UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Representación gráfica interactiva de funciones matemáticas, figuras geométricas y animaciones en la Web.

Proyecto MateFun

Diego Rey

diego.rey@fing.edu.uy

Ignacio Fagian

jose.ignacio.fagian@fing.edu.uy

Leonel Rosano

leonel.rosano@fing.edu.uy

Tutores: Marcos Viera y Gonzalo Tejera

Proyecto de Grado en Ingeniería en Computación
Facultad de Ingeniería, Instituto de Computación
Universidad de la República

Montevideo – Uruguay
Abril de 2019

RESUMEN

El presente proyecto de tesis tiene como objetivo analizar e implementar una solución para la representación gráfica interactiva de funciones matemáticas, figuras y transformaciones geométricas en la aplicación web del proyecto MateFun. Como parte del objetivo planteado, hemos analizado las diferentes tecnologías utilizadas actualmente para el renderizado de gráficos interactivos en la Web, y se ha implementado una solución con base en SVG, D3.js y FunctionPlot para la representación gráfica en 2D y Three.js para la representación gráfica en 3D. Hemos decidido seguir un diseño basado en módulos que ha facilitado el desarrollo y la integración con la aplicación web de MateFun. Para esto, se ha extendido la biblioteca FunctionPlot con las funcionalidades de representar y trabajar con figuras geométricas y con conjuntos dominio y codominio de una función matemática. También hemos creado una nueva biblioteca sobre Three.js para el trabajo con figuras geométricas, curvas paramétricas y superficies en 3D, mientras que en el ámbito de la aplicación Web de MateFun, se han creado dos nuevos módulos que conectan estas dos bibliotecas y la aplicación.

Los principales aportes de este trabajo se centran en el estudio del marco teórico necesario para la representación de los elementos gráficos mencionados, así como el trabajo realizado en las dos bibliotecas, que puede utilizarse en otro sistema, extenderse para dar soporte a nuevos elementos y funcionalidades, o como base para un nuevo desarrollo que mejore el actual.

Palabras claves:
Representación gráfica, Funciones matemáticas, Figuras geométricas, Function Plot, WebGL.

Tabla de contenidos

Capítulo 1

Introducción.

Los últimos años se han caracterizado por un avance acelerado y significativo de la tecnología, pasando a formar una parte muy importante en nuestras vidas. El aumento en el acceso a Internet, así como la velocidad de conexión y la variedad cada vez mayor de dispositivos conectados, han hecho posible que más personas utilicen esta red para comunicarse, informarse, resolver tareas o simplemente entretenerse.

Estos cambios también se ven reflejados en las aulas educativas. La presencia de Tecnologías de Información y Comunicación (TIC) en ámbitos educativos, ponen a disposición nuevas herramientas que pueden beneficiar los procesos de enseñanza y aprendizaje tradicionales. La tecnología está cambiando el mercado laboral y educativo, redefiniendo los procesos laborales y exigiendo desarrollar nuevas capacidades necesarias para que las personas mejoren sus competencias.

Paralelamente, la enseñanza y aprendizaje de la matemática es tema de preocupación en la actualidad y los resultados de las últimas pruebas PISA, realizadas en 2015, no muestran avances en esta área. Con mayores volúmenes de información disponibles para una creciente cantidad de personas, y con más interconexiones entre los distintos ámbitos de la actividad y del conocimiento humano, es fundamental preparar personas que sean capaces de desarrollar habilidades que les permitan construir y fundamentar ideas propias, y contribuir a la cultura de la comprensión, el análisis crítico y la reflexión (13).

En este contexto, se diseñó y desarrolló un lenguaje de programación funcional simple denominado MateFun, creado para ser utilizado en centros edu-

cativos de secundaria, el cual permitie relacionar conceptos y procedimientos matemáticos con programación y viceversa. Por otra parte, el proyecto MateFun también cuenta con un entorno web desarrollado como proyecto de Tesis por Gonzalo Cameto y Martín Méndez (8), que permite la interacción con el lenguaje MateFun desde una aplicación web.

En nuestro caso, el proyecto de tesis que presentamos en este trabajo surge con el objetivo de mejorar la representación gráfica de funciones matemáticas, figuras geométricas y animaciones en el entorno web del proyecto MateFun. Para su resolución, se decidió utilizar la biblioteca FunctionPlot en la representación en 2D y extender las funcionalidades de esta biblioteca, agregando la capacidad de representar figuras geométricas y funciones definidas sobre dominios y codominios discretos, enumerados, y subconjuntos de los Reales. Este trabajo fue integrado por completo a MateFun, creando un nuevo módulo (Módulo 2D) en el entorno web que actúa de intermediario entre la biblioteca y lo que el usuario desea representar en 2D utilizando el lenguaje MateFun. En el caso de la representación en 3D, se creó una nueva biblioteca denominada Graph3D, que fue desarrollada utilizando Three.js y que es capaz de representar figuras geométricas, curvas paramétricas y superficies en 3D. Al igual que con FunctionPlot, el trabajo realizado en la biblioteca Graph 3D, fue integrado al entorno web de MateFun utilizando un nuevo módulo (Módulo 3D), aunque en este caso, y por razones de alcance de este proyecto, únicamente se integró el trabajo realizado para representar figuras en 3D.

El presente trabajo se encuentra organizado de la siguiente manera. En el Capítulo 2 se presenta el estado del arte, con una breve revisión histórica y un análisis de las diferentes tecnologías utilizadas actualmente para crear gráficos interactivos en la Web. En el Capítulo 3 se definen el estado del lenguaje MateFun y el módulo de gráficas al comenzar este proyecto. El Capítulo 4, presenta las decisiones y arquitectura, así como el trabajo realizado en cada una de las bibliotecas y en los módulos creados en el entorno web. Finalmente, en el capítulo 5 se presentan las conclusiones y las líneas de trabajo a futuro.

Capítulo 2

Estado del arte.

El presente capítulo comenzará con una breve revisión histórica de cómo ha sido la evolución de la Web en el ámbito de las tecnologías relacionadas a la representación gráfica, posteriormente se revisarán las decisiones que se han tomado acerca de esta temática, y cuál es la tendencia actual seguida por la comunidad y por las principales empresas interesadas en la Web. Luego, se describirán las características que presentan las tecnologías usadas actualmente, para finalmente en la última sección, analizar algunas de las bibliotecas que hacen uso de estas tecnologías, y que pueden ser útiles para la representación gráfica de figuras geométricas y funciones matemáticas en 2D y 3D.

2.1. Revisión histórica.

Desde la creación de la Web en 1989 hasta la fecha, la misma se ha ido transformando, los lenguajes y navegadores utilizados han ido evolucionando, se han creado nuevas tecnologías y estándares, y se han dejado de utilizar otras. En octubre de 1990, el principal creador de la Web, Tim Bernes Lee, desarrolla el primer servidor de páginas web y la primera versión de lo que luego sería el lenguaje HTML (9). A fines de 1994, Bernes Lee funda World Wide Web Consortium (W3C) (4) , un consorcio que genera recomendaciones y estándares para el desarrollo de la Web, mientras que un año más tarde, en 1995, se crea la primera versión oficial de HTML, denominada HTML 2.0, pero no es hasta la versión 3.2 recomendada por W3C en enero de 1997, que HTML incluye etiquetas para trabajar con tablas, imágenes, hojas de estilos, applets y otras etiquetas que enriquecen el entorno gráfico de la Web (15). En

ese mismo año, la organización ECMA (1) comienza a publicar normas para unificar el desarrollo del lenguaje JavaScript, lenguaje que había sido creado en 1995 por Braian Etich para el navegador Netscape 2.0. Desde ese entonces y durante los años siguientes, las páginas web comienzan a ser más dinámicas con el progresivo uso y avance del lenguaje JavaScript y las hojas de estilo. Paralelamente con el uso de la etiqueta applet, y posteriormente la etiqueta object en HTML, se dió lugar a la aparición de pequeños programas escritos en Java (Java applets), que pueden ser ejecutados por el navegador, y que su principal uso radicó en crear animaciones, efectos visuales e interactuar con el usuario. 1997 fue el año en el que la empresa Macromedia crea su applet denominado Flash, con el objetivo de crear e interactuar con gráficos vectoriales en la Web. En los años posteriores la popularidad de Flash aumentó, una gran cantidad de páginas web utilizaban Flash para insertar contenido multimedia en su sitio, y crear animaciones y juegos en 2D y 3D. En 1999, W3C comienza a trabajar en un nuevo estándar para la creación y manipulación de gráficos vectoriales. Este nuevo estándar sería denominado SVG (del inglés Scalable Vector Graphics), y pasaría a ser una recomendación de el W3C en setiembre de 2001. En 2004 antiguos empleados de Apple, Mozilla y Opera crean WHATWG (del inglés, Web Hypertext Application Technology Working Group) (5), un nuevo grupo de trabajo para mantener y desarrollar HTML y APIs para la aplicación web. Este grupo comenzó a elaborar especificaciones para una nueva versión de HTML, que luego sería llamado HTML5. En los años siguientes, WHATWG pasaría a encargarse del desarrollo de HTML5, los navegadores a la par de WHATWG fueron implementando este HTML, mientras que en W3C se creó un grupo de trabajo (14), que a partir de la primer versión de HTML5 en 2014, comenzó a fijar versiones de este HTML cada ciertos años. Durante este período, la tecnología Flash fue perdiendo apoyo, el nuevo HTML5 incluía etiquetas para trabajar con el estándar SVG, y también incluía la tecnología Canvas para el desarrollo interactivo con imágenes en mapa de bits, tecnología de la cual Apple era propietario y había liberado para su uso. Estas dos tecnologías son de código abierto y forman parte de las recomendaciones por W3C y WHATWG. Dos años después de la salida de HTML5, aparecería otra tecnología en escena, WebGL, la cual surge a partir de experimentos hechos en Canvas para la representación de objetos en 3D, y permite implementar gráficos 3D interactivos en la Web. También es de código abierto, y desde el año 2009, su desarrollo y especificación se ha realizado por

el consorcio Kronos Group (2), con la participación de las principales empresas interesadas en la Web. En la actualidad, estas tres tecnologías son las dominantes para crear gráficos en la Web, han madurado y proporcionan muchas de las funcionalidades que ofrece Flash, sin tener que instalar complementos para su ejecución. Flash, en 2004, había sido adquirida por Adobe y su código permanecía como código propietario. En 2017, en un comunicado en su blog oficial (10), Adobe anuncia que dejará de distribuir y actualizar Flash para el año 2020. Al igual que Flash, otras tecnologías como Microsoft Silverligh, han aparecido en estos últimos años pero no han tenido el éxito que tuvo Flash y tampoco el que tienen en este momento SVG, Canvas y WebGL.

2.2. Tecnologías para la representación gráfica interactiva en la Web.

Gráficos vectoriales escalables (SVG (16) por sus siglas en inglés) es una parte de la familia de gráficos basados en vectores. Este tipo de gráficos construye la imagen a través de diferentes objetos geométricos y para eso guarda y utiliza información que describe de forma matemática cada objeto geométrico, las transformaciones que se le aplicaron, e información de las relaciones que tiene con el resto de los objetos que conforman la imagen. SVG es un lenguaje que está basado en XML, en el cual cada objeto geométrico, sus transformaciones y relaciones, se definen mediante etiquetas y atributos. SVG y los gráficos vectoriales, conforman un sistema que funciona en modo retenido, es decir, se renderiza la imagen y se retiene la información de cómo fue renderizada. Esta característica de los gráficos vectoriales, en la web, fue realmente explotada con la aparición de HTML5, que permite incluir el XML que define a una imagen SVG en el código HTML, y modificar dinámicamente esta información mediante JavaScript y CSS. Cuando esta información es modificada, el navegador sólo vuelve a renderizar el objeto que fue modificado o agregado, sin renderizar nuevamente el resto de los objetos. Actualmente, SVG y HTML5 están implementados de forma nativa en la mayoría de los navegadores modernos. Antes de la aparición de HTML5, un SVG sólo podía ser insertado en HTML mediante la etiqueta HTML Image, sin la posibilidad de modificar de forma individual los objetos geométricos que componen la imagen.

Canvas, es otra tecnología utilizada para el renderizado de imágenes. Es una tecnología que permite renderizar imágenes raster, también llamadas mapas de bits, mediante scripting en JavaScript. Dicha tecnología funciona en modo inmediato; la imagen es renderizada y sólo se tiene información del color de cada píxel y para agregar nuevos objetos o modificar los existentes, es necesario renderizar nuevamente la imagen. Fue introducido en la Web con el estándar HTML5; se definió una nueva etiqueta HTML Canvas, y una interfaz DOM con un conjunto de métodos y funciones útiles para definir y crear formas geométricas, trazos, gradientes y aplicar diversas transformaciones geométricas sobre un mapa de bits. La etiqueta HTML Canvas es útil únicamente para ubicar la imagen creada con Canvas dentro del documento HTML, mientras que para crear y modificar la imagen se utiliza la interfaz DOM de Canvas. A través de esta Interfaz, se puede definir cada objeto que será renderizado en la imagen y manipular su posición, color, forma, etc, utilizando un objeto en JavaScript denominado "contexto" de la imagen, donde se guarda la información de cada objeto gráfico creado con la interfaz y renderizado en la imagen. Al igual que SVG, Canvas es soportado por la mayoría de los navegadores modernos y para su utilización, sólo es necesario el uso de las tecnologías HTML5 y JavaScript.

WebGL, a diferencia de SVG y Canvas, es una tecnología que tiene menos años de desarrollo pero que se está volviendo muy popular en la Web. Hay una comunidad activa trabajando en ella y tiene un gran apoyo de los principales navegadores. La última versión estable de WebGL está basada en OpenGL ES 2.0, y actualmente se está trabajando en una nueva versión que estará desarrollada sobre OpenGL ES 3.0. WebGL utiliza el elemento HTML Canvas al que se accede mediante interfaces DOM. Los programas escritos en WebGL consisten en un código de control escrito en JavaScript, y un código escrito en lenguaje GLSL que es ejecutado directamente por la Unidad de Procesamiento Gráfico (GPU, por sus siglas en inglés). El lenguaje GLSL permite que cálculos pesados, como el cálculo de la posición en 3D de los vértices, el de la luz en los objetos, las tramas o los efectos, sean manejados paralelamente por hardware especializado mediante el uso de la GPU. Esto acelera el renderizado de los objetos 3D, pero agrega la complejidad que tiene el lenguaje GLSL. Con este último, es necesario comprender la matemática que hace posible la represen-

tación de objetos 3D en dispositivos 2D; la cual involucra vectores y matrices para el cálculo de posición, transformaciones geométricas, color, etc. Por esta razón, la curva de aprendizaje de WebGL es alta, y en los últimos años se han desarrollado varias bibliotecas de uso gratuito y pagas, desarrolladas sobre WebGL, con la finalidad de facilitar el desarrollo con esta tecnología. Dos de las bibliotecas de código abierto más populares son Three.js y BabylonJS.

Tabla 2.1: Comparación de tecnologías.

	SVG	**Canvas**	**WebGL**
Modo	Retenido	Inmediato	Inmediato
Tipo de Imagen	Vectorial	Mapa de bits	Mapa de bits
Uso de GPU	No	No	Si
Manejadores de Eventos	Nativo	Por coordenadas del mouse	Por coordenadas del mouse
Requerimientos	HTML5	HTML5	HTML5
Curva de aprendizaje	Medio/Bajo	Medio/Bajo	Alta

2.3. Bibliotecas útiles para la representación de funciones y figuras matemáticas.

SVG, Canvas y WebGL son tecnologías utilizadas en una gran variedad de casos de uso, desde representaciones de objetos simples hasta aplicaciones y juegos interactivos muy complejos. Para cada uno de estos diferentes usos, se han desarrollado bibliotecas que utilizan estas tecnologías, y resuelven o mejoran algún aspecto específico de la representación gráfica en la Web. Bibliotecas como: GeoGebra(6), Desmos, D3.js, Plotly, FunctionPlot (11), Three.js (3), BabylonJS, entre otras, pueden ser usadas para la representación dinámica e interactiva de gráficas de funciones, figuras geométricas y animaciones. De hecho, estas son las bibliotecas que se han encontrado más relevantes para el desarrollo específico de este caso de uso, por tener desarrollado diferentes aspectos relacionados a esta problemática o por brindar diferentes funcionalidades que facilitan el desarrollo del mismo. Cabe señalar, también, que no todas estas bibliotecas fueron desarrolladas únicamente con el objetivo de facilitar el desarrollo que implica la representación gráfica de funciones o figuras

geométricas, el potencial de diferentes cosas que se pueden desarrollar con bibliotecas como D3.js, Three.js y BabylonJS es mucho más amplio.

GeoGebra(6) y Desmos, son herramientas creadas con el objetivo de que estudiantes del área de la matemática puedan utilizar dichas tecnologías para aprender esta disciplina de forma interactiva, reuniendo la geometría, álgebra, estadística y cálculo, y para utilizarlas en la resolución de cálculos simbólicos, evaluación de funciones, trabajo con figuras geométricas, entre varias cosas más relacionadas a estas áreas. Ambas herramientas utilizan Canvas para la representación gráfica en la Web de objetos 2D, y en el caso de GeoGebra (6), también permite representar objetos 3D a través de WebGL. Inicialmente, GeoGebra (6) fue desarrollado en código Java en 2001, y actualmente su uso está disponible como programa de escritorio, plataforma web, mediante applets o mediante una API desarrollada en JavaScript. Desmos, creado en 2011, fue desarrollado únicamente para su uso en la Web; posee una plataforma web y una API en JavaScript similares a las que posee GeoGebra (6). Son distribuidas mediante una licencia dual, su código es software libre pero no está disponible libremente para fines comerciales.

D3.js, es una biblioteca que permite crear todo tipo de gráficos dinámicos e interactivos haciendo uso de SVG, HTML5 y CSS. La principal característica de D3.js es que permite vincular datos a elementos del DOM, y crear una representación visual para estos datos en HTML o SVG. Un ejemplo sencillo de esta característica es crear un tabla en HTML o un gráfico de barras en SVG, a partir de una matriz de datos. D3.js tiene una licencia BSD, haciendo su código fuente libre hasta para su uso en software no libre o comercial, está desarrollado en módulos que pueden ser utilizados de forma individual o todos juntos, donde cada uno de ellos provee métodos para desarrollar y trabajar con conceptos específicos como crear el efecto zoom en un gráfico, trabajar con escalas, animaciones, etc.

Plotly, al igual que Geogebra(6) y Desmos, es una aplicación que puede ser utilizada de diferentes maneras, y la cual posee distintas licencias para su uso, brinda varias bibliotecas para trabajar en Python, R, Matlab, Node.js, Julia, Arduino y JavaScript. La biblioteca en JavaScript, denominada Plotly.js, tiene licencia MIT, con lo que su código puede ser utilizado y modificado libremen-

te. Esta biblioteca está construida sobre D3.js y stack.gl, siendo esta última creada para facilitar el desarrollo con WebGL. Plotly.js admite 20 tipos de gráficos, incluidos gráficos 3D, mapas geográficos y gráficos estadísticos como de densidad, histogramas, gráficos de caja y de contorno.

FunctionPlot, a diferencia del resto de las bibliotecas mencionadas, es una biblioteca pequeña y liviana desarrollada sobre D3.js, con licencia MIT, que únicamente está destinada al renderizado de funciones utilizando una pequeña configuración para ello. Actualmente posee un conjunto variado de funciones que pueden ser graficadas con la biblioteca, y a través de D3.js posee implementaciones para elementos como ejes, grilla, escalas, zoom, etc.

Three.js y BabylonJS, son dos bibliotecas muy parecidas y las más populares en el ámbito del renderizado 3D en la Web. Mientras que BabylonJS está construida sobre WebGL y utiliza esta tecnología para el renderizado en 3D, Three.js puede ser configurada para utilizar Canvas, SVG o WebGL. Ambas tecnologías permiten trabajar conceptos propios de la representación de objetos 3D en 2D, como es la cámara, luces, materiales, shaders, efectos, escenas, etc. Con estas bibliotecas se pueden crear gráficas de funciones, figuras, juegos y hasta contenido de realidad virtual.

Tabla 2.2: Bibliotecas.

	Geogebra	**Desmos**	**D3.js**	**Plotly**
Representación de funciones	Parcial	Parcial	No	Parcial
Representación de figuras geométricas	Si	Si	Si	Si
Tecnologías	Canvas WebGL	Canvas	SVG	SVG WebGL
Soporte 3D	Si	No	No	Si
Tamaño	2 MB — 8 MB	1.7 MB	242 KB	3.09 MB
Licencia	Dual	Dual	BSD	MIT
	FunctionPlot	**Three.js**	**BabylonJS**	
Representación de funciones	Parcial	No	No	
Representación de figuras geométricas	No	Si	Si	
Tecnologías	SVG	Canvas SVG WebGL	WebGL	
Soporte 3D	No	Si	Si	
Tamaño	142 KB	557 KB	2.1 MB	
Licencia	MIT	MIT	Apache License 2.0	

Capítulo 3

Estado inicial del proyecto MateFun.

En este capítulo se describe el estado del lenguaje MateFun y el módulo de gráficas cuando se comenzó este trabajo de tesis. Es importante comprender los diferentes elementos del lenguaje que pueden ser representados en una gráfica 2D y 3D, y evaluar el trabajo realizado hasta el momento y las dificultades que tuvieron que ser abordadas.

3.1. Lenguaje MateFun.

Al comienzo de nuestro trabajo de tesis, el lenguaje MateFun estaba desarrollado para trabajar con diferentes figuras en 2D y funciones en una y varias dimensiones. Las figuras primitivas del lenguaje en ese entonces eran: rectángulo (rect), círculo (circ), polígono (poli) y línea (line). Al ser MateFun un lenguaje funcional, estas figuras se crean con funciones que reciben parámetros que definen sus atributos. El uso de estas funciones es el siguiente:

```
rect(ancho,alto)
circ(radio)
poli((x1,y1):(x2,y2): ... :(xn,yn):[R X R])
line((x1,y1),(x2,y2))
```

Los parámetros toman valores reales (R) y [R X R] es el dominio de una secuencia de pares de números. Las figuras también pueden ser manipuladas utilizando tres funciones primitivas que brinda el lenguaje:

```
join(fig1,fig2)
```

Retorna una nueva figura como resultado de la unión de $fig1$ y $fig2$.

```
move(fig,(x,y))
```

Retorna una nueva figura como resultado de trasladar el centro de la figura fig a la posición (x, y).

```
rotate(fig, x)
```

Retorna una nueva figura como resultado de rotar el centro de la figura fig, x grados.

MateFun también permite definir nuevas funciones, distintas a las definidas de forma primitiva. Para esto, es necesario que se defina: dominio, codominio y cuerpo de la función. Ejemplo:

```
f :: R -> Fig
f(x) = circ(x+x)
```

La función f tiene como dominio el conjunto R y como codominio el conjunto Fig. El término conjunto en MateFun es utilizado para definir el dominio y codominio de una función. Al igual que las funciones, MateFun tiene definidos conjuntos primitivos, y permite definir nuevos a partir de otros, utilizando una notación por enumeración o comprensión. Los conjuntos R (Reales) y Fig (Figuras) utilizados en la función f son dos conjuntos primitivos. Esta definición permite crear una gran variedad de dominios y codominios, dando lugar a una familia más amplia de funciones que pueden definirse con el lenguaje. Ejemplo:

```
conj Dias = { Lunes, Martes, Miercoles, Jueves, Viernes,
              Sabado, Domingo }
conj Z = { x en R | x == red(x) }
```

```
map :: Dias -> Z
map(d) = 1 si d = Lunes
         2 si d = Martes
         3 si d = Miercoles
         4 si d = Jueves
         5 si d = Viernes
         6 si d = Sabado
         7 si d = Domingo
```

La función *map*, es una función por partes la cual utiliza dos conjuntos definidos por el usuario, uno por enumeración y otro por comprensión. Como último caso, las funciones también pueden ser definidas en dominios y codominios de 2 o más variables.

```
rodar :: Fig -> Fig*
rodar(f) = f:rotate(f,45):rotate(f,90):rotate(f,135):
            rotate(f,180):rotate(f,225):rotate(f,270):
            rotate(f,315)

f :: R X R -> R
f (x,y) = x * y
```

La primera función tiene como codominio el conjunto Fig* que es un conjunto primitivo utilizado para definir animaciones de figuras. Esta función toma una figura y retorna una secuencia de ellas aplicando en cada paso la transformación rotar. La segunda función, es una función de dos variables que en caso de ser graficada, debe realizarse en una gráfica de tres dimensiones.

En resumen, MateFun sólo permite crear y manipular figuras 2D mediante funciones primitivas que el lenguaje posee para esto. Se pueden crear nuevas funciones, las cuales a partir de primitivas, retornan nuevas figuras, pero la representación gráfica de esto siempre será a partir de las figuras y transformaciones primitivas. Para el caso de funciones con dominio y codominio R, un subconjunto de R o un enumerado, el comportamiento es diferente. Estas funciones pueden tener un dominio infinito o presentar diferentes tipos de singularidades y ser funciones continuas, discontinuas, discretas o por partes. La representación gráfica de este tipo de funciones debe de tener en cuenta los diferentes aspectos de la función. Para el caso de 3D, el lenguaje no posee

funciones primitivas para trabajar con figuras en 3D, pero sí con funciones que pueden ser representadas en una gráfica 3D. Estas, al igual que lo que sucede con las funciones en 2D, pueden presentar diferentes aspectos que deben ser considerados para su correcta representación gráfica.

3.2. Módulo de gráficas.

Al comenzar el presente trabajo de tesis, el entorno integrado web de MateFun, poseía un módulo de gráficas capaz de renderizar figuras y funciones en 2D, y fue implementado utilizando la tecnología HTML5 Canvas sin utilizar ninguna biblioteca externa. Este módulo es capaz de renderizar todas las figuras primitivas del lenguaje Matefun y las figuras, transformaciones y animaciones definidas por el usuario. Para el resto de las funciones, dicho módulo sólo es capaz de renderizar funciones numéricas de una sola dimensión, con dominio y codominio R o un subconjunto de R.

La implementación del mismo se lleva a cabo a través de diferentes funciones desarrolladas en TypeScript utilizando Angular 4, que en conjunto funcionan como un analizador sintáctico. Cuando la respuesta del servidor a un comando ejecutado por el usuario en el intérprete es una figura, una animación, o una función a graficar, la respuesta es procesada y analizada por el módulo. Esa respuesta está escrita en formato JSON y contiene una estructura específica de datos que indica qué tipo de función, figura o animación se debe renderizar. El módulo se encarga de analizar la respuesta, crear el Canvas, y brindar un conjunto de controles en la interfaz web para que el usuario pueda interactuar con el Canvas generado.

Para el caso de figuras y animaciones, la representación es directa. MateFun permite generar un conjunto finito de figuras y para cada una de las que se quiera renderizar, la respuesta del servidor contiene todos los datos necesarios para saber qué tipo de figura es, cuáles son sus atributos y qué transformaciones geométricas se deben aplicar. El caso de funciones es diferente. El mencionado módulo debe ser capaz de renderizar cualquier función numérica con dominio y codominio R, o un subconjunto de R que haya definido el usuario. Este conjunto de funciones, como se mencionó anteriormente, es infinito y variado y, en este

caso, la respuesta del servidor contiene como dato únicamente la definición de la función (dominio, codominio y cuerpo de la función). El dominio, en estos casos, puede ser un conjunto infinito, y no es posible que el módulo evalúe la función en todos los puntos del dominio, por el contrario, el módulo debe ser capaz de analizar la función en diferentes puntos del dominio para conocer el comportamiento de la misma. Para resolver esto, se desarrolló el presente módulo con dos algoritmos que analizan la continuidad de la función en distintos puntos del domino. El siguiente pseudocódigo detalla los pasos que sigue el mismo desde que toma la definición de la función hasta que renderiza la misma:

1. Se crea una función JavaScript análoga a la definición de la función obtenida en la respuesta del servidor.

2. Se divide en 1000 puntos equiespaciados el rango en que será visualizada la gráfica de la función en Canvas.

3. Para cada uno de los 1000 puntos:

    a) Se evalúa la función en el punto.

    b) Se ejecutan dos algoritmos para analizar la continuidad de la función en ese punto. Si la función es continua, en Canvas se traza una recta entre la evaluación de este punto y la evaluación del punto anterior, de lo contrario, sólo se dibuja la evaluación de este punto.

El primer algoritmo desarrollado es utilizado para renderizar correctamente continuidades y discontinuidades en funciones discretas o por partes, que en el cuerpo de la función tienen definida una condición de igualdad. Un ejemplo de este tipo de funciones, es la de la Figura 3.1. Al evaluar dicha función en 1000 puntos equiespaciados, la probabilidad de que el punto 1 del dominio pertenezca al conjunto elegido de 1000 puntos es prácticamente 0, esto hará que la función en este punto no sea evaluada y este punto no sea renderizado en la gráfica. Dicho algoritmo entonces, relaja la condición de igualdad agregando un parámetro Δ igual al paso de la iteración ($\Delta = \frac{rango}{1000}$), y cambia las condiciones de igualdad en las funciones por una condición de diferencia. De esta forma se detecta que en x=1, f(x) vale 2.

El segundo algoritmo es utilizado para detectar el resto de las discontinuidades de una función. Sea x_n el punto que se está evaluando y x_{n-1} el punto

```
porParte :: R -> R
porParte(x) = x si x < 1
              o 2 si x == 1
              o 3
```

Función por partes con condición de igualdad.

```
porParte :: R -> R
porParte(x) = x si x < 1
              o 2 si x - 1 < Δ
              o 3
```

Función por partes con parámetro Δ.

Figura 3.1

evaluado en el paso anterior, se define un ángulo θ y se toma la recta tangente a la gráfica de la función por el punto x_{n-1} como la bisectriz del ángulo θ, ver Figura 3.2. Si la evaluación del punto x_n cae dentro de la ventana determinada por el ángulo θ, entonces se considera que en este punto no hay discontinuidad, de lo contrario, existe una discontinuidad y no se traza la recta que une la evaluación de x_{n-1} y x_n. Dicho algoritmo tuvo que ser probado con diferentes valores θ, para encontrar el ángulo θ que lo hace óptimo definiendo $\theta = \frac{\Pi}{8}$.

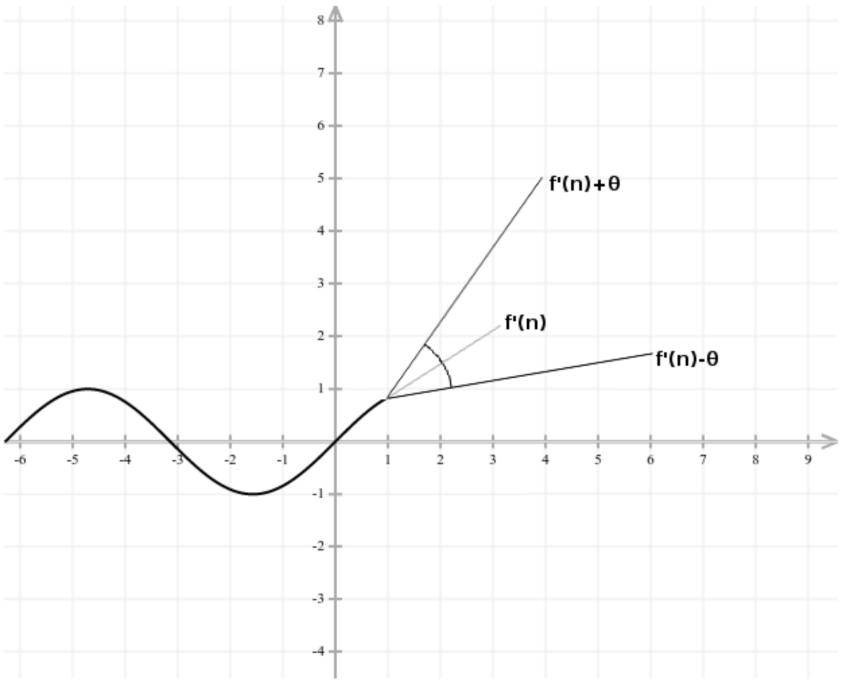


Figura 3.2: Algoritmo de detección de discontinuidades.

El resultado de aplicar los mencionados algoritmos genera una representación gráfica en Canvas de la función. Dicha representación gráfica es una aproximación a la función real debido a que, en entornos donde la función es

continua, se aproxima la función por rectas. A mayor cantidad de puntos en lo que se evalúe la función, mejor será la aproximación, pero decrementa el rendimiento del sistema. En general, con 1000 puntos la aproximación es bastante buena, encontrándose limitado el módulo en la representación gráfica de las funciones. No es posible representar todas las funciones que se pueden generar con el lenguaje MateFun, sólo es posible graficar funciones definidas en un dominio R o un subconjunto de R. Lo segundo, y no menos importante, es que los algoritmos implementados para analizar la continuidad de la función, no siempre funcionan como se espera. En funciones que oscilan demasiado rápido, como el caso de $sen(x^3)$ el segundo algoritmo definido detecta discontinuidades de la función en entornos donde la función es continua (falsos positivos).

Capítulo 4

Desarrollo.

4.1. Decisiones generales.

En el marco del análisis realizado para conocer los diferentes elementos que pueden ser creados con el lenguaje MateFun y las decisiones tomadas para desarrollar el primer módulo de gráficas, se llega a la conclusión de que las funciones con dominio y codomino distinto al conjunto Fig, son el principal elemento a considerar para el diseño del módulo de gráficas. Las figuras, como se mencionó anteriormente, son elementos que pertenecen a un conjunto finito, y la representación gráfica de cada figura y transformación, queda totalmente definida a partir de un conjunto finito de atributos. De aquí en más, el énfasis se pondrá en este tipo de funciones para justificar las decisiones tomadas. La primera decisión que se toma, es dividir el desarrollo en dos módulos, un módulo para la representación de funciones en 2D y otro para la representación en 3D. La razón de esto, surge del hecho de que actualmente para representar elementos en 3D de forma óptima, la mejor solución es utilizar la tecnología WebGL o alguna biblioteca que haga uso de esta tecnología; mientras que para el caso de 2D, si bien también dicha tecnología puede usarse, es más sencillo y liviano utilizar una tecnología como Canvas o SVG, y aprovechar el desarrollo hecho en el área por algunas de las bibliotecas que utilizan estas tecnologías. Cada uno de estos módulos, sus funcionalidades y la representación gráfica serán independientes, y es por eso que se decide separarlos.

Para el desarrollo del módulo de gráficas 2D se decide utilizar la biblioteca FunctionPlot, implementada sobre D3.js y que hace uso de SVG para la re-

presentación gráfica. Esta biblioteca es liviana, y en comparación con el resto de las mencionadas es fácil extender su funcionalidad; no se encontró una que representara todos los tipos de funciones diferentes que el lenguaje MateFun permite trabajar, y es importante entonces que la biblioteca utilizada sea fácil de extender. Bibliotecas como la que brinda el sistema GeoGebra son muy potentes pero a su vez difíciles de extender. En el caso de GeoGebra, es un sistema más pesado, complejo, con varios años de desarrollo, y que posee un gran número de funcionalidades diferentes, muchas más de las que serán utilizadas en MateFun. También, posee una API en JavaScript para integrar el sistema en un sitio web, que tiene muy poca documentación y que el trabajo con la API se hace mediante comandos —que pueden ser complejos— escritos como cadena de texto, y que dificulta la integración con un sistema que utiliza el formato JSON para definir las funciones como es el caso del sistema MateFun. GeoGebra, sin dudas es uno de los sistemas más completos para el trabajo interactivo con la matemática y su representación gráfica, pero en el marco de este proyecto, Function Plot es más flexible para crear un módulo liviano que permita representar en una gráfica las diferentes funciones del lenguaje.

FunctionPlot, actualmente está desarrollada para representar un conjunto variado de funciones definidas en dominios reales o en un subconjunto de los reales y además, mejora el algoritmo que actualmente utiliza el módulo de gráficas de MateFun para aproximar funciones y trazar las gráficas de las mismas. Esta biblioteca utiliza un algoritmo de aritmética de intervalos, haciendo uso de un número finito de muestras o evaluaciones de la función, mientras que la implementación actual del módulo de gráfica, utiliza puntos igualmente espaciados unidos por segmentos de recta. En funciones que oscilan demasiado rápido, se puede apreciar la diferencia de estos dos algoritmos, dando mejores resultados el utilizado por la biblioteca. Además de este aspecto, otra diferencia importante, es que la biblioteca utiliza SVG para la representación gráfica. Las funciones del lenguaje MateFun que pueden ser representadas en una gráfica 2D, son funciones cuya representación no requiere de una gran cantidad de objetos diferentes, pero sí requiere de un alta interacción con el usuario a través de eventos. En SVG, los eventos se manejan de igual forma que se hace con el resto de los elementos en HTML, siendo muy fácil de utilizar en comparación con Canvas, donde los eventos se deben programar a partir de las coordenadas del ratón. La biblioteca también, es de código abierto con una

comunidad activa, y en el futuro, nuevas funcionalidades que la comunidad desarrolle, podrán ser incorporadas a MateFun. El último aspecto a destacar es el uso de D3.js. Funcionalidades necesarias aquí, como las funcionalidades de arrastrar, acercar y alejar la gráfica, están resueltas mediante D3, la cual es una biblioteca altamente aceptada y con una gran comunidad trabajando en ella.

En resumen, para el caso de 2D, se decidió implementar un nuevo módulo de gráficas que utilice FunctionPlot. Las funciones definidas en dominios y codominios R o un subconjunto de R, que actualmente el módulo de gráficas implementado en MateFun renderiza, ya están desarrolladas en FunctionPlot. Para el resto de las funciones, definidas en dominios y codominios enumerados y discretos (que actualmente no renderiza el módulo de gráficas de MateFun), o el caso de figuras, se decidió extender la biblioteca FunctionPlot para que soporte este tipo de funciones y figuras.

En el ámbito de 3D, se analizaron las bibliotecas GeoGebra, Plotly y Three.js, siendo esta última la elegida para llevar a cabo el desarrollo del módulo. WebGL es una tecnología que tiene una alta curva de aprendizaje, la necesidad de familiarizarse y manejar conceptos propios de la representación de objetos 3D en un dispositivo 2D, como la pantalla de un computador, hace que esta tecnología pueda ser difícil de comprender. Three.js es una biblioteca que funciona sobre WebGL, su sintaxis declarativa abstrae mucho de los conceptos utilizados en WebGL, facilitando el desarrollo de esta tecnología. Como se menciona en la documentación oficial del repositorio en GitHub, el objetivo con Three.js es crear una biblioteca 3D liviana y fácil de usar.

En el caso de GeoGebra y Plotly, son dos sistemas muy completos que permiten crear y configurar una gran variedad de gráficos, entre los que se encuentran los gráficos en 3D. Ambos sistemas, poseen bibliotecas desarrolladas para su uso en la Web mediante JavaScript y utilizan WebGL para la representación gráfica en 3D. Si bien MateFun se puede ver beneficiado por la extensa variedad de funcionalidades que ya están implementadas en estas bibliotecas, resulta compleja la tarea de agregar a éstas, características nuevas o adaptar alguna existente para ser utilizada en MateFun. Asimismo, no se encontró una forma sencilla de agregar a MateFun únicamente los tipos de gráficos y las funcionalidades de las bibliotecas que iban a ser utilizadas, lo que hace aún

más difícil la tarea de extenderlas y agregar código que no será utilizado. La biblioteca de GeoGebra aumenta de 2MB a casi 8MB al incluir el soporte para 3D y la biblioteca de Plotly es de 3MB, esto se traduce en un retardo en la carga del archivo JavaScript, que puede apreciarse al momento de cargar una gráfica. Por último, en el caso de GeoGebra, también resulta difícil traducir el código de las funciones que se quieren graficar (código que proviene del servidor de MateFun en formato JSON), al código que utiliza GeoGebra para representar una función. GeoGebra utiliza diferentes comandos escritos como cadena de texto que son evaluados por un método de la biblioteca para crear la representación gráfica de la función.

Three.js es de más bajo nivel que las bibliotecas de GeoGebra y Plotly, no resuelve todo lo que estas bibliotecas resuelven, pero brinda funciones más sencillas para trabajar con objetos y conceptos de la representación gráfica en 3D. Puede ser utilizado mediante módulos, donde el núcleo de Three.js se centra en los componentes más importantes para la representación en 3D, y se tiene muchos otros módulos que el desarrollador puede importar para utilizar en su proyecto.

Teniendo en cuenta lo expuesto precedentemente, se decide crear una biblioteca tomando como base Three.js, la cual sea capaz de renderizar diferentes funciones, figuras y animaciones en 3D, y un módulo para integrar esta biblioteca al sistema MateFun. La biblioteca se denominó Graph3D y puede ser utilizada de forma independiente a MateFun. Es necesario, también, extender el lenguaje MateFun para que soporte figuras en 3D y animaciones de estas figuras, para ello se agregan al lenguaje las siguientes figuras: Esfera, Cubo, Prisma, Anillo de Taurus, Cilindro y Líneas. También se decide, implementar la biblioteca para que brinde soporte para graficar funciones en tres dimensiones, pero no integrar esta característica con el sistema MateFun. Es necesario ajustar algunos aspectos del lenguaje MateFun para que las funciones en 3D funcionen correctamente, siendo una tarea compleja, y que queda por fuera del alcance de este proyecto.

4.2. Arquitectura.

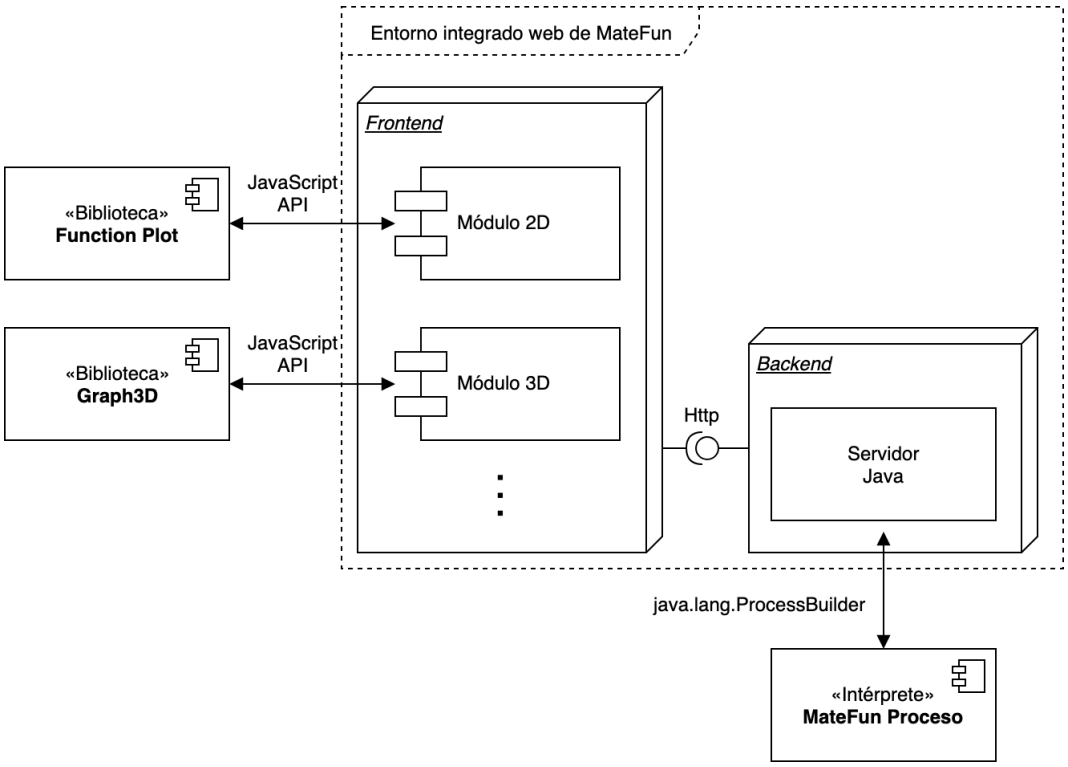


Figura 4.1: Arquitectura.

MateFun Proceso:
Instancia del intérprete de MateFun.

Servidor Java:
Componente responsable de levantar una instancia del intérprete de MateFun, y de manejar la comunicación entre esta instancia y el Frontend.

Módulo 2D:
Módulo encargado de generar la representación gráfica en 2D de una función, figura o animación en el entorno web de MateFun. Cuando la respuesta del Backend es una representación gráfica en 2D, este módulo procesa la respuesta y utiliza la biblioteca Function Plot para generar la representación gráfica en SVG.

Function Plot:
Extensión de la biblioteca Function Plot con funcionalidades para representar figuras geométricas y funciones 2D en diferentes dominios.

Módulo 3D:
Análogo al Módulo 2D para generar representaciones gráficas de figuras y animaciones en 3D para el entorno web de MateFun. Este módulo utiliza la biblioteca Graph3D para generar la representación gráfica.

Graph3D:
Biblioteca creada para representar funciones, figuras y animaciones 3D en Canvas. La biblioteca se crea utilizando la biblioteca Three.js que hace uso de WebGL para crear la representación en 3D.

La comunicación entre el *Frontend* y el *Backend* se realiza a través de servicios REST y una conexión Websocket. El Servidor Java expone una API REST, que brinda funcionalidades para el manejo de usuarios, archivos y grupos. En cuanto a la conexión Websocket, es utilizada para la comunicación bidireccional entre el *Frontend* y la instancia del intérprete de MateFun, siendo el Servidor Java intermediario de esta comunicación. Mediante la conexión Websocket, el *Frontend* envía comandos que deben ser ejecutados en el intérprete MateFun, y la respuesta a dichos comandos es enviada nuevamente al *Frontend* por la conexión Websocket. Los mensajes enviados por la mencionada conexión son en formato JSON siguiendo la siguiente estructura:

Mensajes enviados desde el *Frontend*:

```
{
    "token" : "<token>",
    "comando" : "<comando>"
}
```

token
Id que tiene asociado el cliente con el servidor Java. Permite identificar a qué cliente corresponde el mensaje.

comando
Comando ingresado por el usuario a través de una consola expuesta en el entorno web de MateFun.

Mensajes enviados desde el *Backend*:

```
{
    "tipo" : "<tipo respuesta>",
    "resultado" : "<resultado>"
}
```

tipo
Según el comando ingresado en el mensaje enviado desde el *Frontend*, el tipo de respuesta devuelto por el *Backend* puede ser diferente. En el caso de que la respuesta del intérprete de MateFun sea una representación gráfica en 2D, en 3D, o una animación, el tipo de respuesta devuelto es graph, graph3D y animation, respectivamente.

resultado
Resultado devuelto por el intérprete de MateFun al comando enviado.

Para el caso de la comunicación entre las bibliotecas FunctionPlot y Graph3D con los Módulos 2D y 3D, se realiza mediante una API en JavaScript que expone cada biblioteca. Estas APIs serán desarrolladas más en detalle en las secciones siguientes.

4.3. Gráficos 2D.

4.3.1. Extensión de biblioteca FunctionPlot.

FunctionPlot es una biblioteca en JavaScript que hace uso de la biblioteca Browserify para organizar el desarrollo en módulos, de la misma forma que se hace el desarrollo en módulos en Node.js. La extensión de FunctionPlot se desarrolló respetando esta estructura, modificando la funcionalidad de los módulos existentes, así como agregando nuevas funcionalidades y opciones que permitan representar elementos de MateFun, que originalmente esta biblioteca no permite representar. La arquitectura de dicha biblioteca se puede apreciar en la Figura 4.2. El módulo "Shape", marcado en azul, es un módulo nuevo, producto de la extensión realizada en este proyecto, el cual se detallará en la presente sección.

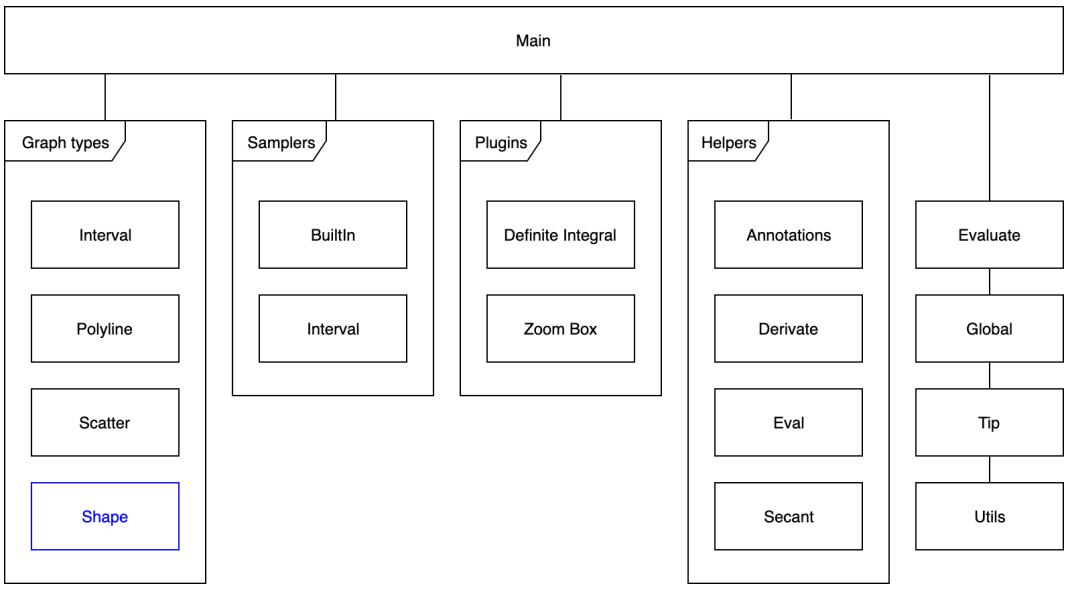
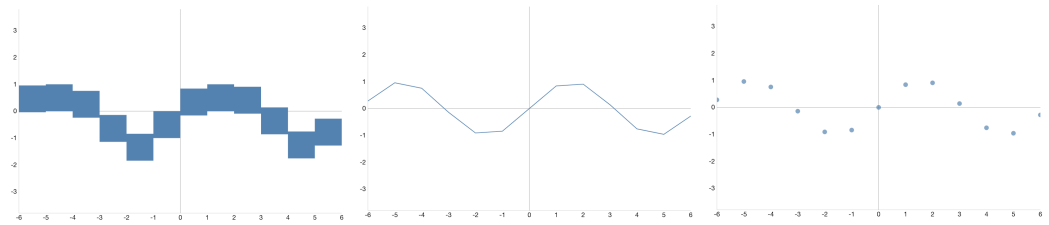


Figura 4.2: Arquitectura de FunctionPlot.

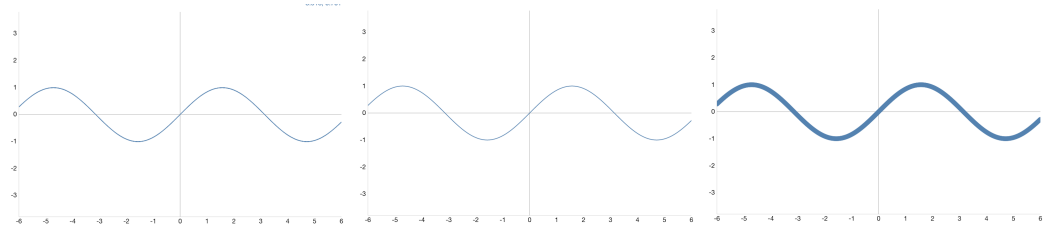
Los módulos de la biblioteca están organizados en cinco grupos, Graph Type, Samplers, Plugins y Helpers. Los módulos que están por fuera de estos grupos son los que se encuentran en la raíz del proyecto, junto con el módulo principal (Main), y con métodos y funcionalidades más generales.

El grupo Graph Types, es uno de los principales grupos de módulos, e implementa diferentes tipos de gráficos. Los tres primeros módulos de este grupo (Interval, Polyline y Scatter) son útiles para representar una función aplicando métodos diferentes. Interval utiliza aritmética de intervalos para aproximar la función, Polyline utiliza segmentos de líneas, mientras que Scatter, utiliza puntos aislados en cada una de las evaluaciones que se hace de la función. En las figuras 4.3 y 4.4, se puede ver la diferencia de estos módulos al representar la función $sen(x)$. Interval y Polyline, utilizan el elemento path de SVG para dibujar la función, y la representación o aproximación de la función mejora al aumentar el número de evaluación que se hace de la misma en el rango en el que se está visualizando la gráfica. Scatter, por el contrario, utiliza el elemento circle de SVG y su finalidad es representar una función en puntos. Este módulo no fue implementado para representar funciones continuas definidas sobre conjuntos densos, dado que se necesitaría un número muy grande de evaluaciones, lo que aumentaría el número de cálculos que se deben hacer, y también el número de etiquetas circle en el SVG.



(a) Interval (b) Polyline (c) Scatter

Figura 4.3: Gráfica de la función sen(x) utilizando 13 evaluaciones de la función en el rango [-6, 6].



(a) Interval (b) Polyline (c) Scatter

Figura 4.4: Gráfica de la función sen(x) utilizando 1000 evaluaciones de la función en el rango [-6, 6].

Con estos módulos se pueden representar funciones continuas y discontinuas definidas en los reales, así como también, se puede trabajar con ecuaciones paramétricas y polares, funciones implícitas, o representar polilíneas, vectores y puntos. En el sitio oficial de la biblioteca se muestra, con un ejemplo, cada uno de estos casos de uso.

El grupo de módulos Samplers, es otro grupo importante. Los dos módulos de este grupo son utilizados de forma excluyente, y realizan una tarea fundamental para la generación de la gráfica de una función. Ambos son los responsables de dividir el rango de la gráfica en intervalos con puntos equiespaciados, y utilizar estos puntos para evaluar la función. El número de puntos en los que se divide el rango es configurable por el usuario. En las figuras 4.3 y 4.4, se pueden ver las gráficas de la función $sen(x)$, utilizando 13 y 1000 puntos respectivamente. Los módulos también son responsables de analizar la continuidad de la función en cada intervalo generado. Los puntos, las evaluaciones de la función, y el análisis de la continuidad, son utilizados posteriormente por los módulos del grupo Graph Types para dibujar la función.

La diferencia entre los módulos de este grupo, radica en que, BuiltIn es uti-

lizado por Polyline y Scatter, mientras que Interval es utilizado por el módulo Interval de Graph Types. BuiltIn, retorna un arreglo con los puntos en los que se evaluó la función, y la evaluación de la función en cada uno de estos puntos, mientras que Interval, retorna un arreglo con los puntos extremos de cada intervalo generado, y la evaluación de la función que se hizo en estos puntos.

El análisis de la continuidad de la función en cada intervalo, fue implementado en ambos módulos para detectar asíntotas y salto infinito en el intervalo. Discontinuidades como las que puede presentar una función definida en trozos o en partes, no fueron contempladas en esta biblioteca. En el anexo se detallan los algoritmos utilizados por BuiltIn e Interval para detectar este tipo de discontinuidades.

Los siguientes dos grupos de módulos generan nuevos elementos que pueden ser añadidos a la gráfica de la función. Estos nuevos elementos pueden ser rectas tangentes o secantes en puntos de la función, texto que se puede agregar en el gráfico, cálculo de integrales definidas, manejadores de eventos para acercar el gráfico en regiones que el usuario defina, etc. En el sitio oficial de la biblioteca también se detallan ejemplos de estos módulos y sus posibles usos.

Por último, el módulo Main, es el que controla todo el flujo de trabajo que se realiza con la biblioteca. Este módulo expone métodos para crear una instancia de una gráfica, y expone otros que pueden ser utilizados para interactuar con una gráfica ya creada. Además, en dicho módulo es donde se crean elementos de la gráfica como la grilla, los ejes, las escalas, y donde se configuran manejadores de eventos para acercar, alejar, arrastrar el gráfico, o para mostrar valores de la función cuando se pasa el ratón por encima de la misma. Para crear una instancia de una gráfica se define un constructor denominado FunctionPlot, de forma global dentro del objeto window en JavaScript. Este método recibe un objeto que tiene propiedades específicas, que el usuario puede usar para definir la función que desea graficar, y para configurar los diferentes elementos que componen la gráfica. En la Figura 4.5 se crea la gráfica de x^2 y se configuran ciertas propiedades del gráfico.

La propiedad *data*, es un arreglo de objetos donde cada uno de ellos define una función a ser graficada. También aquí se puede definir qué módulo se

```
FunctionPlot({
    target: '#ejemplo',
    width: 580,
    height: 400,
    xAxis: {domain:[-10, 10]},
    yAxis: {domain:[-2, 8]},
    disableZoom: true,
    data: [{
      fn: 'x^2',
      graphType: 'polyline',
      nSamples: 500
    }]
})
```

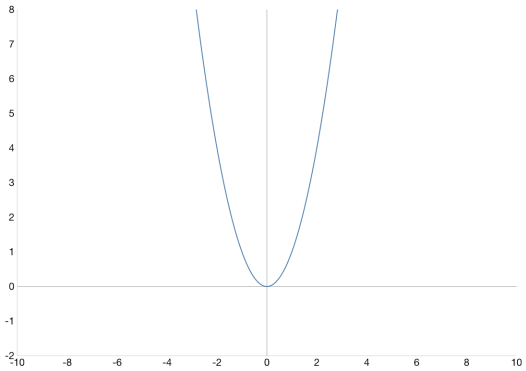


Figura 4.5: Inicialización de una gráfica utilizando la biblioteca FunctionPlot.

quiere utilizar, cuántas evaluaciones se deben hacer en el rango de la gráfica, entre otras propiedades más. Para ver una lista completa de todas las propiedades que brinda FunctionPlot, ver apéndice. El resto de las propiedades que acompañan la propiedad *data*, son las que definen diferentes elementos de la gráfica, como el rango en el eje x, en el eje y, ancho y alto de la gráfica, etc. Las únicas propiedades requeridas que deben ser definidas, son la propiedad *fn* de cada objeto de *data*, y la propiedad *target*, que es el id del elemento HTML donde será insertada la gráfica. Para el resto de las propiedades, de no definirse, FunctionPlot toma valores por defecto.

4.3.1.1. Figuras.

Como se acaba de ver, FunctionPlot es una biblioteca que permite crear representaciones gráficas interactivas de funciones, pero en el caso de las figuras geométricas, sólo es posible representar puntos y polilíneas. Se decide entonces extender la funcionalidad de FunctionPlot, para que sea posible representar con la biblioteca las figuras geométricas en 2D definidas en el lenguaje MateFun. Estas figuras son:

Rectángulo, Círculo, Texto, Polilínea y Polígono.

Para las figuras, Rectángulo, Círculo y Texto, se decide implementar un nuevo módulo en el grupo Graph Types denominado Shape. Las figuras están definidas por atributos geométricos que definen cómo deben ser renderizadas. No son funciones que deban ser evaluadas en diferentes puntos para generar una aproximación, como lo hacen los módulos Interval, Polyline y Scatter.

Por esta razón, se decide separar la implementación de estas tres figuras en un nuevo módulo. Para las figuras Polilínea y Polígono, se puede utilizar la implementación que tiene FunctionPlot en el módulo Polyline para representar polilíneas y puntos.

Módulo Shape

Los módulos del grupo Graph Types, pueden ser seleccionados cuando se crea la gráfica mediante la propiedad *graphType*. A dicha propiedad se le agrega el valor "shape"para poder seleccionar este nuevo módulo.

```
graphType: interval|polyline|scatter|shape
```

También se define una nueva propiedad *shapeType* que permita seleccionar el tipo de figura que se desea representar con este módulo. Los posibles valores que esta propiedad puede tomar son:

```
shapeType: rect|circle|text
```

Por último, para definir cada uno de los atributos de la figura seleccionada, se define un nuevo objeto denominado *shape*, con propiedades que varían según la figura que se esté representando. En la Figura 4.6 se puede ver un ejemplo de todas las propiedades disponibles en el objeto *shape* para definir un Rectángulo. Para las figuras Círculo y Texto puede consultar el anexo. Algunas propiedades del objeto *shape* varían, pero el proceso es análogo.

```
data: [{
    shape: {
      w : 6.0,
      h : 3.0,
      x : 0,
      y : 0,
      fill :"red",
      stroke: "#7f8c8d",
      rotation : -45
    },
    graphType: "shape",
    shapeType: "rect"
}]
```

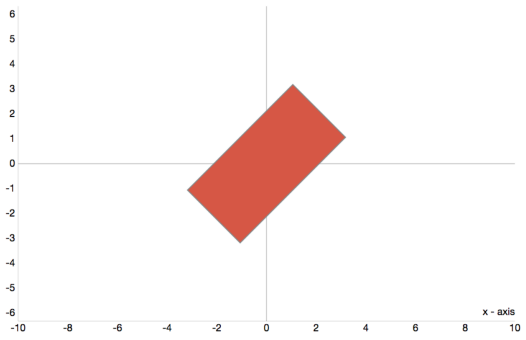


Figura 4.6: Representación de un rectángulo utilizando el módulo Shape.

Módulo Polyline

Los Polígonos y Polilíneas, a diferencia del resto de las figuras, quedan definidos mediante un conjunto de puntos unidos por segmentos de rectas. Se hace uso de la representación de puntos que FunctionPlot posee en este módulo para representar estas dos figuras.

La primera propiedad que se debe definir es para activar el uso del módulo Polyline:

```
graphType: polyline
```

La segunda propiedad es para especificar para qué se desea utilizar la representación de puntos:

```
fnType: points
```

También, como el caso del módulo Shape, se define una nueva propiedad polylineType para poder seleccionar el tipo de figura que se desea representar. Los posibles valores son:

```
polylineType: line|polygon
```

Al igual que con las figuras del módulo Shape, se deben determinar los atributos que definen la figura a representar. En la Figura 4.7 se muestra un ejemplo de Polígono y en el anexo se puede consultar el caso de Polilínea. Un Polígono consta de diferentes puntos (x,y) unidos por segmentos, donde el último punto debe unirse con el primero, y para definir dichos puntos se utiliza la propiedad *points* de FunctionPlot. Esta propiedad consta de una lista de puntos, donde cada punto (x,y) está representado por una lista de largo dos que representa a x e y, como se puede apreciar en el siguiente pseudocódigo:

```
points: [
    [x0, y0],
    [x1, y1],
    ....
```

```
]
```

```
data: [{
    points: [
      [-8,3],
      [-6,1],
      [-4,2],
      [-6,6]
    ],
    fill :"white",
    stroke: "green",
    rotation : 0,
    boundingBox: true,
    fnType: 'points',
    polylineType : "polygon",
    graphType: 'polyline',
}]
```

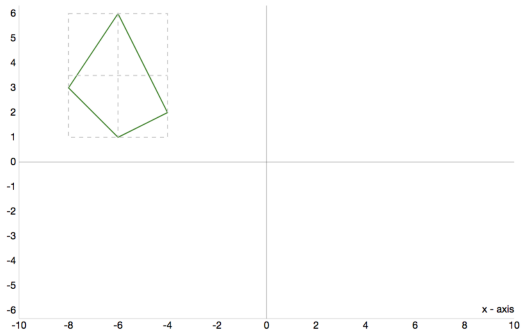


Figura 4.7: Representación de un polígono utilizando el módulo Polyline.

Para el caso de Polígonos y Polilíneas hay un elemento nuevo que se agrega, y que es útil para calcular el centro de rotación de la figura, denominado BoundingBox, el cual puede ser visible en el gráfico utilizando la propiedad BoundingBox en true, como se puede ver en el ejemplo. Por defecto, esta propiedad fue establecida en false.

4.3.1.2. Funciones.

Para el caso de funciones se analizaron los diferentes tipos de funciones que pueden ser definidas con el lenguaje MateFun, así como las diferentes discontinuidades que puede presentar una función. En primer lugar, señalar que FunctionPlot no permite definir el dominio y codominio de una función, por el contrario, en esta biblioteca se considera que la función a representar está definida para todo punto que pertenece al conjunto de los reales (R). Esto implica que funciones definidas sobre conjuntos diferentes a R o sobre subconjuntos de R, no puedan ser representadas por esta biblioteca o son representadas de forma errónea. En segundo lugar, destacar que la biblioteca sólo posee un método para detectar discontinuidades asintóticas, para el resto de las discontinuidades será necesario implementar nuevos métodos que detecten esto.

Junto al objeto *data* utilizado en FunctionPlot para definir las funciones y figuras geométricas a representar, se agrega un nuevo objeto denominado *conj*,

utilizado para definir los conjuntos dominio y codominio de cada función. En este objeto se podrán utilizar los siguientes conjuntos:

```
Reales (R) | Enteros (Z) | Enumerados (Numer) |
Subconjuntos de R y Z (Func)
```

Internamente, FunctionPlot tuvo que ser modificada para que evalúe la función únicamente en puntos del dominio, y aplique diferentes métodos dependiendo el conjunto que se esté utilizando. Una función definida sobre un dominio discreto como es el caso de Z, un subconjunto de Z o un enumerado, se representa mediante puntos utilizando el módulo Scatter de FunctionPlot, mientras que para funciones definidas sobre R o un subconjunto de R, se utilizan los módulos Polyline e Interval, que aplican los métodos de aproximación de la función por líneas o intervalos, y analizan la discontinuidad asintótica de la función. Además, para dominios discretos, se modificó el método que selecciona los puntos del dominio en los que es evaluada la función. Este método, originalmente, selecciona un número finito de puntos equiespaciados del dominio, para que luego la función sea evaluada en estos puntos y aproximada en los puntos intermedios por una recta o intervalo. Esto funciona bien en dominios reales, donde no es posible evaluar la función en cada punto del domino, pero en dominios discretos, es necesario evaluar la función en cada punto que se represente en la gráfica. Se modificó entonces el método para que retorne todos los puntos del dominio que se encuentren en el rango de visualización de la gráfica, y esos serán los puntos utilizados para evaluar la función. Este cambio permitió representar correctamente funciones definidas sobre dominios discretos, pero también, logró degradar notablemente el sistema, lo cual se debe a que, a diferencia del método original donde la función es evaluada siempre en un número constante de puntos, en dominios discretos el número de puntos depende del tamaño del rango de la gráfica y crece linealmente con éste. Si bien la gráfica es creada con un rango determinado (ej. rango de -10 a 10), el usuario puede alejar la gráfica, aumentando así este rango y generando más puntos que pueden pertenecer al dominio discreto y en los que será evaluada la función (ej. rango de -10000000 a 10000000). La solución que se implementó fue introducir un parámetro (*zoom*) que defina la cantidad máxima de puntos enteros que puede tener el rango de la gráfica. De

esta forma, un usuario puede alejar la gráfica como máximo hasta generar un rango de puntos enteros igual al parámetro *zoom*. Por defecto, se estableció el valor de *zoom* en 2000, pudiendo ser modificado por el usuario.
Al agregar información del dominio y codominio de una función, es posible ahora representar con la biblioteca funciones discretas, o funciones definidas sobre conjuntos densos, pero que no necesariamente sean el conjunto R. También se utiliza esta información para definir los valores que se muestran en los ejes x e y de la gráfica. La Figura 4.8 muestra un ejemplo de uso de la propiedad *conj*.

```
conj: [{
  radio: 2,
  baseDom: 'Z',
  baseCod: 'Z',
  dom: 'Func',
  cod: 'Z',
  sets: {
    fcod:'Z',
    fdom: function (x) {
      var cond = (x > 2 && x!=
          4 && x!=5)
      return cond;
    }
  }
}],
data: [{
 graphType: 'scatter',
 id: 0,
 fn: function (scope) {
   var x = scope.x
   return x
 }
]
```

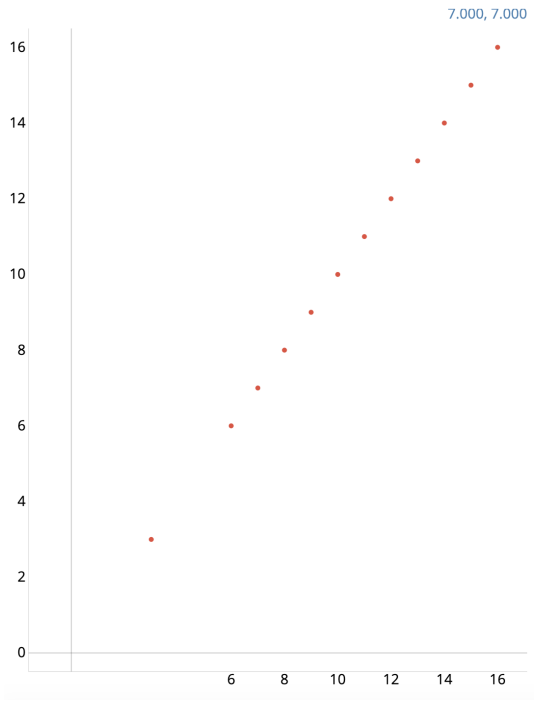


Figura 4.8: Función $f(x) = x$ definida en un subconjunto de Z.

Las propiedades *baseDom* y *baseCod* son utilizadas para definir el conjunto base del dominio y codominio respectivamente, pudiendo ser este conjunto base R o Z, mientras que en las propiedades *dom* y *cod*, se define si los conjuntos son R o Z, subconjuntos de estos (Func), o un enumerado (Enum). Por último, la propiedad *sets* es utilizada para indicar las funciones que definen a dichos conjuntos, y la propiedad *radio* fue agregada para parametrizar el tamaño del círculo.

Con respecto al estudio de la continuidad de las funciones, se pueden separar los tipos de discontinuidades de salto infinito y asintótica del resto de las discontinuidades. Para las primeras, el método que ya está desarrollado en FunctionPlot, detecta este tipo de discontinuidades para una gran familia diferente de funciones, mientras que el resto de las discontinuidades (evitables, de salto finito y de segunda especie) se presentan en funciones que son definidas por partes y donde el método mencionado anteriormente no fue desarrollado para detectarlas. Cabe aclarar, también, que hay un tipo de discontinuidad de segunda especie que se presenta en funciones que oscilan demasiado rápido cerca de un punto, que aparecen en funciones que no necesariamente están definidas por partes, como es el caso de la función $f(x) = sen(1/x)$ en el punto $x = 0$. De todas formas, para este tipo particular de discontinuidad, los métodos desarrollados en los módulos Polyline e Interval renderizan correctamente la gráfica de la función en ese punto, y no es necesario crear un método nuevo.

Retomando el caso de las funciones definidas por partes, se decidió desarrollar un método que contemple dicho caso y que sea el Módulo 2D el que lleve a cabo esta tarea. Este método consiste en que el Módulo 2D identifique si la función a renderizar es una definida por partes, y en ese caso, debe dividirla en funciones disjuntas (una función diferente por cada parte o condición de la función original). Luego, las mencionadas funciones son enviadas mediante la propiedad *data* a FunctionPlot, y al ser diferentes, FunctionPlot analizará y representará cada una de forma independiente una de otra. De esta forma, se evita tener que analizar internamente en FunctionPlot cómo se comporta la función en los extremos de cada parte o intervalo. En la Figura 4.9 se muestra un ejemplo de cómo renderiza FunctionPlot una función por partes donde no se aplique este método, mientras que en la Figura 4.10 se muestra el mismo ejemplo aplicando este método.

```
data: [{
  graphType: 'polyline',
  fn: function (scope) {
    var x = scope.x
    if (x < -4) {
        return -x
    } else if (x >= -4 && x<=5)
        {
        return Math.sin(x)
    } else {
        return -x
    }
  }
}]
```

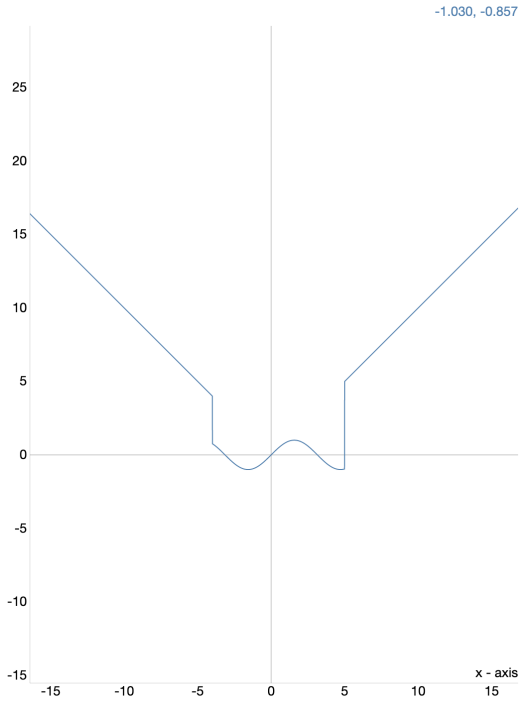


Figura 4.9: Función definida por parte sin dividirla en funciones diferentes.

```
data: [{
  graphType: 'polyline',
  fn: function (scope) {
    var x = scope.x
    if (x < -4) {
      return -x
    }
  }
  },{
  graphType: 'polyline',
  fn: function (scope) {
    var x = scope.x
    if (x >= -4 && x <= 5) {
      return Math.sin(x)
    }
  }
  }, {
  graphType: 'polyline',
  fn: function (scope) {
    var x = scope.x
    if (!(x < -4) && !(x >= -4
        && x <= 5)) {
      return -x
    }
  }
}]
```

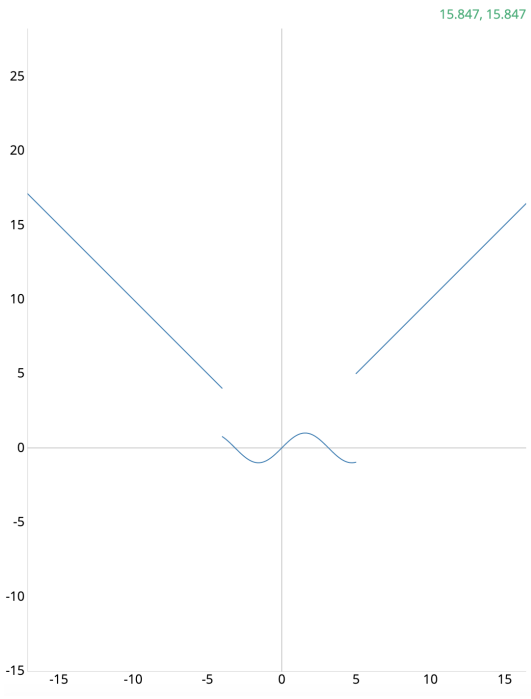


Figura 4.10: Función definida por parte dividiendo cada parte en una función diferente.

4.3.2. Módulo 2D.

El módulo 2D forma parte de un conjunto de módulos implementados en Angular 4, los cuales funcionan conjuntamente y definen la interfaz del entorno web de MateFun. Este módulo es utilizado específicamente cuando la respuesta del intérprete a un comando ingresado por el usuario en la interfaz, es la representación gráfica en 2D de una función, figura o animación. El módulo 2D, debe procesar este tipo de respuesta, generar la representación gráfica e implementar los controles necesarios para interactuar con la gráfica. Para esto, hace uso de la biblioteca FunctionPlot extendida.

Como se mencionó en la sección arquitectura (ref), la respuesta del servidor es un objeto JSON con 2 propiedades, tipo y resultado. Los posibles valores que la propiedad tipo puede tomar cuando la respuesta es una representación gráfica son:

```
tipo: canvas|animacion|graph
```

Los valores *canvas* y *animación* son utilizados para indicar que la respuesta contiene la definición de figuras y animación respectivamente, mientras que el valor *graph* es utilizado para el caso de funciones. En las siguientes secciones se detallan las decisiones tomadas para cada uno de estos tipos, algunas de las cuales implicaron modificaciones al lenguaje MateFun.

4.3.2.1. Figuras.

```
tipo: canvas
```

Este tipo de respuesta tiene en la propiedad resultado una lista de figuras a graficar. Cada figura en esta lista, queda definida por un objeto con propiedades específicas que definen geométricamente a la figura y sus atributos. En la Figura 4.11, se muestra un ejemplo de la propiedad resultado para este tipo de respuesta. En el anexo se detalla el objeto utilizado para definir cada una de las figuras de MateFun.

Los objetos que definen cada figura dentro de la propiedad resultado, son muy similares a los implementados en FunctionPlot para representar figuras.

```
resultado : [{
    tipo: "circle",
    x: 1,
    y: 1,
    r: 0.5,
    color: 'white',
    rotacion: 0
}, {
    tipo: "rect",
    x: 2,
    y: 4,
    w: 10,
    ...
}]
```

Figura 4.11: Ejemplo de propiedad resultado para el caso de figuras.

El módulo 2D sólo debe crear el objeto que utiliza FunctionPlot con estos datos, y agregar el resto de las propiedades utilizadas por la biblioteca, como la propiedad para seleccionar el tipo de módulo que se usará, Polyline o Shape.

4.3.2.2. Animaciones.

```
tipo: animacion
```

El formato de la propiedad resultado para el caso de animaciones es muy similar al caso de figuras, con la diferencia que en animaciones, resultado es una lista de *frames*, donde cada *frame* es una lista de figuras. Cada figura se representa con las mismas propiedades que en el caso anterior. Ver ejemplo en la Figura 4.12,

En este caso, es necesario crear la lógica para ejecutar la animación, los controles para que el usuario pueda comenzar y pausar la animación, y los controles para aumentar y disminuir la frecuencia con que se cambia cada *frame*. Mientras no se pause la animación, ésta es ejecutada en un bucle infinito, retornando al primer *frame* cuando se llega al último. Para dibujar cada *frame* se utiliza la misma solución que en la sección anterior para el caso de figuras, y se utilizan los métodos implementados en FunctionPlot para limpiar y dibujar nuevos objetos en la misma gráfica. Estos métodos de FunctionPlot permi-

```
resultado : [
    [{
        tipo: "circle",
        x: 1,
        y: 1,
        ...
    }, {
        tipo: "rect",
        ...
    }], [{
        tipo: "text",
        text: "texto a mostrar",
        x: 1,
        ...
    }]
]
```

Figura 4.12: Ejemplo de propiedad resultado para el caso de animación.

ten que objetos del gráfico que no cambian, como la grilla o los ejes, no sean dibujados nuevamente en cada *frame*. En este módulo 2D se decide implementar el cambio de *frame* y la lógica de la animación, dejando a FunctionPlot únicamente la responsabilidad de representar correctamente las figuras.

4.3.2.3. Funciones.

```
tipo: graph
```

En los dos casos anteriores, el formato del objeto JSON que retorna el servidor permaneció sin cambios y no fue necesario modificar el lenguaje MateFun, siendo solamente necesario implementar una nueva lógica que utilizara FunctionPlot y lo implementado como extensión en esta biblioteca. La situación con las funciones fue diferente, se decidió agregar nuevos elementos al lenguaje MateFun, que mejoraran la representación gráfica de algunos tipos de funciones. También fue necesario modificar el objeto JSON, para incluir estos nuevos elementos y para mejorar algunos aspectos de la estructura del objeto.

Se agrega al lenguaje MateFun, la posibilidad de trabajar de forma primitiva con el conjunto de los enteros (Z). Hasta el momento, para poder trabajar

con conjuntos numéricos, MateFun sólo tenía como conjunto primitivo al conjunto de los reales (R). El conjunto Z, como otros subconjuntos de R, puede ser definido por el usuario utilizando la notación por comprensión, y utilizando el conjunto R como conjunto primitivo, ver Figura 4.13.

```
Z = { x en R | x == red(x) }
N = { x en Z | x >= 0 }
```

Figura 4.13: Conjunto de los enteros (Z) y naturales (N), definidos a partir del conjunto de los reales (R). La función $red(x)$ es el redondeo de x.

Definir los elementos de un conjunto como los números reales, que cumplen una condición de igualdad, dificulta la representación gráfica de la función. Como se vió en la sección 3.1, sería necesario agregar un parámetro Δ que relajara la condición de igualdad, o implementar algún otro método para encontrar los puntos reales que cumplen esa igualdad. El nuevo conjunto primitivo Z, en el módulo 2D, permite obtener más información sobre el dominio de una función, y saber si este dominio es un conjunto discreto (Z, o un subconjunto de Z), o un conjunto denso (R, o un subconjunto de R). Con esto es posible entonces, tratar cada caso por separado, e implementar diferentes algoritmos que mejoren la representación gráfica de la función, como se hizo al extender FunctionPlot en el ámbito de funciones.

La Figura 4.14, muestra la estructura general de la propiedad resultado del objeto JSON, retornado por el servidor para el caso de funciones. La propiedad *graph* definida en este objeto, tiene el nombre de la función que se debe graficar. La definición de esta función y de otras funciones utilizadas, se encuentran en el arreglo de la propiedad *funs*.

Las propiedades *fun*, *args* y *bdy*, son utilizadas para definir nombre, argumentos y cuerpo de la función respectivamente, mientras que *dom*, *cod* y *sets*, son propiedades nuevas que se deciden agregar para definir los conjuntos dominio y codominio de la función. Anteriormente estas propiedades no existían, únicamente se definía el dominio de la función y éste era incluido junto a la definición del cuerpo de la función en la propiedad *bdy*. La estructura de los objetos utilizados para definir conjunto dominio y cuerpo de la función es la

```
resultado : {
    graph: "Nombre de funcion 1",
    funs: [{
        fun: "Nombre de funcion 1",
        args: [
            "arg 1",
            "arg 2",
            "...",
        ],
        dom: "C1",
        cod: "C2",
        sets: [
            C1: "Objeto que define conjunto C1",
            C2: "Objeto que define conjunto C2"
        ]
        bdy: "Objeto que define cuerpo de funcion 1"
    }, {
        fun: "Nombre de funcion 2",
        ...
    }]
}
```

Figura 4.14: Estructura general de propiedad resultado para el caso de funciones.

misma, por lo que al incluirlos juntos en la propiedad *bdy*, dificulta reconocer cada uno de estos objetos y tratarlos de forma independiente. En la Figura 4.15, se muestra un ejemplo de definición de conjunto y de cuerpo de una función.

```
cond: {
    kind: "bop",
    op: ">=",
    exp1: {
        kind: "var",
        var: "x"
    },
    exp2: {
        kind: "lit",
        val: 0
    }
}
```

Objeto utilizado para definir el conjunto $x \in R \mid x >= 0$.

```
cond: {
    kind: "bop",
    op: "+",
    exp1: {
        kind: "var",
        var: "x"
    },
    exp2: {
        kind: "lit",
        val: 5
    }
}
```

Objeto utilizado para definir $x + 5$ como cuerpo de una función.

Figura 4.15: Estructura utilizada para representar funciones y condiciones

Se decide entonces, separar en propiedades nuevas la definición del conjunto dominio, e incluir también aquí, la definición del conjunto codominio de la función. Además, ahora con el nuevo conjunto primitivo Z, estas definiciones también deben incluir información sobre el conjunto primitivo que se está utilizando (R o Z). Las propiedades *dom* y *cod*, son utilizadas para definir el nombre del conjunto dominio y codominio de la función respectivamente, mientras que la propiedad *sets*, es donde se incluye la definición de éstos. Los conjuntos mencionados hasta aquí, fueron definidos en el lenguaje MateFun utilizando la notación por comprensión. Para los definidos por enumeración, el objeto utilizado en la propiedad *sets*, es un arreglo que contiene el valor de cada elemento del enumerado. En la figura 4.18, se muestra un ejemplo de estas tres nuevas propiedades; la propiedad *set* utilizada en la definición del conjunto N, es para definir el conjunto primitivo que se utiliza en N.

```
dom: "E",
cod: "N",
sets: [
    E: ["Lunes", "Martes", "Miercoles", "Jueves", "Viernes"],
    N: {
        set: "Z",
        var: "x",
        cond: {
            kind: "bop",
            op: ">=",
            exp1: {
                kind: "var",
                var: "x"
            },
            exp2: {
                kind: "lit",
                val: 0
            }
        }
    }
]
```

Figura 4.16: Se utilizan dos conjuntos E y N para definir el dominio y codominio de una función. E es un conjunto por enumeración y N un conjunto por comprensión, definido sobre el conjunto primitivo Z.

Por último, al igual que en el caso de figuras y animaciones, fue necesario crear un algoritmo de parsing para procesar estos datos, y crear el objeto uti-

lizado en FunctionPlot para la representación de funciones en R o Z.

El algoritmo toma como entrada el JSON recibido y retorna un objeto JavaScript. Para poder lograr esto, el algoritmo primero itera sobre el objeto *bdy* generando la función correspondiente y luego recorre los objetos *dom*, *cod* y *sets* para generar el conjunto asociado.

A modo de ejemplo, si se desea graficar la función de la Figura 4.17.

```
funZ :: Z -> Z
funZ (x) = x * x
```

Figura 4.17: Función x^2 en un Z.

Se obtendrá del Servidor el JSON de la Figura 4.18. Como se trata de una función será derivada al algoritmo creado.

```
{ tipo : graph,
   resultado: { graph : funZ,
           funs: [
                      { fun : funZ,
                        args: [x],
                        dom : Z,
                        cod : Z,
                        sets: [{}],
                        bdy: { kind : bop,
                               op : *,
                               exp1 : { kind :var,
                                        var: x},
                               exp2: { kind : var,
                               var : x }
                    }
                      }]
               }
}
```

Figura 4.18: JSON proveniente del Servidor, correspondiente a la función x^2 en Z

En la Figura 4.19 se puede observar el resultado de procesar con el algoritmo el JSON de la Figura 4.18

```
target: '#built-in-eval-
    function',
zoom:2000,
conj:[

{
  radio: 2,
  baseDom: 'Z',
  dom: 'Z',
  baseCod: 'Z',
  cod: 'Z',
  sets: {
    fdom:Z,
    fcod:Z}
  }
],
data: [{
 graphType: 'scatter',
   id: 0,
   fn: function (scope) {
    var x = scope.x
    return ((x))
  },       color:'blue'}
]
```

Figura 4.19: Representación de objeto JavaScript resultante procesar el JSON de la Figura 4.18 por el algoritmo de parsing.

Para finalizar en la Figura 4.20 se observa el resultado obtenido por Function-Plot al enviarle el objeto JavaScript de la Figura 4.19.

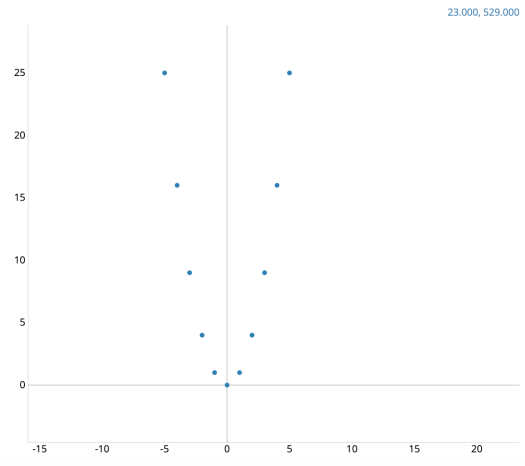


Figura 4.20: Renderización de la funcíon x^2 por Function-Plot con un dominio Entero

Para los tres casos se crearon controles en la interfaz manejados por este módulo, que utilizan los métodos de FunctionPlot y que permiten alejar o

acercar la gráfica, mostrar u ocultar la grilla o los ejes, o limpiar la gráfica entre otras cosas.

4.4. Gráficos 3D

4.4.1. Biblioteca Graph3D.

Graph3D es una biblioteca escrita en código JavaScript, que tiene como objetivo la representación de objetos 3D en un browser, así como ofrecer un conjunto de diferentes controles para la interacción del usuario con el visualizador 3D.

Esta biblioteca permite que, a través de diferentes configuraciones en formato JSON, se puedan representar objetos en 3D.

Es importante destacar que Three.js hace gran parte del trabajo de la representación de objetos 3D en un browser, y permite abstraernos de la complejidad que tiene la representación de figuras 3D.

La diferencia entre Graph3D y Three.js radica en que esta última brinda un gran conjunto de objetos elementales necesarios para la representación en 3D, y Graph3D usa gran parte de los mismos para que puedan ser creados a partir de configuraciones en formato JSON, como también hacer diferentes controles customizados para el visualizador, los cuales serán explicados a lo largo del capítulo.

Para comenzar a conocer a Graph3D, a continuación se muestra su arquitectura en la Figura 4.21

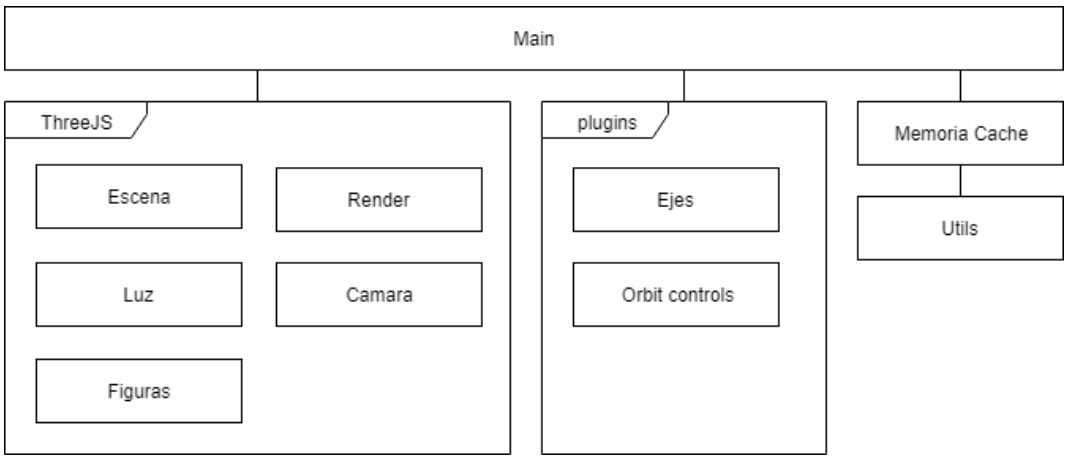


Figura 4.21: Arquitectura de Graph3D.

El grupo Three.js está constituido por los principales módulos utilizados de la biblioteca Three.js, con el agregado de un nuevo módulo denominado Figuras, que se desarrolló para representar las figuras 3D en el visualizador. Este módulo interactúa como un intérprete de Figuras, el cual recibe cierta configuración en formato JSON y lo transforma a objetos de Three.js.
En el grupo Plugins se presentan dos módulos, uno denominado Ejes y otro denominado Orbit Controls. El módulo Ejes brinda la funcionalidad de poder dibujar los ejes tridimensionales, incluyendo una grilla que sirve como guía para la visualización de la posición de los objetos 3D.

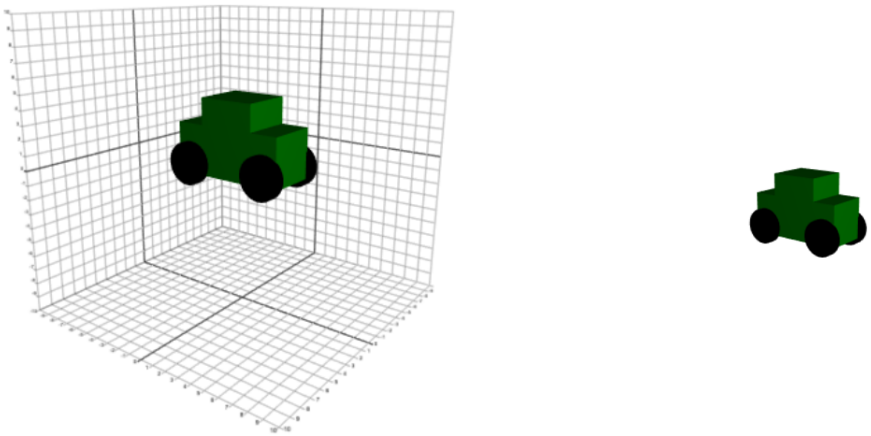


Figura 4.22: Representación con ejes y sin ejes

El módulo Orbit Controls brinda la funcionalidad de poder mover la cámara en forma de órbita alrededor de un objetivo, y gracias a ello se puede conseguir una visualización completa del mundo 3D.

Por último, encontramos dos módulos, por un lado Memoria Caché y por otro un módulo utilitario de funciones llamado Utils.

La Memoria Caché es un módulo desarrollado para mejorar la performance de las animaciones, lo cual será analizado posteriormente en la sección 4.4.2

Three.js es una biblioteca para mostrar y animar gráficos 3D en el navegador, el cual posee dentro de sus cualidades, la capacidad de proveer una gran variedad de objetos 3D elementales que facilitan el uso y la comprensión de sus funcionalidades.

Para construir un modelo de objetos en 3D usando Three.js, se deben utilizar, en conjunto, diferentes objetos que provee la biblioteca, los cuales serán analizados con la finalidad de conocer su función y la forma en la cual son

utilizados.
En primer lugar, se encuentra el objeto Renderer, que es el encargado de realizar el procesamiento de todos los objetos representados en Three.js para que puedan ser visualizados a través del elemento Canvas de HTML5.
Todos los objetos del mundo 3D se encuentran contenidos dentro de una escena, la cual también está conformada por dos objetos fundamentes que son las cámaras y luces.
La cámara representa los ojos con los que se visualiza el mundo 3D. A su vez, la cámara puede usar dos tipos distintos de proyecciones: en perspectiva e isométrica. La proyección en perspectiva refiere a la forma que se ve el mundo real, deformando los objetos dependiendo de su posición y distancia en la que se encuentren respecto a la cámara. En cambio, la isométrica mantiene el tamaño de los objetos, independientemente de la distancia en que se encuentren de la cámara.
La luz permite ver con claridad los colores de los diferentes objetos de la escena, los cuales están constituidos por una geometría representada por un conjunto de vértices y caras. Los vértices son puntos en el espacio 3D y las caras son triángulos formados por la unión de tres puntos.
Además de la geometría, los objetos se componen de un material que representa el revestimiento de éstos, indica de qué color será el objeto 3D y de qué forma impactará la luz sobre el mismo.

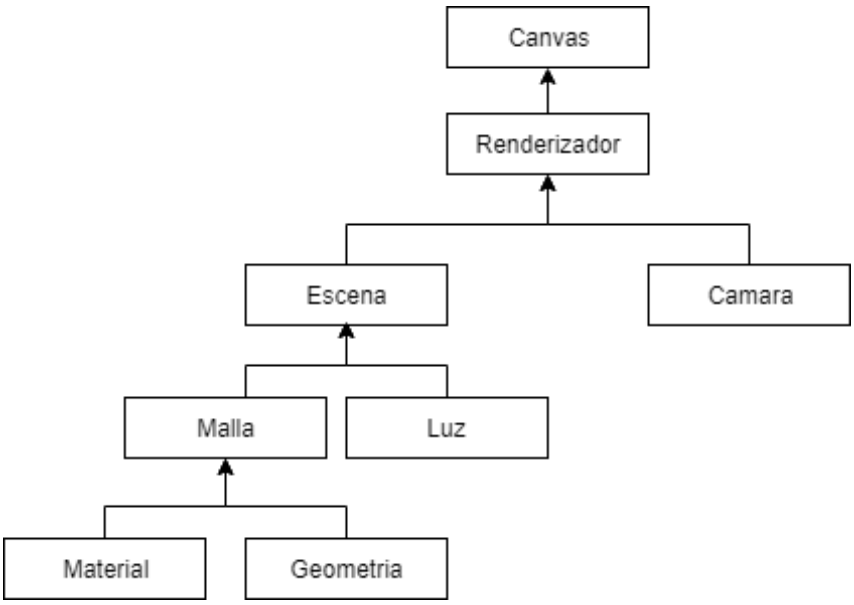


Figura 4.23: Relación entre los diferentes objetos de Three.js

Para poder comprender cómo se consigue generar la visualización de objetos 3D, a continuación se describe un breve ejemplo de cómo se genera un ambiente en 3D.

```
// Se crea la escena
var scene = new THREE.Scene();
// Se configura la cámara, indicando fov (campo de visión), aspecto, near, far
var camera = new THREE.PerspectiveCamera(45, width/height, 0.1, 1000);
// Se crea el renderizador WebGL
var renderer = new THREE.WebGLRenderer();
// Se crea el punto de luz, de color blanco
var light = new THREE.PointLight(0xffffff);
// Se inserta la luz a la cámara
camera.add(light);
// Se inserta a la escena la cámara
scene.add(camera);
// Se renderiza
renderer.render(scene, camera);
```

En el ejemplo anterior, se logró ver en forma simplificada cómo se genera el ambiente 3D en Graph3D. Luego de que se dispone del ambiente 3D, se da paso a la adición de figuras en 3D a la escena. A continuación se muestra un ejemplo de como Three.js simplifica el trabajo abstrayéndonos de la complejidad en cuanto a la representación de objetos 3D. Se procede a definir la geometría del objeto y el material del mismo. Además, se muestra cómo añadir

un prisma a la escena.

```
// Se crea la geometría de un cubo con parámetros
    (ancho, altura, profundidad)
var geometry = new THREE.CubeGeometry(2,2,3);
// Se crea el material del objeto 3D, que en este
    ejemplo será de color azul
var material = new THREE.MeshLambertMaterial({
    color: new THREE.Color('Blue')
});
// Se genera la malla para la geometría y el
    material dado
var mesh = new THREE.Mesh(geometry, material);
// Se agrega a la escena
scene.add(mesh);
```

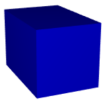


Figura 4.24: Ejemplo de representación

Otro de los módulos desarrollados es el módulo de los Ejes que fue desarrollado a medida para mejorar la visualización de Graph3D.

Para su implementación se generó un conjunto de líneas equiespaceadas, por una separación dada por parámetro. Las líneas son representadas usando el objeto Line de Three.js.

```
// Se crea el material para el objeto Line
var material = new THREE.LineBasicMaterial({
    color: 'black',
    linewidth: 0.1
});
// Se crea la geometría de la línea, con los puntos (x0,y0,z0) y (x1,y1,z1)
var geometry = new THREE.Geometry();
geometry.vertices.push(new THREE.Vector3(x0, y0, z0));
geometry.vertices.push(new THREE.Vector3(x1, y1, z1));
// Se crea la línea con la geometría y el material designado
var line = new THREE.Line(geometry, material);
```

Entre las diferentes funcionalidades que ofrece Graph3D se encuentra la de poder representar figuras estáticas, tales como esfera, prisma, cilindro, anillo y línea. Para poder representar cada una de las figuras, se debió especificar un conjunto de parámetros de configuración para cada una de ellas, dependiendo del tipo de figura. Para el caso de un prisma se debe especificar el ancho, la altura y la profundidad, los cuales son parámetros específicos para ese tipo de figura. Además, hay otros parámetros que representan la posición del objeto: el color y la rotación del mismo, los cuales son comunes a todos los objetos.

Estas configuraciones son representadas en formato JSON y enviadas como parámetros a la función drawFigures, que provee la API de Graph3D.

```
{
  "tipo": "cube",
  "x": 4,
  "y": 8,
  "z": 5,
  "w": 1.5,
  "h": 2,
  "l": 1,
  "color": "verde",
  "rotacion": {
    "x": 45, "y": 0, "z": 0
  }
}
```

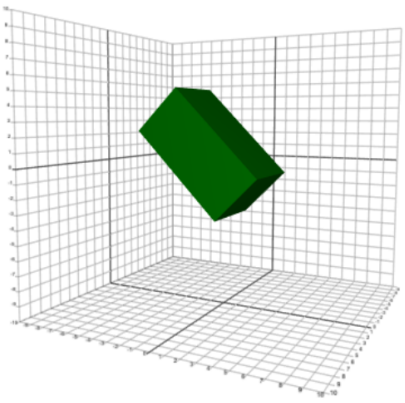


Figura 4.25: Representación de un prisma utilizando Graph3D.

Para las demás figuras como la esfera, cilindro, entre otros, la configuración de JSON es similiar a la mostrada en la Figura 4.25, variando en la cantidad de parámetros que requiere la representación, ya que para el caso de la esfera sólo es necesario indicar cuál es su radio.

4.4.1.1. Figuras 2D en tres dimensiones.

Otra de las funcionalidades que se implementó en Graph3D es la de poder representar shapes en 3D, tales como círculo, rectángulo y, de forma más general, polígonos. Estos objetos son representados con la misma lógica que en el esquema de configuración que ya vimos para los objetos 3D.

En este caso, Three.js no provee geometrías predefinidas para poder representar los shapes, pero lo que sí brinda es un objeto llamado Shape con diferentes métodos que nos permite obtener como resultado las figuras que deseamos renderizar.

Para poder comprender cómo se obtienen los resultados a través del objeto Shape, se muestra en el siguiente ejemplo cómo se representa el rectángulo en Graph3D.

```
var shape = new THREE.Shape();
// Se posiciona en el vértice desde el cual se comienza
shape.moveTo(x - w, y - h);
// Se traza la primer línea
```

```
shape.lineTo(x, y - h);
// Se traza la segunda línea
shape.lineTo(x, y);
// Se traza la tercera línea
shape.lineTo(x - w, y);
// Se traza la línea que cierra el rectángulo
shape.lineTo(x - w, y - h);
```

Para el caso de representación de polígonos con más caras, el código es bastante similar, pero en el caso del círculo se deja de utilizar el método *lineTo* y se pasa usar el método *arcTo*. Ambos métodos difieren en el punto de si la curva que une dos puntos se debe de representar como una línea recta o como un arco.

En la Figura 4.26 se presenta un ejemplo con varias figuras representadas

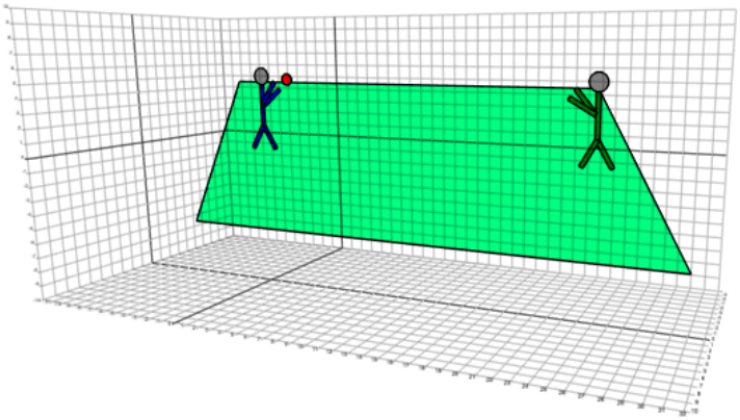


Figura 4.26: Ejemplo 2D en 3D

Cabe destacar que la representación de figuras 2D en 3D dimensiones no se encuentra integrada en MateFun, ya que no fue agregada al núcleo. Se espera que como trabajo a futuro la misma pueda ser integrada.

4.4.1.2. Representación de curvas paramétricas y superficies.

Es de gran interés la visualización de funciones matemáticas en 3D, más precisamente las superficies y curvas paramétricas, por lo cual se llegó a una representación de las mismas utilizando Graph3D.

Durante la investigación de la misma se encontró una fuente con ejemplos de ThreeJS (12) que ya solucionaba gran parte del problema, por lo que se

decidió utilizarla y extender dicha solución.

La solución encontrada en la fuente mencionada, cubría gran parte de la representación de superficies y curvas paramétricas utilizando Three.js. Nuestro foco fue puesto en poder añadirla a la librería Graph3D, sumándole la capacidad de que pueda ser representada a través de una configuración en formato JSON.

Una limitación importante, que aún resta resolver, es que la evaluación de las diferentes funciones matemáticas sólo son consideradas en los casos en que dicha función existe, quedando pendiente el análisis de los casos en lo que es discontinua o la función no existe.

A continuación, se presentan ejemplos de dos funciones que son representadas usando Graph3D. El primer ejemplo muestra la representación para una superficie y el segundo para una curva paramétrica.

```
{
  "tipo": "surface",
  "fn": "sin(sqrt(pow(x,2) +
      pow(y,2)))"
}
```

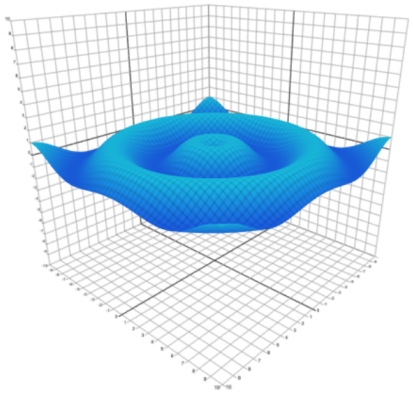


Figura 4.27: Representación de una superficie utilizando Graph3D.

Para la representación de este ejemplo se definió el tipo de gráfica como 'surface' y la función matemática la cual consta de dos variables libres.

```
{
  "tipo": "parametric",
  "u0": 0,
  "u1": 2*PI,
  "v0": 0,
  "v1": 2*PI,
  "fn": {
    "x": "cos(u)*(6 + 2*cos(v))
        ",
    "y": "sin(u)*(6 + 2*cos(v))
        ",
    "z": "2*sin(v)"
  }
}
```

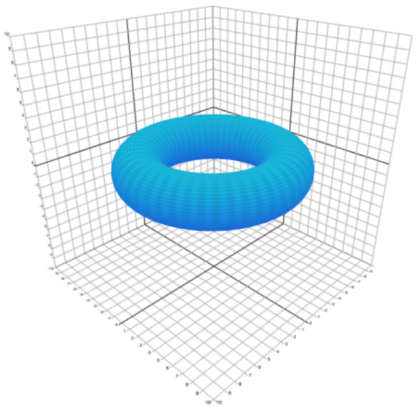


Figura 4.28: Representación de una curva paramétrica utilizando Graph3D.

Para la representación de funciones paramétricas se requiere definir los intervalos angulares de la función a representar. Es por ello que, en este ejemplo, además de definir el tipo de gráfica y la función matemática, se definen u y v como los intervalos angulares.

Al igual que en la sección anterior, esta funcionalidad no se encuentra integrada en MateFun.

4.4.1.3. Animaciones

Las animaciones están representadas por una lista de frames, donde cada uno está compuesto por una lista de objetos 3D. Para renderizar animaciones en el renderizador se siguen los siguientes pasos:

1. Se limpian los objetos que se encuentran en el visualizador.
2. Se espera un tiempo programado de retardo entre cada frame.
3. Se añaden al visualizador los objetos del frame siguiente.

Para lograr una animación atractiva al usuario, el tiempo entre cada frame debe ser despreciable. Debido a que la tarea de crear un objeto 3D es algo costoso, se diseñó una lógica de código para dar soporte a esta problemática. En la gran mayoría de las animaciones hay muchos objetos que se mantienen y no desaparecen, lo único que varía es la posición o rotación en la escena, por lo que se implementó una memoria caché para poder mantener los objetos en memoria y que puedan ser reutilizados en las próximas escenas, lo cual ahorra el tiempo de procesamiento. La memoria caché implementada funciona

almacenando una colección de items, de tamaño reducido, donde cada item está compuesto por la siguiente estructura:

```
item: {
    clave: "<clave>"
    objeto: [Object 3D]
    acceso: "<time-stamp>"
]
```

El campo clave representa un identificador del objeto 3D, y está compuesto por los atributos de la figura. Para el caso de una esfera de radio igual a 2 y color azul, la clave sería: **sphere-2-azul**, en objeto se almacena la instancia de la figura 3D y en acceso se asigna un time-stamp de la última vez que se hizo uso del objeto. A su vez, durante el procesamiento de cada frame de la animación se mantiene una lista de los objetos utilizados de la caché, ya que podríamos tener dos objetos que tengan diferentes posicionamientos pero su clave es la misma y llevaría a que para la representación de dos figuras diferentes se esté manipulando la referencia hacia un mismo objeto. Es por eso que en cada paso de la animación se inicializa una lista vacía y durante la iteración de figuras presentes en el frame se van agregando sus respectivas claves como visitadas, y si hay repeticiones de acceso a una misma clave se realiza un clonado del objeto. Como la memoria caché implementada tiene capacidad acotada, se debió definir una política de reemplazo para el caso que ésta se encuentre completa y se desee añadir un nuevo elemento. Para decidir qué elemento debe ser reemplazado, se itera sobre todos los elementos y se reemplaza aquel que no haya sido accedido por más de una cierta cantidad de segundos, definido paramétricamente.

4.4.2. Análisis del rendimiento de la caché

Tal como fue nombrado en la sección 4.4.1, el módulo Memoria Caché fue creado con el fin de mejorar la performance de las animaciones en 3D. Dado que los frames recibidos para las animaciones en 3D pueden contener objetos repetidos entre dos frames consecutivos, se ideó una caché para poder almacenar dichos objetos y no tener que renderizarlos nuevamente.

Para poder analizar el rendimiento de la memoria caché desarrollada, se realizaron varias pruebas en las cuales se observaron resultados similares.
A modo de ejemplo exponemos los resultados obtenidos sobre dos conjuntos

de datos diferentes.

En el primero, se consideró una animación que consta de 24 frames, en el que gran parte de los objetos presentes sólo sufren cambios de posicionamiento en cada frame.

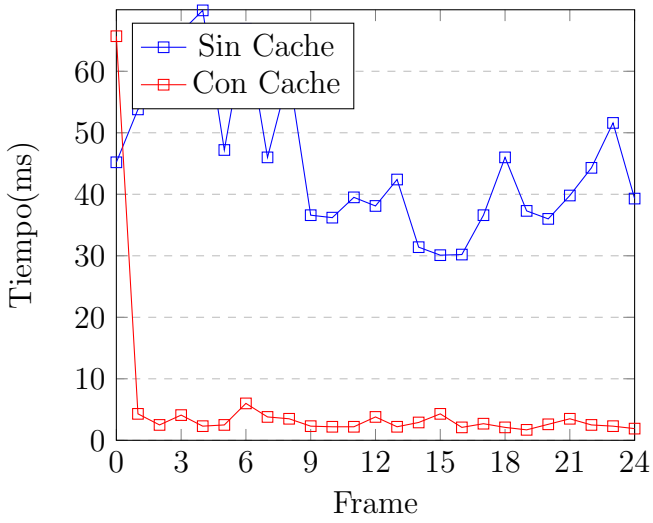


Figura 4.29: Gráfica de Tiempo/Frame sobre una animación en 3D con objetos repetidos.

Podemos observar que en la Figura 4.29 la utilización de la caché mejora notoriamente el tiempo en que se procesan los frames.

En el segundo caso de estudio, utilizamos una animación de 24 frames, en el que varían los distintos objetos en cada frame, de forma que la caché no sea de utilidad.

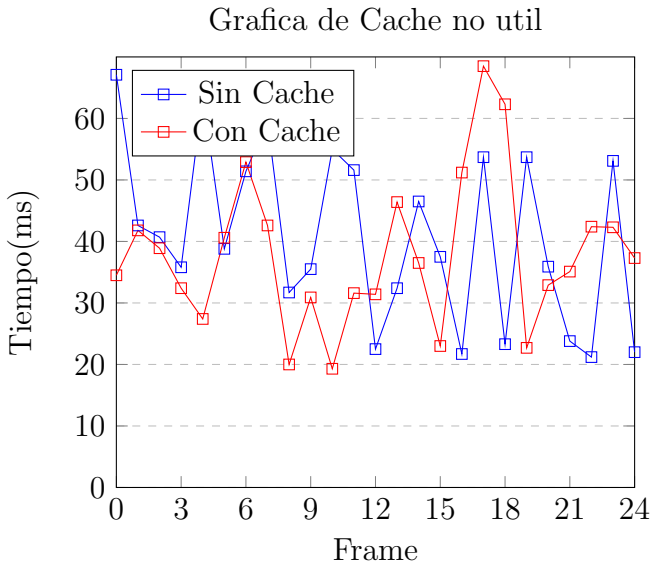


Figura 4.30: Gráfica de Tiempo/Frame sobre una animación en 3D con objetos diferentes.

En la Figura 4.30 se puede apreciar que los tiempos en promedio son similares, lo cual nos permite concluir que si existen objetos repetidos, la performance mejorará de forma notoria, mientras que si la animación a renderizar contiene frame con objetos variantes, la performance no mejorará pero tampoco tendrá un impacto negativo.
Ambas pruebas fueron realizadas sobre una máquina con sistema operativo Windows 10 de 64 bits, procesador Intel Core i7-8550U 1.8GHz, memoria RAM de 16 GB y 3 MB de memoria Caché.

Capítulo 5

Conclusiones y trabajo a futuro.

Como conclusión general del proyecto, destacamos que cumplimos con todos los requerimientos presentados dentro del alcance de este proyecto de tesis. La representación en 2D fue creada a partir de una nueva solución, que mejora lo que ya estaba implementado, y extiende las funcionalidades para dar soporte a funciones definidas en dominios y codominios diferente a los reales. Esta nueva solución, además, está basada en la biblioteca FunctionPlot, de código abierto, que se encuentra mantenida por su autor Mauricio Poppe y por una comunidad activa, lo que implica que en el futuro nuevas funcionalidades y arreglos desarrollados en la biblioteca puedan ser incluidas al proyecto MateFun. Para la representación en 3D, se creó una nueva biblioteca desde cero y se extendió el lenguaje MateFun para brindar la posibilidad de trabajar con figuras geométricas en 3D. Ambos desarrollos de las bibliotecas, son independientes entre sí y entre la aplicación web de MateFun, facilitando la tareas de mantenimiento y extensión.

En relación a las tecnologías utilizadas para el renderizado se encontró que SVG y Three.js tienen curvas de aprendizaje liviana y se adaptan muy bien al uso que se les da aquí. SVG, al ser un imagen vectorial, hace sencillo el trabajo con figuras geométricas teniendo muchas opciones para representar y modificar individualmente cada figura. D3.js también facilitó aún más el trabajo con SVG, tiene una curva de aprendizaje más pronunciada pero permite asociar los datos que se desean representar con su representación gráfica, facilitando el trabajo con ambos y dando como resultado un código más limpio. Además, posee soluciones ya implementadas para ciertos aspectos de la representación gráfica como el zoom, arrastre del gráfico, escalas, etc. Three.js, al igual que

D3.js, resuelve muchos de estos aspectos y oculta mucha de la complejidad que tiene representar elementos en 3D con WebGL.

A continuación, nos parece pertinente listar algunos puntos, que si bien quedan fuera del alcance de nuestro trabajo, creemos que pueden tenerse en cuenta para profundizar, a futuro, lo planteado en este trabajo.

- Mejorar los métodos implementados para detectar discontinuidades en gráficas de funciones en 2D, y adaptar estos métodos para incluir casos que no estén contemplados.
- Agregar nuevas opciones a la interfaz web de MateFun, para poder señalar en la gráfica de una función conceptos matemáticos como raíces, derivada en un punto, función derivada, integral definida, etc. Function-Plot fue creado originalmente para poder trabajar con alguno de estos conceptos y para incluirlos, sería necesario implementar la lógica del lado de la aplicación web de MateFun. En el caso de 3D, cada uno de estos conceptos deben ser implementados en la biblioteca Graph3D y en la aplicación web de MateFun.
- Adaptar el lenguaje MateFun para poder incluir el desarrollo hecho en la biblioteca Graph 3D, con relación a curvas paramétricas, superficies y figuras en 2D.
- Extender la biblioteca Graf3D para que pueda graficar funciones discontinuas.
- Mejorar la evaluación de funciones matemáticas para las funciones a representar en 3D.
- Ampliar la variedad de tipos de figuras a representar en 3D.

Referencias bibliográficas

[1] *ECMA*. URL: https://www.ecma-international.org/.

[2] *Khronos Group - Webgl*. URL: https://www.khronos.org/webgl/.

[3] *Three.js - JavaScript 3D library*. URL: http://mrdoob.github.com/three.js/ [Último acceso 28 Mayo 2019].

[4] *W3C*. URL: https://www.w3c.es/Consorcio/.

[5] *WHATWG*. 2018. URL: https://whatwg.org/.

[6] *Geogebra.js - JavaScript library*. 2019. URL: https://wiki.geogebra.org/es/Publicaciones#Art.C3.ADculos [Último acceso 31 Mayo 2019].

[7] M. G. Bruno Juliá-Díaz. *Análisis matemático de una variable*. Publicacions i Edicions de la Universitat de Barcelona, Barcelona, 2008.

[8] G. Cameto and M. Méndez. *MateFun (IDE Web + Lenguaje Funcional Específico)*. Proyecto de grado, Facultad de Ingeniería, Montevideo, Uruguay, 2017.

[9] CERN. *A short history of the Web*. URL: https://home.cern/science/computing/birth-web/short-history-web.

[10] A. C. Communications. *Flash The Future of Interactive Content*. 2017. URL: https://theblog.adobe.com/adobe-flash-update/ [Último acceso 25 de Julio 2017].

[11] M. Poppe. *Function Plot*. 2015. URL: https://mauriciopoppe.github.io/function-plot/ [Último acceso 31 Mayo 2019].

[12] L. Stemkoski. *Three.js Examples*. 2013. URL: https://stemkoski.github.io/Three.js/ [Último acceso 31 Mayo 2019].

[13] UNESCO. *Aportes para la enseñanza de la matemática*. Organización de las Naciones Unidas para la Educación, la Ciencia y la Cultura, 7, place de Fontenoy, 75352 París 07 SP, Francia y la Oficina Regional de Educación de la UNESCO para América Latina y el Caribe, OREALC/UNESCO Santiago, 2016. URL: `http://www.unesco.org/new/es/santiago/resources/single-publication/news/aportes_para_la_ensenanza_de_la_matematica/` [Último acceso 11 de Mayo de 2016].

[14] W3C. *W3CHTML Media Extensions Working Group*. 2007. URL: `https://www.w3.org/html/wg/` [Último acceso 18 de Mayo 2017].

[15] W3C. *Introducing HTML 3.2*. 2014. URL: `https://www.w3.org/MarkUp/Wilbur/` [Último acceso 24 Febrero 2014].

[16] E. Willigers, C. Lilley, D. Schulze, B. Brinza, D. Storey, and A. Bellamy-Royds. *Scalable Vector Graphics (SVG) 2*. W3C, 2018. URL: `https://www.w3.org/TR/2018/CR-SVG2-20181004/` [Último acceso 28 Mayo 2019].

APÉNDICES

Apéndice 1

Figuras en Function Plot.

Se extendió la funcionalidad de la biblioteca FunctionPlot para que mediante ésta, sea posible representar en una gráfica figuras geométricas. Estas figuras son: rectángulo, círculo, polígono y polilínea, a las cuales se suma otro elemento denominado *texto*, que fue implementado junto con las figuras y que es útil para insertar texto en una gráfica. Rectángulo, círculo y texto fueron implementados en un nuevo módulo denominado *Shape* y se crea una nueva propiedad, también denominada *shape*, útil para definir la figura que se desea representar. Polígono y polilínea se implementaron en el módulo *Polyline*, debiendo utilizarse la propiedad *points* para definir a estas figuras. Todas las propiedades utilizadas aquí, deben establecerse como un objeto del arreglo *data*, el cual es utilizado para definir cada una de las representaciones gráficas que se desea crear con la biblioteca.

Figura	Elemento en SVG
Rectángulo	`<rect>`
Círculo	`<circ>`
Texto	`<text>`
Linea	`<path>`
Polígono	`<path>`

Tabla 1.1: Elementos de SVG utilizados para representar cada figura.

Las figuras mencionadas fueron implementadas utilizando elementos de SVG y, al igual que el resto del objetos representados por esta biblioteca, se hizo uso de D3.js para la creación y trabajo con estos elementos. En la Tabla 1.1, se muestra el elemento en SVG utilizado para crear cada una de estas

figuras.

A continuación, se listan cada una de las figuras, las opciones disponibles para trabajar con ellas y se mostrarán ejemplos de cómo representar cada una con lo desarrollado en la biblioteca.

1.1. Rectángulo, Círculo y Texto

Estas tres figuras forman parte del módulo *Shape* y es necesario utilizar dos propiedades, una para indicar que se utilizará este módulo y otra para indicar qué tipo de figura se desea representar. Estas dos propiedades y sus posibles valores son:

```
graphType: "shape"
shapeType: "rect" | "circ" | "text"
```

Para definir los atributos de cada una de estas figuras se utiliza la propiedad *shape*, la cual es un objeto que contiene propiedades comunes, utilizadas por las tres figuras, y propiedades específicas a la figura que se esté representando.

Las propiedades comunes a las tres figuras son:

```
// Coordenada x del centro de la figura.
shape.x: (integer|float) *
// Coordenada y del centro del figura.
shape.y: (integer|float) *
// Color de relleno de la figura.
shape.fill: (hexa|rgb|color name)
// Color de linea de la figura.
shape.stroke: (hexa|rgb|color name)
// Valor de rotacion en grados y en sentido
// horario de la figura.
shape.rotation: (integer|float)
```

* Propiedad requerida que debe estar definida en el objeto *shape*.

Mientras que las propiedades específicas a cada figura son:

Rectángulo

```
// Ancho del rectangulo.
shape.w: (integer|float) *
// Alto del rectangulo.
shape.h: (integer|float) *
```

Círculo

```
// Radio del circulo.
shape.r : (integer|float) *
```

Texto

```
// Texto a mostrar.
shape.text : (integer|float) *
```

* Propiedad requerida que debe estar definida en el objeto *shape*

.

Estas son todas las propiedades disponibles para trabajar con las figuras del módulo *Shape*. En las Figuras 1.1, 1.2, 1.3, se muestran ejemplos de la representación de un rectángulo, un círculo y texto respectivamente y dónde se puede apreciar el uso de dichas propiedades.

```
data: [{
    shape: {
      w : 6.0,
      h : 3.0,
      x : 0,
      y : 0,
      fill :"red",
      stroke: "#7f8c8d",
      rotation : -45
    },
    graphType: "shape",
    shapeType: "rect"
}]
```

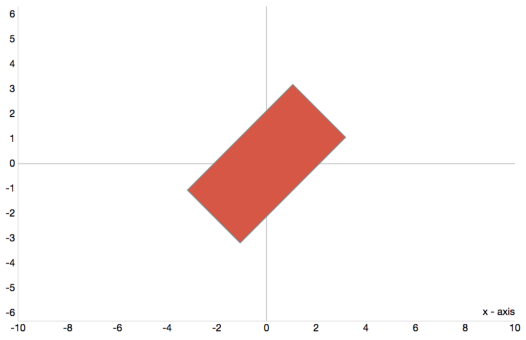


Figura 1.1: Representación de un rectángulo utilizando el módulo Shape.

```
data: [{
    shape: {
      r : 2.0,
      x : -2.0,
      y : 0,
      fill :"#2980b9",
      stroke: "#ecf0f1",
      rotation : 0.0
    },
    graphType: 'shape',
    shapeType: 'circle'
}]
```

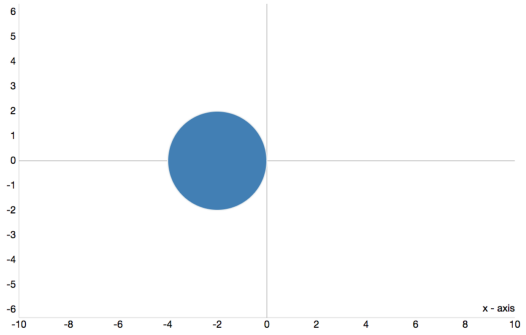


Figura 1.2: Representación de un círculo utilizando el módulo Shape.

```
data: [{
    shape: {
      text:"Here goes the text"
          ,
      size: 24,
      x : -3.5,
      y : 0,
      fill :"#27ae60",
      rotation : -90
    },
    graphType: 'shape',
    shapeType: 'text'
}]
```

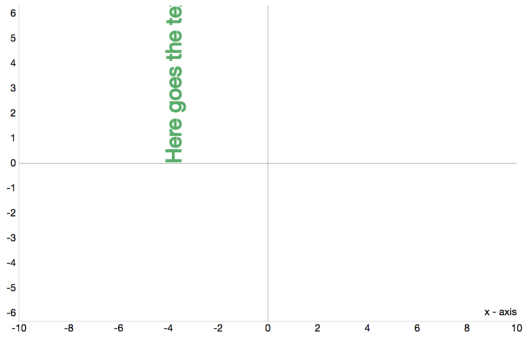


Figura 1.3: Representación de texto utilizando el módulo Shape.

1.2. Polilínea y Polígono

Estas figuras fueron agregadas al módulo *Polyline*. La razón de dicha acción surge del hecho que este modulo ya tiene implementado propiedades para representar puntos y líneas, siendo esto útil para la representación de polilíneas y polígonos.

Para definir el uso del módulo *Polyline*, la representación de puntos y el tipo de figura que se quiere representar con este módulo, se utilizan tres propiedades. Estas tres propiedades y sus posibles valores son:

```
graphType: "polyline"
fnType: "points"
polylineType: "line" | "polygon"
```

Tanto la polilínea como un polígono quedan definidos por un conjunto de puntos, donde los puntos luego serán unidos por segmentos de recta. Los mencionados puntos se definen en la propiedad *points* de este módulo, y la diferencia entre una figura u otra radica en que, en el caso del polígono, el último punto definido en esta propiedad será unido por un segmento de recta al primer punto, mientras que en la polilínea estos dos puntos no son unidos. La propiedad *points* es una lista de puntos donde cada uno de ellos (x,y) está representado por una lista de largo 2 que representa a x e y, como se puede ver a continuación:

```
points: [
    [x0, y0],
    [x1, y1],
    ...
    [xn, yn]
]
```

Además, al igual que las figuras del módulo *Shape*, se utilizan propiedades para definir la rotación de la figura, color de línea, etc, siendo estas las siguientes:

```
// Recuadro delimitador de la figura.
boundingBox: (true, false)
// Color de relleno de la figura.
fill: (hexa|rgb|color name)
// Color de linea de la figura.
stroke: (hexa|rgb|color name)
// Valor de rotacion en grados y en sentido
// horario de la figura.
rotation: (integer|float)
```

En las Figuras 1.4y 1.5 se muestran ejemplos de la representación de una polilínea y un polígono respectivamente y se puede apreciar de que forma son utilizadas estas propiedades. El ejemplo utilizado para representar un polígono, el punto (-8,3) es unido por un segmento de recta al punto (-6, 6).

Además, cabe destacar que el *Bounding Box* de una figura es utilizado aquí para calcular el centro de rotación de la figura y la propiedad *boundingBox*, es útil para mostrar u ocultar este recuadro; por defecto, esta propiedad tiene

el valor de "false". En suma, en el caso de polilínea, la propiedad *stroke* no es relevante, utilizándose únicamente la propiedad *fill* para establecer el color de la polilínea.

```
data: [{
    points: [
      [-10,-2],
      [0,-4],
      [2,6],
      [15,-10]
    ],
    fill :"orange",
    rotation : -90,
    fnType: 'points',
    polylineType : "line",
    graphType: 'polyline'
}]
```

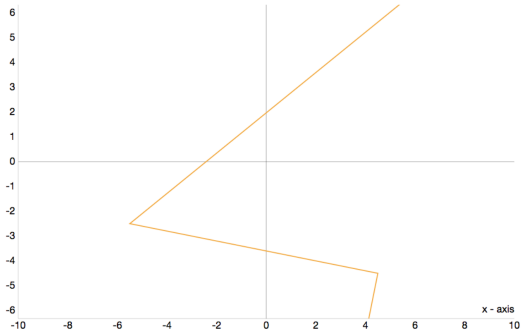


Figura 1.4: Representación de una polilínea utilizando el módulo Polyline.

```
data: [{
    points: [
      [-8,3],
      [-6,1],
      [-4,2],
      [-6,6]
    ],
    fill :"white",
    stroke: "green",
    rotation : 0,
    boundingBox: true,
    fnType: 'points',
    polylineType : "polygon",
    graphType: 'polyline',
}]
```

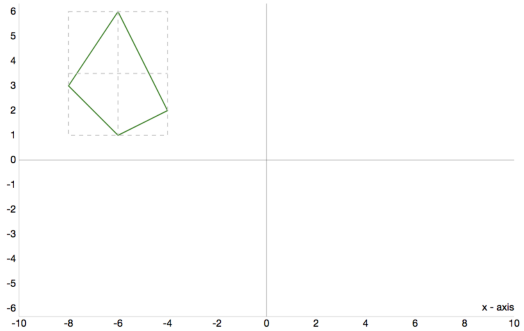


Figura 1.5: Representación de una polígono utilizando el módulo Polyline.

Apéndice 2

Algoritmos para la detección de discontinuidades.

A continuación se detallan los algoritmos utilizados para representar correctamente gráficas en 2D, de funciones que presenta discontinuidades en el rango en que se visualizarán dichas gráficas.

2.1. Detección de asíntotas

Este algoritmo fue desarrollado por Mauricio Poppe, autor de la biblioteca FunctionPlot, para detectar y renderizar correctamente las funciones que presentan asíntotas en el rango en que se visualiza la gráfica. En FunctionPlot, la representación gráfica de una función es creada a partir de dividir el rango en un número finito de intervalos de igual tamaño, evaluar la función en los puntos extremos de cada intervalo y analizar el comportamiento de la función en cada uno de éstos. El algoritmo de detección de asíntotas consta de dos etapas. Una primera etapa, en la cual se toma cada intervalo y se analizan las evaluaciones realizadas en los puntos extremos del intervalo. De cumplirse ciertas condiciones en estos puntos, se ejecuta una segunda etapa, la cual analiza con mayor profundidad el intervalo para detectar si la función presenta o no una asíntota.

La primer etapa del algoritmo utiliza la diferencia entre las dos evaluaciones de la función en ese intervalo, y el cambio de signo de la función entre ese intervalo y el intervalo anterior. Si la diferencia entre las evaluaciones es mayor que un Δ (grande), y la función cambia de signo, se pasa a la segunda etapa.

En esta segunda etapa, se divide el intervalo en subintervalos de igual tamaño, se evalúa la función en los extremos de cada uno de éstos y se vuelve a analizar la función en cada uno de ellos. De cumplirse cierta condición, se considera que la función presenta asíntota en el intervalo y las evaluaciones realizadas en cada extremo del intervalo no son unidas en la gráfica de la función. En las Figuras 2.1 y 2.2, se detallan la primera y segunda etapa del algoritmo respectivamente.

Figura 2.1: Primer etapa del algoritmo. Análisis de la función en los puntos extremos de cada intervalo.

1. Sea f la función a representar. Para cada punto x_n seleccionado para evaluar f, se toma el siguiente punto seleccionado x_{n+1}, y se calcula la diferencia $d_{n+1} = f(x_{n+1})$ - $f(x_n)$.
2. Sea $sig(d_n)$, signo de d_n. Si $sig(d_{n+1}) <> sig(d_n)$ y $d_{n+1} > \delta$ para un cierto δ, entonces:
   a) Se divide el entorno $[x_n, x_{n+1}]$ en 10 puntos y se analiza en mas detalle f en ese entorno. Para esto, se utiliza el algoritmo definido en la Figura 2.2.

Figura 2.2: Segunda etapa del algoritmo. Análisis de la función en subintervalos del intervalo seleccionado en la etapa 1.

Sea $y_1..y_{10}$ los puntos elegidos en el entorno $[x_n, x_{n+1}]$ entonces:

1. Si este proceso se ha realizado 3 veces entonces: Se considera que la función f es discontinua en el entorno $[x_n, x_{n+1}]$
2. Sino:
   a) Para cada punto y_m, con m en 1..10: Si signo de $f(y_{m+1})$ - $f(y_m)$ es igual a signo de d_{n+1} entonces Se repite este proceso, dividiendo el entorno $[y_m, y_{m+1}]$ en 10 puntos.
   b) Si signo $f(y_{m+1})$ - $f(y_m)$ es distinto a d_{n+1} para todo m entonces: Se considera que la función es continua en el entorno $[x_n, x_{n+1}]$.

2.2. Discontinuidad en funciones por partes

En funciones definidas por partes, es probable que se generen discontinuidades en los puntos extremos de cada intervalo de definición, pudiendo ser además, discontinuidades diferentes a la asintótica como son las discontinuidades evitables o de salto finito (7). FunctionPlot, fue creando únicamente con el algoritmo de detección de asíntotas. Para el resto de las discontinuidades que pueden presentarse aquí, FunctionPlot representa la función en ese punto como continua.

Para representar correctamente estos tipos de discontinuidades, se decidió crear un algoritmo que tome la función original y la divida en varias funciones diferentes, una por cada parte de la función original. Luego, cada una de dichas funciones puede ser representada de forma independiente una de otra utilizando FunctionPlot. En la Figura 2.3 se muestra el problema que se generaba al querer representar una función por partes y en la Figura 2.4 se muestra el mismo ejemplo aplicando este algoritmo y dividiendo la función en dos funciones diferentes. Al aplicar el algoritmo, la función en los puntos -2 y 2 no fue unida por una recta y se logró representar correctamente las discontinuidades que presenta la función original en esos puntos.

```
data: [{
  fn: function(x) {
    if (x >= -2 && x <= 2) {
      return x - 4;
    } else {
      return sin(x);
    }
  },
  graphType: 'polyline'
}]
```

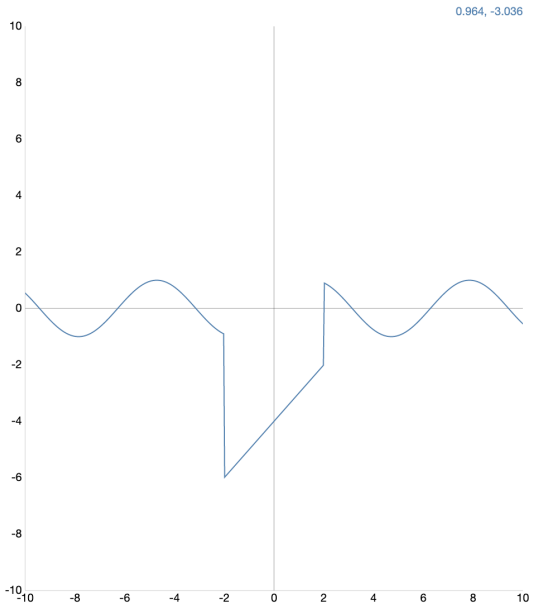


Figura 2.3: Función por partes sin aplicar el algoritmo de discontinuidad.

```
data: [{
  fn: function (x) {
    if (x >= -2 && x <= 2) {
      return x - 4;
    }
  },
  graphType: 'polyline'
},{
  fn: function (x) {
    if (!(x >= -2 && x <= 2)) {
      return sin(x);
    }
  },
  graphType: 'polyline'
}]
```

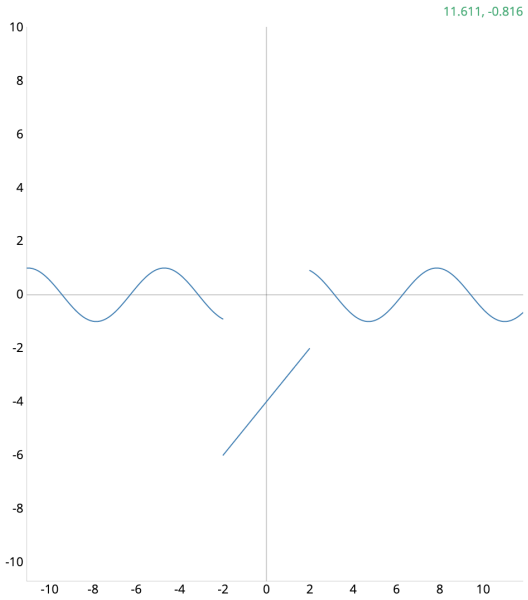


Figura 2.4: Función por partes aplicando el algoritmo de discontinuidad.

ANEXOS

Anexo 1

Prototipos Implementados

1.1. Prototipo 3D

Para facilitar el desarrollo de la biblioteca Graph3D se implementó un prototipo que interactúe con ésta, sin la necesidad de utilizar MateFun, es decir que sea totalmente independiente a MateFun.

Una de las motivaciones que llevó a la realización del mismo fue facilitar el desarrollo y pruebas de la biblioteca Graph3D, ya que al trabajar por fuera de MateFun, nos permite definir el modelo de datos de entrada a la biblioteca que deseamos, sin la necesidad de que los mismos sean generados a través de MateFun. Otro punto importante a resaltar es que durante el desarrollo de la misma, no se contaba con el soporte de MateFun para definir conjuntos de datos para 3D, por lo que el prototipo nos permitió avanzar con el desarrollo de Graph3D sin la necesidad de contar con ese requerimiento.

El prototipo desarrollado es un sitio en HTML5 y JavaScript, que tiene entre sus dependencias a la biblioteca Graph3D. El sitio está compuesto por dos paneles principales, un panel izquierdo en el cual se muestra un listado de diferentes ejemplos de los tipos de funcionalidades de Graph3D, y un panel derecho en el cual se despliega el resultado para dicha funcionalidad.

A continuación se muestra una captura del sitio:

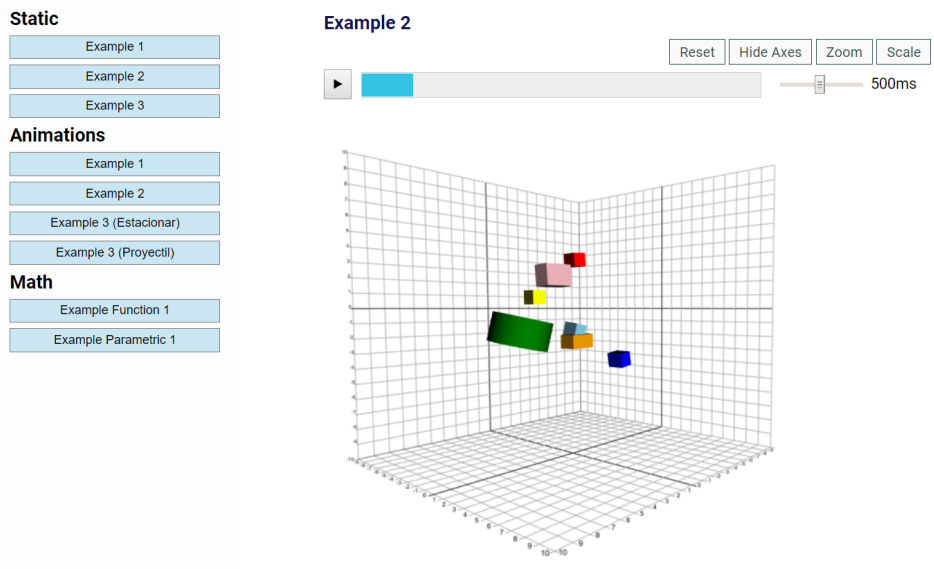


Figura 1.1: Captura del prototipo

Como se mencionó en la sección 4.4 (Graph3D), Graph3D cuenta con funcionalidades como la representación de Figuras 2D en tres dimensiones y la renderización de funciones matemáticas, pero que no son contempladas por el intérprete MateFun.

Si bien a través de MateFun no se pueden ver dichas funcionalidades, en el prototipo si se pueden visualizar diferentes ejemplos.

Anexo 2

Interfaz para la interacción con la biblioteca Graph3D

En este apartado se detalla la interfaz para la interacción con la librería desarrollada, y se listan los diferentes métodos, añadiendo una breve descripción de la funcionalidad de cada uno.

`drawFigures(figures: array<figure>)`
Permite dibujar una lista de figuras, las cuales vienen como parámetro.

`initializeAnimation(frames: array<array<figure>>, callback: ref)`
Carga los datos para visualizar una animación en 3D.

`playAnimation()`
Pone en ejecución la animación.

`changeSpeedAnimation(delay: int)`
Cambia la velocidad con la cual la animación se ejecuta. El tiempo delay pasado como parámetro significa el tiempo que debe transcurrir para alternar cada frame.

`clear()`
Limpia todo el contenido que hay en el renderizador.

`changeZoomType(type: ZoomType)`
Establece el tipo de zoom que se realiza al hacer scroll sobre el renderizador. Los tipos de zoom que soporta son:

All El zoom es aplicado sobre todos los ejes por igual.

XAxis El zoom es aplicado solamente en el eje x.

YAxis El zoom es aplicado solamente en el eje y.

ZAxis El zoom es aplicado solamente en el eje z.

`changeZoom(increase: bool)`

Cambia el zoom del visualizador, aumentando el mismo o disminuyéndolo dependiendo del parámetro increase.

`changeOptions(quality: int)`

Establece el nivel de calidad de los objetos 3D representados, a menor valor implica una menor calidad y a mayor valor una mayor calidad. Aumentar demasiado la calidad trae consigo demoras en el renderizado de los objetos.

`changeAxesSize(axes)`

Permite alterar la escala de cada uno de los 3 ejes (x, y, z).

`reset()`

Restablece la configuración del visualizador, poniendo como opciones a las por defecto.