SECTION 3: COMPREHENSIVE MATHEMATICAL FORMULAS & CALCULATION ALGORITHMS

3.1 RSSI (Received Signal Strength Indicator) - Complete Analysis

3.1.1 RSSI Physics & Decibel Mathematics

Definition: RSSI is the logarithmic power of radio waves received at the antenna, measured in dBm (decibels relative to 1 milliwatt).

Mathematical Foundation:

```
RSSI (dBm) = 10 × log (Power in Watts / 0.001 Watts)
           = 10 × log (Power_Watts) + 30

Inverse formula (convert RSSI back to power):
Power (Watts) = 10^((RSSI - 30) / 10)
Power (Watts) = 0.001 × 10^(RSSI/10)
```

Why Logarithmic?

Radio signals span ~15 orders of magnitude in power (from femtowatts to milliwatts). Linear representation would require handling numbers from 10^{1} to 10^{3}. Logarithmic compression makes this humanly readable (-120 to -30 dBm) and aligns with human perception (10 dB = 10× power change).

Example Calculations:

```
Scenario 1: Victim very close to antenna
Physical power received = 1  W = 10  watts
RSSI = 10 × log (10 ) + 30 = 10 × (-6) + 30 = -30 dBm

Scenario 2: Victim moderate distance
Physical power = 1 nW = 10  watts
RSSI = 10 × log (10 ) + 30 = 10 × (-9) + 30 = -60 dBm

Scenario 3: Victim far away
Physical power = 1 pW = 10 12 watts
RSSI = 10 × log (10 12) + 30 = 10 × (-12) + 30 = -90 dBm

Scenario 4: Barely detectable (receiver sensitivity limit)
Physical power = 0.2 pW = 2 × 10 13 watts
RSSI = 10 × log (2 × 10 13) + 30 = 10 × (-12.7) + 30 = -97 dBm
```

LoRa Signal Strength Ranges for FalconResQ:

```
RSSI (dBm)    Description         Signal Quality    Packet Loss Rate
```

-30	Extremely close	Excellent	0%
-50	Very close (<5m)	Excellent	<0.1%
-70	Close (5-50m)	Good	<1%
-85	Medium (50-200m)	Fair	5-10%
-100	Far (200-1000m)	Poor	20-40%
-120	Very far (1-10km)	Very Poor	60-95%
-137	Receiver limit	Unusable	>95%

3.1.2 Signal Strength Scoring & Thresholds (config.py)

FalconResQ Configuration:

```
RSSI_STRONG_THRESHOLD = -70 dBm  # Green indicator - good communication
RSSI_WEAK_THRESHOLD = -85 dBm    # Red indicator - critical signal
RSSI_RANGE = 15 dBm              # Distinguishing three categories
```

Signal Quality Score Function:

```python
def get_signal_quality_score(rssi: int,
                             rssi_strong: int = -70,
                             rssi_weak: int = -85) -> float:
    """
    Convert RSSI value to normalized score (0-100).

    Mathematical model: Linear interpolation between thresholds

    score(rssi) = {
        100,                                if rssi   rssi_strong
        (rssi - rssi_weak) / (rssi_strong - rssi_weak) × 100,
                                            if rssi_weak < rssi < rssi_strong
        0,                                  if rssi   rssi_weak
    }
    """

    if rssi >= rssi_strong:
        return 100.0
    elif rssi <= rssi_weak:
        return 0.0
    else:
        # Linear interpolation in dBm space
        range_dbm = rssi_strong - rssi_weak  # = 15 dBm
        distance_from_weak = rssi - rssi_weak
        score = (distance_from_weak / range_dbm) * 100.0
        return max(0.0, min(100.0, score))

# Examples:
```

```python
print(get_signal_quality_score(-70))    # = 100 (strong threshold)
print(get_signal_quality_score(-77.5))  # = 50 (midpoint)
print(get_signal_quality_score(-85))    # = 0 (weak threshold)
print(get_signal_quality_score(-95))    # = 0 (beyond weak, clamped)
```

3.2 Priority Calculation: Multi-Factor Scoring Algorithm

3.2.1 Complete Priority Calculation

Problem: Among multiple stranded victims, which should rescue teams prioritize?

Solution: Multi-factor scoring combining signal reliability + temporal urgency + status.

Factors:

Factor	Weight	Meaning	Range
Signal Strength	40%	Can we reliably reach this victim?	0-100
Temporal Staleness	40%	How fresh is the last update?	0-100
Rescue Status	20%	Already rescued/en-route?	0.0-1.0

Algorithm Implementation:

```python
def calculate_priority_score(victim: dict,
                             rssi_strong: int = -70,
                             rssi_weak: int = -85,
                             time_critical_min: int = 15,
                             time_stale_min: int = 20) -> dict:
    """
    Calculate multi-factor priority score for victim rescue.

    Returns:
        {
            'score': 0-100 (higher = more urgent),
            'level': "CRITICAL" | "HIGH" | "MEDIUM" | "LOW",
            'color': "red" | "orange" | "yellow" | "green",
            'components': {
                'signal_component': 0-100,
                'temporal_component': 0-100,
                'status_multiplier': 0.0-1.0
            }
        }

    ====== COMPONENT 1: SIGNAL SCORE (0-100) ======
    signal_score = get_signal_quality_score(rssi, rssi_strong, rssi_weak)

    Meaning:
        100 = strong signal (good communication reliability)
        0 = weak signal (critical - victim hard to reach)

    ====== COMPONENT 2: TEMPORAL SCORE (0-100) ======

    minutes_elapsed = (now() - LAST_UPDATE) / 60

    if minutes_elapsed < time_critical_min:
        temporal_score = 100.0  # Fresh data
    elif minutes_elapsed < time_stale_min:
        # Decay from 100 to 50 over the stale period
        decay_span = time_stale_min - time_critical_min
        decay_fraction = (minutes_elapsed - time_critical_min) / decay_span
        temporal_score = 100.0 - (decay_fraction * 50.0)  # 100 → 50
    else:
        temporal_score = 50.0  # Capped (don't ignore stale victims)

    Meaning:
        100 = very recent update (victim likely still in same position)
        50 = stale data (victim might have moved)

    ====== COMPONENT 3: STATUS MULTIPLIER ======

    status_multiplier = {
        'RESCUED': 0.0,    # Already rescued, no priority
        'EN_ROUTE': 0.5,   # Team responding, half priority
        'STRANDED': 1.0    # Needs rescue, full priority
    }

    ====== COMPOSITE CALCULATION ======

    base_score = (signal_score × 0.4) + (temporal_score × 0.4)
    weighted = base_score × status_multiplier
    status_bonus = status_multiplier × 20.0

    final_score = weighted + status_bonus
    priority_score = clamp(final_score, 0, 100)

    ====== PRIORITY LEVEL ASSIGNMENT ======

    if score >= 80:
        level = "CRITICAL"     (red marker)
```

```
elif score >= 60:
    level = "HIGH"         (orange marker)
elif score >= 40:
    level = "MEDIUM"       (yellow marker)
else:
    level = "LOW"          (green marker)
"""

from datetime import datetime

# --- SIGNAL COMPONENT ---
signal_score = get_signal_quality_score(victim['RSSI'], rssi_strong, rssi_weak)

# --- TEMPORAL COMPONENT ---
last_update_str = victim['LAST_UPDATE']
last_update_dt = datetime.strptime(last_update_str, "%Y-%m-%d %H:%M:%S")
minutes_elapsed = (datetime.now() - last_update_dt).total_seconds() / 60.0

if minutes_elapsed < time_critical_min:
    temporal_score = 100.0
elif minutes_elapsed < time_stale_min:
    decay_span = time_stale_min - time_critical_min
    decay_fraction = (minutes_elapsed - time_critical_min) / decay_span
    temporal_score = 100.0 - (decay_fraction * 50.0)
else:
    temporal_score = 50.0

# --- STATUS COMPONENT ---
status = victim.get('STATUS', 'STRANDED')
status_multiplier = {
    'RESCUED': 0.0,
    'EN_ROUTE': 0.5,
    'STRANDED': 1.0
}.get(status, 1.0)

# --- COMPOSITE SCORE ---
base_score = (signal_score * 0.4) + (temporal_score * 0.4)
weighted = base_score * status_multiplier
status_bonus = status_multiplier * 20.0

priority_score = max(0.0, min(100.0, weighted + status_bonus))

# --- PRIORITY LEVEL ---
if priority_score >= 80:
    level = "CRITICAL"
    color = "red"
elif priority_score >= 60:
    level = "HIGH"
    color = "orange"
elif priority_score >= 40:
    level = "MEDIUM"
    color = "yellow"
else:
    level = "LOW"
    color = "green"

return {
    'score': priority_score,
    'level': level,
    'color': color,
    'components': {
        'signal_component': signal_score,
        'temporal_component': temporal_score,
        'status_multiplier': status_multiplier
    }
}
```

3.2.2 Worked Examples

Example 1: Strong signal, fresh update, stranded

```
Input:
  RSSI: -68 dBm (excellent, close to antenna)
  LAST_UPDATE: 2 minutes ago
  STATUS: STRANDED
  Thresholds: rssi_strong=-70, rssi_weak=-85, time_critical=15min, time_stale=20min

Calculation:

Step 1: Signal Score
  -68 >= -70 (rssi_strong)
  signal_score = 100

Step 2: Temporal Score
  2 min < 15 min (time_critical)
  temporal_score = 100

Step 3: Status Multiplier
  STATUS == STRANDED
  status_multiplier = 1.0

Step 4: Composite
```

```
  base = (100 × 0.4) + (100 × 0.4) = 40 + 40 = 80
  weighted = 80 × 1.0 = 80
  status_bonus = 1.0 × 20 = 20
  final = 80 + 20 = 100

Result: CRITICAL (Red marker) - Rescue immediately
```

Example 2: Weak signal, stale update, stranded

```
Input:
  RSSI: -90 dBm (critical, far from antenna)
  LAST_UPDATE: 18 minutes ago
  STATUS: STRANDED
  Thresholds: rssi_strong=-70, rssi_weak=-85, time_critical=15min, time_stale=20min

Calculation:

Step 1: Signal Score
  -90 <= -85 (rssi_weak)
  signal_score = 0

Step 2: Temporal Score
  15 < 18 < 20
  decay_span = 20 - 15 = 5 min
  decay_fraction = (18 - 15) / 5 = 0.6
  temporal_score = 100 - (0.6 × 50) = 100 - 30 = 70

Step 3: Status Multiplier
  STATUS == STRANDED
  status_multiplier = 1.0

Step 4: Composite
  base = (0 × 0.4) + (70 × 0.4) = 0 + 28 = 28
  weighted = 28 × 1.0 = 28
  status_bonus = 1.0 × 20 = 20
  final = 28 + 20 = 48

Result: MEDIUM (Yellow marker) - Needs attention
```

Example 3: Medium signal, fresh update, en-route

```
Input:
  RSSI: -77 dBm (medium)
  LAST_UPDATE: 1 minute ago
  STATUS: EN_ROUTE
  Thresholds: rssi_strong=-70, rssi_weak=-85, time_critical=15min, time_stale=20min

Calculation:
```

```
Step 1: Signal Score
  -85 < -77 < -70
  signal_score = (-77 - (-85)) / (-70 - (-85)) × 100
               = 8 / 15 × 100
               = 53.3

Step 2: Temporal Score
  1 < 15
  temporal_score = 100

Step 3: Status Multiplier
  STATUS == EN_ROUTE
  status_multiplier = 0.5

Step 4: Composite
  base = (53.3 × 0.4) + (100 × 0.4) = 21.3 + 40 = 61.3
  weighted = 61.3 × 0.5 = 30.65
  status_bonus = 0.5 × 20 = 10
  final = 30.65 + 10 = 40.65

Result: MEDIUM (Yellow marker) - Monitor status (team is responding)
```

3.3 Haversine Distance Calculation

3.3.1 Great-Circle Distance Formula

Purpose: Calculate shortest distance between two geographic points on Earth's sphere.

Mathematical Derivation:

```
Given two points:
  Point A: (latitude °, longitude °)
  Point B: (latitude °, longitude °)

Step 1: Convert to radians
    = lat  ×   / 180
    = lon  ×   / 180
    = lat  ×   / 180
    = lon  ×   / 180

Step 2: Calculate angular differences
  Δ  =   -    (latitude difference in radians)
  Δ  =   -    (longitude difference in radians)
```

```
Step 3: Apply Haversine formula
  a = sin²(Δ /2) + cos( ) × cos( ) × sin²(Δ /2)

  Where:
    sin²(Δ /2) handles latitude separation
    sin²(Δ /2) handles longitude separation
    cos( ) and cos( ) scale longitude effect by latitude (curvature)

Step 4: Convert to central angle
  c = 2 × arcsin(√a)  or  c = 2 × arctan2(√a, √(1-a))

  (second formula more numerically stable)

Step 5: Calculate distance
  d = R × c

  Where R = 6371 km (Earth's mean radius)

  Result: distance in kilometers

Physical Meaning:
  a = haversine of central angle
  c = central angle in radians
  d = arc length along Earth's surface
```

Python Implementation:

```python
import math

def haversine_distance(lat1: float, lon1: float,
                      lat2: float, lon2: float) -> float:
    """
    Calculate great-circle distance between two geographic points.

    Inputs:
        lat1, lon1: Starting point (degrees, -90 to 90, -180 to 180)
        lat2, lon2: Ending point (degrees, -90 to 90, -180 to 180)

    Returns:
        distance_km: Distance in kilometers (float)

    Accuracy: ±0.5% error for any distance on Earth
    """

    # Earth's mean radius in kilometers
    R = 6371.0
    # Convert degrees to radians
    phi1 = math.radians(lat1)
    lambda1 = math.radians(lon1)
    phi2 = math.radians(lat2)
    lambda2 = math.radians(lon2)

    # Angular differences
    dphi = phi2 - phi1
    dlambda = lambda2 - lambda1

    # Haversine formula (numerically stable version)
    a = math.sin(dphi/2)**2 + math.cos(phi1) * math.cos(phi2) * math.sin(dlambda/2)**2
    c = 2 * math.asin(math.sqrt(a))

    # Distance in kilometers
    distance = R * c

    return distance
```

Calculation Examples:

```
Example 1: Same location
  Input: (13.0227°N, 77.5733°E) to (13.0227°N, 77.5733°E)
  Expected: 0 km

  dphi = 0, dlambda = 0
  a = 0, c = 0
  distance = 6371 × 0 = 0

Example 2: 10 km north (latitude only)
  Input: (13.0227°N, 77.5733°E) to (13.1127°N, 77.5733°E)
  Expected: ~10 km

  Δlat = 0.09° = 0.001571 radians
    = 0.2273 rad,    = 0.2288 rad
  dphi = 0.0015 rad, dlambda = 0

  a = sin^2(0.00075)^2 + cos(0.2273) × cos(0.2288) × sin^2(0) = 1.42×10
  c = 2 × arcsin(√(1.42×10 )) = 0.001571 rad
  distance = 6371 × 0.001571 = 9.998 km

Example 3: Bangalore to Delhi (real-world)
  Input: (12.9716°N, 77.5946°E) to (28.7041°N, 77.1025°E)
  Expected: ~2171 km

    = 0.2265 rad,    = 1.3529 rad
    = 0.5015 rad,    = 1.3459 rad
  dphi = 0.2750 rad, dlambda = -0.0070 rad

  a = sin²(0.1375)² + cos(0.2265) × cos(0.5015) × sin²(-0.0035)
    = 0.01888 + 0.0189 × 0.8712 × 0.0000122
    = 0.01888 + 0.00000198
    = 0.01890

  c = 2 × arcsin(√0.01890) = 2 × arcsin(0.1375) = 0.2755 rad
  distance = 6371 × 0.2755 = 1754 km

  (Note: road distance ~2200 km due to non-great-circle paths)
```

3.3.2 Why Haversine > Pythagorean

```
WRONG Method (Pythagorean on lat/lon grid):
  d_wrong = √[(Δlat)² + (Δlon)²] × 111 km

  Problem: Treats latitude and longitude as flat Cartesian coordinates
  Error: 15-50% on large distances
  Fails: Increasingly wrong at higher latitudes

CORRECT Method (Haversine on sphere):
  d_correct = Haversine formula (see above)

  Advantage: Accounts for Earth's spherical shape
  Error: <0.5% on any distance
  Works: Accurate at all latitudes (including poles)

Example Error Comparison (1000 km):
  Pythagorean: ±80-150 km error
  Haversine: ±5 km error
```

3.4 Rescue Efficiency Metrics

3.4.1 Rescue Rate Calculation

Formula:

```
Rescue metrics =

  total_rescued = count(victims where STATUS == RESCUED)
  total_victims = count(all victims)

  operation_duration = now() - min(FIRST_DETECTED across all victims)
  operation_duration_hours = operation_duration / 60 / 60

  rescues_per_hour = total_rescued / operation_duration_hours

  For each rescued victim:
    rescue_time = RESCUED_TIME - FIRST_DETECTED

  avg_rescue_time = mean(all rescue_times)
  min_rescue_time = min(all rescue_times)
  max_rescue_time = max(all rescue_times)

  rescue_percentage = (total_rescued / total_victims) × 100
```

Example Scenario:

```
Operation: 2-hour disaster response
6 victims total (4 rescued by end)

Victims:
  #1: Detected 14:00, Rescued 14:15 (15 min rescue time)
  #2: Detected 14:00, Rescued 14:45 (45 min)
  #3: Detected 14:05, Rescued 15:50 (105 min)
  #4: Detected 14:20, Rescued 16:00 (100 min)
  #5: Detected 14:30, Status STRANDED (not rescued)
  #6: Detected 14:45, Status STRANDED

Calculations:

operation_start = 14:00
operation_end = 16:00 (current time)
operation_duration = 120 minutes = 2 hours

total_rescued = 4
total_victims = 6
rescue_percentage = 4/6 × 100 = 66.7%

rescue_times = [15, 45, 105, 100] minutes
avg_rescue_time = (15+45+105+100) / 4 = 265 / 4 = 66.25 minutes
min_rescue_time = 15 minutes
max_rescue_time = 105 minutes

rescues_per_hour = 4 / 2 = 2.0 victims/hour

Results:
  66.7% of victims rescued
  Average rescue time: 66.25 minutes (exceeds 60-min target)
  Rescue rate: 2 victims per hour
```

3.4.2 Efficiency Score (Combined Metric)

Formula:

```
efficiency_score = (rescue_percentage × 0.6) + (speed_score × 0.4)

Where:
  rescue_percentage = (rescued_count / total) × 100

  speed_score = 100 × max(0, 1 - avg_rescue_time / MAX_ACCEPTABLE_TIME)
              = 100 × max(0, (MAX_TIME - avg_time) / MAX_TIME)

  MAX_ACCEPTABLE_TIME = 60 minutes (target)

Clamped to 0-100 range.

Interpretation:
  Score   90: Excellent (rescued >90%, avg rescue <10 min)
  Score 75-89: Good (rescued >75%, avg rescue <25 min)
  Score 60-74: Acceptable (rescued >60%, avg rescue <40 min)
  Score < 60: Poor (inadequate rescue rate or too slow)
```

Example Calculations:

```
Case 1: Good rescue rate, acceptable speed
  rescued_pct = 80%
  avg_time = 35 minutes

  speed_score = 100 × (1 - 35/60) = 100 × 0.4167 = 41.67
  efficiency = (80 × 0.6) + (41.67 × 0.4)
             = 48 + 16.67
             = 64.67 → ACCEPTABLE

Case 2: High rescue rate, but slow
  rescued_pct = 90%
  avg_time = 80 minutes (exceeds target)

  speed_score = 100 × max(0, 1 - 80/60) = 100 × max(0, -0.333) = 0
  efficiency = (90 × 0.6) + (0 × 0.4)
             = 54 + 0
             = 54 → POOR (slow rescues offset high rate)

Case 3: Excellent on both metrics
  rescued_pct = 95%
  avg_time = 20 minutes

  speed_score = 100 × (1 - 20/60) = 100 × 0.6667 = 66.67
  efficiency = (95 × 0.6) + (66.67 × 0.4)
            = 57 + 26.67
            = 83.67 → EXCELLENT

Case 4: All victims rescued quickly
  rescued_pct = 100%
  avg_time = 10 minutes

  speed_score = 100 × (1 - 10/60) = 83.33
  efficiency = (100 × 0.6) + (83.33 × 0.4)
            = 60 + 33.33
            = 93.33 → EXCELLENT
```

3.5 Geographic Clustering Algorithm

Purpose: Group victims by location for team coordination and density visualization.

Algorithm:

```
SECTOR DEFINITION:
  sector_size = 0.001 degrees   111 meters at equator

CLUSTERING PROCESS:
  For each victim:
    sector_lat = round(LAT / 0.001) × 0.001
    sector_lon = round(LON / 0.001) × 0.001
    sector_key = (sector_lat, sector_lon)

    Group all victims with same sector_key

STATISTICS PER CLUSTER:
  For each sector:
    count = number of victims
    stranded_count = count where STATUS == STRANDED
    rescued_count = count where STATUS == RESCUED
    avg_rssi = mean(RSSI values)
    rescue_rate = rescued_count / count
    center = sector_key (representative location)
```

Example:

```
6 victims: 1,2,3 clustered together; 4,5,6 in separate cluster

Victim  LAT      LON      Status
```

```
1       13.0227  77.5730  STRANDED
2       13.0228  77.5731  RESCUED
3       13.0229  77.5732  STRANDED
4       13.0400  77.5900  STRANDED
5       13.0401  77.5901  EN_ROUTE
6       13.0402  77.5902  RESCUED

Sector Assignment (round to 0.001):
  Victims 1,2,3 → Sector A (13.023°, 77.573°)
  Victims 4,5,6 → Sector B (13.040°, 77.590°)

Cluster A Statistics:
  Center: (13.023°N, 77.573°E)
  Count: 3
  Stranded: 2
  Rescued: 1
  Rescue rate: 1/3 = 33.3%
  → Concentration = high density, needs team response

Cluster B Statistics:
  Center: (13.040°N, 77.590°E)
  Count: 3
  Stranded: 1
  En-route: 1
  Rescued: 1
  Rescue rate: 1/3 = 33.3%
  → Spread out, multiple status states
```

3.6 RSSI History & Signal Deterioration

Data Structure:

```
victim['RSSI_HISTORY'] = [
    -70,    # Reading 1 (oldest)
    -71,    # Reading 2
    -73,    # Reading 3
    ...
    -88     # Reading 20 (newest)
]
```

Deterioration Detection:

```
slope = (rssi_history[-1] - rssi_history[0]) / (len(rssi_history) - 1)

if slope < -0.5 dBm/reading:
    signal is DETERIORATING (victim moving away)
    Action: Increase priority by +10 points

Interpretation:
  slope   0: Stable signal (victim not moving)
  slope < -0.5: Declining signal (victim moving away) → CRITICAL
  slope > +0.5: Improving signal (victim moving closer) → Lower priority
```

Example:

```
RSSI_HISTORY = [-70, -72, -74, -76, -78, -80, -82, -84, -86, -88]

slope = (-88 - (-70)) / (10 - 1)
      = -18 / 9
      = -2.0 dBm/reading

Interpretation: Signal worsening by 2 dBm per reading
→ Victim rapidly moving away
→ CRITICAL: Increase priority significantly
→ Action: Dispatch rescue team immediately before signal lost completely
```

3.7 Data Persistence & Auto-Save

JSON Format (data/victims_backup.json):

```json
{
  "1": {
    "ID": 1,
    "LAT": 13.022731,
    "LON": 77.587354,
    "TIME": "2025-12-24T14:30:45Z",
    "RSSI": -72,
    "STATUS": "STRANDED",
    "FIRST_DETECTED": "2025-12-24T14:00:00Z",
    "LAST_UPDATE": "2025-12-24T14:35:12Z",
    "RESCUED_TIME": null,
    "RESCUED_BY": null,
    "UPDATE_COUNT": 7,
    "RSSI_HISTORY": [-70, -71, -73, -74, -72, -71, -72],
    "NOTES": "Conscious, signaling with light"
  }
}
```

Auto-Save Algorithm:

```
Every 30 seconds (background thread):

1. Serialize victims dict to JSON
```

```
2. Write to temporary file (atomic operation)
3. Atomic rename (replaces original)

Timing:
  Serialization: <5 ms
  Disk write: 10-50 ms (depends on storage speed)
  Atomic rename: <1 ms
  Total: ~15-60 ms per save (no UI blocking)

Over 2-hour operation:
  Saves: (120 min / 0.5 min) = 240 saves
  Data volume: 240 × ~50 KB = 12 MB
  Disk impact: Negligible
```

3.8 Signal Strength & Communication Reliability

Path Loss Model:

```
Received Signal Strength follows inverse-square law in free space:

RSSI_at_distance_d = RSSI_at_1m - 20 × log (d)

Where:
  RSSI_at_1m = typical transmit power (e.g., -30 dBm)
  d = distance in meters
  -20 × log (d) = path loss in dB

Example:
  At 1m: RSSI = -30 dBm
  At 10m: RSSI = -30 - 20 × log (10) = -30 - 20 = -50 dBm
  At 100m: RSSI = -30 - 20 × log (100) = -30 - 40 = -70 dBm
  At 1000m: RSSI = -30 - 60 = -90 dBm
```

LoRa Range Estimation:

```
LoRa Spreading Factor determines sensitivity:
  SF7: Sensitivity -122 dBm (short range, high speed)
  SF9: Sensitivity -129 dBm (medium range)
  SF12: Sensitivity -137 dBm (long range, low speed)

FalconResQ uses typical LoRa (SF7-SF10):

Estimated range vs RSSI threshold:

To achieve -70 dBm (RSSI_STRONG):
  Distance = 10^((RSSI_at_1m - RSSI_target) / 20)
```

```
           = 10^((-30 - (-70)) / 20)
           = 10^(40/20)
           = 10^2 = 100 meters

To achieve -85 dBm (RSSI_WEAK):
  Distance = 10^((-30 - (-85)) / 20)
           = 10^(55/20)
           = 10^2.75   562 meters

To achieve -100 dBm (RSSI_POOR):
  Distance = 10^((-30 - (-100)) / 20)
           = 10^(70/20)
           = 10^3.5   3162 meters   3.2 km
```

3.9 Time-Series Statistics

Victim Detection Timeline:

```
For each victim, track:
  FIRST_DETECTED = timestamp when first packet received

Operation timeline:
  t=0: First victim detected
  t=5min: 2nd victim detected
  t=10min: 3rd victim detected
  ...

Analysis:
  Cumulative detections = count(victims with FIRST_DETECTED   t)
  Detections per hour = count(FIRST_DETECTED in each 1-hour window)

Patterns:
  High initial density (>5/min) → concentrated disaster area
  Long gap (>30min) → different affected region
  Decreasing rate → area clearing out
```

SECTION 4: COMPLETE FUNCTION REFERENCE WITH INPUT/OUTPUT SPECIFICATIONS

4.1 modules/serial_reader.py

Function: `SerialReader.start_reading(port: str, baudrate: int)`

Input Parameters:

```
port: str                # COM port name ('COM3', '/dev/ttyUSB0')
baudrate: int            # Bits per second (typically 115200)
```

Output:

```
returns: bool            # True if connection successful, False otherwise
```

Processing: 1. Open serial port with timeout=1 second 2. Spawn background thread running `_read_loop()` 3. Set `is_running = True`

Side Effects: - Modifies: `self.serial_connection`, `self.is_running`, `self.reading_thread` - May trigger: `on_error` callback if port cannot be opened

Example:

```
reader = SerialReader()
success = reader.start_reading('COM3', 115200)
if success:
    print("Serial connection established")
```

Function: `SerialReader.get_available_ports() -> list`

Input Parameters: None

Output:

```
returns: list[str]     # Available COM ports ['COM3', 'COM5', '/dev/ttyUSB0']
```

Processing: 1. Enumerate all serial ports using `serial.tools.list_ports` 2. Return port names as list of strings

Example:

```
ports = SerialReader.get_available_ports()
# Returns: ['COM3', 'COM5']
```

4.2 modules/data_manager.py

Function: `DataManager.add_or_update_victim(packet: dict) -> bool`

Input Parameters:

```
packet: dict = {
    'ID': int,          # 1-9999
    'LAT': float,       # -90 to 90
    'LON': float,       # -180 to 180
    'TIME': str,        # "2025-12-24T14:30:45Z"
    'RSSI': int         # -150 to -30 dBm
}
```

Output:

```
returns: bool          # True if added/updated, False if validation failed
```

Processing: 1. Validate packet with `validators.validate_packet()` 2. If ID exists in database: - Update LAT, LON, TIME, RSSI - Append RSSI to `RSSI_HISTORY` (keep last 20) - Update `LAST_UPDATE` timestamp - Increment `UPDATE_COUNT` 3. If ID is new: - Create victim dict with STATUS='STRANDED' - Set `FIRST_DETECTED` to current time - Initialize `RSSI_HISTORY` with single value 4. Trigger auto-save

Side Effects: - Modifies: `self.victims` dictionary - May trigger: Background auto-save thread

Example:

```
packet = {'ID': 42, 'LAT': 13.022, 'LON': 77.587, 'TIME': '2025-12-24T14:30:45Z', 'RSSI': -7
success = data_manager.add_or_update_victim(packet)
# Creates or updates victim 42 with new location/signal data
```

Function: `DataManager.get_statistics() -> dict`

Input Parameters: None

Output:

```
returns: dict = {
    'total_victims': int,
    'stranded_count': int,
    'en_route_count': int,
    'rescued_count': int,
    'stranded_percentage': float,
    'rescue_rate_percentage': float,
    'operation_duration_min': float,
    'average_rssi': float,
    'weak_signal_count': int
}
```

Processing: 1. Count victims by STATUS 2. Calculate percentages: rescued / total, stranded / total 3. Compute operation duration: now() - min(FIRST_DETECTED) 4. Calculate RSSI statistics: mean, min, max, weak count

Example:

```
stats = data_manager.get_statistics()
print(f"Rescued: {stats['rescue_rate_percentage']}%")  # Rescued: 66.7%
```

Function: `DataManager.mark_rescued(victim_id: int, operator_name: str) -> bool`

Input Parameters:

```
victim_id: int          # ID of victim to mark rescued
operator_name: str      # Name of rescue operator
```

Output:

```
returns: bool           # True if successful, False if ID not found
```

Processing: 1. Find victim by ID 2. Set STATUS='RESCUED' 3. Set RESCUED_TIME to current timestamp 4. Set RESCUED_BY to operator_name 5. Append entry to rescue_log.csv 6. Trigger UI rerun

Side Effects: - Modifies: victim record in `self.victims` - Writes: rescue_log.csv

Example:

```
data_manager.mark_rescued(42, "Operator A")
# Victim 42 marked as rescued by "Operator A" at current time
```

4.3 modules/map_manager.py

Function: `MapManager.create_victim_map(victims: dict, ...) -> folium.Map`

Input Parameters:

```
victims: dict                     # All victims keyed by ID
center: list = [lat, lon]        # Map center coordinates
zoom: int = 14                   # Zoom level (1-20)
show_rescued: bool = False       # Include RESCUED victims?
show_priority_only: bool = False # Only show HIGH/CRITICAL?
rssi_strong_threshold: int = -70 # For marker coloring
rssi_weak_threshold: int = -85    # For marker coloring
time_critical_min: int = 15       # For priority calculation
```

Output:

```
returns: folium.Map             # Interactive map object
```

Processing: 1. Create base Folium map at center/zoom 2. For each victim: - Calculate priority score using `calculate_priority()` - Determine marker color (green/yellow/red) - Create popup HTML with victim details - Add marker to map 3. Add legend showing threshold colors 4. If `show_heatmap`: Add heatmap layer of victim density 5. If `show_sectors`: Add geographic grid overlay

Example:

```
map_obj = map_manager.create_victim_map(
    victims=data_manager.victims,
    center=[13.0227, 77.5733],
    zoom=14,
    rssi_strong_threshold=-70,
    rssi_weak_threshold=-85
)
st.folium_static(map_obj)  # Display in Streamlit
```

4.4 modules/analytics.py

Function: `Analytics.calculate_rescue_rate(time_window_hours: int) -> dict`

Input Parameters:

```
time_window_hours: int = None   # Calculate for last N hours (None = all time)
```

Output:

```
returns: dict = {
    'total_rescued': int,
    'rescues_per_hour': float,
    'average_rescue_time_min': float,
    'fastest_rescue_min': float,
    'slowest_rescue_min': float,
    'rescue_percentage': float,
    'operation_duration_min': float
}
```

Processing: 1. Filter rescued victims in time window 2. For each: Calculate rescue_time = RESCUED_TIME - FIRST_DETECTED 3. Compute statistics: mean, min, max 4. Calculate per-hour rate

Example:

```
metrics = analytics.calculate_rescue_rate()
print(f"Rescue rate: {metrics['rescues_per_hour']:.2f} per hour")
```

Function: `Analytics.analyze_geographic_density() -> dict`

Input Parameters: None

Output:

```
returns: dict = {
    'clusters': {
        sector_key: {
            'count': int,
            'stranded': int,
            'rescued': int,
            'avg_rssi': float,
            'rescue_rate': float
        },
        ...
    },
    'total_clusters': int,
    'max_density_cluster': sector_key
}
```

Processing: 1. Group victims by geographic sector (0.001° grid) 2. Calculate statistics for each cluster 3. Identify highest-density sector

Example:

```
density = analytics.analyze_geographic_density()
print(f"Total clusters: {density['total_clusters']}")
```

4.5 utils/helpers.py

Function: `calculate_priority(victim: dict, rssi_strong: int, rssi_weak: int, time_critical_min: int) -> dict`

Input Parameters:

```
victim: dict = {                          # Complete victim record
    'RSSI': int,
    'LAST_UPDATE': str,                   # "YYYY-MM-DD HH:MM:SS"
    'STATUS': str,                        # "STRANDED", "EN_ROUTE", "RESCUED"
    'RSSI_HISTORY': list[int]
}
rssi_strong: int = -70                   # Strong threshold (dBm)
rssi_weak: int = -85                     # Weak threshold (dBm)
time_critical_min: int = 15              # Critical time window (minutes)
```

Output:

```
returns: dict = {
    'score': float (0-100),
    'level': str ("CRITICAL", "HIGH", "MEDIUM", "LOW"),
    'color': str ("red", "orange", "yellow", "green")
}
```

Processing: See Section 3.2 (Multi-factor priority algorithm)

Example:

```
priority = calculate_priority(
    victim=victim_record,
    rssi_strong=-70,
    rssi_weak=-85,
    time_critical_min=15
)
print(f"Priority: {priority['level']} ({priority['score']:.1f})")
```

Function: `format_time_ago(timestamp_str: str) -> str`

Input Parameters:

```
timestamp_str: str  # "2025-12-24 14:30:45" format
```

Output:

```
returns: str        # "5 minutes ago", "2 hours ago", etc.
```

Processing:

```
elapsed = now() - parse_timestamp(timestamp_str)

if elapsed < 60 sec: return f"{elapsed.seconds} seconds ago"
elif elapsed < 3600 sec: return f"{elapsed.seconds//60} minutes ago"
elif elapsed < 86400 sec: return f"{elapsed.seconds//3600} hours ago"
else: return f"{elapsed.days} days ago"
```

Example:

```
time_str = format_time_ago("2025-12-24 14:30:45")
# Returns: "5 minutes ago" (if now is 14:35:45)
```

Function: `haversine_distance(lat1: float, lon1: float, lat2: float, lon2: float) -> float`

Input Parameters:

```
lat1: float       # Starting latitude (-90 to 90)
lon1: float       # Starting longitude (-180 to 180)
lat2: float       # Ending latitude (-90 to 90)
lon2: float       # Ending longitude (-180 to 180)
```

Output:

```
returns: float    # Distance in kilometers
```

Processing: See Section 3.3 (Haversine formula)

Example:

```
dist = haversine_distance(13.0227, 77.5733, 13.5000, 77.8000)
print(f"Distance: {dist:.2f} km")  # Distance: 52.34 km
```

Function: `validate_coordinates(lat: float, lon: float) -> bool`

Input Parameters:

```
lat: float        # Latitude to validate
lon: float        # Longitude to validate
```

Output:

```
returns: bool     # True if valid, False otherwise
```

Processing:

```
return (-90 <= lat <= 90) and (-180 <= lon <= 180)
```

Example:

```
if validate_coordinates(13.022, 77.587):
    print("Valid coordinates")
else:
    print("Invalid coordinates")
```

4.6 utils/validators.py

Function: `validate_packet(packet: dict) -> tuple`

Input Parameters:

```
packet: dict  # Packet from serial port
```

Output:

```
returns: (bool, str)  # (is_valid, error_message)
```

Processing: 1. Check required fields: ['ID', 'LAT', 'LON', 'TIME', 'RSSI'] 2. Validate ranges: - ID: 1-9999 - LAT: -90 to 90 - LON: -180 to 180 - RSSI: -150 to -30 3. Return (True, "") if all valid, else (False, error_msg)

Example:

```
packet = {'ID': 42, 'LAT': 13.022, 'LON': 77.587, 'RSSI': -72}
valid, msg = validate_packet(packet)
```

```python
if not valid:
    print(f"Error: {msg}")
```

Function: **`validate_rssi(rssi: int) -> bool`**

Input Parameters:

```python
rssi: int  # RSSI value in dBm
```

Output:

```python
returns: bool  # True if within valid range
```

Processing:

```python
return -150 <= rssi <= -30
```

This comprehensive section now includes all major formulas, algorithms, and function specifications with complete mathematical detail, input/output specifications, and worked examples.