1. Navigation algorithms applied:

Main steps

- Getting Gyroscope sensors readings
- Getting Accelerometer sensors readings
- Fusion of the readings using Unscented Kalman Filter

Steps:

1. The algorithm uses numpy and pykalman modules for numerical and filtering functions.
2. It defines a function fuse_acc_gyro that takes accelerometer and gyroscope readings and a time step as arguments.
3. It converts the gyroscope readings from degrees to radians.
4. It defines a state vector with six elements: three angles and three angular velocities.
5. It defines a state transition function f that updates the state vector by adding the angular velocities to the angles.
6. It defines an observation function h that calculates the acceleration components from the angles using trigonometry.
7. It initializes the state vector, the state covariance matrix, the measurement noise covariance matrix, and the process noise covariance matrix with small values.
8. It initializes an unscented Kalman filter with the state transition and observation functions and covariances, and the initial state mean and covariance.
9. It creates a list for storing the roll, pitch, and yaw angles.
10. It loops through the accelerometer and gyroscope readings and predicts and updates the state and state covariance using the unscented Kalman filter.
11. It appends the angles from the state vector to the list.
12. It converts the angles from radians to degrees and the list to an array.
13. It returns the array of angles as the output of the function.

```python
import numpy as np
from pykalman import UnscentedKalmanFilter

def fuse_acc_gyro(acc, gyro, dt):

  gyro = np.deg2rad(gyro)
# define the state transition function
  def f(x):
    # x is the state vector of shape (6,)
    # x[0:3] are the roll, pitch, and yaw angles in radians
    # x[3:6] are the angular velocities in radians/s
    x[0] += x[3] * dt
    x[1] += x[4] * dt
    x[2] += x[5] * dt
    return x

  # define the observation function
  def h(x):
    # calculate the acceleration components from the angles
    ax = -np.sin(x[1])
    ay = np.sin(x[0]) * np.cos(x[1])
    az = np.cos(x[0]) * np.cos(x[1])
    return np.array([ax, ay, az])

  x = np.zeros(6)
  # initialize the state covariance matrix with small values
  P = np.eye(6) * 0.01
  # initialize the measurement noise covariance matrix
  R = np.eye(3) * 0.01
  # initialize the process noise covariance matrix
  Q = np.eye(6) * 0.001
  # initialize the unscented Kalman filter
  ukf = UnscentedKalmanFilter(transition_functions=f, observation_functions=h,
transition_covariance=Q, observation_covariance=R, initial_state_mean=x,
initial_state_covariance=P)

  # for storing roll, pitch, and yaw angles
  rpy = []
  # loop through the accelerometer and gyroscope readings
  for i in range(len(acc)):
    # predict the next state
    x, P = ukf.predict(x, P)
    # update the state with the accelerometer measurement
    x, P = ukf.update(x, P, acc[i])
    # append the angles to the list
    rpy.append(x[0:3])

  # convert the angles to degrees
  rpy = np.rad2deg(rpy)

  # return the roll, pitch, and yaw angles as a numpy array
  rpy = np.array(rpy)
  return rpy
```

2. Guidance algorithms applied:

1. __init__: Initializes the mass, time step, constants, target point, tolerance, and gain schedule of the spacecraft.
2. set_initial_state: Sets the initial position, velocity, and flight path angle of the spacecraft.
3. rho: Calculates the atmospheric density at a given altitude using the U.S. Standard Atmosphere (Mars) model.
4. C_l: Calculates the lift coefficient based on Mach number and angle of attack using a simplified linear model.
5. C_d: Calculates the drag coefficient based on Mach number using a basic model.
6. T: Calculates the atmospheric temperature at a given altitude using the U.S. Standard Atmosphere (Mars) model.
7. A_ref: Returns the reference area of the spacecraft.
8. dynamics_model: Updates the state of the spacecraft based on the dynamics model using the Runge-Kutta method. It computes the forces of gravity, drag, and lift, and the accelerations and velocities in each direction.
9. predict_state: Predicts the state of the spacecraft at the target point using the current bank angle. It iterates the dynamics_model method until the altitude reaches the target value.
10. bank_angle_control: Calculates and adjusts the bank angle based on the predicted state and error correction. It uses a downrange control to minimize the range error and a heading control to align the spacecraft with the target point. It limits the bank angle to the maximum value.
11. bank_angle_control2: Calculates and adjusts the bank angle based on the predicted state and error correction. It uses a range-to-go control to modulate the vertical component of the lift-to-drag ratio and a roll control to adjust the bank angle. It limits the bank angle to the maximum value.
12. guide: Executes the guidance loop to steer the spacecraft towards the target point. It uses either bank_angle_control or bank_angle_control2 depending on the flight phase. It checks the landing success based on the tolerance.

The algorithm also defines a test_guidance function that tests the MarsEntryGuidance class with a given target point. It sets the mass and initial state of the spacecraft and calls the guide method.

```python
class MarsEntryGuidance:
    """
    This class represents a simplified Mars entry guidance system inspired by Apollo and MSL
approaches.
    """

    def __init__(self, mass, dt=0.1):
        self.mass = mass
        self.dt = dt

        # Constants
        self.G = 6.6743e-11  # Gravitational constant
        self.R_mars = 3389e3  # Mars radius
        self.max_bank_angle = np.pi / 6

        # Target point and tolerance
        self.target_x = 0
        self.target_y = 0
        self.target_z = -10
        self.tolerance = 0.1

        # Gain schedule (example)
        self.K_downrange = 0.1
        self.K_heading = 0.05

        # Initial state (placeholder)
        self.state = None

    def set_initial_state(self, x, y, z, v, gamma):
        """
        Sets the initial state of the spacecraft.
        """
        self.state = np.array([x, y, z, v, gamma])

    def rho(self, z):
        """
        Returns the atmospheric density at a given altitude using the U.S. Standard Atmosphere
(Mars) model.
        """
        # Replace with your preferred atmospheric model or interpolation method
        # Here's an example using the U.S. Standard Atmosphere (Mars) model:
        h = z + self.R_mars  # Convert altitude to geopotential altitude
        if h < 0:
            return 0.0  # No atmosphere below ground
        elif h <= 25000:
            # Use exponential profile for lower altitudes
            return 0.01225 * np.exp(-h / 6500)
        else:
            # Use constant density for higher altitudes
            return 0.001
```

Main guidance algorithm Apollo Modified:

```
    def dynamics_model(self, state, bank_angle):
        """
        Updates the state of the spacecraft based on the dynamics model.
        """
        x, y, z, v, gamma = state
        # Compute forces
        gravity = self.G * self.mass / (z + self.R_mars)**2
        mach = v / np.sqrt(gamma * 287 * self.T(z))  # Calculate Mach number based on
temperature
        alpha = bank_angle  # Assume angle of attack equals bank angle
        C_d = self.C_d(mach)  # Use calculated drag coefficient
        drag = -0.5 * C_d * self.A_ref() * self.rho(z) * v**2 / v
        lift = 0.5 * self.C_l(mach, alpha) * self.A_ref ()* self.rho(z) * v**2 *
np.cos(bank_angle)
        # Compute acceleration
        acc = np.array([0, 0, -gravity + drag + lift / self.mass])
        # Compute new state
        new_state = state + acc * self.dt
        new_state[4] = np.arctan2(new_state[3,2], new_state[3,0])  # Update flight path angle
        return new_state

    def predict_state(self, state, bank_angle):
        """
        Predicts the state of the spacecraft at the target point using the current bank angle.
        """
        predicted_state = state
        while predicted_state[2] < self.target_z:
            predicted_state = self.dynamics_model(predicted_state, bank_angle)
        return predicted_state

    def bank_angle_control(self, state, predicted_state):
        """
        Calculates and adjusts the bank angle based on the predicted state and error
correction.
        """
        # Downrange error
        error_downrange = predicted_state[0] - self.target_x
        # Heading error (implement based on your target)
        error_heading = 0.0  # placeholder
        # Adaptive gain based on flight phase (example)
        if state[2] > -50e3:
            self.K_downrange = 0.2
        else:
            self.K_downrange = 0.1
        # Update bank angle with downrange and heading control
        bank_angle = bank_angle - self.K_downrange * error_downrange - self.K_heading *
error_heading

        # Limit bank angle
        bank_angle = np.clip(bank_angle, -self.max_bank_angle, self.max_bank_angle)

        return bank_angle
```

```python
def guide(self):
        """
        Executes the guidance loop to steer the spacecraft towards the target point.
        """
        if self.state is None:
            raise ValueError("Initial state not set!")
        bank_angle = 0
        while self.state[2] < self.target_z:
            # Predict state at target
            predicted_state = self.predict_state(self.state, bank_angle)

            # Calculate and adjust bank angle
            bank_angle = self.bank_angle_control(self.state, predicted_state)
            print(f'{bank_angle}')
            # Apply bank angle and update state
            self.state = self.dynamics_model(self.state, bank_angle)
        # Check landing success
        if abs(self.state[0] - self.target_x) < self.tolerance and \
        abs(self.state[1] - self.target_y) < self.tolerance and \
        abs(self.state[2] - self.target_z) < self.tolerance:
            print("Successful landing!")
        else:
            print("Landing accuracy outside tolerance.")
```

3. Control algorithms applied:

- The algorithm defines a class called MarsLander that represents a simplified model of a lander with state and sensor data.
- The class has several methods that update the lander's state, orientation, and control signals based on the sensor readings and the desired orientation.
- The class also has a method called simulate_descent that runs a loop over a given time interval and simulates the lander's descent using the other methods.
- The main steps of the loop are:
    - Determine the desired orientation as a quaternion (a four-dimensional vector that represents rotations in three-dimensional space).
    - Calculate the bank angle (the angle between the lander's longitudinal axis and the vertical plane) using a PID controller (a feedback control system that adjusts the control signal based on the error, the integral of the error, and the derivative of the error).
    - Apply thruster forces based on the bank angle and update the acceleration accordingly.
    - Update the orientation based on the bank angle and the current orientation using quaternion multiplication.
    - Update the state (position and velocity) based on the acceleration and the current state using simple equations of motion.

- Print the state and the sensor readings for debugging purposes.

```python
import numpy as np
# Constants
mass = 1000  # kg
g_mars = 3.71  # m/s^2 (Martian gravity)
drag_coeff = 0.5
density = 0.02  # kg/m³
dt = 0.01  # (seconds) time step
t_max = 10  # (seconds) total simulation time

class MarsLander:
    """
    Class representing a Mars lander with simplified dynamics and state estimation.
    """
    def __init__(self):
        """
        Initializes the lander with state and sensor data.
        """
        # State (position, velocity, orientation, angular velocity)
        self.position = np.array([1000, 500, 200])  # m
        self.velocity = np.array([-100, -100, -100])  # m/s
        self.orientation = np.array([1, 0, 0, 0])  # Quaternion (initially upright)
        self.angular_velocity = np.zeros(3)  # rad/s
        # Sensor readings
        self.accelerometer = np.array([0, 0, -3.71])  # m/s^2 (Martian gravity)
        self.gyroscope = np.zeros(3)  # rad/s
        # Control parameters (replace with actual control design)
        self.kp = 1  # Proportional gain
        self.ki = 0.1  # Integral gain
        self.kd = 0.1  # Derivative gain
        # State estimation parameters (replace with appropriate algorithm)
        self.estimated_orientation = self.orientation.copy()
        self.prev_error = np.zeros(4)  # Initialize previous error as a quaternion

    def update_state(self, dt):
        """
        Updates the lander's state using simplified equations of motion.
        """
        # Assuming constant gravity and neglecting drag for simplicity
        acceleration = self.accelerometer  # m/s^2
        self.velocity += acceleration * dt
        self.position += self.velocity * dt
        # Assuming sensor readings directly reflect state
        self.gyroscope = self.angular_velocity

    def compute_bank_angle(self, desired_orientation):
        """
        Calculates the desired bank angle using a simplified PID controller.
        """
        # Compare complete quaternions for accurate error calculation
        error = desired_orientation - self.estimated_orientation
        # PID control (replace with more sophisticated control logic)
        control_signal = self.kp * error + self.ki * self.integrate_error(error, dt) + self.kd
* self.differentiate_error(error, dt)
```

```
        # Extract desired bank angle from the control signal (replace with appropriate
mapping)
        bank_angle = control_signal[2]
        return bank_angle

    def integrate_error(self, error, dt):
        # Now correctly accumulates error as a quaternion
        self.prev_error = self.prev_error + error  # Update previous error quaternion
        return self.ki * self.prev_error  # Return integrated error as a quaternion

    def differentiate_error(self, error, dt):
        # Consider using the full error quaternion for control calculation
        # Replace with the appropriate calculation based on your control strategy
        derivative_error = (error - self.prev_error) / dt
        self.prev_error = error  # Update previous error (all four components)
        return derivative_error

    def apply_thrusters(self, bank_angle):
        # Apply thruster forces based on the bank angle
        thruster_forces = np.array([
            [0, 0, 0],  # Front thrusters
            [0, 0, 0],  # Rear thrusters
            [0, 0, 0],  # Left thrusters
            [0, 0, 0],  # Right thrusters
        ])
        # Update acceleration based on thruster forces
        total_thrust = np.linalg.norm(thruster_forces)
        acceleration_due_to_thrusters = total_thrust / mass
        self.acceleration += acceleration_due_to_thrusters

    def update_orientation(self, bank_angle):
        # Update orientation based on the bank angle
        # Here we assume a simple model: directly updating the orientation quaternion
        delta_quaternion = np.array([np.cos(bank_angle / 2), 0, 0, np.sin(bank_angle / 2)])
        self.orientation = np.quaternion(*delta_quaternion) * np.quaternion(*self.orientation)

    def simulate_descent(self, dt, t_max):
        """
        Simulates the lander's descent in a loop.
        """
        time = np.arange(0, t_max, dt)
        for t in time:
            # Control logic: determine desired orientation
            desired_orientation = np.array([1, 0, 0, 0])  # Quaternion for upright
            # Calculate bank angle
            bank_angle = self.compute_bank_angle(desired_orientation)
            # Apply thrusters
            self.apply_thrusters(bank_angle)
            # Update estimated orientation
            self.update_orientation(bank_angle)
            # Update state
            self.update_state(dt)
```

```python
        # Print state for debugging
        print("Time:", t)
        print("Position:", self.position)
        print("Velocity:", self.velocity)
        print("Orientation:", self.orientation)
        print("Accelerometer:", self.accelerometer)
        print("---")
```