Optimización de Ascenso de Aeronaves Familia AIRBUS A320

Autor: Marco A. Erazo, Dir: Leonardo A. Pachon

25 de octubre de 2024

Contents

Metodología y Pasos Seguidos

El proceso de optimización desarrollado para el problema del ascenso de aeronaves sigue una secuencia rigurosa de pasos que permiten abordar la complejidad inherente a las ecuaciones no lineales y las múltiples restricciones involucradas. A continuación, se describe los pasos seguidos comenzando con la manipulación de ecuaciones hasta la implementación.

1 Variables de Decisión

Las variables que se buscan optimizar para cada punto discretizado (i) de la trayectoria son:

- v_i: Velocidad aerodinámica real (TAS).
- γ_i: Ángulo de ascenso.
- m_i: Masa de la aeronave.
- t_i: Tiempo transcurrido.
- s_i: Distancia recorrida.
- Cz_i: Coeficiente de sustentación.
- λ_i: Fracción del empuje máximo de ascenso (MCL).

Estas variables, en conjunto, definen el estado de la aeronave y su trayectoria en cada punto discretizado del ascenso.

Para determinar si el problema de optimización de la trayectoria de ascenso de una aeronave, tal como se describe, matemáticamente, efectivamente involucra 364 variables, se llevó a cabo un análisis del Jacobiano de las ecuaciones de actualización de estado. Dado que la discretización de la trayectoria se realiza en $N = 53$ puntos, cada uno con 7 variables (velocidad, ángulo de ascenso, masa, tiempo, distancia, coeficiente de sustentación y fracción de empuje), el número total de variables asciende a $7 * (N - 1) = 364$. El Jacobiano, en este contexto, es una matriz que contiene las derivadas parciales de las ecuaciones de actualización de estado con respecto a cada una de las 364 variables. Este análisis se centra en las ecuaciones de actualización de estado porque son las que determinan la factibilidad de una trayectoria, relacionando las variables en cada punto discretizado.

El Análisis de Independencia Diferencial se realiza numéricamente, evaluando las derivadas parciales en un punto específico del espacio de soluciones. Esto se debe a la complejidad de las ecuaciones, que dificulta un análisis simbólico completo. La evaluación numérica permite obtener una aproximación del rango del Jacobiano, que es un indicador del número de ecuaciones de actualización de estado linealmente independientes. Un rango igual a 364 confirmaría que todas las ecuaciones son independientes y que, por lo tanto, el problema está bien definido con 364 variables. Un rango menor a 364 sugeriría la existencia de dependencias lineales entre las ecuaciones, lo que podría simplificar el problema. Un rango mayor a 364, indicaría un sistema sobredeterminado, posiblemente inconsistente, este proceso fué hecho, programado y ejecutado en python.

1.1 Análisis de Independencia Diferencial

El análisis y su implementación en código python se divide en varias secciones:

1. Definición de variables simbólicas y constantes.
2. Definición de funciones auxiliares (densidad del aire y empuje máximo).
3. Definición de las ecuaciones de actualización de estado.
4. Implementación de funciones para calcular derivadas parciales y el Jacobiano.
5. Definición de valores numéricos para las variables y constantes.
6. Cálculo del rango del Jacobiano y el número de variables independientes.

A continuación se muestra la fracción del código:

```python
import numpy as np
import sympy

# Define symbolic variables.  Incluyendo constantes
i = sympy.Symbol('i', integer=True)
N = 53  # Define N como un entero (valor de prueba)
Zp_I = sympy.Symbol('Zp_I')
Zp_F = sympy.Symbol('Zp_F')
v = sympy.Function('v')
```

```
gamma = sympy.Function('gamma')
m = sympy.Function('m')
t = sympy.Function('t')
s = sympy.Function('s')
F_N_MCL = sympy.Function('F_N_MCL')
rho = sympy.Function('rho')
Cx_0 = sympy.Symbol('Cx_0')
k = sympy.Symbol('k')
Cz = sympy.Function('Cz')
S_REF = sympy.Symbol('S_REF')
g_0 = sympy.Symbol('g_0')
lambda_ = sympy.Function('lambda_') # evitar conflicto con palabra clave lambda
eta = sympy.Symbol('eta')
R = sympy.Symbol('R')
L_z = sympy.Symbol('L_z') # Mantiene consistencia
Ts_0 = sympy.Symbol('Ts_0')
rho_0 = sympy.Symbol('rho_0')
alpha_0 = sympy.Symbol('alpha_0')
CI = sympy.Symbol('CI')
M_CRZ = sympy.Symbol('M_CRZ')
s_F = sympy.Symbol('s_F')
CAS_I = sympy.Symbol('CAS_I')
m_I = sympy.Symbol('m_I')
```

1.2 Variables Simbólicas y Constantes

- Se importan las bibliotecas NumPy (para operaciones numéricas) y SymPy (para cálculo simbólico).
- Se definen variables simbólicas utilizando SymPy, incluyendo la variable de índice ‘i‘, el número de puntos de discretización ‘N‘, y las variables de estado y constantes del problema.
- Se utiliza ‘sympy.Function‘ para definir variables que son funciones de ‘i‘, como la velocidad ‘v(i)‘, el ángulo de ascenso ‘gamma(i)‘, etc.
- Se utiliza ‘sympy.Symbol‘ para definir constantes y variables que no son funciones de ‘i‘.
- Se renombra la variable ‘lambda‘ a ‘lambda_‘ para evitar conflictos con la palabra clave ‘lambda‘ de Python.

1.3 Funciones Auxiliares

- Se define ‘Zp(i)‘ como una función lambda que calcula la altitud en el punto ‘i‘ utilizando una interpolación lineal entre la altitud inicial ‘Zp_I‘ y la altitud final ‘Zp_F‘.
- Se define ‘rho(i)‘ como una función lambda que calcula la densidad del aire en el punto ‘i‘ utilizando la ecuación de la atmósfera estándar internacional.
- Se define ‘F_N_MCL(i)‘ como una función lambda que calcula el empuje máximo disponible en el punto ‘i‘ (asumiendo una dependencia lineal con la altitud).

```
    Zp = sympy.Lambda(i, Zp_I + i * (Zp_F - Zp_I) / (N - 1))

    # Define rho(Zp(i))
    rho = sympy.Lambda(i, rho_0*((Ts_0 + L_z*Zp(i))/Ts_0)**(alpha_0 - 1))

    # Define F_N_MCL(Zp(i))
    F_N_MCL = sympy.Lambda(i, 140000 - 2.53*Zp(i))
```

1.4 Ecuaciones Fundamentales de Recurrencia

- Se definen las ecuaciones de recurrencia ‘g_v‘, ‘g_gamma‘, ‘g_m‘, ‘g_t‘, y ‘g_s‘. Estas ecuaciones describen cómo cambian la velocidad, el ángulo de ascenso, la masa, el tiempo y la distancia entre dos puntos consecutivos de la trayectoria discretizada.

```
# Se define las ecuaciones usando directamente Zp(i)
g_v = (v(i+1) - v(i))/(Zp(i+1) - Zp(i)) - \
    (1/2) * ( (lambda_(i+1) * F_N_MCL(i+1)) / (m(i+1) * v(i+1) * sympy.sin(gamma(i+1))) - \
             ( (1/2) * rho(i+1) * v(i+1) * S_REF * (Cx_0 + k*Cz(i+1)**2) ) / (m(i+1) * sympy.sin(gamma(i+1))) - \
              g_0 / v(i+1) + \
             (lambda_(i) * F_N_MCL(i)) / (m(i) * v(i) * sympy.sin(gamma(i))) - \
             ( (1/2) * rho(i) * v(i) * S_REF * (Cx_0 + k*Cz(i)**2) ) / (m(i) * sympy.sin(gamma(i))) - \
              g_0 / v(i) )

g_gamma = (gamma(i+1) - gamma(i))/(Zp(i+1) - Zp(i)) - \
    (1/2) * ( ( (1/2)*rho(i+1)*S_REF*Cz(i+1) ) / (m(i+1) * sympy.sin(gamma(i+1))) - \
              g_0 / (v(i+1)**2 * sympy.tan(gamma(i+1))) + \
              ( (1/2)*rho(i)*S_REF*Cz(i) ) / (m(i) * sympy.sin(gamma(i))) - \
              g_0 / (v(i)**2 * sympy.tan(gamma(i))) )

g_m = (m(i+1) - m(i))/(Zp(i+1) - Zp(i)) + \
      (1/2) * eta * ( (lambda_(i+1) * F_N_MCL(i+1)) / (v(i+1) * sympy.sin(gamma(i+1))) + \
                      (lambda_(i) * F_N_MCL(i)) / (v(i) * sympy.sin(gamma(i))) )

g_t = (t(i+1) - t(i))/(Zp(i+1) - Zp(i)) - \
      (1/2) * ( 1/(v(i+1)*sympy.sin(gamma(i+1))) + 1/(v(i)*sympy.sin(gamma(i))) )

g_s = (s(i+1) - s(i))/(Zp(i+1) - Zp(i)) - \
      (1/2) * ( 1/sympy.tan(gamma(i+1)) + 1/sympy.tan(gamma(i)) )
```

1.5 Implementación de Funciones para Calcular Derivadas Parciales y el Jacobiano

- La función 'derivada_parcial' calcula la derivada parcial de una función 'funcion' con respecto a una variable 'variable' en los puntos 'i_funcion' e 'i_variable'. - Matemáticamente, esto se representa como:

$$\frac{\partial \text{funcion}(i_{\text{funcion}})}{\partial \text{variable}(i_{\text{variable}})}$$

  Por ejemplo, si 'funcion' es 'g_v' y 'variable' es 'v', con 'i_funcion = 2' e 'i_variable = 3', la función calcularía:

$$\frac{\partial g_v(2)}{\partial v(3)}$$

- La función 'calcular_jacobiano' calcula el Jacobiano de las ecuaciones de actualización de estado.

  El Jacobiano es una matriz que contiene las derivadas parciales de las ecuaciones de actualización de estado con respecto a cada una de las variables.

  La estructura del Jacobiano se puede representar como:

$$J = \begin{bmatrix} \frac{\partial g_v(0)}{\partial v(1)} & \frac{\partial g_v(0)}{\partial v(2)} & \cdots & \frac{\partial g_v(0)}{\partial \gamma(1)} & \cdots & \frac{\partial g_v(0)}{\partial \lambda(N-1)} \\ \frac{\partial g_v(1)}{\partial v(1)} & \frac{\partial g_v(1)}{\partial v(2)} & \cdots & \frac{\partial g_v(1)}{\partial \gamma(1)} & \cdots & \frac{\partial g_v(1)}{\partial \lambda(N-1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \frac{\partial g_\gamma(0)}{\partial v(1)} & \frac{\partial g_\gamma(0)}{\partial v(2)} & \cdots & \frac{\partial g_\gamma(0)}{\partial \gamma(1)} & \cdots & \frac{\partial g_\gamma(0)}{\partial \lambda(N-1)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \frac{\partial g_s(N-2)}{\partial v(1)} & \frac{\partial g_s(N-2)}{\partial v(2)} & \cdots & \frac{\partial g_s(N-2)}{\partial \gamma(1)} & \cdots & \frac{\partial g_s(N-2)}{\partial \lambda(N-1)} \end{bmatrix} \tag{1}$$

  Donde: $g_v, g_\gamma, g_m, g_t, g_s$ son las ecuaciones de actualización de estado.

  $v, \gamma, m, t, s, Cz, \lambda$ son las variables. N es el número de puntos de discretización.

- Se itera sobre todas las ecuaciones y variables para calcular las derivadas parciales y construir el Jacobiano.
- Se asegura que los indices de las variables sean correctos al calcular las derivadas parciales (se suma 1 a 'j_variable').

  Esto se debe a que las ecuaciones de actualización de estado se definen en términos del subindice iniciando desde 0 mientras que el índice 'j_variable' comienza en 1.

```python
def derivada_parcial(funcion, variable, i_funcion, i_variable):
    """
    Calcula la derivada parcial de una funcion respecto a una variable con subindices especificos.
    """
    derivada = funcion.subs(i, i_funcion).diff(variable(i_variable)).simplify().trigsimp()
    return derivada


def calcular_jacobiano(N):
    """Calcula el Jacobiano de las restricciones"""

    funciones = [g_v, g_gamma, g_m, g_t, g_s]
    variables = [v, gamma, m, t, s, Cz, lambda_]

    num_funciones = len(funciones) * (N - 1)
    num_variables = len(variables) * (N - 1)

    jacobiano = sympy.zeros(num_funciones, num_variables)

    for i_funcion in range(N - 1):
        for j_variable in range(N - 1):  # j_variable va de 0 a N-2, representando indices de 1 a N-1
            for k_func, func in enumerate(funciones):
                for l_var, var in enumerate(variables):
                    #  Se suma 1 a j_variable al derivar para representar indices de 1 a N-1
                    derivada = derivada_parcial(func, var, i_funcion, j_variable + 1)  # asegura
                    jacobiano[i_funcion * len(funciones) + k_func, j_variable * len(variables) + l_var] = derivada

    #return jacobiano
    return jacobiano, num_variables
```

1.6 Definición de Valores Numéricos para las Variables y Constantes

- Se define un diccionario 'valores_numericos' que contiene valores numéricos para las constantes del problema.
- Se utiliza la función 'generar_valores_numericos' para generar valores de ejemplo para las variables en cada punto de la trayectoria.
- Se calculan las constantes con subíndice 0 (valores iniciales) utilizando las expresiones dadas en el documento original de AIRBUS.
- Se define un diccionario 'valores_numericos_0' que contiene los valores iniciales de las variables.
- Se realizan las conversiones de unidades necesarias (pies a metros, nudos a metros por segundo).

```python
# Valores numericos (con unidades correctas y valores para N=5)

valores_numericos = {
    Zp_I: 10000 * 0.3048,  # Convertir ft a m
    Zp_F: 36000 * 0.3048,  # Convertir ft a m
    Cx_0: 0.014,
    k: 0.09,
    S_REF: 120,
    g_0: 9.80665,
```

```
    eta: 0.06 / 3600,  # kg/(N*s)
    R: 287.05287,
    L_z: -0.0065,
    Ts_0: 288.15,
    rho_0: 1.225,
    alpha_0: -9.80665 / (287.05287 * -0.0065),  # Valor calculado, corregir signo
    m_I: 60000,  # kg
    CAS_I: 250 * 0.514444,  # Convertir kt a m/s
    CI: 30 / 60, # kg/s
    M_CRZ : 0.8,
    s_F : 400000
}

def generar_valores_numericos(N):
    """Genera un diccionario con valores numericos de ejemplo para un N dado."""

    valores_numericos = {
        Zp_I: 10000 * 0.3048,  # Convertir ft a m
        Zp_F: 36000 * 0.3048,  # Convertir ft a m
        Cx_0: 0.014,
        k: 0.09,
        S_REF: 120,
        g_0: 9.80665,
        eta: 0.06 / 3600,  # kg/(N*s)
        R: 287.05287,
        L_z: -0.0065,
        Ts_0: 288.15,
        rho_0: 1.225,
        alpha_0: -9.80665 / (287.05287 * -0.0065),
        m_I: 60000,  # kg
        CAS_I: 250 * 0.514444,  # Convertir kt a m/s
        CI: 30 / 60,
        M_CRZ: 0.8,
        s_F: 400000
    }

    # Generar valores para las variables
    for i in range(1, N):
        velo = [136.723, 139.576 ...]
        valores_numericos[v(i)] = velo[i-1]
        gammas = [2.357, 1.463 ...]*
        valores_numericos[gamma(i)] = gammas[i-1]  # Ejemplo
        eme = [59987.32, 59948.48 ...]
        valores_numericos[m(i)] = eme[i-1]  # Ejemplo
        te = [19.878, 54.137...]
        valores_numericos[t(i)] = te[i-1]  # Ejemplo
        ese = [2791.07, 7532.51...]
        valores_numericos[s(i)] = ese[i-1]  # Ejemplo
        Czz = [0.565, 0.581...]
        valores_numericos[Cz(i)] = Czz[i-1]  # Ejemplo
        lamdas = [0.0033, 0.988...]
        valores_numericos[lambda_(i)] = lamdas[i-1]  # Ejemplo
    return valores_numericos

valores_numericos = generar_valores_numericos(N)

# Calcular constantes con subindice 0 (con unidades consistentes)
v0 = sympy.sqrt(7 * valores_numericos[R] * (valores_numericos[Ts_0] + valores_numericos[
    L_z] * valores_numericos[Zp_I]) *
             (((1 + valores_numericos[CAS_I]**2 / (7 * valores_numericos[R] *
    valores_numericos[Ts_0]))**(3.5) - 1) *
              (valores_numericos[Ts_0] / (valores_numericos[Ts_0] + valores_numericos[
    L_z] * valores_numericos[Zp_I]))**(-valores_numericos[alpha_0]) + 1)**(1/3.5) - 1)


rho0_val = rho(0).subs(valores_numericos).evalf()


Cz0 = (valores_numericos[m_I] * valores_numericos[g_0]) / (0.5 * rho0_val * v0**2 *
    valores_numericos[S_REF])


gamma0 = sympy.asin((F_N_MCL(0).subs(valores_numericos).evalf() - 0.5 * rho0_val * v0**2
```

```
     * valores_numericos[S_REF] * (valores_numericos[Cx_0] + valores_numericos[k] * Cz0
    **2)) / (valores_numericos[m_I] * valores_numericos[g_0]))


valores_numericos_0 = {
    v(0): v0,
    gamma(0): gamma0,
    m(0): valores_numericos[m_I],
    t(0): 0,
    s(0): 0,
    lambda_(0): 1,
    Cz(0): Cz0
}
```

1.7 Cálculo del Rango del Jacobiano y el Número de Variables Independientes

- Se calcula el Jacobiano numéricamente utilizando los valores numéricos definidos anteriormente.
- Se convierte el Jacobiano simbólico a una matriz NumPy para poder calcular su rango.
- Se calcula el rango del Jacobiano utilizando la función 'np.linalg.matrix_rank'.
- Se calcula el número de variables independientes como la diferencia entre el número total de variables y el rango del Jacobiano.
- Se imprimen el rango del Jacobiano y el número de variables independientes.

```
jacobiano_N5, num_variables = calcular_jacobiano(N)


jacobiano_numerico = jacobiano_N5.subs(valores_numericos).subs(valores_numericos_0)



jacobiano_numpy = np.array([[float(sympy.re(expr).evalf()) if expr.is_complex else float
    (expr.evalf()) if expr.is_number else float(sympy.re(expr.subs({s: 0 for s in expr.
    free_symbols}))).evalf()) for expr in row] for row in jacobiano_numerico.tolist()],
    dtype=float)


rango = np.linalg.matrix_rank(jacobiano_numpy)
print("Rango del Jacobiano:", rango)

variables_independientes = num_variables - rango
print("Numero de variables independientes:", variables_independientes)
```

1.8 Análisis de la Salida del Código

Al ejecutar el código, se obtiene la siguiente salida:

```
Rango del Jacobiano: 260
Numero de variables independientes: 104
```

Interpretación de la Salida:

- **Rango del Jacobiano (260):** El rango del Jacobiano indica el número de ecuaciones de actualización de estado linealmente independientes. En este caso, el rango es 260, lo que significa que solo 260 de las 364 ecuaciones son linealmente independientes.
- **Número de Variables Independientes (104):** La diferencia entre el número total de variables (364) y el rango del Jacobiano (260) nos da el número de variables independientes. En este caso, hay 104 variables independientes. Esto significa que podemos expresar las 364 variables en función de estas 104 variables independientes.

Implicaciones para la Optimización:

- El hecho de que el rango del Jacobiano sea menor que el número total de variables indica que existen dependencias lineales entre las ecuaciones de actualización de estado.
- Esto sugiere que el problema de optimización puede simplificarse mediante técnicas de reducción de dimensionalidad. En lugar de optimizar las 364 variables, podemos optimizar solo las 104 variables independientes.
- La elección del algoritmo de optimización también se ve influenciada por este resultado. Algunos algoritmos son más eficientes cuando se aplican a problemas con un número reducido de variables independientes.

2 Restricciones del Problema

El problema está sujeto a una serie de restricciones que limitan las soluciones factibles:

- **Ecuaciones de Movimiento**: Las ecuaciones (3), (4), (5), (6) y (7) del enunciado original del problema describen la dinámica de la aeronave y relacionan las variables de decisión entre puntos consecutivos de la trayectoria. Estas ecuaciones representan las leyes físicas que gobiernan el movimiento de la aeronave y deben ser satisfechas en todo momento.
- **Restricciones Operativas**: Además de las ecuaciones de movimiento, existen restricciones operativas que limitan el rango de valores permisibles para ciertas variables:
  - **VMO**: Velocidad calibrada máxima (CAS).
  - **MMO**: Número de Mach máximo.
  - **Vz_min**: Velocidad de ascenso vertical mínima.
  - **Cz_max**: Coeficiente de sustentación máximo.
  - λ**_max**: Empuje máximo (normalizado a 1).

  Estas restricciones garantizan que la aeronave opere dentro de límites seguros y eficientes.

3 Discretización de la Trayectoria

La trayectoria de ascenso se discretiza en N puntos, donde cada punto representa un estado específico de la aeronave. El enunciado del problema sugiere un valor de $N = 53$, lo que implica una discretización cada 500 pies de altitud. Esta discretización transforma el problema de optimización continuo en un problema discreto, facilitando su tratamiento computacional.

4 Parámetros del Problema

El problema incluye una serie de parámetros constantes que definen las características de la aeronave y las condiciones ambientales, como:

- **Coeficientes aerodinámicos** (Cx_0**,** k**)**: Describen la resistencia aerodinámica de la aeronave.
- **SREF**: Superficie de referencia aerodinámica.
- η: Consumo específico de combustible.
- Zp_I**,** Zp_F: Altitud inicial y final.
- m_I: Masa inicial de la aeronave.
- **CAS_I**: Velocidad calibrada inicial.
- **VMO, MMO**: Velocidades máximas.
- M_CRZ: Número de Mach de crucero.
- **L**: Distancia total de la trayectoria.

- **Vz_min**: Velocidad de ascenso vertical mínima.
- g_0: Aceleración de la gravedad.
- **CI**: Índice de costo.

5 Complejidad del Problema

La combinación de la no linealidad de las ecuaciones de movimiento, el gran número de variables de decisión ($7 * (N - 1) = 364$ para $N = 53$) y las restricciones operativas hacen que este sea un problema de optimización complejo. Los métodos clásicos de optimización pueden encontrar dificultades para encontrar la solución óptima global en un tiempo razonable, especialmente para valores grandes de N. Esto motiva la exploración de enfoques alternativos. Sin embargo dado el análisis del numero total de variables hecho a partir del rango del Jacobino antes presentado tenemos una reducción significativa de la complejidad al tratar ahora con 104 variables.

Consideraciones

- **Modelo de Empuje**: El modelo de empuje máximo de ascenso (FN_MCL) es una función lineal de la altitud, lo que simplifica ligeramente el problema.
- **Modelo Atmosférico**: El modelo atmosférico utilizado considera la variación de la densidad del aire con la altitud.
- **Fases Adicionales**: El problema incluye dos fases adicionales después del ascenso: una aceleración a altitud constante hasta alcanzar la velocidad de crucero y un segmento de crucero a altitud y velocidad constantes. Aunque estas fases no están implícitas en el trayecto que se busca optimizar.

6 Formulación Matemática del Problema

El problema de optimización se formula matemáticamente como la minimización de la función de costo ϕ, sujeta a un conjunto de restricciones. La función de costo ϕ se define como:

$$\phi = \text{consumo} + CI \times \text{tiempo} \tag{2}$$

donde "consumo" representa el consumo de combustible y "tiempo" es la duración del ascenso. El Índice de Coste (CI) pondera la importancia relativa de estos dos factores.

Las restricciones del problema incluyen:

- **Velocidad calibrada (CAS)**: La velocidad calibrada debe estar dentro de los límites operacionales, es decir, $CAS(v_i, Zp_i) \leq VMO$, donde VMO es la velocidad calibrada máxima.
- **Número de Mach**: El número de Mach también debe estar limitado, $MACH(v_i, Zp_i) \leq MMO$, donde MMO es el número de Mach máximo.
- **Velocidad vertical (Vz)**: La velocidad vertical debe ser mayor que un valor mínimo, $Vz_{min} \leq v_i \sin \gamma_i$.
- **Coeficiente de sustentación (Cz)**: El coeficiente de sustentación está limitado por un valor máximo, $Cz_i \leq Cz_{max}$.
- **Configuración de empuje (λ)**: La configuración de empuje debe estar entre 0 y 1, $0 \leq \lambda_i \leq 1$.

Además de estas restricciones, el modelo matemático incluye ecuaciones que describen la dinámica del vuelo, como las ecuaciones de movimiento y la ecuación de la energía. Estas ecuaciones relacionan las variables de estado de la aeronave, como la velocidad, el ángulo de ascenso, la masa y la distancia recorrida.

6.1 Parámetros y Variables

El modelo utiliza una serie de parámetros que caracterizan las propiedades de la aeronave y las condiciones atmosféricas. Estos parámetros incluyen coeficientes aerodinámicos (Cx_0, k), el área de referencia aerodinámica (S_{REF}), el consumo específico de combustible (η), las altitudes inicial y final (Zp_I, Zp_F), la masa inicial (m_I), la velocidad calibrada inicial (CAS_I), la longitud total de la trayectoria (L), y la aceleración debida a la gravedad (g_0).

Las variables del problema incluyen la velocidad verdadera (v_i), el ángulo de ascenso (γ_i), la masa (m_i), el tiempo (t_i), la distancia recorrida (s_i), el coeficiente de sustentación (Cz_i), y la configuración de empuje (λ_i), para cada segmento i del ascenso. El objetivo de la optimización es encontrar los valores de estas variables que minimicen la función de costo ϕ, sujeta a las restricciones mencionadas.

6.2 Función Objetivo y Restricciones

La función objetivo ϕ representa el costo total del ascenso, combinando el consumo de combustible y el tiempo de vuelo. El Índice de Coste (CI) permite ajustar la importancia relativa de estos dos componentes. Minimizar ϕ implica encontrar la trayectoria de ascenso que equilibre de manera óptima el consumo de combustible y la duración del ascenso.

Las restricciones del problema representan limitaciones físicas y operacionales. Los límites en la velocidad calibrada (VMO) y el número de Mach (MMO) garantizan que la aeronave opere dentro de su envolvente de vuelo segura. La restricción en la velocidad vertical (Vz_{min}) asegura que la aeronave ascienda a una velocidad suficiente para mantener la sustentación. El límite en el coeficiente de sustentación (Cz_{max}) impide que la aeronave exceda su capacidad de generar sustentación. Finalmente, la restricción en la configuración de empuje (λ) limita la potencia del motor disponible.

6.3 Discretización del Problema

Para resolver numéricamente el problema, la trayectoria de ascenso se discretiza en N segmentos. Cada segmento se caracteriza por un conjunto de variables $\{v_i, \gamma_i, m_i, t_i, s_i, Cz_i, \lambda_i\}$, donde i varía de 0 a $N-1$. La altitud en cada segmento Zp_i se define como:

$$Zp_i = Zp_I + i\frac{Zp_F - Zp_I}{N-1} \quad (0 \leq i \leq N-1) \tag{3}$$

Esta discretización transforma el problema de optimización continuo en un problema discreto con un número finito de variables. El valor de N determina la granularidad de la discretización y afecta la precisión de la solución. En el código proporcionado, se utiliza $N = 53$, lo que resulta en un problema con 364 incógnitas y 520 restricciones.

6.4 Transformación del sistema a forma matemática equivalente

Para enfrentar la complejidad del sistema de ecuaciones no lineales que gobierna la dinámica del ascenso, un primer paso necesario fue reordenar las ecuaciones y aislar ciertas variables clave. Este enfoque permite reducir el número de incógnitas y facilita el proceso de solución iterativa. El objetivo es expresar λ_{i+1} y Cz_{i+1} en función de las variables en el paso i.

- **Despeje de la configuración de empuje (λ_{i+1})**: El empuje durante el ascenso está relacionado con el cambio de masa de la aeronave, ya que el empuje impulsa el consumo de combustible. Partiendo de la ecuación de restricción de masa (gm):

$$g_{m_i} = \frac{m_{i+1} - m_i}{Zp_{i+1} - Zp_i} + \frac{1}{2}\eta\left(\frac{\lambda_{i+1}F_{N_{MCL_{i+1}}}}{v_{i+1}\sin\gamma_{i+1}} + \frac{\lambda_i F_{N_{MCL_i}}}{v_i \sin\gamma_i}\right) = 0 \tag{4}$$

  Despejando λ_{i+1} obtenemos:

$$\lambda_{i+1} = -\frac{2v_{i+1}\sin\gamma_{i+1}}{\eta F_{N_{MCL_{i+1}}}}\left(\frac{m_{i+1} - m_i}{Zp_{i+1} - Zp_i}\right) - \frac{\lambda_i F_{N_{MCL_i}} v_{i+1}\sin\gamma_{i+1}}{F_{N_{MCL_{i+1}}} v_i \sin\gamma_i} \tag{5}$$

  Esta expresión se implementa en el código Python como la función ‘λ_ip(i):‘, así:

```
    def λ_ip(i):
        term1 = -2 * (v[i+1] * sin(γ[i+1])) / (η * F_N_MCL(i+1))
        term2 = (m[i+1] - m[i]) / (Zp(i+1) - Zp(i))
        term3 = (v[i+1] * sin(γ[i+1]) * λ[i] * F_N_MCL(i)) / (F_N_MCL(i+1) * v[i] *
     sin(γ[i]))
        return (term1 * term2) - term3
```

- **Despeje del coeficiente de sustentación (Cz_{i+1})**: La ecuación de restricción de ángulo de ascenso (g_γ) relaciona el cambio en el ángulo de ascenso con el coeficiente de sustentación.

$$g_{\gamma_i} = \frac{\gamma_{i+1} - \gamma_i}{Zp_{i+1} - Zp_i} - \frac{1}{2}\left(\frac{\frac{1}{2}\rho_{i+1} S_{REF} Cz_{i+1}}{m_{i+1}\sin\gamma_{i+1}} - \frac{g_0}{v_{i+1}^2 \tan\gamma_{i+1}} + \frac{\frac{1}{2}\rho_i S_{REF} Cz_i}{m_i \sin\gamma_i} - \frac{g_0}{v_i^2 \tan\gamma_i}\right) = 0 \tag{6}$$

Despejando Cz_{i+1}:

$$Cz_{i+1} = \frac{2m_{i+1}\sin\gamma_{i+1}}{\rho_{i+1} S_{REF}}\left[\frac{2(\gamma_{i+1} - \gamma_i)}{Zp_{i+1} - Zp_i} - \left(\frac{\rho_i S_{REF} Cz_i}{2m_i \sin\gamma_i} - \frac{g_0}{v_i^2 \tan\gamma_i} - \frac{g_0}{v_{i+1}^2 \tan\gamma_{i+1}}\right)\right] \tag{7}$$

Esta expresión se traduce en la función 'Cz_ip(i):' en el código:

```
    def Cz_ip(i):
        return (2 * m[i+1] * sin(γ[i+1]) * ((2 * γ[i+1] - 2 * γ[i]) / (Zp(i+1) - Zp(
    i)) - (ρ(i) * S_REF * Cz[i]) / (2 * m[i] * sin(γ[i])) + g_0 / (v[i+1]**2 * tan(γ
    [i+1])) + g_0 / (v[i]**2 * tan(γ[i]))))/(ρ(i+1) * S_REF)
```

6.5 Solución iterativa para la masa y otras variables

Una vez que las expresiones para λ_{i+1} y Cz_{i+1} han sido aisladas en función de las variables del segmento i y de m_{i+1}, el siguiente paso crucial es determinar precisamente el valor de la masa de la aeronave en el segmento siguiente, m_{i+1}. Este proceso se realiza de manera iterativa, aprovechando las condiciones conocidas en el segmento i y las relaciones establecidas entre las variables.

La clave para resolver m_{i+1} radica en la ecuación de restricción de velocidad (g_v), que relaciona el cambio en la velocidad verdadera con la altitud, el empuje, la resistencia aerodinámica y la fuerza gravitacional:

$$g_{v_i} = \frac{v_{i+1} - v_i}{Zp_{i+1} - Zp_i} - \frac{1}{2}\left(\frac{\lambda_{i+1} F_{N_{MCL_{i+1}}}}{m_{i+1} v_{i+1}\sin\gamma_{i+1}} - \frac{\frac{1}{2}\rho(Zp_{i+1}) v_{i+1} S_{REF}(Cx_0 + kCz_{i+1}^2)}{m_{i+1}\sin\gamma_{i+1}}\right. \tag{8}$$

$$\left. - \frac{g_0}{v_{i+1}} + \frac{\lambda_i F_{N_{MCL_i}}}{m_i v_i \sin\gamma_i} - \frac{\frac{1}{2}\rho(Zp_i) v_i S_{REF}(Cx_0 + kCz_i^2)}{m_i \sin\gamma_i} - \frac{g_0}{v_i}\right) = 0 \tag{9}$$

Recordemos que ya hemos obtenido expresiones para λ_{i+1} y Cz_{i+1} en función de m_{i+1} y de las variables del segmento i. Sustituyendo estas expresiones en la ecuación g_{v_i}, obtenemos una ecuación no lineal donde la única incógnita es m_{i+1}.

Para ilustrar este proceso, consideremos la expresión de λ_{i+1}:

$$\lambda_{i+1} = -\frac{2v_{i+1}\sin\gamma_{i+1}}{\eta F_{N_{MCL_{i+1}}}}\left(\frac{m_{i+1} - m_i}{Zp_{i+1} - Zp_i}\right) - \frac{\lambda_i F_{N_{MCL_i}} v_{i+1}\sin\gamma_{i+1}}{F_{N_{MCL_{i+1}}} v_i \sin\gamma_i} \tag{10}$$

Y la expresión de Cz_{i+1}:

$$Cz_{i+1} = \frac{2m_{i+1}\sin\gamma_{i+1}}{\rho_{i+1} S_{REF}}\left[\frac{2(\gamma_{i+1} - \gamma_i)}{Zp_{i+1} - Zp_i} - \left(\frac{\rho_i S_{REF} Cz_i}{2m_i \sin\gamma_i} - \frac{g_0}{v_i^2 \tan\gamma_i} - \frac{g_0}{v_{i+1}^2 \tan\gamma_{i+1}}\right)\right] \tag{11}$$

Al sustituir estas expresiones en g_{v_i}, la ecuación resultante, aunque compleja, puede ser resuelta numéricamente para m_{i+1} utilizando métodos como el método de Newton-Raphson o métodos de bisección.

En la implementación en Python, esta resolución numérica se encapsula dentro de la función 'm_ip(i):', la cual devuelve el valor calculado de m_{i+1}. Es importante destacar que la precisión de la solución para m_{i+1} impacta directamente en la precisión del cálculo de las demás variables en el segmento $i+1$ y, por ende, en la precisión de la trayectoria de ascenso óptima.

Una vez que m_{i+1} se ha determinado, se procede a calcular las demás variables del segmento $i+1$ (Cz_{i+1}, λ_{i+1}, s_{i+1}, t_{i+1}) utilizando las expresiones que ya hemos definido en función de m_{i+1} y de las variables del segmento i. Este proceso iterativo se repite para cada segmento del ascenso, construyendo la trayectoria completa de la aeronave.

- **Determinación de la masa (m_{i+1})**: Con los valores conocidos de la velocidad verdadera v_{i+1} y el ángulo de ascenso γ_{i+1} en el segmento siguiente, se utilizan las ecuaciones de movimiento y la ecuación de la energía para calcular m_{i+1}. El cambio de masa está directamente relacionado con el consumo de combustible, el cual depende del empuje y la configuración de vuelo. Resolver para m_{i+1} garantiza que la masa de la aeronave refleje con precisión la cantidad de combustible consumido hasta ese punto.

- **Actualización de otras variables**: Una vez calculada la masa m_{i+1}, se utiliza este valor, junto con v_{i+1} y γ_{i+1}, para calcular las demás variables del sistema en el siguiente intervalo. Esto incluye el coeficiente de sustentación Cz_{i+1}, el tiempo t_{i+1}, la distancia recorrida s_{i+1} y la configuración de empuje λ_{i+1}. Este proceso iterativo asegura que todas las variables permanezcan consistentes con las ecuaciones dinámicas que gobiernan el comportamiento de la aeronave.

6.6 Implementación de la optimización numérica

Una vez establecidas las ecuaciones clave y las relaciones entre las variables, el siguiente paso es implementar un método de optimización numérica para encontrar los valores óptimos de las variables que minimicen la función de costo ϕ. Esta función de costo está compuesta por dos términos principales: el consumo de combustible y el tiempo de vuelo, ponderados por el Índice de Coste (CI).

- **Definición de la función objetivo**: La función objetivo ϕ se define en términos de las variables de estado de la aeronave en cada segmento del ascenso. Esta función se evalúa para un conjunto dado de valores $\{v_i, \gamma_i, m_i, t_i, s_i, Cz_i, \lambda_i\}$ y devuelve el costo total de la trayectoria. La optimización busca minimizar esta función, lo que implica encontrar la combinación de variables que produzca el menor costo total, considerando tanto el tiempo como el consumo de combustible.

- **Manejo de restricciones**: El problema está sujeto a un conjunto de restricciones operacionales y físicas. Estas incluyen límites en la velocidad máxima calibrada (VMO) y el número de Mach máximo (MMO), la velocidad mínima de ascenso (Vzmin), y el coeficiente de sustentación máximo Cz_{max}. Durante el proceso de optimización, estas restricciones se imponen de manera que cualquier solución que las viole sea penalizada o descartada. Esto garantiza que las trayectorias generadas no solo sean óptimas en términos de costo, sino también seguras y factibles desde el punto de vista operacional.

7 Código que presenta explícitamente la función de costo

El código se divide en varias secciones:

1. Definición de constantes y parámetros del problema.
2. Definición de funciones auxiliares para calcular variables aerodinámicas y de la atmósfera.
3. Definición de la función objetivo (función de costo).
4. Implementación del algoritmo de Evolución Diferencial.

7.1 Definición de Constantes y Parámetros

- Se definen las constantes del problema, como los coeficientes aerodinámicos, la eficiencia del combustible, la altitud inicial y final, la masa inicial, la velocidad inicial, etc.

- Se definen las unidades y se realizan las conversiones necesarias para mantener la consistencia dimensional.
- Se definen los límites de las variables de optimización (velocidad y ángulo de ascenso).

```
from math import sin, tan, sqrt, pow, asin, atanh, log, exp, pi, nan
import numpy as np

Cx_0 = 0.014
k = 0.09
Cz_max = 0.7
S_REF = 120
η = 0.06/3600
Zp_I = 10000 * 0.3048
Zp_F = 36000 * 0.3048

π = pi

m_I = 60000
CAS_I = 250*0.5144444444444445
VMO = 350*0.5144444444444445
MMO = 0.82
M_CRZ = 0.80
L = 400000
s_F=L

Vz_min = 1.52400
g_0 = 9.80665
CI = 30/60

m_0 = m_I
t_0 = 0
s_0 = 0
λ_0 = 1

Ts_0 = 288.15
ρ_0 = 1.225
L_z = -0.0065
g_0 = 9.80665
R = 287.05287
α_0 = -g_0/R/L_z
```

7.2 Definición de Funciones Auxiliares

- Se definen funciones para calcular la altitud ‘Zp(i)‘, el empuje máximo ‘F_N_MCL(i)‘, la densidad del aire ‘ρ(i)‘, el número de Mach ‘M(l)‘, la velocidad calibrada del aire ‘CAS(l)‘ y la velocidad aerodinámica real ‘TAS_I‘.
- Estas funciones se utilizan para calcular la función objetivo y las restricciones del problema.

```
def F(x):
    array_ejemplo = np.array(x)
    x1, x2 = np.array_split(array_ejemplo, 2)
    N=len (x1)+1
    x1 = np.array(x1)
    x2 = np.array(x2)

    def Zp(i):
        return Zp_I + i*(Zp_F - Zp_I)/(N)

    def F_N_MCL(i):
        return 140000 - 2.53*Zp(i)/0.3048

    def ρ(i):
        return ρ_0 * ((Ts_0 + L_z*Zp(i))/Ts_0)**(α_0 - 1)

    def M(l):
        return v[l]/sqrt(1.4*R*((Ts_0) + L_z*Zp(l)))

    def CAS(l):
```

```
        arg = (7*R*Ts_0) * (((Ts_0/(Ts_0 + L_z*Zp(l)))**-α_0 *
                                (pow((1 + (v[l]**2/(7*R*(Ts_0 + L_z*Zp(l))))),3.5) - 1)
 + 1)**(1/3.5) - 1)
        if arg < 0:
            return nan
        else:
            return sqrt(arg)

    TAS_I = sqrt     (7*R*(Ts_0 + L_z*Zp_I) * ((((Ts_0 + L_z*Zp_I)/Ts_0)**-α_0 * ((1 +
    CAS_I**2/(7*R*Ts_0))**3.5 - 1) + 1)**(1/3.5) - 1))

    v_0 = TAS_I
    Cz_0 = m_0*g_0/(0.5*ρ(0)*v_0**2*S_REF)
    γ_0 = asin   ((F_N_MCL(0)-0.5*ρ(0)*v_0**2*S_REF*(Cx_0 + k*Cz_0))/(m_0*g_0))
    ρ_F = ρ_0 * ((Ts_0 + L_z*Zp_F)/Ts_0)**(α_0-1)
    v_F = M_CRZ * sqrt(1.4*R*(Ts_0 + L_z*Zp_F))
```

7.3 Definición de la Función Objetivo (Función de Costo)

- Se define la función 'F(x)', que representa la función de costo a minimizar.
- La función 'F(x)' recibe un vector 'x' que contiene las velocidades y los ángulos de ascenso en cada punto de la trayectoria.
- Dentro de la función 'F(x)', se definen varias funciones auxiliares que calculan los valores de las variables de estado en cada punto de la trayectoria, utilizando las ecuaciones de actualización de estado.
- Se define la función '$\Theta(N\_, \_v, \gamma\_)$', que calcula las variables de estado (masa, distancia, tiempo, etc.) en cada punto de la trayectoria, teniendo en cuenta las restricciones del problema.
- Se definen funciones para calcular los valores de las variables en el punto final de la trayectoria (después de la fase de ascenso).
- La función '$\phi(v, \gamma)$' calcula el costo total de la trayectoria, que es una combinación del consumo de combustible y el tiempo de vuelo.

```
    def Cz_ip(i):
        return (2 * m[i+1] * sin(γ[i+1]) * ((2 * γ[i+1] - 2 * γ[i]) / (Zp(i+1) - Zp(i))
    - (ρ(i) * S_REF * Cz[i]) / (2 * m[i] * sin(γ[i])) \
                                            + g_0 / (v[i+1]**2 * tan(γ[i+1])) + g_0 / (v
    [i]**2 * tan(γ[i]))))/(ρ(i+1) * S_REF)

    def m_ip(i):

        A = (v[i+1]-v[i])/(Zp(i+1)-Zp(i))

        L = (-g_0/v[i+1] + (λ[i]*F_N_MCL(i))/(m[i]*v[i]*sin(γ[i]))
            - (0.5*ρ(i)*v[i]*S_REF*(Cx_0+k*Cz[i]**2))/(m[i]*sin(γ[i]))
            - g_0/v[i])

        H = (4*sin(γ[i+1])/(ρ(i+1)*S_REF)) * ((γ[i+1]-γ[i])/(Zp(i+1)-Zp(i)) +
            g_0/(2*v[i+1]**2*tan(γ[i+1])) - (ρ(i)*S_REF*Cz[i])/(4*m[i]*sin(γ[i])) +
            g_0/(2*v[i]**2*tan(γ[i])))

        I = (-2*v[i+1]*sin(γ[i+1])/(η*F_N_MCL(i+1))) * (1/(Zp(i+1)-Zp(i)))

        J = (2*v[i+1]*sin(γ[i+1])/(η*F_N_MCL(i+1))) * (m[i]/(Zp(i+1)-Zp(i))) - (v[i+1]*
    sin(γ[i+1])*λ[i]*F_N_MCL(i))/(F_N_MCL(i+1)*v[i]*sin(γ[i]))

        numerator_1 = (-2.0*A*v[i+1]*sin(γ[i+1]) + F_N_MCL(i+1)*I + L*v[i+1]*sin(γ[i+1])
    )
        numerator_2 = (2.0*sqrt(A**2*v[i+1]**2*sin(γ[i+1])**2 - A*F_N_MCL(i+1)*I*v[i+1]*
    sin(γ[i+1]) -
                                    A*L*v[i+1]**2*sin(γ[i+1])**2 - 0.25*Cx_0*H**2*S_REF
    **2*k*ρ(i+1)**2*v[i+1]**4 +
                                    0.25*F_N_MCL(i+1)**2*I**2 + 0.5*F_N_MCL(i+1)*H**2*J*
    S_REF*k*ρ(i+1)*v[i+1]**2 +
                                    0.5*F_N_MCL(i+1)*I*L*v[i+1]*sin(γ[i+1]) + 0.25*L**2*
    v[i+1]**2*sin(γ[i+1])**2))
```

```
        denominator = H**2*S_REF*k*ρ(i+1)*v[i+1]**2

        m_i_plus_1_positive = (numerator_1 + numerator_2)/denominator
        m_i_plus_1_negative = (numerator_1 - numerator_2)/denominator

        return m_i_plus_1_positive

    def s_ip(i):
        return s[i] + 0.5 * (Zp(i+1) / tan(γ[i+1]) + Zp(i+1) / tan(γ[i]) - Zp(i) / tan(γ[i+1]) - Zp(i) / tan(γ[i]))

    def t_ip(i):
        return t[i] + 0.5 * ((Zp(i+1) - Zp(i)) / (v[i+1] * sin(γ[i+1])) + (Zp(i+1) - Zp(i)) / (v[i] * sin(γ[i])))

    def λ_ip(i):
        term1 = -2 * (v[i+1] * sin(γ[i+1])) / (η * F_N_MCL(i+1))
        term2 = (m[i+1] - m[i]) / (Zp(i+1) - Zp(i))
        term3 = (v[i+1] * sin(γ[i+1]) * λ[i] * F_N_MCL(i)) / (F_N_MCL(i+1) * v[i] * sin(γ[i]))
        return (term1 * term2) - term3

    def Θ(N_,v,γ):
        global N, v, γ, m, s, t, λ, Cz, v_, P
        N=N_ ; P=True ; v=v; γ=γ

        m = [m_0]; s=[s_0]; t=[t_0]; λ=[λ_0]; Cz=[Cz_0]
        Cz_i = Cz_0
        for i in range (0, N-1, 1):
            if len(v) <= (i+1) or v[i+1]*sin(γ[i+1]) < Vz_min or CAS(i+1) > VMO : P=False; N = i+1; break
            m.append(m_ip(i))
            Cz.append(Cz_ip (i))
            λ.append(λ_ip(i))
            s.append(s_ip(i))
            t.append(t_ip(i))
            if (λ[i+1] > 1 or λ[i+1] < 0) or (Cz[i+1] > Cz_max or M(i+1) > MMO):
                P=False; break
        return


    def A():
        return (-ρ_F * S_REF * Cx_0) / (2 * m[N-1]) - (6 * k * m[N-1] * g_0**2) / (ρ_F * S_REF * v[N-1]**4)

    def B():
        return (16 * k * m[N-1] * g_0**2) / (ρ_F * S_REF * v[N-1]**3)

    def C():
        return (F_N_MCL(N-1) / m[N-1]) - (12 * k * m[N-1] * g_0**2) / (ρ_F * S_REF * v[N-1]**2)

    def D(A, B, C):
        return (B**2 - 4 * A * C)**0.5

    def t_B():
        global kuo45789
        kuo45789 = (2/D(A(), B(), C())) * (atanh((2*A()*v[N-1] + B())/D(A(), B(), C())) - atanh((2*A()*v_F + B())/D(A(), B(), C())))
        return t[N-1] + kuo45789


    def m_B():
        return m[N-1] - η * λ[N-1] * F_N_MCL(N-1) * (t_B() - t[N-1])

    def s_B():
        return s[N-1] + (1/A()) * log((D(A(), B(), C())-2*A()*v_F-B())/(D(A(), B(), C())-2*A()*v[N-1]-B())) - (B()+D(A(), B(), C()))/(2*A()) * (t_B() - t[N-1])

    def m_F():
        return m_B() * exp((-2 * η * g_0 * sqrt(k * Cx_0) / v_F) * (s_F - s_B()))

    def t_F():
```

```
        return t_B() + (s_F - s_B()) / v_F

    def ϕ(v,γ):
        Θ(N,v,γ)
        if P == True:
            nada = -m_F()  + CI*(t_B() - s_B()/v_F)
            dt = [j-i for i, j in zip(t[:-1], t[1:])]
            dtp = dt + [kuo45789]
            aaa = [aa *  bb for aa, bb in zip(v.tolist(), np.sin(γ).tolist())]
            ccc = [aa *  bb for aa, bb in zip(v.tolist(), np.cos(γ).tolist())]
            bbb = [aa_ *  bb_ for aa_, bb_ in zip(aaa, dtp)]
            ddd = [aa_ *  bb_ for aa_, bb_ in zip(ccc, dtp)]
            if sum(ddd) <= 350000:
                return sum(bbb)
            else:
                return nan
        else:
            return nan


    return ϕ(np.concatenate(([v_0], x1)), np.concatenate(([γ_0], x2*π/180)))
```

7.4 Implementación del Algoritmo de Evolución Diferencial

- Se define la función 'objective_function(x)', que se utiliza como la función objetivo para el algoritmo de Evolución Diferencial.
- Se define la función 'FF(x)', que es una versión modificada de la función objetivo que se utiliza para manejar los casos en los que la función 'F(x)' retorna valores no numéricos (NaN).
- Se definen los límites de las variables de optimización en la lista 'bounds'.
- Se define un punto de referencia inicial 'x_ref'.
- Se define un vector de perturbaciones 'perturbations' que se utiliza para generar la población inicial del algoritmo de Evolución Diferencial.
- Se define el tamaño de la población 'population_size'.
- Se implementa un bucle que ejecuta el algoritmo de Evolución Diferencial varias veces, utilizando el punto de referencia anterior como punto de partida para la siguiente iteración.
- Se utiliza la función 'differential_evolution' de la biblioteca SciPy para implementar el algoritmo de Evolución Diferencial.
- Se define una función 'callback' que se utiliza para registrar el historial de la optimización.
- Se define una función 'generate_individual_perturbations' que se utiliza para generar la población inicial del algoritmo de Evolución Diferencial, utilizando el punto de referencia y el vector de perturbaciones.
- Se imprimen los resultados de la optimización (el punto óptimo encontrado y el valor mínimo de la función objetivo).

```
def objective_function(x):
    a = -F(x)
    if a > 0:
        return a
    else:
        return 88000

def FF(x):
    a = F(x)
    if a > 0:
        return a
    else:
        return 99999999999 # simplemente un valor alto
from scipy.optimize import differential_evolution
```

```
import time
bounds = [(80, 201.13351856), (80, 201.13351856), ... ... ... (0, 0.1), (0, 0.1)] # Limites de exploracion considerados en la busqueda
population_size = 15
x_ref = np.array([140.4982793587414, 138.93222706627756, 149.16832 ...
3.0197379365411465, 2.2377882124286907, 1.968444894...]) # Vector velocidades - angulos

perturbations = [0.1 for _ in range(52)] + [0.01 for _ in range(52)]

print("len (bounds) = ",len(bounds), "     len(x_ref) = ",len(x_ref),"   perturbations = ",len(perturbations) )


atsk = 8014.7

for i in range (2,5000):
    SS = True

    while SS:
        history = []
        def callback(xk, convergence):
            history.append(xk.copy())


        def generate_individual_perturbations(reference_point, perturbations):
            return [ref + np.random.uniform(-perturb, perturb) for ref, perturb in zip(reference_point, perturbations)]


        initial_population = [generate_individual_perturbations(x_ref, perturbations) for _ in range(population_size)]
        result = differential_evolution(FF, bounds, callback=callback, init=np.array(initial_population))

        if result.fun < atsk:
            SS = False

    atsk = result.fun
    print("PUNTO encontrado Angulo:", result.x.tolist())
    print("El valor minimo es:", result.fun)
    print("i = ",i)
    x_ref = result.x
```

8 Notas finales

El presente trabajo ha abordado el desafío de la optimización de la trayectoria de ascenso de aeronaves, un problema complejo que involucra la minimización de una función de costo no lineal sujeta a un conjunto de restricciones operacionales. La implementación del algoritmo de Evolución Diferencial, adaptado a las características del problema y combinado con un análisis del Jacobiano para la reducción de la dimensionalidad, ha demostrado ser una estrategia efectiva para encontrar soluciones que minimizan el costo total del ascenso, considerando tanto el consumo de combustible como el tiempo de vuelo.

A continuación, se presentan las conclusiones derivadas de este trabajo:

1. Efectividad del Algoritmo de Evolución Diferencial:

- El algoritmo de Evolución Diferencial ha mostrado ser capaz de encontrar soluciones óptimas para el problema de optimización, incluso en presencia de una función de costo compleja y no lineal, y un gran número de variables y restricciones.
- La capacidad del algoritmo para explorar el espacio de soluciones de manera eficiente, escapando de mínimos locales, ha sido crucial para obtener resultados satisfactorios.
- La estrategia de utilizar el punto óptimo encontrado en cada iteración como punto de partida para la siguiente ha permitido refinar la solución iterativamente, mejorando su calidad.

2. Importancia del Análisis de Independencia Diferencial:

- El análisis del Jacobiano y el cálculo del rango han revela la existencia o no de dependencias lineales entre las ecuaciones de actualización de estado, lo que ha permitido reducir la dimensionalidad del problema.
- Esta reducción de dimensionalidad ha simplificado el problema de optimización, haciéndolo más tratable y mejorando la eficiencia del algoritmo de Evolución Diferencial.
- El análisis de la estructura del Jacobiano ha proporcionado información valiosa sobre las relaciones entre las variables del problema, lo que podría ser útil para futuras investigaciones.

3. Complejidad del Problema de Optimización:

- El problema de optimización de la trayectoria de ascenso de aeronaves es inherentemente complejo debido a la no linealidad de las ecuaciones de movimiento, el gran número de variables y las restricciones operacionales.
- La elección del algoritmo de optimización y la configuración de sus parámetros han sido cruciales para obtener resultados satisfactorios en un tiempo razonable.

4. Implicaciones para la Industria Aeronáutica:

- El desarrollo de herramientas de optimización de trayectorias de vuelo puede contribuir a la reducción del impacto ambiental de la aviación, minimizando el consumo de combustible y las emisiones de gases de efecto invernadero.
- La optimización de las operaciones aéreas puede generar ahorros significativos en costos para las aerolíneas, mejorando su eficiencia y competitividad.
- La investigación en este campo puede contribuir al desarrollo de nuevas tecnologías y estrategias para la gestión del tráfico aéreo, mejorando la seguridad y la eficiencia del sistema.