Appendix A: Computational Implementation

The following sections present the Python code used for the coupled hydro-geophysical finite-element modeling, implemented with the pyGIMLi library (Rücker et al., 2017). Each part includes a brief description of the code's purpose and functionality, with key parameters retained as in the original simulation.

A.1 Primary Geometry Design and Mesh Generation

A 2D finite-element mesh was generated to represent the modeling domain, consisting of a world domain with layered structures and a mineralized block mimicking a porphyry copper deposit. The mesh was created with specified quality and area constraints to ensure sufficient resolution for subsequent flow and transport simulations.

```
import numpy as np
import pygimli as pg
import pygimli.meshtools as mt
import pygimli.physics.ert as ert
import pygimli.physics.petro as petro
from pygimli.physics import ERTManager
import matplotlib.pyplot as plt

 #Create geometry definition for the modeling domain
world = mt.createWorld(start=[-24, 0], end=[24, -19.2], layers=[-2.4, -9.6], worldMarker=False)

 #Create a mineralized block (porphyry copper deposit)
block = mt.createRectangle(start=[-7.2, -4.2], end=[7.2, -7.2], marker=4, boundaryMarker=10,
area=0.1)

 #Merge geometrical entities
geom = world + block

 #Create a mesh from the geometry definition
mesh = mt.createMesh(geom, quality=32, area=0.2, smooth=[1, 10])

 #Calculate and print min, max, and mean cell sizes
cell_areas = mesh.cellSizes()  # Get cell areas in m²
cell_sizes = np.sqrt(cell_areas)  # Characteristic cell size as sqrt(area)
print(f"Minimum cell size: {np.min(cell_sizes):.2e} m")
print(f"Maximum cell size: {np.max(cell_sizes):.2e} m")
print(f"Mean cell size: {np.mean(cell_sizes):.2e} m")
print(f"Forward mesh: Nodes: {mesh.nodeCount()}, Cells: {mesh.cellCount()}")
```

A.2 Mesh Quality Analysis

Mesh quality was assessed to ensure numerical accuracy in the finite-element simulations. The quality metric, based on the ratio of triangle incircle to circumcircle radii, was computed for each cell, with statistics calculated to evaluate the mesh's suitability for modeling flow and ion transport.

```
 #Calculate mesh quality
```

```
qualities = pg.meshtools.quality(mesh)
 #Calculate statistics for mesh quality
min_quality = np.min(qualities)
max_quality = np.max(qualities)
mean_quality = np.mean(qualities)
print(f"Minimum mesh quality: {min_quality:.2f}")
print(f"Maximum mesh quality: {max_quality:.2f}")
print(f"Mean mesh quality: {mean_quality:.2f}")
```

A.3 Material Property Assignment

Hydraulic conductivity and electrical resistivity values were assigned to the mesh regions to represent the aquifer, host rock, basement, and mineralized zone. These properties were mapped to each cell based on predefined region markers, enabling accurate simulation of subsurface flow and electrical resistivity tomography.

```
# Map regions to hydraulic conductivity in m/s
kMap = [
    [1, 1e-5],  # Aquifer (0 to -2.4 m, high porosity, sandstone)
    [2, 1e-6],  # Host rock (-2.4 to -9.6 m, moderate fracturing, altered granodiorite)
    [3, 1e-8],  # Basement (-9.6 to -19.2 m, impermeable, granite)
    [4, 5e-6]   # Mineralized zone (-7.2 to 7.2 m, -4.2 to -7.2 m, fractured, sulfide-rich)
]
K = pg.solver.parseMapToCellArray(kMap, mesh)

# Map regions to resistivity in Ω·m
rhoMap = [
    [1, 50],    # Aquifer (0 to -2.4 m, high porosity, water-saturated)
    [2, 500],   # Host rock (-2.4 to -9.6 m, moderate fracturing)
    [3, 2000],  # Basement (-9.6 to -19.2 m, impermeable, highly resistive)
    [4, 10]     # Mineralized zone (-7.2 to 7.2 m, -4.2 to -7.2 m, fractured, sulfide-rich)
]
rho = pg.solver.parseMapToCellArray(rhoMap, mesh)
```

A.3 Material Property Assignment

Hydraulic conductivity and electrical resistivity values were assigned to the mesh regions to represent the aquifer, host rock, basement, and mineralized zone. These properties were mapped to each cell based on predefined region markers, enabling accurate simulation of subsurface flow and electrical resistivity tomography.

```
# Dirichlet conditions for hydraulic potential
left = 20
right = 0
pBound = {"Dirichlet": {1: left, 2: left, 3: left, 4: right, 5: right, 6: right}}

# Solve for hydraulic potential
p = pg.solver.solveFiniteElements(mesh, a=K, bc=pBound)

# Solve velocity as gradient of hydraulic potential
vel = -pg.solver.grad(mesh, p) * np.asarray([K, K, K]).T
```

```
# Calculate and print min, max, and mean velocity magnitudes in mm/s
vel_magnitude = pg.abs(vel) * 1000  # Convert to mm/s
print(f"Minimum velocity: {np.min(vel_magnitude):.2e} mm/s")
print(f"Maximum velocity: {np.max(vel_magnitude):.2e} mm/s")
print(f"Mean velocity: {np.mean(vel_magnitude):.2e} mm/s")

# Initialize injection source vector
S = pg.Vector(mesh.cellCount(), 0.0)

# Fill injection source vector for a fixed injection position
source_pos = [-22.8, -5.4]  # Near mineralized zone (porphyry copper deposit)
sourceCell = mesh.findCell(source_pos)
if sourceCell is None:
    raise ValueError(f"No cell found at injection point {source_pos}")
S[sourceCell.id()] = 1.0 / sourceCell.size()  # Source strength: 1.0 g/(l s), normalized by cell size

# Set up advection-diffusion simulation parameters
time = 250 * 24 * 3600  # 250 days in seconds
n_steps = 50000  # Number of time steps
t = pg.utils.grange(0, time, n=n_steps)  # Time array
dispersivity = 1e-2  # Dispersivity (m)
dispersion = pg.abs(vel) * dispersivity

# Solve for injection time (first phase), using velocities on cell nodes
veln = mt.cellDataToNodeData(mesh, vel)
c1 = pg.solver.solveFiniteVolume(
    mesh,
    a=dispersion,
f=S,
    vel=veln,
    times=t,
    scheme='PS',
    verbose=True
)

# Solve without injection (second phase), starting with last result
c2 = pg.solver.solveFiniteVolume(
    mesh,
    a=dispersion,
    f=0,
    vel=veln,
    u0=c1[-1],
    times=t,
    scheme='PS',
    verbose=True
)

# Stack results together
c = np.vstack((c1, c2))
```

A.5 ERT Forward Modeling

An electrical resistivity tomography (ERT) forward model was constructed using a dipole-dipole measurement scheme. Fluid conductivity was derived from ion concentrations using an empirical scaling, and bulk resistivity was calculated via Archie's Law. The ERT simulation was performed for 18 time frames to capture the temporal evolution of resistivity distributions.

```
# Create dipole-dipole ERT scheme
electrode_positions = np.arange(-24, 24.1, 1.2)  # 1.2 m spacing
scheme = ert.createData(elecs=electrode_positions, schemeName='dd')

# Create ERT mesh
meshERT = mt.createParaMesh(scheme, quality=33, paraMaxCellSize=0.2, boundaryMaxCellSize=50, smooth=[1, 2])

# Select 18 time frames for ERT simulation
timesERT = np.array(np.linspace(0, len(c) - 1, 18, dtype=int))

# Calculate fluid conductivity from ion concentration (S/m)
sigmaFluid = c[timesERT] * 0.05 + 0.01  # Scaling: 0.05 S/m per g/l, background: 0.01 S/m

# Calculate bulk resistivity using Archie's Law
porosity = 0.3
m = 1.3  # Cementation exponent
resBulk = np.zeros((len(timesERT), mesh.cellCount()))
for i, sigma_f in enumerate(sigmaFluid):
    rho_fluid = 1. / sigma_f  # Fluid resistivity (Ω·m)
    resBulk[i] = rho_fluid * porosity ** (-m)  # Bulk resistivity (Ω·m)

# Define background resistivity model (Ω·m)
rho0 = np.zeros(mesh.cellCount())
for cell in mesh.cells():
    y = cell.center()[1]
    if y < -9.6:  # Basement
        rho0[cell.id()] = 2000
    elif y < -2.4:  # Host rock
        rho0[cell.id()] = 500.0
    else:  # Aquifer
        rho0[cell.id()] = 50.0
    if cell.marker() == 4:  # Mineralized zone
        rho0[cell.id()] = resBulk[0, cell.id()]  # Initial time frame

# Combine resistivities
resis = pg.Matrix(resBulk)

# Run ERT simulation
ERT = ert.ERTManager(verbose=False)
rhoa = ERT.simulate(mesh, res=resis, scheme=scheme, returnArray=True, verbose=False)
```

A.6 ERT Inversion

Time-lapse ERT data for four selected time frames were inverted using smoothness-constrained Tikhonov regularization. The inversion process incorporated error estimates and constrained resistivity values to ensure physically realistic models, with parameters optimized for the mineralized zone's response.

```
# Initialize ERT manager for inversion
ERT = ert.ERTManager(verbose=True)

# Perform inversion for each of the 4 time frames
inverted_models = []
for i, rhoa_i in enumerate(rhoa[:4]):  # Limit to 4 time frames
    # Create data container for inversion
    data = pg.DataContainerERT(scheme)
    data["rhoa"] = rhoa_i  # Set apparent resistivity values

    # Adjust error estimate
    relative_error = 0.05
    data["err"] = ert.estimateError(data, relativeError=relative_error, absoluteError=1e-4)
# Perform inversion with specified parameters
    inv_model = ERT.invert(zWeight=0.1, data=data, lam=5, paraMaxCellSize=1, paraDepth=25,
                           paraBoundary=0.1, minModel=1, maxModel=5000, verbose=True)
    inverted_models.append(inv_model)
```

A.7 Model Validation and Misfit Analysis

Predicted apparent resistivities were computed from the inverted resistivity models for four time frames, and relative misfits were calculated to assess the accuracy of the inversion results. The misfit, defined as the absolute difference between observed and predicted apparent resistivities relative to observed values, was quantified to evaluate model performance.

```
# Compute predicted apparent resistivities for 4 time frames
predicted_data = []
for i in range(4):
    inv_mesh = ERT.paraDomain  # Inversion mesh
    inv_res = inverted_models[i]  # Inverted resistivity values
    res_model = pg.solver.parseArgToArray(inv_res, inv_mesh.cellCount())  # Ensure correct
format
    predicted = ERT.simulate(inv_mesh, res=res_model, scheme=scheme, returnArray=True,
verbose=False)
    predicted_data.append(predicted)

# Compute relative misfit for each time frame
misfit_data = []
for i in range(4):
    observed = np.array(rhoa[i])
    predicted = np.array(predicted_data[i])
    epsilon = 1e-10  # Avoid division by zero
```

```
    relative_misfit = np.abs(observed - predicted) / np.maximum(np.abs(observed), epsilon)
    misfit_data.append(relative_misfit)
```

A.8 Inversion Mesh Quality Analysis

The quality of the inversion mesh was evaluated to ensure numerical reliability of the ERT inversion results. The quality metric, calculated as the ratio of four times the square root of three times the triangle area to the sum of squared edge lengths, was computed for each triangular cell to assess mesh suitability.

```
# Extract the inversion mesh
mesh = ERT.paraDomain

# Compute mesh quality for triangular elements
quality = np.zeros(mesh.cellCount())
for cell in mesh.cells():
    if cell.nodeCount() == 3:  # Ensure the cell is a triangle
        nodes = [cell.node(i).pos() for i in range(3)]
        l1 = np.sqrt((nodes[0][0] - nodes[1][0])**2 + (nodes[0][1] - nodes[1][1])**2)
        l2 = np.sqrt((nodes[1][0] - nodes[2][0])**2 + (nodes[1][1] - nodes[2][1])**2)
        l3 = np.sqrt((nodes[2][0] - nodes[0][0])**2 + (nodes[2][1] - nodes[0][1])**2)
        edge_sum_sq = l1**2 + l2**2 + l3**2
        area = cell.size()
        q = (4 * np.sqrt(3) * area) / edge_sum_sq if edge_sum_sq > 0 else 0
        quality[cell.id()] = min(max(q, 0), 1)  # Clamp between 0 and 1
    else:
        quality[cell.id()] = 0  # Non-triangular cells get quality 0

# Calculate statistics for mesh quality
min_quality = np.min(quality)
max_quality = np.max(quality)
mean_quality = np.mean(quality)
print(f"Minimum inversion mesh quality: {min_quality:.2f}")
print(f"Maximum inversion mesh quality: {max_quality:.2f}")
print(f"Mean inversion mesh quality: {mean_quality:.2f}")
```