🏔️ AI-Powered Slope Stability Monitoring System

Complete Pipeline from Drone Images to FOS Calculation

📷 PHASE 1: Data Acquisition & Preprocessing

1.1 Drone Image Capture Strategy

python

```
# Optimal flight parameters for slope monitoring
Flight Planning:
├── Overlap: 80% front, 70% side overlap
├── Resolution: <2cm/pixel for crack detection
├── Flight pattern: Grid + oblique angles (45°)
├── Multiple flights: Different seasons/conditions
└── GPS accuracy: RTK/PPK for precise georeferencing

# Camera settings
├── RAW format for maximum detail
├── Fixed exposure for consistent lighting
├── Multiple angles: Nadir + oblique views
└── Stereo pairs for 3D reconstruction
```

1.2 Data Types to Collect

- **RGB Images**: Visual crack detection
- **LiDAR Data**: Precise 3D geometry (if available)
- **Multispectral**: Vegetation health, moisture detection
- **Thermal**: Temperature variations, seepage detection
- **GPS Coordinates**: Precise georeferencing

🏗️ PHASE 2: 3D Model Construction

2.1 Photogrammetry Pipeline

A) Structure from Motion (SfM)

python

```
# Software options:
Commercial:
├── Agisoft Metashape (Best accuracy)
├── Pix4D (Industry standard)
└── RealityCapture (Fast processing)

Open Source:
├── OpenDroneMap (Free, good quality)
├── COLMAP (Research-grade)
└── MeshRoom (Alembic ecosystem)

# Process flow:
Images → Feature matching → Camera poses → Dense point cloud → 3D mesh
```

B) Point Cloud Processing

```python
# Key steps:
1. Noise removal ───► Remove outliers, artifacts
2. Classification ──► Ground, vegetation, rock faces
3. Meshing ──────────► Convert points to 3D surfaces
4. Texture mapping ► Apply original images to 3D model
5. Georeferencing ──► Align to real-world coordinates

# Tools:
├── CloudCompare (Free, powerful)
├── PCL (Programming library)
├── Open3D (Python-based)
└── MeshLab (Mesh processing)
```

2.2 Quality Assessment

```python
# Check 3D model quality:
Point density ───────► >100 points/m² for crack detection
Accuracy ────────────► <5cm error vs ground truth
Completeness ────────► No holes in critical areas
Texture quality ──► Sharp, well-aligned textures
```

🤖 PHASE 3: AI Detection Models

3.1 Multi-Target Detection System

A) Crack Detection

```python
# Best models for slope cracks:
Primary: SAM (Segment Anything)
├── Zero-shot crack segmentation
├── Works on 3D texture maps
└── Highly accurate for linear features

Alternative: EfficientNet-UNet
├── 95%+ accuracy on crack datasets
├── Fine-tuned for geological features
└── Pixel-level crack mapping

# Implementation:
from segment_anything import SamPredictor, sam_model_registry
import cv2

def detect_cracks_on_3d_model(texture_image):
    sam = sam_model_registry["vit_h"](checkpoint="sam_vit_h.pth")
    predictor = SamPredictor(sam)

    # Process each texture patch
    predictor.set_image(texture_image)
    # Use automatic mask generation for full coverage
    masks = predictor.generate_masks()

    return crack_masks
```

B) Motion Detection (Displacement Analysis)

```python
```

```
# Multi-temporal analysis:
Time Series Analysis:
├── Compare 3D models from different dates
├── Calculate displacement vectors
├── Identify movement patterns
└── Detect acceleration/deceleration

# Tools & Methods:
CloudCompare M3C2:
├── Precise distance calculations between point clouds
├── Statistical significance testing
├── Uncertainty quantification
└── 3D displacement vectors

# Python implementation:
import open3d as o3d
import numpy as np

def detect_motion(cloud_t1, cloud_t2, threshold=0.05):
    # Register point clouds
    transformation = register_point_clouds(cloud_t1, cloud_t2)

    # Calculate distances
    distances = np.asarray(cloud_t1.compute_point_cloud_distance(cloud_t2))

    # Identify significant movements
    moving_points = distances > threshold
    return moving_points, distances
```

C) Joint Set Analysis

```python
```

```
# Discontinuity detection:
Automated Joint Analysis:
├── Plane fitting algorithms
├── Orientation analysis (dip/dip direction)
├── Spacing measurements
├── Persistence calculations
└── Joint set classification

# Implementation approach:
def analyze_joint_sets(point_cloud):
    # 1. Segment planar surfaces
    planes = segment_planes_ransac(point_cloud)

    # 2. Calculate orientations
    orientations = []
    for plane in planes:
        dip, dip_direction = calculate_orientation(plane.normal)
        orientations.append((dip, dip_direction))

    # 3. Cluster similar orientations
    joint_sets = cluster_orientations(orientations)

    # 4. Calculate geometric properties
    for joint_set in joint_sets:
        spacing = calculate_spacing(joint_set)
        persistence = calculate_persistence(joint_set)
        roughness = calculate_roughness(joint_set)

    return joint_sets
```

3.2 Integration with 3D Models

```
python
```

```python
# Map 2D detections to 3D coordinates:
def map_2d_to_3d(detection_mask, texture_coordinates, point_cloud):
    # Convert 2D pixel coordinates to 3D world coordinates
    world_coords = []
    for pixel in detection_mask:
        if pixel in texture_coordinates:
            world_coord = texture_coordinates[pixel]
            world_coords.append(world_coord)

    return world_coords
```

📊 PHASE 4: Geotechnical Parameter Extraction

4.1 Rock Mass Classification Parameters

A) From AI Detection Results

python

```python
# Extract geotechnical parameters:
def extract_geotechnical_params(cracks, joints, motion_data):
    parameters = {}

    # RQD (Rock Quality Designation)
    parameters['RQD'] = calculate_rqd_from_cracks(cracks)

    # Joint parameters
    parameters['joint_spacing'] = calculate_joint_spacing(joints)
    parameters['joint_orientation'] = get_joint_orientations(joints)
    parameters['joint_persistence'] = calculate_persistence(joints)
    parameters['joint_roughness'] = estimate_roughness(joints)
    parameters['joint_aperture'] = measure_aperture(cracks)

    # Weathering assessment
    parameters['weathering_grade'] = assess_weathering_from_texture()

    # Groundwater conditions
    parameters['groundwater'] = detect_seepage_signs()

    return parameters
```

B) Rock Mass Rating (RMR) Calculation

python

```python
def calculate_rmr(parameters):
    rmr_score = 0

    # 1. Uniaxial compressive strength (from rock type/field tests)
    rmr_score += get_ucs_rating(parameters['rock_type'])

    # 2. RQD
    rmr_score += get_rqd_rating(parameters['RQD'])

    # 3. Joint spacing
    rmr_score += get_spacing_rating(parameters['joint_spacing'])

    # 4. Joint condition
    rmr_score += get_joint_condition_rating(
        parameters['joint_persistence'],
        parameters['joint_roughness'],
        parameters['joint_aperture']
    )

    # 5. Groundwater conditions
    rmr_score += get_groundwater_rating(parameters['groundwater'])

    return rmr_score
```

4.2 Strength Parameters

python

```python
# Convert RMR to strength parameters:
def rmr_to_strength_params(rmr_score):
    # Hoek-Brown parameters
    mb = np.exp((rmr_score - 100) / 28)
    s = np.exp((rmr_score - 100) / 9)
    a = 0.5 + (np.exp(-rmr_score/15) - np.exp(-20/3)) / 6

    # Mohr-Coulomb parameters
    friction_angle = 20 + 30 * (rmr_score / 100)
    cohesion = 0.05 + 0.25 * (rmr_score / 100)  # MPa

    return {
        'friction_angle': friction_angle,
        'cohesion': cohesion,
        'mb': mb, 's': s, 'a': a
    }
```

⚖️ PHASE 5: Factor of Safety (FOS) Calculation

5.1 Slope Stability Analysis Methods

A) Limit Equilibrium Methods

python

```python
# Bishop's Simplified Method
def bishops_method(slope_geometry, soil_params, water_table):
    slices = discretize_slope(slope_geometry)

    for iteration in range(max_iterations):
        F = 0  # Factor of safety

        for slice in slices:
            # Calculate forces on each slice
            W = slice.weight
            u = calculate_pore_pressure(slice, water_table)

            # Normal and shear forces
            N = W * np.cos(slice.alpha) + (slice.X_right - slice.X_left)
            T = W * np.sin(slice.alpha) + (slice.E_right - slice.E_left)

            # Available shear strength
            s_available = (slice.c * slice.width +
                           (N - u * slice.width) * np.tan(slice.phi))

            F += s_available / T

        if converged(F):
            break

    return F

# Janbu's Method (for non-circular surfaces)
def janbu_method(slope_geometry, soil_params):
    # Similar implementation for irregular failure surfaces
    pass
```

B) Finite Element Method (Advanced)

```python
```

```python
# For complex geometries and stress analysis
import numpy as np
from scipy.sparse import csc_matrix
from scipy.sparse.linalg import spsolve

def fem_slope_analysis(mesh, material_properties, boundary_conditions):
    # Assemble stiffness matrix
    K = assemble_stiffness_matrix(mesh, material_properties)

    # Apply boundary conditions
    K, F = apply_boundary_conditions(K, boundary_conditions)

    # Solve for displacements
    displacements = spsolve(K, F)

    # Calculate stresses and safety factors
    stresses = calculate_stresses(displacements, material_properties)
    local_fos = calculate_local_safety_factors(stresses, material_properties)

    return np.min(local_fos)  # Critical FOS
```

5.2 3D Slope Stability

python

```python
def calculate_3d_fos(point_cloud, joint_sets, strength_params):
    # Identify potential failure surfaces
    failure_surfaces = identify_failure_surfaces(joint_sets)

    fos_values = []
    for surface in failure_surfaces:
        # Extract geometry along failure surface
        surface_geometry = extract_surface_geometry(point_cloud, surface)

        # Calculate 3D limit equilibrium
        fos = calculate_3d_limit_equilibrium(
            surface_geometry,
            strength_params,
            joint_orientations=surface.orientations
        )
        fos_values.append(fos)

    return min(fos_values), fos_values
```

🗺️ PHASE 6: 2D Stability Mapping

6.1 Create Stability Maps

```
python
```

```python
import matplotlib.pyplot as plt
import geopandas as gpd
from shapely.geometry import Point, Polygon

def create_stability_map(slope_area, fos_values, coordinates):
    # Create grid for interpolation
    grid_x, grid_y = np.meshgrid(
        np.linspace(coordinates.x.min(), coordinates.x.max(), 100),
        np.linspace(coordinates.y.min(), coordinates.y.max(), 100)
    )

    # Interpolate FOS values
    from scipy.interpolate import griddata
    fos_grid = griddata(
        (coordinates.x, coordinates.y),
        fos_values,
        (grid_x, grid_y),
        method='cubic'
    )

    # Create stability classification
    stability_classes = np.zeros_like(fos_grid)
    stability_classes[fos_grid > 1.5] = 4  # Safe
    stability_classes[(fos_grid > 1.2) & (fos_grid <= 1.5)] = 3  # Good
    stability_classes[(fos_grid > 1.0) & (fos_grid <= 1.2)] = 2  # Caution
    stability_classes[fos_grid <= 1.0] = 1  # Critical

    return grid_x, grid_y, stability_classes
```

6.2 Visualization

```python
```

```python
def plot_stability_map(grid_x, grid_y, stability_classes, detected_features):
    fig, ax = plt.subplots(1, 1, figsize=(12, 8))

    # Color map for stability zones
    colors = ['red', 'orange', 'yellow', 'green']
    im = ax.contourf(grid_x, grid_y, stability_classes,
                     levels=4, colors=colors, alpha=0.7)

    # Overlay detected features
    for crack in detected_features['cracks']:
        ax.plot(crack.x, crack.y, 'k-', linewidth=2, label='Cracks')

    for joint in detected_features['joints']:
        ax.plot(joint.x, joint.y, 'b--', linewidth=1, label='Joints')

    # Add legend and labels
    ax.set_xlabel('Easting (m)')
    ax.set_ylabel('Northing (m)')
    ax.set_title('Slope Stability Map')

    # Color bar
    cbar = plt.colorbar(im, ax=ax)
    cbar.set_label('Stability Class')
    cbar.set_ticks([1, 2, 3, 4])
    cbar.set_ticklabels(['Critical', 'Caution', 'Good', 'Safe'])

    return fig, ax
```

🚨 PHASE 7: Alert System & Monitoring

7.1 Risk Assessment

```python
```

```python
def assess_risk_level(fos, crack_density, motion_rate):
    risk_score = 0

    # FOS contribution (50% weight)
    if fos < 1.0:
        risk_score += 50
    elif fos < 1.2:
        risk_score += 30
    elif fos < 1.5:
        risk_score += 15

    # Crack density contribution (30% weight)
    if crack_density > 0.8:  # cracks per m²
        risk_score += 30
    elif crack_density > 0.5:
        risk_score += 20
    elif crack_density > 0.2:
        risk_score += 10

    # Motion rate contribution (20% weight)
    if motion_rate > 10:  # mm/year
        risk_score += 20
    elif motion_rate > 5:
        risk_score += 15
    elif motion_rate > 2:
        risk_score += 10

    # Classify risk level
    if risk_score >= 70:
        return "CRITICAL"
    elif risk_score >= 50:
        return "HIGH"
    elif risk_score >= 30:
        return "MODERATE"
    else:
        return "LOW"
```

7.2 Automated Monitoring

```
python
```

```python
class SlopeMonitoringSystem:
    def __init__(self):
        self.baseline_model = None
        self.alert_thresholds = {
            'fos_critical': 1.0,
            'fos_warning': 1.2,
            'motion_critical': 10,  # mm/year
            'new_crack_threshold': 0.1  # m length
        }

    def process_new_survey(self, new_images, new_lidar=None):
        # 1. Build 3D model
        new_model = build_3d_model(new_images, new_lidar)

        # 2. Detect features
        cracks = detect_cracks(new_model)
        joints = analyze_joint_sets(new_model)
        motion = detect_motion(self.baseline_model, new_model)

        # 3. Calculate FOS
        fos_map = calculate_fos_distribution(new_model, cracks, joints)

        # 4. Assess changes
        alerts = self.check_for_alerts(cracks, motion, fos_map)

        # 5. Update baseline
        self.baseline_model = new_model

        return {
            'stability_map': fos_map,
            'detected_features': {'cracks': cracks, 'joints': joints},
            'motion_analysis': motion,
            'alerts': alerts
        }

    def check_for_alerts(self, cracks, motion, fos_map):
        alerts = []

        # Check FOS values
        critical_fos_areas = np.where(fos_map < self.alert_thresholds['fos_critical'])
        if len(critical_fos_areas[0]) > 0:
            alerts.append({
                'type': 'CRITICAL_FOS',
```

```
            'severity': 'HIGH',
            'locations': critical_fos_areas,
            'message': 'Factor of Safety below 1.0 detected'
        })

    # Check for new cracks
    new_major_cracks = [c for c in cracks if c.length > self.alert_thresholds['new_crack_threshold']]
    if new_major_cracks:
        alerts.append({
            'type': 'NEW_CRACKS',
            'severity': 'MEDIUM',
            'count': len(new_major_cracks),
            'message': f'{len(new_major_cracks)} new significant cracks detected'
        })

    # Check motion rates
    high_motion_areas = np.where(motion.annual_rate > self.alert_thresholds['motion_critical'])
    if len(high_motion_areas[0]) > 0:
        alerts.append({
            'type': 'EXCESSIVE_MOTION',
            'severity': 'HIGH',
            'locations': high_motion_areas,
            'message': 'Excessive ground movement detected'
        })

    return alerts
```

🔄 PHASE 8: Complete Integration Pipeline

8.1 End-to-End Workflow

```
python
```

```python
def complete_slope_monitoring_pipeline(drone_images, previous_model=None):
    """Complete pipeline from drone images to FOS calculation"""

    # Phase 1: 3D Reconstruction
    print("Building 3D model...")
    model_3d = build_photogrammetric_model(drone_images)

    # Phase 2: AI Feature Detection
    print("Detecting features with AI...")
    cracks = detect_cracks_sam(model_3d.texture_images)
    joints = analyze_joint_sets_3d(model_3d.point_cloud)

    if previous_model:
        motion = detect_motion_multitemporal(previous_model, model_3d)
    else:
        motion = None

    # Phase 3: Geotechnical Parameter Extraction
    print("Extracting geotechnical parameters...")
    geo_params = extract_geotechnical_parameters(cracks, joints, motion)
    strength_params = calculate_strength_parameters(geo_params)

    # Phase 4: FOS Calculation
    print("Calculating Factor of Safety...")
    fos_distribution = calculate_3d_fos_distribution(
        model_3d, joints, strength_params
    )

    # Phase 5: Create Stability Map
    print("Generating stability map...")
    stability_map = create_2d_stability_map(
        model_3d.coordinates, fos_distribution
    )

    # Phase 6: Risk Assessment & Alerts
    risk_assessment = assess_overall_risk(
        fos_distribution, cracks, motion
    )

    return {
        '3d_model': model_3d,
        'detected_features': {
            'cracks': cracks,
```

```
            'joints': joints,
            'motion': motion
        },
        'geotechnical_params': geo_params,
        'fos_distribution': fos_distribution,
        'stability_map': stability_map,
        'risk_assessment': risk_assessment
    }
```

This complete pipeline takes you from raw drone images all the way to actionable slope stability insights with automated FOS calculations and risk mapping!