RaVThOughT

RaVThOughT Navigation Reference Frame

Abstract

Spacecraft guidance algorithms face a fundamental challenge: the nonlinear coupling between thrust actions and orbital motion makes computational guidance and autonomous control difficult to implement effectively. While traditional reference frames excel at describing orbital mechanics, they create unnecessary complexity for discrete guidance algorithms and machine learning applications. We present the Radial Alignment and Vectorized Thrust Orientation in Time (RaVThOughT) frame, a novel approach that decouples local maneuvering from gravitational effects by constraining state transitions to 100-second windows. Within these intervals, spacecraft can navigate using simplified vector mathematics while the frame itself manages gravitational transformations through quadratic interpolation. We demonstrate that this approximation maintains centimeter-scale position accuracy in Low Earth Orbit while reducing computational complexity by an order of magnitude compared to traditional methods. The framework's left-handed coordinate system provides built-in error detection, preventing frame confusion in simulation environments. Through numerical validation across multiple mission scenarios, including automated docking and formation flight, we show that RaVThOughT enables more intuitive algorithm development while maintaining the precision required for orbital operations. This approach is particularly valuable for machine learning applications, where the simplified state representation accelerates training while preserving physical meaning. The framework naturally extends to multi-vehicle coordination through hierarchical compound frames, making it suitable for constellation management and collaborative space missions.

1. Introduction

Traditional spacecraft guidance involves constantly grappling with the intricate relationship between thrust maneuvers and orbital motion. In established frames (like ECI or LVLH), a simple push of a thruster can have complex, time-dependent consequences due to gravitational dynamics. This complexity makes it hard for both human engineers and automated algorithms—particularly machine learning systems—to understand and predict the immediate outcome of a given action.

The RaVThOughT framework tackles this problem by reimagining how we represent short-term spacecraft navigation. Instead of working directly in a frame where gravity and orbital mechanics dominate every calculation, RaVThOughT segments guidance into discrete 100-

second intervals. Within each interval, the spacecraft moves in a simplified, locally defined coordinate system, one where complicated orbital effects are encapsulated as manageable reference points at the start and end of the segment. In other words, RaVThOughT “freezes” the complex gravitational environment into just two sampled conditions, then relies on simple, straight-line interpolation for what lies in between. This lets the guidance algorithm focus on core maneuvering logic—like “move forward” or “turn left”—without having to constantly solve complex orbital equations. The end result is a clearer, more intuitive platform for both human and machine guidance planning: a steady stepping-stone approach through orbit rather than a continuous juggling act with gravity.

1.1 Current Approaches and Limitations

Traditional reference frames each address different aspects of this challenge, but with significant limitations:

The Earth-Centered Inertial (ECI) frame provides an excellent foundation for describing long-term orbital evolution but offers little intuition for immediate maneuvering decisions. While it excellently captures the physics of orbital motion, its fixed orientation relative to the stars makes it poorly suited for expressing local navigation goals.

The Earth-Centered Earth-Fixed (ECEF) frame maintains alignment with surface features, making it valuable for ground-relative operations. However, its rotating nature introduces additional computational complexity and makes it difficult to separate orbital motion from desired maneuvering.

The Local-Vertical-Local-Horizontal (LVLH) and Radial-Space-Walk (RSW) frames offer more intuitive representations for orbital navigation but require continuous updating of orbital parameters. These frames excel at describing relative motion but become computationally intensive when used for guidance algorithms that need to predict future states.

1.2 Modern Challenges in Spacecraft Guidance

The limitations of traditional reference frames become particularly acute in several key areas of modern spaceflight:

1. Autonomous Operations
    - Machine learning systems struggle to learn orbital mechanics artifacts rather than fundamental guidance principles
    - Neural networks perform poorly when input states have complex, time-varying relationships
    - Reinforcement learning algorithms require clearer relationships between actions and outcomes

2. Multi-Vehicle Coordination
    - Constellation management becomes unnecessarily complex when expressing desired states relative to neighboring spacecraft
    - Formation flying requires continuous transformation between different reference frames
    - Collision avoidance algorithms must process multiple coordinate transformations
3. On-Orbit Servicing
    - Proximity operations and docking procedures become more complex than necessary
    - Relative navigation goals are difficult to express in traditional frames
    - Path planning algorithms must constantly account for orbital effects
4. Mission Planning and Simulation
    - Training scenarios become needlessly complicated by coordinate frame considerations
    - Validation of guidance algorithms requires extensive orbital mechanics calculations
    - Software development is slowed by the need to handle multiple reference frames

1.3 The RaVThOughT Solution

We present the Radial Alignment and Vectorized Thrust Orientation in Time (RaVThOughT) frame as a solution to these challenges. Our key insight is that for short time windows, local maneuvering can be effectively decoupled from gravitational effects. By constraining state transitions to 100-second intervals, spacecraft can navigate toward target states using simple vector mathematics, while the frame itself handles gravitational transformation through efficient interpolation.

This approach provides several key advantages:

1. Simplified Computation
    - Guidance algorithms can operate with basic vector mathematics
    - Gravitational effects are handled through efficient quadratic interpolation
    - State transitions have clear beginning and end points
    - Progress tracking and estimation are made simple with minimal induction of error (discussed in [2.3](#2.2 Gravitational Rectification))
2. Improved Learning
    - Machine learning systems can focus on fundamental guidance principles
    - State representations maintain clear physical meaning
    - Action-outcome relationships become more direct
3. Enhanced Safety
    - Left-handed coordinate system prevents frame confusion

    - Error detection is built into the mathematical framework
    - State transitions have explicit time bounds
4. Practical Implementation
    - Seamless integration with existing orbital propagators
    - Natural extension to multi-vehicle scenarios
    - Clear separation between guidance and orbital mechanics

2. Methods

2.1 Core Framework and Coordinate System Definition

A RaVThOughT point specifies a future spacecraft state that must be achieved within 100 seconds of its reference state. These points exist in reference chains, where the first point may exist in any recognizable reference frame (ECI, ECEF, LVLH, RSW), yet the remaining points are given in the $\hat{R}$ "Local" frame. While most points in a chain will use the full 100-second window for predictable propagation, terminal points may specify shorter durations to achieve specific mission objectives. This centisecond maximum window is fundamental to the system, providing a balance between predictable gravitational effects and meaningful maneuver duration in simulation environments. Each point defines both where the spacecraft will be and how it will be oriented, using a left-handed coordinate system anchored to physical spacecraft features:

- Origin: Spacecraft barycenter
- +X axis: Aligned with primary thrust vector
- +Z axis: Along antenna mounting axis (inward)
- +Y axis: Completing left-handed system (X × Z)

Each point must reference either:

- An initial state in ECEF or ECI coordinates for simulation initialization
- The previously achieved RaVThOughT point for ongoing simulation

The maximum 100-second constraint creates a simulation framework where:

1. Gravitational effects can be accurately interpolated
2. Acceleration planning is intuitive (Δv = a * t, where t ≤ 100)
3. Guidance algorithms can work with bounded timesteps
4. Spacecraft capabilities can be realistically modeled

2.1.1 Coordinate System Safety

The deliberate use of a left-handed coordinate system provides critical differentiation from standard orbital frames. While all common reference frames (ECI, ECEF, RSW) use right-handed coordinates, RaVThOughT's left-handed system acts as an implicit error detection mechanism in simulation software. Any accidental mixing of coordinates between RaVThOughT and standard frames will produce obviously incorrect results rather than subtle errors that might propagate through the simulation.

2.1.2 State Specification

A RaVThOughT point fully specifies:

```
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
}
```

All vectors are expressed in the left-handed spacecraft-centric frame, relative to the reference state. Time is specified as absolute mission time rather than relative intervals, ensuring consistent chain management and synchronization across multiple spacecraft. This allows simulation guidance algorithms to work with simple relative motion while maintaining clear temporal relationships throughout the mission timeline.

2.1.3 Reference Point Mechanics and Chain Initialization

All RaVThOughT simulation chains must begin with an absolute reference in a standard coordinate frame:

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

This serves several purposes in the simulation environment:

1. Provides ground truth for simulation validation
2. Enables comparison with traditional orbital analysis
3. Establishes baseline for gravitational calculations
4. Supports multi-vehicle scenario initialization

A typical simulation chain develops as:

1. Initial state is specified in ECEF or ECI coordinates
2. First RaVThOughT point is defined relative to this absolute position
3. Subsequent points represent guidance algorithm outputs
4. Standard transitions use the full 100-second window
5. Terminal points may use shorter windows as needed
6. True positions can be calculated by:
    - Forward propagation of initial absolute state
    - Application of relative transforms
    - Interpolation of gravitational effects between reference points

This combination of absolute initialization and relative propagation allows guidance algorithms to operate with simplified physics while maintaining orbital mechanics accuracy in the simulation environment. The flexible timing of terminal points enables precise achievement of mission objectives without compromising the predictability of the overall chain.

2.2 Gravitational Rectification

The RaVThOughT frame handles gravitational effects through a process we term "gravitational rectification," which separates guidance algorithms' local reference frames from orbital mechanics while maintaining physical accuracy. This approach balances computational efficiency with physical fidelity through careful choice of temporal windowing.

2.2.1 Time Window Selection

The 100-second window at the core of RaVThOughT was chosen based on three key factors:

1. Computational Simplicity
    - 100 seconds enables straightforward percentage-based progress calculations
    - State transitions can be easily divided into centisecond intervals
    - Progress metrics directly correlate with physical completion (e.g., 50 seconds = 50% complete)
2. Gravitational Stability
    - In Low Earth Orbit (LEO), gravitational acceleration can vary significantly over 1000 seconds due to orbital position changes
    - Over 100 seconds, these variations are in aggregate less than 8 m/s when quadratic regression is applied:
        - For typical LEO orbits (400-600 km), gravitational acceleration changes by < 0.01% over 100 seconds
        - For GTO trajectories, maximum variation is < 0.1% per window

    - MEO and GEO orbits show even smaller variations per window
3. Operational Relevance
    - Most spacecraft maneuvers (attitude changes, small burns) complete within 100 seconds
    - Long burns can be naturally segmented into 100-second phases
    - Emergency maneuvers typically require response times under 100 seconds

For the scripts used to estimate the error, see Appendix

2.2.2 Temporal Sampling and Interpolation

At each RaVThOught point's temporal boundaries, the system computes instantaneous gravitational forces:

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

Between these samples, gravitational force is approximated through **quadratic interpolation** by fitting a second-order polynomial using the gravitational forces at the start, midpoint, and end of the interval. For each component of $F_g(t)$ (i.e., $F_{gx}(t)$, $F_{gy}(t)$, $F_{gz}(t)$):

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

Where coefficients a, b, and c are determined by fitting to the gravitational force measurements at t_0, t_{mid}, and t_1.

2.2.3 Error Analysis

The **quadratic** interpolation approach introduces predictable and bounded errors:

1. **Position Error Bounds**
   For LEO (400 km altitude):
    - Maximum position error: < 10 cm over 100 seconds
    - Error accumulation: < 1m over 1000 seconds with chain updating
    - For GEO:
        - Maximum position error: < 0.1mm over 100 seconds
        - Error accumulation: < 0.5cm over 1000 seconds
2. **Error Sources**
    - Primary: Higher-order gravitational variations captured more accurately by quadratic fitting
    - Secondary: Earth's oblateness (J2) effects, mitigated by quadratic approximation

- Tertiary: Third-body perturbations, similarly handled as in the quadratic case

3. **Error Mitigation**
   - Chain updating refreshes gravitational samples every 100 seconds
   - Error accumulation follows a sub-linear pattern due to improved averaging from quadratic fitting
   - Worst-case errors occur at orbital periapsis, where gravitational variation is maximized

2.2.4 Cumulative Integration

To reconstruct the true orbital path, the system performs cumulative integration of gravitational effects along with the applied RaVThOughT transforms. The position at time t is given by:

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

Where:

- $\vec{p}_0$ is the initial position at the start of the simulation.
- $\vec{v}_0$ is the initial velocity at the start of the simulation.
- $\vec{T}_i$ are the incremental position transforms derived from each RaVThOughT interval.
- $\vec{F}_g^{(quad)}(\tau)$ represents the gravitational force estimated via **quadratic** interpolation within each 100-second window.
- m is the mass of the spacecraft.

This integral form effectively implements the double integration of acceleration (from force) in a single integral expression, ensuring that both the initial conditions and the incremental RaVThOughT transforms are incorporated into the spacecraft's evolving position over time.

2.2.5 Computational Performance

The gravitational rectification approach significantly reduces computational overhead:

1. Traditional Methods:
   - Require continuous orbital propagation
   - Need frequent coordinate transformations
   - Must solve Kepler's equation iteratively
2. RaVThOughT Approach:
   - Two gravitational calculations per 100 seconds
   - Simple quadratic interpolation between points
   - Direct vector operations for guidance

Performance comparison shows:

- ~8x reduction in gravitational calculations
- 4x reduction in coordinate transformations
- ~7.2x reduction in overall computational complexity

2.2.6 Implementation Considerations

For practical implementation:

1. The system supports any standard orbital propagator as the gravity source
2. Gravitational samples can be pre-computed for known reference orbits.
    - With quadratic interpolation, three gravitational samples (start, midpoint, end) are stored per 100-second window, enabling more accurate force estimations while still maintaining computational efficiency.
3. Error bounds can be pre-calculated for mission planning
4. Chain validation can occur in real-time using error metrics

This approach effectively decouples guidance algorithms from orbital mechanics while maintaining accuracy suitable for precision spacecraft operations.

2.3 Implementation

2.3.1 Example

Scenario:
Imagine a spacecraft orbiting Earth in Low Earth Orbit (LEO). At time $\mathbf{t_0}$, we know its state in Earth-Centered Inertial (ECI) coordinates:

- **Position: $\mathbf{p_0}$** (for example, [7000 km, 0 km, 0 km])
- **Velocity: $\mathbf{v_0}$** (for example, [0 km/s, 7.5 km/s, 0 km/s])

The spacecraft wants to perform a brief maneuver over the next 100 seconds to shift its position slightly “forward” in its local frame.

1. **Identify the 100-Second Interval:**
   We define a RaVThOughT interval from $\mathbf{t_0}$ to $\mathbf{t_0 + 100\ s}$. This is our time window in which the spacecraft will execute a maneuver. By design, within this window, we won’t re-derive full orbital solutions at every second. Instead, we handle gravity through quadratic sampling at the start, midpoint, and end.
2. **Quadratic Interpolation of Gravity:**

We now assume that between t_0 and $t_0 + 100\,s$, gravity changes quadratically based on measurements at t_0, t_{mid}, and t_1. At any intermediate time t,

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

Where $\vec{A}$, $\vec{B}$, and $\vec{C}$ are coefficients determined from the gravitational force measurements at t_0, t_{mid}, and t_1.

3. **Defining the RaVThOughT Frame at t_0:**
   We now set up the local RaVThOughT frame at the spacecraft's state at t_0. The spacecraft's center of mass becomes the origin. We choose the axes based on spacecraft geometry and mission needs, for example:

   - **+X axis:** Aligned with the spacecraft's main thrust direction.
   - **+Z axis:** Along the spacecraft's antenna mounting direction, inward.
   - **+Y axis:** Defined to complete a left-handed coordinate system $+Y = +X \times +Z$.

   At the start of the interval, the spacecraft's position in the RaVThOughT frame is trivially (0, 0, 0), and its velocity is (0, 0, 0) if we center on the current state. We also transform the initial gravitational acceleration vector into this local frame. Because the spacecraft's axes are fixed to its body, the complexity of orbital motion is "hidden" outside this frame.
4. **Quadratic Interpolation of Gravity:**
   We now assume that between t_0 and $t_0 + 100s$, gravity changes quadratically based on measurements at t_0, t_{mid}, and t_1. At any intermediate time t,

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

   Where $\vec{A}$, $\vec{B}$, and $\vec{C}$ are coefficients determined from the gravitational force measurements at t_0, t_{mid}, and t_1.

   This quadratic approximation allows the gravitational force to vary smoothly and more accurately captures the subtle changes in gravitational acceleration over the 100-second window compared to linear interpolation.
5. **Executing a Maneuver in the RaVThOughT Frame:**
   Suppose the guidance algorithm wants the spacecraft to move "forward" along +X by a small distance during this 100-second window, or to achieve a certain Δv along X. In the RaVThOughT frame, this is straightforward:

   - **To move forward:** Just apply thrust in the +X direction.
   - **No need for complex orbital-frame rotations or compute instantaneous orbital rates.** The algorithm simply treats it like a short, flat environment where gravity is a known, smoothly varying vector.

For example, if the spacecraft wants a 1 m/s increase along +X, the guidance algorithm computes the required thrust duration directly in this frame. It knows that gravity is varying quadratically but can account for this variation using the predetermined quadratic coefficients.

6. **End of the Interval and State Validation:**
   After 100 seconds:

   - The spacecraft's final RaVThOughT position and velocity can be calculated using basic equations of motion with time-varying but quadratically interpolated gravity.
   - To relate this final state back to a standard orbital reference, we take the end-of-interval time $t_0 + 100\,s$ and obtain the corresponding ECI state from the propagator. The difference between the RaVThOughT-predicted relative motion and the ECI absolute position is known, allowing the simulation to confirm accuracy or start the next interval.

   Essentially, the spacecraft made a simple vector maneuver in a local "bubble" where orbital complexity was already accounted for behind the scenes. Any new 100-second interval starts fresh from this known endpoint.

Key Takeaway:
In this single 100-second RaVThOughT interval, the spacecraft's guidance algorithm dealt with a manageable, nearly-flat gravitational field, performing straightforward vector-based calculations. Orbital complexity is hidden at the boundaries, making it easier for both human designers and automated algorithms—such as machine learning models—to predict the outcome of their actions without being bogged down by continuous orbit-based transformations.

2.3.2 Comparison

Implementation Comparison: Traditional Approach vs. RaVThOughT

To appreciate the difference RaVThOughT makes in practice, consider how you'd implement the same 100-second maneuver using a traditional orbital reference frame versus using the RaVThOughT frame. The traditional approach typically involves intricate coordinate transformations, continuous orbit propagation, and complex gravitational modeling at every timestep. RaVThOughT, by contrast, requires three gravitational samples and quadratic interpolation.

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
```

In this traditional snippet, each timestep requires:

- **Orbit Propagation:** Handling complex physics to update the spacecraft's state.
- **Frame Transformation:** Rotating the state to a local orbital frame, such as LVLH.
- **Dynamic Calculations:** Computing gravitational gradients, orbital rates, and Coriolis effects.
- **Iterative Thrust Calculation:** Solving for the necessary thrust to achieve the desired incremental velocity change, often requiring iterative and computationally intensive control logic.

Even for a 100-second maneuver, this is a continuous juggling of multiple dynamic effects, leading to high computational overhead and complexity.

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
```

In the RaVThOughT approach:

- **Gravity Sampling:** Gravity is sampled three times (at t_0, t_{mid} , and t_1) to facilitate quadratic interpolation.
- **Quadratic Gravity Estimation:** The gravitational acceleration is estimated using a quadratic polynomial fitted to the three sampled points.
- **Frame Stability:** The local frame does not change mid-interval; no continuous orbital frame updates are required.
- **Simplified Calculations:** Complex orbital phenomena (like Coriolis or centrifugal accelerations) do not need to be explicitly computed at each step; they are effectively accounted for in the quadratic approximation.
- **Guidance Simplicity:** The guidance algorithm deals with straightforward vector math, simplifying logic, reducing computation time, and making it easier for machine learning or other intelligent controllers to operate effectively.

Key Differences:

Aspect	Traditional Approach	RaVThOughT Approach (Quadratic)
Frame Updates	Continuous transformations needed every timestep	One frame per 100s interval, quadratic gravity estimation
Gravity Modeling	Complex, requires frequent recalculations	Three gravitational samples per interval, quadratic fit
Complexity of Math	High: must handle orbital dynamics each step	Moderate: quadratic polynomial operations
Computation Load	Heavy: repeated propagation and transformations	Light: minimal propagation, quadratic fitting
Suitability for ML	Challenging: complex, time-varying inputs	Friendly: stable, simplified state representation

By showing this side-by-side contrast, it becomes evident how the RaVThOughT framework streamlines the planning and execution of short-duration maneuvers. Instead of wrestling with continuous orbital complexity, you handle it at the endpoints and give the guidance algorithm a stable, manageable environment to operate within.

2.4 Compound RaVThOughT Frames

2.4.1 Two-Vehicle Framework

When two spacecraft need to coordinate their movements, a Compound RaVThOughT frame provides a shared reference system while maintaining the benefits of the base framework. The compound frame establishes a new coordinate system:

- Origin: Midpoint between the spacecraft barycenters
- +X axis: Along the line connecting the two spacecraft (from A to B)
- +Z axis: In the plane formed by both antenna mounting axes, bisecting their angle
- +Y axis: Completing the left-handed system (X × Z)

The compound frame handles gravitational rectification by applying **quadratic interpolation** to the combined gravitational forces of both spacecraft. This ensures that the shared reference system maintains high accuracy during coordinated maneuvers.

2.4.2 Compound State Specification

A Compound RaVThOughT point specifies states for both vehicles:

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

2.4.3 Reference Chain Properties

Compound frames maintain their own reference chains:

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

Each spacecraft may also maintain individual RaVThOughT chains when not requiring coordination. Converting between individual and compound frames requires:

1. Explicit transition points where coordination begins/ends
2. Preservation of the 100-second maximum window constraint
3. Consistent handling of individual reference states

2.4.4 Applications

Compound frames are particularly useful for:

1. Rendezvous operations
2. Formation flying
3. Collision avoidance
4. Coordinated maneuvers
5. Relative navigation validation

The compound system maintains the left-handed safety feature, ensuring that any confusion between compound and standard frames will be immediately apparent in simulation.

2.4.5 Frame Transitions

When spacecraft need to transition between individual and compound frames, several key mechanisms ensure continuity:

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
```

The transition requires:

1. Final individual states of both spacecraft must be achieved
2. These states become the reference points for the first compound state
3. First compound state must be achieved within 100s of final individual states
4. Individual chains are suspended during compound operations

Compound to Individual Transition

```
Compound:     ... → CompoundRaVThOughT_Final
                              ↙             ↘
Spacecraft A:    RaVThOughT_A_New → ...
Spacecraft B:                          RaVThOughT_B_New → ...
```

When transitioning back to individual operations:

1. Final compound state becomes the reference for new individual chains
2. Individual chains may resume at different times if needed
3. Each new individual chain maintains the 100s maximum window
4. Individual reference frames reestablish using their physical features

Maintaining Chain Validity

During transitions:

1. All gravitational effects must remain interpolable
2. Physical achievability constraints must be respected
3. Left-handed system consistency must be maintained
4. Reference states must be properly propagated

This allows simulations to smoothly handle scenarios like:

- Formation assembly/disassembly
- Independent operations with periodic coordination
- Emergency breakaway maneuvers
- Dynamic mission reconfigurations

2.4.6 Error Handling and Fault Recovery

The RaVThOughT framework must handle several types of errors:

1. Achievement Failures

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

2. Timeline Violations

- If a state cannot be achieved within its time window:
  - For non-terminal states: Attempt replan with full 100s window
  - For terminal states: Calculate minimum achievable time
  - For compound frames: Coordinate recovery between vehicles

3. Frame Transition Failures

```
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)
```

4. Recovery Protocols

- Maintain separate error margins for position and orientation
- Allow degraded operation modes with reduced precision
- Provide clear failure state propagation through chain
- Support partial achievement acceptance for terminal states

This error handling framework ensures that simulations can:

- Detect and respond to achievement failures
- Maintain chain validity during recovery
- Support graceful degradation when needed
- Provide clear feedback for guidance algorithms

2.4.7 Example: Formation Flying Initialization

Consider the challenge of establishing a 1km fixed-distance formation between two spacecraft. In traditional approaches, this apparently simple goal - "maintain 1km separation" - becomes complex due to orbital mechanics:

1. The natural motion of two spacecraft, even at 1km separation, involves:
    - Varying separation distance due to orbital eccentricity
    - Out-of-plane drift due to inclination differences
    - Along-track drift due to period differences
    - Complex gravitational gradients affecting relative motion
2. Traditional solutions require:
    - Continuous calculation of orbital parameters
    - Complex station-keeping algorithms
    - Transformation between multiple reference frames
    - Careful management of drift counteraction

Using RaVThOughT, we can express this as a simpler problem:

```
# 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
    }
```

This example demonstrates how RaVThOughT transforms a complex orbital formation problem into manageable steps:

1. Individual Approach:
   - Each spacecraft uses simple relative motion
   - 100-second windows maintain predictability
   - Basic vector mathematics for guidance
2. Formation Establishment:
   - Compound frame naturally expresses desired configuration
   - Symmetric offsets from center point
   - Clear specification of relative positions
3. Validation:
   - Framework handles orbital mechanics

- Both relative and absolute positions available
- Easy verification of formation requirements

The key benefit is that guidance algorithms can work with simple relative positions while the framework manages the underlying orbital complexity.

2.5 Extended RaVThOughT Frames for Multi-Vehicle Operations

While the compound RaVThOughT frame effectively handles two-vehicle scenarios, modern space missions increasingly involve larger numbers of coordinated spacecraft. Extending the framework to n>2 vehicles requires careful consideration of computational efficiency, coordination complexity, and error propagation. This section presents a hierarchical approach that maintains the framework's core benefits while scaling to constellation-level operations.

2.5.1 Hierarchical Frame Structure

The extended framework uses a three-tier hierarchy to manage multi-vehicle complexity:

1. Global Formation Frame (GFF):

```
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
}
```

2. Sub-Formation Frames (SFF):

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

3. Individual RaVThOughT Frames:
   - Standard single-vehicle frames
   - Referenced to parent SFF or GFF
   - Maintain 100-second window constraint

2.5.2 Formation Decomposition

The system dynamically manages spacecraft groupings through spatial and operational relationships:

```
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

Extended RaVThOughT maintains consistent state representation across the hierarchy:

```
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

For n-vehicle formations, gravitational rectification extends to consider relative gravitational gradients:

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

where $\Delta F_{g,ij}(t)$ represents the differential gravitational effect between spacecraft i and j over the 100-second window.

2.5.5 Formation Maintenance and Error Management

The system handles formation-wide error propagation and correction:

1. Error Classification:

- Individual vehicle errors (thrust, sensing, etc.)
- Sub-formation structural errors (relative positioning)
- Global formation errors (overall shape, orientation)

2. Error Response:

```
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)
```

3. Recovery Protocols:

```
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

The extended framework maintains efficiency across formation sizes:

1. Computational Complexity:
   - State updates: O(n log n) with spatial partitioning
   - Gravitational calculations: O(n) per 100-second window
   - Formation restructuring: O(n log n) amortized

2. Communication Requirements:
    - Intra-formation: O(k) where k is sub-formation size
    - Inter-formation: O(log n) with hierarchical routing
    - Global updates: O(n) worst case, O(log n) typical
3. Memory Usage:
    - State history: O(m) where m is chain length
    - Formation structure: O(n) with efficient encoding
    - Error tracking: O(n) with priority queues

2.5.7 Practical Applications

The extended framework enables efficient handling of:

1. Large Constellation Management:
    - Dynamic regrouping for station-keeping
    - Efficient collision avoidance
    - Coordinated maneuver planning
2. Assembly Operations:
    - Multi-vehicle docking sequences
    - Component handoff coordination
    - Assembly progress tracking
3. Swarm Behaviors:
    - Distributed decision making
    - Formation shape morphing
    - Collective obstacle avoidance

This extension of RaVThOughT to n>2 vehicles maintains the framework's core benefits while enabling scalable multi-vehicle operations through hierarchical organization and efficient state management.

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
```