RaVThOughT

RaVThOughT Navigation Reference Frame

Abstract

1. Introduction

1.1 Current Approaches and Limitations

1.2 Modern Challenges in Spacecraft Guidance

-
-
-

1.3 The RaVThOughT Solution

2. Methods

2.1 Core Framework and Coordinate System Definition

$\hat{R}$

2.1.1 Coordinate System Safety

2.1.2 State Specification

```
RaVThOughT = {
    position: (x, y, z),     // Target position relative to reference
    velocity: (vx, vy, vz),  // Required velocity to achieve next state
    theta: (θx, θy, θz),     // Required orientation for maneuver
    omega: (wx, wy, wz),     // Required angular rates for maneuver
    time: absolute_t,        // Absolute mission time of achievement
    reference: prev_state    // Previous validated state
}
```

2.1.3 Reference Point Mechanics and Chain Initialization

```
ECEF_Initial or ECI_Initial → RaVThOughT_1 → RaVThOughT_2 → ... →
RaVThOughT_Terminal
```

2.2 Gravitational Rectification

2.2.1 Time Window Selection

-
-
-
-

2.2.2 Temporal Sampling and Interpolation

```
F_g(t₀) = propagator.gravity(state₀)  # Start of window
F_g(t₁) = propagator.gravity(state₁)  # End of window (t₀ + 100s)
```

quadratic inte

$F_g(t)$ $F_{gx}(t)$ $F_{gy}(t)$ $F_{gz}(t)$

$$F_g(t) = a(t - t_0)^2 + b(t - t_0) + c$$

a b c

t_0 t_{mid} t_1

2.2.3 Error Analysis

quadratic

Position Error Bounds

-
-
-
  -
  -

Error Sources

-
-

- Error Mitigation
-
-
-

2.2.4 Cumulative Integration

t

$$\vec{p}(t) = \vec{p}_0 + \sum_{i=1}^{n} \vec{T}_i + \vec{v}_0 t \; + \; \frac{1}{m} \int_0^t (t - \tau)\, \vec{F}_g^{(quad)}(\tau)\, d\tau$$

- $\vec{p}_0$
- $\vec{v}_0$
- $\vec{T}_i$
- $\vec{F}_g^{(quad)}(\tau)$ quadratic
- m

2.2.5 Computational Performance

-
-
-
-
-
-

-
-
-

2.2.6 Implementation Considerations

-

2.3 Implementation

2.3.1 Example

Scenario:

t_0

- Positpi$_0$on:
- Velocvi$_0$ty:

Identify the 100-Second Interval:

t_0 t_0 + 100 s

Quadratic Interpolation of Gravity:

t_0 $t_0 + 100\,s$

t_0 t_{mid} t_1 t

$$\vec{F}_g(t) = \vec{A}(t - t_0)^2 + \vec{B}(t - t_0) + \vec{C}$$

$\vec{A}$ $\vec{B}$ $\vec{C}$

$t_0, \$t_{mid}$ t_1

Defining the RaVThOughT Frame at t

t_0

- +X axis:
- +Z axis:
- +Y axis: $+Y = +X \times +Z$

Quadratic Interpolation of Gravity:

t_0 $t_0 + 100s$

t_0 t_{mid} t_1 t

$$\vec{F}_g(t) = \vec{A}(t - t_0)^2 + \vec{B}(t - t_0) + \vec{C}$$

$\vec{A}$ $\vec{B}$ $\vec{C}$

t_0 t_{mid} t_1

Executing a Maneuver in the RaVThOughT Fra

Δv

- To move forward:
- No need for complex orbital-frame rotati rates.

End of the Interval Data Approach: State Vector

-
- $t_0 + 100\,s$

Key Takeaway:

2.3.2 Comparison

Implementation Complexity: Traditional Approach vs. Raw Thought

Traditional Implementation Sketch:

```
def plan_maneuver_traditional(initial_state_eci, desired_delta_v):
    planned_thrust = []
    dt = 1.0  # second-step increments for simulation
    current_state = initial_state_eci
```

```
    for t in range(100):
        # 1. Propagate orbit from current_state for dt
        propagated_state = propagate_orbit(current_state, dt)

        # 2. Transform to a local orbital frame (e.g., LVLH)
        local_frame_state = transform_to_LVLH(propagated_state)

        # 3. Compute gravitational accelerations, orbital rates, and Coriolis
terms
        gravity = compute_gravity(propagated_state)
        orbital_rates = compute_orbital_rates(propagated_state)
        coriolis_acc = compute_coriolis(orbital_rates,
local_frame_state.velocity)

        # 4. Determine thrust needed this second to achieve part of the
desired Δv
        #    This may require iterative adjustment and complex control logic.
        incremental_thrust = solve_for_required_thrust(
            local_frame_state, gravity, coriolis_acc, desired_delta_v
        )

        planned_thrust.append(incremental_thrust)
        current_state = apply_thrust(current_state, incremental_thrust, dt)

    return planned_thrust
```

- Orbit Propagation:
- Frame Transformation:
- Dynamic Calculations:

- Iterative Thrust Calculation:

RaVThOughT Implementation Sketch:

```
def plan_maneuver_ravthought(initial_state_eci, desired_delta_v):
    # Step 1: Define the RaVThOughT interval (t0 to t0+100s)
```

```
    t0 = initial_state_eci.time
    t1 = t0 + 100

    # Step 2: Get gravitational acceleration at start, midpoint, and end
    t_mid = t0 + 50  # Midpoint for quadratic interpolation
    g0 = compute_gravity(initial_state_eci, t0)             # Gravity at t0
    g_mid = compute_gravity(initial_state_eci, t_mid)       # Gravity at t_mid
    final_state_eci = propagate_orbit(initial_state_eci, 100)
    g1 = compute_gravity(final_state_eci, t1)              # Gravity at t1

    # Step 3: Construct local RaVThOughT frame at t0
    local_state = transform_eci_to_ravthought(initial_state_eci)

    # Step 4: Desired Δv is simple: just add it to local x-axis (for example)
    # Apply quadratic interpolation for gravity
    incremental_thrust = compute_quadratic_thrust(local_state,
desired_delta_v, g0, g_mid, g1, 100)

    # In RaVThOughT, this can be done analytically or with minimal iteration
    # since gravity is approximated quadratically and there is no frame re-
derivation each second.

    return [incremental_thrust]  # Often just one planned maneuver command for
the whole interval
```

- Gravity Sampling: t_0 t_{mid} t_1
- Quadratic Gravity Estimation:
- Frame Stability:
- Simplified Calculations:
- Guidance Simplicity:

Key Differences:

Aspect	TraditAipomradach	RaVThOuAgphTroach (Quadratic)
Frame Upd		
Gravity Modeling		
Complexi Math		
Computat Load		
Suitabil ML		

2.4 Compound RaVThOughT Frames

2.4.1 Two-Vehicle Framework

-
-
-
-

quadratic inte

2.4.2 Compound State Specification

```
CompoundRaVThOughT = {
    spacecraft_A: {
        position: (x, y, z),     // Relative to compound origin
        velocity: (vx, vy, vz),  // Required velocity
        theta: (θx, θy, θz),     // Required orientation
        omega: (ωx, ωy, ωz)      // Required angular rates
    },
    spacecraft_B: {
        position: (x, y, z),
        velocity: (vx, vy, vz),
        theta: (θx, θy, θz),
        omega: (ωx, ωy, ωz)
    },
    time: t,                     // Achievement time (≤ 100s)
    reference: prev_compound     // Previous compound state
}
```

2.4.3 Reference Chain Properties

```
Initial_Compound → CompoundRaVThOughT_1 → CompoundRaVThOughT_2 → ...
```

2.4.4 Applications

2.4.5 Frame Transitions

Individual to Compound Transition

```
Spacecraft A:  RaVThOughT_A1 → RaVThOughT_A2 → RaVThOughT_A_Final
                                                        ↓
Compound:                                       CompoundRaVThOughT_1 → ...
                                                        ↑
Spacecraft B:  RaVThOughT_B1 → RaVThOughT_B2 → RaVThOughT_B_Final
```

Compound to Individual Transition

```
Compound:     ... → CompoundRaVThOughT_Final

Spacecraft A:    RaVThOughT_A_New → ...
Spacecraft B:                         RaVThOughT_B_New → ...
```

Maintaining Chain Validity

-
-
-
-

2.4.6 Error Handling and Fault Recovery

```
class StateAchievementError(Exception):
    def __init__(self, intended_state, actual_state, time_delta):
        self.recovery_options = calculate_recovery_paths(
            actual_state,
            time_delta
        )
```

-
    -
    -
    -

```
def handle_transition_failure(failed_transition):
    # Options in priority order:
    # 1. Extend individual chains briefly
    # 2. Establish new compound frame
    # 3. Return to independent operations
    return select_recovery_strategy(failed_transition)
```

-
-
-
-

-
-
-
-

2.4.7 Example: Formation Flying Initialization

-
-
-
-

-
-
-
-

```
# Initial states in ECI (showing the complexity we're abstracting away)
spacecraft_A_initial = ECIState(
    position=(6878.0, 0.0, 0.0),      # km
    velocity=(0.0, 7.67, 0.0),        # km/s
    time=0                            # mission start
)

spacecraft_B_initial = ECIState(
    position=(6878.0, 0.1, 0.0),      # slightly ahead
    velocity=(0.0, 7.67, 0.02),       # slight drift
    time=0
```

```
)

# First, each spacecraft establishes its individual RaVThOughT chain
# moving toward the desired formation position

# Spacecraft A needs to adjust position to prepare for formation
chain_A = [
    # Start with reference to absolute position
    RaVThOughT(
        position=(0, 0, 0),           # starting point
        velocity=(0, 0, 0),           # current velocity
        theta=(0, 0, 0),              # current attitude
        omega=(0, 0, 0),              # no rotation
        time=0,
        reference=spacecraft_A_initial
    ),
    # Move to approximate formation position
    RaVThOughT(
        position=(0.2, 0, 0.1),       # rough correction
        velocity=(0.002, 0, 0.001),   # ~2 m/s approach
        theta=(0, 0.1, 0),            # slight reorientation
        omega=(0, 0, 0),
        time=100,                     # using full window
        reference=chain_A[0]          # reference previous state
    )
]

# Spacecraft B does the same from its position
chain_B = [
    RaVThOughT(
        position=(0, 0, 0),
        velocity=(0, 0, 0),
        theta=(0, 0, 0),
        omega=(0, 0, 0),
        time=0,
        reference=spacecraft_B_initial
    ),
    RaVThOughT(
        position=(-0.15, 0, -0.1),    # moving to meet A
        velocity=(-0.0015, 0, -0.001),
        theta=(0, -0.1, 0),
        omega=(0, 0, 0),
        time=100,
        reference=chain_B[0]
    )
]
```

```
# Once both spacecraft are roughly positioned, establish compound frame
# Note how the formation is now expressed in simple relative terms
compound_chain = [
    CompoundRaVThOughT(
        spacecraft_A={
            position=(-0.5, 0, 0),     # 500m back from midpoint
            velocity=(0, 0, 0),        # matched velocities
            theta=(0, 0, 0),           # aligned attitudes
            omega=(0, 0, 0)
        },
        spacecraft_B={
            position=(0.5, 0, 0),      # 500m forward from midpoint
            velocity=(0, 0, 0),
            theta=(0, 0, 0),
            omega=(0, 0, 0)
        },
        time=200,                      # next 100s window
        reference=(chain_A[-1], chain_B[-1])
    ),
    # Maintain formation
    CompoundRaVThOughT(
        spacecraft_A={
            position=(-0.5, 0, 0),     # maintaining position
            velocity=(0, 0, 0),
            theta=(0, 0, 0),
            omega=(0, 0, 0)
        },
        spacecraft_B={
            position=(0.5, 0, 0),
            velocity=(0, 0, 0),
            theta=(0, 0, 0),
            omega=(0, 0, 0)
        },
        time=300,
        reference=compound_chain[0]
    )
]

# The framework handles conversion to true orbital positions
def get_formation_truth(compound_state):
    # Propagate individual states to current time
    truth_A = propagate_orbit(spacecraft_A_initial, compound_state.time)
    truth_B = propagate_orbit(spacecraft_B_initial, compound_state.time)

    # Calculate compound frame origin in true coordinates
```

```
    origin = (truth_A + truth_B) / 2

    # Convert compound frame positions to ECI
    eci_positions = transform_compound_to_eci(
        compound_state,
        origin,
        truth_A,
        truth_B
    )

    return eci_positions

# Validation shows both relative and absolute accuracy
def validate_formation(compound_state):
    # Check relative spacing
    positions = get_formation_truth(compound_state)
    actual_separation = magnitude(
        positions.spacecraft_B - positions.spacecraft_A
    )

    # Verify orbital parameters
    orbital_state = calculate_orbital_elements(positions)

    return {
        'separation_error': abs(actual_separation - 1.0),  # should be < 0.01
km
        'orbital_elements': orbital_state,
        'is_valid': abs(actual_separation - 1.0) < 0.01
    }
```

-
-
-
-
-
-
-

2.5 Extended RaVThOughT Frames for Multi-Vehicle Operations

2.5.1 Hierarchical Frame Structure

```
GFF = {
    origin: barycenter(spacecraft_list),
    x_axis: formation_primary_axis,      # Mission-specific definition
    z_axis: -R_earth,                    # Toward Earth center
    y_axis: cross(z_axis, x_axis)        # Maintains left-handed system
}
```

```
SFF = {
    origin: barycenter(subgroup),
    orientation: relative_to_GFF,
    members: spacecraft_list,
    bounds: spatial_limits          # For dynamic regrouping
}
```

2.5.2 Formation Decomposition

```
def decompose_formation(spacecraft_list, mission_params):
    # Initial spatial clustering
    clusters = octree_spatial_decomposition(spacecraft_list)

    # Refine based on operational requirements
    sub_formations = []
    for cluster in clusters:
        if requires_tight_coordination(cluster):
            sub_formations.append(CompoundFrame(cluster))
        elif len(cluster) == 1:
            sub_formations.append(IndividualFrame(cluster[0]))
        else:
            sub_formations.append(
                LooseFormationFrame(cluster, coordination_params)
            )

    return sub_formations

def update_formation_structure(current_formation, new_states):
    # Check for restructuring needs
    if needs_reorganization(current_formation, new_states):
        new_formation = decompose_formation(
            new_states.spacecraft_list,
            new_states.mission_params
        )
        return plan_formation_transition(
            current_formation,
            new_formation
        )
    return current_formation
```

2.5.3 State Management and Propagation

```
ExtendedRaVThOughTState = {
    global_frame: {
        origin: Vector3,           # ECI coordinates
        orientation: Quaternion,   # Global frame orientation
        time: float               # Mission time
```

```
    },
    sub_formations: [
        {
            type: enum{'compound', 'loose', 'individual'},
            origin: Vector3,       # Relative to global frame
            spacecraft: [
                {
                    id: string,
                    position: Vector3,    # Relative to sub-formation
                    velocity: Vector3,
                    attitude: Quaternion,
                    angular_velocity: Vector3,
                    constraints: {
                        max_acceleration: float,
                        fuel_remaining: float,
                        operational_status: enum
                    }
                }
            ],
            coordination_params: {
                update_frequency: float,
                communication_latency: float,
                position_tolerance: float
            }
        }
    ],
    reference_chain: [
        previous_states: ExtendedRaVThOughTState
    ]
}
```

2.5.4 Multi-Vehicle Gravitational Rectification

$$F_{g,i}(t) = F_{g,i}(t_0) + \sum_{j \neq i} \Delta F_{g,ij}(t)$$

$$\Delta F_{g,ij}(t)$$

2.5.5 Formation Maintenance and Error Management

-
-
-

```
def handle_formation_error(error, formation_state):
    if error.severity > CRITICAL_THRESHOLD:
        return emergency_reformation(formation_state)

    if affects_multiple_subformations(error):
        return restructure_formation(formation_state, error)

    return local_error_correction(formation_state, error)
```

```
def recover_formation_integrity(formation_state):
    # Assess current formation health
    health = evaluate_formation_health(formation_state)

    # Generate recovery options
    options = generate_recovery_paths(
        formation_state,
        health.issues
    )

    # Select optimal recovery strategy
    strategy = optimize_recovery_path(
        options,
        formation_state.constraints
    )

    return execute_recovery(strategy)
```

2.5.6 Scalability Analysis

-
-
-

2.5.7 Practical Applications

Appendix

Demonstration of 100 second window error in elliptical LEO orbit

```
import numpy as np
from scipy.integrate import odeint, quad
from scipy.interpolate import interp1d
```

```python
import matplotlib.pyplot as plt

# ------------------------------------------------------
# This script:
# 1. Defines a high-fidelity gravity model (with J2).
# 2. Sets up an initial elliptical orbit and simulates from t0 to t1.
# 3. Measures gravity at t0, t_mid, and t1.
# 4. Uses quadratic interpolation of gravity based on three points as a model.
# 5. Uses calculus-based integration to compute aggregate and average errors.
# 6. Prints the aggregate error and average error in m/s².
# 7. Visualizes the errors and acceleration components.
# ------------------------------------------------------

# ------------------------------------------------------
# Constants and Initial Conditions
# ------------------------------------------------------
GM_EARTH = 3.986004418e14  # Earth's gravitational parameter [m^3/s^2]
R_EARTH = 6378137.0        # Earth's mean radius [m]
J2 = 1.08262668e-3          # Earth's J2 term

# Define an elliptical orbit with specific perigee and apogee
perigee_altitude = 400e3    # Perigee altitude [m]
apogee_altitude = 800e3     # Apogee altitude [m]

# Calculate orbital radii
r_p = R_EARTH + perigee_altitude  # Perigee radius [m]
r_a = R_EARTH + apogee_altitude   # Apogee radius [m]

# Calculate semi-major axis and eccentricity
a = (r_p + r_a) / 2.0              # Semi-major axis [m]
e = (r_a - r_p) / (r_a + r_p)      # Eccentricity

print(f"Defined Elliptical Orbit:")
print(f"Perigee Radius (m): {r_p}")
print(f"Apogee Radius (m): {r_a}")
print(f"Semi-Major Axis (m): {a}")
print(f"Eccentricity: {e:.4f}\n")

# Calculate initial velocity at perigee using the vis-viva equation
v_p = np.sqrt(GM_EARTH * (2.0 / r_p - 1.0 / a))
print(f"Calculated Initial Velocity at Perigee (m/s): {v_p:.2f}\n")

# Initial state vector at perigee: [x, y, z, vx, vy, vz]
# Position at perigee along the x-axis, velocity perpendicular in the y-direction
state0 = np.array([r_p, 0.0, 0.0, 0.0, v_p, 0.0])
```

```
# ------------------------------------------------------
# Gravity Model with J2
# ------------------------------------------------------
def gravity_full_j2(state):
    """
    Compute gravitational acceleration with J2 perturbation.    Output:
Acceleration vector [m/s^2].    """    x, y, z = state[0], state[1], state[2]
    r = np.sqrt(x ** 2 + y ** 2 + z ** 2)

    if r == 0:
        raise ValueError("Division by zero encountered in gravity
calculation.")

    ux, uy, uz = x / r, y / r, z / r

    # Central gravity
    a_cx = -GM_EARTH * ux / (r ** 2)
    a_cy = -GM_EARTH * uy / (r ** 2)
    a_cz = -GM_EARTH * uz / (r ** 2)

    # J2 perturbation
    factor = (3.0 / 2.0) * J2 * (GM_EARTH / (r ** 2)) * ((R_EARTH ** 2) / (r
** 2))
    a_j2x = factor * (ux * (5.0 * uz ** 2 - 1.0))
    a_j2y = factor * (uy * (5.0 * uz ** 2 - 1.0))
    a_j2z = factor * (uz * (5.0 * uz ** 2 - 3.0))

    return np.array([a_cx + a_j2x, a_cy + a_j2y, a_cz + a_j2z])

def ode_full(state, t):
    """
    Defines the ODE system for orbit propagation.    """    a =
gravity_full_j2(state)
    return [state[3], state[4], state[5], a[0], a[1], a[2]]

# ------------------------------------------------------
# Step 1: Propagate from t0 to t1 with the full model
# ------------------------------------------------------
# Define the simulation time span
t0 = 0.0                      # Start time [s]
t1 = 100                     # End time [s] (adjust as needed for sufficient
orbit variation)
num_points = 501              # Number of points for ODE integration and
interpolation
t_span = np.linspace(t0, t1, num_points)
```

```
print(f"Simulating orbit from {t0} s to {t1} s with {num_points} time
points.\n")

# Propagate the orbit using ODE integration
full_solution = odeint(ode_full, state0, t_span)
final_state_full_model = full_solution[-1]

# Create interpolation functions for state variables using cubic splines
state_interpolators = [
    interp1d(t_span, full_solution[:, i], kind='cubic',
fill_value="extrapolate")
    for i in range(6)
]

def get_state(t):
    """Retrieve the state vector at time t using interpolation."""
    return np.array([f(t) for f in state_interpolators])

# ------------------------------------------------------
# Step 2: Measure gravity at t0, t_mid, and t1
# ------------------------------------------------------
t_mid = (t0 + t1) / 2.0
state_mid = get_state(t_mid)
a0 = gravity_full_j2(state0)                # Gravity at t0 [m/s^2]
a_mid = gravity_full_j2(state_mid)          # Gravity at t_mid [m/s^2]
a1 = gravity_full_j2(final_state_full_model)  # Gravity at t1 [m/s^2]

print(f"Gravitational Acceleration Measurements:")
print(f"a(t0) = {a0} m/s²")
print(f"a(t_mid) = {a_mid} m/s²")
print(f"a(t1) = {a1} m/s²\n")

# ------------------------------------------------------
# Step 3: Quadratic gravity model function based on three points
# ------------------------------------------------------
# Fit a quadratic polynomial for each acceleration component
# using t0, t_mid, and t1

# Define the time points for fitting
times_fit = np.array([t0, t_mid, t1])

# Acceleration components for fitting
a_components = {
    'x': np.array([a0[0], a_mid[0], a1[0]]),
    'y': np.array([a0[1], a_mid[1], a1[1]]),
```

```
    'z': np.array([a0[2], a_mid[2], a1[2]])
}

# Fit quadratic polynomials for each component
coeffs_quadratic = {}
for comp in ['x', 'y', 'z']:
    # Fit a second-order polynomial (quadratic)
    coeffs = np.polyfit(times_fit, a_components[comp], 2)
    coeffs_quadratic[comp] = coeffs  # [A, B, C] for Ax^2 + Bx + C

def quadratic_gravity(t):
    """
    Estimates gravitational acceleration using a quadratic fit.    Output:
Estimated acceleration [m/s^2].    """    a_est = np.array([
        np.polyval(coeffs_quadratic['x'], t),
        np.polyval(coeffs_quadratic['y'], t),
        np.polyval(coeffs_quadratic['z'], t)
    ])
    return a_est

# ----------------------------------------------------------
# Step 4: Define the error magnitude function for quadratic model
# ----------------------------------------------------------
def error_magnitude_quadratic(t):
    """
    Computes the magnitude of the difference between actual and quadratic-
estimated gravity at time t.    """    current_state = get_state(t)
    a_actual = gravity_full_j2(current_state)
    a_est = quadratic_gravity(t)
    error_vec = a_actual - a_est
    return np.linalg.norm(error_vec)

# ----------------------------------------------------------
# Step 5: Integrate the error over [t0, t1] using calculus
# ----------------------------------------------------------
# Quadratic Model Errors
print("Integrating errors for the quadratic interpolation model...\n")
sum_error_quad, _ = quad(
    error_magnitude_quadratic, t0, t1, limit=100, epsabs=1e-9, epsrel=1e-9
)
average_error_quad = sum_error_quad / (t1 - t0)

# Find maximum error using fine sampling
num_max_points = 1001
t_max_span = np.linspace(t0, t1, num_max_points)
max_error_quad = max(error_magnitude_quadratic(t) for t in t_max_span)
```

```python
# ------------------------------------------------------
# Step 6: Results
# ------------------------------------------------------
print("---- Quadratic Interpolation Results ----")
print(f"Sum of Errors over Interval [m/s²·s]: {sum_error_quad:.6f}")
print(f"Average Error [m/s²]:                 {average_error_quad:.6f}")
print(f"Max Error [m/s²]:                     {max_error_quad:.6f}\n")

# ------------------------------------------------------
# Step 7: Visualization
# ------------------------------------------------------
# Compute errors at discrete points for plotting
print("Generating plots...\n")
errors_quad = [error_magnitude_quadratic(t) for t in t_max_span]

# Plot Error Magnitude Over Time
plt.figure(figsize=(12, 6))
plt.plot(t_max_span, errors_quad, label='Quadratic Interpolation Error',
color='blue')
plt.xlabel('Time [s]')
plt.ylabel('Error Magnitude [m/s²]')
plt.title('Gravitational Acceleration Error Over Time (Quadratic
Interpolation)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Optional: Compare Estimated vs. Actual Acceleration Components
# Choose a subset of times to plot for clarity
num_plot_points = 200
indices_plot = np.linspace(0, num_max_points - 1, num_plot_points, dtype=int)
times_plot = t_max_span[indices_plot]

a_actual_plot = np.array([gravity_full_j2(get_state(t)) for t in times_plot])
a_quadratic_plot = np.array([quadratic_gravity(t) for t in times_plot])

# Plot each acceleration component
components = ['x', 'y', 'z']
for i, comp in enumerate(components):
    plt.figure(figsize=(12, 6))
    plt.plot(times_plot, a_actual_plot[:, i], label='Actual Acceleration',
color='black')
    plt.plot(times_plot, a_quadratic_plot[:, i], label='Quadratic Estimation',
linestyle='--', color='blue')
```

```
    plt.xlabel('Time [s]')
    plt.ylabel(f'Acceleration Component {comp.upper()} [m/s²]')
    plt.title(f'Acceleration Component {comp.upper()} Over Time')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

# ----------------------------------------------------
# Entry Point
# ----------------------------------------------------
if __name__ == "__main__":

    """
    Defined Elliptical Orbit:
    Perigee Radius (m): 6778137.0
    Apogee Radius (m): 7178137.0
    Semi-Major Axis (m): 6978137.0
    Eccentricity: 0.0287
    Calculated Initial Velocity at Perigee (m/s): 7777.68
        Simulating orbit from 0.0 s to 100 s with 501 time points.

        Gravitational Acceleration Measurements:
        a(t0) = [-8.68842639 -0.         -0.        ] m/s²
        a(t_mid) = [-8.67337494 -0.498152   -0.        ] m/s²
        a(t1) = [-8.62829505 -0.99431868 -0.        ] m/s²

        Integrating errors for the quadratic interpolation model...
        ---- Quadratic Interpolation Results ----
        Sum of Errors over Interval [m/s²·s]: 0.008279
        Average Error [m/s²]:              0.000083
        Max Error [m/s²]:                  0.000127
    """

    pass
```

Demonstration of error in circular orbit

```
import numpy as np
from scipy.integrate import odeint, quad
from scipy.interpolate import interp1d
import matplotlib.pyplot as plt

# ----------------------------------------------------
# This script:
```

```
# 1. Defines a high-fidelity gravity model (with J2).
# 2. Sets up an initial orbit and simulates from t0 to t1.
# 3. Measures gravity at t0, t_mid, and t1.
# 4. Uses quadratic interpolation of gravity based on three points as a model.
# 5. Uses calculus-based integration to compute aggregate and average errors.
# 6. Prints the aggregate error and average error in m/s².
# 7. Visualizes the errors and acceleration components.
# ------------------------------------------------------

# ------------------------------------------------------
# Constants and Initial Conditions
# ------------------------------------------------------
GM_EARTH = 3.986004418e14  # Earth's gravitational parameter [m^3/s^2]
R_EARTH = 6378137.0        # Earth's mean radius [m]
J2 = 1.08262668e-3          # Earth's J2 term

t0 = 0.0
t1 = 100.0                  # seconds over which we will model gravity

# Define a nominal LEO orbit ~400 km above Earth's surface
altitude = 400e3
r0 = R_EARTH + altitude
v_circ = np.sqrt(GM_EARTH / r0)
# Initial state: circular orbit in the equatorial plane
# state = [x, y, z, vx, vy, vz]
state0 = np.array([r0, 0.0, 0.0, 0.0, v_circ, 0.0])

# ------------------------------------------------------
# Gravity Model with J2
# ------------------------------------------------------
def gravity_full_j2(state):
    """
    Compute gravitational acceleration with J2 perturbation.    Output:
Acceleration vector [m/s^2].    """    x, y, z = state[0], state[1], state[2]
    r = np.sqrt(x ** 2 + y ** 2 + z ** 2)

    if r == 0:
        raise ValueError("Division by zero encountered in gravity
calculation.")

    ux, uy, uz = x / r, y / r, z / r

    # Central gravity
    a_cx = -GM_EARTH * ux / (r ** 2)
    a_cy = -GM_EARTH * uy / (r ** 2)
    a_cz = -GM_EARTH * uz / (r ** 2)
```

```
    # J2 perturbation
    factor = (3.0 / 2.0) * J2 * (GM_EARTH / (r ** 2)) * ((R_EARTH ** 2) / (r
** 2))
    a_j2x = factor * (ux * (5.0 * uz ** 2 - 1.0))
    a_j2y = factor * (uy * (5.0 * uz ** 2 - 1.0))
    a_j2z = factor * (uz * (5.0 * uz ** 2 - 3.0))

    return np.array([a_cx + a_j2x, a_cy + a_j2y, a_cz + a_j2z])

def ode_full(state, t):
    """
    Defines the ODE system for orbit propagation.    """    a =
gravity_full_j2(state)
    return [state[3], state[4], state[5], a[0], a[1], a[2]]

# ------------------------------------------------------
# Step 1: Propagate from t0 to t1 with the full model
# ------------------------------------------------------
num_points = 501  # Number of points for ODE integration and interpolation
t_span = np.linspace(t0, t1, num_points)
full_solution = odeint(ode_full, state0, t_span)
final_state_full_model = full_solution[-1]

# Create interpolation functions for state variables
state_interpolators = [
    interp1d(t_span, full_solution[:, i], kind='cubic',
fill_value="extrapolate")
    for i in range(6)
]

def get_state(t):
    """Retrieve the state vector at time t using interpolation."""
    return np.array([f(t) for f in state_interpolators])

# ------------------------------------------------------
# Step 2: Measure gravity at t0, t_mid, and t1
# ------------------------------------------------------
t_mid = (t0 + t1) / 2.0
state_mid = get_state(t_mid)
a0 = gravity_full_j2(state0)          # Gravity at t0 [m/s^2]
a_mid = gravity_full_j2(state_mid)    # Gravity at t_mid [m/s^2]
a1 = gravity_full_j2(final_state_full_model)  # Gravity at t1 [m/s^2]

# ------------------------------------------------------
# Step 3: Quadratic gravity model function based on three points
```

```
# ------------------------------------------------------
# Fit a quadratic polynomial for each acceleration component
# using t0, t_mid, and t1

# Define the time points
times_fit = np.array([t0, t_mid, t1])

# Acceleration components for fitting
a_components = {
    'x': np.array([a0[0], a_mid[0], a1[0]]),
    'y': np.array([a0[1], a_mid[1], a1[1]]),
    'z': np.array([a0[2], a_mid[2], a1[2]])
}

# Fit quadratic polynomials for each component
coeffs_quadratic = {}
for comp in ['x', 'y', 'z']:
    # Fit a second-order polynomial (quadratic)
    coeffs = np.polyfit(times_fit, a_components[comp], 2)
    coeffs_quadratic[comp] = coeffs  # [A, B, C] for Ax^2 + Bx + C

def quadratic_gravity(t):
    """
    Estimates gravitational acceleration using a quadratic fit.    Output:
Estimated acceleration [m/s^2].    """    a_est = np.array([
        np.polyval(coeffs_quadratic['x'], t),
        np.polyval(coeffs_quadratic['y'], t),
        np.polyval(coeffs_quadratic['z'], t)
    ])
    return a_est

# ------------------------------------------------------
# Step 4: Define the error magnitude function for quadratic model
# ------------------------------------------------------
def error_magnitude_quadratic(t):
    """
    Computes the magnitude of the difference between actual and quadratic-
estimated gravity at time t.    """    current_state = get_state(t)
    a_actual = gravity_full_j2(current_state)
    a_est = quadratic_gravity(t)
    error_vec = a_actual - a_est
    return np.linalg.norm(error_vec)

# ------------------------------------------------------
# Step 5: Integrate the error over [t0, t1] using calculus
# ------------------------------------------------------
```

```python
# Quadratic Model Errors
sum_error_quad, _ = quad(
    error_magnitude_quadratic, t0, t1, limit=100, epsabs=1e-9, epsrel=1e-9
)
average_error_quad = sum_error_quad / (t1 - t0)

# Find maximum error using fine sampling
num_max_points = 1001
t_max_span = np.linspace(t0, t1, num_max_points)
max_error_quad = max(error_magnitude_quadratic(t) for t in t_max_span)

# ------------------------------------------------------
# Step 6: Results
# ------------------------------------------------------
print("---- Quadratic Interpolation ----")
print(f"Sum of Errors over Interval [m/s²·s]: {sum_error_quad:.6f}")
print(f"Average Error [m/s²]:                 {average_error_quad:.6f}")
print(f"Max Error [m/s²]:                     {max_error_quad:.6f}\n")

# ------------------------------------------------------
# Step 7: Visualization
# ------------------------------------------------------
# Compute errors at discrete points for plotting
errors_quad = [error_magnitude_quadratic(t) for t in t_max_span]

plt.figure(figsize=(12, 6))
plt.plot(t_max_span, errors_quad, label='Quadratic Interpolation Error',
color='blue')
plt.xlabel('Time [s]')
plt.ylabel('Error Magnitude [m/s²]')
plt.title('Gravitational Acceleration Error Over Time (Quadratic
Interpolation)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Optional: Compare Estimated vs. Actual Acceleration
# Choose a subset of times to plot
num_plot_points = 200
indices_plot = np.linspace(0, num_max_points - 1, num_plot_points, dtype=int)
times_plot = t_max_span[indices_plot]

a_actual_plot = np.array([gravity_full_j2(get_state(t)) for t in times_plot])
a_quadratic_plot = np.array([quadratic_gravity(t) for t in times_plot])
```

```
# Plot each component
components = ['x', 'y', 'z']
for i, comp in enumerate(components):
    plt.figure(figsize=(12, 6))
    plt.plot(times_plot, a_actual_plot[:, i], label='Actual Acceleration',
color='black')
    plt.plot(times_plot, a_quadratic_plot[:, i], label='Quadratic Estimation',
linestyle='--', color='blue')
    plt.xlabel('Time [s]')
    plt.ylabel(f'Acceleration Component {comp.upper()} [m/s²]')
    plt.title(f'Acceleration Component {comp.upper()} Over Time')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

# ------------------------------------------------------
# Entry Point
# ------------------------------------------------------
if __name__ == "__main__":
    """
    ---- Quadratic Interpolation ----
    Sum of Errors over Interval [m/s²·s]: 0.006477
    Average Error [m/s²]:              0.000065
    Max Error [m/s²]:                  0.000100
    """
    pass
```