Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

Análisis Dinámico de una Montaña Rusa

Asignatura: MNAF

Grado en Física - Universidad de Oviedo

Autor: Edgar Fernández Vigil
Fecha: January 13, 2026

Contents

1 *Configuración*

1.1 *Módulos importados y configuración extra*

[1]:
```
# Modulo de cálculos específicos de la montaña rusa
import calculos_montaña as cmr

# Module de gestión de fechas y rutas
import os

# Modulos de cálculo y gestión simbólica
import numpy as np
import sympy as sp

# Modulos de graficación
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Modulo de interpolación
import scipy.interpolate as scip
from scipy.interpolate import BSpline

# Funcion de resolución de EDOs
from scipy.integrate import solve_ivp

# Modulos de animación
from matplotlib.animation import FuncAnimation, PillowWriter
from IPython.display import HTML

# Supresión de warnings
import warnings
warnings.filterwarnings('ignore')

# Configurar para animaciones interactivas
%matplotlib widget
```

```
has imported calculos_montaña package, contiene lo necesario para cálculos de
montaña rusa
```

1.2 *Funciones necesarias*

[2]:
```
# Función que genera la expresión paramétrica
def montaña_rusa_parametrica(X_data=None, Z_data=None, Y_data=None, grado=3):
    '''
    Construye la expresión paramétrica r(t) mediante B-splines.

    Parámetros:
    -----------
```

```
    X_data : array-like
        Coordenadas horizontales (distancia)
    Z_data : array-like
        Coordenadas verticales (altura)
    grado : int
        Grado del B-spline (default=3, cúbico)

    Retorna:
    --------
    curva_bspline: expresión parámetrica evaluable con un parámetro [0,1]
    '''

    # Como hacemos una función 3D, si falta alguna coordenada la llenamos con ceros
    if X_data is None:
        X_data = np.full(len(Z_data), 0)
    elif Z_data is None:
        Z_data = np.full(len(X_data), 0)
    elif Y_data is None:
        Y_data = np.full(len(X_data), 0)

    # definimos los Puntos de control, serán los mismos datos
    puntos_control = np.array([X_data, Y_data, Z_data])  # Cada fila es una coordenada (X, Y, Z)

    # Definimos los nodos, debe cumplir la condición len(nodos) = len(puntos_control) + grado + 1
    nodos = np.array([0]*(grado) + list(np.linspace(0, 1, len(X_data)-grado)) + [1]*(grado)) #Ponemos los nodos al inicio y al final repetidos grado+1 veces para forzar a que la curva pase por los puntos extremos

    # Usamos la clase BSpline de scipy.interpolate
    curva_bspline = BSpline(nodos, puntos_control.T, grado)

    return curva_bspline
```

2 Simulación en 2D

2.1 *Definición de las curvas a usar*

En el paquete definido como calculos_montaña tenemos en esencia lo necesario para hacer una montaña rusa, lo primero que tenemos que hacer antes de generar una función interpolante es definir las partes que queremos que tenga nuestra montaña rusa. Los elementos que mínimo debe de tener esta montaña rusa son una gran bajada, una montañita y un looping, por lo tanto nuestra montaña rusa será de la siguiente manera - gran bajada - looping - montañita (gaussiana) - pequeña bajada - looping - recta final de frenado

En calculos de montaña tenemos lo necesario para calcular por partes estas secciones, luego las uniremos de forma que quede mas o menos uniforme y se parezca a una montaña rusa.

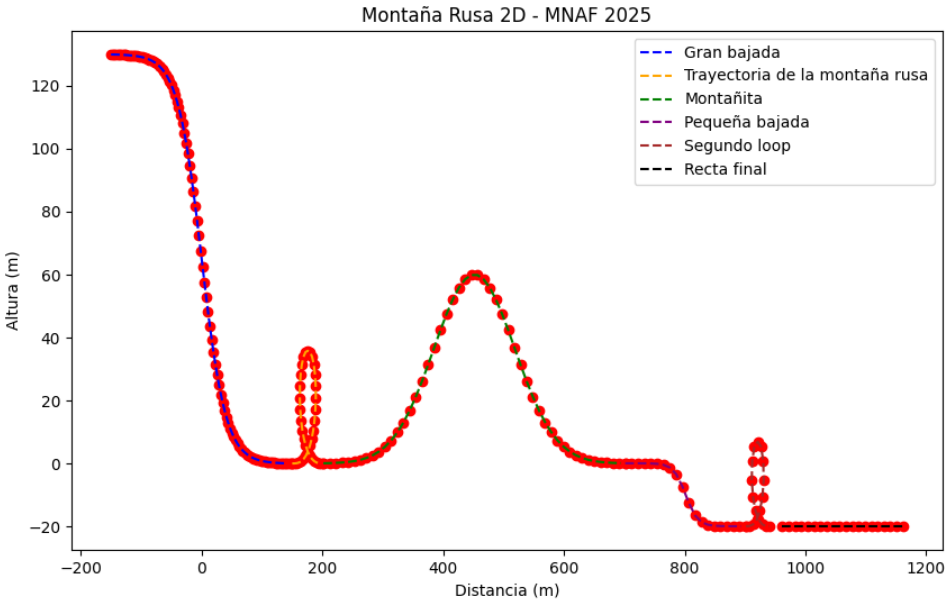
[3]:
```
# Definimos la gran bajada de la montaña rusa
Gran_bajada = cmr.curva3d('s',np.linspace(-150,150,100),A=130,args=[0.05])

# Definimos el primer loop de la montaña rusa
Loop1 = cmr.curva3d('l',np.linspace(0,1.
 ↪79,20),A=40,C=[Gran_bajada[0][-1],0,Gran_bajada[2][-1]],args=[0])

# Definimos montañita (gaussina)
Montañita = cmr.curva3d('g',np.
 ↪linspace(-250,250,50),A=60,C=[Loop1[0][-1]+250,0,Loop1[2][-1]],args=[95])

# Definimos la pequeña bajada
Pequeña_bajada = cmr.curva3d('s',np.
 ↪linspace(-100,100,20),A=20,C=[Montañita[0][-1]+100,0,Montañita[2][-1]-20],args=[0.
 ↪1])

# Definimos el segundo loop de la montaña rusa
Loop2 = cmr.curva3d('l',np.linspace(0,1.
 ↪79,10),A=30,C=[Pequeña_bajada[0][-1],0,Pequeña_bajada[2][-1]],args=[0])

# Recta final frenado
Recta_final = [Pequeña_bajada[0][-1]+np.linspace(0,200,20)+60,np.zeros(20),np.
 ↪full(20,Loop2[2][-1])]
```

2.2 *Dibujo de la montaña rusa*

Una vez definidas podremos ver la forma que nos gustaría que tuviese usando matplotlib

[4]:
```
# ============================================
# Gráfica de la montaña rusa
# ============================================

plt.figure(figsize=(10,6))

# Graficar la gran bajada
```

```
plt.plot(Gran_bajada[0], Gran_bajada[2], 'o', color='red')
plt.plot(Gran_bajada[0], Gran_bajada[2], '--', color='blue', label='Gran␣
 ↪bajada')

# Graficar el primer loop
plt.plot(Loop1[0], Loop1[2], 'o', color='red')
plt.plot(Loop1[0], Loop1[2], '--', color='orange', label='Trayectoria de la␣
 ↪montaña rusa')

# Graficar la montañita
plt.plot(Montañita[0], Montañita[2], 'o', color='red')
plt.plot(Montañita[0], Montañita[2], '--', color='green', label='Montañita')

# Graficar la pequeña bajada
plt.plot(Pequeña_bajada[0], Pequeña_bajada[2], 'o', color='red')
plt.plot(Pequeña_bajada[0], Pequeña_bajada[2], '--', color='purple',␣
 ↪label='Pequeña bajada')

# Graficar el segundo loop
plt.plot(Loop2[0], Loop2[2], 'o', color='red')
plt.plot(Loop2[0], Loop2[2], '--', color='brown', label='Segundo loop')

# Graficar la recta final
plt.plot(Recta_final[0], Recta_final[2], 'o', color='red')
plt.plot(Recta_final[0], Recta_final[2], '--', color='black', label='Recta␣
 ↪final')

# Configuraciones de la gráfica
plt.legend()
plt.title('Montaña Rusa 2D - MNAF 2025')
plt.xlabel('Distancia (m)')
plt.ylabel('Altura (m)')
plt.show()
```



2.3 *Construimos la expresión paramétrica*

Una vez sabemos la forma que nos gustaría que tuviese debemos de construir una función que podamos usar para los cálculos, ya que usando unicamente los puntos no nos servirá, contruieremos una expresión parmétrica, tal que r(u)=(x(u),z(u)). Definimos así la trayectoría completa de la montaña rusa.

Nos ayudaremos de B-splines, existe una implementación en scipy.interpolate llamada justo BSpline, necesitaremos de unos puntos de control, que serán justamente nuestros datos, y un vector de nodos.

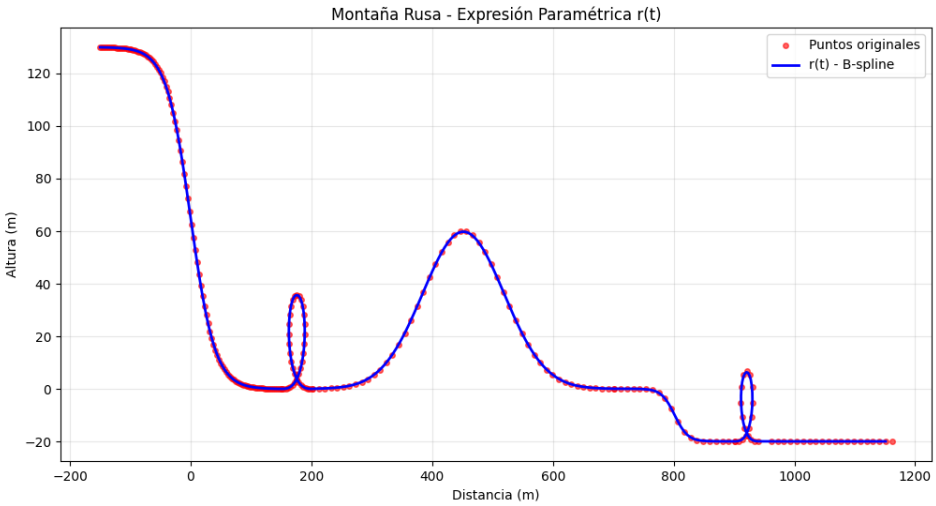
El vector de nodos no está construido así porque si, de hecho a parte de las propiedades mismas que ha de cumplir por si solo, los cuatro 0 al principio y los cuatro 1. al final fuerzan a la expresión a pasar por los extremos, el resto de nodos servirán como atractores, y se le podría dar más importancia a unos o a otros dependiendo de la distancia relativa entre los nodos.

Así la expresión construida será de la forma $\sum_{n=0}^{m-k-2} c_i B_{i,k} \quad t \in [t_{k-1}, t_{m-k}]$,donde k es el grado y m la longitud del vector de nodos, es decir el número de elementos.

2.3.1 *Visualización de lo obtenido tras interpolar*

[5]:
```
# Visualización usando la expresión paramétrica r(t)

#Construimos la expresión
Curva_parametrica = montaña_rusa_parametrica(
    X_data = np.concatenate([Gran_bajada[0], Loop1[0], Montañita[0],␣
 ↪Pequeña_bajada[0], Loop2[0], Recta_final[0]]),
```

```
    Z_data = np.concatenate([Gran_bajada[2], Loop1[2], Montañita[2],␣
 ↪Pequeña_bajada[2], Loop2[2], Recta_final[2]]),
)

# Evaluamos
t_eval = np.linspace(0, 1, 500)
Puntos_de_interpolación=Curva_parametrica(t_eval)

plt.figure(figsize=(12, 6))

# Gráfica comparativa
plt.plot(np.concatenate([Gran_bajada[0], Loop1[0], Montañita[0],␣
 ↪Pequeña_bajada[0], Loop2[0], Recta_final[0]]),
         np.concatenate([Gran_bajada[2], Loop1[2], Montañita[2],␣
 ↪Pequeña_bajada[2], Loop2[2], Recta_final[2]]),
         'o', color='red', markersize=4, label='Puntos originales', alpha=0.6)
plt.plot(Puntos_de_interpolación[:,0], Puntos_de_interpolación[:,2], '-',␣
 ↪color='blue', linewidth=2, label='r(t) - B-spline')

plt.legend()
plt.title('Montaña Rusa - Expresión Paramétrica r(t)')
plt.xlabel('Distancia (m)')
plt.ylabel('Altura (m)')
plt.grid(True, alpha=0.3)
plt.show()
```



2.4 *Caso sin fricción*

Una vez parametrizada la curva debemos plantear el sistema y resolverlo, el cual nos da su dinámica de movimiento. Primero debemos de saber a que problema nos enfrentamos y como debemos de abordarlo.

Para plantear las ecuaciones de la dinámica de movimiento obviamente usaremos la famosa ecuación de Newton la cual es $\sum F = ma$, evidentemente de forma simplificada, ya que no consideraremos posibles cambios de masa durante el trayecto.

A diferencia de otras ocasiones donde trabajamos con la base cartesiana, lo cual se podría hacer, trabajaremos con otra vas ortonormal, conocida como el Triedo de Fernet, vectores tangente, normal y binormal a la superficie de la curva en cada punto. Estos se pueden definir directamente con las derivadas de la curva la cual ya tenemos parametrizada, y nos otroga alguna que otra facilidad a la hora de resolver el problema.

el problema queda de la siguiente manera:

$$\frac{d}{dt}\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \frac{v}{\|\dot{r}\|} \\ -g(\vec{k}\cdot\vec{t}) - \left(\mu\left|\kappa v^2 + g(\vec{k}\cdot\vec{n})\right| + c_v v^2\right)\operatorname{sgn}(v) \end{pmatrix} \tag{1}$$

Siendo estas Edos ordinarias que solve_ivp sabe resolver, por lo tanto tendremos que definir las funciones que se igualan a las derivadas ordinarias de u(parámetro) y v(velocidad), que es lo que necesita solve_ivp para darnos una solución.

2.4.1 *Preparación del sistema*

[6]:
```
# ============================================
# PREPARACIÓN DEL SISTEMA EDO
# ============================================

# Concatenar datos completos de la montaña rusa
Distancia_data = np.concatenate([Gran_bajada[0], Loop1[0], Montañita[0],␣
 ↪Pequeña_bajada[0], Loop2[0], Recta_final[0]])
Altura_data = np.concatenate([Gran_bajada[2], Loop1[2], Montañita[2],␣
 ↪Pequeña_bajada[2], Loop2[2], Recta_final[2]])

# Calcular derivadas de la curva usando pkgmrusa
derivadas_curva = cmr.trayec_der(Curva_parametrica)

# Parámetros físicos del sistema
masa = 500.0          # kg - masa del vagón
mu = 0.0              # coeficiente de rozamiento (0 = conservativo)
ca = 0.0              # coeficiente de arrastre aerodinámico (0 = conservativo)
Sf = 2.0              # m² - superficie frontal
rho_aire = 1.225      # kg/m³ - densidad del aire
g = 9.81              # m/s² - gravedad (VALOR POSITIVO, la función ya aplica el␣
 ↪signo)
```

```
cvefVis = ca * Sf * rho_aire / (2 * masa)

# Condiciones iniciales [parámetro u0, velocidad v0]
u0 = 0.0              # inicio de la curva
v0 = 1.0              # m/s - velocidad inicial
y0 = [u0, v0]

#============================================
# Sistema Sedo
#============================================
def edo_montaña_rusa(t, y):
    '''
    Define el sistema de EDOs para la montaña rusa.

    Parámetros:
    -----------
    t : float
        Tiempo (no se usa explícitamente en este sistema)
    y : list
        Estado del sistema [posición u, velocidad v]

    Retorna:
    --------
    dydt : list
        Derivadas [du/dt, dv/dt]
    '''

    return cmr.edofun_mr(t, y, derivadas_curva, mu, cvefVis, g)

#Información del sistema
print("="*70)
print("SISTEMA EDO - CASO CONSERVATIVO")
print("="*70)
print(f"Parámetros físicos:")
print(f"  • Masa: {masa} kg")
print(f"  • Coef. rozamiento  : {mu} (SIN FRICCIÓN)")
print(f"  • Coef. arrastre ca: {ca} (SIN ROZAMIENTO)")
print(f"  • Superficie frontal Sf: {Sf} m²")
print(f"  • Coeficiente viscoso cvefVis: {cvefVis} 1/m")
print(f"  • Gravedad: {g} m/s² (positivo)")
print(f"\nCondiciones iniciales:")
print(f"  • Posición inicial: u  = {u0}")
print(f"  • Velocidad inicial: v  = {v0} m/s")
```

```
======================================================================
SISTEMA EDO - CASO CONSERVATIVO
======================================================================
Parámetros físicos:
```

- Masa: 500.0 kg
- Coef. rozamiento : 0.0 (SIN FRICCIÓN)
- Coef. arrastre ca: 0.0 (SIN ROZAMIENTO)
- Superficie frontal Sf: 2.0 m^2
- Coeficiente viscoso cvefVis: 0.0 1/m
- Gravedad: 9.81 m/s^2 (positivo)

Condiciones iniciales:
- Posición inicial: u = 0.0
- Velocidad inicial: v = 1.0 m/s

2.4.2 *Comparación de Métodos Numéricos*

Una vez definido el problema querremos saber que método es el mejor para la resolución de este problema concreto, ya que solve_ivp nos ofrece unos cuantos métodos numericos de resolución.

Nos basaremos para escoger en un parámetro clave siempre en física, la energía.

Al ser un problema conservativo la Energía se ha de mantener constante a lo largo de toda la trayectoria, el que menos fluctuaciones presente en la energía lo escogeremos como método optimo de resolución

Solución por los distintos metodos conocidos

[7]:
```
# Metodos a usar
Metodos=['RK45','RK23','DOP853','Radau','BDF','LSODA']

# Diccionario para almacenar las soluciones
soluciones = {}

# Tiempo de evaluación
t_eval = np.linspace(0, 70, 1000)

# Resolución del Sedo por cada método
for metodo in Metodos:
    # Aquí iría la llamada a la función que resuelve la SEDO usando el método especificado
    x = solve_ivp(edo_montaña_rusa,[0,70],y0,method=metodo,t_eval=t_eval,dense_output=True)
    soluciones[metodo] = x

for solucion in soluciones:
    if soluciones[solucion].success == True:
        print(f"La solución con el método {solucion} fue exitosa.")
    else:
        pass
```

```
La solución con el método RK45 fue exitosa.
La solución con el método RK23 fue exitosa.
La solución con el método DOP853 fue exitosa.
```

```
La solución con el método Radau fue exitosa.
La solución con el método BDF fue exitosa.
La solución con el método LSODA fue exitosa.
```

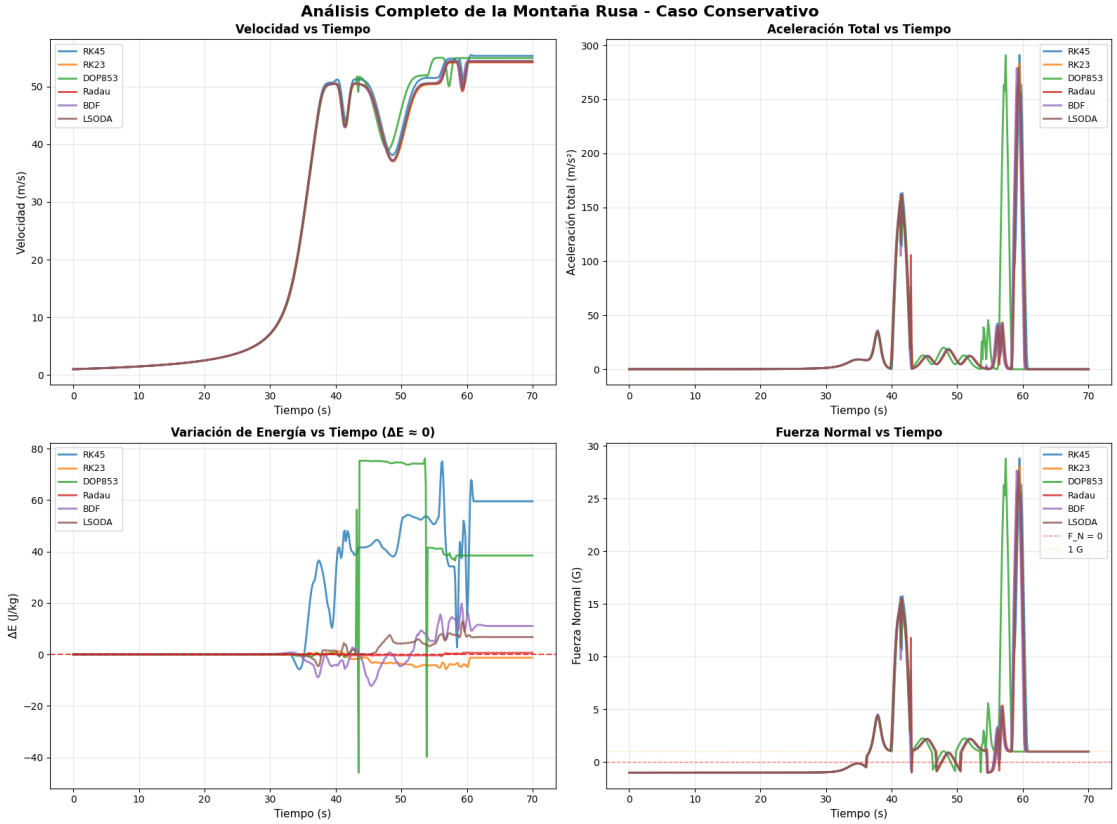
Visualicación de lo obtenido

[8]:
```
# ================================================
# GRÁFICAS COMBINADAS: Análisis completo del movimiento
# ================================================

fig, axes = plt.subplots(2, 2, figsize=(16, 12))
fig.suptitle('Análisis Completo de la Montaña Rusa - Caso Conservativo',␣
 ↪fontsize=16, fontweight='bold')

# ================================================
# 1. VELOCIDAD vs TIEMPO (subplot superior izquierdo)
# ================================================
ax1 = axes[0, 0]
for metodo, sol in soluciones.items():
    ax1.plot(sol.t, sol.y[1], label=f'{metodo}', linewidth=2, alpha=0.8)
ax1.set_xlabel('Tiempo (s)', fontsize=11)
ax1.set_ylabel('Velocidad (m/s)', fontsize=11)
ax1.set_title('Velocidad vs Tiempo', fontsize=12, fontweight='bold')
ax1.legend(loc='best', fontsize=9)
ax1.grid(True, alpha=0.3)

# ================================================
# 2. ACELERACIÓN TOTAL vs TIEMPO (subplot superior derecho)
# ================================================
ax2 = axes[0, 1]
for metodo, sol in soluciones.items():
    # Calcular aceleración total para cada punto
    t_dense = np.linspace(sol.t[0], sol.t[-1], 500)
    datos = sol.sol(t_dense)
    u_vals = datos[0]
    v_vals = datos[1]

    aceleraciones = []
    for i in range(len(u_vals)):
        fuerzaN, baseLocal, ctes = cmr.fuerzaNormal(u_vals[i], v_vals[i],␣
 ↪derivadas_curva, g)
        ace_total, ace_tg, ace_nr = cmr.aceleracion(v_vals[i], baseLocal, mu,␣
 ↪cvefVis, fuerzaN, ctes[1], g)
        aceleraciones.append(ace_total)

    ax2.plot(t_dense, aceleraciones, label=f'{metodo}', linewidth=2, alpha=0.8)

ax2.set_xlabel('Tiempo (s)', fontsize=11)
```

```
ax2.set_ylabel('Aceleración total (m/s²)', fontsize=11)
ax2.set_title('Aceleración Total vs Tiempo', fontsize=12, fontweight='bold')
ax2.legend(loc='best', fontsize=9)
ax2.grid(True, alpha=0.3)

# ================================================
# 3. VARIACIÓN DE ENERGÍA vs TIEMPO (subplot inferior izquierdo)
# ================================================
ax3 = axes[1, 0]
variaciones_energia = {}  # Guardar para análisis posterior

for metodo, sol in soluciones.items():
    # Calcular energía para cada punto
    t_dense = np.linspace(sol.t[0], sol.t[-1], 500)
    datos = sol.sol(t_dense)
    u_vals = datos[0]
    v_vals = datos[1]

    energias = []
    for i in range(len(u_vals)):
        E = cmr.energia(u_vals[i], v_vals[i], derivadas_curva[0], g)
        energias.append(E)

    # Energía inicial
    E0 = energias[0]
    # Variación de energía
    delta_E = np.array(energias) - E0
    variaciones_energia[metodo] = delta_E

    ax3.plot(t_dense, delta_E, label=f'{metodo}', linewidth=2, alpha=0.8)

ax3.set_xlabel('Tiempo (s)', fontsize=11)
ax3.set_ylabel('ΔE (J/kg)', fontsize=11)
ax3.set_title('Variación de Energía vs Tiempo (ΔE  0)', fontsize=12,␣
 ↪fontweight='bold')
ax3.legend(loc='best', fontsize=9)
ax3.grid(True, alpha=0.3)
ax3.axhline(y=0, color='r', linestyle='--', linewidth=1.5, alpha=0.7)

# ================================================
# 4. FUERZA NORMAL vs TIEMPO en unidades de G (subplot inferior derecho)
# ================================================
ax4 = axes[1, 1]

# Graficar todos los métodos
for metodo, sol in soluciones.items():
    t_dense = np.linspace(sol.t[0], sol.t[-1], 500)
```

```
    datos = sol.sol(t_dense)
    u_vals = datos[0]
    v_vals = datos[1]

    fuerzas_normales_G = []

    for i in range(len(u_vals)):
        fuerzaN, baseLocal, ctes = cmr.fuerzaNormal(u_vals[i], v_vals[i],␣
 ↪derivadas_curva, g)
        # Convertir a unidades de G (1 G = 9.81 m/s²)
        fuerzaN_G = fuerzaN / g
        fuerzas_normales_G.append(fuerzaN_G)

    ax4.plot(t_dense, fuerzas_normales_G, linewidth=2, alpha=0.8, label=metodo)

ax4.set_xlabel('Tiempo (s)', fontsize=11)
ax4.set_ylabel('Fuerza Normal (G)', fontsize=11)
ax4.set_title('Fuerza Normal vs Tiempo', fontsize=12, fontweight='bold')
ax4.grid(True, alpha=0.3)
ax4.axhline(y=0, color='r', linestyle='--', linewidth=1, alpha=0.5, label='F_N␣
 ↪= 0')
ax4.axhline(y=1, color='orange', linestyle=':', linewidth=1, alpha=0.5,␣
 ↪label='1 G')
ax4.legend(loc='best', fontsize=9)

plt.tight_layout()
plt.show()

# ================================================
# ANÁLISIS:
# ================================================
print("\n" + "="*80)
print("ANÁLISIS DE CONSERVACIÓN DE ENERGÍA:")
print("="*80)

# Calcular la desviación máxima de energía para cada método
desviaciones = {}
for metodo, delta_E in variaciones_energia.items():
    desviacion_max = np.max(np.abs(delta_E))
    desviacion_media = np.mean(np.abs(delta_E))
    desviaciones[metodo] = (desviacion_max, desviacion_media)
    print(f"\n{metodo}:")
    print(f"  • Desviación máxima de energía: {desviacion_max:.6e} J/kg")
    print(f"  • Desviación media de energía:  {desviacion_media:.6e} J/kg")

# Identificar el mejor método
mejor_metodo = min(desviaciones.items(), key=lambda x: x[1][0])
```

```
print("\n" + "-"*80)
print(f"MÉTODO CON MENOR DESVIACIÓN: {mejor_metodo[0]}")
print(f"  → Desviación máxima: {mejor_metodo[1][0]:.6e} J/kg")
print("-"*80)
```




```
================================================================================
ANÁLISIS DE CONSERVACIÓN DE ENERGÍA:
================================================================================

RK45:
  • Desviación máxima de energía: 7.505167e+01 J/kg
  • Desviación media de energía:  2.281063e+01 J/kg

RK23:
  • Desviación máxima de energía: 5.791126e+00 J/kg
  • Desviación media de energía:  1.156529e+00 J/kg

DOP853:
  • Desviación máxima de energía: 7.620358e+01 J/kg
  • Desviación media de energía:  2.017535e+01 J/kg
```

```
Radau:
  • Desviación máxima de energía: 7.016928e-01 J/kg
  • Desviación media de energía:  1.846564e-01 J/kg

BDF:
  • Desviación máxima de energía: 1.978074e+01 J/kg
  • Desviación media de energía:  3.706252e+00 J/kg

LSODA:
  • Desviación máxima de energía: 1.285455e+01 J/kg
  • Desviación media de energía:  2.304471e+00 J/kg

--------------------------------------------------------------------------------
MÉTODO CON MENOR DESVIACIÓN: Radau
  → Desviación máxima: 7.016928e-01 J/kg
--------------------------------------------------------------------------------
```

Como pudimos ver y ya el código identifico Radau es el mejor método para la resolución de este problema siendo el que minimiza la variación de energía, y será por tanto el que usaremos de ahora en adelante para la resolución

2.5 *Caso con fricción*

Ahora estudiaremos el caso real, el caso con fricción, la sistemática será la misma, solo que ahora la energía no se conserva, y algunos términos que antes eran 0 ahora no lo son.

Trabajamos con la misma curva de antes, luego no tenemos que volver a hacer parametrizaciones ni nada por el estilo

2.5.1 *Generación del sistema Sedo*

[9]:
```python
g = 9.8 #m/s^2 #aceleración de la gravedad
m = 800 #kg #masa del vagón con 4 pasajeros a bordo
mu = 0.015 #coeficiente de fricción
c_a = 0.4 #coeficiente de resistencia aerodinámica
S_f = 2 #m^2 #superficie frontal del vagón
ro_a = 1.225 #kg/m^3 #densidad del aire
c_v = (c_a*S_f*ro_a)/(2*m) #m^{-1} #coeficiente de la fuerza viscosa

# Condiciones iniciales [parámetro u0, velocidad v0]
u0 = 0.0              # inicio de la curva
v0 = 10.0              # m/s - velocidad inicial
y0 = [u0, v0]

#============================================
# Sistema Sedo
#============================================
def edo_montaña_rusa_no_conservativa(t, y):
```

```
    '''
    Define el sistema de EDOs para la montaña rusa.

    Parámetros:
    -----------
    t : float
        Tiempo (no se usa explícitamente en este sistema)
    y : list
        Estado del sistema [posición u, velocidad v]

    Retorna:
    --------
    dydt : list
        Derivadas [du/dt, dv/dt]
    '''

    return cmr.edofun_mr(t, y, derivadas_curva, mu, c_v, g)


#Información del sistema
print("="*70)
print("SISTEMA EDO - CASO NO CONSERVATIVO")
print("="*70)
print(f"Parámetros físicos:")
print(f"  • Masa: {m} kg")
print(f"  • Coef. rozamiento  : {mu}")
print(f"  • Coef. arrastre ca: {c_a}")
print(f"  • Superficie frontal Sf: {S_f} m²")
print(f"  • Coeficiente viscoso cvefVis: {c_v} 1/m")
print(f"  • Gravedad: {g} m/s² (positivo)")
print(f"\nCondiciones iniciales:")
print(f"  • Posición inicial: u  = {u0}")
print(f"  • Velocidad inicial: v  = {v0} m/s")
```

```
======================================================================
SISTEMA EDO - CASO NO CONSERVATIVO
======================================================================
Parámetros físicos:
  • Masa: 800 kg
  • Coef. rozamiento  : 0.015
  • Coef. arrastre ca: 0.4
  • Superficie frontal Sf: 2 m²
  • Coeficiente viscoso cvefVis: 0.0006125000000000001 1/m
  • Gravedad: 9.8 m/s² (positivo)

Condiciones iniciales:
  • Posición inicial: u  = 0.0
  • Velocidad inicial: v  = 10.0 m/s
```

2.5.2 *Resolución del SEDO*

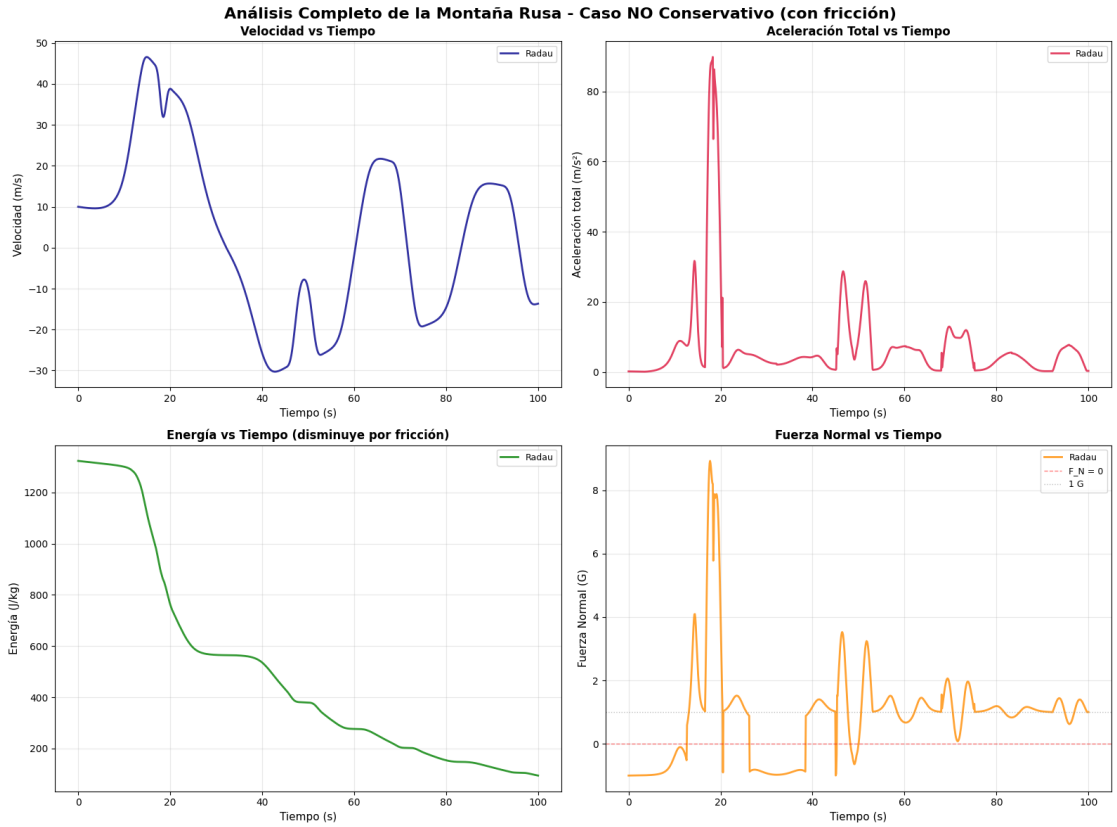
[10]:
```
# Tiempo de evaluación
t_eval = np.linspace(0, 100, 1000)

# Resolución del Sedo por el método Radau
sol =␣
 ↪solve_ivp(edo_montaña_rusa_no_conservativa,[0,100],y0,method="Radau",t_eval=t_eval,dense_ou
```

2.5.3 *Visualización de lo obtenido*

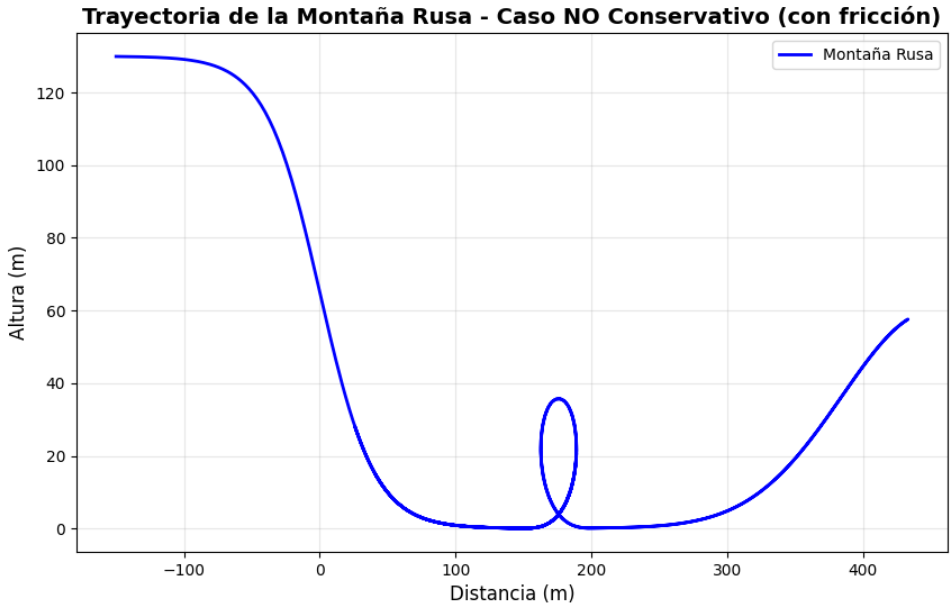
[11]:
```
# ================================================
# GRÁFICAS COMBINADAS: Análisis del movimiento con fricción
# ================================================

fig, axes = plt.subplots(2, 2, figsize=(16, 12))
fig.suptitle('Análisis Completo de la Montaña Rusa - Caso NO Conservativo (con␣
 ↪fricción)', fontsize=16, fontweight='bold')

# Generar puntos densos para cálculos
t_dense = np.linspace(sol.t[0], sol.t[-1], 1000)
datos = sol.sol(t_dense)
u_vals = datos[0]
v_vals = datos[1]

# ================================================
# 1. VELOCIDAD vs TIEMPO (subplot superior izquierdo)
# ================================================
ax1 = axes[0, 0]
ax1.plot(t_dense, v_vals, linewidth=2, color='darkblue', alpha=0.8,␣
 ↪label='Radau')
ax1.set_xlabel('Tiempo (s)', fontsize=11)
ax1.set_ylabel('Velocidad (m/s)', fontsize=11)
ax1.set_title('Velocidad vs Tiempo', fontsize=12, fontweight='bold')
ax1.legend(loc='best', fontsize=9)
ax1.grid(True, alpha=0.3)

# ================================================
# 2. ACELERACIÓN TOTAL vs TIEMPO (subplot superior derecho)
# ================================================
ax2 = axes[0, 1]

aceleraciones = []
for i in range(len(u_vals)):
    fuerzaN, baseLocal, ctes = cmr.fuerzaNormal(u_vals[i], v_vals[i],␣
 ↪derivadas_curva, g)
```

```
    ace_total, ace_tg, ace_nr = cmr.aceleracion(v_vals[i], baseLocal, mu, c_v,␣
 ↪fuerzaN, ctes[1], g)
    aceleraciones.append(ace_total)

ax2.plot(t_dense, aceleraciones, linewidth=2, color='crimson', alpha=0.8,␣
 ↪label='Radau')
ax2.set_xlabel('Tiempo (s)', fontsize=11)
ax2.set_ylabel('Aceleración total (m/s²)', fontsize=11)
ax2.set_title('Aceleración Total vs Tiempo', fontsize=12, fontweight='bold')
ax2.legend(loc='best', fontsize=9)
ax2.grid(True, alpha=0.3)

# ================================================
# 3. ENERGÍA vs TIEMPO (subplot inferior izquierdo)
# ================================================
ax3 = axes[1, 0]

energias = []
for i in range(len(u_vals)):
    E = cmr.energia(u_vals[i], v_vals[i], derivadas_curva[0], g)
    energias.append(E)

ax3.plot(t_dense, energias, linewidth=2, color='green', alpha=0.8,␣
 ↪label='Radau')
ax3.set_xlabel('Tiempo (s)', fontsize=11)
ax3.set_ylabel('Energía (J/kg)', fontsize=11)
ax3.set_title('Energía vs Tiempo (disminuye por fricción)', fontsize=12,␣
 ↪fontweight='bold')
ax3.legend(loc='best', fontsize=9)
ax3.grid(True, alpha=0.3)

# ================================================
# 4. FUERZA NORMAL vs TIEMPO en unidades de G (subplot inferior derecho)
# ================================================
ax4 = axes[1, 1]

fuerzas_normales_G = []

for i in range(len(u_vals)):
    fuerzaN, baseLocal, ctes = cmr.fuerzaNormal(u_vals[i], v_vals[i],␣
 ↪derivadas_curva, g)
    # Convertir a unidades de G (1 G = 9.81 m/s²)
    fuerzaN_G = fuerzaN / g
    fuerzas_normales_G.append(fuerzaN_G)

ax4.plot(t_dense, fuerzas_normales_G, linewidth=2, color='darkorange', alpha=0.
 ↪8, label='Radau')
```

```
ax4.set_xlabel('Tiempo (s)', fontsize=11)
ax4.set_ylabel('Fuerza Normal (G)', fontsize=11)
ax4.set_title('Fuerza Normal vs Tiempo', fontsize=12, fontweight='bold')
ax4.grid(True, alpha=0.3)
ax4.axhline(y=0, color='r', linestyle='--', linewidth=1, alpha=0.5, label='F_N␣
 ↪= 0')
ax4.axhline(y=1, color='gray', linestyle=':', linewidth=1, alpha=0.5, label='1␣
 ↪G')
ax4.legend(loc='best', fontsize=9)

plt.tight_layout()
plt.show()

# ================================================
# ANÁLISIS DE PÉRDIDA DE ENERGÍA
# ================================================
print("\n" + "="*80)
print("ANÁLISIS DE PÉRDIDA DE ENERGÍA - CASO CON FRICCIÓN")
print("="*80)
E_inicial = energias[0]
E_final = energias[-1]
perdida_energia = E_inicial - E_final
porcentaje_perdida = (perdida_energia / E_inicial) * 100

print(f"\nEnergía inicial:  {E_inicial:.2f} J/kg")
print(f"Energía final:    {E_final:.2f} J/kg")
print(f"Pérdida total:    {perdida_energia:.2f} J/kg ({porcentaje_perdida:.
 ↪2f}%)")
print(f"\nVelocidad inicial: {v_vals[0]:.2f} m/s")
print(f"Velocidad final:   {v_vals[-1]:.2f} m/s")
print(f"\n  La energía disminuye gradualmente debido a:")
print(f"    • Fricción con la pista (  = {mu})")
print(f"    • Resistencia aerodinámica (c_v = {c_v:.6f} m ¹)")
print("="*80 + "\n")

# Vista de la trayectoria 2D de la montaña rusa con fricción
plt.figure(figsize=(10,6))
plt.plot(Curva_parametrica(sol.sol(t_dense)[0])[:,0], Curva_parametrica(sol.
 ↪sol(t_dense)[0])[:,2], 'b-', linewidth=2, label='Montaña Rusa')
plt.title('Trayectoria de la Montaña Rusa - Caso NO Conservativo (con␣
 ↪fricción)', fontsize=14, fontweight='bold')
plt.xlabel('Distancia (m)', fontsize=12)
plt.ylabel('Altura (m)', fontsize=12)
plt.grid(True, alpha=0.3)
plt.legend()
plt.show()
```




```
================================================================================
ANÁLISIS DE PÉRDIDA DE ENERGÍA - CASO CON FRICCIÓN
================================================================================

Energía inicial:  1323.30 J/kg
Energía final:    94.28 J/kg
Pérdida total:    1229.01 J/kg (92.88%)

Velocidad inicial: 10.00 m/s
Velocidad final:   -13.67 m/s

  La energía disminuye gradualmente debido a:
    • Fricción con la pista (  = 0.015)
    • Resistencia aerodinámica (c_v = 0.000613 m ¹)
================================================================================
```



Como podemos ver esta montaña rusa no tiene viabilidad en la vida real, ya que a aparte de superar el umbral de fuerza Normal recomendada, que es de 5G, y ni si quiera termina el recorrido

2.6 *Animación: Sin Fricción vs Con Fricción*

[12]:
```
# Resolver ambos casos con el mismo tiempo de evaluación para comparación
t_max = 70  # tiempo máximo de simulación
t_eval_comp = np.linspace(0, t_max, 1000)

# Parámetros SIN fricción
mu_sin = 0.0
cv_sin = 0.0
v0_sin = 1.0
y0_sin = [0.0, v0_sin]



sol_sin_friccion = soluciones['Radau']

# Resolver CON fricción (ya tenemos estos parámetros definidos)
mu_con = 0.015
cv_con = c_v
v0_con = 10.0
y0_con = [0.0, v0_con]

sol_con_friccion = sol
```

```
print("="*70)
print("SOLUCIONES CALCULADAS PARA LA ANIMACIÓN")
print("="*70)
print(f"Caso SIN fricción:")
print(f"  •   = {mu_sin}, c_v = {cv_sin}")
print(f"  • v  = {v0_sin} m/s")
print(f"  • Tiempo final alcanzado: {sol_sin_friccion.t[-1]:.2f} s")
print(f"\nCaso CON fricción:")
print(f"  •   = {mu_con}, c_v = {cv_con:.6f} m ¹")
print(f"  • v  = {v0_con} m/s")
print(f"  • Tiempo final alcanzado: {sol_con_friccion.t[-1]:.2f} s")
print("="*70)
```

```
======================================================================
SOLUCIONES CALCULADAS PARA LA ANIMACIÓN
======================================================================
Caso SIN fricción:
  •   = 0.0, c_v = 0.0
  • v  = 1.0 m/s
  • Tiempo final alcanzado: 70.00 s

Caso CON fricción:
  •   = 0.015, c_v = 0.000613 m ¹
  • v  = 10.0 m/s
  • Tiempo final alcanzado: 100.00 s
======================================================================
```

[ ]:
```
# ================================================
# ANIMACIÓN COMPARATIVA: Sin Fricción vs Con Fricción
# ================================================

# Configurar matplotlib para animaciones en notebook (VS Code usa widget)
%matplotlib widget

# Crear figura para la animación
fig, (ax_sin, ax_con) = plt.subplots(1, 2, figsize=(12, 6))
fig.suptitle('Comparación: Movimiento con y sin Fricción', fontsize=14,␣
 ↪fontweight='bold')

# ================================================
# Subplot IZQUIERDO: SIN FRICCIÓN
# ================================================
ax_sin.plot(Distancia_data, Altura_data, 'k-', linewidth=2, alpha=0.3,␣
 ↪label='Pista')
vagon_sin, = ax_sin.plot([], [], 'ro', markersize=12, label='Vagón')
```

```
estela_sin, = ax_sin.plot([], [], 'r--', linewidth=1, alpha=0.4,␣
 ↪label='Trayectoria')
ax_sin.set_xlabel('Distancia (m)', fontsize=10)
ax_sin.set_ylabel('Altura (m)', fontsize=10)
ax_sin.set_title('SIN Fricción ( =0, c_v=0)', fontsize=11, fontweight='bold')
ax_sin.legend(loc='upper right', fontsize=8)
ax_sin.grid(True, alpha=0.3)
ax_sin.set_xlim(Distancia_data.min()-50, Distancia_data.max()+50)
ax_sin.set_ylim(Altura_data.min()-20, Altura_data.max()+20)

# Texto informativo para sin fricción
texto_sin = ax_sin.text(0.02, 0.98, '', transform=ax_sin.transAxes,
                        fontsize=9, verticalalignment='top',
                        bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.
 ↪8))

# ================================================
# Subplot DERECHO: CON FRICCIÓN
# ================================================
ax_con.plot(Distancia_data, Altura_data, 'k-', linewidth=2, alpha=0.3,␣
 ↪label='Pista')
vagon_con, = ax_con.plot([], [], 'bo', markersize=12, label='Vagón')
estela_con, = ax_con.plot([], [], 'b--', linewidth=1, alpha=0.4,␣
 ↪label='Trayectoria')
ax_con.set_xlabel('Distancia (m)', fontsize=10)
ax_con.set_ylabel('Altura (m)', fontsize=10)
ax_con.set_title(f'CON Fricción ( ={mu_con}, c_v={cv_con:.4f})', fontsize=11,␣
 ↪fontweight='bold')
ax_con.legend(loc='upper right', fontsize=8)
ax_con.grid(True, alpha=0.3)
ax_con.set_xlim(Distancia_data.min()-50, Distancia_data.max()+50)
ax_con.set_ylim(Altura_data.min()-20, Altura_data.max()+20)

# Texto informativo para con fricción
texto_con = ax_con.text(0.02, 0.98, '', transform=ax_con.transAxes,
                        fontsize=9, verticalalignment='top',
                        bbox=dict(boxstyle='round', facecolor='lightblue',␣
 ↪alpha=0.8))

# ================================================
# Preparar datos para la animación
# ================================================

# Obtener valores de u(t) y v(t) para ambos casos
t_max_sin = sol_sin_friccion.t[-1]
t_max_con = sol_con_friccion.t[-1]
```

```
t_max_anim = min(t_max_sin, t_max_con)  # Usar el mínimo para sincronizar

# Crear array de tiempos para la animación
n_frames = 500
t_anim = np.linspace(0, t_max_anim, n_frames)

# Obtener datos interpolados para cada frame
u_sin = sol_sin_friccion.sol(t_anim)[0]
v_sin = sol_sin_friccion.sol(t_anim)[1]

u_con = sol_con_friccion.sol(t_anim)[0]
v_con = sol_con_friccion.sol(t_anim)[1]

# Convertir u -> posiciones (x,y,z) usando la curva paramétrica
pos_sin = Curva_parametrica(u_sin)  # shape: (n_frames, 3)
pos_con = Curva_parametrica(u_con)

# Calcular energías para mostrar en la animación
energias_sin = [cmr.energia(u_sin[i], v_sin[i], derivadas_curva[0], g) for i in␣
 ↪range(n_frames)]
energias_con = [cmr.energia(u_con[i], v_con[i], derivadas_curva[0], g) for i in␣
 ↪range(n_frames)]

print(f"\nDatos preparados para animación:")
print(f"  • Número de frames: {n_frames}")
print(f"  • Duración: {t_max_anim:.2f} s")
print(f"  • Caso SIN fricción: energía inicial = {energias_sin[0]:.2f} J/kg")
print(f"  • Caso CON fricción: energía inicial = {energias_con[0]:.2f} J/kg")

# ================================================
# Función de inicialización
# ================================================
def init():
    vagon_sin.set_data([], [])
    estela_sin.set_data([], [])
    vagon_con.set_data([], [])
    estela_con.set_data([], [])
    texto_sin.set_text('')
    texto_con.set_text('')
    return vagon_sin, estela_sin, vagon_con, estela_con, texto_sin, texto_con

# ================================================
# Función de actualización de frames
# ================================================
def animate(frame):
    # Actualizar VAGÓN y ESTELA sin fricción
    x_sin = pos_sin[frame, 0]
```

```
    z_sin = pos_sin[frame, 2]
    vagon_sin.set_data([x_sin], [z_sin])

    # Estela (últimos 50 puntos)
    inicio_estela = max(0, frame-50)
    estela_sin.set_data(pos_sin[inicio_estela:frame+1, 0],␣
 ↪pos_sin[inicio_estela:frame+1, 2])

    # Texto informativo sin fricción
    info_sin = (f"Tiempo: {t_anim[frame]:.2f} s\n"
                f"Velocidad: {v_sin[frame]:.2f} m/s\n"
                f"Energía: {energias_sin[frame]:.2f} J/kg\n"
                f"Posición u: {u_sin[frame]:.3f}")
    texto_sin.set_text(info_sin)

    # Actualizar VAGÓN y ESTELA con fricción
    x_con = pos_con[frame, 0]
    z_con = pos_con[frame, 2]
    vagon_con.set_data([x_con], [z_con])

    # Estela (últimos 50 puntos)
    estela_con.set_data(pos_con[inicio_estela:frame+1, 0],␣
 ↪pos_con[inicio_estela:frame+1, 2])

    # Texto informativo con fricción
    info_con = (f"Tiempo: {t_anim[frame]:.2f} s\n"
                f"Velocidad: {v_con[frame]:.2f} m/s\n"
                f"Energía: {energias_con[frame]:.2f} J/kg\n"
                f"Posición u: {u_con[frame]:.3f}")
    texto_con.set_text(info_con)

    return vagon_sin, estela_sin, vagon_con, estela_con, texto_sin, texto_con

# ================================================
# Crear animación
# ================================================
anim = FuncAnimation(fig, animate, init_func=init, frames=n_frames,
                     interval=20, blit=True, repeat=True)

plt.tight_layout()
plt.show()
```

[ ]:
```
# ================================================
# Convertir animación a HTML5 video
# ================================================

from IPython.display import HTML
```

```
# Convertir la animación a video HTML5 embebido
# Esto funciona en HTML exportado
html_video = anim.to_html5_video()
html_anim = HTML(html_video)

# Mostrar en el notebook (también funcionará en HTML exportado)
display(html_anim)
```

2.7 *Caso realmente Viable*

Al ver que en el caso anteriror no era viable, modificaremos nuestra montaña rusa para que se ajuste a los parámetros ideales, que son:

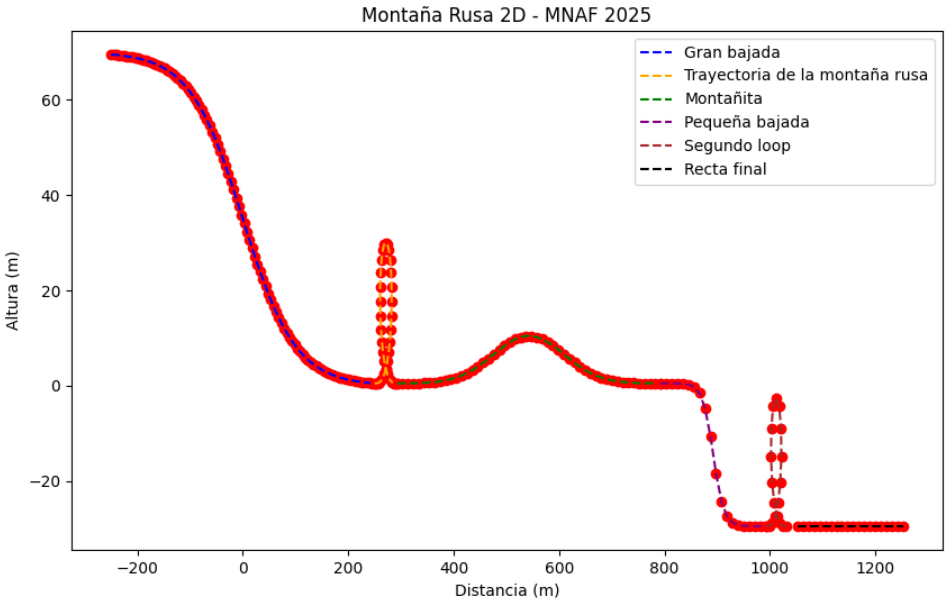
- Completar el recorrido
- Fuerza normal máxima menor que 5G

2.7.1 *Defínción de la nueva trayectoria*

[15]:
```
# Definimos la gran bajada de la montaña rusa
Gran_bajada = cmr.curva3d('s',np.linspace(-250,250,100),A=70,args=[0.02])

# Definimos el primer loop de la montaña rusa
Loop1 = cmr.curva3d('l',np.linspace(0,1.
 ↪79,20),A=33,C=[Gran_bajada[0][-1],0,Gran_bajada[2][-1]],args=[0])

# Definimos montañita (gaussina)
Montañita = cmr.curva3d('g',np.
 ↪linspace(-250,250,50),A=10,C=[Loop1[0][-1]+250,0,Loop1[2][-1]],args=[95])

# Definimos la pequeña bajada
Pequeña_bajada = cmr.curva3d('s',np.
 ↪linspace(-100,100,20),A=30,C=[Montañita[0][-1]+100,0,Montañita[2][-1]-30],args=[0.
 ↪1])

# Definimos el segundo loop de la montaña rusa
Loop2 = cmr.curva3d('l',np.linspace(0,1.
 ↪79,10),A=30,C=[Pequeña_bajada[0][-1],0,Pequeña_bajada[2][-1]],args=[0])

# Recta final frenado
Recta_final = [Pequeña_bajada[0][-1]+np.linspace(0,200,20)+60,np.zeros(20),np.
 ↪full(20,Loop2[2][-1])]
```

2.7.2 *Visualización de la nueva trayectroia*

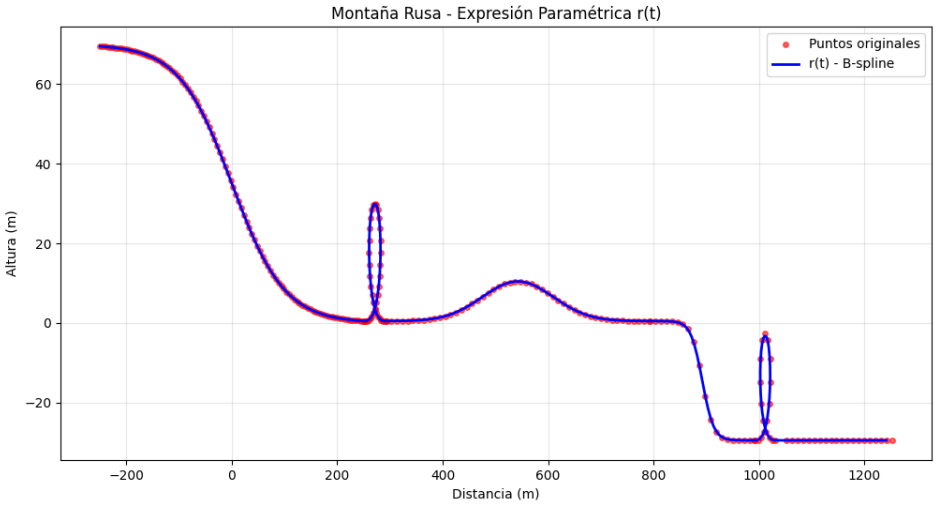
[16]:
```
# ============================================
# Gráfica de la montaña rusa
# ============================================

plt.figure(figsize=(10,6))

# Graficar la gran bajada
plt.plot(Gran_bajada[0], Gran_bajada[2], 'o', color='red')
plt.plot(Gran_bajada[0], Gran_bajada[2], '--', color='blue', label='Gran␣
 ↪bajada')

# Graficar el primer loop
plt.plot(Loop1[0], Loop1[2], 'o', color='red')
plt.plot(Loop1[0], Loop1[2], '--', color='orange', label='Trayectoria de la␣
 ↪montaña rusa')

# Graficar la montañita
plt.plot(Montañita[0], Montañita[2], 'o', color='red')
plt.plot(Montañita[0], Montañita[2], '--', color='green', label='Montañita')

# Graficar la pequeña bajada
plt.plot(Pequeña_bajada[0], Pequeña_bajada[2], 'o', color='red')
plt.plot(Pequeña_bajada[0], Pequeña_bajada[2], '--', color='purple',␣
 ↪label='Pequeña bajada')

# Graficar el segundo loop
plt.plot(Loop2[0], Loop2[2], 'o', color='red')
plt.plot(Loop2[0], Loop2[2], '--', color='brown', label='Segundo loop')

# Graficar la recta final
plt.plot(Recta_final[0], Recta_final[2], 'o', color='red')
plt.plot(Recta_final[0], Recta_final[2], '--', color='black', label='Recta␣
 ↪final')

# Configuraciones de la gráfica
plt.legend()
plt.title('Montaña Rusa 2D - MNAF 2025')
plt.xlabel('Distancia (m)')
plt.ylabel('Altura (m)')
plt.show()
```



2.7.3 *Construcción y visualización de la curva paramétrica*

[17]:
```
# Visualización usando la expresión paramétrica r(t)

#Construimos la expresión
Curva_parametrica = montaña_rusa_parametrica(
    X_data = np.concatenate([Gran_bajada[0], Loop1[0], Montañita[0],␣
 ↪Pequeña_bajada[0], Loop2[0], Recta_final[0]]),
    Z_data = np.concatenate([Gran_bajada[2], Loop1[2], Montañita[2],␣
 ↪Pequeña_bajada[2], Loop2[2], Recta_final[2]]),
)

# Evaluamos
t_eval = np.linspace(0, 1, 500)
Puntos_de_interpolación=Curva_parametrica(t_eval)

plt.figure(figsize=(12, 6))

# Gráfica comparativa
plt.plot(np.concatenate([Gran_bajada[0], Loop1[0], Montañita[0],␣
 ↪Pequeña_bajada[0], Loop2[0], Recta_final[0]]),
         np.concatenate([Gran_bajada[2], Loop1[2], Montañita[2],␣
 ↪Pequeña_bajada[2], Loop2[2], Recta_final[2]]),
         'o', color='red', markersize=4, label='Puntos originales', alpha=0.6)
```

```
plt.plot(Puntos_de_interpolación[:,0], Puntos_de_interpolación[:,2], '-',␣
 ↪color='blue', linewidth=2, label='r(t) - B-spline')

plt.legend()
plt.title('Montaña Rusa - Expresión Paramétrica r(t)')
plt.xlabel('Distancia (m)')
plt.ylabel('Altura (m)')
plt.grid(True, alpha=0.3)
plt.show()
```



2.7.4 *Generación del sistema SEDO*

[18]:
```
g = 9.8 #m/s^2 #aceleración de la gravedad
m = 800 #kg #masa del vagón con 4 pasajeros a bordo
mu = 0.015 #coeficiente de fricción
c_a = 0.4 #coeficiente de resistencia aerodinámica
S_f = 2 #m^2 #superficie frontal del vagón
ro_a = 1.225 #kg/m^3 #densidad del aire
c_v = (c_a*S_f*ro_a)/(2*m) #m^{-1} #coeficiente de la fuerza viscosa

# Condiciones iniciales [parámetro u0, velocidad v0]
u0 = 0.0              # inicio de la curva
v0 = 7.0              # m/s - velocidad inicial
y0 = [u0, v0]

#===========================================
# Sistema Sedo
#===========================================
```

```python
# Calcular derivadas de la curva usando pkgmrusa
derivadas_curva = cmr.trayec_der(Curva_parametrica)


def edo_montaña_rusa_no_conservativa(t, y):
    '''
    Define el sistema de EDOs para la montaña rusa.

    Parámetros:
    -----------
    t : float
        Tiempo (no se usa explícitamente en este sistema)
    y : list
        Estado del sistema [posición u, velocidad v]

    Retorna:
    --------
    dydt : list
        Derivadas [du/dt, dv/dt]
    '''

    return cmr.edofun_mr(t, y, derivadas_curva, mu, c_v, g)


#Información del sistema
print("="*70)
print("SISTEMA EDO - CASO NO CONSERVATIVO")
print("="*70)
print(f"Parámetros físicos:")
print(f"  • Masa: {m} kg")
print(f"  • Coef. rozamiento  : {mu}")
print(f"  • Coef. arrastre ca: {c_a} (SIN ROZAMIENTO)")
print(f"  • Superficie frontal Sf: {S_f} m²")
print(f"  • Coeficiente viscoso cvefVis: {c_v} 1/m")
print(f"  • Gravedad: {g} m/s² (positivo)")
print(f"\nCondiciones iniciales:")
print(f"  • Posición inicial: u  = {u0}")
print(f"  • Velocidad inicial: v  = {v0} m/s")
```

```
======================================================================
SISTEMA EDO - CASO NO CONSERVATIVO
======================================================================
Parámetros físicos:
  • Masa: 800 kg
  • Coef. rozamiento  : 0.015
  • Coef. arrastre ca: 0.4 (SIN ROZAMIENTO)
  • Superficie frontal Sf: 2 m²
```

- Coeficiente viscoso cvefVis: 0.0006125000000000001 1/m
- Gravedad: 9.8 m/s² (positivo)

Condiciones iniciales:
- Posición inicial: u = 0.0
- Velocidad inicial: v = 7.0 m/s

2.7.5 *Resolución del sistema SEDO*

[19]:
```
# Tiempo de evaluación
t_eval = np.linspace(0, 100, 1000)

# Resolución del Sedo por el método Radau
sol_viable = solve_ivp(edo_montaña_rusa_no_conservativa,[0,100],y0,method="Radau",t_eval=t_eval,dense_ou
```

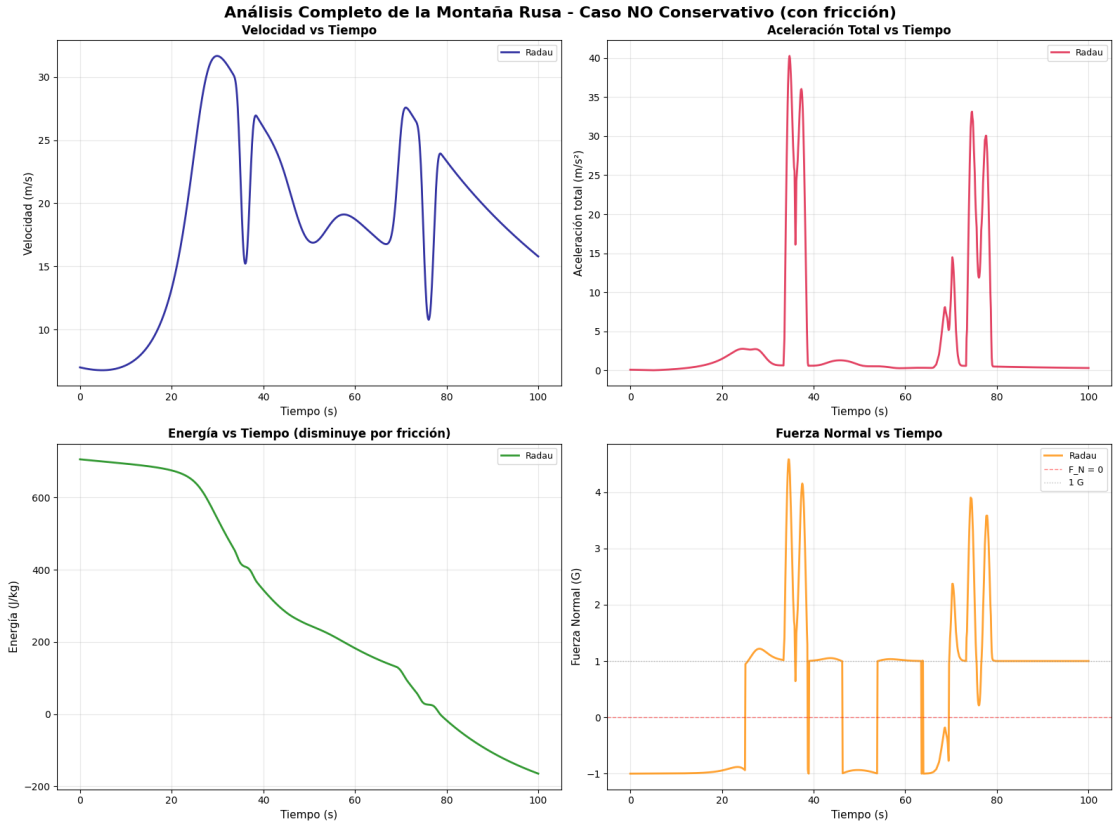
2.7.6 *Analisis de lo obtenido*

[20]:
```
# ================================================
# GRÁFICAS COMBINADAS: Análisis del movimiento con fricción
# ================================================

fig, axes = plt.subplots(2, 2, figsize=(16, 12))
fig.suptitle('Análisis Completo de la Montaña Rusa - Caso NO Conservativo (con fricción)', fontsize=16, fontweight='bold')

# Generar puntos densos para cálculos
t_dense = np.linspace(sol_viable.t[0], sol_viable.t[-1], 1000)
datos = sol_viable.sol(t_dense)
u_vals = datos[0]
v_vals = datos[1]

# ================================================
# 1. VELOCIDAD vs TIEMPO (subplot superior izquierdo)
# ================================================
ax1 = axes[0, 0]
ax1.plot(t_dense, v_vals, linewidth=2, color='darkblue', alpha=0.8, label='Radau')
ax1.set_xlabel('Tiempo (s)', fontsize=11)
ax1.set_ylabel('Velocidad (m/s)', fontsize=11)
ax1.set_title('Velocidad vs Tiempo', fontsize=12, fontweight='bold')
ax1.legend(loc='best', fontsize=9)
ax1.grid(True, alpha=0.3)

# ================================================
# 2. ACELERACIÓN TOTAL vs TIEMPO (subplot superior derecho)
# ================================================
```

```
ax2 = axes[0, 1]

aceleraciones = []
for i in range(len(u_vals)):
    fuerzaN, baseLocal, ctes = cmr.fuerzaNormal(u_vals[i], v_vals[i],␣
 ↪derivadas_curva, g)
    ace_total, ace_tg, ace_nr = cmr.aceleracion(v_vals[i], baseLocal, mu, c_v,␣
 ↪fuerzaN, ctes[1], g)
    aceleraciones.append(ace_total)

ax2.plot(t_dense, aceleraciones, linewidth=2, color='crimson', alpha=0.8,␣
 ↪label='Radau')
ax2.set_xlabel('Tiempo (s)', fontsize=11)
ax2.set_ylabel('Aceleración total (m/s²)', fontsize=11)
ax2.set_title('Aceleración Total vs Tiempo', fontsize=12, fontweight='bold')
ax2.legend(loc='best', fontsize=9)
ax2.grid(True, alpha=0.3)

# ================================================
# 3. ENERGÍA vs TIEMPO (subplot inferior izquierdo)
# ================================================
ax3 = axes[1, 0]

energias = []
for i in range(len(u_vals)):
    E = cmr.energia(u_vals[i], v_vals[i], derivadas_curva[0], g)
    energias.append(E)

ax3.plot(t_dense, energias, linewidth=2, color='green', alpha=0.8,␣
 ↪label='Radau')
ax3.set_xlabel('Tiempo (s)', fontsize=11)
ax3.set_ylabel('Energía (J/kg)', fontsize=11)
ax3.set_title('Energía vs Tiempo (disminuye por fricción)', fontsize=12,␣
 ↪fontweight='bold')
ax3.legend(loc='best', fontsize=9)
ax3.grid(True, alpha=0.3)

# ================================================
# 4. FUERZA NORMAL vs TIEMPO en unidades de G (subplot inferior derecho)
# ================================================
ax4 = axes[1, 1]

fuerzas_normales_G = []

for i in range(len(u_vals)):
    fuerzaN, baseLocal, ctes = cmr.fuerzaNormal(u_vals[i], v_vals[i],␣
 ↪derivadas_curva, g)
```

```
    # Convertir a unidades de G (1 G = 9.81 m/s²)
    fuerzaN_G = fuerzaN / g
    fuerzas_normales_G.append(fuerzaN_G)

ax4.plot(t_dense, fuerzas_normales_G, linewidth=2, color='darkorange', alpha=0.
 ↪8, label='Radau')
ax4.set_xlabel('Tiempo (s)', fontsize=11)
ax4.set_ylabel('Fuerza Normal (G)', fontsize=11)
ax4.set_title('Fuerza Normal vs Tiempo', fontsize=12, fontweight='bold')
ax4.grid(True, alpha=0.3)
ax4.axhline(y=0, color='r', linestyle='--', linewidth=1, alpha=0.5, label='F_N␣
 ↪= 0')
ax4.axhline(y=1, color='gray', linestyle=':', linewidth=1, alpha=0.5, label='1␣
 ↪G')
ax4.legend(loc='best', fontsize=9)

plt.tight_layout()
plt.show()

# ================================================
# ANÁLISIS DE PÉRDIDA DE ENERGÍA
# ================================================
print("\n" + "="*80)
print("ANÁLISIS DE PÉRDIDA DE ENERGÍA - CASO CON FRICCIÓN")
print("="*80)
E_inicial = energias[0]
E_final = energias[-1]
perdida_energia = E_inicial - E_final
porcentaje_perdida = (perdida_energia / E_inicial) * 100

print(f"\nEnergía inicial:  {E_inicial:.2f} J/kg")
print(f"Energía final:    {E_final:.2f} J/kg")
print(f"Pérdida total:    {perdida_energia:.2f} J/kg ({porcentaje_perdida:.
 ↪2f}%)")
print(f"\nVelocidad inicial: {v_vals[0]:.2f} m/s")
print(f"Velocidad final:   {v_vals[-1]:.2f} m/s")
print(f"\n  La energía disminuye gradualmente debido a:")
print(f"    • Fricción con la pista ( = {mu})")
print(f"    • Resistencia aerodinámica (c_v = {c_v:.6f} m ¹)")
print("="*80 + "\n")


plt.figure(figsize=(10,6))
plt.plot(Curva_parametrica(sol_viable.sol(t_dense)[0])[:,0],␣
 ↪Curva_parametrica(sol_viable.sol(t_dense)[0])[:,2], 'b-', linewidth=2,␣
 ↪label='Montaña Rusa')
```

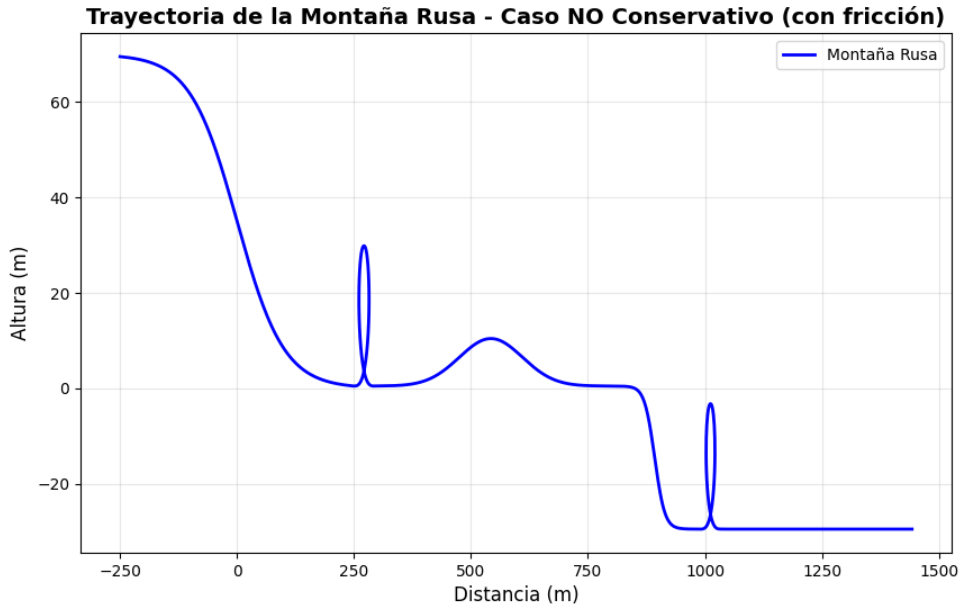
```
plt.title('Trayectoria de la Montaña Rusa - Caso NO Conservativo (con␣
  ↪fricción)', fontsize=14, fontweight='bold')
plt.xlabel('Distancia (m)', fontsize=12)
plt.ylabel('Altura (m)', fontsize=12)
plt.grid(True, alpha=0.3)
plt.legend()
plt.show()
```




```
================================================================================
ANÁLISIS DE PÉRDIDA DE ENERGÍA - CASO CON FRICCIÓN
================================================================================

Energía inicial:  705.91 J/kg
Energía final:    -164.70 J/kg
Pérdida total:    870.61 J/kg (123.33%)

Velocidad inicial: 7.00 m/s
Velocidad final:   15.79 m/s

  La energía disminuye gradualmente debido a:
```

- Fricción con la pista (= 0.015)
- Resistencia aerodinámica (c_v = 0.000613 m ¹)

==========



Como podemos ver en este caso si que cumple con los parámetros que planteamos como ideales, por lo tanto sería una montaña rusa perfectamente viable

2.7.7 *Animación de la trayectoria*

[ ]:
```
# ============================================
# ANIMACIÓN DEL CASO VIABLE
# ============================================

# Configurar matplotlib para animaciones en notebook
%matplotlib widget

# Crear figura para la animación
fig_viable, ax_viable = plt.subplots(figsize=(14, 7))
fig_viable.suptitle('Montaña Rusa - Caso Viable con Fricción', fontsize=14,␣
 ↪fontweight='bold')

# Concatenar datos completos de la pista
Distancia_data_viable = np.concatenate([Gran_bajada[0], Loop1[0], Montañita[0],␣
 ↪Pequeña_bajada[0], Loop2[0], Recta_final[0]])
```

```
Altura_data_viable = np.concatenate([Gran_bajada[2], Loop1[2], Montañita[2],␣
 ↪Pequeña_bajada[2], Loop2[2], Recta_final[2]])

# Dibujar la pista
ax_viable.plot(Distancia_data_viable, Altura_data_viable, 'k-', linewidth=2,␣
 ↪alpha=0.3, label='Pista')
vagon_viable, = ax_viable.plot([], [], 'go', markersize=14, label='Vagón',␣
 ↪zorder=5)
estela_viable, = ax_viable.plot([], [], 'g--', linewidth=1.5, alpha=0.5,␣
 ↪label='Trayectoria')

ax_viable.set_xlabel('Distancia (m)', fontsize=11)
ax_viable.set_ylabel('Altura (m)', fontsize=11)
ax_viable.set_title(f'  = {mu}, c_v = {c_v:.6f} m ¹, v  = {v0} m/s', fontsize=11)
ax_viable.legend(loc='upper right', fontsize=10)
ax_viable.grid(True, alpha=0.3)
ax_viable.set_xlim(Distancia_data_viable.min()-50, Distancia_data_viable.
 ↪max()+50)
ax_viable.set_ylim(Altura_data_viable.min()-20, Altura_data_viable.max()+20)

# Texto informativo
texto_viable = ax_viable.text(0.02, 0.98, '', transform=ax_viable.transAxes,
                              fontsize=10, verticalalignment='top',
                              bbox=dict(boxstyle='round',␣
 ↪facecolor='lightgreen', alpha=0.8))

# ============================================
# Preparar datos para la animación
# ============================================

t_max_viable = sol_viable.t[-1]
n_frames_viable = 500
t_anim_viable = np.linspace(0, t_max_viable, n_frames_viable)

# Obtener datos interpolados
u_viable = sol_viable.sol(t_anim_viable)[0]
v_viable = sol_viable.sol(t_anim_viable)[1]

# Convertir u -> posiciones (x,y,z)
pos_viable = Curva_parametrica(u_viable)

# Calcular energías y fuerzas
energias_viable = []
fuerzas_G_viable = []

for i in range(n_frames_viable):
```

```
    E = cmr.energia(u_viable[i], v_viable[i], derivadas_curva[0], g)
    energias_viable.append(E)

    fuerzaN, baseLocal, ctes = cmr.fuerzaNormal(u_viable[i], v_viable[i],␣
 ↪derivadas_curva, g)
    fuerzas_G_viable.append(fuerzaN / g)

print(f"\nDatos preparados para animación del caso viable:")
print(f"  • Número de frames: {n_frames_viable}")
print(f"  • Duración: {t_max_viable:.2f} s")
print(f"  • Energía inicial: {energias_viable[0]:.2f} J/kg")
print(f"  • Energía final: {energias_viable[-1]:.2f} J/kg")
print(f"  • Pérdida de energía: {energias_viable[0] - energias_viable[-1]:.2f}␣
 ↪J/kg")

# ============================================
# Funciones de animación
# ============================================

def init_viable():
    vagon_viable.set_data([], [])
    estela_viable.set_data([], [])
    texto_viable.set_text('')
    return vagon_viable, estela_viable, texto_viable

def animate_viable(frame):
    # Actualizar vagón
    x_viable = pos_viable[frame, 0]
    z_viable = pos_viable[frame, 2]
    vagon_viable.set_data([x_viable], [z_viable])

    # Estela (últimos 80 puntos)
    inicio_estela = max(0, frame-80)
    estela_viable.set_data(pos_viable[inicio_estela:frame+1, 0],
                           pos_viable[inicio_estela:frame+1, 2])

    # Texto informativo
    info = (f"  Tiempo: {t_anim_viable[frame]:.2f} s\n"
            f"  Velocidad: {v_viable[frame]:.2f} m/s\n"
            f"  Energía: {energias_viable[frame]:.2f} J/kg\n"
            f"  Fuerza Normal: {fuerzas_G_viable[frame]:.2f} G\n"
            f"  Posición: {u_viable[frame]:.3f}")
    texto_viable.set_text(info)

    return vagon_viable, estela_viable, texto_viable

# Crear animación
```

```
anim_viable = FuncAnimation(fig_viable, animate_viable, init_func=init_viable,
                            frames=n_frames_viable, interval=20, blit=True,␣
 ↪repeat=True)

plt.tight_layout()
plt.show()
```

[ ]:
```
# ============================================
# Convertir a HTML5 video (funciona en HTML exportado)
# ============================================

html_video_viable = anim_viable.to_html5_video()
html_anim_viable = HTML(html_video_viable)

# Mostrar en el notebook (también funcionará en HTML exportado)
display(html_anim_viable)
```

2.8 *Parámetros válidos*

Una vez conocido el caso viable nos interesa saber también en que condiciones, si alguno de los otros parametros de definición del problema cambiase en que intevalo sería viable.

Es decir para esta parte mantendremos constante el resto de parametros haciendo variar uno de ellos para encontrar los intervalos de viabilidad

2.8.1 *Restricciones del diseño*

Para que la montaña rusa sea viable, debe cumplir con las siguientes restricciones:

1. **Fuerza Normal máxima**: $F_N \leq 5g$ (limitación de seguridad y comodidad)
2. **Completar el recorrido**: El vagón debe llegar al final con velocidad $v > 0$

A continuación estudiaremos los intervalos válidos para cada parámetro: {m, , ca, Sf, v }

2.8.2 *Función de estudio de parámetros validos*

[23]:
```
# ============================================
# ESTUDIO DE PARÁMETROS VÁLIDOS
# ============================================

def simular_montaña(m_val, mu_val, ca_val, Sf_val, v0_val, tiempo_max=100):
    """
    Simula la montaña rusa con los parámetros dados y verifica las condiciones.

    Retorna:
    --------
    dict con:
        - 'valido': bool (si cumple ambas condiciones)
        - 'completa_recorrido': bool
```

```
        - 'FN_max_G': float (fuerza normal máxima en unidades de G)
        - 'u_final': float (posición final alcanzada)
        - 'sol': objeto solución (si fue exitosa)
    """

    # Calcular coeficiente viscoso
    c_v_sim = (ca_val * Sf_val * ro_a) / (2 * m_val)

    # Condiciones iniciales
    y0_sim = [0.0, v0_val]

    # Sistema EDO
    def edo_sim(t, y):
        return cmr.edofun_mr(t, y, derivadas_curva, mu_val, c_v_sim, g)

    # Resolver
    try:
        sol_sim = solve_ivp(edo_sim, [0, tiempo_max], y0_sim,
                            method="Radau", dense_output=True,
                            events=[cmr.finalVia,cmr.paradaVagon])  # Se␣
↪detiene si velocidad llega a 0

        # Verificar si completó el recorrido (u_final >= 0.95)
        u_final = sol_sim.sol(sol_sim.t[-1])[0]
        completa = u_final >= 0.95

        # Calcular fuerza normal máxima
        t_check = np.linspace(sol_sim.t[0], sol_sim.t[-1], 500)
        datos_check = sol_sim.sol(t_check)
        u_check = datos_check[0]
        v_check = datos_check[1]

        FN_max = 0
        for i in range(len(u_check)):
            if u_check[i] >= 1.0:  # No revisar más allá del recorrido
                break
            fuerzaN, _, _ = cmr.fuerzaNormal(u_check[i], v_check[i],␣
↪derivadas_curva, g)
            FN_G = fuerzaN / g
            if FN_G > FN_max:
                FN_max = FN_G

        # Verificar condiciones
        FN_ok = FN_max <= 5.0
        valido = completa and FN_ok

        return {
```

```
            'valido': valido,
            'completa_recorrido': completa,
            'FN_max_G': FN_max,
            'u_final': u_final,
            'sol': sol_sim,
            'FN_ok': FN_ok
        }

    except Exception as e:
        return {
            'valido': False,
            'completa_recorrido': False,
            'FN_max_G': np.inf,
            'u_final': 0,
            'sol': None,
            'FN_ok': False,
            'error': str(e)
        }

print("="*80)
print("FUNCIÓN DE SIMULACIÓN CREADA")
print("="*80)
print("Condiciones para validez:")
print("  1. Completar el recorrido: u_final   0.95")
print("  2. Fuerza normal máxima: FN_max   5g")
print("="*80)
```

```
================================================================================
FUNCIÓN DE SIMULACIÓN CREADA
================================================================================
Condiciones para validez:
  1. Completar el recorrido: u_final   0.95
  2. Fuerza normal máxima: FN_max   5g
================================================================================
```

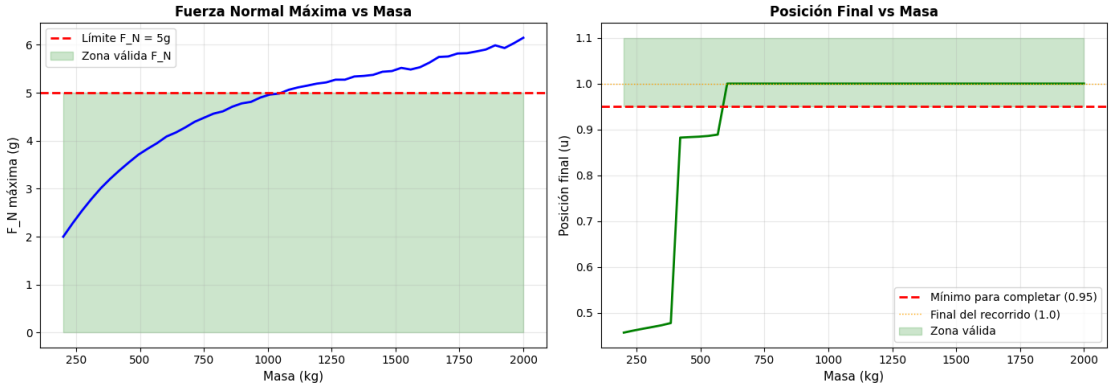
2.8.3 *Variación de Masa*

[24]:
```
# ============================================
# ESTUDIO 1: Variación de MASA (m)
# ============================================

# Parámetros base (del caso viable)
mu_base = 0.015
ca_base = 0.4
Sf_base = 2.0
v0_base = 7.0

# Rango de masas a explorar (kg)
```

```
masas = np.linspace(200, 2000, 50)

resultados_masa = []
for m_test in masas:
    resultado = simular_montaña(m_test, mu_base, ca_base, Sf_base, v0_base)
    resultados_masa.append({
        'm': m_test,
        'valido': resultado['valido'],
        'completa': resultado['completa_recorrido'],
        'FN_max': resultado['FN_max_G'],
        'u_final': resultado['u_final']
    })

# Análisis de resultados
masas_validas = [r['m'] for r in resultados_masa if r['valido']]
masas_completan = [r['m'] for r in resultados_masa if r['completa']]
masas_FN_ok = [r['m'] for r in resultados_masa if r['FN_max'] <= 5.0]

print("\n" + "="*80)
print("ESTUDIO 1: VARIACIÓN DE MASA (m)")
print("="*80)
print(f"Parámetros fijos:  ={mu_base}, c_a={ca_base}, S_f={Sf_base} m²,␣
 ↪v ={v0_base} m/s")
print(f"\nRango explorado: {masas.min():.0f} - {masas.max():.0f} kg")
if masas_validas:
    print(f"  Rango válido: {min(masas_validas):.1f} - {max(masas_validas):.1f}␣
 ↪kg")
else:
    print("  No hay valores válidos en el rango explorado")
print("="*80)

# Gráfica
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# FN_max vs masa
FN_values = [r['FN_max'] for r in resultados_masa]
masas_array = [r['m'] for r in resultados_masa]
ax1.plot(masas_array, FN_values, 'b-', linewidth=2)
ax1.axhline(y=5, color='r', linestyle='--', linewidth=2, label='Límite F_N =␣
 ↪5g')
ax1.fill_between(masas_array, 0, 5, alpha=0.2, color='green', label='Zona␣
 ↪válida F_N')
ax1.set_xlabel('Masa (kg)', fontsize=11)
ax1.set_ylabel('F_N máxima (g)', fontsize=11)
ax1.set_title('Fuerza Normal Máxima vs Masa', fontsize=12, fontweight='bold')
ax1.legend()
ax1.grid(True, alpha=0.3)
```

```python
# Posición final vs masa
u_final_values = [r['u_final'] for r in resultados_masa]
ax2.plot(masas_array, u_final_values, 'g-', linewidth=2)
ax2.axhline(y=0.95, color='r', linestyle='--', linewidth=2, label='Mínimo para␣
 ↪completar (0.95)')
ax2.axhline(y=1.0, color='orange', linestyle=':', linewidth=1, label='Final del␣
 ↪recorrido (1.0)')
ax2.fill_between(masas_array, 0.95, 1.1, alpha=0.2, color='green', label='Zona␣
 ↪válida')
ax2.set_xlabel('Masa (kg)', fontsize=11)
ax2.set_ylabel('Posición final (u)', fontsize=11)
ax2.set_title('Posición Final vs Masa', fontsize=12, fontweight='bold')
ax2.legend()
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

```
================================================================================
ESTUDIO 1: VARIACIÓN DE MASA (m)
================================================================================
Parámetros fijos:  =0.015, c_a=0.4, S_f=2.0 m², v =7.0 m/s

Rango explorado: 200 - 2000 kg
  Rango válido: 604.1 - 1044.9 kg
================================================================================
```

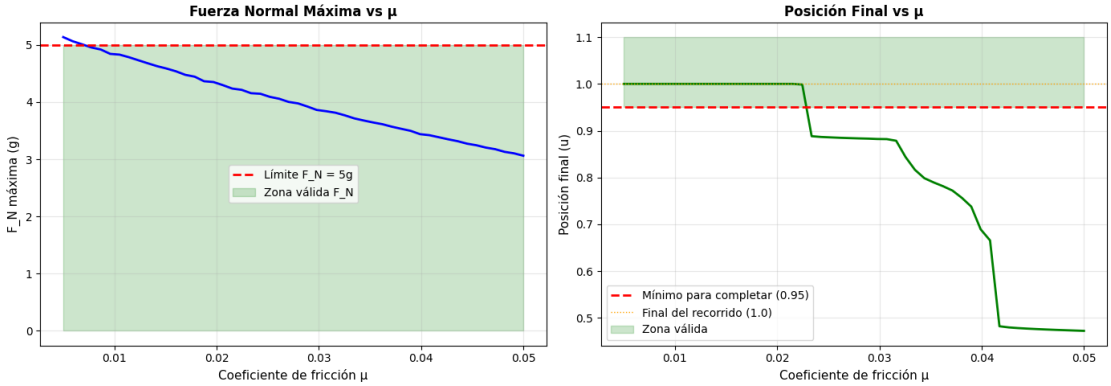


2.8.4 *Variación coef de fricción*

[25]:
```
# ============================================
# ESTUDIO 2: Variación de COEFICIENTE DE FRICCIÓN ( )
# ============================================

# Parámetros base
m_base = 800
ca_base = 0.4
Sf_base = 2.0
v0_base = 7.0

# Rango de   a explorar
mu_values = np.linspace(0.005, 0.05, 50)

resultados_mu = []
for mu_test in mu_values:
    resultado = simular_montaña(m_base, mu_test, ca_base, Sf_base, v0_base)
    resultados_mu.append({
        'mu': mu_test,
        'valido': resultado['valido'],
        'completa': resultado['completa_recorrido'],
        'FN_max': resultado['FN_max_G'],
        'u_final': resultado['u_final']
    })

# Análisis
mu_validos = [r['mu'] for r in resultados_mu if r['valido']]

print("\n" + "="*80)
print("ESTUDIO 2: VARIACIÓN DE COEFICIENTE DE FRICCIÓN ( )")
print("="*80)
print(f"Parámetros fijos: m={m_base} kg, c_a={ca_base}, S_f={Sf_base} m²,␣
 ↪v ={v0_base} m/s")
print(f"\nRango explorado: {mu_values.min():.4f} - {mu_values.max():.4f}")
if mu_validos:
    print(f"  Rango válido: {min(mu_validos):.4f} - {max(mu_validos):.4f}")
else:
    print("  No hay valores válidos en el rango explorado")
print("="*80)

# Gráfica
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# FN_max vs
FN_values = [r['FN_max'] for r in resultados_mu]
mu_array = [r['mu'] for r in resultados_mu]
```

```
ax1.plot(mu_array, FN_values, 'b-', linewidth=2)
ax1.axhline(y=5, color='r', linestyle='--', linewidth=2, label='Límite F_N =␣
 ↪5g')
ax1.fill_between(mu_array, 0, 5, alpha=0.2, color='green', label='Zona válida␣
 ↪F_N')
ax1.set_xlabel('Coeficiente de fricción  ', fontsize=11)
ax1.set_ylabel('F_N máxima (g)', fontsize=11)
ax1.set_title('Fuerza Normal Máxima vs  ', fontsize=12, fontweight='bold')
ax1.legend()
ax1.grid(True, alpha=0.3)

# Posición final vs
u_final_values = [r['u_final'] for r in resultados_mu]
ax2.plot(mu_array, u_final_values, 'g-', linewidth=2)
ax2.axhline(y=0.95, color='r', linestyle='--', linewidth=2, label='Mínimo para␣
 ↪completar (0.95)')
ax2.axhline(y=1.0, color='orange', linestyle=':', linewidth=1, label='Final del␣
 ↪recorrido (1.0)')
ax2.fill_between(mu_array, 0.95, 1.1, alpha=0.2, color='green', label='Zona␣
 ↪válida')
ax2.set_xlabel('Coeficiente de fricción  ', fontsize=11)
ax2.set_ylabel('Posición final (u)', fontsize=11)
ax2.set_title('Posición Final vs  ', fontsize=12, fontweight='bold')
ax2.legend()
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

```
================================================================================
ESTUDIO 2: VARIACIÓN DE COEFICIENTE DE FRICCIÓN ( )
================================================================================
Parámetros fijos: m=800 kg, c_a=0.4, S_f=2.0 m², v =7.0 m/s

Rango explorado: 0.0050 - 0.0500
  Rango válido: 0.0078 - 0.0224
================================================================================
```



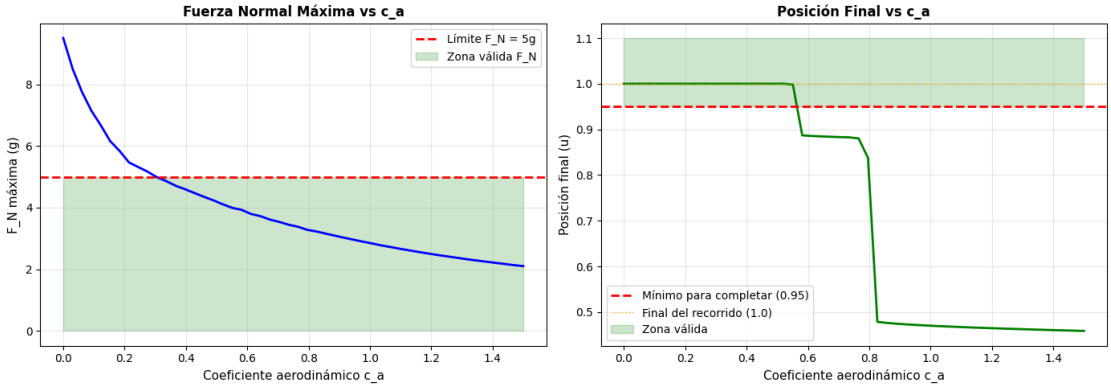
2.8.5 *Variación coef aerodinámico*

[26]:
```
# ============================================
# ESTUDIO 3: Variación de COEFICIENTE AERODINÁMICO (c_a)
# ============================================

# Parámetros base
m_base = 800
mu_base = 0.015
Sf_base = 2.0
v0_base = 7.0

# Rango de c_a a explorar
ca_values = np.linspace(0.0, 1.5, 50)

resultados_ca = []
for ca_test in ca_values:
    resultado = simular_montaña(m_base, mu_base, ca_test, Sf_base, v0_base)
    resultados_ca.append({
        'ca': ca_test,
        'valido': resultado['valido'],
        'completa': resultado['completa_recorrido'],
        'FN_max': resultado['FN_max_G'],
        'u_final': resultado['u_final']
    })

# Análisis
ca_validos = [r['ca'] for r in resultados_ca if r['valido']]

print("\n" + "="*80)
print("ESTUDIO 3: VARIACIÓN DE COEFICIENTE AERODINÁMICO (c_a)")
print("="*80)
```

```
print(f"Parámetros fijos: m={m_base} kg,  ={mu_base}, S_f={Sf_base} m²,␣
 ↪v ={v0_base} m/s")
print(f"\nRango explorado: {ca_values.min():.3f} - {ca_values.max():.3f}")
if ca_validos:
    print(f"  Rango válido: {min(ca_validos):.3f} - {max(ca_validos):.3f}")
else:
    print("  No hay valores válidos en el rango explorado")
print("="*80)

# Gráfica
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# FN_max vs c_a
FN_values = [r['FN_max'] for r in resultados_ca]
ca_array = [r['ca'] for r in resultados_ca]
ax1.plot(ca_array, FN_values, 'b-', linewidth=2)
ax1.axhline(y=5, color='r', linestyle='--', linewidth=2, label='Límite F_N =␣
 ↪5g')
ax1.fill_between(ca_array, 0, 5, alpha=0.2, color='green', label='Zona válida␣
 ↪F_N')
ax1.set_xlabel('Coeficiente aerodinámico c_a', fontsize=11)
ax1.set_ylabel('F_N máxima (g)', fontsize=11)
ax1.set_title('Fuerza Normal Máxima vs c_a', fontsize=12, fontweight='bold')
ax1.legend()
ax1.grid(True, alpha=0.3)

# Posición final vs c_a
u_final_values = [r['u_final'] for r in resultados_ca]
ax2.plot(ca_array, u_final_values, 'g-', linewidth=2)
ax2.axhline(y=0.95, color='r', linestyle='--', linewidth=2, label='Mínimo para␣
 ↪completar (0.95)')
ax2.axhline(y=1.0, color='orange', linestyle=':', linewidth=1, label='Final del␣
 ↪recorrido (1.0)')
ax2.fill_between(ca_array, 0.95, 1.1, alpha=0.2, color='green', label='Zona␣
 ↪válida')
ax2.set_xlabel('Coeficiente aerodinámico c_a', fontsize=11)
ax2.set_ylabel('Posición final (u)', fontsize=11)
ax2.set_title('Posición Final vs c_a', fontsize=12, fontweight='bold')
ax2.legend()
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

```
================================================================================
ESTUDIO 3: VARIACIÓN DE COEFICIENTE AERODINÁMICO (c_a)
```

```
================================================================================
Parámetros fijos: m=800 kg,  =0.015, S_f=2.0 m², v =7.0 m/s

Rango explorado: 0.000 - 1.500
  Rango válido: 0.306 - 0.551
================================================================================
```



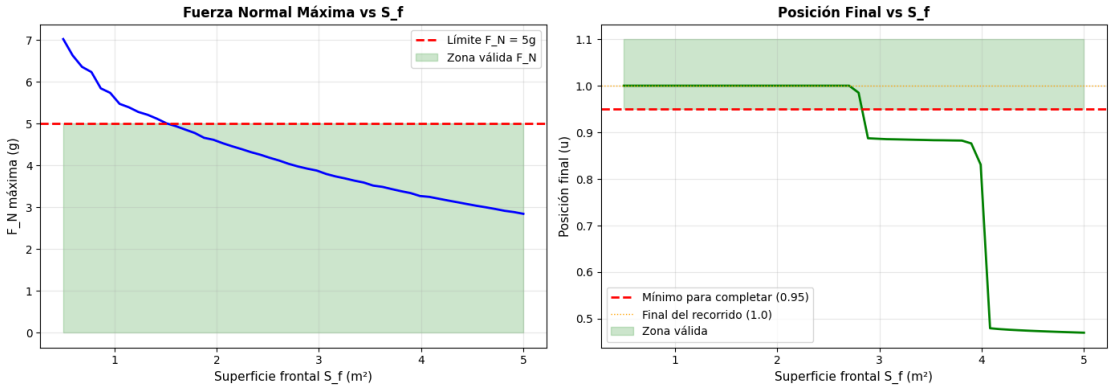
2.8.6 *Variación superficie frontal*

[27]:
```python
# ============================================
# ESTUDIO 4: Variación de SUPERFICIE FRONTAL (S_f)
# ============================================

# Parámetros base
m_base = 800
mu_base = 0.015
ca_base = 0.4
v0_base = 7.0

# Rango de S_f a explorar (m²)
Sf_values = np.linspace(0.5, 5.0, 50)

resultados_Sf = []
for Sf_test in Sf_values:
    resultado = simular_montaña(m_base, mu_base, ca_base, Sf_test, v0_base)
    resultados_Sf.append({
        'Sf': Sf_test,
        'valido': resultado['valido'],
        'completa': resultado['completa_recorrido'],
        'FN_max': resultado['FN_max_G'],
        'u_final': resultado['u_final']
    })
```

```
# Análisis
Sf_validos = [r['Sf'] for r in resultados_Sf if r['valido']]

print("\n" + "="*80)
print("ESTUDIO 4: VARIACIÓN DE SUPERFICIE FRONTAL (S_f)")
print("="*80)
print(f"Parámetros fijos: m={m_base} kg,  ={mu_base}, c_a={ca_base},␣
 ↪v ={v0_base} m/s")
print(f"\nRango explorado: {Sf_values.min():.2f} - {Sf_values.max():.2f} m²")
if Sf_validos:
    print(f"  Rango válido: {min(Sf_validos):.2f} - {max(Sf_validos):.2f} m²")
else:
    print("  No hay valores válidos en el rango explorado")
print("="*80)

# Gráfica
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# FN_max vs S_f
FN_values = [r['FN_max'] for r in resultados_Sf]
Sf_array = [r['Sf'] for r in resultados_Sf]
ax1.plot(Sf_array, FN_values, 'b-', linewidth=2)
ax1.axhline(y=5, color='r', linestyle='--', linewidth=2, label='Límite F_N =␣
 ↪5g')
ax1.fill_between(Sf_array, 0, 5, alpha=0.2, color='green', label='Zona válida␣
 ↪F_N')
ax1.set_xlabel('Superficie frontal S_f (m²)', fontsize=11)
ax1.set_ylabel('F_N máxima (g)', fontsize=11)
ax1.set_title('Fuerza Normal Máxima vs S_f', fontsize=12, fontweight='bold')
ax1.legend()
ax1.grid(True, alpha=0.3)

# Posición final vs S_f
u_final_values = [r['u_final'] for r in resultados_Sf]
ax2.plot(Sf_array, u_final_values, 'g-', linewidth=2)
ax2.axhline(y=0.95, color='r', linestyle='--', linewidth=2, label='Mínimo para␣
 ↪completar (0.95)')
ax2.axhline(y=1.0, color='orange', linestyle=':', linewidth=1, label='Final del␣
 ↪recorrido (1.0)')
ax2.fill_between(Sf_array, 0.95, 1.1, alpha=0.2, color='green', label='Zona␣
 ↪válida')
ax2.set_xlabel('Superficie frontal S_f (m²)', fontsize=11)
ax2.set_ylabel('Posición final (u)', fontsize=11)
ax2.set_title('Posición Final vs S_f', fontsize=12, fontweight='bold')
ax2.legend()
ax2.grid(True, alpha=0.3)
```

```
plt.tight_layout()
plt.show()
```

```
================================================================================
ESTUDIO 4: VARIACIÓN DE SUPERFICIE FRONTAL (S_f)
================================================================================
Parámetros fijos: m=800 kg,  =0.015, c_a=0.4, v =7.0 m/s

Rango explorado: 0.50 - 5.00 m²
  Rango válido: 1.60 - 2.80 m²
================================================================================
```



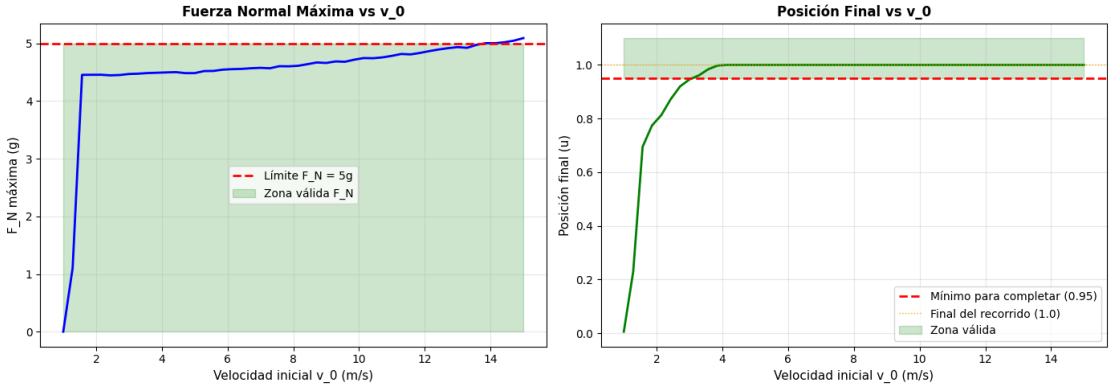
2.8.7 *Variación velocidad incial*

[28]:
```
# ============================================
# ESTUDIO 5: Variación de VELOCIDAD INICIAL (v_0)
# ============================================

# Parámetros base
m_base = 800
mu_base = 0.015
ca_base = 0.4
Sf_base = 2.0

# Rango de v_0 a explorar (m/s)
v0_values = np.linspace(1.0, 15.0, 50)

resultados_v0 = []
for v0_test in v0_values:
    resultado = simular_montaña(m_base, mu_base, ca_base, Sf_base, v0_test)
    resultados_v0.append({
```

```
        'v0': v0_test,
        'valido': resultado['valido'],
        'completa': resultado['completa_recorrido'],
        'FN_max': resultado['FN_max_G'],
        'u_final': resultado['u_final']
    })

# Análisis
v0_validos = [r['v0'] for r in resultados_v0 if r['valido']]

print("\n" + "="*80)
print("ESTUDIO 5: VARIACIÓN DE VELOCIDAD INICIAL (v_0)")
print("="*80)
print(f"Parámetros fijos: m={m_base} kg,  ={mu_base}, c_a={ca_base},␣
 ↪S_f={Sf_base} m²")
print(f"\nRango explorado: {v0_values.min():.2f} - {v0_values.max():.2f} m/s")
if v0_validos:
    print(f"  Rango válido: {min(v0_validos):.2f} - {max(v0_validos):.2f} m/s")
else:
    print("  No hay valores válidos en el rango explorado")
print("="*80)

# Gráfica
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# FN_max vs v_0
FN_values = [r['FN_max'] for r in resultados_v0]
v0_array = [r['v0'] for r in resultados_v0]
ax1.plot(v0_array, FN_values, 'b-', linewidth=2)
ax1.axhline(y=5, color='r', linestyle='--', linewidth=2, label='Límite F_N =␣
 ↪5g')
ax1.fill_between(v0_array, 0, 5, alpha=0.2, color='green', label='Zona válida␣
 ↪F_N')
ax1.set_xlabel('Velocidad inicial v_0 (m/s)', fontsize=11)
ax1.set_ylabel('F_N máxima (g)', fontsize=11)
ax1.set_title('Fuerza Normal Máxima vs v_0', fontsize=12, fontweight='bold')
ax1.legend()
ax1.grid(True, alpha=0.3)

# Posición final vs v_0
u_final_values = [r['u_final'] for r in resultados_v0]
ax2.plot(v0_array, u_final_values, 'g-', linewidth=2)
ax2.axhline(y=0.95, color='r', linestyle='--', linewidth=2, label='Mínimo para␣
 ↪completar (0.95)')
ax2.axhline(y=1.0, color='orange', linestyle=':', linewidth=1, label='Final del␣
 ↪recorrido (1.0)')
```

```
ax2.fill_between(v0_array, 0.95, 1.1, alpha=0.2, color='green', label='Zona␣
 ↪válida')
ax2.set_xlabel('Velocidad inicial v_0 (m/s)', fontsize=11)
ax2.set_ylabel('Posición final (u)', fontsize=11)
ax2.set_title('Posición Final vs v_0', fontsize=12, fontweight='bold')
ax2.legend()
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

```
================================================================================
ESTUDIO 5: VARIACIÓN DE VELOCIDAD INICIAL (v_0)
================================================================================
Parámetros fijos: m=800 kg,  =0.015, c_a=0.4, S_f=2.0 m²

Rango explorado: 1.00 - 15.00 m/s
  Rango válido: 3.29 - 13.57 m/s
================================================================================
```



3 *Diseño 3D*

Tras haber hecho en dos dimensiones pasaremos a definirla en 3D tal y como una montaña rusa debería ser, seguiremos los mismos pasos que antes, solo que ahora extendido a una dimensión más, pero en realidad nada cambia.

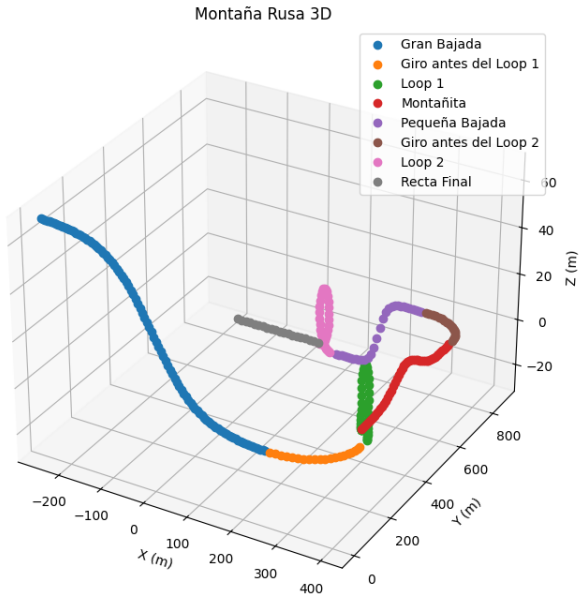
3.1 *Generación de la montaña*

[29]:
```
# Definimos la gran bajada de la montaña rusa
Gran_bajada = cmr.curva3d('s',np.linspace(-250,250,100),A=70,args=[0.02])

# giro antes del loop
giro1 = cmr.curva3d('h',np.linspace(0,np.pi/
 ↪2,20),A=140,C=[Gran_bajada[0][-1]+10,-140,Gran_bajada[2][-1]],args=[0],plano='xy')

# Definimos el primer loop de la montaña rusa
Loop1 = cmr.curva3d('l',np.linspace(0,1.
 ↪79,20),A=33,C=[giro1[0][0],-giro1[1][0]+40,0],args=[0],plano='yz',paso=-30)

# Definimos montañita (gaussina)
Montañita = cmr.curva3d('g',np.
 ↪linspace(-250,250,50),A=10,C=[Loop1[0][-1],Loop1[1][-1]+250,0],args=[95],plano='yz')

# giro antes de la pequeña bajada
giro2 = cmr.curva3d('h',np.linspace(np.pi/
 ↪2,0,20),A=100,C=[Montañita[0][-1]-100,Montañita[1][-1]+20,Montañita[2][-1]],args=[0],plano=

# Definimos la pequeña bajada
Pequeña_bajada = cmr.curva3d('s',np.
 ↪linspace(-100,100,30),A=30,C=[-giro2[0][0]+110,giro2[1][0],Montañita[2][-1]-30],args=[0.
 ↪1])

# Definimos el segundo loop de la montaña rusa
Loop2 = cmr.curva3d('l',np.linspace(0,1.
 ↪79,20),A=30,C=[Pequeña_bajada[0][0]+265,Pequeña_bajada[1][0]+30,Pequeña_bajada[2][-1]],args

# Recta final frenado
Recta_final = [Loop2[0][0]-np.linspace(1,201,20),np.full(20,Loop2[1][0]),np.
 ↪full(20,Loop2[2][-1])]
```

3.2 *Visualización de la montaña rusa*

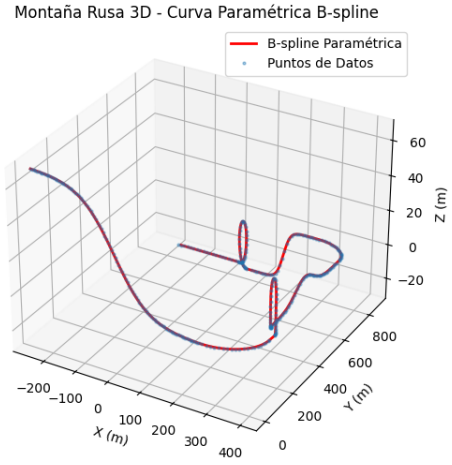
[30]:
```
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')

ax.plot(Gran_bajada[0], Gran_bajada[1], Gran_bajada[2], 'o', linewidth=2,␣
 ↪label='Gran Bajada')
```

```
ax.plot(giro1[0], -giro1[1], giro1[2], 'o', linewidth=2, label='Giro antes del␣
 ↪Loop 1')

ax.plot(Loop1[0], Loop1[1], Loop1[2], 'o', linewidth=2, label='Loop 1')

ax.plot(Montañita[0], Montañita[1], Montañita[2], 'o', linewidth=2,␣
 ↪label='Montañita')

ax.plot(-Pequeña_bajada[0], Pequeña_bajada[1], Pequeña_bajada[2], 'o',␣
 ↪linewidth=2, label='Pequeña Bajada')

ax.plot(giro2[0], giro2[1], giro2[2], 'o', linewidth=2, label='Giro antes del␣
 ↪Loop 2')

ax.plot(Loop2[0], Loop2[1], Loop2[2], 'o', linewidth=2, label='Loop 2')

ax.plot(Recta_final[0], Recta_final[1], Recta_final[2], 'o', linewidth=2,␣
 ↪label='Recta Final')
ax.set_xlabel('X (m)')
ax.set_ylabel('Y (m)')
ax.set_zlabel('Z (m)')
ax.set_title('Montaña Rusa 3D')
ax.legend()
plt.show()
```



3.3 *Construcción y visualización de curva paramétrica*

[31]:
```
# Visualización usando la expresión paramétrica r(t)

#Construimos la expresión
Curva_parametrica = montaña_rusa_parametrica(
    X_data=np.concatenate([Gran_bajada[0],giro1[0][::
 ↪-1],Loop1[0],Montañita[0],giro2[0][::-1],-Pequeña_bajada[0],Loop2[0][::
 ↪-1],Recta_final[0]]),
    Z_data=np.concatenate([Gran_bajada[2],giro1[2][::
 ↪-1],Loop1[2],Montañita[2],giro2[2][::-1],Pequeña_bajada[2],Loop2[2][::
 ↪-1],Recta_final[2]]),
    Y_data=np.concatenate([Gran_bajada[1],-giro1[1][::
 ↪-1],Loop1[1],Montañita[1],giro2[1][::-1],Pequeña_bajada[1],Loop2[1][::
 ↪-1],Recta_final[1]]),
    grado=3
)
# Evaluamos
t_eval = np.linspace(0, 1, 1000)

fig=plt.figure(figsize=(12, 6))
ax = fig.add_subplot(111, projection='3d')

# Gráfica comparativa
ax.plot(Curva_parametrica(t_eval)[:,0], Curva_parametrica(t_eval)[:,1],␣
 ↪Curva_parametrica(t_eval)[:,2], 'r-', linewidth=2, label='B-spline␣
 ↪Paramétrica')
ax.plot(np.concatenate([Gran_bajada[0],giro1[0][::
 ↪-1],Loop1[0],Montañita[0],giro2[0][::-1],-Pequeña_bajada[0],Loop2[0][::
 ↪-1],Recta_final[0]]),
        np.concatenate([Gran_bajada[1],-giro1[1][::
 ↪-1],Loop1[1],Montañita[1],giro2[1][::-1],Pequeña_bajada[1],Loop2[1][::
 ↪-1],Recta_final[1]]),
        np.concatenate([Gran_bajada[2],giro1[2][::
 ↪-1],Loop1[2],Montañita[2],giro2[2][::-1],Pequeña_bajada[2],Loop2[2][::
 ↪-1],Recta_final[2]]),
        'o', markersize=2, label='Puntos de Datos',alpha=0.4)
ax.set_xlabel('X (m)')
ax.set_ylabel('Y (m)')
ax.set_zlabel('Z (m)')
ax.set_title('Montaña Rusa 3D - Curva Paramétrica B-spline')
ax.legend()
plt.show()
```



3.4 *Caso conservativo*

3.4.1 *Sistema SEDO*

[32]:
```
g = 9.8 #m/s^2 #aceleración de la gravedad
m = 800 #kg #masa del vagón con 4 pasajeros a bordo
mu = 0.0 #coeficiente de fricción
c_a = 0 #coeficiente de resistencia aerodinámica
S_f = 2 #m^2 #superficie frontal del vagón
ro_a = 1.225 #kg/m^3 #densidad del aire
c_v = (c_a*S_f*ro_a)/(2*m) #m^{-1} #coeficiente de la fuerza viscosa

# Condiciones iniciales [parámetro u0, velocidad v0]
u0 = 0.0              # inicio de la curva
v0 = 7             # m/s - velocidad inicial
y0 = [u0, v0]

#============================================
# Sistema Sedo
#============================================

# Calcular derivadas de la curva usando pkgmrusa UNA SOLA VEZ
derivadas_curva = cmr.trayec_der(Curva_parametrica)


def edo_montaña_rusa_conservativa(t, y):
    '''
    Define el sistema de EDOs para la montaña rusa.
```

```
    Parámetros:
    -----------
    t : float
        Tiempo (no se usa explícitamente en este sistema)
    y : list
        Estado del sistema [posición u, velocidad v]

    Retorna:
    --------
    dydt : list
        Derivadas [du/dt, dv/dt]
    '''

    return cmr.edofun_mr(t, y, derivadas_curva, mu, c_v, g)


#Información del sistema
print("="*70)
print("SISTEMA EDO - CASO CONSERVATIVO")
print("="*70)
print(f"Parámetros físicos:")
print(f"  • Masa: {m} kg")
print(f"  • Coef. rozamiento  : {mu} (SIN FRICCIÓN)")
print(f"  • Coef. arrastre ca: {c_a} (SIN ROZAMIENTO)")
print(f"  • Superficie frontal Sf: {S_f} m²")
print(f"  • Coeficiente viscoso cvefVis: {c_v} 1/m")
print(f"  • Gravedad: {g} m/s² (positivo)")
print(f"\nCondiciones iniciales:")
print(f"  • Posición inicial: u  = {u0}")
print(f"  • Velocidad inicial: v  = {v0} m/s")
```

```
======================================================================
SISTEMA EDO - CASO CONSERVATIVO
======================================================================
Parámetros físicos:
  • Masa: 800 kg
  • Coef. rozamiento  : 0.0 (SIN FRICCIÓN)
  • Coef. arrastre ca: 0 (SIN ROZAMIENTO)
  • Superficie frontal Sf: 2 m²
  • Coeficiente viscoso cvefVis: 0.0 1/m
  • Gravedad: 9.8 m/s² (positivo)

Condiciones iniciales:
  • Posición inicial: u  = 0.0
  • Velocidad inicial: v  = 7 m/s
```

3.4.2 *Solución al SEDO*

[33]:
```
# t_eval para solución densa
t_eval = np.linspace(0, 200, 1000)

# Resolver EDO no conservativa
solución_3D =␣
 ↪solve_ivp(edo_montaña_rusa_no_conservativa,[0,200],y0,method="Radau",t_eval=t_eval,dense_ou
 ↪finalVia,cmr.paradaVagon])
```

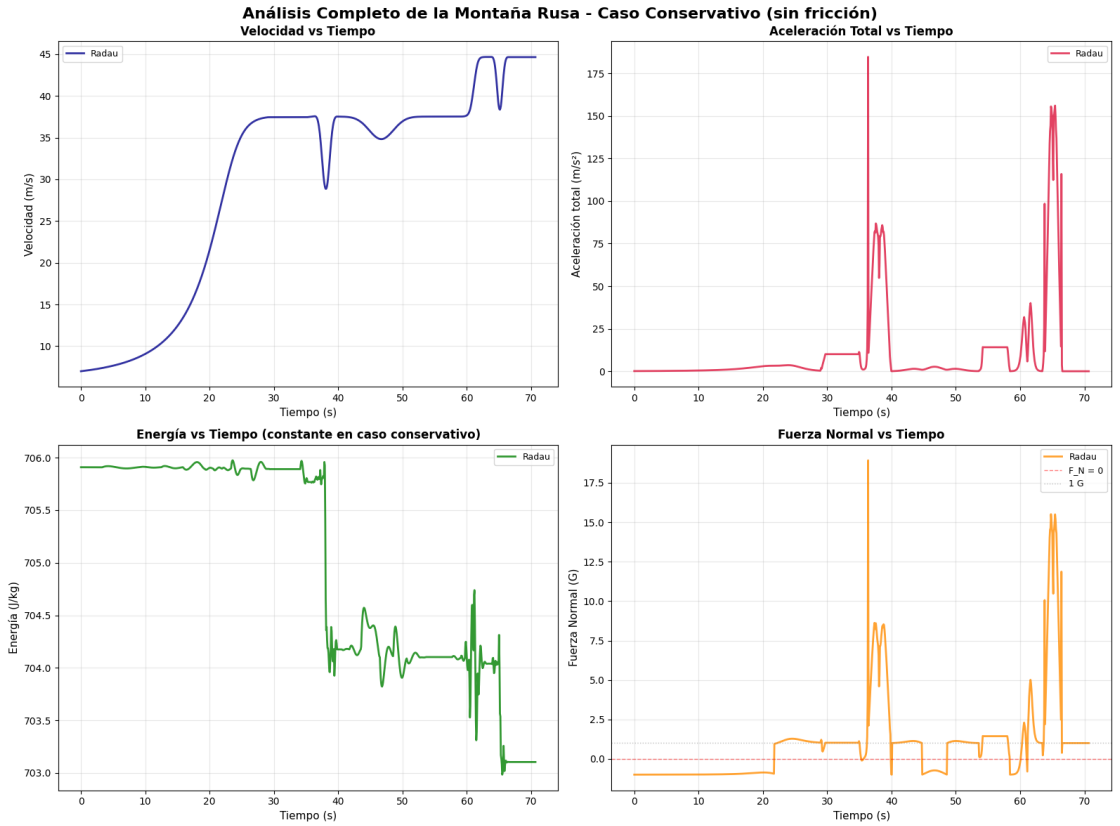
3.4.3 *Análisis de lo obtenido*

[34]:
```
# ================================================
# GRÁFICAS COMBINADAS: Análisis del movimiento con fricción
# ================================================

fig, axes = plt.subplots(2, 2, figsize=(16, 12))
fig.suptitle('Análisis Completo de la Montaña Rusa - Caso Conservativo (sin␣
 ↪fricción)', fontsize=16, fontweight='bold')

# Generar puntos densos para cálculos
t_dense = np.linspace(solución_3D.t[0], solución_3D.t[-1], 1000)
datos = solución_3D.sol(t_dense)
u_vals = datos[0]
v_vals = datos[1]

# ================================================
# 1. VELOCIDAD vs TIEMPO (subplot superior izquierdo)
# ================================================
ax1 = axes[0, 0]
ax1.plot(t_dense, v_vals, linewidth=2, color='darkblue', alpha=0.8,␣
 ↪label='Radau')
ax1.set_xlabel('Tiempo (s)', fontsize=11)
ax1.set_ylabel('Velocidad (m/s)', fontsize=11)
ax1.set_title('Velocidad vs Tiempo', fontsize=12, fontweight='bold')
ax1.legend(loc='best', fontsize=9)
ax1.grid(True, alpha=0.3)

# ================================================
# 2. ACELERACIÓN TOTAL vs TIEMPO (subplot superior derecho)
# ================================================
ax2 = axes[0, 1]

aceleraciones = []
for i in range(len(u_vals)):
    fuerzaN, baseLocal, ctes = cmr.fuerzaNormal(u_vals[i], v_vals[i],␣
 ↪derivadas_curva, g)
```

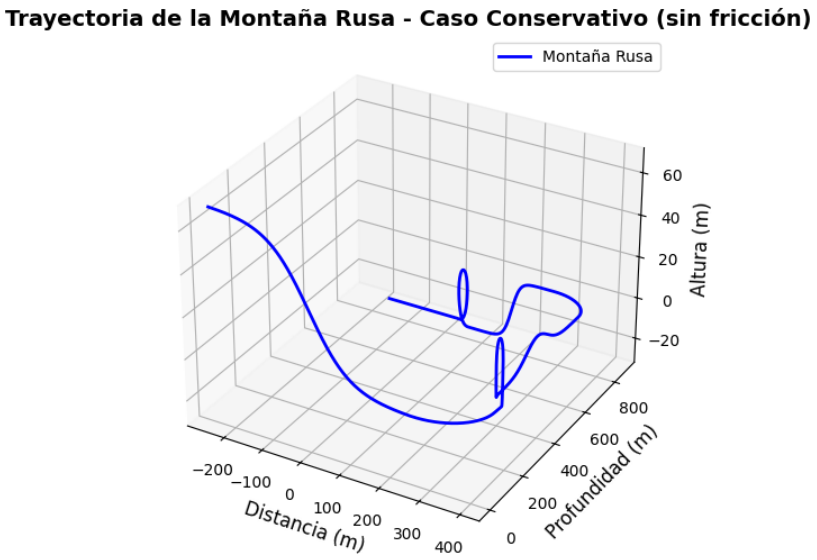
```
    ace_total, ace_tg, ace_nr = cmr.aceleracion(v_vals[i], baseLocal, mu, c_v,␣
 ↪fuerzaN, ctes[1], g)
    aceleraciones.append(ace_total)

ax2.plot(t_dense, aceleraciones, linewidth=2, color='crimson', alpha=0.8,␣
 ↪label='Radau')
ax2.set_xlabel('Tiempo (s)', fontsize=11)
ax2.set_ylabel('Aceleración total (m/s²)', fontsize=11)
ax2.set_title('Aceleración Total vs Tiempo', fontsize=12, fontweight='bold')
ax2.legend(loc='best', fontsize=9)
ax2.grid(True, alpha=0.3)

# ================================================
# 3. ENERGÍA vs TIEMPO (subplot inferior izquierdo)
# ================================================
ax3 = axes[1, 0]

energias = []
for i in range(len(u_vals)):
    E = cmr.energia(u_vals[i], v_vals[i], derivadas_curva[0], g)
    energias.append(E)

ax3.plot(t_dense, energias, linewidth=2, color='green', alpha=0.8,␣
 ↪label='Radau')
ax3.set_xlabel('Tiempo (s)', fontsize=11)
ax3.set_ylabel('Energía (J/kg)', fontsize=11)
ax3.set_title('Energía vs Tiempo (constante en caso conservativo)',␣
 ↪fontsize=12, fontweight='bold')
ax3.legend(loc='best', fontsize=9)
ax3.grid(True, alpha=0.3)

# ================================================
# 4. FUERZA NORMAL vs TIEMPO en unidades de G (subplot inferior derecho)
# ================================================
ax4 = axes[1, 1]

fuerzas_normales_G = []

for i in range(len(u_vals)):
    fuerzaN, baseLocal, ctes = cmr.fuerzaNormal(u_vals[i], v_vals[i],␣
 ↪derivadas_curva, g)
    # Convertir a unidades de G (1 G = 9.81 m/s²)
    fuerzaN_G = fuerzaN / g
    fuerzas_normales_G.append(fuerzaN_G)

ax4.plot(t_dense, fuerzas_normales_G, linewidth=2, color='darkorange', alpha=0.
 ↪8, label='Radau')
```

```
ax4.set_xlabel('Tiempo (s)', fontsize=11)
ax4.set_ylabel('Fuerza Normal (G)', fontsize=11)
ax4.set_title('Fuerza Normal vs Tiempo', fontsize=12, fontweight='bold')
ax4.grid(True, alpha=0.3)
ax4.axhline(y=0, color='r', linestyle='--', linewidth=1, alpha=0.5, label='F_N␣
 ↪= 0')
ax4.axhline(y=1, color='gray', linestyle=':', linewidth=1, alpha=0.5, label='1␣
 ↪G')
ax4.legend(loc='best', fontsize=9)

plt.tight_layout()
plt.show()

# ================================================
# ANÁLISIS DE PÉRDIDA DE ENERGÍA
# ================================================
print("\n" + "="*80)
print("ANÁLISIS DE PÉRDIDA DE ENERGÍA - CASO CONSERVATIVO")
print("="*80)
E_inicial = energias[0]
E_final = energias[-1]
perdida_energia = E_inicial - E_final
porcentaje_perdida = (perdida_energia / E_inicial) * 100

print(f"\nEnergía inicial:  {E_inicial:.2f} J/kg")
print(f"Energía final:    {E_final:.2f} J/kg")
print(f"Pérdida total:    {perdida_energia:.2f} J/kg ({porcentaje_perdida:.
 ↪2f}%)")
print(f"\nVelocidad inicial: {v_vals[0]:.2f} m/s")
print(f"Velocidad final:   {v_vals[-1]:.2f} m/s")
print(f"\n  La energía disminuye gradualmente debido a:")
print(f"    • Fricción con la pista (  = {mu})")
print(f"    • Resistencia aerodinámica (c_v = {c_v:.6f} m ¹)")
print("="*80 + "\n")


fig=plt.figure(figsize=(10,6))
ax = fig.add_subplot(111, projection='3d')
ax.plot(Curva_parametrica(solución_3D.sol(t_dense)[0])[:,0],␣
 ↪Curva_parametrica(solución_3D.sol(t_dense)[0])[:,1],␣
 ↪Curva_parametrica(solución_3D.sol(t_dense)[0])[:,2], 'b-', linewidth=2,␣
 ↪label='Montaña Rusa')
ax.set_title('Trayectoria de la Montaña Rusa - Caso Conservativo (sin␣
 ↪fricción)', fontsize=14, fontweight='bold')
ax.set_xlabel('Distancia (m)', fontsize=12)
ax.set_ylabel('Profundidad (m)', fontsize=12)
ax.set_zlabel('Altura (m)', fontsize=12)
```

```
ax.grid(True, alpha=0.3)
ax.legend()
plt.show()
```




```
================================================================================
ANÁLISIS DE PÉRDIDA DE ENERGÍA - CASO CONSERVATIVO
================================================================================

Energía inicial:  705.91 J/kg
Energía final:    703.10 J/kg
Pérdida total:    2.80 J/kg (0.40%)

Velocidad inicial: 7.00 m/s
Velocidad final:   44.65 m/s

  La energía disminuye gradualmente debido a:
    • Fricción con la pista (  = 0.0)
    • Resistencia aerodinámica (c_v = 0.000000 m ¹)
================================================================================
```



3.5 *Caso no conservativo*

3.5.1 *Sistema SEDO*

[35]:
```
g = 9.8 #m/s^2 #aceleración de la gravedad
m = 800 #kg #masa del vagón con 4 pasajeros a bordo
mu = 0.015 #coeficiente de fricción
c_a = 0.4 #coeficiente de resistencia aerodinámica
S_f = 2 #m^2 #superficie frontal del vagón
ro_a = 1.225 #kg/m^3 #densidad del aire
c_v = (c_a*S_f*ro_a)/(2*m) #m^{-1} #coeficiente de la fuerza viscosa

# Condiciones iniciales [parámetro u0, velocidad v0]
u0 = 0.0              # inicio de la curva
v0 = 7             # m/s - velocidad inicial
y0 = [u0, v0]

#============================================
# Sistema Sedo
#============================================

# Calcular derivadas de la curva usando pkgmrusa UNA SOLA VEZ
derivadas_curva = cmr.trayec_der(Curva_parametrica)


def edo_montaña_rusa_no_conservativa(t, y):
```

```
    '''
    Define el sistema de EDOs para la montaña rusa.

    Parámetros:
    -----------
    t : float
        Tiempo (no se usa explícitamente en este sistema)
    y : list
        Estado del sistema [posición u, velocidad v]

    Retorna:
    --------
    dydt : list
        Derivadas [du/dt, dv/dt]
    '''

    return cmr.edofun_mr(t, y, derivadas_curva, mu, c_v, g)


#Información del sistema
print("="*70)
print("SISTEMA EDO - CASO NO CONSERVATIVO")
print("="*70)
print(f"Parámetros físicos:")
print(f"  • Masa: {m} kg")
print(f"  • Coef. rozamiento  : {mu} (SIN FRICCIÓN)")
print(f"  • Coef. arrastre ca: {c_a} (SIN ROZAMIENTO)")
print(f"  • Superficie frontal Sf: {S_f} m²")
print(f"  • Coeficiente viscoso cvefVis: {c_v} 1/m")
print(f"  • Gravedad: {g} m/s² (positivo)")
print(f"\nCondiciones iniciales:")
print(f"  • Posición inicial: u  = {u0}")
print(f"  • Velocidad inicial: v  = {v0} m/s")
```

```
======================================================================
SISTEMA EDO - CASO NO CONSERVATIVO
======================================================================
Parámetros físicos:
  • Masa: 800 kg
  • Coef. rozamiento  : 0.015 (SIN FRICCIÓN)
  • Coef. arrastre ca: 0.4 (SIN ROZAMIENTO)
  • Superficie frontal Sf: 2 m²
  • Coeficiente viscoso cvefVis: 0.0006125000000000001 1/m
  • Gravedad: 9.8 m/s² (positivo)

Condiciones iniciales:
  • Posición inicial: u  = 0.0
  • Velocidad inicial: v  = 7 m/s
```

3.5.2 *Resolución del SEDO*

[36]:
```
# t_eval para solución densa
t_eval = np.linspace(0, 200, 1000)

# Resolver EDO no conservativa
solución_3D_fric =␣
 ↪solve_ivp(edo_montaña_rusa_no_conservativa,[0,200],y0,method="Radau",t_eval=t_eval,dense_ou
 ↪finalVia,cmr.paradaVagon])
```

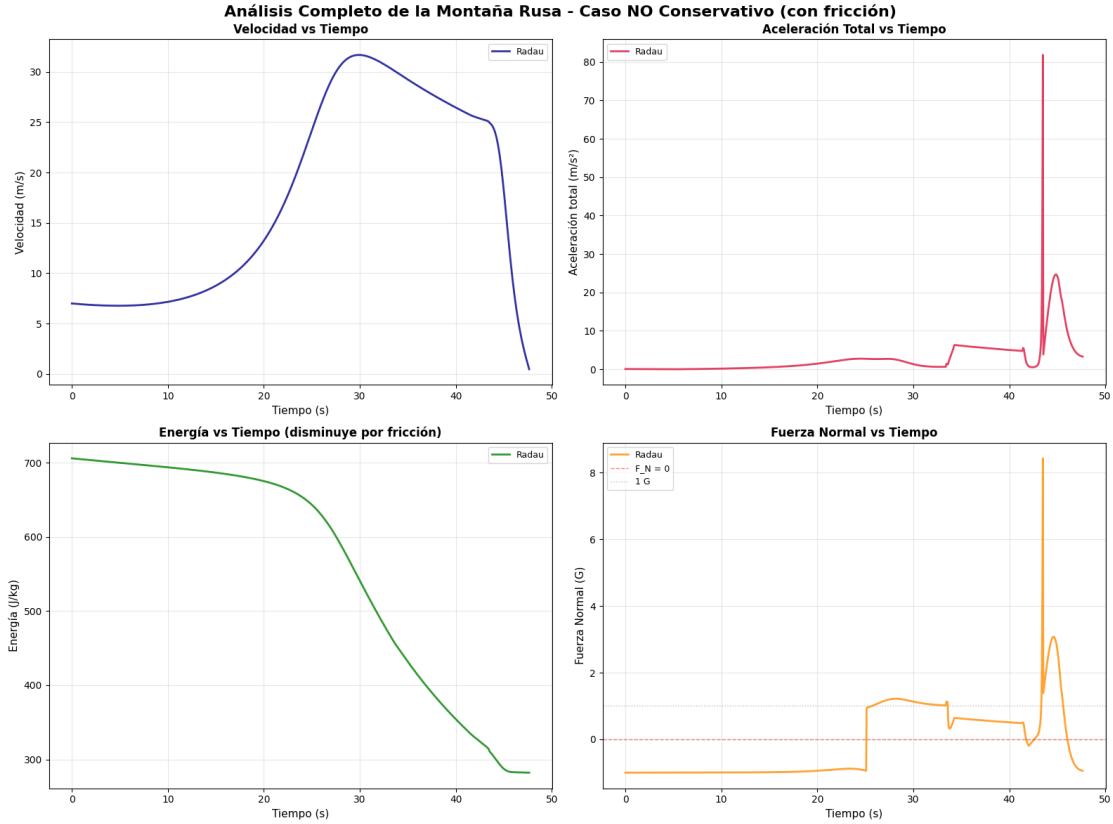
3.5.3 *Análisis de lo obtenido*

[37]:
```
# ================================================
# GRÁFICAS COMBINADAS: Análisis del movimiento con fricción
# ================================================

fig, axes = plt.subplots(2, 2, figsize=(16, 12))
fig.suptitle('Análisis Completo de la Montaña Rusa - Caso NO Conservativo (con␣
 ↪fricción)', fontsize=16, fontweight='bold')

# Generar puntos densos para cálculos
t_dense = np.linspace(solución_3D_fric.t[0], solución_3D_fric.t[-1], 1000)
datos = solución_3D_fric.sol(t_dense)
u_vals = datos[0]
v_vals = datos[1]

# ================================================
# 1. VELOCIDAD vs TIEMPO (subplot superior izquierdo)
# ================================================
ax1 = axes[0, 0]
ax1.plot(t_dense, v_vals, linewidth=2, color='darkblue', alpha=0.8,␣
 ↪label='Radau')
ax1.set_xlabel('Tiempo (s)', fontsize=11)
ax1.set_ylabel('Velocidad (m/s)', fontsize=11)
ax1.set_title('Velocidad vs Tiempo', fontsize=12, fontweight='bold')
ax1.legend(loc='best', fontsize=9)
ax1.grid(True, alpha=0.3)

# ================================================
# 2. ACELERACIÓN TOTAL vs TIEMPO (subplot superior derecho)
# ================================================
ax2 = axes[0, 1]

aceleraciones = []
for i in range(len(u_vals)):
    fuerzaN, baseLocal, ctes = cmr.fuerzaNormal(u_vals[i], v_vals[i],␣
 ↪derivadas_curva, g)
```

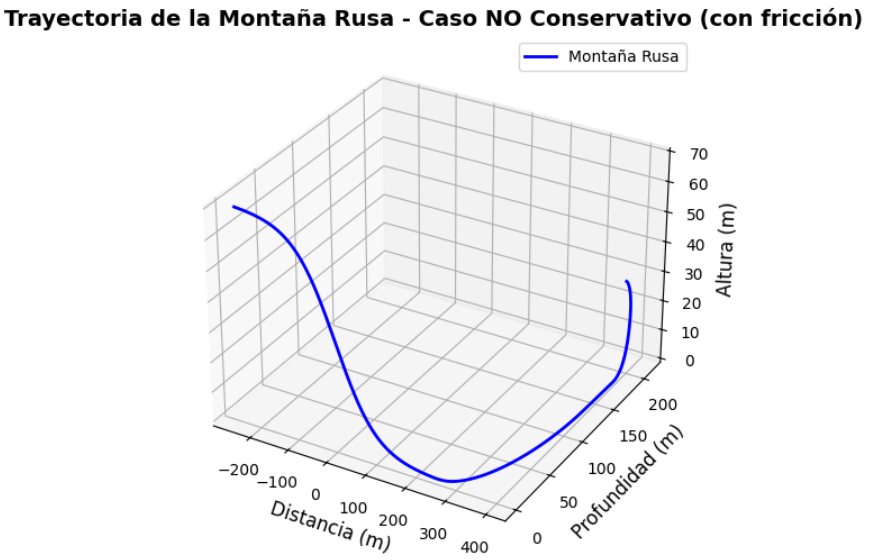
```
    ace_total, ace_tg, ace_nr = cmr.aceleracion(v_vals[i], baseLocal, mu, c_v,␣
 ↪fuerzaN, ctes[1], g)
    aceleraciones.append(ace_total)

ax2.plot(t_dense, aceleraciones, linewidth=2, color='crimson', alpha=0.8,␣
 ↪label='Radau')
ax2.set_xlabel('Tiempo (s)', fontsize=11)
ax2.set_ylabel('Aceleración total (m/s²)', fontsize=11)
ax2.set_title('Aceleración Total vs Tiempo', fontsize=12, fontweight='bold')
ax2.legend(loc='best', fontsize=9)
ax2.grid(True, alpha=0.3)

# ================================================
# 3. ENERGÍA vs TIEMPO (subplot inferior izquierdo)
# ================================================
ax3 = axes[1, 0]

energias = []
for i in range(len(u_vals)):
    E = cmr.energia(u_vals[i], v_vals[i], derivadas_curva[0], g)
    energias.append(E)

ax3.plot(t_dense, energias, linewidth=2, color='green', alpha=0.8,␣
 ↪label='Radau')
ax3.set_xlabel('Tiempo (s)', fontsize=11)
ax3.set_ylabel('Energía (J/kg)', fontsize=11)
ax3.set_title('Energía vs Tiempo (disminuye por fricción)', fontsize=12,␣
 ↪fontweight='bold')
ax3.legend(loc='best', fontsize=9)
ax3.grid(True, alpha=0.3)

# ================================================
# 4. FUERZA NORMAL vs TIEMPO en unidades de G (subplot inferior derecho)
# ================================================
ax4 = axes[1, 1]

fuerzas_normales_G = []

for i in range(len(u_vals)):
    fuerzaN, baseLocal, ctes = cmr.fuerzaNormal(u_vals[i], v_vals[i],␣
 ↪derivadas_curva, g)
    # Convertir a unidades de G (1 G = 9.81 m/s²)
    fuerzaN_G = fuerzaN / g
    fuerzas_normales_G.append(fuerzaN_G)

ax4.plot(t_dense, fuerzas_normales_G, linewidth=2, color='darkorange', alpha=0.
 ↪8, label='Radau')
```

```
ax4.set_xlabel('Tiempo (s)', fontsize=11)
ax4.set_ylabel('Fuerza Normal (G)', fontsize=11)
ax4.set_title('Fuerza Normal vs Tiempo', fontsize=12, fontweight='bold')
ax4.grid(True, alpha=0.3)
ax4.axhline(y=0, color='r', linestyle='--', linewidth=1, alpha=0.5, label='F_N␣
 ↪= 0')
ax4.axhline(y=1, color='gray', linestyle=':', linewidth=1, alpha=0.5, label='1␣
 ↪G')
ax4.legend(loc='best', fontsize=9)

plt.tight_layout()
plt.show()

# ================================================
# ANÁLISIS DE PÉRDIDA DE ENERGÍA
# ================================================
print("\n" + "="*80)
print("ANÁLISIS DE PÉRDIDA DE ENERGÍA - CASO CON FRICCIÓN")
print("="*80)
E_inicial = energias[0]
E_final = energias[-1]
perdida_energia = E_inicial - E_final
porcentaje_perdida = (perdida_energia / E_inicial) * 100

print(f"\nEnergía inicial:  {E_inicial:.2f} J/kg")
print(f"Energía final:    {E_final:.2f} J/kg")
print(f"Pérdida total:    {perdida_energia:.2f} J/kg ({porcentaje_perdida:.
 ↪2f}%)")
print(f"\nVelocidad inicial: {v_vals[0]:.2f} m/s")
print(f"Velocidad final:   {v_vals[-1]:.2f} m/s")
print(f"\n  La energía disminuye gradualmente debido a:")
print(f"    • Fricción con la pista (  = {mu})")
print(f"    • Resistencia aerodinámica (c_v = {c_v:.6f} m ¹)")
print("="*80 + "\n")


fig=plt.figure(figsize=(10,6))
ax = fig.add_subplot(111, projection='3d')
ax.plot(Curva_parametrica(solución_3D_fric.sol(t_dense)[0])[:,0],␣
 ↪Curva_parametrica(solución_3D_fric.sol(t_dense)[0])[:,1],␣
 ↪Curva_parametrica(solución_3D_fric.sol(t_dense)[0])[:,2], 'b-', linewidth=2,␣
 ↪label='Montaña Rusa')
ax.set_title('Trayectoria de la Montaña Rusa - Caso NO Conservativo (con␣
 ↪fricción)', fontsize=14, fontweight='bold')
ax.set_xlabel('Distancia (m)', fontsize=12)
ax.set_ylabel('Profundidad (m)', fontsize=12)
ax.set_zlabel('Altura (m)', fontsize=12)
```

```
ax.grid(True, alpha=0.3)
ax.legend()
plt.show()
```




```
================================================================================
ANÁLISIS DE PÉRDIDA DE ENERGÍA - CASO CON FRICCIÓN
================================================================================

Energía inicial:  705.91 J/kg
Energía final:    282.21 J/kg
Pérdida total:    423.70 J/kg (60.02%)

Velocidad inicial: 7.00 m/s
Velocidad final:   0.47 m/s

  La energía disminuye gradualmente debido a:
    • Fricción con la pista (  = 0.015)
    • Resistencia aerodinámica (c_v = 0.000613 m ¹)
================================================================================
```



Como podemos ver exteniendolo a 3 dimensiones si quiera es capaz de superar el primer loop, luego tendremos que hacer algunos ajustes para que nuestra montaña sea viable, bajo las mismas condiciones de antes

3.6 *Caso viable*

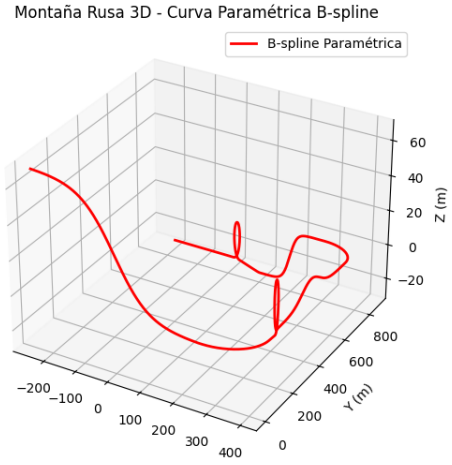
3.6.1 *Nueva definición de la montaña*

[38]:
```
# Definimos la gran bajada de la montaña rusa
Gran_bajada = cmr.curva3d('s',np.linspace(-250,250,100),A=70,args=[0.02])

# giro antes del loop
giro1 = cmr.curva3d('h',np.linspace(0,np.pi/
 ↪2,20),A=140,C=[Gran_bajada[0][-1]+10,-140,Gran_bajada[2][-1]],args=[0],plano='xy')

# Definimos el primer loop de la montaña rusa
Loop1 = cmr.curva3d('l',np.linspace(0,1.
 ↪79,20),A=33,C=[giro1[0][0],-giro1[1][0]+37,0],args=[0],plano='yz',paso=-10)

# Definimos montañita (gaussina)
Montañita = cmr.curva3d('g',np.
 ↪linspace(-250,250,50),A=10,C=[Loop1[0][-1],Loop1[1][-1]+250,0],args=[95],plano='yz')

# giro antes de la pequeña bajada
giro2 = cmr.curva3d('h',np.linspace(np.pi/
 ↪2,0,20),A=100,C=[Montañita[0][-1]-100,Montañita[1][-1]+20,Montañita[2][-1]],args=[0],plano=
```

```
# Definimos la pequeña bajada
Pequeña_bajada = cmr.curva3d('s',np.
 ↪linspace(-100,100,30),A=30,C=[-giro2[0][0]+110,giro2[1][0],Montañita[2][-1]-30],args=[0.
 ↪1])

# Definimos el segundo loop de la montaña rusa
Loop2 = cmr.curva3d('l',np.linspace(0,1.
 ↪79,20),A=25,C=[Pequeña_bajada[0][0]+265,Pequeña_bajada[1][0]+30,Pequeña_bajada[2][-1]+2],ar

# Recta final frenado
Recta_final = [Loop2[0][0]-np.linspace(1,201,20),np.full(20,Loop2[1][0]),np.
 ↪full(20,Loop2[2][-1])]
```

3.6.2 *Visualización directa de la parametrización*

[39]:
```
# Visualización usando la expresión paramétrica r(t)

#Construimos la expresión
Curva_parametrica = montaña_rusa_parametrica(
    X_data=np.concatenate([Gran_bajada[0],giro1[0][::
 ↪-1],Loop1[0],Montañita[0],giro2[0][::-1],-Pequeña_bajada[0],Loop2[0][::
 ↪-1],Recta_final[0]]),
    Z_data=np.concatenate([Gran_bajada[2],giro1[2][::
 ↪-1],Loop1[2],Montañita[2],giro2[2][::-1],Pequeña_bajada[2],Loop2[2][::
 ↪-1],Recta_final[2]]),
    Y_data=np.concatenate([Gran_bajada[1],-giro1[1][::
 ↪-1],Loop1[1],Montañita[1],giro2[1][::-1],Pequeña_bajada[1],Loop2[1][::
 ↪-1],Recta_final[1]]),
    grado=3
)
# Evaluamos
t_eval = np.linspace(0, 1, 1000)

fig=plt.figure(figsize=(12, 6))
ax = fig.add_subplot(111, projection='3d')

# Gráfica comparativa
ax.plot(Curva_parametrica(t_eval)[:,0], Curva_parametrica(t_eval)[:,1],␣
 ↪Curva_parametrica(t_eval)[:,2], 'r-', linewidth=2, label='B-spline␣
 ↪Paramétrica')

ax.set_ylabel('Y (m)')
ax.set_zlabel('Z (m)')
ax.set_title('Montaña Rusa 3D - Curva Paramétrica B-spline')
ax.legend()
plt.show()
```



3.6.3 *Sistema SEDO*

[40]:
```
g = 9.8 #m/s^2 #aceleración de la gravedad
m = 800 #kg #masa del vagón con 4 pasajeros a bordo
mu = 0.015 #coeficiente de fricción
c_a = 0.4 #coeficiente de resistencia aerodinámica
S_f = 2 #m^2 #superficie frontal del vagón
ro_a = 1.225 #kg/m^3 #densidad del aire
c_v = (c_a*S_f*ro_a)/(2*m) #m^{-1} #coeficiente de la fuerza viscosa

# Condiciones iniciales [parámetro u0, velocidad v0]
u0 = 0.0              # inicio de la curva
v0 = 11.8        # m/s - velocidad inicial
y0 = [u0, v0]

#============================================
# Sistema Sedo
#============================================

# Calcular derivadas de la curva usando pkgmrusa UNA SOLA VEZ
derivadas_curva = cmr.trayec_der(Curva_parametrica)


def edo_montaña_rusa_no_conservativa(t, y):
    '''
    Define el sistema de EDOs para la montaña rusa.

    Parámetros:
```

```
    -----------
    t : float
        Tiempo (no se usa explícitamente en este sistema)
    y : list
        Estado del sistema [posición u, velocidad v]

    Retorna:
    --------
    dydt : list
        Derivadas [du/dt, dv/dt]
    '''

    return cmr.edofun_mr(t, y, derivadas_curva, mu, c_v, g)


#Información del sistema
print("="*70)
print("SISTEMA EDO - CASO NO CONSERVATIVO")
print("="*70)
print(f"Parámetros físicos:")
print(f"  • Masa: {m} kg")
print(f"  • Coef. rozamiento  : {mu} (SIN FRICCIÓN)")
print(f"  • Coef. arrastre ca: {c_a} (SIN ROZAMIENTO)")
print(f"  • Superficie frontal Sf: {S_f} m²")
print(f"  • Coeficiente viscoso cvefVis: {c_v} 1/m")
print(f"  • Gravedad: {g} m/s² (positivo)")
print(f"\nCondiciones iniciales:")
print(f"  • Posición inicial: u  = {u0}")
print(f"  • Velocidad inicial: v  = {v0} m/s")
```

```
======================================================================
SISTEMA EDO - CASO NO CONSERVATIVO
======================================================================
Parámetros físicos:
  • Masa: 800 kg
  • Coef. rozamiento  : 0.015 (SIN FRICCIÓN)
  • Coef. arrastre ca: 0.4 (SIN ROZAMIENTO)
  • Superficie frontal Sf: 2 m²
  • Coeficiente viscoso cvefVis: 0.0006125000000000001 1/m
  • Gravedad: 9.8 m/s² (positivo)

Condiciones iniciales:
  • Posición inicial: u  = 0.0
  • Velocidad inicial: v  = 11.8 m/s
```

3.6.4 *Resolución del SEDO*

[41]:
```
# Tiempo de evaluación
t_eval = np.linspace(0, 300, 600)

# Solución del sistema EDOs
sol_viable =␣
 ↪solve_ivp(edo_montaña_rusa_no_conservativa,[0,300],y0,method="Radau",t_eval=t_eval,dense_ou
 ↪finalVia,cmr.paradaVagon],max_step=0.1)
```

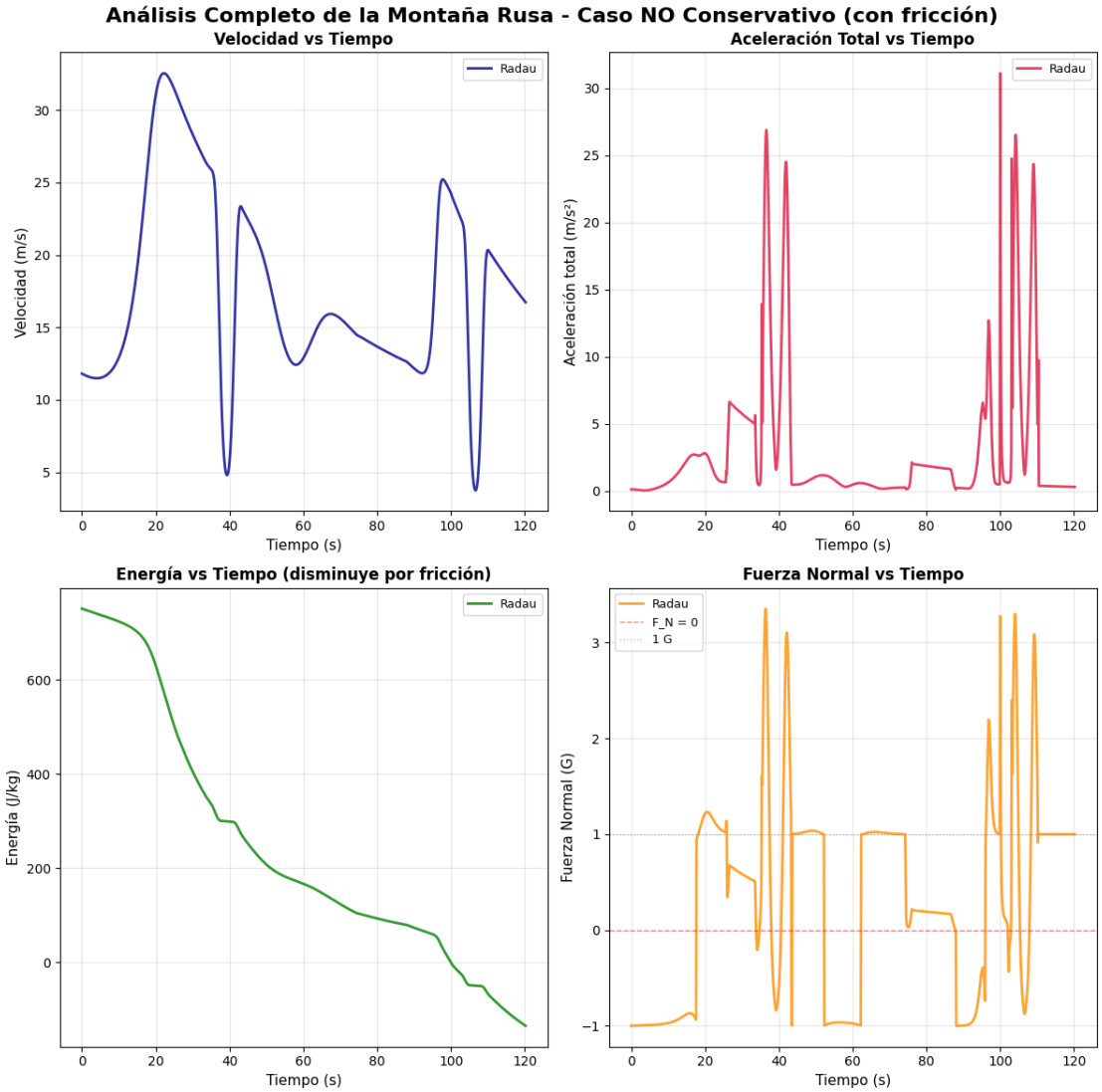
3.6.5 *Visualización de lo obtenido*

[42]:
```
# ================================================
# GRÁFICAS COMBINADAS: Análisis del movimiento con fricción
# ================================================

fig, axes = plt.subplots(2, 2, figsize=(12, 12))
fig.suptitle('Análisis Completo de la Montaña Rusa - Caso NO Conservativo (con␣
 ↪fricción)', fontsize=16, fontweight='bold')

# Generar puntos densos para cálculos
t_dense = np.linspace(sol_viable.t[0], sol_viable.t[-1], 1000)
datos = sol_viable.sol(t_dense)
u_vals = datos[0]
v_vals = datos[1]

# ================================================
# 1. VELOCIDAD vs TIEMPO (subplot superior izquierdo)
# ================================================
ax1 = axes[0, 0]
ax1.plot(t_dense, v_vals, linewidth=2, color='darkblue', alpha=0.8,␣
 ↪label='Radau')
ax1.set_xlabel('Tiempo (s)', fontsize=11)
ax1.set_ylabel('Velocidad (m/s)', fontsize=11)
ax1.set_title('Velocidad vs Tiempo', fontsize=12, fontweight='bold')
ax1.legend(loc='best', fontsize=9)
ax1.grid(True, alpha=0.3)

# ================================================
# 2. ACELERACIÓN TOTAL vs TIEMPO (subplot superior derecho)
# ================================================
ax2 = axes[0, 1]

aceleraciones = []
for i in range(len(u_vals)):
    fuerzaN, baseLocal, ctes = cmr.fuerzaNormal(u_vals[i], v_vals[i],␣
 ↪derivadas_curva, g)
```

```
    ace_total, ace_tg, ace_nr = cmr.aceleracion(v_vals[i], baseLocal, mu, c_v,␣
 ↪fuerzaN, ctes[1], g)
    aceleraciones.append(ace_total)

ax2.plot(t_dense, aceleraciones, linewidth=2, color='crimson', alpha=0.8,␣
 ↪label='Radau')
ax2.set_xlabel('Tiempo (s)', fontsize=11)
ax2.set_ylabel('Aceleración total (m/s²)', fontsize=11)
ax2.set_title('Aceleración Total vs Tiempo', fontsize=12, fontweight='bold')
ax2.legend(loc='best', fontsize=9)
ax2.grid(True, alpha=0.3)

# ================================================
# 3. ENERGÍA vs TIEMPO (subplot inferior izquierdo)
# ================================================
ax3 = axes[1, 0]

energias = []
for i in range(len(u_vals)):
    E = cmr.energia(u_vals[i], v_vals[i], derivadas_curva[0], g)
    energias.append(E)

ax3.plot(t_dense, energias, linewidth=2, color='green', alpha=0.8,␣
 ↪label='Radau')
ax3.set_xlabel('Tiempo (s)', fontsize=11)
ax3.set_ylabel('Energía (J/kg)', fontsize=11)
ax3.set_title('Energía vs Tiempo (disminuye por fricción)', fontsize=12,␣
 ↪fontweight='bold')
ax3.legend(loc='best', fontsize=9)
ax3.grid(True, alpha=0.3)

# ================================================
# 4. FUERZA NORMAL vs TIEMPO en unidades de G (subplot inferior derecho)
# ================================================
ax4 = axes[1, 1]

fuerzas_normales_G = []

for i in range(len(u_vals)):
    fuerzaN, baseLocal, ctes = cmr.fuerzaNormal(u_vals[i], v_vals[i],␣
 ↪derivadas_curva, g)
    # Convertir a unidades de G (1 G = 9.81 m/s²)
    fuerzaN_G = fuerzaN / g
    fuerzas_normales_G.append(fuerzaN_G)

ax4.plot(t_dense, fuerzas_normales_G, linewidth=2, color='darkorange', alpha=0.
 ↪8, label='Radau')
```

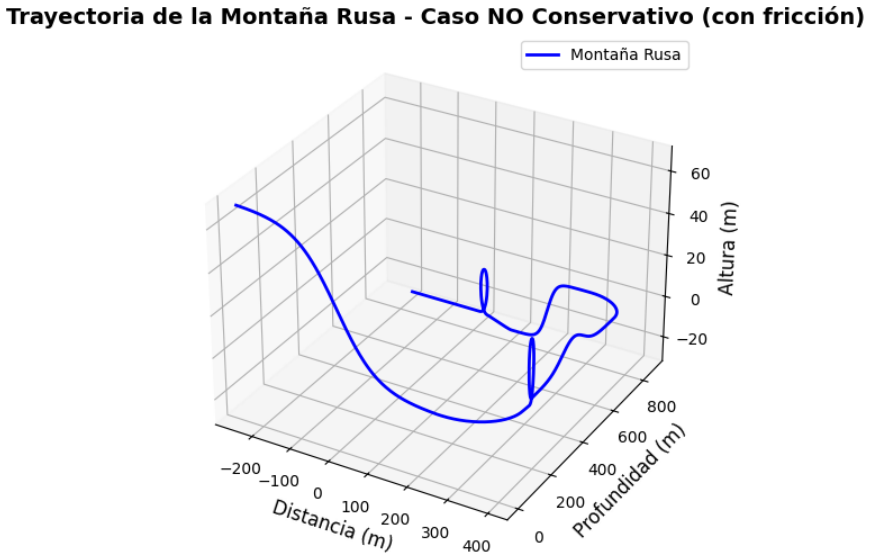
```
ax4.set_xlabel('Tiempo (s)', fontsize=11)
ax4.set_ylabel('Fuerza Normal (G)', fontsize=11)
ax4.set_title('Fuerza Normal vs Tiempo', fontsize=12, fontweight='bold')
ax4.grid(True, alpha=0.3)
ax4.axhline(y=0, color='r', linestyle='--', linewidth=1, alpha=0.5, label='F_N␣
 ↪= 0')
ax4.axhline(y=1, color='gray', linestyle=':', linewidth=1, alpha=0.5, label='1␣
 ↪G')
ax4.legend(loc='best', fontsize=9)

plt.tight_layout()
plt.show()

# ================================================
# ANÁLISIS DE PÉRDIDA DE ENERGÍA
# ================================================
print("\n" + "="*80)
print("ANÁLISIS DE PÉRDIDA DE ENERGÍA - CASO CON FRICCIÓN")
print("="*80)
E_inicial = energias[0]
E_final = energias[-1]
perdida_energia = E_inicial - E_final
porcentaje_perdida = (perdida_energia / E_inicial) * 100

print(f"\nEnergía inicial:  {E_inicial:.2f} J/kg")
print(f"Energía final:    {E_final:.2f} J/kg")
print(f"Pérdida total:    {perdida_energia:.2f} J/kg ({porcentaje_perdida:.
 ↪2f}%)")
print(f"\nVelocidad inicial: {v_vals[0]:.2f} m/s")
print(f"Velocidad final:   {v_vals[-1]:.2f} m/s")
print(f"\n  La energía disminuye gradualmente debido a:")
print(f"    • Fricción con la pista (  = {mu})")
print(f"    • Resistencia aerodinámica (c_v = {c_v:.6f} m ¹)")
print("="*80 + "\n")


fig=plt.figure(figsize=(10,6))
ax = fig.add_subplot(111, projection='3d')
ax.plot(Curva_parametrica(sol_viable.sol(t_dense)[0])[:,0],␣
 ↪Curva_parametrica(sol_viable.sol(t_dense)[0])[:,1],␣
 ↪Curva_parametrica(sol_viable.sol(t_dense)[0])[:,2], 'b-', linewidth=2,␣
 ↪label='Montaña Rusa')
ax.set_title('Trayectoria de la Montaña Rusa - Caso NO Conservativo (con␣
 ↪fricción)', fontsize=14, fontweight='bold')
ax.set_xlabel('Distancia (m)', fontsize=12)
ax.set_ylabel('Profundidad (m)', fontsize=12)
ax.set_zlabel('Altura (m)', fontsize=12)
```

```
ax.grid(True, alpha=0.3)
ax.legend()
plt.show()
```




```
================================================================================
ANÁLISIS DE PÉRDIDA DE ENERGÍA - CASO CON FRICCIÓN
================================================================================

Energía inicial:  751.03 J/kg
Energía final:    -134.31 J/kg
Pérdida total:    885.34 J/kg (117.88%)
```

```
Velocidad inicial: 11.80 m/s
Velocidad final:   16.73 m/s

  La energía disminuye gradualmente debido a:
    • Fricción con la pista (  = 0.015)
    • Resistencia aerodinámica (c_v = 0.000613 m ¹)
================================================================================
```



Podemos ver que en efecto ahora si que cumple con los estandares por lo tanto es perfectamente viable

3.7 *Intervalos válidos*

Igual que en 2D estudiaremos que intervalos de parámetros son viables manteniendo el resto constantes

3.7.1 *Función de estudio*

[43]:
```python
# ============================================
# ESTUDIO DE PARÁMETROS VÁLIDOS
# ============================================

def simular_montaña(m_val, mu_val, ca_val, Sf_val, v0_val, tiempo_max=300):
    """
    Simula la montaña rusa con los parámetros dados y verifica las condiciones.
```

```
    Retorna:
    --------
    dict con:
        - 'valido': bool (si cumple ambas condiciones)
        - 'completa_recorrido': bool
        - 'FN_max_G': float (fuerza normal máxima en unidades de G)
        - 'u_final': float (posición final alcanzada)
        - 'sol': objeto solución (si fue exitosa)
    """

    # Calcular coeficiente viscoso
    c_v_sim = (ca_val * Sf_val * ro_a) / (2 * m_val)


    # Condiciones iniciales
    y0_sim = [0.0, v0_val]

    # Sistema EDO
    def edo_sim(t, y):
        return cmr.edofun_mr(t, y, derivadas_curva, mu_val, c_v_sim, g)

    # Resolver
    try:
        sol_sim = solve_ivp(edo_sim, [0, tiempo_max], y0_sim,
                            method="Radau", dense_output=True,
                            events=[cmr.finalVia,cmr.paradaVagon],max_step=0.
↪1,t_eval=np.linspace(0, tiempo_max, 600))  # Se detiene si velocidad llega a␣
↪0

        # Verificar si completó el recorrido (u_final >= 0.95)
        u_final = sol_sim.sol(sol_sim.t[-1])[0]
        completa = u_final >= 0.95

        # Calcular fuerza normal máxima
        t_check = np.linspace(sol_sim.t[0], sol_sim.t[-1], 500)
        datos_check = sol_sim.sol(t_check)
        u_check = datos_check[0]
        v_check = datos_check[1]

        FN_max = 0
        for i in range(len(u_check)):
            if u_check[i] >= 1.0:  # No revisar más allá del recorrido
                break
            fuerzaN, _, _ = cmr.fuerzaNormal(u_check[i], v_check[i],␣
↪derivadas_curva, g)
            FN_G = fuerzaN / g
```

```
            if FN_G > FN_max:
                FN_max = FN_G

        # Verificar condiciones
        FN_ok = FN_max <= 5.0
        valido = completa and FN_ok

        return {
            'valido': valido,
            'completa_recorrido': completa,
            'FN_max_G': FN_max,
            'u_final': u_final,
            'sol': sol_sim,
            'FN_ok': FN_ok
        }

    except Exception as e:
        return {
            'valido': False,
            'completa_recorrido': False,
            'FN_max_G': np.inf,
            'u_final': 0,
            'sol': None,
            'FN_ok': False,
            'error': str(e)
        }

print("="*80)
print("FUNCIÓN DE SIMULACIÓN CREADA")
print("="*80)
print("Condiciones para validez:")
print("  1. Completar el recorrido: u_final  0.95")
print("  2. Fuerza normal máxima: FN_max  5g")
print("="*80)
```

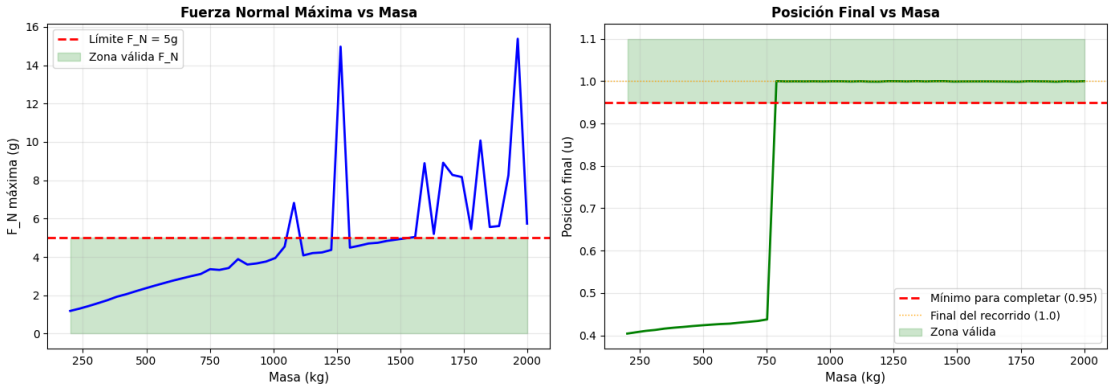
```
================================================================================
FUNCIÓN DE SIMULACIÓN CREADA
================================================================================
Condiciones para validez:
  1. Completar el recorrido: u_final  0.95
  2. Fuerza normal máxima: FN_max  5g
================================================================================
```

3.7.2 *Intervalo de masa*

[44]:
```
# ============================================
# ESTUDIO 1: Variación de MASA (m)
# ============================================

# Parámetros base (del caso viable)
mu_base = 0.015
ca_base = 0.4
Sf_base = 2.0
v0_base = 11.8

# Rango de masas a explorar (kg)
masas = np.linspace(200, 2000, 50)

resultados_masa = []
for m_test in masas:
    resultado = simular_montaña(m_test, mu_base, ca_base, Sf_base, v0_base)
    resultados_masa.append({
        'm': m_test,
        'valido': resultado['valido'],
        'completa': resultado['completa_recorrido'],
        'FN_max': resultado['FN_max_G'],
        'u_final': resultado['u_final']
    })

# Análisis de resultados
masas_validas = [r['m'] for r in resultados_masa if r['valido']]
masas_completan = [r['m'] for r in resultados_masa if r['completa']]
masas_FN_ok = [r['m'] for r in resultados_masa if r['FN_max'] <= 5.0]

print("\n" + "="*80)
print("ESTUDIO 1: VARIACIÓN DE MASA (m)")
print("="*80)
print(f"Parámetros fijos:  ={mu_base}, c_a={ca_base}, S_f={Sf_base} m²,␣
 ↪v ={v0_base} m/s")
print(f"\nRango explorado: {masas.min():.0f} - {masas.max():.0f} kg")
if masas_validas:
    print(f"  Rango válido: {min(masas_validas):.1f} - {max(masas_validas):.1f}␣
 ↪kg")
else:
    print("  No hay valores válidos en el rango explorado")
print("="*80)

# Gráfica
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))
```

```
# FN_max vs masa
FN_values = [r['FN_max'] for r in resultados_masa]
masas_array = [r['m'] for r in resultados_masa]
ax1.plot(masas_array, FN_values, 'b-', linewidth=2)
ax1.axhline(y=5, color='r', linestyle='--', linewidth=2, label='Límite F_N =␣
 ↪5g')
ax1.fill_between(masas_array, 0, 5, alpha=0.2, color='green', label='Zona␣
 ↪válida F_N')
ax1.set_xlabel('Masa (kg)', fontsize=11)
ax1.set_ylabel('F_N máxima (g)', fontsize=11)
ax1.set_title('Fuerza Normal Máxima vs Masa', fontsize=12, fontweight='bold')
ax1.legend()
ax1.grid(True, alpha=0.3)

# Posición final vs masa
u_final_values = [r['u_final'] for r in resultados_masa]
ax2.plot(masas_array, u_final_values, 'g-', linewidth=2)
ax2.axhline(y=0.95, color='r', linestyle='--', linewidth=2, label='Mínimo para␣
 ↪completar (0.95)')
ax2.axhline(y=1.0, color='orange', linestyle=':', linewidth=1, label='Final del␣
 ↪recorrido (1.0)')
ax2.fill_between(masas_array, 0.95, 1.1, alpha=0.2, color='green', label='Zona␣
 ↪válida')
ax2.set_xlabel('Masa (kg)', fontsize=11)
ax2.set_ylabel('Posición final (u)', fontsize=11)
ax2.set_title('Posición Final vs Masa', fontsize=12, fontweight='bold')
ax2.legend()
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

```
================================================================================
ESTUDIO 1: VARIACIÓN DE MASA (m)
================================================================================
Parámetros fijos:  =0.015, c_a=0.4, S_f=2.0 m², v =11.8 m/s

Rango explorado: 200 - 2000 kg
  Rango válido: 787.8 - 1522.4 kg
================================================================================
```



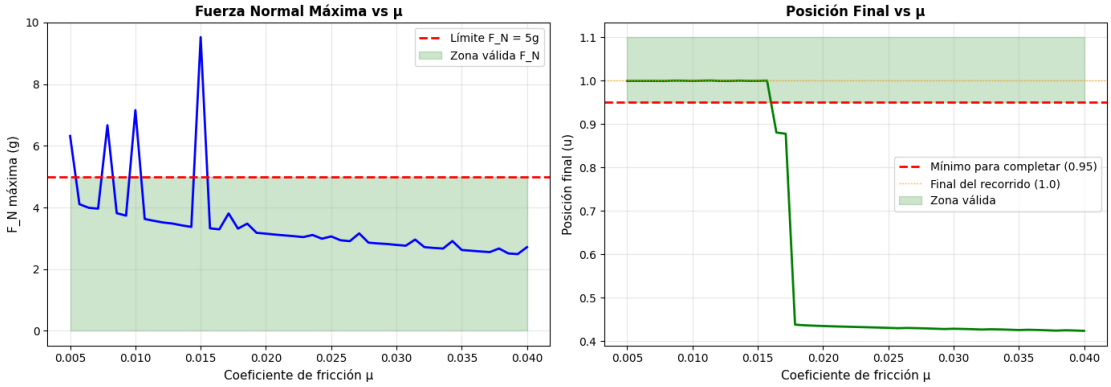
3.7.3 *Intervalo de coef de fricción*

[45]:
```
# ============================================
# ESTUDIO 2: Variación de COEFICIENTE DE FRICCIÓN ( )
# ============================================

# Parámetros base
m_base = 800
ca_base = 0.4
Sf_base = 2.0
v0_base = 11.8

# Rango de   a explorar
mu_values = np.linspace(0.005, 0.04, 50)

resultados_mu = []
for mu_test in mu_values:
    resultado = simular_montaña(m_base, mu_test, ca_base, Sf_base, v0_base)
    resultados_mu.append({
        'mu': mu_test,
        'valido': resultado['valido'],
        'completa': resultado['completa_recorrido'],
        'FN_max': resultado['FN_max_G'],
        'u_final': resultado['u_final']
    })

# Análisis
mu_validos = [r['mu'] for r in resultados_mu if r['valido']]

print("\n" + "="*80)
print("ESTUDIO 2: VARIACIÓN DE COEFICIENTE DE FRICCIÓN ( )")
print("="*80)
```

```
print(f"Parámetros fijos: m={m_base} kg, c_a={ca_base}, S_f={Sf_base} m²,␣
 ↪v ={v0_base} m/s")
print(f"\nRango explorado: {mu_values.min():.4f} - {mu_values.max():.4f}")
if mu_validos:
    print(f"  Rango válido: {min(mu_validos):.4f} - {max(mu_validos):.4f}")
else:
    print("  No hay valores válidos en el rango explorado")
print("="*80)

# Gráfica
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# FN_max vs
FN_values = [r['FN_max'] for r in resultados_mu]
mu_array = [r['mu'] for r in resultados_mu]
ax1.plot(mu_array, FN_values, 'b-', linewidth=2)
ax1.axhline(y=5, color='r', linestyle='--', linewidth=2, label='Límite F_N =␣
 ↪5g')
ax1.fill_between(mu_array, 0, 5, alpha=0.2, color='green', label='Zona válida␣
 ↪F_N')
ax1.set_xlabel('Coeficiente de fricción  ', fontsize=11)
ax1.set_ylabel('F_N máxima (g)', fontsize=11)
ax1.set_title('Fuerza Normal Máxima vs  ', fontsize=12, fontweight='bold')
ax1.legend()
ax1.grid(True, alpha=0.3)

# Posición final vs
u_final_values = [r['u_final'] for r in resultados_mu]
ax2.plot(mu_array, u_final_values, 'g-', linewidth=2)
ax2.axhline(y=0.95, color='r', linestyle='--', linewidth=2, label='Mínimo para␣
 ↪completar (0.95)')
ax2.axhline(y=1.0, color='orange', linestyle=':', linewidth=1, label='Final del␣
 ↪recorrido (1.0)')
ax2.fill_between(mu_array, 0.95, 1.1, alpha=0.2, color='green', label='Zona␣
 ↪válida')
ax2.set_xlabel('Coeficiente de fricción  ', fontsize=11)
ax2.set_ylabel('Posición final (u)', fontsize=11)
ax2.set_title('Posición Final vs  ', fontsize=12, fontweight='bold')
ax2.legend()
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

```
================================================================================
ESTUDIO 2: VARIACIÓN DE COEFICIENTE DE FRICCIÓN ( )
```

```
================================================================================
Parámetros fijos: m=800 kg, c_a=0.4, S_f=2.0 m², v =11.8 m/s

Rango explorado: 0.0050 - 0.0400
  Rango válido: 0.0057 - 0.0157
================================================================================
```



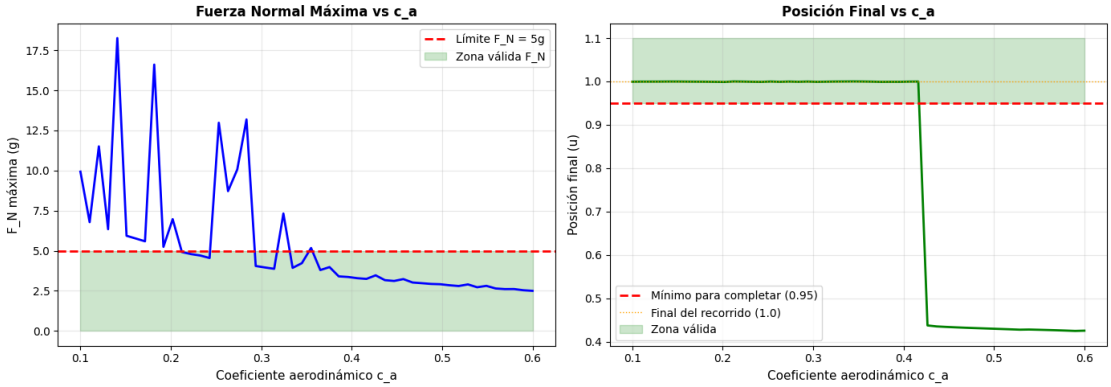
3.7.4 *Intervalo de coef aerodinámico*

[46]:
```python
# ============================================
# ESTUDIO 3: Variación de COEFICIENTE AERODINÁMICO (c_a)
# ============================================

# Parámetros base
m_base = 800
mu_base = 0.015
Sf_base = 2.0
v0_base = 11.8

# Rango de c_a a explorar
ca_values = np.linspace(0.1, 0.6, 50)

resultados_ca = []
for ca_test in ca_values:
    resultado = simular_montaña(m_base, mu_base, ca_test, Sf_base, v0_base)
    resultados_ca.append({
        'ca': ca_test,
        'valido': resultado['valido'],
        'completa': resultado['completa_recorrido'],
        'FN_max': resultado['FN_max_G'],
        'u_final': resultado['u_final']
    })
```

```python
# Análisis
ca_validos = [r['ca'] for r in resultados_ca if r['valido']]

print("\n" + "="*80)
print("ESTUDIO 3: VARIACIÓN DE COEFICIENTE AERODINÁMICO (c_a)")
print("="*80)
print(f"Parámetros fijos: m={m_base} kg,  ={mu_base}, S_f={Sf_base} m²,␣
 ↪v ={v0_base} m/s")
print(f"\nRango explorado: {ca_values.min():.3f} - {ca_values.max():.3f}")
if ca_validos:
    print(f"  Rango válido: {min(ca_validos):.3f} - {max(ca_validos):.3f}")
else:
    print("  No hay valores válidos en el rango explorado")
print("="*80)

# Gráfica
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# FN_max vs c_a
FN_values = [r['FN_max'] for r in resultados_ca]
ca_array = [r['ca'] for r in resultados_ca]
ax1.plot(ca_array, FN_values, 'b-', linewidth=2)
ax1.axhline(y=5, color='r', linestyle='--', linewidth=2, label='Límite F_N =␣
 ↪5g')
ax1.fill_between(ca_array, 0, 5, alpha=0.2, color='green', label='Zona válida␣
 ↪F_N')
ax1.set_xlabel('Coeficiente aerodinámico c_a', fontsize=11)
ax1.set_ylabel('F_N máxima (g)', fontsize=11)
ax1.set_title('Fuerza Normal Máxima vs c_a', fontsize=12, fontweight='bold')
ax1.legend()
ax1.grid(True, alpha=0.3)

# Posición final vs c_a
u_final_values = [r['u_final'] for r in resultados_ca]
ax2.plot(ca_array, u_final_values, 'g-', linewidth=2)
ax2.axhline(y=0.95, color='r', linestyle='--', linewidth=2, label='Mínimo para␣
 ↪completar (0.95)')
ax2.axhline(y=1.0, color='orange', linestyle=':', linewidth=1, label='Final del␣
 ↪recorrido (1.0)')
ax2.fill_between(ca_array, 0.95, 1.1, alpha=0.2, color='green', label='Zona␣
 ↪válida')
ax2.set_xlabel('Coeficiente aerodinámico c_a', fontsize=11)
ax2.set_ylabel('Posición final (u)', fontsize=11)
ax2.set_title('Posición Final vs c_a', fontsize=12, fontweight='bold')
ax2.legend()
ax2.grid(True, alpha=0.3)
```

```
plt.tight_layout()
plt.show()
```

```
================================================================================
ESTUDIO 3: VARIACIÓN DE COEFICIENTE AERODINÁMICO (c_a)
================================================================================
Parámetros fijos: m=800 kg,  =0.015, S_f=2.0 m², v =11.8 m/s

Rango explorado: 0.100 - 0.600
  Rango válido: 0.212 - 0.416
================================================================================
```



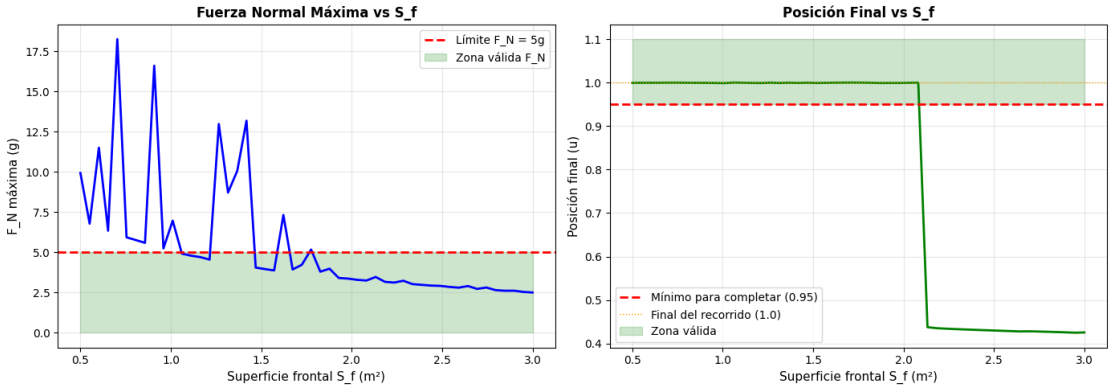
3.7.5 *Intervalo superficie frontal*

[47]:
```
# ============================================
# ESTUDIO 4: Variación de SUPERFICIE FRONTAL (S_f)
# ============================================

# Parámetros base
m_base = 800
mu_base = 0.015
ca_base = 0.4
v0_base = 11.8

# Rango de S_f a explorar (m²)
Sf_values = np.linspace(0.5, 3.0, 50)

resultados_Sf = []
for Sf_test in Sf_values:
    resultado = simular_montaña(m_base, mu_base, ca_base, Sf_test, v0_base)
    resultados_Sf.append({
```

```
        'Sf': Sf_test,
        'valido': resultado['valido'],
        'completa': resultado['completa_recorrido'],
        'FN_max': resultado['FN_max_G'],
        'u_final': resultado['u_final']
    })

# Análisis
Sf_validos = [r['Sf'] for r in resultados_Sf if r['valido']]

print("\n" + "="*80)
print("ESTUDIO 4: VARIACIÓN DE SUPERFICIE FRONTAL (S_f)")
print("="*80)
print(f"Parámetros fijos: m={m_base} kg,  ={mu_base}, c_a={ca_base},␣
 ↪v ={v0_base} m/s")
print(f"\nRango explorado: {Sf_values.min():.2f} - {Sf_values.max():.2f} m²")
if Sf_validos:
    print(f"  Rango válido: {min(Sf_validos):.2f} - {max(Sf_validos):.2f} m²")
else:
    print("  No hay valores válidos en el rango explorado")
print("="*80)

# Gráfica
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# FN_max vs S_f
FN_values = [r['FN_max'] for r in resultados_Sf]
Sf_array = [r['Sf'] for r in resultados_Sf]
ax1.plot(Sf_array, FN_values, 'b-', linewidth=2)
ax1.axhline(y=5, color='r', linestyle='--', linewidth=2, label='Límite F_N =␣
 ↪5g')
ax1.fill_between(Sf_array, 0, 5, alpha=0.2, color='green', label='Zona válida␣
 ↪F_N')
ax1.set_xlabel('Superficie frontal S_f (m²)', fontsize=11)
ax1.set_ylabel('F_N máxima (g)', fontsize=11)
ax1.set_title('Fuerza Normal Máxima vs S_f', fontsize=12, fontweight='bold')
ax1.legend()
ax1.grid(True, alpha=0.3)

# Posición final vs S_f
u_final_values = [r['u_final'] for r in resultados_Sf]
ax2.plot(Sf_array, u_final_values, 'g-', linewidth=2)
ax2.axhline(y=0.95, color='r', linestyle='--', linewidth=2, label='Mínimo para␣
 ↪completar (0.95)')
ax2.axhline(y=1.0, color='orange', linestyle=':', linewidth=1, label='Final del␣
 ↪recorrido (1.0)')
```

```
ax2.fill_between(Sf_array, 0.95, 1.1, alpha=0.2, color='green', label='Zona␣
 ↪válida')
ax2.set_xlabel('Superficie frontal S_f (m²)', fontsize=11)
ax2.set_ylabel('Posición final (u)', fontsize=11)
ax2.set_title('Posición Final vs S_f', fontsize=12, fontweight='bold')
ax2.legend()
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

```
================================================================================
ESTUDIO 4: VARIACIÓN DE SUPERFICIE FRONTAL (S_f)
================================================================================
Parámetros fijos: m=800 kg,  =0.015, c_a=0.4, v =11.8 m/s

Rango explorado: 0.50 - 3.00 m²
 Rango válido: 1.06 - 2.08 m²
================================================================================
```



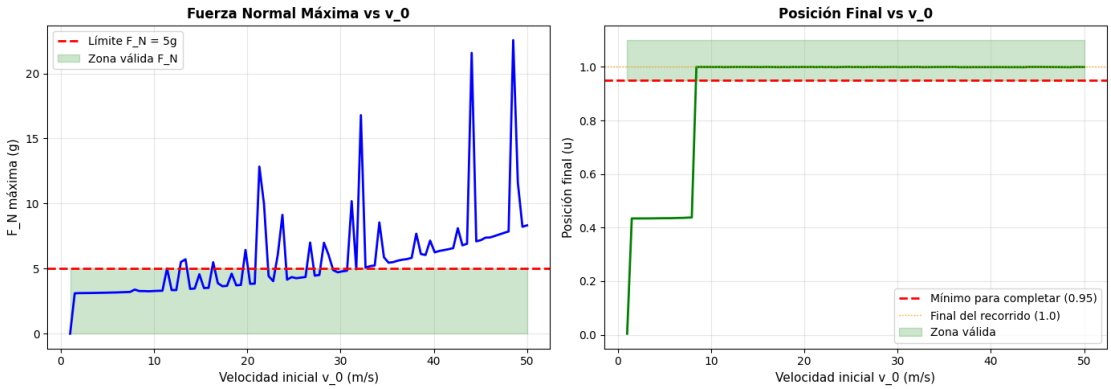
3.7.6 *Intervalo velocidad incial*

[48]:
```
# ============================================
# ESTUDIO 5: Variación de VELOCIDAD INICIAL (v_0)
# ============================================

# Parámetros base
m_base = 800
mu_base = 0.015
ca_base = 0.4
Sf_base = 2.0
```

```python
# Rango de v_0 a explorar (m/s)
v0_values = np.linspace(1.0, 50, 100)

resultados_v0 = []
for v0_test in v0_values:
    resultado = simular_montaña(m_base, mu_base, ca_base, Sf_base, v0_test)
    resultados_v0.append({
        'v0': v0_test,
        'valido': resultado['valido'],
        'completa': resultado['completa_recorrido'],
        'FN_max': resultado['FN_max_G'],
        'u_final': resultado['u_final']
    })

# Análisis
v0_validos = [r['v0'] for r in resultados_v0 if r['valido']]

print("\n" + "="*80)
print("ESTUDIO 5: VARIACIÓN DE VELOCIDAD INICIAL (v_0)")
print("="*80)
print(f"Parámetros fijos: m={m_base} kg,  ={mu_base}, c_a={ca_base},␣
 ↪S_f={Sf_base} m²")
print(f"\nRango explorado: {v0_values.min():.2f} - {v0_values.max():.2f} m/s")
if v0_validos:
    print(f"  Rango válido: {min(v0_validos):.2f} - {max(v0_validos):.2f} m/s")
else:
    print("  No hay valores válidos en el rango explorado")
print("="*80)

# Gráfica
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# FN_max vs v_0
FN_values = [r['FN_max'] for r in resultados_v0]
v0_array = [r['v0'] for r in resultados_v0]
ax1.plot(v0_array, FN_values, 'b-', linewidth=2)
ax1.axhline(y=5, color='r', linestyle='--', linewidth=2, label='Límite F_N =␣
 ↪5g')
ax1.fill_between(v0_array, 0, 5, alpha=0.2, color='green', label='Zona válida␣
 ↪F_N')
ax1.set_xlabel('Velocidad inicial v_0 (m/s)', fontsize=11)
ax1.set_ylabel('F_N máxima (g)', fontsize=11)
ax1.set_title('Fuerza Normal Máxima vs v_0', fontsize=12, fontweight='bold')
ax1.legend()
ax1.grid(True, alpha=0.3)

# Posición final vs v_0
```

```
u_final_values = [r['u_final'] for r in resultados_v0]
ax2.plot(v0_array, u_final_values, 'g-', linewidth=2)
ax2.axhline(y=0.95, color='r', linestyle='--', linewidth=2, label='Mínimo para␣
 ↪completar (0.95)')
ax2.axhline(y=1.0, color='orange', linestyle=':', linewidth=1, label='Final del␣
 ↪recorrido (1.0)')
ax2.fill_between(v0_array, 0.95, 1.1, alpha=0.2, color='green', label='Zona␣
 ↪válida')
ax2.set_xlabel('Velocidad inicial v_0 (m/s)', fontsize=11)
ax2.set_ylabel('Posición final (u)', fontsize=11)
ax2.set_title('Posición Final vs v_0', fontsize=12, fontweight='bold')
ax2.legend()
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

```
================================================================================
ESTUDIO 5: VARIACIÓN DE VELOCIDAD INICIAL (v_0)
================================================================================
Parámetros fijos: m=800 kg,  =0.015, c_a=0.4, S_f=2.0 m²

Rango explorado: 1.00 - 50.00 m/s
  Rango válido: 8.42 - 31.69 m/s
================================================================================
```



3.7.7 *Resumen final*

Hemos podido ver los intervalos para los cuales es válido, aunque parece un poco extraño presentando picos aparentemente anómalos.

La suposición que hago acerca de esto es como construe solve_ivp las soluciones y como se construye la curva paramétrica. Si dos puntos resulta que no están perfectamente alineados, algo que resulta

un poco complicado, podría hacer que se generarse una curvatura muy grande, casi inapreciable para el ojo humano. solve_ivp podría encontrarse o no con ese punto exacto de mucha curvatura, cosa que no pasa habitualmente, pero puede pasar. En 2D es más difícil que esto ocurra pero las definiciones en 3D son más suceptibles a esto. Por lo tanto queda creer que estos puntos son anómalos y nos interesa mas la tendencia como podíamos ver en 2D, que podemos ver como las tendencias son realmente similares.

3.8 *Animación final*

[49]:
```
# ======================================================================
# 1. PREPARAR DATOS PARA ANIMACIÓN
# ======================================================================

print("="*80)
print("PREPARANDO DATOS PARA ANIMACIÓN 3D")
print("="*80)

# Parámetros de interpolación
t_max_sim = sol_viable.t[-1]
n_frames = 300
t_anim = np.linspace(0, t_max_sim, n_frames)

# Datos interpolados de la solución
u_anim = sol_viable.sol(t_anim)[0]
v_anim = sol_viable.sol(t_anim)[1]

# Convertir u -> posiciones 3D (x, y, z)
pos_anim = np.array([Curva_parametrica(u) for u in u_anim])

# Calcular magnitudes para cada frame
energias_anim = []
fuerzas_G_anim = []
aceleraciones_anim = []

for i in range(n_frames):
    # Energía
    E_frame = cmr.energia(u_anim[i], v_anim[i], derivadas_curva[0], g)
    energias_anim.append(E_frame)

    # Fuerza Normal
    try:
        fuerzaN, baseLocal, ctes = cmr.fuerzaNormal(u_anim[i], v_anim[i],␣
 ↪derivadas_curva, g)
        fuerzas_G_anim.append(fuerzaN / g)
    except:
        fuerzas_G_anim.append(0)
```

```
    # Aceleración
    aceleraciones_anim.append(v_anim[i])

print(f"  Frames totales: {n_frames}")
print(f"  Duración simulación: {t_max_sim:.2f} segundos")
print(f"  Rango de posición: u   [0, {u_anim[-1]:.3f}]")
print(f"  Velocidad: v   [{v_anim.min():.2f}, {v_anim.max():.2f}] m/s")
print(f"  F_N máxima: {max(fuerzas_G_anim):.2f} g")
print("="*80 + "\n")
```

```
================================================================================
PREPARANDO DATOS PARA ANIMACIÓN 3D
================================================================================
  Frames totales: 300
  Duración simulación: 120.20 segundos
  Rango de posición: u   [0, 0.999]
  Velocidad: v   [3.76, 32.54] m/s
  F_N máxima: 3.28 g
================================================================================
```

[ ]:
```
# ============================================================================
# 2. CREAR FIGURA 3D CON SUBPLOTS
# ============================================================================

fig = plt.figure(figsize=(18, 11))
fig.suptitle('Montaña Rusa 3D - Animación Interactiva del Movimiento',
             fontsize=16, fontweight='bold', y=0.98)

# Subplot principal: Vista 3D (ocupa 2x2)
ax_3d = fig.add_subplot(2, 3, (1, 4), projection='3d')

# Dibujar pista completa
t_pista = np.linspace(0, 1, 1500)
pista_3d = np.array([Curva_parametrica(u) for u in t_pista])

ax_3d.plot(pista_3d[:, 0], pista_3d[:, 1], pista_3d[:, 2],
           'k-', linewidth=2.5, alpha=0.3, label='Pista completa', zorder=1)

# Elementos animables en 3D
vagon_3d, = ax_3d.plot([], [], [], 'o', color='#FF0000', markersize=12,
                       label='Vagón', zorder=5, markeredgecolor='darkred',␣
 ↪markeredgewidth=1.5)
estela_3d, = ax_3d.plot([], [], [], '--', color='#FF6B6B', linewidth=1.2,
                        alpha=0.6, label='Trayectoria reciente', zorder=2)

# Configuración de ejes 3D
```

```
ax_3d.set_xlabel('X (m)', fontsize=10, fontweight='bold')
ax_3d.set_ylabel('Y (m)', fontsize=10, fontweight='bold')
ax_3d.set_zlabel('Z (m)', fontsize=10, fontweight='bold')
ax_3d.set_title('Vista 3D del Movimiento', fontsize=12, fontweight='bold',␣
 ↪pad=15)
ax_3d.legend(loc='upper left', fontsize=9)

# Limites
margen_x = 40
margen_y = 40
margen_z = 15
ax_3d.set_xlim(pista_3d[:, 0].min() - margen_x, pista_3d[:, 0].max() + margen_x)
ax_3d.set_ylim(pista_3d[:, 1].min() - margen_y, pista_3d[:, 1].max() + margen_y)
ax_3d.set_zlim(pista_3d[:, 2].min() - margen_z, pista_3d[:, 2].max() + margen_z)

# ======================================================================
# Subplot 2: Velocidad vs Tiempo
# ======================================================================

ax_vel = fig.add_subplot(2, 3, 2)
ax_vel.plot(t_anim, v_anim, 'b-', linewidth=2.5, alpha=0.8, label='Velocidad')
ax_vel.fill_between(t_anim, 0, v_anim, alpha=0.15, color='blue')
linea_tiempo_vel, = ax_vel.plot([], [], 'o', color='red', markersize=8,␣
 ↪label='Posición actual')
ax_vel.set_xlabel('Tiempo (s)', fontsize=10, fontweight='bold')
ax_vel.set_ylabel('Velocidad (m/s)', fontsize=10, fontweight='bold')
ax_vel.set_title('Velocidad vs Tiempo', fontsize=11, fontweight='bold')
ax_vel.grid(True, alpha=0.3, linestyle='--')
ax_vel.legend(fontsize=9, loc='best')
ax_vel.set_xlim(0, t_max_sim)

# ======================================================================
# Subplot 3: Fuerza Normal vs Tiempo
# ======================================================================

ax_fn = fig.add_subplot(2, 3, 3)
ax_fn.plot(t_anim, fuerzas_G_anim, 'g-', linewidth=2.5, alpha=0.8,␣
 ↪label='Fuerza Normal (F_N)')
ax_fn.fill_between(t_anim, 0, fuerzas_G_anim, alpha=0.15, color='green')
ax_fn.axhline(y=5, color='r', linestyle='--', linewidth=2, label='Límite (5g)',␣
 ↪zorder=3)
ax_fn.fill_between(t_anim, 0, 5, alpha=0.08, color='red')
linea_tiempo_fn, = ax_fn.plot([], [], 'o', color='red', markersize=8,␣
 ↪label='Posición actual')
ax_fn.set_xlabel('Tiempo (s)', fontsize=10, fontweight='bold')
ax_fn.set_ylabel('Fuerza Normal (g)', fontsize=10, fontweight='bold')
ax_fn.set_title('Fuerza Normal vs Tiempo', fontsize=11, fontweight='bold')
```

```
ax_fn.grid(True, alpha=0.3, linestyle='--')
ax_fn.legend(fontsize=9, loc='best')
ax_fn.set_xlim(0, t_max_sim)
ax_fn.set_ylim(min(fuerzas_G_anim) - 0.5, 6)

# ======================================================================
# Subplot 4: Energía vs Tiempo
# ======================================================================

ax_E = fig.add_subplot(2, 3, 5)
ax_E.plot(t_anim, energias_anim, 'm-', linewidth=2.5, alpha=0.8, label='Energía␣
 ↪Total')
ax_E.fill_between(t_anim, 0, energias_anim, alpha=0.15, color='magenta')
linea_tiempo_E, = ax_E.plot([], [], 'o', color='red', markersize=8,␣
 ↪label='Posición actual')
ax_E.set_xlabel('Tiempo (s)', fontsize=10, fontweight='bold')
ax_E.set_ylabel('Energía (J/kg)', fontsize=10, fontweight='bold')
ax_E.set_title('Energía Total vs Tiempo', fontsize=11, fontweight='bold')
ax_E.grid(True, alpha=0.3, linestyle='--')
ax_E.legend(fontsize=9, loc='best')
ax_E.set_xlim(0, t_max_sim)

# ======================================================================
# Subplot 5: Información General (texto)
# ======================================================================

ax_info = fig.add_subplot(2, 3, 6)
ax_info.axis('off')

texto_info = ax_info.text(0.05, 0.95, '', transform=ax_info.transAxes,
                          fontsize=10, verticalalignment='top',␣
 ↪family='monospace',
                          bbox=dict(boxstyle='round', facecolor='#FFFFCC',
                                    alpha=0.85, pad=1, linewidth=1.5),
                          linespacing=1.8)

print("  Figura 3D multi-subplot creada")
```

[ ]:
```
# ======================================================================
# 3. FUNCIONES DE ANIMACIÓN
# ======================================================================

def init():
    """Inicializar animación"""
    vagon_3d.set_data([], [])
    vagon_3d.set_3d_properties([])
    estela_3d.set_data([], [])
```

```
    estela_3d.set_3d_properties([])
    linea_tiempo_vel.set_data([], [])
    linea_tiempo_fn.set_data([], [])
    linea_tiempo_E.set_data([], [])
    texto_info.set_text('')
    return vagon_3d, estela_3d, linea_tiempo_vel, linea_tiempo_fn,␣
 ↪linea_tiempo_E, texto_info

def animate(frame):
    """Actualizar animación en cada frame"""

    # Posición del vagón en 3D
    x_vagon = pos_anim[frame, 0]
    y_vagon = pos_anim[frame, 1]
    z_vagon = pos_anim[frame, 2]

    vagon_3d.set_data([x_vagon], [y_vagon])
    vagon_3d.set_3d_properties([z_vagon])

    # Estela (últimos 80 frames)
    inicio_estela = max(0, frame - 80)
    estela_3d.set_data(pos_anim[inicio_estela:frame+1, 0],
                       pos_anim[inicio_estela:frame+1, 1])
    estela_3d.set_3d_properties(pos_anim[inicio_estela:frame+1, 2])

    # Marcadores en gráficas
    linea_tiempo_vel.set_data([t_anim[frame]], [v_anim[frame]])
    linea_tiempo_fn.set_data([t_anim[frame]], [fuerzas_G_anim[frame]])
    linea_tiempo_E.set_data([t_anim[frame]], [energias_anim[frame]])

    # Rotación suave de vista 3D
    if frame % 3 == 0:
        azim = 30 + (frame / n_frames) * 330
        ax_3d.view_init(elev=20, azim=azim)

    # Texto informativo
    porcentaje = (u_anim[frame] / u_anim[-1]) * 100 if u_anim[-1] > 0 else 0
    energia_perdida = energias_anim[0] - energias_anim[frame]
    potencia = energia_perdida / (t_anim[frame] + 1e-6)

    info_text = (
        f"  TIEMPO\n"
        f"   {t_anim[frame]:.2f} s\n\n"
        f" VELOCIDAD\n"
        f"   {v_anim[frame]:.2f} m/s\n\n"
        f" ENERGÍA\n"
        f"   {energias_anim[frame]:.2f} J/kg\n"
```

```
        f"   Pérdida: {energia_perdida:.2f} J/kg\n\n"
        f"  FUERZA NORMAL\n"
        f"   {fuerzas_G_anim[frame]:.2f} g\n"
        f"   Límite: 5.0 g\n\n"
        f" POSICIÓN\n"
        f"   {porcentaje:.1f}% recorrido\n"
        f"   u = {u_anim[frame]:.3f}"
    )
    texto_info.set_text(info_text)

    return vagon_3d, estela_3d, linea_tiempo_vel, linea_tiempo_fn,␣
 ↪linea_tiempo_E, texto_info

# Crear animación
anim = FuncAnimation(fig, animate, init_func=init,
                     frames=n_frames, interval=50, blit=True, repeat=True)

plt.tight_layout()
plt.show()

print("  Animación 3D creada correctamente")
```

[ ]:
```
# ======================================================================
# 6. MOSTRAR ANIMACIÓN EN EL NOTEBOOK
# ======================================================================

# Convertir animación a HTML5 para desplegar en notebook
html_anim_3d = HTML(anim.to_html5_video())

# Mostrar en el notebook
html_anim_3d
```

4 *Conclusiones*

Hemos observado las dificultades de hacer una montaña rusa y sus requerimentos.

Durante la ejecución de la misma me he encontrado con un montón de problemas, ya sea que no se haga bien la curva parámetrica, averiguar que importa el orden a la hora de hacerla, y lo dificil que es a veces ajustar las cosas.

Con un ordenador mejor podrían obtenerse resultados más certeros, y con más tiempo de computación obviamente podriamos aumentar el número de puntos de la solución densa de solve_ivp y hacerle que los pasos decrezcan para no encontrarnos con estas anomalías, además de poder añadir puntos y tratar de hacer curvas más suaves a la hora de interpolar.

Pero en resumen creo que este documento recoge todo lo necesario para desarrollar una montaña rusa, desde pequeñas sutilezas de teoría y todo el codigo necesario para hacerlo.