



UNIVERSIDAD NACIONAL DE COLOMBIA

Desarrollo de un programa de computador para el análisis lineal de estructuras aporticadas tridimensionales sometidas a cargas estáticas

Cristian Danilo Ramirez Vargas

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de Ingeniería Civil y Agrícola
Bogotá D. C., Colombia
2022

Desarrollo de un programa de computador para el análisis lineal de estructuras aporticadas tridimensionales sometidas a cargas estáticas

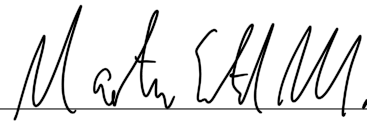
Cristian Danilo Ramirez Vargas

Tesis presentada como requisito parcial para optar al título de:
Magíster en Estructuras

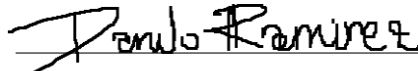
Director(a):
Ph. D. Martín Estrada Mejía

Línea de Investigación:
Análisis de estructuras
Grupo de Investigación:
Análisis, diseño y materiales - GIES

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de Ingeniería Civil y Agrícola
Bogotá D. C., Colombia
2022

A handwritten signature in black ink, appearing to read 'Martín Estrada Mejía', written over a horizontal line.

Firma del director de la tesis
Ph. D. Martín Estrada Mejía

A handwritten signature in black ink, appearing to read 'Cristian Danilo Ramirez Vargas', written over a horizontal line.

Firma del autor de la tesis
Cristian Danilo Ramirez Vargas

"The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform. It can follow analysis; but it has no power of anticipating any analytical relations or truths."

— Augusta Ada King, Countess of Lovelace (1815-1852)

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

— Martin Fowler

Agradecimientos

No habría podido hacer este trabajo sin la dirección del profesor Martín Estrada. Su conocimiento del mundo de la programación me ayudó en momentos decisivos durante el desarrollo del código. Gracias a él trabajé con la librería *Three.js*. No sé como hacer para agradecerle por su paciencia.

Este trabajo también se debe al curso Computación Visual del profesor Jean Pierre Charalambos. Su descripción del *grafo* y de como trabajar con la *escena* fue lo que me permitió hacer *mas.js*.

Adicionalmente, apliqué el concepto de cuaternión en el problema de la rotación de los ejes de referencia tiempo después de haberlo estudiado en una de sus clases, lo que me permitió implementar el método de análisis matricial de manera innovadora. Gracias a su curso ahora creo entender muchas cosas que de adolescente siempre quise saber.

También quiero agradecer a la profesora Maritzabel Molina ya que mi entendimiento del método de análisis matricial proviene de su curso de Análisis Estructural Básico. A ella nos debemos muchos ingenieros estructurales.

Así mismo, quiero agradecer al profesor Fernando Ramírez, de la Universidad de los Andes, por enseñarme el método de los elementos finitos, y al profesor Dorian Linero por enseñarme a implementarlo. A ellos gracias por haberme permitido ganar confianza con el método.

Finalmente, quiero agradecer la ayuda de la Coordinación Curricular del Posgrado en Estructuras, especialmente a la profesora Caori Takeuchi que no tuvo reparos en dejarme ver el curso de Computación Visual. Ese día comenzó la verdadera tesis.

Resumen

Desde hace más de 50 años, el método directo de rigideces ha sido ampliamente utilizado para analizar estructuras aporticadas. En esta tesis se implementó el método en la librería **pymas**, la cual podría ser el motor de análisis de cualquier programa de computador. De esta manera, los usuarios de la librería sólo tendrán que preocuparse por describir el problema estructural.

Una novedad en esta implementación es la aplicación de los cuaterniones en el cálculo de la matriz de transformación de rotación. Los cuaterniones son una extensión de los números reales, similar a la de los números complejos. Fueron descritos por primera vez por el matemático irlandés William Hamilton en 1843 y tienen uso práctico en cálculos que involucran rotaciones en el espacio.

También se desarrolló el programa de computador **mas.js** para visualizar estructuras tridimensionales y los resultados del análisis. Este programa usa tecnologías web (HTML, CSS y JavaScript), tendencia entre los desarrolladores actualmente, gracias a la posibilidad de ejecutar aplicaciones a través del navegador de internet.

Estos programas de computador importan y exportan la información del modelo a través de archivos de texto en formato JSON, donde la información se almacena en un esquema cómodo para su lectura. De esta manera, **pymas** puede ser instalado en un servidor web que reciba peticiones para solucionar modelos estructurales desde cualquier parte con conexión a internet. El resultado, a su vez, puede ser interpretado por **mas.js**, ejecutándose en el navegador web.

Este fue el esquema que se utilizó en la página de internet <https://rvcristiand.github.io/mas.js> para proveer el servicio de análisis estructural en línea. Sin embargo, ni **pymas** ni **mas.js** se limitan a este uso y sus aplicaciones pueden llegar a ser muy variadas.

Palabras clave: análisis estructural, resistencia de materiales, computación gráfica, gráficos de computadora en 3D.

Abstract

For more than 50 years, the direct method stiffness has been widely used to analyze frames structures. In this thesis, the method was implemented in the **pymas** library, which could be the analysis engine of any computer program. In this way, users of the library will not have to worry about how to solve the structural problem.

In this implementation, the use of quaternions is the main method to compute the rotation transformation matrix. Quaternions are an extension of real numbers, similar to that of complex numbers. They were first described once by the irish mathematician William Hamilton in 1843 and have practical uses in calculations that involve rotations in space.

The computer program **mas.js** was also developed to visualize three-dimensional structures and the results of the analysis. This program uses web technologies (HTML, CSS, and JavaScript), a trend among today's developers due to the possibility of running applications on the internet browser.

These programs comunicate with each other through text files in JSON format, where the information of the model is stored in a comfortable readable way. With this scheme **pymas** can be installed on a web server that receives requests to solve structural models. The result can then be interpreted by **mas.js**, executed in the web browser.

This is what was used on the website <https://rvcristiand.github.io/mas.js> to provide the online structural analysis service. However, neither **pymas** nor **mas.js** are limited to that, and their applications could be unlimited.

Keywords: structural analysis, direct stiffness method, computer graphics, 3D computer graphics.

Contenido

Agradecimientos	IX
Resumen	XI
Lista de figuras	XIX
Lista de tablas	XXI
Lista de algoritmos	XXI
1 Introducción	1
1.1 Argumento	6
1.2 Objetivos	7
1.2.1 General	7
1.2.2 Específicos	7
1.3 Metodología	7
1.3.1 Lenguajes de programación	8
1.3.2 Licencias	11
1.3.3 Control de versiones	12
1.3.4 Formato de intercambio de datos	13
1.3.5 Programas desarrollados	13
1.4 Estructura del documento	16
2 Análisis matricial de estructuras aporticadas	17
2.1 Método de rigideces	18
2.2 Matriz de rigidez de elementos los aporticados	24
2.3 Formalización del método de rigideces	25
2.4 Método directo de rigideces	27
2.4.1 Matriz de transformación de rotaciones	28
2.4.2 Solucion del sistema de ecuaciones	29

3 Programa de análisis estructural pymas	31
3.1 Generalidades de pymas	32
3.1.1 Arquitectura del programa	32
3.1.2 Paradigma de programación	34
3.1.3 Estandarización de sintáxis	35
3.2 Clase Structure (primera parte)	37
3.3 Clases específicas	43
3.3.1 Material	44
3.3.2 Section	45
3.3.3 RectangularSection	46
3.3.4 Joint	48
3.3.5 Frame	49
3.3.6 Support	68
3.3.7 LoadPattern	70
3.3.8 PointLoadAtJoint	76
3.3.9 PointLoadAtFrame	79
3.3.10 DistributedLoad	82
3.3.11 Displacement	84
3.3.12 EndActions	86
3.3.13 Reaction	89
3.3.14 InternalForces	91
3.3.15 InternalDisplacements	92
3.4 Clase Structure (segunda parte)	94
3.4.1 set_flags_active_joint_displacements	96
3.4.2 get_flags_active_joint_displacements()	96
3.4.3 get_number_active_joint_displacements()	97
3.4.4 set_indexes()	97
3.4.5 get_indexes()	98
3.4.6 get_stiffness_matrix()	98
3.4.7 set_stiffness_matrix_with_support()	100
3.4.8 get_stiffness_matrix_with_supports()	101
3.4.9 solve_load_pattern()	102
3.4.10 solve()	105
3.4.11 export()	106
3.5 Otras clases	108
3.5.1 AttrDisplay	108

4 Programa de pre y posproceso mas.js	110
4.1 Generalidades de mas.js	110
4.1.1 Arquitectura del programa	111
4.1.2 Visualización y funcionamiento interno del programa	112
4.2 Interacción del usuario con el programa	120
4.2.1 open()	120
4.2.2 addMaterial()	128
4.2.3 addSection()	128
4.2.4 addRectangularSection()	129
4.2.5 addJoint()	130
4.2.6 addFrame()	132
4.2.7 addLoadPattern()	135
4.2.8 addLoadAtJoint()	136
4.2.9 addUniformlyDistributedLoadAtFrame()	138
4.2.10 setFrameView()	141
4.2.11 setSupportModel()	142
4.3 La variable model	143
5 Ejemplos de aplicación	146
5.1 Funcionamiento de la librería pymas	146
5.1.1 Pórtico tridimensional	148
5.2 Funcionamiento del programa de computador mas.js	152
5.2.1 Cercha tridimensional	155
6 Conclusiones	164
Bibliografía	167

Lista de Figuras

1-1	Flujo de información completo entre <code>mas.js</code> y <code>pymas</code>	16
2-1	Viga con un extremo empotrado y el otro simplemente apoyado, solicitada a una carga uniformemente distribuida.	18
2-2	Estructura restringida de la viga de la figura 2-1 solicitada a la carga externa y al desplazamiento desconocido.	19
2-3	Estructura restringida de la viga de la figura 2-1 solicitada a un desplazamiento desconocido unitario.	20
2-4	Viga de dos luces con un extremo empotrado y el otro simplemente apoyado, solicitada a cargas aplicadas en las luces y sobre los apoyos. .	21
2-5	Estructura restringida de la viga de la figura 2-4 solicitada a las cargas externas.	22
2-6	Desplazamientos unitarios D_1 y D_2 introducidos separadamente en la estructura restringida.	23
2-7	Elemento aporticado con sus grados de libertad numerados y su sistema de coordenadas local.	24
3-1	Estructura de los archivos del programa <code>pymas</code>	32
3-2	Licencia del programa <code>pymas</code>	33
3-3	Ejemplo de métodos y atributos de la clase hipotética <code>Perro</code>	34
3-4	Métodos y atributos de la clase <code>Structure</code>	38
3-5	Métodos y atributos de la clase <code>Frame</code>	50
3-6	Elemento aporticado con sus grados de libertad y su sistema de coordenadas local.	51
3-7	Descripción de la rotación de una esfera a través de una rotación alrededor de un eje que pasa por su centro.	53
3-8	Métodos y atributos de la clase <code>LoadPattern</code>	70
3-9	Métodos y atributos de la clase <code>Structure</code>	95
4-1	Estructura de los archivos del programa <code>mas.js</code>	111
4-2	Licencia del programa <code>mas.js</code>	112
4-3	<code>mas.js</code> ejecutándose en el navegador web Firefox.	113

4-4	mas.js con colores del fondo de la escena arbitrarios.	117
4-5	mas.js almacenando la configuración del usuario en la variable <code>localStorage</code>	117
4-6	Archivo <code>space_frame.json</code> abierto con mas.js.	122
4-7	Fuerzas internas de los elementos aporticados del archivo <code>space_frame.json</code> en mas.js.	123
4-8	Configuración deformada del archivo <code>space_frame.json</code> en mas.js. . .	124
4-9	Representación del <code>space_frame.json</code> en <i>estructura de palillos</i> o <i>extruído</i> . 142	
4-10	Representación de los apoyos del archivo <code>space_frame.json</code> en mas.js	143
4-11	Eje local z de la variable <code>model</code> apuntando hacia arriba de la pantalla.	143
4-12	Grafo de la variable <code>model</code> después de agregar nodos, elementos aporticados y cargas al modelo.	145
5-1	Estructura de los archivos del programa <code>pymas</code>	147
5-2	Pórtico tridimensional del <i>Ejemplo 7.6</i> de Uribe-Escamilla, 1995. . . .	149
5-3	Estructura de los archivos del programa <code>mas.js</code>	152
5-4	El programa de computador <code>mas.js</code> ejecutándose en el navegador de internet Firefox.	153
5-5	El archivo <code>space_frame.json</code> en el programa de computador <code>mas.js</code> . .	154
5-6	Cargas externas actuando en la estructura aporticada y la estructura deformada debida a dichas cargas.	154
5-7	Momentos flectores z a lo largo de los elementos aporticados con diferentes escalas.	155
5-8	Cercha tridimensional en <code>mas.js</code>	155
5-9	Opciones disponibles de <code>mas.js</code> a medida que se escribe en la terminal. .	156
5-10	Contenido del modelo al ejecutar la funcion <code>getStructure()</code>	157
5-11	Modelo en <code>mas.js</code> después de agregar los nodos de la estructura.	158
5-12	Modelo en <code>mas.js</code> después de agregar los elementos aporticados de la estructura.	159
5-13	Modelo en <code>mas.js</code> después de agregar los apoyos de la estructura.	160
5-14	Modelo en <code>mas.js</code> después de agregar las cargas puntuales.	161
5-15	Fuerza axial de los elementos de la cercha tridimensional sujetas a las cargas externas.	163

Lista de Tablas

1-1	Programas para el análisis estructural.	4
1-1	Programas para el análisis estructural.	5
1-2	Lenguajes de programación más populares según la encuesta de Stack Overflow del año 2021.	9
5-1	Desplazamientos en los nodos del modelo para el patrón de carga <i>distributed loads</i>	151
5-2	Reacciones de los apoyos del modelo para el patrón de carga <i>distributed loads</i>	151
5-3	Desplazamientos en los nodos del modelo para el patrón de carga <i>point loads</i>	162
5-4	Reacciones de los apoyos del modelo para el patrón de carga <i>point loads</i>	162

Lista de Algoritmos

3.1. Documentación de las clases en Python	36
3.2. Constructor de la clase <code>Structure</code>	37
3.3. Método <code>add_section()</code> de la clase <code>Structure</code>	40
3.4. Método <code>add_rectangular_section()</code> de la clase <code>Structure</code>	41
3.5. Método <code>add_support()</code> de la clase <code>Structure</code>	41
3.6. Método <code>add_load_at_joint()</code> de la clase <code>Structure</code>	42
3.7. Clase <code>Material</code> implementada en el archivo <code>primitives.py</code>	44
3.8. Clase <code>Section</code> implementada en el archivo <code>primitives.py</code>	45
3.9. Clase <code>RectangularSection</code> implementada en el archivo <code>primitives.py</code>	47
3.10. Clase <code>Joint</code> implementada en el archivo <code>primitives.py</code>	48
3.11. Constructor de la clase <code>Frame</code>	51
3.12. Método <code>get_length()</code> de la clase <code>Frame</code>	52
3.13. Método <code>get_direction_cosines()</code> de la clase <code>Frame</code>	52
3.14. Método <code>get_rotation()</code> de la clase <code>Frame</code>	55
3.15. Método <code>get_rotation_matrix()</code> de la clase <code>Frame</code>	57
3.16. Método <code>get_local_stiffness_matrix()</code> de la clase <code>Frame</code>	59
3.17. Método <code>get_global_stiffness_matrix()</code> de la clase <code>Frame</code>	62
3.18. Método <code>get_internal_forces()</code> de la clase <code>Frame</code>	62
3.19. Método <code>get_internal_displacements()</code> de la clase <code>Frame</code>	65
3.20. Clase <code>Support</code> implementada en el archivo <code>primitives.py</code>	68
3.21. Constructor de la clase <code>LoadPattern</code>	71
3.22. Método <code>add_point_load_at_joint()</code> de la clase <code>LoadPattern</code>	71
3.23. Método <code>add_point_load_at_frame()</code> de la clase <code>LoadPattern</code>	72
3.24. Método <code>add_distributed_load()</code> de la clase <code>LoadPattern</code>	73
3.25. Método <code>get_f()</code> de la clase <code>LoadPattern</code>	74
3.26. Método <code>get_f_fixed()</code> de la clase <code>LoadPattern</code>	75
3.27. Clase <code>PointLoad</code> implementada en el archivo <code>primitives.py</code>	77
3.28. Clase <code>PointLoadAtFrame</code> implementada en el archivo <code>primitives.py</code>	79
3.29. Clase <code>DistributedLoad</code> implementada en el archivo <code>primitives.py</code>	82
3.30. Clase <code>Displacement</code> implementada en el archivo <code>primitives.py</code>	84

3.31. Clase <code>EndActions</code> implementada en el archivo <code>primitives.py</code>	86
3.32. Clase <code>Reaction</code> implementada en el archivo <code>primitives.py</code>	89
3.33. Clase <code>InternalForces</code> implementada en el archivo <code>primitives.py</code> . . .	91
3.34. Clase <code>InternalDisplacements</code> implementada en el archivo <code>primitives.py</code> . .	93
3.35. Método <code>set_flags_active_joint_displacements()</code> implementado en el archivo <code>core.py</code>	96
3.36. Método <code>get_flag_active_joint_displacements()</code> de la clase <code>Structure</code> . .	96
3.37. Método <code>get_number_active_joint_displacements()</code> de la clase <code>Struc</code> <code>ture</code>	97
3.38. Método <code>set_indexes()</code> de la clase <code>Structure</code>	98
3.39. Método <code>get_indexes()</code> de la clase <code>Structure</code>	98
3.40. Método <code>get_stiffness_matrix()</code> de la clase <code>Structure</code>	99
3.41. Método <code>set_stiffness_matrix_with_support()</code> de la clase <code>Structure</code> . .	100
3.42. Método <code>get_stiffness_matrix_with_support()</code> de la clase <code>Structure</code> . .	101
3.43. Método <code>solve_load_pattern()</code> de la clase <code>Structure</code>	102
3.44. Método <code>solve()</code> de la clase <code>Structure</code>	105
3.45. Clase <code>AttrDisplay</code> implementada en el archivo <code>classtools.py</code>	108
4.1. Pseudocódigo de la función <code>init()</code> del archivo <code>mas.js</code>	113
4.2. Valores por defecto para configurar <code>mas.js</code>	115
4.3. Implementación de la función <code>createModel()</code> del archivo <code>mas.js</code>	118
4.4. Implementación de la función <code>createStructure()</code> del archivo <code>mas.js</code> . .	119
4.5. Implementación de la función <code>render()</code> del archivo <code>mas.js</code>	120
4.6. Implementación de la función <code>open()</code> del archivo <code>mas.js</code>	121
4.7. Función <code>addMaterial()</code> implementada en el archivo <code>mas.js</code>	128
4.8. Función <code>addSection()</code> implementada en el archivo <code>mas.js</code>	129
4.9. Función <code>addRectangularSection()</code> implementada en el archivo <code>mas.js</code> . .	130
4.10. Función <code>addJoint()</code> implementada en el archivo <code>mas.js</code>	131
4.11. Función <code>addFrame()</code> implementada en el archivo <code>mas.js</code>	132
4.12. Función <code>addLoadPattern()</code> implementada en el archivo <code>mas.js</code>	135
4.13. Función <code>addLoadAtJoint()</code> implementada en el archivo <code>mas.js</code>	136
4.14. Función <code>addUniformlyDistributedLoadAtFrame()</code> implementada en el archivo <code>mas.js</code>	139
4.15. Función <code>setFrameView()</code> implementada en el archivo <code>mas.js</code>	141
4.16. Función <code>setSupportMode()</code> implementada en el archivo <code>mas.js</code>	142
4.17. Implementación de la función <code>setAxesShaftLength()</code> del archivo <code>mas.js</code> . .	144
5.1. Archivo <code>makepath</code> de la carpeta <code>examples</code> del repositorio <code>pymas</code>	147

5.2.	Ingreso de los datos del modelo de la estructura a <code>pymas</code>	148
5.3.	Ejecutar el archivo <code>space_frame.py</code> con el intérprete de Python. . . .	150
5.4.	Ejecutar un servidor local con Python para abrir el programa <code>mas.js</code> . .	153
5.5.	Algoritmo del archivo <code>mas.js.py</code> para analizar el archivo <code>space_truss</code> <code>.json</code> , creado con el programa <code>mas.js</code>	161

1 Introducción

Los programas de computador para el análisis de estructuras aporticadas constituyen la herramienta tecnológica más poderosa que tienen los ingenieros para calcular la respuesta mecánica de estructuras compuestas por elementos prismáticos unidimensionales sometidos a diferentes patrones de carga. Algunos de estos programas de computador implementan el método directo de rigideces, el más común entre los métodos de análisis estructural, tanto para programas comerciales como a código abierto.

El fundamento teórico de los método matriciales fue propuesto por George A. Maney en 1915, quien desarrolló el método de pendiente deflexión. Por esto, algunos lo consideran como el precursor del método matricial de las rigideces. A pesar de sus bondades como método de análisis, su principal desventaja en esa época fue la solución simultanea de múltiples ecuaciones algebraicas (Kassimali, 2011).

Con la llegada de los computadores electrónicos a comienzos de la década de 1960, aparecieron los primeros programas de análisis estructural. Uno de esos programas fue STRESS (*STRuctural Engineering System Solver*), desarrollado bajo la dirección del profesor S. J. Fenves, de la Universidad de Illinois, durante su estancia como profesor invitado en el Massachusetts Institute Technology, en el año académico 1962-1963.

El propósito de STRESS fue facilitar el uso del computador en el análisis de estructuras. Para lograrlo, se debía mejorar la comunicación entre el ingeniero estructural, que normalmente no es experto en programación, y el computador. El sistema STRESS fue un intento por lograr este objetivo, mediante el uso de un lenguaje de programación orientado al problema estructural, que le permitía al ingeniero escribir completamente el programa de entrada en tarjetas perforadas para resolver un problema estructural, aunque no tuviera experiencia en programación (Fenves, 1965).

Debido a que muchos ingenieros han escrito programas de análisis estructural para aprovechar la capacidad de cálculo de los computadores, una reseña que los incluya a todos sería casi interminable. En Uribe-Escamilla (1995) se presenta una buena selección de algunos de estos programas de uso común en América Latina, entre los que se destaca

ETABS (*Three Dimensional Analysis of Building Systems - Extended Version*).

ETABS es un programa de computador creado por Edward Wilson, Jeffery Hollings y Henry Dovey en 1975. Según Wilson *et al.* (1975), este programa de computador fue desarrollado para el análisis estructural lineal de edificios de pórticos y muros a cortante sujetos a cargas estáticas y sísmicas. En este programa, el edificio es idealizado como un sistema estructural aporticado y muros a cortante independientes, interconectado por losas de entrepiso rígidas en su propio plano.

Este programa es una extensión de TABS (*Three Dimensional Analysis of Building Systems*), reemplazando los elementos vigas y columnas programados por los elementos aporticados tridimensionales de SAP IV, para resolver el problema de compatibilidad de deformaciones axiales en las columnas comunes a varias subestructuras (EERI, 2016).

Según Wilson y Dovey (1972), una de las razones para desarrollar TABS fue darle una retroalimentación a los usuarios de los programas de computador FRMSTC (*Static Load Analysis of High-Rise Buildings*), FRMDYN (*Dynamic Analysis of Multistory Buildings*), LATERAL y SOLID SAP (*Static Analysis Program for Three-Dimensional Solid Structures*), usados en el diseño de varias estructuras importantes en varios países para ese entonces.

Edward L. Wilson y Henry Harvey Dovey convirtieron FRMDYN en un sistema de análisis de edificios en tres dimensiones, al que llamaron TABS, para modelar el comportamiento torsional de los edificios sometidos a los desplazamientos debidos a un sismo (EERI, 2016).

El programa FRMSTC permitía analizar edificios simétricos de elementos aporticados y muros a cortante paralelos sometidos a cargas estáticas. Los modos laterales y las frecuencias también eran evaluadas.

El programa FRMDYN era el mismo que FRMSTC a excepción que la carga es la aceleración del terreno debido al acelerograma de un sismo, pero los resultados no se combinaban con los de las cargas estáticas.

El programa LATERAL es una extensión de FRMSTC para analizar subestructuras aporticadas y muros a cortante no necesariamente paralelos entre si. Existían tres grados de libertad en cada nivel. Este programa no tenía opciones de análisis dinámico.

El programa SOLID SAP se escribió como un programa general de elementos finitos con una opción particular para considerar la aproximación de diafragma rígido. Se podían realizar análisis estáticos y dinámicos a las estructuras.

Los programas enunciados anteriormente fueron algunos de los precursores de los programas actuales para el análisis de estructuras, pero su interacción con los usuarios, exclusivamente mediante archivos de texto y ventanas de comandos, no tuvo una completa aceptación entre los ingenieros. Esto evolucionó, naturalmente, en desarrollos importantes en dos frentes: los programas de análisis y las interfaces de pre y posproceso.

En la actualidad, ETABS se encuentra en la versión 19.1.0 y permite hacer análisis estructurales estáticos y dinámicos a una gran variedad de estructuras, considerando linealidad o no linealidad material o geométrica (Computers & Structures, 2021). Además, cuenta con una interfaz gráfica amigable. A través de múltiples cuadros de diálogo el usuario puede modelar la estructura que desea analizar al describir los materiales, las secciones transversales, los elementos estructurales, las condiciones de apoyo, los diafragmas y las cargas, entre otras características.

Según Computers y Structures (2017), ETABS analiza el modelo usando el motor de análisis SAPFire, el cual es utilizado también por otros programas de la misma compañía (SAP2000, SAFE y CSiBridge). SAPFire es la última versión de la serie de programas SAP y ofrece las siguientes funcionalidades, después de 60 años de desarrollo:

- análisis estático y dinámico,
- análisis lineal y no lineal,
- análisis sísmico y análisis incremental no lineal (*pushover*),
- análisis de cargas móviles,
- no linealidad geométrica, incluyendo efectos P-delta y grandes desplazamientos,
- etapas constructivas,
- fluencia lenta (*creep*), retracción (*shrinkage*) y envejecimiento,
- análisis de pandeo,
- análisis de densidad espectral de potencia y estado estacionario,

- elementos aporticados y laminares, incluyendo el comportamiento de vigas, columnas, cerchas, membranas y placas,
- elementos tipo cable y tendón,
- elementos bidimensionales planos y elementos sólidos asimétricos,
- elementos sólidos tridimensionales,
- resortes no lineales y apoyos, y
- propiedades de los resortes y apoyos dependientes de la frecuencia.

Algunas empresas, universidades, grupos de investigación y desarrolladores independientes han hecho programas de análisis estructural generales o para aplicaciones específicas, pero no todos ellos son conocidos o ampliamente utilizados. En la tabla **1-1** se presenta una lista de algunos programas de computador populares para el análisis estructural, en donde se incluye el tipo de licencia de cada uno, algunas características generales y si tienen o no interfaz gráfica de usuario.

Tabla 1-1: Programas para el análisis estructural.

Programa de computador	Desarrollador	Licencia y lenguaje de programación	Características	Interfaz gráfica
SAP2000, ETABS y CSiBridge	Computers & Structures (Estados Unidos)	Privativo FORTRAN	Paquete de software integrado para el análisis y diseño de edificios.	Sí
STAAD	Bentley Systems (Estados Unidos)	Privativo	<i>Framework</i> para simular la respuesta de sistemas estructurales y geotécnicos sujetos a sismos.	Sí
Midas Civil	MIDAS IT (Corea del Sur)	Privativo	Programa de elementos finitos para modelar, analizar y diseñar puentes	Sí

Tabla 1-1: Programas para el análisis estructural.

Programa de computador	Desarrollador	Licencia y lenguaje de programación	Características	Interfaz gráfica
SkyCiv	SkyCiv Engineering (Australia)	Privativo JavaScript	Software de análisis estructural en línea	Sí
RC-BUILDINGS	Engineering Solutions & Services, Inc. (Estados Unidos)	Privativo	Programa para el análisis y diseño de pórticos de concreto	Sí
OpenSees	Universidad de California (Estados Unidos)	No comercial C++	<i>Framework</i> para simular la respuesta de sistemas estructurales y geotécnicos sujetos a sismos.	No
ONSAS	Universidad de la República (Uruguay)	GPL Octave	Análisis estático y dinámico no lineal de vigas, cerchas y sólidos tridimensionales.	No

Adicionalmente, en el mercado existen programas que permiten resolver problemas numéricos con el método de los elementos finitos. Aunque es posible realizar análisis estructurales con algunos de ellos, no están diseñados específicamente para ello, por lo cual no se incluyen en la lista anterior.

En la Universidad Nacional de Colombia se han desarrollado varios programas de computador de análisis estructural, entre los cuales se encuentra ANALEST (Uribe-Escamilla, 1995); una serie de programas cuyo objeto es servir de ayuda en el análisis y diseño de estructuras.

Los programas están escritos en BASIC y su característica principal fue la facilidad de adaptación que tenía a cualquiera de los microcomputadores existentes en el mercado de la época. Permite analizar vigas continuas, armaduras planas, armaduras en el espacio, pórticos planos, pórticos en el espacio y parrillas planas. Además, los programas de

vigas y armaduras permiten encontrar la envolvente de momentos en los apoyos y de fuerzas axiales, respectivamente, como paso preliminar para el diseño de los miembros.

1.1. Argumento

Los ingenieros civiles se preparan para generar soluciones a problemas de infraestructura de gran escala, para lo cual usan herramientas computacionales en la resolución matemática de los modelos que utilizan. El desarrollo de un programa de computador o herramienta computacional de análisis estructural es el resultado de la aplicación del conocimiento del ingeniero con el fin obtener la respuesta mecánica de una estructura. Algunas características deseables en este tipo de desarrollo son:

- programarse en un lenguaje de programación versátil y amigable,
- tener un código fuente completamente abierto,
- permitir extender sus algoritmos en cualquier dirección, y
- ser libre para descargar y distribuir.

Por su parte, una interfaz gráfica de usuario se espera que cuente con los siguientes aspectos:

- conexión directa con el programa de análisis,
- exportación e importación de datos en formatos estándar,
- interfaz de usuario amigable, y
- expandible a plataformas BIM.

El área de análisis estructural no es ajena a esto. Y si bien existe un amplio conjunto de programas de computador disponible para analizar estructuras, muchos de ellos no son lo suficientemente versátiles para acomplarse en otros flujos de trabajo. Esto resulta limitando el potencial y las capacidades del ingeniero contemporáneo.

Actualmente, los ingenieros tienden a usar tecnologías y plataformas que le permiten trabajar en diferentes tipos de dispositivo, de manera colaborativa, y con la libertad necesaria para realizar cualquier tipo de análisis que se requiera. Sin embargo, los programas disponibles no fueron diseñados para tal fin. Si bien han aparecido recientemente

programas de computador con la intención de suplir algunas de las características nombradas, como el programa SkyCiv (tabla **1-1**), su carácter privativo no le permite a sus clientes revisar, entender o analizar su código fuente, ni mucho menos mejorarlo o extenderlo en su favor. Eso limita el potencial de dichos programas.

Con el presente trabajo se pretende continuar la línea de desarrollo de programas de computador para ingeniería, incluyendo la de análisis estructural, que ofrezcan las características nombradas en cuanto a libertades de uso, lectura, mejoría, distribución, versatilidad y uso de tecnologías actuales. De esta manera, este trabajo pretende contribuir con un programa de análisis estructural amigable y didáctico en tres ámbitos: primero, un ejercicio completamente libre de la profesión del ingeniero estructural; segundo, a la enseñanza del análisis de las estructuras en carreras de pregrado; y tercero, a la investigación en el área del análisis estructural.

1.2. Objetivos

1.2.1. General

Desarrollar un programa de computador a código abierto para el análisis lineal de estructuras aporticadas tridimensionales sometidas a cargas estáticas.

1.2.2. Específicos

- Desarrollar el módulo de análisis estructural para calcular el desplazamiento de los nodos, el valor de las reacciones y de las fuerzas internas de los elementos de estructuras sometida a cargas estáticas.
- Desarrollar el ambiente gráfico y la interfaz gráfica de usuario del programa de computador para permitirle al usuario ingresar los datos que describen la estructura, las cargas a las cuales se encuentra sometida y visualizar los resultados del análisis estructural.

1.3. Metodología

El desarrollo de programas o librerías de computador es una tarea que se puede ejecutar de múltiples maneras. Sin embargo, se considera necesario establecer una metodología de desarrollo organizada y normalizada para respetar los principios establecidos en la

sección 1.1. Esto permite desarrollar programas que respondan a las necesidades generales de los ingenieros estructurales, investigadores o desarrolladores y no a las necesidades personales de un individuo. A continuación se establecen algunos criterios generales que se tuvieron en cuenta en la tesis y más adelante se explican las tecnologías elegidas con mayor profundidad en los capítulos específicos para cada programa desarrollado.

1.3.1. Lenguajes de programación

Para que los computadores, tabletas, teléfonos y otros dispositivos realicen actividades específicas, se requiere un programa de computador que indique los comandos o instrucciones necesarios en un orden específico. Para ello existen diferentes tipos y características de lenguajes de programación, dependiendo de su filosofía y objetivos.

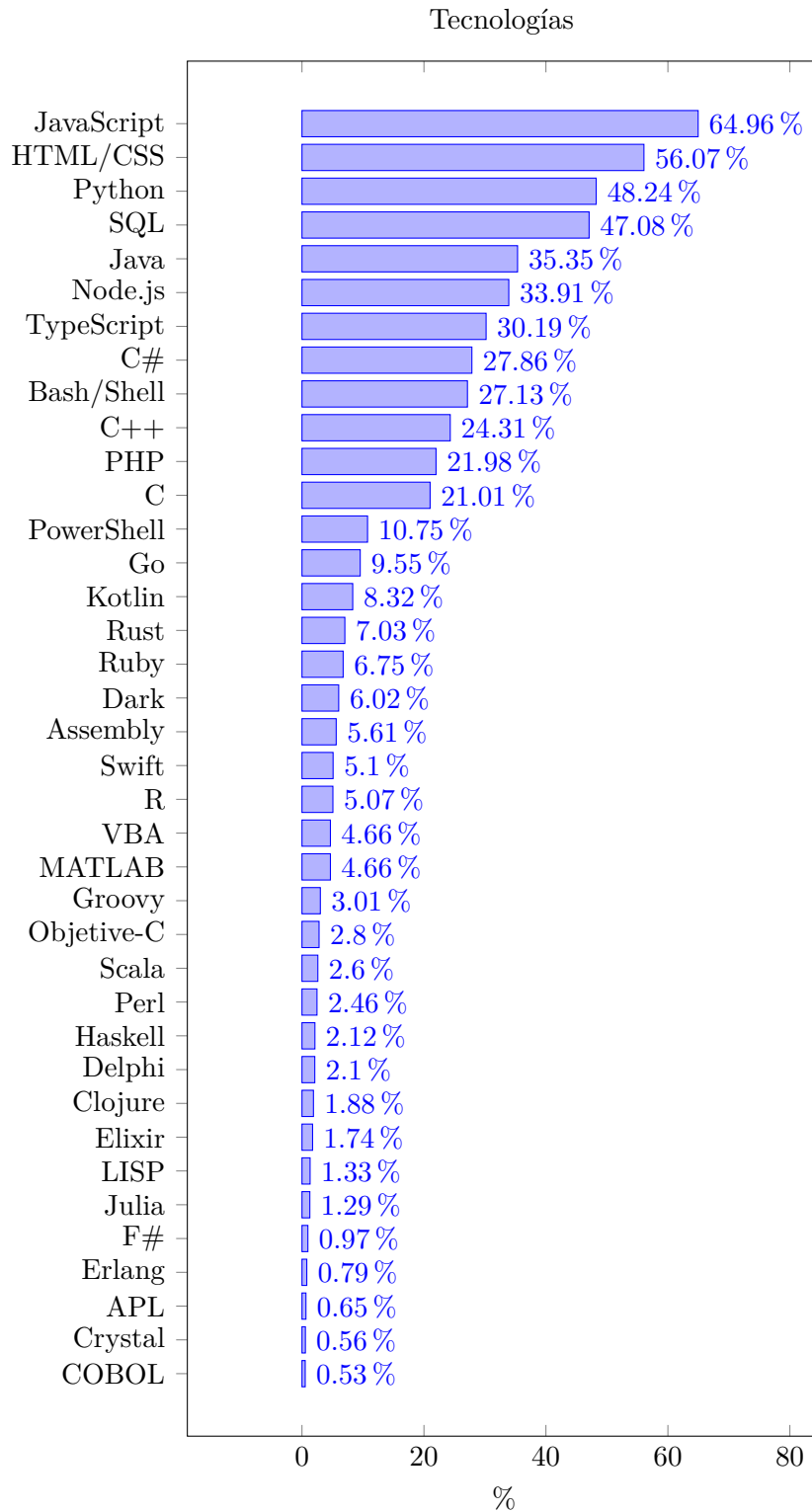
Un lenguaje de programación es un conjunto de reglas gramaticales que permiten dar instrucciones al computador. Dichas reglas gramaticales, más un conjunto de palabras clave, constituyen la sintáxis del lenguaje. De esta manera, cada lenguaje de programación tiene su propia sintáxis, diferente de la que usan los demás.

Existen muchos lenguajes de programación que pueden clasificarse de múltiples maneras. La escala más general de clasificación es el nivel: bajo o alto. Los lenguajes de bajo nivel consisten en ensambladores y lenguajes de máquina, lo que los hace muy rápidos, pero difíciles de leer por un humano. Por su parte, los lenguajes de alto nivel están diseñados para ser fáciles de leer, lo que los hace muy atractivos para los programadores. El trabajo colaborativo y portabilidad de los programas que se desarrollan en este trabajo requiere el uso de lenguajes de alto nivel.

Se realizó una revisión general a los lenguajes de programación más usados hoy en día. Se tomaron en cuenta algunos lenguajes de programación mencionados en la encuesta que cada año realiza la página de internet *Stack Overflow* a sus miembros (Overflow, 2020). Stack Overflow es una página de internet de preguntas y respuestas para programadores profesionales y aficionados. En la tabla **1-2** se presentan los resultados de los lenguajes de programación más populares entre los miembros de su comunidad para el año 2021.

A continuación se enuncian algunos de los lenguajes de programación de alto nivel más populares, acompañados de una pequeña descripción.

- **Python:** lenguaje de programación de propósito general que combina paradigmas procedimentales, funcionales y orientado a objetos.

Tabla 1-2: Lenguajes de programación más populares según la encuesta de Stack Overflow del año 2021.

- **MATLAB:** (MATrix LABoratory): es un lenguaje de programación patentado privativo de múltiples paradigmas y un entorno informático numérico desarrollado por MathWorks. MATLAB permite manipulaciones de matrices, trazado de funciones y datos, implementación de algoritmos, creación de interfaces de usuario e interactuar con programas escritos en otros lenguajes. Su principal diferencia con otros lenguajes consiste en que trabaja directamente con elementos tipo matriz y no con números separados.
- **JavaScript:** es un lenguaje de programación de alto nivel interpretado que soporta múltiples paradigmas y una escritura dinámica. Junto con HTML y CSS, JavaScript es una de las tecnologías principales de la web.
- **C:** es un lenguaje de programación de tipos de datos estáticos, debidamente tipado, de medio nivel, que dispone de las estructuras típicas de los lenguajes de alto nivel pero, a su vez, dispone de construcciones del lenguaje que le permite un control de bajo nivel.
- **C++:** es un lenguaje de programación que extiende al lenguaje de programación C para manipular objetos. El lenguaje se ha expandido significativamente a lo largo del tiempo, y actualmente cuenta con los paradigmas de programación orientado a objetos, genericos y funcionales, además de facilidades para la manipulación de memoria de bajo nivel. Por lo anterior, los tiempos de ejecución con C y C++ son considerablemente bajos.
- **Fortran:** es un lenguaje de programación de alto nivel que está especialmente adaptado al cálculo numérico y a la computación científica. Es uno de los lenguajes más populares en el área de la computación de alto rendimiento.
- **Rust:** es un lenguaje de programación compilado, de propósito general y multiparadigma que está siendo desarrollado por Mozilla. Soporta la programación funcional pura, por procedimientos, imperativa y orientada a objetos.
- **Go:** es un lenguaje de programación concurrente y compilado inspirado en la sintaxis de C, que intenta ser dinámico como Python y con el rendimiento de C o C++.

Paradigmas de programación

Independientemente del lenguaje de programación que se utilice, un problema se puede resolver de múltiples maneras o estilos. Los paradigmas de programación se refieren a las reglas que definen cada estilos de programación. Esto existe para ayudar al desarrollo

colaborativo y que diferentes personas puedan entender y modificar los códigos de otra persona. Los paradigmas de programación más comunes se encuentran enmarcados en dos grandes grupos: programación imperativa y programación declarativa. La programación imperativa consiste en desarrollar un proceso paso a paso, mientras que en la declarativa se pretende que el camino que toma el computador se haga de forma lógica, algo más parecido al razonamiento humano. A continuación se describen los paradigmas de programación más recurrentes.

- Imperativo
 - **procedimental:** las instrucciones se pueden indicar una tras otra en una sola ruta o se agrupan en subrutinas que se llaman en un orden específico.
 - **orientada a objetos:** las instrucciones y propiedades se almacenan en objetos que tienen la capacidad de alterar su propio estado y, en ocasiones, el de otros objetos. Por ejemplo, un objeto puede representar un carro, con características como color o marca, actividades como acelerar o frenar, y guardar el estado del tanque de combustible.
- Declarativo
 - **funcional:** el estado de los elementos o variables cambia a través de la aplicación de funciones. Se diferencia de los paradigmas de programación procedimental en tanto que las funciones no son simples partes del procedimiento general que se utilizan para mejorar la legibilidad, sino que sirven para alterar el estado de un elemento, como un texto o una matriz.
 - **lógico:** hay actividades o instrucciones que ocurren dependiendo del estado de algún elemento. Por ejemplo, un video reduce su resolución cuando la velocidad del internet es baja.

Los paradigmas de programación orientado a objetos y funcional tienen una característica en común importante. Ambos pretenden que la resolución de un problema se reduzca a la solución de muchos procesos pequeños, lo cual evita un gran número de errores durante el desarrollo.

1.3.2. Licencias

Las licencias de los programas de computador definen legalmente su uso y redistribución. Según los derechos que cada autor reserva sobre su obra, las licencias se pueden clasificar en: licencias de código abierto permisivas, licencias de código abierto robustas

y licencias de código cerrado.

Las licencias de código cerrado o privativas pueden aplicarse a programas gratis o de pago. Su carácter cerrado está en prohibir leer, estudiar, mejorar o distribuir su código fuente. Estas licencias impiden el desarrollo colaborativo de cualquier programa que las tenga.

Las licencias de código abierto robustas imponen algunas limitaciones en las obras derivadas, pudiendo obligar a hacerlas públicas. El objetivo de estas licencias es propiciar el libre uso y distribución de una obra, y evitar los intentos de apropiación que terminen reduciendo dichas libertades. Entre las licencias más famosas de este tipo se encuentra la *GNU General Public License*. La principal característica de esta licencia es que obliga a las copias y derivados de una obra a usar la misma licencia.

Las licencias de código abierto permisivas permiten que obras derivadas no tengan obligación con el autor original. La idea de estas licencias es imponer el mínimo de requisitos para redistribuir el programa de computador. Entre estas licencias vale la pena destacar la licencia MIT, la cual impone muy pocas limitaciones y es compatible con muchas otras. La licencia permite reutilizar el programa de computador en programas privativos. Así mismo, esta licencia es compatible con las licencias de código abierto robusta, como la GNU General Public License.

1.3.3. Control de versiones

Los sistemas de control de versiones son herramientas que ayudan a los desarrolladores a gestionar los cambios en el código fuente a lo largo del tiempo. A medida que los entornos de desarrollo se aceleran, los sistemas de control de versiones ayudan a los equipos de programación a trabajar de forma más rápida y ordenada.

Entre los sistemas de control de versiones disponibles se encuentran:

- **Git:** es un programa de computador para el control de versiones de archivos y coordinar el trabajo que varias personas realizan sobre los archivos de un repositorio (proyecto).
- **Subversion:** es un programa de computador para el control y revisión de versiones distribuidas. Su objetivo es ser un sucesor compatible del programa de computador ampliamente usado Concurrent Version System.

- **Mercurial:** es un programa de computador para el control de revisión distribuida. Está pensado para tener gran rendimiento y escalabilidad, completamente distribuido, sin necesidad de un servidor.

1.3.4. Formato de intercambio de datos

Los archivos de texto plano almacenan cadenas de texto para ser consultados más tarde. Así mismo, pueden ser editados para actualizarlos o modificarlos. Si la información almacenada es estructurada, podría ser interpretada por diferentes programas de computador fácilmente.

La información puede ser estructurada para cierto dominio en específico, como las hojas de cálculo, o almacenada usando algún formato independiente del problema. Estos últimos han evolucionado para permitir aún más la codificación de metadatos que describen los atributos estructurales de la información.

Algunos de los formatos de intercambio de datos más famosos son:

- **XML** (Extensible Markup Language): es un lenguaje de marcado de etiquetas para almacenar, transmitir y reconstruir datos arbitrarios. Define un conjunto de reglas para codificar documentos en un formato que puede ser leído por los humanos y las máquinas.
- **JSON** (JavaScript Object Notation): es un formato de archivo estándar y un formato de intercambio de datos que utiliza texto legible por humanos para almacenar y transmitir objetos de datos que consisten en pares de atributos-valores y matrices (u otros valores serializables).
- **YAML** es un lenguaje de serialización de datos legible por humanos. Se usa comúnmente para archivos de configuración y en aplicaciones donde se almacenan o transmiten datos. YAML se dirige a muchas de las mismas aplicaciones de comunicaciones que el lenguaje de marcado extensible (XML), pero tiene una sintaxis mínima que difiere intencionalmente de SGML.

1.3.5. Programas desarrollados

Las tecnologías descritas tienen diferentes propósitos y, naturalmente, se utilizan para diferentes finalidades. Para responder a lo planteado en los objetivos, en esta tesis se desarrollan dos programas de computador: `pymas` y `mas.js`. El primero para analizar

estructuras aporticadas lineales sometidas a cargas estáticas y el segundo para representarlas visualmente. Esta separación de programas tiene ventajas, como el uso de lenguajes de programación adaptados para cada finalidad o la posibilidad de ser utilizados en diferentes ámbitos, sin depender uno del otro. Además, también proporciona a los usuarios la libertad de utilizar cualquier interfaz gráfica de su preferencia, para luego analizar con **pymas**, o al contrario, utilizar **mas.js** para representar de forma gráfica una estructura que será analizada con otro programa o librería de análisis.

El trabajo colaborativo y la mayor libertad de uso de los programas son dos aspectos que permean todo el desarrollo de ambos programas. Por ello su programación se hizo utilizando la herramienta Git, para el control de versiones, y se alojó una copia pública en la plataforma GitHub para acceder, descargar y colaborar en cualquiera de los dos proyectos.

Programa de análisis - pymas

pymas fue programado en Python. Según Lutz (2013), Python es un lenguaje de programación interpretado orientado a objetos cuya filosofía hace énfasis en la legibilidad de su código. Al disponer con una gran variedad de librerías gratuitas; sobre todo las librerías de cálculo científico, y la facilidad de poder ser instalado en una gran variedad de dispositivos, este lenguaje de programación hace de **pymas** una herramienta versátil y fácil de usar para usuarios novatos o expertos. Además, la forma y estilo utilizados permiten un desarrollo colaborativo en cualquier aspecto, como algoritmos específicos de mecánica estructural u optimizaciones para mejorar tiempos de cálculo, entre otros.

El programa o librería **pymas** es de libre uso y cualquier contribución es bienvenida. Se encuentra alojado en el enlace <https://github.com/rvcristiand/pymas>.

Programa de pre y post proceso gráfico - mas.js

mas.js fue desarrollado en el lenguaje de programación JavaScript, con ayuda de la librería *Three.js*. Según Dirksen (2015), *Three.js* es un API (de sus siglas en inglés, *application programming interface*) programada en JavaScript para WebGL que permite crear escenas tridimensionales en el navegador de internet. Ya que JavaScript es el lenguaje de las páginas web y *Three.js* está programada en ese lenguaje, el programa desarrollado **mas.js** puede ejecutarse en el navegador web de cualquier dispositivo, sin

que el usuario tenga que instalar nada adicional en su equipo.

Gracias a la tecnología de los navegadores web (HTML, CSS y JavaScript), **mas.js** se puede ejecutar localmente en diferentes dispositivos, lo que permite ejecutar el mismo código en diferentes plataformas. Así mismo, **mas.js** se beneficia de los últimos desarrollos tecnológicos web, como el renderizado y computación mediante la GPU (de sus siglas en inglés, *graphic processing unit*) o PWA (de sus siglas en inglés, *progressive web apps*). Eso proporciona rapidez en el renderizado de escenas tridimensionales.

En la página de internet <https://rv cristiand.github.io/mas.js> se puede ejecutar directamente la aplicación, sin necesidad de ver el código ni instalar programas. El navegador ejecuta **mas.js**. Además, **mas.js** se conecta con un servidor alojado en la página web <http://rv cristiand.pythonanywhere.com/> que ejecuta **pymas**. De esta manera se conectan ambos programas y cualquier persona podría modelar y analizar estructuras solo con una conexión a internet.

Transferencia de información

Cada uno de los dos programas desarrollados se hizo para un propósito particular, y por lo tanto utiliza tecnologías diferentes. Aunque esto tiene ventajas para el desarrollo y uso de cada programa independientemente, es necesario establecer una forma para transferir la información entre los dos programas. Tradicionalmente, los programas de análisis estructural, de elementos finitos y otros, han utilizado archivos de texto plano con sintaxis particulares para cada programa. En vista de que esta tesis pretende facilitar el uso y desarrollo de herramientas para los ingenieros estructurales, se prefirió el uso de un formato estándar de transferencia de información.

pymas y **mas.js** utilizan un esquema en archivos formato JSON (por sus siglas en inglés, *JavaScript Object Notation*) para almacenar la información de los modelos. Tanto **pymas** como **mas.js** interpretan estos archivos apropiadamente y, además, resulta fácil de leer para los humanos. Este es el esquema que se viene aplicando en la actualidad para realizar trabajos colaborativos en la nube, donde un servidor provee almacenamiento y tiempo de cálculo mientras que los usuarios pueden acceder al servicio desde cualquier dispositivo conectado a internet.

Aunque cada programa, **pymas** o **mas.js**, puede ejecutarse por separado, el formato JSON permite ejecutar de manera cómoda el flujo de información que se muestra en la figura

1-1. Sin embargo, el proceso y transferencia de información puede empezar o terminar en cualquier etapa. Por ejemplo, se puede generar una estructura directamente en **pymas** utilizando Python, y luego exportar la información hacia **mas.js**, o también se puede crear la estructura en **mas.js** y exportar el archivo JSON para ser analizado en **pymas**, y ver algún resultado particular directamente en Python.

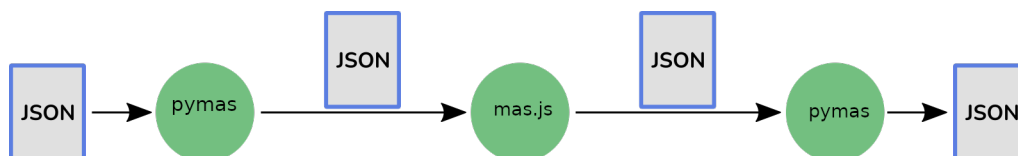


Figura 1-1: Flujo de información completo entre **mas.js** y **pymas**.

El uso del formato JSON, además de permitir un flujo de información rápido, cómodo y fluído entre **pymas** y **mas.js**, también permite transferir fácilmente la información hacia y desde otros programas o APIs.

1.4. Estructura del documento

En el capítulo 2 se presentan los conceptos técnicos fundamentales del análisis estructural mediante el método directo de rigideces. Teniendo en cuenta que el propósito del presente trabajo, allí se expone el método de manera general, sin desarrollar con detalle casos particulares, como los elementos tipo viga o parrilla, entre otros. En consecuencia, se presenta la formulación de elementos prismáticos tridimensionales, se explica una técnica novedosa y útil para trabajar la rotación de elementos en el espacio mediante cuaterniones y la metodología elegida para la solución numérica del sistema de ecuaciones.

En los capítulos 3 y 4 se presentan las consideraciones tomadas para el desarrollo de los programas de computador de análisis (**pymas**) y de representación gráfica (**mas.js**), así como las funciones, variables y algoritmos en cada uno de ellos. En el capítulo 5 se presentan algunas aplicaciones de estos programas a manera de ejemplos y tutoriales, con el propósito de demostrar la versatilidad, extensibilidad y libertad de las herramientas desarrolladas.

Finalmente, en el capítulo 6 se hacen unas reflexiones finales sobre el trabajo realizado y se propone una línea de trabajo y desarrollo para el Departamento de Ingeniería Civil de la Universidad Nacional de Colombia.

2 Análisis matricial de estructuras aporticadas

El *método de rigideces* es el principal método usado en el análisis matricial de estructuras. Una de sus ventajas sobre el método de flexibilidades es su implementación en un programa de computador; una vez el modelo analítico de la estructura ha sido definido, no se requiere tomar más decisiones para llevar a cabo el análisis. En relación con esto difiere del método de flexibilidades, aunque los dos enfoques tienen un desarrollo matemático similar. En el método de flexibilidades las cantidades desconocidas son las reacciones redundantes que deben ser escogidas arbitrariamente, mientras que en el método de rigideces las cantidades desconocidas son los desplazamientos de los nodos de la estructura, los cuales son automáticamente especificados. Es decir que en el método de rigideces el número de incógnitas a resolver es igual al *grado de indeterminación cinemático* de la estructura (Weaver & Gere, 1990).

Si bien la versión formal del método es esclarecedora y bien organizada, involucra grandes matrices dispersas de compatibilidad con términos que no son fáciles de evaluar correctamente. Una mejor metodología basada en este enfoque y aplicando algunas técnicas computacionales se desarrolló hasta convertirse en lo que se conoce como el *método directo de rigideces*.

En este capítulo se desarrolla el método de rigideces escribiendo ecuaciones de equilibrio en los nodos de la estructura en términos de coeficientes de rigidez y desplazamientos desconocidos. Una versión cinemáticamente determinada de la estructura, es decir, con los desplazamientos de los todos nodos restringidos, resulta ser de utilidad para caracterizar tales condiciones de equilibrio como la superposición de diferentes acciones. Una vez discutidos los conceptos básicos, el método es formalizado en un procedimiento para ensamblar la matriz de rigidez de la estructura a partir de la matriz de rigidez de cada elemento.

2.1. Método de rigideces

El método de rigideces para el análisis de estructuras estáticamente indeterminadas considera los desplazamientos como las magnitudes desconocidas, por lo que también se le denomina *método de desplazamientos*. Los desplazamientos desconocidos se obtienen al resolver ecuaciones de equilibrio que contienen coeficientes en la forma de rigideces. Este método es completamente general y puede emplearse para una gran diversidad de estructuras. Sin embargo, está limitado, al igual que el método de flexibilidades, a estructuras cuyos materiales se comporten en forma lineal y elástica, y donde las deformaciones se consideren infinitesimales (Gere, 1986).

Con el fin de describir el método de rigideces, analicemos la viga de la figura 2-1. Esta viga tiene un apoyo empotrado en el nodo A , un apoyo de patín en el nodo B , y está solicitada a una carga uniformemente distribuida de intensidad w . La viga es cinemáticamente indeterminada en primer grado (considerando despreciables las deformaciones axiales), porque el único desplazamiento desconocido es la rotación θ_B del nodo B . La primera fase para analizar la viga es determinar el valor de dicha rotación, después se pueden calcular las fuerzas internas y los desplazamientos a lo largo del elemento.

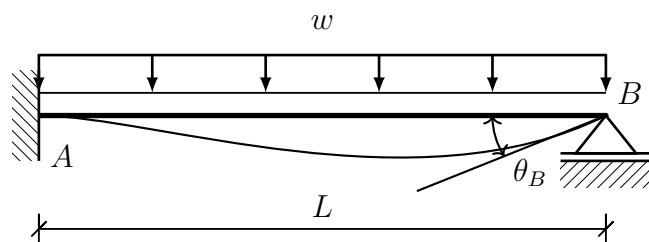


Figura 2-1: Viga con un extremo empotrado y el otro simplemente apoyado, solicitada a una carga uniformemente distribuida.

El análisis de la viga se puede considerar como la superposición de los efectos de las cargas externas y los desplazamientos desconocidos actuando en una estructura alterada, donde todos los desplazamientos desconocidos son iguales a cero. Ya que los desplazamientos desconocidos son las translaciones y rotaciones en los nodos de la estructura, éstos se pueden hacer iguales a cero restringiendo todos los nodos para que no se desplacen. La estructura obtenida al restringir todos los nodos de la estructura original se llama *estructura restringida*. En el caso de la viga de la figura 2-1, la estructura restringida se obtiene impidiendo la rotación θ_B del nodo B . Entonces la estructura restringida

es una viga empotrada en sus extremos, como se muestra en la figura **2-2**.

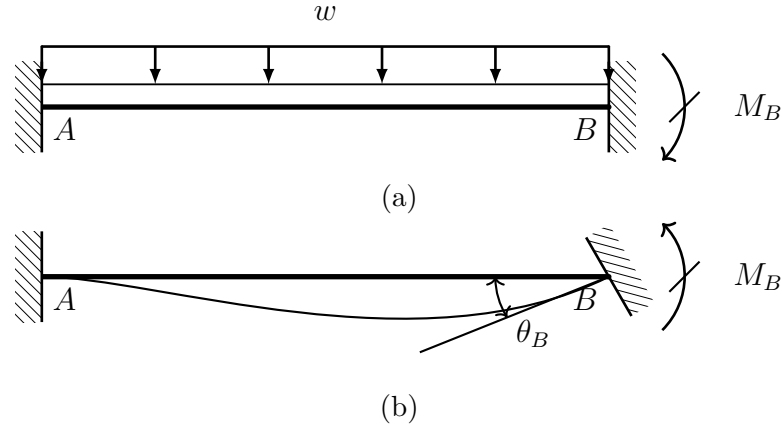


Figura 2-2: Estructura restringida de la viga de la figura **2-1** solicitada a la carga externa y al desplazamiento desconocido.

Cuando la carga uniformemente distribuida w actúa en la viga doblemente empotrada, el momento M_B en el apoyo B es igual a

$$M_B = \frac{wL^2}{12} \quad (2-1)$$

Ya que en la viga original de la figura **2-1** no se restringe la rotación del apoyo B, es necesario considerar la estructura restringida sometida a un momento igual y opuesto al momento M_B . La superposición de las solicitaciones de la figura **2-2** actuando en la estructura restringida producen las solicitaciones en la viga original. Por ende, la rotación θ_B es igual a la rotación desconocida en la viga original.

La relación entre el momento M_B y la rotación θ_B en la viga restringida es

$$M_B = \frac{4EI}{L}\theta_B \quad (2-2)$$

donde EI es la rigidez a flexión de la viga. Igualando las dos expresiones para el momento M_B de las ecuaciones 2-1 y 2-2 se obtiene

$$\frac{wL^2}{12} = \frac{4EI}{L}\theta_B$$

de donde

$$\theta_B = \frac{wL^3}{48EI}$$

De manera que la rotación del nodo B de la viga ha sido determinada.

En el ejemplo anterior es conveniente considerar el efecto de un desplazamiento desconocido unitario actuando en la estructura restringida. También, es más sistemático formular la ecuación para la rotación como una ecuación de superposición y usar una convención de signos consistente para todos los términos, como se presenta a continuación.

En la figura 2-3 se presenta la viga restringida sometida a un desplazamiento unitario. Ya que el momento m_B es la acción que corresponde a la rotación θ_B y es causada por un desplazamiento unitario de esta, mientras los demás desplazamientos son iguales a cero, m_B es conocido como *coeficiente de rigidez* de la estructura restringida.



Figura 2-3: Estructura restringida de la viga de la figura 2-1 solicitada a un desplazamiento desconocido unitario.

El valor del momento m_B es igual a (véase la ecuación 2-2)

$$m_B = \frac{4EI}{L} \quad (2-3)$$

La ecuación de superposición de momentos para el apoyo B se obtiene al igualar el momento M_B de la viga restringida sometida a la carga uniformemente distribuida y el momento m_B multiplicado por θ_B , con el momento actuando en la viga original. Todos los términos de la ecuación de superposición se expresan en la misma convención de signos. De acuerdo con esto, el momento M_B es igual a

$$M_B = -\frac{wL^2}{12}$$

La ecuación de superposición de momentos para el apoyo B es

$$M_B + m_B\theta_B = 0$$

o

$$-\frac{wL^2}{12} + \frac{4EI}{L}\theta_B = 0$$

Solucionando esta ecuación para la rotación θ_B resulta

$$\theta_B = \frac{wL^3}{48EI}$$

el cual es el mismo resultado obtenido anteriormente.

Si una estructura es cinemáticamente indeterminada en más de un grado, debe introducirse un planteamiento de la solución más organizado. Por ejemplo, analicemos la viga de dos luces de la figura 2-4. La viga tiene una rigidez a flexión constante EI y está sujeta a las cargas P_1 , M , P_2 y P_3 . Ya que la viga puede girar en los nodos B y C , la estructura es cinemáticamente indeterminada en grado dos (considerando despreciables las deformaciones axiales). Sean D_1 y D_2 las rotaciones desconocidas en dichos apoyos, respectivamente, y asumamos que son positivas en el sentido contrario al de las manecillas del reloj. Estos desplazamientos desconocidos pueden ser encontrados solucionando ecuaciones de superposición en los nodos B y C , como se presenta a continuación.

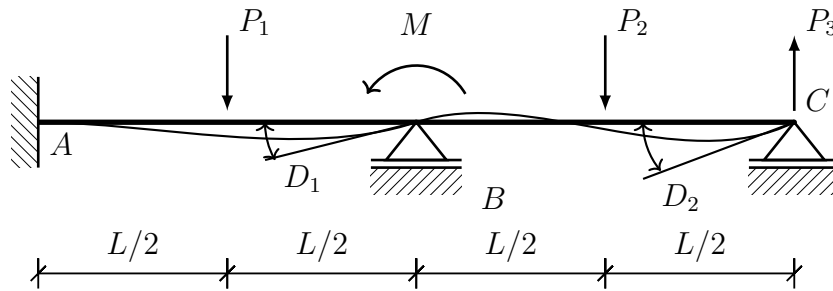


Figura 2-4: Viga de dos luces con un extremo empotrado y el otro simplemente apoyado, solicitada a cargas aplicadas en las luces y sobre los apoyos.

El primer paso para analizar la viga consiste en aplicar apoyos imaginarios en los nodos para restringir los desplazamientos. La estructura restringida son las dos vigas doblemente empotradas de la figura 2-5. Se asume que en la estructura restringida actúan todas las cargas a excepción de aquellas que actúan en los grados de libertad desconocidos, es decir, sólo las cargas P_1 , P_2 y P_3 . Todas las otras cargas que correspondan a desplazamientos desconocidos, como el momento M en este ejemplo, serán tomadas en cuenta más adelante. Los momentos A_{DL1} y A_{DL2} son las reacciones de los apoyos de

la estructura restringida, correspondientes a los desplazamientos D_1 y D_2 , respectivamente, debidas a las cargas actuando en la estructura. Por ejemplo, la reacción A_{DL1} es la suma de la reacción en B debido a la carga P_1 actuando en el elemento AB y la reacción en B debido a la carga P_2 actuando en el elemento BC .

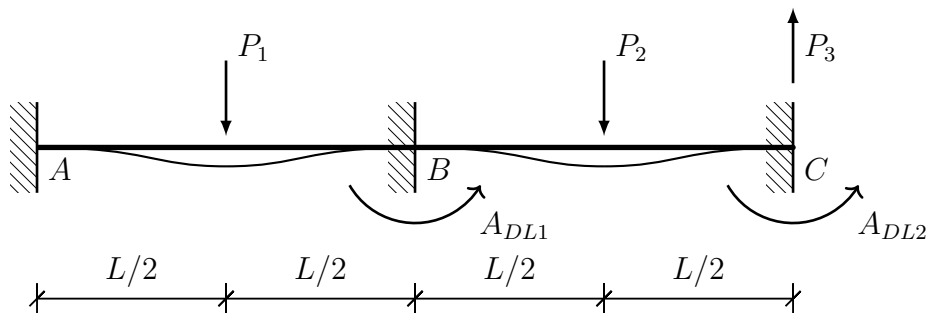


Figura 2-5: Estructura restringida de la viga de la figura 2-4 solicitada a las cargas externas.

Para generar los coeficientes de rigidez en los nodos B y C , se introducen desplazamientos unitarios D_1 y D_2 por separado en la estructura restringida. Un desplazamiento unitario D_1 consiste en una rotación unitaria del nodo B , tal como se presenta en la figura 2-6a. El desplazamiento D_2 permanece igual a cero en esta viga. Por lo tanto, las reacciones correspondientes a los desplazamientos D_1 y D_2 son los coeficientes de rigidez S_{11} y S_{21} , respectivamente. Estas rigideces consisten en los momentos generados por las reacciones de los apoyos B y C de la estructura, respectivamente. El caso en que D_2 es igual a 1 mientras que D_1 es igual a cero se presenta en la figura 2-6b. En esta figura la rigidez S_{12} es la reacción que corresponde al desplazamiento D_1 mientras que la rigidez S_{22} es la reacción correspondiente to D_2 .

Ahora se pueden plantear dos ecuaciones de superposición en los apoyos B y C para expresar los momentos que actúan en la estructura original (figura 2-4). Sean A_{D1} y A_{D2} las acciones en la estructura original correspondiente a los desplazamientos D_1 y D_2 respectivamente. Estas acciones serán iguales a cero a menos que una carga concentrada externa actúe en alguno de los grados de libertad desconocidos. En la figura 2-5, la acción A_{D1} es igual al momento M mientras que la acción A_{D2} es igual a cero. Las ecuaciones de superposición relacionan las cargas aplicadas en los grados de libertad desconocidos de la estructura original con las reacciones de los correspondientes grados de libertad en la estructura restringida, debidas a las cargas externas aplicadas en los elementos aporticados y a desplazamientos unitarios.

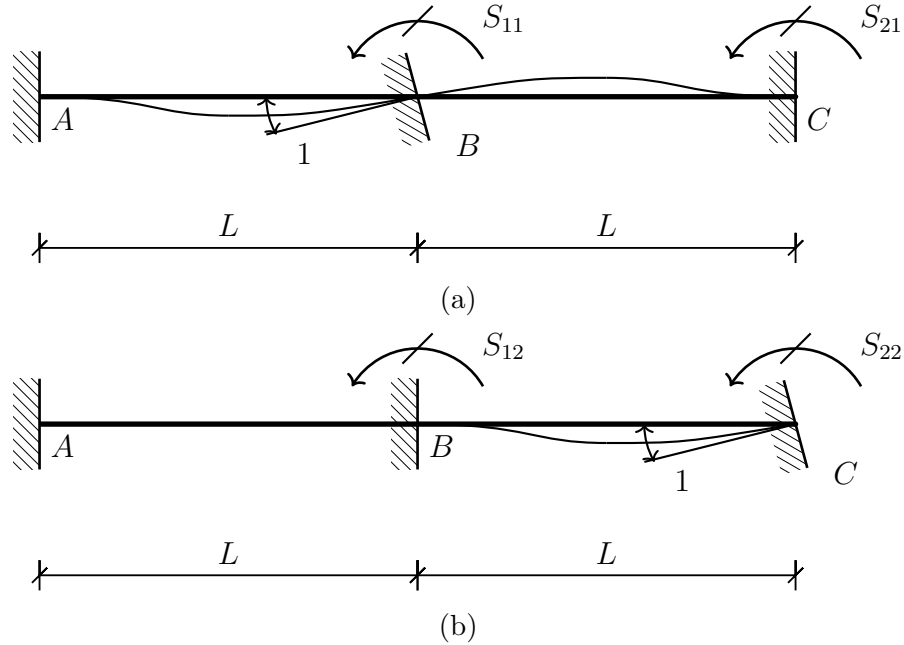


Figura 2-6: Desplazamientos unitarios D_1 y D_2 introducidos separadamente en la estructura restringida.

Las ecuaciones de superposición en los apoyos B y C de la viga de la figura 2-4 son

$$\begin{aligned} A_{D1} &= A_{DL1} + S_{11}D_1 + S_{12}D_2 \\ A_{D2} &= A_{DL2} + S_{21}D_1 + S_{22}D_2 \end{aligned} \quad (2-4)$$

La convención de signos usada en estas ecuaciones para los momentos es la misma que la usada para los desplazamientos, positivos en el sentido contrario al de las manecillas del reloj.

Reescribiendo la ecuación 2-4 en forma matricial se obtiene

$$\mathbf{A}_D = \mathbf{A}_{DL} + \mathbf{S}\mathbf{D} \quad (2-5)$$

En esta ecuación el vector $\mathbf{A}_D$ representa las acciones externas correspondientes a los desplazamientos desconocidos $\mathbf{D}$, el vector $\mathbf{A}_{DL}$ representa las reacciones en la estructura restringida, debidas a las cargas externas (a excepción de aquellas actuando en los grados de libertad desconocidos), correspondientes a los desplazamientos desconocidos, y $\mathbf{S}$ es la matriz de rigidez correspondiente a los desplazamientos desconocidos.

Solucionando la ecuación 2-5 para los desplazamientos desconocidos se obtiene:

$$\mathbf{D} = \mathbf{S}^{-1}(\mathbf{A}_D - \mathbf{A}_{DL}) \quad (2-6)$$

Esta ecuación representa la solución de los desplazamientos en forma matricial, ya que los términos $\mathbf{A}_D$, $\mathbf{A}_{DL}$ y $\mathbf{S}$ son conocidos o pueden ser calculados de la estructura restringida. El término $-\mathbf{A}_{DL}$ representa el *vector de cargas equivalentes en los nodos*. Tales cargas se definen como la acción contraria de las reacciones correspondiente a los desplazamientos desconocidos, causadas por las cargas aplicadas a los miembros de la estructura restringida. Cuando estas cargas equivalentes en los nodos se suman con las cargas que actúan en los nodos $\mathbf{A}_D$, el resultado se conoce como el *vector de cargas combinadas en los nodos*.

2.2. Matriz de rigidez de elementos los aporticados

En la figura 3-6 se presenta un elemento aporticado i con sus nodos j y k empotrados. El sistema de coordenadas local del elemento tiene como origen el nodo j , el eje x coincide con el eje centroidal del elemento y es positivo en el sentido del nodo j al nodo k .

Los ejes y y z son los ejes principales del elemento de manera que los planos xy y zx son los planos principales de flexión. Se asume que el centro de cortante y el centroide del elemento coinciden de tal forma que la flexión y la torsión se presentan una independiente de la otra.

Los grados de libertad se numeran del 1 al 12, empezando por las translaciones y las rotaciones del nodo j , tomados en orden x, y, z respectivamente.

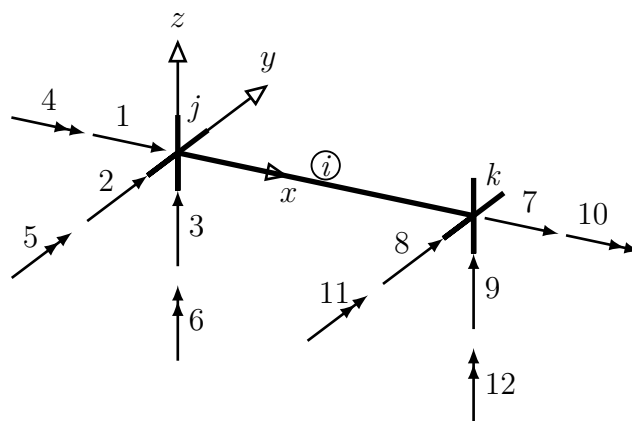


Figura 2-7: Elemento aporticado con sus grados de libertad numerados y su sistema de coordenadas local.

Los coeficientes de rigidez de los elementos aporticados pueden ser organizados dentro de *matrices de rigidez*. Los términos en dichas matrices están definidos como las reacciones en los extremos del elemento debido a desplazamientos unitarios de un extremo relativo a los demás. En la ecuación 2-7 se presenta la matriz de rigidez $\mathbf{S}_{Mi}$ del elemento aporticado.

$$\mathbf{S}_{Mi} = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \end{matrix} & \left[\begin{array}{cccccccccccc} \frac{EA}{L} & 0 & 0 & 0 & 0 & 0 & -\frac{EA}{L} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{12EI_z}{L^3} & 0 & 0 & 0 & \frac{6EI_z}{L^2} & 0 & -\frac{12EI_z}{L^3} & 0 & 0 & 0 & \frac{6EI_z}{L^2} \\ 0 & 0 & \frac{12EI_y}{L^3} & 0 & -\frac{6EI_y}{L^2} & 0 & 0 & 0 & -\frac{12EI_y}{L^3} & 0 & -\frac{6EI_y}{L^2} & 0 \\ 0 & 0 & 0 & \frac{GJ}{L} & 0 & 0 & 0 & 0 & 0 & -\frac{GJ}{L} & 0 & 0 \\ 0 & 0 & -\frac{6EI_y}{L^2} & 0 & \frac{4EI_y}{L} & 0 & 0 & 0 & \frac{6EI_y}{L^2} & 0 & \frac{2EI_y}{L} & 0 \\ 0 & \frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{4EI_z}{L} & 0 & -\frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{2EI_z}{L} \\ -\frac{EA}{L} & 0 & 0 & 0 & 0 & 0 & \frac{EA}{L} & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{12EI_z}{L^3} & 0 & 0 & 0 & \frac{12EI_z}{L^3} & 0 & 0 & 0 & 0 & -\frac{6EI_z}{L^2} & 0 \\ 0 & 0 & \frac{12EI_y}{L^3} & 0 & \frac{6EI_y}{L^2} & 0 & 0 & 0 & \frac{12EI_y}{L^3} & 0 & 0 & -\frac{6EI_y}{L^2} \\ 0 & 0 & 0 & -\frac{GJ}{L} & 0 & 0 & 0 & 0 & 0 & \frac{GJ}{L} & 0 & 0 \\ 0 & 0 & \frac{2EI_y}{L} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{4EI_y}{L} & 0 \\ 0 & \frac{2EI_z}{L} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{4EI_z}{L} \end{array} \right] \end{matrix} \quad (2-7)$$

donde E es el módulo de Young, G es el módulo de elasticidad a cortante, A es el área de la sección trasversal, J es la constante de torsión, y I_y y I_z son los momentos de inercia con respecto a los ejes y y z , respectivamente.

2.3. Formalización del método de rigideces

La matriz de rigidez $\mathbf{S}_{Mi}$ de un elemento i relaciona las acciones en los extremos de dicho elemento $\mathbf{A}_{Mi}$ con los correspondientes desplazamientos $\mathbf{D}_{Mi}$ de la manera

$$\mathbf{A}_{Mi} = \mathbf{S}_{Mi} \mathbf{D}_{Mi} \quad (2-8)$$

Si se escribe consecutivamente la ecuación 2-8 para todos los m elementos que componen la estructura, se obtiene la expresión

$$\left\{ \begin{matrix} \mathbf{A}_{M1} \\ \mathbf{A}_{M2} \\ \mathbf{A}_{M3} \\ \dots \\ \mathbf{A}_{Mi} \\ \dots \\ \mathbf{A}_{Mm} \end{matrix} \right\} = \left[\begin{array}{ccccccc} \mathbf{S}_{M1} & 0 & 0 & \dots & 0 & \dots & 0 \\ 0 & \mathbf{S}_{M2} & 0 & \dots & 0 & \dots & 0 \\ 0 & 0 & \mathbf{S}_{M3} & \dots & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & \mathbf{S}_{Mi} & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 0 & \dots & \mathbf{S}_{Mi} \end{array} \right] \left\{ \begin{matrix} \mathbf{D}_{M1} \\ \mathbf{D}_{M2} \\ \mathbf{D}_{M3} \\ \dots \\ \mathbf{D}_{Mi} \\ \dots \\ \mathbf{D}_{Mm} \end{matrix} \right\} \rightarrow \mathbf{A}_M = \mathbf{S}_M \mathbf{D}_M \quad (2-9)$$

La ecuación anterior relaciona las acciones en los extremos de cada elemento con sus desplazamientos, sin tener en cuenta el efecto de unos elementos sobre otros en la estructura. De esa manera, la ecuación podría representar casos en los que los desplazamientos de los extremos de dos elementos que llegan al mismo punto no son iguales o incompatibles. A continuación se presenta una formalización del proceso matemático que se puede ejecutar para representar un sistema con desplazamientos compatibles.

Primero, se introduce un vector $\mathbf{D}_J$ que contiene los desplazamientos en los nodos de toda la estructura. Más adelante, $\mathbf{D}_J$ será el resultado directo de la solución del método. La relación entre los desplazamientos de los nodos (en $\mathbf{D}_J$) y los desplazamientos en los extremos de los elementos (en $\mathbf{D}_M$) se puede hacer matricialmente como

$$\mathbf{D}_M = \mathbf{C}_{MJ}\mathbf{D}_J \quad (2-10)$$

El término $\mathbf{C}_{MJ}$ en esta ecuación es una matriz de transformación de los desplazamientos. En la práctica, dicha matriz permite extraer los desplazamientos de los extremos de cada elemento $\mathbf{D}_M$ a partir de los desplazamientos de los nodos de toda la estructura $\mathbf{D}_J$. Los componentes de $\mathbf{C}_{MJ}$ son determinados a partir de las condiciones de compatibilidad de desplazamientos que existen entre los elementos de la estructura, por lo que se conoce como *matriz de compatibilidad*.

Luego se impone una pequeña cantidad de desplazamiento virtual arbitrario en todos los nodos de la estructura. Los desplazamientos virtuales de los extremos de los elementos $\delta\mathbf{D}_M$ se pueden escribir en términos de $\delta\mathbf{D}_J$ mediante la misma matriz de compatibilidad anterior, como

$$\delta\mathbf{D}_M = \mathbf{C}_{MJ}\delta\mathbf{D}_J \quad (2-11)$$

Ahora, el trabajo virtual externo δW generado por las acciones reales $\mathbf{A}_J$ actuando en la estructura al desplazarse los desplazamientos virtuales en los nodos $\delta\mathbf{D}_J$ está dado por

$$\delta W = \mathbf{A}_J^T \delta\mathbf{D}_J \quad (2-12)$$

Por su parte, el trabajo virtual interno δU proveniente de las acciones en los extremos de los elementos $\mathbf{A}_M$ y sus respectivos desplazamientos virtuales $\delta\mathbf{D}_M$ se puede escribir como

$$\delta U = \mathbf{A}_M^T \delta\mathbf{D}_M \quad (2-13)$$

$$\mathbf{A}_J^T \delta \mathbf{D}_J = \mathbf{A}_M^T \delta \mathbf{D}_M \quad (2-14)$$
$$\mathbf{A}_J^T \delta \mathbf{D}_J = \mathbf{D}_J^T \mathbf{C}_{MJ}^T \mathbf{S}_M \mathbf{C}_{MJ} \delta \mathbf{D}_J \quad (2-15)$$
$$\mathbf{A}_J = \mathbf{C}_{MJ}^T \mathbf{S}_M \mathbf{C}_{MJ} \mathbf{D}_J = \mathbf{S}_J \mathbf{D}_J \quad \rightarrow \quad \mathbf{S}_J = \mathbf{C}_{MJ}^T \mathbf{S}_M \mathbf{C}_{MJ} \quad (2-16)$$

2.4. Método directo de rigideces

El ensamblaje de la matriz de rigidez de la estructura $\mathbf{S}_J$ se puede escribir como:

$$S_J = \bigwedge_{i=1}^m S_{MSi} \quad (2-17)$$

En esta ecuación, $\mathbf{S}_{MSi}$ representa la matriz de rigidez del i -ésimo elemento con las acciones en los extremos y los desplazamientos tomadas con respecto al sistema de

coordenadas de la estructura, lo cual se indica con la letra S del subíndice MSi .

De manera similar, el vector de cargas elementales equivalentes en los nodos $\mathbf{A}_E$ (conocido también como las fuerzas de empotramiento) puede ser ensamblado a partir de la contribución de cada elemento como:

$$\mathbf{A}_E = - \bigtriangleup_{i=1}^m \mathbf{A}_{MSi} \quad (2-18)$$

en donde $\mathbf{A}_{MSi}$ es el vector de acciones fijas en los extremos del i -ésimo elemento con respecto al sistema de coordenadas de la estructura.

2.4.1. Matriz de transformación de rotaciones

La matriz de rigidez $\mathbf{S}_{MSi}$ en coordenadas globales se puede escribir en función de la matriz de rigidez $\mathbf{S}_{Mi}$ en coordenadas locales del elemento i como:

$$\mathbf{S}_{MSi} = \mathbf{R}_{Ti} \mathbf{S}_{Mi} \mathbf{R}_{Ti}^T \quad (2-19)$$

donde $\mathbf{R}_{Ti}$ es la *matriz de transformación de rotación* del elemento i .

La matriz $\mathbf{R}_T$ para un elemento aporticado orientado arbitrariamente en el espacio contiene cuatro matrices de rotación en su interior, ya que cada elemento representa un vector de desplazamientos y uno de rotación o uno de fuerzas y uno de momentos para cada nodo. Esto es, cuatro vectores para cada elemento. Dicha matriz se puede escribir de la siguiente forma

$$\mathbf{R}_T = \begin{bmatrix} \mathbf{R} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{R} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{R} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{R} \end{bmatrix} \quad (2-20)$$

donde $\mathbf{R}$ es la matriz de rotación de los ejes locales del elemento a los ejes globales de la estructura para un único vector.

La matriz de rotación $\mathbf{R}$ se ha calculado tradicionalmente mediante cosenos directores o ángulos de Euler. A pesar de que ese método es conocido y ensañado en los cursos de álgebra lineal, presenta dificultades para representar transformaciones en el espacio. Por ejemplo, al intentar rotar un elemento alrededor de su propio eje longitudinal. Según

Dunn (2002), la posición de un vector después de una rotación θ alrededor de un eje $\mathbf{n}$ que pasa por el origen se puede describir mediante un *cuaternión* como:

$$\mathbf{q} = [\cos(\theta/2) \quad \sin(\theta/2)\mathbf{n}] \quad (2-21)$$

el cual se compone de una parte escalar $\cos(\theta/2)$ y una parte vectorial $\sin(\theta/2)\mathbf{n}$. Esta es una herramienta matemática conveniente y compacta para calcular la matriz de rotación de los elementos estructurales. La matriz $\mathbf{R}$ se puede obtener a partir de un cuaternión de la siguiente manera

$$\mathbf{R} = \begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy + 2wz & 2xz - 2wy \\ 2xy - 2wz & 1 - 2x^2 - 2z^2 & 2yz + 2wx \\ 2xz + 2wy & 2yz - 2wx & 1 - 2x^2 - 2y^2 \end{bmatrix} \quad (2-22)$$

donde w es la parte escalar $\cos(\theta/2)$ y x , y y z la parte vectorial del cuaternión $\sin(\theta/2)\mathbf{n}$.

2.4.2. Solucion del sistema de ecuaciones

El sistema de ecuaciones que representa la relación entre las acciones externas y los desplazamientos de los nodos de una estructura en coordenadas globales está dado por

$$\mathbf{S}_J \mathbf{D}_J = \mathbf{A}_J \quad (2-23)$$

en donde el vector $\mathbf{D}_J$ contiene algunas componentes conocidas iguales o diferentes de cero, establecidas por las condiciones de apoyo.

Una posible manera de encontrar los desplazamientos desconocidos es mediante la re-organización del sistema, intercambiando filas y columnas. Sin embargo, una manera más eficiente computacionalmente de solucionar el sistema consiste en mantener las ecuaciones en su orden original, modificando temporalmente las matrices de rigidez y los vectores de carga. Esto implica convertir las ecuaciones de las reacciones en los apoyos en simples ecuaciones triviales. Así, todo el conjunto de ecuaciones puede ser solucionado simultáneamente.

Por ejemplo, en la ecuación 2-24 se presenta las ecuaciones de equilibrio para una estructura con cuatro grados de libertad.

$$\begin{bmatrix} S_{J11} & S_{J12} & S_{J13} & S_{J14} \\ S_{J21} & S_{J22} & S_{J23} & S_{J24} \\ S_{J31} & S_{J32} & S_{J33} & S_{J34} \\ S_{J41} & S_{J42} & S_{J43} & S_{J44} \end{bmatrix} \begin{Bmatrix} D_{J1} \\ D_{J2} \\ D_{J3} \\ D_{J4} \end{Bmatrix} = \begin{Bmatrix} A_{J1} \\ A_{J2} \\ A_{J3} \\ A_{J4} \end{Bmatrix} \quad (2-24)$$

Suponemos que el tercer desplazamiento es conocido y diferente de cero. Entonces, los términos relacionados con D_{J3} pueden ser restados de ambos lados de la ecuación 2-24, y la tercera ecuación reemplazada por la expresión trivial $D_{J3} = D_{J3}$, para obtener

$$\begin{bmatrix} S_{J11} & S_{J12} & 0 & S_{J14} \\ S_{J21} & S_{J22} & 0 & S_{J24} \\ 0 & 0 & 1 & 0 \\ S_{J41} & S_{J42} & 0 & S_{J44} \end{bmatrix} \begin{Bmatrix} D_{J1} \\ D_{J2} \\ D_{J3} \\ D_{J4} \end{Bmatrix} = \begin{Bmatrix} A_{J1} - S_{J13}D_{J3} \\ A_{J2} - S_{J23}D_{J3} \\ D_{J3} \\ A_{J4} - S_{J43}D_{J3} \end{Bmatrix} \quad (2-25)$$

Ahora estas ecuaciones pueden ser resueltas para los cuatro desplazamientos, incluyendo el desplazamiento D_{J3} .

Los términos negativos del lado derecho de la ecuación 2-25 representan las cargas equivalentes en los nodos debido al desplazamiento conocido D_{J3} . Por supuesto, si D_{J3} es igual a cero, dichas cargas equivalentes son iguales a cero.

3 Programa de análisis estructural **pymas**

En la introducción se explicó que en este trabajo se pretende que los productos desarrollados cumplan ciertos propósitos en cuanto a funcionamiento, portabilidad o códigos de conducta en la programación, entre otros, como se estableció en los objetivos y la introducción. Para ello, la principal decisión consistió en separar los procesos de análisis estructural de la representación gráfica del modelo. Esto permite establecer un flujo de desarrollo adecuado para determinar el lenguaje de programación más razonable, apropiado y eficiente para cada etapa, mantener un código extensible y ordenado, permitir a los usuarios utilizar de manera cómoda cada programa según sus funcionalidades, etc.

A partir de lo anterior, el desarrollo del análisis estructural por el método de la rigidez toma diferentes caminos y se decanta en una herramienta computacional orientada principalmente a tres tipos de usuario diferentes. Primero, para ingenieros civiles recientemente graduados o estudiantes terminando sus estudios, con la cual puedan realizar cálculos sencillos sin sacrificar mucho tiempo, dinero u otros recursos. Segundo, para investigadores y estudiantes de posgrados en estructuras, quienes requieren la suficiente versatilidad para involucrar otras librerías, algoritmos o programas y extender el análisis hacia la aplicabilidad o proceso que cada uno requiera. Y tercero, para profesores y estudiantes de cursos básicos de análisis estructural, quienes necesitan ver, estudiar, entender, modificar y explorar el método de la rigidez en el análisis de estructuras. Como consecuencia de esto, se desarrolla una herramienta computacional, que en adelante llamaremos programa o librería, en el lenguaje Python, con el paradigma de programación orientado a objetos, llamado **pymas**. A continuación, se describen aspectos generales, arquitectura, código implementado, clases (objetos), atributos (variables) y métodos (funciones).

3.1. Generalidades de pymas

`pymas` es un programa de computador desarrollado en Python para realizar el análisis lineal de estructuras aporticadas tridimensionales, sometidas a cargas estáticas. En la página web de GitHub <https://github.com/rvcristiand/pymas> se encuentra alojada una copia actualizada del programa.

3.1.1. Arquitectura del programa

En la figura 5-1 se presenta la estructura de los archivos del programa.

Los archivos que se encuentran dentro de la carpeta principal `pymas` son informativos,

```
pymas/
├── LICENSE
├── README.md
├── examples/
│   ├── 01_simplest_beam/
│   │   ├── makepath.py
│   │   ├── simplest_beam.json
│   │   └── simplest_beam.py
│   ├── 02_space_truss/
│   │   ├── makepath.py
│   │   ├── space_truss.json
│   │   └── space_truss.py
│   └── 03_space_frame/
│       ├── makepath.py
│       ├── space_frame.json
│       └── space_frame.py
└── src/
    ├── pymas/
    │   ├── __init__.py
    │   ├── classtools.py
    │   ├── core.py
    │   └── primitives.py
```

Figura 3-1: Estructura de los archivos del programa `pymas`.

que ayudan a los usuarios a tener una primera mirada del programa, así como su alcance y limitaciones. A continuación se presenta una explicación breve de cada uno de ellos:

- El archivo `LICENSE` contiene información sobre la licencia del programa, la cual establece lo que pueden realizar los usuarios con el código descargado. En virtud de proporcionar la mayor libertad de uso, lectura, modificación y distribución del programa, `pymas` contiene una licencia MIT. En la figura **3-2** se presenta la licencia de `pymas`.

The MIT License (MIT)

Copyright (c) 2019 pymas

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Figura 3-2: Licencia del programa `pymas`.

Es importante notar que la licencia elegida establece por escrito la mayor libertad a los usuarios y desarrolladores de `pymas`. Esto pretende que el programa se extienda y pueda ser utilizado sin límites por cualquier persona en el mundo. Como se puede leer en la licencia MIT, la única restricción es mantener este mismo tipo de licencia en cualquier copia del programa que sea distribuida.

- El archivo `README.md` contiene información general sobre el programa, como la manera de descargarlo e instalarlo. Adicionalmente se presenta un ejemplo sencillo para resolver una viga simplemente apoyada con carga uniformemente distribuida, de manera que los potenciales usuarios puedan tener una primera idea del funcionamiento del programa y su sintaxis. Este archivo es el primer lugar que debe visitar cualquier usuario nuevo del programa para comenzar a utilizarlo.

Adicionalmente a los archivos descritos, se encuentran dos carpetas:

- La carpeta `examples/` contiene algunos ejemplos de ejercicios resueltos con `pymas`. Más adelante se presenta un capítulo que muestra y explica los ejemplos

incluidos en el programa. La colección de ejemplos cubre diferentes ejercicios con la intención de mostrar el potencial del programa.

- En la carpeta **src/** se almacena la totalidad de algoritmos y funciones que componen el programa o librería **pymas**. Dentro de esta carpeta, el archivo `__init__.py` contiene una única sentencia, cuya función es activar automáticamente el uso de la clase **Structure**, una vez sea importada la librería en algún proyecto de Python. Los otros archivos de esta carpeta contienen los procedimientos específicos del método de la rigidez, los cuales se explican más adelante.

3.1.2. Paradigma de programación

Los paradigmas de programación son una clasificación de los códigos de programación, que responde a la forma y características del programa desarrollado, como es explico en la introducción. **pymas** se programó con el paradigma de programación orientado a objetos y, por lo tanto, cualquier elemento, característica, aspecto o parte del programa estará representado con un objeto de diferente tipo, al cual se le atribuyen unos atributos o propiedades y unos métodos o procedimientos.

En principio, los objetos son una representación virtual de un objeto real. Por ejemplo, un perro labrador, café, de 2 años de edad que puede ladrar y comer se puede ver como un objeto de la clase *Perro*. La figura **3-3** muestra una representación gráfica de dicha clase. Allí se puede ver que cualquier objeto de la clase *Perro* puede tener tres atributos: nombre, color y edad, y además puede realizar dos acciones: ladrar y comer. Al definir esta clase **Perro** se pueden instanciar muchos objetos del mismo tipo, pero cada uno con sus atributos propios.

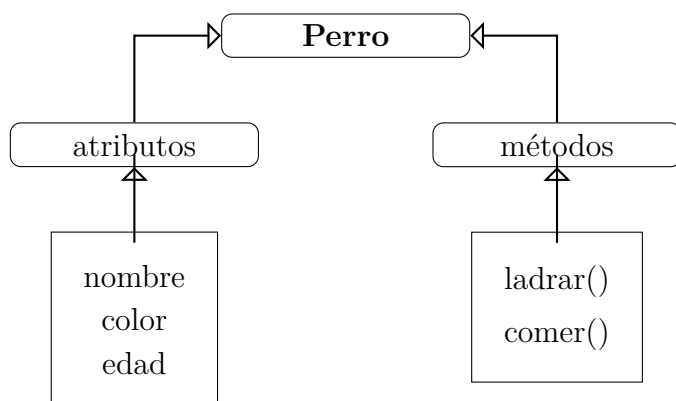


Figura 3-3: Ejemplo de métodos y atributos de la clase hipotética **Perro**.

Además de representar las características y acciones de objetos físicos, también pueden existir clases para la creación de objetos un poco más abstractos. Por ejemplo, las cuentas bancarias pueden representarse como objetos con atributos (propiedades) como dueño o saldo y con métodos (acciones) como consignar o retirar.

Como consecuencia del paradigma utilizado, **pymas** define clases para cada aspecto del modelo estructural. Por ejemplo, existe una clase **Material** para los material, una clase **Frame** para los elementos estructurales, o hasta la clase general **Structure** que contiene todos los objetos que describen la estructura.

Las clases que describen una estructura se encuentran en el archivo `primitives.py`, mientras que la clase **Structure**, que contiene los objetos que describen una estructura en particular y con la cual se puede solucionar el problema estructura, se encuentra en el archivo `core.py`.

3.1.3. Estandarización de sintáxis

El programa **pymas** se encuentra escrito en el idioma inglés en su totalidad. Aunque esto pueda generar alguna dificultad en usuarios hispanoparlantes, es una ventaja y una necesidad para permitir e incentivar el desarrollo colaborativo por personas de otros países. El código desarrollado cumple con estándares de programación en cuando a claridad, orden y legibilidad, entre otros. Y además, se establecieron algunas reglas que facilitan su entendimiento y trabajabilidad, aunque en algunos casos esto pueda resultar en procedimientos que no han sido optimizados para la velocidad de cálculo.

Características de las clases

- Los nombres de todas las clases siguen la convención *CamelCase*. Esto implica que dichos nombres empiezan siempre con letra mayúscula y en los casos que alguna clase tenga un nombre compuesto por más de una palabra, cada una de ellas deberá empezar también por mayúscula. Además, se intentan evitar nombres confusos, en favor de nombres descriptivos: **Structure**, **Material**, **Section**, **DistributedLoad**, etc.
- Todas las clases contienen un bloque de comentario después de su definición. Allí se presenta una descripción corta de la clase, los nombres de los atributos de la clase con su tipo y una descripción de una línea, y los nombres de todos los

métodos permitidos para la clase con una descripción de una línea. El bloque de comentarios siempre se presenta con el formato del algoritmo 3.1.

Algoritmo 3.1: Documentación de las clases en Python

```
class MiClase:
    """
    Descripción de la clase.

    Attributes
    -----
    atributo1 : tipo
        Descripción corta del atributo 1.
    atributo2 : tipo
        Descripción corta del atributo 2.
    atributo3 : tipo
        Descripción corta del atributo 3.

    Methods
    -----
    metodo1()
        Descripción del método 1.
    metodo2()
        Descripción del método 2.
    """
    pass
```

Características de los atributos

- Los nombres de los atributos se establecen de la manera más comprensible, teniendo en cuenta los nombres que se utilizan tradicionalmente para resolver problemas de ingeniería estructural. En ese sentido todos los desplazamientos comienzan con la letra **u** si se refieren a traslación y la letra **r** para rotación, seguida de la dirección **x**, **y**, o **z**, lo cual resulta en los nombres de variables **ux**, **uy** y **uz** para las traslaciones en dirección *x*, *y* y *z*, y **rx**, **ry** y **rz** para rotaciones alrededor de los ejes *x*, *y* y *z*.
- Los nombres de los atributos que determinan las variables de las clases se escriben con la convención *snake_case*, es decir, todas las letras en minúsculas y las palabras separadas por una barra al piso, por ejemplo **load_patern**, **ux**, **fx**, **fx_j**, etc. Únicamente se rompe esta regla cuando el nombre de la variable sea una letra o sigla comúnmente utilizada en ingeniería estructural o mecánica de materiales, como **E** o **G** para representar al módulo de Young y módulo de cortante, respectivamente.

Características de los métodos

- Todos los nombres de los métodos en las clases siguen la convención *snake_case* sin excepción. Todas las letras en minúsculas y las palabras separadas por barras al piso.

- Todos los métodos utilizados para crear o agregar objetos comienzan con `add_`, seguido del nombre de la clase apropiada. Por ejemplo `add_section()`, `add_frame()`, `add_load_patern()`, etc.
- Los métodos utilizados para establecer alguna característica específica a un objeto comienzan con `set_`. Por ejemplo, `set_indexes()`.
- Los métodos con los cuales se solicita realizar un cálculo y tomar un resultado comienzan con `get_` seguido de una palabra descriptiva del procedimiento a realizar. Algunos de estos métodos son `get_stiffness_matrix()`, `get_interntal_forces()` o `get_length()`, entre otros.
- La clase **Structure** contiene tres métodos que no comienzan con ninguno de los tres prefijos explicados anteriormente. Estas son `solve_load_pattern()` para solucionar un caso de carga específico, `solve()` para resolver la estructura y `export()` para exportar una archivo con la información del modelo en formato JSON.

3.2. Clase **Structure** (primera parte)

El usuario de esta librería puede generar (o instanciar) objetos de la clase **Structure**, definida en el archivo `core.py`, para describir y analizar linealmente modelos de estructuras aporticadas tridimensionales sometidas a fuerzas estáticas. En la figura 3-4 se presentan los principales métodos y atributos de esta clase.

El constructor de la clase **Structure** recibe seis argumentos de entrada opcionales, uno para cada grado de libertad (tres translaciones y tres rotaciones), los cuales tienen `None` como valor por defecto. Cuando el usuario crea un objeto de esta clase debe indicar qué grados de libertad quiere tener en cuenta para analizar el modelo.

En el algoritmo 3.2 se presenta el constructor de la clase **Structure**. El constructor de la clase le asigna a los atributos `ux`, `uy`, `uz`, `rx`, `ry` y `rz` los respectivos valores de los argumentos de entrada. A los demás atribututos les asigna un diccionario vacío.

Algoritmo 3.2: Constructor de la clase **Structure**.

```
def __init__(self, ux=None, uy=None, uz=None, rx=None, ry=None, rz=None):
    """
    Instantiate a Structure object.
```

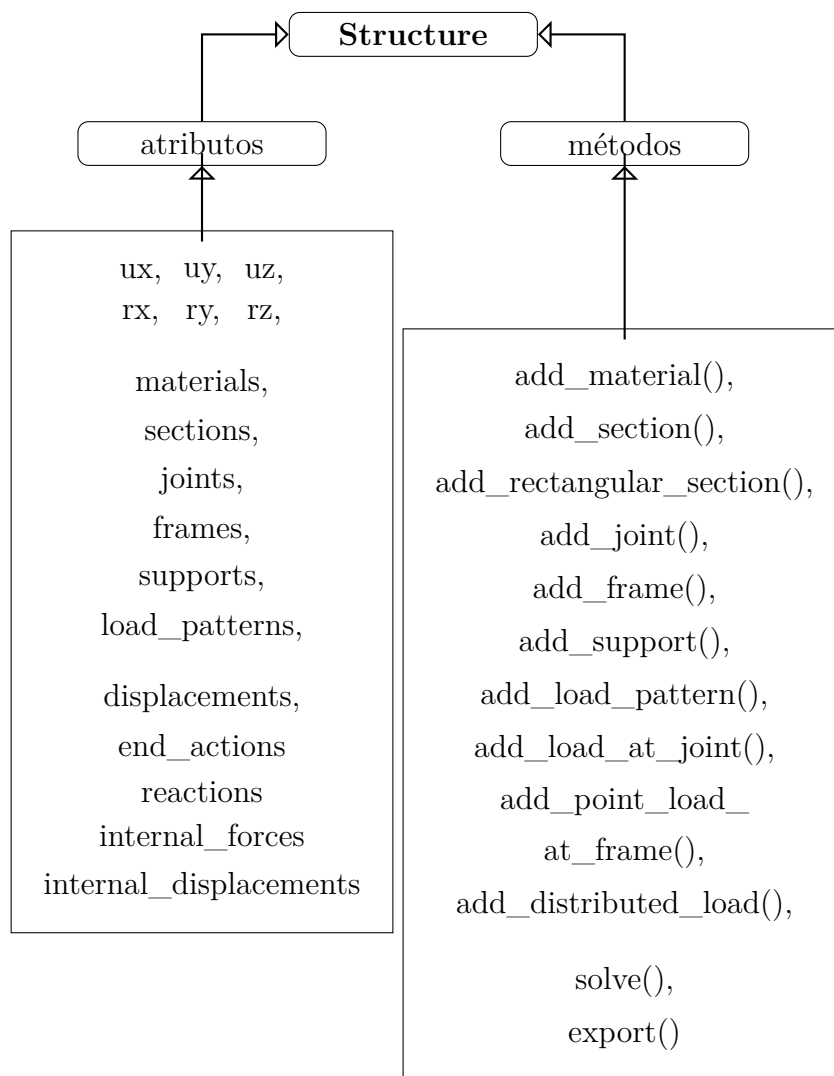


Figura 3-4: Métodos y atributos de la clase **Structure**.

Parameters

ux : bool, optional
Flag analyze translation along x-axis.

uy : bool, optional
Flag analyze translation along y-axis.

uz : bool, optional
Flag analyze translation along z-axis.

rx : bool, optional
Flag analyze rotation around x-axis.

ry : bool, optional

```

        Flag analyze rotation around y-axis.
        rz : bool, optional
        Flag analyze rotation around z-axis.
        """
        self._active_joint_displacements = np.array([])
        self._indexes = {}
        self._ksupport = np.array([])

        # flags active joint displacements
        self.ux = ux
        self.uy = uy
        self.uz = uz
        self.rx = rx
        self.ry = ry
        self.rz = rz

        # dict materials and sections
        self.materials = {}
        self.sections = {}

        # dict joints and frames
        self.joints = {}
        self.frames = {}

        # dict supports and load patterns
        self.supports = {}
        self.load_patterns = {}

        # dict displacements, end actions and reactions
        self.displacements = {}
        self.end_actions = {}
        self.reactions = {}

        # dict internal forces and displacements
        self.internal_forces = {}
        self.internal_displacements = {}

```

El usuario puede describir el modelo con los objetos tipo **Structure** agregándole objetos que representan materiales, secciones transversales, nodos, elementos aporticados, apoyos, patrones de carga, fuerzas puntuales aplicadas en los nodos o en los elementos aporticados, y cargas distribuidas en los elementos aporticados, mediante los métodos que comienzan con *add*.

Estos métodos reciben uno o varios argumentos de entrada obligatorios y una serie de argumentos de entrada opcionales para crear los objetos y almacenarlos en los respectivos diccionarios del objeto tipo **Structure**.

Dichos métodos son similares entre sí, pero cambia el diccionario al cual se le está agregando la nueva entrada y el objeto que se está creando. Por ejemplo, en el algoritmo 3.3 se presenta el método `add_section()`. Los argumentos de entrada **name**, ***args** y ****kwargs** son pasados al constructor de la clase **Section**, junto con el objeto **Structure**, para crear un objeto tipo **Section**, mientras que el argumento **name** es usado como *llave* para almacenar dicho objeto en el diccionario **sections**. El método devuelve el objeto creado.

Algoritmo 3.3: Método `add_section()` de la clase **Structure**.

```
def add_section(self, name, *args, **kwargs):
    """
    Add a section.

    Parameters
    -----
    name : str
        Section name.

    Returns
    -----
    section : Section
        Section.
    """
    section = self.sections[name] = Section(self, name, *args, **kwargs)

    return section
```

En el caso de los métodos `add_material()`, `add_joint()`, `add_frame()` y `add_load_pattern()`, el diccionario ya no es **sections** sino **materials**, **joints**, **frames** o **load_patterns**, según corresponda, y el objeto a crear ya no es de tipo **Sections** sino de tipo **Material**, **Joint**, **Frame** o **LoadPattern**, respectivamente.

El método `add_rectangular_section()` permite agregar objetos tipo **Rectangular Section** de manera similar a como lo hace el método `add_section()`, con la diferencia que este método recibe como argumentos de entrada, además del argumento **name**, el ancho y el alto la sección transversal rectangular. En el algoritmo 3.4 se presenta el

método `add_rectangular_section()`. El objeto creado se almacena en el diccionario `sections`.

Algoritmo 3.4: Método `add_rectangular_section()` de la clase **Structure**.

```
def add_rectangular_section(self, name, width, height):
    """
    Add a rectangular section.

    Parameters
    -----
    name : str
        Rectangular section name.
    width : float
        Width.
    height : float
        Height.

    Returns
    -----
    rect_sect : RectangularSection
        RectangularSection.
    """
    rect_sect = self.sections[name] = RectangularSection(self, name,
width, height)

    return rect_sect
```

El método `add_support()` es similar a los anteriores, con la diferencia que recibe como parámetro de entrada `joint`, el *nombre* del nodo donde se van a aplicar las restricciones, tal como se presenta en el algoritmo 3.5.

Algoritmo 3.5: Método `add_support()` de la clase **Structure**.

```
def add_support(self, joint, *args, **kwargs):
    """
    Add a support.

    Parameters
    -----
    joint : str
        Joint name.

    Returns
```

```

support : Support
    Support.
    """
    support = self.supports[joint] = Support(self, joint, *args, **kwargs)

```

Por su parte, los métodos `add_load_at_joint()`, `add_load_at_frame()` y `add_distributed_load()` reciben dos argumentos de entrada obligatorios y una serie de argumentos de entrada opcionales. El primer argumento de entrada obligatorio es el nombre del objeto tipo `LoadPattern` y el segundo es el nombre del objeto tipo `Joint` o del objeto tipo `Frame`, según el método. En el algoritmo 3.6 se presenta el método `add_load_at_joint()`. Con el nombre del patrón de carga se recupera el objeto tipo `LoadPattern` del diccionario `load_patterns`. El nombre del nodo y los demás argumentos de entrada opcionales son pasados al método `add_point_load_at_joint()` de la clase `LoadPattern`.

Algoritmo 3.6: Método `add_load_at_joint()` de la clase `Structure`.

```

def add_load_at_joint(self, load_pattern, joint, *args, **kwargs):
    """
    Add a point load at joint.

    Parameters
    

---


    load_pattern : str
        Load pattern name.
    joint : str
        Joint name.

    Returns
    

---


    pointLoad : PointLoadAtJoint
        Point load at joint.
    """
    pointLoad = self.load_patterns[load_pattern].add_point_load_at_joint(
        joint, *args, **kwargs)

    return pointLoad

```

En el caso de los métodos `add_point_load_at_frame()` y `add_distributed_loads()`, el nombre del objeto tipo `Frame` y los demás argumentos de entrada opcionales son

pasados a los métodos `add_point_load_at_frame()` y `add_distributed_load()` del objeto tipo `LoadPattern`.

Cuando el usuario termina de describir el modelo puede ejecutar el método `solve()` de la clase `Structure` para realizar el análisis de la estructura. `pymas` permite analizar diferentes patrones de carga, almacenando los resultados de desplazamientos, fuerzas en los extremos de los elementos, reacciones, fuerzas internas y desplazamientos al interior de los elemento en los atributos `displacements`, `end_actions`, `reactions`, `internal_forces` y `internal_displacements`, respectivamente.

En las siguientes secciones se presenta la implementación de las clases específicas, con las cuales el usuario puede describir el modelo (`Material`, `Section`, `Joint`, etc.) y después la implementación de los demás métodos de la clase `Structure`.

3.3. Clases específicas

A continuación se presentan las clases específicas de `pymas` que se encuentran en los archivos `primitives.py` y `classtools.py` y que permiten describir el modelo:

- `Material`,
- `Section`,
- `RectangularSection`,
- `Joint`,
- `Frame`,
- `Support`,
- `LoadPattern`,
- `PointLoadAtJoint`,
- `PointLoadAtFrame`,
- `DistributedLoad`,
- `Displacement`,
- `EndActions`,
- `Reaction`,
- `InternalForces`,
- `InternalDisplacements`.

En el archivo `classtools.py` se define la clase `AttrDisplay`, que implementa una representación de los objetos más cómoda para los usuarios. A continuación se presenta la implementación de todas las clases de `pymas`.

3.3.1. Material

La clase `Material` representa un material lineal elástico al definir los valores del módulo de Young y del módulo a cortante. En el algoritmo 3.7 se presenta la implementación de la clase `Material`. El constructor de la clase recibe los argumentos de entrada `E` y `G`, los cuales tienen `None` como valor por defecto. Estos valores son pasados a los atributos `E` y `G` del objeto tipo `Material` respectivamente.

Algoritmo 3.7: Clase `Material` implementada en el archivo `primitives.py`.

```
class Material(AttrDisplay):
    """
    Linear elastic material.

    Attributes
    -----
    name : str
        Material name.
    E : float
        Young's modulus.
    G : float
        Shear modulus.
    """

    def __init__(self, parent, name, E=None, G=None):
        """
        Instantiate a Material object.

        Parameters
        -----
        parent : Structure
            Structure.
        name : str
            Material name.
        E : float, optional
            Young's modulus.
        G : float, optional
            Shear modulus.
        """
        self.__parent = parent
        self.name = name
        self.E = E
        self.G = G
```

3.3.2. Section

La clase **Section** representa la sección transversal de los elementos aporticados de manera general, al definir los valores del área transversal, de la constante de torsión y de las inercias alrededor de los ejes principales.

En el algoritmo 3.8 se presenta la implementación de la clase **Section**. El constructor de la clase recibe los argumentos de entrada **A**, **J**, **Iy** y **Iz**, los cuales tienen **None** como valor por defecto. Estos valores son pasados a los atributos **A**, **J**, **Iy** y **Iz**, respectivamente.

Algoritmo 3.8: Clase **Section** implementada en el archivo **primitives.py**.

```
class Section(AttrDisplay):
    """
    Cross section.

    Attributes
    -----
    name : str
        Section name.
    A : float
        Area.
    J : float
        Torsional constant.
    Iy : float
        Inertia around y-axis.
    Iz : float
        Inertia around z-axis.
    """

    def __init__(self, parent, name, A=None, J=None, Iy=None, Iz=None):
        """
        Instantiate a Section object.

        Parameters
        -----
        parent : Structure
            Structure.
        name : str
            Section name.
        A : float, optional
            Area.
        J : float, optional
```

```

        Torsional constant.
        Iy : float , optional
        Inertia around y-axis.
        Iz : float , optional
        Inertia around z-axis.
        """
        self.__parent = parent
        self.name = name
        self.A = A
        self.J = J
        self.Iy = Iy
        self.Iz = Iz

```

3.3.3. RectangularSection

La clase **RectangularSection** representa la sección transversal de forma rectangular de los elementos aporticados, al definir los valores de su base y altura.

En el algoritmo 3.9 se presenta la implementación de la clase **RectangularSection**. Esta clase hereda todos los métodos y atributos de la clase **Section**, para evitar duplicar el código a lo largo del programa, al pasar dicha clase como argumento de entrada cuando se construye la clase **RectangularSection**.

El constructor de la clase recibe los argumentos de entrada **name**, **width** y **height**, los cuales son obligatorios (a diferencia de los argumentos de entrada del constructor de la clase **Section**). Los valores de los argumentos de entrada son asignados a los respectivos atributos de los objeto tipo **RectangularSection**.

Se asume que el valor del parámetro **width** corresponde a la dimensión del elemento aporticado de sección transversal rectangular a lo largo del eje *y* del sistema de coordenadas local, mientras que el parámetro **height** corresponde a la dimensión del elemento aporticado a lo largo del eje *z*.

Teniendo en cuenta esto se calcula el área, la constante de torsión y los momentos de inercia alrededor de los ejes *y* y *z*. Para calcular la constante de torsión se utiliza la expresión (3-1), la misma que se presenta en Uribe-Escamilla (1995),

$$J = \left(\frac{1}{3} - 0.21 \frac{a}{b} \left(1 - (1/12)(a/b)^4 \right) \right) ba^3 \quad (3-1)$$

donde a es la dimensión menor del rectángulo y b su dimensión mayor.

Finalmente, las propiedades de la sección transversal son pasadas al constructor de la clase `Section` para asignárselas a los atributos del objeto tipo `RectangularSection`. Esto se hace mediante la función `super()` que trae Python por defecto, la cual genera una referencia a la *clase padre*, en este caso, la clase `Section`.

Algoritmo 3.9: Clase `RectangularSection` implementada en el archivo `primitives.py`.

```
class RectangularSection(Section):
    """
    Rectangular cross section.

    Attributes
    -----
    name : str
        Section name.
    width : float
        Width.
    height : float
        Height.
    A : float
        Area.
    J : float
        Torsion.
    Iy : float
        Inertia around y-axis.
    Iz : float
        Inertia around z-axis.
    """

    def __init__(self, parent, name, width, height):
        """
        Instantiate a RectangularSection object.

        Parameters
        -----
        parent : Structure
            Structure.
        name : str
            Section name.
        width : float
            Width.
```

```

    height : float
    Height.
    """
    a = min(width, height)
    b = max(width, height)

    A = width * height
    J = (1/3 - 0.21 * (a / b) * (1 - (1/12) * (a/b)**4)) * b * a ** 3
    Iy = (1 / 12) * height * width ** 3
    Iz = (1 / 12) * width * height ** 3

    super().__init__(parent, name, A, J, Iy, Iz)
    self.width = width
    self.height = height

```

3.3.4. Joint

La clase `Joint` representa los nodos de la estructura, al definir sus coordenadas en el sistema de coordenadas global. En el algoritmo 3.10 se presenta la implementación de la clase `Joint`. El constructor de la clase recibe tres argumentos de entrada opcionales, uno para cada una de las coordenadas, los cuales tienen `None` como valor por defecto. Estos valores son pasados a los atributos `x`, `y` y `z` del objeto respectivamente.

La clase `Joint` implementa el método `get_coordinate()` que genera una *array* con las coordenadas del objeto. En caso que alguna coordenada no haya sido especificada, se toma como igual a cero.

Algoritmo 3.10: Clase `Joint` implementada en el archivo `primitives.py`.

```

class Joint(AttrDisplay):
    """
    End of frames.

    Attributes
    -----
    name : str
        Joint name.
    x : float
        X coordinate.
    y : float
        Y coordinate.
    z : float

```

```

        Z coordinate.

Methods


---


get_coordinate()
    Get joint coordinate.
    """

def __init__(self, parent, name, x=None, y=None, z=None):
    """
    Instantiate a Joint object.

    Parameters
    

---


    parent : Structure
        Structure.
    name : str
        Joint name.
    x : float, optional
        X coordinate.
    y : float, optional
        Y coordinate.
    z : float, optional
        Z coordinate.
    """
    self.__parent = parent
    self.name = name
    self.x = x
    self.y = y
    self.z = z

def get_coordinate(self):
    """Get coordinate"""
    x = self.x if self.x is not None else 0
    y = self.y if self.y is not None else 0
    z = self.z if self.z is not None else 0

    return np.array([x, y, z])

```

3.3.5. Frame

La clase **Frame** representa los elementos aporticados de la estructura, al definir sus nodos, material y sección transversal. En la figura **3-5** se presentan los métodos y atributos

de esta clase.

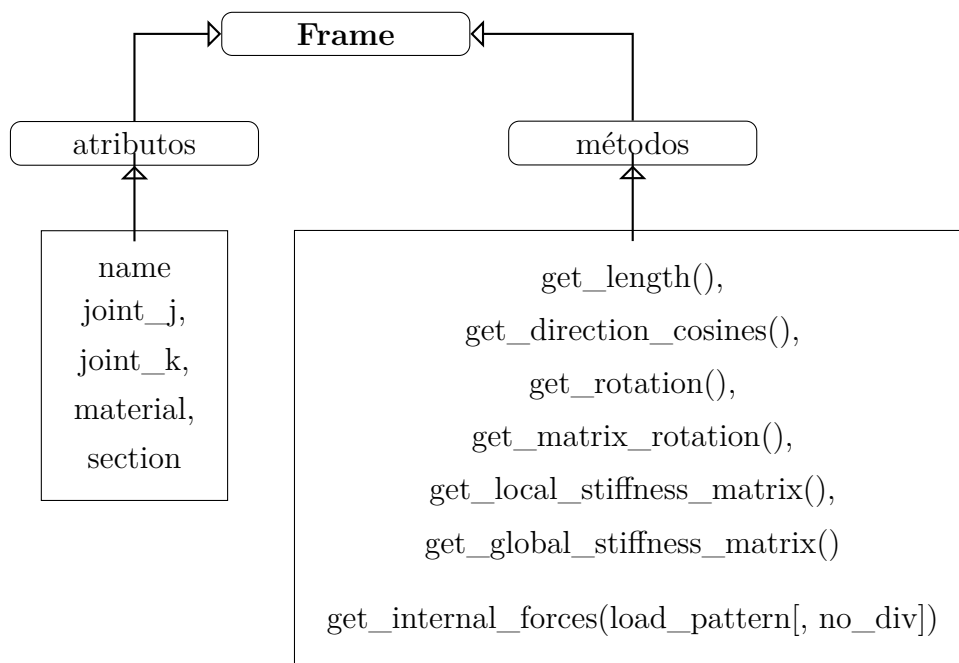


Figura 3-5: Métodos y atributos de la clase **Frame**.

En la figura **3-6** se presenta un elemento aporticado i con sus nodos j y k empotrados. El sistema de coordenadas local del elemento tiene como origen el nodo j ; el eje x coincide con el eje centroidal del elemento y es positivo en el sentido del nodo j al nodo k .

Los ejes y y z son los ejes principales del elemento de manera que los planos xy y zx son los planos principales de flexión. Se asume que el centro de cortante y el centroide del elemento coinciden de tal forma que el momento flector es independiente de la fuerza cortante.

Los grados de libertad se numeran del 1 al 12, empezando por las translaciones y las rotaciones del nodo j , tomados en orden x , y , z respectivamente.

En el algoritmo 3.11 se presenta el constructor de la clase **Frame**. El constructor de la clase **Frame** recibe cinco argumentos de entrada; el nombre del elemento, el nodo cercano, el nodo lejano, el material y la sección. Los argumentos de entrada son pasados a los atributos `joint_j`, `joint_k`, `material` y `section` respectivamente.

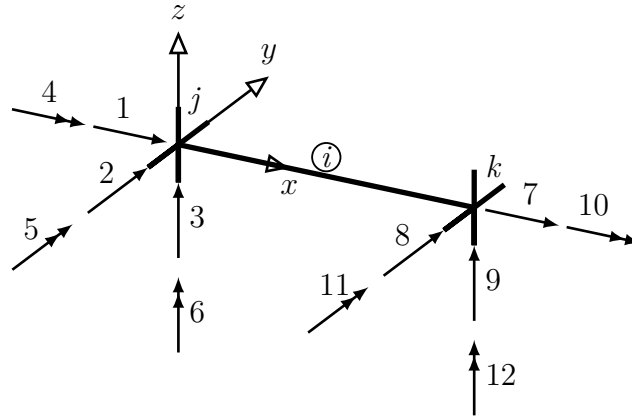


Figura 3-6: Elemento aporticado con sus grados de libertad y su sistema de coordenadas local.

Algoritmo 3.11: Constructor de la clase **Frame**.

```
def __init__(self, parent, name, joint_j=None, joint_k=None, material=
None, section=None):
    """
    Instantiate a Frame object.

    Parameters
    -----
    parent : Structure
        Structure.
    name : str
        Frame name.
    joint_j : str, optional
        Near Joint.
    joint_k : str, optional
        Far Joint.
    material : str, optional
        Material.
    section : str, optional
        Section.
    """
    self.__parent = parent
    self.name = name
    self.joint_j = joint_j
    self.joint_k = joint_k
    self.material = material
    self.section = section
```

A continuación se presentan los métodos de la clase **Frame**, con los cuales se puede, entre otras cosas, calcular la matriz de rigidez de los elementos tipo **Frame**.

get_length()

El método **get_length()** de la clase **Frame** permite calcular la longitud de los elementos aporticados representado por objetos tipo **Frame**.

En el algoritmo 3.12 se presenta la implementación del método **get_length()**, con el cual se calcula la distancia que hay entre las coordenadas de los nodos del elemento aporticado. Para esto hace un llamado a la función **np.linalg.norm()** con la resta entre las coordenadas de los objeto tipo **Joint**, las cuales obtiene con el método **get_coordinate()** (véase el algoritmo 3.10).

Algoritmo 3.12: Método **get_length()** de la clase **Frame**.

```
def get_length(self):
    """Get length"""
    j = self._parent.joints[self.joint_j].get_coordinate()
    k = self._parent.joints[self.joint_k].get_coordinate()

    return np.linalg.norm(k - j)
```

get_direction_cosines()

El método **get_direction_cosines()** de la clase **Frame** permite calcular los cosenos directores del eje x del sistema de coordenadas local de los elementos aporticados, representados por objetos tipo **Frame**, con respecto al sistema de coordenadas global.

En el algoritmo 3.13 se presenta la implementación del método **get_direction_cosines()**. El método resta las coordenadas de los nodos del elemento aporticado entre si y almacena el resultado en la variable **vector**. Después divide cada uno de los elementos de **vector** por la norma de dicha variable.

Algoritmo 3.13: Método **get_direction_cosines()** de la clase **Frame**.

```
def get_direction_cosines(self):
    """Get direction cosines"""
    j = self._parent.joints[self.joint_j].get_coordinate()
```

```

k = self._parent.joints[self.joint_k].get_coordinate()
vector = k - j

return vector / np.linalg.norm(vector)

```

get_rotation()

El método `get_rotation()` de la clase `Frame` permite calcular la rotación de los elementos aportados, representados por objetos tipo `Frame`, con respecto al sistema de coordenadas global.

Según el *teorema de rotación de Euler* (Sten, 2022), siempre es posible encontrar un diámetro de una esfera cuya posición es la misma después de rotar la esfera alrededor de su centro, por lo que cualquier secuencia de rotaciones de un sistema coordenado tridimensional es equivalente a una única rotación alrededor de un eje que pase por el origen.

Por ejemplo, en la figura 3-7 se presenta una esfera que rota alrededor de un eje $\mathbf{n}$ que pasa por su centro un ángulo θ . Durante la rotación, el diámetro marcado sobre la superficie de la esfera no cambia de posición. Esta rotación queda completamente descrita entonces por el eje $\mathbf{n}$, que es perpendicular al plano que contiene al diámetro, y el ángulo θ .

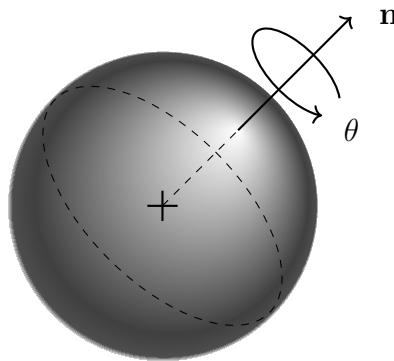


Figura 3-7: Descripción de la rotación de una esfera a través de una rotación alrededor de un eje que pasa por su centro.

El ángulo θ y el vector $\mathbf{n}$ que definen la rotación del eje x del sistema de coordenadas global hacia el eje x_m del sistema de coordenadas local de un elemento aportado se puede calcular como

$$\begin{aligned}\mathbf{n} &= (1, 0, 0) \times \mathbf{x}_m \\ \theta &= \arccos((1, 0, 0) \cdot \mathbf{x}_m)\end{aligned}\tag{3-2}$$

Según Dunn (2002), la rotación de un sistema de coordenadas tridimensionales alrededor del eje $\mathbf{n}$ una cantidad θ se puede describir mediante un *cuaternión* como

$$\mathbf{q} = [\cos(\theta/2) \quad \sin(\theta/2)(x \mathbf{i}, y \mathbf{j}, z \mathbf{k})]\tag{3-3}$$

donde x , y y z son las componentes del vector $\mathbf{n}$.

Se puede obtener la matriz de rotación a partir de un cuaternión como:

$$\mathbf{R} = \begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy + 2wz & 2xz - 2wy \\ 2xy - 2wz & 1 - 2x^2 - 2z^2 & 2yz + 2wx \\ 2xz + 2wy & 2yz - 2wx & 1 - 2x^2 - 2y^2 \end{bmatrix}\tag{3-4}$$

donde w es la parte escalar y x , y y z la parte vectorial del cuaternión.

En el algoritmo 3.14 se presenta la implementación del método `get_rotation()`. El método calcula el *cuaternión* que representa la rotación del eje x del sistema de coordenadas global hacia el eje x del sistema de coordenadas local del elemento aporticado.

Para esto, se almacena el eje global x y el eje local x en las variables `v_from` y `v_to` respectivamente. El eje global x es igual a $(1, 0, 0)$ mientras que el eje local x se calcula mediante el método `get_direction_cosines()` (algoritmo 3.13).

Después, se verifica si las variables `v_from` y `v_to` son iguales entre sí, o si una variable es el inverso aditivo de la otra.

En el caso que las variables sean iguales entre sí, no hay rotación y el ángulo θ es igual a cero, por lo tanto el cuaternión es igual a $(1, 0 \times \mathbf{n})$ (véase la ecuación 3-3). En caso contrario, el ángulo θ que describe la rotación es igual a 180° . Como eje se asume el eje global z , por lo que el cuaternión es igual a $(0, 1 \times (0, 0, 1))$.

Si las variables `v_from` y `v_to` no son iguales entre sí, y una no es el inverso aditivo de la otra, entonces se calcula el eje y el ángulo que describen la rotación del eje global x hacia el eje local x del elemento aporticado, aplicando las expresiones de ecuación (3-2).

Para calcular el eje se halla el producto cruz entre el eje global x y el eje local x mediante la función `cross()`. Después se normaliza el resultado dividiéndolo por su norma, con

ayuda de la función `linalg.norm()`.

El ángulo se halla calculando el arcocoseno del producto punto entre el eje global x y el eje local x . Esto se calcula mediante las funciones `dot()` y `arccos()`, respectivamente.

Finalmente, se aplican las expresiones de la ecuación (3-3) para crear un objeto tipo `Rotation`, mediante la función `Rotation.from_quat()`.

Según Virtanen *et al.* (2020), la función `from_quat()` permite crear objetos tipo `Rotation`, los cuales son una interfaz para inicializar y representar rotaciones en el espacio, mediante un cuaternión.

Algoritmo 3.14: Método `get_rotation()` de la clase `Frame`.

```
def get_rotation(self):
    """Get rotation"""
    v_from = np.array([1, 0, 0])
    v_to = self.get_direction_cosines()

    if np.all(v_from == v_to):
        return Rotation.from_quat([0, 0, 0, 1])

    elif np.all(v_from == -v_to):
        return Rotation.from_quat([0, 0, 1, 0])

    else:
        w = np.cross(v_from, v_to)
        w = w / linalg.norm(w)
        theta = np.arccos(np.dot(v_from, v_to))

        return Rotation.from_quat([x * np.sin(theta/2) for x in w] + [np.
            cos(theta/2)])
```

`get_rotation_matrix()`

El método `get_rotation_matrix()` de la clase `Frame` permite calcular la matriz de transformación de rotación de los elementos aportados, representados por objetos tipo `Frame`, con respecto al sistema de coordenadas global.

Según Weaver y Gere (1990), la matriz de transformación de rotación $\mathbf{R}_T$ para un elemento aporticado es

$$\mathbf{R}_T = \begin{bmatrix} \mathbf{R} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{R} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{R} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{R} \end{bmatrix} \quad (3-5)$$

donde $\mathbf{R}$ es la matriz de rotación presentada en (3-4).

En el algoritmo 3.15 se presenta la implementación del método `get_rotation_matrix()`. Según los grados de libertad activados, la matriz de transformación de rotación se genera a partir del cuaterión del elemento aporticado, calculada con el método `get_rotation()` de la clase **Frame** (véase el algoritmo 3.14), y la función `bsr_matrix()`.

Según Virtanen *et al.* (2020), el método `as_matrix()` de la clase **Rotation** calcula la matriz de rotación de los objetos tipo **Rotation**, y la función `bsr_matrix()` crea matrices dispersas con submatrices densas describiéndolas en la representación estándar (`data`, `indices`, `indptr`). En dicha representación los índices de la columna de cada submatriz en la fila `i` de la matriz dispersa están almacenados en `indices[indptr[i]:indptr[i+1]]` y los valores correspondientes almacenados en `data[indptr[i]:indptr[i+1]]`.

En las variable `indptr` e `indices` se almacenan los *arrays* `[0, 1, 2]` y `[0, 1]` respectivamente. Estas variables describen en la representación estándar las posiciones que ocupan dos submatrices en la diagonal principal de una matriz dispersa.

En la primer fila hay una submatriz en la primer columna (`indices[indptr[0]:indptr[1]] -> indices[0:1] -> indices[0] -> 0`) y en la segunda fila hay una submatriz en la segunda columna (`indices[indptr[1]:indptr[2]] -> indices[1:2] -> indices[1] -> 1`).

Inicialmente se calcula la matriz de transformación de rotación para un solo nodo. Después se seleccionan las filas y las columnas de esta matriz asociadas a los grados de libertad activados. Finalmente se crea toda la matriz de transformación de rotación duplicando los valores seleccionados.

Para crear la matriz de transformación de rotación para un solo nodo se duplica la matriz de rotación del elemento aporticado, mediante la función `tile`, y se almacena en la variable `data`. Después se pasa junto con las variables `indptr` e `indices` a la

función `bsr_matrix()`.

La matriz de transformación de rotación del elemento aporticado se crea al indicar dos matrices de transformación de rotación para un solo nodo en la diagonal principal de una matriz dispersa, después de haber seleccionado las filas y columnas asociadas a los grados de libertad activados. El tamaño de la matriz de transformación de rotación se calcula en función de la cantidad de grados de libertad activados.

Algoritmo 3.15: Método `get_rotation_matrix()` de la clase `Frame`.

```
def get_matrix_rotation(self):
    """Get matrix rotation"""

    # rotation as direction cosine matrix
    indptr = np.array([0, 1, 2])
    indices = np.array([0, 1])
    data = np.tile(self.get_rotation().as_matrix(), (2, 1, 1))

    # matrix rotation for a joint
    t1 = bsr_matrix((data, indices, indptr), shape=(6, 6)).toarray()

    active_joint_displacements = np.nonzero(self._parent.
get_flags_active_joint_displacements())[0]
    n = 2 * np.size(active_joint_displacements)

    t1 = t1[active_joint_displacements[:, None],
active_joint_displacements]
    data = np.tile(t1, (2, 1, 1))

    return bsr_matrix((data, indices, indptr), shape=(n, n)).toarray()
```

`get_local_stiffness_matrix()`

El método `get_local_stiffness_matrix()` de la clase `Frame` permite calcular la matriz de rigidez de los elementos aporticados, representados por objetos tipo `Frame`, con respecto al sistema de coordenadas local.

Según Weaver y Gere (1990), la ecuación (3-6) es la matrix de rigidez del elemento aporticado en coordenadas locales, donde E es el módulo de Young y G es el módulo de elasticidad a cortante del material, L es la longitud del elemento y A , J , I_y e I_z son el área, la constante de torsión y los momentos principales de inercia de la sección

transversal.

$$\begin{matrix}
 & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{matrix} \\
 \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \end{matrix} & \left[\begin{array}{cccccccccccc}
 \frac{EA}{L} & 0 & 0 & 0 & 0 & 0 & -\frac{EA}{L} & 0 & 0 & 0 & 0 & 0 \\
 0 & \frac{12EI_z}{L^3} & 0 & 0 & 0 & \frac{6EI_z}{L^2} & 0 & -\frac{12EI_z}{L^3} & 0 & 0 & 0 & \frac{6EI_z}{L^2} \\
 0 & 0 & \frac{12EI_y}{L^3} & 0 & -\frac{6EI_y}{L^2} & 0 & 0 & 0 & -\frac{12EI_y}{L^3} & 0 & -\frac{6EI_y}{L^2} & 0 \\
 0 & 0 & 0 & \frac{GJ}{L} & 0 & 0 & 0 & 0 & 0 & -\frac{GJ}{L} & 0 & 0 \\
 0 & 0 & 0 & 0 & \frac{4EI_y}{L} & 0 & 0 & 0 & \frac{6EI_y}{L^2} & 0 & \frac{2EI_y}{L} & 0 \\
 0 & 0 & 0 & 0 & 0 & \frac{4EI_z}{L} & 0 & -\frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{2EI_z}{L} \\
 0 & 0 & 0 & 0 & 0 & 0 & \frac{EA}{L} & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{12EI_z}{L^3} & 0 & 0 & 0 & -\frac{6EI_z}{L^2} \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{12EI_y}{L^3} & 0 & \frac{6EI_y}{L^2} & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{GJ}{L} & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{4EI_y}{L} & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{4EI_z}{L}
 \end{array} \right]
 \end{matrix} \quad (3-6)$$

En el algoritmo 3.16 se presenta la implementación del método `get_local_stiffness_matrix()`, que genera la matriz de rigidez en el sistema de coordenadas local.

La matriz de rigidez en el sistema de coordenadas local se calcula con los atributos del material, de la sección transversal, de los nodos de los elementos aporticados y la función `coo_matrix()`.

Según Virtanen *et al.* (2020), con la función `coo_matrix()` se pueden crear matrices dispersas en el formato coordenado, también conocido como el formato *ijv* o el formato triple. En este formato los índices de las filas, de las columnas y los respectivos valores de la matriz dispersa son almacenados en tres *arrays* independientes `i`, `j` y `data` de tal manera que se cumpla `A[i[k], j[k]] = data[k]`.

En las variables `l`, `e`, `g`, `j`, `iy` y `iz` se almacenan la longitud del elemento aporticado (véase el algoritmo 3.12), el módulo de Young y el módulo a cortante del material, y las inercias principales de la sección transversal con respecto a los ejes x , y y z del sistema de coordenadas local.

Después se calcula el módulo de Young dividido entre varias potencias de la longitud del elemento aporticado y los resultados se almacena en las variables `e1`, `e12` y `e13` respectivamente. El número al final del nombre de estas variables indica la potencia de la longitud del elemento.

Con estas variables se calculan los términos EA/L , GJ/L , EI_y/L , EI_z/L , $6EI_y/L^2$, $6EI_z/L^2$, $12EI_y/L^3$ y $12EI_z/L^3$, los cuales son almacenados en las variables `ael`, `gjl`,

`e_iy_1`, `e_iz_1`, `e_iy_12`, `e_iz_12`, `e_iy_13` y `e_iz_13`, respectivamente.

Con estas variables se describe la matriz de rigidez en coordenadas locales como una matriz dispersa en el formato *ijv*. Los índices de las filas y las columnas se almacenan en los arreglos `rows` y `cols`, respectivamente, mientras que los valores de la matriz se almacenan en el arreglo `data`.

Por ejemplo, para describir los términos de la matriz de rigidez asociados a las sollicitaciones axiales, se le pasa a los arreglos `rows` y `cols` los valores `[0, 6, 0, 6]` y `[0, 6, 6, 0]`, y al arreglo `data` se le pasa los valores `[ael, ael, -eal, -eal]`. Para los otros términos de la matriz de rigidez se procede de manera similar.

Algoritmo 3.16: Método `get_local_stiffness_matrix()` de la clase `Frame`.

```
def get_local_stiffness_matrix(self):
    """Get local stiffness matrix"""
    l = self.get_length()

    material = self._parent.materials[self.material]
    e = material.E if material.E is not None else 0
    g = material.G if material.G is not None else 0

    section = self._parent.sections[self.section]
    a = section.A if section.A is not None else 0
    j = section.J if section.J is not None else 0
    iy = section.Iy if section.Iy is not None else 0
    iz = section.Iz if section.Iz is not None else 0

    el = e / l
    el2 = e / l ** 2
    el3 = e / l ** 3

    ael = a * el
    gjl = j * g / l

    e_iy_1 = iy * el
    e_iz_1 = iz * el

    e_iy_12 = 6 * iy * el2
    e_iz_12 = 6 * iz * el2

    e_iy_13 = 12 * iy * el3
    e_iz_13 = 12 * iz * el3
```

```

rows = np.empty(40, dtype=int)
cols = np.empty(40, dtype=int)
data = np.empty(40)

# AE / L
rows[:4] = np.array([0, 6, 0, 6])
cols[:4] = np.array([0, 6, 6, 0])
data[:4] = np.array([ael, ael, -ael, -ael])

# GJ / L
rows[4:8] = np.array([3, 9, 3, 9])
cols[4:8] = np.array([3, 9, 9, 3])
data[4:8] = np.array([gjl, gjl, -gjl, -gjl])

# 12EI / L^3
rows[8:12] = np.array([1, 7, 1, 7])
cols[8:12] = np.array([1, 7, 7, 1])
data[8:12] = np.array([e_iz_l3, e_iz_l3, -e_iz_l3, -e_iz_l3])

rows[12:16] = np.array([2, 8, 2, 8])
cols[12:16] = np.array([2, 8, 8, 2])
data[12:16] = np.array([e_iy_l3, e_iy_l3, -e_iy_l3, -e_iy_l3])

# 6EI / L^2
rows[16:20] = np.array([1, 5, 1, 11])
cols[16:20] = np.array([5, 1, 11, 1])
data[16:20] = np.array(4 * [e_iz_l2])

rows[20:24] = np.array([5, 7, 7, 11])
cols[20:24] = np.array([7, 5, 11, 7])
data[20:24] = np.array(4 * [-e_iz_l2])

rows[24:28] = np.array([2, 4, 2, 10])
cols[24:28] = np.array([4, 2, 10, 2])
data[24:28] = np.array(4 * [-e_iy_l2])

rows[28:32] = np.array([4, 8, 8, 10])
cols[28:32] = np.array([8, 4, 10, 8])
data[28:32] = np.array(4 * [e_iy_l2])

# 4EI / L
rows[32:36] = np.array([4, 10, 5, 11])
cols[32:36] = np.array([4, 10, 5, 11])

```

```

    data[32:36] = np.array([4 * e_iy_l, 4 * e_iy_l, 4 * e_iz_l, 4 *
e_iz_l])

    rows[36:] = np.array([10, 4, 11, 5])
    cols[36:] = np.array([4, 10, 5, 11])
    data[36:] = np.array([2 * e_iy_l, 2 * e_iy_l, 2 * e_iz_l, 2 * e_iz_l
])

    return coo_matrix((data, (rows, cols)), shape=(12, 12)).toarray()

```

get_global_stiffness_matrix()

El método `get_global_stiffness_matrix()` de la clase `Frame` permite calcular la matriz de rigidez de los elementos aporticados, representados por objetos tipo `Frame`, con respecto al sistema de coordenadas global.

Según Weaver y Gere (1990), la matriz de rigidez de los elementos aporticados con respecto al sistema de coordenadas global se puede calcular como

$$\mathbf{S}_{MS} = \mathbf{R}_T \mathbf{S}_M \mathbf{R}_T^T \quad (3-7)$$

donde $\mathbf{R}_T$ y $\mathbf{S}_M$ son la matriz de transformación de rotación y la matriz de rigidez en el sistema de coordenadas local del elemento aporticado.

En el algoritmo 3.17 se presenta la implementación del método `get_global_stiffness_matrix()`. Según los grados de libertad activados se genera la matriz de rigidez en el sistema de coordenadas global.

La matriz de rigidez en el sistema de coordenadas global se calcula con la matriz de rigidez en el sistema de coordenadas local y la matriz de transformación de rotación del elemento aporticado. Estas matrices son calculadas con los métodos `get_matrix_rotation()` (algoritmo 3.15) y `get_local_stiffness_matrix()` (algoritmo 3.16), y almacenadas en las variables `k` y `t`, respectivamente.

Finalmente, se operan las matrices obtenidas según la ecuación (3-7) para calcular la matriz de rigidez del elemento aporticado en el sistema de coordenadas global con las funciones `dot()` y `transpose()`.

Algoritmo 3.17: Método `get_global_stiffness_matrix()` de la clase `Frame`.

```
def get_global_stiffness_matrix(self):
    """Get global stiffness matrix"""

    flags_active_joint_displacements = self._parent.
    get_flags_active_joint_displacements()
    active_frame_displacement = np.nonzero(np.tile(
    flags_active_joint_displacements, 2))[0]

    k_local = self.get_local_stiffness_matrix()
    t = self.get_matrix_rotation()
    k_global = np.dot(np.dot(t, k_local), np.transpose(t))

    return k_global[active_frame_displacement[:, None],
    active_frame_displacement]
```

`get_internal_forces()`

El método `get_internal_forces()` de la clase `Frame` permite calcular las fuerzas internas a lo largo de los elementos aporticados, representados por objetos tipo `Frame`.

En el algoritmo 3.18 se presenta la implementación del método `get_internal_forces()`. El método recibe como argumentos de entrada el nombre del patrón de carga y el número de divisiones donde se calculan las solicitaciones, por defecto igual a 100. Se genera un diccionario con las entradas `fx`, `fy`, `fz`, `mx`, `my` y `mz`, las cuales contienen una lista con la respectiva solución a lo largo del elemento.

Las fuerzas al interior del elemento se calculan con base en las fuerzas en sus extremos, calculados una vez se ha solucionado el modelo, y las diferentes cargas aplicadas en él.

Algoritmo 3.18: Método `get_internal_forces()` de la clase `Frame`.

```
def get_internal_forces(self, load_pattern, no_div=100):
    """
    Get internal forces.

    Parameters
    -----
    load_pattern : str
        Load pattern name.
    no_div : float, optional
        Number divisions.
```

Returns

```
internal_forces : dict
    Internal forces.
    """
```

```
loadPattern = self._parent.load_patterns[load_pattern]
end_actions = self._parent.end_actions[load_pattern][self.name]
```

```
length = self.get_length()
```

```
fx_j = end_actions.fx_j if end_actions.fx_j is not None else 0
fy_j = end_actions.fy_j if end_actions.fy_j is not None else 0
fz_j = end_actions.fz_j if end_actions.fz_j is not None else 0
mx_j = end_actions.mx_j if end_actions.mx_j is not None else 0
my_j = end_actions.my_j if end_actions.my_j is not None else 0
mz_j = end_actions.mz_j if end_actions.mz_j is not None else 0
```

```
internal_forces = {}
internal_forces['fx'] = np.full(shape=no_div+1, fill_value=-fx_j)
internal_forces['fy'] = np.full(shape=no_div+1, fill_value= fy_j)
internal_forces['fz'] = np.full(shape=no_div+1, fill_value= fz_j)
internal_forces['mx'] = np.full(shape=no_div+1, fill_value=-mx_j)
internal_forces['my'] = np.full(shape=no_div+1, fill_value= my_j)
internal_forces['mz'] = np.full(shape=no_div+1, fill_value=-mz_j)
```

```
for i in range(no_div+1):
    x = (i / no_div) * length
    internal_forces['my'][i] += fz_j * x
    internal_forces['mz'][i] += fy_j * x
```

```
if self.name in loadPattern.distributed_loads:
    for distributed_load in loadPattern.distributed_loads[self.name]:
        fx = distributed_load.fx if distributed_load.fx is not None
else 0
        fy = distributed_load.fy if distributed_load.fy is not None
else 0
        fz = distributed_load.fz if distributed_load.fz is not None
else 0
```

```
for i in range(no_div+1):
    x = (i / no_div) * length
    internal_forces['fx'][i] -= fx * x
    internal_forces['fy'][i] += fy * x
    internal_forces['fz'][i] += fz * x
```

```

        internal_forces['my'][i] += fz * x ** 2 / 2
        internal_forces['mz'][i] += fy * x ** 2 / 2

    if self.name in loadPattern.point_loads_at_frames:
        for point_load in loadPattern.point_loads_at_frames[self.name]:
            fx = point_load.fx if point_load.fx is not None else (0, 0)
            fy = point_load.fy if point_load.fy is not None else (0, 0)
            fz = point_load.fz if point_load.fz is not None else (0, 0)
            mx = point_load.mx if point_load.mx is not None else (0, 0)
            my = point_load.my if point_load.my is not None else (0, 0)
            mz = point_load.mz if point_load.mz is not None else (0, 0)

            for i in range(no_div+1):
                x = (i / no_div)
                internal_forces['fx'][i] -= fx[0] if x > fx[1] else 0
                internal_forces['fy'][i] += fy[0] if x > fy[1] else 0
                internal_forces['fz'][i] += fz[0] if x > fz[1] else 0
                internal_forces['mx'][i] -= mx[0] if x > mx[1] else 0
                internal_forces['my'][i] += my[0] if x > my[1] else 0
                internal_forces['mz'][i] -= mz[0] if x > mz[1] else 0

                internal_forces['my'][i] += fz[0] * (x - fz[1]) if x > fz
[1] else 0
                internal_forces['mz'][i] += fy[0] * (x - fy[1]) if x > fy
[1] else 0

            internal_forces['fx'] = internal_forces['fx'].tolist()
            internal_forces['fy'] = internal_forces['fy'].tolist()
            internal_forces['fz'] = internal_forces['fz'].tolist()
            internal_forces['mx'] = internal_forces['mx'].tolist()
            internal_forces['my'] = internal_forces['my'].tolist()
            internal_forces['mz'] = internal_forces['mz'].tolist()

    return internal_forces

```

get_internal_displacements()

El método `get_internal_displacements()` de la clase `Frame` permite calcular los desplazamientos internos a lo largo de los elementos aportricados, representados por objetos tipo `Frame`.

En el algoritmo 3.19 se presenta la implementación del método `get_internal_displacements()`. El método recibe como argumentos de entrada el nombre del patrón de

carga y el número de divisiones donde se calcula los desplazamientos, por defecto igual 100. Se genera un diccionario con las entradas `ux`, `uy`, `uz`, `rx`, `ry` y `rz`, las cuales contienen una lista con la respectiva solución a lo largo del elemento.

Los desplazamientos al interior del elemento se calculan con base en las fuerzas en sus extremos, calculados una vez se ha solucionado el modelo, y las diferentes cargas aplicadas en él.

Algoritmo 3.19: Método `get_internal_displacements()` de la clase `Frame`.

```
def get_internal_displacements(self, load_pattern, no_div=100):
    """
    Get internal displacements.

    Parameters
    -----
    load_pattern : str
        Load pattern name.
    np_div : float, optional
        Number divisions.

    Returns
    -----
    internal_displacements : dict
        Internal displacements.
    """
    material = self._parent.materials[self.material]
    section = self._parent.sections[self.section]
    loadPattern = self._parent.load_patterns[load_pattern]
    end_actions = self._parent.end_actions[load_pattern][self.name]
    j_joint_displacement = self._parent.displacements[load_pattern][self.joint_j]

    length = self.get_length()
    E = material.E if material.E is not None else 0
    G = material.G if material.G is not None else 0
    A = section.A if section.A is not None else 0
    J = section.J if section.J is not None else 0
    Iy = section.Iy if section.Iy is not None else 0
    Iz = section.Iz if section.Iz is not None else 0

    end_actions = self._parent.end_actions[load_pattern][self.name]
    fx_j, fy_j, fz_j, mx_j, my_j, mz_j = end_actions.get_end_actions()[:6]
```

```

j_joint_displacmement = self._parent.displacements[load_pattern][self
.joint_j].get_displacements()
j_joint_displacmement = np.dot(np.transpose(self.get_matrix_rotation
())[:6, :6], j_joint_displacmement)
ux_j, uy_j, uz_j, rx_j, ry_j, rz_j = j_joint_displacmement

internal_displacements = {}
internal_displacements['ux'] = np.full(shape=no_div+1, fill_value=
ux_j)
internal_displacements['uy'] = np.full(shape=no_div+1, fill_value=
uy_j)
internal_displacements['uz'] = np.full(shape=no_div+1, fill_value=
uz_j)
internal_displacements['rx'] = np.full(shape=no_div+1, fill_value=
rx_j)
internal_displacements['ry'] = np.full(shape=no_div+1, fill_value=
ry_j)
internal_displacements['rz'] = np.full(shape=no_div+1, fill_value=
rz_j)

for i in range(no_div+1):
    x = (i / no_div) * length
    internal_displacements['ux'][i] -= fx_j * x / (E * A)
    internal_displacements['uy'][i] += fy_j * x ** 3 / (6 * E * Iz)
    internal_displacements['uy'][i] -= mz_j * x ** 2 / (2 * E * Iz)
    internal_displacements['uy'][i] += rz_j * x
    internal_displacements['uz'][i] += fz_j * x ** 3 / (6 * E * Iy)
    internal_displacements['uz'][i] += my_j * x ** 2 / (2 * E * Iy)
    internal_displacements['uz'][i] -= ry_j * x
    internal_displacements['rx'][i] -= mx_j * x / (G * J)
    internal_displacements['ry'][i] -= fz_j * x ** 2 / (2 * E * Iz)
    internal_displacements['ry'][i] -= my_j * x / (E * Iz)
    internal_displacements['rz'][i] += fy_j * x ** 2 / (2 * E * Iz)
    internal_displacements['rz'][i] += mz_j * x / (E * Iy)

if self.name in loadPattern.distributed_loads:
    for distributed_load in loadPattern.distributed_loads[self.name]:
        fx = distributed_load.fx if distributed_load.fx is not None
else 0
        fy = distributed_load.fy if distributed_load.fy is not None
else 0
        fz = distributed_load.fz if distributed_load.fz is not None
else 0

```

```

        for i in range(no_div+1):
            x = (i / no_div) * length
            internal_displacements['ux'][i] -= fx * x ** 2 / (2 * E *
A)
            internal_displacements['uy'][i] += fy * x ** 4 / (24 * E
* Iz)
            internal_displacements['uz'][i] += fz * x ** 4 / (24 * E
* Iy)

            internal_displacements['ry'][i] -= fz * x ** 3 / (6 * E *
Iz)
            internal_displacements['rz'][i] += fy * x ** 3 / (6 * E *
Iz)

    if self.name in loadPattern.point_loads_at_frames:
        for point_load in loadPattern.point_loads_at_frames[self.name]:
            fx = point_load.fx if point_load.fx is not None else (0, 0)
            fy = point_load.fy if point_load.fy is not None else (0, 0)
            fz = point_load.fz if point_load.fz is not None else (0, 0)
            mx = point_load.mx if point_load.mx is not None else (0, 0)
            my = point_load.my if point_load.my is not None else (0, 0)
            mz = point_load.mz if point_load.mz is not None else (0, 0)

            for i in range(no_div+1):
                x = (i / no_div)
                internal_displacements['ux'][i] -= fx[0] * x / (E * A) if
x > fx[1] else 0
                internal_displacements['uy'][i] -= fy[0] * (x ** 3 / 6 -
fy[1] * x ** 2 / 2) / (E * Iz) if x > fy[1] else 0
                internal_displacements['uy'][i] += mz[0] * x ** 2 / (2 *
E * Iz) if x > mz[1] else 0
                internal_displacements['uz'][i] -= fz[0] * (x ** 3 / 6 -
fz[1] * x ** 2 / 2) / (E * Iy) if x > fz[1] else 0
                internal_displacements['uz'][i] -= my[0] * x ** 2 / (2 *
E * Iy) if x > my[1] else 0
                internal_displacements['rx'][i] -= mx[0] * x / (G * J) if
x > fz[1] else 0
                internal_displacements['ry'][i] -= fz[0] * (x ** 2 / 2 -
fy[1] * x) / (E * Iz) if x > fy[1] else 0
                internal_displacements['ry'][i] -= my[0] * x / (E * Iz)
if x > my[1] else 0
                internal_displacements['rz'][i] -= fy[0] * (x ** 2 / 2 -
fy[1] * x) / (E * Iy) if x > fy[1] else 0
                internal_displacements['rz'][i] += mz[0] * x / (E * Iy)
if x > mz[1] else 0

```

```

internal_displacements[ 'ux' ] = internal_displacements[ 'ux' ].tolist()
internal_displacements[ 'uy' ] = internal_displacements[ 'uy' ].tolist()
internal_displacements[ 'uz' ] = internal_displacements[ 'uz' ].tolist()
internal_displacements[ 'rx' ] = internal_displacements[ 'rx' ].tolist()
internal_displacements[ 'ry' ] = internal_displacements[ 'ry' ].tolist()
internal_displacements[ 'rz' ] = internal_displacements[ 'rz' ].tolist()

return internal_displacements

```

3.3.6. Support

La clase **Support** representa los apoyos de la estructura, al establecer los desplazamientos restringidos de los nodos.

En el algoritmo 3.20 se presenta la implementación de la clase **Support**. El constructor de la clase recibe seis argumentos de entrada opcionales, para cada uno de los grados de libertad, los cuales tienen **None** como valor por defecto. El usuario debe indicar qué grados de libertad están restringidos.

La clase **Support** implementa el método **get_restrains()** que genera un arreglo que indica para cada grado de libertad activado si está o no restringido.

Algoritmo 3.20: Clase **Support** implementada en el archivo **primitives.py**.

```

class Support(AttrDisplay):
    """
    Point of support.

    Attributes
    -----
    joint : str
        Joint name.
    ux : bool
        Flag translation along x-axis restraint.
    uy : bool
        Flag translation along y-axis restraint.
    uz : bool
        Flag translation along z-axis restraint.
    rx : bool
        Flag rotation around x-axis restraint.
    ry : bool

```

```

        Flag rotation around y-axis restraint.
    rz : bool
        Flag rotation around z-axis restraint.

    Methods
    -----
    get_restraints()
        Get flags restraints.
    """

def __init__(self, parent, joint, ux=None, uy=None, uz=None, rx=None,
             ry=None, rz=None):
    """
        Instantiate a Support object.

    Parameters
    -----
    parent : Structure
        Structure.
    joint : str
        Joint name.
    ux : bool, optional
        Flag translation along x-axis restraint.
    uy : bool, optional
        Flag translation along y-axis restraint.
    uz : bool, optional
        Flag translation along z-axis restraint.
    rx : bool, optional
        Flag rotation around x-axis restraint.
    ry : bool, optional
        Flag rotation around y-axis restraint.
    rz : bool, optional
        Flag rotation around z-axis restraint.
    """
    self._parent = parent
    self.joint = joint
    self.ux = ux
    self.uy = uy
    self.uz = uz
    self.rx = rx
    self.ry = ry
    self.rz = rz

def get_restraints(self):
    """Get flags restraints"""

```

```

ux = self.ux if self.ux is not None else False
uy = self.uy if self.uy is not None else False
uz = self.uz if self.uz is not None else False
rx = self.rx if self.rx is not None else False
ry = self.ry if self.ry is not None else False
rz = self.rz if self.rz is not None else False

restrains = np.array([ux, uy, uz, rx, ry, rz])

return restrains[self.__parent.
get_flags_active_joint_displacements()]

```

3.3.7. LoadPattern

La clase `LoadPattern` representa los patrones de carga a los que está sometida la estructura, al establecer la magnitud de las fuerzas y las cargas distribuidas que actúan en los nodos y en los elementos aporticados. En la figura 3-8 se presentan los métodos y atributos de esta clase.

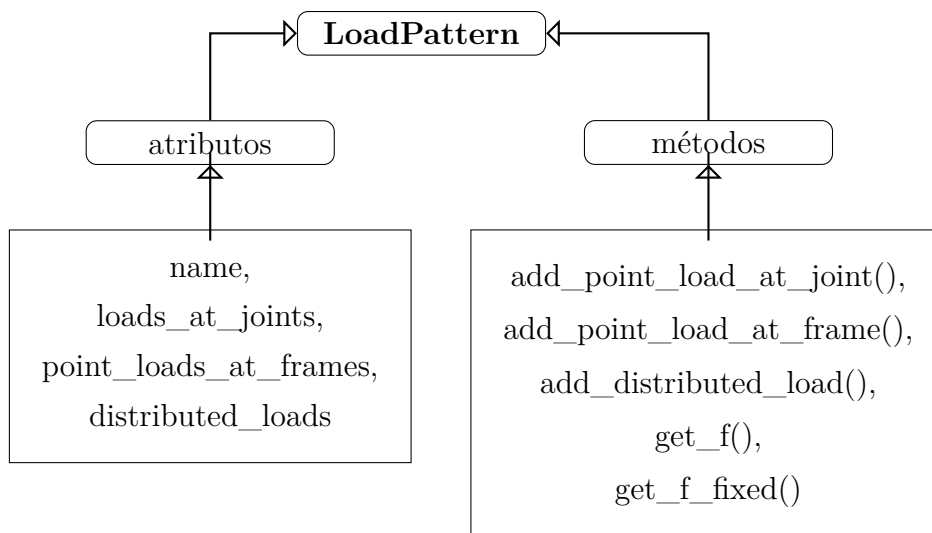


Figura 3-8: Métodos y atributos de la clase `LoadPattern`.

En el algoritmo 3.21 se presenta el constructor de la clase `LoadPattern`. El constructor de la clase no tiene argumentos de entrada opcionales. Sin embargo, asigna un diccionario vacío a los atributos `loads_at_joints`, `point_loads_at_frames` y `distributed_loads`.

Algoritmo 3.21: Constructor de la clase `LoadPattern`.

```
def __init__(self, parent, name):
    """
    Instantiate a LoadPattern object.

    Parameters
    -----
    parent : Structure
        Structure
    name : str
        Load pattern name.
    """
    self.__parent = parent
    self.name = name
    self.loads_at_joints = {}
    self.point_loads_at_frames = {}
    self.distributed_loads = {}
```

A continuación se presentan los métodos de la clase `LoadPattern`, con los cuales se puede, entre otras cosas, calcular el vector de fuerzas en los nodos de la estructura del caso de carga.

`add_point_load_at_joint()`

El método `add_point_load_at_joint()` de la clase `LoadPattern` permite agregar fuerzas en los nodos.

En el algoritmo 3.22 se presenta la implementación del método `add_point_load_at_joint()`. Los argumentos de entrada opcionales `*args` y `**kwargs` son pasados al constructor de la clase `PointLoadAtJoint`, mientras que el argumento `joint` es usado como llave para almacenar el objeto creado en una lista con las demás cargas aplicadas a ese mismo nodo.

Algoritmo 3.22: Método `add_point_load_at_joint()` de la clase `LoadPattern`.

```
def add_point_load_at_joint(self, joint, *args, **kwargs):
    """
    Add a point load at joint.

    Parameters
    -----
    joint : Joint
```

```

        Joint name.

Returns


---


pointLoad : PointLoadAtJoint
        Point load at joint.
    """
    pointLoad = PointLoadAtJoint(self._parent, self.name, joint, *args,
    **kwargs)

    if joint in self.loads_at_joints:
        self.loads_at_joints[joint].append(pointLoad)
    else:
        self.loads_at_joints[joint] = [pointLoad]

    return pointLoad

```

add_point_load_at_frame()

El método `add_point_load_at_frame()` de la clase `LoadPattern` permite agregar fuerzas puntuales en los elementos aporticados.

En el algoritmo 3.23 se presenta la implementación del método `add_point_load_at_frame()`. Los argumentos de entrada opcionales `*args` y `**kwargs` son pasados al constructor de la clase `PointLoadAtFrame`, mientras que el argumento `frame` es usado como llave para almacenar el objeto creado en una lista con las demás cargas puntuales aplicadas a ese elemento aporticado.

Algoritmo 3.23: Método `add_point_load_at_frame()` de la clase `LoadPattern`.

```

def add_point_load_at_frame(self, frame, *args, **kwargs):
    """
        Add a point load at frame.

        Parameters
        

---


frame : Frame
        Frame name.

        Returns
        

---


pointLoad : PointLoadAtFrame
        Point load at frame.
    """

```

```

    pointLoad = PointLoadAtFrame(self._parent, self.name, frame, *args,
**kwargs)

    if frame in self.point_loads_at_frames:
        self.point_loads_at_frames[frame].append(pointLoad)
    else:
        self.point_loads_at_frames[frame] = [pointLoad]

    return pointLoad

```

add_distributed_load()

El método `add_distributed_load()` de la clase `LoadPattern` permite agregar cargas distribuidas en los elementos aporticados.

En el algoritmo 3.24 se presenta la implementación del método `add_distributed_load()`. Los argumentos de entrada opcionales `*args` y `**kwargs` son pasados al constructor de la clase `DistributedLoad`, mientras que el argumento `frame` es usado como llave para almacenar el objeto creado en una lista con las demás cargas distribuidas aplicadas a ese elemento aporticado.

Algoritmo 3.24: Método `add_distributed_load()` de la clase `LoadPattern`.

```

def add_distributed_load(self, frame, *args, **kwargs):
    """
    Add a distributed load at frame.

    Parameters
    -----
    frame : Frame
        Frame name.

    Returns
    -----
    distributedLoad : DistributedLoad
        Distributed load.
    """
    distributedLoad = DistributedLoad(self._parent, self.name, frame, *
args, **kwargs)

    if frame in self.distributed_loads:
        self.distributed_loads[frame].append(distributedLoad)
    else:

```

```

        self.distributed_loads[frame] = [distributedLoad]

    return distributedLoad

```

get_f()

El método `get_f()` de la clase `LoadPattern` calcula el vector de fuerzas total en los nodos de la estructura del caso de carga, representado por objetos tipo `LoadPattern`, con respecto al sistema de coordenadas global.

Según Weaver y Gere (1990), el vector de fuerzas equivalente en los nodos de la estructura A_E debido a las cargas en los elementos aporticados se calcula como

$$\mathbf{A}_E = - \sum_{i=1}^m \mathbf{A}_{MSi} \quad (3-8)$$

donde A_{MSi} es el vector de acciones fijas en los nodos del elemento aporticado i en el sistema de coordenadas global. Este vector de fuerzas equivalentes se suma con el vector de fuerzas aplicadas en los nodos de la estructura para formar el vector de fuerzas total.

En el algoritmo 3.25 se presenta la implementación del método `get_f()`. El vector de fuerzas aplicadas en los nodos de la estructura se ensambla, a partir de las fuerzas aplicadas en cada nodo de la estructura y sus respectivos grados de libertad, con la función `coo_matrix()`.

Algoritmo 3.25: Método `get_f()` de la clase `LoadPattern`.

```

def get_f(self):
    """
    Get the load vector.

    Returns
    -----
    coo_matrix
    Load vector.
    """
    no = self._parent.get_number_active_joint_displacements()
    indexes = self._parent.get_indexes()
    n = no * len(self.loads_at_joints)

    rows = np.empty(n, dtype=int)
    cols = np.zeros(n, dtype=int)

```

```

data = np.zeros(n)

for i, (joint, point_loads) in enumerate(self.loads_at_joints.items()):
    rows[i * no:(i + 1) * no] = indexes[joint]
    for point_load in point_loads:
        data[i * no:(i + 1) * no] += point_load.get_load()

return coo_matrix((data, (rows, cols)), (no * len(self._parent.joints), 1)) - self.get_f_fixed()

```

get_f_fixed()

El método `get_f_fixed()` de la clase `LoadPattern` calcula el vector de fuerzas equivalente en los nodos de la estructura del caso de carga, representado por objetos tipo `LoadPattern`, con respecto al sistema de coordenadas global.

En el algoritmo 3.26 se presenta la implementación del método `get_f_fixed()`. Según los grados de libertad activados se genera el vector de fuerzas equivalentes en los nodos de la estructura del caso de carga. El vector de fuerzas equivalentes en los nodos de la estructura se ensambla, a partir de las cargas aplicadas en los elementos aporticados y sus respectivos grados de libertad, con la función `coo_matrix()`.

Algoritmo 3.26: Método `get_f_fixed()` de la clase `LoadPattern`.

```

def get_f_fixed(self):
    """
    Get f fixed.

    Returns
    -----
    coo_matrix
        f fixed.
    """
    no = self._parent.get_number_active_joint_displacements()
    indexes = self._parent.get_indexes()

    # point loads
    n = 2 * no * len(self.point_loads_at_frames)

    rows = np.empty(n, dtype=int)
    cols = np.zeros(n, dtype=int)
    data = np.zeros(n)

```

```

    for i, (frame, point_loads) in enumerate(self.point_loads_at_frames.
items()):
        frame = self._parent.frames[frame]
        joint_j, joint_k = frame.joint_j, frame.joint_k

        rows[i * 2 * no:(i + 1) * 2 * no] = np.concatenate((indexes[
joint_j], indexes[joint_k]))
        for point_load in point_loads:
            data[i * 2 * no:(i + 1) * 2 * no] += point_load.get_f_fixed().
.flatten()

    point_loads = coo_matrix((data, (rows, cols)), (no * len(self._parent
.joints), 1))

    # distributed loads
    n = 2 * no * len(self.distributed_loads)

    rows = np.empty(n, dtype=int)
    cols = np.zeros(n, dtype=int)
    data = np.zeros(n)

    for i, (frame, distributed_loads) in enumerate(self.distributed_loads
.items()):
        frame = self._parent.frames[frame]
        joint_j, joint_k = frame.joint_j, frame.joint_k

        rows[i * 2 * no:(i + 1) * 2 * no] = np.concatenate((indexes[
joint_j], indexes[joint_k]))
        for distributed_load in distributed_loads:
            data[i * 2 * no:(i + 1) * 2 * no] += distributed_load.
get_f_fixed().flatten()

    distributed_loads = coo_matrix((data, (rows, cols)), (no * len(
indexes), 1))

    return point_loads + distributed_loads

```

3.3.8. PointLoadAtJoint

La clase `PointLoadAtJoint` representa las fuerzas aplicadas en los nodos de la estructura, al establecer el valor de las fuerzas en el sistema de coordenadas global.

En el algoritmo 3.27 se presenta la implementación de la clase `PointLoadAtJoint`. El constructor de la clase recibe seis argumentos de entrada opcionales, para cada uno de los grados de libertad, los cuales tienen `None` como valor por defecto.

La clase `PointLoadAtJoint` implementa el método `get_load()` que genera un *array* que indica, para cada grado de libertad activado, el valor de la fuerza.

Algoritmo 3.27: Clase `PointLoad` implementada en el archivo `primitives.py`.

```
class PointLoadAtJoint(AttrDisplay):
    """
    Point load at joint.

    Attributes
    -----
    load_pattern : str
        Load pattern name.
    joint : str
        Joint name.
    fx : float
        Force along x-axis.
    fy : float
        Force along y-axis.
    fz : float
        Force along z-axis.
    mx : float
        Force around x-axis.
    my : float
        Force around y-axis.
    mz : float
        Force around z-axis.

    Methods
    -----
    get_load()
        Get the load vector.
    """

    def __init__(self, parent, load_pattern, joint, fx=None, fy=None, fz=
None, mx=None, my=None, mz=None):
        """
        Instantiate a PointLoadAtJoint object.

        Parameters
        -----
```

```

    parent : Structure.
        Structure.
    load_pattern : str
        Load pattern name.
    joint : str
        Joint name.
    fx : float
        Force along x-axis, optional.
    fy : float
        Force along y-axis, optional.
    fz : float
        Force along z-axis, optional.
    mx : float
        Force around x-axis, optional.
    my : float
        Force around y-axis, optional.
    mz : float
        Force around z-axis, optional.
    """
    self.__parent = parent
    self.load_pattern = load_pattern
    self.joint = joint
    self.fx = fx
    self.fy = fy
    self.fz = fz
    self.mx = mx
    self.my = my
    self.mz = mz

def get_load(self):
    """Get load"""

    fx = self.fx if self.fx is not None else 0
    fy = self.fy if self.fy is not None else 0
    fz = self.fz if self.fz is not None else 0
    mx = self.mx if self.mx is not None else 0
    my = self.my if self.my is not None else 0
    mz = self.mz if self.mz is not None else 0

    return np.array([fx, fy, fz, mx, my, mz])[self.__parent.
get_flags_active_joint_displacements()]

```

3.3.9. PointLoadAtFrame

La clase `PointLoadAtFrame` representa las fuerzas puntuales aplicadas en los elementos aporticados de la estructura, al establecer el valor de las cargas y su punto de aplicación en el sistema de coordenadas local.

En el algoritmo 3.28 se presenta la implementación de la clase `PointLoadAtFrame`. El constructor de la clase recibe seis argumentos de entrada opcionales, para cada uno de los grados de libertad, los cuales tienen `None` como valor por defecto.

La clase `PointLoadAtFrame` implementa el método `get_f_fixed()` que genera un *array* que indica, para cada grado de libertad activado, las fuerzas equivalentes en los nodos del elemento aporticado en el sistema de coordenadas global.

Algoritmo 3.28: Clase `PointLoadAtFrame` implementada en el archivo `primitives.py`.

```
class PointLoadAtFrame( AttrDisplay ):
    """
    Point load at frame.

    Attributes
    -----
    load_pattern : str
        Load pattern name.
    frame : str
        Frame name.
    fx : tuple
        (value, position).
    fy : tuple
        (value, position).
    fz : tuple
        (value, position).
    mx : tuple
        (value, position).
    my : tuple
        (value, position).
    mz : tuple
        (value, position).
    """

    def __init__( self , parent , load_pattern , frame , fx=None , fy=None , fz=
None , mx=None , my=None , mz=None ):
    """
```

Instantiate a PointLoadAtFrame object.

Parameters

```

parent : Structure
        Structure.
load_pattern : str
        Load pattern name.
frame : str
        Frame name.
fx : tuple, optional
        (value, position).
fy : tuple, optional
        (value, position).
fz : tuple, optional
        (value, position).
mx : tuple, optional
        (value, position).
my : tuple, optional
        (value, position).
mz : tuple, optional
        (value, position).
"""
self._parent = parent
self.load_pattern = load_pattern
self.frame = frame
self.fx = fx
self.fy = fy
self.fz = fz
self.mx = mx
self.my = my
self.mz = mz

def get_f_fixed(self):
    """Get f fixed """
    frame = self._parent.frames[self.frame]
    fx = self.fx if self.fx is not None else (0, 0)
    fy = self.fy if self.fy is not None else (0, 0)
    fz = self.fz if self.fz is not None else (0, 0)
    mx = self.mx if self.mx is not None else (0, 0)
    my = self.my if self.my is not None else (0, 0)
    mz = self.mz if self.mz is not None else (0, 0)
    L = frame.get_length()

    f_local = np.empty((2 * 6, 1))

```

```

# fx
P = fx[0]
a = fx[1] * L
b = L - a

f_local[0] = -P*b / L
f_local[6] = -P*a / L

# fy
P = -fy[0]
a = fy[1] * L
b = L - a

f_local[1] = P*b**2*(3*a+b) / L ** 3
f_local[7] = P*a**2*(a+3*b) / L ** 3

f_local[5] = P*a*b**2 / L**2
f_local[11] = -P*a**2*b / L**2

# fz
P = -fz[0]
a = fz[1] * L
b = L - a

f_local[2] = P*b**2*(3*a+b) / L ** 3
f_local[8] = P*a**2*(a+3*b) / L ** 3

f_local[4] = -P*a*b**2 / L**2
f_local[10] = P*a**2*b / L**2

# mx
T = mx[0]
a = mx[1] * L
b = L - b

f_local[3] = -T*b / L
f_local[9] = -T*a / L

# my
M = my[0]
a = my[1] * L
b = L - a

f_local[2] += -6*M*a*b / L ** 3

```

```

    f_local[8] += 6*M*a*b / L **3

    f_local[4] += M*b*(2*a-b) / L ** 2
    f_local[10] += M*a*(2*b-a) / L ** 2

    # mz
    M = mz[0]
    a = mz[1] * L
    b = L - a

    f_local[1] += 6*M*a*b / L ** 3
    f_local[7] += -6*M*a*b / L ** 3

    f_local[5] += M*b*(2*a-b) / L ** 2
    f_local[11] += M*a*(2*b-a) / L ** 2

    f_local = f_local[np.nonzero(np.tile(flag_joint_displacements, 2)
    )[0], :]

    return np.dot(frame.get_matrix_rotation(), f_local)

```

3.3.10. DistributedLoad

La clase `DistributedLoad` representa las cargas distribuidas aplicadas en los elementos aporticados de la estructura, al establecer el valor de las cargas en el sistema de coordenadas local.

En el algoritmo 3.29 se presenta la implementación de la clase `DistributedLoad`. El constructor de la clase recibe tres argumentos de entrada opcionales, para cada una de las cargas distribuidas a lo largo de los ejes principales del sistema de coordenadas local. El usuario debe indicar el valor de las cargas distribuidas diferentes de cero.

La clase `DistributedLoad` implementa el método `get_f_fixed()` que genera un *array* que indica, para cada grado de libertad activado, las fuerzas equivalentes en los nodos de la estructura en el sistema de coordenadas global.

Algoritmo 3.29: Clase `DistributedLoad` implementada en el archivo `primitives.py`.

```

class DistributedLoad(AttrDisplay):
    """
    Distributed load at frame.

```

Attributes

```

load_pattern : str
    Load pattern name.
frame : str
    Frame name.
system: str
    Coordinate system ('local' by default).
fx : float
    Distributed force along x-axis.
fy : float
    Distributed force along y-axis.
fz : float
    Distributed force along z-axis.

```

Methods

```

get_load()
    Get the load vector.
"""

```

```

def __init__(self, parent, load_pattern, frame, fx=None, fy=None, fz=
None):
    """
    Instantiate a DistributedLoad object.

```

Parameters

```

parent : Structure
    Structure.
load_pattern : str
    Load pattern name.
frame : str
    Frame name.
fx : float, optional
    Distributed force along x-axis.
fy : float, optional
    Distributed force along y-axis.
fz : float, optional
    Distributed force along z-axis.
"""
self.__parent = parent
self.load_pattern = load_pattern
self.frame = frame
self.system = 'local'

```

```

        self.fx = fx
        self.fy = fy
        self.fz = fz

    def get_f_fixed(self):
        """Get f fixed"""
        frame = self._parent.frames[self.frame]
        length = frame.get_length()

        fx = self.fx if self.fx is not None else 0
        fy = self.fy if self.fy is not None else 0
        fz = self.fz if self.fz is not None else 0

        fx_2 = fx * length / 2
        fy_2 = fy * length / 2
        fz_2 = fz * length / 2

        fy_12 = fy * length ** 2 / 12
        fz_12 = fz * length ** 2 / 12

        f_local = np.array([[-fx_2, -fy_2, -fz_2, 0, fz_12, -fy_12, -fx_2,
        , -fy_2, -fz_2, 0, -fz_12, fy_12]]).T

        return np.dot(frame.get_matrix_rotation(), f_local[np.tile(self._parent.get_flags_active_joint_displacements(), 2)])

```

3.3.11. Displacement

La clase **Displacement** representa los desplazamientos de los nodos de la estructura, al establecer el valor de las translaciones y rotaciones en el sistema de coordenadas global.

En el algoritmo 3.30 se presenta la implementación de la clase **Displacement**. El constructor de la clase recibe seis argumentos de entrada opcionales, para cada uno de los posibles desplazamientos de los nodos en el sistema de coordenadas global.

La clase **Displacement** implementa el método **get_displacements()** que genera un *array* que indica, para cada grado de libertad activado, el valor del desplazamiento.

Algoritmo 3.30: Clase **Displacement** implementada en el archivo **primitives.py**.

```

class Displacements(AttrDisplay):
    """

```

Displacements.

Attributes

```
load_pattern : str
    Load pattern.
joint : str
    Joint name.
ux : float
    Translation along x-axis.
uy : float
    Translation along y-axis.
uz : float
    Translation along z-axis.
rx : float
    Rotation around x-axis.
ry : float
    Rotation around y-axis.
rz : float
    Rotation around z-axis.
```

Methods

```
get_displacements()
    Get the displacement vector.
"""
```

```
def __init__(self, parent, load_pattern, joint, ux=None, uy=None, uz=
None, rx=None, ry=None, rz=None):
    """
    Instantiate a Displacements object.
```

Parameters

```
parent : Structure
    Structure.
load_pattern : str
    Load pattern name.
joint : str
    Joint name.
ux : float, optional
    Translation along x-axis.
uy : float, optional
    Translation along y-axis.
uz : float, optional
```

```

        Translation along z-axis.
    rx : float , optional
        Rotation around x-axis.
    ry : float , optional
        Rotation around y-axis.
    rz : float , optional
        Rotation around z-axis.
    """
    self._parent = parent
    self.load_pattern = load_pattern
    self.joint = joint
    self.ux = ux
    self.uy = uy
    self.uz = uz
    self.rx = rx
    self.ry = ry
    self.rz = rz

    def get_displacement(self):
        """Get displacements"""
        return np.array([self.ux, self.uy, self.uz, self.rx, self.ry,
self.rz])[self._parent.get_flags_active_joint_displacements()]

```

3.3.12. EndActions

La clase **EndActions** representa las fuerzas en los extremos de los elementos aporticados de la estructura, al establecer el valor de las fuerzas en el sistema de coordenadas local.

En el algoritmo 3.31 se presenta la implementación de la clase **EndActions**. El constructor de la clase recibe doce argumentos de entrada opcionales, para cada una de las posibles fuerzas en los extremos de los elementos aporticados.

Algoritmo 3.31: Clase **EndActions** implementada en el archivo **primitives.py**.

```

class EndActions(AttrDisplay):
    """
    End actions.

    Attributes
    -----
    load_pattern : str
        Load pattern name.
    frame : str

```

```

        Frame name.
    fx_j : float
        Force along x-axis at near joint.
    fy_j : float
        Force along y-axis at near joint.
    fz_j : float
        Force along z-axis at near joint.
    mx_j : float
        Moment around x-axis at near joint.
    my_j : float
        Moment around y-axis at near joint.
    mz_j : float
        Moment around z-axis at near joint.
    fx_k : float
        Force along x-axis at far joint.
    fy_k : float
        Force along y-axis at far joint.
    fz_k : float
        Force along z-axis at far joint.
    mx_k : float
        Moment around x-axis at far joint.
    my_k : float
        Moment around y-axis at far joint.
    mz_k : float
        Moment around z-axis at far joint.

    Methods
    -----
    get_end_actions()
        Get the end actions vector.
    """

def __init__(self, parent, load_pattern, frame, fx_j=None, fy_j=None,
    fz_j=None, mx_j=None, my_j=None, mz_j=None, fx_k=None, fy_k=None,
    fz_k=None, mx_k=None, my_k=None, mz_k=None):
    """
        Instantiate a EndActions object.

    Parameters
    -----
    parent : Structure
        Structure
    load_pattern : str
        Load pattern name.
    frame : str

```

```

        Frame name.
        fx_j : float , optional
            Force along x-axis at joint_j.
        fy_j : float , optional
            Force along y-axis at joint_j.
        fz_j : float , optional
            Force along z-axis at joint_j.
        mx_j : float , optional
            Moment around xaxis at joint_j.
        my_j : float , optional
            Moment around y-axis at joint_j.
        mz_j : float , optional
            Moment around z-axis at joint_j.
        fx_k : float , optional
            Force along x-axis at joint_k.
        fy_k : float , optional
            Force along y-axis at joint_k.
        fz_k : float , optional
            Force along z-axis at joint_k.
        mx_k : float , optional
            Moment around x-axis at joint_k.
        my_k : float , optional
            Moment around y-axis at joint_k.
        mz_k : float , optional
            Moment around z-axis at joint_k.
        """
        self.__parent = parent
        self.frame = frame
        self.load_pattern = load_pattern
        self.fx_j = fx_j
        self.fy_j = fy_j
        self.fz_j = fz_j
        self.mx_j = mx_j
        self.my_j = my_j
        self.mz_j = mz_j
        self.fx_k = fx_k
        self.fy_k = fy_k
        self.fz_k = fz_k
        self.mx_k = mx_k
        self.my_k = my_k
        self.mz_k = mz_k

def get_end_actions(self):
    """Get end actions """

```

```

    return np.array([self.fx_j, self.fy_j, self.fz_j, self.mx_j, self
        .my_j, self.mz_j, self.fx_k, self.fy_k, self.fz_k, self.mx_k, self
        .my_k, self.mz_k])[np.tile(self._parent.
        get_flags_active_joint_displacements(), 2)]

```

3.3.13. Reaction

La clase **Reaction** representa las reacciones de los apoyos de la estructura, al establecer el valor de las reacciones en el sistema de coordenadas global.

En el algoritmo 3.32 se presenta la implementación de la clase **Reaction**. El constructor de la clase recibe seis argumentos de entrada opcionales, para cada una de las posibles reacciones en el sistema de coordenadas global.

La clase **Reaction** implementa el método **get_reactions()** que genera un *array* que indica, para cada grado de libertad activado, el valor de la reacción.

Algoritmo 3.32: Clase **Reaction** implementada en el archivo **primitives.py**.

```

class Reaction(AttrDisplay):
    """
    Reaction.

    Attributes
    -----
    load_pattern : str
        Load pattern name.
    joint : str
        Joint name.
    fx : float
        Force along x-axis.
    fy : float
        Force along y-axis.
    fz : float
        Force along z-axis.
    mx : float
        Moment around x-axis.
    my : float
        Moment around y-axis.
    mz : float
        Moment around z-axis.

```

Methods

```

get_reactions()
    Get the load vector.
    """

```

```

def __init__(self, parent, load_pattern, joint, fx=None, fy=None, fz=
None, mx=None, my=None, mz=None):
    """

```

```

    Instantiate a Reaction.

```

Parameters

```

parent : Structure
    Structure.
load_pattern : str
    Load pattern name.
joint : str
    Joint name.
fx : float, optional
    Force along 'x' axis.
fy : float, optional
    Force along 'y' axis.
fz : float, optional
    Force along 'z' axis.
mx : float, optional
    Moment around 'x' axis.
my : float, optional
    Moment around 'y' axis.
mz : float, optional
    Moment around 'z' axis.
    """

self.__parent = parent
self.load_pattern = load_pattern
self.joint = joint
self.fx = fx
self.fy = fy
self.fz = fz
self.mx = mx
self.my = my
self.mz = mz

```

```

def get_reactions(self):
    """Get reactions"""

```

```

    return np.array([self.fx, self.fy, self.fz, self.mx, self.my,
self.mz])[self._parent.get_flags_active_joint_displacements()]

```

3.3.14. InternalForces

La clase **InternalForces** representa las fuerzas internas de los elementos aporticados, al establecer el valor de las fuerzas internas a lo largo del elemento en el sistema de coordenadas local.

En el algoritmo 3.33 se presenta la implementación de la clase **InternalForces**. El constructor de la clase recibe seis argumentos de entrada opcionales, para cada una de las fuerzas internas de los elementos aporticados. Cada uno de estos argumento de entrada opcional es una lista con los valores de una de las fuerzas internas, tomados en puntos uniformemente distribuidos a lo largo del elemento aporticado.

Algoritmo 3.33: Clase **InternalForces** implementada en el archivo `primitives.py`.

```

class InternalForces(AttrDisplay):
    """
    Internal forces.

    Attributes
    -----
    load_pattern : str
        Load pattern name.
    frame : str
        Frame name.
    fx : list, optional
        Internal forces along x-axis.
    fy : list, optional
        Internal forces along y-axis.
    fz : list, optional
        Internal forces along z-axis.
    mx : list, optional
        Internal forces around x-axis.
    my : list, optional
        Internal forces around y-axis.
    mz : list, optional
        Internal forces around z-axis.
    """

    def __init__(self, parent, load_pattern, frame, fx=None, fy=None, fz=
None, mx=None, my=None, mz=None):

```

```

"""
    Instantiate a InternalForces object.

    Parameters
    -----
    parent : Structure
        Structure.
    load_pattern : str
        Load pattern name
    frame : str
        Frame name.
    fx : list, optional
        Internal forces along x-axis.
    fy : list, optional
        Internal forces along y-axis.
    fz : list, optional
        Internal forces along z-axis.
    mx : list, optional
        Internal forces around x-axis.
    my : list, optional
        Internal forces around y-axis.
    mz : list, optional
        Internal forces around z-axis.
"""
self.__parent = parent
self.load_pattern = load_pattern
self.frame = frame
self.fx = fx
self.fy = fy
self.fz = fz
self.mx = mx
self.my = my
self.mz = mz

```

3.3.15. InternalDisplacements

La clase **InternalDisplacements** representa los desplazamientos internos de los elementos aporticados, al establecer el valor de los desplazamientos a lo largo del elemento en el sistema de coordenadas local.

En el algoritmo 3.34 se presenta la implementación de la clase **InternalDisplacements**. El constructor de la clase recibe seis argumentos de entrada opcionales, para cada uno de los desplazamientos a lo largo de los elementos aporticados. Cada uno de estos argu-

mentos de entrada opcionales es una lista con los valores de una de las fuerzas internas, tomados en puntos uniformemente distribuidos a lo largo del elemento aporticado.

Algoritmo 3.34: Clase `InternalDisplacements` implementada en el archivo `primitives.py`.

```
class InternalDisplacements(AttrDisplay):
    """
    Internal displacement.

    Attributes
    -----
    load_pattern : str
        Load pattern name.
    frame : str
        Frame name.
    ux : list, optional
        Internal displacements along x-axis.
    uy : list, optional
        Internal displacements along y-axis.
    uz : list, optional
        Internal displacements along z-axis.
    rx : list, optional
        Internal displacements around x-axis.
    ry : list, optional
        Internal displacements around y-axis.
    rz : list, optional
        Internal displacements around z-axis.
    """

    def __init__(self, parent, load_pattern, frame, ux=None, uy=None, uz=
None, rx=None, ry=None, rz=None):
        """
        Instantiate a InternalDisplacements object.

        Parameters
        -----
        parent : Structure
            Structure
        load_pattern : str
            Load pattern name.
        frame : str
            Frame name.
        ux : list, optional
            Internal displacements along x-axis.
```

```

    uy : list , optional
        Internal displacements along y-axis.
    uz : list , optional
        Internal displacements along z-axis.
    rx : list , optional
        Internal displacements around x-axis.
    ry : list , optional
        Internal displacements around y-axis.
    rz : list , optional
        Internal displacements around z-axis.
    """
    self.__parent = parent
    self.load_pattern = load_pattern
    self.frame = frame
    self.ux = ux
    self.uy = uy
    self.uz = uz
    self.rx = rx
    self.ry = ry
    self.rz = rz

```

3.4. Clase Structure (segunda parte)

La clase **Structure** representa el modelo de estructuras aporticadas tridimensionales sometidas a cargas estáticas, al agregar objetos que describen la geometría de la estructura, sus condiciones de apoyo y las sollicitaciones externas. En la figura **3-9** se presentan los métodos y atributos de esta clase.

Como se mencionó anteriormente, el constructor de la clase recibe seis argumentos de entrada opcionales, uno para cada grado de libertad, los cuales tienen **None** como valor por defecto. Cuando el usuario crea un objeto de esta clase debe indicar qué grados de libertad quiere tener en cuenta para analizar el modelo (véase el algoritmo 3.2).

Con los métodos `add_material()`, `add_section()`, `add_rectangular_section()`, `add_joint()`, `add_frame()` y `add_support()` se pueden agregar objetos tipo **Material**, **Section**, **RectangularSection**, **Joint**, **Frame** y **Support**, respectivamente, para describir la geometría y condiciones de apoyo de la estructura. Con los métodos `add_load_pattern()`, `add_load_at_joint()`, `add_point_load_at_frame()` y `add_distributed_load()` se pueden agregar objetos tipo **LoadPattern**, **PointLoadAtJoin**, **PointLoadAtFrame** y **DistributedLoad**, respectivamente, para describir las cargas de los patrones

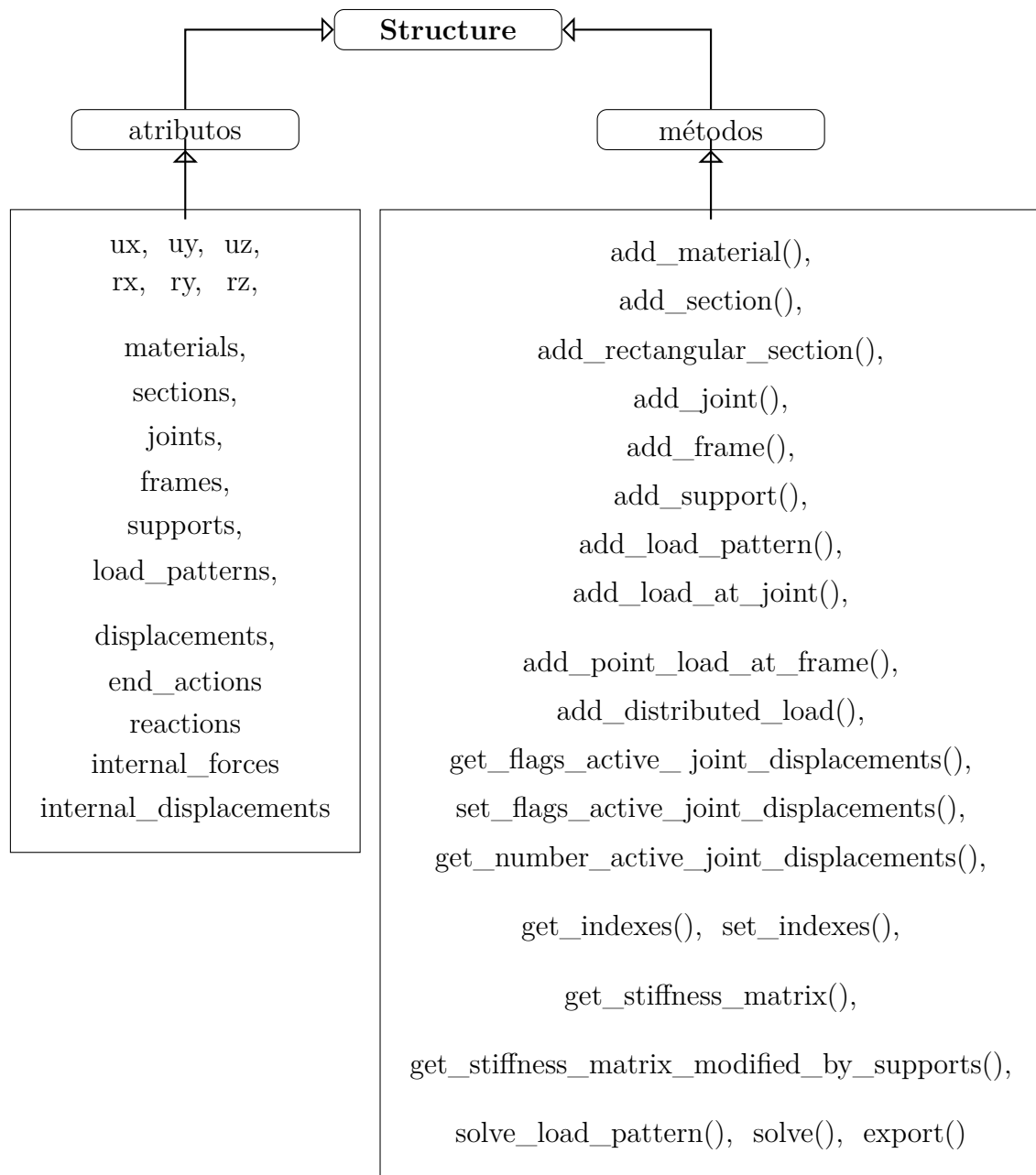


Figura 3-9: Métodos y atributos de la clase **Structure**.

de carga a los que se encuentra sometida la estructura.

A continuación se presentan los demás métodos de la clase **Structure**, con los cuales se puede, entre otras cosas, analizar linealmente el modelo para encontrar los despla-

mientos y reacciones de la estructura sometida a los diferentes patrones de carga.

3.4.1. set_flags_active_joint_displacements

El método `set_flags_active_joint_displacements()`, de la clase `Structure`, almacena un arreglo en el atributo `_active_joint_displacements` que indica para cada grado de libertad si está o no activado.

Algoritmo 3.35: Método `set_flags_active_joint_displacements()` implementado en el archivo `core.py`.

```
def set_flags_active_joint_displacements(self):
    """Set flags active joint displacements"""

    ux = self.ux if self.ux is not None else False
    uy = self.uy if self.uy is not None else False
    uz = self.uz if self.uz is not None else False
    rx = self.rx if self.rx is not None else False
    ry = self.ry if self.ry is not None else False
    rz = self.rz if self.rz is not None else False

    self._active_joint_displacements = np.array([ux, uy, uz, rx, ry, rz])
```

3.4.2. get_flags_active_joint_displacements()

El método `get_flags_active_joint_displacements()` de la clase `Structure` genera un arreglo que indica para cada grado de libertad si está o no activado.

En el algoritmo 3.36 se presenta la implementación del método `get_flags_active_joint_displacements()`. El método genera el arreglo almacenado en el atributo `_active_joint_displacements`.

Algoritmo 3.36: Método `get_flag_active_joint_displacements()` de la clase `Structure`.

```
def get_flags_active_joint_displacements(self):
    """Get flags active joint displacements.

    Returns
    -----
    array
```

```

        Flags active joint displacements.
        """

    return self._active_joint_displacements

```

3.4.3. `get_number_active_joint_displacements()`

El método `get_number_active_joint_displacements()` de la clase `Structure` calcula el número de grados de libertad activados.

En el algoritmo 3.37 se presenta la implementación del método `get_number_active_joint_displacements()`. El método calcula la cantidad de entradas iguales a `True` del *array* generado por el método `get_flags_active_joint_displacements()` (véase el algoritmo 3.36).

Algoritmo 3.37: Método `get_number_active_joint_displacements()` de la clase `Structure`.

```

def get_number_active_joint_displacements(self):
    """
        Get number active joint displacements.

        Returns
        _____
        int
        Number of active joint displacements.
        """
    return np.count_nonzero(self.get_flags_active_joint_displacements())

```

3.4.4. `set_indexes()`

El método `set_indexes()`, de la clase `Structure`, almacena un diccionario en el atributo `_indexes`, donde las llaves son los nodos de la estructura y los valores los respectivos grados de libertad.

En el algoritmo 3.38 se presenta la implementación del método `set_indexes()`. El método crea un diccionario donde las llaves son los nombres de los objetos tipo `Joint` del diccionario `joints` y los valores *arrays* con los respectivos grados de libertad.

Los grados de libertad de cada nodo de la estructura se asignan de manera secuencial en función de la cantidad de grados de libertad activados, calculada con el método `get_number_active_joint_displacements()` (véase el algoritmo 3.37). Al primer nodo se le asignan los primeros n índices, comenzando en cero, al segundo nodo los siguientes n índices y así sucesivamente para cada uno de los demás nodos.

Algoritmo 3.38: Método `set_indexes()` de la clase `Structure`.

```
def set_indexes(self):
    """Set joint indexes"""
    n = self.get_number_active_joint_displacements()

    self._indexes = {joint: np.arange(n * i, n * (i + 1)) for i, joint in
                     enumerate(self.joints)}
```

3.4.5. `get_indexes()`

El método `get_indexes()` de la clase `Structure` genera un *diccionario* donde las llaves son los nodos de la estructura y los valores los respectivos grados de libertad.

En el algoritmo 3.39 se presenta la implementación del método `get_indexes()`. El método genera el diccionario almacenado en el atributo `_indexes`.

Algoritmo 3.39: Método `get_indexes()` de la clase `Structure`.

```
def get_indexes(self):
    """Get joint indexes.

    Returns
    -----
    dict
        Key value pairs joints and indexes.
    """
    return self._indexes
```

3.4.6. `get_stiffness_matrix()`

El método `get_stiffness_matrix()` de la clase `Structure` permite calcular la matriz de rigidez de la estructura.

En el algoritmo 3.40 se presenta la implementación del método `get_stiffness_matrix()`. Según los grados de libertad de los nodos de la estructura se ensamblan las matrices

de rigidez de los elementos aporticados con la función `coo_matrix()`. Para cada objeto tipo **Frame** del diccionario `frames` se calcula su matriz de rigidez en el sistema de coordenadas global, mediante el método `get_global_stiffness_matrix()` (véase el algoritmo 3.17), y se extraen los grados de libertad de los nodos del elemento aporticado del diccionario `indexes` (véase el algoritmo 3.39).

Con estas variables se describe la matriz de rigidez en coordenadas globales como una matriz dispersa en el formato *ijv*. Los índices de las filas y las columnas se almacenan en los arreglos `rows` y `cols`, respectivamente, mientras que los valores de la matriz se almacenan en el arreglo `data`.

Finalmente, se genera la matriz de rigidez de la estructura indicando que se trata de una matriz cuadrada de tamaño del número de grados de libertad activados por la cantidad de nodos de la estructura.

Algoritmo 3.40: Método `get_stiffness_matrix()` de la clase **Structure**.

```
def get_stiffness_matrix(self):
    """
    Get stiffness matrix of the structure.

    Returns
    -----
    array
        Stiffness matrix of the structure.
    """
    flags_active_joint_displacements = self.
    get_flags_active_joint_displacements()
    no_active_joint_displacements = self.
    get_number_active_joint_displacements()
    indexes = self.get_indexes()

    no_joints = len(self.joints)
    no_frames = len(self.frames)

    n = 2 * no_active_joint_displacements
    n_2 = n ** 2

    rows = np.empty(no_frames * n_2, dtype=int)
    cols = np.empty(no_frames * n_2, dtype=int)
    data = np.empty(no_frames * n_2)

    for i, frame in enumerate(self.frames.values()):
```

```

        k_element = frame.get_global_stiffness_matrix()
        indexes_element = np.concatenate((indexes[frame.joint_j], indexes
[frame.joint_k]))
        indexes_element = np.broadcast_to(indexes_element, (n, n))

        rows[i * n_2:(i + 1) * n_2] = indexes_element.flatten('F')
        cols[i * n_2:(i + 1) * n_2] = indexes_element.flatten()
        data[i * n_2:(i + 1) * n_2] = k_element.flatten()

    return coo_matrix((data, (rows, cols)), 2 * (
no_active_joint_displacements * no_joints,)).toarray()

```

3.4.7. set_stiffness_matrix_with_support()

El método `set_stiffness_matrix_with_support()` de la clase `Structure` modifica la matriz de rigidez de la estructura, calculada con el método `get_stiffness_matrix()` (véase el algoritmo 3.40), para tener en cuenta las condiciones de apoyo, y la almacena en el atributo `_k_support`.

Según Reddy (1993), para tener en cuenta las condiciones de apoyo de la estructura en la matriz de rigidez, se deben reemplazar los valores de las filas y las columnas asociadas a los grados de libertad restringidos por ceros, a excepción de los valores en la diagonal principal, los cuales deben ser reemplazados por 1.

En el algoritmo 3.41 se presenta la implementación del método `set_stiffness_matrix_with_support()`. Para cada objeto tipo `Support` del diccionario `supports` se extraen los grados de libertad del diccionario `indexes` y se calculan las restricciones del apoyo con el método `get_restrains()` (véase el algoritmo 3.20). Estos valores se almacenan en las variables `joint_indexes` y `restrains` respectivamente.

Finalmente, para cada grado de libertad restringido se reemplazan los valores asociados de la fila y la columna de la matriz de rigidez de la estructura por ceros y el valor en la diagonal principal por 1.

Algoritmo 3.41: Método `set_stiffness_matrix_with_support()` de la clase `Structure`.

```

def set_stiffness_matrix_modified_by_supports(self):
    """Set stiffness matrix of the structure modified by supports"""

```

```

        flags_active_joint_displacements = self.
get_flags_active_joint_displacements()
        indexes = self.get_indexes()

        stiffness_matrix = self.get_stiffness_matrix()
        n = np.shape(stiffness_matrix)[0]

        for joint, support in self.supports.items():
            joint_indexes = indexes[joint]
            restraints = support.get_restraints()

            for index in joint_indexes[restraints]:
                stiffness_matrix[index] = stiffness_matrix[:, index] = np
                .zeros(n)
                stiffness_matrix[index, index] = 1

        self._ksupport = stiffness_matrix

```

3.4.8. **get_stiffness_matrix_with_supports()**

El método `get_stiffness_matrix_with_supports()` de la clase **Structure** genera la matriz de rigidez de la estructura modificada por las condiciones de apoyo.

En el algoritmo 3.42 se presenta la implementación del método `get_stiffness_matrix_with_supports()`. El método genera el arreglo almacenado en el atributo `_k_support`.

Algoritmo 3.42: Método `get_stiffness_matrix_with_support()` de la clase **Structure**.

```

def get_stiffness_matrix_modified_by_supports(self):
    """Get stiffness matrix of the structure modified by supports.

    Returns
    -----
    stiffness_matrix_with_supports : array
        Stiffness matrix of the structure modified by supports.
    """
    return self._ksupport

```

3.4.9. solve_load_pattern()

El método `solve_load_pattern()` de la clase `Structure` calcula los desplazamientos de los nodos de la estructura, las fuerzas en los extremos de los elementos aporticados, las reacciones en los apoyos y las fuerzas internas, debidos a un patrón de carga.

En el algoritmo 3.43 se presenta la implementación del método `solve_load_pattern()`. El método recibe como argumento de entrada el nombre del patrón de carga, representado por objetos tipo `LoadPattern` (véase el algoritmo 3.21).

Según Reddy (1993), para tener en cuenta las condiciones de apoyo de la estructura en el vector de fuerzas totales en los nodos, se deben reemplazar los valores asociados a los grados de libertad restringidos por cero.

El vector de fuerzas totales en los nodos de la estructura del caso de carga se calcula con el método `get_f()` (véase el algoritmo 3.25). Para cada objeto tipo `Support` del diccionario `supports` se extraen los respectivos grados de libertad del diccionario `indexes` y se calculan las restricciones del apoyo con el método `get_restrains()` (véase el algoritmo 3.20), para reemplazar los valores asociados a los grados de libertad restringidos del vector de fuerzas en los nodos de la estructura por cero.

Una vez modificado el vector de fuerzas totales en los nodos, se calculan los desplazamientos de los nodos de la estructura, usando la matriz de la estructura modificada por las condiciones de apoyo, calculada con el método `get_stiffness_matrix_modified_by_supports()` (véase el algoritmo 3.42). Calculados los desplazamientos, se calculan las fuerzas en los extremos de los elementos, las reacciones en los apoyos y las fuerzas al interior de los elementos.

Algoritmo 3.43: Método `solve_load_pattern()` de la clase `Structure`.

```
def solve_load_pattern(self, load_pattern):
    """
    Solve the structure due to a load pattern.

    Parameters
    -----
    load_pattern : str
        Load pattern name.
    """
```

```

    flags_joint_displacements = self.get_flags_active_joint_displacements
    ()
    indexes = self.get_indexes()
    loadPattern = self.load_patterns[load_pattern]

    f = loadPattern.get_f().toarray()
    f_support = np.copy(f)

    for joint, support in self.supports.items():
        joint_indexes = indexes[joint]
        restrains = support.get_restrains()

        for index in joint_indexes[restrains]:
            f_support[index, 0] = 0

    # find displacements
    u = np.linalg.solve(self.get_stiffness_matrix_modified_by_supports(),
        f_support)

    # store displacements
    load_pattern_displacements = {}

    for joint in self.joints:
        joint_indexes = indexes[joint]
        displacements = np.array(6 * [None])
        displacements[flags_joint_displacements] = u[joint_indexes, 0]
        load_pattern_displacements[joint] = Displacements(self,
load_pattern, joint, *displacements)

    self.displacements[load_pattern] = load_pattern_displacements

    # store frame end actions
    flags_frame_displacements = np.tile(flags_joint_displacements, 2)
    n = np.count_nonzero(flags_frame_displacements)

    rows = []
    cols = []
    data = []

    load_pattern_end_actions = {}

    for frame, _frame in self.frames.items():
        t = _frame.get_matrix_rotation()
        indexes_element = np.concatenate((indexes[_frame.joint_j],
indexes[_frame.joint_k]))

```

```

k_element = _frame.get_local_stiffness_matrix()
u_element = np.dot(np.transpose(t), u[indexes_element])

f_fixed = np.zeros((n, 1))

if frame in loadPattern.point_loads_at_frames:
    for point_load in loadPattern.point_loads_at_frames[frame]:
        f_fixed += point_load.get_f_fixed()

if frame in loadPattern.distributed_loads:
    for distributed_load in loadPattern.distributed_loads[frame]:
        f_fixed += distributed_load.get_f_fixed()

f_end_actions = np.array(12*[None])
f_end_actions[flags_frame_displacements] = np.dot(k_element,
u_element).flatten() + np.dot(np.transpose(t), f_fixed).flatten()
load_pattern_end_actions[frame] = EndActions(self, load_pattern,
frame, *f_end_actions)

# reactions
if any(joint in self.supports for joint in (_frame.joint_k,
_frame.joint_j)):
    rows.extend(list(indexes_element))
    cols.extend(n * [0])
    data.extend(list(np.dot(t, load_pattern_end_actions[frame].
get_end_actions()).flatten()))

self.end_actions[load_pattern] = load_pattern_end_actions

# store reactions
number_joints = len(self.joints)
n = np.count_nonzero(flags_frame_displacements)

f += loadPattern.get_f_fixed().toarray()
f_end_actions = coo_matrix((data, (rows, cols)), (number_joints * n,
1)).toarray()

load_pattern_reactions = {}

for joint in self.supports:
    joint_indexes = indexes[joint]
    _restrains = self.supports[joint].get_restraints()
    restrains = np.array(6 * [False])
    restrains[flags_joint_displacements] = _restrains

```

```

        reactions = np.array(6 * [None])
        reactions[flags_joint_displacements & restrains] = f_end_actions[
joint_indexes, 0][_restrains] - f[joint_indexes, 0][_restrains]
        load_pattern_reactions[joint] = Reaction(self, load_pattern,
joint, *reactions)

        self.reactions[load_pattern] = load_pattern_reactions

        # store internal forces
        load_pattern_internal_forces = {}

        for frame, _frame in self.frames.items():
            load_pattern_internal_forces[frame] = InternalForces(self,
load_pattern, frame, **_frame.get_internal_forces(load_pattern))

        self.internal_forces[load_pattern] = load_pattern_internal_forces

```

3.4.10. solve()

El método `solve()` de la clase **Structure** analiza el modelo de la estructura sometida a los diferentes patrones de carga y almacena los resultados en los diccionarios `displacements`, `end_actions`, `reactions`, `internal_forces` y `internal_displacements`.

En el algoritmo 3.44 se presenta la implementación del método `solve()`. El método calcula el arreglo que indica los grados de libertad activados, con el método `set_flags_active_joint_displacement()` (véase el algoritmo 3.35), el diccionario que relaciona los nodos de la estructura con sus respectivos grados de libertad, con el método `set_indexes()` (véase el algoritmo 3.38), y la matriz de rigidez modificada por las condiciones de apoyo, con el método `get_stiffness_matrix_with_supports()` (véase el algoritmo 3.42).

Para cada patrón de carga se analiza la estructura mediante el método `solve_load_pattern()` (véase el algoritmo 3.43).

Algoritmo 3.44: Método `solve()` de la clase **Structure**.

```

def solve(self):
    """Solve the structure"""
    self.set_flags_active_joint_displacements()
    self.set_indexes()

```

```
self.set_stiffness_matrix_modified_by_supports()

for load_pattern in self.load_patterns:
    self.solve_load_pattern(load_pattern)
```

3.4.11. export()

El método `export()` de la clase `Structure` genera un archivo de texto en formato JSON con la descripción del modelo para ser interpretado por el programa de computador *FEM.js*.

El método almacena los objetos que representan los materiales, las secciones transversales, los nodos, los elementos aporticados, las condiciones de apoyo y los patrones de carga, con sus respectivas cargas, en las entradas `materials`, `sections`, `joints`, `frames`, `supports` y `load_patterns`, respectivamente, usando las mismas llaves con las que fueron agregados al modelo.

A continuación se presenta la estructura general que tiene un archivo generado por este método.

```
{
  "materials": {
    "key_material": {
      "E": 0.0,
      "G": 0.0
    },
    ...
  },
  "sections": {
    "key_section": {
      "area": 0.0,
      "Ix": 0.0,
      "Iy": 0.0,
      "Iz": 0.0,
      "type": "Section"
    },
    "another_key": {
      "area": 0.0,
      "Ix": 0.0,
      "Iy": 0.0,
      "Iz": 0.0,
```

```

        "type": "RectangularSection",
        width: 0.0,
        height: 0.0
    },
    ...
},
"joints": {
    "key": {
        "x": 0.0,
        "y": 0.0,
        "z": 0.0
    },
    ...
},
"frames": {
    "key": {
        "j": "key_joint",
        "k": "another_key_joint",
        "material": "key_material",
        "section": "key_section"
    },
    ...
},
"supports": {
    "key_joint": {
        "ux": bool,
        "uy": bool,
        "uz": bool,
        "rx": bool,
        "ry": bool,
        "rz": bool
    },
    ...
},
"load_patterns": {
    "key_load_pattern": {
        "joints": {
            "key_joint": [
                {
                    "fx": 0.0,
                    "fy": 0.0,
                    "fz": 0.0,
                    "mx": 0.0,
                    "my": 0.0,
                    "mz": 0.0
                }
            ]
        }
    }
}

```

}

Las clases presentadas hasta aquí permiten analizar linealmente estructuras aporticadas tridimensionales sometidas a cargas estáticas. Adicional a estas clases, en el archivo `classtools.py` se desarrolló la clase `AttrDisplay`, la cual es heredada por las demás clases.

La clase `AttrDisplay` implementa una representación más cómoda de los objetos al redefinir el método `__repr__()`, lo cual permite mostrar en pantalla información de los objetos con un formato amigable.

Algoritmo 3.45: Clase `AttrDisplay` implementada en el archivo `classtools.py`.

```
class AttrDisplay:
    def __repr__(self):
        """
        Get representation object.

        Returns
        -----
        str
            Object representation.
        """
        return "{}".format({key: value for key, value in self.__dict__.
items() if not key.startswith('_') and value is not None})
```

4 Programa de pre y posproceso mas.js

Todas las implementaciones del método directo de rigideces, en algún momento de su desarrollo, se ven enfrentadas al problema de comunicarse con sus usuarios. Inicialmente, los ingenieros describían sus modelos y recibían los resultados del análisis en archivos de texto. Sin embargo, esta forma de interactuar con los programas de computador, aunque fue ampliamente adoptada por necesidad, no resultó atractiva para todos.

Con la aparición de los monitores, el principal dispositivo de salida de los computadores, se comenzaron a desarrollar interfaces encargadas del pre y posprocesamiento de la solución, las cuales gozan de mayor aceptación entre el público en general. Estas interfaces presentan la información del modelo y los resultados del análisis en una escena tridimensional, permitiéndole a sus usuarios interactuar fácilmente con él.

Actualmente, el desarrollo web sugiere grandes ventajas, principalmente por la posibilidad de ejecutar programas de computador a través del navegador de internet, lo cual abre la posibilidad del trabajo colaborativo y computación en la nube. Por ello, para este trabajo se desarrolló **mas.js**, un programa de computador donde los usuarios pueden modelar estructuras aporticadas sometidas a cargas estáticas y ver los resultados del análisis directamente en el navegador de internet.

4.1. Generalidades de mas.js

mas.js es una aplicación web desarrollada con Three.js (una *API* de JavaScript para crear escenas tridimensionales en el navegador de internet) para modelar estructuras aporticadas sometidas a cargas estáticas. Una copia del programa se encuentra alojada en la página web de GitHub <https://github.com/rvcristiand/mas.js>.

4.1.1. Arquitectura del programa

En la figura 5-3 se presenta la estructura de los principales archivos del programa.

```
mas.js/  
├── LICENSE  
├── README.md  
├── css/  
│   └── style.css  
├── space_frame.json  
├── index.html  
├── libs/  
│   ├── CSS2DRenderer.js  
│   ├── OrbitControls.js  
│   ├── Projector.js  
│   ├── dat.gui.min.js  
│   ├── stats.js  
│   └── three.js  
├── main.js  
└── modules/  
    ├── mas.js  
    └── terminal.js
```

Figura 4-1: Estructura de los archivos del programa mas.js.

Igual que para el programa **pymas**, la carpeta de **mas.js** contiene algunos archivos informativos, los cuales se utilizan principalmente para tener una idea general del proyecto y para orientar a los colaboradores.

- El archivo **LICENSE** contiene información sobre la licencia del programa, la cual establece lo que pueden realizar los usuarios con el código descargado. En virtud de proporcionar la mayor libertad de uso, lectura, modificación y distribución del programa, **mas.js** tiene una licencia MIT (figura 4-2).
- El archivo **README.md** contiene información general sobre el programa, como la manera de descargarlo o de ejecutarlo directamente desde internet.
- El archivo **space_frame.json** contiene la definición de un modelo estructural, que es de gran utilidad para conocer la sintáxis utilizada. Los ejemplos se explican y resuelven más adelante en el capítulo 5.

The MIT License (MIT)

Copyright (c) 2020 mas.js

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Figura 4-2: Licencia del programa mas.js.

- El archivo `index.html` es necesario para enviar el contenido de la aplicación al navegador web. Allí se determina la posición y tamaño de los diferentes elementos que componen la interfaz gráfica.
- El archivo `main.js` centraliza y controla el funcionamiento de la aplicación. Desde este archivo se hacen llamados a los diferentes módulos desarrollados, principalmente el `canvas` donde se muestra la estructura tridimensional y el `terminal` que constituye una línea de comandos para ejecutar acciones sobre el modelo.
- La carpeta `modules` contiene los módulos desarrollados, que se explican en la siguiente sección.
- La carpeta `libs` contiene las librerías de JavaScript utilizadas y la carpeta `css` contiene reglas de formato para definir detalles como colores y estilos de letra, entre otros.

4.1.2. Visualización y funcionamiento interno del programa

El usuario puede acceder al programa visitando la página web <https://rv cristiand.github.io/mas.js>, o a través de un servidor local, para modelar estructuras aporticadas tridimensionales sometidas a cargas estáticas. En la figura 4-3 se presenta mas.js

ejecutándose por primera vez en el navegador de internet Firefox.

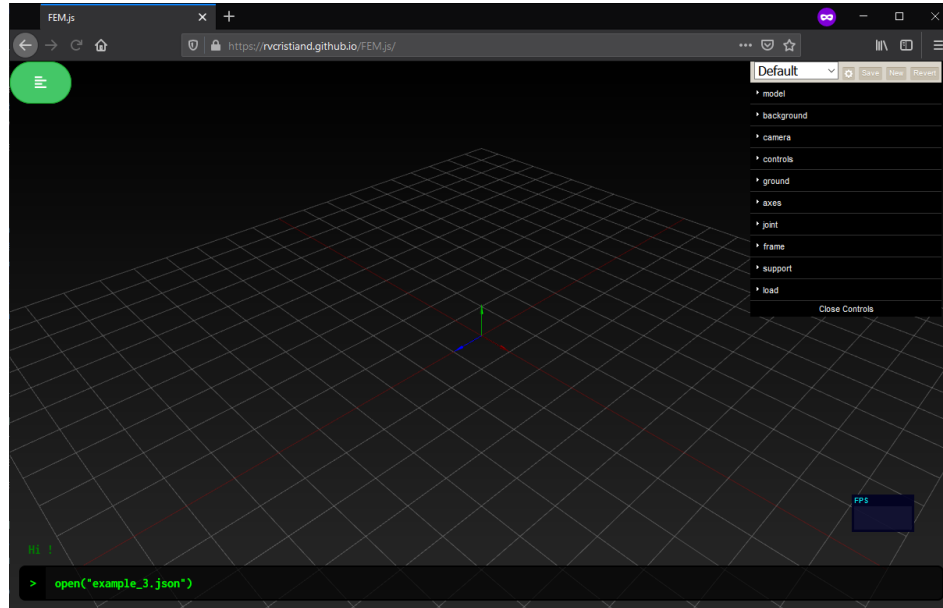


Figura 4-3: mas.js ejecutándose en el navegador web Firefox.

Una vez toda la aplicación web ha sido cargada en el navegador, se ejecuta la función `init()`, definida en el archivo `mas.js`. En esta función se utilizan algunos valores de configuración por defecto para desplegar la página web y una escena tridimensional vacía. Los valores por defecto se encuentran almacenados en la variable `config`, la cual puede ser modificada libremente por los usuarios, según sus gustos de colores y formato.

En el algoritmo 4.1 se presenta el pseudocódigo de la función `init()`. Esta función reemplaza los valores por defecto con los que el usuario haya guardado en una sesión previa, y con ellos configura la escena tridimensional y algunos elementos asociados a esta, como las luces de la escena o los materiales con los cuales se van a representar los elementos de la estructura.

Algoritmo 4.1: Pseudocódigo de la función `init()` del archivo `mas.js`.

```
function init() {  
  // refresh the config  
  
  // set the background  
  
  // create the scene
```

```
// create the camera
...

// create the controls

// set the materials

// create the model
model = createModel();
setModelRotation( config[ 'model.axisUpwards' ] );
scene.add( model );

// create the ground
var ground = createGround( config[ 'ground.size' ], config[ 'ground.
    grid.divisions' ], config[ 'ground.plane.color' ], config[ 'ground.
    plane.transparent' ], config[ 'ground.plane.opacity' ], config[ '
    ground.grid.major' ], config[ 'ground.grid.minor' ] );
scene.add( ground );

// create the structure
structure = createStructure();

// create the stats

// create the dat gui

render();
}
```

La escena tridimensional consiste en una jerarquía de nodos en un grafo, donde cada nodo representa un espacio local (Threejsfundamentals authors, 2021). Estos nodos definen una relación padre/hijo, donde los nodos hijos son posicionados y orientados en la escena con relación a su padre.

La raíz de la escena es el objeto `scene`, a la cual se le agregan otros dos objetos, `model` y `ground`. Los objetos `model` y `ground` son posicionados y orientados en la escena tridimensional con relación al sistema de coordenadas del espacio `scene`. El objeto `model` se orienta con alguno de sus ejes principales apuntando hacia la parte superior de la pantalla, según sea el valor de la variable `config[ 'model.axisUpwards' ]`, haciéndolo rotar con respecto al objeto `scene` con la función `setModelRotation()`.

Los objetos que describen el problema estructural se agregan al objeto `model`, por lo que el sistema de coordenadas del espacio del objeto `model` puede ser considerado como el *sistema de coordenadas global*.

El objeto `ground` consiste en otros dos objetos, `plane` y `grid`. El primer objeto representa un plano y el segundo una grilla, ambos del mismo tamaño y orientados horizontalmente, representando el nivel cero en la escena. Finalmente, la función `createStructure()` crea una variable para almacenar la información de la estructura.

En cuanto a la manera o formato para mostrar los diferentes elementos en el navegador, la variable `config` contiene toda la información necesaria. En el algoritmo 4.2 se presentan algunas entradas de dicha variable.

Algoritmo 4.2: Valores por defecto para configurar mas.js.

```
var config = {  
  // background  
  'background.topColor': '#000000',  
  'background.bottomColor': '#282828',  
  
  // model  
  'model.axisUpwards': 'y',  
  
  'model.axes.visible': true,  
  'model.axes.size': 1,  
  
  'model.axes.head.radius': 0.04,  
  'model.axes.head.height': 0.3,  
  
  'model.axes.shaft.length': 0.7,  
  'model.axes.shaft.radius': 0.01,  
  
  // camera  
  'camera.type': 'perspective',  
  
  'camera.perspective.fov': 45,  
  'camera.perspective.near': 0.1,  
  'camera.perspective.far': 1000,  
  
  'camera.position.x': 10,  
  'camera.position.y': 10,  
  'camera.position.z': 10,
```

```
'camera.target.x': 0,
'camera.target.y': 0,
'camera.target.z': 0,

// controls
...

// axes
...

// ground
...

// joint
...

// frame
...

// support
...

// load
...
};
```

En la figura 4-4 se presenta `mas.js` con unos colores del fondo de la escena alternativos a los valores estándar, modificados con los controles `top` y `bottom` de la sección `background` de la barra de herramientas.

El usuario puede generar múltiples configuraciones de estas opciones a través del panel superior de la barra de herramientas o directamente modificando el valor de diferentes objetos dentro de la variable `config`. Una manera de establecer una configuración de colores personalizada es haciendo clic sobre el botón con un piñón, mediante el cual se puede copiar el objeto que describe sus configuraciones. Posteriormente configurar la aplicación web en otro dispositivo, o guardar la configuración en la variable `localStorage` para ser usadas en próximas sesiones. En la figura 4-5 se presenta `mas.js` después de haber hecho clic sobre dicho botón, indicando que se guardarán las configuraciones en la variable `localStorage`.

En el algoritmo 4.3 se presenta la implementación de la función `createModel()`. La función genera un objeto tipo `THREE.Group` al cual se le ha agregado los objetos

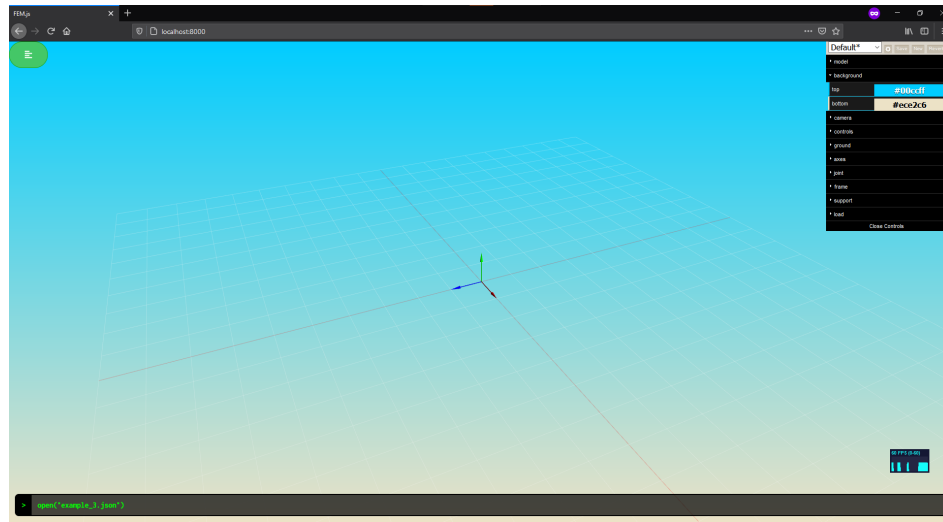


Figura 4-4: mas.js con colores del fondo de la escena arbitrarios.

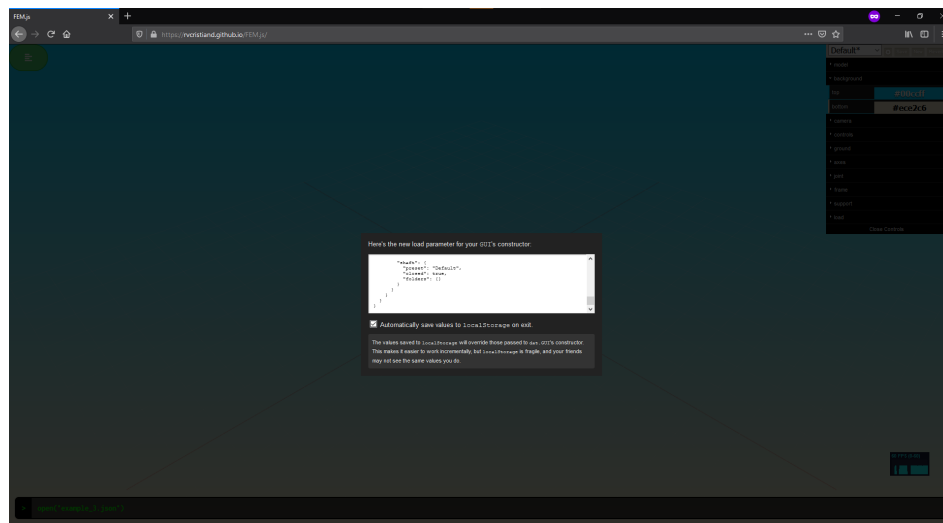


Figura 4-5: mas.js almacenando la configuración del usuario en la variable localStorage.

axes, joints, frames, loads, internalForces, deformedStructure, deformedJoints y deformedFrames, mediante la función add().

THREE.Object3D es la clase base de la mayoría de los objetos en Three.js, al proveer un conjunto de métodos, funciones y propiedades para manipular objetos en la escena tridimensional (Three.js authors, 2021a). La clase THREE.Group es casi idéntica a la clase THREE.Object3D. Su propósito es permitir trabajar con grupos de objetos de manera

sintáctica más clara.

Algoritmo 4.3: Implementación de la función `createModel()` del archivo `mas.js`.

```
function createModel() {  
    // create the model  
  
    // model  
    // -joints  
    // -frames  
    // -loads  
    // -internal forces  
    // -deformed structure  
    // --joints  
    // --deformed frames  
  
    var model = new THREE.Group();  
    model.name = "model";  
  
    // add axes  
    var axes = createAxes( config[ 'model.axes.shaft.length' ], config[ '  
        model.axes.shaft.radius' ], config[ 'model.axes.head.height' ], config[ '  
        'model.axes.head.radius' ] );  
    axes.name = 'axes';  
    axes.visible = config[ 'model.axes.visible' ];  
    axes.scale.setScalar( config[ 'model.axes.size' ] );  
    model.add( axes );  
  
    // add joints  
    var joints = new THREE.Group();  
    joints.name = 'joints';  
    model.add( joints );  
  
    // add frames  
    var frames = new THREE.Group();  
    frames.name = 'frames';  
    frames.visible = config[ 'frame.visible' ];  
    model.add( frames );  
  
    // add loads  
    var loads = new THREE.Group();  
    loads.name = 'loads';  
    loads.visible = config[ 'load.visible' ];  
    model.add( loads );  
}
```

```

// add internal forces
var internalForces = new THREE.Group();
internalForces.name = 'internal forces';
internalForces.visible = config[ 'internal_forces.visible' ];
model.add( internalForces );

// add deformed structure
var deformedStructure = new THREE.Group();
deformedStructure.name = 'deformed structure';
deformedStructure.visible = config[ 'displacements.visible' ];
model.add( deformedStructure );

// add deformed joints
var deformedJoints = new THREE.Group();
deformedJoints.name = 'joints';
deformedStructure.add( deformedJoints );

// add deformed frames
var deformedFrames = new THREE.Group();
deformedFrames.name = 'deformed frames';
deformedStructure.add( deformedFrames );

return model;
}

```

Los nodos de la estructura se agregarán al objeto `joints`. Así mismo, los elementos aporticados, las cargas, las fuerzas internas y la estructura deformada, se agregarán en los objetos `frames`, `loads`, `internalForces` y `deformedFrames`, respectivamente.

En el algoritmo 4.4 se presenta la implementación de la función `createStructure()`. La función crea una nueva variable con las entradas `joints`, `materials`, `sections`, `frames`, `supports` y `load_patterns`.

Algoritmo 4.4: Implementación de la función `createStructure()` del archivo `mas.js`.

```

function createStructure() { return { joints: {}, materials: {}, sections
: {}, frames: {}, supports: {}, load_patterns: {} } };

```

En el algoritmo 4.5 se presenta la implementación de la función `render()`. Esta función se ejecuta repetidamente cada cierto tiempo, mediante la función `requestAnimationFrame()`, para redibujar la escena tridimensional, actualizar el monitor de desempeño

y los controles de la cámara, de ser necesario.

Según MDN, 2021, la función `requestAnimationFrame()` le indica al navegador web que se desea hacer una animación y se requiere llamar una función en específico (en este caso la función `render()`) que actualice la animación antes de ser dibujada nuevamente. En general, la función es ejecutada 60 veces por segundo, pero puede variar levemente según las características del internet y el computador.

Algoritmo 4.5: Implementación de la función `render()` del archivo `mas.js`.

```
function render() {  
  // render the scene  
  
  requestAnimationFrame( render );  
  
  stats.update();  
  
  webGLRenderer.render( scene , camera );  
  CSS2DRenderer.render( scene , camera );  
  
  if ( controls.enableDamping ) controls.update();  
}
```

4.2. Interacción del usuario con el programa

Una vez la aplicación web está desplegada, el usuario puede comenzar a crear la estructura ejecutando un conjunto de funciones a través de la línea de comandos de `mas.js`. Dicho conjunto de funciones están listadas en el archivo `main.js`, aunque la implementación de las mismas se encuentran en el archivo `mas.js`.

Estas funciones tienen por objeto modificar las variables `model` y `structure`, las cuales almacena los objetos tridimensionales del modelo y su información, respectivamente. A continuación se presenta la implementación de cada una de estas funciones.

4.2.1. `open()`

La función `open()` permite abrir archivos con modelos de estructuras almacenados en formato JSON. La función recibe el nombre del archivo, y mediante un conjunto de

funciones creadas para tal fin, lee cada uno de los objetos allí almacenados y los agrega al programa.

En el algoritmo 4.6 se presenta el pseudocódigo de la función `open()`. Esta función elimina cualquier objeto que se le haya agregado al objeto `model` y le asigna una nueva variable a `structure`, con la función `createStructure()` (algoritmo 4.4), antes de agregar cada uno de los objetos definidos en el archivo indicado por el usuario.

Algoritmo 4.6: Implementación de la función `open()` del archivo `mas.js`.

```
export function open( filename ) {  
  // open a file  
  
  var promise = loadJSON( filename )  
  .then( json => {  
    // delete labels  
    ...  
  
    // delete objects  
    ...  
  
    // create structure  
    structure = createStructure();  
  
    // add materials  
    ...  
  
    // add sections  
    ...  
  
    // add joints  
    ...  
  
    // add frames  
    ...  
  
    // add supports  
    ...  
  
    // add load patterns  
    ...  
  
    // add joint displacements  
    ...  
  }  
}
```

```
    // add internal forces
    ...

    // add internal displacements
    ...

    return "the '" + filename + "' model has been loaded";
  });

  return promise;
}
```

En la figura 4-6 se presenta `mas.js` después de ejecutar la función `open()` para abrir el archivo `space_frame.json`. Este archivo ha sido generado con `pymas`, un programa de computador desarrollado en Python para analizar estructuras aporticadas sometidas a cargas estáticas.

A través de la barra de herramientas el usuario puede modificar la apariencia de los objetos que se muestran en la escena. Para este caso en particular, se le indicó a `mas.js` que mostrara los nombres de los nodos, los cuales se presentan en blanco sobre un rectángulo negro, que ocultara los ejes locales de los elementos aporticados y mostrara las cargas externas.

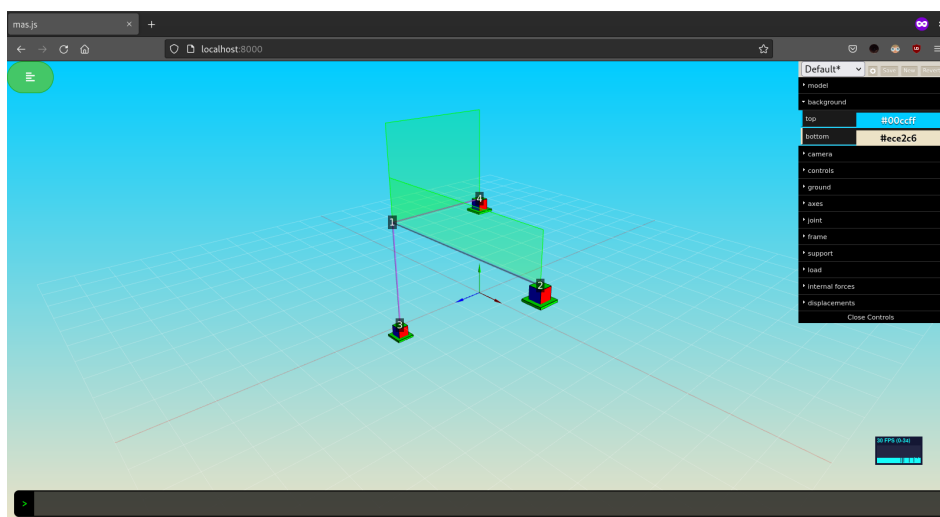


Figura 4-6: Archivo `space_frame.json` abierto con `mas.js`.

A través de la barra de herramientas, el usuario también puede ver los resultados del análisis. Para eso, la barra de herramientas cuenta con dos apartados, *internal forces* y *displacements*, con varios controles para modificar algunos objetos de la escena.

Por ejemplo, el apartado *internal forces* tiene cuatro controles: *visible*, *load pattern*, *action* y *scale*. Con el control *visible* el usuario puede hacer aparecer y desaparecer las fuerzas internas de los elementos aporticados. Con el control *load pattern* el usuario puede definir que fuerzas internas ver según el caso de carga. Con el control *action* el usuario puede ver cada una de seis las fuerzas internas. Y, finalmente, con el control *scale* el usuario puede escalar los resultados a conveniencia. En la figura 4-7 se presentan las fuerzas internas a lo largo de los elementos aporticados.

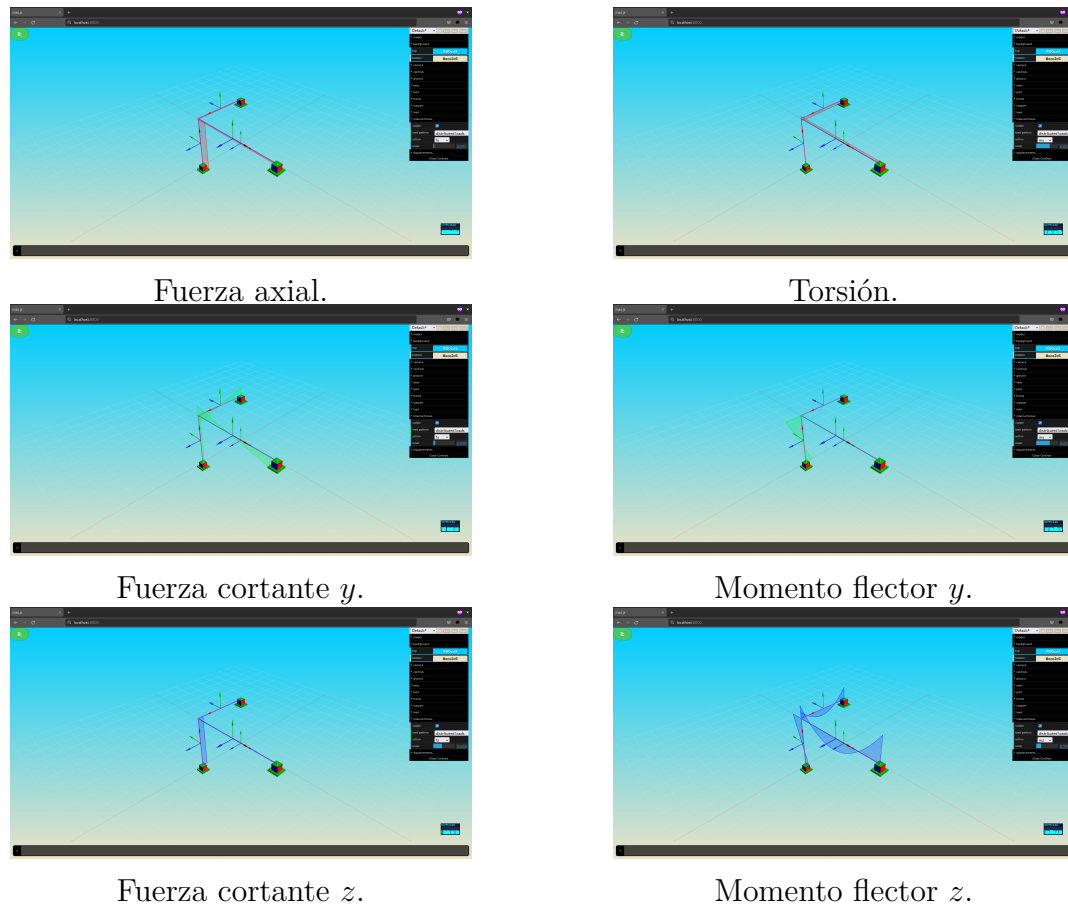


Figura 4-7: Fuerzas internas de los elementos aporticados del archivo `space_frame.json` en `mas.js`.

El apartado *displacements* tiene tres controles, muy similares a los controles del apartado *internal forces*, con la diferencia que estos modifican los objetos de la escena relacionados con los desplazamientos de la estructura. Por ejemplo, en la figura 4-8 se presenta la estructura deformada debido a las cargas externas.

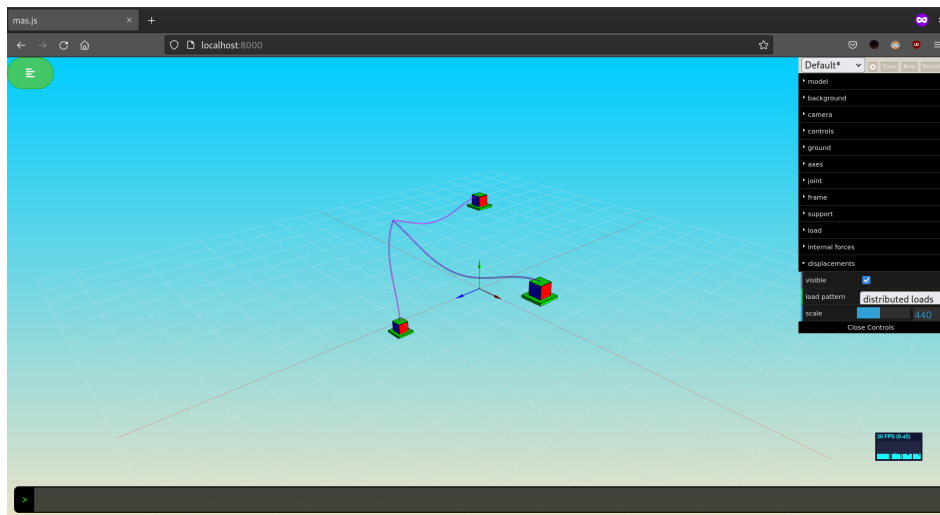


Figura 4-8: Configuración deformada del archivo `space_frame.json` en `mas.js`.

A continuación se presenta el contenido del archivo `space_frame.json`. `mas.js` lee: materiales, secciones transversales, nodos, elementos aporticados, patrones de carga, desplazamientos de los nodos y deformaciones de los elementos aporticados, los agrega a la variable `structure` y crea su correspondiente representación en la escena, mediante una serie de funciones desarrolladas para tal fin.

```
{
  "materials": {
    "material1": {
      "name": "material1",
      "E": 2200000.0,
      "G": 850000.0
    }
  },
  "sections": {
    "section1": {
      "type": "Section",
      "name": "section1",
      "A": 0.12,
      "J": 0.001944,
      "Iy": 0.0009,
      "Iz": 0.0016
    },
    "section2": {
      "type": "Section",
      "name": "section2",
      "A": 0.1,
      "J": 0.0012734,
      "Iy": 0.001333,
      "Iz": 0.0005208
    }
  },
  "joints": {
    "1": {
      "name": "1",
      "x": 0,

```

```

        "y": 3,
        "z": 3
    },
    "2": {
        "name": "2",
        "x": 5,
        "y": 3,
        "z": 3
    },
    "3": {
        "name": "3",
        "x": 0,
        "y": 0,
        "z": 3
    },
    "4": {
        "name": "4",
        "x": 0,
        "y": 3,
        "z": 0
    }
},
"frames": {
    "1-2": {
        "name": "1-2",
        "joint_j": "1",
        "joint_k": "2",
        "material": "material1",
        "section": "section1"
    },
    "4-1": {
        "name": "4-1",
        "joint_j": "4",
        "joint_k": "1",
        "material": "material1",
        "section": "section2"
    },
    "3-1": {
        "name": "3-1",
        "joint_j": "3",
        "joint_k": "1",
        "material": "material1",
        "section": "section1"
    }
},
"supports": {
    "2": {
        "joint": "2",
        "ux": true,
        "uy": true,
        "uz": true,
        "rx": true,
        "ry": true,
        "rz": true
    },
    "3": {
        "joint": "3",

```

```

        "ux": true,
        "uy": true,
        "uz": true,
        "rx": true,
        "ry": true,
        "rz": true
    },
    "4": {
        "joint": "4",
        "ux": true,
        "uy": true,
        "uz": true,
        "rx": true,
        "ry": true,
        "rz": true
    }
},
"load_patterns": {
    "distributed loads": {
        "name": "distributed loads",
        "frames": {
            "1-2": [
                {
                    "type": "
DistributedLoad",
                    "load_pattern": "
distributed loads",
                    "frame": "1-2",
                    "system": "local",
                    "fy": -2.4
                }
            ],
            "4-1": [
                {
                    "type": "
DistributedLoad",
                    "load_pattern": "
distributed loads",
                    "frame": "4-1",
                    "system": "local",
                    "fy": -3.5
                }
            ]
        }
    }
},
"displacements": {
    "distributed loads": {
        "1": {
            "load_pattern": "distributed
loads",
            "joint": "1",
            "ux": 2.687315391431904e-05,
            "uy":
-0.00011574968996290452,
            "uz": -1.0006115108507558e
-05,

```

```

        "rx": -0.0005668526305782732,
        "ry": 7.904785530558646e-06,
        "rz": -0.0006309015221398214
    },
    "2": {
        "load_pattern": "distributed
loads",
        "joint": "2",
        "ux": 0.0,
        "uy": 0.0,
        "uz": 0.0,
        "rx": 0.0,
        "ry": 0.0,
        "rz": 0.0
    },
    "3": {
        "load_pattern": "distributed
loads",
        "joint": "3",
        "ux": 0.0,
        "uy": 0.0,
        "uz": 0.0,
        "rx": 0.0,
        "ry": 0.0,
        "rz": 0.0
    },
    "4": {
        "load_pattern": "distributed
loads",
        "joint": "4",
        "ux": 0.0,
        "uy": 0.0,
        "uz": 0.0,
        "rx": 0.0,
        "ry": 0.0,
        "rz": 0.0
    }
},
"reactions": {
    "distributed loads": {
        "2": {
            "load_pattern": "distributed
loads",
            "joint": "2",
            "fx": -1.4189025266760453,
            "fy": 6.572099741135986,
            "fz": 0.005658316443946586,
            "mx": 0.18733345735350773,
            "my": 0.01101549603976524,
            "mz": -5.9860946812535305
        },
        "3": {
            "load_pattern": "distributed
loads",
            "joint": "3",
            "fx": 1.4384740156088265,

```

```

        "fy": 10.185972716735598,
        "fz": -0.7394400910678338,
        "mx": -0.7350374004200906,
        "my": -0.004353955870231471,
        "mz": -1.4174532374358457
    },
    "4": {
        "load_pattern": "distributed
loads",
        "joint": "4",
        "fx": -0.019571488932780986,
        "fy": 5.741927542128418,
        "fz": 0.7337817746238872,
        "mx": -3.146398956522173,
        "my": -0.03708442474814382,
        "mz": 0.22762716618297452
    }
},
"end_actions": {
    "distributed loads": {
        "1-2": {
            "frame": "1-2",
            "load_pattern": "distributed
loads",
            "fx_j": 1.4189025266760453,
            "fy_j": 5.427900258864014,
            "fz_j":
-0.005658316443946586,
            "mx_j": -0.18733345735350773,
            "my_j": 0.017276086179967686,
            "mz_j": 3.125595975573601,
            "fx_k": -1.4189025266760453,
            "fy_k": 6.572099741135986,
            "fz_k": 0.005658316443946586,
            "mx_k": 0.18733345735350773,
            "my_k": 0.01101549603976524,
            "mz_k": -5.9860946812535305
        },
        "4-1": {
            "frame": "4-1",
            "load_pattern": "distributed
loads",
            "fx_j": 0.7337817746238872,
            "fy_j": 5.741927542128418,
            "fz_j": 0.01957148893278115,
            "mx_j": 0.22762716618297382,
            "my_j": -0.03708442474814382,
            "mz_j": 3.146398956522173,
            "fx_k": -0.7337817746238872,
            "fy_k": 4.758072457871582,
            "fz_k": -0.01957148893278115,
            "mx_k": -0.22762716618297382,
            "my_k": -0.02163004205019963,
            "mz_k": -1.6706163301369188
        },
        "3-1": {

```

```

        "frame": "3-1",
        "load_pattern": "distributed
loads",
        "fx_j": 10.185972716735598,
        "fy_j": -1.4384740156088243,
        "fz_j": -0.7394400910678338,
        "mx_j":
-0.004353955870231634,
        "my_j": 0.7350374004200906,
        "mz_j": -1.4174532374358457,
        "fx_k": -10.185972716735598,
        "fy_k": 1.4384740156088243,
        "fz_k": 0.7394400910678338,
        "mx_k": 0.004353955870231634,
        "my_k": 1.483282872783411,
        "mz_k": -2.8979688093906266
    }
},
"internal_forces": {
    "distributed loads": {
        "1-2": {
            "load_pattern": "distributed
loads",
            "frame": "1-2",
            "fx": [...],
            "fy": [...],
            "fz": [...],
            "mx": [...],
            "my": [...],
            "mz": [...]
        },
        "4-1": {
            "load_pattern": "distributed
loads",
            "frame": "4-1",
            "fx": [...],
            "fy": [...],
            "fz": [...],
            "mx": [...],
            "my": [...],
            "mz": [...]
        },
        "3-1": {
            "load_pattern": "distributed
loads",
            "frame": "3-1",
            "fx": [...],
            "fy": [...],
            "fz": [...],
            "mx": [...],
            "my": [...],
            "mz": [...]
        }
    }
}

```

El usuario es capaz de ejecutar las mismas funciones para agregar estos elementos al programa, permitiéndole modelar sus estructuras. A continuación se presentan dichas funciones.

4.2.2. addMaterial()

La función `addMaterial()` permite agregar materiales al programa. La función recibe los valores del módulo de Young y el módulo de elasticidad a cortante del material.

En el algoritmo 4.7 se presenta la implementación de la función `addMaterial()`. Antes de agregar la nueva entrada a la variable `structure`, la función verifica que no haya otro material con el mismo nombre.

Algoritmo 4.7: Función `addMaterial()` implementada en el archivo `mas.js`.

```
export function addMaterial( name, e, g ) {  
  // add a material  
  
  var promise = new Promise( ( resolve , reject ) => {  
    // only strings accepted as name  
    name = name.toString();  
  
    // check if material's name already exists  
    if ( structure.materials.hasOwnProperty( name ) ) {  
      reject( new Error( "material's name '" + name + "' already exist" ) )  
    };  
    } else {  
      // add material to structure  
      structure.materials[ name ] = { "E": e, "G": g };  
  
      resolve( "material '" + name + "' was added" );  
    }  
  });  
  
  return promise;  
}
```

4.2.3. addSection()

La función `addSection()` permite agregar secciones transversales generales al programa. La función recibe el nombre de la sección.

En el algoritmo 4.8 se presenta la implementación de la función `addSection()`. Antes de agregar las nuevas entradas en las variables `structure` y `sections`, la función verifica que no haya otra sección transversal con el mismo nombre.

En la variable **structure** se almacena la información de la sección transversal general, mientras que en la variable **sections** se almacena un objeto tipo **THREE.Shape** que representa un círculo de radio unitario creado con la función **createSection()**.

Según Three.js authors, 2021b, con los objetos tipo **THREE.Shape** se pueden definir figuras planas en dos dimensiones usando *paths*. Estos objetos pueden ser extrudidos para crear geometrías tridimensionales.

Algoritmo 4.8: Función **addSection()** implementada en el archivo **mas.js**.

```
export function addSection( name ) {
  // add a section

  var promise = new Promise( ( resolve , reject ) => {
    // only strings accepted as name
    name = name.toString();

    // check if section's name already exists
    if ( structure.sections.hasOwnProperty( name ) ) {
      reject( new Error( "section's name '" + name + "' already exists" ) );
    } else {
      structure.sections[ name ] = { type: "Section" };
      // create section
      sections[ name ] = createSection();

      resolve( "section '" + name + "' was added" );
    }
  });

  return promise;
}
```

4.2.4. addRectangularSection()

La función **addRectangularSection()** permite agregar secciones transversales rectangulares al programa. La función recibe el nombre, la base y el alto de la sección transversal.

En el algoritmo 4.9 se presenta la implementación de la función **addRectangularSection()**. Antes de agregar las nuevas entradas en las variables **structure** y **section**,

la función verifica que no haya otra sección transversal con el mismo nombre.

En tal caso, se agrega la información de la sección transversal rectangular en la variable **structure**, mientras que en la variable **sections** se almacena un objeto **THREE.Shape** que representa un rectángulo de las dimensiones dadas, creado con la función **createRectangularSection()**.

Algoritmo 4.9: Función **addRectangularSection()** implementada en el archivo **mas.js**.

```
export function addRectangularSection( name, width, height ) {
  // add a rectangular section

  var promise = new Promise( ( resolve, reject ) => {
    // only strings accepted as name
    name = name.toString();

    // check if section's name already exists
    if ( structure.sections.hasOwnProperty( name ) ) {
      reject( new Error( "section's name '" + name + "' already exists" ) );
    } else {
      // add section to structure
      structure.sections[ name ] = { type: "RectangularSection", width:
width, height: height };
      // create rectangular section
      sections[ name ] = createRectangularSection( width, height );

      resolve( "rectangular section '" + name + "' was added" );
    }
  });

  return promise;
}
```

4.2.5. addJoint()

La función **addJoint()** permite agregar nodos al programa. La función recibe el nombre del nodo y sus coordenadas.

En el algoritmo 4.10 se presenta la implementación de la función **addJoint()**. Antes de agregar el nodo al programa, la función verifica que no haya otro con el mismo nombre

o con las mismas coordenadas. En el caso que no haya ningún inconveniente, se agrega la información del nodo a la variable `structure`, se crea un objeto tipo `THREE.Group`, que se asigna a la variable `parent`, y se agrega a la variable `model`.

A este objeto se le modifica su posición, asignándole las coordenadas del nodo, y se le agrega el objeto `joint`, al cual, a su vez, se le ha agregado el objeto `label`. El objeto `joint`, creado con la función `createJoint()`, representa el nodo con una esfera mientras que el objeto `label` presenta el nombre del nodo con una etiqueta html (véase la figura 4-6).

Algoritmo 4.10: Función `addJoint()` implementada en el archivo `mas.js`.

```
export function addJoint( name, x, y, z ) {
  // add a joint

  var promise = new Promise( ( resolve, reject ) => {
    // only strings accepted as name
    name = name.toString();

    // check if joint's name or joint's coordinate already exists
    if ( structure.joints.hasOwnProperty( name ) || Object.values(
      structure.joints ).some( joint => joint.x === x && joint.y === y &&
      joint.z === z ) ) {
      if ( structure.joints.hasOwnProperty( name ) ) {
        reject( new Error( "joint's name '" + name + "' already exist" ) )
      };
    } else {
      reject( new Error( "joint's coordinate [" + x + ", " + y + ", " +
        z + "] already exist" ) );
    }
  } else {
    // add joint to structure
    structure.joints[ name ] = { x: x, y: y, z: z };

    // parent
    var parent = new THREE.Group();
    parent.name = name;
    parent.position.set( x, y, z );
    model.getObjectByName( 'joints' ).add( parent );

    // joint
    var joint = createJoint( config[ 'joint.size' ] );
    parent.add( joint );
  }
}
```

```

    // label
    var label = document.createElement( 'div' );
    label.className = 'joint';
    label.textContent = name;
    label = new THREE.CSS2DObject( label );
    label.name = 'label';
    label.visible = config[ 'joint.label' ];
    label.position.set( 0.5, 0.5, 0.5 );
    joint.add( label );

    resolve( "joint '" + name + "' was added" );
  }
});

return promise;
}

```

4.2.6. addFrame()

La función `addFrame()` permite agregar elementos aporticados al programa. La función recibe el nombre del elemento aporticado, el nodo cercano, el nodo lejano, el material y la sección transversal.

En el algoritmo 4.11 se presenta la implementación de la función `addFrame()`. Antes de agregar el elemento aporticado al programa, la función verifica que no haya otro con el mismo nombre o con los mismos nodos.

En el caso que no haya ningún inconveniente, se agrega la información del elemento aporticado a la variable `structure`, se almacena un objeto tipo `THREE.Group` creado con la función `createFrame()` en la variable `frame` y se agrega a la variable `model`.

A este objeto se le modifica su posición y orientación, del tal manera que quede entre los nodos, y se le agregan los objetos `axes` y `label`. El objeto `axes`, creado con la función `createAxes()`, representa los ejes locales del elemento aporticado mientras que el objeto `label` presenta su nombre con una etiqueta html.

Algoritmo 4.11: Función `addFrame()` implementada en el archivo `mas.js`.

```

export function addFrame( name, j, k, material, section ) {
  // add a frame

```

```

var promise = new Promise( ( resolve , reject ) => {
  // only strings accepted as name
  name = name.toString();

  j = j.toString();
  k = k.toString();

  material = material.toString();
  section = section.toString();

  // check if frame's name of frame's joints already exists
  if ( structure.frames.hasOwnProperty( name ) || Object.values(
structure.frames ).some( frame => frame.j === j && frame.k === k ) ) {
    if ( structure.frames.hasOwnProperty( name ) ) {
      reject( new Error( "frame's name '" + name + "' already exists" )
);
    } else {
      reject( new Error( "frame's joints [" + j + ", " + k + "] already
taked" ) );
    }
  } else {
    // check if joints, material and section exists
    if ( structure.joints.hasOwnProperty( j ) && structure.joints.
hasOwnProperty( k ) && structure.materials.hasOwnProperty( material )
&& structure.sections.hasOwnProperty( section ) ) {
      // add frame to structure
      structure.frames[ name ] = { j: j, k: k, material: material,
section: section };

      // get frame's joints
      j = model.getObjectByName( 'joints' ).getObjectByName( j );
      k = model.getObjectByName( 'joints' ).getObjectByName( k );

      // calculate local axis
      var x_local = k.position.clone().sub( j.position );

      // create frame
      var frame = createFrame( x_local.length(), structure.frames[ name
].section );
      frame.name = name;
      frame.position.copy( x_local.clone().multiplyScalar(0.5).add( j.
position ) );
      frame.quaternion.setFromUnitVectors( new THREE.Vector3( 1, 0, 0 )
, x_local.clone().normalize() );
    }
  }
} );

```

```

        // add axes
        var axes = createAxes( config[ 'frame.axes.shaft.length' ], config
[ 'frame.axes.shaft.radius' ], config[ 'frame.axes.head.height' ],
config[ 'frame.axes.head.radius' ] );
        axes.name = 'axes';
        axes.visible = config[ 'frame.axes.visible' ];
        frame.add( axes );

        // add label
        var label = document.createElement( 'div' );
        label.className = 'frame';
        label.textContent = name;
        label = new THREE.CSS2DObject( label );
        label.name = 'label';
        label.visible = config[ 'frame.label' ];
        frame.add( label );

        // add frame to scene
        model.getObjectByName( 'frames' ).add( frame );

        resolve( "frame '" + name + "' was added" );
    } else {
        if ( !structure.joints.hasOwnProperty( j ) ) reject( new Error("
joint '" + j + "' does not exists" ) );
        if ( !structure.joints.hasOwnProperty( k ) ) reject( new Error("
joint '" + k + "' does not exists" ) );
        if ( !structure.materials.hasOwnProperty( material ) ) reject(
new Error( "material '" + material + "' does not exists" ) );
        if ( !structure.sections.hasOwnProperty( section ) ) reject( new
Error( "section '" + section + "' does not exists" ) );
    }
}
});

return promise;
}

```

La función `craeteFrame()` crea un objeto tipo `THREE.Group` al que le agrega dos objetos también tipo `THREE.Group`, a los cuales se les asigna los nombres `wireFrame` y `extrudeFrame` respectivamente. Estos dos objetos representan el elemento aporticado en *forma de palillo* y extruido. En caso que la sección transversal del elemento aporticado sea general, el objeto `extrudeFrame` se copia del objeto `wireFrame`.

4.2.7. addLoadPattern()

La función `addLoadPattern()` permite agregar patrones de carga al programa. La función recibe el nombre del patrón de carga.

En el algoritmo 4.12 se presenta la implementación de la función `addLoadPattern()`. Antes de agregar el patrón de carga al programa, la función verifica que no haya otro con el mismo nombre.

Si no hay otro patrón de carga con el mismo nombre, se agregar una nueva entrada a la variable `structure`, se almacena un objeto tipo `THREE.Group` en la variable `loadPattern` y se agrega a la variable `model`.

Finalmente, la función actualiza la barra de herramientas, específicamente la lista de patrones de carga de la sección `load`, para poder alternar el caso de carga visible en la escena.

Algoritmo 4.12: Función `addLoadPattern()` implementada en el archivo `mas.js`.

```
export function addLoadPattern( name ) {  
  // add a load pattern  
  
  var promise = new Promise( ( resolve , reject ) => {  
    // only strings accepted as name  
    name = name.toString();  
  
    // check if load pattern's name already exists  
    if ( structure.load_patterns.hasOwnProperty( name ) ) {  
      reject( new Error( "load pattern's name '" + name + "' already exists" ) );  
    } else {  
      // add load pattern to structure  
      structure.load_patterns[ name ] = {};  
  
      // add load pattern to model  
      var loadPattern = new THREE.Group();  
      loadPattern.name = name;  
      loadPattern.visible = name === config[ 'load.loadPattern' ];  
      model.children.find( obj => obj.name === "loads" ).add( loadPattern );  
    }  
  
    // add load pattern to controller  
  } );  
}
```

```

        var str, innerHTMLStr = "<option value='" + "" + "'" + ">" + "" + "</options>";
        Object.keys( structure.load_patterns ).forEach( loadPattern => {
            str = "<option value='" + loadPattern + "'" + ">" + loadPattern + "</options>";
            innerHTMLStr += str;
        });
        loadPatternController.domElement.children[ 0 ].innerHTML = innerHTMLStr;
        loadPatternController.updateDisplay();

        resolve( "load pattern '" + name + "' was added" );
    }
});

return promise;
}

```

4.2.8. addLoadAtJoint()

La función `addLoadAtJoint()` permite agregar cargas puntuales en los nodos de la estructura. La función recibe el patrón de carga asociado a la carga, el nodo en que actúa, y la magnitud de la fuerza en sus componentes con relación al sistema de coordenadas global.

En el algoritmo 4.13 se presenta la implementación de la función `addLoadPattern()`. Antes de agregar la carga puntual al programa, la función verifica que tanto el patrón de carga como el nodo existan.

En el caso que no haya ningún inconveniente, se agrega la información de la carga a la variable `structure`, se almacena un objeto tipo `THREE.Group` creado con la función `createLoadAtJoint()` en la variable `load` y se agrega a la variable `model`.

Algoritmo 4.13: Función `addLoadAtJoint()` implementada en el archivo `mas.js`.

```

export function addLoadAtJoint( loadPattern, joint, fx, fy, fz, mx, my,
    mz ) {
    // add a load at joint

    var promise = new Promise( ( resolve, reject ) => {
        // only strings accepted as name
        loadPattern = loadPattern.toString();
    }

```

```
joint = joint.toString();

// only numbers accepted as values
fx = fx ? fx : 0;
fy = fy ? fy : 0;
fz = fz ? fz : 0;
mx = mx ? mx : 0;
my = my ? my : 0;
mz = mz ? mz : 0;

// check if loadPattern & joint exists
if ( structure.load_patterns.hasOwnProperty( loadPattern ) &&
    structure.joints.hasOwnProperty( joint ) ) {
    // add load to structure

    if ( !structure.load_patterns[ loadPattern ].hasOwnProperty( '
joints' ) ) structure.load_patterns[ loadPattern ].joints = {};
    if ( !structure.load_patterns[ loadPattern ].joints.hasOwnProperty(
joint ) ) structure.load_patterns[ loadPattern ].joints[ joint ] =
[];

    structure.load_patterns[ loadPattern ].joints[ joint ].push( { 'fx'
: fx, 'fy': fy, 'fz': fz, 'mx': mx, 'my': my, 'mz': mz } );

    // add loads to joint
    if ( !model.getObjectByName( 'joints' ).getObjectByName( joint ).
getObjectByName( 'loads' ) ) {
        var loads = new THREE.Group();
        loads.name = 'loads';
        loads.visible = config[ 'load.visible' ];
        model.getObjectByName( 'joints' ).getObjectByName( joint ).add(
loads );
    }

    // remove loadPattern
    if ( model.getObjectByName( 'joints' ).getObjectByName( joint ).
getObjectByName( 'loads' ).getObjectByName( loadPattern ) ) model.
getObjectByName( 'joints' ).getObjectByName( joint ).getObjectByName(
'loads' ).remove( model.getObjectByName( 'joints' ).getObjectByName(
joint ).getObjectByName( 'loads' ).getObjectByName( loadPattern ) );

    // add load to model
    var load = createLoadAtJoint( loadPattern, joint );
    load.visible = loadPattern == config[ 'load.loadPattern' ];
```

```

    model.getObjectByName( 'joints' ).getObjectByName( joint ).
    getObjectByName( 'loads' ).add( load );

    // set force scale
    setLoadForceScale( config[ 'load.force.scale' ] );

    // set torque scale
    setLoadTorqueScale( config[ 'load.torque.scale' ] );

    resolve( "load added to joint '" + joint + "' in load pattern '" +
loadPattern + "'" );
  } else {
    if ( structure.load_patterns.hasOwnProperty( loadPattern ) ) {
      reject( new Error( "joint '" + joint + "' does not exist" ) );
    } else {
      reject( new Error( "load pattern '" + loadPattern + "' does not
exist" ) );
    }
  }
});

return promise;
}

```

La función `createLoadAtJoint()` crea un objeto tipo `THREE.Group` al que le agrega dos objetos también tipo `THREE.Group`, a los cuales se les asigna los nombres `components` y `resultant` respectivamente. Estos dos objetos representan la carga puntual en sus componentes con respecto al sistema de coordenadas global y como resultante, mediante flechas de colas rectas y curvas.

4.2.9. addUniformlyDistributedLoadAtFrame()

La función `addUniformlyDistributedLoadAtFrame()` permite agregar cargas distribuidas en los elementos aporticados de la estructura. La función recibe el patrón de cargas asociado a la carga, el elemento aporticado en que actúa, el sistema de coordenadas de referencia y la magnitud de la fuerza en sus componentes con respecto a dicho sistema.

En el algoritmo 4.14 se presenta la implementación de la función `addUniformlyDistributedLoadAtFrame()`. Antes de agregar la carga distribuida al programa, la función

verifica que tanto el patrón de carga como el elemento aporticado existan.

Sí las dos entradas existen, se agrega la información de la carga a la variable `structure` y se agrega un objeto tipo `THEE.Group` creado con la función `createGlobalLoadAtFrame()` a la variable `model`.

Algoritmo 4.14: Función `addUniformlyDistributedLoadAtFrame()` implementada en el archivo `mas.js`.

```
export function addUniformlyDistributedLoadAtFrame( loadPattern , frame ,
    system , fx , fy , fz , mx , my , mz ) {
    // add a uniformly distributed load at frame

    var promise = new Promise( ( resolve , reject ) => {
        // only strings accepted as name
        loadPattern = loadPattern.toString();
        frame = frame.toString();

        // check if loadPattern & frame exists
        if ( structure.load_patterns.hasOwnProperty( loadPattern ) &&
            structure.frames.hasOwnProperty( frame ) ) {
            // add load to structure

            if ( !structure.load_patterns[ loadPattern ].hasOwnProperty( '
frames' ) ) structure.load_patterns[ loadPattern ].frames = {};
            if ( !structure.load_patterns[ loadPattern ].frames.hasOwnProperty(
frame ) ) structure.load_patterns[ loadPattern ].frames[ frame ] =
{};
            if ( !structure.load_patterns[ loadPattern ].frames[ frame ].
hasOwnProperty( 'uniformly_distributed' ) ) structure.load_patterns[
loadPattern ].frames[ frame ][ 'uniformly_distributed' ] = {};
            if ( !structure.load_patterns[ loadPattern ].frames[ frame ].
uniformly_distributed.hasOwnProperty( system ) ) structure.
load_patterns[ loadPattern ].frames[ frame ][ 'uniformly_distributed'
][ system ] = [];
            structure.load_patterns[ loadPattern ].frames[ frame ].
uniformly_distributed[ system ].push( { 'fx': fx , 'fy': fy , 'fz': fz ,
'mx': mx , 'my': my , 'mz': mz } );

            // add frame to loads
            if ( !model.children.find( obj => obj.name === "loads" ).
getObjectByName( loadPattern ).getObjectByName( 'frames' ) ) {
                var frames = new THREE.Group();
                frames.name = 'frames';
```

```

        model.children.find( obj => obj.name === "loads" ).getObjectByName
( loadPattern ).add( frames );
    }

    // remove loadPattern
    if ( model.children.find( obj => obj.name === "loads" ).
getObjectByName( loadPattern ).getObjectByName( 'frames' ).
getObjectByName( frame ) ) model.children.find( obj => obj.name === '
loads' ).getObjectByName( loadPattern ).getObjectByName( 'frames' ).
remove( model.children.find( obj => obj.name === 'loads' ).
getObjectByName( loadPattern ).getObjectByName( 'frames' ).
getObjectByName( frame ) );

    // add distributed load to model
    model.children.find( obj => obj.name === "loads" ).getObjectByName(
loadPattern ).getObjectByName( 'frames' ).add( createGlobalLoadAtFrame
( loadPattern , frame ) );

    // set force scale
    setLoadForceScale( config[ 'load.force.scale' ] );

    resolve( "frame distributed load added" );
} else {
    if ( structure.load_patterns.hasOwnProperty( loadPattern ) ) {
        reject( new Error( "frame '" + frame + "' does not exist" ) );
    } else {
        reject( new Error( "load pattern '" + loadPattern + "' does not
exist" ) );
    }
}
});

return promise;
}

```

La función `createGlobalLoadAtFrame()` crea un objeto tipo `THREE.Group` al que se le agrega un objeto también tipo `THREE.Group`, al cual se le asigna el nombre `components`. Este objeto representa la carga distribuida en sus componentes con respecto al sistema de coordenadas global.

4.2.10. setFrameView()

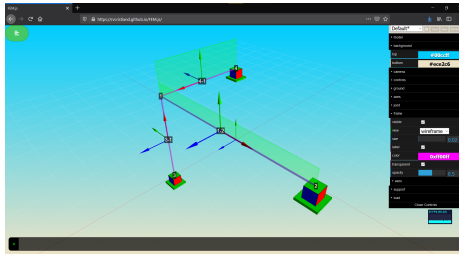
La función `setFrameView()` permite ver en el entorno gráfico los elementos aporticados del modelo como *palillos* o extruídos, según su sección transversal.

En el algoritmo 4.15 se presenta la implementación de la función `setFrameView()`. La función alterna el valor de la propiedad `visible` de los objetos `wireFrame` y `extrudeFrame` entre falso y verdadero, para presentar los elementos aporticados en la escena en una o en otra representación.

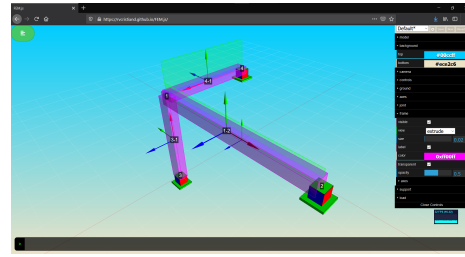
Algoritmo 4.15: Función `setFrameView()` implementada en el archivo `mas.js`.

```
export function setFrameView( view ) {  
  // set frame view  
  
  var promise = new Promise( ( resolve , reject ) => {  
    let wireframeView = view == 'wireframe', extrudeView = view == '  
    extrude';  
  
    if ( wireframeView || extrudeView ) {  
      model.getObjectByName( 'frames' ).children.forEach( frame => {  
        frame.getObjectByName( 'wireFrame' ).visible = wireframeView;  
        frame.getObjectByName( 'extrudeFrame' ).visible = extrudeView;  
      });  
  
      resolve( "" + view + " view setted" );  
    } else {  
      reject( new Error( "" + view + " does not exists" ) );  
    }  
  });  
  
  return promise;  
}
```

En la figura 4-9 se presenta `mas.js` después de ejecutar la función `open()` para abrir el archivo `space_frame.json`. Este archivo también ha sido generado con `pymas` para analizar el ejercicio 7.3 de Uribe-Escamilla (1995). El usuario es capaz de alternar la representación de los elementos aporticados con la barra de herramientas.



Estructura de palillo.



Estructura extruída.

Figura 4-9: Representación del `space_frame.json` en *estructura de palillos* o *extruído*.

4.2.11. `setSupportModel()`

La función `setSupportMode()` permite ver en el entorno gráfico los apoyos del modelo como objetos tridimensionales o como flechas, según los grados de libertad restringidos.

En el algoritmo 4.16 se presenta la implementación de la función `setSupportMode()`. La función alterna el valor de la propiedad `visible` de los objetos `analytical` y `space` entre falso y verdadero, para presentar los apoyos en la escena en una o en otra representación.

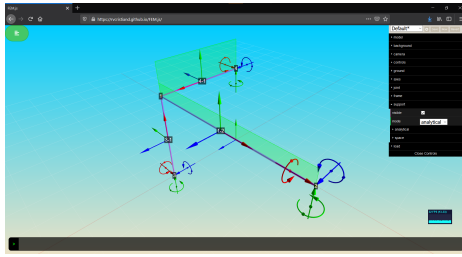
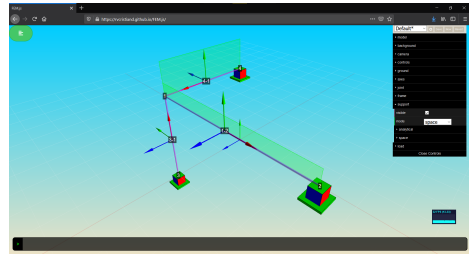
Algoritmo 4.16: Función `setSupportMode()` implementada en el archivo `mas.js`.

```
function setSupportMode( mode ) {
  // set support mode

  var support;

  Object.keys( structure.supports ).forEach( name => {
    support = model.getObjectByName( 'joints' ).getObjectByName( name ).
      getObjectByName( 'support' );
    support.getObjectByName( 'analytical' ).visible = ( mode == '
    analytical' );
    support.getObjectByName( 'space' ).visible = ( mode == 'space' );
  });
}
```

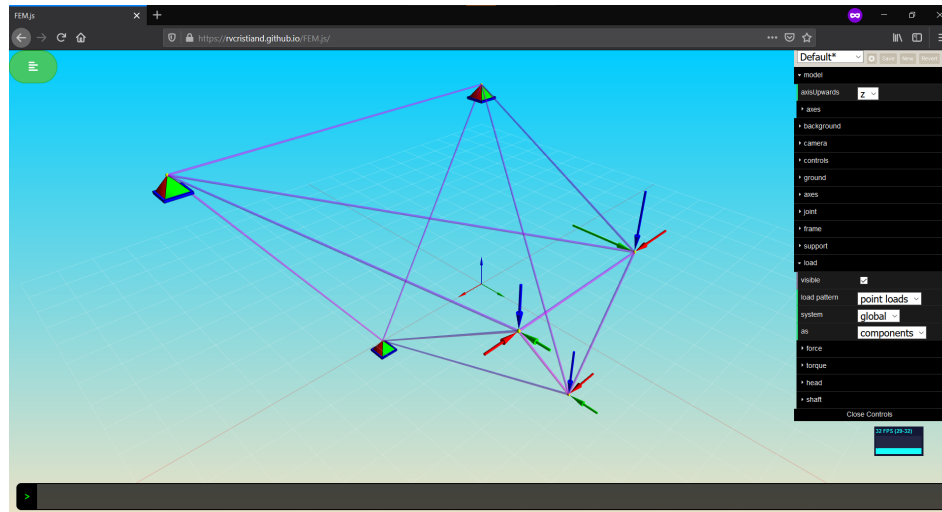
En la figura 4-10 vuelve y se presenta el archivo `space_truss.json` en `mas.js` donde se muestra las diferentes representaciones disponibles de los apoyos del modelo. El usuario es capaz de alternar la presentación de los apoyos con la barra de herramientas. Así mismo, es capaz de modificar las dimensiones de estos objetos.

Apoyos en modo *analytical*.Apoyos en modo *space*.**Figura 4-10:** Representación de los apoyos del archivo `space_frame.json` en `mas.js`

4.3. La variable `model`

Una vez el usuario haya agregado nodos, elementos aporricados y cargas, la variable `model` describe un grafo como el presentado en la figura 4-12.

Según (Threejsfundamentals authors, 2021), cada nodo de este grafo representa un sistema de coordenadas que se ubica en la escena con relación a su *nodo padre*. Por ejemplo, en la figura 4-11 se presenta el modelo después de indicarle a `mas.js` que orientar el modelo con el eje z local apuntando hacia la parte superior de la pantalla.

**Figura 4-11:** Eje local z de la variable `model` apuntando hacia arriba de la pantalla.

Esto es posible mediante la función `setModelRotation()`, definida en el archivo `mas.js`, que gira la variable `model` cierta cantidad alrededor de un eje que pasa por su origen, de

tal manera que uno de los ejes principales apunte hacia la parte de arriba de la pantalla.

Sin embargo, no es necesario modificar las demás variables agregadas a la variable `model` para que ocupen nuevas posiciones en la escena, ya que Three.js se encarga de calcular dichas posiciones en función de su ubicación relativa en el grafo.

La ubicación de los distintos nodos en el grafo presentado en la figura anterior obedece al nivel de dependencia que cada objeto tiene en la escena tridimensional. De esta manera, el grafo también permite visualizar la forma como se agrupan los objetos convenientemente para poder interactuar con ellos. Por ejemplo, en el algoritmo 4.17 se presenta la implementación de la función `setAxesShaftLength()`, que modifica la longitud de las colas de las *flechas* que representan un sistema coordinado. Un juego de estas flechas con el nombre `axes` se agrega a la variable `model` y a cada uno de los elementos aporticados (véase la figura 4-12).

Para modificar la longitud de las colas sólo es necesario modificar su longitud a lo largo de su eje local x y actualizar la posición de la cabeza. Esto es posible gracias a que tanto la cola como la cabeza de la flecha se agregan a un objeto intermedio, `x`, `y` o `z`, de tal manera que siempre están orientados a lo largo del eje x de dicho objeto.

Algoritmo 4.17: Implementación de la función `setAxesShaftLength()` del archivo `mas.js`.

```
function setAxesShaftLength( axes , length ) {  
  // set axes shaft length  
  
  axes.children.forEach( arrow => {  
    arrow.getObjectByName( 'straightShaft' ).scale.setX( length );  
    arrow.getObjectByName( 'head' ).position.setX( length );  
  });  
}
```

A través de la barra de herramientas, o de la variable correspondiente, el usuario puede cambiar las dimensiones de las flechas de los sistemas coordinados de la variable `model` y de los elementos aporticados. Varias funciones similares a esta se implementaron para interactuar con los objetos de la escena. A continuación se presentan algunas de las más relevantes.

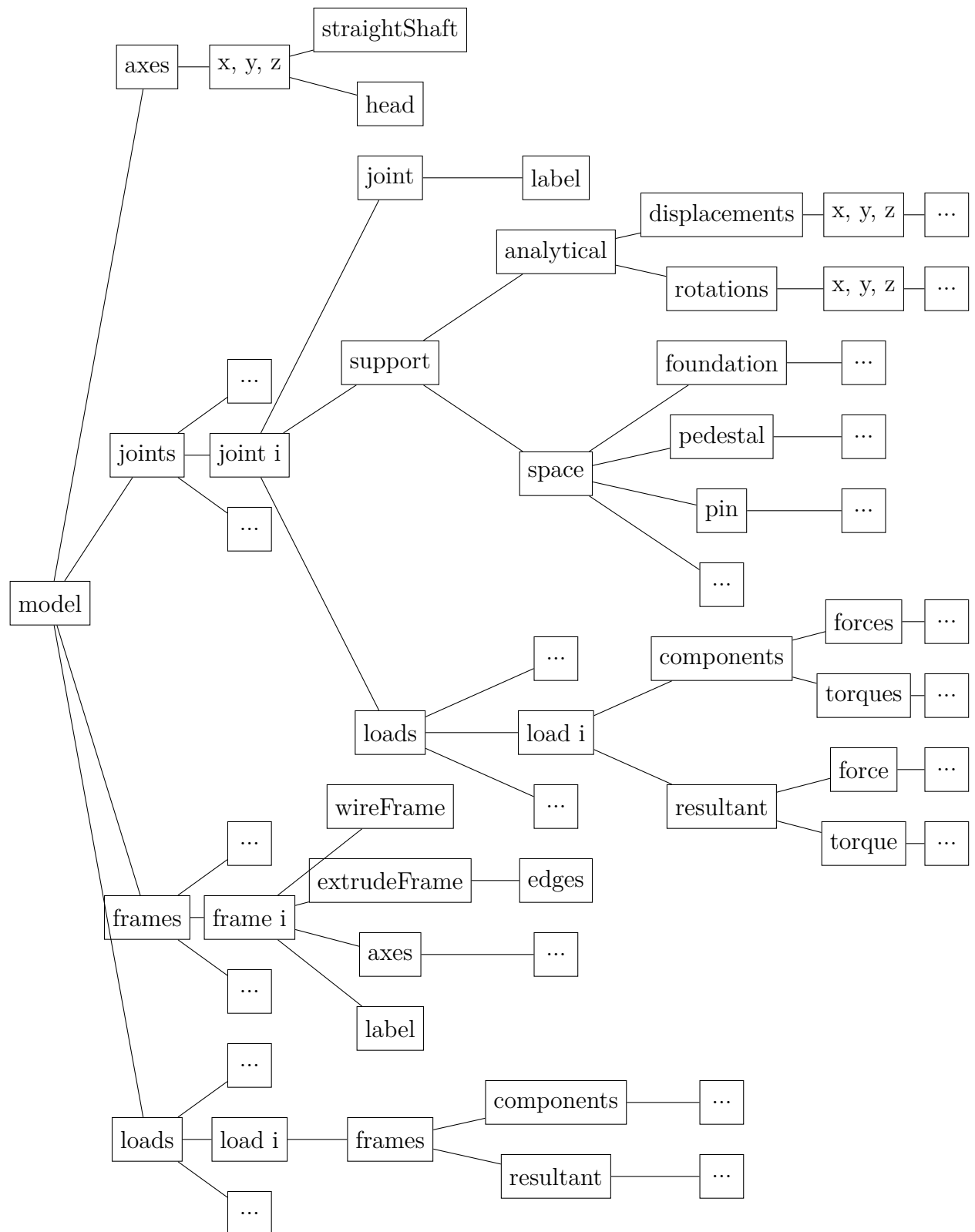


Figura 4-12: Grafo de la variable `model` después de agregar nodos, elementos aporticados y cargas al modelo.

5 Ejemplos de aplicación

Con el fin de mostrar el funcionamiento de los programas desarrollados, en este capítulo se presentan diferentes modelos de estructuras de elementos aporticados sometidos a cargas estáticas, los cuales se resuelven usando la librería **pymas** y los resultados se visualizan en **mas.js**.

5.1. Funcionamiento de la librería **pymas**

pymas es una librería programada en Python para analizar linealmente estructuras aporticadas tridimensionales sometidas a cargas estáticas. Una copia de esta librería se puede descargar de la página web del proyecto <https://github.com/rvcristiand/pymas>.

En la figura 5-1 se presenta la estructura de los archivos del programa. **pymas** utiliza otras librerías de Python para trabajar con matrices y cuaterniones. En específico, usa las librerías **numpy** y **scipy**. **numpy** es una librería que permite trabajar con arreglos multidimensionales, mientras que **scipy** es una librería de algoritmos matemáticos y de funciones convenientes construidas con **numpy**. Con **scipy**, una sesión interactiva de Python se convierte en un entorno de procesamiento de datos y creación de prototipos de sistemas que rivaliza con sistemas como MATLAB, IDL, Octave, R-Lab y SciLab (Harris *et al.*, 2020).

Para ejecutar **pymas** primero se debe instalar Python y dichas librerías. Existen varias formas de instalar Python, según el sistema operativo. Una vez instalado Python se pueden instalar las librerías **numpy** y **scipy** con **pip**, el instalador de paquetes de Python (van Rossum, 2019).

Una alternativa más sencilla consiste en instalar Anaconda Distribution, una distribución de Python y R para ciencias e ingeniería con más de 7.000 paquetes a código abierto. Anaconda Distribution se puede instalar en los sistemas operativos Windows, macOS y Linux. En la página de internet <https://docs.anaconda.com/anaconda/install/>

```
pymas/
├── LICENSE
├── README.md
├── examples/
│   ├── 01_simplest_beam/
│   │   ├── makepath.py
│   │   ├── simplest_beam.json
│   │   └── simplest_beam.py
│   ├── 02_space_truss/
│   │   ├── makepath.py
│   │   ├── space_truss.json
│   │   └── space_truss.py
│   ├── 03_space_frame/
│   │   ├── makepath.py
│   │   ├── space_frame.json
│   │   └── space_frame.py
│   └── 04_mas.js
│       ├── makepath.py
│       ├── mas.js.py
│       ├── space_truss.json
│       └── space_truss-solved.json
└── src/
    ├── pymas/
    │   ├── __init__.py
    │   ├── classtools.py
    │   ├── core.py
    │   └── primitives.py
```

Figura 5-1: Estructura de los archivos del programa pymas.

se encuentran los detalles del proceso de instalación para cada sistema operativo.

Una vez se ha instalado Python y las librerías que utiliza **pymas**, sólo hace falta indicarle al intérprete de Python el lugar donde se encuentra el código fuente del programa. En el caso de los ejemplos almacenados en la carpeta **examples**, se creó un archivo llamado **makepath.py** para tal fin, que agrega la ubicación del código fuente de **pymas** en la variable **sys.path**. La variable **sys.path** contiene la lista de directorios en la que el intérprete buscará por las librerías solicitadas. En el algoritmo 5.1 se presenta el contenido de este archivo.

Algoritmo 5.1: Archivo `makepath` de la carpeta `examples` del repositorio `pymas`.

```
import sys

sys.path.append('/home/rvcristiand/Documents/pymas/src/')
```

Quien desee ejecutar los ejemplos de la carpeta `examples`, debe cambiar la ruta que aparece en el archivo `makepath.py` por la ruta donde tiene almacenado el código fuente de `pymas`. Una vez hecho todo esto, ahora si el usuario le podrá indicar al intérprete de Python que ejecute alguno de los ejemplos.

A continuación, se presentan algunos de los ejemplos que hacen parte del repositorio de `pymas`.

5.1.1. Pórtico tridimensional

En el libro *Microcomputadores en Ingeniería Estructural*, del ingeniero *Jairo Uribe Escamilla* (Uribe-Escamilla, 1995), se presenta el ejemplo y la solución del modelo de un pórtico tridimensional. A continuación se muestra la solución de dicho ejemplo mediante el uso de la librería `pymas`.

Ejemplo 7.6 - *Resuelva matricialmente el pórtico de la figura.*

Solución - La figura a la cual hace referencia el ejemplo 7.6 se presenta en la figura 5-2. En ésta se presenta la geometría del modelo de la estructura, así como las propiedades del material, las cargas a las que están sometidos y las condiciones de apoyo.

Para solucionar el ejemplo anterior usando la librería `pymas`, se ingresa la información necesaria para describir completamente el modelo de la estructura instanciando un objeto de la clase `Structure`. Dicha información consiste en: (1) los materiales, (2) las secciones transversales, (3) los nodos, (4) los elementos tipo pórtico, (5) los apoyos y (6) las cargas. En el algoritmo 5.2 se presenta las instrucciones que debe recibir `pymas` para solucionar el modelo de la estructura.

Algoritmo 5.2: Ingreso de los datos del modelo de la estructura a `pymas`.

```
import makepath

from pymas import Structure
```

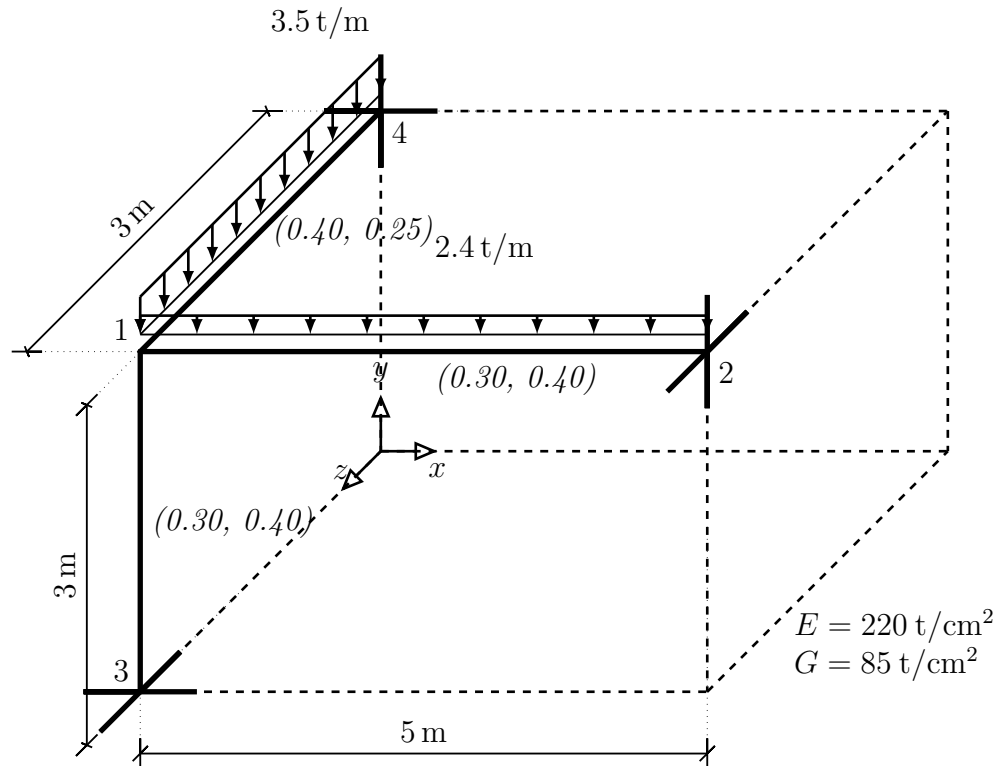


Figura 5-2: Pórtico tridimensional del *Ejemplo 7.6* de Uribe-Escamilla, 1995.

```

"""Solution to problem 7.6 from 'Microcomputadores en Ingeniería
Estructural'"""
# structure
model= Structure(True, True, True, True, True, True)

# add material
model.add_material('material1', 220e4, 85e4)

# add sections
model.add_section('section1', 0.12, 1.944e-3, 9e-4, 1.6e-3)
model.add_section('section2', 0.10, 1.2734e-3, 1.333e-3, 5.208e-4)

# add nodes
model.add_joint('1', 0, 3, 3)
model.add_joint('2', 5, 3, 3)
model.add_joint('3', 0, 0, 3)
model.add_joint('4', 0, 3, 0)

# add frames
model.add_frame('1-2', '1', '2', 'material1', 'section1')

```

```

model.add_frame('4-1', '4', '1', 'material1', 'section2')
model.add_frame('3-1', '3', '1', 'material1', 'section1')

# add supports
model.add_support('2', (True, True, True, True, True, True))
model.add_support('3', (True, True, True, True, True, True))
model.add_support('4', (True, True, True, True, True, True))

# add load pattern
model.add_load_pattern("distributed loads")

# add distributed loads
model.add_distributed_load('distributed loads', '1-2', fy=-2.4)
model.add_distributed_load('distributed loads', '4-1', fy=-3.5)

# solve
model.solve()

# export
model.export('space_frame.json')

```

Las instrucciones consisten en importar los archivos `makepath.py` y `core.py`, instanciar un objeto de la clase **Structure**, al cual se le ha dado el nombre de *structure* y, seguidamente, agregar los materiales, las secciones, los nodos, los elementos tipo pórtico, los apoyos, los patrones de carga y las cargas en los elementos. Una vez los datos del modelo de la estructura han sido ingresado, se puede solucionar el modelo mediante la instrucción `solve()`.

Para ejecutar este programa basta con indicarle a Python la ubicación del mismo. En el algoritmo 5.3 se presentan las instrucciones que se le pasaron al intérprete para solucionar el modelo.

Algoritmo 5.3: Ejecutar el archivo `space_frame.py` con el intérprete de Python.

```
$ python3 /path/to/space_frame.py
```

Cuando se ejecuta la instrucción `structure.solve()`, **pymas** comienza a solucionar el modelo de la estructura con base en la información ingresada, y almacena los resultados en la variable **structure**. Finalmente, con la función `export()` se guarda la información del modelo de la estructura y los resultados del análisis. A continuación se realiza la comparación entre los resultados obtenidos contra los resultados presentados en Uribe-Escamilla (1995).

Desplazamientos de los nodos

En la tabla 5-1 se presenta el desplazamientos de los nodos del modelo de la estructura para el patrón de carga *distributed loads*, los cuales son iguales a los presentados en Uribe-Escamilla, 1995.

Nodo	1	2	3	4
u_x [m]	2.687×10^{-5}	0	0	0
u_y [m]	-1.157×10^{-4}	0	0	0
u_z [m]	-1.001×10^{-5}	0	0	0
r_x [rad]	-5.669×10^{-4}	0	0	0
r_y [rad]	7.905×10^{-6}	0	0	0
r_z [rad]	-6.309×10^{-4}	0	0	0

Tabla 5-1: Desplazamientos en los nodos del modelo para el patrón de carga *distributed loads*.

Reacciones de los apoyos

En la tabla 5-2 se presenta las reacciones de los apoyos del modelo de la estructura para el patrón de carga *distributed loads*, las cuales son practicamente iguales a las presentadas por Uribe-Escamilla (1995).

Nodo	2	3	4
F_x [t]	-1.419	1.438	-0.02
F_y [t]	6.572	1.019×10^1	5.742
F_z [t]	5.658×10^{-3}	-7.394×10^{-1}	0.734
M_x [t m]	1.873×10^{-1}	-7.350×10^{-1}	-3.146
M_y [t m]	1.102×10^{-2}	-4.354×10^{-3}	-0.037
M_z [t m]	-5.986	-1.417	0.228

Tabla 5-2: Reacciones de los apoyos del modelo para el patrón de carga *distributed loads*.

A pesar que en este documento sólo se muestran los resultados de desplazamientos en los nodos y reacciones en los apoyos, el usuario tiene acceso a las variables y funciones trabajadas en el ejemplo anterior. Con dichas funciones y la versatilidad del lenguaje de programación Python, es posible proceder a realizar cálculos adicionales.

En vez de revisar uno a uno los valores de las fuerzas internas y los desplazamientos a lo largo de los elementos aporticados, vamos a revisar los resultados obtenidos visualizándolos en `mas.js`.

5.2. Funcionamiento del programa de computador `mas.js`

`mas.js` es un programa de computador desarrollado en JavaScript para visualizar modelos de estructuras aporticadas. Una copia de este programa se puede descargar de la página web del proyecto <https://github.com/rvcristiand/mas.js.git>.

En la figura 5-3 se presenta la estructura de los archivos del programa.

```
mas.js/  
├── LICENSE  
├── README.md  
├── css/  
│   └── style.css  
├── space_frame.json  
├── index.html  
├── libs/  
│   ├── CSS2DRenderer.js  
│   ├── OrbitControls.js  
│   ├── Projector.js  
│   ├── dat.gui.min.js  
│   ├── stats.js  
│   └── three.js  
├── main.js  
└── modules/  
    ├── mas.js  
    └── terminal.js
```

Figura 5-3: Estructura de los archivos del programa `mas.js`.

Para ejecutar `mas.js` hace falta un servidor local, para poder leer archivos alojados en el computador desde el navegador web. Se puede crear un servidor local con Python, usando la librería estándar `html`. Esta librería permite implementar servidores HTML localmente. En el algoritmo 5.4 se presentan las instrucciones necesarias para ejecutar

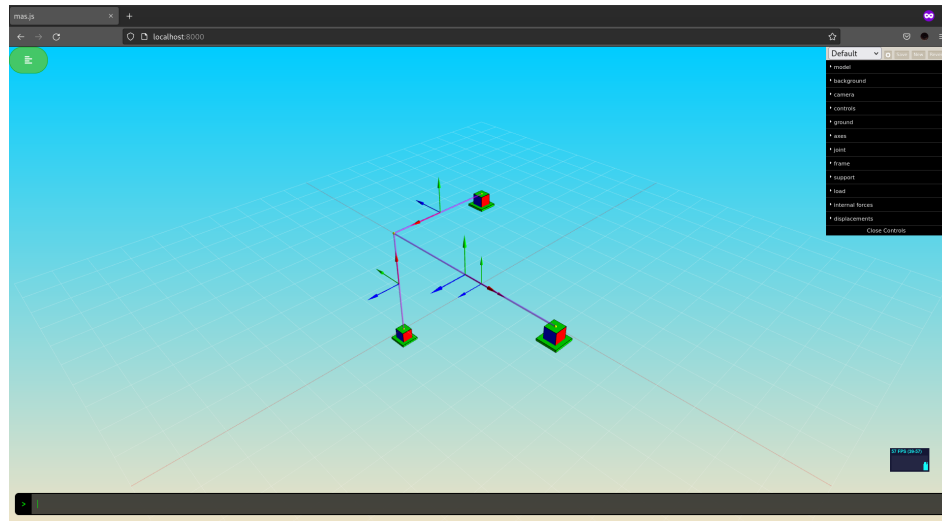


Figura 5-4: El programa de computador `mas.js` ejecutándose en el navegador de internet Firefox.

un servidor local con Python. Por defecto, el servidor local está escuchando peticiones en el puerto 8000. Para poder ver el programa `mas.js` desde el navegador, el usuario puede ir a la dirección `http://localhost:8000`. En la figura 5-4 se presenta `mas.js` en el navegador de internet Firefox.

Algoritmo 5.4: Ejecutar un servidor local con Python para abrir el programa `mas.js`.

```
$ cd /to/folder/mas.js  
$ python3 -m http.server
```

Una vez se ha ejecutado el servidor local, el usuario es capaz de modelar sus propias estructuras con la terminal que se encuentra en la parte inferior de la pantalla, o también puede abrir archivos previamente guardados. En este último caso, los archivos deben estar ubicados en el directorio raíz del programa `mas.js`.

En dicha carpeta se encuentra ubicado el archivo `space_frame.json`, el cual fue generado con el algoritmo 5.2. Así mismo, la terminal de programa viene con una instrucción por defecto, `open("space_frame.json")`, la cual puede ser ejecutada para ver el archivo con `mas.js`. En la figura 5-5 se presenta el programa `mas.js` después de abrir el archivo `space_frame.json`.

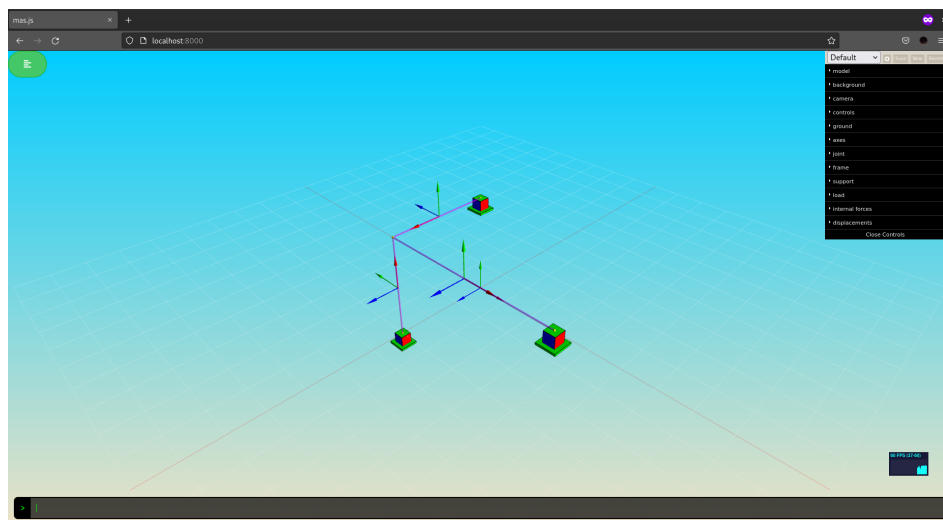
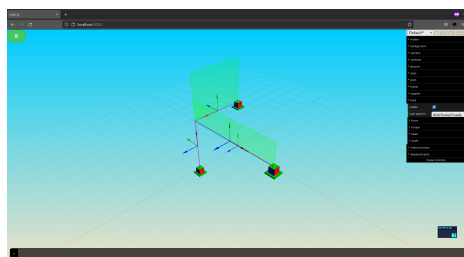
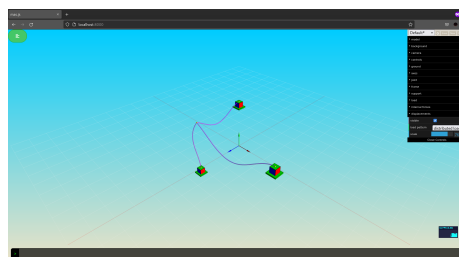


Figura 5-5: El archivo `space_frame.json` en el programa de computador `mas.js`.

El usuario puede, entre otras cosas, ver las cargas externas, las fuerzas internas o la estructura deformada de cada patrón de carga, por medio de la barra de herramientas a la derecha de la pantalla. Por ejemplo, en la figura 5-6 se presenta las cargas a las que está sometida la estructura y la condición deformada.



Cargas externas.



Estructura deformada.

Figura 5-6: Cargas externas actuando en la estructura aporticada y la estructura deformada debida a dichas cargas.

Con la barra de herramientas de `mas.js` es posible escalar algunos elementos de la escena tridimensional. Por ejemplo, en la figure 5-7 se presentan los momentos flectores z a lo largo de los elementos aporticados con diferentes escalas.

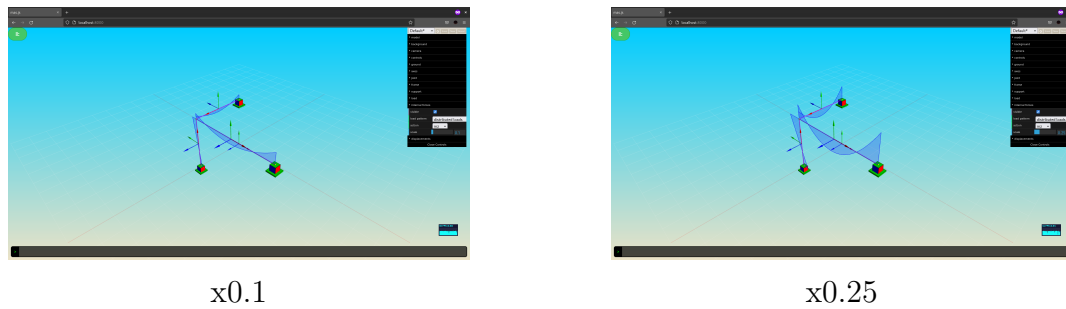


Figura 5-7: Momentos flectores z a lo largo de los elementos aporticados con diferentes escalas.

Con `mas.js` se pueden modelar las estructuras previamente presentadas, o leer los archivos en formato JSON generados con `pymas`. A continuación se presenta algunos ejemplos prácticos.

5.2.1. Cercha tridimensional

En la figura 5-8 se presenta una cercha tridimensional modelada con `mas.js`. Esta estructura puede ser modelada a través de los comandos que `mas.js` tiene disponible.

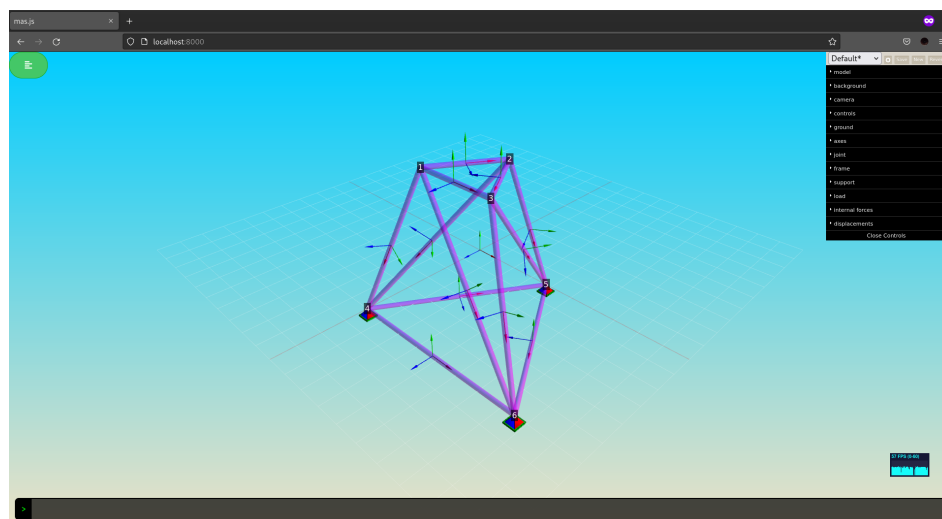


Figura 5-8: Cercha tridimensional en `mas.js`.