Universidad Nacional Autónoma de México

Posgrado en Ciencias Matemáticas
Facultad de Ciencias

Aplicaciones de Cálculo Simbólico y LISP en Geometría Cuántica

T E S I S

QUE PARA OBTENER EL GRADO ACADÉMICO DE:
MAESTRO EN CIENCIAS

PRESENTA:
CÉSAR CARREÓN OTAÑEZ

DIRECTOR DE TESIS: Dr. MICHO ĐURĐEVICH

2012

Захвалница

Посвећено моме професору Мићи, уз захвалност на истинској подршци у тешкоћама, пружању мира и сигурности у току рада, подели искуства, и веровању у мене. Као и на заразном ентузијазму за математику, увођењу у њен фасцинантни свет, и поврх свега на подсећању зашто сам уопште одлучио да је студирам. Хвала, јер као личност и професионисте сам порастао под његовим надзором.

Gracias a mi familia por apoyarme y soportarme durante este proceso, mi mamá, papá y hermanos: Marisol, David y Ana.

A mis amigas que creen en mi y me han visto caer y crecer durante este tiempo: Alejandra García, Adriana, Alma, Lety. A mis amigos de la facultad que siguen ahí o allá: Ofelia, Mónica, Andrés García, Dany, Manuel, Gus y Javier.

A los que cultive en la maestría: Chio, Brenda, Kernel, Irma, Jorge, Magda, César 6708.

A quienes a lo largo de años siguen conmigo, mis almas afines: Judith, Lizbeth, Jane.

A mi alma máter, la Universidad Nacional Autónoma de México por quien soy al día de hoy.

Índice general

Prefacio

El presente trabajo surge por la inquietud de abordar dos líneas de investigación; la primera hace referencia al software libre, en específico Maxima y Lisp, la segunda se enfoca al campo de la Geometría Cuántica.

En el medio algebrista existen operadores que involucran enormes cantidades de cuentas; de aquí emana el objetivo principal de la tesis: tomar algunos de esos operadores y programarlos.

La Geometría Cuántica requiere efectuar cálculos que se prestan a manipulación simbólica; es ahí donde la programación que ofrece el lenguaje en Lisp se acopla perfectamente. Lo cual es una motivación natural para explorar qué se puede hacer utilizando ambas herramientas. Así, al tener dichas implementaciones éstas se pueden usar como métodos de prueba, en investigación o para corroborar con ejemplos concretos lo que se deduce de teoremas y resultados.

Maxima pertenece a una ideolgía de software libre, para su uso y comercialización sin fines de lucro y de código abierto, entre otras características. Lisp por su parte es un lenguaje que utiliza listas como sintaxis principal, con características que lo hacen especial respecto a otros lenguajes, como el hecho de usar funciones como argumentos, los predicados y Macros.

Enteramente programado en Lisp, Maxima hereda su estructura como una poderosa herramienta para el cálculo simbólico, con una ambientación pensada para uso matemático, resuelve una amplia gama de problemas donde se involucran grandes cantidades de cuentas. Con la tesis actual se pretende abarcar un campo casi nulo dentro de sus implementaciones, la Geometría Cuántica.

Con la ayuda del Dr. Durdevich [19] y lo que ha desarrollado por años en el área, surge un conjunto de rutinas enfocadas a problemas específicos y su implementación en Teoría de Grupos Finitos equipados con un Cálculo Diferencial. Este cálculo diferencial es necesariamente de naturaleza cuántica, ya que los espacios discretos no permiten un cálculo diferencial estándar no-trivial.

El primer capítulo muestra el software de cálculo simbólico MAXIMA, su historia, sintaxis y lo presentar como una herramienta útil para realizar cálculos de tipo algebraico y numérico. Con ejemplos se ilustran algunas de las funciones que tiene y como se utiliza.

En el segundo capítulo se describe el lenguaje de programación LISP, usado en principio en el área de Inteligencia Artificial llega a ser la base del software simbólico: MATHEMATICA y MAXIMA. Debido a la programación de tipo declarativo bajo el que está realizado, LISP armoniza con cálculos lógicos y totalmente simbólicos, así se consiguen hacer de forma eficiente y rápida algoritmos que pueden reproducir las operaciones algebraicas que en otros lenguajes costaría más o simplemente no podrían ser realizados.

El tercer capítulo tiene de forma introductora un panorama general de la GEOMETRÍA CUÁNTICA y cómo ésta sirve de inspiración para desarrollar algoritmos alrededor de un muy interesante operador de trenza σ, que aparece de manera natural en el contexto de cálculo diferencial cuántico sobre grupos finitos (en general, sobre grupos cuánticos). Se muestra como este operador permite introducirse en el campo de la GEOMETRÍA TRENZADA. Aquí se detallan los ejemplos que se estudiaron y el enfoque que se abordó para desarrollarlos.

El cuarto y último capítulo contiene de forma concisa las principales rutinas que se hicieron. El código está desglosado con la sintaxis para ser usado, con ejemplos y los resultados obtenidos.

En la sección de apéndices se recapitulan con precisión las transformaciones de simetría que se usaron, código secundario y definiciones que complementan el trabajo. También, se presenta una demostración del Teorema de Gelfand-Naimark, que es la base de la ideología en la Geometría Cuántica.

Capítulo 1

Cálculo Simbólico: Maxima

1.1. Introducción

A lo largo del desarrollo de la computación han surgido corrientes enfocadas a la distribución de software de manera libre, con código abierto.

Maxima se encuentra dentro de los paquetes de software matemático que tiene ese distintivo, Scilab, Octave, GeoGebra, LaTeX, R, entre otros, surgen como una opción a lo que ofrece el mercado en cuanto a necesidades muy específicas y confiables para realizar cálculos.

En la primer parte del trabajo se dará una descripción general de las funciones, comandos y sintaxis que tienen Maxima y Lisp, mostrando en cada caso su utilidad y la forma en que ambos se relacionan.

Maxima es un sistema para la manipulación de expresiones simbólicas y numéricas, entre las aplicaciones que tiene cuenta con diferenciación, integración, expansión en series de Taylor, transformada de Laplace, ecuaciones diferenciales ordinarias, sistemas de ecuaciones lineales, vectores, matrices y tensores.

Dentro de las funciones que realiza cuenta con implementación de gráficas en dos y tres dimensiones, así como un alto grado de precisión usando fracciones exactas y representación con aritmética de punto flotante.

Maxima está basado en un sistema de álgebra computacional desarrollado en la década de los 60 en el Instituto Tecnológico de Massachusetts (MIT) y es portable para los sistemas operativos, Windows, Linux y MacOSX, entre otros.

1.1.1. Macsyma

Maxima evoluciona de un programa llamado Macsyma, que tiene por significado MAC's SYmbolic MAnipulator y que fue el primer sistema simbólico

matemático escrito en LISP por Joel Moses, dentro del "Proyect MAC" (Multiple Access Computers) creado por J C R Licklider dentro del MIT, el cual se desarrolló a lo largo de 1968 a 1982.

MACSYMA crece en medio de severos problemas de licencia y peleas de poder entre los creadores y administradores. En 1992 se crea "Macsyma Inc." por George Carrette y Russell Noftsker, continuando con su desarrollo hasta volverlo, en ese tiempo, un serio competidor para Mathematica y Maple.

"La rama Maxima de Macsyma fue mantenida por William Schelter desde 1982 hasta su muerte en 2001. En 1998 él obtuvo permiso para liberar el código fuente bajo la licencia pública general (GPL[1]) de GNU", véase [20].

1.1.2. wxMaxima

WXMAXIMA [21] es una plataforma GUI (Graphical User Interface) para el sistema de álgebra computacional Maxima, basado en wxWidgets.

La cual incluye las siguientes características:

- Despliega un formato simbólico matemático.
- Menú con distintos comandos y funciones en un panel superior.
- Comandos que requieren más de un argumento pueden ser escritos con diálogos sin necesidad de recordar la sintaxis completa.
- Crea documentos que pueden ser mezclados con cálculos para ser salvados y editados posteriormente.
- Cuenta con animaciones simples.
- Modo predictivo para completar funciones y variables, así como plantillas.
- WXMAXIMA está creado bajo licencia GPL2.

Con WXMAXIMA se tiene un manejo más estético y amigable de las funciones que tiene MAXIMA en cuanto a formato de ventana de salida.

La siguientes figuras muestran dos ambientes para MAXIMA sobre emacs (1.1) y el entorno creado por WXMAXIMA (1.2) en Mac.
Como se puede observar hay una diferencia notable en cuanto el formato de escritura, colores e imagen.

[1] GNU Public License.

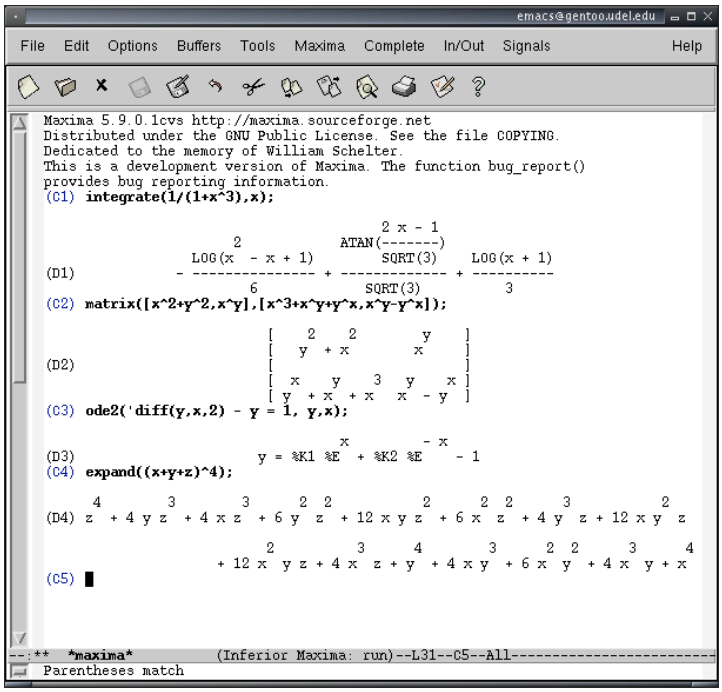


Figura 1.1: Screenshot: `Maxima` en emacs.

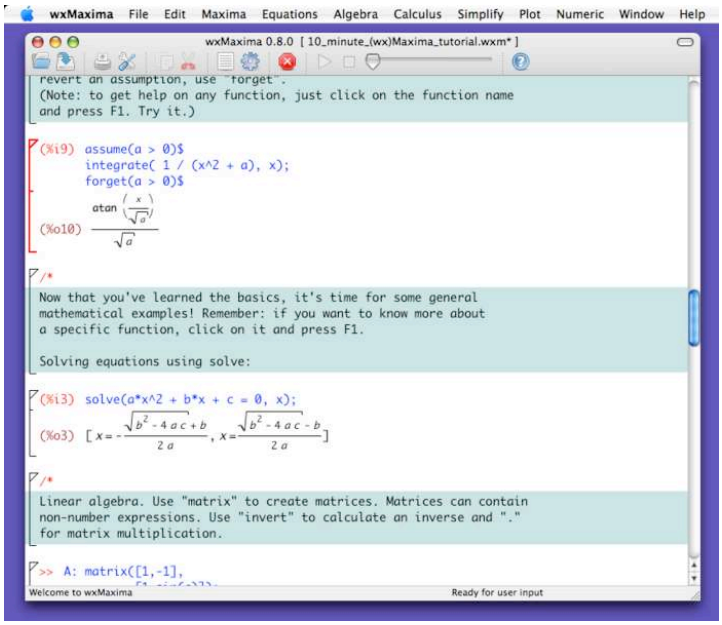


Figura 1.2: Screenshot: `wxMaxima` en Mac.

1.2. Entorno

Se comenzará con una descripción global de MAXIMA, funciones, estructura y características propias, para dar a conocer el ambiente de trabajo.

Sin ser un manual, esta sección tiene el objetivo de dar una visión muy general de lo que es MAXIMA y algunos atributos que tiene como software de cálculo simbólico.

MAXIMA tiene una manera peculiar de manejarse comparado con otros paquetes de software matemático, ello lo hereda de LISP (el cual se verá con detalle posteriormente) ya que está escrito sobre él y con algunos comandos se conserva el formato de listas.

Al iniciar con el programa se tiene la siguiente sintaxis en la ventana de comandos: `(%i1)`

Lo cual indica que MAXIMA espera el "primer **i**nput". Si se ejecuta una instrucción, por ejemplo:

```
(%i1) 2 + 2;
```

Se tiene de salida:

```
(%o1) 4
```

Lo que se lee "primer **o**utput" 4.

Conforme se trabaja se van enumerando los inputs y outputs para tener un control en memoria de lo ejecutado y si se desea llamarlos, posteriormente, poder hacerlo.

Cada instrucción que quiera ser ejecutada tiene que ir al final con ";" seguido de un ENTER, de otra forma es sólo texto lo que resulta. Si no se quiere ver en pantalla lo que se despliega como output, se coloca $ terminando la línea.

Input	Output
`(%i) x:1;`	`(%o) 1`
`(%i) x:1$`	

Cuando se quiere eliminar una variable se utiliza el comando KILL para borrar de memoria su valor, por ejemplo:

Input	Output
`(%i) kill(x);`	`(%o)` *done*

en caso de hacer que todas las asignaciones que se hayan declarado previamente se borren de memoria, basta dar un KILL(ALL).

Los comentarios se escriben con la siguiente sintaxis: /* `comentario` */

1.2.1. Funciones y Comandos Básicos

MAXIMA tiene, por ser un lenguaje para matemáticas, las funciones estándar de la materia.

- Suma "`+`", resta "`- `", multiplicación "`*`", división "`/`", potencia "`^`".
- Raíz cuadrada "`sqrt(x)`", seno "`sin(x)`", coseno "`cos(x)`", tangente "`tan(x)`", etc.

Al ser simbólico el ambiente se tiene en mente una estructura apropiada, ya que el sistema trabaja diferente a lo que uno podría estar acostumbrado. Si por ejemplo se escribiera lo siguiente:

```
(%i) 4 / 3;
```

el output resultante estaría dado por:

(%o) $\frac{4}{3}$

Para poder ver el resultado en un número decimal se tiene el comando BFLOAT que convierte su argumento a un flotante:

```
(%i) bfloat(4/3);
(%o) 1.33333333333b0
```

Lo mismo sucede con las funciones análiticas que están implementadas ya.

Para constantes comúnmente usadas e, π, γ, ϕ MAXIMA reserva caracteres especiales y poder manipularlas, antecediendo el signo %:

`(%i) bfloat(%pi);`	`(%o)` $3.1415926535897 93b0$
`(%i) bfloat(%e);`	`(%o)` $2.7182818284590 45b0$
`(%i) bfloat(%gamma);`	`(%o)` $5.7721566490153 29b-1$
`(%i) bfloat(%phi);`	`(%o)` $1.6180339887498 95b0$

MAXIMA por estar orientado a lenguaje simbólico tiene funciones "extras" para su uso: EXPAND, RATSIMP son algunas de ellas que cumplen con ese cometido.

La tabla de abajo muestra el uso de las mismas y su sintaxis.

Input	output
`(%i)` $(x^2-x-1)*(a+1);$	`(%o)` $(a+1)(x^2-x-1)$
`(%i) expand`$((x^2-x-1)*(a+1));$	`(%o)` $ax^2+x^2-ax-x-a-1$
`(%i) ratsimp`$(ax^2+x^2-ax-x-a-1);$	`(%o)` $(a+1)x^2+(-a-1)x-a-1$

Como se puede observar, mientras que EXPAND sólamente desarrolla los términos de alguna expresión, RATSIMP la simplifica.

La función que realiza el proceso inverso se llama FACTOR y factoriza las expresiones que tiene por argumento:

Input	Output
`(%i) factor(`x^6-1`);`	`(%o)` $(x-1)(x+1)(x^2-x+1)(x^2+x+1)$

Lo siguiente es hacer notar la diferencia entre los caracteres "=","**:**" y ":=", ya que cada uno de ellos se utiliza en distintos contextos.

	Descripción	Ejemplo
=	La igualdad funciona como símbolo exclusivamente, no tiene propiedad de asignar o pasar valores.	`(%i)` $x = 1$; `(%o)` $x = 1$ Aquí sólo se tiene la expresión simbólica.
:	Los dos puntos tienen la función de asignar valores.	`(%i)` $x : 1$; `(%o)` 1 Aquí x tiene asignado el valor de 1 en memoria.
:=	Los dos puntos con el igual sirve para definir funciones.	`(%i)` $f(x)$:=`x^3;` `(%o)` $f(x) := x^3$

Una función interesante dentro del programa es BLOCK, la cual ejecuta una serie de sentencias dentro de un bloque con sus propias variables (locales), aún cuando éstas hayan sido declaradas previamente, dando como resultado la evaluación de la última sentencia.

Input		Output
`(%i)`	`x:-1;`	`(%o)` -1
	`y:1;`	`(%o)` 1
	`block([x,y],x:2,y:5,x*y);`	`(%o)` 10
	`x*y;`	`(%o)` -1

Como lo muestra la tabla con el ejemplo para BLOCK, inicialmente las variables x, y son declaradas con 1 y -1, dentro del bloque se vuelven a definir y se les asignan otros valores y se opera con ellos. El resultado del bloque es 10 la operación definida dentro, mientras que en el último renglón es -1 operación resultante de los valores que se tenían desde el inicio.

1.2.2. Listas

Al ser programado sobre LISP, MAXIMA hereda en algunas instrucciones el formato de listas. Se verán las funciones más comunes con ejemplos.

Lo primero es ver cómo se declara una lista:

Input	Output
`(%i) X : [1,2,3,4];`	`(%o)` $[1,2,3,4]$
`(%i) Y : [5,6,7,8];`	`(%o)` $[5,6,7,8]$

Al manejar listas en las operaciones aritméticas básicas (+,-,*,/) se hace entrada a entrada:

`(%i)` X + Y;	`(%o)` $[6, 8, 10, 12]$
`(%i)` X - Y;	`(%o)` $[-4, -4, -4, -4]$
`(%i)` X * Y;	`(%o)` $[5, 12, 21, 32]$
`(%i)` X / Y;	`(%o)` $[\frac{1}{5}, \frac{1}{3}, \frac{3}{7}, \frac{1}{2}]$

La siguiente tabla contiene algunos de los comando que se manejan con listas.

Función	Descripción	Ejemplo
`append`	Une listas.	`(%i) append(X,Y);` `(%o)` $[1, 2, 3, 4, 5, 6, 7, 8]$
`cons`	Une un elemento en el primer sitio de una lista ya creada.	`(%i) cons(0,X);` `(%o)` $[0, 1, 2, 3, 4]$
`first`	Devuelve el primer elemento de la lista	`(%i) first(X);` `(%o) 1`
`last`	Devuelve el último elemento de la lista	`(%i) last(X);` `(%o) 4`
`makelist`	Crea una lista a partir de una expresión o sentencia dada.	`(%i) makelist(2*i,i,1,4);` `(%o)` $[2, 4, 6, 8]$
`length`	Devuelve la longitud de una lista.	`(%i) length(X);` `(%o) 4`
`listp`	Arroja *true* si el argumento es una lista, *false* en caso contrario.	`(%i) listp(X);` `(%o)` *true*
`reverse`	Invierte el orden de los elementos de una lista.	`(%i) reverse(X);` `(%o)` $[4, 3, 2, 1]$
`create_list`	Crea una lista evaluando una expresión o sentencia sobre una lista de listas. `(%i) create_list ([i,j],i,[1,2],j,[a,b,c]);` `(%o)` $[\ [1,a],[1,b],[1,c],[2,a],[2,b],[2,c]\ ]$	

Una función peculiar es MAP, la cual aplica alguna sentencia o función sobre los elementos de una lista. APPLY por su parte tiene como argumentos una función y argumentos propios los cuales se van a evaluar en la función.

Input	Output
`(%i) map(sin,X);`	`(%o)` $[sin(1), sin(2), sin(3), sin(4)]$
`(%i) apply(min,X);`	`(%o)` 1

1.2.3. Álgebra Lineal y Matrices

Lo siguiente será dar una descripción de las implementaciones que tiene MAXIMA en el área de álgebra lineal y matrices, a manera de introducción, ya que se utilizaran algunos de estos comandos y funciones para el módulo que se creará como objetivo del presente trabajo.

Una matriz se define como lo indica la siguiente tabla:

Nombre	Input	Output
`matrix`	`A: matrix([1,2,3],[4,5,6],[7,8,9]);`	$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$
`matrix`	`B: matrix([a,b,c],[d,e,f],[g,h,i]);`	$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$

Las operaciones básicas (+,-,*,/) entre matrices se hacen entrada a entrada. El producto entre matrices se define por "·"

	Input	Ouput
$+$	`(%i) A+B;`	`(%o)` $\begin{bmatrix} a+1 & b+2 & c+3 \\ d+4 & e+5 & f+6 \\ g+7 & h+8 & i+9 \end{bmatrix}$
$-$	`(%i) A-B;`	`(%o)` $\begin{bmatrix} 1-a & 2-b & 3-c \\ 4-d & 5-e & 6-f \\ 7-g & 8-h & 9-i \end{bmatrix}$
$*$	`(%i) A*B;`	`(%o)` $\begin{bmatrix} a & 2\,b & 3\,c \\ 4\,d & 5\,e & 6\,f \\ 7\,g & 8\,h & 9\,i \end{bmatrix}$
$/$	`(%i) A/B;`	`(%o)` $\begin{bmatrix} \frac{1}{a} & \frac{2}{b} & \frac{3}{c} \\ \frac{4}{d} & \frac{5}{e} & \frac{6}{f} \\ \frac{7}{g} & \frac{8}{h} & \frac{9}{i} \end{bmatrix}$

$\cdot$	`(%i) A·B;`
`(%o)`	$\begin{bmatrix} 3\,g+2\,d+a & 3\,h+2\,e+b & 3\,i+2\,f+c \\ 6\,g+5\,d+4\,a & 6\,h+5\,e+4\,b & 6\,i+5\,f+4\,c \\ 9\,g+8\,d+7\,a & 9\,h+8\,e+7\,b & 9\,i+8\,f+7\,c \end{bmatrix}$

Algunas operaciones básicas entre matrices se muestran a continuación[2].

[2] `mattrace` requiere cargar el paquete `nchrpl`.

Función	Descripción	Ejemplo
`determinant`	Calcula el determinante de una matriz	`(%i) determinant(A);` `(%o)` 0
`col`	Extrae la i-ésima columna de una matriz.	`(%i) col(A,2);` `(%o)` $\begin{bmatrix}2\\5\\8\end{bmatrix}$
`row`	Extrae el j-ésimo renglón	`(%i) row(A,3);` `(%o)` $\begin{bmatrix}7 & 8 & 9\end{bmatrix}$
`mattrace`	Devuelve la suma de los valores de la traza de una matriz.	`(%i) mattrace(A);` `(%o) 15`
`minor`	Devuelve el menor de algún elemento (i,j) de la matriz.	`(%i) minor(A,1,1);` `(%o)` $\begin{bmatrix}5 & 6\\8 & 9\end{bmatrix}$
`submatrix`	Devuelve una matriz sin el i-ésimo renglón ni la j-ésima columna	`(%i) submatrix(1,A,3);` `(%o)` $\begin{bmatrix}4 & 5\\7 & 8\end{bmatrix}$
`matrixp`	Devuelve *true* si el argumento es una matriz, *false* en caso contrario	`(%i) matrixp(A);` `(%o)` *true*
`transpose`	Devuelve la transpuesta de una matriz.	`(%i) transpose(A);` `(%o)` $\begin{bmatrix}1 & 4 & 7\\2 & 5 & 8\\3 & 6 & 9\end{bmatrix}$

`diagmatrix`	Crea una matriz con elementos x en su diagonal, éstos pueden ser valores, listas, matrices. `(%i) diagmatrix(4,[1,2]);` `(%o)` $\begin{bmatrix}[1,2] & 0 & 0 & 0\\0 & [1,2] & 0 & 0\\0 & 0 & [1,2] & 0\\0 & 0 & 0 & [1,2]\end{bmatrix}$
`eigenvalues`	Devuelve los valores propios del argumento (matriz). `(%i) eigenvalues(A);` `(%o)` $[[-\frac{3\sqrt{33}-15}{2}, \frac{3\sqrt{33}+15}{2}, 0], [1,1,1]]$
`invert`	Devuelve la inversa de una matriz. `(%i) invert(matrix([1,-1],[2,1]));` `(%o)` $\begin{bmatrix}\frac{1}{3} & \frac{1}{3}\\-\frac{2}{3} & \frac{1}{3}\end{bmatrix}$

1.2.4. Diferenciación e Integración

Este pequeño apartado que corresponde a diferenciación e integración tiene como objetivo mostrar los atractivos que tiene MAXIMA como lenguaje de cálculo simbólico y por lo que se distingue de todo el software matemático.

La instrucción para derivar DIFF requiere como argumentos una función y la variable sobre la cual derivar.

`(%i) diff(x^8-4*x^6+2*x^2,x);`	`(%o)` $8x^7 - 24x^5 + 4x$
`(%i) diff(cos(x*y)-sin(x),y);`	`(%o)` $-xsin(xy)$
`(%i) diff(x*y*z^3+tan(x*z),z);`	`(%o)` $xsec(xz)^2 + 3xyz^2$

Cuando se busca una derivada de algún order superior basta con agregar un tercer argumento indicando cuál es el grado deseado.

`(%i) diff(x^8-4*x^6+2*x^2,x,2);`	`(%o)` $56x^6 - 120x^4 + 4$
`(%i) diff(cos(x*y)-sin(x),y,2);`	`(%o)` $-x^2cos(xy)$
`(%i) diff(x*y*z^3+tan(x*z),z,2);`	`(%o)` $2x^2sec(xz)^2tan(xz)$ $+6xyz$

Si se desea no evaluar las funciones que DIFF tiene como argumento y trabajar simbólicamente basta poner una coma simple antes del comando.

`(%i) 'diff(x^8-4*x^6+2*x^2,x);`	`(%o)` $\frac{d}{dx}\left(x^8 - 4x^6 + 2x^2\right)$
`(%i) 'diff(cos(x*y)-sin(x),y);`	`(%o)` $\frac{d}{dy}\left(cos(xy) - sin(x)\right)$
`(%i) 'diff(x*y*z^3+tan(x*z),z);`	`(%o)` $\frac{d}{dz}\left(tan(xz) + xyz^3\right)$

El manejo de las derivadas de la forma en que se presentó brinda algunas ventajas, por ejemplo si se quisiera resolver la siguiente ecuación:

$$y''(x) + 3y'(x) + x + y(x) = 0$$

El código en MAXIMA está dado por:

`(%i) 'diff(y,x,2) + 3*'diff(y,x)+x + y;`	`(%o)` $\frac{d^2}{dx^2}y + 3\left(\frac{d}{dx}y\right) + y(x) + x$

Para resolver el sistema se utiliza ODE2[3] que tiene como parámetros, la ecuación, la variable dependiente e independiente:

`(%i) ode2(%,y,x);`
`(%o)` $y = \%k1\, e^{\frac{\left(\sqrt{5}-3\right)x}{2}} + \%k2\, e^{\frac{\left(-\sqrt{5}-3\right)x}{2}} - x + 3$

[3]De primer orden o segundo.

TAYLOR es un comando que da como resultado la expansión en serie de Taylor de alguna función alrededor de un punto y con un cierto grado.

`(%i) taylor(cos(x),x,0,8);`	`(%o)` $1-\frac{x^2}{2}+\frac{x^4}{24}-\frac{x^6}{720}+\frac{x^8}{40320}+...$
`(%i) taylor(1/x,x,1,4);`	`(%o)` $1-(x-1)+(x-1)^2-(x-1)^3+(x-1)^4+...$

Para obtener algunos operadores diferenciales como el gradiente o la divergencia, se requiere cargar en memoria el paquete "vect".

`(%i) load("vect");`
`(%o)` */Applications/ ... /maxima/5.19.2/share/vector/vect.mac*

Así, dada una función se pueden calcular dichas operaciones las cuales se pueden visualizar con el comando EXPRESS que transforma los nombres de los operadores diferenciales en expresiones que contienen derivadas parciales.

Luego, para evaluar los resultados se usa EV con el argumento DIFF para que sean mostradas de manera explícita.

`(%i) grad(x^3+y^3+z^3);`	`(%o)` $grad(z^3+y^3+x^3)$
`(%i) express(%);`	`(%o)` $[\frac{d}{dx}(z^3+y^3+x^3),\frac{d}{dy}(z^3+y^3+x^3),\frac{d}{dz}(z^3+y^3+x^3)]$
`(%i) ev(%,diff);`	`(%o)` $[3x^2,3y^2,3z^2]$
`(%i) div([x^3,y^3,z^3]);`	`(%o)` $div([x^3,y^3,z^3])$
`(%i) express(%);`	`(%o)` $\frac{d}{dz}z^3+\frac{d}{dy}y^3+\frac{d}{dx}x^3$
`(%i) ev(%,diff);`	`(%o)` $3z^2+3y^2+3x^2$
`(%i) laplacian([x^3,y^3,z^3]);`	`(%o)` $laplacian([x^3,y^3,z^3])$
`(%i) express(%);`	`(%o)` $\frac{d^2}{dz^2}[x^3,y^3,z^3]+\frac{d^2}{dy^2}[x^3,y^3,z^3]+\frac{d^2}{dx^2}[x^3,y^3,z^3]$
`(%i) ev(%,diff);`	`(%o)` $[6x,6y,6z]$

En cuanto a integración MAXIMA cuenta con el comando INTEGRATE que devuelve la integral de una función, está orientado al cálculo de funciones elementales y analíticas.

INTEGRATE tiene como argumentos la función a integrar y la variable con respecto a la cual se desea operar.

`(%i) integrate(x^5+x^4-6,x);`	(%o) $\frac{x^6}{6}+\frac{x^5}{5}-6x$
`(%i) integrate(cos(y)^3,y);`	(%o) $sin(y)-\frac{sin(y)^3}{3}$
`(%i) integrate(z*%e^(3*z^2),z);`	(%o) $\frac{\%e^{3z^2}}{6}$

Si se quiere evaluar la integral resultante en un intervalo $[a, b]$, se agregan como argumentos los puntos extremos del intervalo.

`(%i) integrate(x^5+x^4-6,x,0,1);`	(%o) $-\frac{169}{30}$
`(%i) integrate(cos(y)^3,y,0,%pi/2);`	(%o) $\frac{2}{3}$
`(%i) integrate(z*%e^(3*z^2),z,0,1);`	(%o) $\frac{\%e^3}{6}-\frac{1}{6}$

Igual que en el caso de diferenciación, si se opta por querer trabajar de manera simbólica las integrales, se antecede una coma simple a la instrucción:

`(%i) integrate(x^5+x^4-6,x,0,1);`	(%o) $\int_0^1 x^5+x^4-6dx$
`(%i) integrate(cos(y)^3,y,0,%pi/2);`	(%o) $\int_0^{\frac{\%pi}{2}} cos(y)^3 dy$
`(%i) integrate(z*%e^(3*z^2),z,0,1);`	(%o) $\int_0^1 z\,\%e^{3z^2} dz$

CHANGEVAR realiza un cambio de variable para una función $f(u, x) = 0$ en términos de la nueva variable u por x.

Si se tiene $g : [a, b] \to \mathbb{R}$ de clase $\mathcal{C}^1$ en $[a, b]$ y $f : \mathbb{R} \to \mathbb{R}$ continua, entonces el teorema de cambio de variable asegura:

$$\int_a^b (f \circ g) \cdot g' = \int_{g(a)}^{g(b)} f$$

tomando

$$f(y) = e^{\sqrt{ay}},\ a > 0; \qquad y \in [0, 4]$$

$$g(z) = \frac{z^2}{a} \ \Rightarrow\ g'(z) = 2\frac{z}{a}$$

el cambio de variable es:

$$\int_0^4 e^{\sqrt{ay}} dy = \frac{2}{a} \int_0^{-2\sqrt{a}} z e^{|z|} dz$$

En MAXIMA los comandos son los siguientes:

`(%i) assume (a>0);`
`(%i) 'integrate (%e**sqrt(a*y),y,0,4);`
`(%o)` $\int_0^4 e^{\sqrt{a}}\sqrt{y}dy$
`(%i) changevar(%,y-z^2/a,z,y);`
`(%o)` $-\frac{2\int_{-2\sqrt{a}}^{0} z\, e^{\lvert z\rvert}dz}{a}$

Como tema ligado a la integración está la de sumas, las cuales son dadas por dos tipos, las finitas e infinitas.

`(%i) sum(i^2,i,1,100);`	`(%o)` 338350
`(%i) sum(1/x^2,x,1,inf);`	`(%o)` $\sum_{x=1}^{inf}\frac{1}{x^2}$

Si se desea ver el valor de la suma, en caso de converger, se requiere de la instrucción SIMPSUM: ,

`(%i) sum(1/x^2,x,1,inf),simpsum;`	`(%o)` $\frac{\pi^2}{6}$
`(%i) sum(x^2,x,1,inf),simpsum;`	`(%o)` *sum: sum is divergent. – an error. To debug this try debugmode(true);*

Como se observa, cuando diverge la suma aparece un "error" con su correspondiente aviso.

MAXIMA posee cálculo para integrales dobles del estilo de (1.1) con el paquete DBLINT que previo uso se carga en memoria.

$$\int_a^b \int_{r(x)}^{s(x)} f(x,y)dxdy \tag{1.1}$$

De igual forma, posee una biblioteca llamada QUADPACK la cual contiene un conjunto de funciones y rutinas especializadas en el cálculo numérico de integrales definidas por una variable. Está formada por rutinas de un trabajo conjunto de Piessens, E. de Doncker, C. Ueberhuber y D. Kahaner.

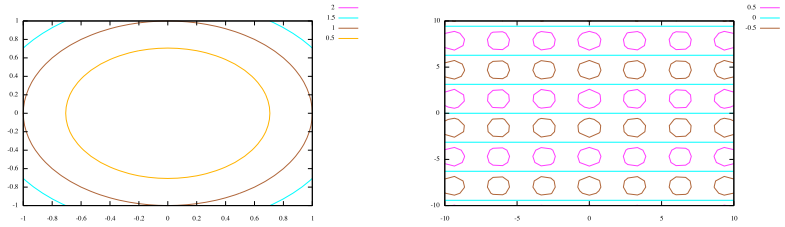
Para más detalles de lo anterior se puede consultar la página de MAXIMA [20] donde se encuentran manuales con las especificaciones de cada uno.

1.2.5. Gráficos

Entre las opciones que `Maxima` tiene se encuentran las implementaciones para gráficos, los hay en 2D y 3D. Se dará una descripción de los principales comandos y algunos ejemplos.

`contour_plot`: Grafica las curvas de nivel de una expresión sobre un dominio, Figura 1.3.

```
(%i) contour_plot(x^2 + y^2, [x, -1, 1], [y, -1, 1]);
(%i) contour_plot(sin(y)*cos(x)^2,[x, -10, 10],[y, -10, 10]);
```

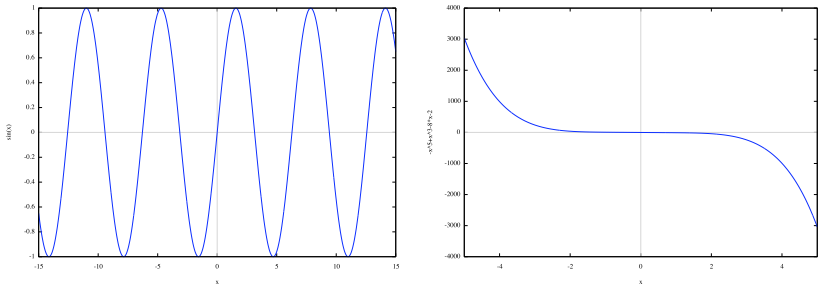


(a) x^2+y^2 (b) sin(y)*cos(x)^2

Figura 1.3: `contour_plot`

`plot2d`: Grafica una función para un cierto rango de las variables que tenga por argumento, Figura 1.4.

```
(%i) plot2d(sin(x), [x,-15, 15]);
(%i) plot2d(-x^5+x^3-8*x-2,[x,-5,5]);
```



(a) sin(x) (b) -x^5+x^3-8*x-2

Figura 1.4: `plot2d`

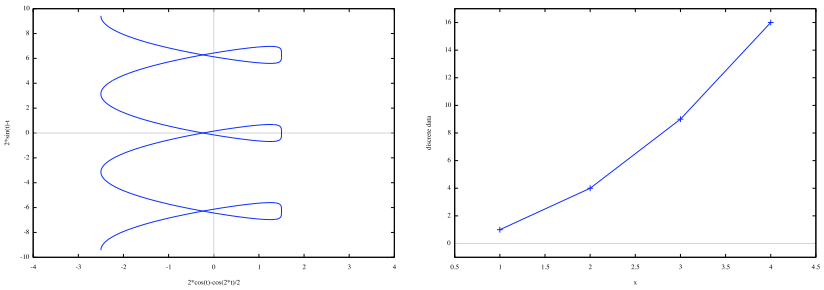
`plot2d` tiene dos opciones de graficado, la forma *parametric* (paramétrica) y *discrete* (discreta). La figura 1.5 (a) muestra un ejemplo del formato paramétrico y en 1.5 (b) uno del formato discreto.

```
(%i) plot2d([parametric,2*cos(t)-1/2*cos(2*t),
```

```
2*sin(t)-1/2*(2*t),[t,-%pi*3,%pi*3],[nticks,800]],[x,-4,4]);
```

```
xx : [1,2,3,4]$
yy : [1,4,9,16]$
plot2d([discrete,xx,yy],[style,points],[x,0.5,4.5],[y,-1,17]);
```

Las Figuras 1.6 y 1.7 muestran algunos ejemplos para el comando `plot3d`.



(a) *parametric* (b) *discrete*

Figura 1.5: `plot2d`

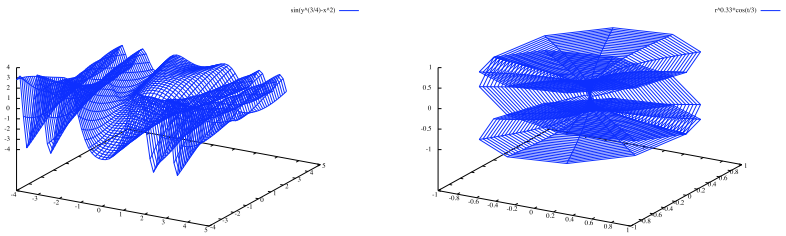


Figura 1.6: `plot3d`

Para llamar los gráficos desde Mac hay que agregar un argumento extra [GNUPLOT_TERM,AQUA] que hace el llamado a la terminal propia del sistema operativo.

Para más detalles de las funciones y comandos que se presentaron, se pueden consultar las páginas de MAXIMA [20] y WXMAXIMA [21] para encontrar manuales completos de ellas.

Ya que el presente trabajo no tiene como objetivo detallar minuciosamente las funciones de MAXIMA si no presentar un panorama general de ellas, para

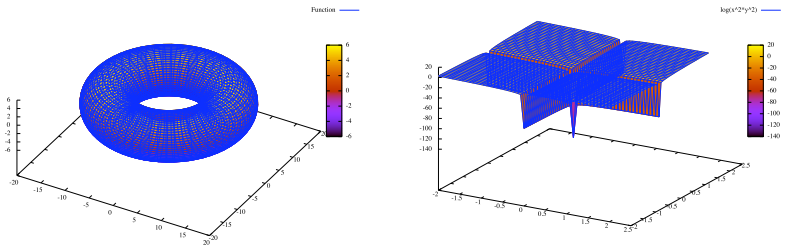


Figura 1.7: `plot3d`

luego enfocarse en el módulo de geometría cuántica, se muestraron los comandos comunes para introducir al lector al ambiente.

1.3. Álgebra Computacional

El área del Álgebra Computacional y Simbólica, (SAC por sus siglas en inglés) tiene como objetivo automatizar de forma computacional procesos matemáticos de todo tipo. Resultando en sistemas computacionales, comerciales y experimentales sumamente poderosos como herramientas para científicos, ingenieros y profesores. El SAC usualmente combina matemáticas con avanzadas técnicas computacionales. Para referencias e historia puede consultarse, [2], [5], [4] y [13].

La siguiente es una lista tomada de [20] haciendo referencia al software existente en el área del Álgebra Computacional de uso libre o de código abierto.

- AXIOM. Un sistema de Álgebra Computacional de propósito general. Útil para hacer matemáticas por computadora para investigación y desarrollo de algoritmos matemáticos. Define una tipificación muy fuerte, una jerarquía de datos matemáticamente correcta. Tiene un lenguaje de programación e incorpora un compilador.

  Hay también una herramienta: Rosetta Stone la cual ofrece traducciones de muchas de las operaciones básicas para varios sistemas de álgebra computacional, incluyendo Maxima.

- GAP. Es un sistema para trabajo con álgebra discreta, con particular énfasis en Teoría de Grupos Computacional.

- JASYMCA. Es una calculadora simbólica escrita para teléfonos móbiles y PDAs. Resuelve y manipula ecuaciones, maneja problemas básicos de cálculo y provee un poco más de las típicas funciones de un sistema de álgebra computacional. La sintáxis está relacionada con GNU-Maxima.

- REDUCE. Es un sistema interactivo para cálculos algebraicos generales de interés para matemáticos, científicos e ingenieros.
- SINGULAR. Es un Sistema de Álgebra Computacional para manipulación polinomial con especial énfasis en álgebra conmutativa, geometría algebraica y teoría de la singularidad.
- YACAS. Es un sistema de Álgebra Computacional de fácil uso y de propósito general, un programa para manipulación simbólica de expresiones matemáticas. Usa su propio lenguaje de programación diseñado para cálculos tanto simbólicos como numéricos con precisión arbitraria.

Capítulo 2

Lisp

2.1. Introducción

Lisp (LISt Processing) uno de los dos lenguajes de programación más antiguos[1] que maneja una sintaxis a base de listas, es creado en 1959 por John McCarthy cuando residía en el Instituto Tecnológico de Massachusetts (MIT) y en un principio es usado para programación en AI (Artifitial Intelligence).

John McCarthy mostró como un conjunto pequeño de operaciones simples y la noción de función se puede construir un lenguaje de programaciôm completo. Luego que Fortran fuera mostrado como una innovadora forma de poder realizar cálculos matemáticos sin necesidad de realizarlo en lenguaje de ensamblador, si no con notación acorde a lo que entiende uno en la matemática usual, Lisp en sus inicios fue una forma de repetir lo logrado pero a base de listas.

Incluso en su primera versión se hizo llamar FLPL (FORTRAN List Processing Language) para luego crecer como Lisp y se desarrollasen una cantidad de versiones en torno a él.

Lisp tiene muchos dialectos los cuales han ido evolucionando conforme a las necesidades de los programadores, dos de los más usados son Common Lisp y Scheme. Algunos compiladores y versiones son: ABCL, Allegro Common Lisp, CLiCC, CLISP, Clojure, Clozure CL[2], CMUCL, Corman Lisp, DotLips, ECL, GCL, Hedgehog, ... para más referencia puede consultarse la página [22].

La manera en que se construyó Lisp es ahora uno de los tipos de programación que se trabajan, uno precisamente bajo el modelo de Lisp (declarativo) y el otro del tipo en que se maneja C (imperativo).

[1] El primero es FORTRAN.

[2] Sobre esta versión de Lisp se trabajará para efectos de pruebas y ejemplos.

2.2. Entorno

2.2.1. Sintaxis, Funciones y Comandos Básicos

Lisp es un lenguaje que se maneja a base de listas casi en su totalidad. La forma en que se declaran variables, números, cadenas.

Las funciones también son declaradas con ese formato, de la siguiente forma:

(*nombre función*, *argumento 1*, *argumento 2*, ...).

ejemplo:

$>$ (+ 2 3) $\Rightarrow$ $2+3$ $\quad$ $>$ (* 2 3) $\Rightarrow$ $2*3$

5 $\quad$ 6

$>$ (− 2 3) $\Rightarrow$ $2-3$ $\quad$ $>$ (/ 2 3) $\Rightarrow$ $2/3$

-1 $\quad$ $2/3$

Las funciones analíticas que por default tiene ya programadas responden a esta lógica[3].

$>$ (cos 1) $\Rightarrow$ $cos(1)$ $\quad$ $>$ (exp 1) $\Rightarrow$ e^1

0.54030234 $\quad$ 2.7182817

$>$ (log 1) $\Rightarrow$ $log(1)$ $\quad$ $>$ (sqrt 1) $\Rightarrow$ $\sqrt{1}$

0 $\quad$ 1.0

Si se deseará escribir una combinación de ellas, se tienen que seguir las mismas reglas:

$>$ (− (* (+ (cos 1) (exp 1)) (log 2)) 2) $\Rightarrow$ $(cos(1)+e^1)*log(2)-2$

0.25867844

La forma en que se opera es, si la lista está anidada como arriba, calcular de adentro hacia afuera.

Una característica que Lisp tiene por ser un lenguaje simbólico es el manejo de fracciones como tal, es decir, sin convertirlas a números flotantes:

$>$ (+ (/ 1 2) (/ 1 3)) $\Rightarrow$ $\frac{1}{2}+\frac{1}{3}$

5/6

Cuando alguna de las operaciones básicas tiene más de un argumento, realiza la operación dos a dos, donde el primer número que opera es el resultado de los anteriores.

[3]El símbolo “>” es el cursor que se utiliza en la pantalla de comandos, a veces se tiene “$” como cursor, depende de la versión.

$$
\begin{aligned}
&> (+\ 1\ 2\ 3\ 4) \quad 10\\
&> (-\ 1\ 2\ 3\ 4) \quad -8\\
&> (*\ 1\ 2\ 3\ 4) \quad 24\\
&> (/\ 1\ 2\ 3\ 4) \quad 1/24
\end{aligned}
$$

Para asignar valores a una variable se utiliza SETQ:

> (`setq` x 1) x tiene el valor asignado de 1.

2.2.2. Listas

Una lista está hecha de átomos[4] u otras listas. Para definir una lista se utiliza una comilla simple ‘ antes de ella, de otra forma genera un error, ya que LISP tiene reservado el primer sitio de cualquier arreglo para funciones y los consecutivos para los argumentos de éstas. Por ejemplo si se escribiera:

> (1 2 3 4)
Error: Car of (1 2 3 4) *is not a function name or lambda-expression.*

La forma correcta de definir la lista es,

> ‘(1 2 3 4)
(1 2 3 4)

Se asigna a la variable *list* la lista formada por ‘(1 2 3 4).

> (`setq` list ‘(1 2 3 4))

Cada lista tiene dos formas de ser representada, interna y externamente.[5]

La forma externa es cómoda porque es compacta y fácil de usar. La representación interna es la manera en que la lista se almacena en memoria, hay una representación gráfica para entenderlas.

La representación interna de lista no involucra paréntesis, en memoria son organizadas como cadenas de CONS en celdas. Las celdas son ligadas por punteros, cada celda tiene dos punteros uno que señala al elemento de la lista y el otro hacia la siguiente celda en la cadena que se representa.

La Figura 2.1 nos muestra la representación interna (a nivel de memoria) de la lista (a b c). Como se observa la lista al final apunta a NIL convención que LISP tiene.

La lista vacío es denotada por () e internamente es representada por NIL.

[4] Un átomo es la unidad básica en la estructura de dato que LISP tiene para si.
[5] Entendiendo externamente como se despliega en pantalla para el usuario.

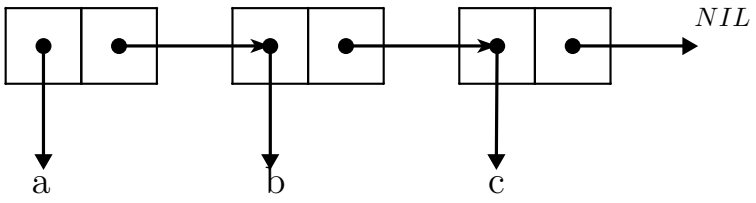


Figura 2.1: Diagrama representando la lista (a b c).

Otras funciones bastante utilizadas son CAR (*Contents of Address portion of Register*) y CDR (*Contents of Decrement portion of Register*) socorridas a la hora de programar. Éstas reciben como argumentos listas y a base de los punteros que tienen asignados se ejecutan, mientras que CAR toma lo que esté dentro de la primera celda y lo muestra como salida, CDR brinca a la segunda celda y toma lo que hay de ahí en adelante.

Comando	Descripción	Ejemplo
CAR	Extrae el primer elemento de una lista.	> (car list) 1
CDR	Extrae los últimos $n-1$ elementos de una lista de n.	> (cdr list) (2 3 4)

La función CONS (*CONStruct*) toma su primer argumento (el cual puede ser un átomo o lista) y lo inserta delante del primer elemento de su segundo argumento.

Tiene la implementación "inversa" de lo que realizan CAR y CDR ya que mientras las segundas extraen elementos de una lista el primero crea listas nuevas a partir de dos.

Comando	Descripción	Ejemplo
CONS	Crea listas.	> (cons 1 '(2 3 4) (1 2 3 4)

A continuación se muestran algunas funciones para crear listas.

Comando	Descripción	Ejemplo
LIST	Crea lista a partir de sus argumentos.	> (`list` ‘a ‘b ‘c ‘d) (A B C D)
APPEND	Tiene como argumentos dos listas, devuelve una sóla lista con la unión de las primeras.	> (`append` ‘(a b) ‘(c d)) (A B C D)
REVERSE	Devuelve la lista que tiene como argumento pero en orden inverso.	> (`reverse` ‘(a b c d)) (D C B A)
FIRTS	Devuelve el primer elemento de una lista.	> (`first` ‘(a b c d)) A
LAST	Devuelve el último elemento de una lista.	> (`last` ‘(a b c d)) D
REMOVE	Quita el elemento de una lista que lo contenga.	> (`remove` ‘a ‘(a a b c d) (B C D)
MEMBER	Verifica si su argumento pertenece a una lista, de encontrarse devuelve T (como la lista completa) y NIL en caso contrario.	> (`member` ‘a ‘(a b c d)) (A B C D)

2.2.3. Predicados

Algo que distingue a Lisp es el tener en su estructura el concepto de “predicado”, el cual es una prueba de veracidad sobre alguna propiedad.

El predicado es una función que devuelve T (*true*) o NIL (*false*) si la prueba que realiza a su argumento se cumple como cierta o no. Lisp tiene esta categoría dentro de sus funciones.

El predicado ATOM devuelve T sí su argumento es un número, cadena, caracter, símbolo, etc, o visto como todo aquello que no sea un CONS.

```
> (atom 1)          > (atom (car ‘(a b c d))
T                   T
```

Algunos de los predicados usados son lo siguientes:

Comando	Descripción	Ejemplo
NUMBERP	Verifica si el argumento es un número.	> (numberp 3) T > (numberp ‘e) NIL
INTEGERP	Verifica si el argumento es un número entero.	> (integerp 3) T
RATIONALP	Verifica si el argumento es un número racional.	> (rationalp 3) T
FLOATP	Verifica si el argumento es un número flotante.	> (floatp 3) NIL > (floatp 3.0) T
COMPLEXP	Verifica si el argumento es un número complejo.	> (complexp 3) NIL
SYMBOLP	Verifica si el argumento es un símbolo.	> (symbolp ‘e) T > (symbolp 3) NIL
LISTP	Verifica si el argumento es una lista.	> (listp ‘(a b)) T
ZEROP	Verifica si el argumento es 0.	> (zerop 3) NIL
ODDP	Verifica si el argumento es par.	> (oddp 3) NIL
EVENP	Verifica si el argumento es impar.	> (evenp 3) T
<	Verifica si sus argumentos cumplen con la relación “menor que”.	> (< 2 3) T
>	Verifica si sus argumentos cumplen con la relación “mayor que”.	> (> 2 3) NIL
=	Verifica si sus argumentos cumplen con la relación “igual que”.	> (= 2 3) NIL

Los predicados presentados son de los más comúnmente usados sin embargo existen más, por mencionar algunos extras: NULL, CONSP, CHARACTERP, STRINGP, BIT-VECTOR-P, VECTORP, SIMPLE-VECTOR-P, SIMPLE-STRING-P, SIMPLE-BIT-VECTOR-P, ARRAYP, PACKAGEP, FUNCTIONP, COMPILED-FUNCTION-P, COMMONP, STREAMP, RANDOM-STATE-P, READTABLEP, HASH-TABLE-P, PATHNAMEP.

2.2.4. Conjuntos

LISP tiene dentro de sus implementaciones un apartado de funciones para manipular conjuntos, con las operaciones que se conocen para ellos, unión, intersección, diferencia, se mostrará una breve descripción de ellas.

INTERSECTION	Devuelve la lista formada por la intersección de las listas que tiene como argumentos. Si no tienen elementos en común devuelve NIL.
Ejemplo:	> (`intersection` '(a 1 b 2 c) '(1 a 2 b 3)) (2 B 1 A)
UNION	Devuelve la unión de las listas que tiene como argumentos.
Ejemplo:	> (`union` '(a 1 b 2 c) '(1 a 2 c b 3)) (1 A 2 C B 3)
SET-DIFFERENCE	Devuelve una lista con los elementos resultantes luego de quitar los elementos de una lista a otra.
Ejemplo:	> (`set-difference` '(a 1 b 2 c) '(1 a 2 b 3)) (C) > (`set-difference` '(1 a 2 b 3) '(a 1 b 2 c)) (3)
REMOVE-DUPLICATES	Retira los valores repetidos de una lista.
Ejemplo:	> (`remove-duplicates` '(a 1 a d 3 a b 4 4 c 5)) (1 D 3 A B 4 C 5)
SUBSETP	Predicado que devuelve T cuando uno de sus dos argumentos es subconjunto del segundo y en caso de no serlo NIL.
Ejemplo:	> (`subsetp` '(1 a) '(1 a 2 b 3)) T > (`subsetp` '(b 4) '(1 a 2 b 3)) NIL

2.2.5. Funciones

Cuando se define una función, creada por el usuario, en LISP ésta tiene la misma jerarquía que las ya implementadas. Lo que significa que cualquier función nueva será considerada como una extensión del mismo LISP. Se tienen varias implementaciones para definir una función, se verán algunas de ellas.

QUOTE	Regresa sus argumentos los cuales **no son evaluados**.
	Sintaxis
Ejemplo:	> (quote (* n n n)) (* N N N) > (quote (3 3 3)) (3 3 3)
DEFUN	Es un tipo de función llamada MACRO FUNCTION la cual no evalúa sus argumentos. El primero de ellos es el **nombre** de la función a crear, el segundo es, el o los **argumentos** a utilizar para la nueva función y el resto es para detallar el **cuerpo**.
	Sintaxis para declarar una función.
Ejemplo:	> (defun potencia3 (n) (* n n n)) POTENCIA3
	Modo de llamado.
	> (potencia3 4) 64

DEFUN tiene dentro de la sintaxis para declarar los argumentos de las funciones a crear los llamados **lambda-list keywords** que es una lista extra de argumentos que permiten hacer pruebas adicionales a lo que la función en si va a realizar.

& OPTIONAL: Permite agregar argumentos opcionales a la función sin que éstos necesariamente se utilicen.

& REST: Agrega una cantidad ilimitada de argumentos a la nueva función.

& KEY: Permite incluir una lista de argumentos extras sin que exista problema alguno si no se evalúan todos.

& AUX: Da flexibilidad a crear variables locales a las cuales se les puede asignar un valor predeterminado en caso contrario toma el valor de NIL.

La expresión LAMBDA sirve para crear funciones que son evaluadas al momento de ejecutarse.

LAMBDA	La expresión lambda sirve para definir funciones, requiere del o los **argumentos** y el **cuerpo** de dicha expresión.
	Si se desea programar $\lambda(n) = n^3$.
Ejemplo:	> (lambda (n) (* n n n))

APPLY Toma una función (como argumento) y una lista como segundo argumento, aplicando la función a la lista. Se tiene que escribir con #‘ para ser ejecutada.

Sintaxis.

Ejemplo:
```
> (apply #'* '(4 4 4))
64
> (apply #'= '(1 2))
NIL
```

Otra de las características que tiene LISP, como ya se había mencionado, es el hecho de tener funciones que llamen funciones como argumentos, FUNCALL tiene esta propiedad.

FUNCALL Recibe como datos de entrada, una función y los argumentos de la misma para evaluarlos sobre ellos.

Sintaxis: (`funcall` #‘*función argumento-1 ... argumento-n*)

Ejemplo:
```
> (funcall #'* 4 4 4)
64
> (funcall #'potencia3 4)
64
```

MAPCAR Función que evalúa una función (que tiene por argumento) en una lista.

Sintaxis: (`mapcar` #‘*función lista*)

Ejemplo:
```
> (mapcar #'sqrt '(1 4 9 16))
(1 2 3 4)
> (mapcar #' potencia3 '(1 2 3 4))
(1 8 27 64)
```

Dentro del entorno que maneja LISP las variables tienen distintas formas de ser declaradas, las que actúan a nivel local y las globales. LET es una función que permite manejo de variables locales dentro del cuerpo de la misma.

LET Permite definir variables locales y dentro de su entorno ejecutar una serie de sentencias, devolviendo como valor el último calculado.

Sintaxis: (LET (*variable-1 valor-1 ... variable-n valor-n*) *cuerpo*)

Ejemplo:
```
>  (let ((n3 (* 4 4 4))
         (s (+ 4 4)))
      (list n3 s))
(64 8)
```

LET* Realiza las mismas funciones que LET salvo que aquí las variables locales que se crean pueden interactuar entre ellas.

Sintaxis: (LET (*variable-1 valor-1 ... variable-n valor-n*) *cuerpo*)

Ejemplo:
```
>  (let* ((n3 (* 4 4 4))
          (s (+ 4 n3)))
      (list n3 s))
(64 68)
```

2.2.6. Condicionales y Recursión

Condicionales

Lisp por ser lenguaje de programación tiene sus condicionales, unos estándares y otros propios, se revisaran algunos de ellos, con una breve descripción y un ejemplo para fijar ideas.

IF — El condicional ejecuta una instrucción si la sentencia lógica que tiene asignada es verdadera (T) y otra en caso de ser falsa (NIL).

Sintaxis: (IF (*Sentencia lógica*) (Si T *hacer*) (Si NIL *hacer*))

Ejemplo:
```
> (if (> 1 0) 'positivo 'negativo)
POSITIVO
> (if (> -1 0) 'positivo 'negativo)
NEGATIVO
```

COND — Consiste en una lista de pruebas con sus respectivas instrucciones. Va verificando cada instrucción y de ser cierta se ejecuta el cuerpo de ese nivel y da como salida el valor calculado, de ser negativa sigue al siguiente nivel haciendo lo mismo hasta acabar, si ninguna prueba se cumple como verdadera da NIL como salida.

Todas la pruebas ciertas y se evalúa la primera.
```
> (cond ((= 0 0) 'Cero-es-igual-a-cero)
        ((< 0 1) 'Uno-es-positivo)
        ((> 0 -1) 'Uno-es-negativo))
CERO-ES-IGUAL-A-CERO
```

Dos últimas pruebas ciertas, se evalúa la primera.
```
> (cond ((= -1 0) 'Cero-es-igual-a-cero)
        ((< 0 1) 'Uno-es-positivo)
        ((> 0 -1) 'Uno-es-negativo))
UNO-ES-POSITIVO
```

Ninguna prueba cierta, se devuelve el valor NIL
```
> (cond ((= -1 0) 'Cero-es-igual-a-cero)
        ((> 0 1) 'Uno-es-positivo)
        ((< 0 -1) 'Uno-es-negativo))
NIL
```

Para asegurar que siempre sea devuelto un valor en COND, no necesariamente NIL, se puede poner al final como última prueba el valor T (verdadero) con un sentencia a ejecutar. De resultar todas las pruebas falsas T siempre devolverá el valor que se le haya asignado predeterminadamente.

	Todas la pruebas falsas y la última verdadera (T).
>	(cond ((= -1 0) 'Cero-es-igual-a-cero) ((> 0 1) 'Uno-es-positivo) ((< 0 -1) 'Uno-es-negativo) **(T 'Ningun-resultado))** NINGUN-RESULTADO

Las macros AND y OR sirven para construir instrucciones que requieren ser evaluadas de forma excluyente o parcial.

AND Ejecuta una serie de sentencias siempre y cuando estas sean verdaderas, cuando encuentra la primera falsa se detiene y devuelve NIL. De cumplirse todas devuelve el valor de la última.

Sintaxis: (AND (*sentencia 1*) (*sentencia 2*) ... (*sentencia n*))	
Las dos sentencias verdaderas.	La primera sentencia falsa.
> (and (< 0.5 1) (> 0.5 0)) T	> (and (> 0.5 1) (> 0.5 0)) NIL
Segunda sentencia falsa.	Ambas sentencias falsas.
> (and (< 0.5 1) (< 0.5 0)) NIL	> (and (> 0.5 1)) (< 0.5 0)) NIL

OR Ejecuta una serie de sentencias hasta hallar la primera verdadera, de otra forma sigue hasta terminar, si este es el caso devuelve NIL.

Sintaxis: (AND (*sentencia 1*) (*sentencia 2*) ... (*sentencia n*))	
Las dos sentencias verdaderas.	La primera sentencia falsa.
> (or (< 0.5 1) (> 0.5 0)) T	> (or (> 0.5 1) (> 0.5 0)) T
Segunda sentencia falsa.	Ambas sentencias falsas.
> (or (< 0.5 1) (< 0.5 0)) T	> (or (> 0.5 1)) (< 0.5 0)) NIL

Recursión

Con las herramientas que se tienen hasta el momento y con ejemplos en concreto se mostrará el concepto de recursión que tiene Lisp.

Supóngase que se desea calcular:

$$S(n) = \sum_{i=1}^{n} i$$

la cual se puede ver de forma recursiva como:

$$S(n) = n + S(n-1)$$

si se toma $S(0) = 0$ se define la siguiente función:

```
(defun suma (n)
   (cond ((zerop n) 0)
      (t (+ n (suma (- n 1))))))
```

La segunda línea verifica con el predicado ZEROP, si el argumento es cero, de serlo asigna a la función SUMA el valor de cero. La tercera línea tiene la recursión ya que se suma $n +$ SUMA$(n-1)$.

```
Ejemplos:  >  (suma 1)    > (suma 4)
              1           10
           >  (suma 20)   > (suma 100)
              210         5050
```

Por ejemplo si se deseará programar el valor n-ésimo para la sucesión de Fibonacci dada por $F(n) = F(n-1) + F(n-2)$ con $F(1) = 1$, $F(0) = 1$ y para $n \geq 2$. Su función en LISP estaría dada:

```
(defun fib (n)
   (cond ((zerop n) 1)
         ((= n 1) 1)
         (t (+ (fib (- n 1)) (fib (- n 2))))))
```

Las líneas dos y tres calculan las condiciones iniciales mientras que la cuarta da la recursividad.

```
Ejemplos:  >  (fib 1)   > (fib 0)
              1         1
           >  (fib 5)   > (fib 20)
              8         10946
```

Hay una manera equivalente de escribir (- n 1) que LISP tiene dado por: (1- n). De esta forma las funciones definidas antes pueden ser escritas también como:

```
(defun suma (n)
   (cond ((zerop n) 0)
      (t (+ n (suma (1- n))))))
(defun fib (n)
   (cond ((zerop n) 1)
         ((= n 1) 1)
         (t (+ (fib (1- n)) (fib (- n 2))))))
```

Dos funciones que tienen comportamiento recursivo son DOLIST y DOTIMES.

DOLIST Evalúa un ciclo sobre los elementos de una lista que se ejecutan en la instrucción definida.

Sintaxis:(DOLIST (*variable lista*) *Sentencia a evaluar*)

```
Ejemplo:  > (dolist (x '(1 2 3 4)) (print (potencia3 x)))
            1
            8
            27
            64
            NIL
```

DOTIMES Evalúa un ciclo sobre un índice creando una lista como resultado de ejecutar una sentencia definida.

Sintaxis: (DOTIMES (*variable sobre un contador*) *Sentencia a evaluar*)

```
Ejemplo:  > (dotimes (i 5) (print (potencia3 i)))
            0
            1
            8
            27
            64
            NIL
```

LISP cuenta con un poderoso bucle[6] para realizar iteraciones, éste es DO y su variante DO*

DO Es un macro que realiza ciclos hasta que la condición de paro se cumple. Puede ir actualizando variables al mismo tiempo a distintos tamaños de paso de forma independiente.

Sintaxis:

```
(DO ((variable-1 inicialización-1 [Actualización-1])
     (variable-2 inicialización-2 [Actualización-2])
     ...)
     ( prueba action-1 ... action-n)
  cuerpo)
```

```
Ejemplo:  (defun fundo (n)
            (do ((A 1 B)
                 (B 1 (+ A B))
                 (k n (1- k)))
                ((<= k 0) B)))
```

```
Llamado:  (mapcar #'fundo '(1 2 3 4 5 6 7 8 9 10))
          (2 3 5 8 13 21 34 55 89 144)
```

[6] Bucle = ciclo

DO* Tiene las mismas funciones que el macro DO con la diferencia que éste macro permite inicializar las variables sin que se les haya necesariamente asignado un valor específico.

```
Ejemplo:  (defun fundo* (n)
            (do ((A 1 B)
                 (B A (+ A B))
                 (k n (1- k)))
                ((<= k 0) B)))
Llamado:  (mapcar #'fundo* '(1 2 3 4 5 6 7 8 9 10))
          ((2 4 8 16 32 64 128 256 512 1024)
```

Ambas funciones tienen la siguiente estructura: se inicializan A y B en 1 y 1 para FUNDO, 1 y A para FUNDO* (como $A = 1$, entonces B también toma el valor de 1). La variable k va decrementándose en cada iteración en una unidad y la condición de paro es, sí $k \leq 0$ devolver B.

Mientras que FUNDO da los elementos de la sucesión de Fibonacci, FUNDO* las potencias n-ésimas de 2. Ambas funciones tienen esencialmente la misma sintaxis incluso las dos en $n = 1$ toman el valor de 2.

Las dos tablas siguientes muestran la sútil diferencia entre DO y DO*, como se puede observar la primera actualiza el paso k-ésimo con las variables sólo del paso (k-1)-ésimo en tanto la segunda actualiza el paso k-ésimo con todas la variables que antes ya hayan cambiado su valor.

Primeras cuatro iteraciones para FUNDO.			
$A_1 = 1$	$A_2 = 1$	$A_3 = 2$	$A_4 = 3$
$B_1 = 1$	$B_2 = A_1 + B_1$	$B_3 = A_2 + B_2$	$B_4 = A_3 + B_3$
	$2 = 1 + 1$	$3 = 1 + 2$	$5 = 2 + 3$

Primeras cuatro iteraciones para FUNDO*.			
$A_1 = 1$	$A_2 = 1$	$A_3 = 2$	$A_4 = 4$
$B_1 = 1$	$B_2 = A_2 + B_1$	$B_3 = A_3 + B_2$	$B_4 = A_4 + B_3$
	$2 = 1 + 1$	$4 = 2 + 2$	$8 = 4 + 4$

2.3. Macros

Las Macros son una forma de extender la sintaxis en LISP, un programa que escribe programas. Brinda una ventaja al escribir de forma compacta algunas funciones.

Una Macro no es por si sóla una función, ya que escribe expresiones que al ser evaluadas dan resultados concretos.

La forma en que se define una MACRO es similar a como se hace con las funciones, la instrucción DEFMACRO permite comenzar. Se verá como:

Ejemplo

```
(defmacro opera2 (var1 var2)
  (list 'setq var1 (list '+ var1 var2) var2 (list '- var1 var2)))
```

La macro que se construye tiene como argumentos de entrada dos variables *var1* y *var2*, utilizando SETQ se le pasan los valores *var1 = (+ var1 var2)* y *var2 = (- var1 var2)*. El caracter ' (quote) sirve para no evaluar expresiones cuando se antecede a las funciones SETQ, + y -, éstas no se ejecutan.

Cuando la MACRO es cargada en memoria para posteriormente usarle, ésta lee las líneas que tiene como parte de su código y las evalúa en los argumentos que tiene.

Cabe mencionar que OPERA2 hace su labor sobre variables ya tengan un valor predeterminado, de otra forma marca error.

Paso 1: Ejecutar en LISP
> `opera2`

Paso 2: Asignar valores a dos variables.	
> (`setq` a 1)	> (`setq` b 1)
1	1

Paso 3: Llamado en LISP
> (`opera2` a b)
1

Paso 4: Se muestra en una lista los valores que ahora tienen a y b.
> (`list` a b)
(2 1)

Se crea una lista para observar el comportamiento.	
> (`opera2` a b)	> (`list` a b)
2	(3 2)
> (`opera2` a b)	> (`list` a b)
3	(5 3)
> (`opera2` a b)	> (`list` a b)
5	(8 5)

Como se puede observar, las variables A y B van cambiando su valor, ya que son modificadas dentro de OPERA2.

Hay dos caracteres que sirven para simplificar la sintaxis a la hora de definir MACROS, ‘ (backquote) , (coma) y @ (arroba).

La comilla invertida o backquote, cuando se usa sola tiene la misma función que la comilla simple.

Crear listas.	
> '(a b c d)	> ‘(a b c d)
(A B C D)	(A B C D)

La comilla invertida al combinarse con la coma tiene el siguiente resultado:

Se asigna a `c` una lista.	Se crea otra lista.
> (`setq` c '(1 2 3 4)) (1 2 3 4)	> '(a b c d) (A B C D)
Se combina la comilla invertida con la coma.	
> `'(a b ,c d)` (A B (1 2 3 4) D)	

Se puede observa que en lugar de pasar el símbolo "c" como tal pasa lo que tiene asignado previamente.

Cuando se utiliza una mezcla con @ el resultado es el siguiente:

> `'(a b ,@c d)` (A B 1 2 3 4 D)

El efecto que tiene @ es destructivo en el sentido de "olvidar" los paréntesis que trae "c" como lista propia.

Utilizar backquote dentro de la sintaxis al definir una macro ayuda a simplificar el código.

Definición original.
(`defmacro opera2` (var1 var2) (`list` '`setq` var1 (`list` '+ var1 var2) var2 (`list` '- var1 var2)))
Utilizando backquotey coma.
(`defmacro opera2` (var1 var2) `'(setq ,var1 (+ ,var1 ,var2) ,var2 (- ,var1 ,var2)))`

Se tiene lo mismo pero reducido en líneas que a la hora que son demasiado grandes las Macros, lo cual ayuda tanto a su lectura como a su escritura.
Para referencia detallada se puede consultar [14].

2.4. Maxima-Lisp

Debido a que Lisp es el lenguaje donde Maxima se elaboró hay manera de trabajar éste desde el ambiente de wxMaxima con la siguiente sintaxis:

`:lisp #$`*`código`*`$`

Utilizando la sintaxis descrita arriba, en Maxima para la simple instrucción de definir una lista `X = [a,b,c]`, el `output` correspondiente se muestra en la tabla de abajo,

Input	`(%i):lisp #$X = [a,b,c]$;`
Output	((MEQUAL SIMP) $x ((MLIST SIMP) $A $B $C))

El `output` que me visualiza es la manera en que internamente Lisp ve la lista de Maxima, como era de esperarse una lista de listas con parámetros especiales. La lectura de éstos puede leerse abajo.

- MEQUAL = Maxima EQUAL → "Igualdad en Maxima".
- SIMP = SIMPle.
- MLIST = Maxima LIST → "Lista de Maxima".
- $x → Símbolo "x" en Maxima.

La sintaxis nos puede mostrar la diferencia entre las distintas clases de asignación, por ejemplo al tomar en MAXIMA los siguiente:

Input	`(%i):lisp #$X : [a,b,c]$;`
Output	((MLIST SIMP) $A $B $C))

A la hora de asignar valores de una función:

Input	`(%i):lisp #$f(x):= x^2$;`
Output	((MDEFINE SIMP) (($F) $X) ((MEXPT) $X 2))

Se tiene Maxima DEFINE del símbolo $F con argumento $X la regla de correspondencia MaximaEXPT (exponente) 2.

Cuando se tiene una simple evaluación, dígase $cos(2)$,

Input	`(%i):lisp #$cos(2)$;`
Output	((%COS SIMP) 2)

Al tener ya LISP las funciones básicas sólo se llaman y evalúan.

Una matriz es una lista de lista como se observa debajo.

Input	`(%i):lisp #$matrix([1,-1],[-1,1])$;`
Output	(($MATRIX SIMP) ((MLIST SIMP) 1 -1) ((MLIST SIMP) -1 1))

Con la instrucción `to_lisp()` se va a una sesión de LISP dentro del mismo MAXIMA:

Input	`(%i) to_lisp();`
Output	*Type (to-maxima) to restart, ($quit) to quit Maxima.* MAXIMA>

Así se tiene una sesión de LISP desde donde se puede hacer todo lo que anteriormente se describió.

Se pueden ver las variables que están activas en la sesión de MAXIMA:

```
MAXIMA>   #$X$
((MLIST SIMP) $A $B $C)
MAXIMA>
```

Para regresar a MAXIMA se teclea: `(to-maxima)` y se vuelve al entorno simbólico.

Una virtud que tiene dicha relación es que desde la sesión de LISP se pueden crear, definir variables-funciones o asignar valores y las mismas serán reconocidas desde MAXIMA, sólo se antecede el signo $.

Sesión de LISP sobre MAXIMA	
MAXIMA>	`(defun $misuma (x y) (+ x y))`
`$MISUMA`	
MAXIMA>	`($misuma 4 4)`
8	
MAXIMA>	`(to-maxima)`

Sesión en MAXIMA	
`(% i)`	`misuma(4,4);`
`(% o)`	8

La función `misuma` se ejecuta libremente desde MAXIMA y desde LISP como `$misuma`.

Al haber visto como interactúan ambos lenguajes se cierra la parte que corresponde a presentar el software donde se desarrollaran las aplicaciones para dar paso al marco teórico donde se exploran los problemas a programar.

Capítulo 3

Geometría Cuántica

3.1. Preliminares

La parte correspondiente a esta sección es la introducción y desarrollo del marco teórico de lo que se implementará. Un panorama general de lo que es la Geometría Cuántica y conceptos básicos.

La Geometría Cuántica presenta un nuevo concepto de espacio que la geometría estándar tiene por si sola.

En geometría clásica (estándar) el espacio se ve como un conjunto de puntos equipados con una estructura extra. Por ejemplo en geometría euclidiana dicha estructura está dada por rectas y planos, la relación entre ellas, así como la especificación de los ángulos entre sus intersecciones. En geometría diferencial es dada por sistemas locales de coordenadas, es decir, especificando los atlas. El espacio topológico está caracterizado por la colección de abiertos en él, en la misma lógica la estructura de un espacio medible son los conjuntos medibles que lo conforman.

Se puede decir que en todas las modalidades de la geometría clásica la estructura correspondiente está codificada en un álgebra, cuyos elementos son ciertas funciones sobre el espacio dado. En topología por ejemplo son las funciones continuas sobre el espacio con valores en los complejos, en geometría diferencial son las funciones suaves mientras que en un espacio de medida son las funciones medibles.

El plano metodológico de la Geometría Cuántica es traducir todos los conceptos geométricos al lenguaje de álgebras de funciones y luego observar que en esta forma traducida ya no es necesario mantener la suposición sobre la naturaleza de la estructura algebraica (como las funciones sobre el espacio correspondiente) aquí es importante observar que en todos los ejemplos las álgebras de funciones son *conmutativas*, la parte central de generalización cuántica es trabajar con

álgebras apropiadas y *no conmutativas*.

La figura 3.1 muestra gráficamente el universo clásico dentro del cuántico.

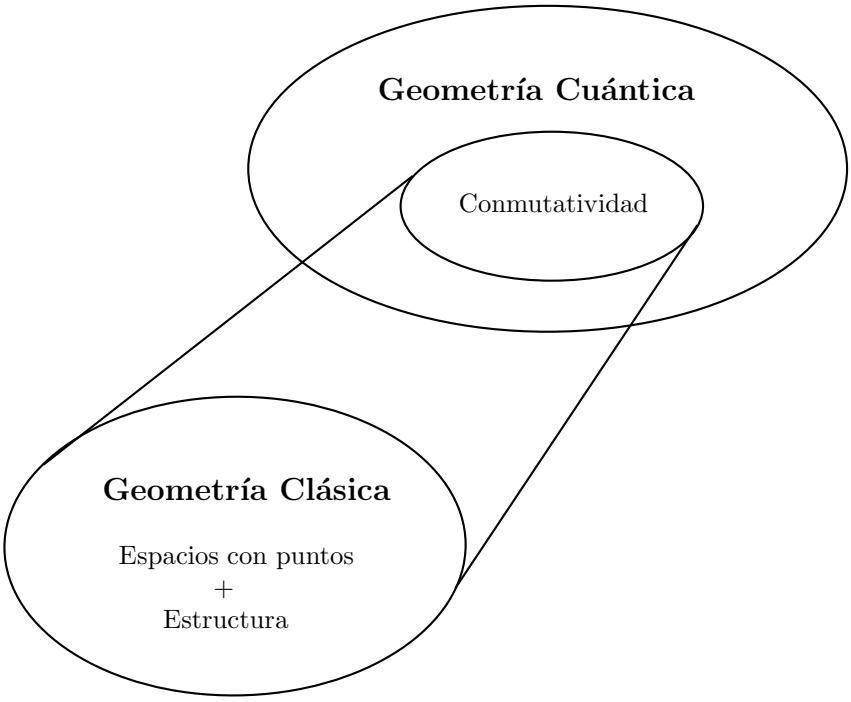


Figura 3.1

Así no hay necesidad de hablar de puntos o estructuras de espacios, se extienden las ideas fundamentales en geometría clásica y se pueden volver a recuperar al agregar la condición de conmutatividad.

Al tener una estructura geométrica clásica en X siempre se le puede asociar una álgebra $*$. Véase para referencia [19].

Aquí vale la pena observar el teorema clásico de Gelfand-Naimark[1] que demuestra, a cada álgebra A conmutativa C^* de manera natural se puede representar como $A \cong \mathcal{C}_0(X)$ para un espacio localmente compacto X. Donde $\mathcal{C}_0(X)$ son funciones continuas que desaparecen en la infinidad de X, es decir, para cada cota $\epsilon > 0$ existe un compacto K en X tal que, módulo de la función, ésta no sobre pasa sus valores fuera de una vecindad de radio ϵ, fuera de K. En caso particular cuando X es compacto se tiene $\mathcal{C}_0(X) = \mathcal{C}(X)$, lo que es equivalente a decir que A es unital.

[1] La demostración puede consultarse en el área de apéndices.

Al desarrollar estas ideas con mayor profundidad, se puede demostrar que toda teoría de álgebras C^* conmutativas es equivalente a teoría de espacios topológicos localmente compactos.

Se puede decir que la Geometría Cuántica rompe el esquema clásico de puntos con estructura, en realidad la idea de punto, parte, aquí-allá se pierde a favor de una descripción global e integral en términos de enteras álgebras no conmutativas.

Una forma de ir haciendo un diccionario de los conceptos geométricos (clásicos) a los algebraicos se ve en la Tabla 3.1 donde se resumen las ideas y relaciones que hay entre ambos [19].

Geometría	**Álgebra**
Espacio topológico compacto X.	Álgebra C^*, unital y conmutativa A.
Puntos $x \in X$.	Caracteres $\kappa = \kappa_x$ de A.
Mapeos continuos entre espacios compacto X y $Y \leftrightarrow B$.	$*$ Homomorfismo Unital de B a A.
Producto $X \times Y$.	Producto C^* Tensorial $A \otimes B$.
Simetrías de X.	Automorfismos de A.
Estructura de Grupo en X.	Mapeo Coproducto $\phi : A \to A \otimes A$.
Medidas de probabilidad en un espacio compacto metrizable X.	Funcionales lineales positivos normalizados en A.
Teoría de la Medida.	Álgebras Conmutativas von Neumann.
Variedad compacta suave X.	$A = \mathcal{C}^\infty(X) = \{$ funciones complejas suaves en X $\}$.

Tabla 3.1: Diccionario elemental Geometría-Álgebra.

3.2. Grupos Cuánticos y Cálculo Diferencial

Una vez mostrado un esquema general de la Geometría Cuántica y la forma en que extiende algunos conceptos e ideas, se presentará de manera general lo que es un grupo cuántico y el cálculo diferencial que existe en ellos.

Sea G un espacio, la estructura de grupo es una aplicación de $G \times G$ en G, usualmente llamada producto, tal que cumple asociatividad, existencia de elemento neutro e inversos.

Dada la estructura del grupo es natural considerar que G es además un espacio topológico. En este caso suponer, por ejemplo, la continuidad en los mapeos producto y inverso, con esto se tiene un grupo topológico, similarmente en caso de variedades suaves, suponiendo las suavidad de dichos mapeos, se llega al concepto de grupo de Lie.

Lo siguiente es ver, cómo llevar la estructura de grupo a nivel cuántico, qué se entenderá por grupo cuántico.

En resonancia con lo descrito previamente y para motivar la extensión, dado G grupo topológico compacto se le asocia A el álgebra C^* de las funciones continuas definidas sobre éste a los complejos y un $*$ homomorfismo $\phi : A \to A \otimes A$ el cual dualiza el concepto de producto a nivel de álgebras. La idea geométrica detrás, es la de tomar ϕ imagen inversa asociado al producto.

Se explorará un poco más el concepto de producto tensorial $\otimes$ entre álgebras para entender como está constituido $A \otimes A$. Véase lo siguiente, sean A y B álgebras, se construye $A \otimes B$ un espacio vectorial en la manera estándar como:

$$A \otimes B = \left\{ \sum a \otimes b \;\middle|\; a \in A,\ b \in B \right\}$$

donde se define un producto y una operación $*$ entre sus elementos, aprovechando la estructura de las álgebras,

$$(a \otimes b)(a' \otimes b') = aa' \otimes bb'$$
$$(a \otimes b)^* = a^* \otimes b^*$$

En caso cuando A y B son además álgebras C^* se puede definir [15] una norma C^* cumpliendo:

$$\|a \otimes b\| = \|a\|\|b\|, \quad a \in A, b \in B.$$

y la completación de $A \otimes B$ al tomar su cerradura

$$A \otimes_* B = \overline{\left\{ \sum a \otimes b \;\middle|\; a \in A,\ b \in B \right\}}$$

Así $A \otimes_* B$ es completa con estructura de álgebra C^*. Lo cual lleva a un concepto de producto tensorial C^*.

Si $A = \mathcal{C}_0(X)$ y $B = \mathcal{C}_0(Y)$ entonces $A \otimes B \cong C_0(X \times Y)$ de manera natural.

El enfoque de la tesis es abordar **Grupos Clásicos y Finitos** caso muy partícular de grupos cuánticos, ya que todos los grupos clásicos (localmente) compactos se pueden ver como grupos cuánticos, con álgebras C^* conmutativas.

Lo que lleva a la pregunta ¿qué es lo cuántico en el presente trabajo?. Lo cuántico resulta del *cálculo diferencial* que se analiza en el contexto de grupos

finitos y que fue desarrollado para todo grupo cuántico por Woronowicz, [16], [17].

En particular los conjuntos finitos no pueden tener ningún comportamiento diferencial desde un punto de vista clásico, situación que varia en el ámbito cuántico ya que se les descubre la "vida secreta" que hay dentro de ellos en un mundo más amplio.

Ahora se presentan las ideas principales que conforman el cálculo diferencial sobre espacios cuánticos.

Cálculo Diferencial

Para estudiar el cálculo diferencial en la forma más general se comienza con un $*$ álgebra $\mathcal{A}$, representando funciones "suaves" sobre el espacio cuántico X, el *cálculo diferencial* sería representado por un $*$ álgebra graduada diferencial Ω tal que

$$\Omega = \sum_{k\geq 0} \oplus \Omega^k \qquad \Omega^0 = \mathcal{A} \qquad \Omega^k \Omega^l \subset \Omega^{k+l}$$

y se supone que elementos de $\mathcal{A}$ generan Ω como álgebra diferencial. En otras palabras Ω está equipado con el diferencial $d : \Omega \to \Omega$, donde $d(\Omega^k) \subset \Omega^{k+1}$ y tal que:

$$d(xy) = d(x)y + (-)^{\partial x} x d(y) \qquad \forall\ x, y \in \Omega$$

es decir, se cumple la regla graduada de Leibniz y además se supone que $d^2 = 0$.

El hecho que $\mathcal{A}$ genera Ω es equivalente a decir que los espacios Ω^k son hechos de sumas de elementos de la forma $a_0 d(a_1)\ ...\ d(a_k)$ donde $a_i \in \mathcal{A}$. Aquí también se está suponiendo que $*d = d*$ y que la estructura $*$ es graduadamente antimultiplicativa, es decir,

$$(xy)^* = (-)^{\partial x \partial y} y^* x^* \qquad \forall\ x, y \in \Omega$$

la interpretación intuitiva es que elementos de Ω representan "formas diferenciales" sobre el espacio cuántico dado.

Resulta que dada el álgebra $\mathcal{A}$ en general existiran multiples formas de extenderla a un cálculo diferencial Ω, cada una de esas multiples formas en principio pueden tener un significado geométrico y representar alguna estructura algebraico-geométrica interesante e importante.

Es también interesante mencionar que siempre se puede construir, lo que se llama *Cálculo Diferencial Universal* $\Omega = \Omega(\mathcal{A})$ definiéndolo como álgebra graduada diferencial generada por $\mathcal{A}$ y *diferenciales formales* $d(a)$, $a \in \mathcal{A}$ con la única regla dada por la propiedad de Leibniz $d(ab) = d(a)b + ad(b)$ donde $a, b \in \mathcal{A}$. El operador d y la estructura $*$ luego se extienden a todo $\Omega(\mathcal{A})$.

Por la construcción cada cálculo diferencial sobre $\mathcal{A}$ se puede ver como proyección de éste *cálculo universal*, en caso cuando el espacio tiene la estructura de grupo cuántico representada por el coproducto $\phi : \mathcal{A} \to \mathcal{A} \otimes \mathcal{A}$, es natural

considerar tales cálculos diferenciales compatibles con la estrutura de grupo, es decir, pedir que ϕ se extiende a homomorfismo $\widehat{\phi} : \Omega \to \Omega \,\widehat{\otimes}\, \Omega$ de álgebras graduadas diferenciales correspondientes. Aqui $\widehat{\otimes}$ es producto tensorial de álgebras graduadas diferenciales, donde producto y estructura $*$ son dados vía,

$$(x \otimes y)(z \otimes t) = (-)^{\partial y \partial z} xz \otimes yt$$

$$(x \otimes y)^* = x^* \otimes y^*$$

mientras el diferencial $d : \Omega \,\widehat{\otimes}\, \Omega \to \Omega \,\widehat{\otimes}\, \Omega$ está definido por:

$$d(x \otimes y) = d(x) \otimes y + (-)^{\partial x} x \otimes d(y)$$

Todo fue estudiado por Woronovicz en [16], [17] y también Durdevich en [6] apéndice B.

En tal caso el cálculo diferencial es intrínsecamente relacionado con su parte $\Omega^1 = \Gamma$ que representa uno formas sobre el grupo y como ha demostrado Woronovicz el espacio de uno formas Γ se puede naturalmente descomponer como $\Gamma \leftrightarrow \mathcal{A} \otimes \Gamma_{inv}$ donde Γ_{inv} es el espacio de uno formas invariantes a la izquierda, que en caso clásico corresponde al dual de álgebra de Lie asociado al grupo G.

Así que toda la estructura de cálculo está de una manera codificada en el espacio Γ_{inv}, para efectos del presente trabajo se enfocó en grupos clásicos y finitos, donde el espacio Γ_{inv} siempre será asociado a un subconjunto S de un grupo G que no contiene el elemento neutro, que está hecho de enteras clases de conjugación y cerrado bajo inversos, en otras palabras, $\Gamma_{inv} \cong L(S)$ el espacio vectorial generado por S.

Un cálculo de primer orden es completamente determinado por un espacio vectorial Γ_{inv} que corresponde a "uno formas" de Γ_{inv} invariantes a la izquierda respecto a la acción natural de G sobre Γ. Se puede demostrar que hay una correspondencia entre los siguientes espacios:

$$\Gamma \leftrightarrow \mathcal{A} \otimes \Gamma_{inv} \leftrightarrow \Gamma_{inv} \otimes \mathcal{A}$$

como $\mathcal{A}$ módulos izquierdos y derechos respectivamente.

Es importante observar, que la condición de la extensibilidad de ϕ hacia homomorfismo de álgebras diferenciales $\widehat{\phi}$, al nivel de cálculo diferencial de primer orden, se manifiesta como una condición de covariancia izquierda y derecha.

Si se considera la restricción de primer orden, se obtiene

$$\widehat{\phi} \colon \Gamma \to \mathcal{A} \otimes \Gamma + \Gamma \otimes \mathcal{A}$$

y las respectivas proyecciones en $\mathcal{A} \otimes \Gamma$ y $\Gamma \otimes \mathcal{A}$ nos dan las acciones izquierda y derecha

$$_\Gamma\phi \colon \Gamma \to \mathcal{A} \otimes \Gamma \qquad \phi_\Gamma : \Gamma \to \Gamma \otimes \mathcal{A}$$

Como resultado de la coasociatividad del mapeo $\widehat{\phi}$ se obtiene

$$(\phi \otimes \mathrm{id})_\Gamma\phi = (\mathrm{id} \otimes {}_\Gamma\phi)_\Gamma\phi$$
$$(\phi_\Gamma \otimes \mathrm{id})\phi_\Gamma = (\mathrm{id} \otimes \phi)\phi_\Gamma$$
$$(\mathrm{id} \otimes \phi_\Gamma)_\Gamma\phi = ({}_\Gamma\phi \otimes \mathrm{id})\phi_\Gamma$$

Los mapeos ${}_\Gamma\phi$ y ϕ_Γ también satisfacen:

$$\begin{aligned} {}_\Gamma\phi(\theta a) &= {}_\Gamma\phi(\theta)\phi(a) \qquad & {}_\Gamma\phi(a\theta) &= \phi(a)_\Gamma\phi(\theta) \\ \phi_\Gamma(\theta a) &= \phi_\Gamma(\theta)\phi(a) \qquad & \phi_\Gamma(a\theta) &= \phi(a)\phi_\Gamma(\theta) \\ {}_\Gamma\phi d(a) &= (\mathrm{id} \otimes d)\phi(a) \qquad & \phi_\Gamma d(a) &= (d \otimes \mathrm{id})\phi(a) \end{aligned}$$

en otras palabras, bimódulo Γ con el diferencial $d\colon \mathcal{A} \to \Gamma$ es un cálculo diferencial bicovariante [16]. Se puede demostrar que a partir de un cálculo diferencial bicovariante, se puede construir todo el cálculo diferencial (con las formas de más alto orden) tal que las dos acciones ${}_\Gamma\phi$ y ϕ_Γ se combinan en el coproducto diferencial $\widehat{\phi}$ en la manera explicada. Hay dos construcciones universales, una maximal mediante envolventes universales diferenciales, [6] apéndice B, y otra minimal mediante álgebras trenzadas exteriores [16].

El cálculo diferencial Γ se descompone naturalmente (como $\mathcal{A}$-módulo izquierdo) en la forma $\Gamma \leftrightarrow \mathcal{A}\otimes\Gamma_{inv}$ donde Γ_{inv} es el espacio de todos los elementos invariantes a la izquierda, en otras palabras

$$\theta \in \Gamma_{inv} \quad \Leftrightarrow \quad {}_\Gamma\phi(\theta) = 1 \otimes \theta$$

El espacio Γ_{inv} en teoría clásica corresponde al dual del álgebra de Lie,

$$\Gamma_{inv} \leftrightarrow lie(G)^*$$

como lo demostró Woronowicz [16], todo cálculo naturalmente se puede construir a partir de Γ_{inv}. Uno de los fenómenos cuánticos muy interesante es la existencia de un canónico operador de trenza

$$\sigma : \Gamma_{inv} \otimes \Gamma_{inv} \to \Gamma_{inv} \otimes \Gamma_{inv}$$

es decir, σ cumple la ecuación de trenza:

$$(\sigma \otimes \mathrm{id})(\mathrm{id} \otimes \sigma)(\sigma \otimes \mathrm{id}) = (\mathrm{id} \otimes \sigma)(\sigma \otimes \mathrm{id})(\mathrm{id} \otimes \sigma)$$

además σ es G-equivariante y en caso clásico se reduce a la transposición estándar.

Como se ha mencionado son grupos finitos con cálculos cuánticos, en este caso, como lo ha demostrado el Dr. Durdevich [7] en el caso de grupos finitos clásicos los espacios Γ_{inv} son naturalmente etiquetados por ciertos subconjuntos S de G, más precisamente, S cerrado bajo conjugaciones e inversos. Dado S el espacio Γ_{inv} se recupera como $L(S)$, de acuerdo con lo anteriormente mencionado.

3.3. Conjunto S

Ahora se mostrarán los conceptos que darán origen a las rutinas que motivaron el tópico principal en el presente trabajo.

Dentro de la teoría clásica de grupos el concepto de conjugación es el siguiente.

DEFINICIÓN. *Sean x, y elementos de un grupo G, se dice que x es* ***conjugado*** *de y si,*

$$gxg^{-1} = y$$

para algún $g \in G$.

La colección de todos los elementos conjugados a un elemento dado es llamada *clase de conjugación* y define [3] una partición de G. La relación de conjugación es una relación de equivalencia. De hecho las clases de conjugación son órbitas de la acción de G sobre G, vía automorfismos internos, dados por:

$$(g, x) \mapsto gxg^{-1}$$

Cerradura de S

Sea G un grupo finito y $S \subset G$, se define a S como conjunto cerrado si dado $x \in S$ entonces:

1. $e \notin S$.
2. $[x] \subset S$.
3. $x^{-1} \in S$.

Donce $[x]$ es la clase de conjugación de x. En este contexto se dirá que S es es "cerrado" bajo inversos y clases de conjugación, sin elemento neutro e.

DEFINICIÓN. *Si $S \subset G$, cumple con las propiedades anteriores se dirá que S es perfecto.*

Los conjuntos perfectos S llevan, de acuerdo con lo mencionado anteriormente, a cálculos diferenciales cuánticos sobre G.

3.4. Operador σ

Ahora se introducirá de manera explícita, el canónico operador de trenza $\sigma : \Gamma_{inv} \otimes \Gamma_{inv} \to \Gamma_{inv} \otimes \Gamma_{inv}$.

Sea $S \subset G$ perfecto de G grupo finito y defínase primero el mapeo $\sigma : S \times S \to S \times S$ por,

$$\sigma(q, g) = (qgq^{-1}, q)$$

con $q, g \in S$.

Con lo ya descrito se define el espacio vectorial $\Gamma_{inv} = L(S)$ como todas las combinaciones lineales de elementos de S, con escalares en $\mathbb{C}$. En otras palabras:

$$L(S) = \left\{ \sum c \cdot s \;\middle|\; s \in S, \quad c \in \mathbb{C} \right\}$$

Los elementos $s \in S$ forman una base natural en $L(S)$. Por lo tanto, elementos $s \otimes t$ forman una base natural en $\Gamma_{inv} \otimes \Gamma_{inv}$ donde $s, t \in S$. Ahora se puede extender la definición anterior de σ al nivel de operador lineal $\sigma : \Gamma_{inv} \otimes \Gamma_{inv} \to \Gamma_{inv} \otimes \Gamma_{inv}$ postulando que

$$\sigma(q \otimes g) = qgq^{-1} \otimes q$$

Como se puede observar, se utiliza la misma notación para el operador σ, tanto el que se define en el producto cruz como en el producto tensorial, para elementos de S se puede pensar en la correspondencia,

$$(q, p) \leftrightarrow q \otimes p, \qquad q, g \in S.$$

por lo tanto, una correspondencia entre los espacios $L(S \times S)$ y $L(S) \otimes L(S)$.

Propiedades básicas de σ

Primero se demostrará que $\sigma : \Gamma_{inv} \otimes \Gamma_{inv} \to \Gamma_{inv} \otimes \Gamma_{inv}$ es biyectivo, especificando su inversa.

PROPOSICIÓN. *El operador σ es biyectivo y su inversa es dada por:*

$$\sigma^{-1}(p \otimes f) = f \otimes f^{-1}pf$$

Demostración. Sean $f, p, q, g \in S$, se tiene:

$$\begin{aligned} \sigma(\sigma^{-1}(p \otimes f)) &= \sigma(f \otimes f^{-1}pf) \\ &= (ff^{-1}pff^{-1} \otimes f) \\ &= (p \otimes f) \end{aligned} \qquad \begin{aligned} \sigma^{-1}(\sigma(q \otimes g)) &= \sigma^{-1}(qgq^{-1} \otimes q) \\ &= (q \otimes q^{-1}qgq^{-1}q) \\ &= (q \otimes g) \end{aligned}$$

Lo que demuestra la proposición. □

PROPOSICIÓN. *Se cumple $\sigma(q \otimes g) = g \otimes q$ sí y sólamente sí:*

$$qg = gq$$

Demostración. En sus dos implicaciones se tiene lo siguiente.

$\Rightarrow$) Suponiendo $\sigma(q \otimes g) = g \otimes q$ por un lado y por otro $\sigma(q \otimes g) = qgq^{-1} \otimes q$ implica:

$$g \otimes q = qgq^{-1} \otimes q \;\Leftrightarrow g = qgq^{-1}$$

multiplicando por la derecha la última expresión por q,

$$qg = gq$$

queda demostrado la conmutatividad.

$\Leftarrow$) Suponiendo $qg = gq$ véase:

$$\begin{aligned}\sigma(q \otimes g) &= qgq^{-1} \otimes q \\ &= gqq^{-1} \otimes q \\ &= g \otimes q\end{aligned}$$

Así quede demostrada la afirmación. □

Proposición. *El operador σ cumple con la ecuación de trenza, es decir,*

$$(\sigma \otimes \mathrm{id})(\mathrm{id} \otimes \sigma)(\sigma \otimes \mathrm{id}) = (\mathrm{id} \otimes \sigma)(\sigma \otimes \mathrm{id})(\mathrm{id} \otimes \sigma)$$

Demostración. Véase:

$$\begin{aligned}(\sigma \otimes \mathrm{id})(\mathrm{id} \otimes \sigma)(\sigma \otimes \mathrm{id})(q \otimes g \otimes p) &= (\sigma \otimes \mathrm{id})(\mathrm{id} \otimes \sigma)(\sigma(q \otimes g) \otimes p) \\ &= (\sigma \otimes \mathrm{id})(\mathrm{id} \otimes \sigma)(qgq^{-1} \otimes q \otimes p) \\ &= (\sigma \otimes \mathrm{id})(qgq^{-1}, \sigma(q \otimes p)) \\ &= (\sigma \otimes \mathrm{id})(qgq^{-1} \otimes qpq^{-1} \otimes q) \\ &= \sigma(qgq^{-1} \otimes qpq^{-1}) \otimes q \\ &= qgq^{-1}qpq^{-1}(qgq^{-1})^{-1} \otimes qgq^{-1} \otimes q \\ &= qgpq^{-1}qg^{-1}q^{-1} \otimes qgq^{-1} \otimes q \\ &= qgpg^{-1}q^{-1} \otimes qgq^{-1} \otimes q\end{aligned}$$

por otro lado,

$$\begin{aligned}(\mathrm{id} \otimes \sigma)(\sigma \otimes \mathrm{id})(\mathrm{id} \otimes \sigma)(q \otimes g \otimes p) &= (\mathrm{id} \otimes \sigma)(\sigma \otimes \mathrm{id})(q \otimes \sigma(g \otimes p)) \\ &= (\mathrm{id} \otimes \sigma)(\sigma \otimes \mathrm{id})(q \otimes gpg^{-1} \otimes g) \\ &= (\mathrm{id} \otimes \sigma)(\sigma(q \otimes gpg^{-1}) \otimes g) \\ &= (\mathrm{id} \otimes \sigma)(qgpg^{-1}q^{-1} \otimes q \otimes p) \\ &= qgpg^{-1}q^{-1} \otimes \sigma(q \otimes g) \\ &= qgpg^{-1}q^{-1} \otimes qgq^{-1} \otimes q\end{aligned}$$

En otras palabras, se cumple la ecuación de trenza. □

Vale la pena mencionar que ambas propiedades se cumplen en caso general de los grupos cuánticos compactos de Woronowicz y sus cálculos bicovariantes, aquí se han presentado demostraciones independientes para grupos finitos clásicos.

Diagonalización de σ

Antes de entrar en materia se verá de manera general como diagonalizar un operador con características más generales que el operador de trenza σ.

Sea T conjunto finito de cardinalidad n y considérese $L(T)$ como el espacio lineal generado por todas la combinaciones lineales de elementos en T,

$$L(T) = \left\{ \sum c \cdot t \;\middle|\; t \in T,\ c \in \mathbb{C} \right\}$$

los elementos de T resultan ser una base para $L(T)$.

Considérese la función $\Sigma : T \to T$ biyectiva. Existe única extensión lineal (denotada con el mismo símbolo) $\Sigma : L(T) \to L(T)$. Por la construcción, esta transformación lineal es biyectiva.

Ya que Σ es biyectiva la matriz M_Σ que representa Σ en la base dada por T resulta tener un único 1 por renglón y columna. Por ejemplo:

$$M_\Sigma = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 1 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

Observaciones:

1. La transformación $\Sigma : T \to T$ puede tener puntos fijos, es decir, $\Sigma(t) = t$ para algunos $t \in T$.

2. La transformación Σ tiene ciclos de orden a lo más n, es decir, para algún subcojunto $S = \{s_1, s_2, \ldots, s_n\}$ de T:

   $$\Sigma(s_1) = s_2, \Sigma(s_2) = s_3, \cdots, \Sigma(s_k) = s_1$$

   donde $2 \leq k \leq n$.

   Los elementos de los ciclos forman una partición del conjunto T en subconjuntos de cardinalidad k_i tal que $k_1 + \cdots + k_l = n$.

   Se denotará como Σ_k el ciclo interno donde k indica en qué paso regresa la función Σ.

Se calcularan los valores propios de la aplicación $\Sigma : L(T) \to L(T)$ considerando $|T| = n$ y sin puntos fijos.

Caso $n = 1$

Si $T = \{t_1\}$ se tiene el valor propio $\omega = 1$ el mapeo es trivial (identidad). Aquí se pueden considerar (y agrupar) cuando se cuenta con puntos fijos.

Caso $n = 2$

Si $T = \{t_1, t_2\}$ se elige[2]

$$\Sigma_2(t_1) = t_2,\ \Sigma_2(t_2) = t_1$$

la matriz asociada es:

$$M_{\Sigma_2} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Si se toman $\omega_1 = 1$ y $\omega_2 = -1$:

$$\Sigma_2\left[\frac{t_1 + t_2}{\sqrt{2}}\right] = \frac{t_2 + t_1}{\sqrt{2}} \Rightarrow \Sigma_2\left[\frac{t_1 + t_2}{\sqrt{2}}\right] = \omega_1\frac{t_1 + t_2}{\sqrt{2}}$$
$$\Sigma_2\left[\frac{t_1 - t_2}{\sqrt{2}}\right] = \frac{t_2 - t_1}{\sqrt{2}} \Rightarrow \Sigma_2\left[\frac{t_1 - t_2}{\sqrt{2}}\right] = \omega_2\frac{t_1 - t_2}{\sqrt{2}}$$

Así, ω_1 y ω_2 son valores propios de Σ_2.

Caso $n = 3$

Si $T = \{t_1, t_2, t_3\}$ un posible[3] arreglo para Σ_3:

$$\Sigma_3(t_1) = t_3,\ \Sigma_3(t_2) = t_1,\ \Sigma_3(t_3) = t_2$$

y

$$M_{\Sigma_3} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

Si $\omega_1 = 1$, $\omega_2 = (-1 - \sqrt{3}\imath)/2$, $\omega_3 = (-1 + \sqrt{3}\imath)/2$ raíces cúbicas de la unidad:

$$\Sigma_3\left[(t_1 + t_2 + t_3)/\sqrt{3}\right] = (t_3 + t_1 + t_2)/\sqrt{3} = \omega_1(t_1 + t_2 + t_3)/\sqrt{3} \qquad (3.1)$$

y

$$\Sigma_3\left[\left(t_1 + \frac{t_2}{\omega_2} + \frac{t_3}{\omega_2^2}\right)\Big/\sqrt{3}\right] = \left(t_3 + \frac{t_1}{\omega_2} + \frac{t_2}{\omega_2^2}\right)\Big/\sqrt{3}$$
$$= \frac{1}{\omega_2}\left(t_1 + \frac{t_2}{\omega_2} + \omega_2 t_3\right)\Big/\sqrt{3}$$

se sabe que $\omega_2^{-1} = \overline{\omega_2} = \omega_3$ y $\omega_2 = \omega_2^{-2}$, así se concluye:

$$\Sigma_3\left[\left(t_1 + \frac{t_2}{\omega_2} + \frac{t_3}{\omega_2^2}\right)\Big/\sqrt{3}\right] = \omega_3\left(t_1 + \frac{t_2}{\omega_2} + \frac{t_3}{\omega_2^2}\right)\Big/\sqrt{3} \qquad (3.2)$$

[2]La otra combinación resultaría en la identidad: $\Sigma_2(t_1) = t_1,\ \Sigma_2(t_2) = t_2$, excluyendo puntos fijos.

[3]Quitando los casos en que deja fijo algún elemento se tiene otro arreglo distinto: $\Sigma_3(t_1) = t_2,\ \Sigma_3(t_2) = t_3,\ \Sigma_3(t_3) = t_1$.

repitiendo el proceso con ω_3,

$$\Sigma_3\left[\left(t_1+\frac{t_2}{\omega_3}+\frac{t_3}{\omega_3^2}\right)\Big/\sqrt{3}\right]=\omega_2\left(t_1+\frac{t_2}{\omega_3}+\frac{t_3}{\omega_3^2}\right)\Big/\sqrt{3} \tag{3.3}$$

De las expresiones en 3.1, 3.2 y 3.3 se concluye que las raíces cúbicas de la unidad son valores propios para Σ_3.

En un caso más extenso para $T=\{t_1,t_2,\cdots,t_n\}$ y algún posible ordenamiento en sus elementos, sin que hayan puntos fijos, entonces la n-ésima raíz ω de la unidad cumple con ser valor propio para el operador Σ con el siguiente arreglo,

$$\Sigma_n\left[\left(t_1+t_2/\omega+\cdots+t_n/\omega^{n-1}\right)\,/\,\sqrt{n}\right]=\omega\left(t_1+t_2/\omega+\cdots+t_n/\omega^{n-1}\right)\,/\,\sqrt{n}$$

En general para un conjunto arbitrario finito T, éste puede contener subconjuntos invariantes bajo Σ, un caso particular son los puntos fijos. Si se nombran a éstos como ciclos, es natural una descomposición por ciclos del conjunto T bajo la acción Σ.

Sea $\Lambda\subset T$ tal que se cumple:

$$\Sigma(t_1)=t_2,\ \Sigma(t_2)=t_3,\ \cdots,\ \Sigma(t_k)=t_1$$

para $t_i\in\Lambda$.

Al conjunto Λ se le llamará ciclo por la propiedad que tiene bajo el operador Σ.

El conjunto T se descompone en sus ciclos Λ, disjuntos, tal que al tomar el subespacio $L(\Lambda)$ éste resulta ser T-invariante, además se tiene:

$$L(T)=\bigoplus_{\Lambda}L(\Lambda),\ \Lambda\in\text{Ciclos}(T)$$

En cada uno de los espacios $L(\Lambda)$ el operador Σ se diagonaliza como se ha explicado para cuando se tienen ciclos. Es decir, los valores propios de $\Sigma|_{L(\Lambda)}$ son las k-ésimas raíces de la unidad, donde $k=|\Lambda|$.

Al tomar $\Sigma=\sigma$, $T=S\times S$ consecuentemente $L(T)=L(S\times S)$, los valores propios de σ resultan ser las raíces de la unidad.

Descomposición en Órbitas de σ

Dado Λ ciclo en $S\times S$ se tiene:

$$\sigma(t_1)=t_2,\sigma(t_2)=t_3,\cdots,\sigma(t_k)=t_1$$

con $t_i\in\Lambda$.

Los valores propios para el operador σ sobre Λ resultan ser, como se ha visto, las k-ésimas raíces de la unidad. Para ilustrar véase lo siguiente:

$$\begin{aligned}
\sigma(t_1) &= t_2 \\
\sigma(\sigma(t_1)) &= \sigma(t_2) = t_3 \\
\sigma(\sigma^2(t_1)) &= \sigma(t_3) \\
\sigma(\sigma^3(t_1)) &= \sigma(t_4) \\
\vdots &= \vdots \\
\sigma(\sigma^{k-1}(t_1)) &= \sigma(t_k) \\
\sigma^k(t_1) &= t_1
\end{aligned}$$

Ya que el procedimiento anterior no depende de qué elemento se tome, siempre se tiene:

$$\sigma^k = \mathrm{id}$$

para elementos de Λ.

Ahora, si $S \times S$ se ve como la unión disjunta de sus ciclos, es decir,

$$S \times S = \Lambda_1 \cup \Lambda_2 \cup ... \cup \Lambda_j$$

y además

$$m_j = |\Lambda_j|$$

entonces

$$\sigma^m = \mathrm{id}, \text{ con } m = \mathrm{mcm}\{m_1, m_2, \cdots, m_j\}$$

Para ilustrar lo anterior tómese la siguiente matriz para alguna σ

$$M_\sigma = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

Aquí se tienen dos puntos fijos, los que corresponden a las esquinas de la matriz, y un ciclo de longitud 2.

Los valores para m_j son: $m_1 = 1, m_2 = 2$ y $m_3 = 1$. Con m =mcm$\{1, 2, 1\}$, en otras palabras, para este ejemplo en concreto el operador σ es la identidad en $m = 2$.

$$\sigma^2 = \mathrm{id}$$

Polinomio Mínimo de σ

Dado que el polinomio mínimo de un operador tiene las mismas raíces que el polinomio característico, una órbita o ciclo Λ de tamaño k dentro de $S \times S$ tendrá el siguiente polinomio:

$$p_\sigma(x) = (x - \omega_1)(x - \omega_2) \cdots (x - \omega_k)$$

con ω_i raíz de la unidad.

En general si $S \times S$ se ve como la unión de sus ciclos, el polinomio característico asociado al operador σ estará dado como el producto de los polinomios formados por cada ciclo,

$$p_\sigma(x) = p_1(x)p_2(x)\cdots p_j(x)$$

Se observa que si hay ciclos de la misma longitud, éstos se verán reflejados en las multiplicidades de sus raíces correspondientes. En particular la unidad tendrá siempre multiplicidad j, la misma que el total de ciclos existentes en $S\times S$.

Para la matriz M_σ en 3.4 el polinomio característico es:

$$p_\sigma(x) = (x-1)^3(x+1)$$

mientras el polinomio mínimo es:

$$p_\sigma(x) = (x-1)(x+1)$$

Ciclos-Órbitas

Teniendo M_σ se le asocia el vector V_M dado por:

$$V_M(j) = i, \qquad \text{si} \qquad M_\sigma(i,j) = 1. \tag{3.5}$$

El vector V_M resulta ser de permutación, por lo que las órbitas que tiene M_σ se ven reflejados en éste.

Véase lo siguiente, al tener un vector de permutaciones $V = (1\ 2\ \ldots\ n)$ éste puede ser descompuestos en permutaciones "locales" o ciclos.

Por ejemplo:

$$V_M = (1\ 4\ 8\ 2\ 6\ 5\ 9\ 3\ 7)$$

la permutación tiene 5 ciclos, listados por:

I) (**1** 4 8 2 6 5 9 3 7), $1 \to 1$.

II) (1 **4** 8 **2** 6 5 9 3 7), $2 \to 4 \to 2$.

III) (1 4 **8** 2 6 5 9 **3** 7), $3 \to 8 \to 3$.

IV) (1 4 8 2 **6 5** 9 3 7), $5 \to 6 \to 5$.

V) (1 4 8 2 6 5 **9** 3 **7**), $9 \to 7 \to 9$.

Y la matriz asociada es:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Como se puede observar se genera una partición de la permutación a través de sus ciclos, ello se ve reflejado en el siguiente resultado:

TEOREMA 1. ***Descomposición en ciclos disjuntos***
Toda permutación de P_n se puede descomponer en un producto de ciclos disjuntos. Esta descomposición es única, salvo el orden de los factores [23].

Con el teorema anterior y la relación que se describió previamente entre M_σ y V_M, teniendo los ciclos en V_M inmediatamente se tienen los ciclos en la matriz M_σ, por lo tanto las órbitas que se forman en $S \times S$.

3.5. Ejemplos

Luego de haber planteado el marco teórico sobre el cual este trabajo se basó, se dispone a presentar los grupos en los cuales se ejemplificaron los conceptos antes mencionados.

Dentro de la teoría de grupos se decidió trabajar sobre dos clases en específico. La primera consta de isometrías resultantes del tetraedro regular, cuadrado y triángulo. La segunda, los generados por las permutaciones de n elementos.

En el presente capítulo se toman algunos conjuntos para ir mostrando los conceptos descritos previamente, para detalle de las permutaciones o isometrías en el apartado de apéndices se desglosan.

3.5.1. Tetraedro

Considérese el tetraedro regular (Platónico), Figura 3.2, y las isometrías que se derivan de él por reflexiones y rotaciones, a partir de sus vértices y aristas.

Dichas transformaciones forman un grupo, el cual se describirá para efectos de ejemplificar los conceptos antes mencionados.

El primer grupo de transformaciones son las que consisten de, dado un vértice fijo rotar a la derecha el tetraedro. Figura 3.3. El segundo grupo es el que

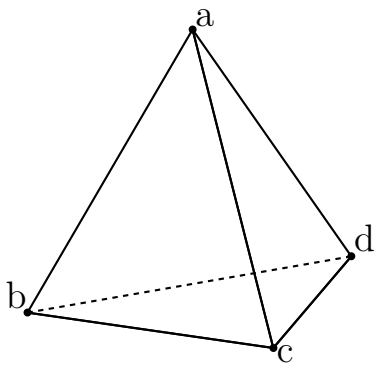


Figura 3.2: Tetraedro

tiene el giro a la izquierda, fijando igualmente un vértice. Figura 3.4.

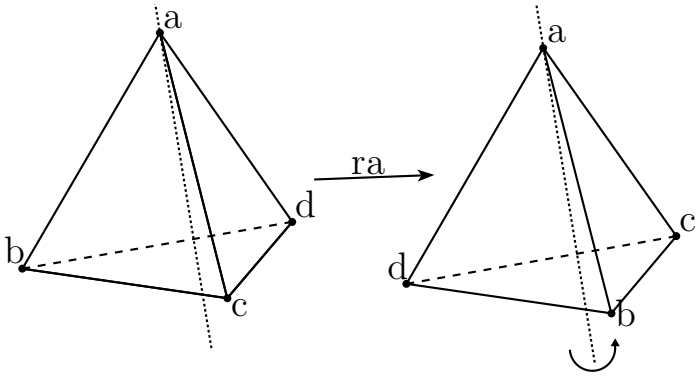


Figura 3.3: Isometría: *ra*

El último grupo de transformaciones son las rotaciones que vienen dado por aristas opuestas. Figura 3.5. En el apéndice A se tiene a detalle el resto de las transformaciones.

Al formar estas transformaciones un grupo, donde la operación es la composición de transformaciones se verá como se decidió manejarlas.

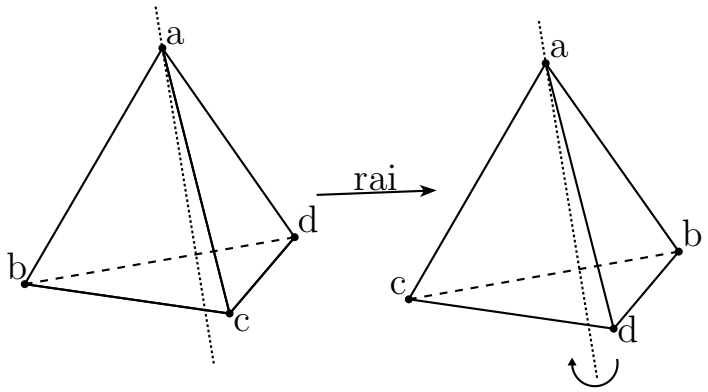


Figura 3.4: Isometría: *rai*

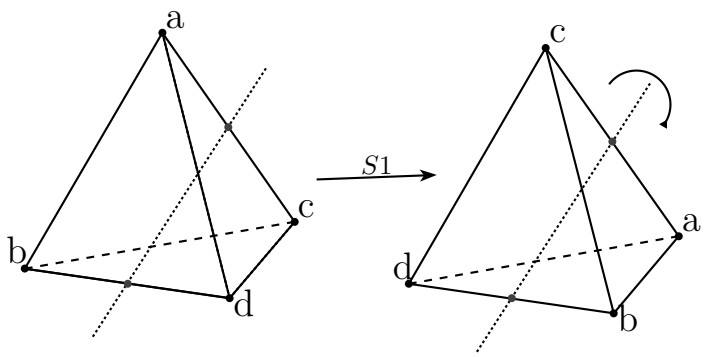


Figura 3.5: Isometría: s1

Tómese la primera transformación, listada, que gira a la derecha[4], denotada como ra, la cual tiene la siguiente regla de correspondencia entre los vértices del tetraedro:

$$ra : \begin{bmatrix} a & b & c & d \\ a & c & d & b \end{bmatrix}$$

lo que indica el arreglo es, a lo deja fijo, b lo coloca en la posición de c, c a la posición de d y d a la posición de b.

Así, el total de transformaciones del Tetraedro, están dadas por,

$$ra : \begin{bmatrix} a & b & c & d \\ a & c & d & b \end{bmatrix} rb : \begin{bmatrix} a & b & c & d \\ d & b & a & c \end{bmatrix} rc : \begin{bmatrix} a & b & c & d \\ b & d & c & a \end{bmatrix} rd : \begin{bmatrix} a & b & c & d \\ c & a & b & d \end{bmatrix}$$

$$rai : \begin{bmatrix} a & b & c & d \\ a & d & b & c \end{bmatrix} rbi : \begin{bmatrix} a & b & c & d \\ c & b & d & a \end{bmatrix} rci : \begin{bmatrix} a & b & c & d \\ d & a & c & b \end{bmatrix} rdi : \begin{bmatrix} a & b & c & d \\ b & c & a & d \end{bmatrix}$$

$$s1 : \begin{bmatrix} a & b & c & d \\ c & d & a & b \end{bmatrix} s2 : \begin{bmatrix} a & b & c & d \\ d & c & b & a \end{bmatrix} s3 : \begin{bmatrix} a & b & c & d \\ b & a & d & c \end{bmatrix}$$

Ahora se verá que éstas transformaciones son todas las posibles. Para ello considérese lo siguiente, hágase una correspondencia entre los vértices del tetraedro y los números 1,2,3 y 4 como sigue:

$$a \to 1,\ b \to 2,\ c \to 3,\ d \to 4$$

así, reescritas en términos de los números quedan como:

$$ra : \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 3 & 4 & 2 \end{bmatrix} rb : \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 2 & 1 & 3 \end{bmatrix} rc : \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 3 & 1 \end{bmatrix} rd : \begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 2 & 4 \end{bmatrix}$$

$$rai : \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 2 & 3 \end{bmatrix} rbi : \begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 2 & 4 & 1 \end{bmatrix} rci : \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 3 & 2 \end{bmatrix} rdi : \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 1 & 4 \end{bmatrix}$$

$$s1 : \begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \end{bmatrix} s2 : \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \end{bmatrix} s3 : \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \end{bmatrix}$$

$$e : \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

Se determinará la paridad de los elementos antes descritos para ver que constituyen el subgrupo alternante A_4 del grupo de permutaciones S_4 de cuatro elementos, [9], [10], [11], [12]. Para ello se tiene las siguientes definiciones

DEFINICIÓN. *Inversión*

Sea $\phi \in S_n$. Una inversión en ϕ es un par de índices (i, j) tal que $1 \leq i < j \leq n$ y $\phi(i) > \phi(j)$.

[4]Haciendo la nota que girar a la "derecha" puede ser relativo, pero se menciona así para efectos de ilustrar y como punto de referencia para el ejemplo.

DEFINICIÓN. *Inversiones de una permutación*

Dada $\phi \in S_n$, se denota como $\boldsymbol{inv}(\phi)$ el número de inversiones en ϕ, y está dada como:

$$\boldsymbol{inv}(\phi) = \sum_{i=1}^{n-1} |\{j > i : \phi(j) < \phi(i)\}|$$

DEFINICIÓN. *Se dice que una permutación es de orden par, si tiene un número par de inversiones.*

DEFINICIÓN. *Se dice que una permutación es de orden impar, si tiene un número impar de inversiones.*

En el área de apéndices está de manera explícita cómo cada una de las transformaciones listadas antes son pares.

Con la siguiente proposición [18] se garantiza que son 12 transformaciones, listadas ya, las únicas.

PROPOSICIÓN. *Hay tantas permutaciones pares como impares. Por lo tanto el orden del grupo alternante A_n es $n!/2$.*

Multiplicación

El conjunto antes descrito, compuesto por las rotaciones y reflexiones en el tetraedro, bajo la composición forma un grupo. Ahora se procede a describir la forma en que se trabajo esta operación entre los elementos.

Por ejemplo, $ra * s2 = rb$, viendo los diagramas se tiene:

$$ra * s2 = rb \;\Rightarrow\; \begin{bmatrix} a & b & c & d \\ a & c & d & b \end{bmatrix} * \begin{bmatrix} a & b & c & d \\ d & c & b & a \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ d & b & a & c \end{bmatrix}$$

Véase para el elemento a, ra lo deja intacto pero $s2$ lo coloca en la posición de d; es lo mismo que hace la transformación rb hace con a. Con el elemento b, ra lo coloca en la posición de c y al aplicar $s2$ ésta manda desde la posición de c a la posición de b, es decir, lo deja intacto, como rb. De forma similar con las dos restantes.

Si agregamos la transformación nuetra (identidad), la tabla de multiplicación está dada como:

	ra	rb	rc	rd	rai	rbi	rci	rdi	$s1$	$s2$	$s3$
ra	rai	rci	rdi	rbi	e	$s1$	$s2$	$s3$	rd	rb	rc
rb	rdi	rbi	rai	rci	$s1$	e	$s3$	$s2$	rc	ra	rd
rc	rbi	rdi	rci	rai	$s2$	$s3$	e	$s1$	rb	rd	ra
rd	rci	rai	rbi	rdi	$s3$	$s2$	$s1$	e	ra	rc	rb
rai	e	$s2$	$s3$	$s1$	ra	rd	rb	rc	rbi	rci	rdi
rbi	$s2$	e	$s1$	$s3$	rc	rb	rd	ra	rai	rdi	rci
rci	$s3$	$s1$	e	$s2$	rd	ra	rc	rb	rdi	rai	rbi
rci	$s1$	$s3$	$s2$	e	rb	rc	ra	rd	rci	rbi	rai
$s1$	rb	ra	rd	rc	rdi	rci	rbi	rai	e	$s3$	$s2$
$s2$	rc	rd	ra	rb	rbi	rai	rdi	rci	$s3$	e	$s1$
$s3$	rd	rc	rb	ra	rci	rdi	rai	rbi	$s2$	$s1$	e

Para el grupo de las isometrías del tetraedro se tiene que las correspondientes clases de conjugación son:

$$\{(e), (s1\ s2\ s3), (ra\ rb\ rc\ rd), (rai\ rbi\ rci\ rdi)\}$$

Conjuntos Perfectos

Para el caso particular de las isometrías, los conjuntos que son perfectos se pueden listar como las uniones de los siguientes subconjuntos:

- $\{s1, s2, s3\}$
- $\{ra, rb, rc, rd, rai, rbi, rci, rdi\}$

Nótese que cada uno tiene la clase de equivalencia correspondiente a cada elemento así como los inversos.

Vale la pena hacer notar que el conjunto $\{s1, s2, s3\}$ con la unidad e, es decir $\{e, s1, s2, s3\}$ representa por sí mismo un subgrupo invariante, el cual es conocido como grupo de Klein, un subgrupo Abeliano de A_4.

Su tabla de multipliación puede verse en la del grupo de isometrías listada arriba.

Operador σ

Tómese $S = \{s1, s2, s3\}$ conjunto perfecto. La relación dada por σ para cada elemento en S es:

$$\begin{array}{lll} \sigma(s1 \otimes s1) = s1 \otimes s1, & \sigma(s2 \otimes s1) = s1 \otimes s2, & \sigma(s3 \otimes s1) = s1 \otimes s3 \\ \sigma(s1 \otimes s2) = s2 \otimes s1, & \sigma(s2 \otimes s2) = s2 \otimes s2, & \sigma(s3 \otimes s2) = s2 \otimes s3 \\ \sigma(s1 \otimes s3) = s3 \otimes s1, & \sigma(s2 \otimes s3) = s3 \otimes s2, & \sigma(s3 \otimes s3) = s3 \otimes s3 \end{array}$$

Para construir la matriz M_σ asociada al conjunto S y obedeciendo la regla de correspondencia, antes mencionada, se puede visualizar como lo muestra la Tabla 3.2

	$s1\otimes s1$	$s1\otimes s2$	$s1\otimes s3$	$s2\otimes s1$	$s2\otimes s2$	$s2\otimes s3$	$s3\otimes s1$	$s3\otimes s2$	$s3\otimes s3$
$s1\otimes s1$	1	0	0	0	0	0	0	0	0
$s1\otimes s2$	0	0	0	1	0	0	0	0	0
$s1\otimes s3$	0	0	0	0	0	0	1	0	0
$s2\otimes s1$	0	1	0	0	0	0	0	0	0
$s2\otimes s2$	0	0	0	0	1	0	0	0	0
$s2\otimes s3$	0	0	0	0	0	0	0	1	0
$s3\otimes s1$	0	0	1	0	0	0	0	0	0
$s3\otimes s2$	0	0	0	0	0	1	0	0	0
$s3\otimes s3$	0	0	0	0	0	0	0	0	1

Tabla 3.2: Arreglo previo a M_σ.

El arreglo está construido de forma que las columnas se leen primero, es decir, el 1 en la primera entrada indica

$$\sigma(s1\otimes s1) = s1\otimes s1$$

el segundo 1 en el cuarto renglón es debido a:

$$\sigma(s1\otimes s2) = s2\otimes s1$$

Por la proposición antes mencionada, al ser éste un grupo Abeliano[5] resulta que la operación bajo el operador σ es la transposición usual:

$$\sigma(s\otimes t) = t\otimes s$$

con $s,t\in\{s1,s2,s3\}$.

De forma sucesiva se construye la matriz asociada a σ y S dada como:

$$M_\sigma = \begin{pmatrix} 1&0&0&0&0&0&0&0&0\\ 0&0&0&1&0&0&0&0&0\\ 0&0&0&0&0&0&1&0&0\\ 0&1&0&0&0&0&0&0&0\\ 0&0&0&0&1&0&0&0&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&1&0&0&0&0&0&0\\ 0&0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&0&0&1 \end{pmatrix} \tag{3.6}$$

Órbitas

Para la matriz en (3.6) el vector de permutación asociado es:

$$V_M = (1\ 4\ 7\ 2\ 5\ 8\ 3\ 6\ 9)$$

[5] Añadiendo e.

Los ciclos para el vector con los correspondientes elementos de S son:

I) $(\mathbf{1}\ 4\ 7\ 2\ 5\ 8\ 3\ 6\ 9) \Rightarrow \Lambda_1 = \{S1 \otimes S1\}$.

II) $(1\ \mathbf{4}\ 7\ \mathbf{2}\ 5\ 8\ 3\ 6\ 9) \Rightarrow \Lambda_2 = \{S2 \otimes S1, S1 \otimes S2\}$.

III) $(1\ 4\ \mathbf{7}\ 2\ 5\ 8\ \mathbf{3}\ 6\ 9) \Rightarrow \Lambda_3 = \{S3 \otimes S1, S1 \otimes S3\}$.

IV) $(1\ 4\ 7\ 2\ \mathbf{5}\ 8\ 3\ 6\ 9) \Rightarrow \Lambda_4 = \{S2 \otimes S2\}$.

V) $(1\ 4\ 7\ 2\ 5\ \mathbf{8}\ 3\ \mathbf{6}\ 9) \Rightarrow \Lambda_5 = \{S3 \otimes S2, S2 \otimes S3)\}$.

VI) $(1\ 4\ 7\ 2\ 5\ 8\ 3\ 6\ \mathbf{9}) \Rightarrow \Lambda_6 = \{S3 \otimes S3\}$.

Hay 6 ciclos, 3 de longitud 1 (puntos fijos) dados por $\Lambda_1, \Lambda_4, \Lambda_6$ y 3 de longitud 2 dados por $\Lambda_2, \Lambda_3, \Lambda_5$.

I) $\sigma(S1 \otimes S1) = S1 \otimes S1$.

II) $\sigma(S2 \otimes S1) = S1 \otimes S2$ y $\sigma(S1 \otimes S2) = S2 \otimes S1$.

III) $\sigma(S3 \otimes S1) = S1 \otimes S3$ y $\sigma(S1 \otimes S3) = S3 \otimes S1$.

IV) $\sigma(S2 \otimes S2) = S2 \otimes S2$.

V) $\sigma(S3 \otimes S2) = S2 \otimes S3$ y $\sigma(S2 \otimes S3) = S3 \otimes S2$.

VI) $\sigma(S3 \otimes S3) = S3 \otimes S3$.

De manera que las m_j son: $m_1 = m_4 = m_6 = 1$ y $m_2 = m_3 = m_5 = 2$. Con lo que $\text{mcm}\{1, 2\} = 2$, es decir, $\sigma^2 = id$.

El polinomio característico es:

$$p_\sigma(x) = (x+1)^3(x-1)^6$$

y el polinomio mínimo:

$$p_\sigma(x) = (x+1)(x-1)$$

Si se toma $S = \{ra, rb, rc, rd, rai, rbi, rci, rdi\}$ entonces $S \times S$ va a constar de 64 elementos al igual que el tamaño de la matriz M_σ será de 64×64. No se listaran ninguno de los dos conceptos por cuestión de espacio, basta comentar que utilizando los programas realizados, posteriormente descritos, se mostraran sólo los ciclos que tiene S.

$\Lambda_1 = \{ra \otimes ra\}$

$\Lambda_2 = \{rc \otimes ra, rb \otimes rc, ra \otimes rb\}$

$\Lambda_3 = \{rd \otimes ra, rc \otimes rd, ra \otimes rc\}$

$\Lambda_4 = \{rb \otimes ra, rd \otimes rb, ra \otimes rd\}$

$\Lambda_5 = \{rai \otimes ra, ra \otimes rai\}$

$$\Lambda_6 = \{rdi \otimes ra, rb \otimes rdi, rci \otimes rb, ra \otimes rci\}$$

$$\Lambda_7 = \{rbi \otimes ra, rc \otimes rbi, rdi \otimes rc, ra \otimes rdi\}$$

$$\Lambda_8 = \{rb \otimes rb\}$$

$$\Lambda_9 = \{rdi \otimes rb, rc \otimes rdi, rai \otimes rc, rb \otimes rai\}$$

$$\Lambda_{10} = \{rbi \otimes rb, rb \otimes rbi\}$$

$$\Lambda_{11} = \{rai \otimes rb, rd \otimes rai, rci \otimes rd, rb \otimes rci\}$$

$$\Lambda_{12} = \{rc \otimes rc\}$$

$$\Lambda_{13} = \{rbi \otimes rc, rd \otimes rbi, rai \otimes rd, rc \otimes rai\}$$

$$\Lambda_{14} = \{rci \otimes rc, rc \otimes rci\}$$

$$\Lambda_{15} = \{rd \otimes rd\}$$

$$\Lambda_{16} = \{rdi \otimes rd, rd \otimes rdi\}$$

$$\Lambda_{17} = \{rai \otimes rai\}$$

$$\Lambda_{18} = \{rdi \otimes rai, rbi \otimes rdi, rai \otimes rbi\}$$

$$\Lambda_{19} = \{rbi \otimes rai, rci \otimes rbi, rai \otimes rci\}$$

$$\Lambda_{20} = \{rci \otimes rai, rdi \otimes rci, rai \otimes rdi\}$$

$$\Lambda_{21} = \{rbi \otimes rbi\}$$

$$\Lambda_{22} = \{rdi \otimes rbi, rci \otimes rdi, rbi \otimes rci\}$$

$$\Lambda_{23} = \{rci \otimes rci\}$$

$$\Lambda_{24} = \{rdi \otimes rdi\}$$

$$\Lambda_{25} = \{rci \otimes ra, rd \otimes rci, rbi \otimes rd, ra \otimes rbi\}$$

$$\Lambda_{26} = \{rc \otimes rb, rd \otimes rc, rb \otimes rd\}$$

Como puede observarse en los ciclos, éstos no cumplen precisamente con la condición de transposición estándar para sus elementos.

Las longitudes de los ciclos son:

$1 = m_1 = m_8 = m_{12} = m_{15} = m_{17} = m_{21} = m_{23} = m_{24}.$

$2 = m_5 = m_{10} = m_{14} = m_{16}.$

$3 = m_2 = m_3 = m_4 = m_{18} = m_{19} = m_{20} = m_{22} = m_{26}.$

$4 = m_6 = m_7 = m_9 = m_{11} = m_{13} = m_{25}.$

Donde $12 = \text{mcm}\{1, 2, 3, 4\}$ por lo que $\sigma^{12} = \text{id}$.

El polinomio característico es:

$$p_\sigma(x) = [p_1(x)]^8[p_2(x)]^4[p_3(x)]^8[p_4(x)]^6$$

donde $p_k(x)$ es el polinomio que tiene las k-ésimas raíces de la unidad.

$$\begin{aligned}
p_1(x) &= x - 1 \\
p_2(x) &= (x-1)(x+1) \\
p_3(x) &= \left(x - \frac{\sqrt{3}\,i - 1}{2}\right)\left(x + \frac{\sqrt{3}\,i + 1}{2}\right)(x-1) \\
p_4(x) &= (x-i)(x+1)(x+i)(x-1)
\end{aligned}$$

Por lo tanto el polinomio mínimo es:

$$p_\sigma(x) = (x-1)(x+1)(x-i)(x+i)\left(x - \frac{\sqrt{3}\,i - 1}{2}\right)\left(x + \frac{\sqrt{3}\,i + 1}{2}\right)$$

3.5.2. Permutaciones

El segundo grupo se utilizará en los ejemplos es el de las permutaciones de n elementos.

Si X es un conjunto de cardinalidad n, una permutación se puede ver como una biyección de X en X.

La colección de todas las permutaciones de X forma un grupo S_X bajo la composición de funciones, si $\alpha : X \to X$ y $\beta : X \to X$ son permutaciones entonces $\alpha\beta : X \to X$ definida por $\alpha\beta(x) = \alpha(\beta(x))$ es una permutación.

Se llama S_n el grupo de permutaciones y es llamado *Grupo Simétrico* de grado n, con orden $n!$.

Al tomar $X = \{1, 2, \cdots, n\}$ se tiene:

$$\alpha(1\ 2\ 3\ \ldots\ i\ \ldots\ j\ \ldots\ n-1\ n) \to (1\ 2\ 3\ \ldots\ j\ \ldots\ i\ \ldots\ n-1\ n)$$

una permutación es un arreglo de la misma dimensión en la que se intercambian dos elementos cualesquiera de posición.

Ejemplos

Los primeros 4 grupos de permutaciones son los siguientes:

$S_1 = \{(1)\}$.

$S_2 = \{(1\ 2),\ (2\ 1)\}$.

$S_3 = \{(2\ 1\ 3),\ (2\ 3\ 1),\ (3\ 2\ 1),\ (1\ 2\ 3),\ (1\ 3\ 2),\ (3\ 1\ 2)\}$.

$S_4 = \{(3\ 1\ 2\ 4),\ (3\ 1\ 4\ 2),\ (3\ 4\ 1\ 2),\ (4\ 3\ 1\ 2),\ (1\ 3\ 2\ 4),\ (1\ 3\ 4\ 2),$
$(1\ 4\ 3\ 2),\ (4\ 1\ 3\ 2),\ (1\ 2\ 3\ 4),\ (1\ 2\ 4\ 3),\ (1\ 4\ 2\ 3),\ (4\ 1\ 2\ 3),\ (3\ 2\ 1\ 4),$
$(3\ 2\ 4\ 1),\ (3\ 4\ 2\ 1),\ (4\ 3\ 2\ 1),\ (2\ 3\ 1\ 4),\ (2\ 3\ 4\ 1),\ (2\ 4\ 3\ 1),\ (4\ 2\ 3\ 1),$
$(2\ 1\ 3\ 4),\ (2\ 1\ 4\ 3),\ (2\ 4\ 1\ 3),\ (4\ 2\ 1\ 3)\}$

Los elementos del grupo tienen la siguiente estructura al leerlos, si (3 1 2 4) se traduce como: el número 1 lo traslada a la posición 3, el número 2 lo traslada a la posición 1, el número 3 a la posición 2 y el número 4 a la posición 4.

De forma desglosada:

$$(3\ 1\ 2\ 4) = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 2 & 4 \end{bmatrix}$$

El elemento neutro en cada S_n está dado por $(1\ 2\ 3\ \ldots\ n)$.

Multiplicación

Si $x, y \in S_n$ entonces $xy \in S_n$, es decir, la multiplicación-composición de permutaciones es una permutación. Al operar xy se aplica primero y y posteriormente x. Véase el siguiente diagrama:

$$(3\ 1\ 2\ 4) * (4\ 2\ 1\ 3) = (4\ 1\ 3\ 2)$$

o de forma equivalente

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 2 & 4 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 2 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 3 & 2 \end{bmatrix}$$

Para S_4 y luego de realizar los cálculos se tienen las clases de conjugación del mismo.

$\{[(1\ 2\ 3\ 4)],\ [(3\ 4\ 1\ 2),\ (4\ 3\ 2\ 1),\ (2\ 1\ 4\ 3)],$
$[(4\ 1\ 2\ 3),\ (4\ 3\ 1\ 2),\ (2\ 3\ 4\ 1),\ (2\ 4\ 1\ 3),\ (3\ 1\ 4\ 2),\ (3\ 4\ 2\ 1)],$
$[(1\ 4\ 3\ 2),\ (1\ 2\ 4\ 3),\ (1\ 3\ 2\ 4),\ (3\ 2\ 1\ 4),\ (4\ 2\ 3\ 1),\ (2\ 1\ 3\ 4)],$
$[(1\ 4\ 2\ 3),\ (1\ 3\ 4\ 2),\ (2\ 4\ 3\ 1),\ (4\ 1\ 3\ 2),\ (2\ 3\ 1\ 4),\ (4\ 2\ 1\ 3),\ (3\ 2\ 4\ 1),\ (3\ 1\ 2\ 4)]\}$

Conjuntos Perfectos

Si se toman las Permutaciones de grado 3 los siguientes son algunos subconjuntos perfectos:

- $\{(2\ 3\ 1), (3\ 1\ 2)\}$.
- $\{(2\ 1\ 3), (1\ 3\ 2), (3\ 2\ 1)\}$.

Como se puede observar se encuentran todos los elementos del grupo, salvo la identidad que el propio concepto de ser conjunto perfecto excluye, así que las uniones de los subconjuntos también son perfectos. Nota: el grupo S_3 es isomorfo a las isometrías del triángulo equilátero.

Para las Permutaciones de grado 4 se tienen los siguientes conjuntos perfectos:

- $\{(2\ 1\ 4\ 3), (3\ 4\ 1\ 2), (4\ 3\ 2\ 1)\}$.
- $\{(3\ 4\ 2\ 1), (3\ 1\ 4\ 2), (4\ 1\ 2\ 3), (4\ 3\ 1\ 2), (2\ 4\ 1\ 3), (2\ 3\ 4\ 1)\}$.
- $\{(2\ 1\ 3\ 4), (4\ 2\ 3\ 1), (1\ 4\ 3\ 2), (1\ 3\ 2\ 4), (1\ 2\ 4\ 3), (3\ 2\ 1\ 4)\}$.
- $\{(1\ 3\ 4\ 2), (2\ 3\ 1\ 4), (4\ 1\ 3\ 2), (2\ 4\ 3\ 1), (3\ 1\ 2\ 4), (3\ 2\ 4\ 1), (4\ 2\ 1\ 3)$ $(1\ 4\ 2\ 3)\}$.

Vale la pena mencionar que el grupo S_4 es isomorfo al grupo de movimientos isométricos del cubo regular (o bien, su poliedro dual, el octaedro).

Operador σ

Si $S = \{(1\ 3\ 2)\ (2\ 1\ 3)\ (3\ 2\ 1)\}$ se tiene que $\sigma : L(S) \otimes L(S) \to L(S) \otimes L(S)$ actúa :

$$\sigma((1\ 3\ 2) \otimes (1\ 3\ 2)) = (1\ 3\ 2) \otimes (1\ 3\ 2); \quad \sigma((1\ 3\ 2) \otimes (2\ 1\ 3)) = (3\ 2\ 1) \otimes (1\ 3\ 2)$$
$$\sigma((1\ 3\ 2) \otimes (3\ 2\ 1)) = (2\ 1\ 3) \otimes (1\ 3\ 2); \quad \sigma((2\ 1\ 3) \otimes (1\ 3\ 2)) = (3\ 2\ 1) \otimes (2\ 1\ 3)$$
$$\sigma((2\ 1\ 3) \otimes (2\ 1\ 3)) = (2\ 1\ 3) \otimes (2\ 1\ 3); \quad \sigma((2\ 1\ 3) \otimes (3\ 2\ 1)) = (1\ 3\ 2) \otimes (2\ 1\ 3)$$
$$\sigma((3\ 2\ 1) \otimes (1\ 3\ 2)) = (2\ 1\ 3) \otimes (3\ 2\ 1); \quad \sigma((3\ 2\ 1) \otimes (2\ 1\ 3)) = (1\ 3\ 2) \otimes (3\ 2\ 1)$$
$$\sigma((3\ 2\ 1) \otimes (3\ 2\ 1)) = (3\ 2\ 1) \otimes (3\ 2\ 1).$$

Visto en forma de Tabla (3.3) y matricialmente la expresión 3.7.

	$(1\ 3\ 2)\otimes(1\ 3\ 2)$	$(1\ 3\ 2)\otimes(2\ 1\ 3)$	$(1\ 3\ 2)\otimes(3\ 2\ 1)$	$(2\ 1\ 3)\otimes(1\ 3\ 2)$	$(2\ 1\ 3)\otimes(2\ 1\ 3)$	$(2\ 1\ 3)\otimes(3\ 2\ 1)$	$(3\ 2\ 1)\otimes(1\ 3\ 2)$	$(3\ 2\ 1)\otimes(2\ 1\ 3)$	$(3\ 2\ 1)\otimes(3\ 2\ 1)$
$(1\ 3\ 2)\otimes(1\ 3\ 2)$	1	0	0	0	0	0	0	0	0
$(1\ 3\ 2)\otimes(2\ 1\ 3)$	0	0	0	0	0	1	0	0	0
$(1\ 3\ 2)\otimes(3\ 2\ 1)$	0	0	0	0	0	0	0	1	0
$(2\ 1\ 3)\otimes(1\ 3\ 2)$	0	0	1	0	0	0	0	0	0
$(2\ 1\ 3)\otimes(2\ 1\ 3)$	0	0	0	0	1	0	0	0	0
$(2\ 1\ 3)\otimes(3\ 2\ 1)$	0	0	0	0	0	0	1	0	0
$(3\ 2\ 1)\otimes(1\ 3\ 2)$	0	1	0	0	0	0	0	0	0
$(3\ 2\ 1)\otimes(2\ 1\ 3)$	0	0	0	1	0	0	0	0	0
$(3\ 2\ 1)\otimes(3\ 2\ 1)$	0	0	0	0	0	0	0	0	1

Tabla 3.3: Arreglo previo a M_S.

$$M_\sigma = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.7}$$

Órbitas

Dada la matriz (3.7) el vector V_M asociado es:

$$V_M = (1\ 7\ 4\ 8\ 5\ 2\ 6\ 3\ 9)$$

de donde se tiene los siguientes ciclos con los elementos correspondientes a S:

I) $(\mathbf{1}\ 7\ 4\ 8\ 5\ 2\ 6\ 3\ 9) \Rightarrow\ \Lambda_1 = \{(1\ 3\ 2)\otimes(1\ 3\ 2)\}$

II) $(1\ \mathbf{7}\ 4\ 8\ 5\ \mathbf{2}\ \mathbf{6}\ 3\ 9) \Rightarrow$
$\Lambda_2 = \{(3\ 2\ 1)\otimes(1\ 3\ 2), (1\ 3\ 2)\otimes(2\ 1\ 3), (2\ 1\ 3)\otimes(3\ 2\ 1)\}$

III) $(1\ 7\ \mathbf{4}\ \mathbf{8}\ 5\ 2\ 6\ \mathbf{3}\ 9) \Rightarrow$
$\Lambda_3 = \{(2\ 1\ 3)\otimes(1\ 3\ 2), (3\ 2\ 1)\otimes(2\ 1\ 3), (1\ 3\ 2)\otimes(3\ 2\ 1)\}$

IV) $(1\ 7\ 4\ 8\ 5\ 2\ 6\ 3\ \mathbf{9}) \Rightarrow\ \Lambda_4 = \{(3\ 2\ 1)\otimes(3\ 2\ 1)\}$

Los valores de las m_j son los siguientes:

$1 = \ m_1 = m_4$

$3 = \ m_2 = m_3$

De manera explícita la relación del operador σ para S es:

I) $\sigma((1\ 3\ 2) \otimes (1\ 3\ 2)) = (1\ 3\ 2) \otimes (1\ 3\ 2)$.

II)
- $\sigma((3\ 2\ 1) \otimes (1\ 3\ 2)) = (1\ 3\ 2) \otimes (2\ 1\ 3)$.
- $\sigma((1\ 3\ 2) \otimes (2\ 1\ 3)) = (2\ 1\ 3) \otimes (3\ 2\ 1)$.
- $\sigma((2\ 1\ 3) \otimes (3\ 2\ 1)) = (3\ 2\ 1) \otimes (1\ 3\ 2)$.

III)
- $\sigma((2\ 1\ 3) \otimes (1\ 3\ 2)) = (3\ 2\ 1) \otimes (2\ 1\ 3)$.
- $\sigma((3\ 2\ 1) \otimes (2\ 1\ 3)) = (1\ 3\ 2) \otimes (3\ 2\ 1)$.
- $\sigma((1\ 3\ 2) \otimes (3\ 2\ 1)) = (2\ 1\ 3) \otimes (1\ 3\ 2)$.

IV) $\sigma((3\ 2\ 1) \otimes (3\ 2\ 1)) = (3\ 2\ 1) \otimes (3\ 2\ 1)$.

Con $m = \text{mcm}\ \{1, 3\} = 3 \ \Rightarrow \sigma^3 = id$. Donde el polinomio característico es:

$$p_\sigma(x) = (x-1)^4(x-\omega)^2(x-\overline{\omega})^2, \qquad \omega = \frac{-1+\sqrt{3}\,i}{2}$$

y el polinomio mínimo es

$$p_\sigma(x) = (x-1)(x-\omega)(x-\overline{\omega})$$

Una vez que se presentó la teoría involucrada y los ejemplos correspondientes, se procede a detallar qué tópicos se programaron, mostrando el código con una descripción de las rutinas.

Capítulo 4

Módulo

En esta sección se describirán las rutinas que se realizaron en LISP y MAXIMA como parte de la implementación de la tesis.

El trabajo se enfoco en dos tópicos principales: cálculo de las clases de conjugación para un grupo, para poder determinar conjuntos perfectos S que definen los cálculos diferenciales. Así como el cálculo del operador σ sobre S y determinar las órbitas que tiene.

Las rutinas que conforman la primera parte requieren tres argumentos esenciales para ser ejecutadas:

1. El conjunto G.
2. Función Producto.
3. Función Inverso.

Definición del Grupo

La problemática que esencialmente tiene el Grupo es, la forma en que se plantea el mismo como argumento de entrada. Los programas fueron probados con dos clases, las permutaciones de n elementos y las isometrías[1] que se derivan del triángulo, cuadrado y tetraedro.

En el segundo caso se opto por capturar una lista de listas que consta de la definición de multiplicación de dos elementos del grupo, donde el primer miembro de la lista son todos los elementos del conjunto, empezando con el elemento neutro y definiéndolos previamente como variables simbólicas, el resto es para la multipiplicación de dos elementos.

[1] El apéndice A muestra con detalle dichas transformaciones y sus tablas de multiplicación.

Para el caso del triángulo se tiene:

Definición simbólica de cada elemento.

```
(setf rho 'rho)
(setf rhoi 'rhoi)
(setf sa 'sa)
(setf sb 'sb)
(setf sc 'sc)
(setf e 'e)
```

Definición de la tabla de multiplicación para el Grupo.

```
(setf list '((e rho rhoi sa sb sc)
  (rho rho rhoi) (rho rhoi e) (rho sa sc) (rho sb sa)
  (rho sc sb) (rhoi rho e) (rhoi rhoi rho) (rhoi sa sb)
  (rhoi sb sc) (rhoi sc sa) (sa rho sb) (sa rhoi sc)
  (sa sa e) (sa sb rhoi) (sa sc rho) (sb rho sc)
  (sb rhoi sa) (sb sa rho) (sb sb e) (sb sc rhoi)
  (sc rho sa) (sc rhoi sb) (sc sa rhoi) (sc sb rho)
  (sc sc e)))
```

Obsérvese que el primer elemento de la lista está compuesto por los integrantes del grupo, los demás son la multiplicación definida para los mismo. Por ejemplo si se tiene `(sc rhoi sb)` se entenderá `sc*rhoi = sb`.

Para el caso de las permutaciones se creo una función que las genera:

```
(permutation n)
```

Ejecutada desde LISP devuelve una lista de listas con las permutaciones correspondientes al tamaño dado, n.

```
> (permutation 4)
((3 1 2 4)(3 1 4 2)(3 4 1 2)(4 3 1 2)(1 3 2 4)
(1 3 4 2)(1 4 3 2)(4 1 3 2)(1 2 3 4)(1 2 4 3)
(1 4 2 3)(4 1 2 3)(3 2 1 4)(3 2 4 1)(3 4 2 1)
(4 3 2 1)(2 3 1 4)(2 3 4 1)(2 4 3 1)(4 2 3 1)
(2 1 3 4)(2 1 4 3)(2 4 1 3)(4 2 1 3))
```

Así, para estos ejemplos dichas listas son el argumento de entrada principal para las funciones que calculan las clases de conjugación.

Producto

El conjunto de rutinas está pensando para que el usuario pueda introducir su propio grupo, con sus respectivas reglas de producto y asignación de inversos.

Al trabajar con las isometrías mencionadas y por la forma en que se decidió capturar la lista con el grupo, la función Producto para las isometrías es:

```
(product-i x y list)
```

realiza la búsqueda de la pareja `x y` en la lista y devuelve `x*y`, tercer argumento de la triada.

```
> (product-i sa s2 list)
RHO
```

Al trabajar con el grupo de Permutaciones la función

```
(product-p x y)
```

efectúa para `x` y `y` la multiplicación-composición usual descrita previamente.

```
> (product-p '(3 2 4 1) '(2 4 1 3))
(2 1 3 4)
```

Inverso

Con el grupo derivado de las isometrías se tiene

```
(inverse-i x list)
```

dado `x` se busca al mismo en la primera posición de las listas y luego se verifica que en la tercera posisición se halle `e`, lo que quiere decir que el miembro en la segunda posición (`y`) es el inverso quien cumple con `x*y = e`.

```
> (inverse-i sa list)
SA
```

Al trabajar con las permutaciones la función

```
(inverse-p x)
```

da el inverso de `x` calculándolo de forma directa gracias a las relaciones que se tienen entre los elementos del grupo.

```
> (inverse-p '(3 2 4 1))
(4 2 1 3)
```

Todas las funciones se detallaran en el área de apéndices.

4.1. Clases de Conjugación

Las funciones que se crearon para calcular las clases de conjugación de un grupo G, tienen como argumentos de entrada, la lista con los elementos del grupo, un función Producto y una función Inverso.

Ya que, dependiendo del tipo de grupo que se agregue como entrada se requerirá una específica regla de producto para sus elementos y el cálculo de los inversos, se pensó dejar abierta la posibilidad y libertad al usuario de introducir sus propias funciones, debido a la clase de conceptos que se manejaron y sumado al hecho que LISP permite tener argumentos de entrada como funciones, los programas resultantes permitieron dejar todo el código lo más general posible.

En resonancia a como está pensado LISP permite conservar la estructura matemática que hay detrás de los problemas que se plantearon.

Como se describió previamente la relación $x \sim y$ dada por la conjugación de elementos en un grupo G, está dada por:

$$gxg^{-1} = y, \text{ para algún } g \in G.$$

Por lo que la primera función que se construye es `conjugado`, la cual dado $x \in G$ y algún $g \in G$ devuelve y conjugado de x por g. Ver Tabla 4.1.

La siguiente función se llama `conjuga`, la cual dado `x` $\in G$ encuentra la clase de conjugación del mismo, la Tabla 4.2 muestra su estructura.

Por último `conjugacy-class` que se encarga de almacenar las clases de conjugación del argumento (lista de elementos del grupo), devolviendo una lista de listas con las mismas. Dentro del cuerpo se llama una función dentro del contexto `labels` llamada `remove-sub`[2] que se encarga de remover subconjuntos de un conjunto. Véase Tabla 4.3.

Ejemplos

En la Tabla 4.4 muestra como se puede ejecutar desde LISP la función `conjugacy-class` con la sintaxis:

```
#'(lambda (x y) (product-i x y list))
```

y

```
#'(lambda (x) (inverse-i x list))
```

que define a las funciones `product-i` e `inverse-i` como argumentos con sus propias variables `x` y `y`. Cabe mencionar que al tener estas rutinas el argumento `list` que contiene la tabla de multiplicación, ésta se carga de manera global y se hace un `(car list)` para ver solamente los elementos del grupo.

[2]Detallada en el apartado de apéndices.

Con las permutaciones las funciones argumento son:

```
#'(lambda (x y) (product-p x y))
```

y

```
#'(lambda (x) (inverse-p x))
```

las cuales sólo requieren de los elementos a multiplicar o en su caso, el miembro del cual se requiere su inverso.

En la Tabla 4.5 y 4.6 se pueden ver los resultados que arroja `conjugacy-class` para los diferentes ejemplos.

4.2. Conjunto Perfecto

Este bloque ejemplifica cómo se construyo un conjunto perfecto dado cualquier subconjunto, sin identidad, del grupo G.

La función `close-set` es la que dado un subconjunto de un Grupo lo cierra bajo inversos y clases de conjugación, la Tabla 4.9 muestra su descripción. La Tabla 4.10 tiene el código para ejecutar en Lisp con los dos grupos descritos previamente y sus respectivas funciones de Producto e Inverso. Algunos ejemplos se pueden ver en la Tabla 4.11.

Predicado para un Conjunto Perfecto

Para efectos de cálculos posteriores se creo la función `perfect-set-p` predicado que indica si el conjunto (argumento) es perfecto. La Tabla 4.12 muestra su estructura.

Las Tablas 4.13 y 4.14 tienen ejemplos de los grupos trabajados.

4.3. Operador σ

La parte correspondiente al cálculo del producto tensorial consta de tres rutinas:

– `ptensorial`: Se encarga de calcular para $q, g \in S$, $q \otimes g$.

Las Tablas 4.15, 4.16, 4.17 nos muestran la sintaxis y ejemplos.

– `sub-tensorial`: Dado $q \in G$ y $S \subset G$, genera los $q \otimes g$, $\forall\ g \in S$.

Para cada $g \in S$ la rutina halla el conjunto $g \otimes S = \{q \otimes g \mid g \in S,\ q \in G\}$. La Tabla 4.18 muestra el código y los datos en las Tablas 4.19 y 4.20 son ejemplos y como se llama desde Lisp.

– `prod-tensorial`: Calcula $q \otimes g$, $\forall\ q, g \in S$ con $S \subset G$.

Por último la función que utiliza las dos anteriores, tiene como argumentos de entrada $S \subset G$ y realiza el producto tensorial entre los elementos de S, se entenderá que de aquí en adelante que S es perfecto.

La Tabla 4.21 tiene el código de dicha rutina, en la Tabla 4.23 se muestra como se llama en LISP y algunos ejemplos.

Matriz M_σ

La construcción de la matriz asociada a σ y S se formuló, para efectos de programación, de una forma especial. Ya que M_σ está compuesta de ceros y unos, por su regla de correspondencia (3.5), no es necesario construirla o almacenarla en su totalidad, en su lugar se trabaja con el vector V_M asociado, ver (3.5).

La función que calcula dicho vector es `sub-vec`:

```
sub-vec (plist &optional (produc #'identity) (inverse #'identity)
                (clist nil) (alist nil) (cont nil))
```

la cual es independiente en el sentido que sólo requiere como argumento (`plist` × `plist`), donde `plist` es un conjunto perfecto e internamente calcula el producto tensorial elemento a elemento, que no almacena, para ir construyendo el vector de permutaciones.

La Tabla 4.24 muestra el código de la rutina, para implementarla se construyeron dos rutinas auxiliares: `prod-cruz` y `cerolist`.[3]

La Tabla 4.26 tiene algunos ejemplos de los vectores de permutación para conjuntos Perfectos.

La rutina está pensada para ser ejecutada desde `Maxima` y desde ahí reconstruir la matriz M_S.

Órbitas

Para encontrar las órbitas del vector de permutaciones derivado de la matriz M_σ se construyo la rutina

`find-cycles`

La Tabla 4.27 muestra el código y la síntesis de sus funciones, en la Tabla 4.29 algunos ejemplos.

Una vez que se tienen los ciclos de un vector de permutación se pueden extraer los elementos de $S \times S$ que les corresponden mostrando la partición del conjunto. La rutina que realiza dicho calculo es:

[3] Detallas en el área de apéndices.

`cycles-class`

La Tabla 4.31 tiene el código y en la Tabla 4.33 los ejemplos.

Cabe mencionar que para efectos de programación, una vez elegido un conjunto S con cualquier orden, se sigue trabajando sobre la forma en que es introducido como argumento de entrada y varias rutinas dependen de ello.

Las siguientes son las tablas que contienen los códigos, descripciones, ejemplos y modos de uso de las principales rutinas.

Función: `conjugado`

```
(defun conjugado (g x &optional (produc #'identity) (inverse #'identity))
         (let ((y (funcall produc (funcall produc g x) (funcall inverse g)))) y))
```

Descripción:

La función calcula, dado un `x` $\in G$ (argumento principal) y algún `g` $\in G$ el conjugado de `x` por `g`.

Argumentos:

`x`	Elemento del grupo al cual se busca su conjugado.
`g`	Elemento del grupo por el cual se conjugara `x`.
`produc`	Argumento &opcional, Función Producto específica para el grupo.
`inverse`	Argumento &opcional, Función Inverso específica para el grupo.

Tabla 4.1: Función: `conjugado`

Función: `conjuga`

```
(defun conjuga (x list &optional (produc #'identity) (inverse #'identity) (nlist nil))
  (cond  ((null list) nlist)
         (T (let* ((elem (conjugado (car list) x produc inverse))
          (nlist (find-insert elem nlist)))
         (conjuga x (cdr list) produc inverse nlist)))))
```

Descripción:

La función devuelve la clase de conjugación del elemento `x`.

Argumentos:	
`x`	Elemento del grupo al cual se quiere encontrar su clase de conjugación.
`list`	Lista con los elementos del grupo.
`produc`	Argumento &opcional, Función Producto específica para el grupo.
`inverse`	Argumento &opcional, Función Inverso específica para el grupo.

Tabla 4.2: Función: `conjuga`

Función: `conjugacy-class`

```
(defun conjugacy-class (list &optional (produc #'identity) (inverse #'identity) (nlist nil) (alist nil))
  (labels ((remove-sub (list sublist)
    (cond ((null sublist) list)
      (T (if (equalp (car list) (car sublist))
        (let* ((sublist (cdr sublist))
          (list (cdr list)))
        (remove-sub list sublist))
      (let* ((list (append (cdr list) (list (car list)))))
        (remove-sub list sublist)))))))
  (cond ((null list) nlist)
    (T (if (null alist)
      (let* ((alist list)
        (clist (conjuga (car alist) alist produc inverse))
          (list (remove-sub list clist))
          (nlist (push clist nlist)))
        (conjugacy-class list produc inverse nlist alist))
      (let* ((clist (conjuga (car list) alist produc inverse))
          (list (remove-sub list clist))
          (nlist (push clist nlist)))
        (conjugacy-class list produc inverse nlist alist)))) )))
```

Descripción:
La función esencialmente lo que realiza es, para los elementos del grupo extraer las clases de conjugación. Gracias a que éstas forman una partición la función optimiza el procedimiento, al obtener una clase la retira del grupo y sigue la búsqueda con los elementos restantes.

Argumentos:	
`list`	Lista con los elementos del grupo.
`produc`	Argumento &opcional, Función Producto específica para el grupo.
`inverse`	Argumento &opcional, Función Inverso específica para el grupo.
`nlist`	Argumento auxiliar que va almacenando las clases de conjugación.
`alist`	Argumento auxiliar que guarda al grupo completo.

Tabla 4.3: Función: `conjugacy-class`

Ejemplos:

Isometrías.

```
>(conjugacy-class (car list) #'(lambda (x y) (product-i x y list)) #'(lambda (x) (inverse-i x list)))
```

Permutaciones

```
>(conjugacy-class list #'(lambda (x y) (product-p x y)) #'(lambda (x) (inverse-p x)))
```

Tabla 4.4: Código ejecutado desde `Lisp`.

Ejemplos de Clases de Conjugación derivados de Isometrías.

Figura	Grupo	Clases de Conjugación
Triángulo	`(e rho rhoi sa sb sc)`	`((SC SB SA) (RHOI) (RHO) (E))`
Cuadrado	`(e rho rhoi rho2 sa sb s1 s2)`	`((S2 S1) (SB SA) (RHO2) (RHOI RHO) (E))`
Tetraedro	`(e ra rb rc rd rai rbi rdi rci s1 s2 s3)`	`((S3 S2 S1) (RCI RBI RDI RAI) (RC RB RD RA) (E))`

Tabla 4.5: Ejemplos para las Isometrías.

Ejemplos de Clases de Conjugación derivadas de grupos de Permutaciones.

Elementos	Clases de Conjugación
2	`(((2 1)) ((1 2)))`
3	`(((1 2 3)) ((2 3 1) (3 1 2)) ((3 2 1) (1 3 2) (2 1 3)))`
4	`(((1 2 3 4)) ((1 4 3 2) (1 2 4 3) (1 3 2 4) (3 2 1 4) (4 2 3 1) (2 1 3 4)) ((3 4 1 2) (4 3 2 1) (2 1 4 3)) ((4 1 2 3) (4 3 1 2) (2 3 4 1) (2 4 1 3) (3 1 4 2) (3 4 2 1)) ((1 4 2 3) (1 3 4 2) (2 4 3 1) (4 1 3 2) (2 3 1 4) (4 2 1 3) (3 2 4 1) (3 1 2 4)))`

Tabla 4.6: Ejemplos para las Permutaciones.

Función: `add-inverse`

```
(defun add-inverse (slist list &optional (inverse #'identity) (clist slist))
   (labels ((seek-elem (elem list &optional (nlist nil))
          (cond ((null elem) nlist)
            ((null list) (cons elem nlist))
            (T (if (equalp elem (car list))
              (let ((nlist (append list nlist)))
                (seek-elem nil list nlist))
              (let* ((nlist (cons (car list) nlist))
                   (list (cdr list)))
                (seek-elem elem list nlist)))))))
   (cond ((null slist) clist)
          (T (let* ((elem (funcall inverse (car slist)))
               (clist (seek-elem elem clist)))
          (add-inverse (cdr slist) list inverse clist)) ))))
```

Descripción:

Dado el subconjunto `slist` del conjunto `list`, agrega los inversos de cada elemento de `slist` al mismo.

Argumentos:

`slist`	Lista con elementos de `list`.
`list`	Lista con los elementos del grupo.
`inverse`	Argumento &optional, Función Inverso específica para el grupo.
`clist`	Argumento auxilar que almacena la cerradura de slist.

Tabla 4.7: Función que agrega inversos a un subconjunto de un grupo.

Función: `add-class`

```
(defun add-class (slist list &optional (produc #'identity) (inverse #'identity) (clist nil))
   (cond ((null slist) clist)
         (T (let* ((clist (union (conjuga (car slist) list produc inverse) clist)))
              (add-class (cdr slist) list produc inverse clist)))))
```

Descripción:

Dado el subconjunto `slist` de `list`, la función agrega las clases de conjugación de cada elemento en `slist`.

Argumentos:

`slist`	Subconjunto de un Grupo.
`list`	Lista que contiene los elementos de un grupo.
`produc`	Argumento &opcional, Función Producto específica para el grupo.
`inverse`	Argumento &opcional, Función Inverso específica para el grupo.
`clist`	Argumento auxiliar donde se almacena el grupo cerrado bajo clases de conjugación.

Tabla 4.8: Función que cierra un conjunto bajo clases de conjugación.

Función: `close-set`

```
(defun close-set (slist list &optional (produc #'identity) (inverse #'identity))
   (add-inverse (add-class slist list produc inverse) list inverse))
```

Descripción:

Cierra el subconjunto `slist` bajo inversos y clases de conjugación de un grupo.

Argumentos:	
`slist`	Subconjunto de un Grupo.
`list`	Lista que contiene los elementos de un grupo.
`produc`	Argumento &opcional, Función Producto específica para el grupo.
`inverse`	Argumento &opcional, Función Inverso específica para el grupo.

Tabla 4.9: Función que cierra un subconjunto bajo inversos y clases de conjugación.

Ejemplos:

Isometrías.
`>(close-set slist (car list) #'(lambda (x y) (product-i x y list)) #'(lambda (x) (inverse-i x list)))`
Permutaciones
`>(close-set slist list #'(lambda (x y) (product-p x y)) #'(lambda (x) (inverse-p x)))`

Tabla 4.10: Código ejecutado desde `Lisp`.

Grupo de Isometrías del Tetraedro.	
Subconjunto	Cerradura
`(s3)`	`(S2 S3 S1)`
`(rb)`	`(RAI RCI RB RC RD RA RBI RDI)`
`(ra rbi s2)`	`(RCI RDI RAI RA RD RC S2 S3 RB S1 RBI)`

Grupo de permutaciones de 3 elementos.	
Subconjunto	Cerradura
`((3 1 2))`	`((2 3 1) (3 1 2))`
`((2 1 3))`	`((3 2 1) (2 1 3) (1 3 2))`

Grupo de permutaciones de 4 elementos.	
Subconjunto	Cerradura
`((1 2 4 3))`	`((3 2 1 4) (1 4 3 2) (2 1 3 4) (1 3 2 4) (1 2 4 3) (4 2 3 1))`
`((2 3 1 4))`	`((1 4 2 3) (4 2 1 3) (3 2 4 1) (3 1 2 4) (2 4 3 1) (4 1 3 2) (2 3 1 4) (1 3 4 2))`
`((2 3 4 1) (1 4 3 2))`	`((2 3 4 1) (4 3 1 2) (3 4 2 1) (4 1 2 3) (3 1 4 2) (4 2 3 1) (2 1 3 4) (1 4 3 2) (1 3 2 4) (1 2 4 3) (3 2 1 4) (2 4 1 3))`

Tabla 4.11: Ejemplos de subconjuntos que se cierran bajo Inversos y Clases de Conjugación.

Función: `perfect-set-p`

`(defun perfect-set-p (set group &optional (produc #'identity) (inverse #'identity))`
`  (let ((pset (close-set set group produc inverse)))`
`  (if (equal-list-p set pset) T nil)))`

Descripción:
Verifica si un conjunto es Perfecto, devolviendo T, en caso contrario NIL. Internamente genera la cerradura de `set` y prueba si es igual a él mismo.
Argumentos:
`set` Subconjunto del grupo principal.
`group` Grupo.
`produc` Argumento &opcional, Función Producto específica para el grupo.
`inverse` Argumento &opcional, Función Inverso específica para el grupo.

Tabla 4.12: Predicado para verificar si un conjunto es Perfecto.

Ejemplos:

Isometrías.
`>(perfect-set-p set group #'(lambda (x y) (product-i x y list)) #'(lambda (x) (inverse-i x list)))`
Permutaciones
`>(perfect-set-p set group #'(lambda (x y) (product-p x y)) #'(lambda (x) (inverse-p x)))`

Tabla 4.13: Código ejecutado desde `Lisp`.

Grupo de Isometrías del Tetraedro.	
Subconjunto	Predicado
`(s3)`	`NIL`
`(ra rb rc rd)`	`NIL`
`(s1 s2 s3)`	`T`
`(ra rb rc rd rai rbi rci rdi)`	`T`

Grupo de permutaciones de 3 elementos.	
Subconjunto	Predicado
`((2 3 1))`	`NIL`
`((2 3 1) (3 1 2))`	`T`
`((2 1 3) (1 3 2) (2 3 1) (3 2 1) (3 1 2))`	`T`

Grupo de permutaciones de 4 elementos.	
Subconjunto	Predicado
`((1 3 2 4) (3 2 1 4) (4 2 3 1) (2 1 3 4) (1 4 3 2))`	`T`
`((4 3 1 2) (1 3 2 4))`	`NIL`
`((3 1 4 2) (2 1 3 4) (4 2 3 1) (3 2 1 4) (1 3 2 4) (1 2 4 3) (1 4 3 2) (2 3 4 1) (4 3 1 2) (3 4 2 1) (4 1 2 3) (2 4 1 3))`	`T`

Tabla 4.14: Ejemplos del Predicado `perfect-set-p`.

Función: `ptensorial`

`(defun ptensorial (q g &optional (produc #'identity) (inverse #'identity))` `(list (conjugado q g produc inverse) q))`
Descripción:
Dados `q` y `g` elementos de un grupo, calcula `q` $\otimes$ `g`.
Argumentos:
`q` Elemento de un grupo dado.
`g` Elemento de un grupo dado.
`produc` Argumento &opcional, Función Producto específica para el grupo.
`inverse` Argumento &opcional, Función Inverso específica para el grupo.

Tabla 4.15: Función que calcula el producto tensorial entre dos elementos de un subconjunto perfecto.

Ejemplos:

Isometrías.

```
>(ptensorial q g #'(lambda (x y) (product-i x y list)) #'(lambda (x) (inverse-i x list)))
```

Permutaciones

```
>(ptensorial q g #'(lambda (x y) (product-p x y)) #'(lambda (x) (inverse-p x)))
```

Tabla 4.16: Código ejecutado desde `Lisp`.

Grupo de Isometrías del Cuadrado				Grupo de Isometrías del Tetraedro.		
q	g	$q \otimes g$		q	g	$q \otimes g$
e	sa	(SA E)		s3	s2	(S2 S3)
rho	rho	(RHO RHO)		ra	s1	(S2 RA)
rho2	sb	(SB RHO2)		rci	rci	(RCI RCI)
s1	s2	(S2 S1)		s3	rdi	(RCI S3)

Grupo de Permutaciones de 3 elementos				Grupo de Permutaciones de 4 elementos.		
q	g	$q \otimes g$		q	g	$q \otimes g$
(2 3 1)	(3 1 2)	((3 1 2) (2 3 1))		(2 3 1 4)	(3 1 4 2)	((4 1 2 3) (2 3 1 4))
(1 2 3)	(3 1 2)	((3 1 2) (1 2 3))		(4 3 2 1)	(4 3 2 1)	((4 3 2 1) (4 3 2 1))
(1 3 2)	(1 3 2)	((1 3 2) (1 3 2))		(3 4 1 2)	(1 2 3 4)	((1 2 3 4) (3 4 1 2))

Tabla 4.17: Ejemplos de producto tensorial.

Función: `sub-tensorial`

```
(defun sub-tensorial (q set &optional (produc #'identity) (inverse #'identity) (nlist nil))
  (cond ((null set) nlist)
     (T (if (null nlist)
       (let ((nlist (push (ptensorial q (car set) produc inverse) nlist)))
         (sub-tensorial q (cdr set) produc inverse nlist))
       (let ((nlist (push (ptensorial q (car set) produc inverse) nlist)))
        (sub-tensorial q (cdr set) produc inverse nlist))))))
```

Tabla 4.18: Código para el cálculo del conjunto $q \otimes S$, con $q \in G$.

Descripción:
Dado un elemento `q` de un grupo y `set` subconjunto del mismo también, calcula el producto tensorial `q` $\otimes$ `g`, para todo `g` $\in$ `set`.

Argumentos:	
`q`	Elemento de un grupo.
`set`	Subconjunto de un grupo.
`produc`	Argumento &opcional, Función Producto específica para el grupo.
`inverse`	Argumento &opcional, Función Inverso específica para el grupo.
`nlist`	Argumento auxiliar que va almacenando el producto tensorial de `q` con `set`.

Ejemplos:

Isometrías.

```
>(sub-tensorial q set #'(lambda (x y) (product-i x y list)) #'(lambda (x) (inverse-i x list)))
```

Permutaciones

```
>(sub-tensorial q set #'(lambda (x y) (product-p x y)) #'(lambda (x) (inverse-p x)))
```

Tabla 4.19: Código ejecutado desde `Lisp`.

Grupo de Isometrías del Tetraedro.

q	S	$q \otimes S$
ra	(s1 s2 s3)	((S1 RA) (S3 RA) (S2 RA))
rci	(ra s1 rdi e)	((E RCI) (RBI RCI) (S3 RCI) (RD RCI))
s1	(ra rb rc rd)	((RB S1) (RA S1) (RD S1) (RC S1))

Grupo de Permutaciones de 3 elementos.		
q	S	$q \otimes S$
(3 1 2)	((3 2 1) (1 2 3))	(((1 2 3) (3 1 2)) ((1 3 2) (3 1 2)))
(2 1 3)	((2 3 1) (1 2 3) (3 2 1))	(((1 3 2) (2 1 3)) ((1 2 3) (2 1 3)) ((3 1 2) (2 1 3)))

Tabla 4.20: Ejemplos del producto tensorial $q \otimes S$, con $q \in G,\ S \subset G$.

Función: `prod-tensorial`

```
(defun prod-tensorial (set &optional (produc #'identity) (inverse #'identity) (nlist nil) (alist nil))
        (cond ((null set) (reverse nlist))
          (T (if (null nlist)
            (let ((alist set)
              (nlist (append (sub-tensorial (car set) set produc inverse) nlist)))
            (prod-tensorial (cdr set) produc inverse nlist alist))
          (let ((nlist (append (sub-tensorial (car set) alist produc inverse) nlist)))
            (prod-tensorial (cdr set) produc inverse nlist alist))))))
```

Descripción:

Función que calcula, dado $S \subset G$, $S \times S$. S tiene que ser un conjunto Perfecto.
Los elementos dados como $q, p \in S \times S$ se hacen corresponder con $q \otimes p \in L(S) \otimes L(S)$.

Argumentos:

`set`	Subconjunto Perfecto de un grupo dado.
`produc`	Argumento &opcional, Función Producto específica para el grupo.
`inverse`	Argumento &opcional, Función Inverso específica para el grupo.
`nlist`	Argumento auxiliar encargado de ir almacenando el producto tensorial de `set`.
`alist`	Argumento auxiliar que almacena los datos originales de `set`.

Tabla 4.21: Función que calcula en producto tensorial entre los elementos de S.

Ejemplos:

Isometrías.

```
>(prod-tensorial set #'(lambda (x y) (product-i x y list)) #'(lambda (x) (inverse-i x list)))
```

Permutaciones

```
>(prod-tensorial set #'(lambda (x y) (product-p x y)) #'(lambda (x) (inverse-p x)))
```

Tabla 4.22: Código ejecutado desde `Lisp`.

Grupo de Isometrías del Tetraedro.	
S	$S \times S$ bajo la relación $(q, p) \leftrightarrow q \otimes p$.
`(e s1 s2 s3)`	`((E E) (S1 E) (S2 E) (S3 E) (E S1) (S1 S1) (S2 S1) (S3 S1) (E S2) (S1 S2) (S2 S2) (S3 S2) (E S3) (S1 S3) (S2 S3) (S3 S3))`
`(ra rb rc rd rai rbi rci rdi)`	`((RBI RBI) (RCI RBI) (RD RBI) (RB RBI) (RA RBI) (RC RBI) (RAI RBI) (RDI RBI) (RDI RAI) (RAI RAI) (RB RAI) (RD RAI) (RC RAI) (RA RAI) (RCI RAI) (RBI RAI) (RDI RC) (RBI RC) (RC RC) (RD RC) (RA RC) (RB RC) (RAI RC) (RCI RC) (RBI RB) (RDI RB) (RA RB) (RB RB) (RC RB) (RD RB) (RCI RB) (RAI RB) (RAI RD) (RCI RD) (RB RD) (RA RD) (RD RD) (RC RD) (RDI RD) (RBI RD) (RCI RA) (RAI RA) (RD RA) (RC RA) (RB RA) (RA RA) (RBI RA) (RDI RA) (RCI RDI) (RBI RDI) (RA RDI) (RC RDI) (RD RDI) (RB RDI) (RDI RDI) (RAI RDI) (RAI RCI) (RDI RCI) (RC RCI) (RA RCI) (RB RCI) (RD RCI) (RBI RCI) (RCI RCI))`
Grupo de Permutaciones de 3 elementos.	
`((2 1 3) (1 3 2) (3 2 1))`	`(((2 1 3) (2 1 3)) ((3 2 1) (2 1 3)) ((1 3 2) (2 1 3)) ((3 2 1) (1 3 2)) ((1 3 2) (1 3 2)) ((2 1 3) (1 3 2)) ((1 3 2) (3 2 1)) ((2 1 3) (3 2 1)) ((3 2 1) (3 2 1)))`

Tabla 4.23: Ejemplos del producto tensorial para $S \subset G$ Perfecto.

Función:`sub-vec`

```
(defun sub-vec (plist &optional (produc #'identity) (inverse #'identity) (clist nil) (alist nil) (cont nil))
          (cond ((null plist) clist)
            (T (if (null alist)
              (let ((alist plist)
                   (cont 0))
                (sub-vec plist produc inverse clist alist cont))
              (let* ((elem (ptensorial (caar plist) (cadar plist) produc inverse))
                   (pos (pos-elem elem alist))
                   (cont (+ cont 1))
                   (clist (insert pos clist cont)))
              (sub-vec (cdr plist) produc inverse clist alist cont))))))
```

Descripción:

Rutina que dado el producto cruz del conjunto Perfecto S calcula el vector V_M asociado a la transformación σ y el conjunto S.

Argumentos:

`plist`	Producto cruz de un conjunto perfecto.
`produc`	Argumento &opcional, Función Producto específica para el grupo.
`inverse`	Argumento &opcional, Función Inverso específica para el grupo.
`clist`	Argumento auxiliar encargado de ir almacenando el vector V_M. Inicializado con un vector de ceros igual a la dimensión del conjunto Perfecto.
`alist`	Argumento auxiliar igual a `plist` encargado de almacenar los datos originales de éste.
`cont`	Contador interno que va avanzando sobre las entradas del vector V_M.

Tabla 4.24: Código: `sub-vec`.

Ejemplos:

Isometrías.

```
>(sub-vec plist #'(lambda (x y) (product-i x y list)) #'(lambda (x) (inverse-i x list)) clist)
```

Permutaciones

```
>(sub-vec plist #'(lambda (x y) (product-p x y)) #'(lambda (x) (inverse-p x)) clist)
```

Tabla 4.25: Código ejecutado desde `Lisp`.

Grupo de Isometrías del Tetraedro.	
S	V_M
`(s1 s2 s3)`	(1 4 7 2 5 8 3 6 9)
`(ra rb rd rc rai rbi rci rdi)`	(1 17 25 9 33 49 57 41 26 10 2 18 58 42 34 50 11 27 19 3 43 59 51 35 20 4 12 28 52 36 44 60 5 29 13 21 37 61 45 53 22 14 30 6 54 46 62 38 31 7 23 15 63 39 55 47 16 24 8 32 48 56 40 64)
Grupo de Permutaciones de 3 elementos.	
`(2 3 1) (3 1 2)`	(1 3 2 4)
`(1 3 2) (2 1 3)`	(1 7 4 8 5 2 6 3 9)
`(3 2 1)`	
P_3	(1 31 25 19 13 7 26 8 2 20 14 32 27 33 15 21 3 9 4 10 16 22 28 34 17 35 5 23 29 11 18 12 30 24 6 36)

Tabla 4.26: Ejemplos del Vector V_M.

Función: `find-cycles`

```
(defun find-cycles (vec &optional (pvec nil) (auxvec nil) (pos 1))
   (labels ((f-nonzero (list &optional (pos 1))
         (cond ((null (equalp (car list) 0)) pos)
           (T (let ((pos (1+ pos)))
             (f-nonzero (cdr list) pos))))))
     (cond ((zero-list-p vec) (push auxvec pvec))
         (T (if (= (nth (1- pos) vec) 0)
           (find-cycles vec (push auxvec pvec) nil (f-nonzero vec))
           (find-cycles (insert 0 vec pos) pvec (push (nth (1- pos) vec) auxvec) (nth (1- pos) vec)))))))
```

Descripción:

La función halla, dado un vector V_M, los ciclos que hay dentro del mismo. Dentro tiene una función auxiliar `f-nonzero` que devuelve la posición del primer elemento distinto de cero en su argumento, `list`.

Argumentos:

`vec`	Vector V_M.
`pvec`	Argumento auxiliar encargado de almacenar los ciclos que se van encontrando.
`auxvec`	Argumento auxiliar encargado de colectar los elementos de un ciclo.
`pos`	Contador interno para ubicar los elementos del vector V_M.

Tabla 4.27: Código: `find-cycles`.

```
(find-cycles vec)
```

Tabla 4.28: Código ejecutado desde `Lisp`.

Grupo de Isometrías del Tetraedro.	
V_M	Ciclos
(1 4 7 2 5 8 3 6 9)	((1) (2 4) (3 7) (5) (6 8) (9))
(1 17 25 9 33 49 57 41 26 10 2 18 58 42 34 50 11 27 19 3 43 59 51 35 20 4 12 28 52 36 44 60 5 29 13 21 37 61 45 53 22 14 30 6 54 46 62 38 31 7 23 15 63 39 55 47 16 24 8 32 48 56 40 64)	((1) (2 11 17) (3 20 25) (4 26 9) (5 33) (6 44 31 49) (7 50 16 57) (8 59 22 41) (10) (12 27 18) (13 35 24 58) (14 42) (15 52 29 34) (19) (21 36 30 43) (23 51) (28) (32 60) (37) (38 48 61) (39 54 45) (40 63 53) (46) (47 56 62) (55) (64))
Grupo de Permutaciones de 3 elementos.	
(1 3 2 4)	((1) (2 3) (4))
(1 7 4 8 5 2 6 3 9)	((1) (2 6 7) (3 8 4) (5) (9))
(1 31 25 19 13 7 26 8 2 20 14 32 27 33 15 21 3 9 4 10 16 22 28 34 17 35 5 23 29 11 18 12 30 24 6 36)	((1) (2 9 18 31) (3 17 25) (4 19) (5 27 13) (6 35 26 7) (8) (10 20) (11 30 33 14) (12 32) (15) (16 21) (22) (23 28) (24 34) (29) (36))

Tabla 4.29: Ejemplos de Ciclos en Vectores V_M.

Función: cycles-class

```
(defun cycles-class (subvec cset &optional (asubvec nil) (auxset nil) (class nil) (pos 0))
   (cond ((null subvec) class)
     (T (if (= pos 0)
       (cycles-class subvec cset (car subvec) auxset class 1)
      (if (null asubvec)
         (cycles-class (cdr subvec) cset nil nil (push auxset class) 0)
         (cycles-class subvec cset (cdr asubvec) (push (nth (1- (car asubvec)) cset)
           auxset) class (1+ pos)))))))
```

Tabla 4.30: Código: cycles-class

Descripción:
Dado el arreglo `subvec` con los ciclos de un vector V_M y el conjunto `cset` el cual es $S \otimes S$ para algún S perfecto, extrae los subconjuntos asociados a los ciclos.

Argumentos:	
`subvec`	Lista que contiene los ciclos de un vector de permutación.
`cset`	Arreglo que contiene los elementos del conjunto $S \otimes S$.
`asubvec`	Argumento auxiliar que almacena temporalmente los ciclos de `subvec`.
`auxset`	Argumento auxiliar que almacena temporalmente los subconjuntos de $S \otimes S$ asociados a los ciclos.
`class`	Argumento auxiliar que va almacenando los subconjuntos de $S \otimes S$ dando al final las clases de dichos ciclos.
`pos`	Contador interno para ir recorriendo los elementos en `subset`.

Tabla 4.31: Código: cycles-class

```
(cycles-class subvec cset)
```

Tabla 4.32: Código ejecutado desde `Lisp`.

Grupo de Isometrías del Tetraedro								
$S = \{s1\ s2\ s3\}$								
Ciclos	(1)	(2 4)		(3 7)		(5)	(6 8)	(9)
Subconjuntos	(S1 S1)	(S2 S1)	(S1 S2)	(S3 S1)	(S1 S3)	(S2 S2)	(S3 S2) (S2 S3)	(S3 S3)

$S = \{ra\ rb\ rc\ rd\ rai\ rbi\ rci\ rdi\}$		
(1) (RA RA)	(2 11 17) (RC RA) (RB RC) (RA RB)	(3 20 25) (RD RA) (RC RD) (RA RC)
(4 26 9) (RB RA) (RD RB) (RA RD)	(5 33) (RAI RA) (RA RAI)	(6 44 31 49) (RCI RA) (RD RCI) (RBI RD) (RA RBI)
(7 50 16 57) (RDI RA) (RB RDI) (RCI RB) (RA RCI)	(8 59 22 41) (RBI RA) (RC RBI) (RDI RC) (RA RDI)	(10) (RB RB)
(12 27 18) (RC RB) (RD RC) (RB RD)	(13 35 24 58) (RDI RB) (RC RDI) (RAI RC) (RB RAI)	(14 42) (RBI RB) (RB RBI)
(15 52 29 34) (RAI RB) (RD RAI) (RCI RD) (RB RCI)	(19) (RC RC)	(21 36 30 43) (RBI RC) (RD RBI) (RAI RD) (RC RAI)
(23 51) (RCI RC) (RC RCI)	(28) (RD RD)	(32 60) (RDI RD) (RD RDI)
(37) ((RAI RAI)	(38 48 61) (RDI RAI) (RBI RDI) (RAI RBI)	(39 54 45) (RBI RAI) (RCI RBI) (RAI RCI)
(40 63 53) (RCI RAI) (RDI RCI) (RAI RDI)	(46) (RBI RBI)	(47 56 62) (RDI RBI) (RCI RDI) (RBI RCI)
(55) (RCI RCI)	(64) (RDI RDI)	

Grupo de permutaciones de 3 elementos.

$S = \{(2\ 3\ 1)\ (3\ 1\ 2)\}$

Ciclos	(1)	(2 3)	(4)
Subconjunto:	((2 3 1) (2 3 1))	((3 1 2) (2 3 1)) ((2 3 1) (3 1 2))	((3 1 2) (3 1 2))

Tabla 4.33: Ejemplos de Ciclos.

4.4. Maxima

Por último se realizó una rutina sobre MAXIMA para dado un vector V_M poder reconstruir la matriz M_σ y en su ambiente poder hacer cálculos como búsqueda de valores y vectores propios.

Función: msigma

```
msigma(vec):=block(n:length(vec),
   for i:1 thru n do for j:1 thru n do
     if vec[j] = i then M[i,j] : 1 else M[i,j] : 0,
       genmatrix(M,n,n));
```

Descripción:

Dados los elementos del vector $V_M(j) = i$ coloca un 1 en la entrada $M_\sigma(i, j) = 1$, cero en otro caso.

Ejemplos:

Si $V_M = [1, 4, 7, 2, 5, 8, 3, 6, 9]$

$$M_\sigma = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Si $V_M = [1, 7, 4, 8, 5, 2, 6, 3, 9]$

$$M_\sigma = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Apéndice A

Isometrías

Definición. *Si E_1 y E_2 son dos espacios métricos una isometría φ está dada por: $\varphi : E_1 \to E_2$, $\forall\ (x, y) \in E_1 \times E_2$, tal que $d_1(x, y) = d_2(\varphi(x), \varphi(y))$, con d_1 y d_2 las respectivas métricas en E_1 y E_2.*

A.1. Triángulo Equilatero

Las Isometrías del triángulo descritas son las siguientes:

$$rx : \{a, b, c, \} \to \{a, b, c\},\ \text{con } x = a, b, c.$$

$$sa : \begin{bmatrix} a & b & c \\ a & c & b \end{bmatrix}\ sb : \begin{bmatrix} a & b & c \\ c & b & a \end{bmatrix}\ sc : \begin{bmatrix} a & b & c \\ b & a & c \end{bmatrix}$$

$$rho : \begin{bmatrix} a & b & c \\ c & a & b \end{bmatrix}\ rhoi : \begin{bmatrix} a & b & c \\ b & c & a \end{bmatrix}\ e : \begin{bmatrix} a & b & c \\ a & b & c \end{bmatrix}$$

La tabla de multiplicación está dada como:

	sa	sb	sc	rho	$rhoi$
sa	e	$rhoi$	rho	sb	sc
sb	rho	e	$rhoi$	sc	sa
sc	$rhoi$	rho	e	sa	sb
rho	sc	sa	sb	$rhoi$	e
$rhoi$	sb	sc	sa	e	rho

Para el grupo de las isometrías del triángulo se tiene que las correspondientes clases de conjugación son:

$$\{e, sa\ sb\ sc, rho\ rhoi\}$$

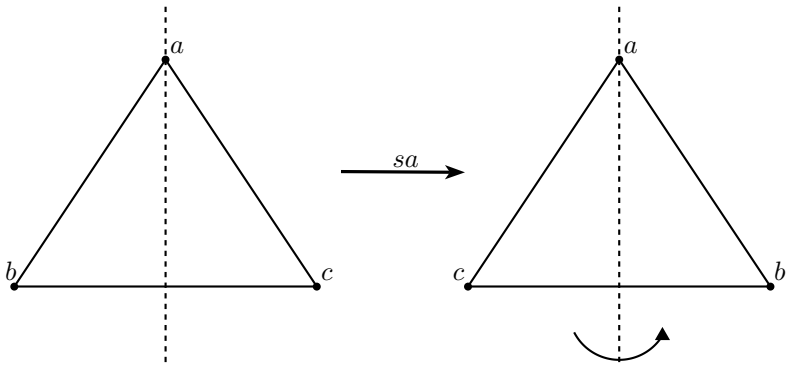


Figura A.1: Isometría *sa*.

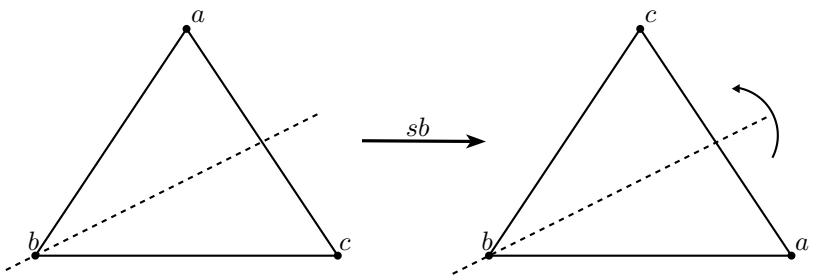


Figura A.2: Isometría *sb*.

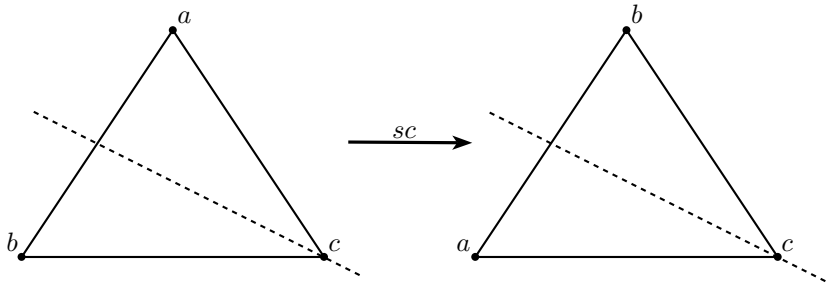


Figura A.3: Isometría *sc*

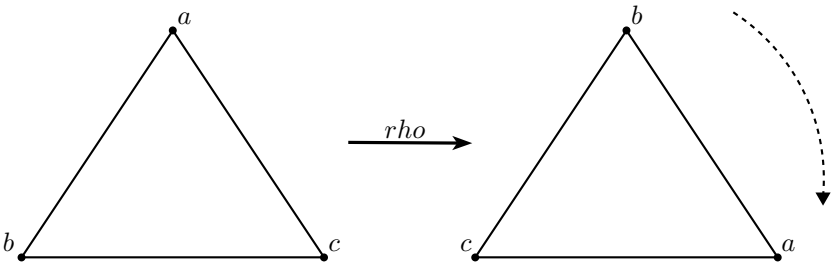


Figura A.4: Isometría *rho*

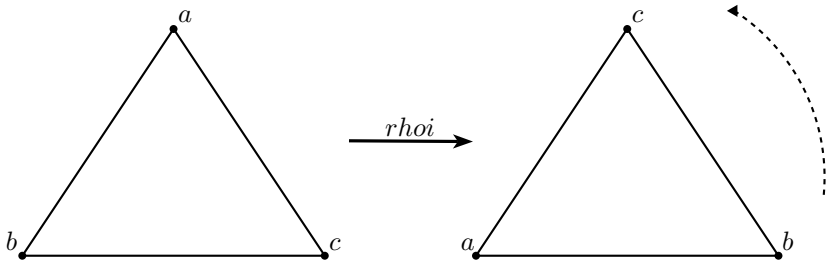


Figura A.5: Isometría *rhoi*

A.2. Cuadrado

Las Isometrías del cuadrado descritas son las siguientes:

$$rx : \{a,b,c,d\} \to \{a,b,c,d\}, \text{ con } x = a,b,c,d.$$

$$sa : \begin{bmatrix} a & b & c & d \\ a & d & c & b \end{bmatrix} \quad sb : \begin{bmatrix} a & b & c & d \\ c & b & a & d \end{bmatrix} \quad s1 : \begin{bmatrix} a & b & c & d \\ b & a & d & c \end{bmatrix} \quad s2 : \begin{bmatrix} a & b & c & d \\ d & c & b & a \end{bmatrix}$$

$$rho : \begin{bmatrix} a & b & c & d \\ d & a & b & c \end{bmatrix} \quad rhoi : \begin{bmatrix} a & b & c & d \\ b & c & d & a \end{bmatrix} \quad rho2 : \begin{bmatrix} a & b & c & d \\ c & d & a & b \end{bmatrix} \quad e : \begin{bmatrix} a & b & c \\ a & b & c \end{bmatrix}$$

La tabla de multiplicación está dada como:

	sa	sb	$s1$	$s2$	rho	$rhoi$	$rho2$
sa	e	$rho2$	rho	$rhoi$	$s1$	$s2$	sb
sb	$rho2$	e	$rhoi$	rho	$s2$	$s1$	sa
$s1$	$rhoi$	rho	e	$rho2$	sb	sa	$s1$
$s2$	rho	$rhoi$	$rho2$	e	sa	sb	$s2$
rho	$s2$	$s1$	sa	sb	$rho2$	e	$rhoi$
$rhoi$	$s1$	$s2$	sb	sa	e	$rho2$	rho
$rho2$	sb	sa	$s2$	$s1$	$rhoi$	rho	e

Para el grupo de las isometrías del cuadrado se tiene que las correspondientes clases de conjugación son:

$$\{e, sa\ sb,\ s1\ s2,\ rho\ rhoi,\ rho2\}$$

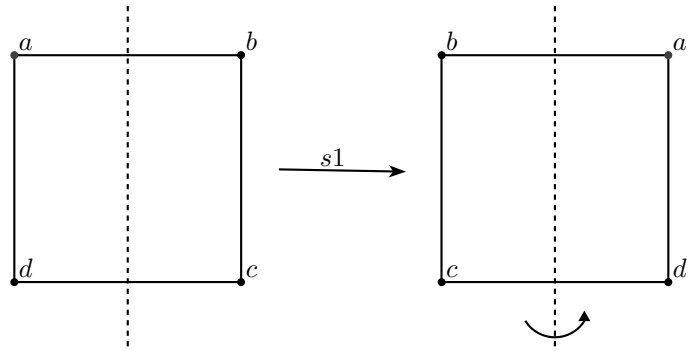


Figura A.6: Isometría $s1$

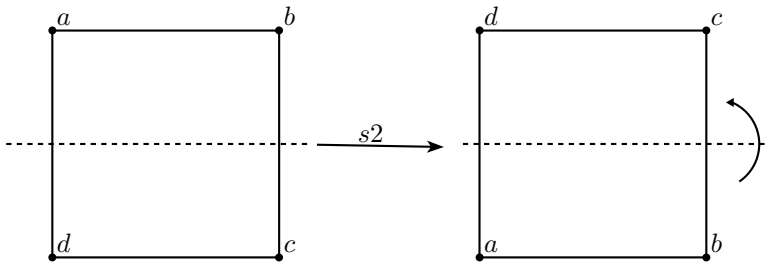


Figura A.7: Isometría $s2$

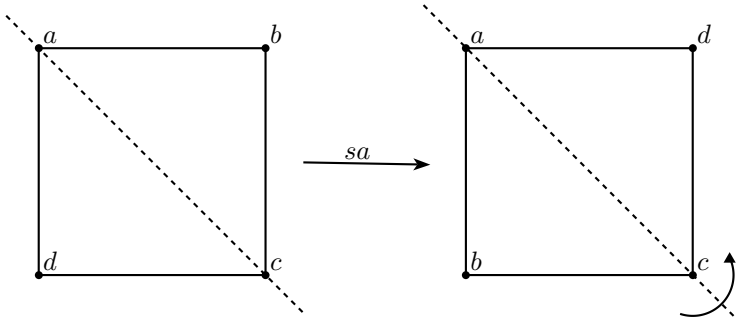


Figura A.8: Isometría *sa*

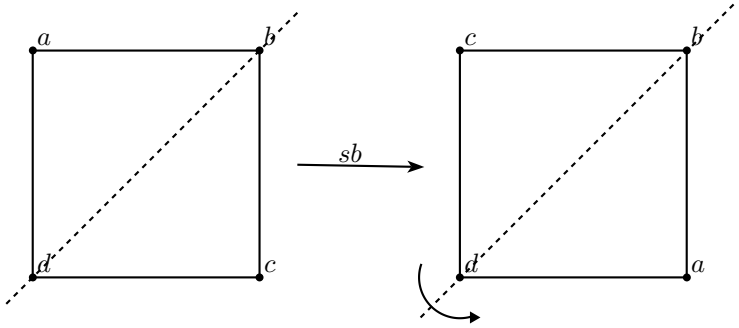


Figura A.9: Isometría *sb*

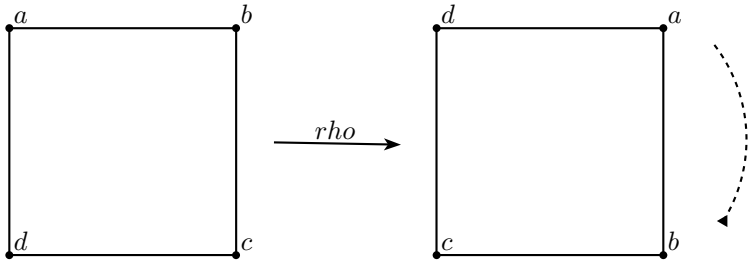


Figura A.10: Isometría *rho*

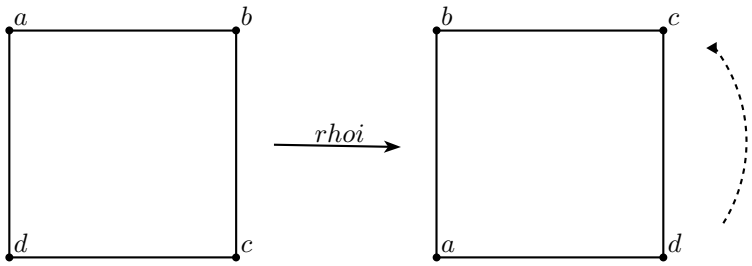


Figura A.11: Isometría *rhoi*

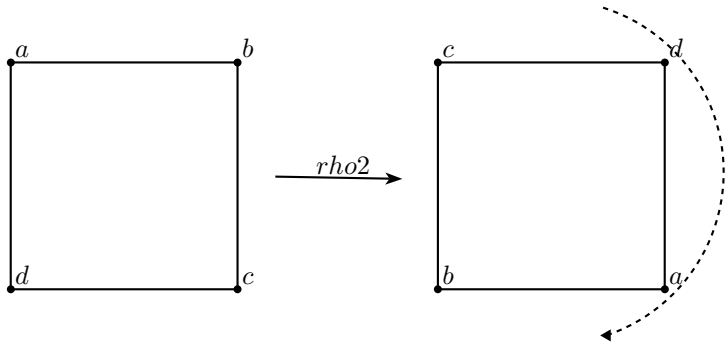


Figura A.12: rho2

A.3. Tetraedro

Las Isometrías que se derivan de rotaciones en el Tetraedro son la siguientes:

Primer Subconjunto.

Éstas transformaciones constan, dado un vértice fijo, rotar[1] el tetraedro a la derecha y ver como mueve los vértices. Si a éstos los indexamos con la letras a, b, c, d y la isometría la denotamos por rx, donde $x = a, b, c, d$. Se tiene;

$$rx : \{a, b, c, d\} \to \{a, b, c, d\}, \text{ con } x = a, b, c, d.$$

$$ra : \begin{bmatrix} a & b & c & d \\ a & c & d & b \end{bmatrix} rb : \begin{bmatrix} a & b & c & d \\ d & b & a & c \end{bmatrix} rc : \begin{bmatrix} a & b & c & d \\ b & d & c & a \end{bmatrix} rd : \begin{bmatrix} a & b & c & d \\ c & a & b & d \end{bmatrix}$$

[1]Entendiéndose rotar de forma subjetiva, ya que el movimiento es de acuerdo al observante.

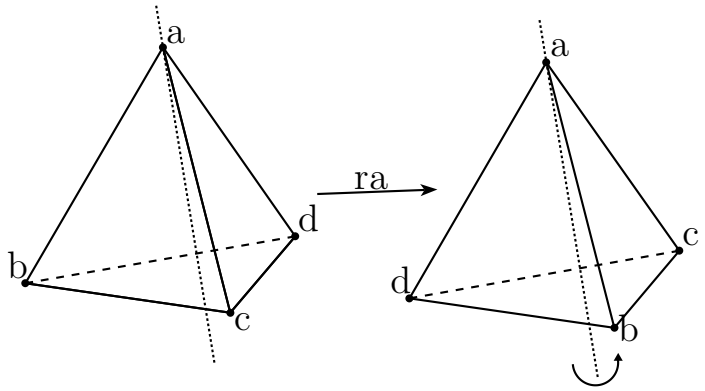


Figura A.13: Isometría: *ra*

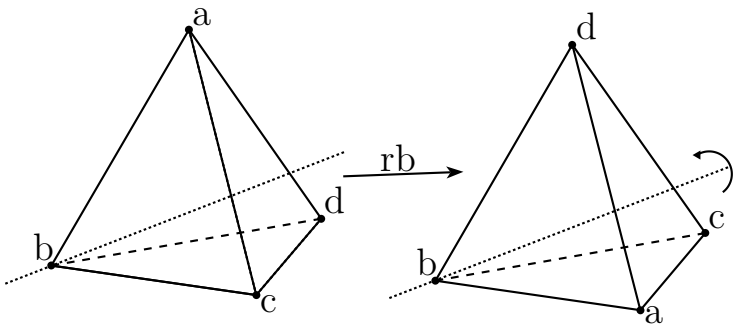


Figura A.14: Isometría: *rb*

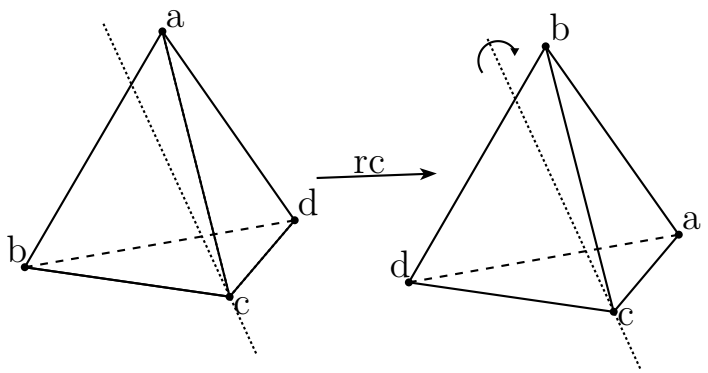


Figura A.15: Isometría: rc

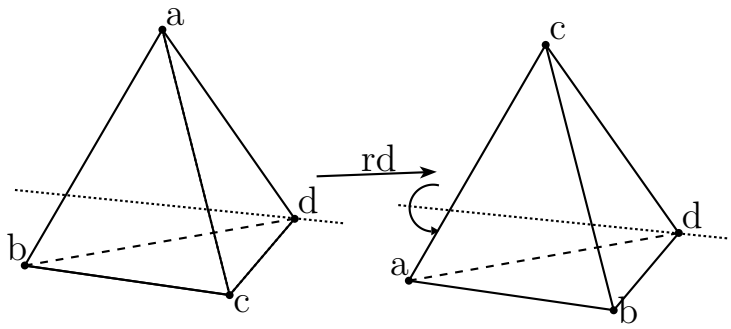


Figura A.16: Isometría: rd

Segundo Subconjunto.

Éstas transformaciones constan, dado un vértice fijo, rotar el tetraedro a la izquierda y ver como mueve los vértices. Si se denotan por rxi, donde $x = a, b, c, d$ con la idea que representan el giro inverso a las anteriores, están dada por:

$$rxi : \{a, b, c, d\} \to \{a, b, c, d\}, \text{ con } x = a, b, c, d.$$

$$rai : \begin{bmatrix} a & b & c & d \\ a & d & b & c \end{bmatrix} rbi : \begin{bmatrix} a & b & c & d \\ c & b & d & a \end{bmatrix} rci : \begin{bmatrix} a & b & c & d \\ d & a & c & b \end{bmatrix} rdi : \begin{bmatrix} a & b & c & d \\ b & c & a & d \end{bmatrix}$$

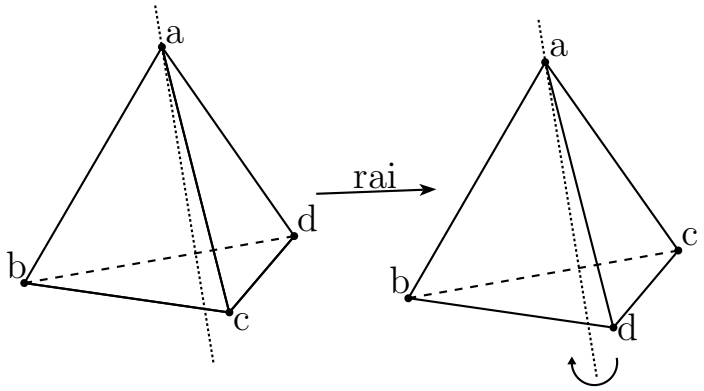


Figura A.17: Isometría: *rai*

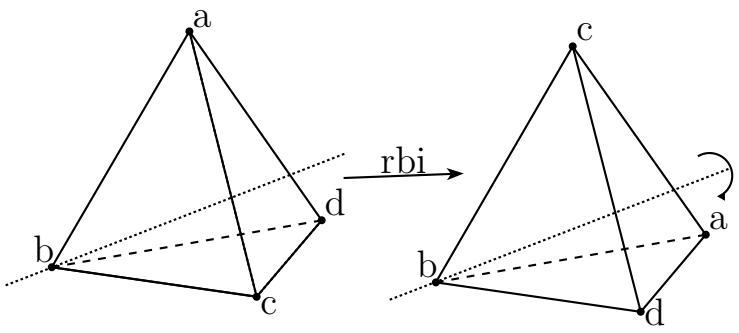


Figura A.18: Isometría: *rbi*

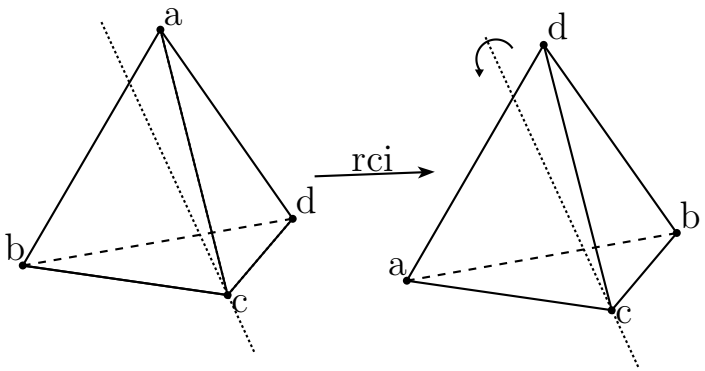


Figura A.19: Isometría: rci

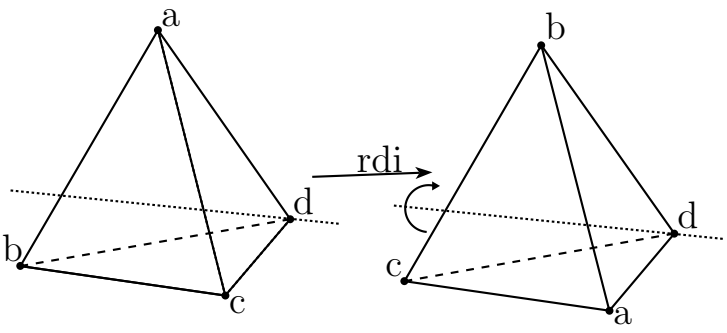


Figura A.20: Isometría: rdi

Tercer Subconjunto.

Las transformaciones que conforman este bloque, son las resultantes de girar el tetraedro por los puntos medios de aristas opuestas. Denotadas por Si, donde $i = 1, 2, 3$ son las siguientes:

$$Si : \{a, b, c, d\} \to \{a, b, c, d\}, \text{ con } i = 1, 2, 3.$$

$$s1 : \begin{bmatrix} a & b & c & d \\ c & d & a & b \end{bmatrix} \; s2 : \begin{bmatrix} a & b & c & d \\ d & c & b & a \end{bmatrix} \; s3 : \begin{bmatrix} a & b & c & d \\ b & a & d & c \end{bmatrix}$$

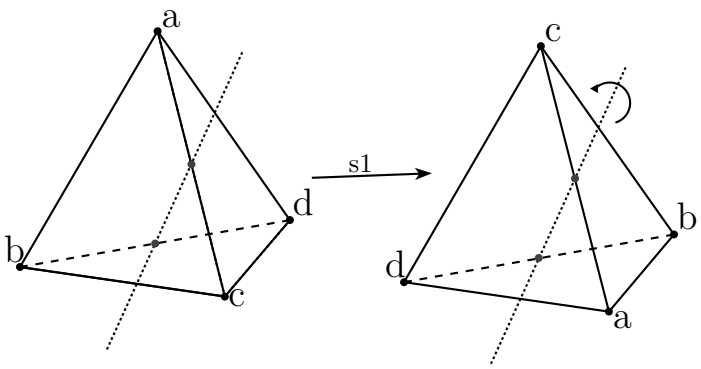


Figura A.21: Isometría: s1

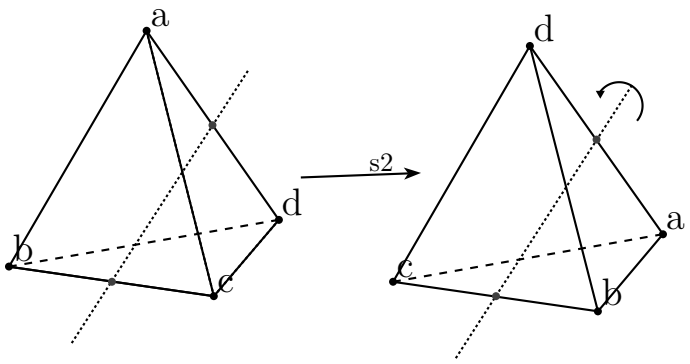


Figura A.22: Isometría: s2

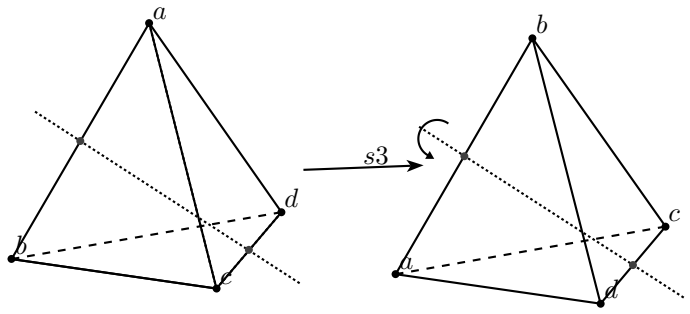


Figura A.23: Isometría: s3

A.4. Paridad de las transformaciones del tetraedro

Se procederá a comprobar que el grupo de isometrías del tetraedro regular isomorfo al subgrupo alternante A_4 tiene todos su elementos pares.

Nota: si se cumple con la condición $|\{j > i : \phi(j) < \phi(i)\}|$ se contabiliza como 1, en caso contrario como 0.

Se denotará en letras negritas los valores que aportan con la aseveración $\phi(j) < \phi(i)$.

ra tiene la siguiente regla de correspondencia:

$$ra(1) = 1, ra(2) = 3, ra(3) = 4, ra(4) = 2$$

$$\begin{aligned}\texttt{inv(ra)} = &|\{2 > 1 : 3 < 1\}| + |\{3 > 1 : 4 < 1\}| + |\{4 > 1 : 2 < 1\}| + \\ &|\{3 > 2 : 4 < 3\}| + |\{\mathbf{4} > \mathbf{2} : \mathbf{2} < \mathbf{3}\}| + |\{\mathbf{4} > \mathbf{3} : \mathbf{2} < \mathbf{4}\}|\end{aligned}$$

$$\texttt{inv(ra)} = 2$$

rb tiene la siguiente regla de correspondencia:

$$rb(1) = 1, rb(2) = 2, rb(3) = 1, rb(4) = 3$$

$$\texttt{inv(rb)} = |\{\mathbf{2} > \mathbf{1} : \mathbf{2} < \mathbf{4}\}| + |\{\mathbf{3} > \mathbf{1} : \mathbf{1} < \mathbf{4}\}| + |\{\mathbf{4} > \mathbf{1} : \mathbf{3} < \mathbf{4}\}| + |\{\mathbf{3} > \mathbf{2} : \mathbf{1} < \mathbf{2}\}| + |\{4 > 2 : 3 < 2\}| + |\{4 > 3 : 4 < 1\}|$$

$$\texttt{inv(rb)} = 4$$

rc tiene la siguiente regla de correspondencia:

$$rc(1) = 2, rc(2) = 4, rc(3) = 3, rc(4) = 1$$

$$\texttt{inv(rc)} = |\{2 > 1 : 4 < 2\}| + |\{3 > 1 : 3 < 2\}| + |\{\mathbf{4} > \mathbf{1} : \mathbf{1} < \mathbf{2}\}| + |\{\mathbf{3} > \mathbf{2} : \mathbf{3} < \mathbf{4}\}| + |\{\mathbf{4} > \mathbf{2} : \mathbf{1} < \mathbf{4}\}| + |\{\mathbf{4} > \mathbf{3} : \mathbf{1} < \mathbf{3}\}|$$

$$\texttt{inv(rc)} = 4$$

rd tiene la siguiente regla de correspondencia:

$$rd(1) = 3, rd(2) = 1, rd(3) = 2, rd(4) = 4$$

$$\texttt{inv(rd)} = |\{\mathbf{2} > \mathbf{1} : \mathbf{1} < \mathbf{3}\}| + |\{\mathbf{3} > \mathbf{1} : \mathbf{2} < \mathbf{3}\}| + |\{4 > 1 : 4 < 3\}| + |\{3 > 2 : 2 < 1\}| + |\{4 > 2 : 4 < 1\}| + |\{4 > 3 : 4 < 2\}|$$

$$\texttt{inv(rd)} = 2$$

rai tiene la siguiente regla de correspondencia:

$$rai(1) = 1, rai(2) = 4, rai(3) = 2, rai(4) = 3$$

$$\texttt{inv(rai)} = |\{2 > 1 : 4 < 1\}| + |\{3 > 1 : 2 < 1\}| + |\{4 > 1 : 3 < 1\}| + |\{\mathbf{3} > \mathbf{2} : \mathbf{2} < \mathbf{4}\}| + |\{\mathbf{4} > \mathbf{2} : \mathbf{3} < \mathbf{4}\}| + |\{4 > 3 : 3 < 2\}|$$

$$\texttt{inv(rai)} = 2$$

rbi tiene la siguiente regla de correspondencia:

$$rbi(1) = 3, rbi(2) = 2, rbi(3) = 4, rbi(4) = 1$$

$$\texttt{inv(rbi)} = |\{\mathbf{2} > \mathbf{1} : \mathbf{2} < \mathbf{3}\}| + |\{3 > 1 : 4 < 3\}| + |\{\mathbf{4} > \mathbf{1} : \mathbf{1} < \mathbf{3}\}| + |\{3 > 2 : 4 < 2\}| + |\{\mathbf{4} > \mathbf{2} : \mathbf{1} < \mathbf{2}\}| + |\{\mathbf{4} > \mathbf{3} : \mathbf{1} < \mathbf{4}\}|$$

$$\texttt{inv(rbi)} = 4$$

rci tiene la siguiente regla de correspondencia:

$$rci(1) = 4, rci(2) = 1, rci(3) = 3, rci(4) = 2$$

$$\begin{aligned}\texttt{inv(rci)} = &|\{\mathbf{2} > \mathbf{1} : \mathbf{1} < \mathbf{4}\}| + |\{\mathbf{3} > \mathbf{1} : \mathbf{3} < \mathbf{4}\}| + |\{\mathbf{4} > \mathbf{1} : \mathbf{2} < \mathbf{4}\}| + \\ &|\{3 > 2 : 3 < 1\}| + |\{4 > 2 : 2 < 1\}| + |\{\mathbf{4} > \mathbf{3} : \mathbf{2} < \mathbf{3}\}|\end{aligned}$$

$$\texttt{inv(rci)} = 4$$

rdi tiene la siguiente regla de correspondencia:

$$rdi(1) = 2, rdi(2) = 3, rdi(3) = 1, rdi(4) = 4$$

$$\begin{aligned}\texttt{inv(rdi)} = &|\{2 > 1 : 3 < 2\}| + |\{\mathbf{3} > \mathbf{1} : \mathbf{1} < \mathbf{2}\}| + |\{4 > 1 : 4 < 2\}| + \\ &|\{\mathbf{3} > \mathbf{2} : \mathbf{1} < \mathbf{3}\}| + |\{4 > 2 : 4 < 3\}| + |\{4 > 3 : 4 < 1\}|\end{aligned}$$

$$\texttt{inv(rdi)} = 2$$

s1 tiene la siguiente regla de correspondencia:

$$s1(1) = 3, s1(2) = 4, s1(3) = 1, s1(4) = 1$$

$$\begin{aligned}\texttt{inv(s1)} = &|\{2 > 1 : 4 < 3\}| + |\{\mathbf{3} > \mathbf{1} : \mathbf{1} < \mathbf{3}\}| + |\{\mathbf{4} > \mathbf{1} : \mathbf{2} < \mathbf{3}\}| + \\ &|\{\mathbf{3} > \mathbf{2} : \mathbf{1} < \mathbf{4}\}| + |\{\mathbf{4} > \mathbf{2} : \mathbf{2} < \mathbf{4}\}| + |\{4 > 3 : 2 < 1\}|\end{aligned}$$

$$\texttt{inv(s1)} = 4$$

s2 tiene la siguiente regla de correspondencia:

$$s2(1) = 4, s2(2) = 3, s2(3) = 2, s2(4) = 1$$

$$\begin{aligned}\texttt{inv(s2)} = &|\{\mathbf{2} > \mathbf{1} : \mathbf{3} < \mathbf{4}\}| + |\{\mathbf{3} > \mathbf{1} : \mathbf{2} < \mathbf{4}\}| + |\{\mathbf{4} > \mathbf{1} : \mathbf{1} < \mathbf{4}\}| + \\ &|\{\mathbf{3} > \mathbf{2} : \mathbf{2} < \mathbf{3}\}| + |\{\mathbf{4} > \mathbf{2} : \mathbf{1} < \mathbf{3}\}| + |\{\mathbf{4} > \mathbf{3} : \mathbf{1} < \mathbf{2}\}|\end{aligned}$$

$$\texttt{inv(s2)} = 6$$

s3 tiene la siguiente regla de correspondencia:

$$s3(1) = 2, s3(2) = 1, s3(3) = 4, s3(4) = 3$$

$$\texttt{inv(s3)} = |\{\mathbf{2} > \mathbf{1} : \mathbf{1} < \mathbf{2}\}| + |\{3 > 1 : 4 < 2\}| + |\{4 > 1 : 3 < 2\}| + |\{3 > 2 : 4 < 1\}| + |\{4 > 2 : 3 < 1\}| + |\{\mathbf{4} > \mathbf{3} : \mathbf{3} < \mathbf{4}\}|$$

$$\texttt{inv(s3)} = 2$$

Por último la identidad no cumple con la definición $|\{j > i : \phi(j) < \phi(i)\}|$ para ninguna i ni j, así que $\texttt{inv(e)} = 0$.

Así queda demostrado que todas las transformaciones de isometrías son pares.

Apéndice B

Teorema Gelfand-Naimark

Un Teorema importante dentro del contexto cuántico es el Teorema de Gelfand-Naimark, se procederán a dar algunos resultados que llevaran a su demostración y que para detalles de los mismos se puede consultar [8].

Se dice que A es un álgebra normada si se tiene definida una norma en A y ésta satisface:

$$||ab|| \leq ||a||||b|| \ \forall \ a, b \in A$$

Una involución en A es un mapeo $* : A \to A$ tal que si cumple

I) $a^{**} = a$

II) $(\lambda a + b)^* = \overline{\lambda} a^* + b^*$

III) $(ab)^* = b^* a^*$ con $\lambda, \in \mathbb{C}$ y $a, b \in A$.

entonces A es llamada álgebra $*$ y si además se tiene la propiedad:

$$||aa^*|| = ||a||^2$$

siendo A espacio de Banach con involución, entonces A es llamada álgebra C^* donde a la última expresión se le conoce como condición C^*.

El espectro de un elemento a de A es definido como el conjunto de todos los números complejos λ tal que $\lambda e - a$ no tiene inversa en A y es denotado por $\sigma(a)$, esto es:

$$\sigma(a) = \{\lambda : \lambda e - a \text{ es singular}\}$$

Algunas definiciones para cuando los elementos de A cumplen con cierta característica son las siguientes:

- Sea $u \in A$, u es unitario sí $uu^* = u^* u = e$.

- Sea $a \in A$, a es hermitiano sí $a^* = a$.
- Sea $a \in A$, a es normal sí $aa^* = a^*a$.

TEOREMA 2. *En un álgebra C^* se cumple:*

I $\sigma(a) = \overline{\sigma(a^*)}$.

II *Sí u es unitario entonces $|\lambda| = 1 \ \forall \ \lambda \in \sigma(u)$.*

III *Si a es hermitiano entonces $\sigma(a)$ es real.*

COROLARIO 1. *Si A es un álgebra C^* y a es normal, entonces:*

$$||a|| = \nu(a)$$

donde el radio espectral está dado por,

$$\nu(a) = \text{máx}\{|\lambda| : \lambda \in \sigma(a)\}$$

DEFINICIÓN. *Un elemento de un álgebra de Banach es llamado quasi nilpotente si el único elemento en su espectro es 0.*

DEFINICIÓN. *Un homomorfismo entre dos álgebras A y B es una función $\phi : A \to B$ que cumple:*

1. $\phi(\lambda a + b) = \lambda\phi(a) + \phi(b)$
2. $\phi(ab) = \phi(a)\phi(b)$

Para toda $a, b \in A$ y $\lambda \in \mathbb{C}$.

DEFINICIÓN. *Un homomorfismo $*$ es un homomorfismo que preserva la involución, es decir,*

$$\phi(a^*) = \phi^*(a)$$

Se puede demostrar que cada homomorfismo $*$ entre C^* álgebras, con $\phi : A \to B$ cumple $\|\phi(a)\| \leq \|a\| \ \forall \ a \in A$ en particular es automáticamente continuo.

Ahora se van a considerar álgebras C^* conmutativas.

Sea Φ el conjunto de los homomorfismos, no cero, de A en $\mathbb{C}$, para cada $\phi \in \Phi$ y $a \in A$ se tiene el número complejo $\phi(a)$, así dada $a \in A$ se define la función $\widehat{a} : \Phi \to \mathbb{C}$ por

$$\widehat{a}(\phi) = \phi(a)$$

También se puede demostrar que tales ϕ son automáticamente $*$ homomorfismos.

TEOREMA 3. *Las siguientes afirmaciones son equivalentes:*

I *La función identidad $\widehat{e}$, es decir, $\widehat{e}(\phi) = 1 \ \forall \ \phi$.*

II $\widehat{a} = 0$ *sí y sólo si* $a \in \bigcap\{\phi^{-1}(0) : \phi \in \Phi\} = \bigcap\{M : M \in \mathcal{M}\}$.

III $\sigma(a) = \{\widehat{a}(\phi) : \phi \in \Phi\}$.

IV $\nu(a) = máx\{|\widehat{a}(\phi)| : \phi \in \Phi\}$.

V *Sí* $\phi_1 \neq \phi_2$ *entonces para alguna* $a \in A$, $\widehat{a}(\phi_1) \neq \widehat{a}(\phi_2)$.

Donde $\mathcal{M}$ es el conjunto de ideales maximales propios de A.

DEFINICIÓN. *Transformación de Gelfand*

Se define para cada $a \in A$ la función $\widehat{a} : \Phi \to \mathbb{C}$ dada por:

$$\widehat{a}(\phi) = \phi(a)$$

TEOREMA 4. *El mapeo $a \to \widehat{a}$ es un homomorfismo algebraico de A en el conjunto de las funciones complejo valuadas de Φ.*

TEOREMA 5. *Mapeo de Gelfand*

Dada cualquier álgebra de Banach conmutativa A unital, existe un homomorfismo de A en $C(\Phi)$, el álgebra de todas las funciones complejo valuadas sobre un espacio compacto Hausdorff Φ. El Kernel del mapeo es el conjunto de todos los elementos quasi nilpotentes de A.

TEOREMA 6. *Gelfand-Naimark*

Dada cualquier álgebra unital C^* conmutativa A, existe un isomorfismo $*$ isométrico de A en $C(\Phi)$, el álgebra de todas las funciones complejo valuadas sobre el espacio compacto Hausdorff Φ.

Demostración. Sea Φ el espectro de A y sea $a \to \widehat{a}$ el Mapeo de Gelfand. Cada elemento del álgebra conmutativa C^* es normal, del Corolario 1 la norma de cada elemento de A es igual a su radio espectral, esto es $||a|| = ||\widehat{a}||$.

A es un espacio de Banach por lo tanto completo, ya que el Mapeo de Gelfand es isométrico, el conjunto $\{\widehat{a} : a \in A\}$ es un subconjunto completo de $C(\Phi)$ y cerrado.

Para mostrar que el Mapeo de Gelfand preserva la involución en $C(\Phi)$, donde ésta es definida por $f^*(\phi) = \overline{f(\phi)}$, véase lo siguiente: para $a \in A$ se tiene $a = x + \imath y$ donde $x = \frac{1}{2}(a + a^*)$, $y = \frac{1}{2\imath}(a - a^*)$ son hermitianos. Del Teorema 2, x y y tienen espectro real y por lo tanto $\widehat{x}(\phi) \in \sigma(x)$, así las funciones $\widehat{x}$ y $\widehat{y}$ son real valuadas. Por lo tanto

$$(\widehat{a})^* = \overline{\widehat{x} + \imath\widehat{y}} = \widehat{x} - \imath\widehat{y} = \widehat{x - \imath y} = \widehat{a^*}$$

Lo cual muestra que la imagen de $\widehat{A}$ es una subalgebra $*$ y que el Mapeo de Gelfand preserva la operación adjunta u operación $*$. Así se tienen las condiciones para aplicar el Teorema de Stone-Weiestrass, es decir, $\widehat{A}$ separa puntos de Φ y contiene la función identidad (Teorema 3), por lo tanto $\widehat{A} = C(\Phi)$.

□

Apéndice C

Rutinas

Aquí se detallaran las funciones auxiliares que se utilizaron dentro de los programas principales.

Función: `find-insert`

```
(defun find-insert (x list &optional (nlist nil))
   (cond ((null list) (cons x nlist))
         ((equalp x (car list)) (append nlist list))
         (T (let* ((nlist (append nlist (list (car list))))
           (list (cdr list)))
         (find-insert x list nlist)))))
```

Descripción:

Verifica si `x` se encuentra en la lista `list` de pertenecer devuelve la lista, en caso contrario lo inserta al inicio.

Argumentos:

`x`	Elemento.
`list`	Lista.
`nlist`	Argumento auxiliar que almacena la lista de salida.

Ejemplos:

Input	Output
`>(find-insert 'a '(1 2 3 4 5))`	`(A 1 2 3 4 5)`
`>(find-insert 2 '(1 2 3 4 5))`	`(1 2 3 4 5)`
`>(find-insert 5 '(1 2 3 4 5))`	`(1 2 3 4 5)`

Función: `remove-sub`[1]

```
(defun remove-sub (list sublist)
   (cond ((null sublist) list)
     (T (if (equalp (car list) (car sublist))
          (let* ((sublist (cdr sublist))
            (list (cdr list)))
          (remove-sub list sublist))
     (let* ((list (append (cdr list) (list (car list)))))
          (remove-sub list sublist))))))
```

Descripción:

Remueve el subconjunto `sublist` del conjunto `list`. Se parte del hecho que `sublist` pertenece a `list`.

Argumentos:

`list`	Lista.
`sublist`	Subconjunto de la lista `list`.

Ejemplos:

Input	Output
`>(remove-sub '((a) (b) (c)) '((b)))`	`((C) (A))`
`>(remove-sub '(1 2 3 4) '(1 2))`	`(3 4)`

Función: `seek-elem`

```
(defun seek-elem (elem list &optional (nlist nil))
   (cond ((null elem) nlist)
     ((null list) (cons elem nlist))
     (T (if (equalp elem (car list))
        (let ((nlist (append list nlist)))
          (seek-elem nil list nlist))
        (let* ((nlist (cons (car list) nlist))
               (list (cdr list)))
            (seek-elem elem list nlist))))))
```

Descripción:

Busca un elemento en una lista `list` de estar devuelve `list`, de no estar lo inserta.

Argumentos:

`elem`	Elemento a insertar.
`list`	Lista.
`nlist`	Arreglo auxiliar donde se almacena la lista de salida.

Ejemplos:

Input	Output
`>(seek-elem 4 '(1 2 3 4 5 6))`	`(1 2 3 6 5 4)`
`>(seek-elem 4 '(1 2 3 5 6))`	`(4 6 5 3 2 1)`

Función: `my-remove-duplicates`[2]

```
(defun my-remove-duplicates (list &optional (tlist nil))
     (cond ((null list) tlist)
       (T (let* ((pos (pos-elem (car list) (cdr list))))
         (if (= pos 0)
           (let ((tlist (push (car list) tlist)))
             (my-remove-duplicates (cdr list) tlist))
           (let* ((alist (remove-elem pos (cdr list)))
                (list (push (car list) alist)))
             (my-remove-duplicates list tlist)))))))
```

Descripción:

Remueve duplicados en una lista, abarcando el caso en que éstos sean listas.Si no hay duplicados devuelve la lista.

Argumentos:

`list`	Lista.
`tlist`	Argumento auxiliar donde se almacena la lista de salida.

Ejemplos:

```
> (my-remove-duplicates '((a) (a) (a) (b) (c) (c) (d)))
  ((A) (B) (C) (D))
> (my-remove-duplicates '(1 2 3 4 4 5 5 6))
  (1 2 3 4 5 6)
> (my-remove-duplicates '(1 2 3 4 5 6))
  (1 2 3 4 5 6)
```

Función: `pos-elem`

```
(defun pos-elem (elem list &optional (pos nil))
   (cond ((null elem) pos)
     ((null list) 0)
     (T (if (null pos)
       (let ((pos 0))
         (if (equalp elem (car list))
           (let ((pos (1+ pos)))
             (pos-elem nil list pos))
           (let ((pos (1+ pos)))
             (pos-elem elem (cdr list) pos))))
         (if (equalp elem (car list))
           (let ((pos (1+ pos)))
             (pos-elem nil list pos))
           (let ((pos (1+ pos)))
             (pos-elem elem (cdr list) pos)))))))
```

Descripción:

Devuelve la posicion de la primera vez que `elem` aparece en la lista `list`, si no está devuelve 0.

Argumentos:

`elem`	Elemento a buscar en la lista.
`list`	Lista donde se buscará.
`pos`	Variable auxiliar donde se almacena la posición.

Ejemplos:

Input	Output
`>(pos-elem ‘d ‘(a b c d e))`	`4`
`>(pos-elem ‘d ‘(a b c d e d d f))`	`4`
`>(pos-elem ‘z ‘(a b c d e))`	`0`

Función: `remove-elem`

```
(defun remove-elem (pos list &optional (nlist nil))
   (cond ((eql pos 1) (append nlist (cdr list)))
     (T (let* ((nlist (append nlist (list (car list))))
          (list (cdr list))
          (pos (1- pos)))
        (remove-elem pos list nlist)))))
```

Descripción:

Elimina un elemento de una lista dada su posicion.

Argumentos:

`pos`	Posición del elemento.
`list`	Lista.
`nlist`	Argumento auxiliar que devuelve la lista modificada.

Ejemplos:

Input	Output
`>(remove-elem 4 '(a b c d e))`	`(A B C E)`
`>(remove-elem 2 '((1 2) (3 4) (5 6)))`	`((1 2) (5 6))`

Función: `equal-list-p`

```
(defun equal-list-p (list1 list2 &optional (flag nil))
  (cond ((and (null list1) (null list2)) T)
    ((or (null list1) (null list2)) nil)
    ((equalp 0 flag) nil)
    (T (let* ((flag (pos-elem (car list1) list2))
         (list2 (if (equalp 0 flag)
           list2
           (remove-elem flag list2))))
      (equal-list-p (cdr list1) list2 flag)))))
```

Descripción:

Predicado de igualdad entre dos listas (T) y (nil).

Argumentos:

`list1`	Primer lista a comparar.
`list2`	Segunda lista a comparar.
`flag`	Argumento auxiliar que funciona como bandera.

Ejemplos:

```
>  (equal-list-p '(1 2 3 4) '(1 2 3 5))
NIL
>  (equal-list-p '(1 2 3 4) '(1 2 3 4))
T
>  (equal-list-p '(1 (2) 3 4) '(1 (2) 3 4))
T
>  (equal-list-p '((a a) 1 (b) 3 4 c) '((a a) 1 (b) 3 4 c))
T
>  (equal-list-p '((a a) 1 (b) 3 4 c) '((a) 1 (b) 3 4 c))
NIL
```

Función: `prod-cruz`

```
(defun prod-cruz (list &optional (nlist nil) (alist nil))
   (labels ((x-cruz (x list &optional (alist nil))
     (cond ((null list) alist)
          (T (if (null alist)
            (let ((alist (push (list x (car list)) alist)))
              (x-cruz x (cdr list) alist))
            (let ((alist (push (list x (car list)) alist)))
              (x-cruz x (cdr list) alist)))))))
     (cond ((null list) (reverse nlist))
       (T (if (null alist)
         (let ((alist list)
              (nlist (x-cruz (car list) list)))
         (prod-cruz (cdr list) nlist alist))
       (let ((nlist (append (x-cruz (car list) alist) nlist)))
         (prod-cruz (cdr list) nlist alist)))))))
```

Descripción:

Realiza el producto cruz de los elementos en `list`.

Argumentos:

`list`	Lista donde se almacena el conjunto.
`nlist`	Lista de salida donde se almacenan por parejas los elementos del `list` × `list`.
`alist`	Argumento auxiliar de almacenamiento.

Ejemplos:

```
>  (prod-cruz '(a b c))
   ((A A) (A B) (A C) (B A) (B B) (B C) (C A) (C B) (C C))
>   (prod-cruz '(a $ 1))
   ((A A) (A $) (A 1) ($ A) ($ $) ($ 1) (1 A) (1 $) (1 1))
>   (prod-cruz '((a b) (1 2) ($ %)))
   (((A B) (A B)) ((A B) (1 2)) ((A B) ($ %)) ((1 2) (A B))
   ((1 2) (1 2)) ((1 2) ($ %)) (($ %) (A B)) (($ %) (1 2)) (($ %)
   ($ %)))
```

Función: `insert`

```
(defun insert (x list pos)
   (if (or (eql pos 1) (eql list nil))
       (cons x (cdr list))
       (cons (car list) (insert x (cdr list) (- pos 1)))))
```

Descripción:

Inserta el elemento `x` en la posición `pos` sustituyendo al miembro existente de la lista `list`.

Argumentos:

`x`	Elemento a insertar.
`list`	Lista.
`pos`	Posición en la lista donde se colocará el nuevo miembro.

Ejemplos:

```
> (insert 'a '(1 2 3 4 5) 3)
  (1 2 A 4 5)
> (insert '(1 2) '(a b c d e) 4)
  (A B C (1 2) E)
> (insert 5 '($ $ $ $ $) 3)
  ($ $ 5 $ $)
```

Función: `cerolist`

```
(defun cerolist (n)
     (do* ((i 0 (1+ i))
           (list nil (cons 0 list)))
        ((<(1- n) i) list)))
```

Descripción:

Genera un vector de ceros de tamaño n.

Argumentos:

`n`	Número natural

Ejemplos:

```
> (cerolist 10)
  (0 0 0 0 0 0 0 0 0 0)
```

Función:

```
(defun zero-list-p (list &optional (flag nil))
   (cond ((and (null list) (equalp flag 0)) T)
       ((null list) nil)
       (T (if (equalp (car list) 0)
         (zero-list-p (cdr list) 0)
         (zero-list-p nil nil)))))
```

Descripción:

Predicado que verifica si una lista es de ceros.

Argumentos:

`list`	Lista.
`flag`	Argumento auxiliar, bandera.

Ejemplos:

```
>     (zero-list-p '(0 0 0 g))
      NIL
>     (zero-list-p '(0 0 0 0))
      T
```

Referencias

[1] Aguilar, B. R. *Álgebras C^* y Grupos Cuánticos Compactos*. Tesis de Licenciatura en Matemáticas, Facultad de Ciencias, UNAM, 2006.

[2] Ahmed K. Noor. *List of Books, Monographs, Conference Proceedings and short courses on Symbolic Computations*, Winter Annual Meeting of the American Society of Mechanical Engineers (ISBN 0-7918-0589-0), pp. vii-xii, 1990.

[3] Armstrong, M.A. *Groups and Symmetry*. Springer, 1988.

[4] Buchberger B, Keppler J. *Journal of Symbolic Computation*, Vol 1, Academic Press, Universität, Institut für Mathematik, A-4040 Linz, Austria 1985.

[5] Caviness B. F. *Computer Algebra: PAST AND FUTURE*, Springer-Verlag Lecture Series on Computer Science, 1985.

[6] Durdevich, M. *Geometry of Quantum Principal Bundles I*, Commun Math Phys 175 (3) 457–521 (1996).

[7] Durdevich, M. *Geometry of Quantum Principal Bundles III–Structure of Calculi and Around, Algebras, Groups and Geometries*, Vol 27, 247-336 (2010).

[8] Erdos, J. A. $C^* - Algebras$. Department of Mathematics King's College London WC2R 2LS England.

[9] Herstein I.N. *Álgebra Abstracta*. Grupo Editorial Iberoamericano.1988.

[10] Hungerford T. W. *Algebra Graduate Texts in Mathematics*. Springer–Verlag. 1974.

[11] Hungerford T. W. *Abstract Algebra An Introduction*. Har-court College Publishers. 1997.

[12] Rotman J. J. *An Introduction to the Theory of Groups*. Graduate Texts in Mathematics. Springer–Verlag. 1994 North-Holland,1977.

[13] Pavelle R, Rothstein M, and Fitch J. *Computer Algebra*, Scientific American, Vol 245, Nr. 6 pp. 136-152. 1981.

[14] Touretzky, D. S. *COMMON LISP: A Gentle Introduction to Symbolic Computation*. Carnegie Mellon University, The Benjamin/Cummings Publishing Company, Inc. 1990.

[15] Wegge-Olsen, N. E. *K-Theory and C*-Algebras*, Oxford Science Publications 1993.

[16] Woronowicz, S. L. *Differential Calculus on Compact Matrix Pseudogroups (Quantum Groups)*, Department of Mathematical Methods in Physics, Faculty of Physics, University of Warsaw, Hoza 74, PL-00-682 Warszawa, Poland, 1989.

[17] Woronowicz, S. L. *Twisted $SU(2)$ Group. An Example of a Non-Commutative Differencial Calculus*. Publi. RIMS Kyoto Univ. 23 (1987), 117–181.

[18] Zaldívar F. *Introdución a la teoría de grupos*. Aportacioens Matemáticas, texto 32. Sociedad Matemática Mexicana 2006.

[19] Sitio web, Micho Durdevich: `http://www.matem.unam.mx/~micho/`

[20] Sitio web, Maxima: `http://maxima.sourceforge.net/es/`

[21] Sitio web, wxMaxima: `http://wxmaxima.sourceforge.net/wiki/index.php/Main_Page`

[22] Sitio web, para las distintas versiones de Lisp `http://www.rodoval.com/paginalen.php?len=Lisp`

[23] Sitio web, Universidad de Sevilla: `http://departamento.us.es/da/planantiguo/notas-ant/algebra/t11.pdf`