ANEXOS

Anexo 1: Script modelo_estructura.py

```
"""
Módulo: modelo_estructura.py
Descripción: Base de datos central del programa. Define las clases y estructuras
de datos para nodos, elementos 1D (vigas y columnas), materiales, secciones y
apoyos necesarios para el Método Matricial de Rigidez.
"""

# ------------------------------------------------------------------------
# VORTEX 3D - Análisis Estructural de Edificaciones 3D y Diseño Normativo
# de Vigas y Columnas
# Copyright (C) 2026 Diego Oliver Vargas Moya & Luis Alberto Ortiz Morales
#
# Este programa es software libre: puedes redistribuirlo y/o modificarlo
# bajo los términos de la Licencia Pública General GNU (GNU GPL) publicada
# por la Free Software Foundation, ya sea la versión 3 de la Licencia, o
# (a tu elección) cualquier versión posterior.
#
# Este programa se distribuye con la esperanza de que sea útil,
# pero SIN GARANTÍA ALGUNA; ni siquiera la garantía implícita
# MERCANTIL o de APTITUD PARA UN PROPÓSITO DETERMINADO.
# Consulta los detalles de la Licencia Pública General GNU para más
# información.
#
# Deberías haber recibido una copia de la Licencia Pública General GNU
# junto a este programa. En caso contrario, consulta
# <https://www.gnu.org/licenses/>.
# ------------------------------------------------------------------------

#====================================================================
# I. IMPORTACIONES Y DEPENDENCIAS
#====================================================================
import json
import numpy as np
import math
from typing import Optional
from procesador_cargas import CombinacionCarga, ProcesadorCargas

#====================================================================
# II. ESTRUCTURA DE DATOS PRINCIPAL DEL MODELO MATEMÁTICO
#====================================================================
class Estructura:
    """Encapsula todos los datos y operaciones de la estructura."""
    def __init__(self):
        # 1. Inyección de dependencias para el motor de procesamiento de cargas
        from procesador_cargas import ProcesadorCargas
        self.procesador_cargas = ProcesadorCargas(self)
        self.reiniciar()

    def reiniciar(self):
        """Restablece todos los datos a un estado vacío."""
        self.nodos = {}
        self.elementos = {}
        self.materiales = {}
        self.apoyos = {}
        self.losas = {}
```

```
        self.hipotesis_de_carga = {}
        self.cargas_nodales = []
        self.cargas_elementos = []
        self.cargas_superficiales = {}
        self.siguiente_id_carga = 1
        self.siguiente_id_carga_sup = 1
        self.combinaciones = ProcesadorCargas(self).combinaciones_norma.copy()
        self.combinaciones_usuario = []
        self.archivo_actual = None
        self.modificado = False
        self.desplazamientos = None
        self.reacciones = None
        self.fuerzas_internas = None
        self.rigidez_global = None
        self.armados_diseno = {}
        self.resultados_calculo = None
        self.datos_reporte_losas = []

    def get_centro_geometrico(self):
        """Calcula el centro del cuadro delimitador de todos los nodos."""
        if not self.nodos:
            return np.array([0.0, 0.0, 0.0])

        coords = np.array(list(self.nodos.values()))
        min_coords = coords.min(axis=0)
        max_coords = coords.max(axis=0)
        return (min_coords + max_coords) / 2.0

    def get_siguiente_id(self, coleccion):
        """Obtiene el siguiente ID disponible en una colección (diccionario)."""
        if not coleccion:
            return 1
        return max(coleccion.keys()) + 1

#====================================================================
# III. GESTIÓN TOPOLÓGICA: NODOS Y ELEMENTOS UNIDIMENSIONALES
#====================================================================
    def agregar_nodo(self, id_nodo, coords):
        if id_nodo in self.nodos:
            raise ValueError(f"El ID de nodo {id_nodo} ya existe.")
        self.nodos[id_nodo] = coords
        self.modificado = True

    def actualizar_nodo(self, id_nodo, coords):
        if id_nodo not in self.nodos:
            raise ValueError(f"El nodo {id_nodo} no existe.")
        self.nodos[id_nodo] = coords
        self.modificado = True

    def eliminar_nodo(self, id_nodo):
        if id_nodo not in self.nodos:
            return
        del self.nodos[id_nodo]

       # 1. Eliminación en cascada de elementos y propiedades dependientes
       elementos_a_eliminar = [eid for eid, (ni, nj, _) in self.elementos.items() if ni == id_nodo or nj ==
id_nodo]
```

```
        for eid in elementos_a_eliminar:
            self.eliminar_elemento(eid)

        if id_nodo in self.apoyos: del self.apoyos[id_nodo]
        if id_nodo in self.cargas_nodales: del self.cargas_nodales[id_nodo]
        self.modificado = True

    def agregar_elemento(self, id_elemento, ni, nj, id_material):
        if id_elemento in self.elementos:
            raise ValueError(f"El ID de elemento {id_elemento} ya existe.")
        if ni not in self.nodos or nj not in self.nodos:
            raise ValueError("Uno o ambos nodos del elemento no existen.")
        if id_material not in self.materiales:
            raise ValueError("El material del elemento no existe.")
        if ni == nj:
            raise ValueError("Los nodos de inicio y fin no pueden ser iguales.")
        self.elementos[id_elemento] = (ni, nj, id_material)
        self.modificado = True

    def actualizar_elemento(self, id_elemento, ni, nj, id_material):
        if id_elemento not in self.elementos: return
        if ni not in self.nodos or nj not in self.nodos: raise ValueError("Nodos no
válidos.")
        if id_material not in self.materiales: raise ValueError("Material no válido.")
        self.elementos[id_elemento] = (ni, nj, id_material)
        self.modificado = True

    def eliminar_elemento(self, id_elemento):
        if id_elemento not in self.elementos: return
        del self.elementos[id_elemento]
        if id_elemento in self.cargas_elementos: del self.cargas_elementos[id_elemento]
        self.modificado = True

    def actualizar_material_de_elementos(self, ids_elementos: list, id_material_nuevo: int):
        """
        Actualiza el material de una lista de elementos 1D (pórticos) de forma masiva.
        Verifica que el material sea compatible (no de placa).

        Args:
            ids_elementos (list): Lista de IDs de elementos a modificar.
            id_material_nuevo (int): ID del nuevo material a asignar.

        Returns:
            int: Cantidad de elementos que fueron exitosamente actualizados.

        Raises:
            ValueError: Si el material no existe.
        """

        # 1. Validar el material nuevo
        if id_material_nuevo not in self.materiales:
            raise ValueError(f"Error: El material ID {id_material_nuevo} no existe.")

        # 2. Iterar y actualizar
        ids_no_encontrados = []
        count_actualizados = 0
```

```
        for id_elem in ids_elementos:
            if id_elem in self.elementos:
                datos_actuales = self.elementos[id_elem]
                self.elementos[id_elem] = (datos_actuales[0], datos_actuales[1],
id_material_nuevo)

                count_actualizados += 1
            else:
                ids_no_encontrados.append(id_elem)

        # 3. Marcar como modificado y reportar
        if count_actualizados > 0:
            self.modificado = True

        if ids_no_encontrados:
            print(f"ADVERTENCIA (actualizar_material_de_elementos): No se encontraron los
elementos IDs: {ids_no_encontrados}")

        return count_actualizados

#====================================================================
# IV. GESTIÓN DE ELEMENTOS BIDIMENSIONALES: LOSAS
#====================================================================
    def _ordenar_nodos_placa(self, nodos_ids):
        """
        Garantiza el ordenamiento secuencial (perimetral) de los nodos
        pertenecientes a un elemento superficial para evitar auto-intersecciones.
        """
        if len(nodos_ids) != 4:
            return nodos_ids

        coords = np.array([self.nodos[nid] for nid in nodos_ids])

        centro = np.mean(coords, axis=0)

        # 1. Determinación del vector normal predominante
        v1 = coords[1] - coords[0]
        v2 = coords[2] - coords[0]
        normal = np.cross(v1, v2)

        # 2. Proyección sobre el plano principal para cálculo de ángulos polares
        if abs(normal[2]) > abs(normal[0]) and abs(normal[2]) > abs(normal[1]):
            idx1, idx2 = 0, 1
        elif abs(normal[1]) > abs(normal[0]):
            idx1, idx2 = 0, 2
        else:
            idx1, idx2 = 1, 2

        angulos = [math.atan2(p[idx2] - centro[idx2], p[idx1] - centro[idx1]) for p in
coords]

        nodos_ids_ordenados = [nid for _, nid in sorted(zip(angulos, nodos_ids))]
        return nodos_ids_ordenados

    def agregar_losa(self, id_losa, nodos_ids, distribucion, eje_uni=None, espesor=0.20,
peso_especifico=24.0):
        """
```

```
        Registra un elemento bidimensional en la base de datos, validando
        previamente la coplanaridad matemática de sus vértices.
        """
        if id_losa in self.losas:
            raise ValueError(f"El ID de losa {id_losa} ya existe.")
        if len(nodos_ids) != 4:
            raise ValueError("Una losa debe tener exactamente 4 nodos.")
        for nid in nodos_ids:
            if nid not in self.nodos:
                raise ValueError(f"El nodo {nid} no existe.")

            # 1. Verificación de coplanaridad mediante Descomposición en Valores Singulares
(SVD)
            tolerancia_coplanar = 0.01
            coords = np.array([self.nodos[nid] for nid in nodos_ids])
            coords_centradas = coords - coords.mean(axis=0)

            _, valores_singulares, vh = np.linalg.svd(coords_centradas)

            normal_plano = vh[2, :]
            distancias_al_plano = np.abs(np.dot(coords_centradas, normal_plano))

            if np.max(distancias_al_plano) > tolerancia_coplanar:
                raise ValueError(f"Los nodos de la losa {id_losa} no son coplanares
(desviación máxima > {tolerancia_coplanar} m).")

        nodos_ordenados = self._ordenar_nodos_placa(nodos_ids)

        self.losas[id_losa] = {
            'nodos': nodos_ordenados,
            'distribucion': distribucion,
            'eje_uni': eje_uni,
            'espesor': espesor,
            'peso_especifico': peso_especifico
        }
        self.modificado = True

    def actualizar_losa(self, id_losa, nodos_ids, distribucion, eje_uni=None, espesor=0.20,
peso_especifico=24.0):
        if id_losa not in self.losas:
            raise ValueError(f"La losa {id_losa} no existe.")
        if len(nodos_ids) != 4:
            raise ValueError("Una losa debe tener exactamente 4 nodos.")
        for nid in nodos_ids:
            if nid not in self.nodos:
                raise ValueError(f"El nodo {nid} no existe.")

            tolerancia_coplanar = 0.01
            coords = np.array([self.nodos[nid] for nid in nodos_ids])
            coords_centradas = coords - coords.mean(axis=0)
            _, _, vh = np.linalg.svd(coords_centradas)
            normal_plano = vh[2, :]
            distancias_al_plano = np.abs(np.dot(coords_centradas, normal_plano))

            if np.max(distancias_al_plano) > tolerancia_coplanar:
                raise ValueError(f"Los nodos de la losa {id_losa} no son coplanares
(desviación máxima > {tolerancia_coplanar} m).")
```

```
        nodos_ordenados = self._ordenar_nodos_placa(nodos_ids)

        self.losas[id_losa]['nodos'] = nodos_ordenados
        self.losas[id_losa]['distribucion'] = distribucion
        self.losas[id_losa]['eje_uni'] = eje_uni
        self.losas[id_losa]['espesor'] = espesor
        self.losas[id_losa]['peso_especifico'] = peso_especifico
        self.modificado = True

    def eliminar_losa(self, id_losa):
        if id_losa in self.losas:
            del self.losas[id_losa]
            self.modificado = True

    def actualizar_propiedades_losas_lote(self,
                                          ids_losas: list,
                                          distribucion: Optional[str] = None,
                                          eje_uni: Optional[str] = None,
                                          espesor: Optional[float] = None,
                                          peso_especifico: Optional[float] = None) -> int:
        """
        Actualiza propiedades seleccionadas (distribución, espesor, PE)
        de una lista de losas de forma masiva.
        Las propiedades enviadas como None no se modificarán.

        Args:
            ids_losas (list): Lista de IDs de losas a modificar.
            distribucion (str, optional): 'bidireccional' o 'unidireccional'.
            eje_uni (str, optional): 'Global X' o 'Global Y'.
            espesor (float, optional): Nuevo valor de espesor.
            peso_especifico (float, optional): Nuevo valor de PE.

        Returns:
            int: Cantidad de losas que fueron exitosamente actualizadas.
        """
        # 1. Validación de consistencia de los parámetros de entrada
        if distribucion is not None:
            if distribucion == 'unidireccional' and eje_uni not in ['Global X', 'Global Y']:
                raise ValueError("El eje de distribución debe ser 'Global X' o 'Global Y'
para el tipo unidireccional.")
            if distribucion == 'bidireccional' and eje_uni is not None:
                 raise ValueError("No se puede especificar un eje para el tipo
bidireccional.")

        if espesor is not None and espesor <= 0:
            raise ValueError("El espesor debe ser un valor positivo.")

        if peso_especifico is not None and peso_especifico < 0:
             raise ValueError("El peso específico no puede ser negativo.")

        count_actualizados = 0

        # 2. Iteración y actualización de parámetros
        for id_losa in ids_losas:
            if id_losa in self.losas:
                if distribucion is not None:
```

```
                    self.losas[id_losa]['distribucion'] = distribucion
                    self.losas[id_losa]['eje_uni'] = eje_uni

                if espesor is not None:
                    self.losas[id_losa]['espesor'] = espesor

                if peso_especifico is not None:
                    self.losas[id_losa]['peso_especifico'] = peso_especifico

                count_actualizados += 1

        if count_actualizados > 0:
            self.modificado = True

        return count_actualizados

#================================================================
# V. GESTIÓN DE MATERIALES Y PROPIEDADES MECÁNICAS
#================================================================
    def agregar_material(self, id_mat, descripcion, tipo_seccion, props,
peso_especifico=0.0):
        if id_mat in self.materiales:
            raise ValueError(f"El material con ID {id_mat} ya existe.")

        if tipo_seccion == 'general':
            if len(props) != 8:
                raise ValueError("Las propiedades para 'general' deben ser 8 valores
positivos (E, G, A, J, Iy, Iz, Ay, Az).")

        elif not all(p > 0 for p in props if p is not None):
             raise ValueError("Las propiedades deben ser valores positivos.")

        self.materiales[id_mat] = {
            'tipo': tipo_seccion,
            'descripcion': descripcion,
            'propiedades': props,
            'peso_especifico': peso_especifico
        }
        self.modificado = True

    def actualizar_material(self, id_mat, descripcion, tipo_seccion, props,
peso_especifico=0.0):
        if id_mat not in self.materiales:
            raise ValueError(f"El material con ID {id_mat} no existe.")

        if tipo_seccion == 'general':
            if len(props) != 8:
                raise ValueError("Las propiedades para 'general' deben ser 8 valores
positivos (E, G, A, J, Iy, Iz, Ay, Az).")

        elif not all(p > 0 for p in props if p is not None):
             raise ValueError("Las propiedades deben ser valores positivos.")

        self.materiales[id_mat] = {
            'tipo': tipo_seccion,
            'descripcion': descripcion,
            'propiedades': props,
```

```
            'peso_especifico': peso_especifico
        }
        self.modificado = True

    def get_propiedades_calculadas(self, id_mat):
        """
        Extrae y computa las constantes mecánicas y geométricas de la sección
        transversal para su integración en la matriz de rigidez [Hibbeler, 2017].
        """
        if id_mat not in self.materiales:
            raise ValueError(f"Material inexistente ID {id_mat}.")

        material = self.materiales[id_mat]
        tipo = material.get('tipo', 'rectangular')

        print(f"--- Obteniendo propiedades para Material ID: {id_mat} (Tipo: {tipo}) ---")

        # 1. Determinación de inercias y módulos para secciones rectangulares estándar
        if tipo == 'rectangular':
            E, nu, b, h, *_ = material['propiedades']
            G = E / (2 * (1 + nu))
            A = b * h
            Iy = (b * (h**3)) / 12
            Iz = (h * (b**3)) / 12
            a, B = (h, b) if h > b else (b, h)
            J = a * (B**3) * (1/3 - 0.21 * (B/a) * (1 - (B**4) / (12 * a**4)))
            Ay = 5/6 * A
            Az = Ay
            print("=============== MATERIAL TIPO RECTANGULAR ===============")
            print(f"G={G:.8e}; E={E:.8e}    b={b}, h={h} -> Iy={Iy:.8e}, Iz={Iz:.8e},
A={A:.8e}, J={J:.8e}, Ay={Ay:.8e}, Az={Az:.8e}")
            print("=========================================================")
            return (E, G, A, J, Iy, Iz, Ay, Az)

        # 2. Extracción de propiedades explícitas para perfiles generales
        elif tipo == 'general':
            props = material['propiedades']
            if len(props) != 8:
                print(f"    ADVERTENCIA: El material 'general' ID {id_mat} tiene 6
propiedades. Se asumirá Ay=Az=A.")
                E, G, A, J, Iy, Iz = props
                Ay, Az = A, A
                return (E, G, A, J, Iy, Iz, Ay, Az)
            print("=============== MATERIAL TIPO GENERAL ===================")
            print(f"    Propiedades generales (8): {props}")
            print("=========================================================")
            return material['propiedades']

        else:
            raise TypeError(f"Tipo de material desconocido: {tipo}")

    def eliminar_material(self, id_material):
        if id_material not in self.materiales: return
        del self.materiales[id_material]
        elementos_a_eliminar = [eid for eid, (_, _, mid) in self.elementos.items() if mid ==
id_material]
        for eid in elementos_a_eliminar:
```

```
            self.eliminar_elemento(eid)
        self.modificado = True

#====================================================================
# VI. CONDICIONES DE CONTORNO Y ESTADOS DE CARGA
#====================================================================
    def agregar_o_actualizar_apoyo(self, id_nodo, restricciones):
        if id_nodo not in self.nodos:
            raise ValueError(f"El nodo {id_nodo} no existe.")
        self.apoyos[id_nodo] = restricciones
        self.modificado = True

    def eliminar_apoyo(self, id_nodo):
        if id_nodo in self.apoyos:
            del self.apoyos[id_nodo]
            self.modificado = True

    def agregar_hipotesis(self, nombre, tipo):
        if any(h['nombre'] == nombre for h in self.hipotesis_de_carga.values()):
            raise ValueError(f"Ya existe una hipótesis con el nombre '{nombre}'.")

        nuevo_id = self.get_siguiente_id(self.hipotesis_de_carga)
        self.hipotesis_de_carga[nuevo_id] = {'nombre': nombre, 'tipo': tipo}
        self.modificado = True
        return nuevo_id

    def eliminar_hipotesis(self, id_hipotesis):
        if id_hipotesis not in self.hipotesis_de_carga:
            return

        del self.hipotesis_de_carga[id_hipotesis]

        self.cargas_nodales = [c for c in self.cargas_nodales if c['id_hipotesis'] !=
id_hipotesis]
        self.cargas_elementos = [c for c in self.cargas_elementos if c['id_hipotesis'] !=
id_hipotesis]

        self.modificado = True

    def actualizar_hipotesis(self, id_hipotesis, nuevo_nombre, nuevo_tipo):
        if id_hipotesis not in self.hipotesis_de_carga:
            raise ValueError(f"La hipótesis con ID {id_hipotesis} no existe.")

        if any(h['nombre'] == nuevo_nombre for id_h, h in self.hipotesis_de_carga.items() if
id_h != id_hipotesis):
            raise ValueError(f"El nombre '{nuevo_nombre}' ya está en uso por otra
hipótesis.")

        self.hipotesis_de_carga[id_hipotesis]['nombre'] = nuevo_nombre
        self.hipotesis_de_carga[id_hipotesis]['tipo'] = nuevo_tipo
        self.modificado = True

    def agregar_carga_nodal(self, id_nodo, id_hipotesis, vector_carga):
        if id_nodo not in self.nodos:
            raise ValueError(f"El nodo {id_nodo} no existe.")
        if id_hipotesis not in self.hipotesis_de_carga:
            raise ValueError(f"La hipótesis de carga con ID {id_hipotesis} no existe.")
```

```
        nueva_carga = {
            'id_carga': self.siguiente_id_carga,
            'id_nodo': id_nodo,
            'id_hipotesis': id_hipotesis,
            'vector': vector_carga
        }
        self.cargas_nodales.append(nueva_carga)
        self.siguiente_id_carga += 1
        self.modificado = True

    def agregar_carga_elemento(self, id_elemento, id_hipotesis, datos_carga):
        if id_elemento not in self.elementos:
            raise ValueError(f"El elemento {id_elemento} no existe.")
        if id_hipotesis not in self.hipotesis_de_carga:
            raise ValueError(f"La hipótesis de carga con ID {id_hipotesis} no existe.")

        nueva_carga = {
            'id_carga': self.siguiente_id_carga,
            'id_elemento': id_elemento,
            'id_hipotesis': id_hipotesis,
            'datos_carga': datos_carga
        }
        self.cargas_elementos.append(nueva_carga)
        self.siguiente_id_carga += 1
        self.modificado = True

    def eliminar_carga(self, id_carga):
        """Elimina una carga individual (nodal o de elemento) por su ID único."""
        len_inicial = len(self.cargas_nodales) + len(self.cargas_elementos)

        self.cargas_nodales = [c for c in self.cargas_nodales if c.get('id_carga') !=
id_carga]
        self.cargas_elementos = [c for c in self.cargas_elementos if c.get('id_carga') !=
id_carga]

        len_final = len(self.cargas_nodales) + len(self.cargas_elementos)

        if len_final < len_inicial:
            self.modificado = True

    def agregar_o_actualizar_carga_superficial(self, id_carga_sup, id_losa, id_hipotesis,
magnitud_wz):
        """Agrega o actualiza una carga superficial definida sobre una losa."""
        if id_losa not in self.losas:
            raise ValueError(f"La losa con ID {id_losa} no existe.")
        if id_hipotesis not in self.hipotesis_de_carga:
            raise ValueError(f"La hipótesis de carga con ID {id_hipotesis} no existe.")

        if id_carga_sup is None or id_carga_sup not in self.cargas_superficiales:
            nuevo_id = self.get_siguiente_id(self.cargas_superficiales)
            self.cargas_superficiales[nuevo_id] = {
                'id_carga': nuevo_id,
                'id_losa': id_losa,
                'id_hipotesis': id_hipotesis,
                'magnitud': magnitud_wz
            }
```

```
            self.modificado = True
            return nuevo_id
        else:
            self.cargas_superficiales[id_carga_sup]['id_losa'] = id_losa
            self.cargas_superficiales[id_carga_sup]['id_hipotesis'] = id_hipotesis
            self.cargas_superficiales[id_carga_sup]['magnitud'] = magnitud_wz
            self.modificado = True
            return id_carga_sup

    def eliminar_carga_superficial(self, id_carga_sup):
        """Elimina una carga superficial por su ID."""
        if id_carga_sup in self.cargas_superficiales:
            del self.cargas_superficiales[id_carga_sup]
            self.modificado = True

#====================================================================
# VII. PERSISTENCIA DE DATOS Y SERIALIZACIÓN
#====================================================================
    def cargar_desde_archivo(self, ruta_archivo):
        """
        Deserializa el estado estructural desde un archivo JSON, reconstruyendo
        el modelo en memoria RAM [Sommerville, 2011].
        """
        try:
            with open(ruta_archivo, 'r') as f:
                datos = json.load(f)

            self.reiniciar()

            # 1. Reconstrucción de diccionarios con casteo de claves a enteros
            self.nodos = {int(k): tuple(v) for k, v in datos.get('nodos', {}).items()}
            self.elementos = {int(k): tuple(v) for k, v in datos.get('elementos',
{}).items()}
            self.materiales = {int(k): v for k, v in datos.get('materiales', {}).items()}
            self.apoyos = {int(k): v for k, v in datos.get('apoyos', {}).items()}
            self.losas = {int(k): v for k, v in datos.get('losas', {}).items()}
            self.cargas_superficiales = {int(k): v for k, v in
datos.get('cargas_superficiales', {}).items()}
            self.armados_diseno = {int(k): v for k, v in datos.get('armados_diseno',
{}).items()}

            self.hipotesis_de_carga = {int(k): v for k, v in datos.get('hipotesis_de_carga',
{}).items()}
            self.cargas_nodales = datos.get('cargas_nodales', [])
            self.cargas_elementos = datos.get('cargas_elementos', [])

            # 2. Recalcular el siguiente ID de carga disponible para evitar colisiones
            max_id = 0
            all_cargas = self.cargas_nodales + self.cargas_elementos
            if all_cargas:
                max_id = max(c.get('id_carga', 0) for c in all_cargas)
            self.siguiente_id_carga = max_id + 1

            max_id_sup = 0
            if self.cargas_superficiales:
                max_id_sup = max(self.cargas_superficiales.keys())
            self.siguiente_id_carga_sup = max_id_sup + 1
```

```
            combos_cargados = datos.get('combinaciones', [])
            if combos_cargados:
                 self.combinaciones = [CombinacionCarga.desde_dict(c) for c in
combos_cargados]

            self.archivo_actual = ruta_archivo
            self.modificado = False
        except Exception as e:
            raise IOError(f"No se pudo cargar el archivo (puede que el formato sea
incompatible): {e}")

    def guardar_en_archivo(self, ruta_archivo):
        """
        Serializa la instancia actual de la estructura y la exporta
        al sistema de archivos en formato JSON.
        """
        try:
            datos = {
                'nodos': self.nodos, 'elementos': self.elementos, 'materiales':
self.materiales,
                'losas': self.losas, 'apoyos': self.apoyos,
                'hipotesis_de_carga': self.hipotesis_de_carga,
                'cargas_nodales': self.cargas_nodales,
                'cargas_elementos': self.cargas_elementos,
                'cargas_superficiales': self.cargas_superficiales,
                'armados_diseno': self.armados_diseno,
                'combinaciones': [c.to_dict() for c in self.combinaciones]
            }
            with open(ruta_archivo, 'w') as f:
                json.dump(datos, f, indent=4)
            self.archivo_actual = ruta_archivo
            self.modificado = False
        except Exception as e:
            raise IOError(f"No se pudo guardar el archivo: {e}")
```

Anexo 2: Script calc.py

```
"""
Módulo: calc.py
Descripción: Subrutinas matemáticas y utilidades de cálculo matricial para el
motor de rigidez. Contiene las definiciones de matrices de rigidez local y
matrices de transformación de coordenadas para elementos 1D espaciales.
"""

# ------------------------------------------------------------------------------
# VORTEX 3D - Análisis Estructural de Edificaciones 3D y Diseño Normativo
# de Vigas y Columnas
# Copyright (C) 2026 Diego Oliver Vargas Moya & Luis Alberto Ortiz Morales
#
# Este programa es software libre: puedes redistribuirlo y/o modificarlo
# bajo los términos de la Licencia Pública General GNU (GNU GPL) publicada
# por la Free Software Foundation, ya sea la versión 3 de la Licencia, o
# (a tu elección) cualquier versión posterior.
#
# Este programa se distribuye con la esperanza de que sea útil,
# pero SIN GARANTÍA ALGUNA; ni siquiera la garantía implícita
# MERCANTIL o de APTITUD PARA UN PROPÓSITO DETERMINADO.
# Consulta los detalles de la Licencia Pública General GNU para más
# información.
#
# Deberías haber recibido una copia de la Licencia Pública General GNU
# junto a este programa. En caso contrario, consulta
# <https://www.gnu.org/licenses/>.
# ------------------------------------------------------------------------------

import numpy as np

#======================================================================
# I. MATRIZ DE TRANSFORMACIÓN DE COORDENADAS PARA PÓRTICOS 3D
#======================================================================
def matriz_transformacion_portico_3d(p1, p2):
    """
    Genera la matriz de transformación de coordenadas espaciales (12x12)
    que relaciona el sistema de coordenadas local del elemento con el
    sistema global de la estructura.
    """
    # 1. Determinación del vector direccional y longitud geométrica del elemento
    v = np.array(p2) - np.array(p1)
    L = np.linalg.norm(v)
    if L < 1e-9: raise ValueError("La longitud del elemento de pórtico no puede ser cero.")

    # 2. Vector unitario correspondiente al eje X local (eje longitudinal de la barra)
    vx = v / L

    # 3. Orientación del eje Z global utilizado como referencia espacial
    eje_Z_global = np.array([0.0, 0.0, 1.0])

    # 4. Cálculo de vectores unitarios transversales (ejes Y y Z locales)
    if np.allclose(np.abs(vx), eje_Z_global):
        # Caso particular: Elemento puramente vertical (columna perfecta)
        vy = np.array([1.0, 0.0, 0.0])
        vz = np.cross(vx, vy)
```

```
    else:
        # Caso general: Elemento con inclinación respecto a la vertical
        vy = np.cross(eje_Z_global, vx)
        vy /= np.linalg.norm(vy)
        vz = np.cross(vx, vy)

    # 5. Ensamblaje de la matriz de rotación (3x3) y proyección a la matriz de transformación
(12x12)
    R = np.vstack([vx, vy, vz]); T = np.zeros((12, 12))
    for i in range(4): T[i*3:(i+1)*3, i*3:(i+1)*3] = R
    return T, L

#====================================================================
# II. MATRIZ DE RIGIDEZ LOCAL DEL ELEMENTO (EULER-BERNOULLI / TIMOSHENKO)
#====================================================================
def matriz_rigidez_local_portico_3d(L, E, G, A, J, Iy, Iz, Ay, Az, usar_timoshenko=True):
    """
    Formulación de la matriz de rigidez elemental (12x12) en el sistema de
    coordenadas locales. El algoritmo permite alternar entre la teoría clásica
    de Euler-Bernoulli y la teoría de Timoshenko.
    """
    # 1. Componentes de rigidez axial y torsional básicas
    EA_L=E*A/L
    GJ_L=G*J/L

    # 2. Factores adimensionales de corrección por cortante (Teoría de Timoshenko)
    if usar_timoshenko and Ay > 1e-12 and Az > 1e-12:
        # Factor de cortante para el plano X-Y (flexión Mz)
        phi_z = (12 * E * Iz) / (G * Ay * L**2)
        # Factor de cortante para el plano X-Z (flexión My)
        phi_y = (12 * E * Iy) / (G * Az * L**2)
    else:
        # Si no se usa Timoshenko (o áreas son cero), los factores son 0
        # y las ecuaciones se reducen a Euler-Bernoulli.
        phi_y = 0.0
        phi_z = 0.0

    # 3. Denominadores modificados para términos de flexión
    den_y = 1.0 + phi_y
    den_z = 1.0 + phi_z

    # 4. Cálculo de términos de rigidez a flexión y cortante (Plano X-Y, Momento Mz, Cortante
Vy)
    EIz_L3 = (12 * E * Iz) / (L**3 * den_z)
    EIz_L2 = (6 * E * Iz) / (L**2 * den_z)
    EIz_L_4 = ((4.0 + phi_z) * E * Iz) / (L * den_z)
    EIz_L_2 = ((2.0 - phi_z) * E * Iz) / (L * den_z)

    # 5. Cálculo de términos de rigidez a flexión y cortante (Plano X-Z, Momento My, Cortante
Vz)
    EIy_L3 = (12 * E * Iy) / (L**3 * den_y)
    EIy_L2 = (6 * E * Iy) / (L**2 * den_y)
    EIy_L_4 = ((4.0 + phi_y) * E * Iy) / (L * den_y)
    EIy_L_2 = ((2.0 - phi_y) * E * Iy) / (L * den_y)

    # 6. Ensamblaje y retorno explícito de la matriz de rigidez local 12x12
    return np.array([
```

```
        [EA_L,0,0,0,0,0,-EA_L,0,0,0,0,0],
        [0,EIz_L3,0,0,0,EIz_L2,0,-EIz_L3,0,0,0,EIz_L2],
        [0,0,EIy_L3,0,-EIy_L2,0,0,0,-EIy_L3,0,-EIy_L2,0],
        [0,0,0,GJ_L,0,0,0,0,0,-GJ_L,0,0],
        [0,0,-EIy_L2,0,EIy_L_4,0,0,0,EIy_L2,0,EIy_L_2,0],
        [0,EIz_L2,0,0,0,EIz_L_4,0,-EIz_L2,0,0,0,EIz_L_2],
        [-EA_L,0,0,0,0,0,EA_L,0,0,0,0,0],
        [0,-EIz_L3,0,0,0,-EIz_L2,0,EIz_L3,0,0,0,-EIz_L2],
        [0,0,-EIy_L3,0,EIy_L2,0,0,0,EIy_L3,0,EIy_L2,0],
        [0,0,0,-GJ_L,0,0,0,0,0,GJ_L,0,0],
        [0,0,-EIy_L2,0,EIy_L_2,0,0,0,EIy_L2,0,EIy_L_4,0],
        [0,EIz_L2,0,0,0,EIz_L_2,0,-EIz_L2,0,0,0,EIz_L_4]])

#====================================================================
# III. CLASE PRINCIPAL: MOTOR DE RESOLUCIÓN MATRICIAL
#====================================================================
class Solucionador3D:
    """
    Clase contenedora que ejecuta el Análisis Matricial de Estructuras.
    Se encarga de orquestar el ensamblaje de la matriz de rigidez global
    y la resolución algorítmica del sistema fundamental de ecuaciones.
    """
    def __init__(self, modelo, usar_timoshenko=True):
        # 1. Instanciación del modelo geométrico y de cargas
        self.modelo = modelo
        self.usar_timoshenko = usar_timoshenko

        # 2. Diccionario de almacenamiento precalculado de propiedades mecánicas y
geométricas
        self.propiedades_porticos = {}
        ids_portico = {e[2] for e in modelo.elementos.values()}
        for id_mat in ids_portico:
            if id_mat in modelo.materiales:
                 self.propiedades_porticos[id_mat] =
modelo.get_propiedades_calculadas(id_mat)

#====================================================================
# IV. ENSAMBLAJE DE LA MATRIZ DE RIGIDEZ GLOBAL
#====================================================================
    def ensamblar_K_global(self):
        """
        Calcula y superpone las contribuciones de rigidez elementales (k) en las
        posiciones topológicas correctas dentro de la Matriz de Rigidez Global (K).
        """
        # 1. Inicialización de la estructura de datos para la Memoria de Cálculo (Auditoría)
        datos_reporte = {
            'log_ensamblaje': [],
            'k_locales': {},
            'T_matrices': {},
            'K_global_elem': {},
            'K_global': None
        }
        log = datos_reporte['log_ensamblaje']

        # 2. Definición de la dimensión de la matriz global (6 GDL totales por cada nodo)
        gdl_totales = len(self.modelo.nodos) * 6
        K_global = np.zeros((gdl_totales, gdl_totales))
```

```
        log.append("--- INICIO: Ensamblaje de Matriz de Rigidez Global ---")

        # 3. Iteración sobre la topología del modelo estructural
        for id_elem, (ni, nj, mat_id) in self.modelo.elementos.items():
            log.append(f"  -> Ensamblando Pórtico E{id_elem} (Nodos {ni}-{nj})")
            p1, p2 = self.modelo.nodos[ni], self.modelo.nodos[nj]

            # 4. Extracción de propiedades y conversión de módulos elásticos a unidades base
(kPa)
            E, G, A, J, Iy, Iz, Ay, Az = self.propiedades_porticos[mat_id]
            EkPa= E*1000
            GkPa= G*1000

            # 5. Cálculo de matrices elementales del pórtico 3D
            T, L = matriz_transformacion_portico_3d(p1, p2)
            K_local = matriz_rigidez_local_portico_3d(L, EkPa, GkPa, A, J, Iy, Iz, Ay, Az,
self.usar_timoshenko)

            # 6. Transformación por congruencia de la matriz elemental al sistema global
            K_global_elem = T.T @ K_local @ T

            # 7. Almacenamiento temporal de matrices para trazabilidad del reporte
            datos_reporte['k_locales'][id_elem] = K_local
            datos_reporte['T_matrices'][id_elem] = T
            datos_reporte['K_global_elem'][id_elem] = K_global_elem

            # 8. Mapeo de sub-índices y superposición aditiva en la matriz K global
            gdl = list(range((ni-1)*6, ni*6)) + list(range((nj-1)*6, nj*6))
            for i in range(12):
                for j in range(12):
                    K_global[gdl[i], gdl[j]] += K_global_elem[i, j]

        log.append(f"--- FIN: Ensamblaje explícito completado. Dimensión de K_global:
{K_global.shape} ---")

        # 9. Inyección del resultado en los datos del reporte y retorno de la Matriz Global
        datos_reporte['K_global'] = K_global
        return K_global, datos_reporte

#==================================================================
# V. RESOLUCIÓN MATRICIAL Y POST-PROCESO DE RESULTADOS
#==================================================================
    def resolver(self, vector_fuerzas, K_global, fep_por_elemento={}):
        """
        Aplica las condiciones de borde, ejecuta la resolución algebraica del sistema
        reducido para hallar los desplazamientos nodales y realiza el post-proceso
        de reacciones y solicitaciones internas.
        """
        # 1. Inicialización de contenedor de datos para trazabilidad algorítmica
        datos_reporte_resolucion = {
            'log_resolucion': [],
            'gdl_totales': 0,
            'gdl_libres': [],
            'gdl_restringidos': [],
            'K_reducida': None,
            'F_reducido': None,
            'cond_K_reducida': 0.0
```

```
        }
        log = datos_reporte_resolucion['log_resolucion']

        # 2. Identificación geométrica de grados de libertad (GDL) libres y restringidos
(apoyos)
        gdl_totales = len(self.modelo.nodos) * 6
        gdl_restringidos = [
            (id_nodo - 1) * 6 + i
            for id_nodo, restr in self.modelo.apoyos.items()
            for i, r_bool in enumerate(restr) if r_bool
        ]
        gdl_libres = [i for i in range(gdl_totales) if i not in gdl_restringidos]

        # Almacenamiento de variables de estado para el reporte
        datos_reporte_resolucion['gdl_totales'] = gdl_totales
        datos_reporte_resolucion['gdl_libres'] = gdl_libres
        datos_reporte_resolucion['gdl_restringidos'] = gdl_restringidos

        log.append(f"--- INICIO: Resolución del sistema de ecuaciones ---")
        log.append(f"  GDL Totales: {gdl_totales}, Libres: {len(gdl_libres)}, Restringidos:
{len(gdl_restringidos)}")

         # 3. Partición de la matriz global para obtener el subsistema matemáticamente
resoluble
        K_reducida = K_global[np.ix_(gdl_libres, gdl_libres)]
        F_reducida = vector_fuerzas.copy()[gdl_libres]

        # 4. Verificación de la estabilidad numérica del sistema estructural
        cond_k = 0.0
        try:
            cond_k = np.linalg.cond(K_reducida)
            print(f"  -> Condición de la matriz de rigidez reducida: {cond_k:.2e}")
            log.append(f"  -> Condición de la matriz de rigidez reducida: {cond_k:.2e}")
        except np.linalg.LinAlgError:
            print("  -> Advertencia: Matriz singular, la condición no puede ser calculada.")
            log.append("  -> Advertencia: Matriz singular, la condición no puede ser
calculada.")

        datos_reporte_resolucion['K_reducida'] = K_reducida
        datos_reporte_resolucion['F_reducido'] = F_reducida
        datos_reporte_resolucion['cond_K_reducida'] = cond_k

        # 5. Resolución computacional del vector de desplazamientos [ d ] delegada a NumPy
        try:
            d_reducidos = np.linalg.solve(K_reducida, F_reducida)
        except np.linalg.LinAlgError:
            raise RuntimeError("La estructura es inestable (matriz singular). Verifique los
apoyos y la conectividad.")

        # 6. Reconstrucción del vector de desplazamientos general incluyendo condiciones de
borde nulas
        desplazamientos = np.zeros(gdl_totales)
        desplazamientos[gdl_libres] = d_reducidos

        print("  -> Desplazamientos calculados.")
        log.append("  -> Desplazamientos calculados.")
```

```
        # 7. Post-proceso: Determinación analítica de las reacciones en los apoyos
        reacciones = (K_global @ desplazamientos) - vector_fuerzas

        print("  -> Reacciones en apoyos calculadas.")
        log.append("  -> Reacciones en apoyos calculadas.")

        # 8. Post-proceso: Extracción de esfuerzos internos elementales
        fuerzas_internas = {}
        print("--- INICIO: Cálculo de Fuerzas Internas en Elementos Locales ---")
        log.append("--- INICIO: Cálculo de Fuerzas Internas en Elementos Locales ---")

        for id_elem, (ni, nj, mat_id) in self.modelo.elementos.items():
            p1, p2 = self.modelo.nodos[ni], self.modelo.nodos[nj]

            # Recálculo de las matrices locales específicas de cada elemento
            E, G, A, J, Iy, Iz, Ay, Az = self.propiedades_porticos[mat_id]
            EkPa = E * 1000
            GkPa = G * 1000
            T, L = matriz_transformacion_portico_3d(p1, p2)
            K_local = matriz_rigidez_local_portico_3d(L, EkPa, GkPa, A, J, Iy, Iz, Ay, Az,
self.usar_timoshenko)

            # Extracción del sub-vector de desplazamientos asociado a los nodos del elemento
            gdl = list(range((ni-1)*6, ni*6)) + list(range((nj-1)*6, nj*6))

            # Transformación de desplazamientos al sistema local (T @ d_global)
            d_local = T @ desplazamientos[gdl]

            # Superposición de efectos: Fuerzas por desplazamiento + Fuerzas de Empotramiento
Perfecto (FEP)
            fep = fep_por_elemento.get(id_elem, np.zeros(12))
            fuerzas_internas[id_elem] = (K_local @ d_local) + fep

        print("--- FIN: Cálculo de Fuerzas y Reacciones completado. ---")
        log.append("--- FIN: Cálculo de Fuerzas y Reacciones completado. ---")

        # 9. Retorno consolidado de los datos computados del análisis estructural
        resultados_completos = {
            'desplazamientos': desplazamientos,
            'reacciones': reacciones,
            'fuerzas_internas': fuerzas_internas,
            'reporte_resolucion': datos_reporte_resolucion
        }
        return resultados_completos
```

Anexo 3: Script procesador_cargas.py

```
"""
Módulo: procesador_cargas.py
Descripción: Gestiona la definición de estados y combinaciones de carga.
Transforma las cargas aplicadas (puntuales, distribuidas, superficiales) en
vectores de fuerzas equivalentes en los nodos para el análisis matricial.
"""

# ------------------------------------------------------------------------
# VORTEX 3D - Análisis Estructural de Edificaciones 3D y Diseño Normativo
# de Vigas y Columnas
# Copyright (C) 2026 Diego Oliver Vargas Moya & Luis Alberto Ortiz Morales
#
# Este programa es software libre: puedes redistribuirlo y/o modificarlo
# bajo los términos de la Licencia Pública General GNU (GNU GPL) publicada
# por la Free Software Foundation, ya sea la versión 3 de la Licencia, o
# (a tu elección) cualquier versión posterior.
#
# Este programa se distribuye con la esperanza de que sea útil,
# pero SIN GARANTÍA ALGUNA; ni siquiera la garantía implícita
# MERCANTIL o de APTITUD PARA UN PROPÓSITO DETERMINADO.
# Consulta los detalles de la Licencia Pública General GNU para más
# información.
#
# Deberías haber recibido una copia de la Licencia Pública General GNU
# junto a este programa. En caso contrario, consulta
# <https://www.gnu.org/licenses/>.
# ------------------------------------------------------------------------


#====================================================================
# I. IMPORTACIONES Y DEPENDENCIAS
#====================================================================
import numpy as np
from collections import defaultdict
import itertools
from PySide6.QtWidgets import QMessageBox
from calc import Solucionador3D, matriz_transformacion_portico_3d
from distribuidor_losas import traducir_carga_losa_a_cargas_lineales

#====================================================================
# II. FUNCIONES ANALÍTICAS: FUERZAS DE EMPOTRAMIENTO PERFECTO (FEP)
#====================================================================
def _fep_uniforme_parcial(L, w, a, b):
    """
    Calcula los FEPs (Reacciones y Momentos anti-horarios) para una carga
    uniforme 'w' aplicada en un tramo [a, b] de una viga de longitud 'L'.
    Devuelve (R_i, M_i, R_j, M_j)
    """
    # 1. Validación geométrica del elemento
    if L < 1e-9: return (0.0, 0.0, 0.0, 0.0)
    L2, L3 = L*L, L*L*L

    # 2. Pre-cálculo de potencias integradas (integrales de x^n de 'a' a 'b')
    b0 = (b - a)
    b1 = (b**2 - a**2) / 2.0
```

```
    b2 = (b**3 - a**3) / 3.0
    b3 = (b**4 - a**4) / 4.0

    # 3. R_i = w * integral( (1 - 3(x/L)^2 + 2(x/L)^3) dx) de 'a' a 'b'
    R_i = w * (b0 - 3*b2/L2 + 2*b3/L3)

    # 4. M_i = w * integral( (x - 2x^2/L + x^3/L^2) dx) de 'a' a 'b'
    M_i = w * (b1 - 2*b2/L + b3/L2)

    # 5. R_j = w * integral( (3(x/L)^2 - 2(x/L)^3) dx) de 'a' a 'b'
    R_j = w * (3*b2/L2 - 2*b3/L3)

    # 6. M_j = w * integral( (x^3/L^2 - x^2/L) dx) de 'a' a 'b'
    M_j = w * (b3/L2 - b2/L)

    return (R_i, M_i, R_j, M_j)

def _fep_triangular_parcial(L, w_pico, a, b):
    """
    Calcula los FEPs para una carga triangular (pico 'w_pico' en x=b,
    cero en x=a) aplicada en [a, b] en una viga de longitud 'L'.
    Devuelve (R_i, M_i, R_j, M_j)
    """
    # 1. Validación de la existencia geométrica del tramo de aplicación
    if L < 1e-9 or abs(b - a) < 1e-9: return (0.0, 0.0, 0.0, 0.0)
    L2, L3 = L*L, L*L*L

    # 2. Constante de la pendiente: w(x) = k * (x - a)
    denominador = (b - a)
    k = w_pico / denominador

    # 3. Pre-cálculo de potencias integradas (integrales de x^n * (x-a) de 'a' a 'b')
    # integral(k * (x-a) * x^n dx) = k * integral(x^(n+1) - a*x^n dx)

    # n=0: k * [x^2/2 - ax]
    i0 = k * ( (b**2 - a**2)/2.0 - a*(b - a) )
    # n=1: k * [x^3/3 - ax^2/2]
    i1 = k * ( (b**3 - a**3)/3.0 - a*(b**2 - a**2)/2.0 )
    # n=2: k * [x^4/4 - ax^3/3]
    i2 = k * ( (b**4 - a**4)/4.0 - a*(b**3 - a**3)/3.0 )
    # n=3: k * [x^5/5 - ax^4/4]
    i3 = k * ( (b**5 - a**5)/5.0 - a*(b**4 - a**4)/4.0 )

    # 4. R_i = integral( w(x) * (1 - 3(x/L)^2 + 2(x/L)^3) dx)
    R_i = i0 - 3*i2/L2 + 2*i3/L3

    # 5. M_i = integral( w(x) * (x - 2x^2/L + x^3/L^2) dx)
    M_i = i1 - 2*i2/L + i3/L2

    # 6. R_j = integral( w(x) * (3(x/L)^2 - 2(x/L)^3) dx)
    R_j = 3*i2/L2 - 2*i3/L3

    # 7. M_j = integral( w(x) * (x^3/L^2 - x^2/L) dx)
    M_j = i3/L2 - i2/L

    return (R_i, M_i, R_j, M_j)
```

```
#=====================================================================
# III. UTILIDADES TOPOLÓGICAS Y MATEMÁTICAS PARA TRAMOS DE CARGA
#=====================================================================

def _interpolar_carga_en_punto(lista_tramos, p_norm_buscado):
    """
    Interpola linealmente el valor de carga 'q' para un 'p_norm_buscado'
    dentro de una lista de tramos [(p1, q1), (p2, q2), ...].
    """
    # 1. Verificación de nulidad en el arreglo de entrada
    if not lista_tramos:
        return 0.0

    # 2. Control de límites de contorno (fuera del dominio interpolable)

        # Caso 1: Antes del inicio de la lista
    if p_norm_buscado < lista_tramos[0][0] - 1e-9:
        return 0.0 # O el valor del primer punto, según convención. 0 es más seguro.

        # Caso 2: Después del final de la lista
    if p_norm_buscado > lista_tramos[-1][0] + 1e-9:
        return 0.0 # O el valor del último punto.

    # 3. Búsqueda iterativa del sub-dominio y cálculo del factor de interpolación
    for i in range(len(lista_tramos) - 1):
        p_a, q_a = lista_tramos[i]
        p_b, q_b = lista_tramos[i+1]

        if p_a - 1e-9 <= p_norm_buscado <= p_b + 1e-9:
            # Evitar división por cero si los puntos son idénticos
            if abs(p_b - p_a) < 1e-9:
                return q_a

            # Interpolación lineal
            factor_t = (p_norm_buscado - p_a) / (p_b - p_a)
            q_interpolado = q_a + factor_t * (q_b - q_a)
            return q_interpolado

    # 4. Tolerancia numérica para convergencia en el nodo final
    if abs(p_norm_buscado - lista_tramos[-1][0]) < 1e-9:
        return lista_tramos[-1][1]

    return 0.0 # Fallback

def _sumar_tramos_lineales(lista_tramos_1, lista_tramos_2):
    """
    Suma dos listas de cargas por tramos [(p, q), ...],
    con fusión de puntos cercanos para evitar errores numéricos.
    """
    # 1. Definición del umbral de tolerancia para convergencia geométrica
    TOLERANCIA_FUSION = 1e-4  # 0.1 mm en una viga de 1m. Ajustable.

    # 2. Extracción y ordenamiento del conjunto de coordenadas paramétricas
    puntos_raw = []
    for p, _ in lista_tramos_1: puntos_raw.append(p)
    for p, _ in lista_tramos_2: puntos_raw.append(p)
```

```
    puntos_raw.sort()

    # 3. Fusionar puntos muy cercanos
    if not puntos_raw: return [(0.0, 0.0), (1.0, 0.0)]

    puntos_filtrados = [puntos_raw[0]]
    for p in puntos_raw[1:]:
        if p - puntos_filtrados[-1] > TOLERANCIA_FUSION:
            puntos_filtrados.append(p)
        # Si está muy cerca, ignoramos 'p' y nos quedamos con el anterior,
        # asumiendo que son el mismo punto geométrico.

    # 4. Interpolar y sumar
    lista_sumada = []
    for p in puntos_filtrados:
        if p < -1e-9 or p > 1.0 + 1e-9: continue

        q1 = _interpolar_carga_en_punto(lista_tramos_1, p)
        q2 = _interpolar_carga_en_punto(lista_tramos_2, p)
        q_total = q1 + q2
        lista_sumada.append((p, q_total))

    # 5. Asegurar extremos 0.0 y 1.0
    if not lista_sumada or lista_sumada[0][0] > 1e-9:
        q_start = _interpolar_carga_en_punto(lista_tramos_1, 0.0) +
_interpolar_carga_en_punto(lista_tramos_2, 0.0)
        lista_sumada.insert(0, (0.0, q_start))

    if lista_sumada[-1][0] < 1.0 - 1e-9:
        q_end = _interpolar_carga_en_punto(lista_tramos_1, 1.0) +
_interpolar_carga_en_punto(lista_tramos_2, 1.0)
        lista_sumada.append((1.0, q_end))

    return lista_sumada

#====================================================================
# IV. DEFINICIÓN DE COMBINACIONES DE ESTADOS DE CARGA
#====================================================================
class CombinacionCarga:
    """Clase para definir una combinación de carga con sus factores."""
    def __init__(self, nombre, factores, tipo='Norma'):
        self.nombre = nombre
        self.factores = factores
        self.tipo = tipo
    def to_dict(self):
        # 1. Serialización del objeto para persistencia de datos
        return {'nombre': self.nombre, 'factores': self.factores, 'tipo': self.tipo}

    @classmethod
    def desde_dict(cls, data):
        # 2. Deserialización del objeto desde un diccionario estructurado
        return cls(data['nombre'], data['factores'], data.get('tipo', 'Usuario'))

#====================================================================
# V. MOTOR DE PROCESAMIENTO Y DISTRIBUCIÓN DE CARGAS
#====================================================================
class ProcesadorCargas:
```

```
    """
    Clase principal responsable de la orquestación, transformación y ensamble
    de los estados de carga aplicados sobre la topología estructural.
    """
    def __init__(self, modelo):
        # 1. Inyección de dependencias del modelo matemático global
        self.modelo = modelo
        self.combinaciones_norma = self._get_combinaciones_nb1225002()

    def _get_combinaciones_nb1225002(self):
        """Define las combinaciones de carga estándar de la norma."""
        return [
            CombinacionCarga("Cálculo Simple", {'D': 1.0}),
            CombinacionCarga("1.4D", {'D': 1.4}),
            CombinacionCarga("1.2D + 1.6L + 0.5Lr", {'D': 1.2, 'L': 1.6, 'Lr': 0.5}),
            CombinacionCarga("1.2D + 1.6Lr + 1.0L", {'D': 1.2, 'Lr': 1.6, 'L': 1.0}),
            CombinacionCarga("1.2D + 1.0W + 1.0L + 0.5Lr", {'D': 1.2, 'W': 1.0, 'L': 1.0,
'Lr': 0.5}),
            CombinacionCarga("0.9D + 1.0W", {'D': 0.9, 'W': 1.0}),
        ]

    def _generar_cargas_pp_auto(self):
        """
        Calcula el Peso Propio (PP) Just-in-Time (JIT) para todos los elementos
        (1D y 2D) y genera los vectores, FEPs, cargas distribuidas y logs
        necesarios tanto para el cálculo como para el reporte.
        """
        # 1. Inicialización de contenedores matriciales
        num_gdl = len(self.modelo.nodos) * 6
        vector_pp_total = np.zeros(num_gdl)
        feps_pp_total_dict = defaultdict(lambda: np.zeros(12))
        cargas_dist_pp_total_dict = defaultdict(lambda: {
            'tramos_y': [(0.0, 0.0), (1.0, 0.0)],
            'tramos_z': [(0.0, 0.0), (1.0, 0.0)],
            'axial_x': 0.0,
            'torsion_mx': 0.0
        })
        log_pp_1d = []

        print("[INFO] Calculando PP (JIT) para Elementos 1D...")

        # 1. Procesar Elementos 1D (Vigas/Columnas)
        for id_elem, (ni, nj, id_mat) in self.modelo.elementos.items():
            if id_mat not in self.modelo.materiales:
                continue

            # Obtener PE y Área
            pe_1d = self.modelo.materiales[id_mat].get('peso_especifico', 0.0)
            if abs(pe_1d) < 1e-9:
                continue

            try:
                # 3. Extracción de propiedades geométricas transversales
                propiedades = self.modelo.get_propiedades_calculadas(id_mat)
                A = propiedades[2]
                if abs(A) < 1e-9:
                    continue
```

```
                # 4. Obtener Transformación
                p1, p2 = self.modelo.nodos[ni], self.modelo.nodos[nj]
                T, L = matriz_transformacion_portico_3d(p1, p2)
                R = T[:3, :3]
                if L < 1e-9:
                    continue

            except Exception as e:
                print(f"[WARN] Omitiendo PP para Elem {id_elem}: {e}")
                continue

            # 5. Calcular carga local
            w_pp = pe_1d * A # Carga (kN/m)
            w_global = np.array([0.0, 0.0, -w_pp]) # Actúa en Z Global negativo
            w_local = R @ w_global

            # 6. Evaluación de las componentes del vector de Fuerzas de Empotramiento
Perfecto
            fep_local = np.zeros(12)
            fep_local[0] = -w_local[0] * L / 2
            fep_local[6] = -w_local[0] * L / 2
            fep_local[1] = -w_local[1] * L / 2
            fep_local[7] = -w_local[1] * L / 2
            fep_local[5] = -w_local[1] * L**2 / 12 # Mz_i
            fep_local[11] = w_local[1] * L**2 / 12 # Mz_j
            fep_local[2] = -w_local[2] * L / 2
            fep_local[8] = -w_local[2] * L / 2
            fep_local[4] = w_local[2] * L**2 / 12  # My_i
            fep_local[10] = -w_local[2] * L**2 / 12 # My_j
            # (No hay torsión 'mt' para PP)

            # 7. Sumar a los totales
            feps_pp_total_dict[id_elem] += fep_local
            fep_global = T.T @ fep_local
            indices_globales = list(range((ni-1)*6, ni*6)) + list(range((nj-1)*6, nj*6))
            vector_pp_total[indices_globales] -= fep_global # FEPs se restan

            # 8. Poblar cargas_dist (para diagramas)
            cargas_dist_pp_total_dict[id_elem]['axial_x'] += w_local[0]

            tramos_y_nuevos = [(0.0, w_local[1]), (1.0, w_local[1])] if abs(w_local[1]) > 1e-
9 else []
            tramos_z_nuevos = [(0.0, w_local[2]), (1.0, w_local[2])] if abs(w_local[2]) > 1e-
9 else []

            if tramos_y_nuevos:
                tramos_y_actuales = cargas_dist_pp_total_dict[id_elem]['tramos_y']
                cargas_dist_pp_total_dict[id_elem]['tramos_y'] =
_sumar_tramos_lineales(tramos_y_actuales, tramos_y_nuevos)

            if tramos_z_nuevos:
                tramos_z_actuales = cargas_dist_pp_total_dict[id_elem]['tramos_z']
                cargas_dist_pp_total_dict[id_elem]['tramos_z'] =
_sumar_tramos_lineales(tramos_z_actuales, tramos_z_nuevos)

            # 9. Registro del proceso para la memoria de cálculo
```

```
            log_pp_1d.append({'id_elem': id_elem, 'w_global_z': -w_pp, 'w_local': w_local})


        print("[INFO] Calculando PP (JIT) para Losas 2D...")

        # 10. Procesar Losas (2D)
        for id_losa, datos_losa in self.modelo.losas.items():
            pe_2d = datos_losa.get('peso_especifico', 0.0)
            espesor = datos_losa.get('espesor', 0.0)

            if abs(pe_2d) < 1e-9 or abs(espesor) < 1e-9:
                continue

            wz_pp = -pe_2d * espesor # Carga superficial (hacia abajo)

            try:
                # 11. Traducción del plano de carga a fuerzas equivalentes en apoyos lineales
                cargas_generadas, log_datos_losa = traducir_carga_losa_a_cargas_lineales(
                    id_losa, wz_pp, -1, self.modelo
                )

                # 12. Inyección de datos al registro global de reporte
                log_datos_losa['tipo'] = f"PP_{log_datos_losa['tipo']}" # Marcarla como PP
                if hasattr(self.modelo, 'datos_reporte_losas'):
                    self.modelo.datos_reporte_losas.append(log_datos_losa)

                # 13. Ensamblaje iterativo de cargas tributarias en pórticos adyacentes
                for carga_elem_gen in cargas_generadas:
                    id_elem = carga_elem_gen['id_elemento']
                    datos_carga = carga_elem_gen['datos_carga']

                    if id_elem not in self.modelo.elementos: continue

                    ni, nj, _ = self.modelo.elementos[id_elem]
                    if ni not in self.modelo.nodos or nj not in self.modelo.nodos: continue

                    p1, p2 = self.modelo.nodos[ni], self.modelo.nodos[nj]

                    try:
                        T, L = matriz_transformacion_portico_3d(p1, p2)
                    except ValueError:
                        continue
                    if L < 1e-9: continue

                    tipo_carga_losa = datos_carga[0]
                    fep_local = np.zeros(12)

                    # 14. Procesamiento condicional según el tipo topológico de la carga
incidente
                    if tipo_carga_losa == 'uniforme':
                        # (uniforme, wx_local, wy_local, wz_local, mt_local)
                        _, wlx, wly, wlz, mlt = datos_carga
                        w_local = np.array([wlx, wly, wlz])
                        mt = mlt

                        # Poblar cargas_dist (para diagramas)
                        cargas_dist_pp_total_dict[id_elem]['axial_x'] += w_local[0]
```

```
                        cargas_dist_pp_total_dict[id_elem]['torsion_mx'] += mt

                        tramos_y_nuevos = [(0.0, w_local[1]), (1.0, w_local[1])] if
abs(w_local[1]) > 1e-9 else []
                        tramos_z_nuevos = [(0.0, w_local[2]), (1.0, w_local[2])] if
abs(w_local[2]) > 1e-9 else []

                        if tramos_y_nuevos:
                            tramos_y_actuales =
cargas_dist_pp_total_dict[id_elem]['tramos_y']
                            cargas_dist_pp_total_dict[id_elem]['tramos_y'] =
_sumar_tramos_lineales(tramos_y_actuales, tramos_y_nuevos)

                        if tramos_z_nuevos:
                            tramos_z_actuales =
cargas_dist_pp_total_dict[id_elem]['tramos_z']
                            cargas_dist_pp_total_dict[id_elem]['tramos_z'] =
_sumar_tramos_lineales(tramos_z_actuales, tramos_z_nuevos)

                        # 15. Formulación del vector FEP para el patrón uniforme sobre el
elemento receptor
                        fep_local[0] = -w_local[0] * L / 2
                        fep_local[6] = -w_local[0] * L / 2
                        fep_local[1] = -w_local[1] * L / 2
                        fep_local[7] = -w_local[1] * L / 2
                        fep_local[5] = -w_local[1] * L**2 / 12 # Mz_i
                        fep_local[11] = w_local[1] * L**2 / 12 # Mz_j
                        fep_local[2] = -w_local[2] * L / 2
                        fep_local[8] = -w_local[2] * L / 2
                        fep_local[4] = w_local[2] * L**2 / 12  # My_i
                        fep_local[10] = -w_local[2] * L**2 / 12 # My_j
                        fep_local[3] = -mt * L / 2
                        fep_local[9] = -mt * L / 2

                    elif tipo_carga_losa == 'tramos_locales':
                        _, eje_local, lista_de_puntos = datos_carga

                        # Poblar cargas_dist (para diagramas)
                        if eje_local == 'y':
                            clave_tramos = 'tramos_y'
                        elif eje_local == 'z':
                            clave_tramos = 'tramos_z'
                        else:
                            continue # Eje no soportado, omitimos esta carga

                        tramos_actuales = cargas_dist_pp_total_dict[id_elem][clave_tramos]
                        cargas_dist_pp_total_dict[id_elem][clave_tramos] =
_sumar_tramos_lineales(tramos_actuales, lista_de_puntos)

                        # 16. Calcular FEPs - Integración de superposición para perfiles de
carga discontinuos
                        fep_superposicion = np.zeros(4) # (R_i, M_i, R_j, M_j)
                        for i in range(len(lista_de_puntos) - 1):
                            (p_a_norm, q_a) = lista_de_puntos[i]
                            (p_b_norm, q_b) = lista_de_puntos[i+1]
                            a = p_a_norm * L
                            b = p_b_norm * L
```

```
                            if abs(b - a) < 1e-9: continue
                            w_unif = q_a
                            w_tria = q_b - q_a
                            if abs(w_unif) > 1e-9:
                                fep_superposicion += np.array(_fep_uniforme_parcial(L,
w_unif, a, b))
                            if abs(w_tria) > 1e-9:
                                fep_superposicion += np.array(_fep_triangular_parcial(L,
w_tria, a, b))

                        R_i_total, M_i_total, R_j_total, M_j_total = fep_superposicion

                        if eje_local == 'y':
                            fep_local[1] = -R_i_total; fep_local[7] = -R_j_total
                            fep_local[5] = -M_i_total; fep_local[11] = -M_j_total
                        elif eje_local == 'z':
                            fep_local[2] = -R_i_total; fep_local[8] = -R_j_total
                            fep_local[4] = M_i_total; fep_local[10] = M_j_total

                    # 17. Ensamblaje final de aportes tributarios hacia el vector general de
solicitación
                    if np.any(fep_local):
                        feps_pp_total_dict[id_elem] += fep_local
                        fep_global = T.T @ fep_local
                        indices_globales = list(range((ni-1)*6, ni*6)) + list(range((nj-1)*6,
nj*6))
                        vector_pp_total[indices_globales] -= fep_global

            except Exception as e:
                print(f"[WARN] Omitiendo PP para Losa {id_losa}: Error en distribución: {e}")

        print("[INFO] Cálculo JIT de PP finalizado.")
        return (vector_pp_total, feps_pp_total_dict, cargas_dist_pp_total_dict, log_pp_1d)

    def _agrupar_cargas_por_hipotesis(self):
        """
        Agrupa todas las cargas del modelo por su ID de hipótesis.
        Devuelve los vectores de fuerza globales, FEPs y cargas distribuidas locales.
        AHORA TAMBIÉN DEVUELVE UN LOG DE DATOS.
        """
        # 1. Definición del registro de trazabilidad procedimental
        log_reporte_agrupacion = []

        # 2. Inicialización de tensores acumuladores
        num_gdl = len(self.modelo.nodos) * 6
        vectores_carga_base = defaultdict(lambda: np.zeros(num_gdl))
        feps_base = defaultdict(dict)

        cargas_dist_base = defaultdict(lambda: defaultdict(lambda: {
            'tramos_y': [(0.0, 0.0), (1.0, 0.0)],
            'tramos_z': [(0.0, 0.0), (1.0, 0.0)],
            'axial_x': 0.0,
            'torsion_mx': 0.0
        }))

        # 3. PROCESAR CARGAS NODALES
        for carga in self.modelo.cargas_nodales:
```

```
            id_hipotesis = carga['id_hipotesis']
            id_nodo = carga['id_nodo']
            idx_inicio = (id_nodo - 1) * 6
            vectores_carga_base[id_hipotesis][idx_inicio : idx_inicio + 6] += carga['vector']

            log_reporte_agrupacion.append({'tipo': 'nodal', 'id_hipotesis': id_hipotesis,
'id_nodo': id_nodo, 'vector': carga['vector']})

        # 4. PROCESAR CARGAS DE ELEMENTOS
        for carga in self.modelo.cargas_elementos:
            id_hipotesis = carga['id_hipotesis']
            id_elem = carga['id_elemento']

            if id_elem not in self.modelo.elementos: continue
            ni, nj, _ = self.modelo.elementos[id_elem]
            p1, p2 = self.modelo.nodos[ni], self.modelo.nodos[nj]

            try:
                T, L = matriz_transformacion_portico_3d(p1, p2)
                R = T[:3, :3]
            except ValueError:
                continue

            if L < 1e-9: continue

            tipo_carga = carga['datos_carga'][0]
            fep_local = np.zeros(12)
            w_global = np.zeros(3)
            w_local = np.zeros(3)
            mt = 0.0

            # 5. Cálculo y acumulación paramétrica para distribuciones uniformes explícitas
            if tipo_carga == 'uniforme':
                _, wx, wy, wz, mt_val = carga['datos_carga']
                mt = mt_val
                w_global = np.array([wx, wy, wz])
                w_local = R @ w_global # [wlx, wly, wlz]

                cargas_dist_base[id_hipotesis][id_elem]['axial_x'] += w_local[0]
                cargas_dist_base[id_hipotesis][id_elem]['torsion_mx'] += mt

                tramos_y_nuevos = []
                if abs(w_local[1]) > 1e-9: # Carga en Y local
                    tramos_y_nuevos = [(0.0, w_local[1]), (1.0, w_local[1])]

                tramos_z_nuevos = []
                if abs(w_local[2]) > 1e-9: # Carga en Z local
                    tramos_z_nuevos = [(0.0, w_local[2]), (1.0, w_local[2])]

                if tramos_y_nuevos:
                    tramos_y_actuales = cargas_dist_base[id_hipotesis][id_elem]['tramos_y']
                    cargas_dist_base[id_hipotesis][id_elem]['tramos_y'] =
_sumar_tramos_lineales(tramos_y_actuales, tramos_y_nuevos)

                if tramos_z_nuevos:
                    tramos_z_actuales = cargas_dist_base[id_hipotesis][id_elem]['tramos_z']
```

```
                    cargas_dist_base[id_hipotesis][id_elem]['tramos_z'] =
_sumar_tramos_lineales(tramos_z_actuales, tramos_z_nuevos)

                # Asignación FEP para uniforme
                fep_local[0] = -w_local[0] * L / 2
                fep_local[6] = -w_local[0] * L / 2
                fep_local[1] = -w_local[1] * L / 2
                fep_local[7] = -w_local[1] * L / 2
                fep_local[5] = -w_local[1] * L**2 / 12 # Mz_i (FEP) es Horario (-)
                fep_local[11] = w_local[1] * L**2 / 12 # Mz_j (FEP) es Anti-Horario (+)
                fep_local[2] = -w_local[2] * L / 2
                fep_local[8] = -w_local[2] * L / 2
                fep_local[4] = w_local[2] * L**2 / 12  # My_i (FEP) es Anti-Horario (+)
                fep_local[10] = -w_local[2] * L**2 / 12 # My_j (FEP) es Horario (-)
                fep_local[3] = -mt * L / 2
                fep_local[9] = -mt * L / 2

            # 6. Ejecución del algoritmo de superposición para distribuciones de tramos
seccionales
            elif tipo_carga == 'tramos_locales':
                # El formato es ('tramos_locales', 'y' o 'z', [(p_norm1, q1), (p_norm2, q2),
...])
                _, eje_local, lista_de_puntos = carga['datos_carga']
                print(f"\n[DEBUG proc_cargas Elem {id_elem} Hip {id_hipotesis}] Procesando
'tramos_locales' Eje='{eje_local}' Puntos={lista_de_puntos}") # <-- PRINT

                if eje_local == 'y':
                    clave_tramos = 'tramos_y'
                elif eje_local == 'z':
                    clave_tramos = 'tramos_z'
                else:
                    print(f"ADVERTENCIA: Eje '{eje_local}' no soportado para tramos en Elem
{id_elem}")
                    continue

                tramos_actuales = cargas_dist_base[id_hipotesis][id_elem][clave_tramos]
                cargas_dist_base[id_hipotesis][id_elem][clave_tramos] =
_sumar_tramos_lineales(tramos_actuales, lista_de_puntos)

                fep_superposicion = np.zeros(4) # (R_i, M_i, R_j, M_j)

                for i in range(len(lista_de_puntos) - 1):
                    (p_a_norm, q_a) = lista_de_puntos[i]
                    (p_b_norm, q_b) = lista_de_puntos[i+1]

                    a = p_a_norm * L
                    b = p_b_norm * L

                    if abs(b - a) < 1e-9: continue

                    w_unif = q_a
                    w_tria = q_b - q_a
                    print(f"  [DEBUG Elem {id_elem} Seg {i}] Tramo [{a:.2f}m - {b:.2f}m],
w_unif={w_unif:.2f}, w_tria={w_tria:.2f}")

                    if abs(w_unif) > 1e-9:
```

```
                        print(f"    [DEBUG Elem {id_elem} Seg {i}] Llamando
_fep_uniforme_parcial(L={L:.2f}, w={w_unif:.2f}, a={a:.2f}, b={b:.2f})") # <-- PRINT
                        fep_u = np.array(_fep_uniforme_parcial(L, w_unif, a, b))
                        print(f"      Resultado FEP Unif: {fep_u}")
                        fep_superposicion += fep_u

                    if abs(w_tria) > 1e-9:
                        print(f"    [DEBUG Elem {id_elem} Seg {i}] Llamando
_fep_triangular_parcial(L={L:.2f}, w_pico={w_tria:.2f}, a={a:.2f}, b={b:.2f})") # <-- PRINT
                        fep_t = np.array(_fep_triangular_parcial(L, w_tria, a, b))
                        print(f"      Resultado FEP Tria: {fep_t}")
                        fep_superposicion += fep_t

                R_i_total, M_i_total, R_j_total, M_j_total = fep_superposicion
                print(f"  [DEBUG Elem {id_elem}] FEP 2D Totales (R_i, M_i, R_j, M_j):
{fep_superposicion}")

                if eje_local == 'y': # Carga en 'y' local -> Momento en 'z' local
                    fep_local[1] = -R_i_total
                    fep_local[7] = -R_j_total
                    fep_local[5] = -M_i_total  # Mz_i (FEP) es Clockwise (-)
                    fep_local[11] = -M_j_total # Mz_j (FEP) es Anti-Clockwise (+) (M_j_total
es neg)
                elif eje_local == 'z': # Carga en 'z' local -> Momento en 'y' local
                    fep_local[2] = -R_i_total
                    fep_local[8] = -R_j_total
                    fep_local[4] = M_i_total   # My_i (FEP) es Anti-Clockwise (+) (M_i_total
es pos)
                    fep_local[10] = M_j_total # My_j (FEP) es Clockwise (-) (M_j_total es
neg)
                print(f"  [DEBUG Elem {id_elem}] fep_local (12 comp) calculado: {fep_local}")

            else:
                pass

            # 7. Ensamblaje (COMÚN a todos los tipos que generan FEPs)
            if np.any(fep_local): # Solo ensamblar si hay FEPs
                feps_base[id_hipotesis][id_elem] = feps_base[id_hipotesis].get(id_elem,
np.zeros(12)) + fep_local

                fep_global = T.T @ fep_local
                indices_globales = list(range((ni-1)*6, ni*6)) + list(range((nj-1)*6, nj*6))
                vectores_carga_base[id_hipotesis][indices_globales] -= fep_global

                log_reporte_agrupacion.append({
                    'tipo': 'elemento', 'id_hipotesis': id_hipotesis, 'id_elem': id_elem,
                    'w_global': w_global, 'w_local': w_local, 'mt': mt, 'fep_local':
fep_local, 'fep_global': fep_global,
                    'tipo_carga_aplicada': tipo_carga
                })

        # 8. Módulo de traducción bidimensional: Recreación y purga de logs previos
        if hasattr(self.modelo, 'datos_reporte_losas'):
            self.modelo.datos_reporte_losas.clear()

        print("--- Iniciando traducción de Cargas Superficiales a Cargas de Elemento ---")
```

```
        for id_carga_sup, carga_sup in self.modelo.cargas_superficiales.items():
            try:
                id_losa = carga_sup['id_losa']
                wz = carga_sup['magnitud']
                id_hipotesis = carga_sup['id_hipotesis']

                # Asegurarnos que la hipótesis existe en nuestro diccionario
                if id_hipotesis not in vectores_carga_base:
                    vectores_carga_base[id_hipotesis] = np.zeros(num_gdl)

                print(f"  -> Procesando CargaSup {id_carga_sup} en Losa {id_losa} (Hip:
{id_hipotesis})...")

                # Llamamos al distribuidor "just-in-time"
                cargas_generadas, log_datos = traducir_carga_losa_a_cargas_lineales(
                    id_losa, wz, id_hipotesis, self.modelo
                )

                # Guardamos el log para el reporte
                if hasattr(self.modelo, 'datos_reporte_losas'):
                    self.modelo.datos_reporte_losas.append(log_datos)

                # 9. Asignación tributaria de cargas de losa hacia los porticos
                for carga_elem_gen in cargas_generadas:
                    id_elem = carga_elem_gen['id_elemento']
                    datos_carga = carga_elem_gen['datos_carga']

                    if id_elem not in self.modelo.elementos: continue

                    ni, nj, _ = self.modelo.elementos[id_elem]

                    if ni not in self.modelo.nodos or nj not in self.modelo.nodos:
                        print(f"Advertencia: Omitiendo carga generada en Elem {id_elem} por
nodos faltantes ({ni}, {nj})")
                        continue
                    p1, p2 = self.modelo.nodos[ni], self.modelo.nodos[nj]

                    try:
                        T, L = matriz_transformacion_portico_3d(p1, p2)
                    except ValueError:
                        continue
                    if L < 1e-9: continue

                    tipo_carga_losa = datos_carga[0]
                    fep_local = np.zeros(12)
                    w_local = np.zeros(3)
                    mt = 0.0

                    if tipo_carga_losa == 'uniforme':

                        _, wlx, wly, wlz, mlt = datos_carga # (uniforme, wx_local, wy_local,
wz_local, mt_local)
                        w_local = np.array([wlx, wly, wlz])
                        mt = mlt

                        cargas_dist_base[id_hipotesis][id_elem]['axial_x'] += w_local[0]
                        cargas_dist_base[id_hipotesis][id_elem]['torsion_mx'] += mt
```

```
                        tramos_y_nuevos = []
                        if abs(w_local[1]) > 1e-9: # Carga en Y local
                            tramos_y_nuevos = [(0.0, w_local[1]), (1.0, w_local[1])]

                        tramos_z_nuevos = []
                        if abs(w_local[2]) > 1e-9: # Carga en Z local
                            tramos_z_nuevos = [(0.0, w_local[2]), (1.0, w_local[2])]

                        if tramos_y_nuevos:
                            tramos_y_actuales =
cargas_dist_base[id_hipotesis][id_elem]['tramos_y']
                            cargas_dist_base[id_hipotesis][id_elem]['tramos_y'] =
_sumar_tramos_lineales(tramos_y_actuales, tramos_y_nuevos)

                        if tramos_z_nuevos:
                            tramos_z_actuales =
cargas_dist_base[id_hipotesis][id_elem]['tramos_z']
                            cargas_dist_base[id_hipotesis][id_elem]['tramos_z'] =
_sumar_tramos_lineales(tramos_z_actuales, tramos_z_nuevos)

                        # Asignación FEP para uniforme
                        fep_local[0] = -w_local[0] * L / 2
                        fep_local[6] = -w_local[0] * L / 2
                        fep_local[1] = -w_local[1] * L / 2
                        fep_local[7] = -w_local[1] * L / 2
                        fep_local[5] = -w_local[1] * L**2 / 12 # Mz_i
                        fep_local[11] = w_local[1] * L**2 / 12 # Mz_j
                        fep_local[2] = -w_local[2] * L / 2
                        fep_local[8] = -w_local[2] * L / 2
                        fep_local[4] = w_local[2] * L**2 / 12  # My_i
                        fep_local[10] = -w_local[2] * L**2 / 12 # My_j
                        fep_local[3] = -mt * L / 2
                        fep_local[9] = -mt * L / 2

                    elif tipo_carga_losa == 'tramos_locales':
                        # El formato es ('tramos_locales', 'y' o 'z', [(p_norm1, q1),
(p_norm2, q2), ...])
                        _, eje_local, lista_de_puntos = datos_carga
                        print(f"\n[DEBUG proc_cargas Losa->Elem {id_elem} Hip {id_hipotesis}]
Procesando 'tramos_locales' Eje='{eje_local}' Puntos={lista_de_puntos}") # <-- PRINT

                        # 10. Bloque de auditoría pos-integración
                        id_viga_target = 3
                        if id_elem == id_viga_target:
                            print(f"\n[DEBUG SUPERPOSICIÓN] Viga {id_elem} recibiendo carga
de Losa...")
                            print(f"   -> Carga Entrante: {lista_de_puntos}")

                            clave_tramos_debug = 'tramos_y' if eje_local == 'y' else
'tramos_z'
                            tramos_antes =
cargas_dist_base[id_hipotesis][id_elem][clave_tramos_debug]
                            print(f"   -> Carga Acumulada ANTES: {tramos_antes}")

                        if eje_local == 'y':
                            clave_tramos = 'tramos_y'
```

```
                        elif eje_local == 'z':
                            clave_tramos = 'tramos_z'
                        else:
                            print(f"ADVERTENCIA: Eje '{eje_local}' no soportado para tramos
en Losa->Elem {id_elem}")
                            continue

                        tramos_actuales =
cargas_dist_base[id_hipotesis][id_elem][clave_tramos]
                        nueva_lista_sumada = _sumar_tramos_lineales(tramos_actuales,
lista_de_puntos)
                        cargas_dist_base[id_hipotesis][id_elem][clave_tramos] =
nueva_lista_sumada

                        if id_elem == id_viga_target:
                            print(f"   -> Carga Acumulada DESPUÉS: {nueva_lista_sumada}")
                            puntos_x = [p[0] for p in nueva_lista_sumada]
                            for i in range(len(puntos_x)-1):
                                if puntos_x[i+1] - puntos_x[i] < 1e-4: # Si hay puntos
demasiado cerca
                                    print(f"   [¡ALERTA!] Puntos muy cercanos detectados en
{puntos_x[i]} y {puntos_x[i+1]}. Posible error de interpolación.")

                        fep_superposicion = np.zeros(4) # (R_i, M_i, R_j, M_j)

                        for i in range(len(lista_de_puntos) - 1):
                            (p_a_norm, q_a) = lista_de_puntos[i]
                            (p_b_norm, q_b) = lista_de_puntos[i+1]

                            a = p_a_norm * L
                            b = p_b_norm * L

                            if abs(b - a) < 1e-9: continue

                            w_unif = q_a
                            w_tria = q_b - q_a

                            print(f"  [DEBUG Losa->Elem {id_elem} Seg {i}] Tramo [{a:.2f}m -
{b:.2f}m], w_unif={w_unif:.2f}, w_tria={w_tria:.2f}")

                            if abs(w_unif) > 1e-9:
                                print(f"    [DEBUG Losa->Elem {id_elem} Seg {i}] Llamando
_fep_uniforme_parcial(L={L:.2f}, w={w_unif:.2f}, a={a:.2f}, b={b:.2f})")
                                fep_u = np.array(_fep_uniforme_parcial(L, w_unif, a, b))
                                print(f"      Resultado FEP Unif: {fep_u}")
                                fep_superposicion += fep_u

                            if abs(w_tria) > 1e-9:
                                print(f"    [DEBUG Losa->Elem {id_elem} Seg {i}] Llamando
_fep_triangular_parcial(L={L:.2f}, w_pico={w_tria:.2f}, a={a:.2f}, b={b:.2f})")
                                fep_t = np.array(_fep_triangular_parcial(L, w_tria, a, b))
                                print(f"      Resultado FEP Tria: {fep_t}")
                                fep_superposicion += fep_t

                        R_i_total, M_i_total, R_j_total, M_j_total = fep_superposicion
                        print(f"  [DEBUG Losa->Elem {id_elem}] FEP 2D Totales (R_i, M_i, R_j,
M_j): {fep_superposicion}")
```

```
                        if eje_local == 'y': # Carga en 'y' local -> Momento en 'z' local
                            fep_local[1] = -R_i_total
                            fep_local[7] = -R_j_total
                            fep_local[5] = -M_i_total # Mz_i (FEP) es Horario (-)
                            fep_local[11] = -M_j_total # Mz_j (FEP) es Anti-Horario (+)
(M_j_total es neg)
                        elif eje_local == 'z': # Carga en 'z' local -> Momento en 'y' local
                            fep_local[2] = -R_i_total
                            fep_local[8] = -R_j_total
                            fep_local[4] = M_i_total # My_i (FEP) es Anti-Horario (+)
(M_i_total es pos)
                            fep_local[10] = M_j_total # My_j (FEP) es Horario (-) (M_j_total
es neg)

                        print(f"  [DEBUG Losa->Elem {id_elem}] fep_local (12 comp) calculado:
{fep_local}")

                        pass

                    else:
                        pass

                    # 11. Sumatoria final sobre el subespacio global correspondiente a la
matriz incidente
                    if np.any(fep_local):
                        feps_base[id_hipotesis][id_elem] =
feps_base[id_hipotesis].get(id_elem, np.zeros(12)) + fep_local

                        fep_global = T.T @ fep_local
                        indices_globales = list(range((ni-1)*6, ni*6)) + list(range((nj-1)*6,
nj*6))
                        vectores_carga_base[id_hipotesis][indices_globales] -= fep_global

                        log_reporte_agrupacion.append({
                            'tipo': 'losa_a_elemento', 'id_hipotesis': id_hipotesis,
'id_elem': id_elem,
                            'id_losa_origen': id_losa, 'w_local': w_local, 'mt': mt,
'fep_local': fep_local,
                            'tipo_carga_aplicada': tipo_carga_losa
                        })

            except Exception as e:
                print(f"ADVERTENCIA: No se pudo procesar la carga superficial {id_carga_sup}.
Error: {e}")

        print("--- Fin de traducción de Cargas Superficiales ---")

        return vectores_carga_base, feps_base, cargas_dist_base, log_reporte_agrupacion

    def resolver_combinaciones(self, usar_timoshenko=True, usar_pp=False):
        """
        Orquestación computacional del método de rigidez para múltiples estados
        de carga combinados. Gestiona la solución estática del sistema evaluando
        la respuesta estructural frente a envolventes de diseño.
        """
        # 1. Validación inicial del número de variables fundamentales del sistema
```

```
        num_gdl = len(self.modelo.nodos) * 6
        if num_gdl == 0: raise ValueError("No hay nodos definidos para el cálculo.")

        # 2. Inicialización de la extracción y categorización de cargas
        vectores_carga_base, feps_base, cargas_dist_base, log_agrupacion =
self._agrupar_cargas_por_hipotesis()

        # 3. Solicitud de ensamblaje de la matriz de rigidez unificada
        solucionador = Solucionador3D(self.modelo, usar_timoshenko=usar_timoshenko)
        K_global, datos_ensamblaje_reporte = solucionador.ensamblar_K_global()

        # 4. Declaración del diccionario contenedor de respuestas numéricas
        resultados_por_combinacion = defaultdict(dict)

        resultados_por_combinacion['reporte_global_data'] = {
            'ensamblaje': datos_ensamblaje_reporte,
            'agrupacion': log_agrupacion
        }

        # 5. Lógica PP=0 (JIT Concurrente)
        if usar_pp:
            print("[INFO] Flag 'usar_pp' detectado. Generando cargas PP (JIT)...")
            # (Esta función la creamos en la Etapa 2)
            (vector_pp, feps_pp, cargas_dist_pp, log_pp_1d) = self._generar_cargas_pp_auto()
            # Guardar el log para el reporte
            resultados_por_combinacion['reporte_global_data']['log_pp_1d'] = log_pp_1d
        else:
            print("[INFO] Flag 'usar_pp' no detectado. Omitiendo cálculo de PP.")
            # Crear contenedores vacíos (Lógica PP=0)
            (vector_pp, feps_pp, cargas_dist_pp) = (np.zeros(num_gdl), {}, defaultdict(dict))
            resultados_por_combinacion['reporte_global_data']['log_pp_1d'] = [] # Log vacío

        # 6. Agrupar Hipótesis de Usuario
        hipotesis_por_tipo = defaultdict(list)
        for id_hip, datos_hip in self.modelo.hipotesis_de_carga.items():
            hipotesis_por_tipo[datos_hip['tipo']].append(id_hip)

        # 7. Bucle Principal de Combinaciones
        for combo in self.modelo.combinaciones:
            tipos_requeridos = combo.factores.keys()

            factor_D = combo.factores.get('D', 0.0)

            # 8. Generador de Casos Alternantes (Usuario)
            grupos_a_combinar = []
            for tipo in tipos_requeridos:
                if hipotesis_por_tipo.get(tipo):
                    grupos_a_combinar.append(hipotesis_por_tipo[tipo])
                else:
                    # Si no hay cargas 'D' de usuario, PERO SÍ hay factor D y SÍ usamos PP,
                    # ponemos un 'None' para que el bucle se ejecute al menos una vez.
                    if tipo == 'D' and usar_pp and abs(factor_D) > 1e-9:
                         grupos_a_combinar.append([None])
                    else:
                         grupos_a_combinar.append([None])

            # 9. Iterar sobre cada caso
```

```
            for sub_combinacion_ids in itertools.product(*grupos_a_combinar):

                # 10. Lógica de Inyección de PP (Concurrente)
                # Inicializamos los totales CON el PP ya escalado
                fuerza_total_sub_combo = vector_pp * factor_D
                feps_sub_combo = defaultdict(lambda: np.zeros(12))
                cargas_dist_sub_combo = defaultdict(lambda: {
                    'tramos_y': [(0.0, 0.0), (1.0, 0.0)],
                    'tramos_z': [(0.0, 0.0), (1.0, 0.0)],
                    'axial_x': 0.0,
                    'torsion_mx': 0.0
                })
                nombre_sub_calculo_partes = []

                if abs(factor_D) > 1e-9 and usar_pp:
                    nombre_sub_calculo_partes.append("PP_Auto")

                    # Inyectar FEPs PP
                    for id_elem, fep_pp in feps_pp.items():
                        feps_sub_combo[id_elem] += fep_pp * factor_D

                    # Inyectar Cargas Dist PP
                    for id_elem, cargas_pp in cargas_dist_pp.items():
                        cargas_dist_sub_combo[id_elem]['axial_x'] += cargas_pp['axial_x'] *
factor_D
                        cargas_dist_sub_combo[id_elem]['torsion_mx'] +=
cargas_pp['torsion_mx'] * factor_D

                        tramos_y_pp_fact = [(p, q * factor_D) for p, q in
cargas_pp.get('tramos_y', [])]
                        tramos_z_pp_fact = [(p, q * factor_D) for p, q in
cargas_pp.get('tramos_z', [])]

                        if tramos_y_pp_fact:
                            cargas_dist_sub_combo[id_elem]['tramos_y'] =
_sumar_tramos_lineales(
                                cargas_dist_sub_combo[id_elem]['tramos_y'], tramos_y_pp_fact
                            )
                        if tramos_z_pp_fact:
                            cargas_dist_sub_combo[id_elem]['tramos_z'] =
_sumar_tramos_lineales(
                                cargas_dist_sub_combo[id_elem]['tramos_z'], tramos_z_pp_fact
                            )

                # 11. Suma de Cargas de Usuario (Alternantes) ---
                # Este bucle ahora suma "encima" de los vectores de PP
                for id_hip in sub_combinacion_ids:
                    if id_hip is None:
                        continue

                    tipo_carga = self.modelo.hipotesis_de_carga[id_hip]['tipo']
                    factor = combo.factores[tipo_carga]

                    if id_hip in vectores_carga_base:
                        fuerza_total_sub_combo += vectores_carga_base[id_hip] * factor

                    if id_hip in feps_base:
```

```
                        for id_elem, fep in feps_base[id_hip].items():
                            feps_sub_combo[id_elem] += fep * factor

                    if id_hip in cargas_dist_base:
                         for id_elem, cargas_elem_base in cargas_dist_base[id_hip].items():
                            cargas_dist_sub_combo[id_elem]['axial_x'] +=
cargas_elem_base['axial_x'] * factor
                            cargas_dist_sub_combo[id_elem]['torsion_mx'] +=
cargas_elem_base['torsion_mx'] * factor

                            tramos_y_fact = [(p, q * factor) for p, q in
cargas_elem_base['tramos_y'] if any(abs(q) > 1e-9 for _, q in cargas_elem_base['tramos_y'])]
                            tramos_z_fact = [(p, q * factor) for p, q in
cargas_elem_base['tramos_z'] if any(abs(q) > 1e-9 for _, q in cargas_elem_base['tramos_z'])]

                            if tramos_y_fact:
                                cargas_dist_sub_combo[id_elem]['tramos_y'] =
_sumar_tramos_lineales(cargas_dist_sub_combo[id_elem]['tramos_y'], tramos_y_fact)
                            if tramos_z_fact:
                                cargas_dist_sub_combo[id_elem]['tramos_z'] =
_sumar_tramos_lineales(cargas_dist_sub_combo[id_elem]['tramos_z'], tramos_z_fact)

                    nombre_sub_calculo_partes.append(self.modelo.hipotesis_de_carga[id_hip]['
nombre'])

                # 12. Resolución
                if not nombre_sub_calculo_partes:
                    continue # No había ni PP ni cargas de usuario para este tipo

                nombre_sub_calculo = "; ".join(sorted(nombre_sub_calculo_partes))

                resultados_completos = solucionador.resolver(fuerza_total_sub_combo,
K_global, dict(feps_sub_combo))

                resultados_por_combinacion[combo.nombre][nombre_sub_calculo] = {
                    'desplazamientos': resultados_completos['desplazamientos'],
                    'reacciones': resultados_completos['reacciones'],
                    'fuerzas_internas': resultados_completos['fuerzas_internas'],
                    'cargas_distribuidas': dict(cargas_dist_sub_combo),
                    'reporte_resolucion': resultados_completos['reporte_resolucion'],
                    'reporte_vectores': {
                        'F_total': fuerza_total_sub_combo,
                        'FEP_elem': dict(feps_sub_combo)
                    }
                }

        if len(resultados_por_combinacion) <= 1:
            raise ValueError("Ninguna combinación de carga pudo ser calculada. Verifique que
las cargas definidas correspondan a los tipos de carga en sus combinaciones, o que al menos
exista una hipótesis de carga.")

        return dict(resultados_por_combinacion)
```

Anexo 4: Script distribuidor_losas.py

```
"""
Módulo: distribuidor_losas.py
Descripción: Módulo fundamental para la transferencia de cargas. Calcula áreas
tributarias de losas y distribuye las cargas superficiales hacia los elementos
1D (vigas) perimetrales en forma de cargas distribuidas lineales y uniformes.
"""

# ------------------------------------------------------------------------
# VORTEX 3D - Análisis Estructural de Edificaciones 3D y Diseño Normativo
# de Vigas y Columnas
# Copyright (C) 2026 Diego Oliver Vargas Moya & Luis Alberto Ortiz Morales
#
# Este programa es software libre: puedes redistribuirlo y/o modificarlo
# bajo los términos de la Licencia Pública General GNU (GNU GPL) publicada
# por la Free Software Foundation, ya sea la versión 3 de la Licencia, o
# (a tu elección) cualquier versión posterior.
#
# Este programa se distribuye con la esperanza de que sea útil,
# pero SIN GARANTÍA ALGUNA; ni siquiera la garantía implícita
# MERCANTIL o de APTITUD PARA UN PROPÓSITO DETERMINADO.
# Consulta los detalles de la Licencia Pública General GNU para más
# información.
#
# Deberías haber recibido una copia de la Licencia Pública General GNU
# junto a este programa. En caso contrario, consulta
# <https://www.gnu.org/licenses/>.
# ------------------------------------------------------------------------

#====================================================================
# I. IMPORTACIONES Y DEPENDENCIAS
#====================================================================
import numpy as np
import math
from calc import matriz_transformacion_portico_3d
from collections import defaultdict # Nuevo

#====================================================================
# II. FUNCIONES GEOMÉTRICAS AUXILIARES
#====================================================================

def _encontrar_vigas_de_borde(losa, modelo):
    """
    Identifica los elementos estructurales que coinciden geométricamente con
    el perímetro de la losa analizada.
    """
    # 1. Definición topológica de los bordes de la losa
    vigas_borde = {}
    nodos_losa = losa['nodos']
    bordes_losa_ordenados = {tuple(sorted((nodos_losa[i], nodos_losa[(i + 1) % 4]))) for i in
range(4)}

    # 2. Mapeo de vigas colindantes
    for id_elem, (ni, nj, _) in modelo.elementos.items():
```

```
            borde_elem_ordenado = tuple(sorted((ni, nj)))
            if borde_elem_ordenado in bordes_losa_ordenados:
                vigas_borde[borde_elem_ordenado] = id_elem

    return vigas_borde

def _encontrar_losa_adyacente(borde_actual, id_losa_actual, modelo):
    """
    Busca losas contiguas que compartan un borde específico para determinar
    condiciones de continuidad.
    """
    # 1. Búsqueda de coincidencia de aristas en el modelo
    borde_ordenado_actual = tuple(sorted(borde_actual))
    for id_losa, datos_losa in modelo.losas.items():
        if id_losa == id_losa_actual:
            continue
        nodos_otra = datos_losa['nodos']
        bordes_otra = [tuple(sorted((nodos_otra[i], nodos_otra[(i + 1) % 4]))) for i in
range(4)]
        if borde_ordenado_actual in bordes_otra:
            return id_losa
    return None

def _rotar_vector_2d(vector, angulo_grados):
    """
    Aplica una matriz de rotación en el plano 2D a un vector dado.
    """
    # 1. Transformación angular y producto matricial
    angulo_rad = math.radians(angulo_grados)
    matriz_rotacion = np.array([
        [math.cos(angulo_rad), -math.sin(angulo_rad)],
        [math.sin(angulo_rad), math.cos(angulo_rad)]
    ])
    return matriz_rotacion @ vector

def _ordenar_vertices_poligono(vertices):
    """
    Ordena un conjunto de coordenadas de forma radial respecto a su centroide.
    """
    # 1. Cálculo de centroide y ordenamiento polar
    if len(vertices) <= 3:
        return vertices
    centroide = np.mean(vertices, axis=0)
    return sorted(vertices, key=lambda v: math.atan2(v[1] - centroide[1], v[0] -
centroide[0]))

def _calcular_interseccion_lineas(p1, v1, p2, v2):
    """
    Calcula el punto de intersección entre dos líneas definidas por un punto y un vector.
    """
    # 1. Resolución del sistema lineal 2x2
    p1_2d, v1_2d = p1[:2], v1[:2]
    p2_2d, v2_2d = p2[:2], v2[:2]
    A = np.array([[v1_2d[0], -v2_2d[0]], [v1_2d[1], -v2_2d[1]]])
    b = p2_2d - p1_2d
    try:
        t, _ = np.linalg.solve(A, b)
```

```
            punto_interseccion = p1 + t * v1
            return punto_interseccion
        except np.linalg.LinAlgError:
            print("      [DEBUG] Las líneas de aporte son paralelas, no se intersectan.")
            return None

    def _calcular_area_poligono(vertices):
        """
        Calcula el área de un polígono cerrado utilizando la fórmula de Gauss.
        """
        # 1. Aplicación analítica de la fórmula de áreas topográficas
        if not isinstance(vertices, np.ndarray):
            vertices = np.array(vertices)
        if vertices.ndim == 1 or vertices.shape[0] < 3:
            return 0.0

        x = vertices[:, 0]
        y = vertices[:, 1]
        return 0.5 * np.abs(np.dot(x, np.roll(y, -1)) - np.dot(y, np.roll(x, -1)))

    #====================================================================
    # III. PROCESAMIENTO DE BORDES Y COLINEALIDAD
    #====================================================================
    def _es_colineal_y_en_segmento(coord_p, coord_a, coord_b, tol=1e-3):
        """
        Verifica si el punto p (coordenadas) es colineal con el segmento AB y está
        dentro o sobre el segmento AB.
        """
        coord_a = np.array(coord_a)
        coord_b = np.array(coord_b)
        coord_p = np.array(coord_p)

        vector_ab = coord_b - coord_a
        vector_ap = coord_p - coord_a

        # 1. Colinealidad: El área del triángulo ABP debe ser cero (producto cruzado).
        cross_product = np.cross(vector_ab, vector_ap)
        if np.linalg.norm(cross_product) > tol:
            return False

        # 2. En el segmento: La proyección de AP sobre AB debe estar entre 0 y ||AB||.
        longitud_ab_sq = np.dot(vector_ab, vector_ab)
        if longitud_ab_sq < tol:
            return np.linalg.norm(vector_ap) < tol

        proyeccion_ap_sobre_ab = np.dot(vector_ap, vector_ab)

        if proyeccion_ap_sobre_ab < -tol:
            return False # P está antes de A
        if proyeccion_ap_sobre_ab > longitud_ab_sq + tol:
            return False # P está después de B

        return True

    def _obtener_elementos_en_borde(coord_borde_inicio, coord_borde_fin, losa_borde_ids, modelo):
        """
        Devuelve la lista de elementos 1D cuyos nodos son colineales con el borde
```

```
    principal de la losa y están contenidos en él. La lista se devuelve ordenada
    por proximidad al inicio del borde.

    Devuelve: [{'id_viga': ID, 'nodos_viga': (ni, nj), 'longitud': L}, ...]
    """
    elementos_de_borde = []

    coord_borde_inicio_np = np.array(coord_borde_inicio)
    coord_borde_fin_np = np.array(coord_borde_fin)

    for id_elem, (ni, nj, _) in modelo.elementos.items():
        coord_ni = np.array(modelo.nodos.get(ni, [0,0,0]))
        coord_nj = np.array(modelo.nodos.get(nj, [0,0,0]))

        # 1. Verificar si AMBOS nodos del elemento son colineales y están en el segmento
principal
        es_ni_en_segmento = _es_colineal_y_en_segmento(coord_ni, coord_borde_inicio_np,
coord_borde_fin_np)
        es_nj_en_segmento = _es_colineal_y_en_segmento(coord_nj, coord_borde_inicio_np,
coord_borde_fin_np)

        if es_ni_en_segmento and es_nj_en_segmento and ni != nj:
             longitud = np.linalg.norm(coord_ni - coord_nj)
             if longitud > 1e-6:
                elementos_de_borde.append({
                    'id_viga': id_elem,
                    'nodos_viga': (ni, nj),
                    'longitud': longitud
                })

    # 2. Ordenar por proximidad a coord_borde_inicio (el primer nodo de la losa)
    def ordenar_por_proximidad(item):
        ni, nj = item['nodos_viga']
        coord_ni = np.array(modelo.nodos[ni])
        coord_nj = np.array(modelo.nodos[nj])

        dist_ni = np.linalg.norm(coord_ni - coord_borde_inicio_np)
        dist_nj = np.linalg.norm(coord_nj - coord_borde_inicio_np)

        return min(dist_ni, dist_nj)

    elementos_de_borde.sort(key=ordenar_por_proximidad)

    # 3. Corregir la orientación de los nodos de la viga para que coincida con el borde
    elementos_finales_ordenados = []
    nodo_anterior = losa_borde_ids[0]

    for item in elementos_de_borde:
        n1, n2 = item['nodos_viga']
        if n2 == nodo_anterior:
            item['nodos_viga'] = (n2, n1)
            nodo_anterior = n1
        elif n1 == nodo_anterior:
            nodo_anterior = n2
        else:
             # Este elemento no está conectado al borde principal, o es un caso de modelado
complejo.
```

```
            # Lo mantenemos para el flujo de la viga, asumiendo que el ordenamiento por
proximidad lo maneja.
            pass

        elementos_finales_ordenados.append(item)

    return [e for e in elementos_finales_ordenados if e['longitud'] > 1e-6]

#======================================================================
# IV. DISTRIBUCIÓN UNIDIRECCIONAL DE CARGAS
#======================================================================
def _manejar_distribucion_unidireccional(losa, wz, id_hipotesis, vigas_borde, modelo):
    """
    Calcula y distribuye cargas superficiales hacia apoyos paralelos mediante
    áreas tributarias rectangulares [McCormac & Brown, 2015].
    """
    # 1. Inicialización de registros de cálculo
    log_datos = {
        'id_losa': losa['id'],
        'tipo': 'unidireccional',
        'wz': wz,
        'ancho_total': 0,
        'ancho_aporte': 0,
        'w_lineal': 0,
        'vigas_cargadas': []
    }
    print(f"\n--- [DEBUG] Iniciando Distribución Unidireccional para Losa ID: {losa['id']} --
-")
    print(f"    Carga Superficial (wz): {wz:.2f} kPa")

    eje_uni = losa.get('eje_uni', 'Global Y')
    nodos_losa = losa['nodos']

    eje_paralelo_viga = np.array([1.0, 0.0, 0.0]) if 'Y' in eje_uni else np.array([0.0, 1.0,
0.0])

    bordes_principales_losa = [(nodos_losa[i], nodos_losa[(i + 1) % 4]) for i in range(4)]

    # 2. Identificar los 2 bordes principales paralelos al eje de distribución
    bordes_paralelos = []
    for borde_original in bordes_principales_losa:
        p_inicio = np.array(modelo.nodos[borde_original[0]])
        p_fin = np.array(modelo.nodos[borde_original[1]])
        vector_borde = p_fin - p_inicio
        norm_vector = np.linalg.norm(vector_borde)
        if norm_vector < 1e-9: continue

        vector_borde_unitario = vector_borde / norm_vector

        if abs(np.dot(vector_borde_unitario, eje_paralelo_viga)) > 0.98:
            bordes_paralelos.append(borde_original)

    if len(bordes_paralelos) != 2:
        raise ValueError(f"Error en Losa {losa['id']}: Para distribución unidireccional, se
requieren exactamente 2 bordes de apoyo paralelos y opuestos. Se encontraron
{len(bordes_paralelos)}.")
```

```
    # 3. Calcular ancho de aporte total
    p1_viga1 = np.array(modelo.nodos[bordes_paralelos[0][0]])
    p2_viga_opuesta = np.array(modelo.nodos[bordes_paralelos[1][0]])
    v_base = np.array(modelo.nodos[bordes_paralelos[0][1]]) - p1_viga1
    v_punto = p2_viga_opuesta - p1_viga1

    norm_v_base = np.linalg.norm(v_base)
    if norm_v_base < 1e-9:
         raise ValueError(f"Error: Longitud de borde en losa {losa['id']} es cero.")

    v_base_unitario = v_base / norm_v_base
    proyeccion = np.dot(v_punto, v_base_unitario) * v_base_unitario
    ancho_aporte_total = np.linalg.norm(v_punto - proyeccion)

    ancho_aporte_por_viga = ancho_aporte_total / 2.0
    w_lineal = wz * ancho_aporte_por_viga

    print(f"    Ancho total de aporte (distancia entre bordes): {ancho_aporte_total:.2f} m")
    print(f"    Ancho de aporte por borde: {ancho_aporte_por_viga:.2f} m")
    print(f"    Cálculo Carga Lineal: {wz:.2f} kPa * {ancho_aporte_por_viga:.2f} m =
{w_lineal:.2f} kN/m")

    log_datos['ancho_total'] = ancho_aporte_total
    log_datos['ancho_aporte'] = ancho_aporte_por_viga
    log_datos['w_lineal'] = w_lineal

    cargas_generadas = []

    # 4. OBTENER *TODOS* LOS ELEMENTOS DE BORDE Y ASIGNAR LA CARGA
    for borde_principal_ids in bordes_paralelos:
        p_inicio_borde = modelo.nodos[borde_principal_ids[0]]
        p_fin_borde = modelo.nodos[borde_principal_ids[1]]

        elementos_en_borde = _obtener_elementos_en_borde(p_inicio_borde, p_fin_borde,
borde_principal_ids, modelo)

        if not elementos_en_borde:
            print(f"    -> ADVERTENCIA: No se encontraron elementos en el borde
{borde_principal_ids[0]}-{borde_principal_ids[1]}.")
            continue

        for elem_info in elementos_en_borde:
            id_viga = elem_info['id_viga']
            longitud_viga = elem_info['longitud']
            ni_viga, nj_viga = elem_info['nodos_viga']
            p_inicio_viga, p_fin_viga = modelo.nodos[ni_viga], modelo.nodos[nj_viga]

            print(f"    -> Asignando a Viga ID: {id_viga} (Longitud: {longitud_viga:.2f} m,
Nodos: {ni_viga}-{nj_viga})")

            # Cálculo de carga local
            T, _ = matriz_transformacion_portico_3d(p_inicio_viga, p_fin_viga)
            R = T[:3, :3]
            w_global = np.array([0.0, 0.0, w_lineal])
            w_local = R @ w_global

            cargas_generadas.append({
```

```
                "id_elemento": id_viga, "id_hipotesis": id_hipotesis,
                "datos_carga": ('uniforme', w_local[0], w_local[1], w_local[2], 0.0)
            })

            log_datos['vigas_cargadas'].append({
                'id_viga': id_viga,
                'longitud': longitud_viga,
                'w_local': w_local
            })

    print(f"--- [DEBUG] Fin Distribución Unidireccional para Losa ID: {losa['id']} ---")
    return cargas_generadas, log_datos

#====================================================================
# V. DISTRIBUCIÓN BIDIRECCIONAL DE CARGAS (ÁREAS TRIBUTARIAS)
#====================================================================
def _calcular_geometria_aporte_bidireccional(id_losa_actual, losa, modelo):
    """
    Establece la partición del área de la losa generando trapecios y triángulos
    basados en líneas de rotura.
    """
    # 1. Extracción de coordenadas y verificación del área general
    nodos_losa_ids = losa['nodos']
    coords = {nid: np.array(modelo.nodos[nid]) for nid in nodos_losa_ids}
    p_esquinas = [coords[nid] for nid in nodos_losa_ids]
    area_losa_real = _calcular_area_poligono(np.array([p[:2] for p in p_esquinas]))
    vigas_borde_dict = _encontrar_vigas_de_borde(losa, modelo)

    # 2. Trazado paramétrico de líneas de influencia (rotura) según continuidad
    lineas_aporte = []
    for i in range(4):
        nodo_esquina_id = nodos_losa_ids[i]
        borde_anterior = (nodos_losa_ids[i-1], nodo_esquina_id)
        borde_siguiente = (nodo_esquina_id, nodos_losa_ids[(i + 1) % 4])

        tiene_viga1 = tuple(sorted(borde_anterior)) in vigas_borde_dict
        es_continuo1 = _encontrar_losa_adyacente(borde_anterior, id_losa_actual, modelo) is
not None and tiene_viga1

        tiene_viga2 = tuple(sorted(borde_siguiente)) in vigas_borde_dict
        es_continuo2 = _encontrar_losa_adyacente(borde_siguiente, id_losa_actual, modelo) is
not None and tiene_viga2

        estatus1 = "continuo" if es_continuo1 else "discontinuo"
        estatus2 = "continuo" if es_continuo2 else "discontinuo"

        angulo = 45.0
        if estatus1 != estatus2:
            if estatus1 == "continuo": angulo = 30.0
            else: angulo = 60.0

        vector_borde2 = coords[nodos_losa_ids[(i + 1) % 4]] - coords[nodo_esquina_id]
        norm_borde2 = np.linalg.norm(vector_borde2[:2])
        if norm_borde2 < 1e-9: continue

        vector_aporte_2d = _rotar_vector_2d(vector_borde2[:2] / norm_borde2, angulo)
        vector_aporte_3d = np.array([vector_aporte_2d[0], vector_aporte_2d[1], 0.0])
```

```
        lineas_aporte.append({'punto': coords[nodo_esquina_id], 'vector': vector_aporte_3d})

    if len(lineas_aporte) != 4: return {'poligonos_finales': [], 'puntos_cresta': []}

    # 3. Intersección de líneas para la construcción de polígonos
    p_h1 = _calcular_interseccion_lineas(lineas_aporte[0]['punto'],
lineas_aporte[0]['vector'], lineas_aporte[3]['punto'], lineas_aporte[3]['vector'])
    p_h2 = _calcular_interseccion_lineas(lineas_aporte[1]['punto'],
lineas_aporte[1]['vector'], lineas_aporte[2]['punto'], lineas_aporte[2]['vector'])
    poligonos_h, area_total_h = [], 0
    if p_h1 is not None and p_h2 is not None:
        poligonos_h_sin_orden = [
            [p_esquinas[0], p_esquinas[1], p_h2, p_h1],
            [p_esquinas[1], p_esquinas[2], p_h2],
            [p_esquinas[2], p_esquinas[3], p_h1, p_h2],
            [p_esquinas[3], p_esquinas[0], p_h1]
        ]
        poligonos_h = [_ordenar_vertices_poligono(p) for p in poligonos_h_sin_orden]
        area_total_h = sum(_calcular_area_poligono(np.array([v[:2] for v in p])) for p in
poligonos_h)

    p_v1 = _calcular_interseccion_lineas(lineas_aporte[0]['punto'],
lineas_aporte[0]['vector'], lineas_aporte[1]['punto'], lineas_aporte[1]['vector'])
    p_v2 = _calcular_interseccion_lineas(lineas_aporte[2]['punto'],
lineas_aporte[2]['vector'], lineas_aporte[3]['punto'], lineas_aporte[3]['vector'])
    poligonos_v, area_total_v = [], 0
    if p_v1 is not None and p_v2 is not None:
        poligonos_v_sin_orden = [
            [p_esquinas[0], p_esquinas[1], p_v1],
            [p_esquinas[1], p_esquinas[2], p_v2, p_v1],
            [p_esquinas[2], p_esquinas[3], p_v2],
            [p_esquinas[3], p_esquinas[0], p_v2, p_v1]
        ]
        poligonos_v = [_ordenar_vertices_poligono(p) for p in poligonos_v_sin_orden]
        area_total_v = sum(_calcular_area_poligono(np.array([v[:2] for v in p])) for p in
poligonos_v)

    # 4. Comprobación y selección de la solución geométrica óptima
    tolerancia_area = 1e-4
    es_valido_h = abs(area_total_h - area_losa_real) < tolerancia_area
    es_valido_v = abs(area_total_v - area_losa_real) < tolerancia_area

    poligonos_finales, puntos_cresta, es_cresta_horizontal = [], [], None
    if es_valido_h and not es_valido_v:
        poligonos_finales, puntos_cresta, es_cresta_horizontal = poligonos_h, [p_h1, p_h2],
True
    elif es_valido_v and not es_valido_h:
        poligonos_finales, puntos_cresta, es_cresta_horizontal = poligonos_v, [p_v1, p_v2],
False
    elif es_valido_h and es_valido_v:
        if np.linalg.norm(p_h1 - p_h2) <= np.linalg.norm(p_v1 - p_v2):
            poligonos_finales, puntos_cresta, es_cresta_horizontal = poligonos_h, [p_h1,
p_h2], True
        else:
            poligonos_finales, puntos_cresta, es_cresta_horizontal = poligonos_v, [p_v1,
p_v2], False
    else:
```

```
        p_central = np.mean(p_esquinas, axis=0)
        poligonos_finales = [[p_esquinas[i], p_esquinas[(i+1)%4], p_central] for i in
range(4)]
        puntos_cresta = [p_central, p_central]

    return {'poligonos_finales': poligonos_finales, 'puntos_cresta': puntos_cresta,
            'es_cresta_horizontal': es_cresta_horizontal,
            'p_h1': p_h1, 'p_h2': p_h2, 'p_v1': p_v1, 'p_v2': p_v2}

def _manejar_distribucion_bidireccional(id_losa_actual, losa, wz, id_hipotesis, modelo):
    """
    Asigna cargas perimetrales variables basadas en el análisis de áreas tributarias
    poligonales.
    """
    # 1. Configuración y cálculo inicial de áreas
    log_datos = {
        'id_losa': id_losa_actual,
        'tipo': 'bidireccional',
        'wz': wz,
        'vigas_cargadas': []
    }
    print(f"\n--- [DEBUG] Iniciando Distribución Bidireccional para Losa ID: {id_losa_actual}
---")
    print(f"    Carga Superficial (wz): {wz:.2f} kPa")

    geometria = _calcular_geometria_aporte_bidireccional(id_losa_actual, losa, modelo)
    poligonos_finales = geometria.get('poligonos_finales')
    if not poligonos_finales:
        print(f"    -> ADVERTENCIA: No se pudo calcular la geometría de aporte para la Losa
{id_losa_actual}.")
        return [], log_datos

    cargas_generadas, nodos_losa_ids = [], losa['nodos']
    p_esquinas = {nid: np.array(modelo.nodos[nid]) for nid in nodos_losa_ids}

    bordes_principales_losa = [(nodos_losa_ids[i], nodos_losa_ids[(i + 1) % 4]) for i in
range(4)]
    elementos_de_borde_por_principal = {}
    for borde_principal_ids in bordes_principales_losa:
        p_inicio = p_esquinas[borde_principal_ids[0]]
        p_fin = p_esquinas[borde_principal_ids[1]]
        elementos_de_borde_por_principal[borde_principal_ids] = _obtener_elementos_en_borde(
            p_inicio, p_fin, borde_principal_ids, modelo
        )

    v_global_load = np.array([0.0, 0.0, 1.0]) # Vector unitario de carga global Z

    # 2. Iteración sobre bordes para trazar funciones de carga
    for i, borde_principal_ids in enumerate(bordes_principales_losa):
        n1_borde, n2_borde = borde_principal_ids

        # Parámetros del borde principal
        p_inicio_borde = p_esquinas[n1_borde]
        p_fin_borde = p_esquinas[n2_borde]
        longitud_borde_total = np.linalg.norm(p_fin_borde - p_inicio_borde)

        if longitud_borde_total < 1e-6: continue
```

```
        # Si no hay elementos en este borde, continuar
        if not elementos_de_borde_por_principal[borde_principal_ids]: continue

        # 3. CALCULAR EL PERFIL DE CARGA Q(X) SOBRE LA LONGITUD TOTAL (0 a 1)

        # a) Lógica de sistema local (debe ser la misma para todos los segmentos)
        T_borde, L_borde = matriz_transformacion_portico_3d(p_inicio_borde, p_fin_borde)
        R_borde = T_borde[:3, :3]
        v_local_y = R_borde[1, :]
        v_local_z = R_borde[2, :]

        comp_y = np.dot(v_global_load, v_local_y)
        comp_z = np.dot(v_global_load, v_local_z)
        eje_local = 'y' if abs(comp_y) > abs(comp_z) else 'z'
        magnitud_base = wz * (comp_y if eje_local == 'y' else comp_z)

        # b) Reutilizar la lógica existente para generar la lista_de_puntos TOTAL
        poligono_aporte = poligonos_finales[i]

        # c) Encontrar los puntos de la "cresta" (los que no están en la viga)
        viga_vector_unitario = (p_fin_borde - p_inicio_borde) / longitud_borde_total
        puntos_cresta_locales = []
        for p_vertice in poligono_aporte:
            v_to_p = p_vertice - p_inicio_borde
            dist_proyectada = np.dot(v_to_p, viga_vector_unitario)
            v_proyeccion = dist_proyectada * viga_vector_unitario
            v_perp = v_to_p - v_proyeccion

            if np.linalg.norm(v_perp) > 1e-6:
                puntos_cresta_locales.append({
                    'p_norm': dist_proyectada / longitud_borde_total,
                    'ancho': np.linalg.norm(v_perp)
                })

        # d) Construir la lista de puntos TOTAL (p_norm_total, q_real)
        lista_de_puntos_total = [(0.0, 0.0)]
        puntos_cresta_locales.sort(key=lambda item: item['p_norm'])
        for punto in puntos_cresta_locales:
            p_norm = max(0.0, min(1.0, punto['p_norm']))
            q_real = magnitud_base * punto['ancho']
            lista_de_puntos_total.append((p_norm, q_real))
        lista_de_puntos_total.append((1.0, 0.0))

        # e) Limpiar puntos duplicados (usamos el limpiador local de la función original)
        clean_puntos_total = []
        if lista_de_puntos_total:
            clean_puntos_total.append(lista_de_puntos_total[0])
            for j in range(1, len(lista_de_puntos_total)):
                if abs(lista_de_puntos_total[j][0] - lista_de_puntos_total[j-1][0]) > 1e-6:
                    clean_puntos_total.append(lista_de_puntos_total[j])
                else:
                    clean_puntos_total[-1] = lista_de_puntos_total[j] # Usar el último valor
q si p_norm es igual

        # Usamos la función auxiliar del procesador de cargas para interpolar q(x)
        from procesador_cargas import _interpolar_carga_en_punto
```

```
        # 4. Asignación paramétrica de fragmentos de carga por elemento soportante
        distancia_acumulada = 0.0

        for elem_info in elementos_de_borde_por_principal[borde_principal_ids]:
            id_viga = elem_info['id_viga']
            L_viga = elem_info['longitud']
            ni_viga, nj_viga = elem_info['nodos_viga']

            ni_real, nj_real, _ = modelo.elementos[id_viga]
            es_invertida = False
            # Si el inicio del segmento procesado (ni_viga) no coincide con el inicio real
(ni_real),
            # significa que la viga está definida al revés respecto al recorrido del borde.
            if ni_viga != ni_real:
                es_invertida = True

            print(f"    -> Procesando Viga ID: {id_viga} (Borde {n1_borde}-{n2_borde},
Segmento {ni_viga}-{nj_viga})")

            # a) Normalizar el tramo de la viga sobre la longitud TOTAL [0, 1]
            a_norm_total = distancia_acumulada / longitud_borde_total
            b_norm_total = (distancia_acumulada + L_viga) / longitud_borde_total

            # b) Crear la lista de puntos LOCAL (0.0 a 1.0 local)
            lista_puntos_local = []

            # Añadir el punto de inicio (0.0 local)
            q_inicio = _interpolar_carga_en_punto(clean_puntos_total, a_norm_total)
            lista_puntos_local.append((0.0, q_inicio))

            # Añadir los puntos de quiebre (kinks) que caen dentro del segmento
            for p_norm_total, q_total in clean_puntos_total:
                if a_norm_total < p_norm_total < b_norm_total:
                    p_norm_local = (p_norm_total - a_norm_total) / (b_norm_total -
a_norm_total) # Re-normalizar
                    lista_puntos_local.append((p_norm_local, q_total))

            # Añadir el punto final (1.0 local)
            q_fin = _interpolar_carga_en_punto(clean_puntos_total, b_norm_total)
            lista_puntos_local.append((1.0, q_fin))

            # c) Limpiar puntos locales duplicados y nulos (usando el limpiador local)
            lista_puntos_local.sort(key=lambda item: item[0])
            clean_puntos_local = []
            if lista_puntos_local:
                clean_puntos_local.append(lista_puntos_local[0])
                for j in range(1, len(lista_puntos_local)):
                    if abs(lista_puntos_local[j][0] - lista_puntos_local[j-1][0]) > 1e-6:
                        clean_puntos_local.append(lista_puntos_local[j])
                    else:
                        clean_puntos_local[-1] = lista_puntos_local[j]

            if es_invertida:
                # Invertimos el eje X local: x_nuevo = 1.0 - x_viejo
                # El valor de carga q se mantiene, solo se mueve de posición.
                clean_puntos_local_invertidos = []
```

```
                for p, q in clean_puntos_local:
                    clean_puntos_local_invertidos.append((1.0 - p, q))

                # Al invertir, el 0 se vuelve 1 y viceversa
                clean_puntos_local_invertidos.sort(key=lambda item: item[0])
                clean_puntos_local = clean_puntos_local_invertidos
            # -----------------------------------------

            # d) Generar la carga
            datos_carga_final = ('tramos_locales', eje_local, clean_puntos_local)

            cargas_generadas.append({
                "id_elemento": id_viga,
                "id_hipotesis": id_hipotesis,
                "datos_carga": datos_carga_final
            })

            log_datos['vigas_cargadas'].append({
                'id_viga': id_viga,
                'borde': f"{n1_borde}-{n2_borde}",
                'longitud': L_viga,
                'datos_carga': datos_carga_final
            })

            print(f"       - Carga generada (tramos): Eje='{eje_local}',
Puntos={clean_puntos_local}")

            distancia_acumulada += L_viga

    print(f"--- [DEBUG] Fin Distribución Bidireccional para Losa ID: {id_losa_actual} ---")
    return cargas_generadas, log_datos

#====================================================================
# VI. FUNCIÓN PRINCIPAL
#====================================================================
def traducir_carga_losa_a_cargas_lineales(id_losa, wz, id_hipotesis, modelo):
    """
    Recibe la especificación de carga superficial de una losa y delega la
    evaluación al sub-sistema distributivo correspondiente.
    """
    # 1. Validación de entrada y enrutamiento estructural
    if id_losa not in modelo.losas:
        raise ValueError(f"La losa con ID {id_losa} no existe.")

    losa = modelo.losas[id_losa]
    losa['id'] = id_losa
    vigas_borde = _encontrar_vigas_de_borde(losa, modelo)

    log_datos = {}
    cargas_generadas = []

    if losa['distribucion'] == 'unidireccional':
        cargas_generadas, log_datos = _manejar_distribucion_unidireccional(losa, wz,
id_hipotesis, vigas_borde, modelo)
        return cargas_generadas, log_datos
    elif losa['distribucion'] == 'bidireccional':
```

```
        cargas_generadas, log_datos = _manejar_distribucion_bidireccional(id_losa, losa, wz,
id_hipotesis, modelo)
        return cargas_generadas, log_datos
    else:
        raise ValueError(f"Tipo de distribución '{losa['distribucion']}' no reconocido.")
```

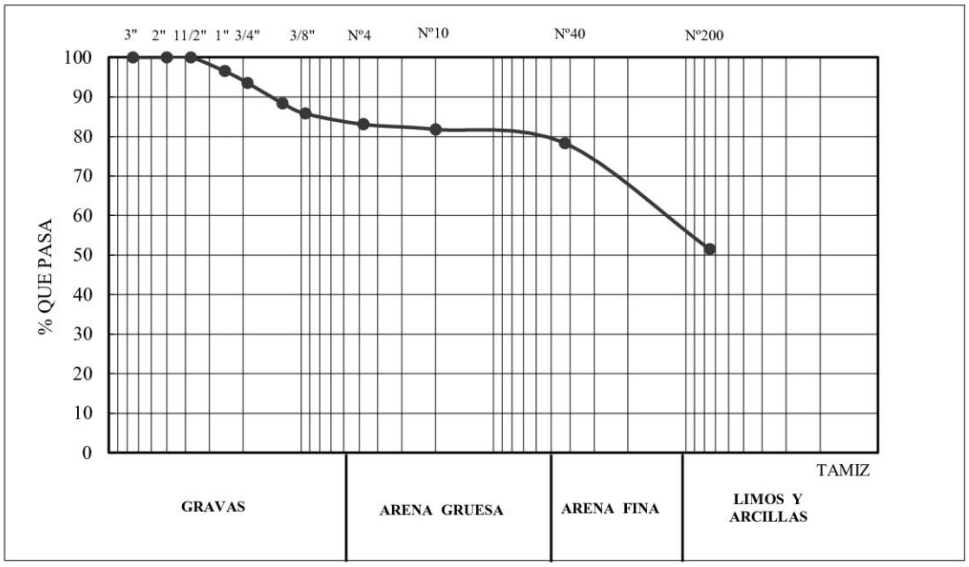
Anexo 5: Estudio de Suelos

UNIVERSIDAD AUTONOMA "JUAN MISAEL SARACHO"
FACULTAD DE CIENCIAS Y TECNOLOGÍA
CARRERA DE INGENIERÍA CIVIL
LABORATORIO DE SUELOS

GRANULOMETRIA

Proyecto:	Desarrollo de una Herramienta de Código Abierto para el Análisis Automatizado de Estructuras Aporticadas en 3D mediante el Método Matricial de Rigidez	**Identificación:**	Pozo 1
Procedencia:	Zona San Luis, Barrio Petrolero	**Fecha:**	10/11/2025

Peso Total (gr.)			**3260**	A.S.T.M.	
Tamices	**Tamaño**	**Peso Ret. (gr)**	**Ret. Acum (gr)**	**% Ret**	**% Que Pasa**
3"	75	0.00	0.00	0.00	**100.00**
2"	50	0.00	0.00	0.00	**100.00**
1 1 /2"	37.50	0.00	0.00	0.00	**100.00**
1"	25.00	113.40	113.40	3.48	**96.52**
3/4"	19.00	97.80	211.20	6.48	**93.52**
1/2"	12.50	167.20	378.40	11.61	**88.39**
3/8"	9.50	85.20	463.60	14.22	**85.78**
Nº4	4.75	88.80	552.40	16.94	**83.06**
Nº10	2.00	41.00	593.40	18.20	**81.80**
Nº40	0.425	116.00	709.40	21.76	**78.24**
Nº200	0.075	872.60	1582.00	48.53	**51.47**



UNIVERSIDAD AUTONOMA "JUAN MISAEL SARACHO"

FACULTAD DE CIENCIAS Y TECNOLOGIA

CARRERA DE INGENIERIA CIVIL

LABORATORIO DE SUELOS

LÍMITES DE ATTERBERG

Proyecto:	Desarrollo de una Herramienta de Código Abierto para el Análisis Automatizado de Estructuras Aporticadas en 3D mediante el Método Matricial de Rigidez	**Identificacion:**	Pozo 1
Procedencia:	Zona San Luis, Barrio Petrolero	**Fecha:**	10/11/2025

Capsula Nº	1	2	3	4
Nº de golpes				
Suelo Húmedo + Cápsula				
Suelo Seco + Cápsula				
Peso del agua				
Peso de la Cápsula				
Peso Suelo seco				
Porcentaje de Humedad				

N.P.

Determinación de Límite Plástico

Cápsula	1	2	3
Peso de suelo húmedo + Cápsula			
Peso de suelo seco + Cápsula			
Peso de cápsula			
Peso de suelo seco			
Peso del agua			
Contenido de humedad			

Límite Líquido (LL)
0
Límite Plástico (LP)
0
Indice de plasticidad (IP)
0
Indice de Grupo (IG)
3

UNIVERSIDAD AUTONOMA "JUAN MISAEL SARACHO"
FACULTAD DE CIENCIAS Y TECNOLOGÍA
CARRERA DE INGENIERÍA CIVIL
LABORATORIO DE SUELOS

HUMEDAD NATURAL Y CLASIFICACIÓN

Proyecto:	Desarrollo de una Herramienta de Código Abierto para el Análisis Automatizado de Estructuras Aporticadas en 3D mediante el Método Matricial de Rigidez	**Identificación:**	Pozo 1
Procedencia:	Zona San Luis, Barrio Petrolero	**Fecha:**	10/11/2025

HUMEDAD NATURAL			
Cápsula	**1**	**2**	**3**
Peso de suelo húmedo + Cápsula	85.04	76.1	92.66
Peso de suelo seco + Cápsula	80.8	72.4	88.2
Peso de cápsula	15.03	14.33	18.9
Peso de suelo seco	65.77	58.07	69.3
Peso del agua	4.24	3.7	4.46
Contenido de humedad	6.45	6.37	6.44
PROMEDIO	**6.42**		

CLASIFICACIÓN DEL SUELO	**SUCS:** **ML - OL** **AASHTO:** **$A\text{-}4_{(3)}$**
DESCRIPCIÓN	Limos inorgánicos, polvo de roca, limo arenosos o arcillosos ligeramente plásticos.

UNIVERSIDAD AUTONOMA "JUAN MISAEL SARACHO"
FACULTAD DE CIENCIAS Y TECNOLOGÍA
PROGRAMA DE INGENIERÍA CIVIL
LABORATORIO DE SUELOS

ENSAYO DE CARGA DIRECTA (S.P.T.)

Proyecto: Desarrollo de una Herramienta de Código Abierto para el Análisis Automatizado de Estructuras Aporticadas en 3D mediante el Método Matricial de Rigidez

Identificación: Pozo 1

Procedencia: Zona San Luis, Barrio Petrolero

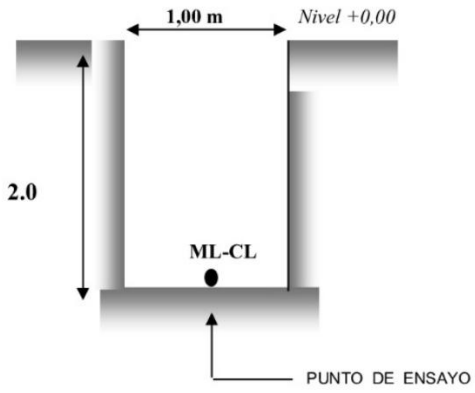
Fecha: 10/11/2025

Datos Standarizados del Equipo

Altura de penetracion:	30 cm
Peso del Martillo:	65 kg
Altura de caida:	75 cm
% Humedad:	6.4

Pozo Nº	Profundidad (m)	Nº Golpes	Resist. Adm. Nat.(Kg/cm²)	Resist. Adm. Seca (Kg/cm²)	Clasificación del Suelo
1	2.00	11	**1.40**	**1.49**	**SUCS:** ML-CL **AASHTO:** A-4(3)

Descripción Gráfica



Características del Suelo

Arcilla limosa arenosa de baja a mediana compresibilidad.

…… ……

Univ. Diego Oliver Vargas Moya — Ing. José Ricardo Arce Avendaño

LABORATORISTA — RESP. LABORATORIO DE SUELO

Nota. El laboratorio de suelos certifica la realización de la presente práctica, mas no se hace responsable de los resultados obtenidos, es enteramente responsabilidad del estudiante investigador

Anexo 6: Determinación de cargas en el edificio Concordia

Calculo de Muros de ladrillo hueco

Altura de muro

$h_{entrepiso} := 3.42\ m$ $h_{viga} := 0.3\ m$

$h_{muro} := h_{entrepiso} - h_{viga} = 3.12\ m$

Determinación de la carga distribuida debido a muros de cierre sobre vigas

$Pe := 10\ \frac{kN}{m^3}$ $A := h_{muro} \cdot 18\ cm = 0.562\ m^2$

$q_{ladrillo} := Pe \cdot A = 5.616\ \frac{kN}{m}$

Peso del mortero

Asumimos revoque de 1 cm en cada cara del ladrillo

$qsuperficial_{yeso} := 2 \cdot 0.13 \cdot \frac{kN}{m^2} = 0.26\ \frac{kN}{m^2}$

$q_{yeso} := qsuperficial_{yeso} \cdot h_{muro} = 0.811\ \frac{kN}{m}$

Carga de Muros Perimetrales

$q_{muro} := q_{ladrillo} + q_{yeso} = 6.427\ \frac{kN}{m}$ Se asume: $q_{muro} := \text{Ceil}\left(q_{muro}, 0.1\ \frac{kN}{m}\right)$

$q_{muro} = 6.5\ \frac{kN}{m}$

Carga de Muros en Ventanales

$q_{muro\_ventanal} := 3\ \frac{kN}{m}$

Carga de Muros en Azoteas

$h_{muro\_azotea} := 1\ m$ $A_{muro\_azotea} := h_{muro\_azotea} \cdot 18\ cm = 0.18\ m^2$

$q_{yeso\_azotea} := qsuperficial_{yeso} \cdot h_{muro\_azotea} = 0.26\ \frac{kN}{m}$

$q_{ladrillo\_azotea} := Pe \cdot A_{muro\_azotea} = 1.8\ \frac{kN}{m}$

$$q_{muro\_azotea} := q_{yeso\_azotea} + q_{ladrillo\_azotea} = 2.06\ \frac{kN}{m}$$

Se asume: $q_{muro\_azotea} := \text{Ceil}\left(q_{muro\_azotea}, 0.1\ \frac{kN}{m}\right)$ $\quad q_{muro\_azotea} = 2.1\ \frac{kN}{m}$

Carga de Muros de la zona de Tanques

$$h_{muro\_tanque} := 3.71\ m \qquad A_{muro\_tanque} := h_{muro\_tanque} \cdot 18\ cm = 0.668\ m^2$$

$$q_{yeso\_tanque} := qsuperficial_{yeso} \cdot h_{muro\_tanque} = 0.965\ \frac{kN}{m}$$

$$q_{ladrillo\_tanque} := Pe \cdot A_{muro\_tanque} = 6.678\ \frac{kN}{m}$$

$$q_{muro\_tanque} := q_{yeso\_tanque} + q_{ladrillo\_tanque} = 7.643\ \frac{kN}{m}$$

Se asume: $q_{muro\_tanque} := \text{Ceil}\left(q_{muro\_tanque}, 0.1\ \frac{kN}{m}\right)$ $\quad q_{muro\_tanque} = 7.7\ \frac{kN}{m}$

Carga de Muros Interiores Sobre Vigas

$$h_{muro\_INT} := 3.12\ m \qquad A_{muro\_INT} := h_{muro\_INT} \cdot 12\ cm = 0.374\ m^2$$

$$q_{yeso\_INT} := qsuperficial_{yeso} \cdot h_{muro\_INT} = 0.811\ \frac{kN}{m}$$

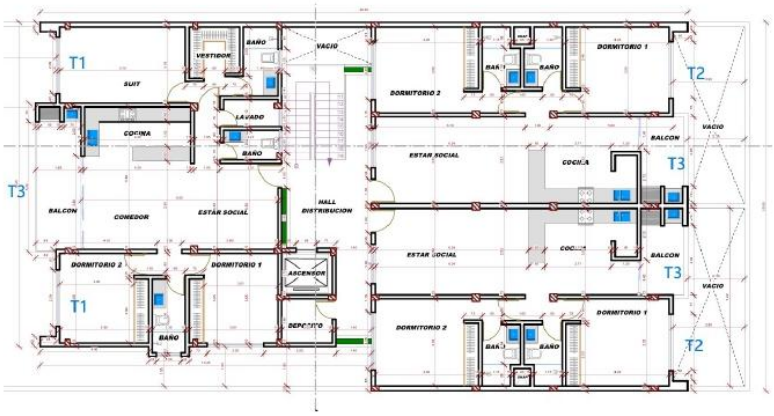
$$q_{ladrillo\_INT} := Pe \cdot A_{muro\_INT} = 3.744\ \frac{kN}{m}$$

$$q_{muro\_INT} := q_{yeso\_INT} + q_{ladrillo\_INT} = 4.555\ \frac{kN}{m}$$

Se asume: $q_{muro\_INT} := \text{Ceil}\left(q_{muro\_INT}, 0.1\ \frac{kN}{m}\right)$

$$q_{muro\_INT} = 4.6\ \frac{kN}{m}$$

Cargas de balcones en Vortex



$q_{muro\_balcón\_cype} := 3\ \frac{kN}{m}$ $q_{L\_balcón\_T1\_T2} := 2\ \frac{kN}{m^2}$ $q_{D\_balcón} := 1.6\ \frac{kN}{m^2}$

$q_{L\_balcón\_T3} := 3\ \frac{kN}{m^2}$

Balcón tipo 1 $q_{DB1} := 3.7\ \frac{kN}{m}$

$L_{b1} := 0.40\ m$ $q_{LB1} := q_{L\_balcón\_T1\_T2} \cdot L_{b1} = 0.8\ \frac{kN}{m}$

$q_{D'B1} := q_{D\_balcón} \cdot L_{b1} = 0.64\ \frac{kN}{m}$

$q_{DB1} := q_{D'B1} + q_{muro\_balcón\_cype} = 3.64\ \frac{kN}{m}$

Balcón tipo 2

$L_{b2} := 0.70\ m$ $q_{LB2} := q_{L\_balcón\_T1\_T2} \cdot L_{b2} = 1.4\ \frac{kN}{m}$

$q_{D'B2} := q_{D\_balcón} \cdot L_{b2} = 1.12\ \frac{kN}{m}$

$q_{DB2} := q_{D'B2} + q_{muro\_balcón\_cype} = 4.12\ \frac{kN}{m}$

Balcón tipo 3 $q_{DB2} := 4.2\ \frac{kN}{m}$

$L_{b3} := 1.25\ m$ $q_{LB3} := q_{L\_balcón\_T3} \cdot L_{b3} = 3.75\ \frac{kN}{m}$

$q_{DB3} := q_{D\_balcón} \cdot L_{b3} = 2\ \frac{kN}{m}$

Cargas permanentes sobre la Losa

$$Q_{cielofalso} := 0.2\ \frac{kN}{m^2}$$

$$Pe_{ceramico} := 18\ \frac{kN}{m^3} \qquad espesor := 8\ mm$$

$$Q_{ceramico} := Pe_{ceramico} \cdot espesor = 0.144\ \frac{kN}{m^2}$$

Sobrecarga de Tabiquería

4.2.2 Sobrecarga de tabiquería movil

Los tabiques móviles y a futuro se tomarán en cuenta como carga equivalente uniformemente repartida por metro cuadrado, igual al 33% del peso por metro lineal de tabique terminado, como un mínimo de 1,20 kN/m^2, salvo que la sobrecarga de uso correspondiente sea igual o superior a 4,00 kN/m^2, en cuyo caso no se requiere considerar el peso de los tabiques.

$$Q_{tabiquería} := 1.2\ \frac{kN}{m^2}$$

Resumen de cargas permanentes

Superficiales

$$Q_{cielofalso} = 0.2\ \frac{kN}{m^2}$$

$$Q_{ceramico} = 0.144\ \frac{kN}{m^2}$$

$$Q_{tabiquería} = 1.2\ \frac{kN}{m^2}$$

Se asume: $Q_D := Q_{cielofalso} + Q_{ceramico} + Q_{tabiquería}$

$$Q_D := \text{Ceil}\left(Q_D, 0.1\ \frac{kN}{m^2}\right) = 1.6\ \frac{kN}{m^2} \qquad Q_D = 1.6\ \frac{kN}{m^2}$$

AGUA

Los tanques ocupan un área de 17 metros

$Ftotal := 90\ kN$ $A_{tanque} := 17\ m^2$ $QF := \frac{Ftotal}{A_{tanque}} = 5.294\ \frac{kN}{m^2}$

$QF := \text{Ceil}\left(QF, 0.1\ \frac{kN}{m^2}\right)$ $QF = 5.3\ \frac{kN}{m^2}$

$A_{tanque\_vortex} := 8.25\ m^2$ $QF_v := \frac{Ftotal}{A_{tanque\_vortex}} = 10.909\ \frac{kN}{m^2}$

$QF_v := \text{Ceil}\left(QF_v, 0.1\ \frac{kN}{m^2}\right)$ $QF_v = 11\ \frac{kN}{m^2}$

Sobrecargas de Uso

$Departamentos := 2\ \frac{kN}{m^2}$

$Balcones := 3\ \frac{kN}{m^2}$

$Azotea\_Accesible := 4\ \frac{kN}{m^2}$

$Azotea\_Inaccesible := 1\ \frac{kN}{m^2}$ para el techo de cuarto de máquinas, azotea inaccesible

$Depósitos := 6\ \frac{kN}{m^2}$

Depósitos (serán diseñados para cargas más pesadas si el almecenamiento previsto lo requiere)	
- Liviano	6,0
- Pesado (Véase 4.13)	12,0

Sobrecarga de granizo

Recomendado por Norma PNB para Tarija

$Q_{granizo} := 110 \ \frac{kgf}{m^2}$

$Q_{granizo} = 1.079 \ \frac{kN}{m^2}$

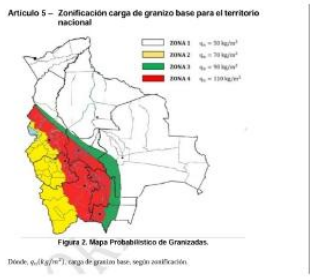


Figura 2. Mapa Probabilístico de Granizadas.

$PE_{granizo} := 9 \ \frac{kN}{m^3}$ asumiendo un espesor de granizo de $e_{gra} := 20 \ cm$

$Q_{granizo2} := PE_{granizo} \cdot e_{gra} = 1.8 \ \frac{kN}{m^2}$

Carga de ascensor

a) Cuando el equipo propulsor se encuentra emplazado sobre la losa:

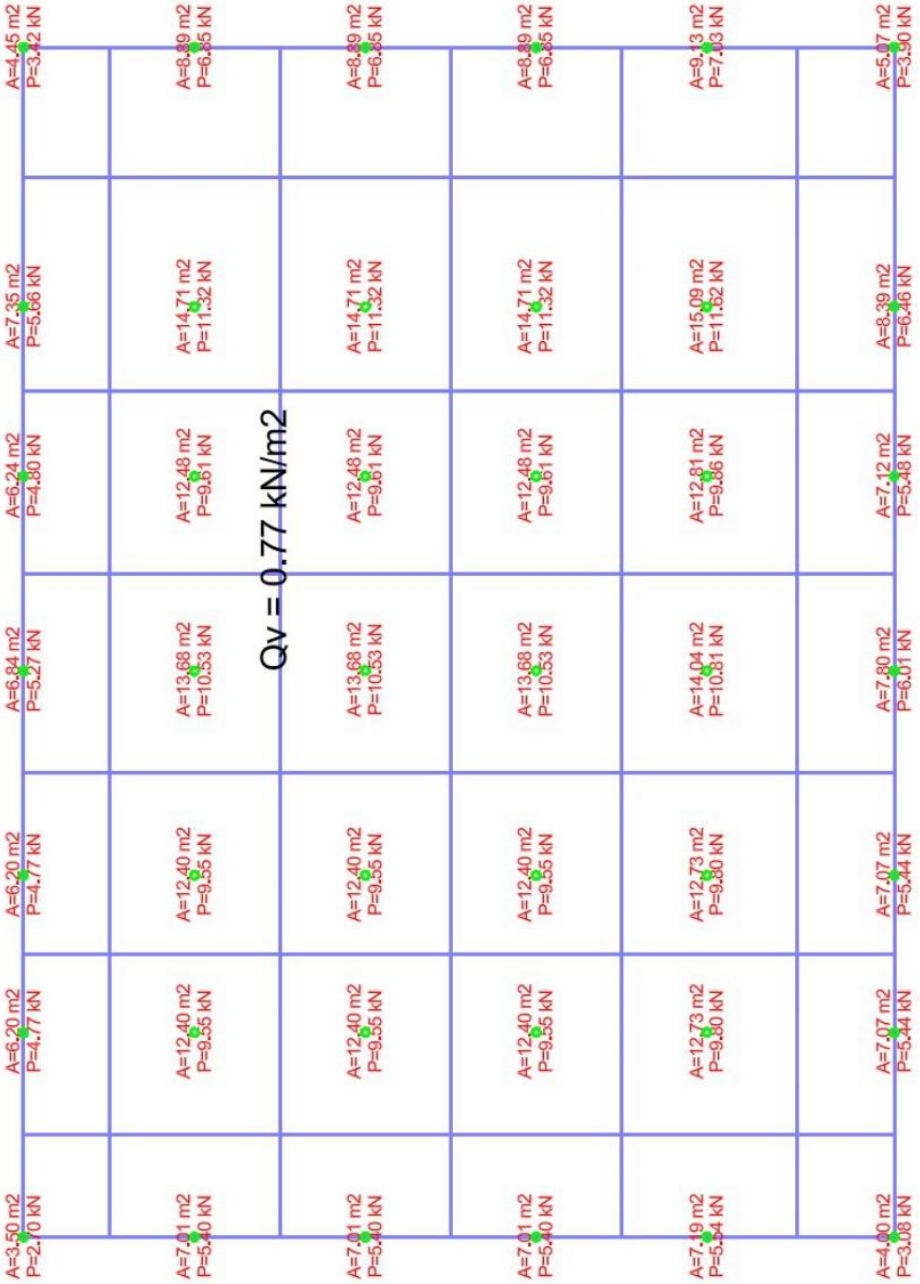
Nº	Área del hueco en m²	Sobrecargas kPa
1	$A_h < 1{,}00$	40,0
2	$1{,}00 < A_h \leq 1{,}50$	35,0
3	$1{,}50 < A_h$	25,0

$Q_{ascensor} := 25 \ \frac{kN}{m^2}$ $IM := 0.33$

$Q_{ascensor} := Q_{ascensor} + Q_{ascensor} \cdot IM = 33.25 \ \frac{kN}{m^2}$

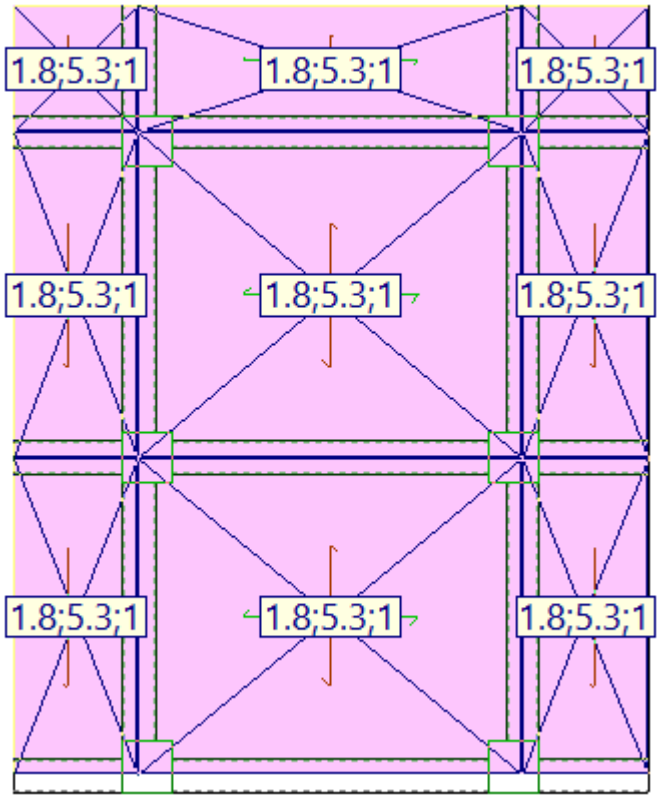
Sobrecarga viento

$$Q := 0.77\ \frac{kN}{m^2}$$

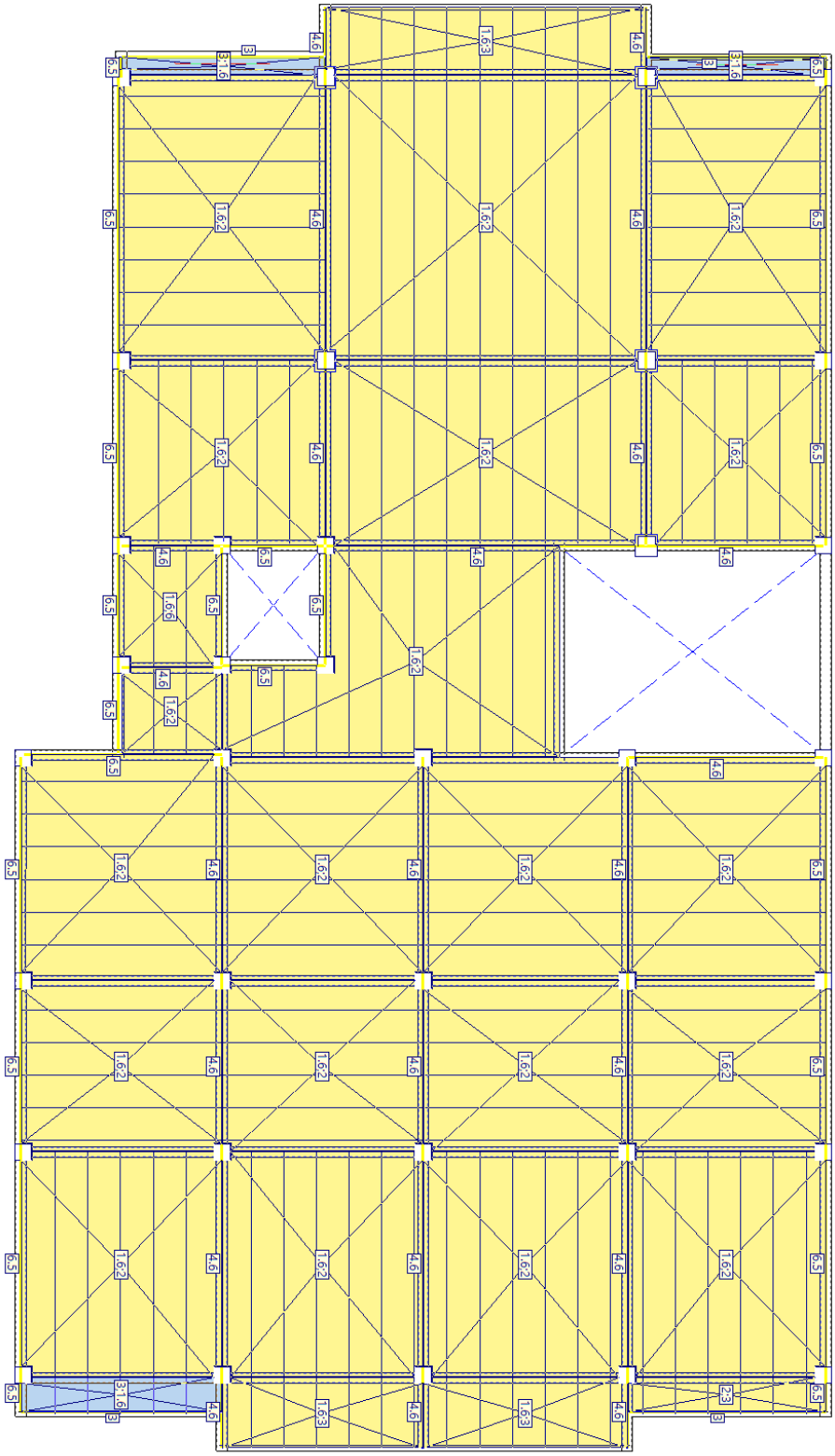


Anexo 7: Cargas Aplicadas en el Edificio Concordia

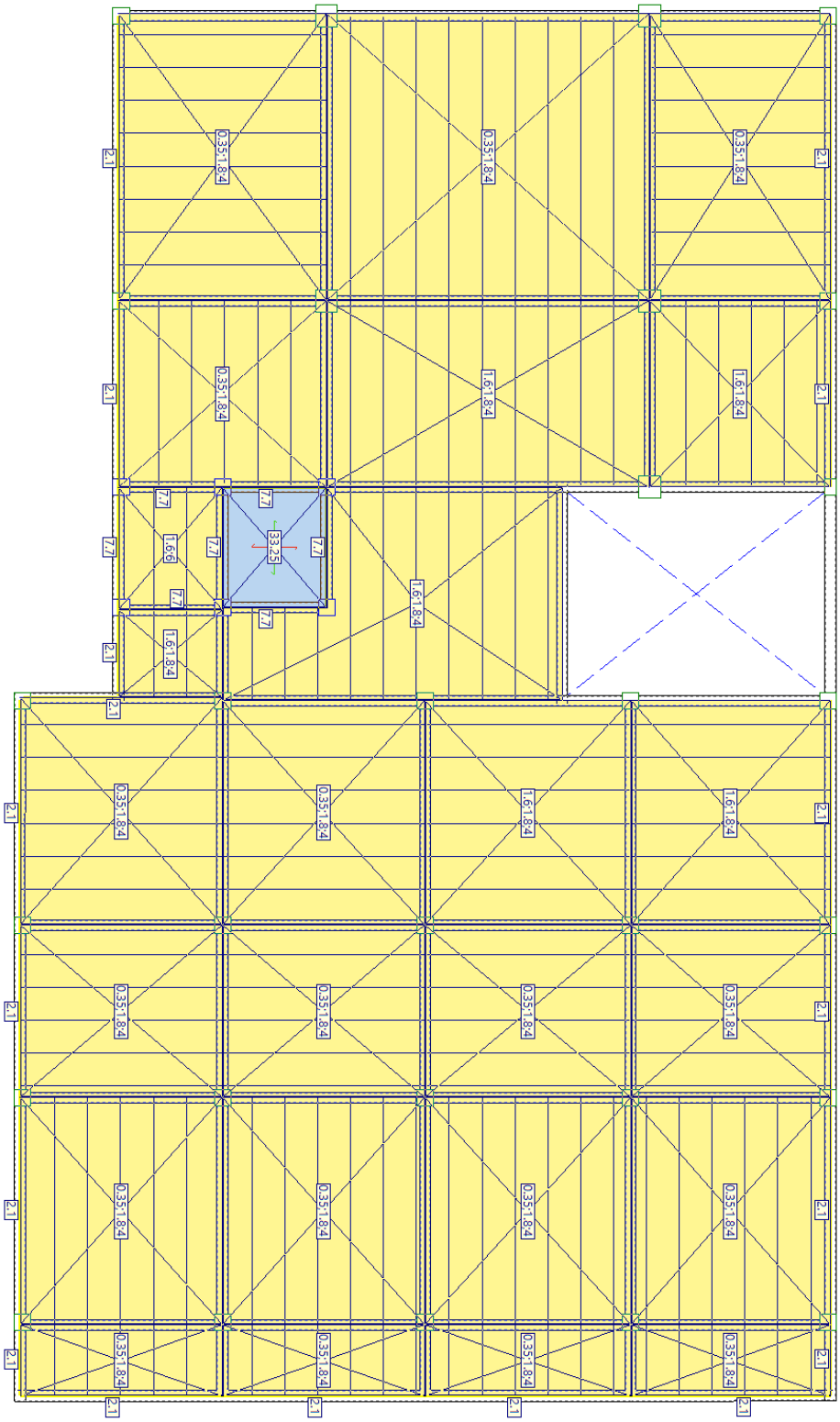
Tanques de Agua:



Pisos 1-4



Azotea



Anexo 8: Manual de Usuario de VORTEX 3D

VORTEX 3D

MANUAL DE USUARIO

Contenido

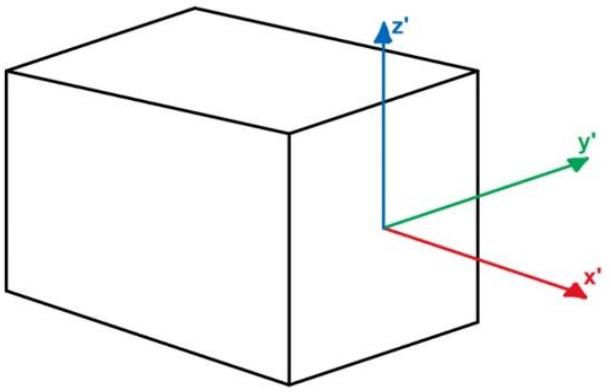
1. INTRODUCCIÓN

VOTX3D es una plataforma integral para el análisis matricial y diseño de estructuras de edificaciones. Diseñado para auditoría de software comercial y realizar el análisis y diseño de estructuras de forma nativa.

Para optimizar la productividad del ingeniero, permite dos flujos de trabajo claros: modelado manual paso a paso o importación automatizada desde planos CAD.

El núcleo del programa resuelve pórticos espaciales (6 grados de libertad por nodo) e integra un distribuidor de cargas de losa que simula la transferencia de fuerzas gravitacionales sin añadir rigidez a la matriz global.

El sistema de coordenadas local sigue la regla de la mano derecha.



2. FLUJO DE TRABAJO MANUAL (NATIVO EN VORTEX)

Este capítulo describe la secuencia lógica para crear un modelo desde cero dentro de la interfaz del programa.

Definición de Nodos

El esqueleto de la estructura. Se ingresan coordenadas (X, Y, Z). Los nodos definen los puntos de inicio y fin de cualquier barra. Los nodos serán guardados con identificadores que serán utilizados posteriormente para definir elementos y cargas puntuales.

	ID	X	Y	Z
1	1	0.000	0.000	3.000
2	2	0.000	3.000	3.000
3	3	0.000	6.000	3.000
4	4	2.500	0.000	3.000
5	5	5.000	0.000	3.000
6	6	2.500	3.000	3.000
7	7	2.500	6.000	3.000
8	8	5.000	3.000	3.000
9	9	5.000	6.000	3.000
10	10	0.000	6.000	0.000

Materiales y Secciones

Los materiales y las secciones se guardan como uno. Antes de crear elementos, defina las propiedades mecánicas según la sección sea de tipo rectangular o de tipo general.

a. Secciones Rectangulares:
    a. Módulo de Young en [MPa]
    b. Coeficiente de Poisson
    c. Base en [m]
    d. Altura en [m]
    e. Peso Específico en [kN/m^3]

b. Secciones Generales:
    a. Módulo de Young en [MPa]
    b. Módulo de Corte en [MPa]
    c. Área en [m^2]
    d. Inercia Torsional en [m^4]
    e. Inercia en Y local
    f. Coeficiente de Poisson
    g. Base en [m]
    h. Altura en [m]
    i. Peso Específico en [kN/m^3]

	ID	Tipo	Desc.	E	v	b	h	G	A	J	Iy	Iz	Ay	Az	PE (kN/m³)
1	1	Rectangular	VIGA	2.10e+04	0.200	0.3000	0.2000								24.00
2	2	Rectangular	COL	2.10e+04	0.200	0.2000	0.2000								24.00
3	3	General	GEN	2.10e+04				8.75e+03	0.080000	0.001000	0.000260	0.001060	0.060000	0.060000	24.00

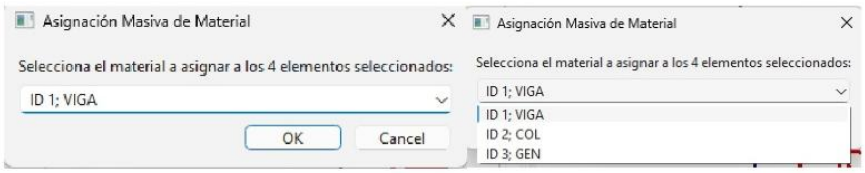
Elementos

Se conectan nodos para crear vigas y columnas, adicionalmente, debe especificarse el material/sección

	ID	Nodo I	Nodo J	Material
1	1	1	2	1
2	2	2	3	1
3	3	1	4	1
4	4	4	5	1
5	5	4	6	1
6	6	6	7	1
7	7	5	8	1
8	8	8	9	1
9	9	2	6	1

Una vez definidos todos los elementos, es posible añadir un nuevo material/sección y hacer una modificación de materiales en lote desde la pestaña de definición de elementos con el botón “Asignar material a selección”. Para usarlo primero deben seleccionarse los elementos que se

desean modificar y luego se pulsa el botón, se abrirá una ventana emergente en la cual se elegirá el material que se desea asignar a la selección:



Apoyos

En este apartado se seleccionan los nodos que representarán apoyos y sus restricciones correspondientes. Es posible añadir un mismo tipo de apoyo a varios nodos al mismo tiempo por medio del uso de comas. No es necesario poner espacios después de las comas.

(Ej. 5,6,7,8,11,12)

	Nodo	Tx	Ty	Tz	Rx	Ry	Rz
1	10	☑	☑	☑	☑	☑	☑
2	11	☑	☑	☑	☑	☑	☑
3	12	☑	☑	☑	☑	☑	☑
4	13	☑	☑	☑	☑	☑	☑
5	14	☑	☑	☑	☑	☑	☑
6	15	☑	☑	☑	☑	☑	☑

Hipótesis de Carga

Antes de poder asignar cargas, deben definirse las hipótesis de carga que serán consideradas en el análisis, para definirlas se les debe asignar un nombre y posteriormente especificar el tipo de carga.

	ID	Nombre	Tipo
1	1	MUERTA	D
2	2	SCdU	L
3	3	VIENTO	W
4	4	SOLO_MUROS	D
5	5	CASO_BIBLIOTECA	L

Definición de Losas

Para definir una losa, primero deben especificarse los 4 puntos que definirán la Losa, las losas deben ser coplanares. Luego se especifica el tipo de distribución, pudiendo ser Bidireccional o Unidireccional, en caso de ser unidirecional, también debe especificarse el sentido en el que se realizará la distribución de cargas.

También debe definirse el espesor y el peso específico de las losas en caso de que se desee hacer un análisis considerando peso propio.

	ID	Nodos	Distribución	Eje Uni.	Espesor (m)	PE (kN/m³)
1	1	2, 6, 7, 3	Unidireccional	Global X	0.200	24.00
2	2	6, 8, 9, 7	Unidireccional	Global X	0.200	24.00
3	3	4, 5, 8, 6	Unidireccional	Global X	0.200	24.00
4	4	1, 4, 6, 2	Unidireccional	Global X	0.200	24.00
5	5	20, 24, 25, 21	Unidireccional	Global X	0.200	24.00
6	6	24, 26, 27, 25	Unidireccional	Global X	0.200	24.00
7	7	22, 23, 26, 24	Unidireccional	Global X	0.200	24.00
8	8	19, 22, 24, 20	Unidireccional	Global X	0.200	24.00

Asignación de Cargas

La asignación de cargas puede ser de tipo puntual, distribuida en elementos y distribuida en superficies. Según el tipo de carga que se seleccione, se deberá especificar los nodos, elementos o losas que recibirán las cargas y las magnitudes de las mismas.

	ID Carga	Hipótesis	Ubicación	Valores
6	14	VIENTO	Nodo 19	Fx: 2.00
7	1	MUERTA	Elem 1	wz: -1.85
8	2	MUERTA	Elem 2	wz: -1.85
9	3	MUERTA	Elem 3	wz: -1.85
10	4	MUERTA	Elem 4	wz: -1.85
11	5	MUERTA	Elem 7	wz: -1.85
12	6	MUERTA	Elem 8	wz: -1.85
13	7	MUERTA	Elem 11	wz: -1.85
14	8	MUERTA	Elem 12	wz: -1.85
15	1	MUERTA	Losa 1	wz: -2.00 kN/m²

Combinaciones de Carga

VORTEX 3D maneja de forma predeterminada las combinaciones de carga establecidas por la norma boliviana NB 1225002, en caso de que el usuario requiera de combinaciones de carga adicionales, podrá crearlos de forma personalizada, para esto se asignará un nombre y

posteriormente se definirán los coeficientes multiplicadores de cada uno de los tipos de carga que entren en dicha combinación. Es posible eliminar añadir, actualizar y eliminar tipos de carga.

	Nombre	Factores	Tipo
1	Cálculo Simple	{'D': 1.0}	Norma
2	1.4D	{'D': 1.4}	Norma
3	1.2D + 1.6L + 0.5Lr	{'D': 1.2, 'L': 1.6, 'Lr': 0.5}	Norma
4	1.2D + 1.6Lr + 1.0L	{'D': 1.2, 'Lr': 1.6, 'L': 1.0}	Norma
5	1.2D + 1.0W + 1.0L + 0.5Lr	{'D': 1.2, 'W': 1.0, 'L': 1.0, 'Lr': 0.5}	Norma
6	0.9D + 1.0W	{'D': 0.9, 'W': 1.0}	Norma
7	PERSONALIZADO 1	{'D': 1.0, 'L': 1.0, 'Lr': 1.0, 'W': 1.0, 'E': 1.0}	Usuario

La forma en la que las combinaciones de carga consideran las hipótesis es mutuamente excluyente a excepción de cuando se ejecuta el cálculo con peso propio incluido, en cuyo caso este si se sumará a cada una de las hipótesis de tipo D que siguen siendo mutuamente excluyentes.

Cálculo

Al ejecutar el análisis, se puede seleccionar:

- Considerar Peso Propio: Al seleccionar esta opción, el programa calcula automáticamente el peso de barras y losas considerando el peso específico y la geometría de cada una de estas.
- Deformación por Cortante: Activar/Desactivar la teoría de TImoshenko (relevante para elementos peraltados y de uso estandarizado en software comercial).

3. FLUJO DE TRABAJO AUTOMATIZADO (DXF)

VORTEX 3D ofrece al usuario la posibilidad de introducir todo el modelo estructural desde dxf. Para esto, debe modelarse la estructura desde un software CAD comercial, guardarse el archivo como ".dxf" e importarse desde VORTEX 3D, la forma en la que se modela la estructura para ser reconocida por el programa es la siguiente:

SINTAXIS DE CAPAS ADMITIDA:

ELEMENTOS (Line): EL_NOMBRE_h_b
Ej: *EL_Viga20x40_40_20* (h y b en cm)

LOSAS (3DFace): LO_DIST_ESP_PE
Ej: *LO_BI_20_24* (Bidireccional, 20cm, 24kN/m^3)
Ej: *LO_UX_15_25* (Uni X, 15cm, 25kN/m^3)

APOYOS (Circle): AP_RESTRICCIONES
Ej: *AP_111000* (Restringe X, Y, Z, Libera rotaciones)

CARGAS UNIFORMES EN ELEMENTO (Line):
CU_TIPO_HIP_EJE_VALOR
Ej: *CU_D_Muro_Z_-15* (Carga Muerta 'Muro', Eje Z, -15 kN/m)
Ej: *CU_W_Viento_X_5* (Carga Viento, Eje X, 5 kN/m)

CARGAS SUP. (3DFace): CS_TIPO_HIP_VALOR
Ej: *CS_L_Uso_-2* (Carga Viva 'Uso' de 2 kN/m^2)

CARGAS PUNT. (Point):
CP_TIPO_HIP_ACCION_VALOR
Ej: *CP_L_Eq_FX_100* (Carga Viva 'Eq', Fuerza X, 100 kN)

Elementos Lineales

Usar entidades LINE (líneas, NO polilíneas). El nombre de la capa que contenga las líneas debe usar la siguiente sintáxis:

EL_NOMBRE_b_h

- **EL:** Identificador de "Elementos"
- **NOMBRE:** Descripción del material/sección.
- **b_h**: Dimensiones en cm (base x altura).

Ejemplos: "EL_VIG_20_30", "EL_COL_20_20"

Ya no es necesario establecer nodos, el programa reconoce automáticamente el inicio y final de las líneas para establecer los nodos.

Losas

Para definir Losas, deben utilizarse entidades 3DFACE con solamente 4 puntos. Actualmente, el programa únicamente acepta losas con 4 lados ortogonales y coplanares.

La sintaxis de la capa que define estas losas es la siguiente:

LO_DISTRIBUCIÓN_ESPESOR_PESOESPECIFICO

- **LO:** Identificador que permite que el programa reconozca los objetos dentro de esta capa como losas.
- **DISTRIBUCIÓN:** El "valor" de DISTRIBUCIÓN puede ser
  - **BI:** Distribución Bidireccional
  - **UX:** Distribución Unidireccional en el Eje Global X.
  - **UY:** Distribución Unidireccional en el Eje Global Y.
- **ESPESOR:** Espesor de las losas que pertenezcan a dicha capa en cm
- **PESO ESPECÍFICO:** Debe ser introducido en kN/m3

Ejemplos: "LO_BI_15_24", "LO_UY_20_17.58"

Apoyos

Se reconocen entidades tipo CIRCLE, cuyo centro coincida con algún nodo.

AP_DxDyDzRxRyRz

- **AP:** Identificador de elementos de apoyo.
- **Restricciones:** En el orden en el que se indica, admitiendo "1" como restringido y "0" como libre

Ejemplos: "AP_111111", "AP_111000"

Cargas Uniformes

Al igual que con elementos lineales, la entidad CAD admitida es LINE (línea), esta debe ser dibujada de nodo a nodo encima de elementos previamente definidos.

CU_TIPO_HIP_EJE_VALOR

- **CU:** Identificador de cargas uniformes.
- **TIPO:** Tipo de carga especificado por NB1225002 (D, L, S, W, etc)
- **HIP:** Nombre del caso de carga o hipótesis que pertenece a cada tipo, esta es una descripción de dicha hipótesis.
- **EJE:** Define el eje en el que se aplicará la carga uniformemente distribuida.
- **VALOR:** Valor de la carga, este puede ser positivo o negativo según requerimiento.

Cargas Superficiales

También reconocidas como entidades 3DFACE, estas deben ser trazadas sobre LOSAS previamente definidas, deben coincidir en los 4 puntos de estas.

CS_TIPO_HIP_VALOR

- **CS:** Identificador de cargas superficiales.
- **TIPO:** Tipo de carga.
- **HIP:** Hipótesis de la carga.
- **VALOR:** Valor de la carga, misma que solamente podrá ser vertical (EJE Z), pero puede ser tanto positiva como negativa.

Cargas Puntuales

Estas serán entidades tipo POINT o PUNTO, deben ser aplicadas directamente sobre un nodo (inicio o fin de un elemento).

CP_TIPO_HIP_ACCIÓN_VALOR

- **CP:** Identificador de cargas puntuales.
- **TIPO:** Tipo de carga.
- **HIP:** Hipótesis de carga.
- **ACCIÓN:** Sentido en el que actuará la carga puntual, pudiendo ser FX, FY, FZ, MX, MY, MZ.
- **VALOR:** Valor de la carga, pudiendo ser positiva o negativa. Los momentos siguen la regla de la mano derecha, es decir, antihorario = positivo y horario = negativo.

4. Resultados

Los resultados presentados por el programa son desplazamientos, reacciones y fuerzas internas, a continuación, se presenta la configuración previa referente a combinaciones, hipótesis y elementos de los cuales se observará cada uno de estos resultados.

Además, también se presentará la configuración y generación de reportes de cálculo.

Combinación

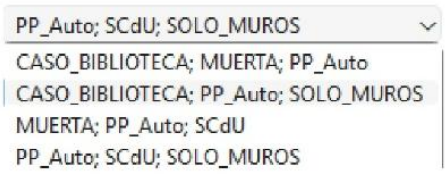
Lo primero que debe seleccionarse es la combinación de cargas de la cual se observarán los resultados.

Ejemplo:

Hipótesis

Nombrada en la pestaña de resultados como "Caso de análisis", indica las distintas combinaciones de hipótesis existentes, teniendo en cuenta que aquellas hipótesis del mismo tipo de carga son mutuamente excluyentes.

Ejemplo:



Elementos

Tanto "combinación" como "hipótesis", seleccionan las condiciones preliminares de la estructura, sin embargo, si lo que se desea es observar las fuerzas internas nodales de cada uno de los elementos de forma preliminar, este debe seleccionarse manualmente en la caja de selección correspondiente.

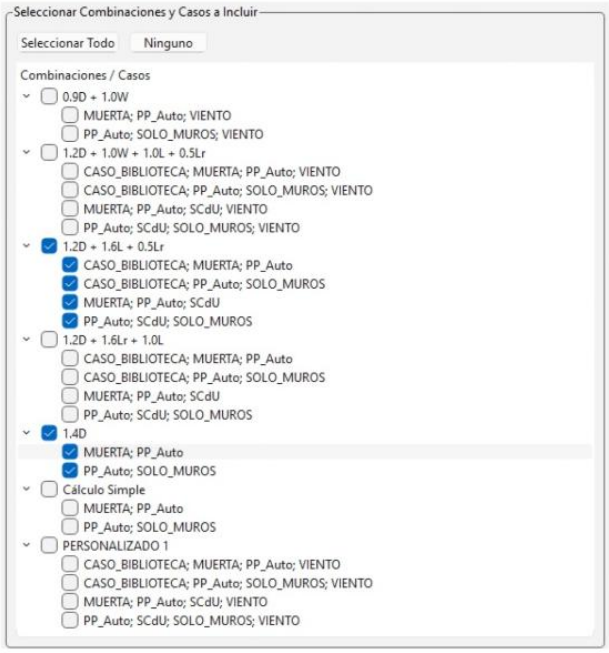
Ejemplo:

Reportes de Cálculo

Al presionar el botón de generar un reporte, aparecerá una ventana emergente con varias configuraciones a realizar antes de generar dicho reporte, a continuación, se explica cada una de dichas configuraciones:

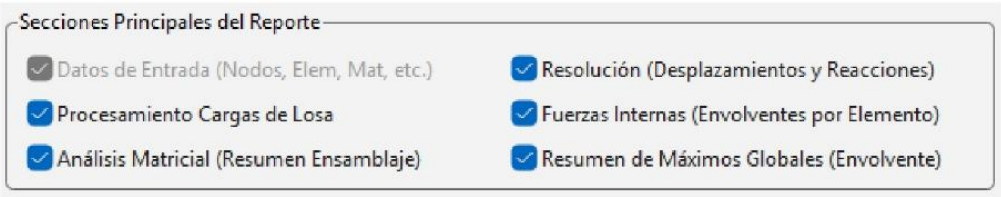
Combinaciones y Casos

En esta sección, el usuario selecciona que combinaciones desea utilizar, a su vez, dentro de cada una de dichas combinaciones, podrá seleccionar los casos o hipótesis que se usarán en cada combinación.



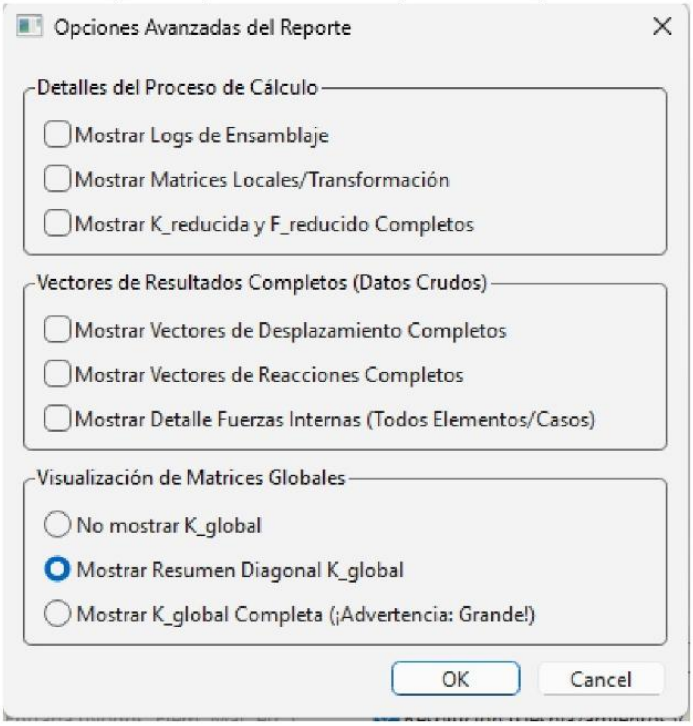
Secciones Principales del Reporte

Estas opciones permiten al usuario configurar de forma rápida las opciones que desea observar en el reporte de cálculo.



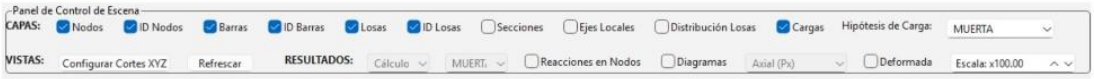
Opciones Avanzadas

Estas permiten al usuario tener un reporte mucho más completo, sin embargo, estos pueden llegar a ser mucho más extensos, motivo por el cual estas opciones son presentadas como avanzadas.



5. Opciones de Visualización 3D

En la parte superior de la Visualización 3D del modelo, se encuentran varias herramientas de visualización, a continuación, se explica cada una de ellas.



Geometría

- Checkbox "Nodos": Activada por defecto, permite mostrar y ocultar nodos.
- Checkbox "ID Nodos": Activada por defecto, permite mostrar y ocultar los ID de los nodos.
- Checkbox "Barras": Activada por defecto, permite mostrar y ocultar los elementos.
- Checkbox "ID Barras": Activada por defecto, permite mostrar y ocultar los ID de los elementos.
- Checkbox "Losas": Activada por defecto, permite mostrar y ocultar las losas.
- Checkbox "ID Barras": Activada por defecto, permite mostrar y ocultar los ID de las losas.
- Checkbox "Secciones": Desactivada por defecto, permite mostrar y ocultar la sección transversal de todos los elementos.
- Checkbox "Ejes Locales": Desactivada por defecto, permite mostrar y ocultar los ejes locales de todos los elementos.
- Checkbox "Distribución Losas": Desactivada por defecto, permite mostrar y ocultar la geometría de la distribución definida en todas las losas.

Cargas

- Checkbox "Cargas": Activada por defecto, permite mostrar y ocultar las cargas que pertenezcan a la hipótesis seleccionada en el selector siguiente.
- Selector "Hipótesis de Carga": Permite al usuario seleccionar la hipótesis de carga que desea ver.

Vistas

- Botón "Configurar Cortes XYZ": Abre una ventana emergente que te permite establecer los máximos y los mínimos de cada uno de los ejes globales, para poder establecer un corte que facilite la visualización del usuario.
- Botón "Refrescar": En caso de percibir que el gráfico no ha sido actualizado, este botón se encarga expresamente de actualizar la visualización.

Resultados

Estas opciones permiten visualizar resultados en el gráfico una vez que ha sido realizado el cálculo de la estructura.

- Selector de combinación: Permite seleccionar la combinación de cargas sobre la cual se evaluarán las reacciones, los diagramas de fuerzas internas y las deformaciones.
- Selector de hipótesis: Permite seleccionar las hipótesis de carga sobre las cuales se evaluarán las reacciones, los diagramas de fuerzas internas y las deformaciones.
- Checkbox "Reacciones en Apoyos": Desactivado por defecto, permite mostrar y ocultar llamadas de texto que muestran las reacciones en cada nodo que tenga un apoyo.
- Checkbox "Diagramas": Desactivado por defecto, permite mostrar y ocultar los diagramas de fuerzas internas
- Selector de efectos: Permite seleccionar los efectos que serán representados por los diagramas de fuerzas internas.
- Checkbox "Deformadas": Desactivado por defecto, permite mostrar y ocultar la deformada de la estructura acompañado de etiquetas en cada nodo que indica el valor del desplazamiento traslacional y rotacional.

6. Diseño de Vigas

La pestaña de Diseño de Vigas en VOTX3D está diseñada para proporcionar un control total sobre las variables de cálculo, permitiendo al usuario realizar verificaciones rápidas y detalladas de secciones específicas de hormigón armado.

Entrada Manual de Datos

En esta zona, el usuario define las condiciones críticas de la sección de manera manual, remitiendo una auditoría directa.

- Propiedades Mecánicas: Resistencia característica del hormigón (f'c) y el límite elástico del acero (fy).
- Solicitaciones de Diseño: Momento flector último (Mu) y esfuerzo cortante último (Vu).
- Geometría de la Sección: Base (b), altura (h) y recubrimiento (r).
- Armado Tentativo: Selección de diámetros para la armadura longitudinal y transversal, utilizados inicialmente para el cálculo del canto efectivo (d).
- Armado Previo: Esta función permite añadir armadura de montaje (u otra ya existente en la sección) para que el motor de cálculo la descuente de los requerimientos totales. De esta forma, el resultado indica exclusivamente el refuerzo adicional.

Resultados y Reporte de Cálculo

Al ejecutar el diseño, el programa devuelve tres valores clave de forma inmediata:

- As Tracción: Área de acero requerida en la zona traccionada.
- A's Compresión: Área de acero necesaria en caso de requerir armadura doble.
- Estribos: Configuración de la armadura transversal.

Simultáneamente a la derecha se abre la pestaña de "Reporte de Cálculo", donde se despliega todo el proceso matemático paso a paso, garantizando la trazabilidad de los resultados según la NB 1225001

Visualización de Diagramas

Esta sección es fundamental para la interpretación de resultados del análisis estructural y la obtención de esfuerzos para el diseño de vigas. Permite inspeccionar el comportamiento de los elementos mediante una interfaz altamente configurable.

Configuración de la Visualización

Se dispone de una serie de selectores para filtrar la información mostrada:

- **Selector de Elementos:** Identificación del miembro estructural a analizar.
- **Filtros de Análisis:** Selectores para Combinaciones de Carga, Hipótesis y Efectos.

- **Visualización de Armado:** Opción para superponer el armado registrado en el elemento sobre los diagramas de esfuerzos.

Análisis de envolventes

Mediante el checkbox “Envolvente”, el programa superpone todas las combinaciones de carga para mostrar los valores máximos y mínimos absolutos (contorneando los trazos de mayor solicitud).

Adicionalmente, al activarse el checkbox “Todas” se detalla con etiquetas y leyendas que identifican el nombre de cada combinación específica.

Herramientas de Presición

Para un análisis detallado, se pueden obtener etiquedas de datos específicos:

- **Valor en X:** Obtiene el valor del efecto seleccionado en una coordenada “x” específica.
- **X para un Valor:** Localiza la coordenada “x” donde ocurre un valor de efecto determinado.

Registro de Armados

El Registro de Armado es la base de datos local donde se consolidan las desiciones de diseño. Aquí se pueden guardar los armados longitudinales y transversales calculados previamente.

Este registro está vinculado directamente al elemento seleccionado en la pestaña de diagramas, permitiendo que la información de refuerzo persista y pueda ser consultada o visualizada en el modelo 3D en etapas posteriores del proyecto.

7. Diseño de Columnas

El módulo de diseño de columnas permite el análisis de secciones rectangulares sometidas a flexo-compresión y corte, con un enfoque en la generación y verificación de diagramas de interacción.

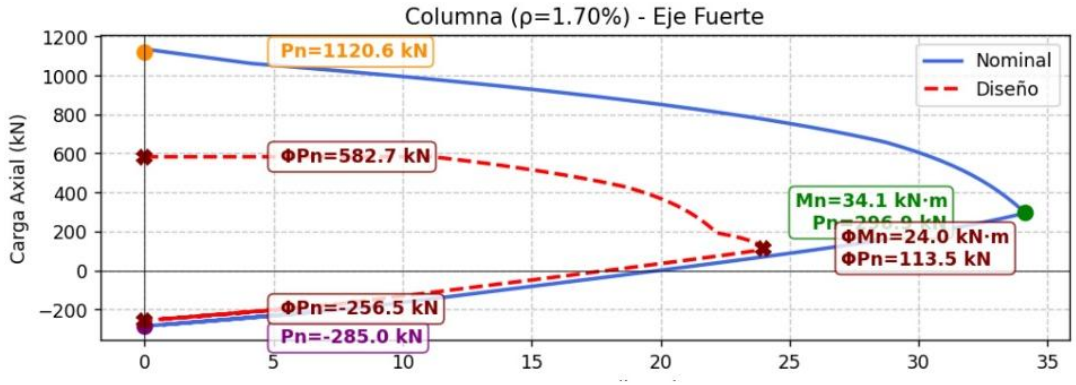
Diseño a Flexo-Compresión

En esta sub sección se configuran los parámetros base para la generación de la capacidad resistente de la sección.

- **Materiales y Geometría:** Entrada manual de f'c, fy, Base, Altura y Recubrimiento.
- **Eje de Análisis:** Permite seleccionar entre el eje fuerte (Mx) y el eje débil (My) para el cálculo del diagrama de interacción correspondiente.
- **Configuración de Refuerzo:** Se definen los diámetros de barra longitudinal y transversal.
- **Distribución Perimetral:** El usuario ingresa el número de barras en las caras H (altura) y B (base). El sistema ubica las barras únicamente en el contorno. Ejemplo: Si se definen 3 barras en H y 3 en B, el total será de 8 barras perimetrales.
- **Retroalimentación Visual:** Un gráfico de la sección transversal se actualiza en tiempo real mostrando la diposición exacta de las barras y las dimensiones de la sección según los datos ingresados.

Diagramas de Interacción

Una vez definidos los datos, el motor genera el diagrama de interacción nominal y reducido (ϕ). El programa identifica puntos críticos denominados Valores Extremos que definen la frontera de falla



Diseño a Corte

Al cambiar a la pestaña de diseño a corte, el usuario debe introducir:

- **Cortante Último (Vu)**
- **Carga Axial Última (Un)**

El programa utiliza la geometría previa y al ejecutar el diseño, abre una memoria de cálculo de corte donde se detalla el procedimiento paso a paso de dicho diseño y sus resultados.

Diagramas de Esfuerzos

Similar al módulo de vigas, permite visualizar los resultados del análisis estructural global para columnas específicas. Sin embargo, no incorpora visor de envolventes, pues se centra en combinaciones e hipótesis individuales.

- **Selectores:** Selector de columna, combinación, hipótesis y efecto.

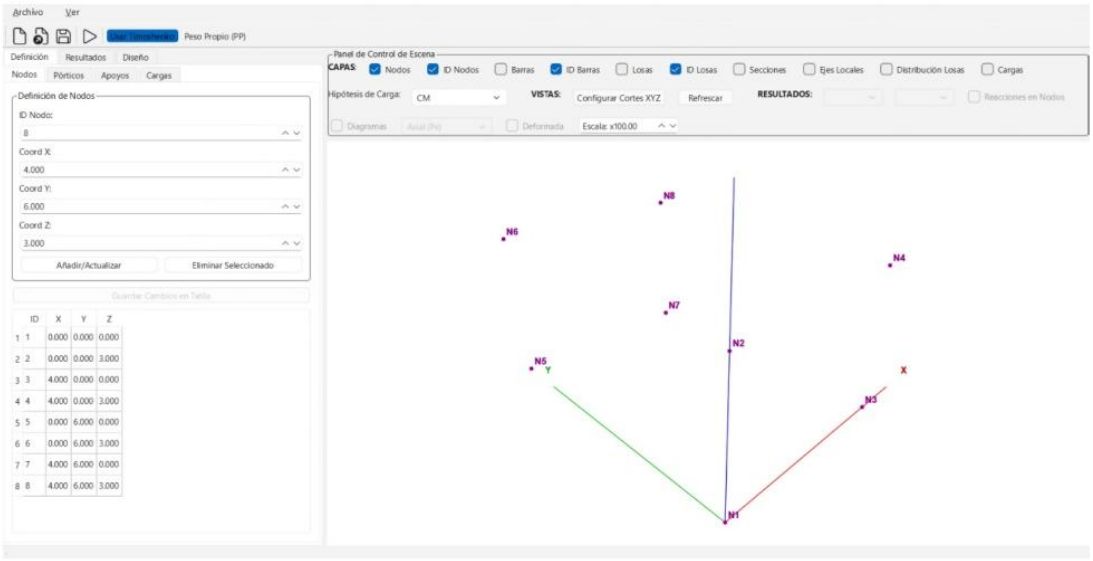
Demandas y Puntos de Diseño

Para validar la seguridad, el usuario puede superponer puntos de demanda sobre el diagrama de interacción

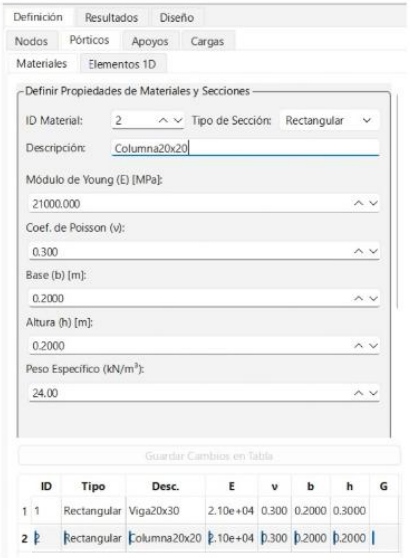
- **Entrada Manual:** Introducción directa de pares (P, M).
- **Importación del Análisis:** Al seleccionar una columna en el diagrama de esfuerzos, el programa puede extraer automáticamente todas las solicitaciones de las combinaciones de carga y graficarlas como puntos dentro o fuera de la curva de capacidad.

8. Ejemplo de Aplicación

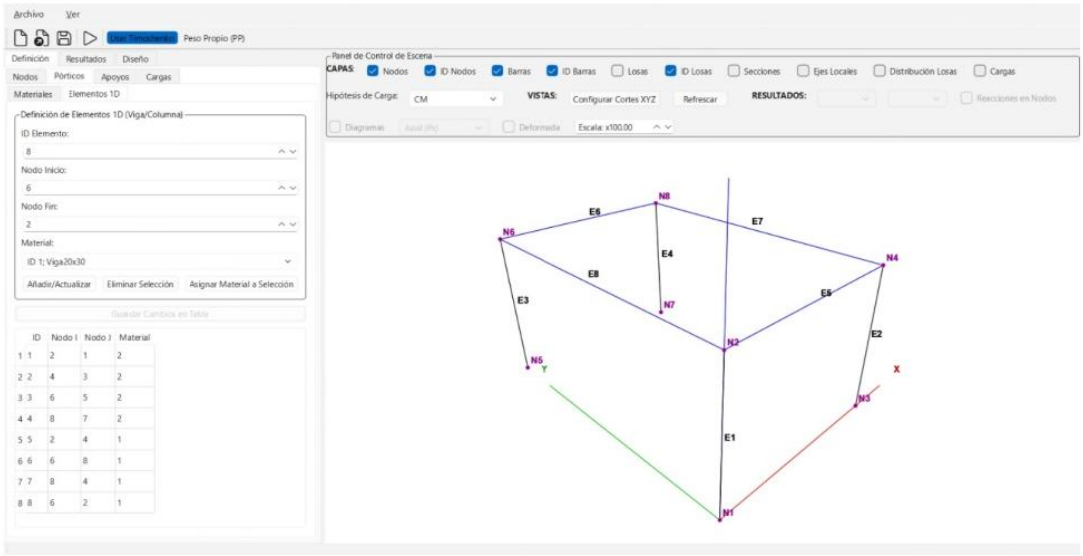
Introducción de Nodos



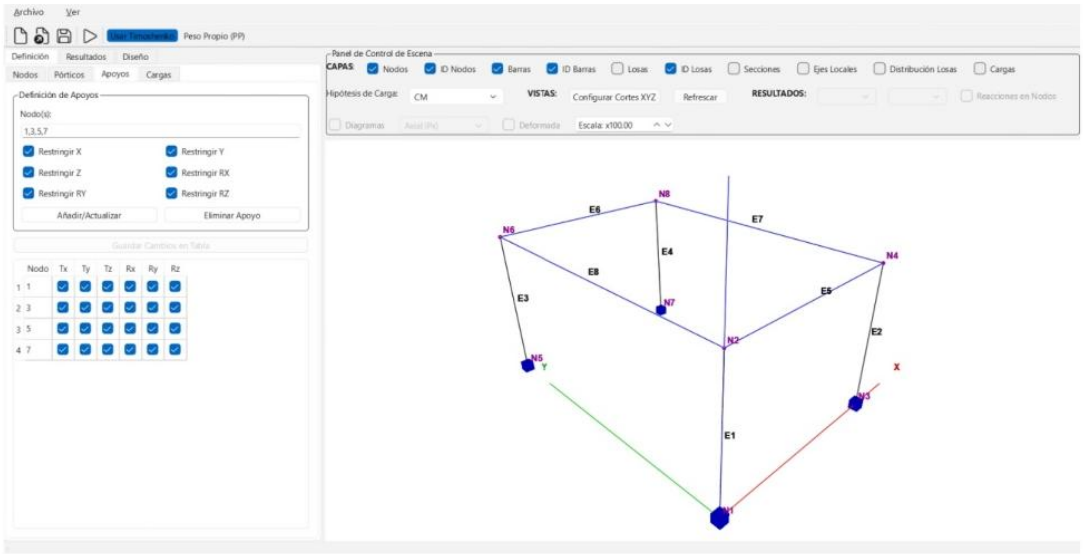
Creación de Materiales y Secciones



Definición de Elementos



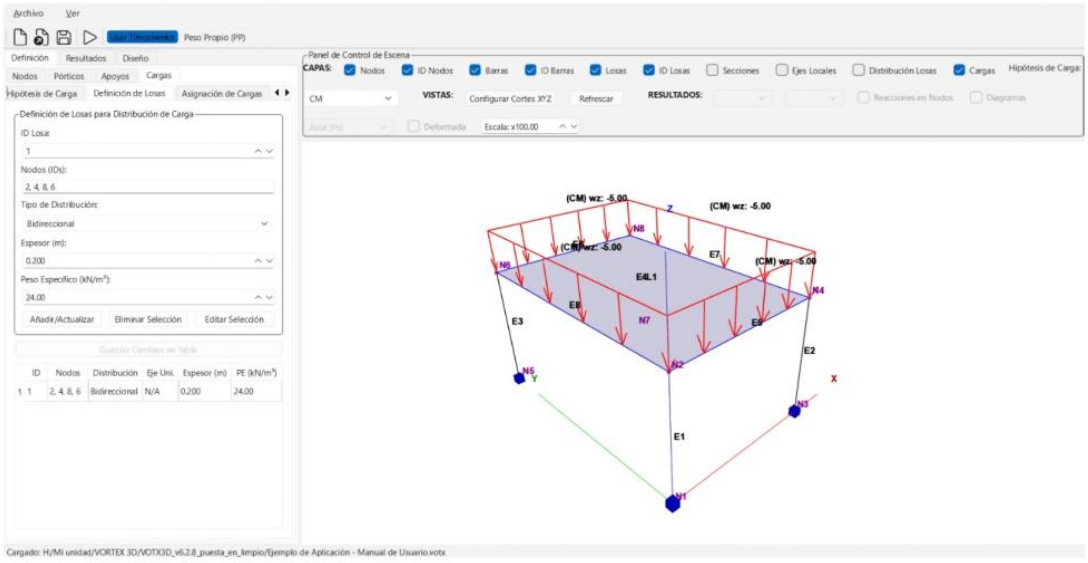
Definición de Apoyos



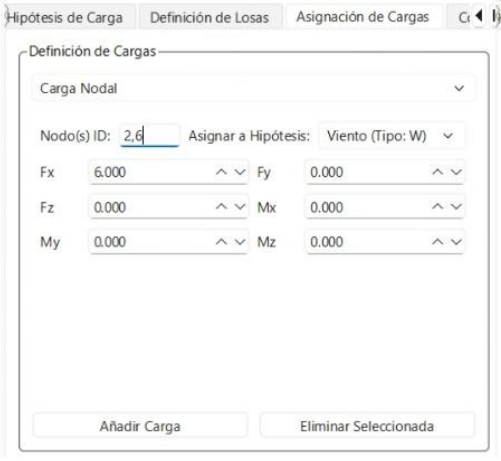
Hipótesis de Carga

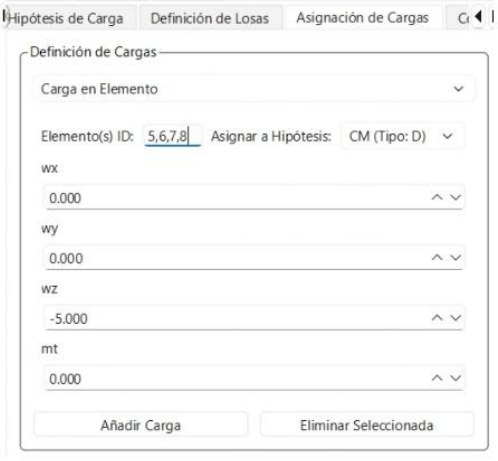
	ID	Nombre	Tipo
1	1	CM	D
2	2	SC	L
3	3	Viento	W
4	4	Granizo	S

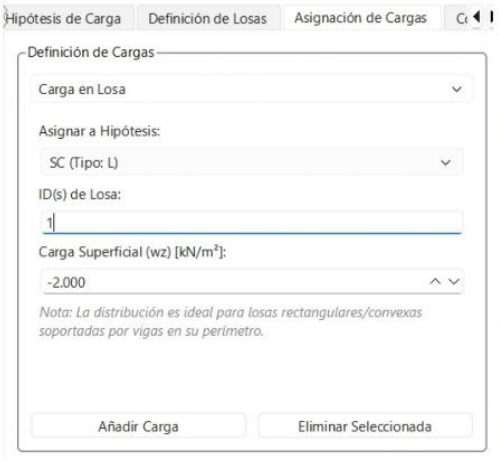
Definición de Losas



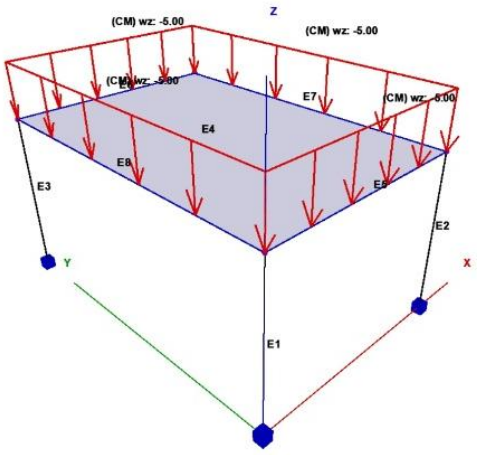
Asignación de Cargas

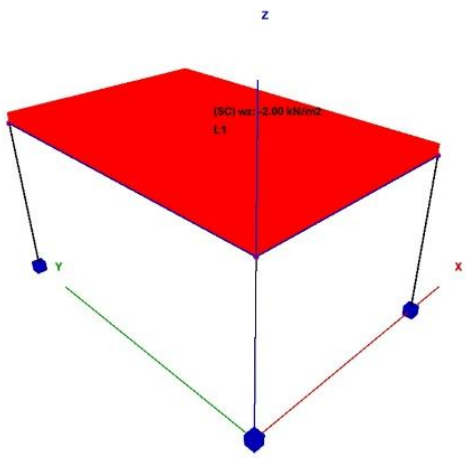


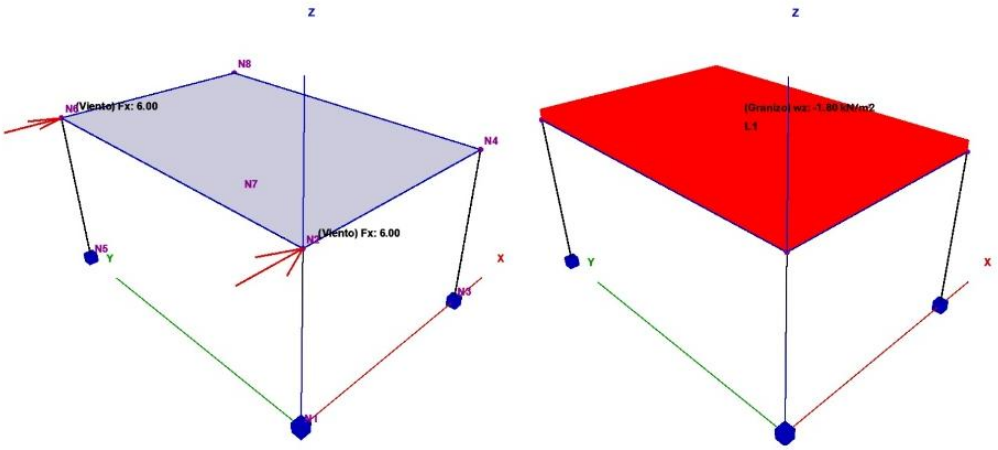




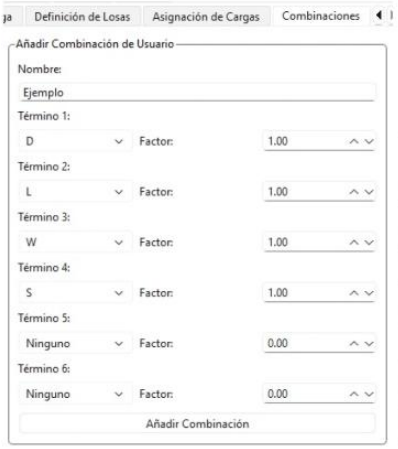
	ID Carga	Hipótesis	Ubicación	Valores
1	1	Viento	Nodo 2	Fx: 6.00
2	2	Viento	Nodo 6	Fx: 6.00
3	3	CM	Elem 5	wz: -5.00
4	4	CM	Elem 6	wz: -5.00
5	5	CM	Elem 7	wz: -5.00
6	6	CM	Elem 8	wz: -5.00
7	1	SC	Losa 1	wz: -2.00 kN/m²
8	2	Granizo	Losa 1	wz: -1.80 kN/m²







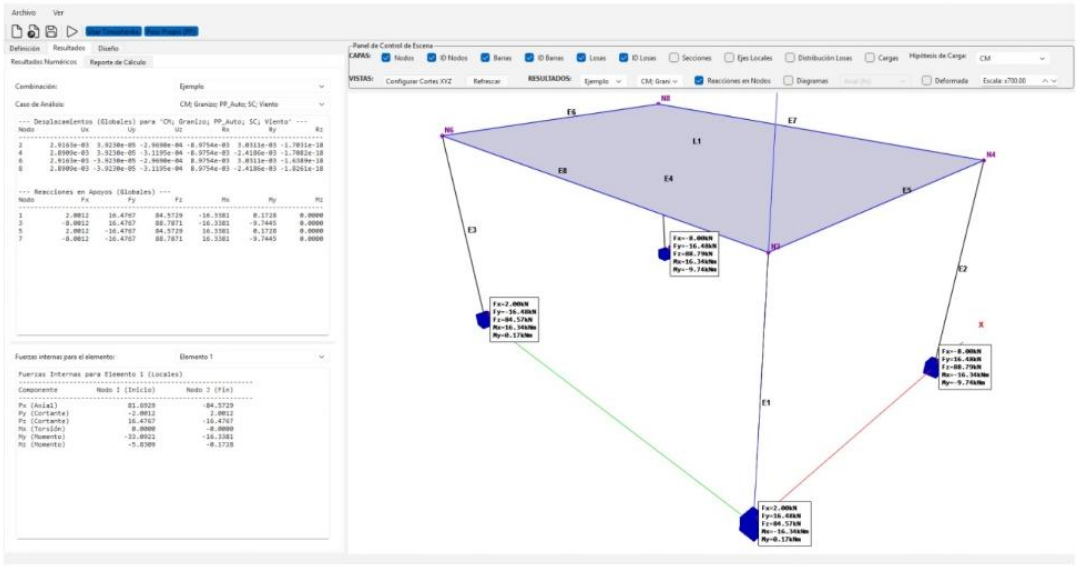
Combinaciones de Carga



	Nombre	Factores	Tipo
1	Cálculo Simple	{'D': 1.0}	Norma
2	1.4D	{'D': 1.4}	Norma
3	1.2D + 1.6L + 0.5Lr	{'D': 1.2, 'L': 1.6, 'Lr': 0.5}	Norma
4	1.2D + 1.6Lr + 1.0L	{'D': 1.2, 'Lr': 1.6, 'L': 1.0}	Norma
5	1.2D + 1.0W + 1.0L + 0.5Lr	{'D': 1.2, 'W': 1.0, 'L': 1.0, 'Lr': 0.5}	Norma
6	0.9D + 1.0W	{'D': 0.9, 'W': 1.0}	Norma
7	Ejemplo	{'D': 1.0, 'L': 1.0, 'W': 1.0, 'S': 1.0}	Usuario

Cálculo

Considerando Teoría de Timoshenko y Peso Propio de la estructura:



9. Descarga del Programa

El programa VORTEX 3D es de distribución gratuita y código abierto. Todo el material del proyecto, incluyendo el código fuente y las versiones compiladas, se encuentra alojado públicamente en la plataforma GitHub.

Para facilitar el uso a ingenieros y estudiantes, se ha empaquetado el programa en un archivo ejecutable (.exe) para el sistema operativo Windows. Esto permite utilizar la herramienta de forma directa y portable, sin necesidad de instalar lenguajes de programación o configurar entornos de desarrollo.

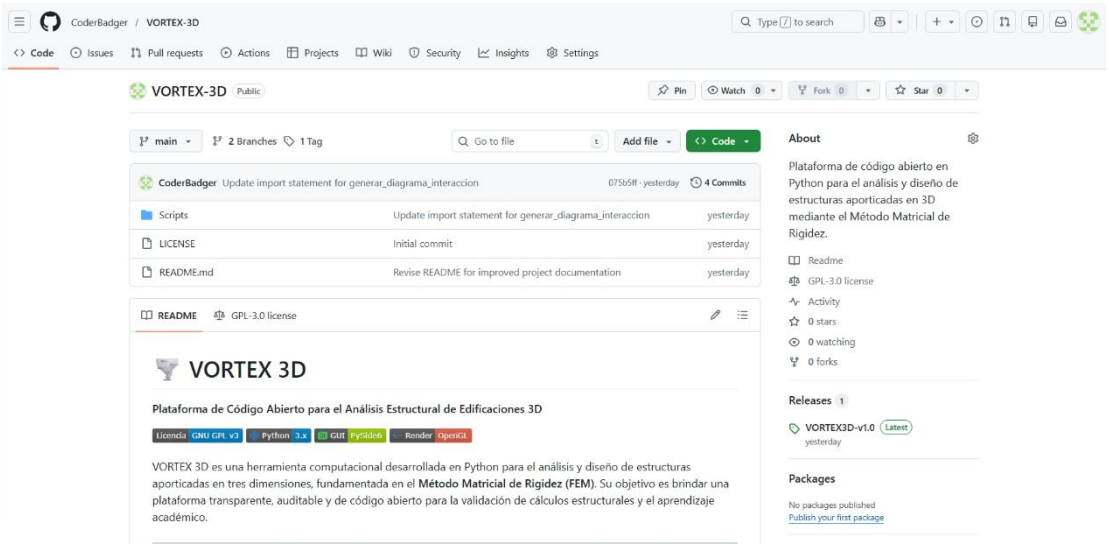
Para descargar el programa, siga los siguientes pasos:

Paso 1: Acceso al Repositorio Oficial

Ingrese a la página principal del repositorio de VORTEX 3D en GitHub a través del siguiente enlace:

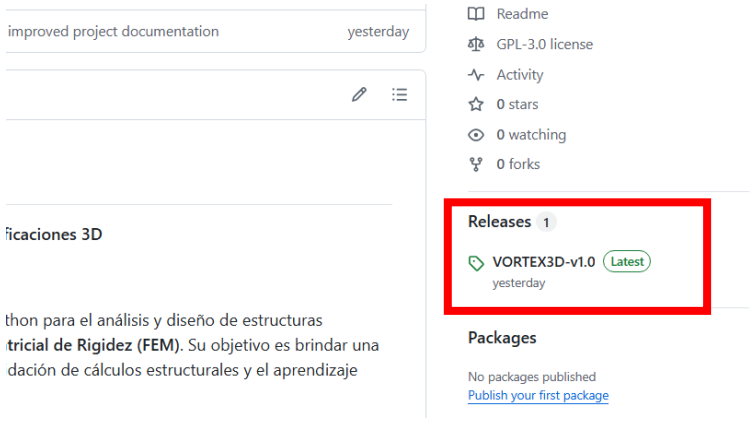
https://github.com/CoderBadger/VORTEX-3D.git

En esta pantalla principal encontrará la descripción del proyecto, el código fuente y el archivo README con la información general.



Paso 2: Ubicar la sección de Lanzamientos (Releases)

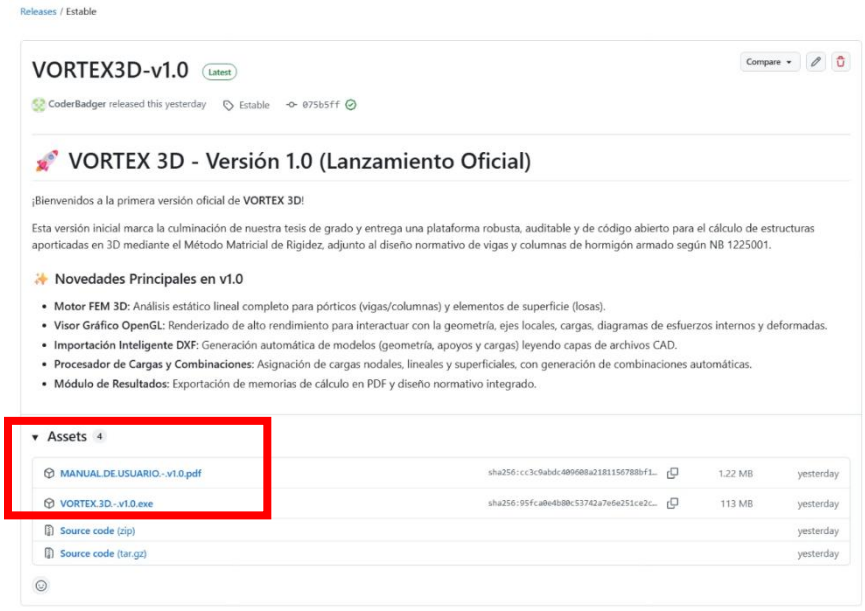
En la parte lateral derecha de la pantalla principal, localice la sección denominada "Releases" (Lanzamientos). Haga clic en la versión más reciente (ej. Versión 1.0) para acceder a los archivos de descarga.



Paso 3: Descarga del Ejecutable

Dentro de la página de la versión seleccionada, diríjase a la sección inferior llamada "Assets" (Recursos). Allí encontrará una lista de archivos adjuntos.

Haga clic sobre el archivo ejecutable del programa para iniciar la descarga en su computador. En esta misma sección también podrá encontrar adjunto este Manual de Usuario en formato PDF y el código fuente del programa.



Paso 4: Ejecución del Programa

Una vez finalizada la descarga, ubique el archivo .exe en su carpeta de descargas. Al ser un programa portable, no requiere de un proceso de instalación tradicional; simplemente haga doble clic sobre el archivo para ejecutar VORTEX 3D y comenzar a modelar.

Desarrollado por:

- Diego Oliver Vargas Moya
- Luis Alberto Ortiz Morales