Análisis del Jacobiano para la Optimización de la Trayectoria de Ascenso de una Aeronave A320

Marco A. Erazo. Dir: Leonardo A. Pachon

20 de octubre de 2024

1. Introducción

Este documento describe el código Python utilizado para analizar el Jacobiano de las ecuaciones de actualización de estado en el problema de optimización de la trayectoria de ascenso de una aeronave. El objetivo principal es determinar el número de variables independientes en el problema, lo cual es crucial para la selección e implementación de algoritmos de optimización eficientes.

El problema se discretiza en $N = 53$ puntos a lo largo de la trayectoria, y en cada punto se consideran 7 variables: velocidad, ángulo de ascenso, masa, tiempo, distancia, coeficiente de sustentación y fracción de empuje. El código calcula el Jacobiano numéricamente, evaluando las derivadas parciales de las ecuaciones de actualización de estado con respecto a cada una de las $7*(N-1) = 364$ variables. El rango del Jacobiano se utiliza para determinar el número de variables independientes.

2. Descripción del Código

El código 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')
```

2.1. Definición de 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.

2.2. Definición de 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))
```

2.3. Definición de las Ecuaciones de Actualización de Estado

- 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.

```
# Define las ecuaciones usando Zp(i) directamente
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)) )
```

2.4. 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:

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

  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.

```
64 def derivada_parcial(funcion, variable, i_funcion, i_variable):
65     """
66     Calcula la derivada parcial de una funcion respecto a una
       variable con subindices especificos.
67     """
68     derivada = funcion.subs(i, i_funcion).diff(variable(i_variable)
       ).simplify().trigsimp()
69     return derivada
70
71
72 def calcular_jacobiano(N):
73     """Calcula el Jacobiano de las restricciones"""
74
75     funciones = [g_v, g_gamma, g_m, g_t, g_s]
76     variables = [v, gamma, m, t, s, Cz, lambda_]
77
78     num_funciones = len(funciones) * (N - 1)
79     num_variables = len(variables) * (N - 1)
80
81     jacobiano = sympy.zeros(num_funciones, num_variables)
82
83     for i_funcion in range(N - 1):
84         for j_variable in range(N - 1):  # j_variable va de 0 a N
       -2, representando indices de 1 a N-1
85             for k_func, func in enumerate(funciones):
86                 for l_var, var in enumerate(variables):
87                     #  Se suma 1 a j_variable al derivar para
       representar indices de 1 a N-1
88                     derivada = derivada_parcial(func, var,
       i_funcion, j_variable + 1)  # asegura
89                     jacobiano[i_funcion * len(funciones) + k_func,
       j_variable * len(variables) + l_var] = derivada
90
91     #return jacobiano
92     return jacobiano, num_variables
```

2.5. 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).

```
# 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
}
```

2.6. 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.

```
# Resto del c digo (similar al anterior)
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)
```

2.7. 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.

3. Conclusión

El código presentado permite calcular el Jacobiano numéricamente y determinar su rango, lo cual facilitaria el análisis del problema y la toma de decisiones informadas sobre la estrategia de optimización a seguir.

El análisis del Jacobiano proporciona información crucial sobre el número de variables independientes en el problema de optimización de la trayectoria de ascenso. El hallazgo de que solo 104 de las 364 variables son independientes tiene importantes implicaciones en la complejidad de la optimización.

Este resultado sugiere que es posible simplificar el problema con una reducción de dimensionalidad, optimizando únicamente sobre las variables independientes. Esto podría conducir a una reducción significativa del tiempo de cómputo y a una mejora en la eficiencia del proceso de optimización.