UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

Entorno de cálculo sobre complejos simpliciales

Autor/es

Pablo Ascorbe Fernández

Director/es

LAUREANO LAMBAN PARDO

Facultad

Facultad de Ciencia y Tecnología

Titulación

Grado en Ingeniería Informática

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2021-22

publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

UNIVERSIDAD DE LA RIOJA

Facultad de Ciencia y Tecnología

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Entrono de cálculo sobre complejos simpliciales

Realizado por: Pablo Ascorbe Fernández

Tutelado por: Laureano Lamban Pardo

Logroño, Julio, 2022

Resumen

El trabajo realizado es un entorno de cálculo sobre complejos simpliciales. Consta de un estudio matemático exhaustivo y su correspondiente implementación en Python.

Su objetivo es comprender y hacer más cercanos los conceptos teóricos para poder trabajar con ellos en una aplicación de escritorio sencilla e intuitiva. Permite realizar cálculos con complejos simpliciales y calcular campos de vectores sobre ellos. También permite definir funciones booleanas y establecer la equivalencia entre complejos y funciones booleanas.

El producto final logrado ha conseguido dicho objetivo. Además, ha sido diseñado para facilitar futuras extensiones y mejoras.

Abstract

The work done is a calculation environment on simplicial complexes. It consists of an exhaustive mathematical study and its corresponding implementation in Python.

Its objective is to understand and make the theoretical concepts closer in order to work with them in a simple and intuitive desktop application. It allows to perform computations with simplicial complexes and to calculate vector fields over them. Also, it allows to define Boolean functions and to establish the equivalence between complexes and Boolean functions.

The final product has achieved this goal. Furthermore, it has been designed to facilitate future extensions and improvements.

Índice

1. Fundamentación matemática

La idea de este Proyecto surgió de la lectura del libro [1], "Discrete Morse Theory", cuyo autor es N.A. Scoville. En realidad, el alcance del Proyecto atañe a una parte modesta de lo que en el libro se puede estudiar, pero se plantea como una primera aproximación a un entorno de cálculo topológico sobre complejos simpliciales.

Esta sección sirve como prolegómeno, previo a la introducción, para presentar conceptos y fundamentos matemáticos básicos, pero necesarios, para dar contexto a este trabajo. Particularmente, se desarrollarán tres: los complejos simpliciales, las funciones booleanas y los campos de vectores gradientes. No se trata de una descripción detallada del tema, solo se pretende hacer entendible, para personas no familiarizadas con la topología simplicial, el entorno de cálculo que se va a desarrollar.

Cabe destacar que lo que esta sección trata es una síntesis del contenido de los capítulos 1, 2 y 6 del libro [1].

1.1. Complejos simpliciales

Antes de introducir la noción de símplice y, con esta, la de complejo simplicial, se ha considerado interesante presentar la idea de convexidad.

Un subconjunto del espacio euclídeo $\mathbb{R}^n$ diremos que es un *conjunto convexo* sí y solo sí, para cualquier par de puntos de dicho conjunto, el segmento que los une está contenido en este. Por ejemplo, un círculo es un conjunto convexo del plano mientras que la circunferencia no lo es.

De esta definición surge una segunda, la de combinación convexa. Dados m puntos de $\mathbb{R}^n$, $x^1, x^2, \dots, x^m$, llamaremos *combinación convexa* de esos puntos, a cualquiera otro vector de $\mathbb{R}^n$, de la forma:

$$p_1 x^1 + \cdots + p_m x^m, donde\ los\ p_i \geq 0, \forall i = 1, \dots, m, \qquad y \sum_{i=1}^{m} p_i = 1$$

Y, terminando con la convexidad, damos la noción de clausura convexa:

Sea $A \subset \mathbb{R}^n$. Se denomina *clausura convexa* de A al menor convexo B de $\mathbb{R}^n$ tal que $B \supset A$.

En particular, si A es un conjunto finito de puntos, la clausura convexa de A coincide con el conjunto de todas sus combinaciones convexas.

Teniendo definido el concepto de convexidad pasamos a definir el de símplice, pudiendo enfocarlo desde dos variantes: la geométrica y la abstracta. Aunque nosotros trabajaremos con la abstracta durante todo el trabajo, nos apoyaremos en la geométrica para una mejor comprensión y clarificar los conceptos posteriores.

Un *símplice geométrico* τ es una clausura convexa de un conjunto finito de puntos linealmente independientes incluidos en $\mathbb{R}^n$. Si $v_0, v_1, \dots, v_k$ son los puntos que determinan el símplice τ, identificaremos τ con el conjunto $\{v_0, v_1, \dots, v_k\}$. Diremos que τ es de *dimensión* k y lo llamaremos *k-símplice*. En la ilustración 1 vemos un ejemplo de símplices geométricos de dimensiones 0, 1 y 2.

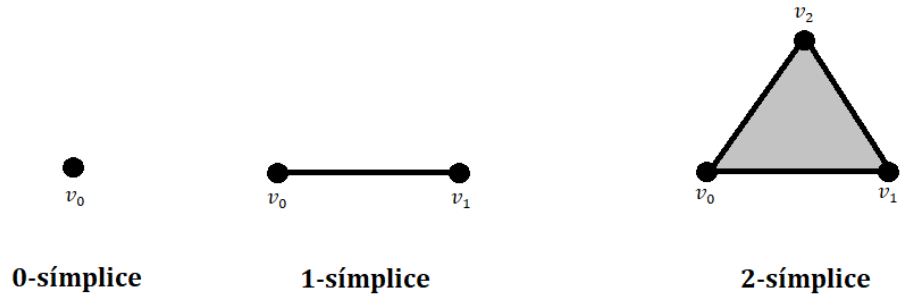


Ilustración 1: Símplice geométricos

Si τ y σ son dos símplices tal que σ está generado por un subconjunto de los vértices generadores de τ, lo denotaremos $\sigma \leq \tau$.

Un *complejo simplicial geométrico* K es un conjunto de símplices geométricos de modo que si $\tau \in K$ y $\sigma \leq \tau$ entonces $\sigma \in K$ y, además, la intersección de dos símplice de K es un símplice de K.

De aquí surge la idea de *cara* de un símplice. Una *cara* de un símplice es el conjunto convexo generado por un subconjunto de sus vértices. Si $\sigma \leq \tau$ entonces σ es cara de τ. Según la definición, todas las caras de un símplice de un complejo simplicial también son símplices.

Habiendo introducido la noción geométrica, podemos definir los complejos simpliciales abstractos.

Llamamos *complejo simplicial abstracto* a una tupla (Ω, K), con Ω un conjunto de n+1 símbolos, y K una colección de subconjuntos de Ω, incluyendo ∅, de modo que:

a) Si $\tau \in K$ y $\sigma \subseteq \tau$ entonces $\sigma \in K$;
b) $\{v_i\} \in K$ para todo $v_i \in \Omega$.

El conjunto Ω es llamado conjunto de puntos o conjunto de vértices y sus elementos son los puntos, vértices o 0-símplices.

No siempre se exige la segunda condición, con lo que los 0-símplices de (Ω, K) son solo un subconjunto de Ω. En nuestro caso, exigimos la condición, lo que nos permite identificar Ω con el conjunto de vértices (0-símplices).

Cabe destacar que la definición abstracta de complejo simplicial tiene un enfoque más combinatorio y por ello, es con la que trabajaremos. A partir de ahora llamaremos complejos simpliciales a los complejos simpliciales abstractos.

Sea K un complejo simplicial. La *dimensión de* K es el máximo de las dimensiones de todos su símplices.

El *c-vector* de un complejo simplicial K es el vector definido por $c_k := (c_0, c_1, \dots, c_{dim(K)})$, donde c_i es el número de símplices de dimensión i, $0 \leq i \leq dim(K)$.

Si $\tau, \sigma \in K$ con $\sigma \subseteq \tau$, se dice que σ es un *cara* de τ y que τ es una *cocara* de σ.

Definimos el concepto de *facet.* Se llama facets a los símplices de K que no están propiamente contenidos en ningún otro símplice de K. Es decir, un facet es un símplice que es maximal respecto del contenido (maximal en el orden parcial que define el contenido de subconjuntos de Ω).

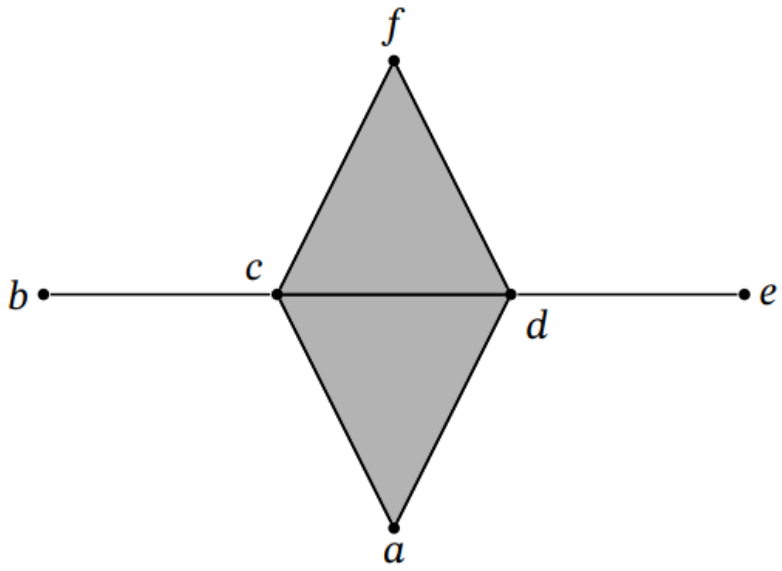


Ilustración 2: Ejemplo de complejo simplicial

Como ejemplo, en la ilustración 2, vemos un complejo simplicial con 6 vértices: a, b, c, d, e y f. Sus aristas, 2-símplices, serían: {a, c}, {a, d}, {c, d}, {c, f}, {d, f}, {b, c}, {d, e}. Sus 3-símplices serían: {a, c, d}, {c, f, d}.

Por último, los facets de este complejo son los símplices: {a, c, d}, {c, d, f}, {b, c} y {d, e}.

El número de caras de un símplice queda bien determinado mediante el triángulo de Pascal. En la ilustración 3 podemos verlo: la columna de la izquierda denota al propio símplice; la siguiente, el número de caras de dimensión estrictamente inferior; la tercera, el número de caras con una dimensión dos veces menor y así sucesivamente hasta la columna de la derecha que contiene el conjunto vacío.

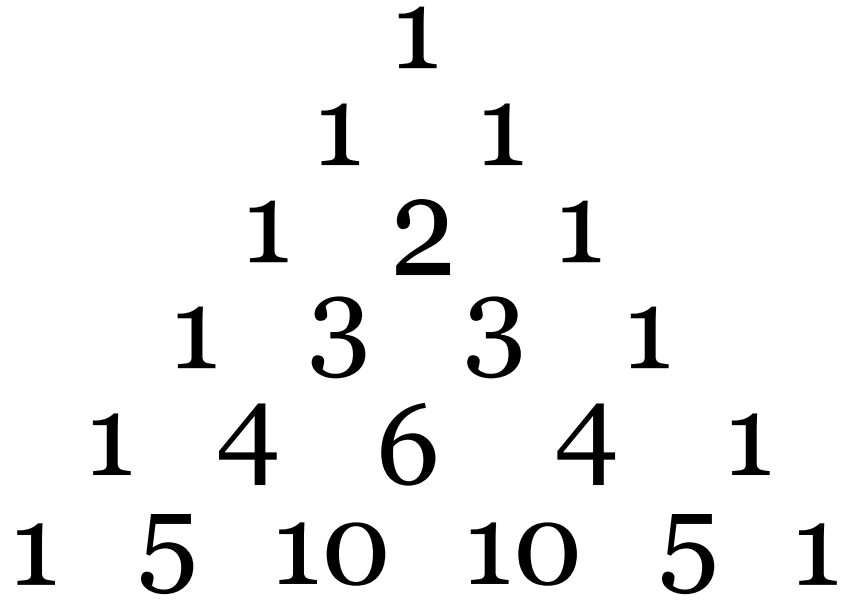


Ilustración 3: Triángulo de Pascal

La *característica de Euler* de un complejo simplicial K viene definida por la expresión: $\sum_{i=0}^{n}(-1)^i c_i(K)$. Es decir, la suma de las componentes del c-vector de K donde las impares son cambiadas de signo.

Se trata de un invariante de tipo topológico, es decir, se conserva por homeomorfismo (isomorfismo entre espacios topológicos) y también se preserva por operaciones como las que se definen a continuación.

Sea K un complejo simplicial y suponemos un par de símplices $\sigma, \tau \in K$ tales que σ es cara de τ y σ no tiene otras cocaras en K. Entonces $K - \{\sigma, \tau\}$ es el complejo simplicial resultante de eliminar en K el par de símplices σ *y* τ. Esta operación se denomina *colapso elemental* de K.

Por otro lado, suponemos una pareja de subconjuntos de Ω, σ *y* τ, que no están en K, σ *es cara de* τ y el resto de caras de τ están en K. Entonces $K \cup \{\sigma, \tau\}$ es el complejo simplicial resultante de añadir a Kese par de símplices. Esta operación se denomina *expansión elemental* de K.

Pues bien, resulta sencillo probar que la característica de Euler se conserva tanto mediante colapsos como por expansiones. No obstante, se trata de un invariante muy débil, en el sentido que espacios "topológicamente muy diferentes" tienen la misma característica de Euler. Así, su uso más frecuente es demostrar la no equivalencia entre espacios.

Por su parte, una invariante más fuerte que la característica de Euler es el tipo de homotopía simple. Dos complejos tienen el mismo tipo de homotopía simple si se puede pasar del primero al segundo mediante colapsos y expansiones.

Ahora definiremos dos conceptos simpliciales importantes: link y star de un vértice. Antes es interesante señalar que hay dos definiciones del star, el star cerrado y el abierto. Nuestro libro de referencia [1] define el star cerrado. La diferencia entre el abierto y el cerrado radica en que este último es un complejo simplicial en sí mismo, mientras que el abierto no.

Sea K un complejo simplicial y $v \in K$ un vértice.

El *star de* v *en* K es el complejo simplicial inducido por el conjunto de todos los símplices de K que contienen a v.

El *link de* v *en* K es el conjunto que resulta de eliminar del star v el conjunto $\{\sigma \in K : v \in \sigma\}$. Es decir, al conjunto star de K restarle el conjunto star abierto.

Así, $link(v) = \{\sigma \in K;\ v \notin \sigma \text{ y } \sigma \cup \{v\} \in K\}$

Para comprenderlo mejor en las ilustraciones 4 y 5 figuran dos ejemplos de star y link respectivamente:

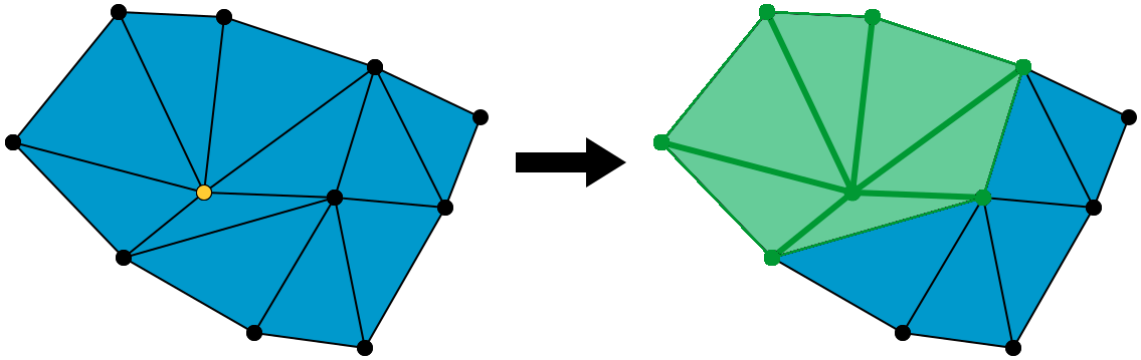


Ilustración 4: Un vértice y su star [2]

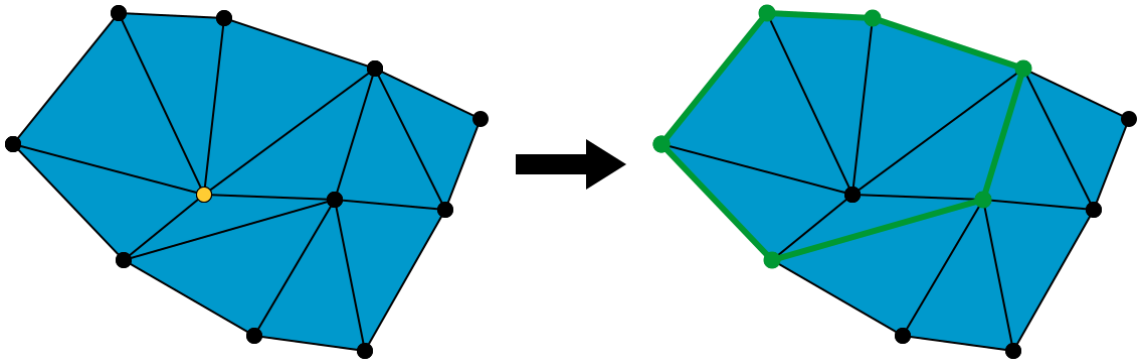


Ilustración 5: Un vértice y su link [2]

En ocasiones, cálculos de invariantes topológicos requieren de la construcción del link y del star de vértices.

Sean K y L dos complejos simpliciales sin ningún vértice en común. Definimos el *join* de K y L, escrito como $K * L$, como el complejo simplicial dado por:

$$K * L := \{\sigma, \tau, \sigma \cup \tau : \sigma \in K, \tau \in L\}.$$

Un *cono* de K, es un join de K en el que $L = \{v\}$, siendo $v \in \Omega$ un vértice que no está en K.

1.2. Campos de vectores gradiente

Este apartado trata de definir la noción de campo de vectores gradiente sobre un complejo simplicial. En realidad, el concepto se enmarca en la conocida como Teoría Discreta de Morse (ver el capítulo correspondiente en [1]). La noción de partida es la de función discreta de Morse. No obstante, en este trabajo no profundizaremos en ella, así que definiremos directamente los campos de vectores gradientes definidas por funciones de Morse.

Un *campo de vectores gradiente*, que denotaremos por V, es un conjunto de tuplas (σ, τ), donde σ es un p-símplice y τ un p+1-símplice, satisfaciendo una serie de condiciones que impondremos más adelante.

Un par $(\sigma, \tau) \in V$, es llamado vector, donde a σ lo llamaremos símplice source (fuente) y a τ target (objetivo).

Los vectores indican una dirección. La idea es que, si marcamos una tupla o pareja de símplices (σ, τ) como vector, el símplice σ podrá recorrer τ hasta llegar al resto de sus caras. Esto nos genera una forma de ver el campo de vectores como una serie de caminos o rutas entre símplices. Intuitivamente puede observarse que, cuando añadimos un vector a un campo, permite que desde un símplice se puedan alcanzar nuevos símplices de su misma dimensión.

La formalización correcta de esta idea llevaría a la noción de *V-camino.*

Cuando un símplice σ se encuentra a sí mismo entre sus símplices alcanzables, decimos que se ha formado un ciclo, es decir, un *V-camino cerrado*.

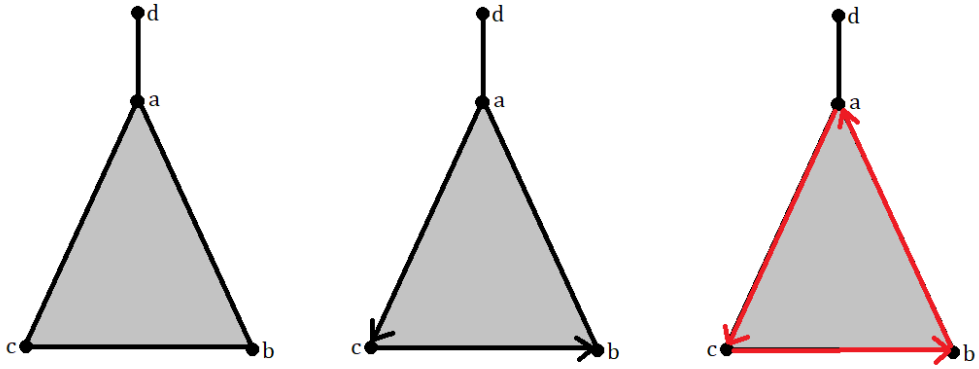


Ilustración 6: Ejemplo de ciclo campo de vectores

En la Ilustración 6 podemos ver un ejemplo de ciclo, inicialmente hay dos vectores (a, ac) y (c, cb), es decir, a puede alcanzar a c y, como c puede alcanzar a b, por transitividad, a puede alcanzar a b. Si ahora se añade como vector (b, ab), entonces a pasa a poder alcanzarse a sí mismo formando un ciclo.

Simplificando algo la terminología, un campo de vectores gradiente debe cumplir las siguientes dos propiedades:

a) Si un símplice pertenece a un vector no puede ser ni source ni target de ningún otro vector del campo.
b) Ningún símplice debe alcanzarse a sí mismo. Es decir, no puede haber ciclos.

Desde el punto de vista del cálculo topológico, la importancia radica en que considerar un campo gradiente sobre un complejo simplicial K, permite obtener un espacio con menor complejidad que K, donde los cálculos requieren menor coste, y conservando invariantes topológicos, como, por ejemplo, los grupos de homología. Por ello, los campos permiten obtener reducciones de los espacios donde resulta más sencilla la computación.

Un buen marco en el que representar un campo de vectores sobre un complejo simplicial K es el diagrama de Hasse asociado a K, es decir, al orden parcial que la relación de contenido define sobre los símplices de K.

Sea K un complejo simplicial, el *diagrama de Hasse* de K, denotado por $\mathcal{H}_{\mathrm{K}}$, es el grafo dirigido definido como el poset de símplices de K ordenado por la relación de cara. Es decir, $\mathcal{H}_{\mathrm{K}}$ puede entenderse como un complejo simplicial de dimensión uno, tal que existe una correspondencia uno a uno entre sus nodos y los símplices de K.

Habiendo definido $\mathcal{H}$, un campo de vectores V sobre K define un nuevo grafo, denotado por $\mathcal{H}_V$, que consiste en invertir en el diagrama de Hasse las aristas que coinciden con los vectores de V. Es decir, una flecha apunta hacia arriba sí y solo sí los dos nodos de la arista son un vector en V (todas las demás apuntan hacia abajo).

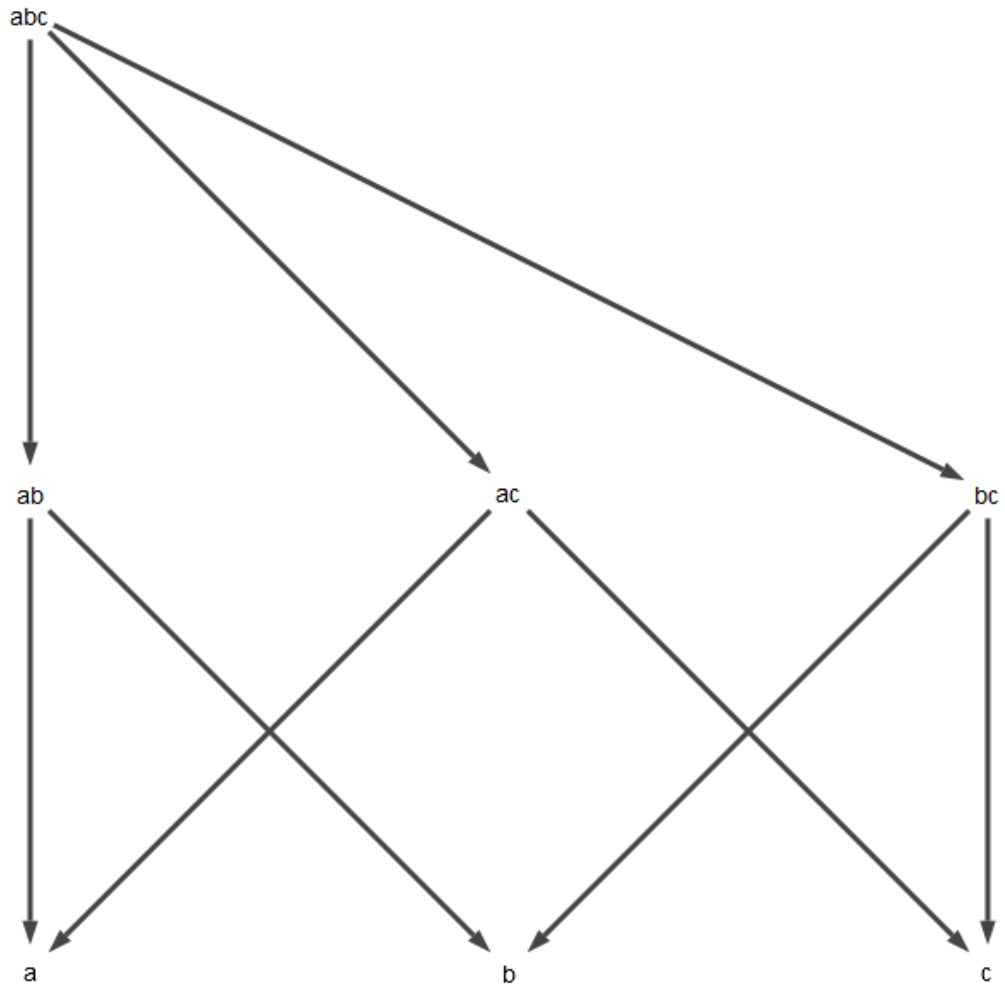


Ilustración 7: Diagrama de Hasse de un complejo simplicial con un 2-símplice

1.3. Funciones booleanas

En este sección vamos a introducir el concepto de función booleana. Así como los complejos simpliciales están determinados por sus vértices, cuáles y cuantos son, de igual manera las funciones booleanas quedan determinadas por sus variables.

Una primera simplificación que vamos a realizar es no utilizar identificadores para las variables, solo nos va a interesar su número. Como es habitual, identificamos 0 con el valor booleano false y 1 con el valor true. De este modo, podemos pensar que una función booleana con n variables tiene la forma: $f: \{0,1\}^n \to \{0,1\}$. Así, para una combinación de las posibles entradas de la función (para sus n variables) obtendremos como salida 0 o 1.

Fijada una ordenación de las variables, la correspondiente tabla de verdad de una función booleana tiene 2^n filas, por lo que podemos identificar una función booleana con la lista (ordenada) de sus 2^n resultados. Es decir, una lista con tamaño 2^n y formada por 0s y 1s.

Antes de introducir el concepto de monotonía, es necesario aclarar que en el libro [1], para establecer una relación entre funciones booleanas y complejos simpliciales, se usa una definición de monotonía creciente. Nosotros usaremos en este trabajo la decreciente, dándole más peso a los 1's que a los 0's.

Sea $f: \{0,1\}^n \to \{0,1\}$ una función booleana. Entonces diremos que f es *monótona* si para cualquier $\vec{x} \leq \vec{y}$, tenemos $f(\vec{x}) \geq f(\vec{y})$. Entendemos que el orden en $\{0,1\}^n$ es el lexicográfico. Por ejemplo, si tenemos dos variables x_0 y x_1 y para sus valores de entrada tenemos $f(1,1) \to 1$, entonces para un valor menor que $(1,1)$, como por ejemplo $(0,1)$, su salida deberá ser obligatoriamente 1. Si no, no será monótona.

Sea f una función booleana monótona. El complejo simplicial K_f inducido por f tiene n-vértices $\{x_1, \dots, x_n\}$ y uno de sus subconjuntos $\{x_{K1}, \dots, x_{Kr}\}$ de lugar a un símplice de dimensión r-1, sí y solo sí la tupla $\vec{x}$ que solo tiene 1's en las posiciones $(k_1, \dots, k_r)$ y 0's en el resto, cumple $f(\vec{x}) = 1$.[1]

Adoptaremos la siguiente notación, si $x_0 = \cdots = x_k = 1$ siendo las k coordenadas de una entrada, entonces el símplice correspondiente en K_f será $\sigma_{\vec{x}} \coloneqq \{x_0, \dots, x_k\}$. Entonces, si solo una variable de entrada vale 1, $\sigma_{\vec{x}}$ estará representando un vértice.

Como ejemplo, en la ilustración 8, la siguiente combinación de valores de salida corresponde con el complejo simplicial de la derecha:

x_0	x_1	Salidas
0	0	1
0	1	1
1	0	1
1	1	1

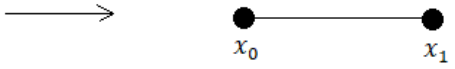


Ilustración 8: Transformación de una función booleana a un complejo simplicial.

2. Introducción

Como ya se ha indicado, el producto final (visible) que se pretende alcanzar con este proyecto es el desarrollo de un entorno amigable para la realización de cálculo sobre complejos simpliciales. No obstante, conviene aclarar que una primera necesidad, que ha requerido un esfuerzo considerable, ha sido el estudio y comprensión exhaustivos de los complejos simpliciales y sus operadores. Así, no hay que olvidar que parte importante del tiempo dedicado y del esfuerzo

[1] Notad que si $f \neq$FALSE entonces $\emptyset \in K_f$, por lo que K_f será no vacío.

requerido ha sido invertido en comprender los conceptos explicados en el capítulo previo.

El objeto de este trabajo, debido a las limitaciones de tiempo de desarrollo y de los conocimientos del tema por mi parte, no es hacer un entorno de cálculo matemático complejo buscando el mayor rigor posible y una eficiencia computacional excelsa. Se trata más bien de programar dichos conceptos matemáticos de una forma sencilla y comprensible, buscando el desarrollo de una aplicación que facilite el aprendizaje y la comprensión de los complejos simpliciales.

2.1. Funcionalidad

La funcionalidad de la aplicación se centra en el trabajo con los complejos simpliciales, las transformaciones entre complejos y funciones booleanas y el cálculo de operadores. Además, de la persistencia y una correcta presentación de los datos.

Con la aplicación se podrá crear, modificar, eliminar y trabajar con funciones booleanas y complejos simpliciales. Como los complejos simpliciales son el objetivo principal de este proyecto casi todos los operadores y cálculos se realizarán sobre ellos. Siempre buscando un enfoque académico, tratando de motivar el estudio y entendimiento de estos magníficos temas a través de la aplicación.

2.2. Comparativa

En este capítulo comentaremos algunas herramientas, librerías o aplicaciones que tengan un propósito similar a la desarrollada en este trabajo.

Además de algunos entornos específicos dedicados al cálculo simbólico en Topología Algebraica, pensadas para las necesidades de

especialistas en el tema, como única librería que trabaja con complejos simpliciales encontramos “simplicial”, desarrollada por Simon Dobson [3]. Con ella se pueden instanciar complejos simpliciales, agregar y eliminar sus símplices, además de calcular una serie de propiedades y operadores. Es una librería profesional y gran parte de su contenido queda fuera del estudio y propósito de la aplicación.

No trabaja con funciones booleanas, ni campos de vectores y hay ciertos conceptos sobre complejos simpliciales que no abarca. Por tanto, aunque para crear, modificar y calcular propiedades de complejos simpliciales sea realmente útil, no contempla gran parte de nuestro alcance, ya que el objetivo de este proyecto era poder traducir la teoría matemática en código, que pudiese ser interpretado y utilizado por usuarios que desearan manipular complejos simpliciales.

2.3. Tecnologías

Inicialmente se deliberó sobre qué lenguaje de programación utilizar, siendo las opciones Python o Java y escogiendo finalmente Python. La elección se debió principalmente a su gran cantidad de librerías útiles para gran parte de nuestro cometido. Sobre todo, dando peso a librerías que permiten crear grafos para los diagramas de Hasse.

Además, se valoró su elegante forma de trabajar con listas y diccionarios, que serán frecuentemente utilizados a los largo del trabajo. Aunque no sea un lenguaje expresamente pensado para la orientación a objetos puede trabajar perfectamente con estos, ya que el modelo de dominio cuenta con clases que modelan los conceptos matemáticos: símplices, complejos simpliciales, funciones booleanas y campos de vectores.

Como IDE (Integrated Development Environment o Entorno de Desarrollo Integrado) se ha escogido a PyCharm. Aunque se podría haber escogido un editor de texto para escribir el código, se ha optado

por un IDE ya que integra también depurador, una integración con un sistema de control de versiones y un intérprete, entre otros.

PyCharm ha sido el IDE seleccionado por: su facilidad de instalación (no solo en Windows si no también en Linux, que son los dos sistemas operativos con los que se ha trabajado), la gran cantidad de plugins y la facilidad de instalación de las librerías necesarias, su sencillez para escribir código y refactorizarlo y, finalmente, por tratarse de una versión gratuita.

Como sistema de control de versiones se ha usado Git, empleando una integración entre GitHub y PyCharm, creando un repositorio para la aplicación donde se han ido subiendo las distintas versiones tras cada iteración.

Para el apartado de persistencia, como no se considera que vaya a ser necesario diseñar e implementar una base de datos SQL, ya que la cantidad de entidades y sus relaciones parece que será limitada, se utilizará JSON. Se emplea este formato de texto debido a que la integración con Python es sencilla, para serializar y deserializar los datos Python cuenta con una librería de manera nativa.

El sistema operativo que se ha empleado para el desarrollo ha sido principalmente Windows, pero también se ha utilizado un sistema Linux.

Como herramienta de modelado y creación de diagramas se ha usado Modelio, ya que los diagramas suponen una parte importante del análisis y diseño, se ha considerado relevante indicar aquí la herramienta.

Por último, para la capa de presentación se ha instalado una librería llamada PyQt5, que es una librería que permite crear interfaces gráficas en Python. Utilizando su interfaz de diseño Qt Designer para el

apartado visual. De nuevo, ha pesado en su elección que es gratuita y permite desarrollo multiplataforma.

2.4. Metodología de desarrollo

Cabe recordar que gran parte del esfuerzo inicial de este proyecto ha sido en torno al estudio y comprensión de los conceptos matemáticos. Es por ello por lo que el ciclo de vida de la aplicación informática no ha sido realmente ambicioso. Tratándose de una aplicación algo volátil, pautar distintas iteraciones no era algo trivial, pero tampoco se ha seguido una metodología tradicional.

Se ha intentado adaptar a una metodología ágil, sin seguir ninguna técnica concreta. Sí que se ha realizado un planteamiento inicial del proyecto, y un análisis continuo de los distintos requisitos que se han ido introduciendo en la aplicación. Estos fueron bastante variables al comienzo, ya que no se tenía del todo claro qué operadores y conceptos serían introducidos finalmente, únicamente teniendo claro que estaría muy centrado en los complejos simpliciales.

La idea ha sido realizar diseño, desarrollo y pruebas unitarias por cada operador o concepto que se introducía en la aplicación. Es imposible decir con precisión cuantas iteraciones se han hecho realmente, ya que ha habido algunas más largas e importantes que otras. Pero generalmente, cada vez que se añadía un requisito a la aplicación se hacía su diseño, se programaba y se realizaban las correspondientes pruebas.

Tras ello, el procedimiento proseguía con la discusión del proceso llevado a cabo con el tutor, recibiendo mejoras o sugerencias para tener en cuenta a la hora de realizar la siguiente iteración. Bien es cierto que no se ha seguido un riguroso control sobre los distintos ciclos de desarrollo, no se han planificado tiempos ni puesto hitos para determinar cuando acababa cada uno. Pero de nuevo, salvando las distancias, el Burndown chart del proyecto nunca se ha visto

comprometido por ello, cada dos semanas, regularmente, se han ido fijando reuniones con el tutor a modo de final de dichos ciclos y comentando lo avanzado con el tutor.

En cuanto a la aplicación se ha seguido un diseño en tres capas: presentación, lógica de negocio y persistencia. Este diseño, evidentemente, ha condicionado la forma en la que sea han ido realizando las distintas iteraciones ya que primero los requisitos de la capa de negocio eran los más prioritarios, siguiendo con los de persistencia, para terminar con iteraciones sobre requisitos de presentación.

3. Gestión del proyecto

3.1. Alcance

Tareas

Las tareas de este proyecto, ya que ha sido algo difícil de adecuar a un proyecto estándar, se centran en la aplicación y su desarrollo, siendo divididas en cinco grandes grupos: modelo de dominio, lógica de negocio, persistencia, presentación y pruebas.

Tareas del proyecto	
Título	Modelo de dominio
Clase complejo simplicial	Modelará a los complejos simpliciales incluyendo sus métodos y operadores.
Clase símplice	Modelará a los símplices abstractos.
Clase función booleana	Modelará a las funciones booleanas con sus métodos y operadores.
Clase campo de vectores	Modelará los campos de vectores y sus operadores.
Métodos auxiliares	Métodos auxiliares para encapsular código necesario para las distintas clases del modelo de dominio.
Título	Lógica de negocio
Gestor complejo simplicial	Gestor que se encargará de las operaciones CRUD de los complejos simpliciales.

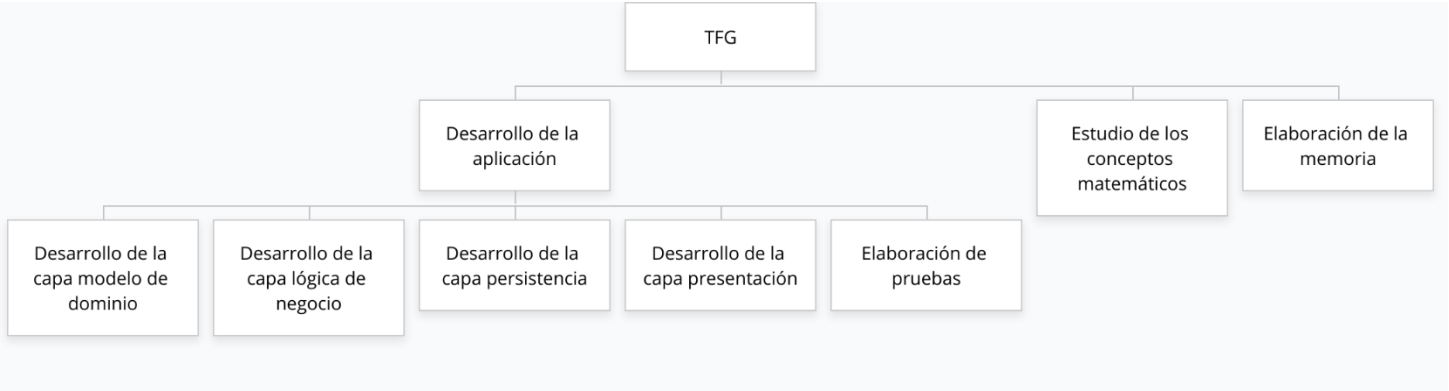
Tareas del proyecto	
Título	Lógica de negocio
Gestor función booleana	Gestor que se encargará de las operaciones CRUS de las funciones booleanas.
Transformadores	Métodos encargados de transformar complejos simpliciales a funciones booleanas y viceversa.
Generador de campos de vectores	Lógica encargada de añadir los vectores correspondientes a un campo de vectores de manera que no se formen ciclos.
Excepciones	Excepciones personalizadas para el contexto de la aplicación.
Título	Persistencia
Persistencia complejo simplicial	Métodos de persistencia de las operaciones CRUD para objetos que representan complejos simpliciales.
Persistencia funciones booleanas	Análogo a la persistencia de complejos simpliciales, pero para funciones booleanas.
JSON	Métodos para serializar y deserializar diccionarios en formato JSON en ficheros .json y viceversa.
Codificador y decodificador	Métodos para codificar y decodificar objetos Python en formato JSON y viceversa, para poder ser serializados y deserializados posteriormente.
Título	Presentación
Menú principal	Menú principal desde el que poder acceder a todas las funcionalidades de la aplicación.
Creación de complejos simpliciales	Formulario de creación de los complejos simpliciales.
Creación de funciones booleanas	Formulario de creación de funciones booleanas.
Lista complejos simpliciales	Lista de todos los complejos simpliciales almacenados, donde poder buscar, ordenar, cargar y eliminarlos.
Lista funciones booleanas	Lista análoga, pero para funciones booleanas.
Menú complejos simpliciales	Menú resultado de cargar un complejo simplicial, donde se incluyen todos sus operadores disponibles y desde el que poder acceder al resto de la aplicación
Menú funciones booleanas	Menú análogo al de complejos simpliciales, pero para funciones booleanas.
Título	Pruebas
Pruebas unitarias	Pruebas unitarias para los métodos más significativos de la aplicación, donde medir la eficacia, eficiencia y efectividad de estos.

Estructura de descomposición del trabajo

La EDT del Proyecto cuenta con tres notables paquetes de trabajo: desarrollo de la aplicación, estudio de los conceptos matemáticos y elaboración de la memoria. Como no se ha realizado un desarrollo de los entregables como tal, y dada la naturaleza de este trabajo, los

paquetes de trabajo de estudio y elaboración de la memoria no tienen asociados entregables. No obstante, estos serían la adquisición de los conocimientos necesarios y la propia memoria.

Y el paquete de trabajo restante, el desarrollo de la aplicación, junto con sus subpaquetes, cuenta con el apartado anterior de tareas como explicación de lo que en él se ha realizado. Se ha dividido según las distintas capas de la aplicación. Se ha de notar que el trabajo referido a desarrollo envuelve las distintas fases de un desarrollo iterativo, siendo estas análisis/diseño, implementación y pruebas.



3.2. Tiempo

Una vez definido el alcance, esta sección contiene un cronograma Gantt del proyecto y una estimación de dedicación para cada grupo de tareas.

Cronograma

El cronograma esta divido por semanas para no hacer una tabla desorbitadamente grande. La primera semana corresponde con la semana del 21 al 27 de febrero y la vigésima con la semana del 4 al 10 de julio.

Cronograma																				
Tareas	Febrero-Julio																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Modelo de dominio																				
Lógica de negocio																				
Persistencia																				
Presentación																				
Pruebas																				

Estimaciones de dedicación

Plan de dedicaciones	
Tareas	Horas
Estudio matemático	50
Desarrollo del modelo de dominio	60
Desarrollo de la lógica de negocio	50
Desarrollo de la persistencia	15
Desarrollo de la presentación	60
Elaboración de las pruebas	25
Elaboración de la memoria y defensa	50
Total	**310**

3.3. Seguimiento y control

En este último apartado se desarrolla ligeramente el seguimiento y control que se ha realizado sobre el Proyecto.

Las tareas han sido realizadas con éxito y pocas incidencias pueden ser destacadas, a excepción del inicio de implementación de la aplicación. El cronograma realizado en la planificación inicial sufrió un ajuste. Debido a alguna dificultad de gestión, todo el proyecto se desplazó de febrero a abril, que fue donde realmente se comenzó a desarrollar la aplicación.

El cronograma real de trabajo se replanteó en el siguiente:

Cronograma																			
Tareas	Febrero-Julio																		
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Modelo de dominio																				
Lógica de negocio																				
Persistencia																				
Presentación																				
Pruebas																				

4. Desarrollo de la aplicación

4.1. Análisis/Diseño

Como se ha comentado previamente el proyecto ha tenido requisitos variables, y no ha sido hasta una etapa avanzada de este, que se han asentado dichos requisitos. Por ello, el análisis y el diseño han ido bastante unidos.

Análisis

Tener bien definidos los objetivos es un factor fundamental para el éxito de un proyecto. Los requisitos analizados en esta etapa suponen darle forma y definir el alcance del trabajo. No obstante, reincidimos en que los requisitos funcionales han ido siendo modificados según se solidificaba la idea final del Proyecto, por tanto, no se ha hecho un exhaustivo análisis inicial, sino que se ha ido formulando conforme avanzaba este. A pesar de esto, se han logrado gestionar exitosamente los requisitos no llegando a suponer un obstáculo para el desarrollo de este trabajo.

Muchos requisitos han sido descartados de la idea inicial y otros tantos se han incorporado posteriormente. la siguiente tabla contiene los requisitos funcionales y no funcionales finales:

Requisitos
Funcionales
Poder crear, cargar y eliminar complejos simpliciales.
Poder crear, cargar y eliminar funciones booleanas.
Un complejo simplicial queda definido por un nombre que lo identifica, su lista de vértices y su lista de símplices.
Una función booleana queda definida por su nombre que la identifica, su número de variables y el nombre de estas y las salidas ordenadas lexicográficamente.
Que una vez creados tanto los complejos simpliciales como las funciones booleanas puedan listarse para recorrerlas fácilmente.
Que el listado de complejos y funciones pueda ordenarse según varios criterios y cuente con un buscador.
Que una vez cargados los objetos de la aplicación[2] se pueda operar con estos.
Que se puedan modificar los objetos de la aplicación sin provocar inconsistencias en estos.
Que los nombres que identifican a los objetos de la aplicación sean alfanuméricos.
Dentro de los operadores disponibles tanto los complejos simpliciales como las funciones booleanas pueden transformarse en su isomorfo correspondiente.
Que sobre los complejos simpliciales se puedan calcular el link y el star de sus vértices.
Como operadores sobre complejos simpliciales se encuentra el join y el cono.
Que se puedan ejecutar expansiones y colapsos elementales sobre el complejo simplicial previamente cargado.
Que se puedan crear campos de vectores gradiente, tanto de forma automática como manual.
Que, si el complejo simplicial cuenta con campos de vectores, estos se puedan cargar y visionar.

[2] Con objetos nos referimos a complejos simpliciales o funciones booleanas, pero para no repetir constantemente empleamos objetos para referirnos a ambos.

Requisitos
Funcionales
Que la creación manual de un campo de vectores cuente con un diagrama de Hasse asociado a dicho campo que se vaya modificando según se añaden vectores a este.
Que si al añadir manualmente vectores al campo se termina formando un ciclo este se advierta y muestren los vectores que ocasionaron dicho ciclo de color rojo.
Todos los operadores que generen un objeto de la aplicación tales como el join, el link y el star entre otros, permitan persistir dicho objeto.
Que cada vez que se desee modificar un objeto de la aplicación y este cuente con un nuevo nombre que coincida con otro ya existente se advierta al usuario.
Que todos los fallos puedan ser revertidos fácilmente, de modo que el usuario tenga el control sobre la aplicación.
Que todos los errores o fallos que pueda cometer el usuario accidentalmente sean evitados previamente y aquellos de los que no sea posible se adviertan claramente.
Que la interfaz de usuario tenga un diseño responsive en todas sus ventanas.
Un apartado de ayuda e información acerca de la aplicación para orientar al usuario.
No funcionales
Implementar la aplicación como una aplicación de escritorio local.
Persistir toda la información en ficheros JSON de manera local.
Dividir el almacén de ficheros JSON en dos carpetas, la que contiene los complejos simpliciales y la que contiene las funciones booleanas.
Adecuar el desarrollo de la aplicación al diseño en tres capas.
Desarrollar la interfaz de usuario siguiendo los principios de usabilidad, para que esta sea efectiva, eficiente, segura, útil, se pueda aprender fácilmente y sea sencillo recordar su uso.
Que los métodos más complejos e importantes de la aplicación busquen la mejor eficiencia, no llegando a sobrepasar tiempos de ejecución significativos.

Habiendo introducido los requisitos de la aplicación, obtenemos los casos de uso representados en la Ilustración 9.

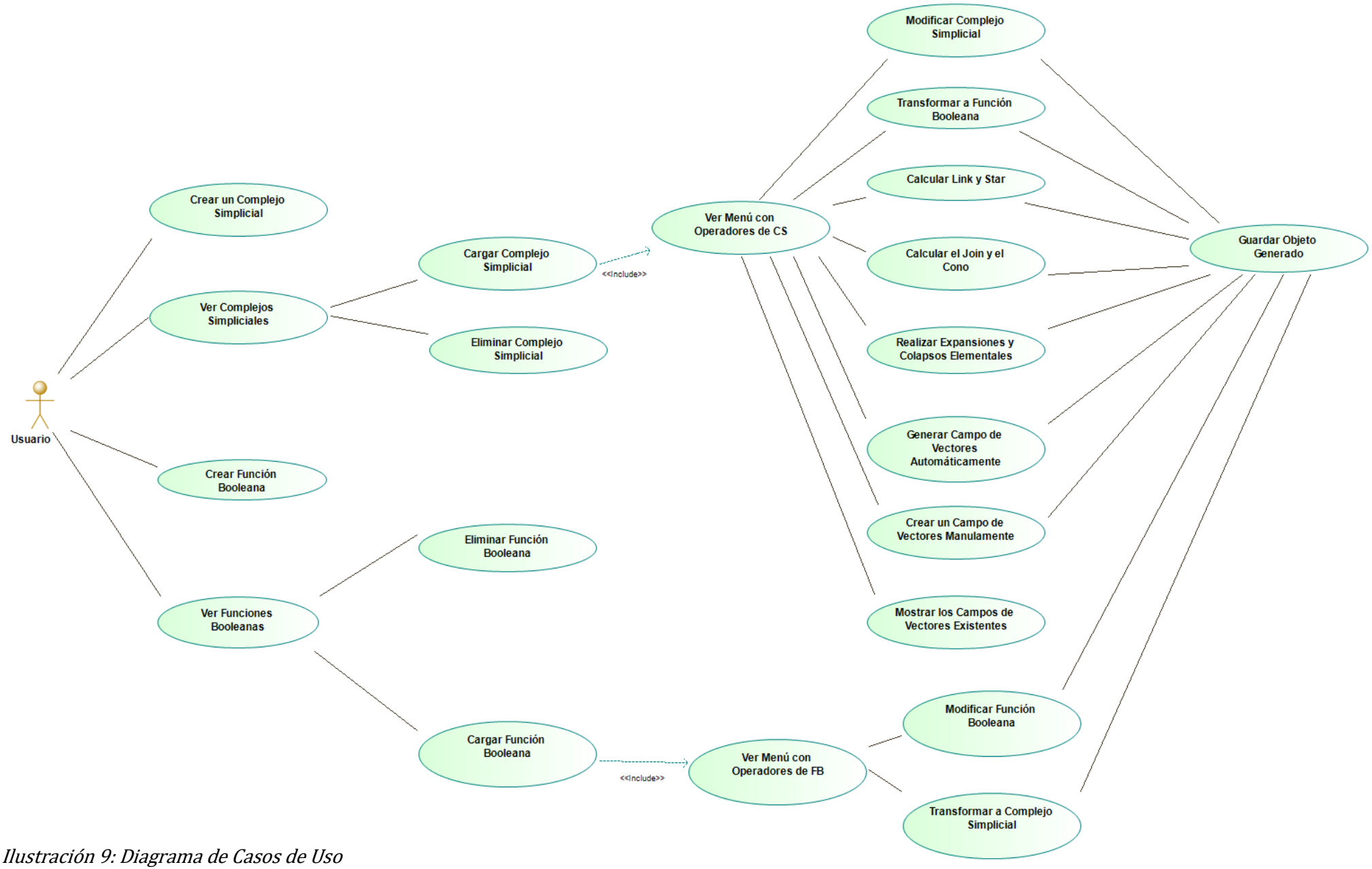


Ilustración 9: Diagrama de Casos de Uso

Diseño

Así como el análisis es una pieza fundamental para una buena definición del alcance y objetivos, el diseño es clave para que el software desarrollado esté bien desacoplado, no haya cohesión entre los distintos módulos, la interacción entre las capas sea con las mínimas dependencias y cuya calidad pueda ser mensurada y evaluada.

Como el análisis del Proyecto no ha podido ser llevado a cabo como debiera haber sido idealmente, pero siendo consistente a pesar de ello, el diseño ha recibido una especial atención debido a que siendo un proyecto que trata temas matemáticos, la rigurosidad de un buen diseño era no solo contingente sino necesaria.

De esta forma y con una exhaustiva dedicación al diseño de cada método, clase, modulo y capa se ha construido una aplicación estable, que no falla al introducir pequeñas modificaciones o sugerencias, mensurable para poder llevar a cabo pruebas y medir la calidad pudiendo detectar necesidades de cambios o mejoras fácilmente.

Para el diseño de la arquitectura de la aplicación se ha escogido un modelo multicapa, donde cada capa compone un subsistema en el que se ubican clases y métodos con responsabilidades propias. De esta forma fomentamos la reutilización de código gracias a la encapsulación y evitando el acoplamiento entre los distintos métodos. Además, podemos programar lógica en componentes separados que puedan ser reutilizados en otras aplicaciones. Por último, esta arquitectura permite poder desarrollar cada capa en paralelo.

Dentro del diseño multicapa se ha escogido, como ya se ha comentado previamente, el diseño en tres capas: presentación, lógica de negocio y persistencia.

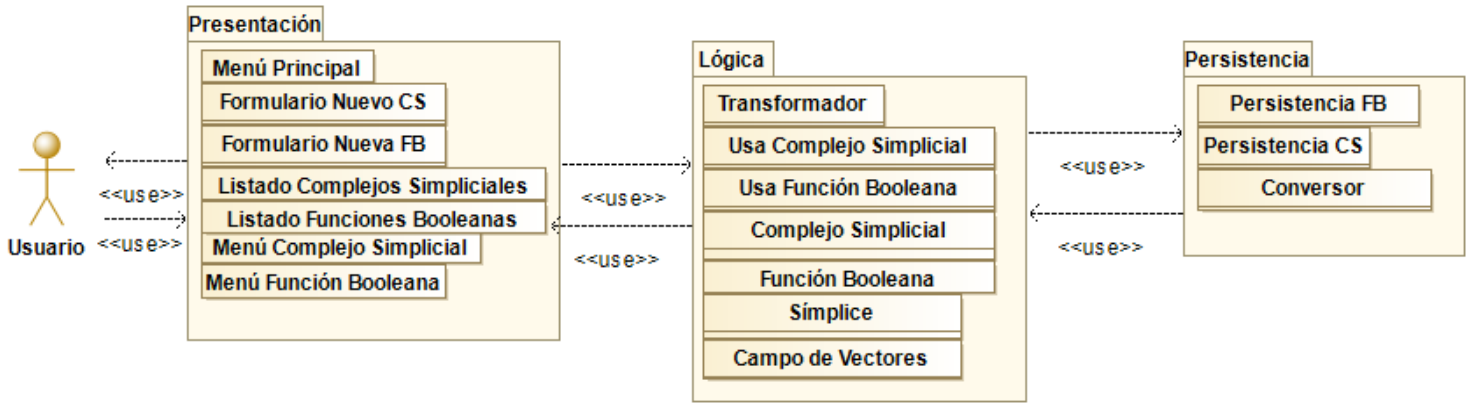


Ilustración 10: Estructura en tres capas de la aplicación

En la Ilustración 10 podemos observar esta división en capas a través de los distintos paquetes, y dentro de cada uno de ellos las clases que lo componen. Todas ellas serán explicadas en posteriores capítulos.

La capa (paquete) presentación contiene todas las clases que representan las ventanas de la interfaz gráfica. La lógica contiene las clases del modelo de dominio además de las clases que se encargan de gestionar los objetos que se persistirán, y una clase transformador que tiene como propósito hacer las transformaciones entre funciones booleanas y complejos simpliciales. Por último, la persistencia contiene las clases que persisten los objetos al sistema de persistencia, en este caso JSON, y un conversor cuyo fin es hacer el paso de objeto a diccionario JSON y viceversa.

Dentro del paquete de lógica encontramos las clases del modelo de dominio, clases que componen las unidades fundamentales y el núcleo de la aplicación. Por ello, en la etapa de diseño es necesario modelar a estas con un diagrama de clases. En él, además de las clases del modelo de dominio, también encontramos las clases de la lógica de negocio que se encargan de trabajar con dichas clases.

Como objetos del modelo de dominio encontramos los siguientes:

- SimplicialComplex: (modela a los complejos simpliciales). Cuenta con un identificador, un entero que define el conjunto omega, tres relaciones con otras clases del modelo de dominio que son sus campos de vectores asociados, los símplices que lo componen y la lista de facets. Además, encontramos tres atributos calculados que son la característica de Euler, el c-vector y la matriz de adyacencias de sus caras.

- BooleanFunction: (modela las funciones booleanas). Está compuesta por un identificador, un entero que define el número de variables de la función, los nombres de cada una de ellas, una lista con las salidas de la función y un flag calculado que determina si la función es o no monótona.

- Simplex: (modela a los símplices). Está compuesta por un identificador, su dimensión y un índice opcional para indexar a dicho símplice si está contenido en un complejo simplicial. Por último, contiene una lista de símplices opcional que modela las caras de dicho símplice.

- VectorField: (modela los campos de vectores). Está compuesto por un identificador, el c-vector del complejo simplicial al que está asociado, una lista de bloques calculada a partir de la matriz de adyacencia de caras, el conjunto de rutas del campo (siendo este conjunto un diccionario donde la clave es el símplice y su valor los símplices accesibles desde este), los vectores añadidos y dos listas auxiliares de símplices que ya han sido target y source.

En la Ilustración 11, encontramos el correspondientes diagrama de clases:

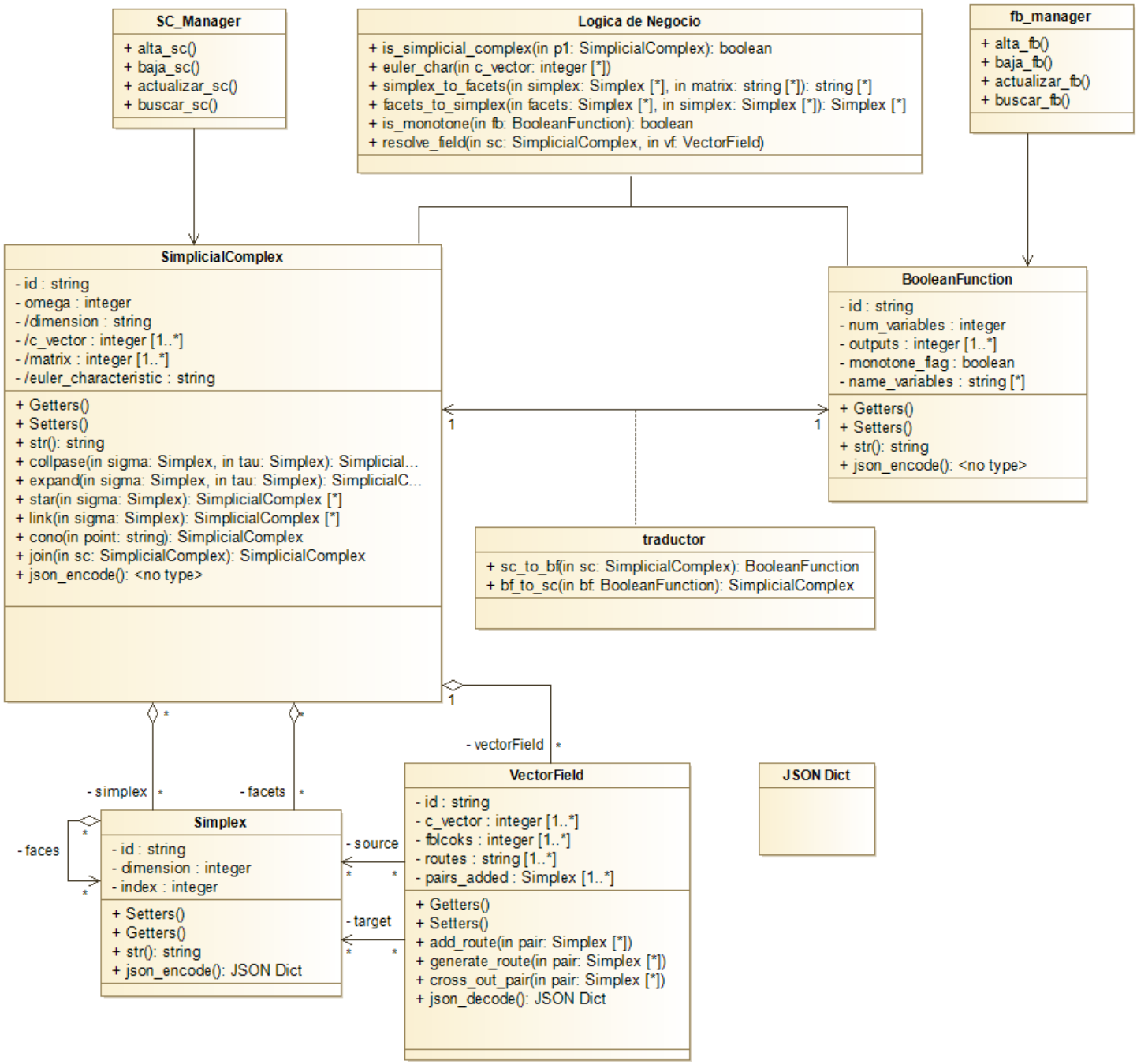


Ilustración 11: Diagrama de Clases

El resto de clases forman parte de la lógica de negocio encargada de trabajar con el modelo de dominio para desarrollar la funcionalidad.

El diseño de la capa de persistencia no supone un gran reto, ya que no vamos a utilizar una base de datos relacional con lo que ello conlleva

en las etapas de análisis, diseño e implementación. Al utilizar ficheros JSON como sistema de persistencia, simplemente es necesario transformar los objetos a un diccionario JSON, eso podemos encontrarlo en el método "json_encode" de cada clase, que se encarga de guardar cada atributo en un valor del diccionario JSON y, además, una serie de métodos de la capa de persistencia encargados de transformar los objetos a los diccionarios y viceversa. Y, por último, para guardar dichos ficheros se utiliza un directorio a modo de almacén, separado en dos subcarpetas, una para complejos simpliciales y otra para funciones booleanas.

Siguiendo con el diseño de la capa de presentación en las Ilustraciones 11 y 12 encontramos dos prototipos a papel de la interfaz, conocido como "low-fi" o de baja fidelidad centrado en el posicionamiento de los distintos controles y en cómo se aprovecharía el espacio de la interfaz.

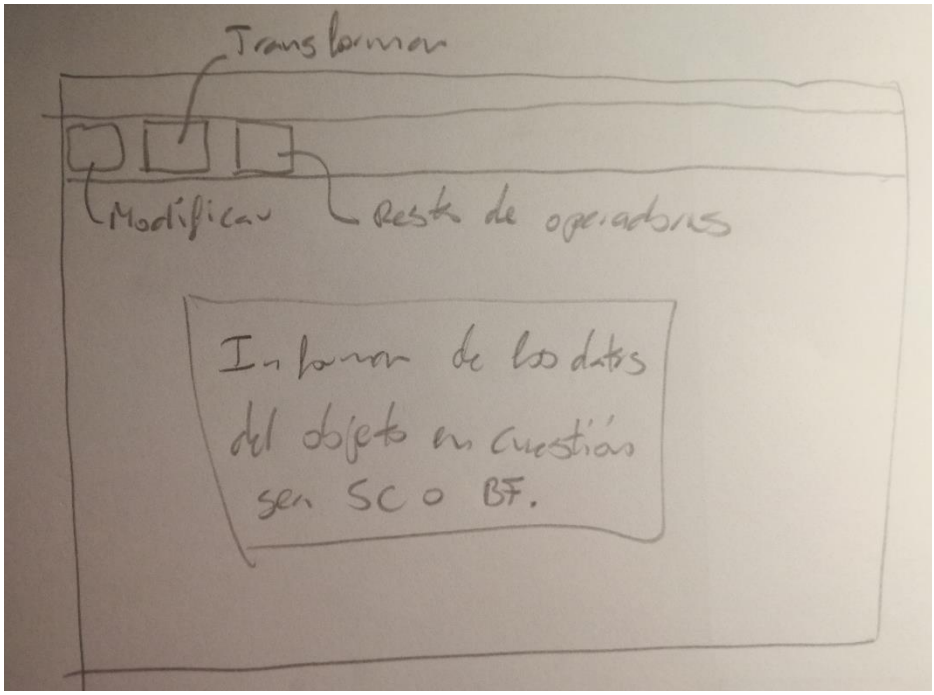


Ilustración 12: Prototipo a papel del área de trabajo

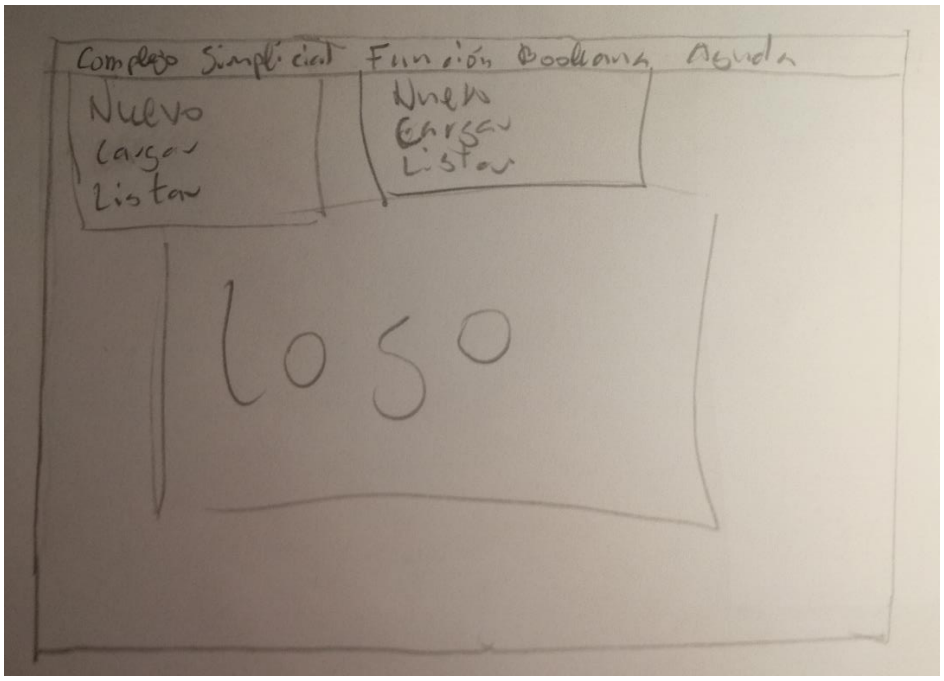


Ilustración 13: Prototipo a papel del menú principal

De los formularios de creación no se hizo un prototipo previo, sino que se diseñaron con Qt Designer. Se fueron adaptando los tamaños y probando diferentes configuraciones hasta encontrar la idónea.

Para finalizar con el apartado de diseño de la interfaz, la Ilustración 14 muestra un diagrama de flujo de las distintas ventanas de la interfaz. La capa de presentación es el intermediario entre la funcionalidad y el usuario. Por ello, su aspecto visual debe ser cuidado y amigable con el usuario, tratando de ser simple y estético pero funcional.

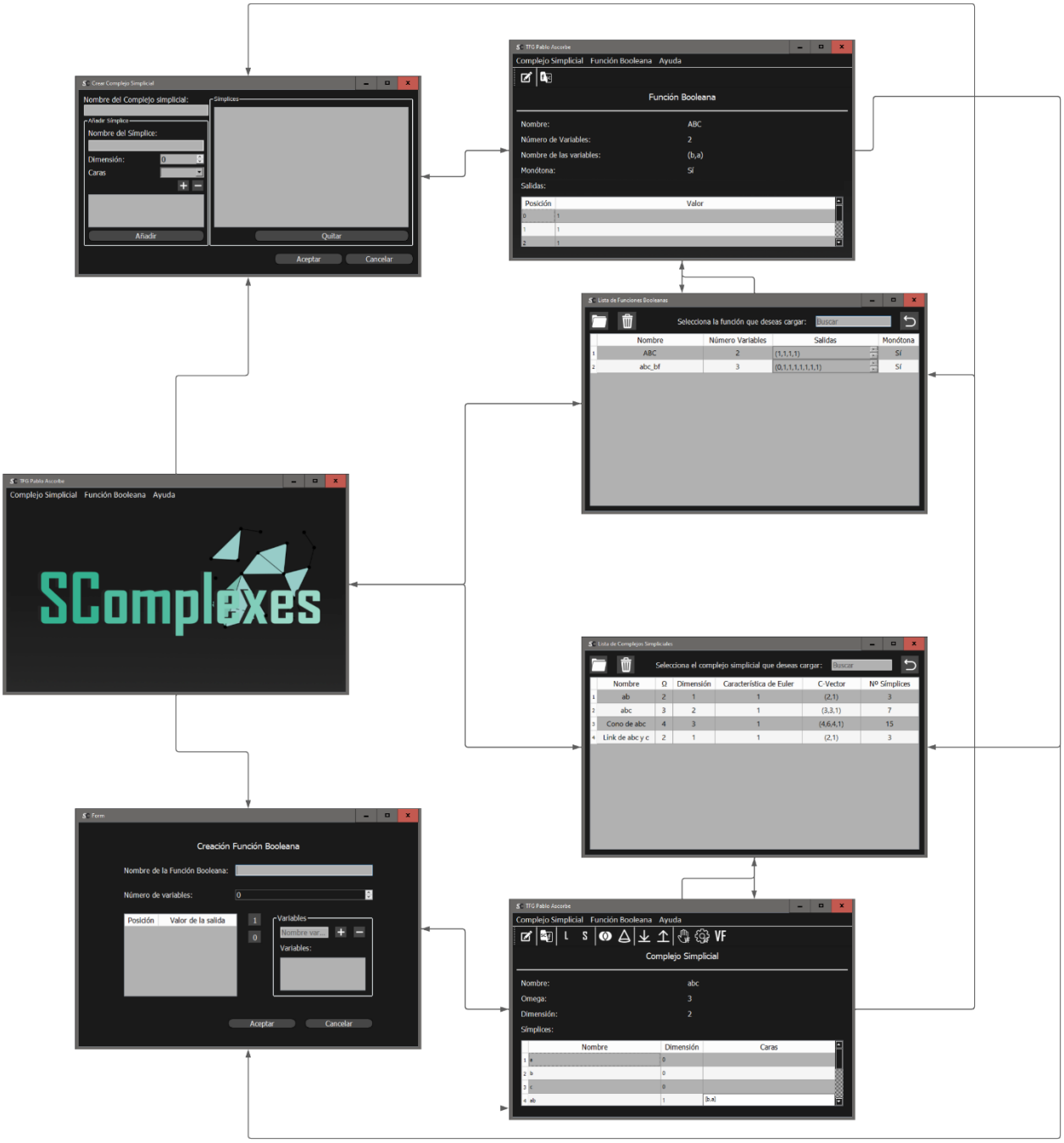


Ilustración 14: Diagrama de Flujo de la interfaz gráfica

En general tenemos dos áreas de trabajo diferenciadas para cada objeto de la aplicación, a las que podemos acceder desde la lista de dichos objetos. Desde el menú principal podremos crear y listar dichos objetos y hasta que no tengamos algún objeto creado no podremos acceder a su correspondiente área de trabajo. Cabe destacar que en los prototipos el menú contaba con las opciones nuevo, cargar y listar.

Pero al construir la aplicación se decidió eliminar la opción de cargar ya que la propia lista cumplía con esta función.

Como último inciso se hace notar que una vez desarrollados unos prototipos de alta fidelidad con la herramienta Qt Designer se trató de validar su usabilidad con dos usuarios con una formación previa dispar. De modo que gracias a esta evaluación de dichos prototipos se recibieron mejoras y sugerencias para hacer la interfaz más intuitiva y usable. Por ejemplo, añadir botones extra o atajos para agilizar algunos procesos y recibir algún aviso de operaciones críticas cuando iban a ser realizadas. Por último, al presentar la interfaz al tutor también se encontraron algunas sugerencias de mejora para hacer la interfaz final más clara e intuitiva para el usuario.

4.2. Implementación

Antes de entrar en detalles he de comentar que para cada clase y método se ha utilizado Doscstring para su correspondiente documentación (en [4] se puede encontrar más información al respecto). Por su gran popularidad y cantidad de posibilidades para hacer una amplia documentación, se ha optado por el estilo de Numpy.

Este proyecto y todo su código fuente es de libre acceso en https://github.com/paascorb/TFG. Así, todos los métodos comentados y código mencionado pueden ser consultados en el anterior enlace. En la memoria solo se explica el código de algunos métodos cuya implementación presente algún tipo de enfoque interesante.

4.2.1. Modelo de dominio

Para la implementación del modelo de dominio y siguiendo el diseño especificado en la etapa anterior, encontramos cuatro clases y un fichero auxiliar, que cuenta con una serie de métodos encargados de encapsular código con un comportamiento más genérico. Bien es cierto

que algunos métodos de este fichero auxiliar se podrían incorporar en su clase correspondiente, siendo que no son tan genéricos como el resto. No obstante, por mantener un diseño consistente se han decidido implementar dichos métodos en el fichero auxiliar.

Aunque en nuestro contexto solo se suele encapsular código para una clase, se ha tomado esa decisión pensando en un diseño extensible y ampliable. Si posteriormente se desease añadir más funcionalidad a la aplicación, estos métodos que han sido especificados como auxiliares contarán con un comportamiento reutilizable en muchos casos.

Para no repetir la misma idea varias veces, señalamos aquí que todas las clases cuentan con los siguientes métodos:

- "__init__", a modo de constructor.
- "__str__" y "__repr__", como métodos que transforman el objeto a una cadena para poder representarlo y mostrarlo.
- "__eq__", para poder realizar comparaciones entre instancias de la clase.
- "json_encode", encargado de transformar los datos de la instancia en un diccionario json.

Complejo simplicial

La clase de SimplicialComplex modela el concepto de complejo simplicial. Esta clase cuenta con los siguientes atributos:

- Obligatorios
    - Identificador, a modo de nombre.
    - Omega, que define el número de vértices del complejo.
- Opcionales[3]
    - Lista de símplices.
    - Lista de facets.
- Calculados
    - Dimensión del complejo simplicial.

[3] Como estos atributos son opcionales, podemos construir el objeto pasándole ambos atributos, ninguno o solo uno de ellos. En todos los casos se construirá un objeto válido.

- Matriz de adyacencia de caras (ordenada por dimensión).
- C-vector.
- Característica de Euler.

El constructor de esta clase hace una serie de comprobaciones para validar que los datos introducidos son correctos. Por ejemplo, si el número de vértices supera omega se lanzará una excepción. Otra manera de haber manejado estas situaciones podría haber sido una detallada precondición que advierta de las condiciones que los parámetros deben cumplir. Pero se ha decidido comprobar estos errores en el modelo de dominio y no delegar esto en capas superiores, prefiriendo limitar esa funcionalidad a la propia clase, ya que podría funcionar como una librería independiente. De modo que, si se utiliza mal, será lanzada una excepción advirtiendo dicho error.

Para los colapsos elementales aparecen dos métodos. El primero recibe una pareja como parámetro, comprobando si dicha pareja es válida para realizar esta operación. El segundo comprueba si puede colapsar llamando al método anterior y, de ser posible, elimina la pareja recibida del complejo simplicial, es decir, realiza el colapso y se devuelve a sí mismo como complejo simplicial derivado de dicho colapso.

La expansión elemental funciona prácticamente igual que el colapso, variando únicamente que en lugar de eliminar la pareja recibida la inserta en el complejo.

Hay otro bloque de métodos que definen operadores sobre símplices, siendo estos: la clausura, el borde, el star y el link.

Para finalizar con esta clase, señalamos que incluye dos métodos para los campos de vectores: uno para añadir a un complejo un campo de vectores ya existente y otro para crear uno nuevo y devolver su referencia.

Simplex

La clase Simplex modela a los símplices. No siendo demasiado compleja cumple una función vital como unidad fundamental para la aplicación. Para su implementación se pensó en dos enfoques, uno más matemático y otro más conceptual e intuitivo. El matemático consistía en modelar los símplices mediante conjuntos de vértices. Esta definición es matemáticamente rigurosa. La dimensión nos la da el cardinal del conjunto y las caras son los distintos subconjuntos. No obstante, se puede perder algo de información, como el nombre de dicho símplice y la posibilidad de verlo como algo independiente al contexto del complejo simplicial. Así, representar los símplices como instancias de una clase termina siendo más sencillo de entender si no se tiene un conocimiento matemático previo. El enfoque matemático era un enfoque más riguroso y posiblemente más eficiente, pero uno de los objetivos de la aplicación es ayudar a facilitar la comprensión y trabajo con los conceptos matemáticos.

Los atributos de la clase son:

- Obligatorios
    - Identificador, a modo de nombre.
    - Dimensión.
- Opcionales
    - Caras, modelada como una lista de símplices.
    - Índice, identifica la posición en el complejo simplicial.

No tiene métodos propios más allá de los genéricos, los getters y setters. El método set de la lista de caras hace una serie de comprobaciones para determinar si la lista de caras es válida. Siguiendo el mismo razonamiento que en la clase anterior, se ha preferido ser restrictivo, no obstante, se podría haber sido más flexible determinando dichas restricciones en la documentación.

VectorField

Clase que modela los campos de vectores. Tiene como atributos:

- Identificador.
- C-vector.
- Bloques de la matriz de adyacencia de caras.
- Lista de vectores incorporados al campo.
- Dos listas auxiliares que registran qué símplices no pueden ser ni source ni target.

Como métodos de esta clase encontramos métodos de comprobación, utilizados a la hora de ver si un posible vector es válido. Por ejemplo, comprobando si el vector es una pareja válida (uno es cara del otro) y si cualquiera de los símplices de dicha pareja no son ni source ni target de vectores ya en el campo.

Si la pareja es válida, el método “add_route”, se encarga de incorporar el vector en dos pasos:

1. Se calcula la lista de símplices alcanzables finales desde el símplice origen. En este paso es donde detectamos si un ciclo ha sido generado, ocurriendo si el origen se encuentra a sí mismo en sus símplices alcanzables.
2. En el resto de claves del diccionario de rutas se actualizan todos aquellos símplices que podían alcanzar al origen, sustituyendo a este por sus nuevos símplices alcanzables finales. En este paso no pueden darse ciclos, por lo tanto, no es necesario hacer dicha comprobación.

BooleanFunction

Clase que modela las funciones booleanas. Cuyos atributos son:

- Identificador.
- Número y nombre de las variables.

- Salidas, modeladas como una lista de 0's y 1's.
- Flag opcional que determina si la función es o no monótona.

Como la aplicación está fundamentalmente basada en los complejos simpliciales no hay más operadores que la transformación para las funciones booleanas, por ello sus métodos son los genéricos comentados anteriormente.

Cabe destacar que el constructor, al igual que las clases anteriores, hace una serie de comprobaciones para verificar si los datos recibidos son correctos. Por ejemplo, si el número de variables no genera el número de salidas correspondientes, salta una excepción.

El método set del flag de monotonía es algo distinto en este aspecto, no comprueba si el valor introducido es congruente con la función, pero se explicita en la documentación que, como precondición, el flag y la monotonía deben coincidir. Este flag tiene como uso el ahorro de tiempo computacional, evitando tener que comprobar si la función es monótona o no cada vez que se necesite.

Auxiliary

Como se ha comentado, este fichero contiene una serie de métodos auxiliares, agrupados según el fin de estos. A modo de ejemplo, solo nos referimos a algunos de ellos. Por ejemplo:

- Complejo Simplicial
    - "is_simplicial_complex". Comprueba si una lista de símplices es válida para generar un complejo simplicial.
    - "simplex_to_facets". Busca aquellos símplices que no tengan ninguna cocara en la matriz de adyacencias proporcionada.
    - "facets_to_simplex". Recorre en profundidad cada facet de la lista hasta generar el conjunto de símplices.

    - “order_and_index”. Ordena e indexa una lista de símplices. El criterio de ordenación es primero por dimensión y segundo por el identificador de los símplices.
- Funciones Booleanas
    - “is_monotone”. Comprueba si una lista de salidas de 0’s y 1’s conforman una función monótona.
    - “check_output”. Encargado de determinar los hijos de una posición proporcionada, que deberían tener un 1 para que la función fuera monótona. Este método se ha refactorizado varias veces debido a que, si el número de variables aumenta considerablemente, al ser potencias de dos, el coste computacional crece de forma exponencial. Por ello, es necesario tratar de buscar la manera óptima de calcularla. En la sección de pruebas se incidirá más en este aspecto.

      Sin entrar en detalles, recorre de manera inversa las lista de salidas de la función. Al encontrar un ‘1’ llama a “check_output” pasándole como parámetro su posición, para así buscar sus “hijos”. Por ejemplo, si nuestro número es el siete (en binario es 111), se calculan sus hijos eliminando iterativamente cada ‘1’ del número en binario de izquierda a derecha. Es decir, la primera iteración nos daría el 011, que es el 3, la segunda el 101, que es el 5 y la última el 110, el 6. De este modo obtenemos todos sus hijos directos, necesitando calcular de igual manera los hijos de forma recursiva. Para no comprobar recursivamente posiciones que ya sepamos que forman parte de los hijos de la “raíz”, siendo en nuestro caso la “raíz” el 7, se añaden los que ya hayan sido comprobados a una lista que se va pasando entre llamadas recursivas.

4.2.2. Lógica de negocio

Manager

En el módulo de lógica de negocio empezamos describiendo los métodos encargados de trabajar con los objetos de la aplicación. Dividido entre complejos simpliciales y funciones booleanas encontramos “SimplicialComplexManager” y “BooleanFuncionManager”. Ambos disponen de métodos CRUD (Create, Read, Update and Delete/Crear, Leer, Actualizar y Borrar en castellano), pero definidos por la clase correspondiente.

Exceptions

Excepciones propias, pensadas para el contexto de la aplicación, que tratan de dar más información a las posibles excepciones generadas por la aplicación.

Traductor

El traductor o transformador “Traductor.py” es el que contiene los métodos encargados de transformar las funciones a complejos y viceversa. A continuación, se explica su código.

Empezando por la transformación de complejo simplicial a función booleana, el método “simplicial_complex_to_boolean_function” se encarga de ir recorriendo la lista de símplices e ir mapeando estos en las correspondientes salidas de lo que será la función booleana resultado. De modo que, si tenemos 3 vértices, estos vértices serán las variables de la función, ocupando cada uno una posición correspondiente a una potencia de 2 (posiciones 1, 2 y 4). A estas posiciones le ponemos un ‘1’ en la lista de salidas.

Una vez se tienen los vértices, basta recorrer hacía arriba el resto de símplices calculando el OR de sus caras. El uso del operador reduce lo simplifica:

```
return reduce(lambda x, y: x | y, faces_pos)
```

(Con reduce estamos aplicando el operador OR consecutivamente a todas las posiciones de las caras del símplice, posiciones que encontramos en la lista “faces_pos”).

Para una mejor comprensión presentamos un ejemplo. Partimos del 2-símplice (triángulo relleno). Sus tres vértices v_0, v_1 y v_2, tendrán las posiciones 1, 2 y 4 en las salidas. Sus cocaras, son las aristas $\{v_0, v_1\}$, $\{v_1, v_2\}$ y $\{v_0, v_2\}$. Para calcular sus posiciones, la operación OR nos proporciona el valor 3 para la primera (001 OR $010 = 011$), 5 para la segunda y 6 para la tercera. Finalmente, para el triángulo $\{v_0, v_1, v_2\}$ tendremos el 111 (7).

El segundo método “boolean_function_to_simplicial_complex” es algo más complejo. Se explicita en la precondición que la función recibida sea monótona, ya que de otra manera no se podrá generar el complejo simplicial. El método comienza iterando de manera revertida la lista de salidas y, por cada ‘1’ que encuentra, genera el símplice correspondiente y su clausura, es decir, su construcción en profundidad hasta los vértices. Esto se consigue, al igual que hacíamos en el método “is_monotone”, encontrando todos los hijos de la posición que tiene un ‘1’. Estos hijos serán las caras del símplice encontrado.

Para determinar a qué símplice corresponde cada posición debemos fijarnos en el número de 1’s que tiene dicha posición en binario. De modo que el 111 (7) será un 2-símplice. Es el proceso inverso a lo que hacíamos en el transformador anterior. Es importante señalar que, al tener toda la lista de símplices traducida de la lista de salidas, los nombres de estos son la posición que ocupan. Por ello, necesitamos reemplazar estos nombres por los nombres de las variables de la función. Los vértices se llamarán como las variables y las cocaras serán la concatenación de dichos nombres (vértice ‘a’ y vértice ‘b’ generan arista ‘ab’).

Para terminar con los transformadores, cabe mencionar que el resto de parámetros tienen una relación directa: omega es el número de

variables, el nombre de las variables es el de los vértices y el identificador asociado es el del objeto traducido.

Join

Aunque originalmente “join” iba a ser un método de la clase “SimplicialComplex”, se ha decidido, por diseño, extraerlo e introducirlo en la lógica de negocio, ya que utiliza dos complejos simpliciales y no se trabaja sobre una instancia única de complejo simplicial.

El método genérico es “join”. Con él se pueden calcular el resto, como suspensiones o, en nuestro caso, “cono”. La funcionalidad de join se fundamenta en el hecho de que los únicos que tienen nombres únicos son los vértices (el resto de símplices son las concatenaciones de cada nombre de los vértices que contienen). De esta forma sabemos que el 2-símplice cuyos vértices son ‘a’, ‘b’ y ‘c’ se llamará ‘abc’. Partiendo de esto, el join crea una lista con todos los símplices de ambos complejos, (recordemos que para poder hacer el join de dos complejos estos no deben compartir ningún vértice). La implementación no requiere ningún comentario especial. Basta con ir construyendo de abajo hacia arriba los distintos símplices e ir generando los nombres correctamente, además de asociarles las caras correctas.

VectorFieldResolver

El método encargado de calcular un campo de vectores hasta no poder añadirle ningún vector nuevo es “resolve_field”. Va recorriendo cada bloque (recordemos que dichos bloques eran cajas de la matriz de adyacencia de caras) y, cuando encuentra el primer ‘1’, añade dicho vector al campo. Como al añadirlo se habrán inhabilitado otros tantos vectores, habrá un momento en el que termine. Para cada bloque se llama a la función “resolve_block” que resuelve de manera más granular cada bloque.

En este método, si al añadir el vector encontrado se detecta una CycleException entonces se elimina dicha pareja y se sigue a la siguiente evitando así introducir ningún ciclo.

4.2.3. Persistencia

Los métodos que se encargan de persistir y realizar las operaciones CRUD, directamente llamando a los métodos de serialización, son los que están contenidos en “PersistenceBooleanFunction” y en “PersistenceSimplicialConplex”. En ellos encontramos esos métodos de creación, lectura, actualización y borrado. Cabe destacar que están asociados a sus correspondientes métodos de la capa de lógica, pero de esta manera evitamos que la lógica conozca el método de persistencia consiguiendo así más interdependencia de módulos.

Una cosa que no se ha comentado en la parte de lógica pero que es interesante destacar aquí es que, desde la capa de lógica, para ciertos métodos como eliminar o leer donde no se necesita todo el objeto, solo su identificador, se crea algo similar a un DTO (Data Transfer Object). Realmente no es una clase implementada para ello, pero como todos los constructores permiten crear objetos vacíos (recibiendo únicamente el identificador), podemos simular estos DTO construyendo un objeto vacío.

Conversor

Sus métodos tienen como función transformar (codificar o decodificar) las instancias de las distintas clases en diccionarios JSON. Como la codificación viene dada por un método interno de cada clase, únicamente necesitamos hacer el paso de diccionario a instancia. Por tanto, es ir leyendo los distintos datos usando el método get del diccionario JSON accediendo a la clave que le dimos al codificar. El único punto para tener en cuenta es seguir un orden a la hora de poner dichos nombres para evitar confusiones.

En una primera versión, se serializaba el complejo simplicial con todos sus símplices y cada uno con todas sus caras hasta llegar a los vértices. Esto hacía que el fichero fuese realmente pesado y poco eficiente. Se cambió para que se persistieran solo los identificadores de las caras y no toda su información, siendo el conversor el encargado de ir asociando cada símplice a sus caras. Se incidirá de nuevo en el apartado de pruebas.

JSONPersistence

Son cuatro métodos:

- Serializar (guardar el diccionario JSON en un fichero .json).
- Deserializar (recuperar el diccionario del fichero asociado).
- Eliminar (los nombres de los ficheros .json coinciden con los identificadores de las clases que persisten).
- Leer (deserializar todos los ficheros de un determinado directorio del almacén).

4.2.4. Presentación

Toda la capa de presentación se ha implementado usando la librería PyQt5 y una herramienta conocida como Qt Designer. En esta encontramos layouts, widgets y ventanas. No es menester de este trabajo desarrollar el funcionamiento de dicha librería, para más información se puede consultar el sitio de referencia [5].

MainWindow

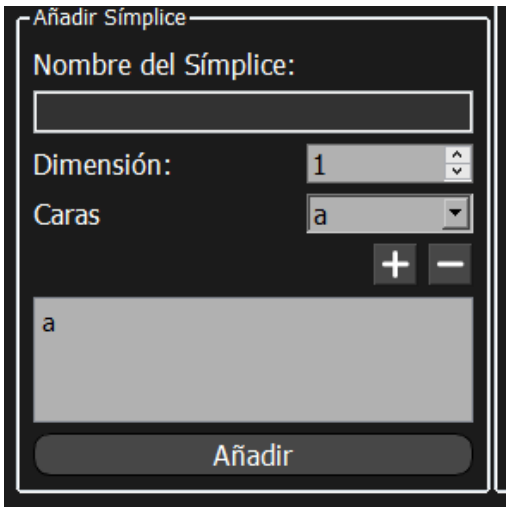
Es la clase que hereda de QMainWindow. Define la interfaz principal, desde la que se puede acceder a la funcionalidad de la aplicación. Aparte de los widgets y su configuración estructural y estética, encontramos una serie de métodos creados como triggers de los eventos asociados a clicar los distintos botones del menú.

Ilustración 15: Menú principal

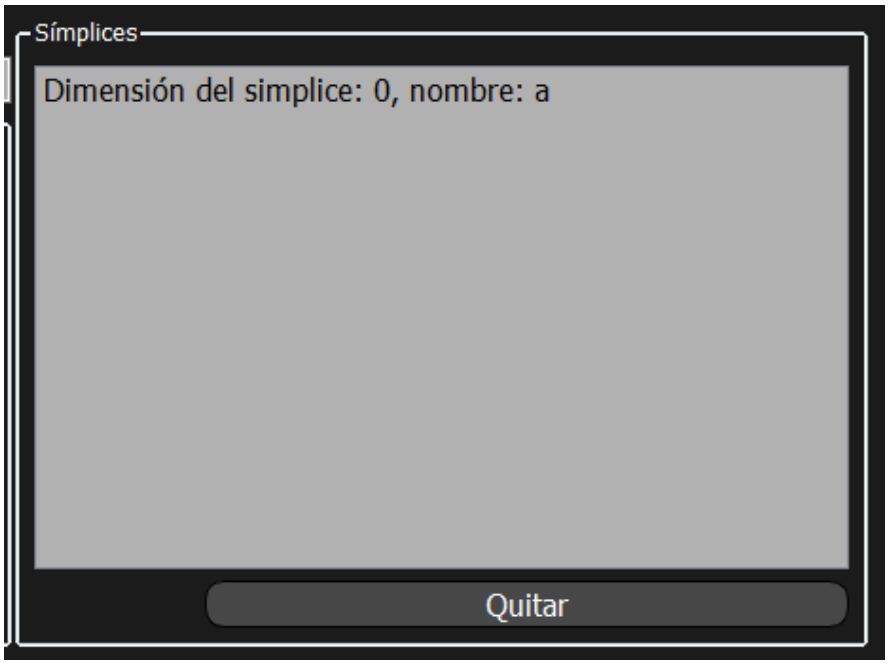
Nuevo_SC

Este formulario contiene los campos necesarios para crear un complejo simplicial. Donde se puede introducir el nombre del complejo y una sección para introducir los datos de los distintos símplices.

A modo de comentario decir que permite dar nombre a los vértices, es decir, cuando en el spinBox figure un ‘0’, se podrá introducir un nombre, si no es así el campo del nombre del símplice se deshabilita. También contiene un comboBox donde aparecerán las caras disponibles que podemos asociarle al símplice (estas caras aparecerán únicamente para símplices con una dimensión superior a 0).



A la derecha del formulario encontramos una lista que va mostrando todos los símplices añadidos por el momento. Si deseáramos eliminar alguno de ellos basta seleccionarlo y pulsar “Quitar”, así lograríamos lo deseado. Nótese que, si se eliminase un símplice que dependa de otro como, por ejemplo, un vértice que es cara de una arista, se eliminará todo el star abierto de dicho vértice, evitando dejar así un complejo simplicial incoherente.



Los métodos que encontramos en la clase Nuevo_SC, que hereda de QWidget, ya que no es una ventana principal, son eventos como, por ejemplo, “closeEvent” que se encarga de advertir al usuario que está saliendo sin guardar (si se ha clicado en aceptar este evento no sucederá).

Y métodos asociados a los distintos componentes del formulario que se encargan de asociar la funcionalidad a cada uno. Validando los datos o rellenando las listas con la información necesaria.

La vista final de la ventana podemos encontrarla en la Ilustración 16.

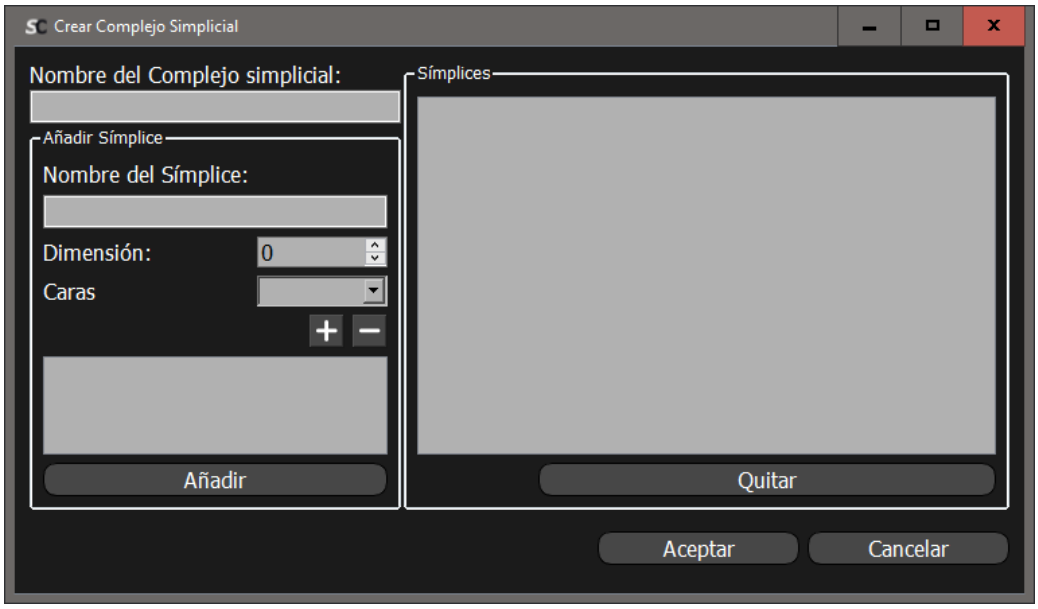


Ilustración 16: Formulario nuevo complejo simplicial

Nuevo_BF

El funcionamiento de este formulario es similar en forma al anterior. Por ello, no conviene desarrollarlo demasiado. Como único apartado interesante en el que detenerse, encontramos el spinBox que gestiona las variables. Si este se incrementa, la lista de salidas lo hace en potencias de dos. La vista del formulario es la siguiente:

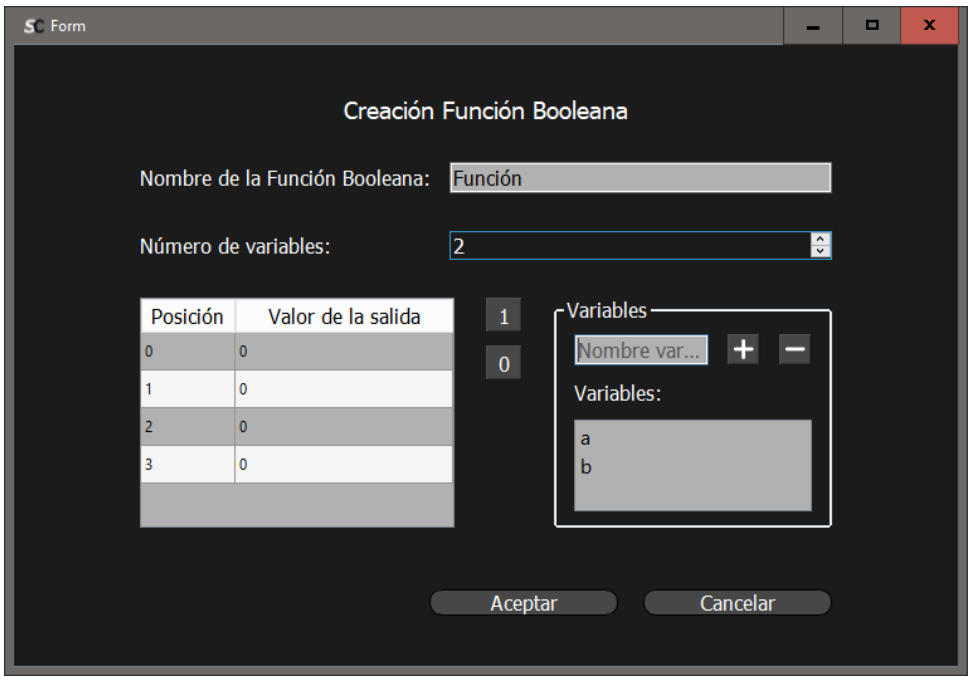
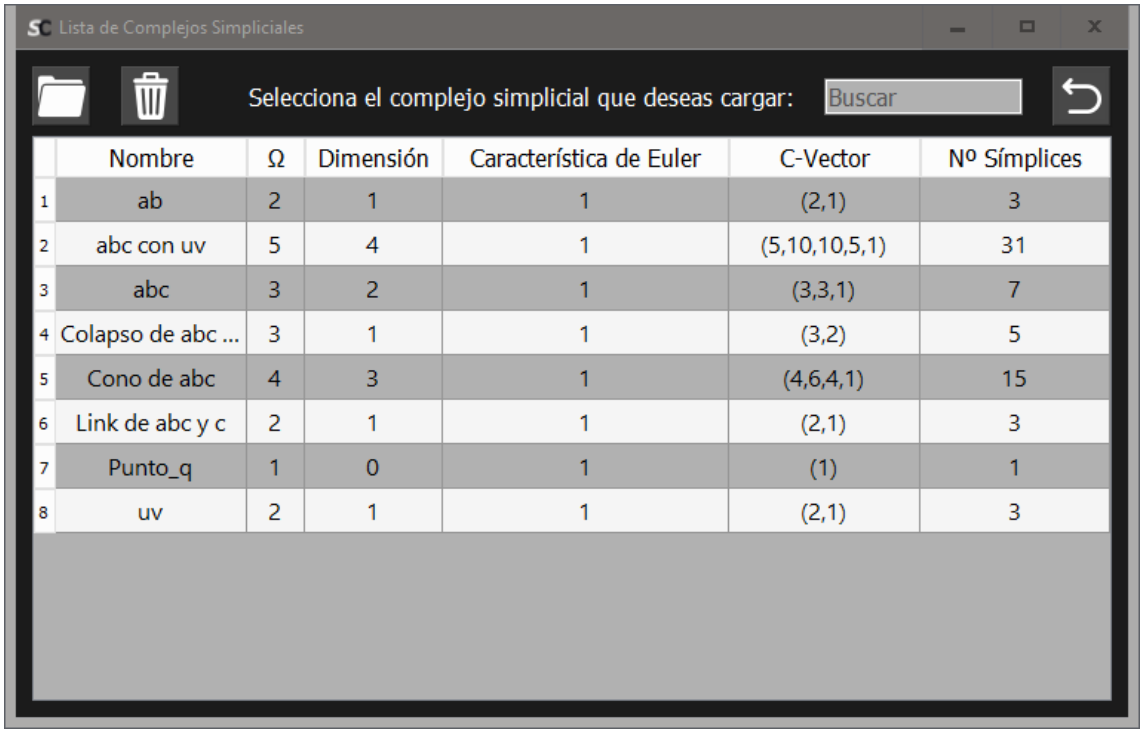


Ilustración 17: Formulario nueva función booleana

Listar_SC

En esta ventana encontramos una tabla que lista todos los complejos simpliciales persistidos en la aplicación. Sobre esta lista podemos cargar y eliminar. Además, la lista puede ser ordenada siguiendo los criterios de cada columna (con un clic se ordena crecientemente y con doble clic decrecientemente). También cuenta con un buscador para hacer más sencilla la interacción del usuario.

La ventana final de la lista es la siguiente:



	Nombre	Ω	Dimensión	Característica de Euler	C-Vector	Nº Símplices
1	ab	2	1	1	(2,1)	3
2	abc con uv	5	4	1	(5,10,10,5,1)	31
3	abc	3	2	1	(3,3,1)	7
4	Colapso de abc ...	3	1	1	(3,2)	5
5	Cono de abc	4	3	1	(4,6,4,1)	15
6	Link de abc y c	2	1	1	(2,1)	3
7	Punto_q	1	0	1	(1)	1
8	uv	2	1	1	(2,1)	3

Ilustración 18: Lista de complejos simpliciales

Listar_BF

Una lista muy similar a la anterior, pero para funciones booleanas, no hay ninguna diferencia significativa como para mencionarla. Su ventana final es la siguiente:

Ilustración 19: Lista de funciones booleanas

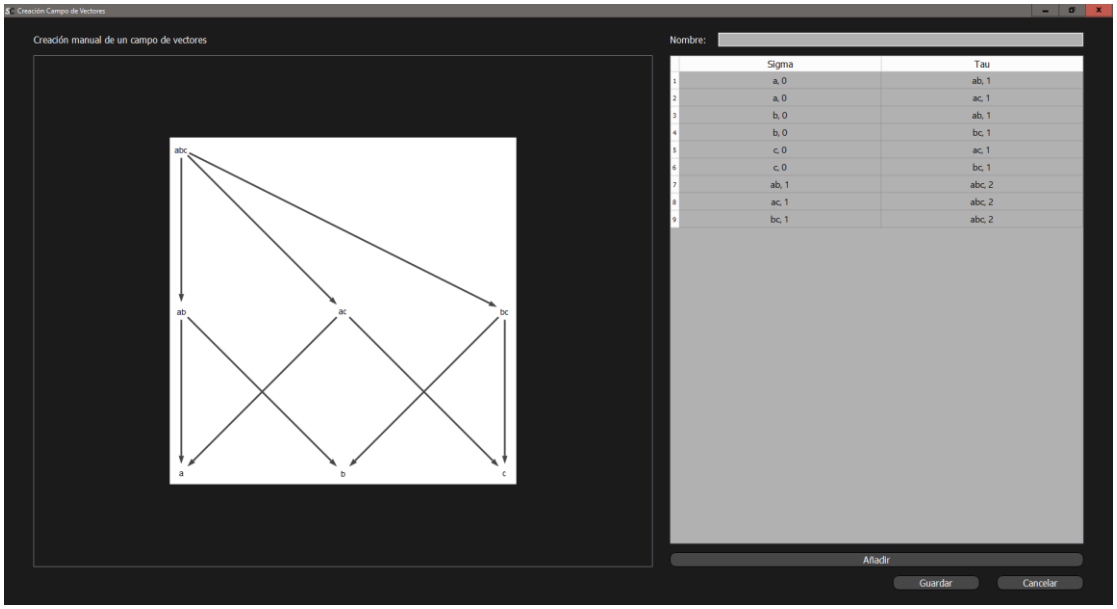
Menu_SC

Es el área de trabajo donde poder aplicar los operadores al correspondiente complejo simplicial cargado, es decir, solo podremos acceder a este menú cuando hayamos creado y cargado un complejo.

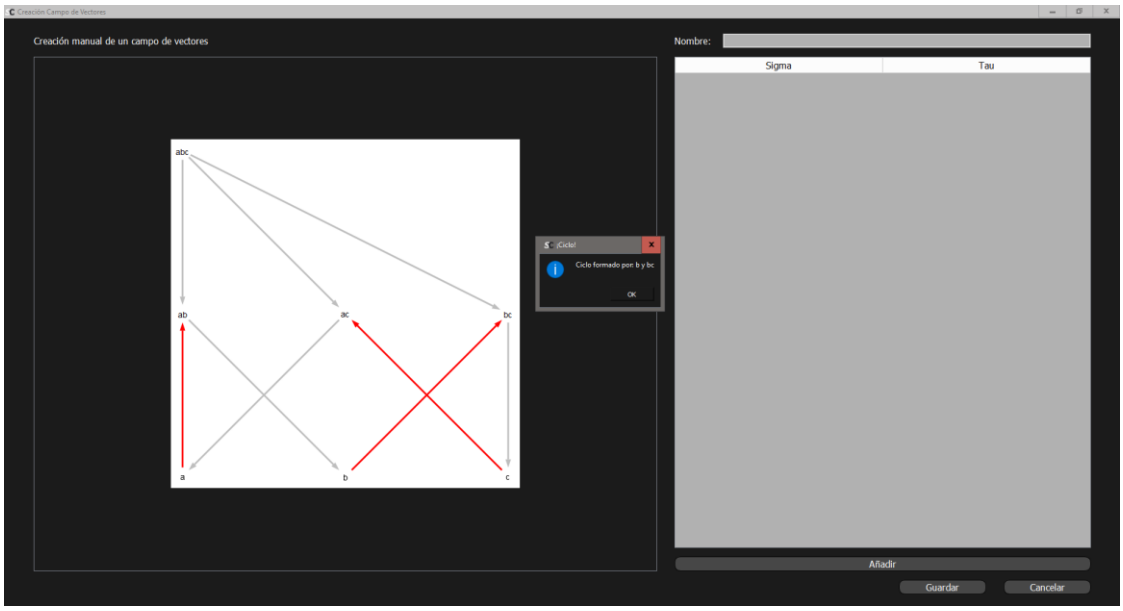
Como se observa en los prototipos hay un submenú con iconos que representa cada operador, estos están divididos por grupos y son:

- Modificar
- Transformar
- Link y star
- Join y cono
- Colapso y expansión
- Campos de vectores: creación manual, automática y listado.

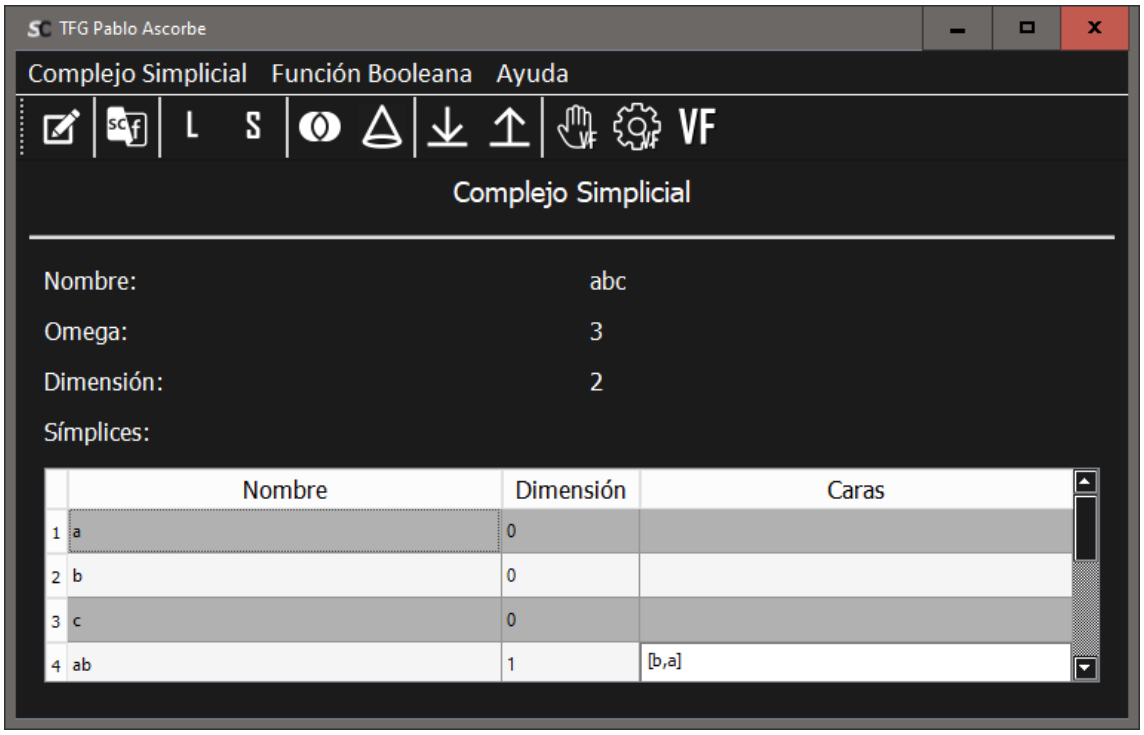
La gran mayoría son muy similares solo que, adaptados a su operador. Internamente el funcionamiento consiste en gestionar los eventos y los datos correctamente para generar los resultados deseados. Siendo la creación de los campos de vectores manuales la única que merece la pena ser mostrada.



Como podemos ver, a la izquierda encontramos el diagrama de Hasse y a la derecha los distintos vectores válidos para añadir al campo. En el momento de encontrar un ciclo este es advertido y los vectores causantes de dicho ciclo son coloreados. Inicialmente, se pensó en colorear todo el ciclo (las flechas que suben y las que bajan), pero no fue posible ya que eso requería una implementación y gestión de los ciclos mucho más compleja que de la que se disponía. Por ello, esta idea terminó siendo descartada.



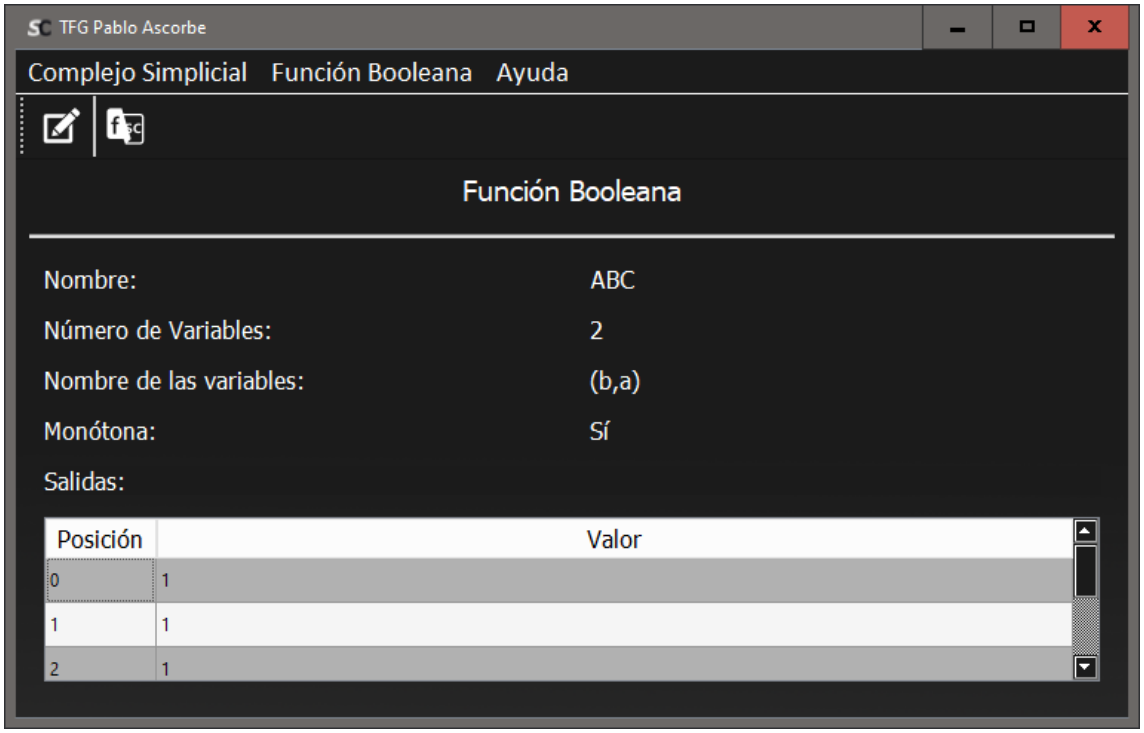
La ventana final del área de trabajo con complejos simpliciales es la siguiente:



Menu_BF

Este menú tiene un comportamiento muy similar al de los complejos simpliciales, salvo que los operadores son los de las funciones booleanas.

La ventana final es la siguiente:



4.3. Pruebas

En este apartado se mostrarán algunos casos destacables en los que se ha mejorado la eficiencia de un método. Se comentan también algunas mediciones que se han llevado a cabo para tomar esas decisiones.

Check_output

Es el método usado para comprobar la monotonía de una función. Las estructuras de datos que se manejan representan un número de datos que es una potencia de dos, luego es muy importante intentar buscar la mayor eficiencia posible.

Se midieron los tiempos para un número de variables considerable. Estos tiempos se midieron para el método antes y después de ser refactorizado y mejorado:

Variables	Sin mejoras	Con mejoras
10	16,191s	0,039s
11	164,239s	0,142s
12	>2.000s	0,65s

Como podemos comprobar las mejoras han supuesto un gran cambio en la eficiencia, mejorando los tiempos de ejecución significativamente. Las mejoras consistieron en comprobar los elementos que ya habían sido calculados y en guardar la posición de la cadena recorrida, de modo que no se comenzase desde el principio en cada iteración.

Persistencia

Se trata de reducir el tamaño de los ficheros en los que se almacenan los objetos. Guardando solamente los identificadores de las caras, en lugar de toda su información, pasamos de 9kb a 2kb con un complejo reducido, incrementando considerablemente con objetos más pesados.

Diccionarios

Hemos trabajado con estos durante todo el desarrollo. Por ello, es interesante que sea eficiente la manera en la que los utilizamos. En el apartado de pruebas unitarias encontramos un método que se encarga de medir tiempos con diccionarios de objetos y con diccionarios de strings. Antes de ningún calculo, parece lógico pensar que las strings serán más eficientes. Pero ¿cuánto más? Ejecutando el código obtenemos lo siguiente:

```
Tiempo para dict mapeado con objetos y 1.000.000 componentes: 1.891635s
Tiempo para dict mapeado con strings y 1.000.000 componentes: 0.8286256999999999s
```

La diferencia es de 1.8/0.8, es decir, un 44.44% más eficiente. Sí que es interesante destacar que esta diferencia se mantiene constante (se han hecho pruebas con 1.000 entradas, con 10.000 y con 1.000.000 y siempre rondaba el 44%-45%).

Comentarios generales

Para terminar con esta sección, en el proyecto hay una colección de pruebas unitarias interesantes, que no solo han servido para comprobar el correcto funcionamiento de los métodos, sino también para encontrar mejoras y medir tiempos. Muchos métodos han sido refactorizados múltiples veces, ya que, aunque funcionaran correctamente podían ser mejorados y simplificados. Otras veces, se encontraban errores para casos muy concretos en los que era necesaria una aproximación totalmente distinta cambiando casi por completo el código.

Como último comentario, en el apartado de la interfaz se utiliza una lista negra para validar los nombres, es decir, permitimos cualquier carácter excepto aquellos que se encuentren en esa lista. Al comienzo, se consideró que el carácter '\' era inocuo, ya que podía ser almacenado sin problemas. Probando casos, se descubrió que, por ejemplo, con el nombre "\..\BooleanFunction\Nombre" se podía persistir un complejo simplicial en el almacén de las funciones booleanas. Así que, se tomaron medidas más estrictas a la hora de validar los nombres.

5. Conclusiones

5.1. Mejoras

Este proyecto está limitado a 300 horas siendo bastantes las posibles mejoras a introducir. Por ello, se deja abierta la puerta a todo aquel que desee ampliarlo y seguir trabajando con él. Recuerdo que es libremente accesible para cualquiera que quiera el código.

Algunas de las posibles mejoras son las siguientes:

- Añadir más operadores, sobre todo las funciones booleanas. En particular, aunque no todas conservan la monotonía, tendría sentido añadir funciones booleanas básicas como AND, OR, NOT, etc.
- Un diseño más estético en la interfaz. Un buen diseñador gráfico podría invertir tiempo en la interfaz, haciéndola más estética y atractiva.
- Portear la aplicación a móvil o a un servicio web cuyo propósito sea similar al de este proyecto.
- Posiblemente todos los métodos puedan ser revisados y medidos para comprobar su eficiencia, aplicando un analizador de código o alguna herramienta con un propósito similar, que detecten algunas mejoras a implementar.

5.2. Conclusión

Personalmente agradezco la oportunidad de haber podido trabajar en este proyecto. No solo me ha permitido ampliar el conocimiento matemático adquirido en el grado, sino que también me ha servido como una prueba real para poner en práctica todos los conocimiento sobre ingeniería del software que he ido recibiendo a lo largo de mi carrera. Ha sido realmente enriquecedor conocer el ámbito de los complejos simpliciales y poder codificar todos los fundamentos matemáticos que con tanto esmero he ido comprendiendo.

También quiero destacar que, aunque muchas de las tecnologías utilizadas para este proyecto fuesen prácticamente nuevas para mí, he podido desenvolverme con soltura al trabajar con ellas gracias a toda la formación adquirida en el grado.

Por último, agradecer a todos mis profesores/as y compañeros/as por estos cuatro fantásticos años y todas las bonitas experiencias que me llevo al haber cursado esta carrera. Y, por supuesto, agradecer a mi tutor Laureano por toda la paciencia y dedicación que ha demostrado tener conmigo. Gracias.

6. Bibliografía

[1] Scoville, N. A. (2019). *Discrete Morse Theory.* Providence, Rhode Island: American Mathematical Society.

[2] Wikipedia. (9 de Febrero de 2022). *Wikipedia Simplicial Complex*. Obtenido de https://en.wikipedia.org/wiki/Simplicial_complex

[3] Dobson, S. (2017). *simplicial.* Obtenido de https://simplicial.readthedocs.io/en/stable/

[4] Sharma, A. (10 de Abril de 2020). *Datacamp.* Obtenido de https://www.datacamp.com/tutorial/docstrings-python

[5] Company, T. Q. (2022). *riverbankcomputing.* Obtenido de https://www.riverbankcomputing.com/static/Docs/PyQt5/