# Optimización de un Sistema de Suspensión Pasiva con el Uso de un Algoritmo Evolutivo

1st Gabriel Orlando Gonzalez Rodriguez, gabrielgr01@estudiantec.cr
2nd David Areus Fuentes Cerdas, davidareus@estudiantec.cr
Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Mecatrónica
MT8008 Inteligencia Artificial

**Resumen**

En el siguiente informe se presenta la implementación de un algoritmo evolutivo para optimizar un sistema de suspensión pasiva. El modelo físico corresponde a un sistema masa-resorte-amortiguador de un solo grado de libertad en el que se busca la minimización de su desplazamiento (estabilidad) y aceleración (confort) vertical. A como se discutirá, el problema se clasifica como multi-objetivo y busca combinaciones de su constante del resorte (o rigidez) $k$ y su constante de amortiguamiento $b$ para su solución. Se obtiene un Frente de Pareto con todas las soluciones posibles, y se implementa una función para obtener una única solución según la preferencia del usuario y la distribución de pesos que se le asigne a los criterios a optimizar.

**Palabras Clave**

Inteligencia Artificial, Algoritmo Evolutivo, Optimización, Suspension Pasiva.

## I. Introducción

### *I-A. Algoritmos Evolutivos*

Los algoritmos evolutivos son un método de optimización que se inspira en la evolución biológica para la obtención de soluciones, por medio de variables que se codifican en cromosomas para representar soluciones candidatas [3]. En primer lugar se genera una población inicial con cromosomas aleatorios. A cada solución, la cual debe estar restringida por los rangos aceptables predefinidos de las variables o espacio de alelos, se le asigna una calidad por medio de un función de evaluación. La calidad es una representación escalar para comparar individuos respecto a los objetivos del problema y ayuda en el proceso de selección. Durante la selección se hace uso de la calidad para decidir cuales de los individuos avanzan a la etapa de cruce. En la mayoría de los casos se busca un equilibrio entre darle prioridad a individuos con las mejores calidades, para aumentar la convergencia hacia una solución (presión selectiva) y tomar otros individuos con calidades menores pero que son diferentes y permiten explorar más soluciones (variabilidad genética).

A partir de los individuos seleccionados se generan nuevas soluciones que toman características de los padres y tienen una probabilidad de mutar. La mutación provoca que los cromosomas de la descendencia varíen alrededor de los cromosomas que heredan de los padres. La mutación es un operador dentro de los algoritmos evolutivos que permite introducir y controlar la variabilidad genética [3]. Una vez se terminan de generar los nuevos individuos es necesario usar un mecanismo de remplazo para que el tamaño de la población se mantenga constante y se evite el crecimiento descontrolado de los recursos computacionales. Para el caso de problemas multiobjetivo, donde los objetivos de optimización son conflictivos, no se obtiene una única solución sino un conjunto de soluciones no dominadas, también llamada frente de pareto.

### *I-B. Sistema de Suspensión Pasiva*

Los sistemas de suspensión pasiva, de un grado de libertad, permiten modelar el desplazamiento vertical de un vehículo respecto a sus ruedas. Este tipo de modelo de suspensión se distingue por tener valores fijos de constantes de amortiguación y resorte [1]. Un diagrama del sistema se puede observar en la figura 1, donde M corresponde a una cuarta parte de la masa del automóvil, b es la constante de amortiguación, k es la constante del resorte y u(t) es la fuerza de entrada al sistema. Las constantes b y k influyen en las características del sistema, las de mayor interés para el caso de estudio, son la ganancia en estado estable, la estabilidad y el sobre impulso. Ya que los parámetros b y k tienen influencia sobre estas características, una combinación adecuada permite reducir el desplazamiento máximo y aceleración máxima para un sistema con la misma masa y fuerza de entrada. En el ámbito de estudio estos son los objetivos que se desean optimizar, el confort (aceleración vertical) y la estabilidad (desplazamiento vertical). Nótese, la diferencia entre el objetivo de estabilidad, el cual forma parte de la evaluación de un individuo y la estabilidad propia del sistema que se refiere a la convergencia en estado estable.

Por lo general, la optimización del confort y la estabilidad en este tipo de sistema representa un problema del tipo multiobjetivo, como es el caso en [2]. Los algoritmos evolutivos son especialmente útiles para este tipo de problemas, este es uno de los principales motivos por el que se implementa el uso de uno. Sin embargo, una verificación del tipo de sistema es necesaria y se puede observar en la sección II-B.

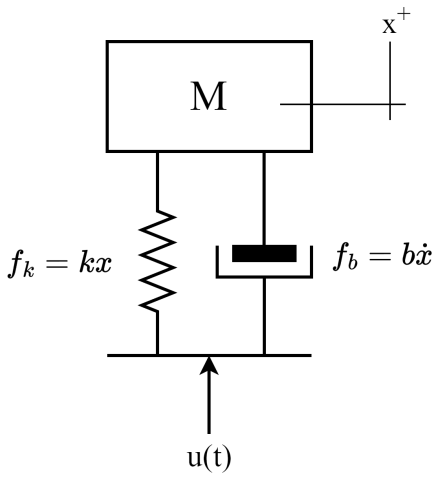


Figura 1: Diagrama de cuerpo libre del sistema.

## II. Análisis del Problema

### *II-A. Modelo del Sistema*

Para el problema de estudio, se busca encontrar la combinación de valores de la constante del resorte (k) y la constante de amortiguamiento (b) del sistema explicado en la sección anterior para poder minimizar el desplazamiento (estabilidad) y la aceleración (confort) del mismo. Estamos entonces ante un problema de optimización, con entradas $k$ y $b$, y salidas $x$ (desplazamiento) y $a$ (aceleración).

Al aplicar la segunda ley de Newton "suma de fuerzas igual a masa por aceleración" en el diagrama de la figura 1 se obtiene la ecuación dinámica 1. Esta ecuación representa el modelo del sistema masa-resorte-amortiguador y se utilizará para poder simular el sistema.

$$\sum F = m \cdot a$$

$$u(t) - f_k - f_b = m \cdot a$$

$$u(t) - kx - b\dot{x} = m\ddot{x} \tag{1}$$

Como se verá en la sección siguiente, se realiza un módulo específico para manejar el modelo del sistema y de este se obtienen las simulaciones que se pueden observar en el ANEXO A. En la figura 3 se muestra una primera prueba de la simulación del sistema, y en la figura 4 se observa otra prueba de la simulación ya con valores de $k$ y $b$ dentro del rango de estudio.

### *II-B. Identificación del Tipo de Problema*

Por ser un problema de optimización, el siguiente paso es identificar con qué tipo de problema se está tratando, si es mono-objetivo o multi-objetivo. Esto va a ser crucial para definir la estrategia correcta para resolver el problema y para no implementar estrategias con complejidad innecesaria si el problema no lo requiere. Para esto se realiza un estudio de sensibilidad o "ceteris paribus" en donde se busca ver el comportamiento de la salida ante variaciones de una de las entradas mientras se mantienen las otras fijas, y así para cada entrada.

A como se explicará en la sección anterior, se realiza un módulo, llamado analysis.py, para realizar este tipo de estudios de datos y del problema. La función analysis.ceteris_paribus() genera una serie de gráficas para combinaciones de $b$ variable y $k$ fijas, y para combinaciones de $k$ variable y $b$ fijas. En el ANEXO B se incluye una muestra de estas gráficas, cuatro en específico, en donde se puede analizar el comportamiento del sistema ante distintas regiones del espacio de entrada. Para algunos casos, el sistema se comporta como un problema mono-objetivo, como se observa en las figuras 5 y 7, sin embargo, basta con que el sistema se comporte de forma no monótona o con un "trade-off", donde al aumentar la entrada variable se minimiza una salida pero se maximiza otra, para poder concluir que el sistema, en general, es multi-objetivo, siendo este el caso para nuestro problema. En las figuras 6 y 8 se observa como en algunas regiones del espacio de entrada el sistema efectivamente se comporta como un problema multi-objetivo ya que no existe una sola solución no dominada, significando esto que, para nuestro problema, no existe ninguna solución que minimice ambas salidas simultáneamente.

## III. Estrategia a Seguir

Para solucionar el problema de optimización (minimización) multi-objetivo se implementa un algoritmo evolutivo, estrategia común para este tipo de situaciones. En este caso se realiza en el lenguaje de programación Python y se organiza el código de la siguiente manera, separando en módulos las funciones según su objetivo:

- main.py : Posee todo lo relacionado al algoritmo evolutivo como tal. (Código en el ANEXO D)
- modules/config.py : Posee las definiciones de variables globales y parámetros a utilizar en el programa. (Código en el ANEXO E)
- modules/model.py : Funciones relacionadas a la generación y resolución del modelo del sistema. (Código en el ANEXO F)
- modules/analysis.py : Funciones relacionadas a analisis de los datos y del problema. (Código en el ANEXO G)
- modules/utils.py : Funciones miscelaneas útiles para el programa. (Código en el ANEXO H)
- modules/__init__.py : Archivo de inicialización necesario para poder acceder a los modulos desde el main.py.

Nota: En el anexo ANEXO C se puede observar una representación más gráfica de la estructura de archivos utilizada y se incluyen además los archivos de imágenes generadas por el programa una vez ejecutado.

### *III-A. Selección de Hiperparámetros y Operadores del Algoritmo Evolutivo*

El primer aspecto a tomar en cuenta para la definición del espacio de alelos son los valores de k y b que generan individuos que representan sistemas inestables y sin sentido físico. La estabilidad del sistema se puede evaluar por medio del factor de amortiguamiento $\zeta$. Para el sistema estudiado el factor de amortiguamiento se puede observar en la ecuación 2, de donde se puede deducir que b y k deben ser ambos números reales positivos. Las variaciones típicas de k se dan entre los 16000 $N/m$ y 73000 $N/m$ según [5]. A partir de este rango se seleccionó un límite superior más conservador de 50000 $N/m$. Y un límite inferior de 1000 $N/m$ para incluir en la exploración soluciones con constantes de resorte más bajas y por lo tanto mayor variabilidad genética. Según [6] las variaciones típicas de b se encuentra de 1000 $Ns/m$ 2500 $Ns/m$. Para expandir la exploración a una mayor número de soluciones se seleccionó un rango de 100 $Ns/m$ a 5000 $Ns/m$.

$$\zeta = \frac{b}{2\sqrt{km}} \tag{2}$$

Rangos seleccionados para las variables de entrada:

- k = [1000, 50000]
- b = [100, 5000]

Además, se detectó que a la hora de graficar es necesario un muestreo de al menos 200, el cual se define en el archivo config.py en la variable “t_samples”. De lo contrario, se obtienen resultados poco precisos y en algunos casos sin sentido.

Para lograr que el algoritmo evolutivo funcionara de la forma correcta, se hizo una selección de hiperparámetros y operadores coherente con la naturaleza del problema y posteriormente, ajustando ligeramente algunos valores y realizando unas pocas pruebas, se obtuvo el algoritmo adecuado y resultados coherentes.

Hiperparámetros seleccionados:

- popu_size = 100 (Tamaño inicial de la población)
  Provee suficiente variabilidad para nuestro problema.
- generations = 50 (Número de generaciones o iteraciones del algoritmo)
  Son suficientes para que el algoritmo converja.
- alpha = 0.5 (Define el rango de valores para generar el cruce)
  Si es cero el indiviuo se genera estrictamente entre los valores de los padres. Si está entre cero y uno se permite una exploración más allá de los padres según su valor.
- mate_chance = 0.75 (Probabilidad de que un par de individuos se crucen)
- parent_popu_size = 25 (Cantidad de individuos a convertirse en padres por cada generación)
  No hace mayor diferencia en nuestro caso.
- child_popu_size = 25 (Cantidad de hijos obtenidos en cada generación)
  No hace mayor diferencia en nuestro caso.
- mutate_chance = 0.2 (Probabilidad de que un individuo mute)
- mu = 0.0 (Parámetro para el operador de mutación)
  Un valor de cero provee una perturbación simétrica del gen.

Operadores seleccionados (de la librería DEAP [4]):

- Selección: tools.selNSGA2
  Es un operador recomendado, prácticamente estándar, para problemas multiobjetivo y provee mejores resultados que operadores básicos como el de ruleta o torneo.

- Cruce: tools.cxBlend
  Es un operador utilizado en sistemas continuos, como es nuestro caso.
- Mutación: tools.mutGaussian
  También es un operador utilizado para sistemas contínuos y realiza una exploración controlada que no se contrapone demasiado con al convergencia del sistema.

## IV. Solución y Resultados

### *IV-A. Obtención del Frente de Pareto*

Una vez ejecutado el algoritmo evolutivo con los parámetros y operadores adecuados, se obtiene un grupo de soluciones no dominadas para el problema. A estas soluciones se les conoce también como el "Hall Of Fame" y representan todas las soluciones posibles en donde se logró optimizar, o minimizar para nuestro caso, la salida del sistema con distintas combinaciones de las variables de entrada. Al graficarlas se obtiene el conocido Frente de Pareto, que se puede observar en la figura 2.

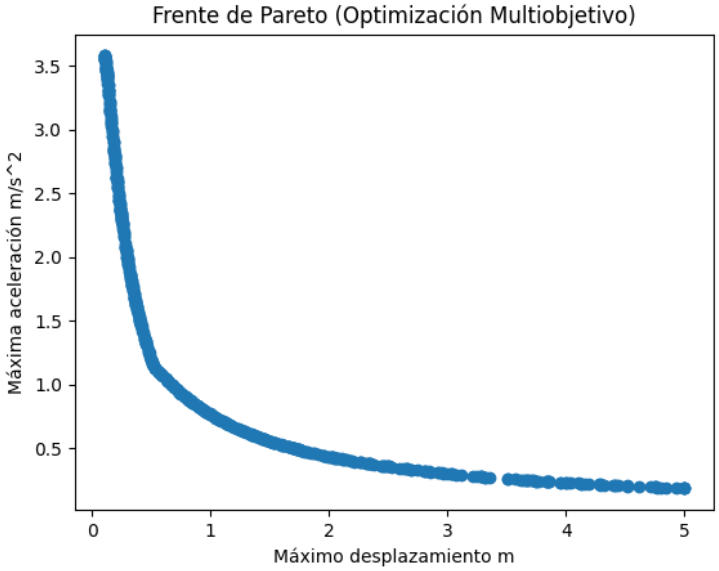


Figura 2: Frente de Pareto

### *IV-B. Selección de la Solución Preferida*

Para cada una de las soluciones del Frente de Pareto existe una combinación de entradas $k$ y $b$ igualmente válidas para resolver el problema de minimización. Sin embargo, en la práctica se buscaría obtener una única solución para poder implementarla en el sistema físico. En estos casos, lo habitual es elegir una de las soluciones según la preferencia que tenga el usuario sobre a cual salida le quiere dar más peso. Para esto, se programó la función get_preferred_solution() que, según el valor porcentual que se le ingrese a la variable "x_to_a_preference" en el módulo config.py, obtiene una única solución según los pesos asociados a cada salida.

La función recorre cada solución del Frente de Pareto y realiza la operación presente en la ecuación 3, la cual asigna un nuevo puntaje a cada una de ellas y la que obtenga un el puntaje mínimo, por ser un problema de minimización, es seleccionada.

$$puntaje = x\_to\_a\_preference \cdot x_{max} + (1 - x\_to\_a\_preference) \cdot a_{max} \tag{3}$$

A continuación se muestra un ejemplo para un valor de $x\_to\_a\_preference = 0{,}7$.

```
--> Getting Prefered Solution ...
- Using preference:  70 % for displacement.  30 % for acceleration.
- Best individual:
        k =  9671.49230215195  b =  5000
- Output:
        Displacement:  0.516983302343006 . Acceleration:  1.1547527631711636
```

## V. Conclusiones

- El problema de optimizar un sistema de suspensión pasiva (masa-resorte-amortiguador) corresponde a un problema multi-objetivo.
- Para rangos de entrada $1000 < k < 50000$ y $100 < b < 5000$, típicos en sistemas de suspensión pasiva, se obtiene el frente de pareto de la figura 2, que minimiza los valores de x (desplazamiento) y a (aceleración).
- Para una preferencia con pesos del 70 % para el desplazamiento y del 30 % para la aceleración, se obtiene que el sistema se minimiza con valores de entrada de k = 9671.5 y b = 5000, dando como resultado un desplazamiento de 0.5 $m$ y una aceleración de 1.16 $m/s^2$

## VI. Observaciones y comentarios Finales

- Los datos obtenidos toman en cuenta una fuerza de entrada al sistema de $u(t) = 5000\ N$, que se buscó que diera resultados más o menos realistas con la naturaleza del problema. Sin embargo, se le puede asignar otro valor a este parámetro según lo que el usuario considere apropiado. De igual forma, cualquier variable definida en el archivo config.py puede ser modificada según la preferencia del usuario.
- El programa solo soporta fuerzas de entrada $u(t)$ de tipo escalón, o en otras palabras, valores escalares.
- En el siguiente enlace se encuentra el repositorio de GitHub en donde se puede descargar el código de forma más accesible: https://github.com/Gabrielgr01/AI-Passive-Suspension-System-Optimization.git

## Referencias


[1] "Physics-Guided Reinforcement Learning System for Realistic Vehicle Active Suspension Control," Arxiv.org, 2023. https://arxiv.org/html/2408.08425v1 (accessed Jun. 18, 2025).

[2] S. Segla and N. Trišović, "Modeling and Optimization of Passive Seat Suspension," American journal of mechanical engineering, vol. 1, no. 7, pp. 407–411, Jan. 2013, doi: https://doi.org/10.12691/ajme-1-7-51.

[3] A. E. Eiben and J. E. Smith, Introduction to Evolutionary Computing, 2nd ed. Berlin, Germany: Springer, 2015.

[4] DEAP 1.4.1 Documentation. "Operators and Algorithms" https://deap.readthedocs.io/en/master/tutorials/basic/part2.html (accessed Jun. 17, 2025.)

[5] T. Shelton and B. Winkel, "STUDENT VERSION Modeling Car Suspensions." Available: https://www.simiode.org/resources/7871/download/3-034-S-CarSuspensions-StudentVersion.pdf

[6] M. Słomczyński, S. Radkowski, and M. Makowski, "Model of a Quarter Car Suspension with a Damper Containing Magnetorheological Fluid and with Damaged Parts Controlled by Backstepping Method," Energies, vol. 16, no. 7, p. 3044, Mar. 2023, doi: https://doi.org/10.3390/en16073044.

# ANEXOS

## ANEXO A. MODELO DEL SISTEMA MASA-RESORTE-AMORTIGUADOR

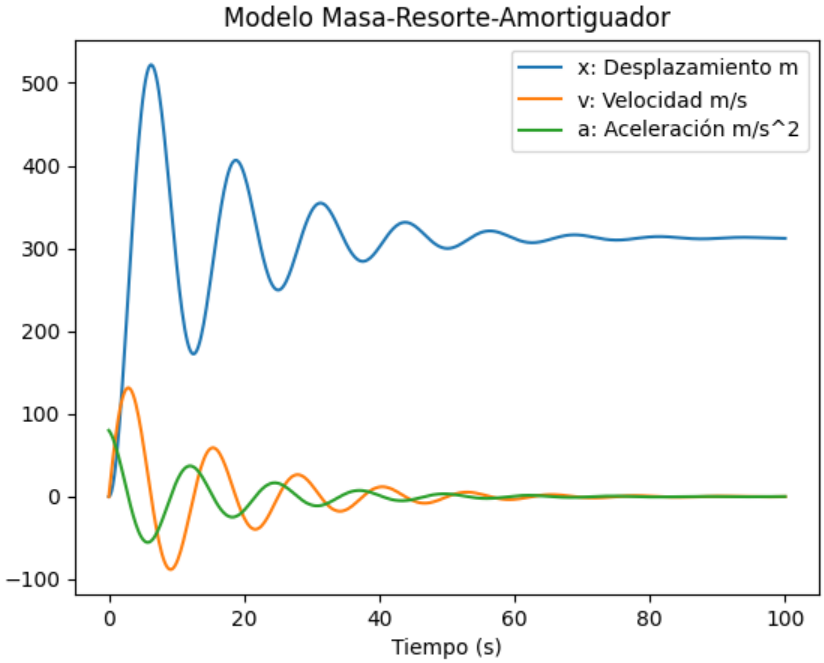


Figura 3: Simulación del sistema para valores de k = 64 y b = 32.

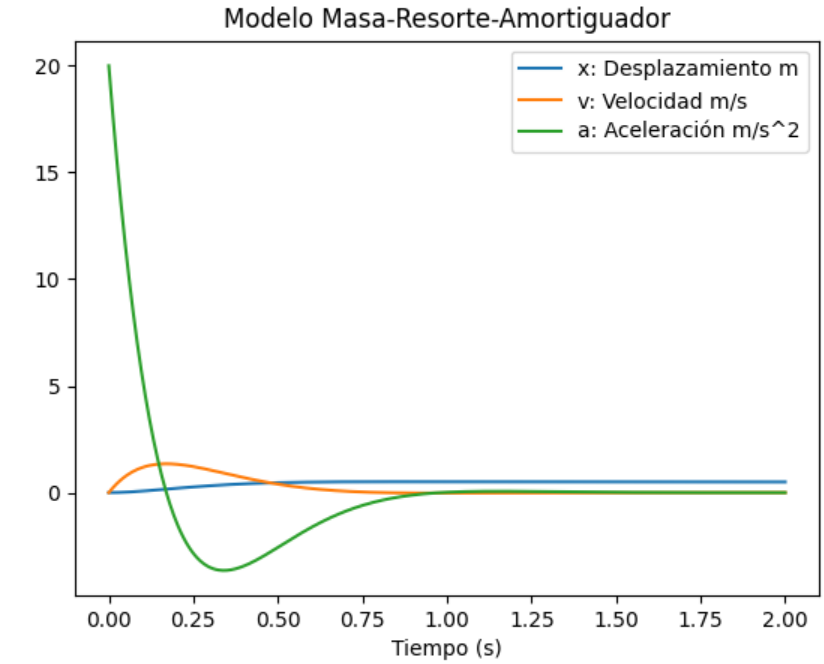


Figura 4: Simulación del sistema para valores de k = 10 000 y b = 2500.

## ANEXO B. IDENTIFICACIÓN DEL TIPO DE PROBLEMA

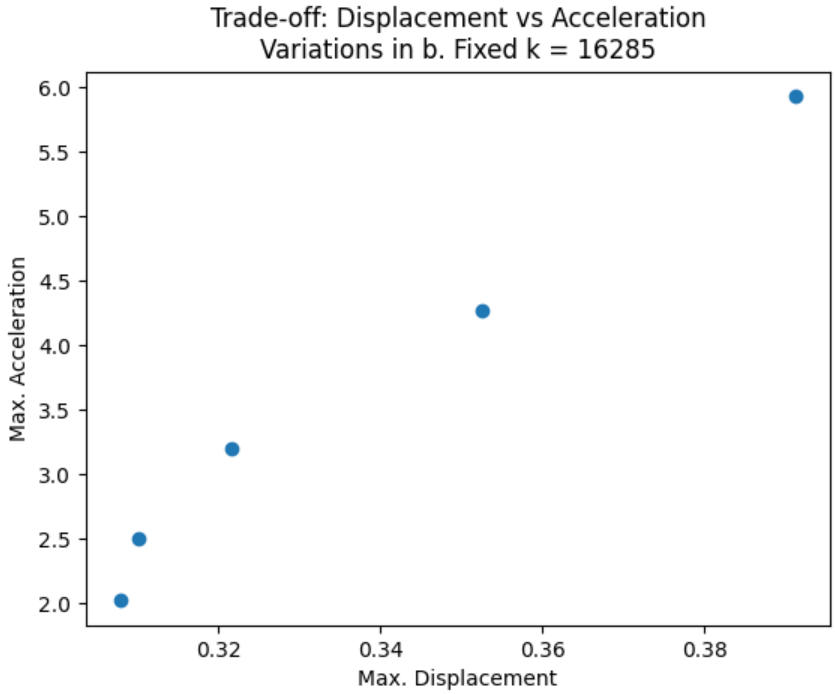


Figura 5: Región del espacio de entrada con comportamiento mono-objetivo para b variable.

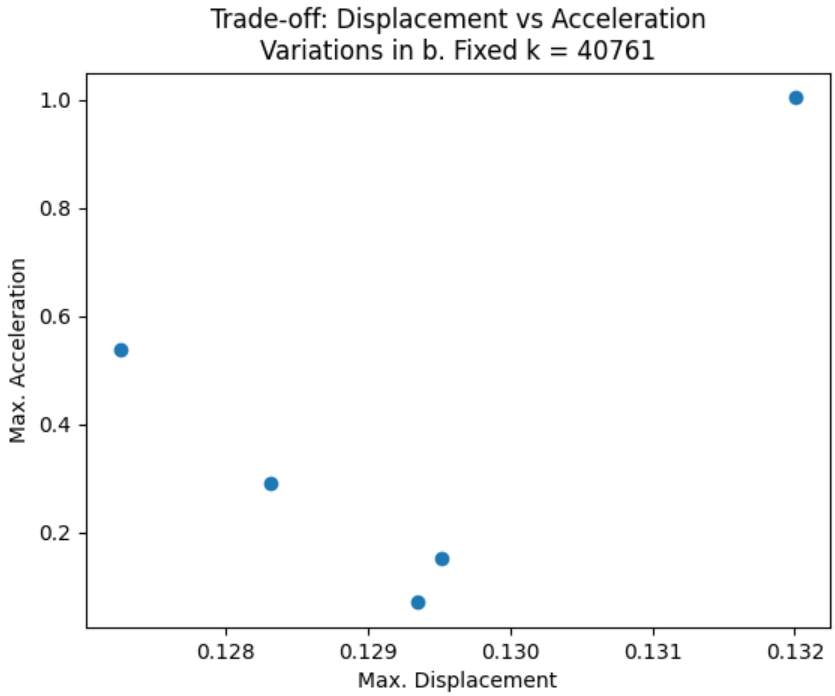


Figura 6: Región del espacio de entrada con comportamiento multi-objetivo para b variable.

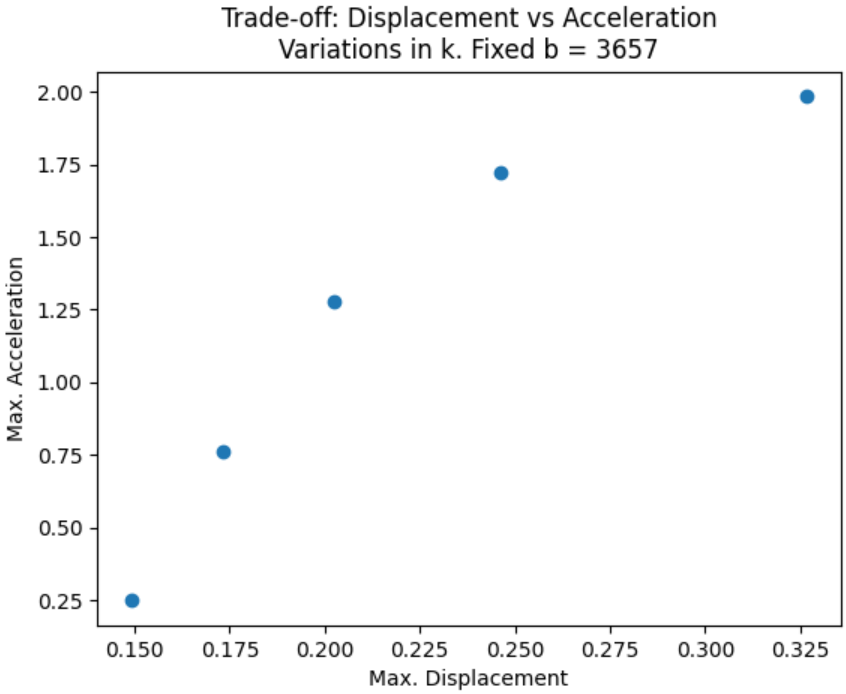


Figura 7: Región del espacio de entrada con comportamiento mono-objetivo para k variable.

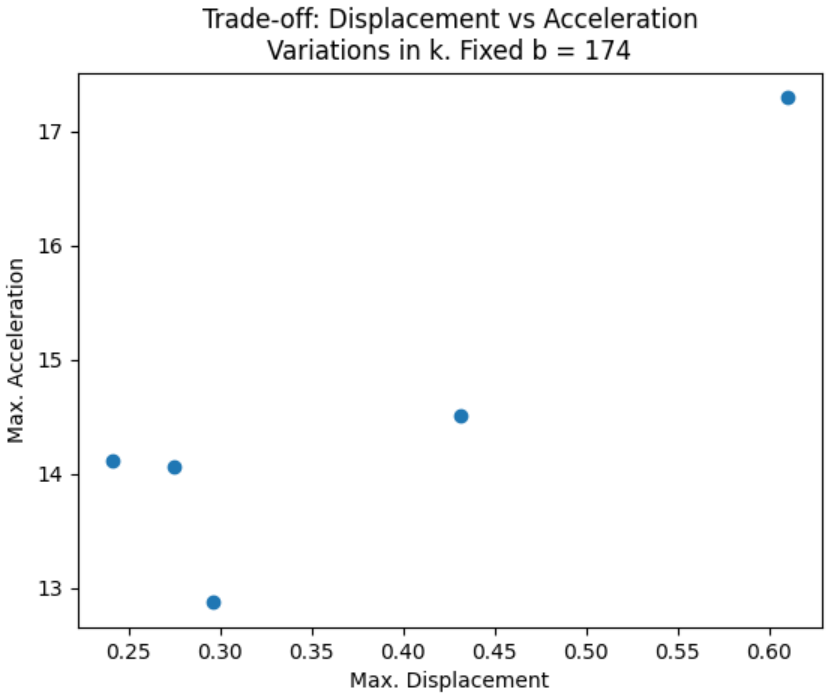


Figura 8: Región del espacio de entrada con comportamiento multi-objetivo para k variable.

# ANEXO C. ESTRUCTURA COMPLETA DE LOS ARCHIVOS DEL PROGRAMA

```
Program File Tree:
C:.
|   main.py
|
|-- modules
|       analysis.py
|       config.py
|       model.py
|       utils.py
|        __init__.py
|
|-- images
    |-- model_tests
    |       model_max_test.png
    |       model_peaks_test.png
    |       model_test.png
    |
    |-- pareto
    |       pareto_front.png
    |
    |-- problem_type_study
            point_1_model_b_var__k_3081.png
            point_1_model_k_var__b_1795.png
            point_2_model_b_var__k_32859.png
            point_2_model_k_var__b_1554.png
            ...
```

# ANEXO D. CÓDIGO MAIN.PY

```
##### IMPORTS #####

# Third party imports
import numpy as np
import matplotlib.pyplot as plt
from deap import creator, base, tools, algorithms

# Built-in imports
import random

# Local imports
from modules.config import *
import modules.utils as utils
import modules.model as model
import modules.analysis as analysis


##### FUNCTIONS DEFINITION #####

toolbox = base.Toolbox()


def evaluation_function(individual, verbose=False):
    """
    Function:
        Used to evaluate every individual.

        Finds the systems response for a given "k" and "b" values.
        Afterwards, it outputs the maximum magnitude of each
        response (displacement and acceleration).

    Parameters:
        individual (list): list containing the chromosomes "k" and "b".
        verbose (bool): Used to display information about each individuals
                        fitness in the terminal.

    Returns:
        fitness (list): Fitness of a given individual.

    """
    k = individual[0]
    b = individual[1]
    t, x_sol, _, a_sol = model.solve_model([k, b], t_max, t_samples, u)
    x_sol_max, _, a_sol_max, t_a_max = model.get_model_maxs(x_sol, _, a_sol, t)

    if verbose == True:
        print(
            f"Evaluando k={k:.2f}, b={b:.2f}, u={u:.2f} -> x_max={x_sol_max:.2f}, a_max={a_sol_max:.2f}"
        )

    return [x_sol_max, a_sol_max]


def check_bounds(lower_bounds, upper_bounds):
    """
    Function:
        Decorator that checks if the mutated individuals are within
        the allele space. If they are not, the specific chromosome is
        modified to enter said space.

    Parameters:
        lower_bounds (list): list containing the minimum values of "b" and "k".
        upper_bounds (list): list containing the maximum values of "b" and "k".

    Returns:
        function: A decorator that wraps functions in order to maintain
                individuals in the defined allele space.

    Source:
        Found in the official DEAP documentation webpage
        https://deap.readthedocs.io/en/master/tutorials/basic/part2.html

    """

    def decorator(func):
```

```
        def wrapper(*args, **kargs):
            offspring = func(*args, **kargs)
            for child in offspring:
                for i in range(len(child)):
                    if child[i] < lower_bounds[i]:
                        child[i] = lower_bounds[i]
                    elif child[i] > upper_bounds[i]:
                        child[i] = upper_bounds[i]
            return offspring

        return wrapper

    return decorator


def plot_pareto_front(
    solutions, img_path, annotate_inputs=False, show=False, verbose=False
):
    """
    Function:
        Generates a plot of the Pareto front from a list of non-dominated
        solutions.

    Parameters:
        solutions (list): A list of non-dominated individuals. Each individual
                          is a list [k, b].
        img_path (str): Path where the output image and directory will be
                        created.
        annotate_inputs (bool, optional): If True, annotates each point in the
                                          scatter plot with its (k, b)
                                          values. Default is False.
        show (bool, optional): If True, displays the plot in a window after
                               saving. Default is False.
        verbose (bool, optional): If True, prints details about each solution
                                  evaluated. Default is False.

    Returns:
        None
    """

    print("\n--> Getting the Pareto Front ...")

    # Manages directory where images will be created
    image_dir_name = "\\images\\pareto"
    image_path = str(img_path) + image_dir_name
    permission_status = utils.manage_directories_gen(image_dir_name)
    if permission_status == 1:
        return

    # Obtains the Pareto Front
    x_values = []
    a_values = []
    k_values = []
    b_values = []
    annotate_vals = []

    if verbose == True:
        print("- Individuos no dominados:")

    for individual in solutions:
        if verbose == True:
            print(individual)
        x_max, a_max = toolbox.evaluate(individual)
        x_values.append(x_max)  # maximum displacements
        a_values.append(a_max)  # maximum accelerations
        k_values.append(individual[0])  # k values
        b_values.append(individual[1])  # b values

    if annotate_inputs == True:
        annotate_vals = [k_values, b_values]

    utils.create_simple_graph(
        x_values=x_values,
        x_title="M ximo desplazamiento",
        y_values=a_values,
        y_title="M xima aceleraci n",
        annotate_values=annotate_vals,
```

```
        plot_type="scatter",
        show_plot=show,
        graph_title="Frente de Pareto (Optimizaci n Multiobjetivo)",
        image_name="pareto_front",
        image_path=image_path,
    )
    print("- Saved in: ", image_path)


def get_preferred_solution(solutions, preference, verbose):
    """
    Function:
        Selects the best individual from a list of solutions based on a weighted
        preference between displacement and acceleration.

    Parameters:
        solutions (list): A list of non-dominated individuals. Each individual
                          is a list [k, b].
        preference (float): A float between 0 and 1 indicating the preference for the displacement. The
    second objective
                            The acceleration is weighted as (1 - preference).
        verbose (bool): If True, print detailed information about the selected individual
                        and the applied preference.

    Returns:
        best_individual (list): The solution with the lowest weighted score.
        best_inputs (list): The chromosome values of selected individual [k, b].
        best_outputs (list): The evaluated objective values [x_max, a_max].
    """

    print("\n--> Getting Prefered Solution ...")
    # 'preference' will be applied to the first output of the evaluation_function
    # '(1 - preference)' will be applied to the second output of the evaluation_function

    best_score = float("inf")
    best_individual = None
    best_outputs = []
    best_inputs = []

    for individual in solutions:
        x_max, a_max = toolbox.evaluate(individual)
        score = preference * x_max + (1 - preference) * a_max
        if score < best_score:
            best_outputs = [x_max, a_max]
            best_score = score
            best_individual = individual

    best_inputs.append(best_individual[0])  # k
    best_inputs.append(best_individual[1])  # b

    if verbose == True:
        print(
            "- Using preference: ",
            int(preference * 100),
            r"% for displacement. ",
            int((1 - preference) * 100),
            r"% for acceleration.",
        )
        print("- Best individual:\n\tk = ", best_inputs[0], " b = ", best_inputs[1])
        print(
            "- Output:\n\tDisplacement: ",
            best_outputs[0],
            ". Acceleration: ",
            best_outputs[1],
        )

    return best_individual, best_inputs, best_outputs


def run_evolutionary_algorithm():
    """
    Function:
        Runs a multi-objective evolutionary algorithm using the DEAP library.

        This function sets up and executes an evolutionary process to optimize two
        conflicting objectives (displacement and acceleration). The function registers
        genetic operators, initializes the population, executes the evolution, and
```

```
    visualizes the results.

Parameters:
    None

Returns:
    None
"""

print("\n--> Running the Evolutionary Algorithm ...")

# Negative weights for minimization
pesos_fitness = (
    -1.0,
    -1.0,
)

# Fitness function definition
creator.create("fitness_function", base.Fitness, weights=pesos_fitness)
# Individual definition
creator.create("individual", list, fitness=creator.fitness_function, typecode="f")

# Alleles
toolbox.register(
    "k_stiffness", random.uniform, a=k_range[0], b=k_range[1]
)  # Stiffness 'k' gene
toolbox.register(
    "b_cushioning", random.uniform, a=b_range[0], b=b_range[1]
)  # Cushioning 'b' gene
# Individual generator
toolbox.register(
    "individual_generation",
    tools.initCycle,
    creator.individual,
    (toolbox.k_stiffness, toolbox.b_cushioning),
    n=1,
)
# Population generator
toolbox.register(
    "population", tools.initRepeat, list, toolbox.individual_generation
)
toolbox.register("evaluate", evaluation_function)
# Evolution operators
toolbox.register("select", tools.selNSGA2)
toolbox.register("mate", tools.cxBlend, alpha=alpha)
toolbox.register(
    "mutate", tools.mutGaussian, mu=mu, sigma=(sigma_k, sigma_b), indpb=0.2
)
toolbox.decorate(
    "mate", check_bounds([k_range[0], b_range[0]], [k_range[1], b_range[1]])
)
toolbox.decorate(
    "mutate", check_bounds([k_range[0], b_range[0]], [k_range[1], b_range[1]])
)

if debug == True:
    # Test for the population and individuals generation
    population_test = toolbox.population(n=popu_size)
    individual_test = toolbox.individual_generation()
    print("Individuo: ", individual_test)
    print("Ejemplo de poblacion: ", population_test)

# Statistics on the general fitness of the population
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("avg", np.mean)  # Generation 'Average'
stats.register("std", np.std)  # Individuals 'Standard Deviation'
stats.register("min", np.min)  # 'Min Fitness' of the generation
stats.register("max", np.max)  # 'Max Fitness' of the generation

hof = tools.ParetoFront()  # Hall of Fame
popu = toolbox.population(n=popu_size)  # Defines the initial population

# Runs the Evolutionary Algorithm
popu, logbook = algorithms.eaMuPlusLambda(
    population=popu,
    toolbox=toolbox,
    mu=parent_popu_size,
```

```
        lambda_=child_popu_size,
        cxpb=mate_chance,
        mutpb=mutate_chance,
        ngen=generations,
        stats=stats,
        halloffame=hof,
        verbose=verbose,
    )

    # Saves/shows the pareto front
    plot_pareto_front(
        solutions=hof,
        img_path=run_area,
        annotate_inputs=display_annotations,
        show=display_plots,
    )

    # Gets/prints the preferred solution
    get_preferred_solution(
        solutions=hof, preference=x_to_a_preference, verbose=True
    )
    print("")


##### MAIN EXECUTION #####

if debug == True:
    # Test for the Evaluation Function.
    # Individual with greater k and b values should have smaller
    # x_max and a_max values.
    print("Evaluaci n:", evaluation_function([30000, 1000]))
    print("Evaluaci n (peor caso):", evaluation_function([1000, 100]))

# Problem Analysis
model.test_model(
    input_vars=[64, 32],
    t_max=t_max,
    t_sample=t_samples,
    u=u,
    img_path=run_area,
    show_plots=display_plots,
)
analysis.ceteris_paribus(
    n_points=10, inputs_range_dict=kb_range_dict, scale_porcentage=0.2
)

# Evolutionary Algorithm
run_evolutionary_algorithm()
```

# ANEXO E. CÓDIGO CONFIG.PY

```
##### IMPORTS #####

# Third party imports

# Built-in imports
import os

# Local imports


##### GLOBAL VARIABLES #####

# General
run_area = os.getcwd()          # Current working directory.
verbose = False                 # To print detailed messages in terminal.
display_plots = False           # To display plots while running algorithm.
display_annotations = False     # To display annotations in plots.
debug = False                   # To print messages useful for debbuging.

# Mass-spring-damper model
k_range = [1000, 50000]         # Stiffness (k) [N/m] -> Range for vehicle suspension.
b_range = [100, 5000]           # Cushioning (b) [Ns/m] -> Range for vehicle suspension.
kb_range_dict = {"k": k_range,
                 "b": b_range}  # Dict with k and b ranges.
car_mass = 1000                 # Car mass [kg].
m = car_mass/4                  # Mass for each suspension.
u = 5000                        # External force (step) - Perturbation.
x_units = "m"                   # Units for the output displacement (x).
v_units = "m/s"                 # Units for the output velocity (v).
a_units = "m/s^2"               # Units for the output acceleration (a).
t_max = 100                     # Maximum time for simulation.
t_samples = 500                 # Number of time samples.


##### HYPERPARAMETERS #####

# General
popu_size = 100         # Population size.
generations = 50        # Number of iterations.
# Mating
alpha = 0.5             # Blend factor.
mate_chance = 0.75      # Chance that a pair of parents will actually mate.
parent_popu_size = 25   # Number of individuals selected to become parents.
child_popu_size = 25    # Number of offspring that are produced during mating.
# Mutation
mutate_chance = 0.2     # Probability that an individual of gene will mutate.
mu = 0.0                #  Mean of the Gaussian distribution. Zero -> symmetric perturbation.
sigma_k = (k_range[1] - k_range[0]) * 0.05  # 5% of the k range.
sigma_b = (b_range[1] - b_range[0]) * 0.05  # 5% of the b range.
# Pareto
x_to_a_preference = 0.7 # Used to select the 'preferred solution'
```

# ANEXO F. CÓDIGO MODEL.PY

```
##### IMPORTS #####

# Third party imports
import numpy as np
from scipy.integrate import odeint
from scipy.signal import find_peaks

# Built-in imports

# Local imports
from .utils import *
from .config import *


##### FUNCTIONS DEFINITION #####

def model(S, t, k, b, u):
    """
    Function:
        Define the differential equations for a mass-spring-damper system.

    Parameters:
        S (list): State vector / Initial conditions [position x, velocity v].
        t (float): Time variable (not used explicitly but required by odeint).
        k (float): Spring constant.
        b (float): Damping coefficient.
        u (float): External force applied.

    Returns:
        List: Derivatives [dx/dt, dv/dt].
    """

    x, v = S
    return [v, (-b * v - k * x + u) / m]


def solve_model(input_vars, t_max, t_samples, u):
    """
    Function:
        Solve the mass-spring-damper system using the given parameters.

    Parameters:
        input_vars (list): List with constants: k (stiffness) and b (cushioning).
        t_max (float): Maximum time for simulation.
        t_samples (int): Number of time samples.
        u (float): External force applied.

    Returns:
        List: time (t), displacement (x), velocity (v), and acceleration a.
    """

    k = input_vars[0]
    b = input_vars[1]
    S_0 = (0, 0)
    t = np.linspace(0, t_max, t_samples)
    solution = odeint(model, y0=S_0, t=t, args=(k, b, u))

    x_sol = solution.T[0]
    v_sol = solution.T[1]
    a_sol = (-b * v_sol - k * x_sol + u) / m

    return t, x_sol, v_sol, a_sol


def get_model_maxs(x_sol, v_sol, a_sol, t):
    """
    Function:
        Obtain maximum absolute values for displacement, velocity, and acceleration.

    Parameters:
        x_sol (list): Max displacement over time.
        v_sol (list): Max velocity over time.
        a_sol (list): max acceleration over time.
        t (float): Time at max acceleration.
```

```
    Returns:
        List: [max abs displacement, max abs velocity, max abs acceleration, time of max acceleration].
    """

    sol_maxs_list = []

    # Abs max for displacement and velocity
    for sol in [x_sol, v_sol]:
        sol_max = np.max(np.abs(sol))
        sol_maxs_list.append(sol_max)

    # Detect local max y min for the acceleration
    peaks_idx, _ = find_peaks(a_sol) # Local max
    valleys_idx, _ = find_peaks(-a_sol)  # Local min
    peaks = a_sol[peaks_idx]
    valleys = a_sol[valleys_idx]

    if len(peaks):
        max_peak = np.max(peaks)
    else:
        max_peak = 0

    if len(valleys):
        min_valley = np.min(valleys)
    else:
        min_valley = 0

    if max_peak > abs(min_valley):
        max_peak_valley = max_peak
        idx = np.where(a_sol == max_peak)
    else:
        max_peak_valley = abs(min_valley)
        # max_peak_valley = min_valley
        idx = np.where(a_sol == min_valley)

    t_a_max = t[idx]
    sol_maxs_list.append(max_peak_valley)
    sol_maxs_list.append(t_a_max)

    return sol_maxs_list


def test_model(input_vars, t_max, t_sample, u, img_path, show_plots):
    """
    Function:
        Test the mass-spring-damper model and generate plots for:
        x(t), v(t), a(t),
        peaks/valleys for the acceleration and
        max absolute acceleration.

    Parameters:
        input_vars (list): input_vars (list): List with constants: k (stiffness) and b (cushioning).
        t_max (float): Maximum time for simulation.
        t_sample (int): Number of time samples.
        u (float): External force applied.
        img_path (str): Path to save the plots.
        show_plots (bool): Whether to display plots while running.

    Returns:
        None
    """

    print("--> Testing model ...")

    # Manages directory where images will be created
    image_dir_name = "\\images\\model_tests"
    image_path = img_path + image_dir_name
    permission_status = manage_directories_gen(image_dir_name)
    if permission_status == 1:
        return

    # Plot of x, v, a vs t
    t, x, v, a = solve_model(input_vars, t_max, t_sample, u)
    _, _, a_max, t_a_max = get_model_maxs(x, v, a, t)

    test_dict = {
        "x: Desplazamiento " + x_units: x,
```

```
        "v: Velocidad " + v_units: v,
        "a: Aceleraci n " + a_units: a,
    }
    create_multi_y_graph(
        t,
        "Tiempo (s)",
        test_dict,
        "plot",
        show_plots,
        "Modelo Masa-Resorte-Amortiguador",
        "model_test",
        image_path,
    )

    # To show peaks and valleys of the acceleration
    peaks, _ = find_peaks(a)
    valleys, _ = find_peaks(-a)

    plt.plot(t, a, label="a(t)")
    plt.plot(t[peaks], a[peaks], "x", label="Picos")
    plt.plot(t[valleys], a[valleys], "o", label="Valles")
    plt.legend()
    plt.xlabel("Tiempo (s)")
    plt.ylabel("Aceleraci n " + a_units)
    plt.title("Picos y valles de la aceleraci n")
    plt.grid(True)
    plt.savefig(f"{image_path}/model_peaks_test")
    if show_plots == True:
        plt.show()
    plt.close()

    # To show max abs peak for the acceleration
    plt.plot(t, a, label="a(t)")
    plt.plot(t_a_max, a_max, "o", label="Max Abs Seleccionado")
    plt.legend()
    plt.xlabel("Tiempo (s)")
    plt.ylabel("Aceleraci n " + a_units)
    plt.title("Maximo absoluto de la aceleraci n")
    plt.grid(True)
    plt.savefig(f"{image_path}/model_max_test")
    if show_plots == True:
        plt.show()
    plt.close()
```

# ANEXO G. CÓDIGO ANALYSIS.PY

```
##### IMPORTS #####

# Third party imports
import numpy as np

# Built-in imports
import random

# Local imports
from .config import *
from .utils import *
from .model import *


##### FUNCTIONS DEFINITION #####

def ceteris_paribus(n_points, inputs_range_dict, scale_porcentage):
    """
    Function:
        Performs a sensitivity study using the Ceteris Paribus method. And creates
        scatter plots of the trade-offs observed due to these variations.

    Parameters:
        n_points (int): Number of random input points to generate and analyze.
        inputs_range_dict (dict): A dictionary where keys are input variable
                                  names (str) and values are tuples
                                  with the range for each input.
        scale_porcentage (float): Percentage that determines how far each input
                                  is varied around its reference value.

    Returns:
        None
    """

    print("\n--> Running Sensibility Study (Ceteris Paribus) ...")

    # Manages directory where images will be created
    image_dir_name = "\\images\\problem_type_study"
    image_path = str(run_area) + image_dir_name
    permission_status = manage_directories_gen(image_dir_name)
    if permission_status == 1:
        return

    # Starting the sensibility study
    for i in range(n_points):
        point_num = i + 1
        # print("----- Point: ", point_num, "-----")

        # Random point generation
        random_point = {}
        input_ref_dict = {}
        for in_name, in_range in inputs_range_dict.items():
            random_num = random.randint(in_range[0], in_range[1])
            ref_num = np.median(np.arange(in_range[0], in_range[1] + 1))
            random_point[in_name] = random_num
            input_ref_dict[in_name] = ref_num
        # print("Random point: ", random_point)

        # Loop through each input
        for in_name, in_ref in input_ref_dict.items():
            # print("--- Inputs: ", in_name, "---")
            current_in_values = [
                in_ref * (1 - scale_porcentage * 2),
                in_ref * (1 - scale_porcentage),
                in_ref,
                in_ref * (1 + scale_porcentage),
                in_ref * (1 + scale_porcentage * 2),
            ]
            random_point_aux = random_point.copy()
            # in_idx = list(input_ref_dict.keys()).index(in_name)

            ### Problem-specific functions to get the 'y' values ###
            y_values_dict = {"Displacement": [], "Acceleration": []}

            ########################################################
```

```
        # Get solutions ('y' values) for current input variations
        for value in current_in_values:
            random_point_aux[in_name] = int(value)
            # print("- Aux point: ", random_point_aux, "-")

            ### Problem-specific functions to get the 'y' values ###
            random_point_aux_list = []
            for in_var_name, in_point_value in random_point_aux.items():
                random_point_aux_list.append(in_point_value)
            t, x, _, a = solve_model(random_point_aux_list, 20, 50, u)

            # model_dict = {"Displacement" : [], "Acceleration" : []}
            # model_dict["Displacement"] = x
            # model_dict["Acceleration"] = a
            # graph_title = "Complete Model"
            # image_name = "point_" + str(point_num) + "_complete_model_k_" + str(random_point_aux_list
[0]) + "_b_" + str(random_point_aux_list[1])
            # image_path = str(run_area) + "\\problem_type_study\\"
            # create_vertical_graphs(t, "time", model_dict, graph_title, image_name, image_path, "red")

            x_max, _, a_max, t_a_max = get_model_maxs(x, _, a, t)
            y_values_dict["Displacement"].append(x_max)
            y_values_dict["Acceleration"].append(a_max)
            ########################################################

        # Create graphs
        graph_title = (
            "Trade-off: Displacement vs Acceleration\nVariations in " + in_name
        )
        image_name = "point_" + str(point_num) + "_model_" + in_name + "_var_"
        for in_var_name, in_point_value in random_point_aux.items():
            if in_var_name != in_name:
                graph_title = (
                    graph_title
                    + ". Fixed "
                    + in_var_name
                    + " = "
                    + str(in_point_value)
                )
                image_name = (
                    image_name + "_" + in_var_name + "_" + str(in_point_value)
                )
        # Creates and adds the image to the corresponding directory
        create_simple_graph(
            x_values=y_values_dict["Displacement"],
            x_title="Max. Displacement",
            y_values=y_values_dict["Acceleration"],
            y_title="Max. Acceleration",
            annotate_values=[],
            plot_type="scatter",
            show_plot=False,
            graph_title=graph_title,
            image_name=image_name,
            image_path=image_path,
        )
```

# ANEXO H. CÓDIGO UTILS.PY

```
##### IMPORTS #####

# Third party imports
import matplotlib.pyplot as plt

# Built-in imports
import stat
import shutil

# Local imports
from .config import *


##### FUNCTIONS DEFINITION #####

def print_files_tree():
    """
    Function:
        Prints the hierarchical tree structure of the main
        program files and modules.Includes images files.
        Used to explain the code in the written report.

    Parameters:
        None

    Returns:
        None
    """
    print ("""
Program File Tree:
C:.
      main.py

      modules
          analysis.py
          config.py
          model.py
          utils.py
           __init__.py

      images
          model_tests
              model_max_test.png
              model_peaks_test.png
              model_test.png

          pareto
              pareto_front.png

          problem_type_study
            point_1_model_b_var__k_3081.png
            point_1_model_k_var__b_1795.png
            point_2_model_b_var__k_32859.png
            point_2_model_k_var__b_1554.png
            ...
       """)


def print_files_tree_short():
    """
    Function:
        Prints the hierarchical tree structure of the main
        program files and modules. Includes only code files.
        Used to explain the code in the written report.

    Parameters:
        None

    Returns:
        None
    """

    print ("""
Program File Tree:
C:.
```

```
      main.py

      modules
        analysis.py
        config.py
        model.py
        utils.py
        __init__.py
    """)


def manage_directories_gen_old(image_dir_name):
    """
    Function:
        Manages image directories by moving the existing directory to a
        backup (_old) and creating a new directory with the original name.

    Parameters:
        image_dir_name (str): Name of the image to generate.

    Returns:
        int: 0 if successful, 1 if PermissionError occurrs.
    """

    image_path = str(run_area) + image_dir_name
    try:
        if os.path.isdir(image_path):
            old_path = str(run_area) + image_dir_name + "_old"
            if os.path.isdir(old_path):
                os.remove(old_path)
            os.replace(image_path, old_path)
            os.chmod(old_path, stat.S_IWRITE)
            os.makedirs(image_path)
            os.chmod(image_path, stat.S_IWRITE)
        else:
            os.makedirs(image_path)
            os.chmod(image_path, stat.S_IWRITE)
        return 0
    except PermissionError:
        print(
            "-E-: 'Access Denied' while trying to modify directories. Files not created."
        )
        print(f"-I-: Possible affected directories:\n\t{image_path}\n\t{old_path}")
        print(
            "-I-: Solution: Execute program as administrator or remove the affected directories.\n"
        )
        return 1


def manage_directories_gen(image_dir_name):
    """
    Function:
        Manages image directories by moving the existing directory to a
        backup (_old) and creating a new directory with the original name.

    Parameters:
        image_dir_name (str): Name of the image to generate.

    Returns:
        None
    """

    image_path = str(run_area) + image_dir_name
    if os.path.exists(image_path):
        old_path = str(run_area) + image_dir_name + "_old"
        if os.path.exists(old_path):
            shutil.rmtree(old_path)
        os.replace(image_path, old_path)
        # shutil.rmtree(image_path)
    os.makedirs(image_path)


def create_simple_graph(
    x_values,
    x_title,
    y_values,
    y_title,
```

```
    annotate_values,
    plot_type,
    show_plot,
    graph_title,
    image_name,
    image_path,
):
    """
    Function:
        Creates and saves a 2D graph (scatter or line).

    Parameters:
        x_values (list): Data points for the x-axis.
        x_title (str): Label for the x-axis.
        y_values (list): Data points for the y-axis.
        y_title (str): Label for the y-axis.
        annotate_values (list): Optional. List of two lists
                                [k_values, b_values] for annotating each
                                (k, b) point.
        plot_type (str): Type of plot to create. Valid options: 'scatter',
                         others default to line plot.
        show_plot (bool): Shows the plot while running if True.
        graph_title (str): Title of the graph.
        image_name (str): Name of the image to generate.
        image_path (str): Path to save the image.

    Returns:
        None
    """

    plt.figure()
    match plot_type:
        case "scatter":
            plt.scatter(x_values, y_values)
        case _:
            plt.plot(x_values, y_values)

    # Annotate each point with k and b values
    if len(annotate_values) == 2:
        k_values = annotate_values[0]
        b_values = annotate_values[1]
        for i in range(len(x_values)):
            plt.annotate(
                f"k={k_values[i]:.1f}\nb={b_values[i]:.1f}",
                (x_values[i], y_values[i]),
                textcoords="offset points",
                xytext=(5, 5),
                ha="left",
                fontsize=8,
            )

    plt.xlabel(x_title)
    plt.ylabel(y_title)
    plt.title(graph_title)
    plt.savefig(f"{image_path}/{image_name}")
    if show_plot == True:
        plt.show()
    plt.close()


def create_multi_y_graph(
    x_values,
    x_title,
    y_values_dict,
    plot_type,
    show_plot,
    graph_title,
    image_name,
    image_path,
):
    """
    Function:
        Creates and saves a 2D graph with multiple y-axis functions
        (scatter or line).

    Parameters:
        x_values (list or array): Data points for the x-axis.
```

```
        x_title (str): Label for the x-axis.
        y_values_dict (dict): Dictionary where keys are labels and values are
                              lists of y data.
        plot_type (str): Type of plot to create. Valid options: 'scatter',
                         others default to line plot.
        show_plot (bool): Shows the plot while running if True.
        graph_title (str): Title of the graph.
        image_name (str): Name of the image to generate.
        image_path (str): Path to save the image.

    Returns:
        None
    """

    plt.figure()
    for y_title, y_values in y_values_dict.items():
        match plot_type:
            case "scatter":
                plt.scatter(x_values, y_values, label=y_title)
            case "plot":
                plt.plot(x_values, y_values, label=y_title)
            case _:
                print(
                    "-W-: For 'create_multi_y_graph' attribute 'plot_type' use: scatter or plot"
                )
    plt.legend(loc="upper right")
    plt.xlabel(x_title)
    plt.title(graph_title)
    plt.savefig(f"{image_path}/{image_name}")
    if show_plot == True:
        plt.show()
    plt.close()


def create_vertical_graphs(
    x_values,
    x_title,
    y_values_dict,
    show_plot,
    graph_title,
    image_name,
    image_path,
    color,
):
    """
    Function:
        Creates and saves a subplots arranged in vertical way
        Subplots share the same x-axis.

    Parameters:
        x_values (list or array): Data points for the x-axis.
        x_title (str): Label for the x-axis.
        y_values_dict (dict): Dictionary where keys are labels and values are
                              lists of y data.
        show_plot (bool): Shows the plot while running if True.
        graph_title (str): Title of the graph.
        image_name (str): Name of the image to generate.
        image_path (str): Path to save the image.
        color (str): Color for the plot lines.

    Returns:
        None
    """

    plt.figure()
    count = 1
    for y_title, y_values in y_values_dict.items():
        plt.subplot(2, 1, count)
        plt.plot(x_values, y_values, label=y_title, color=color)
        count += 1
    plt.legend(loc="upper right")
    plt.xlabel(x_title)
    plt.title(graph_title)
    plt.savefig(f"{image_path}/{image_name}")
    if show_plot == True:
        plt.show()
    plt.close()
```