Trabajo Fin de Grado
Grado en Ingeniería de Tecnologías Industriales

Prototipo de aplicación para la simulación modular de procesos

Autor: Juan Manuel Donado Jiménez

Tutor: Juan Francisco Coronel Toro

Dpto. Ingeniería Energética
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

GRUPOTERMOTECNIA

Trabajo Fin de Grado
Grado en Ingeniería de Tecnologías Industriales

Prototipo de aplicación para la simulación modular de procesos

Autor:

Juan Manuel Donado Jiménez

Tutor:

Juan Francisco Coronel Toro

Profesor Titular

Dpto. Ingeniería Energética
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Trabajo Fin de Grado: Prototipo de aplicación para la simulación modular de procesos

Autor: Juan Manuel Donado Jiménez
Tutor: Juan Francisco Coronel Toro

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:

A mis padres, por su apoyo incondicional durante estos años de carrera

A mi tutor, por el tiempo dedicado

Resumen

El objetivo de este Trabajo Fin de Grado ha sido desarrollar un prototipo de aplicación web para la resolución de problemas del campo de la ingeniería de una forma visual e intuitiva, acoplando el diagrama del problema con sus ecuaciones. Aunque aún necesita la implementación de muchas características para ser realmente útil, la aplicación ya es capaz de resolver numerosos tipos de problemas. En la presente memoria se analiza la aplicación tanto desde un punto de vista del programador, explicando el código y las librerías utilizadas, como desde el punto de vista del usuario, para que cualquier ingeniero o estudiante de ingeniería pueda utilizar la aplicación para resolver problemas.

Abstract

The goal of this project has been to develop a prototype web application for solving problems in the engineering field in a visual and intuitive way, coupling the diagram of the problem with its equations. Although it still needs the implementation of many features to be really useful, the application is already capable of solving numerous types of problems. This report analyzes the application both from a programmer's point of view, explaining the code and libraries used, and from the user's point of view, so that any engineer or engineering student can use the application to solve problems.

Índice

1 Introducción

El objetivo del presente Trabajo Fin de Grado es desarrollar un prototipo de aplicación web para la simulación modular de procesos, que permita a los ingenieros y estudiantes de ingeniería calcular procesos desde el navegador. No obstante, debido al tiempo disponible y a la gran cantidad de características implementables en un programa de este tipo, solamente se han implementado ciertas características, las suficientes para que el software sea utilizable a modo de prototipo. El resultado ha sido un software que, aunque tiene aún muchos aspectos por pulir, es capaz de resolver casi cualquier problema del campo de la ingeniería, siendo el principal inconveniente la comodidad de uso. En su estado actual, el software no proporcionará una ventaja real al ingeniero, siendo otras vías de resolución más factibles. No obstante, con algo más de desarrollo, puede llegar a ser una herramienta muy útil para resolver una gran variedad de problemas.

1.1 Estado del arte de los simuladores modulares

En la actualidad existen ya múltiples aplicaciones para la simulación de procesos, que, en su estado actual, son mucho más avanzadas que el prototipo que hemos creado en este trabajo. Dichas aplicaciones, no obstante, cuentan con varias desventajas que se han querido subsanar con la creación de esta aplicación.

En primer lugar, la mayoría del software disponible, como ASPEN o TRNSYS, es software comercial, y sus licencias son muy costosas para el usuario medio, por lo que acaban siendo usadas sólamente por grandes empresas o instituciones. Por otra parte, aunque existen alternativas de software libre como openModelica y JModelica, basadas en el lenguaje de programación Modelica, estas requieren de un alto nivel de conocimiento de programación por parte del usuario para ser utilizadas, lo que las hace poco prácticas para la mayoría de los ingenieros industriales.

Además, aunque ASPEN y TRNSYS no requieren para ciertos usos conocimientos de programación, sin tener estos conocimientos el software limita mucho las posibilidades que ofrece a los usuarios. Una desventaja añadida es que además utilizan lenguajes de programación antiguos y con pocos recursos de aprendizaje para el usuario medio.

Nuestra aplicación pretende solucionar estos problemas, otorgando al usuario una gran flexibilidad sin tener que programar, pero en caso de que tenga que hacerlo, el software completo está desarrollado en JavaScript, y utiliza librerías modernas y muy aceptadas por la comunidad, lo que hace muy fácil que cualquier usuario pueda aprender a modificar el software.

Por último, otro factor que diferencia nuestro software de los existentes, es que se trata de una aplicación web, y no requiere por tanto de ninguna descarga o instalación tediosa, ya que cualquiera con un navegador moderno puede acceder desde cualquier ordenador.

1.2 Alcance y objetivos

Antes de listar todas las características implementadas, merece la pena detenerse a hablar sobre la arquitectura general del software. El software es una aplicación web, programada en JavaScript, por lo que no requiere de ningún tipo de instalación para utilizarse, cualquier navegador moderno servirá. El software está disponible en el enlace *juanmadonado.github.io/tfgreact*. Para el desarrollo de la aplicación se han utilizado varias librerías, de las cuales las más destacadas son React y Redux. Esta es una combinación bastante popular

en la comunidad de JavaScript, ya que otorga muchas facilidades al programador. Para el diagrama se ha utilizado una librería algo menos conocida, llamada JointJS. En los siguientes capítulos se analizarán con más profundidad estas librerías, pero su documentación oficial, así como otros recursos que el autor de este texto ha utilizado para aprender su funcionamiento, están disponibles en la bibliografía de este trabajo.

Ahora sí, se listan a continuación las características implementadas en el momento de redacción de este Trabajo Fin de Grado:

Diagrama interactivo La aplicación cuenta con un diagrama interactivo que permite la creación de equipos, cuyo objetivo es simular equipos utilizados en procesos térmicos, como intercambiadores, turbinas, etc. y enlaces, cuyo objetivo es simular las conexiones entre los equipos. Actualmente estos equipos y enlaces están en blanco, es decir, no incluyen ninguna ecuación ni variable por defecto, pero se pueden configurar para modelar cualquier comportamiento deseado.

Ecuaciones, datos e incógnitas Una vez creados los equipos y enlaces con los diagramas, es posible añadirles ecuaciones, datos e incógnitas a los mismos haciendo doble click sobre ellos, lo que abrirá un menú desplegable que permite añadir los elementos deseados a cada equipo. Esto permite configurar los equipos a nuestra medida, para simular casi cualquier tipo de comportamiento

Relaciones En los enlaces se ha implementado lo que en la aplicación se ha llamado "relaciones" que no es más que una expresión matemática de igualdad entre dos variables de dos equipos distintos, los dos que conecta el enlace. Esto permite reutilizar variables de un equipo a otro sin tener que volver a crearlas.

Modelo de resolución matemático La aplicación cuenta con un modelo de resolución matemático para resolver iterativamente sistemas de ecuaciones no lineales. Esto le permite en teoría resolver casi cualquier tipo de problema, aunque puede encontrase con problemas de convergencia si no se establecen valores iniciales adecuados.

Ventana de resultados Al resolver el problema matemático, la aplicación mostrará una ventana emergente con los resultados obtenidos, o con un error en caso de que no haya conseguido obtenerlos. Estos resultados se muestran en una tabla con los nombres de las variables y los equipos a los que pertenecen.

Listadas las características implementadas, listaremos ahora algunas características críticas no implementadas por falta de tiempo en la aplicación. En las conclusiones y recomendaciones se expandirá esta lista, pero merece la pena mencionar algunas aquí para evitar confusiones entre los usuarios de la aplicación.

Guardado de problemas Actualmente la aplicación no permite guardar los problemas realizados en la misma, por lo que en cuanto se cierre el navegador todo el progreso realizado se perderá.

Botones de rehacer y deshacer No están tampoco implementados los botones de rehacer y deshacer, por lo que si el usuario comete un error, tendrá que arreglarlo manualmente.

Propiedades de fluidos Se pueden utilizar propiedades termofísicas de fluidos añadiéndolos como datos, pero el software no es todavía capaz de calcular propiedades de fluidos dadas otras variables termodinámicas independientes.

Uso offline Al ser una aplicación exclusivamente web, no está implementado aún el uso offline de la misma, por lo que una conexión de internet inestable podría afectar a su funcionamiento.

1.3 Estructura del código

Para gestionar el código de la aplicación durante su desarrollo, se ha utilizado un repositorio de GitHub, accesible desde la url *github.com/JuanmaDonado/tfgreact*. Al descargar los archivos del repositorio, nos encontramos con la estructura de ficheros que se muestra en la figura 1.1

La aplicación ha sido creada con create-react-app, una herramienta desarrollada por Facebook para generar automáticamente varios ficheros necesarios en todos los proyectos que utilizan React, y crea una estructura de archivos como la de nuestro proyecto. A continuación se describen todos los ficheros que se encuentran en el repositorio.

public En la carpeta public es donde reside el código de nuestra aplicación que realmente llega a ejecutarse en el navegador. create-react-app incorpora el compilador Webpack, una herramienta utilizada para

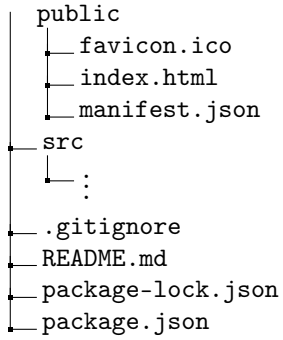

Figura 1.1 Estructura de archivos.

optimizar la ejecución del código en el navegador, además de permitir al programador separar su código en múltiples ficheros. Por ello, nuestro código fuente se encuentra estructurado en múltiples ficheros en la carpeta src, pero en el directorio public solo vemos un único archivo JavaScript (main.js) y un único archivo HTML (index.html). Webpack se ha encargado de transformar todo nuestro código fuente en estos dos archivos, que serán los que cargue el navegador, con la intención de disminuir el tiempo de carga de la aplicación. Además de los principales ficheros de código, también se incluyen el fichero favicon.ico, que no es más que el pequeño icono que se muestra en el título de la web al abrirla con el navegador, y el fichero manifest.json, que contiene información de nuestra aplicación para los navegadores.

src La carpeta src incluye el código fuente de nuestra aplicación. Éste es el código que el programador escribe, pero no llega nunca a ejecutarse en el navegador, sino que es compilado por Webpack en el directorio public, como ya hemos comentado. Esta carpeta tiene a su vez una estructura de archivos algo compleja, pero esperaremos al capítulo 3 para comentarla, ya que será en éste donde analicemos el código fuente de la aplicación.

.gitignore El archivo gitignore es un archivo que le indica a Git, nuestra herramienta de control de versiones, qué archivos tiene que ignorar al realizar la actualización del repositorio. Utilizamos este archivo para que, git no suba al repositorio el código de todas las librerías que se han utilizado en la aplicación, ya que gracias a Webpack no es necesario para ejecutar la misma, aumentaría mucho el tamaño del repositorio, y es fácilmente descargable con un gestor de paquetes como npm, en caso de que se necesite ejecutar la aplicación en un entorno local. No obstante, de nuevo, en el capítulo 3 trataremos con más profundidad cómo utilizar npm para ejecutar la aplicación en un entorno local.

package-lock.json y package.json Estos archivos contienen toda la información necesaria a npm para descargar las librerías que se utilizan en nuestra aplicación. Son generados automáticamente por create-react-app y actualizados por npm.

1.4 Estructura de la memoria

A continuación se listan los capítulos incluidos en esta memoria, así como una breve descripción de lo que contienen para que el lector pueda usarlo de referencia y acceder directamente a los capítulos que le resulten más interesantes.

1. Introducción.
2. Descripción y uso de la aplicación. En este capítulo se describe la aplicación desde un punto de vista del usuario y se muestra su utilización con un ejemplo.
3. Estructura general del software. Se analiza la estructura general del código del software, las librerías utilizadas, y como montar un entorno local para continuar desarrollando la aplicación.
4. Librería gráfica. Se analiza en mayor profundidad la librería gráfica JointJS, utilizada para generar los diagramas interactivos de la aplicación.
5. Modelo del resolución. Se estudia el código utilizado para resolver los sistemas de ecuaciones generados por el usuario tras interactuar con la aplicación.

6. Ejemplos de uso. Se incluyen varios ejemplos de uso de la aplicación para resolver problemas de ingeniería.
7. Conclusiones y recomendaciones.

2 Descripción y uso de la aplicación

En este capítulo nos centraremos en el uso del software desde el punto de vista del usuario. La intención es, que junto al capítulo 6, donde se encontrarán varios ejemplos de uso, el usuario medio pueda aprender a utilizar la aplicación para resolver problemas de ingeniería. Para ello utilizaremos un ejemplo muy básico, crearemos dos equipos conectados entre sí que representan dos intercambiadores por los cuales pasa agua subenfriada, y añadiremos las ecuaciones, variables, relaciones y datos necesarios para resolver el problema con la aplicación.

2.1 Creación de equipos y de enlaces

Al abrir la aplicación, nos encontraremos con la pantalla que se ilustra en la figura 2.1. Como vemos, de momento solo disponemos de un diagrama en blanco y una barra de herramientas inferior con un campo de texto y dos botones: "Nuevo equipo" y "Resolver".

Para crear los dos equipos que necesitamos, bastará con introducir su nombre en el campo de texto y pulsar el botón "Nuevo equipo" para cada uno de ellos. Cabe destacar que todos los nuevos equipos aparecen en el mismo sitio del diagrama, por lo que para poder verlos todos, tendremos que moverlos. Para ello basta con

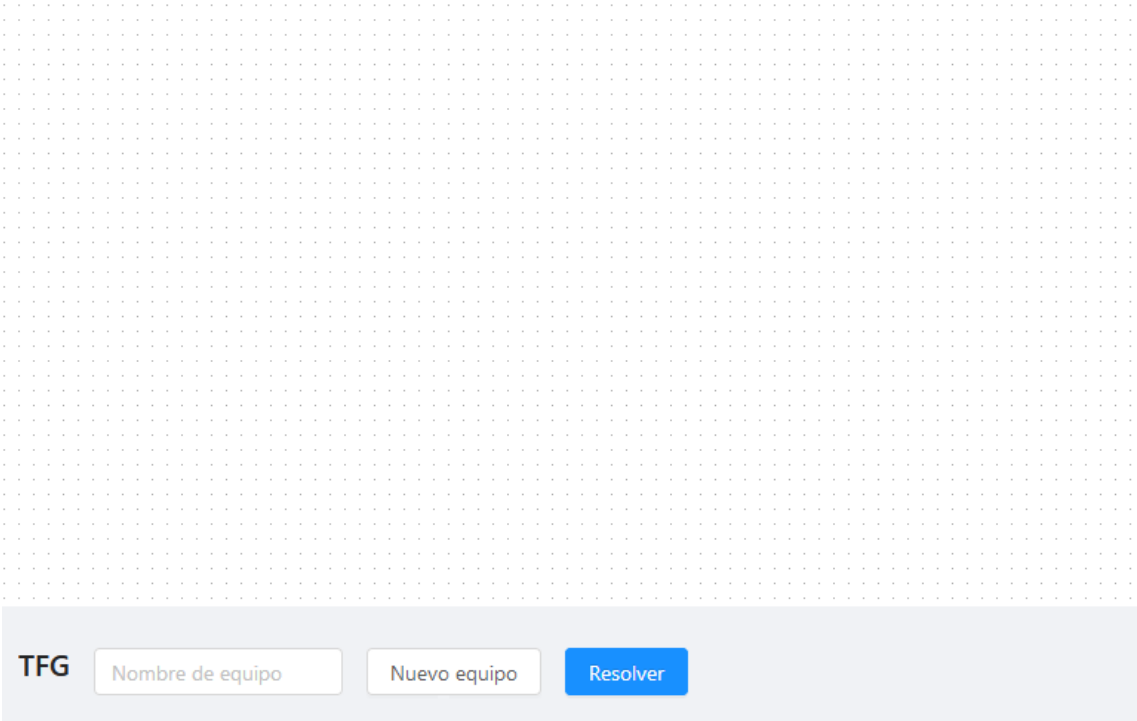

Figura 2.1 Estado inicial de la aplicación.

Figura 2.2 Dos equipos creados.

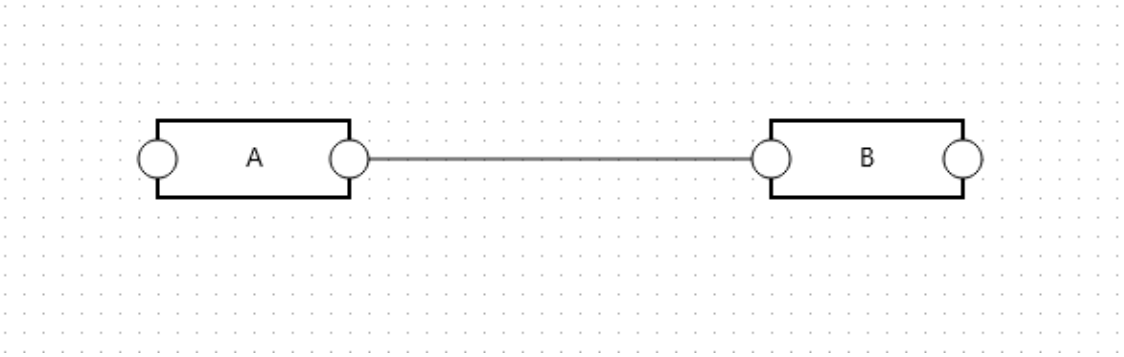

Figura 2.3 Enlaces conectados.

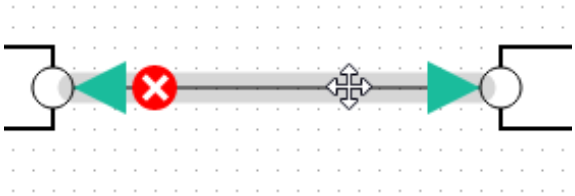

Figura 2.4 Herramientas de enlace.

pulsar y arrastrar en cualquiera de ellos y podremos moverlo a cualquier parte del diagrama. Una vez hecho esto, nos encontraremos en la situación que se muestra en la figura 2.2

Una vez creados los equipos, procederemos a conectarlos. Para ello, basta con pulsar y arrastrar desde cualquiera de los puertos con forma de círculo que tienen los equipos hasta llegar a otro puerto de otro equipo, y tras soltar el equipo quedará conectado. Sin embargo, antes de realizar este paso, es importante tener en cuenta varios aspectos de los enlaces.

En esta aplicación, los enlaces son direccionales, es decir, sí importa el sentido en el que unas los equipos. El motivo de esto es que los enlaces pretenden simular el paso de un fluido por diferentes equipos, y por tanto, debemos conocer la dirección en la que se mueve dicho fluido. A nivel de uso de la aplicación, esto implica que cuando un equipo recibe un enlace, es decir, actúa como destino de un enlace que tiene origen en otro equipo, recibe también todas las variables disponibles en ese equipo, como veremos más adelante. Esto nos permitirá utilizar dichas variables en las ecuaciones, por lo que siempre hay que tener en cuenta la dirección de los enlaces a la hora de plantear el problema. Sabiendo esto, procederemos a conectar el equipo "A" con el equipo "B" en ese orden, obteniendo la configuración que se muestra en la figura 2.3

Una vez tenemos creados los enlaces, podemos pasar el cursor sobre ellos y se mostrarán las herramientas de enlace, que permiten modificarlos, cambiando los equipos de origen y destino, e incluso borrarlos, pulsando en la cruz con fondo rojo.

Una vez creados los equipos y los enlaces necesarios, procederemos en la siguiente sección a añadir los datos necesarios para la resolución del problema.

Figura 2.5 Paneles laterales de equipo (izquierda) y de enlace (derecha).

2.2 Datos, incógnitas, ecuaciones y relaciones

Existen en esta aplicación cuatro elementos importantes a la hora de plantear los problemas que queremos resolver: los datos, las incógnitas, las ecuaciones, y las relaciones. Los dos primeros son triviales: los datos son valores que conocemos, y las incógnitas valores que no. Sin embargo, puede haber confusión entre ecuaciones y relaciones, por lo que intentaremos a continuación explicar la diferencia.

Una ecuación es una expresión matemática de cualquier tipo que relaciona datos e incógnitas declarados en un equipo. De esta descripción es importante destacar dos aspectos fundamentales: primero, que en una ecuación solo pueden aparecer datos e incógnitas que hayamos declarado en la aplicación, y segundo, que las ecuaciones están asociadas a los equipos en los que se declaran, y por tanto solo tienen acceso a los datos e incógnitas que haya declarado en ese equipo. Por tanto, si en el equipo "B" creo una incógnita llamada "t1", no podré utilizar ésta incógnita en una ecuación creada en el equipo "A" o en el equipo "C", ya que estos equipos no tienen acceso a la variable "t1" del equipo "B". La ventaja de esto es que podemos reutilizar los nombres de las variables para cada equipo, es decir, yo puedo definir la variable "presión", con el mismo nombre, en los tres equipos, y será una variable distinta en cada equipo.

Sin embargo, este comportamiento plantea la siguiente cuestión: ¿cómo compartimos información de variables entre equipos? La respuesta es mediante las relaciones. Las relaciones pertenecen a los enlaces, y no son más que una relación de igualdad entre un dato y una incógnita o dos incógnitas de los dos equipos que conecta el enlace. Así, si se quiere hacer que la presión de dos equipos sea la misma, bastará con crear una relación entre ellas.

Entendido el funcionamiento de los cuatro componentes, procederemos ahora a explicar como usarlos en la aplicación. Para ello, tenemos que saber que al hacer doble click sobre un equipo o un enlace, se nos desplegará un panel lateral, como los que se muestran en la figura 2.5 de dicho equipo o enlace, en el cual podremos modificar todos los componentes necesarios.

Empezaremos por el equipo "A". Una vez abierto el panel lateral, añadiremos varios datos nuevos. Para ello, introducimos su nombre en el campo de texto y pulsamos el botón "Añadir". Una vez añadido un nuevo dato, nos aparecerá en la lista de datos, y podremos modificar su valor cambiado el que aparece en el campo de texto junto a su nombre. En este caso, añadiremos cuatros datos nuevos: la temperatura de entrada "Te" con valor de 15, el calor intercambiado "Q" con valor de 2, el "Cp" con valor de 4 y el caudal másico "m" con valor de 1. Si nos hemos equivocado al añadir alguna variable, podemos borrarla pulsando el botón que se encuentra a la derecha del campo numérico. Una vez añadidos deberíamos encontrarnos en la situación que se muestra en la figura 2.6

Ahora, aunque no es necesario para la resolución del resto de equipos, añadiremos una incógnita y una ecuación para calcular el calor intercambiado en el intercambiador A. Para ello, primero nos vamos a la pestaña incógnitas del menú desplegable, que como vemos es idéntica a la de datos. Una vez allí, añadimos una nueva variable "Ts" a la lista. Observaremos que al añadirla también aparecerá un valor numérico al lado del nombre. Este valor no es más que el valor inicial que el modelo de resolución matemática usará

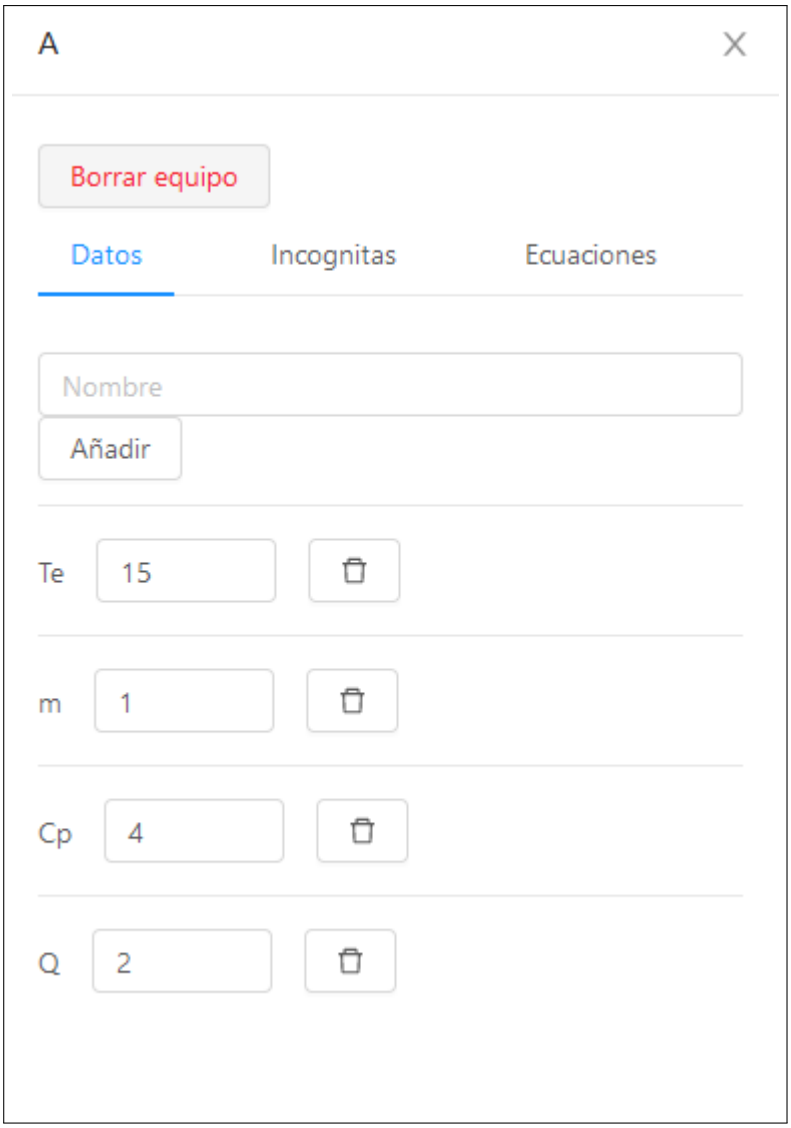

Figura 2.6 Datos del equipo A añadidos.

para iniciar el proceso iterativo, que puede ser necesario modificar en caso de que tengamos problemas de convergencia. Sin embargo, hablaremos más en detalle sobre la resolución en la siguiente sección.

Una vez añadida la incógnita, pulsamos en la pestaña "Ecuaciones" del menú desplegable, que de nuevo es idéntica a las dos anteriores. Para añadir una nueva ecuación, basta con escribirla en el campo de texto y pulsar "Añadir". En este paso es importante utilizar solamente los datos e incógnitas que hemos declarado en sus respectivas pestañas y respetar las mayúsculas y las minúsculas, ya que en caso contrario podemos encontrarnos con errores a la hora de resolver matemáticamente el problema. En este caso, añadiremos la ecuación 2.1 para obtener el calor intercambiado. Una vez hecho esto, deberíamos encontrarnos en la situación que se ilustra en las figuras 2.7 Y 2.8.

$$Q = m * C_p * (T_s - T_e) \tag{2.1}$$

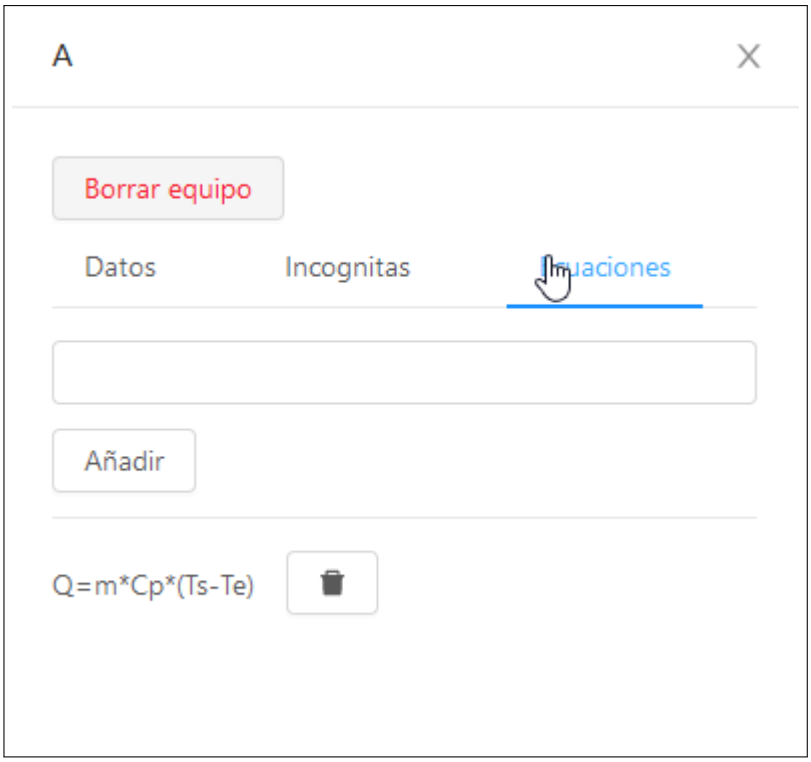

Figura 2.8 Ecuaciones del intercambiador "A".

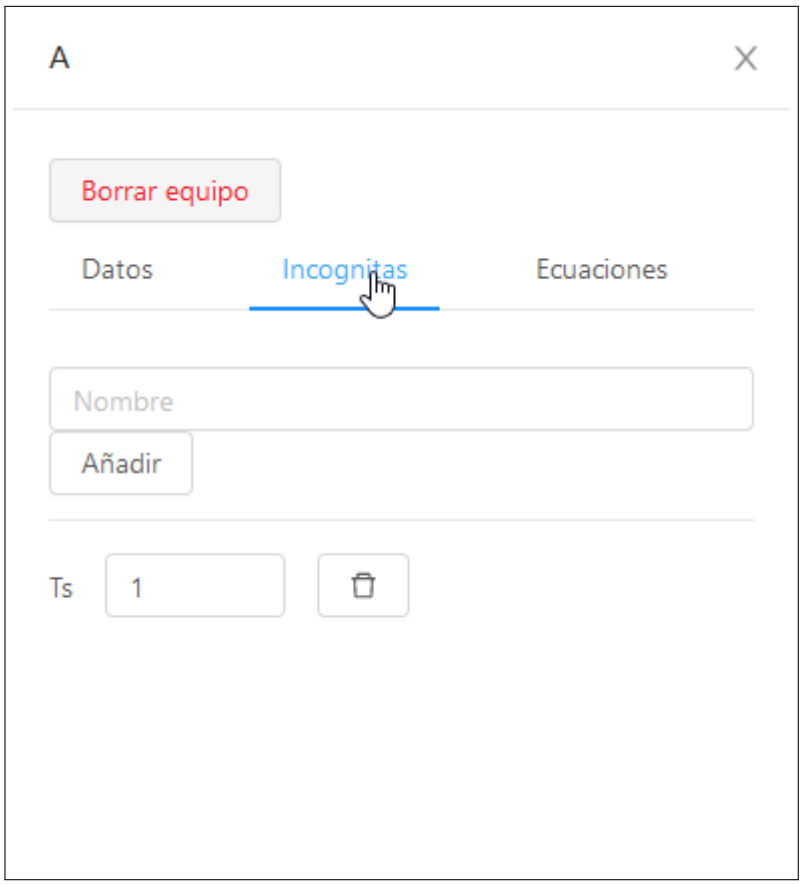

Figura 2.7 Incógnitas del intercambiador "A".

Tabla 2.1 Datos del equipo B.

Datos	Incógnitas	Ecuaciones
$Cp = 4$	Te	$Q = m * Cp * (Ts - Te)$
$Q = 3$	Ts	
	m	

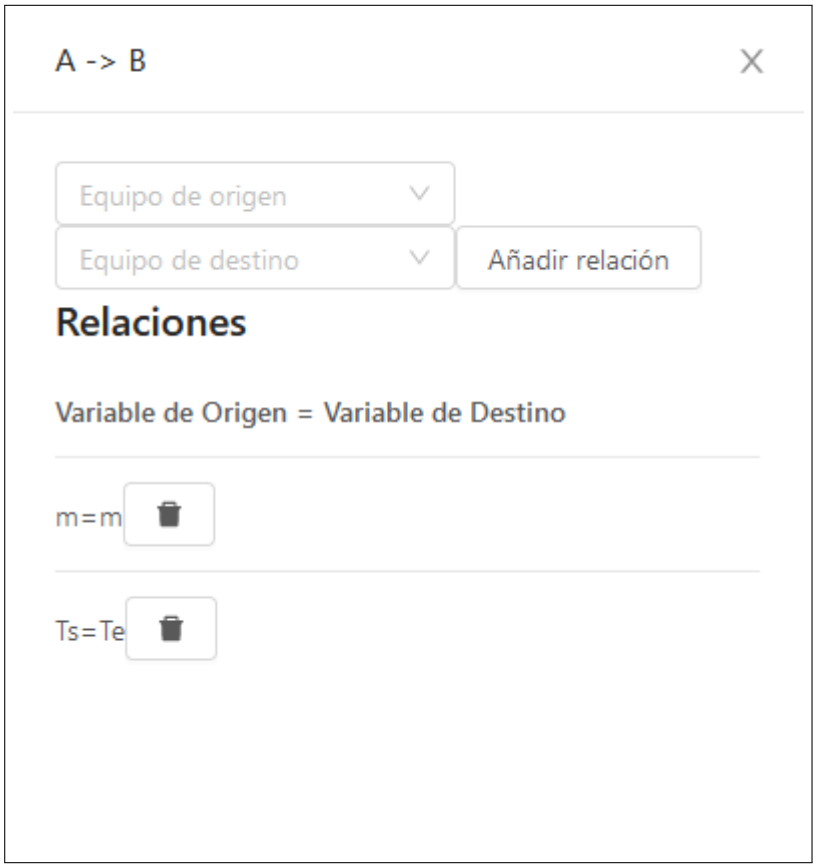

Figura 2.9 Relaciones del enlace A - B.

Una vez aprendido como añadir datos, ecuaciones e incógnitas en los equipos, procederemos añadirlos ahora para el intercambiador "B". En la tabla 2.1 se muestran los datos, incógnitas y ecuaciones a añadir.

Aunque ya hemos añadido toda la información necesaria en los equipos, nuestro problema aún no está planteado por completo, ya que para para poder resolverlo tendremos que añadir las relaciones necesarias en los enlaces. El proceso para añadir una relación es similar a lo realizado hasta ahora, solo que esta vez tendremos que hacer doble click en un enlace para abrir su menú desplegable. Una vez abierto podemos observar varios aspectos importantes de este menú desplegable. En la cabecera del menú, podemos ver la direccionalidad del enlace, es decir, qué equipo es el origen y cuál es el destino del enlace. Esto es importante, ya que en los menús desplegables situados a continuación, aparecen unas u otras variables en función de si el equipo es el origen o destino del enlace.

Sabiendo esto, podemos añadir una relación seleccionando las dos variables que queremos igualar en los menús desplegables, y pulsando el botón "Añadir relación". Es importante saber que en el menú desplegable del equipo de origen tendremos disponibles sus datos y sus incógnitas, pero en el del equipo de destino solo estarán sus incógnitas. Esto es para evitar la creación de una relación dato-dato, que haría el problema irresoluble si los datos tuvieran distintos valores. Si se desea utilizar los datos del destino en el origen, es necesario crear un enlace en la dirección contraria. Así, podemos crear las relaciones necesarias para nuestro problema, que en este caso son $m = m$ y $Ts = Te$, obteniendo una configuración como la que se muestra en la figura 2.9.

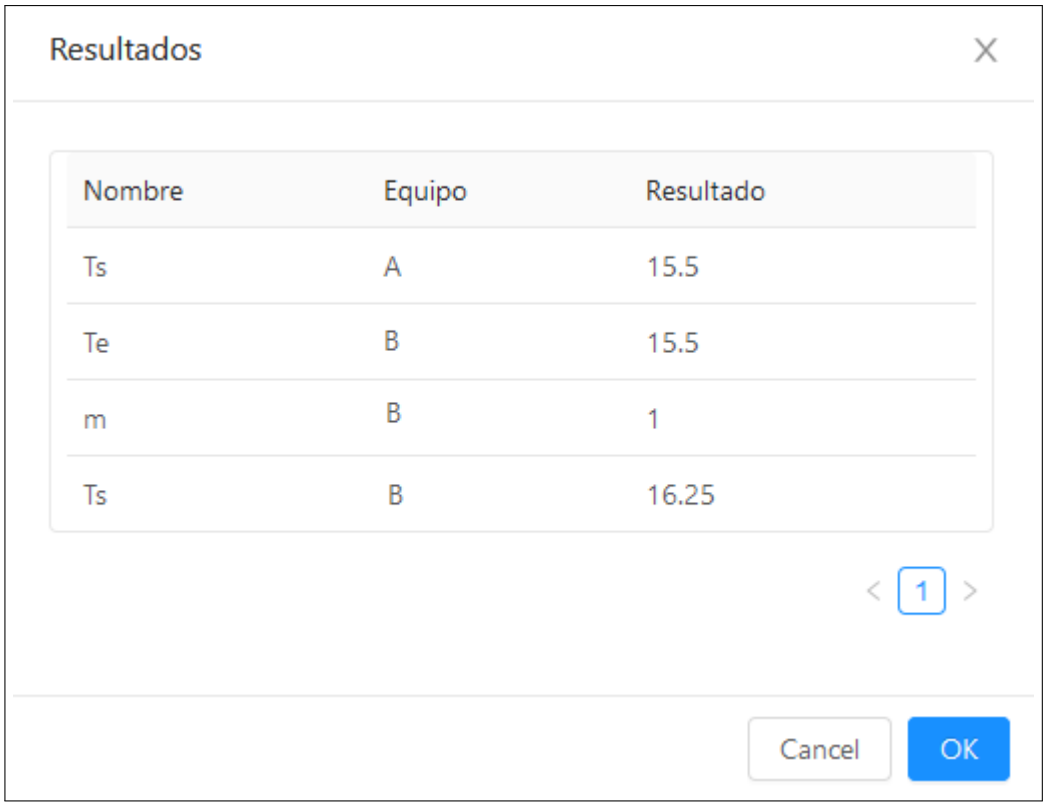

Figura 2.10 Ventana de solución del problema.

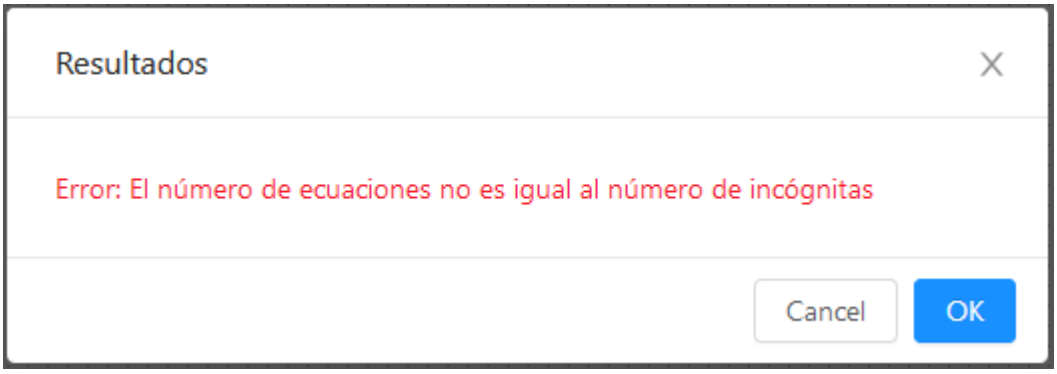

Figura 2.11 Mensaje de error en la ventana de solución.

2.3 Resolución

Para resolver el problema, bastará con que pulsemos el botón "Resolver" situado en la barra de herramientas inferior. Al hacerlo, se nos abrirá una ventana emergente que contendrá la solución del problema, como se muestra en la figura 2.10

Como vemos, en esta ventana se muestra una tabla, siendo cada fila el resultado que ha obtenido el modelo para cada incógnita, con tres columnas, de izquierda a derecha: nombre de la incógnita, equipo al que pertenece, y valor numérico de la solución.

Sin embargo, la ventana de solución puede no mostrar la solución en caso de que haya un error en la misma. En concreto, si el número de ecuaciones no es igual al de incógnitas o si ha superado el número máximo de iteraciones (1000 en la fecha de redacción de este documento, sin ser modificable por el usuario) nos lo mostrará en rojo en lugar de la solución, como vemos en la figura 2.11

Con esto, ya conocemos todo lo necesario para usar la aplicación en la resolución de problemas, por lo que en el capítulo 6 mostraremos algunos ejemplos para demostrar el uso de las herramientas que hemos

aprendido.

3 Estructura general del software

En este capítulo se analizará la estructura general del código del software, así como las diferentes librerías que se han utilizado para el desarrollo del mismo.

3.1 Instalación en un entorno local

En caso de querer continuar con el desarrollo de la aplicación, lo más sensato es establecer un entorno local de desarrollo en el que poder modificar el código sin alterar la versión colgada en la web. Por suerte, el proceso es bastante sencillo, pero requiere que el usuario tenga instalado en su ordenador el gestor de paquetes npm, que se incluye con NodeJS. Además, en caso de que se quieran subir cambios al repositorio en línea existente, también se necesitará instalar Git. Con estos programas instalados, podemos proceder a instalar la aplicación en un entorno local.

El primer paso es descargar el código de la aplicación. Para ello, podemos acceder directamente a la web del repositorio *https://github.com/JuanmaDonado/tfgreact* y pulsar el botón *Clone or download*. Una vez pulsado, se nos darán dos opciones, ambas completamente válidas. La primera es pulsar *Download ZIP*, lo que nos descargará un archivo comprimido con todo el código de nuestro repositorio. Una vez hecho esto, sólo tenemos que descomprimirlo en la carpeta que deseemos y ya tendremos el código disponible en nuestro ordenador. La segunda es usar git desde un terminal, y utilizar el enlace que aparece tras pulsar *Clone or download* con el comando *git clone* de git, y se descargará todo el repositorio al directorio en el que actualmente nos encontremos en el terminal.

Una vez descargado el código, lo primero es abrir un terminal en el directorio donde se encuentre el mismo. Desde dicho terminal, tendremos que ejecutar el comando *npm install* que descargará todas las librerías que son necesarias para la ejecutar la aplicación. Una vez terminada dicha descarga, que puede llevar unos minutos, podremos iniciar un entorno local con el comando *npm start* que nos abrirá una pestaña en el navegador con la aplicación ya ejecutándose. Además, mientras mantengamos el terminal abierto, todos los cambios que hagamos en el código de la aplicación harán que automáticamente se refresque la página donde tenemos abierta la aplicación, y tras hacerlo mostrará los nuevos cambios que hemos añadido.

Con estos dos comandos ya estamos listos para continuar desarrollando la aplicación, pero en caso de querer subir una nueva versión a internet, tendremos que compilar la aplicación y subirla al destino deseado. Por suerte, create-react-app también ofrece herramientas para hacer esto, y su procedimiento se encuentra explicado en el enlace *https://create-react-app.dev/docs/deployment*

3.2 Estructura del código fuente

Al descargar el repositorio de la aplicación, en la carpeta src encontramos el código fuente de la aplicación. A continuación se describen los contenidos de cada directorio de esta carpeta.

actions-creators Contiene las funciones para crear acciones de Redux, para todas las acciones utilizadas en nuestra aplicación

components Contiene un archivo JavaScript para cada componente de React utilizado en nuestra UI

constants Esta carpeta solo contiene un sólo archivo, actionTypes.js, que establece unas constantes para su uso en las acciones de Redux. No es necesario utilizar estas constantes, pero de esta forma aseguramos no escribir los tipos de acción erróneamente

diagrama Por el momento contiene un único archivo, diagrama.js, con todo el código necesario para inicializar el diagrama de JointJS.

mates-jfc Contiene todos los archivos necesarios para el modelo de resolución de ecuaciones, uno por cada objeto que utiliza este modelo.

reducers Un archivo por cada reducer de Redux que utiliza nuestra aplicación.

solver Código necesario para transformar las ecuaciones generadas por nuestra aplicación en ecuaciones que entienda el modelo de resolución de ecuaciones.

store Contiene un único archivo, store.js, con el código de la Store de Redux de nuestra aplicación.

index.js Archivo de entrada para webpack, donde se inicializan la Store, los componentes de React, el diagrama de JointJS.

serviceWorker.js Archivo auxiliar de la librería React creado automáticamente por create-react-app.

3.3 Separación Vista-Modelo

La primera característica a destacar es que desde un principio se ha pretendido separar la lógica de presentación e interactividad del software (lo que llamamos vista) con la lógica de resolución de problemas (el modelo). La ventaja de dicha separación es que si en algún momento necesitamos modificar alguna de las dos partes, ya sea para cambiar librerías, añadir nuevas características al software, o incluso rehacer una de las partes por completo, no tendremos que modificar todo el software existente, sino que bastará con modificar tan sólo la parte necesaria.

Este paradigma es muy utilizado en el campo de la programación, y aunque tiene muchas variantes, el más conocido es el llamado Model-View-Controller (MVC), cuyo esquema de funcionamiento se ilustra en la figura 3.1. La idea es la siguiente: el usuario solamente tiene acceso a la vista, y en función de los comandos que el usuario ejecute, la vista notifica el evento al controlador, que para cada evento manda determinadas órdenes al modelo. El modelo, al recibir éstas ordenes, se actualiza de la forma correspondiente, y manda una notificación de que se ha actualizado. Al recibir esta notificación, la vista se actualiza para reflejar el estado actual del modelo.

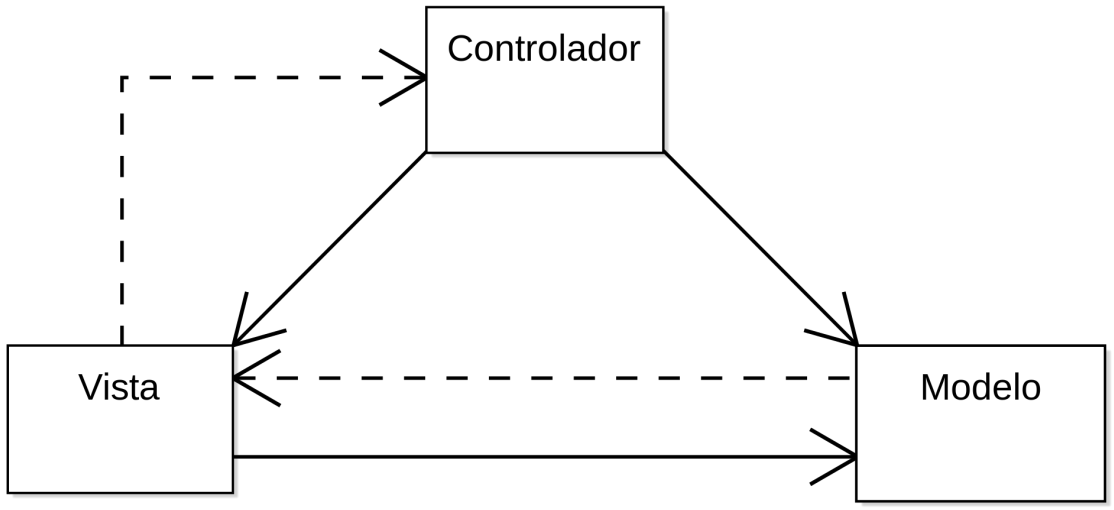

Figura 3.1 Esquema del modelo MVC.

Sin embargo, gracias a la utilización de librerías modernas de JavaScript como son React y Redux, cuyo funcionamiento se explicará con más detalle en las próximas secciones, nuestra aplicación no requiere la

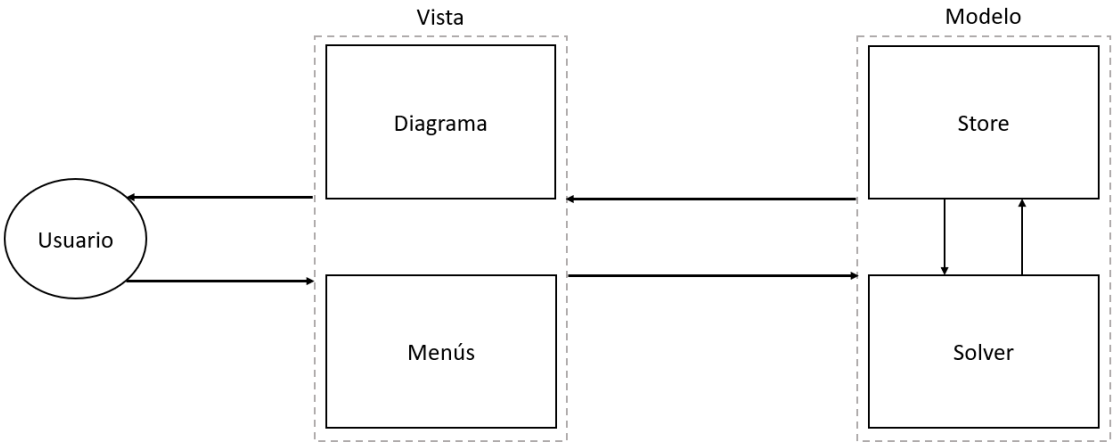

Figura 3.2 Esquema de funcionamiento del software.

programación de un controlador para funcionar, ya que las librerías utilizadas hacen las veces de controlador y se ocupan de gestionar las acciones de los usuarios y las actualizaciones del modelo. Además, tanto la vista como el modelo están a su vez separadas en varias partes independientes como se ilustra en la figura 3.2.

Como vemos, la vista está separada en el diagrama, generado por la librería JointJS, y los menús, programados utilizando la librería React. JointJS se encarga de gestionar toda la lógica correspondiente a la interactividad del diagrama, como es mover equipos o conectar unos con otros. React por su parte gestiona toda la lógica de la UI, como los existentes para crear y borrar equipos, pero también los de añadir y borrar ecuaciones, variables o datos.

Por otro lado, el modelo tiene también dos partes. Por un lado está el "Store", que es la herramienta que aporta la librería Redux para almacenar los datos del modelo. Profundizaremos más en su correspondiente sección, pero de momento lo único que necesitamos saber es que este "Store" almacena toda la información de los equipos, enlaces, variables, datos y ecuaciones, es decir, todo lo necesario para resolver el problema. Por su lado, el "Solver", a cuyo funcionamiento dedicaremos un capítulo completo, no guarda ningún estado del programa, sino que su única función es recibir el estado actual del "Store", resolver matemáticamente el problema, y devolver los resultados a la vista para que ésta los muestre.

Utilizaremos un ejemplo para entender mejor el funcionamiento del software. Al pulsar el botón "Nuevo equipo", React obtiene el nombre del equipo del campo correspondiente, que manda un evento al "Store" solicitando que se cree un nuevo equipo con el nombre que el usuario ha introducido. El "Store" se actualiza, y al estar el diagrama escuchando todas las actualizaciones del "Store", éste se actualiza para mostrar el estado actual del mismo. Exactamente el mismo procedimiento ocurre al modificar enlaces, ecuaciones, datos o incógnitas. Para cada caso, existe un evento distinto que manda la mínima información necesaria al "Store" para que se actualice de la manera correspondiente, y al hacerlo, provoca que la vista se actualice. Al pulsar el botón "Resolver", la vista ejecuta el "Solver", que pide al "Store" su estado actual, realiza las operaciones pertinentes, y devuelve a la vista los resultados, para que ésta los muestre al usuario.

Una vez conocido el funcionamiento general, procederemos en las siguientes secciones a profundizar un poco más en cada una de las partes.

3.4 Vista

Como ya mencionamos en la sección anterior, la vista de nuestro software está dividida en dos partes, el diagrama, desarrollado con la librería JointJS, y los menús, desarrollados con la librería React. La razón de esta separación es que programar un diagrama interactivo con React hubiese llevado demasiado tiempo, que nos ahorramos al utilizar JointJS. Sin embargo, como veremos, reprogramar el diagrama con React tiene algunas ventajas sobre JointJS, por lo que es una mejora a implementar en el futuro. De momento, ésta separación es funcional, y vamos a analizar su funcionamiento en los siguientes apartados.

3.4.1 UI con React

React es una librería desarrollada por Facebook que está muy de moda en la comunidad de JavaScript. Y no es para menos, ya que React permite desarrollar interfaces de usuario de manera fácil y rápida, pudiendo evitar muchísima repetición de código e incluso utilizar código ya escrito por otros usuarios.

La manera en la que permite esto es mediante el uso de lo que React llama *componentes*. Un componente no es más que una parte de tu interfaz, como por ejemplo, un botón, que una vez programado puede reutilizarse en múltiples partes de la interfaz sin tener que volver a programarlo. Cabe destacar, sin embargo, que en React **todo** es un componente, no solo los elementos pequeños, lo que te permite crear una jerarquía de componentes reutilizables con las que puedes construir una interfaz.

Otra de las bondades de React es que la misma librería se encarga de actualizar la vista cuando el modelo se actualiza, pero además hace esto eficientemente, es decir, solo actualiza las partes de la vista que necesitan ser actualizadas, ahorrando así potencia de procesamiento.

Por último, cabe destacar que, como se ha comentado al inicio de esta sección, la separación del código en componentes permite utilizar componentes ya desarrollados por otros usuarios. En nuestro caso, hemos utilizado componentes de Ant Design, que no son más que componentes básicos como botones o campos de texto, que vienen ya estilizados, permitiéndonos darle a la aplicación un aspecto más pulido sin tener que invertir mucho tiempo en ello.

3.4.2 Diagramas con JointJS

JointJS es una librería gratuita de JavaScript para el desarrollo de diagramas interactivos. Aunque dedicaremos un capítulo completo para explicar en profundidad los detalles de la misma, aprovecharemos para dar una idea general del funcionamiento aquí.

JointJS funciona independientemente con su propio modelo MVC. Esto es, al utilizarlo, generas un grafo, que representa el modelo de los equipos y sus conexiones, y lo que la librería llama "paper" que no es más que la vista del grafo. Sin embargo, como hemos explicado anteriormente, nuestra aplicación tiene su propio modelo, por lo que no nos interesa utilizar el modelo que crea automáticamente esta librería.

Esta es una de las razones por las que se mencionó anteriormente que sería interesante cambiar el diagrama a React, ya que eliminaría este modelo que JointJS crea. No obstante, esta situación se puede solucionar simplemente ignorando el modelo de la librería y enlazando la vista de los equipos con nuestro modelo. Para identificar qué vista corresponde con cada equipo en nuestro modelo, utilizamos un identificador único universal (UUID por sus siglas en inglés) que no es más que una cadena de caracteres generada aleatoriamente, cuya longitud es suficientemente grande como para que sea prácticamente imposible que dos UUIDs coincidan.

Al crear un nuevo equipo, JointJS genera un UUID para su propio modelo, que nos es accesible en el momento de su creación, por lo que lo mandamos como información en el evento de creación de equipos de nuestro modelo. Así, cada vez que un nuevo equipo se crea en nuestro modelo, éste tiene el mismo UUID que la vista en JointJS, pudiendo así saber qué vista corresponde a cada modelo, que es importante para futuras modificaciones o eliminaciones de elementos.

Otra de las consecuencias de que JointJS tenga su propio modelo es que su vista no responde a cambios en nuestro modelo, sino en el suyo propio, por lo que, realmente, en el caso del diagrama es la vista la que se actualiza primero, y tras hacerlo manda los eventos correspondientes al nuestro para que se modifique de la misma forma. Esto difiere a la forma en la que se actualiza el resto de la aplicación, ya que en ésta los eventos son mandados directamente a nuestro modelo, y tras actualizarse el modelo, la vista lo hace reflejando el nuevo estado actual. Es importante tener en cuenta esto a la hora de programar modificaciones en el diagrama.

A pesar de estas dificultades, JointJS ha resultado ser una librería muy capaz para generar diagramas interactivos, ya que como veremos en el capítulo dedicado a la misma, proporciona muchas facilidades para gestionar las interacciones de los usuarios con el diagrama.

3.5 Modelo

Como se ha mencionado anteriormente, el modelo de nuestra aplicación consta de dos partes separadas: por un lado una "Store" hecha con la librería Redux, que almacena toda la información de nuestra aplicación, y un Solver, que recibe el estado actual de la aplicación y devuelve la resolución matemática del mismo. Procederemos ahora a analizar cada parte por separado.

3.5.1 Modelo con Redux

Redux es una de las librerías más populares en el ecosistema JavaScript para la programación de modelos de aplicaciones. Aunque está pensada para grandes aplicaciones, se ha decidido utilizarla para nuestra aplicación por dos principales razones: la primera es que hace toda la lógica del modelo consistente, es decir, todas las partes de nuestro modelo están programadas siguiendo la misma lógica, por lo que una vez la aprendes, entiendes toda la aplicación, lo que evita futuras confusiones. La segunda es que permite escalar fácilmente la aplicación, lo que significa que podremos añadir en el futuro nuevas características sin tener que reescribir totalmente el modelo actual.

Sabiendo las razones por las que hemos elegido Redux, explicaremos ahora brevemente su funcionamiento, aunque de nuevo, existen numerosos recursos en la web para aprender en profundidad esta herramienta.

Redux es una librería muy pequeña que tiene tres conceptos básicos que permiten construir todo tipo de modelos para aplicaciones. El primero de ellos es la "Store" que no es más que un objeto de JavaScript que contiene todo el estado de nuestra aplicación. La estructura de este objeto es libre, pero lo que es imperativo es que este objeto es inmutable, es decir, no podemos hacer modificaciones directamente en el propio objeto. Esta limitación puede parecer engorrosa, pero en realidad nos permite escribir código fiable y fácilmente testeable ya que, como veremos, para modificar el estado actual escribiremos exclusivamente funciones puras conocidas como reducers. En el código 3.1 podemos ver la estructura del store de nuestra aplicación.

```
{
  seleccionado: 472eb76b-142a-4dcc-802c-a48a7c822d15,
  isDrawerOpen: false,
  entities: {
    equipos: { /*...*/ },
    enlaces: { /*...*/ },
    datos: { /*...*/ },
    incognitas: { /*...*/ },
    ecuaciones: { /*...*/ },
    relaciones: { /*...*/ },
  }
}
```

Código 3.1 Store de Redux para nuestra aplicación.

Los reducers son el segundo concepto importante de Redux, y no son más que funciones puras, es decir, funciones que para unos mismos argumentos de entrada, devuelven siempre el mismo resultado, por lo que no dependen de otras condiciones externas, permitiendo predecir fiablemente el comportamiento de una función. Este reducer siempre recibe dos argumentos: un estado de la aplicación, y una acción. Esta acción es la que le indica al reducer qué modificaciones realizar en el estado que ha recibido la función, y tras realizarlas, devuelve el estado modificado. Por tanto, cada vez que necesitamos actualizar el estado actual, lo mandamos al reducer correspondiente junto con una acción, y este nos devuelve el nuevo estado, que sustituimos completamente en el Store. Efectivamente, al ser el Store inmutable, para realizar modificaciones tenemos que darle un nuevo estado completo, con las modificaciones que queramos. Aunque esto es técnicamente ineficiente, ya que tenemos que sobrescribir también las partes del estado que no han cambiado, debido a la gran potencia de procesamiento que tienen los ordenadores de hoy en día no supone ningún problema en su uso, mientras que otorga a los programadores las ventajas que comentamos al principio de esta sección. En el código 3.2 podemos ver un ejemplo de un reducer de Redux.

```
function seleccionadoReducer(state = "", action) {
  switch (action.type) {
    case CAMBIAR_SELECCION: {
      return action.payload.nuevaSeleccion;
    }
    case BORRAR_EQUIPO: {
      return "";
    }
    case BORRAR_ENLACE: {
```

```
      return "";
    }
    default: {
      return state;
    }
  }
}
```

Código 3.2 Reducer de Redux.

El último de los conceptos importantes son las acciones, ya mencionados en el párrafo anterior, que no son más que pequeños objetos de JavaScript que contienen toda la información necesaria para que los reducers hagan su trabajo. En nuestro caso, por ejemplo, una acción podría ser la de crear un nuevo equipo, que necesita que se le otorgue un nombre para el equipo y un UUID para identificarlo, así, el objeto de JavaScript contendría únicamente estos tres datos, es decir: el tipo de acción, el nombre, y el UUID. Al pedirle a redux que ejecute la acción, este utilizará el reducer correspondiente y actualizará el estado de la manera pertinente. En el código 3.3 vemos un ejemplo de una acción de Redux.

```
{
    type: CAMBIAR_SELECCION,
    payload: {
      nuevaSeleccion
    }
}
```

Código 3.3 Reducer de Redux.

Una vez entendidas las acciones, cabe subrayar un pequeño atajo que se utiliza para evitar repetición de códigos. Ya que un mismo tipo de acción tiene siempre la misma estructura y lo único que varía son ciertos parámetros, para evitar tener que escribir cada vez el objeto que describe la acción, escribimos funciones llamadas "creadores de acciones" que no son más que funciones que devuelven una acción al suministrarle los argumentos necesarios. Esto no es fundamental para el uso de Redux, ya que podríamos escribir manualmente las acciones, pero como hemos dicho sirve para ahorrar algunas líneas de código.

Con estos tres conceptos claros, solo falta saber como Redux los conecta para que funcionen correctamente. Realmente, aunque estos conceptos son fundamentales en Redux, podríamos haberlos utilizado sin usar la librería, ya que no son más que funciones y objetos nativos de JavaScript. ¿Qué aporta entonces Redux? Pues ciertas utilidades para conectar estas tres herramientas, que nos evitan tener que escribirlas nosotros, además de estar optimizadas, labor complicado para el programador medio. Las dos herramientas más importantes son las funciones subscribe y dispatch. La función subscribe recibe un "listener", una función que ejecutar después de que se produzca un cambio en la store. Esto es útil porque nos permite conectar fácilmente cualquier vista con nuestro modelo, ya que Redux se encargará de ejecutar dicha función cada vez que se produzca un cambio en el modelo. La segunda función, dispatch, no es más que la manera de ejecutar acciones sobre el estado actual. Dispatch simplemente se encarga de recibir una acción, y de ejecutar el reducer correspondiente para modificar el estado. Con esto, ya tenemos una herramienta simple a la vez que potente para poder manejar cualquier tipo de modelo que queramos crear. Redux aporta también otras funciones que resultan muy útiles para el programador, como la función combineReducers, pero no son necesarias para entender su funcionamiento básico, por lo que nos referimos a la bibliografía para aprender sobre estas funciones.

Una vez tenemos estos elementos, es tan simple como utilizar la función dispatch para ejecutar la acción. Podríamos, por ejemplo, hacer que un botón de la vista de nuestra aplicación, llamase a la función dispatch con la acción crear equipo. Una vez modificado el estado, nuestra vista, que estaría suscrita a los cambios del modelo utilizando la función subscribe, se actualizaría para reflejar el estado del modelo.

Estos conceptos son suficientes para entender el papel que tiene Redux en nuestra aplicación, pero reiteramos que se puede ampliar conocimientos con la bibliografía aportada. La desventaja que presenta Redux para una pequeña aplicación como es la nuestra actualmente, es que hace el código bastante más extenso de lo que sería necesario sin utilizar la librería. Sin embargo, el autor ha considerado que por la claridad y la consistencia que otorga Redux merecía la pena escribir algunas líneas más de código.

3.5.2 Solver

Por último, el Solver es la parte del software dedicada a la resolución de los sistemas de ecuaciones generados a partir de los equipos y enlaces. Consta de dos partes principalmente, un modelo de resolución, que tiene un capítulo propio en el que se analiza en profundidad, y una función, llamada solve, implementada para compatibilizar las ecuaciones generadas con el modelo que genera nuestra aplicación.

El modelo de resolución de ecuaciones, como veremos en su capítulo, es una parte del código independiente al resto del programa, por lo que se ha tenido que desarrollar una función que transforme el modelo de nuestra aplicación en datos que el modelo de resolución pueda manejar. Para hacer esto, se ha utilizado una librería de JavaScript llamada RamdaJS, que es una librería de utilidades orientada a la programación funcional, de la que se utilizan varias funciones para transformar unos objetos en otros y así poder utilizar el modelo de resolución.

4 Librería gráfica para el diseño de diagramas: uso de JointJS

Como se ha mencionado en el capítulo 2, para el desarrollo de un diagrama interactivo en nuestra aplicación, hemos utilizado la librería JointJS, que ha resultado ser muy capaz para esta tarea. Esta librería, cuya documentación se encuentra disponible en la web *https://resources.jointjs.com/docs/jointjs/v3.0/joint.html*, es una librería Open Source bajo la licencia MPL 2.0, por lo que permite su uso en aplicaciones de casi cualquier tipo, siempre que se respete su autoría. Los desarrolladores de esta librería también ofrecen una versión de pago, llamada Rappid, que incluye algunas características adicionales, pero para nuestro caso la versión gratuita es suficiente.

Esta librería no es demasiado popular, por lo que no cuenta con mucha comunidad tras de ella, y las guías y tutoriales desarrollados por usuarios son casi inexistentes. El mejor recurso para aprender a utilizarla es la documentación de la librería, que viene también acompañada de un tutorial, disponible en el enlace *https://resources.jointjs.com/tutorial* junto con algunas demos. No obstante, la documentación, aun siendo el recurso más completo, es algo deficiente, por lo que dedicaremos las siguientes secciones a explorar su funcionamiento, así como algunos ejemplos del código del programa para que el lector pueda utilizar cómodamente esta librería.

4.1 El paper y el graph

Los dos elementos principales para entender JointJS son el paper y el graph, que no son más que la vista y el modelo del diagrama. Como mencionaba el anterior capítulo, JointJS utiliza su propio modelo MVC, por lo que obliga a generar un modelo, el graph, para que su vista, el paper, lo muestre en el navegador.

Crear el paper y el graph es relativamente sencillo, solo necesitamos crear un nuevo objeto, con el nombre que creamos apropiado, que copie el objeto modelo que aporta la librería. Podemos ver un ejemplo de esto en el código 4.1

```
1  const graph = new joint.dia.Graph();
2
3  const paper = new joint.dia.Paper({
4    el: document.querySelector(".diagrama"),
5    model: graph,
6    width: "100%",
7    height: "100%",
8    gridSize: 10,
9    drawGrid: true,
10   defaultRouter: { name: "manhattan" },
11   linkPinning: false,
12   interactive: { vertexAdd: false }
13 });
```

Código 4.1 Creación de paper y graph en JointJS.

Como vemos, para crear el graph no añadimos ningún parámetro adicional, pero al paper le suministramos un objeto con varias claves que nos permiten configurarlo a nuestra medida. En la documentación tenemos todas las opciones posibles, pero explicaremos a continuación las utilizadas aquí y por qué se han utilizado.

el Al parámetro "el" hay que suministrarle un selector de JavaScript que contenga el nodo HTML en el cual vamos a mostrar nuestro diagrama. Este parámetro es obligatorio, o el diagrama no se mostrará.

model El modelo a utilizar por nuestro paper. En este caso es la variable graph, que es como hemos llamado a nuestro graph. También obligatorio.

width y height Estilos de CSS de altura y anchura de diagrama. Se utiliza 100 % para que ocupen todo el espacio disponible.

gridSize Este parámetro es opcional, y se puede especificar para crear un grid en nuestro diagrama, que son los pequeños puntos a los que se anclan los objetos del diagrama, y que permiten y posicionamiento más cómodo.

defaultRouter El parámetro defaultRouter también es opcional, y a él se le suministra el tipo de "router" que queremos utilizar en el diagrama. Este "router" es el que se encarga que los caminos de los enlaces sigan cierto patrón al dibujarlo. Por ejemplo, el router Manhattan encuentra el camino ortogonal más corto entre los dos equipos. En la documentación están disponibles todos los routers que incluye JointJS.

linkPinning Este parámetro, por defecto en true, se puede establecer en false para que los enlaces no puedan empezar o acabar en partes en blanco del diagrama. En nuestra aplicación no tiene sentido que un enlace pueda acabar o empezar en partes en blanco, ya que representan conexiones entre equipos, por lo que lo establecemos en false.

interactive Al parámetro interactive se le puede suministrar un objeto con varias opciones a configurar en cuanto a la interactividad del diagrama. En este caso, especificamos el parámetro vertexAdd como false para que no esté permitido añadir nuevos vértices a los enlaces. De nuevo, en la documentación se encuentran todos los parámetros que se le pueden suministrar a interactive.

Creados estos dos objetos, y si existe el nodo HTML que le hemos proporcionado al paper, nuestro diagrama estará inicializado y listo para usar. Sin embargo, éste no dispondrá aun de ningún equipo, por lo que en la siguiente sección veremos cómo se crean los mismos.

4.2 Equipos y enlaces

Para añadir un nuevo equipo, bastará con que especifiquemos una serie de opciones y añadamos el equipo en el graph. JointJS se encarga de que cada vez que el graph se actualice, lo haga el paper de la misma manera, y viceversa, por lo que solo nos tenemos que preocupar de añadir los equipos en el graph para que aparezcan en el paper. Para ello, JointJS nos suministra varios objetos predefinidos para utilizar en nuestro diagrama. La colección es bastante amplia, y se encuentra, una vez más, disponible en la documentación de la librería. Para nuestro caso utilizaremos un simple rectángulo, para los que JointJS proporciona el objeto joint.shapes.standard.Rectangle, al que se le pueden suministrar varios parámetros para poder configurarlo. En concreto, en el código 4.2 vemos el código utilizado en nuestra aplicación.

```
1  const visualizacion = new joint.shapes.standard.Rectangle();
2  visualizacion.position(800, 400); // posición (x,y) en el diagrama
3  visualizacion.resize(100, 40); // tamaño en (anchura, altura)
4  visualizacion.attr({
5    body: { // estilos para el cuerpo del rectángulo
6      fill: "white" // establecemos el color de fondo en blanco
7    },
8    label: { // estilos para el texto del rectángulo
9      text: nombre, // utilizamos una variable declarada anteriormente
10     fill: "black" // color del texto
11   },
12   root: {
```

```
      magnet: "false"
    }
  });
```

Código 4.2 Creación de rectángulo en JointJS.

Además de las propiedades explicadas en los comentarios del código, hay dos aspectos importantes a destacar. El primero es el uso de una variable llamada nombre en la propiedad text. La razón de esto es que como veremos más adelante, para que el usuario pueda crear nuevos equipos, encapsularemos la creación de equipos en una función, y nombre será un parámetro de dicha función. El segundo aspecto a destacar es el parámetro magnet, que establecemos en false, con lo que impedimos que los enlaces se conecten a los rectángulos creados. Esto puede parecer contradictorio, ya que en nuestra aplicación sí queremos que los enlaces se conecten entre rectángulos. Sin embargo, es más conveniente que hagan esto a través de los ports, que explicaremos a continuación.

Los ports son puertos que se añaden a los equipos, desde los cuales pueden crearse enlaces arrastrando y soltando el cursor. Estos ports hay que definirlos e incluirlos en nuestros rectángulos, para lo cual tenemos que cambiar ligeramente el constructor, como ilustra el código 4.3. Los parámetros position, resize y attr, aunque no aparecen en este código, quedan intactos, pero se omiten en este caso para centrarnos en los ports.

```
const port1 = {
  group: "inputs"
};

const port2 = {
  group: "outputs"
};

const visualizacion = new joint.shapes.standard.Rectangle({
  ports: {
    groups: {
      inputs: {
        position: {
          name: "left",
          args: {}
        },
        attrs: { circle: { magnet: true } }
      },
      outputs: {
        position: {
          name: "right",
          args: {}
        },
        attrs: { circle: { magnet: true } }
      }
    },
    items: [port1, port2]
  }
});
```

Código 4.3 Ports en JointJS.

Lo primero es definir dos puertos, que no son más que objetos de JavaScript, con un sólo parámetro, groups, que indican a qué grupo de puertos pertenecen. Estos grupos debemos definirlos en el constructor del rectángulo, y en nuestro caso creamos dos grupos, los inputs, que se posicionarán a la izquierda, y los outputs, que lo harán a la derecha, como establece su parámetro position. Además, como vemos, a los puertos sí que le asignamos el parámetro magnet en true, ya que sí que queremos que los enlaces se conecten a los puertos. Ésta es la razón por la que en los parámetros del rectángulo lo establecimos en false, ya que en caso contrario,

los enlaces podrían haberse conectado a cualquier parte del equipo, mientras que sólo nos interesa que se conecten en los ports. Finalmente, una vez establecidos los grupos, suministramos el array items con los nombres de los puertos que queremos añadir, port1 y port2 en este caso, lo que creará dos puertos, un input y un output, por cada equipo que creemos.

Una vez hecho esto, tenemos un diagrama completamente interactivo con la posibilidad de crear equipos y, una vez creados estos, crear enlaces a través de los ports. Ahora solo falta ver cómo hacer que el diagrama mande información a nuestro modelo sobre las interacciones del usuario.

4.3 Eventos

JointJS ofrece la posibilidad de añadir eventos, tanto en el paper o en el graph, para diferentes acciones del usuario, y ejecutar código cada vez que se produce uno de estos eventos. En la documentación se encuentran disponibles todos los eventos posibles, pero ilustraremos dos aquí para explicar su uso.

```
1   graph.on("remove", cell => {
2     if (cell.isElement()) {
3       store.dispatch(doBorrarEquipo(cell.id));
4     }
5     if (cell.isLink()) {
6       store.dispatch(doBorrarEnlace(cell.id));
7     }
8   });
9
10 paper.on("element:pointerdown", cellView => {
11     limpiarSeleccionados();
12     cellView.highlight(cellView);
13     store.dispatch(doCambiarSeleccion(cellView.model.id));
14   });
```

Código 4.4 Eventos en JointJS.

Como vemos en el código 4.4 podemos definir eventos tanto para el graph como para el paper. Los eventos del graph están relacionados con cambios en el modelo, como por ejemplo, añadir o eliminar un equipo o enlace (JointJS llama "cell" a cada objeto del graph). A cada definición de evento se le suministra primero el tipo de evento, y segundo un callback, una función a ejecutar cuando ocurra el evento. En el primer evento que hemos definido, esperamos el evento "remove", es decir, eliminación de un equipo o enlace. La función callback recibe el cell que se ha eliminado, comprueba si es un equipo (element en JointJS) o enlace (link) y ejecuta la acción apropiada en el modelo de Redux. En el segundo ejemplo, este en el paper, esperamos el evento "element:pointerdown", que se ejecuta cuando se hace click en un equipo. Al hacer esto, resaltamos (highlight) la vista del equipo pulsado (cellView) y ejecutamos la acción apropiada del modelo. Además, también se ejecuta la función limpiarSeleccionados, que se asegura de que no hay ningún otro equipo resaltado cuando resaltamos el actual.

Conociendo los eventos, ya tenemos todos la información necesaria para crear un diagrama interactivo con JointJS. En la siguiente sección, explicaremos algunos detalles sobre la implementación de JointJS en una aplicación con React.

4.4 Implementación en React

Para la implementación del diagrama en una aplicación actual, tenemos dos opciones. La primera es añadir manualmente un nodo HTML en el archivo principal de la web (index.html), para poder suministrarlo en la creación del paper. Esta opción es la que se ha utilizado en nuestro caso por ser más sencilla, ya que una vez añadido el nodo solo tenemos que encapsular la creación del paper y del graph en una función y asegurarnos que esta función se ejecuta al cargar la página. Para crear o borrar equipos, solo tenemos que hacer que sus respectivos botones en la interfaz ejecuten dichas funciones y conseguiremos el comportamiento deseado. Este método se ilustra en el código 4.5.

```
1 //diagrama.js
2 function crearDiagrama() {
3   graph = new joint.dia.Graph();
4
5   paper = new joint.dia.Paper({ /* ... */});
6 }
7
8 //index.js
9 /* ... */
10 crearDiagrama();
```

Código 4.5 Eventos en JointJS.

La segunda opción, más elegante y oficialmente soportada por React, es crear un componente Diagrama, y solicitar a React su referencia, es decir, el nombre que React le ha dado al nodo en el que se aloja su componente. Este método viene explicado en la documentación oficial de React, y aunque, como hemos dicho, es más elegante, no aporta realmente ninguna ventaja con respecto al método utilizado. Para futuras mejoras del diagrama en las que tengamos que implementar opciones más avanzadas, es posible que sí nos sea útil utilizar las referencias de React, pero, como se explicará en las conclusiones de esta memoria, lo ideal en el futuro sería migrar a una librería de diagramas programada enteramente con React, que nos ahorraría muchos problemas en el futuro.

5 Modelo de resolución de ecuaciones

El modelo de resolución de ecuaciones, desarrollado por el tutor de este Trabajo, es un módulo dedicado a la resolución de sistemas de ecuaciones no lineales mediante un proceso iterativo. Su código es independiente al del resto de la aplicación, por lo que puede reutilizarse en otras aplicaciones en JavaScript sin ningún problema. Por tanto, dedicaremos este capítulo a explicar su funcionamiento, no sólo para entenderlo, sino para que otros desarrolladores puedan utilizarlo en sus aplicaciones.

Lo primero a tener en cuenta es que el paradigma utilizado en la programación del modelo es la programación orientada a objetos, por lo que cada elemento programado es un objeto en JavaScript con distintas propiedades y métodos. Explicaremos a continuación los distintos objetos que componen el modelo.

5.1 Variables, datos e incógnitas

5.1.1 Variable

El primer objeto a destacar es el objeto Variable, cuyo constructor acepta un símbolo y un valor. El símbolo no es más que un carácter con el cual se representará la variable en las expresiones matemáticas, que veremos en la siguiente sección. El valor es el valor numérico que tomará dicha variable. Este objeto tiene implementado cuatro métodos, dos getters y dos setters, del valor y el símbolo de la variable. En caso de no aportar un símbolo en el constructor, se le asignará x por defecto, mientras que si no aportamos un valor, se le asignará undefined a éste. El objeto Variable no se utiliza directamente en las ecuaciones, sino que se hace a través de los objetos Dato e Incógnita, que heredan del objeto Variable y que explicaremos a continuación.

Tabla 5.1 API del objeto Variable.

setSimbolo(simbolo)	Recibe una cadena de texto que se establece como nuevo símbolo para representar a la variable
getSimbolo()	Devuelve el símbolo actual de la variable
setValor(valor)	Recibe un número y lo establece como nuevo valor
getValor()	Devuelve el valor actual de la variable

5.1.2 Dato

Los datos son variables cuyo valor numérico conocemos, y debido a que el objeto Variable ya contiene todos los métodos necesarios para asignar un valor numérico y un símbolo, no añaden ninguna funcionalidad extra, sólo se aseguran de que su valor no es undefined, y funcionan como un alias para indicar que conocemos el valor de dicha Variable, lo que será útil a la hora de programar las ecuaciones. Su API es idéntica a la del objeto Variable

5.1.3 Incógnita

Por otro lado, las incógnitas son variables cuyo valor desconocemos, por lo que aunque heredan del objeto Variable, incorporan varios cambios en el constructor. En concreto, su constructor requiere como segundo argumento el valor inicial para el proceso iterativo en lugar de el valor de la Variable, que se establece

automáticamente como undefined. Además, establece límites superiores e inferiores de su valor, que todavía no son modificables por el usuario, por lo que se asignan a más y menos infinito, respectivamente. Su API es, de nuevo, idéntica a la del objeto variable.

5.2 Expresiones y ecuaciones

5.2.1 Expresión

Para poder programar las ecuaciones, primero creamos un objeto Expresion, que representa una expresión matemática de cualquier tipo. Este objeto se sirve de la librería expr-eval para diseccionar dicha expresión y extraer sus variables. En la tabla 5.2 se muestran los métodos disponibles para este objeto.

Tabla 5.2 API del objeto Expresion.

setExpresion(expresion)	Recibe una cadena de texto y la guarda como nueva expresión en el objeto
getListaVariables()	Devuelve la lista de todas las variables encontradas en la expresión.
calcular(datos)	Dado un array de objetos Dato, devuelve el valor numérico de la expresión tras sustituir dichos datos

5.2.2 Ecuaciones

Usando el objeto Expresion, definimos el objeto Ecuacion como dos expresiones separadas por un símbolo =, es decir, un objeto Expresion en cada miembro del objeto Ecuacion. El constructor devolverá un error si no encuentra dicho símbolo igual o si encuentra más de uno, para asegurar que lo que recibe es realmente una ecuación. Además, en el constructor se asigna un valor al número de cifras significativas deseadas y el número de iteraciones máximo. Los métodos que presenta se muestran en la tabla 5.3. El código no se incluye en este texto debido a su longitud, pero está disponible en el código fuente del software. Lo único destacable de este código, además de lo ya explicado, es que incluye un método privado, _haConvergido utilizado por el método resolver para realizar los cálculos iterativos.

Tabla 5.3 API del objeto Ecuacion.

setEcuacion(ecuacion)	Recibe una ecuación como cadena de texto y guarda un objeto Expresion para cada miembro de la ecuación
getListaVariables()	Devuelve la lista de todas las variables encontradas en la ecuación.
getResiduo(variables)	Devuelve el residuo de la ecuación para una lista de variables dada.
resolver()	Resuelve numéricamente la ecuación utilizando un proceso iterativo.

5.3 Sistema de ecuaciones y método Gauss Jordan

Finalmente, tenemos el objeto SistemaEcuaciones, que en su método resolver, el único método público que posee, recibe tres arrays: uno de objetos Ecuacion, otro de objetos Incognita y otro de objetos Dato. Recibidos estos objetos, resuelve iterativamente el sistema de ecuaciones utilizando el método Gauss Jordan, que se programa separadamente, y devuelve un array con los valores numéricos obtenidos para cada incógnita, en el mismo orden en el que se han aportado los objetos Incognita. De nuevo, su código no se muestra en este texto debido a su longitud, pero está disponible en el código fuente de la aplicación.

5.4 Ejemplos de uso

En esta sección mostraremos con dos pequeños ejemplos el manejo de la API que se ha mostrado en las secciones anteriores.

5.4.1 Cálculo de una expresión

Supongamos que tenemos la expresión $x^2 + y + 3$ y queremos calcular su resultado para $x = 2$ e $y = 4$. Para ello lo primero que tendremos que hacer es crear un objeto Expresion que contenga la expresión que queremos calcular, como muestra el código 5.1.

```
let expresion = new Expresion("x^2 + y + 3")
```

Código 5.1 Creación de objeto Expresion.

Ahora tendremos que crear un array de objetos Variable para cada variable que aparece en la expresión. Esto lo podemos hacer de dos formas: la primera es creándolas manualmente, asegurándonos de que su símbolo coincide con el símbolo de las variables en la expresión, y la segunda es utilizando el método getListaVariables para obtener un array de objetos Variable y posteriormente establecerles los valores deseados para el cálculo. Ambas formas se ilustran en el código 5.2. Cabe destacar que, aunque aquí utilicemos el objeto Variable, el objeto Dato sería igualmente válido.

```
//Primera forma
let variables = []
variables.push(new Variable("x", 2))
variables.push(new Variable("y", 4))

//Segunda forma
let variables = expresion.getListaVariables()
variables[0].setValor(2)
variables[1].setValor(4)
```

Código 5.2 Creación de lista de variables.

Creada la lista de variables, utilizamos el método calcular para obtener el resultado numérico, como muestra el código 5.3.

```
expresion.calcular(variables) // Devuelve 11
```

Código 5.3 Cálculo de una expresión.

5.4.2 Resolución de sistema de ecuaciones

Ahora supongamos que tenemos que resolver un sistema de ecuaciones de tres incógnitas, pero en el que conocemos el valor de una de ellas:

$$x^2 + y^2 - 2 + z = 0$$
$$x - y + 1 = z$$
$$z = 1$$

Para ello, tendremos que crear un objeto SistemaEcuaciones, un array de objetos Ecuacion, otro de objetos Incognita, y un último de objetos Dato, que contengan la información de nuestro problema. Finalmente, utilizamos el método resolver del objeto SistemaEcuaciones para obtener el resultado. El procedimiento se muestra en el código 5.4

```
let sistema = new SistemaEcuaciones()

// Creamos array de ecuaciones
let ecuaciones = [
    new Ecuacion("x^2 + y^2 -2 + z = 0"),
    new Ecuacion("x - y + 1 = z")
```

```
]

// Creamos array de incógnitas con valor inicial 1
let incognitas = [
    new Incognita("x", 1),
    new Incognita("y", 1)
]

// Creamos array de datos
let datos = [
    new Dato("z", 1)
]

//Resolvemos el sistema
let solucion = sistema.resolver(ecuaciones, incognitas, datos)

// solucion = [0.7071, 0.7071]

```

Código 5.4 Cálculo de un sistema de ecuaciones.

Como hemos visto, este código permite resolver cualquier sistema de ecuaciones, es totalmente independiente de la aplicación desarrollada en este trabajo, permitiendo su uso en otras aplicaciones. Además, aunque no se utilizan en esta aplicación, se pueden configurar los límites de sus variables, el residuo máximo deseado y el número de iteraciones máximo, haciéndolo un módulo muy versátil para cualquier aplicación.

6 Ejemplos de uso

En este capítulo resolveremos tres problemas del campo de la ingeniería energética para ilustrar el uso de la aplicación. Cabe destacar que, como se ha comentado anteriormente, en su estado actual el software no permite calcular propiedades de fluidos, por lo que las obtendremos externamente y las introduciremos como datos.

6.1 Ciclo simple de refrigeración

Para este ejemplo, resolveremos un problema con el siguiente enunciado:

> Una máquina frigorífica utiliza el ciclo estándar de compresión de vapor. Produce 50 kW de refrigeración utilizando como refrigerante R-22, si su temperatura de condensación es 40°C y la de evaporación -10°C, calcular:
>
> **1.** Efecto frigorífico
>
> **2.** Caudal de refrigerante
>
> **3.** Potencia de compresión
>
> **4.** Coeficiente de eficiencia energética

En primer lugar, crearemos el esquema apropiado de un ciclo de refrigeración simple en la aplicación, dando resultado a lo que vemos en la figura 6.1

Creados los equipos y los enlaces, tendremos que añadir los datos, incógnitas, ecuaciones y relaciones necesarios para resolver el problema. A continuación se muestran tablas con la información a rellenar en cada equipo. En este caso, sólo tendremos que utilizar el evaporador, el compresor, y el enlace entre ambos.

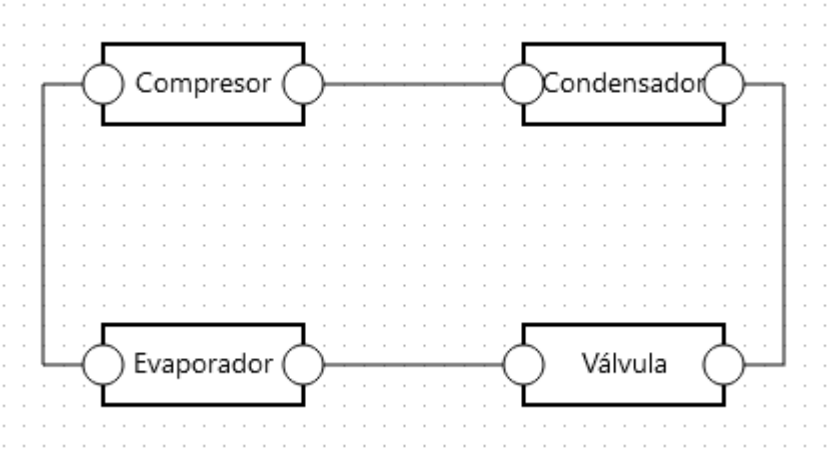

Figura 6.1 Esquema del ciclo simple de refrigeración.

6.1.1 Evaporador

Tabla 6.1 Evaporador del problema 1.

Datos	Incógnitas	Ecuaciones
$h1 = 401$	mr	$Qf = mr(h1 - h4)$
$h4 = 249$	$Efrig$	$Efrig = h1 - h4$
$Qf = 50$		

6.1.2 Compresor

Tabla 6.2 Compresor del problema 1.

Datos	Incógnitas	Ecuaciones
$h2 = 438$	mr	$Pc = mr(h2 - h1)$
	Pc	$COP = Qf/Pc$
	$h1$	
	Qf	
	COP	

6.1.3 Enlace evaporador-compresor

Tabla 6.3 Relaciones del problema 1.

Evaporador	Compresor
mr	mr
h1	h1
Qf	Qf

6.1.4 Resultados

Añadida la información necesaria, podemos finalmente pulsar el botón Resolver para obtener los resultados, que se muestran en la tabla 6.4.

Tabla 6.4 Resultados del problema 1.

Incógnita	Resultado
$Efrig$	152
mr	0.3289
Pc	12.17
COP	4.108

6.2 Turbina de gas

Una central eléctrica de turbina de gas opera con un ciclo Brayton, con una relación de presiones de 8. La temperatura del gas es de 300 K en la entrada del compresor y de 1300K en la entrada de la turbina. Suponiendo que el fluido es aire seco, calcular:

1. Trabajos y calores específicos intercambiados por cada equipo
2. Rendimiento térmico del ciclo

De nuevo, lo primero será crear los equipos y enlaces necesarios para el problema, que se representan en la figura 6.2. A continuación iremos equipo por equipo, y posteriormente enlace por enlace, añadiendo la información necesaria para finalmente obtener los resultados.

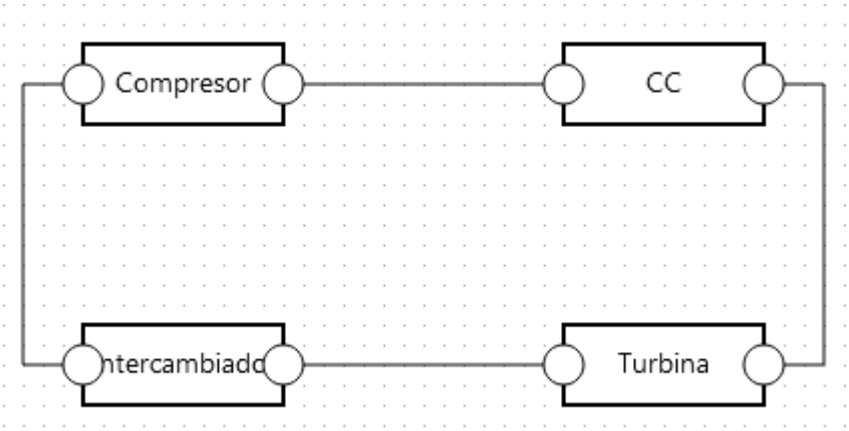

Figura 6.2 Esquema del ciclo Brayton.

6.2.1 Cámara de combustión

Tabla 6.5 Cámara de combustión del problema 2.

Datos	Incógnitas	Ecuaciones
$h3 = 1395$	$h2$	$Qcc = h3 - h2$
	Qcc	$rend = (Wt - Wc)/Qcc$
	Wt	
	Wc	
	$rend$	

6.2.2 Compresor

Tabla 6.6 Compresor del problema 2.

Datos	Incógnitas	Ecuaciones
$h2 = 544$	$h1$	$Wc = h2 - h1$
	Wc	

6.2.3 Intercambiador

Tabla 6.7 Intercambiador del problema 2.

Datos	Incógnitas	Ecuaciones
$h1 = 300$	$h4$	$Qi = h4 - h1$
	Qi	

6.2.4 Turbina

Tabla 6.8 Turbina del problema 2.

Datos	Incógnitas	Ecuaciones
$h4 = 789$	$h3$	$Wt = h3 - h4$
	Wt	

6.2.5 Enlace cámara de combustión - turbina

Tabla 6.9 Enlace cámara de combustión - turbina.

Cámara de combustión	Turbina
$h3$	$h3$
Wt	Wt

6.2.6 Enlace compresor - cámara de combustión

Tabla 6.10 Enlace compresor - cámara de combustión.

Compresor	Cámara de combustión
$h2$	$h2$
Wc	Wc

6.2.7 Enlace intercambiador - compresor

Tabla 6.11 Enlace intercambiador - compresor.

Intercambiador	Compresor
$h1$	$h1$
Wc	Wc

6.2.8 Enlace turbina - intercambiador

Tabla 6.12 Enlace turbina - intercambiador.

Turbina	Intercambiador
$h4$	$h4$

6.2.9 Resultados

Tabla 6.13 Resultados del problema 2.

Incógnita	Resultado
Wc	244
Qcc	851
Wt	606
Qi	489
$rend$	4254

6.3 Problema 3

Dado un caudal de 10 kg/s de una sustancia "A" a 20ºC con calor específico de 1 kJ/kgK, se instala un reactor que convierte la sustancia "A" en otra "B" según la reacción A->B, cuya conversión depende de la temperatura de los reactivos según la expresión $x = \frac{0{,}5T-50}{150}$ para T entre 250ºC y 400ºC. Para calentar el reactivo hasta la temperatura adecuada, se instala un intercambiador a la entrada del reactor. Para obtener 8 kg/s de "B", calcular:

1. Temperatura de entrada de los reactivos
2. Conversión
3. Calor aportado en el intercambiador

En este problema, además del intercambiador y del reactor, añadiremos dos equipos que representarán la entrada y la salida de los caudales, obteniendo un esquema como el de la figura 6.3

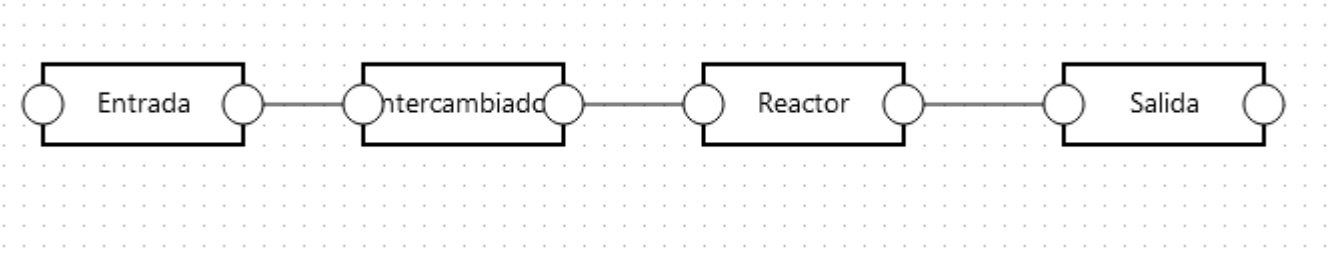

Figura 6.3 Esquema del problema 3.

Una vez más, a continuación se muestra la información a añadir en cada equipo para obtener los resultados deseados.

6.3.1 Entrada

Tabla 6.14 Entrada del problema 3.

Datos	Incógnitas	Ecuaciones
$m = 10$		
$T = 20$		

6.3.2 Intercambiador

Tabla 6.15 Intercambiador del problema 3.

Datos	Incógnitas	Ecuaciones
$Cp = 1$	Q	$Q = m * Cp * (Ts - Te)$
	m	
	Ts	
	Te	

6.3.3 Reactor

Tabla 6.16 Reactor del problema 3.

Datos	Incógnitas	Ecuaciones
	x	$x = (0.5 * T - 50/150)$
	T	$mb = x * me$
	mb	
	me	

6.3.4 Salida

Tabla 6.17 Salida del problema 3.

Datos	Incógnitas	Ecuaciones
$mb = 8$	me	$me = mb$

6.3.5 Enlace entrada-intercambiador

Tabla 6.18 Enlace entrada-intercambiador.

Entrada	Intercambiador
m	m
T	Te

6.3.6 Enlace intercambiador-reactor

Tabla 6.19 Enlace intercambiador-reactor.

Intercambiador	Reactor
Ts	T
m	me

6.3.7 Enlace intercambiador-salida

Tabla 6.20 Enlace reactor-salida.

Reactor	Salida
mb	me

6.3.8 Resultados

Tabla 6.21 Resultados del problema 3.

Incógnita	Resultado
T	340
x	0.8
Q	3200

7 Conclusiones y recomendaciones

Como mencionamos en la introducción de esta memoria, el resultado de este Trabajo Fin de Grado ha sido un prototipo de aplicación para resolver problemas de ingeniería, que aún necesita más características para ser realmente útil en el campo de la ingeniería. Aunque técnicamente ya es posible resolver cualquier tipo de problema con ella, ya que dispone de un modelo de resolución matemático capaz de ello, existen algunas características por añadir que facilitarían mucho la labor del usuario de la aplicación. Por tanto, es el objetivo de este último capítulo elaborar una lista no exhaustiva de características a añadir, así como algunas recomendaciones del autor para llevar a cabo su implementación. Esta lista no tiene ningún orden específico, dejando a discreción de futuros desarrolladores el orden de implementación de las características que aquí se indican.

Migración del diagrama a React Como se comentó en el capítulo dedicado a la librería gráfica, utilizar una librería externa a React para los diagramas tiene sus inconvenientes, principalmente que la librería utilizada (JointJS) utiliza un modelo distinto al de nuestra aplicación. Por tanto, migrar el diagrama a React simplificaría en gran medida el código de la aplicación. Una opción es hacer esto manualmente creando los componentes desde cero y utilizando alguna herramienta como React-DnD para gestionar la interacción con los equipos. No obstante, el principal problema que nos encontramos aquí son los enlaces, cuya vista es difícil de programar en React para obtener un comportamiento similar al actual, ya que no es trivial hacer que estos encuentren una ruta ortogonal de un equipo a otro, y que sigan el cursor mientras se crea dicha ruta. La opción más factible que ha encontrado el autor de este texto es la librería PathFinding.js, que dispone de distintos algoritmos para trazar rutas entre dos puntos, pero la implementación de esta librería en la aplicación requiere de un considerable esfuerzo por parte del programador. La otra opción es utilizar una librería de diagramas ya programada en React, que aunque nos impondría algunos límites al no poder hacerla a nuestra medida, ahorra bastante trabajo al programador. En el momento de redacción de este texto, la única librería adecuada que se ha encontrado es Storm React Diagrams, que podría servir para nuestro caso. No obstante, la ventaja de utilizar React es que su comunidad es muy activa, por lo que es posible que en un futuro cercano existan nuevas librerías capaces de realizar diagramas interactivos.

Cálculo de propiedades de fluidos Uno de los grandes inconvenientes de la aplicación desarrollada en este trabajo es que no incluye la posibilidad de calcular propiedades de fluidos comúnmente utilizados en procesos térmicos. Es posible utilizar calculadoras externas, y añadir las propiedades como datos, pero lo ideal sería poder hacerlo desde la misma aplicación. Una librería que incluye funciones en JavaScript para ello es CoolProp, con la que el autor de este trabajo ha realizado un pequeño ejemplo llamado coolpropweb. Esta librería debería servir para lo que necesitamos, pero su implementación requiere algo de esfuerzo ya que la documentación de la librería es algo deficiente, además de tener que gestionar la conversión de unidades por separado.

Múltiples conexiones por equipo Los equipos que se encuentran ya programados en la aplicación sólo disponen de dos puertos, uno de entrada y otro de salida, pero esto será insuficiente para muchas aplicaciones. Es necesario que el usuario pueda añadir o quitar conexiones, y colocarlas en el sitio apropiado.

Mejora de las expresiones matemáticas Aunque funcional, el actual sistema para añadir datos, incógnitas, ecuaciones y relaciones, es bastante primitivo, y requeriría de algunos cambios para mejorar

la experiencia del usuario. En concreto, sería interesante utilizar alguna librería como MathJax para mostrar las expresiones más elegantemente, y de opciones para editar las ecuaciones y los nombres de las incógnitas y los datos para no tener que borrarlos y escribirlos de nuevo.

Variables globales Una carencia importante de la aplicación actual es la incapacidad de definir variables disponibles en todos los equipos. Reutilizar las variables es posible mediante relaciones, pero la implementación de un apartado donde añadir variables globales ahorraría trabajo al usuario. Esta característica es probablemente bastante fácil de implementar utilizando simples componentes de React.

Relaciones El actual menú para añadir relaciones en los enlaces no parece ser el ideal, ya que resulta algo tedioso de utilizar comparado con otros como los que el software TRSNYS dispone. Además, también sería interesante que las relaciones no fueran sólo relaciones de igualdad, sino relaciones matemáticas simples, como multiplicar por una constante, para simular pérdidas por tuberías.

Equipos predefinidos Actualmente es necesario crear todas las ecuaciones desde cero para resolver el problema, pero ya que las ecuaciones utilizadas para distintos equipos se repiten comúnmente, sería interesante añadir un sistema de equipos predefinidos, como intercambiadores, compresores, turbinas, etc. para que la aplicación añadiese automáticamente las ecuaciones pertinentes. En un principio el programador podría añadir los equipos más comunes en el código, pero lo ideal sería implementar un sistema donde el usuario creara los equipos a su medida y los pudiese utilizar cuantas veces quiera.

Guardado de datos Actualmente la aplicación no dispone de ninguna forma para guardar problemas hechos en ella. Existen varios servicios web para implementar persistencia de datos en aplicaciones sin tener que contratar un servidor centralizado. En concreto, Google ofrece uno bastante popular llamado Firebase. Además del guardado, podría ser interesante implementar un sistema de identificación para que un usuario pueda crear una cuenta y guardar sus problemas, así como un sistema de enlaces únicos con los que poder compartir problemas sólo con el enlace.

Lista de variables y ecuaciones En caso de que el modelo de resolución se tope con algún error, resulta bastante inconveniente tener que ir equipo por equipo comprobando las variables y las ecuaciones. Por tanto, implementar una ventana de variables y ecuaciones como la que incluye el software EES sería bastante interesante y es posible realizarlo utilizando React.

Portar la aplicación a Electron Electron es una librería que permite portar aplicaciones web al escritorio, permitiendo su uso offline. Hacer esto puede ser beneficioso, ya que no siempre se dispone de una conexión de internet estable. El proceso debería ser relativamente sencillo siguiendo la documentación de la web oficial.

Símbolos personalizadas en los equipos Actualmente todos los equipos son rectángulos, que aunque es aceptable para un prototipo, no es lo ideal si queremos que los esquemas del diagrama se entiendan con claridad. Para mejorar esto, sería necesario permitir a los usuarios cambiar los símbolos de cada equipo por unos que ilustren mejor lo que representan. La norma ISO 14617 incluye una lista de símbolos normalizados para los equipos más comúnmente utilizados.

Opción de rehacer y deshacer En la aplicación actual no se pueden corregir errores de los usuarios de forma automática, por lo que sería conveniente implementar una opción de rehacer y deshacer para ahorrar tiempo al usuario. La ventaja de haber usado Redux es que permite hacer esto de manera muy sencilla, ya que podemos simplemente guardar un array con varios estados anteriores, e ir cambiando entre ellos conforme el usuario lo necesite.

Mensajes de error Actualmente sólo existen dos posibles mensajes de error que el programa puede devolver al usuario en caso de que no consiga resolver las ecuaciones. Sin embargo, el número de errores posibles es mucho mayor, por lo que sería conveniente programar mensajes de error descriptivos para facilitar la solución al usuario.

Mejora general del flujo de trabajo Esta es la recomendación más abstracta de esta lista, y el motivo es que mejorar el flujo de trabajo del usuario con la aplicación es un proceso constante y que requiere de gran esfuerzo. Actualmente, debido en parte a que la aplicación no presenta muchas características que aquí se han comentado, no es especialmente cómodo para el usuario resolver problemas con la aplicación. Por ello, es necesario testear la aplicación en profundidad e inspirarse en otras aplicaciones similares, como TRNSYS o ASPEN, para implementar nuevas características que hagan este software más fácil de utilizar

Como se dijo al comienzo, esta es una lista no exhaustiva, de ideas que el autor de este texto ha tenido durante el desarrollo del prototipo inicial. Conforme la aplicación continúe su desarrollo, es probable que surjan nuevas ideas para características útiles en la aplicación. De momento, dejamos aquí el trabajo realizado, y esperamos que la aplicación le sea útil tanto a ingenieros como estudiantes de ingeniería, en el presente y el futuro de la misma.

Índice de Figuras

Índice de Tablas

Índice de Códigos

Bibliografía

[1] “GitHub - facebook/create-react-app: Set up a modern web app by running one command.” [Online]. Available: https://github.com/facebook/create-react-app

[2] “Ant Design of React - Ant Design.” [Online]. Available: https://ant.design/docs/react/introduce

[3] “Documentación | Node.js.” [Online]. Available: https://nodejs.org/es/docs/

[4] “Getting Started with React - An Overview and Walkthrough Tutorial – Tania Rascia.” [Online]. Available: https://www.taniarascia.com/getting-started-with-react/

[5] “Git - Documentation.” [Online]. Available: https://git-scm.com/doc

[6] “GitHub - the-road-to-learn-react/the-road-to-learn-react-spanish: The Road to learn React - Spanish Translation.” [Online]. Available: https://github.com/the-road-to-learn-react/the-road-to-learn-react-spanish

[7] “Integrating with Other Libraries – React.” [Online]. Available: https://reactjs.org/docs/integrating-with-other-libraries.html

[8] “Joint API.” [Online]. Available: https://resources.jointjs.com/docs/jointjs/v3.0/joint.html

[9] “JointJS - JavaScript diagramming library - Getting started.” [Online]. Available: https://resources.jointjs.com/tutorial

[10] “JointJS: Visualize and interact with diagrams and graphs.” [Online]. Available: https://www.jointjs.com/opensource

[11] “npm Documentation.” [Online]. Available: https://docs.npmjs.com/

[12] “Ramda Documentation.” [Online]. Available: https://ramdajs.com/

[13] “React – A JavaScript library for building user interfaces.” [Online]. Available: https://reactjs.org/

[14] “Redux Tutorial by Dan Abramov on egghead.io.” [Online]. Available: https://egghead.io/courses/getting-started-with-redux

[15] “Redux · A Predictable State Container for JS Apps.” [Online]. Available: https://redux.js.org/