Cost Function

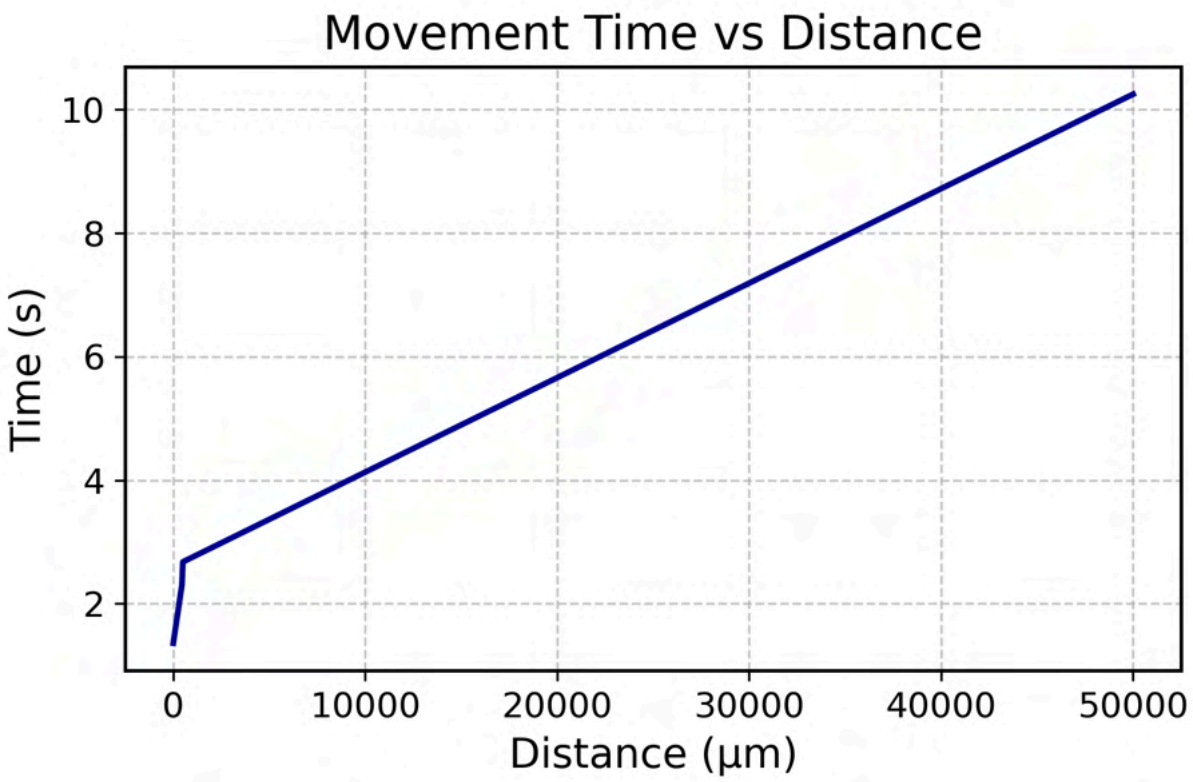
In [1]:
```python
import numpy as np
import pandas as pd

# === Constants ===
t_lift = 30
t_engage = 65
default_t_drift = 300
default_t_measurement = 120
default_t_lift = 25
t_comeback=35

# === Movement model coefficients ===
a, b, c_11, d, e = 4196.41, 4.99e-7, -4195.05, 1.53e-4, 2.60

# === Base movement time (1D, x or y) ===
def movement_time(distance):
    return np.piecewise(distance,
                        [distance < 500, distance >= 500],
                        [lambda x: a * np.exp(b * x) + c_11,
                         lambda x: d * x + e])

# === Cost for moving between indents (x and y), with penalty for long moves ===
def cost_function_indentation(dx, dy, t_drift=default_t_drift, t_lift=default_t_lif
    base_dx = movement_time(dx)
    base_dy = movement_time(dy)
    penalty = 0
    if dx >= 1000 or dy >= 1000:
        penalty = t_engage + t_drift + t_lift
    return base_dx + base_dy + penalty

# === Main cost function returning remaining costs after each indent ===
def evaluate_sample_cost_remaining(x_coords, y_coords, t_drift=default_t_drift, t_m
    assert len(x_coords) == len(y_coords), "Coordinate lists must be of equal lengt

    n = len(x_coords)
    setup_time = t_lift + t_engage + t_drift
    times_per_indent = []

    for i in range(n):
        time = t_measurement
        if i > 0:
            dx = abs(x_coords[i] - x_coords[i-1])
            dy = abs(y_coords[i] - y_coords[i-1])
            time += cost_function_indentation(dx, dy, t_drift)
        times_per_indent.append(time)

    results = [setup_time + sum(times_per_indent)]
    for i in range(1, n):
        results.append(sum(times_per_indent[i:]))
```

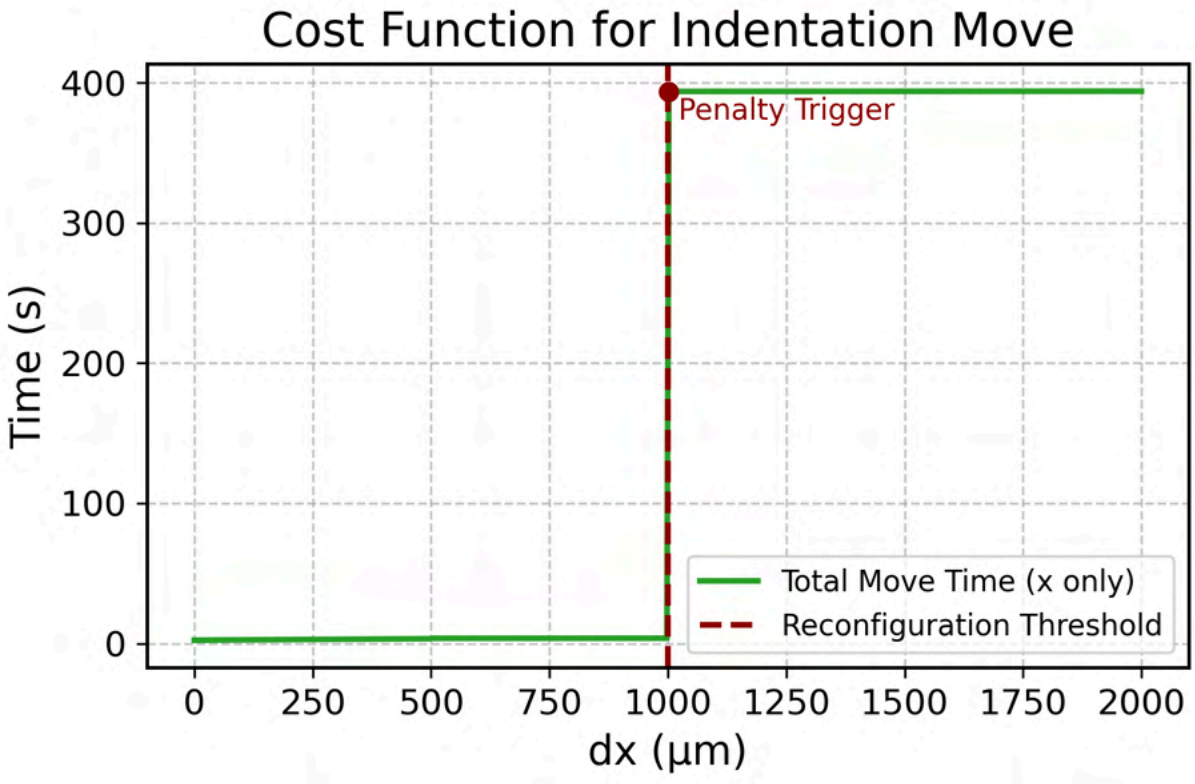
```
    return [r + t_comeback for r in results]
```

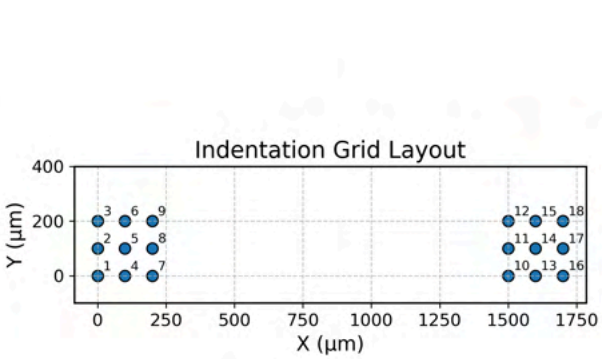
In [2]:

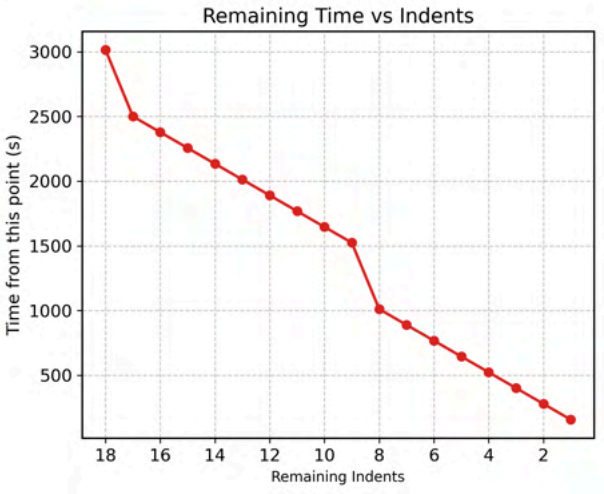
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

# === Set global styles ===
mpl.rcParams.update({
    "font.size": 12,
    "axes.labelsize": 14,
    "axes.titlesize": 16,
    "xtick.labelsize": 12,
    "ytick.labelsize": 12,
    "legend.fontsize": 12,
    "figure.dpi": 300,
    "axes.linewidth": 1.2,
    "lines.linewidth": 2
})

# === Plot 1: movement_time from 0 to 50000 µm ===
x_vals = np.linspace(0, 50000, 1000)
y_vals = movement_time(x_vals)

plt.figure(figsize=(6, 4))
plt.plot(x_vals, y_vals, label="Movement Time", color='darkblue')
plt.xlabel("Distance (µm)")
plt.ylabel("Time (s)")
plt.title("Movement Time vs Distance")
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.savefig("movement_time_vs_distance.png", bbox_inches='tight')
plt.show()

# === Plot 2: cost_function_indentation with reconfiguration penalty marker ===
d_vals = np.linspace(0, 2000, 500)
penalty_times = [cost_function_indentation(d, 0) for d in d_vals]
penalty_point = next(i for i, d in enumerate(d_vals) if d >= 1000)

plt.figure(figsize=(6, 4))
plt.plot(d_vals, penalty_times, label="Total Move Time (x only)", color='tab:green'
plt.axvline(x=1000, color='darkred', linestyle='--', label='Reconfiguration Thresho
plt.scatter(d_vals[penalty_point], penalty_times[penalty_point], color='darkred', z
plt.text(d_vals[penalty_point]+ 20, penalty_times[penalty_point]- 20,
         "Penalty Trigger", color='darkred', fontsize=10)
plt.xlabel("dx (µm)")
plt.ylabel("Time (s)")
plt.title("Cost Function for Indentation Move")
plt.legend(loc='lower right', fontsize=10)
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.savefig("cost_function_penalty.png", bbox_inches='tight')
plt.show()

# === Grid and Remaining Cost Plot ===
x = [i * 100 for i in range(3) for j in range(3)]
```

```
y = [j * 100 for i in range(3) for j in range(3)]
x += [i * 100 + 1500 for i in range(3) for j in range(3)]
y += [j * 100 for i in range(3) for j in range(3)]

remaining_costs = evaluate_sample_cost_remaining(x, y)

fig, axs = plt.subplots(1, 2, figsize=(12, 5))

# Subplot 1: Indentation grid layout
axs[0].scatter(x, y, c='tab:blue', s=60, edgecolors='black')
for i, (xi, yi) in enumerate(zip(x, y)):
    axs[0].text(xi + 20, yi + 20, str(i+1), fontsize=9)
axs[0].set_title("Indentation Grid Layout")
axs[0].set_xlabel("X (µm)")
axs[0].set_ylabel("Y (µm)")
axs[0].grid(True, linestyle='--', alpha=0.7)
axs[0].set_aspect('equal')
axs[0].set_ylim(-100, 400)

# Subplot 2: Remaining time
# --- Subplot 2: Remaining time vs indents ---
axs[1].plot(range(len(remaining_costs), 0, -1), remaining_costs, marker='o', color=
axs[1].set_title("Remaining Time vs Indents", fontsize=14)
axs[1].set_xlabel("Remaining Indents", fontsize=10)
axs[1].set_ylabel("Time from this point (s)", fontsize=12)
axs[1].invert_xaxis()
axs[1].xaxis.set_major_locator(plt.MaxNLocator(integer=True))
axs[1].grid(True, linestyle='--', alpha=0.7)


plt.tight_layout()
plt.savefig("indentation_grid_and_cost.png", bbox_inches='tight')
plt.show()
```

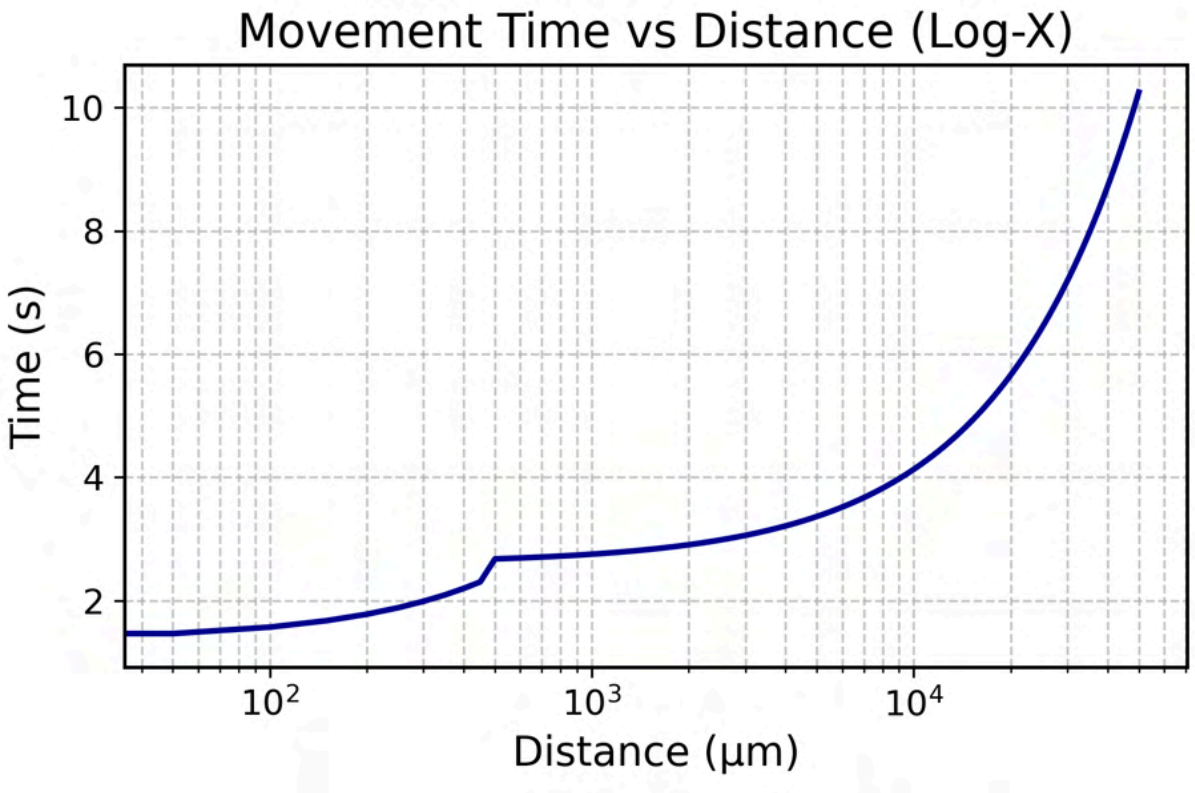

Movement Time vs Distance
Time (s)
Distance (µm)
0
10000
20000
30000
40000
50000
2
4
6
8
10


Cost Function for Indentation Move
Time (s)
dx (µm)
Penalty Trigger
Total Move Time (x only)
Reconfiguration Threshold
0
100
200
300
400
0
250
500
750
1000
1250
1500
1750
2000



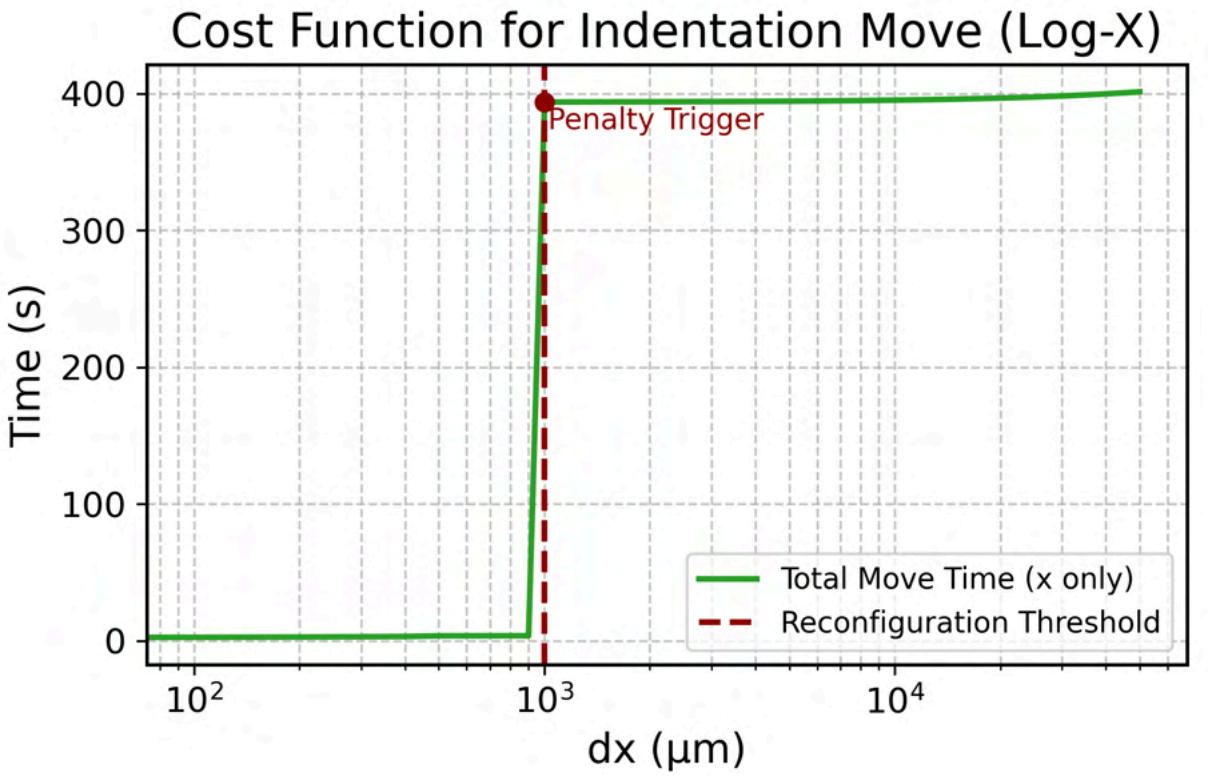


In [3]:
```
plt.figure(figsize=(6, 4))
plt.semilogx(x_vals, y_vals, label="Movement Time", color='darkblue')
plt.xlabel("Distance (µm)")
plt.ylabel("Time (s)")
plt.title("Movement Time vs Distance (Log-X)")
plt.grid(True, which='both', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.savefig("movement_time_vs_distance_semilogx.png", bbox_inches='tight')
plt.show()
```



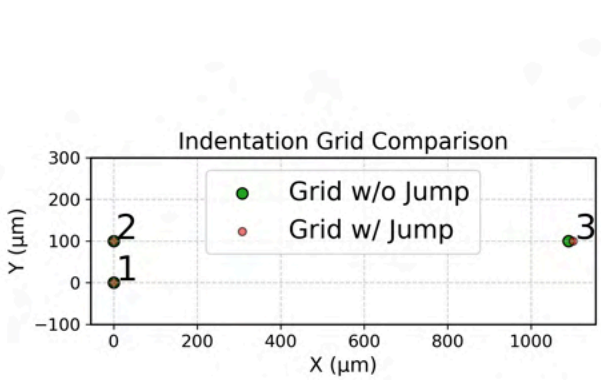
In [4]:
```
d_vals = np.linspace(0, 50000, 500)
penalty_times = [cost_function_indentation(d, 0) for d in d_vals]
penalty_point = next(i for i, d in enumerate(d_vals) if d >= 1000)

plt.figure(figsize=(6, 4))
plt.semilogx(d_vals, penalty_times, label="Total Move Time (x only)", color='tab:gr
```

```
plt.axvline(x=1000, color='darkred', linestyle='--', label='Reconfiguration Thresho
plt.scatter(d_vals[penalty_point], penalty_times[penalty_point], color='darkred', z
plt.text(d_vals[penalty_point] + 20, penalty_times[penalty_point] - 20,
         "Penalty Trigger", color='darkred', fontsize=10)
plt.xlabel("dx (µm)")
plt.ylabel("Time (s)")
plt.title("Cost Function for Indentation Move (Log-X)")
plt.legend(loc='lower right', fontsize=10)
plt.grid(True, which='both', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.savefig("cost_function_penalty_logx.png", bbox_inches='tight')
plt.show()
```

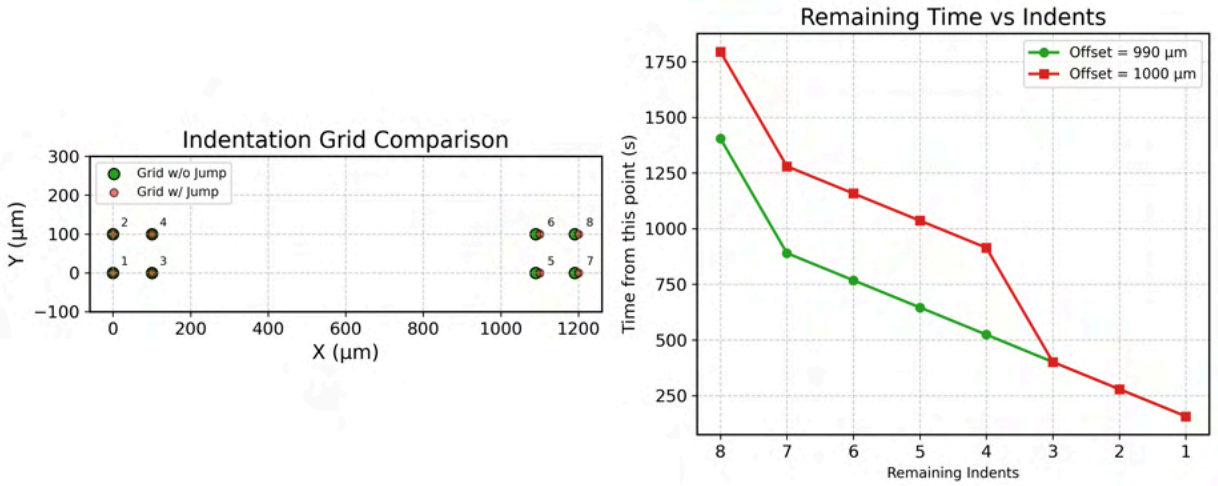


In [5]:
```
import matplotlib.pyplot as plt
import numpy as np

# First 3x3 grid
x1 = [i * 100 for i in range(1) for j in range(2)]
y1 = [j * 100 for i in range(1) for j in range(2)]

# Second 3x3 grid at 990 µm offset (no reconfiguration)
x2_no_jump = [i * 100 + 1090 for i in range(1) for j in range(1)]
y2 = [100+ j * 100 for i in range(1) for j in range(1)]

# Second 3x3 grid at 1000 µm offset (with reconfiguration)
x2_jump = [i * 100 + 1100 for i in range(1) for j in range(1)]

# Full paths
x_no_jump = x1 + x2_no_jump
x_jump = x1 + x2_jump
y_full = y1 + y2
```

```
# Evaluate costs
costs_no_jump = evaluate_sample_cost_remaining(x_no_jump, y_full)
costs_jump = evaluate_sample_cost_remaining(x_jump, y_full)

costs_no_jump[0]=costs_no_jump[0]-400
costs_no_jump[1]=costs_no_jump[1]-400
costs_no_jump[2]=costs_no_jump[2]-400

# Plot comparison
fig, axs = plt.subplots(1, 2, figsize=(12, 5))

# Subplot 1: Grid layout with labels
axs[0].scatter(x_no_jump, y_full, c='tab:green', label='Grid w/o Jump', s=60, edgec
axs[0].scatter(x_jump, y_full, c='tab:red', label='Grid w/ Jump', s=30, edgecolors=
for i, (xi, yi) in enumerate(zip(x_jump, y_full)):
    axs[0].text(xi + 5, yi + 5, str(i+1), fontsize=24)
axs[0].set_title("Indentation Grid Comparison")
axs[0].set_xlabel("X (µm)")
axs[0].set_ylabel("Y (µm)")
axs[0].set_aspect('equal')
axs[0].set_ylim(-100, 300)
axs[0].grid(True, linestyle='--', alpha=0.6)
axs[0].legend(loc='upper center', fontsize=18)

# Subplot 2: Remaining time curves

axs[1].plot(range( len(costs_no_jump),-1, -1),np.append(costs_no_jump, 0) , marker=
axs[1].plot(range(len(costs_jump) ,-1, -1), np.append(costs_jump, 0) , marker='s',
axs[1].set_title("Remaining Time vs Indents")
axs[1].set_xlabel("Indents", fontsize=10)
axs[1].set_ylabel("Time from this point (s)", fontsize=18)
axs[1].invert_xaxis()
axs[1].xaxis.set_major_locator(plt.MaxNLocator(integer=True))
axs[1].grid(True, linestyle='--', alpha=0.6)
axs[1].legend(loc='upper right', fontsize=18)

plt.tight_layout()
plt.savefig("indentation_jump_vs_nojump.png", bbox_inches='tight')
plt.show()
```



In [7]:
```
import matplotlib.pyplot as plt
import numpy as np

# First 3x3 grid
x1 = [i * 100 for i in range(2) for j in range(2)]
y1 = [j * 100 for i in range(2) for j in range(2)]

# Second 3x3 grid at 990 µm offset (no reconfiguration)
x2_no_jump = [i * 100 + 1090 for i in range(2) for j in range(2)]
y2 = [j * 100 for i in range(2) for j in range(2)]

# Second 3x3 grid at 1000 µm offset (with reconfiguration)
x2_jump = [i * 100 + 1100 for i in range(2) for j in range(2)]

# Full paths
x_no_jump = x1 + x2_no_jump
x_jump = x1 + x2_jump
y_full = y1 + y2

# Evaluate costs
costs_no_jump = evaluate_sample_cost_remaining(x_no_jump, y_full)
costs_jump = evaluate_sample_cost_remaining(x_jump, y_full)

# Plot comparison
fig, axs = plt.subplots(1, 2, figsize=(12, 5))

# Subplot 1: Grid layout with labels
axs[0].scatter(x_no_jump, y_full, c='tab:green', label='Grid w/o Jump', s=60, edgec
axs[0].scatter(x_jump, y_full, c='tab:red', label='Grid w/ Jump', s=30, edgecolors=
for i, (xi, yi) in enumerate(zip(x_jump, y_full)):
    axs[0].text(xi + 20, yi + 20, str(i+1), fontsize=8)
axs[0].set_title("Indentation Grid Comparison")
axs[0].set_xlabel("X (µm)")
axs[0].set_ylabel("Y (µm)")
axs[0].set_aspect('equal')
axs[0].set_ylim(-100, 300)
axs[0].grid(True, linestyle='--', alpha=0.6)
axs[0].legend(loc='upper left', fontsize=9)

# Subplot 2: Remaining time curves
axs[1].plot(range(len(costs_no_jump), 0, -1), costs_no_jump, marker='o', color='tab
axs[1].plot(range(len(costs_jump), 0, -1), costs_jump, marker='s', color='tab:red',
axs[1].set_title("Remaining Time vs Indents")
axs[1].set_xlabel("Remaining Indents", fontsize=10)
axs[1].set_ylabel("Time from this point (s)", fontsize=12)
axs[1].invert_xaxis()
axs[1].xaxis.set_major_locator(plt.MaxNLocator(integer=True))
axs[1].grid(True, linestyle='--', alpha=0.6)
axs[1].legend(loc='upper right', fontsize=10)

plt.tight_layout()
plt.savefig("indentation_jump_vs_nojump.png", bbox_inches='tight')
plt.show()
```



Defining GP

In [12]:
```
import torch
import gpytorch
from gpytorch.models import ApproximateGP
from gpytorch.variational import VariationalStrategy, CholeskyVariationalDistributi
from gpytorch.likelihoods import GaussianLikelihood
from gpytorch.mlls import VariationalELBO

class viGP(ApproximateGP):
    def __init__(self, inducing_points, mean_module=None, kernel_module=None):
        variational_distribution = CholeskyVariationalDistribution(inducing_points.
        variational_strategy = VariationalStrategy(
            self, inducing_points, variational_distribution, learn_inducing_locatio
        )
        super().__init__(variational_strategy)

        self.mean_module = mean_module or gpytorch.means.ConstantMean()
        self.covar_module = kernel_module or gpytorch.kernels.ScaleKernel(gpytorch.
        self.likelihood = GaussianLikelihood()

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)

    def fit(self, train_x, train_y, training_iter=500, lr=0.01):
        self.train()
        self.likelihood.train()

        optimizer = torch.optim.Adam(self.parameters(), lr=lr)
        mll = VariationalELBO(self.likelihood, self, train_y.shape[0])  #     Use sl

        for _ in range(training_iter):
            optimizer.zero_grad()
            output = self(train_x)
            loss = -mll(output, train_y)
            loss.backward()
```

```
            optimizer.step()

    def predict(self, test_x, noiseless=False, return_std=False):
        self.eval()
        self.likelihood.eval()
        with torch.no_grad(), gpytorch.settings.fast_pred_var():
            pred = self(test_x) if noiseless else self.likelihood(self(test_x))
        mean = pred.mean.cpu().numpy()  #    Convert to NumPy for safety
        if return_std:
            return mean, pred.stddev.cpu().numpy()  #    Convert std dev as well
        return mean

    def posterior(self, test_x):
        """Computes the full GP posterior (mean & covariance matrix)."""
        self.eval()
        self.likelihood.eval()
        with torch.no_grad(), gpytorch.settings.fast_pred_var():
            posterior = self.likelihood(self(test_x))  #    Includes likelihood no
        return posterior.mean, posterior.covariance_matrix
```

Vanilla Uncertainity based acquisition function

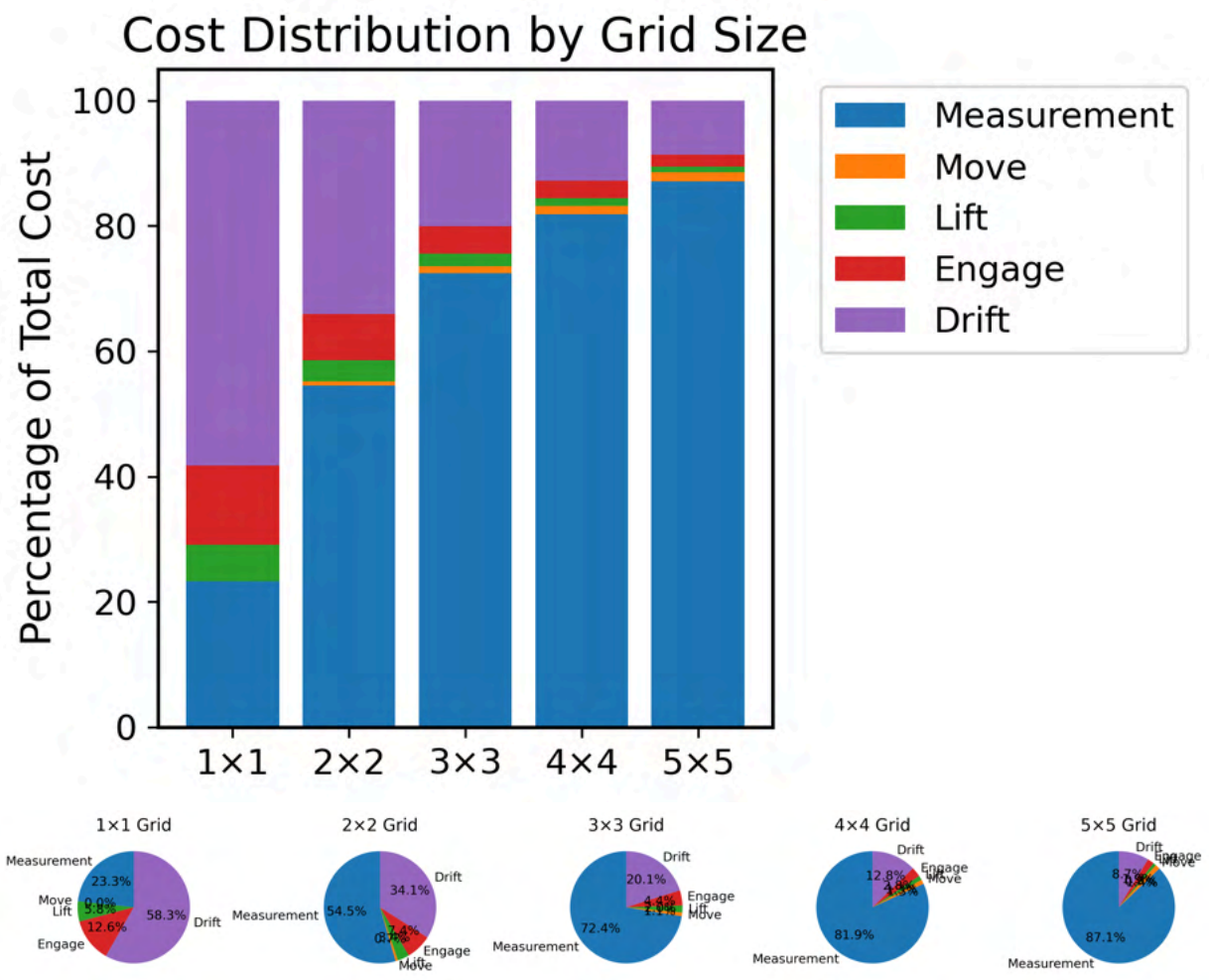
In [6]:
```
def UE(model, X):
    """Computes uncertainty (standard deviation) for given input X using GP posteri

    #    Compute posterior mean and variance
    posterior_mean, posterior_var = model.posterior(X)

    #    Ensure `posterior_var` is a diagonal variance, not a covariance matrix
    if posterior_var.dim() == 2 and posterior_var.shape[0] == posterior_var.shape[1
        posterior_var = posterior_var.diagonal()  # Extract only variances

    #    Compute uncertainty (standard deviation)
    uncertainty = posterior_var.sqrt()

    # print(f"    Posterior Mean Shape: {posterior_mean.shape}")
    # print(f"    Posterior Variance Shape: {posterior_var.shape}")
    # print(f"    Uncertainty Shape: {uncertainty.shape}")

    return uncertainty
```

Cost distribution analysis for measurement

In [10]:
```
import numpy as np
import matplotlib.pyplot as plt

# === Constants ===
t_lift = 30
t_engage = 65
default_t_drift = 300
```

```
default_t_measurement = 120
spacing = 5

# === Movement model ===
def movement_time(distance):
    a, b, c, d, e = 4196.41, 4.99e-7, -4195.05, 1.53e-4, 2.60
    return np.piecewise(distance,
                        [distance < 500, distance >= 500],
                        [lambda x: a * np.exp(b * x) + c,
                         lambda x: d * x + e])

# === Compute cost breakdown for grid sizes ===
grid_sizes = [1,2,3, 4, 5]
results = {}

for g in grid_sizes:
    xs, ys = np.meshgrid(np.arange(g) * spacing, np.arange(g) * spacing)
    xs, ys = xs.ravel(), ys.ravel()
    n = len(xs)

    meas_time = n * default_t_measurement
    lift_time = t_lift
    engage_time = t_engage
    drift_time = default_t_drift

    move_times = []
    for i in range(1, n):
        dx = abs(xs[i] - xs[i-1])
        dy = abs(ys[i] - ys[i-1])
        move_times.append(cost_function_indentation(dx, dy))
    total_move = np.sum(move_times)

    total = meas_time + lift_time + engage_time + drift_time + total_move

    results[g] = {
        'Measurement': meas_time / total * 100,
        'Move': total_move / total * 100,
        'Lift': lift_time / total * 100,
        'Engage': engage_time / total * 100,
        'Drift': drift_time / total * 100,
    }

# === Plot stacked bar chart ===
labels = ['Measurement', 'Move', 'Lift', 'Engage', 'Drift']
fig, ax = plt.subplots(figsize=(6, 4))
bottom = np.zeros(len(grid_sizes))

for label in labels:
    vals = [results[g][label] for g in grid_sizes]
    ax.bar([f'{g}×{g}' for g in grid_sizes], vals, bottom=bottom, label=label)
    bottom += vals

ax.set_ylabel('Percentage of Total Cost')
ax.set_title('Cost Distribution by Grid Size')
ax.legend(loc='upper left', bbox_to_anchor=(1.05, 1))  # legend outside
plt.tight_layout()
```

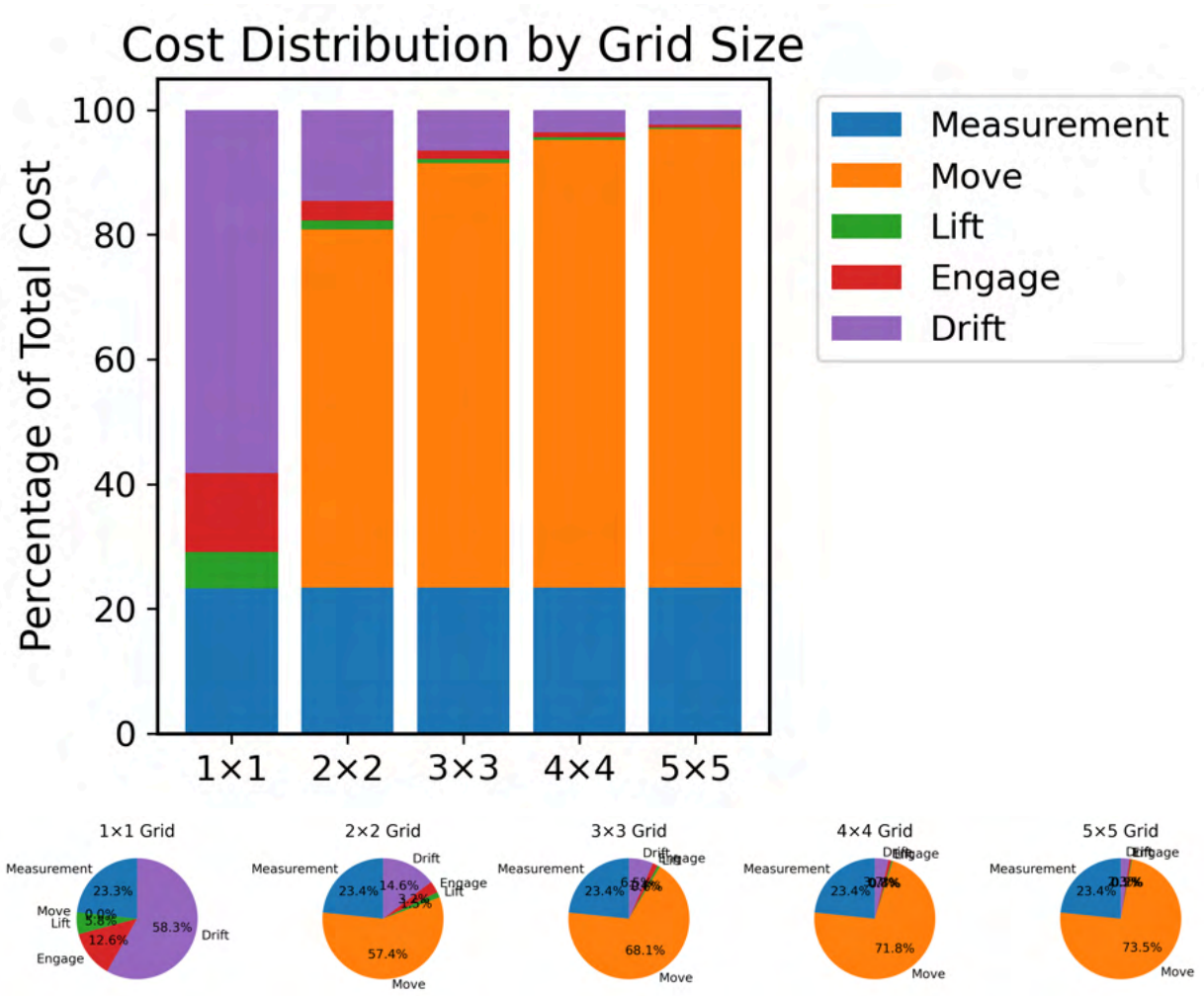
```
plt.show()

# === Plot pie charts ===
fig, axes = plt.subplots(1, len(grid_sizes), figsize=(16, 8))
for ax, g in zip(axes, grid_sizes):
    sizes = [results[g][l] for l in labels]
    ax.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90)
    ax.set_title(f'{g}×{g} Grid')

plt.tight_layout()
plt.show()
```



In [11]:

```
import numpy as np
import matplotlib.pyplot as plt

# === Constants ===
t_lift = 30
t_engage = 65
default_t_drift = 300
default_t_measurement = 120
spacing = 1100


# === Compute cost breakdown for grid sizes ===
grid_sizes = [1,2,3, 4, 5]
results = {}
```

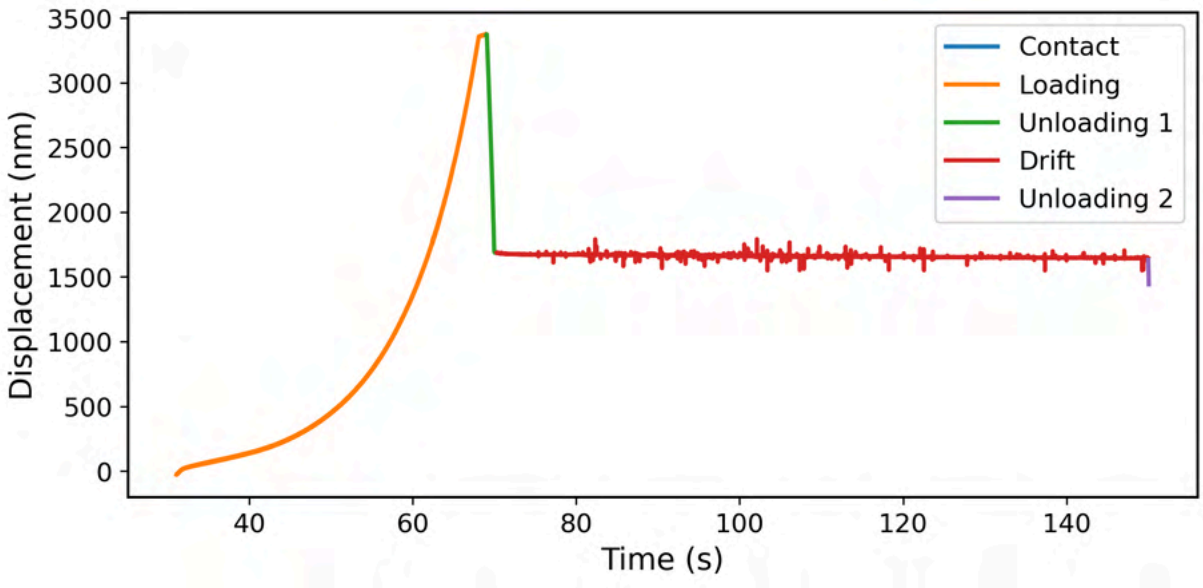
```
for g in grid_sizes:
    xs, ys = np.meshgrid(np.arange(g) * spacing, np.arange(g) * spacing)
    xs, ys = xs.ravel(), ys.ravel()
    n = len(xs)

    meas_time = n * default_t_measurement
    lift_time = t_lift
    engage_time = t_engage
    drift_time = default_t_drift

    move_times = []
    for i in range(1, n):
        dx = abs(xs[i] - xs[i-1])
        dy = abs(ys[i] - ys[i-1])
        move_times.append(cost_function_indentation(dx, dy))
    total_move = np.sum(move_times)

    total = meas_time + lift_time + engage_time + drift_time + total_move

    results[g] = {
        'Measurement': meas_time / total * 100,
        'Move': total_move / total * 100,
        'Lift': lift_time / total * 100,
        'Engage': engage_time / total * 100,
        'Drift': drift_time / total * 100,
    }

# === Plot stacked bar chart ===
labels = ['Measurement', 'Move', 'Lift', 'Engage', 'Drift']
fig, ax = plt.subplots(figsize=(6, 4))
bottom = np.zeros(len(grid_sizes))

for label in labels:
    vals = [results[g][label] for g in grid_sizes]
    ax.bar([f'{g}×{g}' for g in grid_sizes], vals, bottom=bottom, label=label)
    bottom += vals

ax.set_ylabel('Percentage of Total Cost')
ax.set_title('Cost Distribution by Grid Size')
ax.legend(loc='upper left', bbox_to_anchor=(1.05, 1))  # legend outside
plt.tight_layout()
plt.show()

# === Plot pie charts ===
fig, axes = plt.subplots(1, len(grid_sizes), figsize=(16, 8))
for ax, g in zip(axes, grid_sizes):
    sizes = [results[g][l] for l in labels]
    ax.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90)
    ax.set_title(f'{g}×{g} Grid')

plt.tight_layout()
plt.show()
```



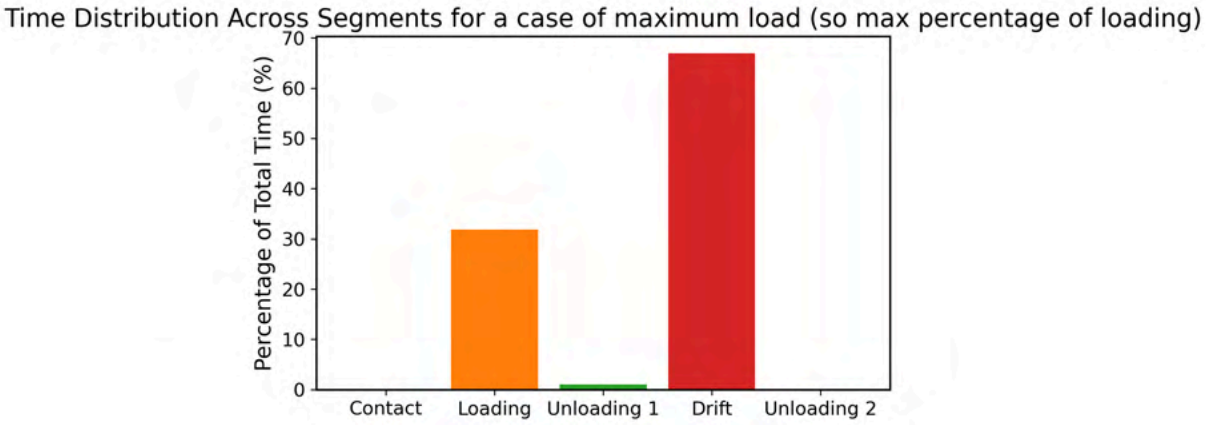
In [8]:
```
import pandas as pd
import matplotlib.pyplot as plt

# 1. Load data, skipping the blank second row
df = pd.read_excel('Book1.xlsx', skiprows=[1])

# 2. Define time boundaries (s)
boundaries = [31.0906149, 69.006935, 70.29, 149.98, 150.06]

# 3. Create masks for each segment
masks = [
    df['Time'] <= boundaries[0],
    (df['Time'] > boundaries[0]) & (df['Time'] <= boundaries[1]),
    (df['Time'] > boundaries[1]) & (df['Time'] <= boundaries[2]),
    (df['Time'] > boundaries[2]) & (df['Time'] <= boundaries[3]),
    (df['Time'] > boundaries[3]) & (df['Time'] <= boundaries[4]),
]
labels = ['Contact', 'Loading', 'Unloading 1', 'Drift', 'Unloading 2']
colors = ['C0', 'C1', 'C2', 'C3', 'C4']

# 4. Plot each segment
fig, ax = plt.subplots(figsize=(8, 4))

# Segment 1: baseline from t=0 to first boundary
# ax.plot([0, boundaries[0]], [0, 0], color=colors[0], label=labels[0])
```

```
for mask, label, color in zip(masks, labels, colors):
    print(color)
    ax.plot(df['Time'][mask], df['Displacement'][mask], color=color, label=label)

ax.set_xlabel('Time (s)')
ax.set_ylabel('Displacement (nm)')
ax.legend(loc='best')
plt.tight_layout()
plt.show()

# Compute and print percentage of total time per segment
total_time = boundaries[-1] - df['Time'].iloc[0]
durations = [(df['Time'][mask].max() - df['Time'][mask].min()) for mask in masks]
percentages = [dur / total_time * 100 for dur in durations]

for label, perc in zip(labels, percentages):
    print(f"{label}: {perc:.2f}%")

# Plot bar chart of percentages
# labels = ['Segment 1', 'Segment 2', 'Segment 3', 'Segment 4', 'Segment 5']
colors = ['C0', 'C1', 'C2', 'C3', 'C4']

fig, ax = plt.subplots(figsize=(6, 4))
ax.bar(labels, percentages, color=colors)
ax.set_ylabel('Percentage of Total Time (%)')
ax.set_title('Time Distribution Across Segments for a case of maximum load (so max
plt.tight_layout()
plt.show()
```

```
C0
C1
C2
C3
C4
```

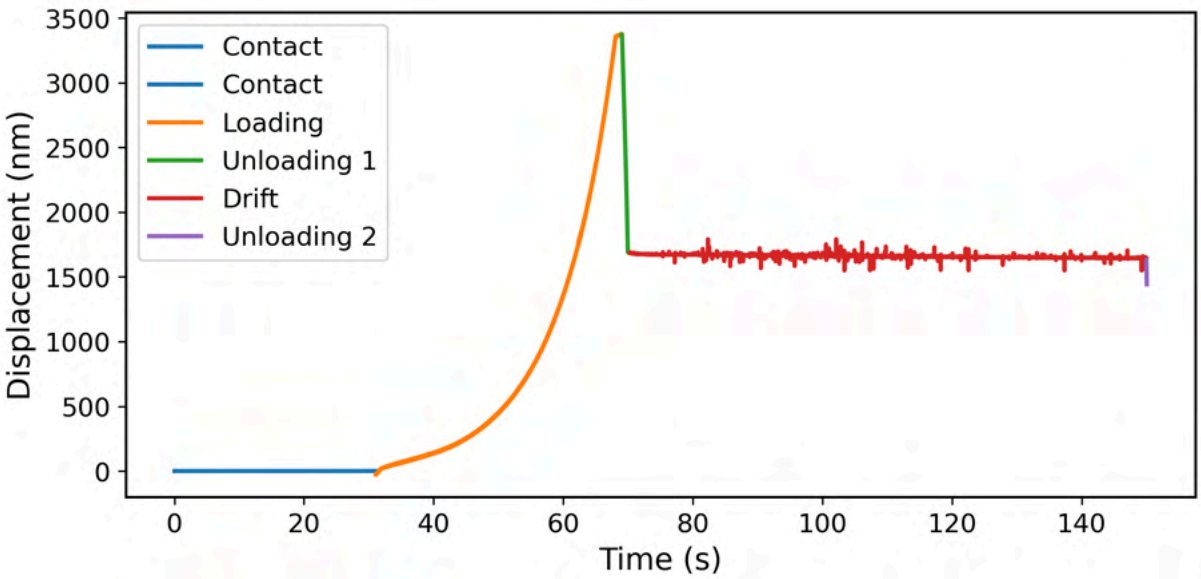


```
Contact: 0.00%
Loading: 31.86%
Unloading 1: 1.07%
Drift: 66.96%
Unloading 2: 0.06%
```

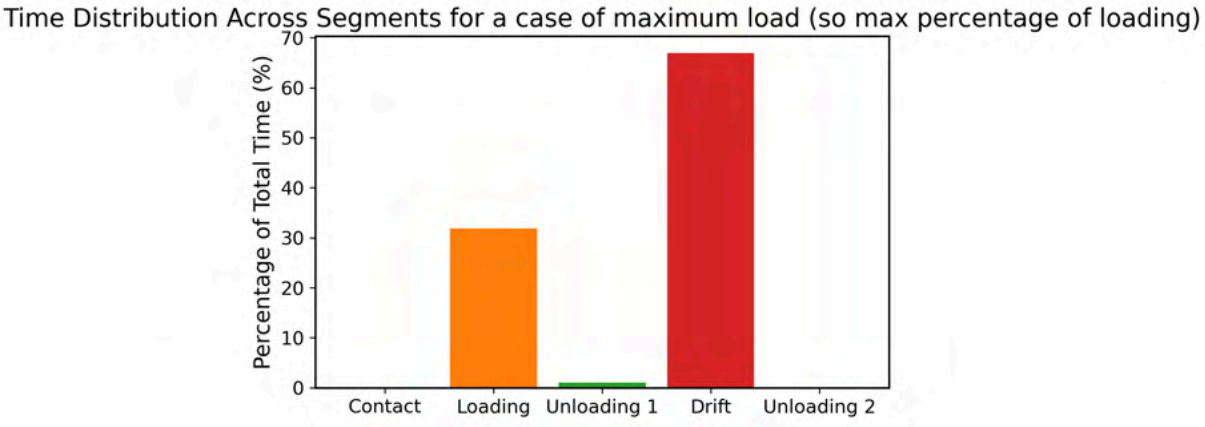


In [9]:
```python
import pandas as pd
import matplotlib.pyplot as plt

# 1. Load data, skipping the blank second row
df = pd.read_excel('Book1.xlsx', skiprows=[1])

# 2. Define time boundaries (s)
boundaries = [31.0906149, 69.006935, 70.29, 149.98, 150.06]

# 3. Create masks for each segment
masks = [
    df['Time'] <= boundaries[0],
    (df['Time'] > boundaries[0]) & (df['Time'] <= boundaries[1]),
    (df['Time'] > boundaries[1]) & (df['Time'] <= boundaries[2]),
    (df['Time'] > boundaries[2]) & (df['Time'] <= boundaries[3]),
    (df['Time'] > boundaries[3]) & (df['Time'] <= boundaries[4]),
]
labels = ['Contact', 'Loading', 'Unloading 1', 'Drift', 'Unloading 2']
colors = ['C0', 'C1', 'C2', 'C3', 'C4']

# 4. Plot each segment
fig, ax = plt.subplots(figsize=(8, 4))

# Segment 1: baseline from t=0 to first boundary
ax.plot([0, boundaries[0]], [0, 0], color=colors[0], label=labels[0])

for mask, label, color in zip(masks, labels, colors):
    ax.plot(df['Time'][mask], df['Displacement'][mask], color=color, label=label)

ax.set_xlabel('Time (s)')
ax.set_ylabel('Displacement (nm)')
ax.legend(loc='best')
plt.tight_layout()
plt.show()

# Compute and print percentage of total time per segment
total_time = boundaries[-1] - df['Time'].iloc[0]
```

```
durations = [(df['Time'][mask].max() - df['Time'][mask].min()) for mask in masks]
percentages = [dur / total_time * 100 for dur in durations]

for label, perc in zip(labels, percentages):
    print(f"{label}: {perc:.2f}%")

# Plot bar chart of percentages
# labels = ['Segment 1', 'Segment 2', 'Segment 3', 'Segment 4', 'Segment 5']
colors = ['C0', 'C1', 'C2', 'C3', 'C4']

fig, ax = plt.subplots(figsize=(6, 4))
ax.bar(labels, percentages, color=colors)
ax.set_ylabel('Percentage of Total Time (%)')
ax.set_title('Time Distribution Across Segments for a case of maximum load (so max
plt.tight_layout()
plt.show()
```




```
Contact: 0.00%
Loading: 31.86%
Unloading 1: 1.07%
Drift: 66.96%
Unloading 2: 0.06%
```



Dynamic Programming of drift

We aim to minimize idle drift-hold time while preserving data quality by dynamically choosing between no hold, a short hold (5 s), or a full hold (80 s) based on measured drift error and intra-grid variation

IMPORTANT!! Drift is not sample dependent but dependent upon surrounding

Key thresholds

- **T_var**: variation threshold = standard deviation of let say hardness measurements in the grid
- **T_big**: "big" threshold = maximum acceptable drift error based on calculation and user definition (e.g. 3 percent or 2* t_var whichever is larger)

Adaptive drift-hold algorithm

1. **Initialize**
   - Start each grid with a 5 s drift-hold.
   - Compute drift error: `drift_err_5 = total depth - drift_rate × test time (20 second)` .
   - Compute drift hardness error: `err_5 = 100* drift_err_5/total depth` # in percentage
   - Compute `T_var` from the measured variability in the 5×5 grid # percentage difference in grid.
2. **Decision branch**
   - **If** `err_5 ≤ T_var` → data quality is "good"
     - **Set** next drift hold to 5 s (5 s hold) -> Note this can be chanaged to see the effects of three hold time strategies
   - **Else if** `T_var < err_5 < T_big` → marginal drift
     - **Set** next drift hold to 5 s again.
   - **Else** (`err_5 ≥ T_big`) → unacceptable drift
     A. Switch to **80 s** drift-hold for the next grid.
     B. Measure both `err_80` and `err_5` on a smaller grid (e.g. shrink 5×5 → 3×3).
     C. **If** `|err_80 – err_5| ≤ ε` (negligible benefit) → revert to 5 s hold.
     D. **Else** keep 80 s hold until drift stabilizes.
3. **Update drift rate**
   - Optionally apply an exponential moving average to `drift_rate` across grids so the algorithm adapts to slow changes in instrument behavior.

This scheme ensures we only pay long drift-hold penalties when they meaningfully improve depth precision, and otherwise run as fast as the instrument noise and surface roughness allow.

Emulator 1 - No particle Size effect / Gradient in composition and not Random

```
Baseline linear contribution (rule-of-mixtures)
base = 3 * A + 5 * B + 7 * C  # GPa (e.g., A=soft, C=hard)

# Solid solution strengthening (how different are each composition)
ss_strength = -4 * A * B - 3 * B * C - 2 * C * A

# Saturation penalty for high C
saturation = -6 * C**2 * (1 - C)

# Total hardness
Hardness= base + ss_strength + saturation
```

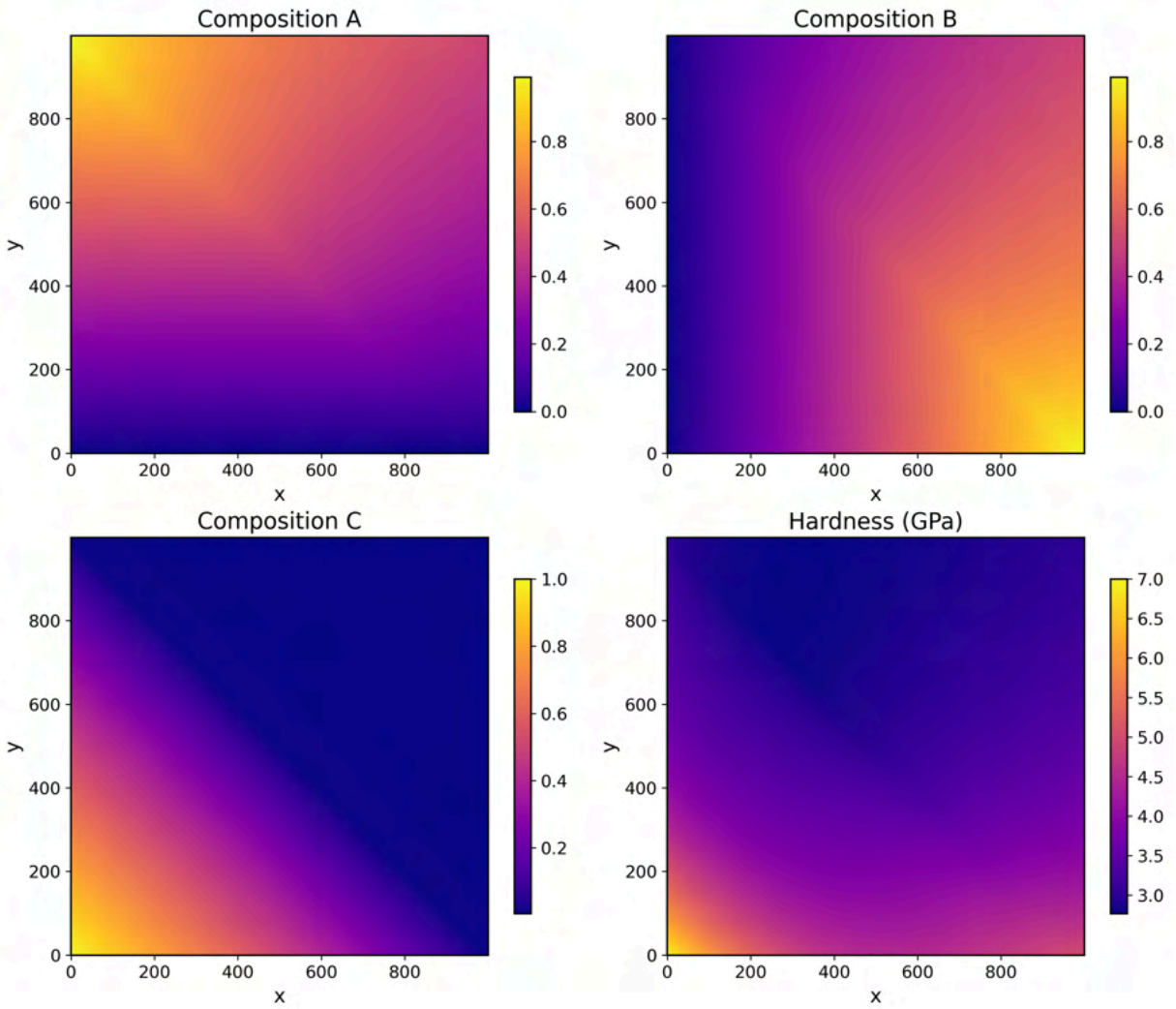
In [5]:
```
import numpy as np
import pandas as pd
def generate_ternary_gradient(nx=100, ny=100):
    """
    Generate a 2D ternary composition gradient over an nx × ny grid.
    A increases along x, B increases along y, C = 1 - A - B (clipped to ≥0).

    Returns:
        pd.DataFrame with columns: x, y, A, B, C
    """
    x_coords, y_coords = np.meshgrid(np.linspace(0, 1, nx), np.linspace(0, 1, ny))
    A = x_coords
    B = y_coords
    C = 1.0 - A - B

    # Clip to avoid negative C (outside triangle)
    A = A[C >= 0]
    B = B[C >= 0]
    C = C[C >= 0]
    x = x_coords[C >= 0]
    y = y_coords[C >= 0]

    return pd.DataFrame({
        "x": x.ravel(),
        "y": y.ravel(),
        "A": A.ravel(),
        "B": B.ravel(),
        "C": C.ravel()
    })


def hardness_model(row):
    A, B, C = row["A"], row["B"], row["C"]

    # Baseline linear contribution (rule-of-mixtures)
    base = 3 * A + 5 * B + 7 * C  # GPa (e.g., A=soft, C=hard)
```

```
    # Solid solution strengthening (how different are each composition)
    ss_strength = -4 * A * B - 3 * B * C - 2 * C * A

    # Saturation penalty for high C
    saturation = -6 * C**2 * (1 - C)

    # Total hardness
    return base + ss_strength + saturation
```

In [6]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# === 1. Generate 100x100 ternary composition grid with gradient ===
nx, ny = 1000, 1000

# Generate linearly varying A (x-direction) and B (y-direction)
x_coords, y_coords = np.meshgrid(np.linspace(0, 1, nx), np.linspace(0, 1, ny))
A = x_coords
B = y_coords
C = np.maximum(1.0 - A - B, 0.01)  # Avoid negative C

# Normalize so A + B + C = 1
total = A + B + C
A /= total
B /= total
C /= total

# Flatten to create DataFrame
composition_df = pd.DataFrame({
    "x": np.arange(nx).repeat(ny),
    "y": np.tile(np.arange(ny), nx),
    "A": A.ravel(),
    "B": B.ravel(),
    "C": C.ravel()
})



composition_df["Hardness"] = composition_df.apply(hardness_model, axis=1)

# === 3. Pivot data into 2D grids for plotting ===
A_grid = composition_df.pivot(index="y", columns="x", values="A")
B_grid = composition_df.pivot(index="y", columns="x", values="B")
C_grid = composition_df.pivot(index="y", columns="x", values="C")
H_grid = composition_df.pivot(index="y", columns="x", values="Hardness")

# === 4. Plot contour maps ===
fig, axs = plt.subplots(2, 2, figsize=(12, 10))

c1 = axs[0, 0].imshow(A_grid, origin='lower', cmap='plasma')
axs[0, 0].set_title("Composition A")
fig.colorbar(c1, ax=axs[0, 0], shrink=0.8)

c2 = axs[0, 1].imshow(B_grid, origin='lower', cmap='plasma')
```

```
axs[0, 1].set_title("Composition B")
fig.colorbar(c2, ax=axs[0, 1], shrink=0.8)

c3 = axs[1, 0].imshow(C_grid, origin='lower', cmap='plasma')
axs[1, 0].set_title("Composition C")
fig.colorbar(c3, ax=axs[1, 0], shrink=0.8)

c4 = axs[1, 1].imshow(H_grid, origin='lower', cmap='plasma')
axs[1, 1].set_title("Hardness (GPa)")
fig.colorbar(c4, ax=axs[1, 1], shrink=0.8)

for ax in axs.flat:
    ax.set_xlabel("x")
    ax.set_ylabel("y")

plt.tight_layout()
plt.savefig("ternary_gradient_library_contours.png", dpi=300, bbox_inches='tight')
plt.show()
```



Ground truth Generation

In [7]:
```python
# Define the measure function using the precomputed H_grid from the user's gradient
def measure_from_Hgrid(x, y, noise_std=0.0):
    """
    Measures the hardness from H_grid at (x, y) with optional Gaussian noise.
    Ensures x and y are within bounds of the H_grid.
    """
    x = int(np.clip(x, 0, H_grid.shape[1] - 1))
    y = int(np.clip(y, 0, H_grid.shape[0] - 1))
    hardness = H_grid.iloc[y, x]
    noise = np.random.normal(0, noise_std)
    return hardness + noise
```

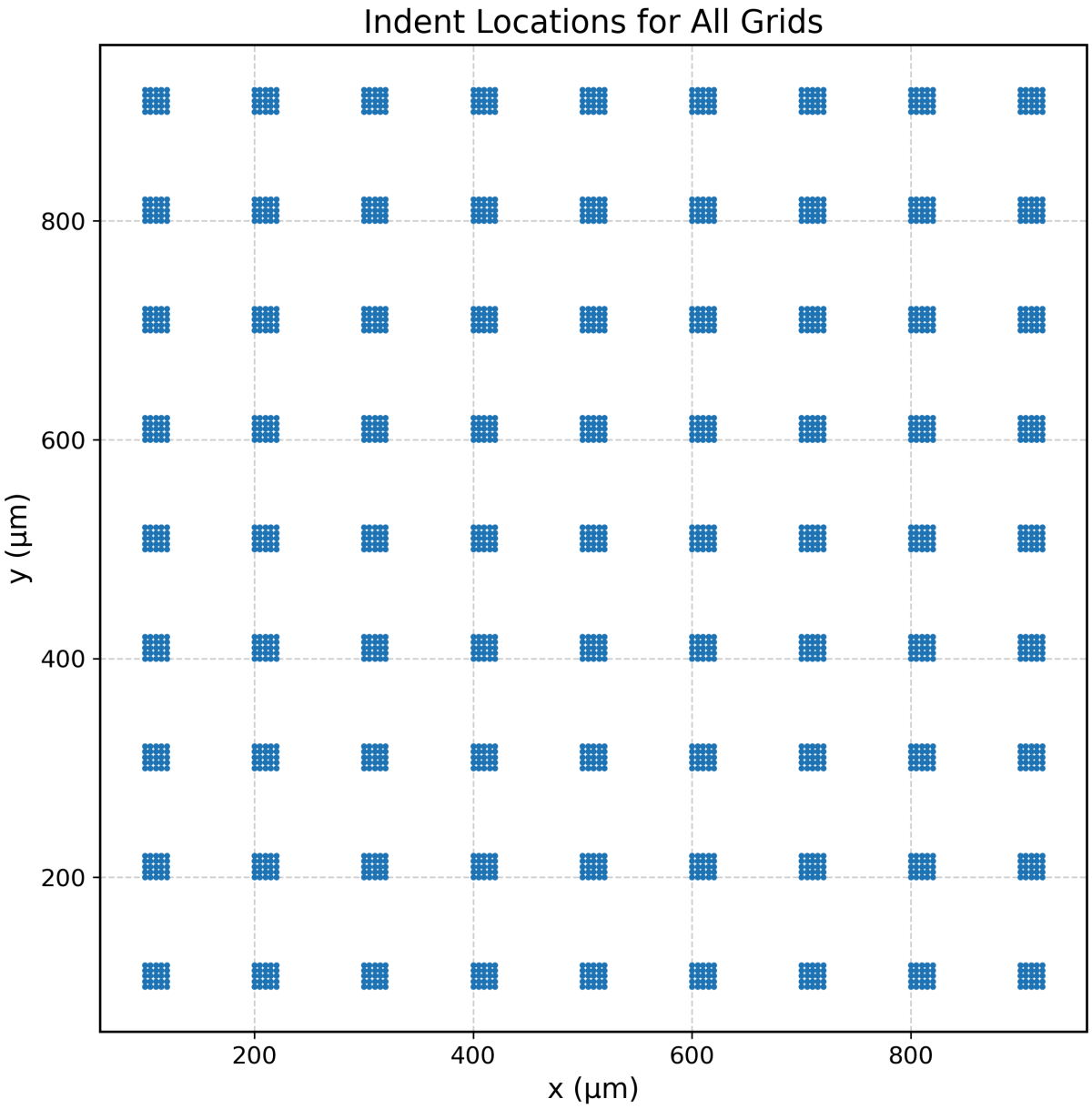
Constants

In [ ]:
```python
t_lift = 30
t_engage = 65
default_t_drift = 300
default_t_measurement = 120
default_t_lift = 25
t_comeback=35
domain_size = 1000
grids_per_dim = 9
grid_size = 5
spacing = 5
t_setup = 20
t_move = 20
```

Grid Search

In [18]:
```python
# Normal way Cost -> An 80 grid measurement

import os

import numpy as np
import matplotlib.pyplot as plt
import torch
from gpytorch.means import ConstantMean
from gpytorch.kernels import ScaleKernel, RBFKernel
from matplotlib import gridspec

# === Your GP model class must be pre-defined as viGP ===

# === Config ===
domain_size = 1000
grids_per_dim = 9
grid_size = 5
spacing = 5
t_setup = 20
t_move = 20

# Grid centers (exclude edge padding)
```

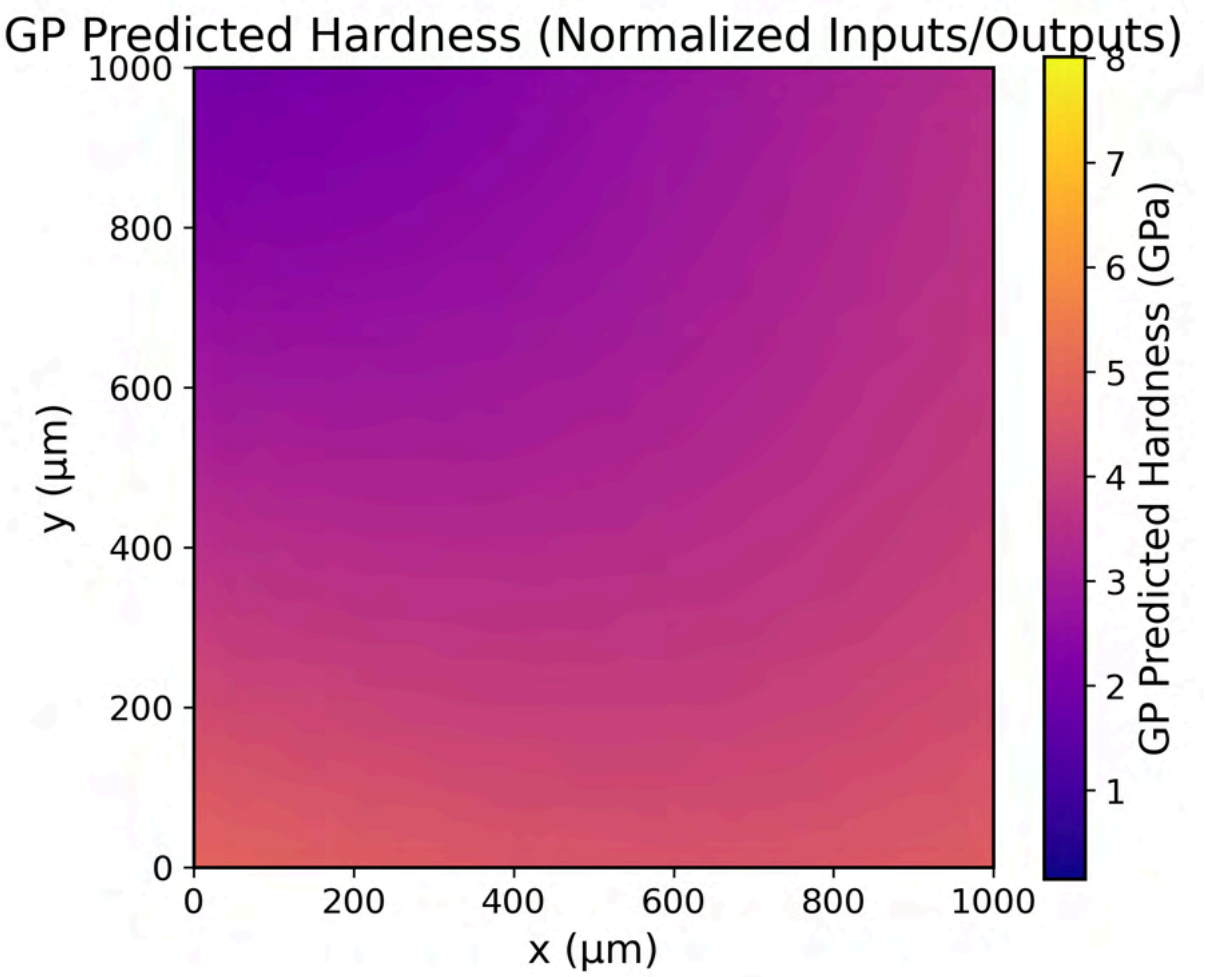
```
grid_centers_x = np.linspace(0, domain_size, grids_per_dim + 2)[1:-1]
grid_centers_y = np.linspace(0, domain_size, grids_per_dim + 2)[1:-1]
grid_centers = [(x, y) for x in grid_centers_x for y in grid_centers_y]

# Indent layout in local grid
ix, iy = np.meshgrid(np.arange(grid_size), np.arange(grid_size))
local_offsets = np.stack([ix.ravel(), iy.ravel()], axis=1) * spacing
# Plot all indent points as 'o' on a single x-y plot
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 8))

for gx, gy in grid_centers:
    positions = local_offsets + np.array([gx, gy])
    x_coords, y_coords = positions[:, 0], positions[:, 1]
    plt.plot(x_coords, y_coords, 'o', markersize=2, color='tab:blue')

plt.title("Indent Locations for All Grids")
plt.xlabel("x (µm)")
plt.ylabel("y (µm)")
plt.grid(True, linestyle='--', alpha=0.6)
plt.gca().set_aspect('equal', adjustable='box')
plt.tight_layout()
plt.savefig("all_indent_locations.png", dpi=300, bbox_inches='tight')
plt.show()
```



In [19]:
```python
import numpy as np
import matplotlib.pyplot as plt
import torch
import gpytorch
from gpytorch.means import ConstantMean
from gpytorch.kernels import ScaleKernel, MaternKernel

# === Assumptions ===
# - domain_size is defined (e.g., 1000)
# - measure_from_Hgrid(x, y, noise_std) is defined
# - viGP is defined

# 1) Sample 1000 random locations
n_samples = 100
xs = np.random.uniform(0, domain_size, n_samples)
ys = np.random.uniform(0, domain_size, n_samples)

# 2) Measure hardness (with optional noise)
```

```
zs = np.array([
    measure_from_Hgrid(int(x), int(y), noise_std=1.4)
    for x, y in zip(xs, ys)
])

# 3) Normalize inputs to [0,1]
X_raw = np.column_stack([xs, ys])
X_norm = X_raw / domain_size
train_x = torch.tensor(X_norm, dtype=torch.float32)

# 4) Normalize outputs to [0,1]
z_min, z_max = zs.min(), zs.max()
z_norm = (zs - z_min) / (z_max - z_min)
train_y = torch.tensor(z_norm, dtype=torch.float32)

# 5) Select inducing points
num_inducing = min(200, train_x.size(0))
inducing_points = train_x[:num_inducing]

# 6) Initialize GP with constant mean and Matern kernel
gp_model = viGP(
    inducing_points,
    mean_module=ConstantMean(),
    kernel_module=ScaleKernel(MaternKernel(nu=2.5))
)

# 7) Fit the GP on normalized data
gp_model.fit(train_x, train_y, training_iter=300, lr=0.01)

# 8) Create a normalized test grid for prediction
res = 100
xt = np.linspace(0, domain_size, res)
yt = np.linspace(0, domain_size, res)
XX, YY = np.meshgrid(xt, yt)
test_pts = np.column_stack([XX.ravel(), YY.ravel()])
test_norm = test_pts / domain_size
test_x = torch.tensor(test_norm, dtype=torch.float32)

# 9) Predict and de-normalize
pred_norm = gp_model.predict(test_x)  # predictions in [0,1]
pred = pred_norm * (z_max - z_min) + z_min
Z = pred.reshape(res, res)

# 10) Plot the de-normalized GP mean
plt.figure(figsize=(6, 5))
plt.imshow(
    Z,
    origin='lower',
    extent=(0, domain_size, 0, domain_size),
    vmin=z_min,
    vmax=z_max,
    cmap='plasma'
)
plt.colorbar(label="GP Predicted Hardness (GPa)")
plt.title("GP Predicted Hardness (Normalized Inputs/Outputs)")
plt.xlabel("x (µm)")
```

```
plt.ylabel("y (µm)")
plt.tight_layout()
plt.show()
```

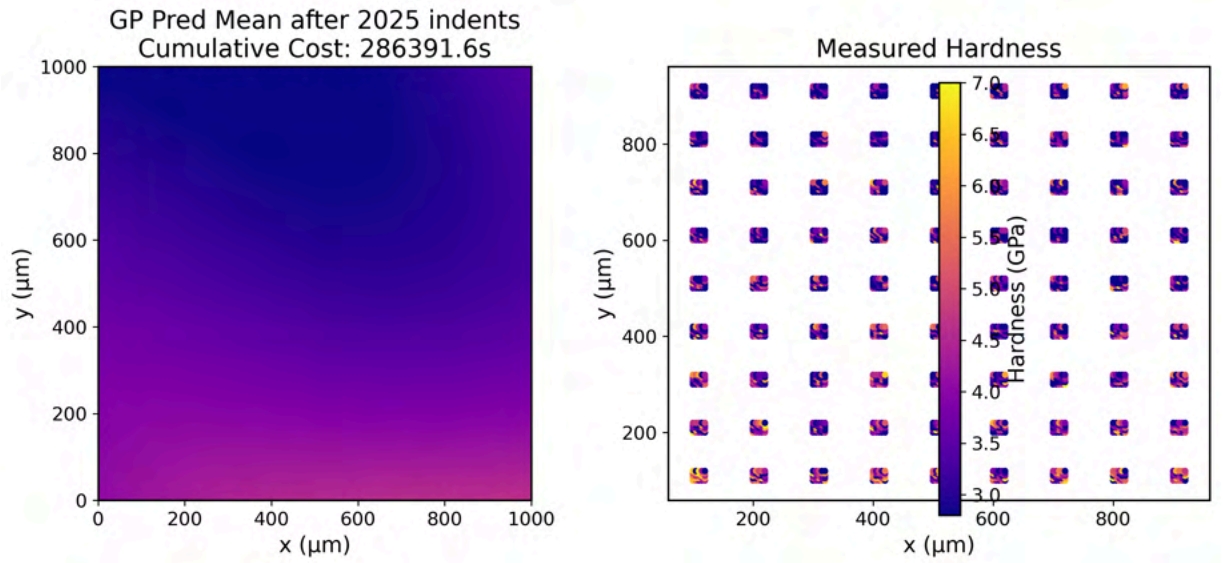


In [20]:
```
import numpy as np
import matplotlib.pyplot as plt
import torch
from gpytorch.means import ConstantMean
from gpytorch.kernels import ScaleKernel, RBFKernel

# === Assumes these are already defined/imported ===
# viGP
# measure_from_Hgrid(x, y, noise_std)
# evaluate_sample_cost_remaining(x_coords, y_coords)
# grid_centers  # list of (gx, gy) for each of the 81 grids
# local_offsets  # array of shape (25,2) for a 5×5 grid, in µm
# t_setup, t_move, domain_size

# Precompute full-domain test grid for GP predictions (normalized)
res = 100
xt_full = np.linspace(0, domain_size, res)
yt_full = np.linspace(0, domain_size, res)
X_full, Y_full = np.meshgrid(xt_full, yt_full)
test_pts_full = np.stack([X_full.ravel(), Y_full.ravel()], axis=1)
test_norm_full = test_pts_full / domain_size  # normalize inputs
test_x_full = torch.tensor(test_norm_full, dtype=torch.float32)
```

```
all_x, all_y, all_val = [], [], []
total_cost = 0.0

for i, (gx, gy) in enumerate(grid_centers, start=1):
    # 1) Indent coords for this grid
    positions = local_offsets + np.array([gx, gy])
    x_coords, y_coords = positions[:, 0], positions[:, 1]

    # 2) Measure hardness
    meas = [measure_from_Hgrid(x, y, noise_std=1.4) for x, y in zip(x_coords, y_coo
    all_x.extend(x_coords)
    all_y.extend(y_coords)
    all_val.extend(meas)

    # 3) Normalize training data
    X_raw = np.stack([all_x, all_y], axis=1)
    X_norm = X_raw / domain_size
    z = np.array(all_val)
    z_min, z_max = z.min(), z.max()
    z_norm = (z - z_min) / (z_max - z_min)

    train_x = torch.tensor(X_norm, dtype=torch.float32)
    train_y = torch.tensor(z_norm, dtype=torch.float32)

    # 4) Fit GP on normalized data
    inducing = train_x[:min(500, train_x.size(0))]
    gp = viGP(inducing, ConstantMean(), ScaleKernel(RBFKernel()))
    gp.fit(train_x, train_y, training_iter=200)

    # 5) Predict over full domain, then de-normalize
    pred_norm = gp.predict(test_x_full)
    pred = pred_norm * (z_max - z_min) + z_min
    Z_pred = pred.reshape(res, res)

    # 6) Compute cost for this grid
    grid_cost = t_setup + t_move + evaluate_sample_cost_remaining(x_coords, y_coord
    total_cost += grid_cost

    # # 7) Plotting
    # fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # # Left: GP predicted mean
    # im = axes[0].imshow(
    #     Z_pred,
    #     origin='lower',
    #     extent=(0, domain_size, 0, domain_size),
    #     vmin=2.8,
    #     vmax=7.0,
    #     cmap='viridis'
    # )
    # axes[0].set_title(f"GP Pred Mean after {len(all_val)} indents\nCumulative Cos
    # axes[0].set_xlabel("x (µm)")
    # axes[0].set_ylabel("y (µm)")

    # # Right: Ground truth measured values so far
```

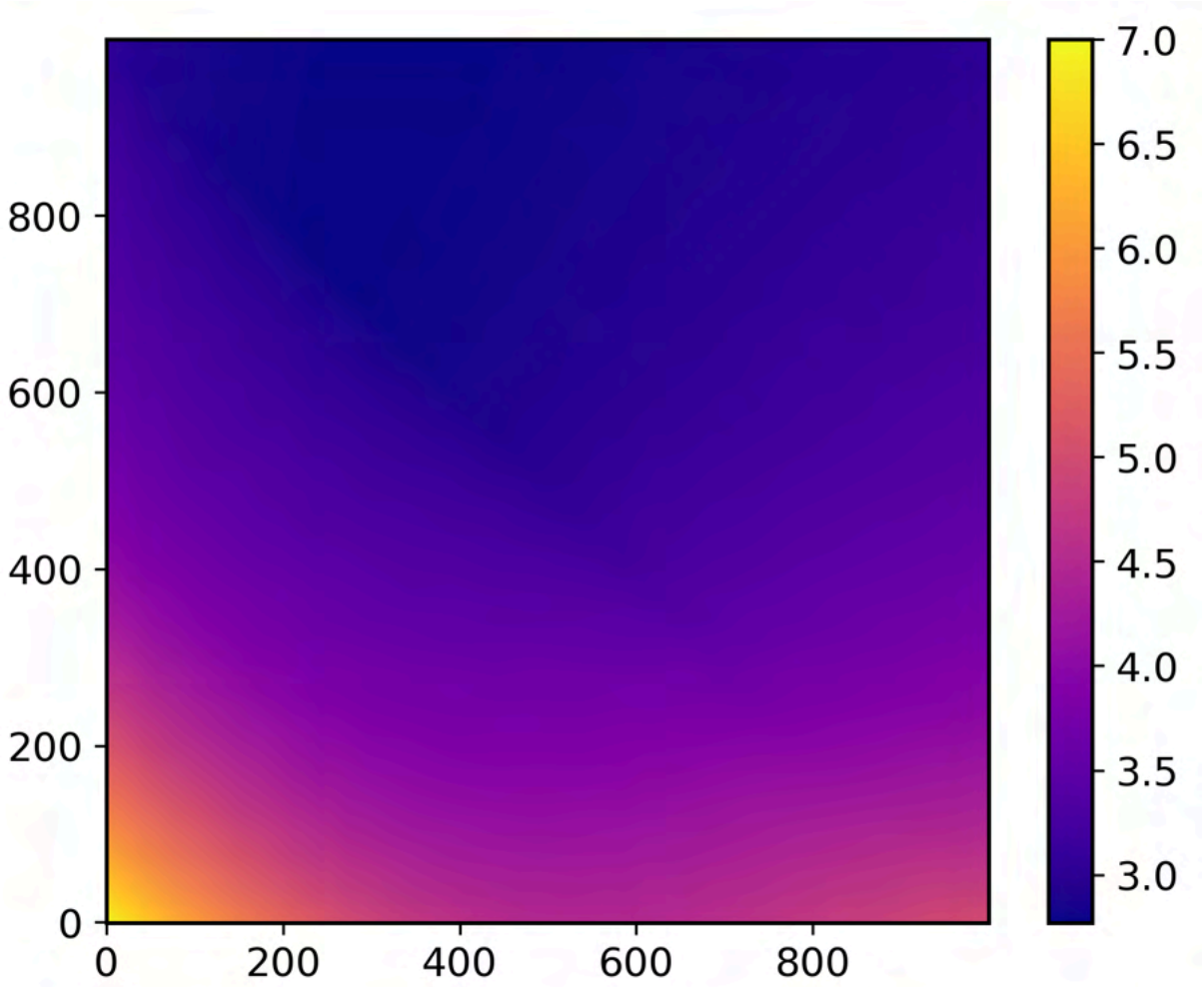
```
    # sc = axes[1].scatter(
    #     all_x, all_y,
    #     c=all_val,
    #     cmap='viridis',
    #     vmin=2.8,
    #     vmax=7.0,
    #     s=10
    # )
    # axes[1].set_title("Measured Hardness")
    # axes[1].set_xlabel("x (µm)")
    # axes[1].set_ylabel("y (µm)")

    # # Shared colorbar
    # fig.colorbar(im, ax=axes.ravel().tolist(), label="Hardness (GPa)")

    # plt.tight_layout()
    # plt.show()


    # 7) Plotting
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Left: GP predicted mean
im = axes[0].imshow(
    Z_pred,
    origin='lower',
    extent=(0, domain_size, 0, domain_size),
    vmin=2.8,
    vmax=7.0,
    cmap='plasma'
)
axes[0].set_title(f"GP Pred Mean after {len(all_val)} indents\nCumulative Cost: {to
axes[0].set_xlabel("x (µm)")
axes[0].set_ylabel("y (µm)")

# Right: Ground truth measured values so far
sc = axes[1].scatter(
    all_x, all_y,
    c=all_val,
    cmap='plasma',
    vmin=2.8,
    vmax=7.0,
    s=10
)
axes[1].set_title("Measured Hardness")
axes[1].set_xlabel("x (µm)")
axes[1].set_ylabel("y (µm)")

# Shared colorbar
fig.colorbar(im, ax=axes.ravel().tolist(), label="Hardness (GPa)")

plt.tight_layout()
plt.show()
```

```
C:\Users\vchawla\AppData\Local\Temp\ipykernel_47884\3830722206.py:130: UserWarning:
This figure includes Axes that are not compatible with tight_layout, so results migh
t be incorrect.
  plt.tight_layout()
```



In [38]:
```
plt.imshow(H_grid, origin='lower', cmap='plasma')
plt.colorbar()
# plt.set_title("Hardness (GPa)")
# fig.colorbar(c4, ax=axs[1, 1], shrink=0.8)
```

Out[38]:
```
<matplotlib.colorbar.Colorbar at 0x1db30ce8090>
```



Describing sample tilt

In [ ]:
```python
import numpy as np

X_bl=0
Y_bl=0
Z_bl=9.67175700
X_tr=1
Y_tr=0
Z_tr=9.671756990
X_br=1
Y_br=1
Z_br=9.671755446293

def fit_plane_xy_to_z(points):
    """
    Fit z ≈ a*x + b*y + c from 3 (or more) (x,y,z) points.
    Returns (a, b, c, predict_fn).
    """
    P = np.asarray(points, dtype=float)
    if P.ndim != 2 or P.shape[1] != 3 or P.shape[0] < 3:
        raise ValueError("Provide an array-like of shape (N,3) with N>=3.")
    X = np.c_[P[:,0], P[:,1], np.ones(len(P))]   # [x y 1]
```

```
    z = P[:,2]
    a, b, c = np.linalg.lstsq(X, z, rcond=None)[0]
    return a, b, c, (lambda x, y: a*x + b*y + c)

# Example:
pts = [(X_bl,Y_bl, Z_bl), (X_tr,Y_tr, Z_tr), (X_br,Y_br, Z_br)]
a, b, c, f = fit_plane_xy_to_z(pts)
print(f"z = {a:.6f}*x + {b:.6f}*y + {c:.6f}")
print("z(0.5, 0.5) =", f(0.5, 0.5))
```

```
z = -0.000000*x + -0.000002*y + 9.671757
z(0.5, 0.5) = 9.6717562231465
```

Now vanilla GP

This code:

1. **Defines** every possible 5×5 indent-block center across the 1000 µm domain.
2. **Randomly picks** `n_initial` blocks and measures them.
3. **Fits** a normalized GP on all measured indents.
4. **Computes** uncertainty-over-cost for each remaining block (skipping any whose 25 indents overlap already measured points).
5. **Selects** the block with maximum acquisition, measures it, removes it from the pool, and updates the total cost. we give penalty if we are close to the edges.
6. **Repeats** for `n_steps` active iterations.

All you need to do is supply `viGP`, `measure_from_Hgrid`, `evaluate_sample_cost_remaining`, and `UE` as before. ::contentReference[oaicite:0]{index=0}

In [ ]:

```
import os, random
import numpy as np
import torch
import matplotlib.pyplot as plt
from gpytorch.means import ConstantMean
from gpytorch.kernels import ScaleKernel, RBFKernel

# ========================
# Reproducibility
# ========================
SEED = 1337
np.random.seed(SEED)
random.seed(SEED)
torch.manual_seed(SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(SEED)
# Safer determinism (may slow down a bit)
try:
    torch.use_deterministic_algorithms(True)
except Exception:
    pass
```

```
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# Output folder
outdir = "al_runs_figs"
os.makedirs(outdir, exist_ok=True)

# Matplotlib "journal" defaults
plt.rcParams.update({
    "figure.dpi": 110,
    "savefig.dpi": 300,
    "font.size": 11,
    "axes.labelsize": 11,
    "axes.titlesize": 12,
    "legend.fontsize": 10,
    "xtick.labelsize": 10,
    "ytick.labelsize": 10,
    "axes.linewidth": 0.8,
    "lines.linewidth": 2.0,
    "grid.alpha": 0.6,
})

# === Assumes these are already defined/imported in your env ===
# viGP
# measure_from_Hgrid(x, y, noise_std)
# evaluate_sample_cost_remaining(x_coords, y_coords)
# UE(model, X) -> tensor or ndarray of uncertainties (per point)  # treated as std
# movement_time, t_engage, default_t_drift, default_t_lift
# Optional: true_from_Hgrid(x, y)  # noiseless ground truth if available

# ========= Config =========
domain_size        = 1000      # µm
grid_size          = 5         # 5×5 indents per block
spacing            = 5         # µm between indents
t_setup            = 20        # s setup per block
t_move             = 20        # s travel per block
n_initial          = 5         # start with TWO blocks
n_steps            = 100        # one block per iteration
train_noise        = 1.4       # measurement noise
res_full           = 100       # visualization + global-error grid resolution
inducing_cap       = 500       # viGP inducing cap
train_iters        = 200       # viGP training iters

# Acquisition options
acq_mode           = "UE"      # "UE" or "UCB"
beta_ucb           = 1.0       # UCB exploration weight (kappa)
normalize_by_cost  = True      # divide acquisition by per-block cost

# ---- Small utilities ----
def to_numpy(x):
    if isinstance(x, torch.Tensor):
        return x.detach().cpu().numpy()
    return np.asarray(x)

def maybe_has_true():
    return 'true_from_Hgrid' in globals() and callable(globals()['true_from_Hgrid'])
```

```
def get_true_vals(xs, ys):
    """Return ground-truth if available, else a low-noise proxy."""
    if maybe_has_true():
        return np.array([true_from_Hgrid(int(x), int(y)) for x, y in zip(xs, ys)],
    else:
        try:
            return np.array([measure_from_Hgrid(int(x), int(y), noise_std=0.0) for
        except Exception:
            reps = 3
            vals = []
            for x, y in zip(xs, ys):
                samples = [measure_from_Hgrid(int(x), int(y), noise_std=max(1e-3, 0
                vals.append(np.mean(samples))
            return np.array(vals, dtype=float)

def mape(y_true, y_pred):
    y_true = np.asarray(y_true, dtype=float)
    y_pred = np.asarray(y_pred, dtype=float)
    eps = 1e-9
    return float(np.mean(np.abs((y_true - y_pred) / np.maximum(eps, np.abs(y_true))

# === Build 100×100 possible block centers across a 1000×1000 domain
max_offset = (grid_size - 1) * spacing
grid_centers_x = np.linspace(0, domain_size - max_offset, 100)
grid_centers_y = np.linspace(0, domain_size - max_offset, 100)
available_centers = [(x, y) for x in grid_centers_x for y in grid_centers_y]

# === Precompute local offsets for each block
ix, iy = np.meshgrid(np.arange(grid_size), np.arange(grid_size))
local_offsets = np.stack([ix.ravel(), iy.ravel()], axis=1) * spacing

# ========= Storage =========
all_x, all_y, all_z = [], [], []          # measured (x,y,z)
visited = []                               # visited block centers
acq_history, cost_history = [], []
total_cost = 0.0

# ========= Global grid for visualization + GLOBAL ERROR =========
xt = np.linspace(0, domain_size, res_full)
yt = np.linspace(0, domain_size, res_full)
XX, YY = np.meshgrid(xt, yt)
global_pts  = np.column_stack([XX.ravel(), YY.ravel()])
global_true = get_true_vals(global_pts[:,0], global_pts[:,1])

global_mape_hist          = []
max_hardness_found_hist   = []

# ========= Initial random sampling (TWO blocks) =========
# NOTE: np.random is seeded above; this choice is repeatable
init_idxs = np.random.choice(len(available_centers), n_initial, replace=False)
for idx in init_idxs:
    gx, gy = available_centers[idx]
    pos = local_offsets + np.array([gx, gy])
    xs, ys = pos[:,0], pos[:,1]
    zs = [measure_from_Hgrid(int(x), int(y), noise_std=train_noise) for x, y in zip
```

```
    all_x.extend(xs); all_y.extend(ys); all_z.extend(zs)
    move_cost = evaluate_sample_cost_remaining(xs, ys)[0]
    total_cost += (t_setup + t_move + move_cost)
    cost_history.append(total_cost)
    visited.append((gx, gy))

# Remove initial centers from pool
available_centers = [c for i, c in enumerate(available_centers) if i not in init_id

# ========= Active learning loop (one block per iteration) =========
for step in range(n_steps):
    # --- Normalize data (XY to [0,1], Z to [0,1]) ---
    X_raw  = np.column_stack([all_x, all_y])
    X_norm = X_raw / domain_size
    z_arr  = np.array(all_z, dtype=float)
    z_min, z_max = z_arr.min(), z_arr.max()
    z_rng = max(1e-12, (z_max - z_min))
    z_norm = (z_arr - z_min) / z_rng

    # --- Fit GP on normalized data ---
    train_x = torch.tensor(X_norm, dtype=torch.float32)
    train_y = torch.tensor(z_norm, dtype=torch.float32)
    inducing = train_x[:min(inducing_cap, train_x.size(0))]
    gp = viGP(inducing, mean_module=ConstantMean(), kernel_module=ScaleKernel(RBFKe
    gp.fit(train_x, train_y, training_iter=train_iters)

    # --- Acquisition for each remaining center ---
    acq_vals = []
    for gx, gy in available_centers:
        pos = local_offsets + np.array([gx, gy])
        xs, ys = pos[:,0], pos[:,1]

        # guard: if any of these points already measured, skip this block
        if any((x, y) in zip(all_x, all_y) for x, y in zip(xs, ys)):
            acq_vals.append(-np.inf)
            continue

        X_test = torch.tensor(pos / domain_size, dtype=torch.float32)
        u      = to_numpy(UE(gp, X_test))              # assumed std on normalized
        u_mean = float(np.mean(u))

        if acq_mode.upper() == "UCB":
            # UCB on normalized scale: mu_norm + beta * sigma_norm
            mu_norm = to_numpy(gp.predict(X_test))
            ucb_val = float(np.mean(mu_norm + beta_ucb * u))
            score = ucb_val
        else:
            # UE mode (default): uncertainty-only
            score = u_mean

        if normalize_by_cost:
            move_cost = evaluate_sample_cost_remaining(xs, ys)[0]
            block_cost = t_setup + t_move + move_cost
            score = score / max(1e-9, block_cost)

        acq_vals.append(score)
```

```
    # --- Select next center ---
    best_idx = int(np.argmax(acq_vals))
    gx, gy = available_centers.pop(best_idx)
    visited.append((gx, gy))
    acq_history.append(acq_vals[best_idx])

    # --- Measure selected block ---
    pos = local_offsets + np.array([gx, gy])
    xs, ys = pos[:,0], pos[:,1]
    zs = [measure_from_Hgrid(int(x), int(y), noise_std=train_noise) for x, y in zip
    all_x.extend(xs); all_y.extend(ys); all_z.extend(zs)

    # --- Update cost ---
    move_cost = evaluate_sample_cost_remaining(xs, ys)[0]
    total_cost += (t_setup + t_move + move_cost)
    cost_history.append(total_cost)

    # --- Predict on FULL GRID (for both visualization and GLOBAL ERROR) ---
    test_x = torch.tensor(global_pts / domain_size, dtype=torch.float32)
    pred_norm = to_numpy(gp.predict(test_x))
    pred_grid = pred_norm * z_rng + z_min

    # --- Global MAPE (percent) ---
    global_mape = mape(global_true, pred_grid) * 100.0  # percent
    global_mape_hist.append(global_mape)

    # --- Track max hardness found so far (prefer truth at measured points if avail
    try:
        measured_true = get_true_vals(all_x, all_y)     # same length as all_z
        current_max = float(np.max(measured_true))
    except Exception:
        current_max = float(np.max(all_z))
    max_hardness_found_hist.append(current_max)
    print(f"[Iter {step+1}] Global MAPE: {global_mape:.2f}% | Max hardness found so

    # --- GP uncertainty over full grid (for map viz only) ---
    ue_full = to_numpy(UE(gp, test_x))

    # --- Build maps for visualization ---
    Z_pred = pred_grid.reshape(res_full, res_full)
    UE_map = ue_full.reshape(res_full, res_full)

    # Acquisition/cost map (optional viz)
    dx = global_pts[:, 0]; dy = global_pts[:, 1]
    base_dx = movement_time(dx); base_dy = movement_time(dy)
    penalty = np.where((dx >= 1000) | (dy >= 1000),
                       t_engage + default_t_drift + default_t_lift, 0)
    move_cost_map = (base_dx + base_dy + penalty)
    ACQ_map = (ue_full / np.maximum(1e-9, move_cost_map)).reshape(res_full, res_ful

    # ==================== Plots ====================
    # ---- 2×2 per-iteration maps ----
    fig, axs = plt.subplots(2, 2, figsize=(10, 8))
    # (0,0) GP Pred Mean
    im0 = axs[0,0].imshow(Z_pred, origin='lower',
```

```
                      extent=(0, domain_size, 0, domain_size),
                      vmin=3, vmax=7, cmap='plasma')
axs[0,0].set_title(f"Iter {step+1}: GP Pred Mean")
axs[0,0].scatter(all_x, all_y, c=np.arange(len(all_x)), cmap='jet', s=5)
fig.colorbar(im0, ax=axs[0,0], shrink=0.8)
# (0,1) GP UE
im1 = axs[0,1].imshow(UE_map, origin='lower',
                      extent=(0, domain_size, 0, domain_size),
                      cmap='plasma')
axs[0,1].set_title("GP Uncertainty (UE)")
vx, vy = zip(*visited)
axs[0,1].scatter(vx, vy, c=np.arange(len(visited)), cmap='jet', s=50)
fig.colorbar(im1, ax=axs[0,1], shrink=0.8)
# (1,0) Acquisition / Cost (UE/cost for viz)
im2 = axs[1,0].imshow(ACQ_map, origin='lower',
                      extent=(0, domain_size, 0, domain_size),
                      cmap='plasma')
axs[1,0].set_title("UE / MoveCost")
axs[1,0].scatter(vx, vy, c=np.arange(len(visited)), cmap='jet', s=50)
fig.colorbar(im2, ax=axs[1,0], shrink=0.8)
# (1,1) Cost progression
steps_arr = np.arange(1, len(cost_history)+1)
axs[1,1].plot(steps_arr, cost_history, marker='o')
axs[1,1].set_title("Cumulative Cost")
axs[1,1].set_xlabel("Iteration")
axs[1,1].set_ylabel("Cost (s)")
axs[1,1].grid(True, linestyle='--', alpha=0.6)

plt.tight_layout()

# Save maps every 4 iterations
if ((step + 1) % 2) == 0:
    fig.savefig(os.path.join(outdir, f"iter_{step+1:03d}_maps.png"),
                bbox_inches="tight", pad_inches=0.02)
plt.show()

# ---- Learning curve: MAPE (%) + Max hardness (right axis) ----
fig2, ax2 = plt.subplots(figsize=(7.8, 4.8))
plt.subplots_adjust(right=0.78)   # Leave room for legend

its = np.arange(1, len(global_mape_hist)+1)

# Left axis: black line
l1 = ax2.plot(
    its, global_mape_hist,
    marker='o',
    label='Global MAPE (%)',
    color='black'
)[0]
ax2.set_xlabel("Iteration")
ax2.set_ylabel("Global MAPE (%)", color='black')
ax2.tick_params(axis='y', labelcolor='black')
ax2.set_title("Model Learning: MAPE vs Iteration (and Max Hardness)")
ax2.grid(True, linestyle='--', alpha=0.6)

# Right axis: blue line and blue label
```

```
    ax2r = ax2.twinx()
    l2 = ax2r.plot(
        its, max_hardness_found_hist,
        marker='^',
        linestyle='--',
        label='Max Hardness Found',
        color='blue'
    )[0]
    ax2r.set_ylabel("Max Hardness Found", color='blue')
    ax2r.tick_params(axis='y', labelcolor='blue')

    # legend outside
    lines  = [l1, l2]
    labels = [ln.get_label() for ln in lines]
    ax2.legend(
        lines, labels,
        loc='center left',
        bbox_to_anchor=(1.20, 0.5),
        borderaxespad=0.0, frameon=False, handlelength=2.0
    )
    # plt.show()


    # Save learning curve every 4 iterations
    if ((step + 1) % 2) == 0:
        fig2.savefig(os.path.join(outdir, f"iter_{step+1:03d}_mape_maxH.png"),
                     bbox_inches="tight", pad_inches=0.02)
    plt.show()

# ======= Final plots (two color variants for the learning curve) =======
its = np.arange(1, len(global_mape_hist)+1)

# Variant A: MAPE in orange
figA, axA = plt.subplots(figsize=(7.8, 4.8))
plt.subplots_adjust(right=0.78)
axA.plot(its, global_mape_hist, marker='o', label='Global MAPE (%)', color='tab:ora
axA.set_xlabel("Iteration")
axA.set_ylabel("Global MAPE (%)")
axA.set_title("Model Learning: MAPE vs Iteration (and Max Hardness)")
axA.grid(True, linestyle='--', alpha=0.6)
axAr = axA.twinx()
axAr.plot(its, max_hardness_found_hist, marker='^', linestyle='--',
          label='Max Hardness Found', color='dimgray')
axAr.set_ylabel("Max Hardness Found")
lines = [axA.lines[0], axAr.lines[0]]
labels = ['Global MAPE (%)', 'Max Hardness Found']
axA.legend(lines, labels, loc='center left', bbox_to_anchor=(1.20, 0.5),
           borderaxespad=0.0, frameon=False, handlelength=2.0)
figA.savefig(os.path.join(outdir, "final_learning_curve_orange.png"),
             bbox_inches="tight", pad_inches=0.02)
plt.show()

# Variant B: MAPE in blue
figB, axB = plt.subplots(figsize=(7.8, 4.8))
plt.subplots_adjust(right=0.78)
axB.plot(its, global_mape_hist, marker='o', label='Global MAPE (%)', color='tab:blu
```

```
axB.set_xlabel("Iteration")
axB.set_ylabel("Global MAPE (%)")
axB.set_title("Model Learning: MAPE vs Iteration (and Max Hardness)")
axB.grid(True, linestyle='--', alpha=0.6)
axBr = axB.twinx()
axBr.plot(its, max_hardness_found_hist, marker='^', linestyle='--',
          label='Max Hardness Found', color='dimgray')
axBr.set_ylabel("Max Hardness Found")
lines = [axB.lines[0], axBr.lines[0]]
labels = ['Global MAPE (%)', 'Max Hardness Found']
axB.legend(lines, labels, loc='center left', bbox_to_anchor=(1.20, 0.5),
           borderaxespad=0.0, frameon=False, handlelength=2.0)
figB.savefig(os.path.join(outdir, "final_learning_curve_blue.png"),
             bbox_inches="tight", pad_inches=0.02)
plt.show()

# ======= Final report =======
if max_hardness_found_hist:
    print(f"\nFinal: Max hardness found = {max_hardness_found_hist[-1]:.6g}")
if global_mape_hist:
    print(f"Final: Global MAPE = {global_mape_hist[-1]:.3f}%")
print(f"Acquisition mode: {acq_mode} "
      f"{'(cost-normalized)' if normalize_by_cost else '(no cost normalization)'} "
      f"{'(beta={:.3g})'.format(beta_ucb) if acq_mode.upper()=='UCB' else ''}")
print(f"Saved per-4-iteration figures and final variants in: {os.path.abspath(outdi
```

Refrence Grid based

In [28]:
```
# Sweep grids_per_dim = 2..10; measure -> fit GP -> compute MAPE, track cost, time,

import os, time, random
import numpy as np
import matplotlib.pyplot as plt
import torch
from gpytorch.means import ConstantMean
from gpytorch.kernels import ScaleKernel, RBFKernel

# ========== Reproducibility ==========
SEED = 1337
np.random.seed(SEED)
random.seed(SEED)
torch.manual_seed(SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(SEED)
try:
    torch.use_deterministic_algorithms(True)
except Exception:
    pass
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

plt.rcParams.update({
    "savefig.dpi": 300,
    "font.size": 11,
```

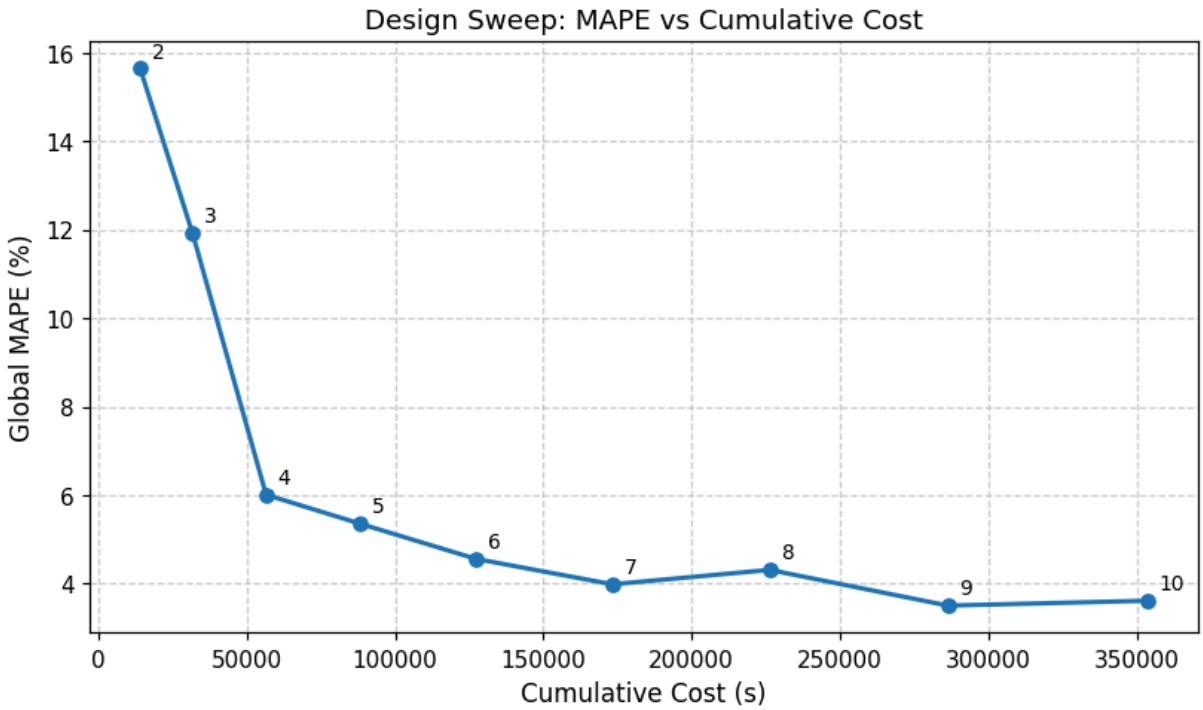
```
    "axes.labelsize": 11,
    "axes.titlesize": 12,
    "legend.fontsize": 10,
    "xtick.labelsize": 10,
    "ytick.labelsize": 10,
    "axes.linewidth": 0.8,
    "lines.linewidth": 2.0,
    "grid.alpha": 0.6,
})

# === Assumes these are already defined/imported in your env ===
# viGP
# measure_from_Hgrid(x, y, noise_std)
# evaluate_sample_cost_remaining(x_coords, y_coords)
# movement_time, t_engage, default_t_drift, default_t_lift  (not used here)
# Optional: true_from_Hgrid(x, y)  # noiseless ground truth if available

# ========= Config (fixed across sweep) =========
domain_size   = 1000      # µm
grid_size     = 5         # 5×5 indents per block
spacing       = 5         # µm between indents
t_setup       = 20        # s setup per block
t_move        = 20        # s travel per block
train_noise   = 1.4       # measurement noise injected
inducing_cap  = 50000     # viGP inducing cap
train_iters   = 200       # viGP training iters
res_full      = 100       # resolution for full-domain prediction & MAPE

# ---------- Helpers ----------
def to_numpy(x):
    if isinstance(x, torch.Tensor):
        return x.detach().cpu().numpy()
    return np.asarray(x)

def maybe_has_true():
    return 'true_from_Hgrid' in globals() and callable(globals()['true_from_Hgrid']

def get_true_vals(xs, ys):
    """Noiseless truth if available; else noise-free (or low-noise) proxy."""
    if maybe_has_true():
        return np.array([true_from_Hgrid(int(x), int(y)) for x, y in zip(xs, ys)],
    try:
        return np.array([measure_from_Hgrid(int(x), int(y), noise_std=0.0) for x, y
    except Exception:
        reps = 3
        vals = []
        for x, y in zip(xs, ys):
            samples = [measure_from_Hgrid(int(x), int(y), noise_std=max(1e-3, 0.1*t
            vals.append(np.mean(samples))
        return np.array(vals, dtype=float)

def mape(y_true, y_pred):
    y_true = np.asarray(y_true, dtype=float)
    y_pred = np.asarray(y_pred, dtype=float)
    eps = 1e-9
    return float(np.mean(np.abs((y_true - y_pred) / np.maximum(eps, np.abs(y_true))
```

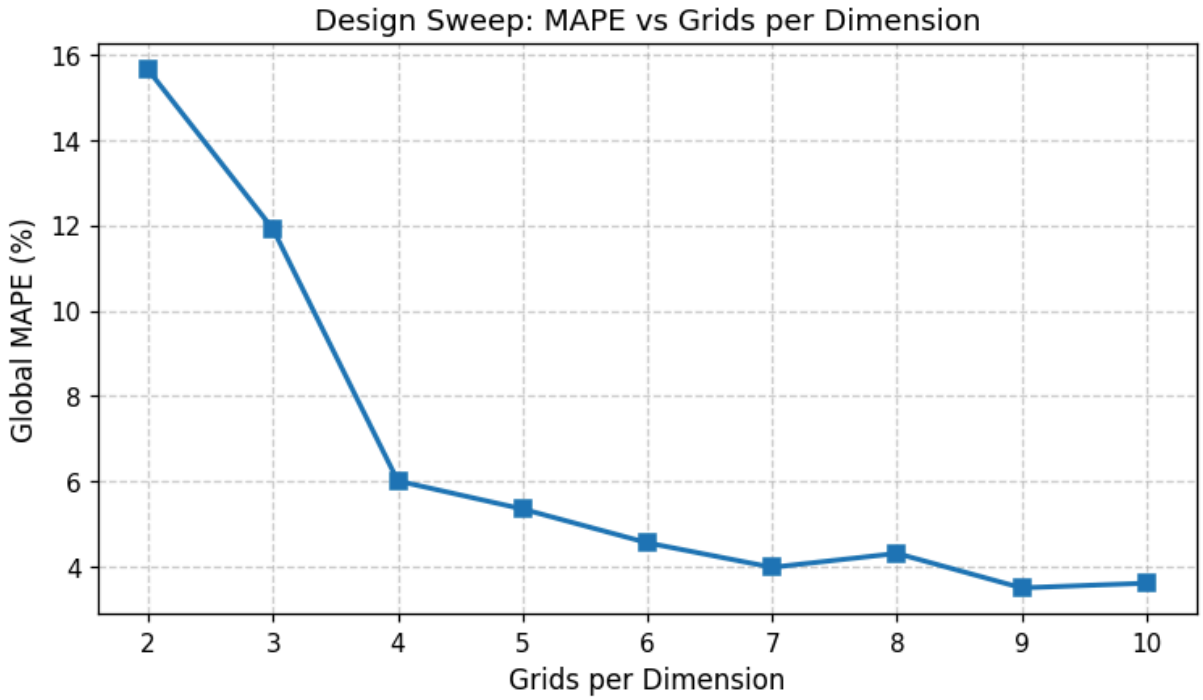
```
# --- Precompute indent layout for a single 5×5 block (relative offsets) ---
ix, iy = np.meshgrid(np.arange(grid_size), np.arange(grid_size))
local_offsets = np.stack([ix.ravel(), iy.ravel()], axis=1) * spacing  # shape (25,2

# --- Full-domain grid for predictions & error ---
xt = np.linspace(0, domain_size, res_full)
yt = np.linspace(0, domain_size, res_full)
XX, YY = np.meshgrid(xt, yt)
full_pts   = np.column_stack([XX.ravel(), YY.ravel()])
full_norm  = full_pts / domain_size
full_x_t   = torch.tensor(full_norm, dtype=torch.float32)
true_full  = get_true_vals(full_pts[:,0], full_pts[:,1])

# ---------- Sweep 2..10 grids per dimension ----------
results = []  # list of dicts per configuration

for gpd in range(2, 11):  # 2,3,...,10
    tic = time.time()

    # Grid centers (exclude outer edges; evenly spaced interior)
    grid_centers_x = np.linspace(0, domain_size, gpd + 2)[1:-1]
    grid_centers_y = np.linspace(0, domain_size, gpd + 2)[1:-1]
    grid_centers   = [(x, y) for x in grid_centers_x for y in grid_centers_y]
    n_blocks       = len(grid_centers)            # gpd^2
    n_indents      = n_blocks * (grid_size**2)    # total points measured

    # Collect all measurements for this design
    all_x, all_y, all_z = [], [], []
    total_cost = 0.0

    for (gx, gy) in grid_centers:
        pos = local_offsets + np.array([gx, gy])         # shape (25,2)
        xs, ys = pos[:,0], pos[:,1]
        zs = [measure_from_Hgrid(int(x), int(y), noise_std=train_noise) for x, y in

        all_x.extend(xs); all_y.extend(ys); all_z.extend(zs)

        # Cost for this block
        move_cost  = evaluate_sample_cost_remaining(xs, ys)[0]
        total_cost += (t_setup + t_move + move_cost)

    # Track max hardness found among measured points (prefer truth if available)
    try:
        measured_true = get_true_vals(all_x, all_y)
        max_hardness_found = float(np.max(measured_true))
    except Exception:
        max_hardness_found = float(np.max(all_z))

    # Fit GP once on all accumulated data
    X_raw  = np.column_stack([all_x, all_y])
    X_norm = X_raw / domain_size
    z_arr  = np.asarray(all_z, dtype=float)
    z_min, z_max = z_arr.min(), z_arr.max()
    z_rng = max(1e-12, (z_max - z_min))
    z_norm = (z_arr - z_min) / z_rng
```

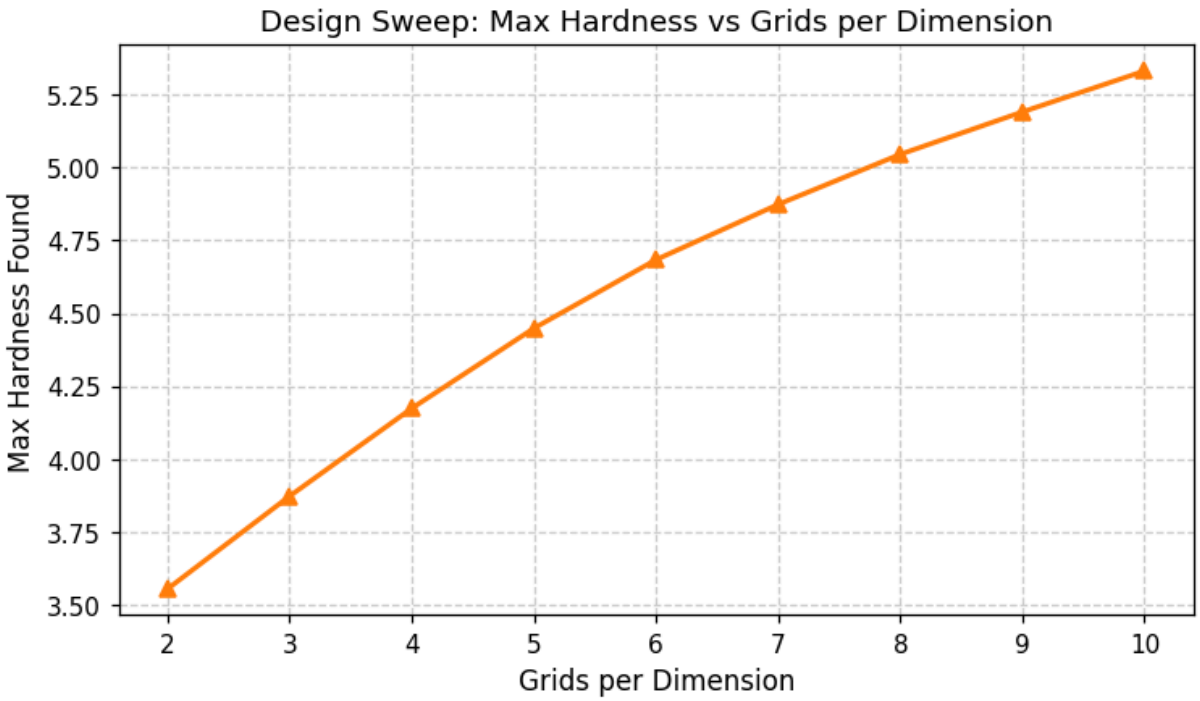
```
    train_x = torch.tensor(X_norm, dtype=torch.float32)
    train_y = torch.tensor(z_norm, dtype=torch.float32)
    inducing = train_x[:min(inducing_cap, train_x.size(0))]

    gp = viGP(inducing, mean_module=ConstantMean(), kernel_module=ScaleKernel(RBFKe
    gp.fit(train_x, train_y, training_iter=train_iters)

    # Predict full domain and compute MAPE (%)
    pred_norm = to_numpy(gp.predict(full_x_t))
    pred_phys = pred_norm * z_rng + z_min
    final_mape_pct = mape(true_full, pred_phys) * 100.0

    wall_time_s = time.time() - tic

    results.append({
        "grids_per_dim": gpd,
        "n_blocks": n_blocks,
        "n_indents": n_indents,
        "cumulative_cost_s": float(total_cost),
        "wall_time_s": float(wall_time_s),
        "mape_percent": float(final_mape_pct),
        "max_hardness_found": max_hardness_found,
    })

    print(f"[gpd={gpd:2d}] blocks={n_blocks:3d} indents={n_indents:5d}  "
          f"cost={total_cost:9.1f}s  MAPE={final_mape_pct:6.2f}%  "
          f"maxH={max_hardness_found:.4g}  time={wall_time_s:7.2f}s")

# ---------- Plot summary ----------
# 1) MAPE (%) vs cumulative cost
fig1, ax1 = plt.subplots(figsize=(7.6, 4.6))
costs  = [r["cumulative_cost_s"] for r in results]
mapes  = [r["mape_percent"]      for r in results]
labels = [r["grids_per_dim"]     for r in results]

ax1.plot(costs, mapes, marker='o')
for c, m, lab in zip(costs, mapes, labels):
    ax1.annotate(str(lab), (c, m), textcoords="offset points", xytext=(5,5), fontsi
ax1.set_xlabel("Cumulative Cost (s)")
ax1.set_ylabel("Global MAPE (%)")
ax1.set_title("Design Sweep: MAPE vs Cumulative Cost")
ax1.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
plt.savefig("sweep_mape_vs_cost.png", dpi=300, bbox_inches="tight")
plt.show()

# 2) MAPE (%) vs grids per dimension
fig2, ax2 = plt.subplots(figsize=(7.0, 4.2))
gpds = [r["grids_per_dim"] for r in results]
ax2.plot(gpds, mapes, marker='s', color='tab:blue')
ax2.set_xlabel("Grids per Dimension")
ax2.set_ylabel("Global MAPE (%)")
ax2.set_title("Design Sweep: MAPE vs Grids per Dimension")
ax2.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
```

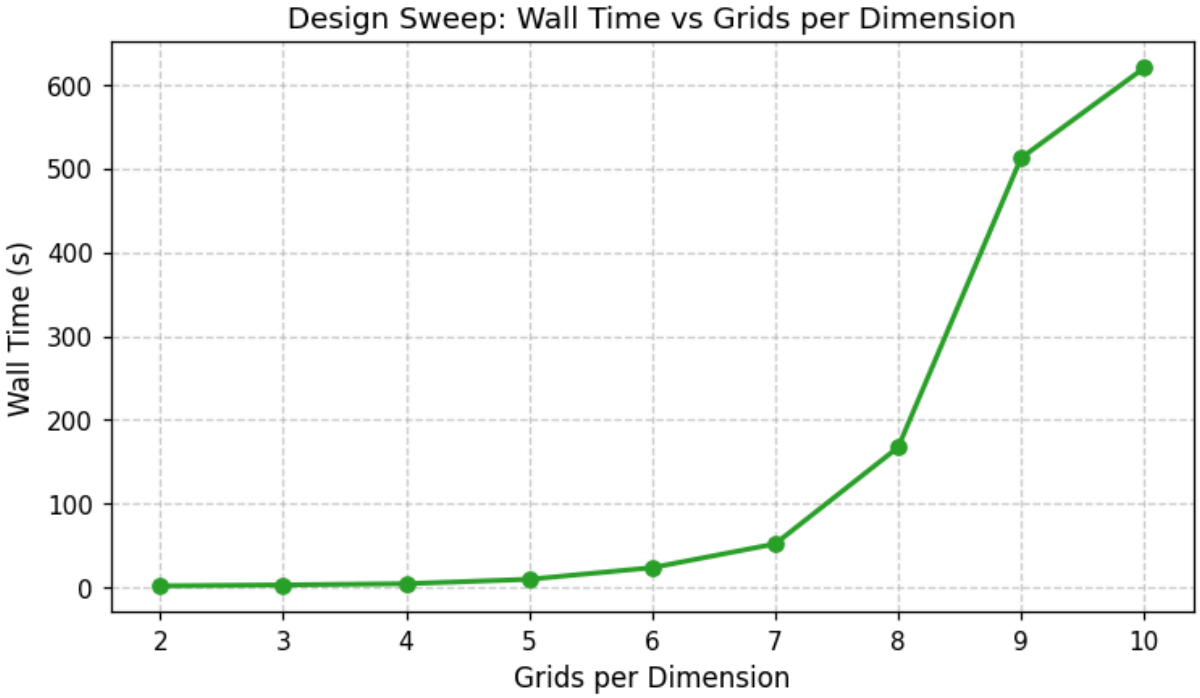
```
plt.savefig("sweep_mape_vs_grids.png", dpi=300, bbox_inches="tight")
plt.show()

# 3) Max hardness found vs grids per dimension
fig3, ax3 = plt.subplots(figsize=(7.0, 4.2))
maxHs = [r["max_hardness_found"] for r in results]
ax3.plot(gpds, maxHs, marker='^', color='tab:orange')
ax3.set_xlabel("Grids per Dimension")
ax3.set_ylabel("Max Hardness Found")
ax3.set_title("Design Sweep: Max Hardness vs Grids per Dimension")
ax3.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
plt.savefig("sweep_maxH_vs_grids.png", dpi=300, bbox_inches="tight")
plt.show()

# 4) (Optional) wall time vs grids per dimension
fig4, ax4 = plt.subplots(figsize=(7.0, 4.2))
times = [r["wall_time_s"] for r in results]
ax4.plot(gpds, times, marker='o', color='tab:green')
ax4.set_xlabel("Grids per Dimension")
ax4.set_ylabel("Wall Time (s)")
ax4.set_title("Design Sweep: Wall Time vs Grids per Dimension")
ax4.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
plt.savefig("sweep_time_vs_grids.png", dpi=300, bbox_inches="tight")
plt.show()

# Print a compact table at the end
print("\n=== Summary (gpd, blocks, indents, cost[s], MAPE[%], maxH, wall_time[s]) =
for r in results:
    print(f"{r['grids_per_dim']:2d}  {r['n_blocks']:4d}  {r['n_indents']:5d}  "
          f"{r['cumulative_cost_s']:10.1f}  {r['mape_percent']:8.3f}  "
          f"{r['max_hardness_found']:7.3f}  {r['wall_time_s']:9.2f}")
```

```
[gpd= 2] blocks=  4 indents=  100  cost=  14142.8s  MAPE= 15.64%  maxH=3.556  time=
2.01s
[gpd= 3] blocks=  9 indents=  225  cost=  31821.3s  MAPE= 11.91%  maxH=3.874  time=
3.17s
[gpd= 4] blocks= 16 indents=  400  cost=  56571.2s  MAPE=  6.01%  maxH=4.175  time=
4.61s
[gpd= 5] blocks= 25 indents=  625  cost=  88392.5s  MAPE=  5.35%  maxH=4.449  time=
9.89s
[gpd= 6] blocks= 36 indents=  900  cost= 127285.2s  MAPE=  4.56%  maxH=4.684  time=
23.74s
[gpd= 7] blocks= 49 indents= 1225  cost= 173249.2s  MAPE=  3.99%  maxH=4.874  time=
52.13s
[gpd= 8] blocks= 64 indents= 1600  cost= 226284.7s  MAPE=  4.31%  maxH=5.045  time=
167.34s
[gpd= 9] blocks= 81 indents= 2025  cost= 286391.6s  MAPE=  3.50%  maxH=5.191  time=
512.69s
[gpd=10] blocks=100 indents= 2500  cost= 353569.9s  MAPE=  3.62%  maxH=5.331  time=
620.05s
```


Design Sweep: MAPE vs Cumulative Cost
Global MAPE (%)
Cumulative Cost (s)
16
14
12
10
8
6
4
0
50000
100000
150000
200000
250000
300000
350000
2
3
4
5
6
7
8
9
10


Design Sweep: MAPE vs Grids per Dimension
Global MAPE (%)
Grids per Dimension
16
14
12
10
8
6
4
2
3
4
5
6
7
8
9
10





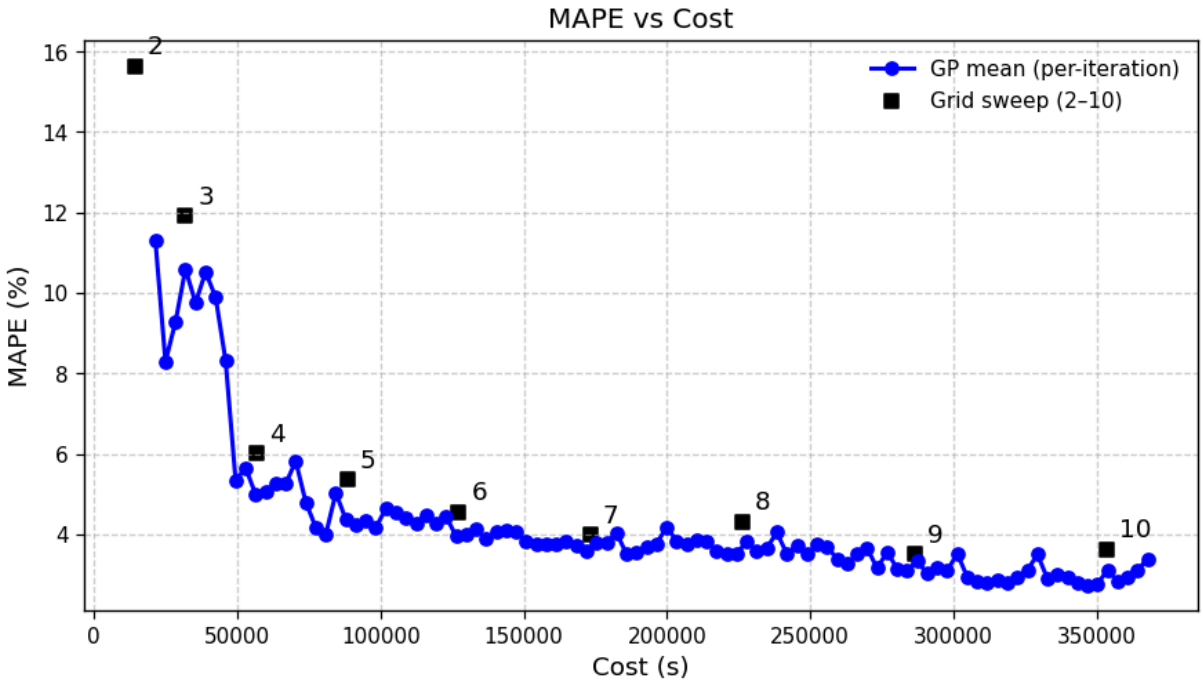

```
=== Summary (gpd, blocks, indents, cost[s], MAPE[%], maxH, wall_time[s]) ===
 2     4    100     14142.8    15.643    3.556       2.01
 3     9    225     31821.3    11.913    3.874       3.17
 4    16    400     56571.2     6.011    4.175       4.61
 5    25    625     88392.5     5.353    4.449       9.89
 6    36    900    127285.2     4.561    4.684      23.74
 7    49   1225    173249.2     3.986    4.874      52.13
 8    64   1600    226284.7     4.313    5.045     167.34
 9    81   2025    286391.6     3.504    5.191     512.69
10   100   2500    353569.9     3.615    5.331     620.05
```

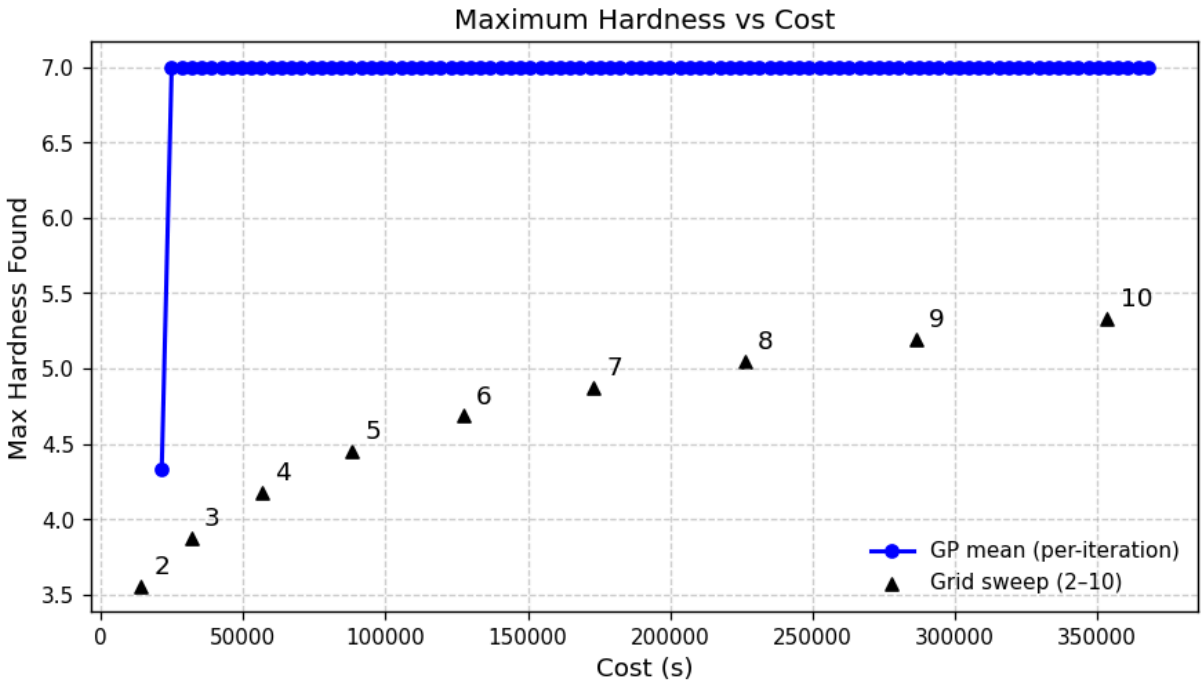
In [ ]:
```
# Two-panel plotting: Cost (s) on X, MAPE or Max Hardness on Y.
# Curve 1: from df (line) where Cost_df = Grid# * 25 * (t_setup + t_move + move_cos
```

```
# Curve 2: from sweep results (10-point scatter) using cumulative_cost_s; points an
#
# If you already have real `df` and `results` in memory, this script will use them.
# Otherwise, it synthesizes plausible demo data with the required schema.

import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# ------------------ Styling ------------------
plt.rcParams.update({
    "savefig.dpi": 300,
    "font.size": 12,
    "axes.labelsize": 12,
    "axes.titlesize": 13,
    "legend.fontsize": 10,
    "xtick.labelsize": 10,
    "ytick.labelsize": 10,
    "axes.linewidth": 0.9,
    "lines.linewidth": 2.0,
})

# ------------------ Inputs ------------------
# Per-indent/block timing terms (seconds)
# t_setup = 20.0
# t_move  = 20.0
# # If you want to plug an empirical average motion cost per indent, set here:
# move_cost_sec = 12.0  # <-- change to your estimate or computed average

per_indent_cost = t_setup + t_move + 100

# ------------------ Load or synthesize df ------------------
if "df" not in globals():
    # Build a demo df with the required columns
    iters = np.arange(1, 101)
    grid_nums = 5 + iters  # starts at 6
    rng = np.random.default_rng(7)
    mape = np.clip(np.linspace(12.0, 3.1, 100) + rng.normal(0, 0.35, 100), 2.5, Non
    maxH = np.maximum.accumulate(np.where(iters < 2, 4.3, 7.0 + rng.normal(0, 0.02,
    df = pd.DataFrame({
        "Grid #": grid_nums,
        "Iteration": iters,
        "MAPE (%)": mape,
        "Max Hardness Found": maxH
    })
else:
    # Ensure columns are trimmed
    df = df.copy()
    df.columns = [c.strip() for c in df.columns]

# Compute Cost_df (s): Grid# * 25 indents per grid * per-indent cost
df["Cost (s)"] = df["Grid #"].astype(float) * 25.0 * per_indent_cost + 400

# ------------------ Load or synthesize sweep results ------------------
if "results" not in globals():
```

```
    # Make a 2..10 sweep demo
    synth = []
    rng = np.random.default_rng(42)
    for gpd in range(2, 11):
        n_blocks = gpd * gpd
        n_indents = n_blocks * 25
        cumulative_cost_s = float(n_indents * per_indent_cost)
        mape_percent = max(2.8, 14.0 - 0.9 * gpd + rng.normal(0, 0.25))
        max_hardness_found = 7.0 + rng.normal(0, 0.015)
        synth.append({
            "grids_per_dim": gpd,
            "cumulative_cost_s": cumulative_cost_s,
            "mape_percent": mape_percent,
            "max_hardness_found": max_hardness_found
        })
    results = synth

sweep_df = pd.DataFrame(results).sort_values("grids_per_dim")

# ------------------ Helpers ------------------
def _finish(ax, title, xlabel, ylabel):
    ax.set_title(title)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.grid(True, linestyle="--", alpha=0.6)
    ax.tick_params(direction="out", length=4, width=0.8)

def _annotate(ax, xs, ys, labels, dx=6, dy=6):
    for x, y, lab in zip(xs, ys, labels):
        ax.annotate(str(lab), (x, y), textcoords="offset points", xytext=(dx, dy))

# ------------------ Figure A: Cost vs MAPE ------------------
figA, axA = plt.subplots(figsize=(8.2, 4.8))

# Curve 1: df (line)
axA.plot(df["Cost (s)"], df["MAPE (%)"], marker="o", label="GP mean (per-iteration)

# Curve 2: sweep (scatter, 10 points), annotated by grid size
axA.scatter(sweep_df["cumulative_cost_s"], sweep_df["mape_percent"], marker="s", la
_annotate(axA, sweep_df["cumulative_cost_s"], sweep_df["mape_percent"], sweep_df["g

_finish(axA, "MAPE vs Cost", "Cost (s)", "MAPE (%)")
axA.legend(frameon=False)
figA.tight_layout()
# figA.savefig("/mnt/data/cost_vs_mape.png", bbox_inches="tight")
# figA.savefig("/mnt/data/cost_vs_mape.pdf", bbox_inches="tight")
# figA.savefig("/mnt/data/cost_vs_mape.svg", bbox_inches="tight")
# plt.close(figA)

# ------------------ Figure B: Cost vs Max Hardness ------------------
figB, axB = plt.subplots(figsize=(8.2, 4.8))

# Curve 1: df (line)
axB.plot(df["Cost (s)"], df["Max Hardness Found"], marker="o", label="GP mean (per-

# Curve 2: sweep (scatter), annotated by grid size
```

```
axB.scatter(sweep_df["cumulative_cost_s"], sweep_df["max_hardness_found"], marker="
_annotate(axB, sweep_df["cumulative_cost_s"], sweep_df["max_hardness_found"], sweep

_finish(axB, "Maximum Hardness vs Cost", "Cost (s)", "Max Hardness Found")
axB.legend(frameon=False)
figB.tight_layout()
# figB.savefig("/mnt/data/cost_vs_maxH.png", bbox_inches="tight")
# figB.savefig("/mnt/data/cost_vs_maxH.pdf", bbox_inches="tight")
# figB.savefig("/mnt/data/cost_vs_maxH.svg", bbox_inches="tight")
# plt.close(figB)

# Report outputs
# sorted([p for p in os.listdir("/mnt/data") if p.startswith(("cost_vs_")) and p.en
```





Dynamic Drift Adaptive GP

Lets see what happens with dynamic drift measurements

Quantitative Comparison: Dynamic vs. Max-Hold GP

We'll use the **Mean Absolute Percentage Error** (MAPE) against our full-domain ground-truth hardness map (`H_grid`) to quantify how well each strategy reconstructs the surface.

1. **Load ground truth**

```
# H_grid is a 1000×1000 DataFrame of true hardness values
Z_true_full = H_grid.values  # shape (1000,1000)

# down-sample / interpolate to our plot resolution (res×res):
from scipy.ndimage import zoom
Z_true = zoom(Z_true_full, res/1000, order=1)
```

Adaptive Drift-Hold & Active-Learning Nanoindentation

We combine an **active-learning sampling** framework with a **dynamic drift-hold policy** to minimize idle time while preserving precision, even when occasional "drift events" temporarily bias depth measurements.

---

1. Cost Model

Each block of measurements incurs three components:

1. **Setup & Travel**
   A fixed time (T_{\rm setup} + T_{\rm move}).

2. **Measurement + Drift-Hold**
   [T_{\rm meas} = T_{\rm base} + H,] where (T_{\rm base}) is the indentation time and (H\in{0,,5,,80}) s is the chosen hold.

3. **Return Penalty**
   A small fixed "come-back" time (T_{\rm comeback}).

Total block cost: [C = T_{\rm setup} + T_{\rm move} + (T_{\rm base}+H) + T_{\rm comeback}.]

---

2. Simulating a Drift Event

- **Onset**: at (t_{\rm event}) measurement depths acquire a one-sided bias that **decays exponentially**: [\varepsilon(t) =

$$\begin{cases} A\, e^{-(t-t_{\text{event}})/\tau}, & t_{\text{event}} \le t \le t_{\text{end}}, \\ 0, & \text{otherwise}, \end{cases}$$

  ] with amplitude (A) and time constant (\tau).

- **Interpretation**: small systematic depth error that gradually vanishes.

---

3. Drift-Hold Decision Logic

At each new block, let (T) be total elapsed time. We estimate:

- **Simulated drift error**
  (\mathrm{err}_5 = \varepsilon(T)), representing what a short (5 s) hold would leave uncorrected.

- **Intra-block variability**
  Sample a quick 5×5 grid to compute its standard deviation (\sigma). Define a “big” threshold [T_{\rm big} = \max!\bigl(p%\times\overline{H},;2,\sigma\bigr),] where (\overline{H}) is mean hardness and (p%) is a user-set tolerance.

Then choose hold time (H):

- If the previous hold was zero, we still force a short check every (k)th block.
- Otherwise: [H =

$$\begin{cases} 0, & \text{err}_5 \le \sigma, \\ 5, & \sigma < \text{err}_5 < T_{\text{big}}, \\ 80, & \text{err}_5 \ge T_{\text{big}}. \end{cases}$$

  ]

---

4. Bias Correction

When measuring with a non-full hold ((H=5) or 0 s) during an event window:

1. Generate raw depth (\tilde z) with base noise.
2. Draw a one-sided bias (\delta\in[0,\varepsilon(T)]).
3. Subtract it: [z_{\rm corrected} = \tilde z - \delta.]

For full holds ((H=80) s) we assume negligible residual bias.

---

5. Active-Learning Sampling

- Maintain two datasets:
  1. **Dynamic-hold** measurements.
  2. **Max-hold** (80 s) measurements as a high-fidelity reference.
- **Gaussian Process (GP)** regression is trained separately on each.
- **Acquisition function** for each candidate block: [\alpha = \frac{\text{GP uncertainty}}{\text{block cost}}.] The next block maximizes (\alpha), balancing information gain vs. time.

---

6. Evaluation & Visualization

Periodically we compare:

- **Predicted mean & uncertainty** maps from both GPs.
- **Measured series** vs. “ground truth” (max-hold branch).
- **Cumulative cost** curves to quantify time savings.
- **Difference map** (\displaystyle 100%\times\frac{Z_{\max} - Z_{\rm dyn}}{Z_{\max}}) and **MAPE** against a down-sampled truth field.

Takeaway

By **adapting drift-hold** based on measured error vs. variability, this strategy:

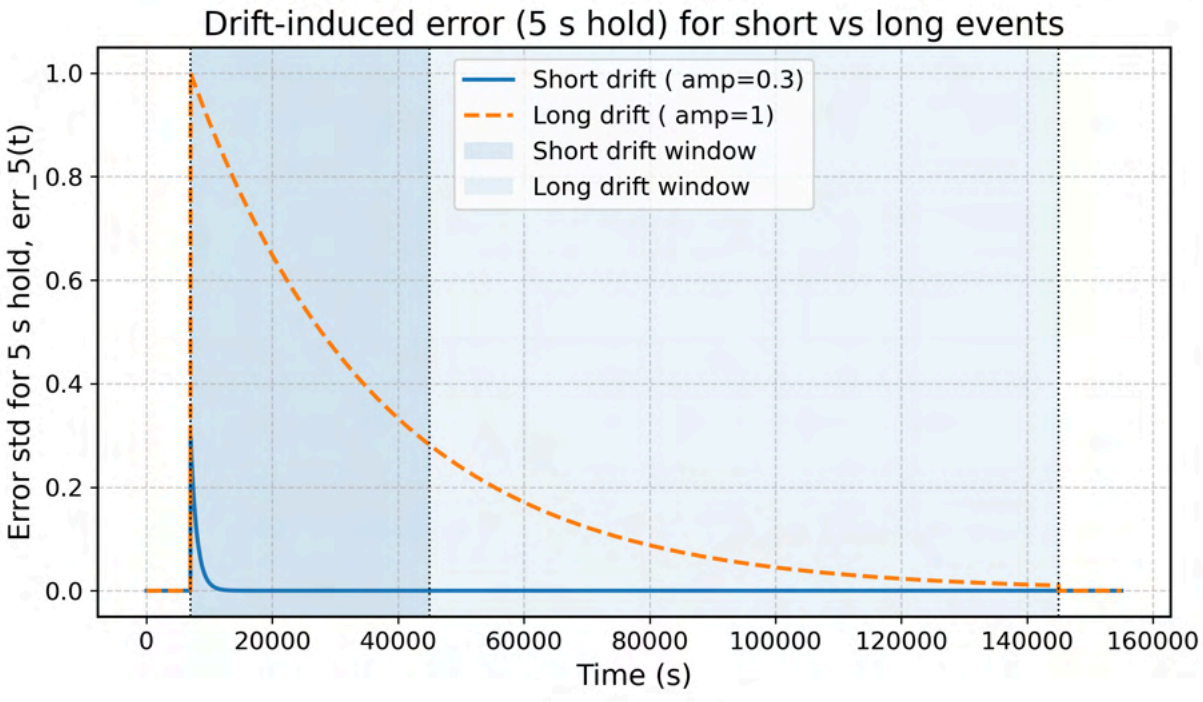
- **Saves** idle hold time when drift is small.
- **Escalates** to longer holds when drift threatens precision.
- **Integrates** seamlessly with active-learning to guide spatial sampling.

In [14]:
```
# === Constants ===
t_lift = 30
t_engage = 65
default_t_drift = 300
default_t_measurement = 120
default_t_lift = 25
t_comeback=35
```

In [23]:
```
import numpy as np
import matplotlib.pyplot as plt

# ---------- Scenario A: small/short drift event ----------
A = dict(
    name="Short drift",
    t_event=7000.0,
```

```
    t_event_end=45000.0,
    decay_tau=1000.0,
    bias_amp5=0.3,   # std at event start for 5 s hold
)

# ---------- Scenario B: long drift event ----------
B = dict(
    name="Long drift",
    t_event=7000.0,
    t_event_end=145000.0,
    decay_tau=30000.0,
    bias_amp5=1,   # keep same amplitude; change tau + window
    # if you want the 5.5*exp(...) flavor, set bias_amp5=5.5 here
)

def err5_vs_time(t, params):
    """err_5(t) = bias_amp5 * exp(-(t - t_event) / decay_tau) inside the window, el
    t0, t1 = params["t_event"], params["t_event_end"]
    tau, amp = params["decay_tau"], params["bias_amp5"]
    out = np.zeros_like(t, dtype=float)
    mask = (t >= t0) & (t <= t1)
    out[mask] = amp * np.exp(-(t[mask] - t0) / tau)
    return out, mask

# ---- Build a common time axis that covers both events ----
t_min = 0.0
t_max = max(A["t_event_end"], B["t_event_end"]) + 10000.0
t = np.linspace(t_min, t_max, 5000)

# ---- Compute curves ----
errA, maskA = err5_vs_time(t, A)
errB, maskB = err5_vs_time(t, B)

# ---- Plot ----
fig, ax = plt.subplots(figsize=(8.5, 5.0))

ax.plot(t, errA, label=f"{A['name']} ( amp={A['bias_amp5']})", linewidth=2)
ax.plot(t, errB, label=f"{B['name']} ( amp={B['bias_amp5']})", linewidth=2, linesty

# Shade each window (different alphas)
ax.axvspan(A["t_event"], A["t_event_end"], alpha=0.12, label=f"{A['name']} window")
ax.axvspan(B["t_event"], B["t_event_end"], alpha=0.08, label=f"{B['name']} window")

# Boundaries
ax.axvline(A["t_event"], color="k", linestyle=":", linewidth=1)
ax.axvline(A["t_event_end"], color="k", linestyle=":", linewidth=1)
ax.axvline(B["t_event"], color="k", linestyle=":", linewidth=1)
ax.axvline(B["t_event_end"], color="k", linestyle=":", linewidth=1)

ax.set_xlabel("Time (s)")
ax.set_ylabel("Error std for 5 s hold, err_5(t)")
ax.set_title("Drift-induced error (5 s hold) for short vs long events")
ax.grid(True, linestyle="--", alpha=0.6)
ax.legend(loc="best")

plt.tight_layout()
```

```
plt.show()

# # Optional: save instead of (or in addition to) showing
# plt.savefig("err5_vs_time_short_vs_long.png", dpi=200, bbox_inches="tight")
```



Low drift case

In [ ]:

```
import os  # NEW
import numpy as np
import torch
import matplotlib.pyplot as plt
from gpytorch.means import ConstantMean
from gpytorch.kernels import ScaleKernel, RBFKernel
import random
from scipy.ndimage import zoom
import pandas as pd  # NEW


SEED = 42
np.random.seed(SEED)
torch.manual_seed(SEED)
random.seed(SEED)

# === Output dir ===
OUTDIR = "outputs_dynamic_vs_max_good drift"  # NEW
os.makedirs(OUTDIR, exist_ok=True) # NEW

def to_np(x):
    """Return a NumPy array regardless of whether x is torch.Tensor or np.ndarray/l
    if isinstance(x, torch.Tensor):
        return x.detach().cpu().numpy()
    return np.asarray(x)
```

```
# === Configuration ===
domain_size           = 1000
default_grid_size     = 5
spacing               = 5
t_setup               = 20
t_move                = 20
default_t_drift       = 300
default_t_measurement = 40   # base measurement time
t_comeback            = 35
n_initial             = 5
n_steps               = 100

# Dynamic measurement hold times
min_hold      = 0
med_hold      = 5
max_hold      = 80
eval_interval = 5
plot_interval = 5
user_threshold_pct = 3.0  # % of mean hardness

# Drift-event parameters
t_event    = 7000.0
t_event_end= 45000.0
decay_tau  = 1000.0
bias_amp5  = 0.3   # one-sided bias std for 5 s hold
bias_amp0  = 0.5   # one-sided bias std for 0 s hold



# H_grid: your pandas DataFrame of shape (1000, 1000)
Z_full = H_grid.values                          # → (1000, 1000)
# res should match the resolution you use for your GP plots
res = 20                                       # for example
scale = res / Z_full.shape[0]                   # = 20 / 1000 = 0.02
# use scipy.ndimage.zoom to interpolate/down-sample
Z_true = zoom(Z_full, zoom=scale, order=1)     # → (res, res)

# Build grid centers
max_offset = (default_grid_size - 1) * spacing
grid_centers_x = np.linspace(0, domain_size - max_offset, 100)
grid_centers_y = np.linspace(0, domain_size - max_offset, 100)
available_centers = [(x, y) for x in grid_centers_x for y in grid_centers_y]

# Precompute local offsets for a grid
ix, iy = np.meshgrid(np.arange(default_grid_size),
                     np.arange(default_grid_size))
local_offsets = np.stack([ix.ravel(), iy.ravel()], axis=1) * spacing

# Storage for dynamic vs max-hold branches
all_x_dyn = []; all_y_dyn = []; all_z_dyn = []
all_x_max = []; all_y_max = []; all_z_max = []
visited_dyn = []; visited_max = []
cost_dyn = []; cost_max = []
total_cost_dyn = 0.0
total_cost_max = 0.0
```

```
rng = np.random.RandomState(0)

# === Metrics storage (for CSV + plots) ===  # NEW
metrics = []  # rows: dict(iter, cost_dyn, cost_max, mape_dyn, mape_max)

# === Helper: safe MAPE ===  # NEW
def mape_percent(y_true, y_pred, eps=1e-8):
    denom = np.maximum(np.abs(y_true), eps)
    return float(np.mean(np.abs((y_true - y_pred) / denom)) * 100.0)

# === Prepare a fixed test grid for GP predictions ===  # NEW
xt = np.linspace(0, domain_size, res)
yt = np.linspace(0, domain_size, res)
XX, YY = np.meshgrid(xt, yt)
test_pts = np.column_stack([XX.ravel(), YY.ravel()])
test_x_torch = torch.tensor(test_pts / domain_size, dtype=torch.float32)

# === 1) Initial warm-up sampling (use max_hold for both) ===
init_idxs = np.random.choice(len(available_centers),
                             n_initial, replace=False)

for idx in init_idxs:
    gx, gy = available_centers[idx]
    pos    = local_offsets + np.array([gx, gy])
    xs, ys = pos[:,0], pos[:,1]

    # simulate raw measurement
    zs0 = [measure_from_Hgrid(int(x), int(y), noise_std=1.4)
           for x,y in zip(xs, ys)]

    # dynamic branch: no bias on first round
    zs_dyn = np.array(zs0)
    zs_max = np.array(zs0)

    # record
    all_x_dyn.extend(xs); all_y_dyn.extend(ys); all_z_dyn.extend(zs_dyn)
    all_x_max.extend(xs); all_y_max.extend(ys); all_z_max.extend(zs_max)
    visited_dyn.append((gx,gy)); visited_max.append((gx,gy))

    # both use max_hold initially
    t_meas = default_t_measurement + max_hold
    c = evaluate_sample_cost_remaining(xs, ys,
                t_drift=default_t_drift,
                t_measurement=t_meas,
                t_comeback=t_comeback)[0]
    print(f"c is {c}")

    total_cost_dyn += t_setup + t_move + c
    total_cost_max += t_setup + t_move + c
    cost_dyn.append(total_cost_dyn)
    cost_max.append(total_cost_max)

# remove initial from availability
available_centers = [c for i,c in enumerate(available_centers)
                     if i not in init_idxs]
```

```
prev_hold = max_hold

# === 2) Active-learning loop with dynamic holds & bias ===
for step in range(n_steps):
    iter_idx = step + 1  # 1-based
    # print(f"1:{step}")

    # --- Fit GP on dynamic data ---
    X_dyn   = np.column_stack([all_x_dyn, all_y_dyn]) / domain_size
    z_dyn   = np.array(all_z_dyn)
    z_min_dyn, z_max_dyn = z_dyn.min(), z_dyn.max()
    train_x = torch.tensor(X_dyn, dtype=torch.float32)
    # guard if all equal
    denom_dyn = (z_max_dyn - z_min_dyn) if (z_max_dyn > z_min_dyn) else 1.0
    train_y = torch.tensor((z_dyn - z_min_dyn)/denom_dyn, dtype=torch.float32)
    inducing = train_x[:min(500, train_x.size(0))]
    gp_dyn = viGP(inducing,
                  mean_module=ConstantMean(),
                  kernel_module=ScaleKernel(RBFKernel()))
    gp_dyn.fit(train_x, train_y, training_iter=200)

    # --- Fit GP on max-hold data ---
    X_max   = np.column_stack([all_x_max, all_y_max]) / domain_size
    z_max_a = np.array(all_z_max)
    z_min_max, z_max_max = z_max_a.min(), z_max_a.max()
    train_xm = torch.tensor(X_max, dtype=torch.float32)
    denom_max = (z_max_max - z_min_max) if (z_max_max > z_min_max) else 1.0
    train_ym = torch.tensor((z_max_a - z_min_max)/denom_max, dtype=torch.float32)
    inducing_m = train_xm[:min(500, train_xm.size(0))]
    gp_max = viGP(inducing_m,
                  mean_module=ConstantMean(),
                  kernel_module=ScaleKernel(RBFKernel()))
    gp_max.fit(train_xm, train_ym, training_iter=200)
    # print(f"2:{step}")

    # --- Acquisition using prev_hold ---
    t_meas_prev = default_t_measurement + prev_hold
    acq_vals = []
    for gx, gy in available_centers:
        pos    = local_offsets + np.array([gx, gy])
        X_test = torch.tensor(pos / domain_size, dtype=torch.float32)
        unc    = UE(gp_dyn, X_test).mean().item()
        c0     = evaluate_sample_cost_remaining(pos[:,0], pos[:,1],
                          t_drift=default_t_drift,
                          t_measurement=t_meas_prev,
                          t_comeback=t_comeback)[0]
        acq_vals.append(unc / (t_setup + t_move + c0))
    best_idx = int(np.argmax(acq_vals))
    gx, gy = available_centers.pop(best_idx)
    visited_dyn.append((gx,gy))
    visited_max.append((gx,gy))

    # --- Determine hold for this block ---
    curr_t = total_cost_dyn
    in_event = (curr_t >= t_event) and (curr_t <= t_event_end)
    err_5    = 0.5 * np.exp(-(curr_t - t_event)/decay_tau) if in_event else 0.0
```

```
if prev_hold == 0:
    hold = med_hold if (iter_idx) % eval_interval == 0 else min_hold
else:
    pos      = local_offsets + np.array([gx, gy])
    hard_vals= np.array([measure_from_Hgrid(int(x), int(y),noise_std=1.4)
                         for x,y in zip(pos[:,0], pos[:,1])])
    T_var =  np.std(hard_vals)
    T_var = -1  # your logic
    T_big = max(user_threshold_pct*np.mean(hard_vals)/100,
                2*T_var)
    if err_5 <= T_var:
        hold = min_hold
    elif err_5 < T_big:
        hold = med_hold
    else:
        hold = max_hold

prev_hold = hold
# print(f"3:{step}")

# --- Final measurement with dynamic bias + max hold ---
pos    = local_offsets + np.array([gx, gy])
xs, ys = pos[:,0], pos[:,1]
zs0    = np.array([measure_from_Hgrid(int(x), int(y),
                    noise_std=1.4)
                   for x,y in zip(xs, ys)])

# compute event bias std
in_event = (curr_t >= t_event) and (curr_t <= t_event_end)
std5 = bias_amp5 * np.exp(-(curr_t - t_event)/decay_tau) if in_event else 0.0
std0 = bias_amp0 if in_event else 0.0

if hold == med_hold:
    biases = rng.uniform(-std5, std5, size=zs0.shape)
elif hold == min_hold:
    biases = rng.uniform(-std0, std0, size=zs0.shape)
else:
    biases = np.zeros_like(zs0)

zs_dyn_new = zs0 + biases
zs_max_new = zs0.copy()  # always no bias, σ=0.1

# append
all_x_dyn.extend(xs); all_y_dyn.extend(ys); all_z_dyn.extend(zs_dyn_new)
all_x_max.extend(xs); all_y_max.extend(ys); all_z_max.extend(zs_max_new)

# update dynamic cost
t_meas_u = default_t_measurement + hold
c_u      = evaluate_sample_cost_remaining(xs, ys,
             t_drift=default_t_drift,
             t_measurement=t_meas_u,
             t_comeback=t_comeback)[0]
total_cost_dyn += t_setup + t_move + c_u
cost_dyn.append(total_cost_dyn)
```

```
    # update max cost
    t_meas_m = default_t_measurement + max_hold
    c_m      = evaluate_sample_cost_remaining(xs, ys,
                 t_drift=default_t_drift,
                 t_measurement=t_meas_m,
                 t_comeback=t_comeback)[0]
    total_cost_max += t_setup + t_move + c_m
    cost_max.append(total_cost_max)

    # --- ALWAYS compute GP predictions + MAPE this iteration (save to metrics) ---
    # --- ALWAYS compute GP predictions + MAPE this iteration (save to metrics) ---
    with torch.no_grad():
        Z_dyn_pred = to_np(gp_dyn.predict(test_x_torch)).reshape(res, res)
        Z_dyn_pred = (Z_dyn_pred * denom_dyn + z_min_dyn)

        Z_max_pred = to_np(gp_max.predict(test_x_torch)).reshape(res, res)
        Z_max_pred = (Z_max_pred * denom_max + z_min_max)


    mape_dyn = mape_percent(Z_true, Z_dyn_pred)
    mape_max = mape_percent(Z_true, Z_max_pred)

    metrics.append({
        "iteration": iter_idx,
        "cost_dyn_s": total_cost_dyn,
        "cost_max_s": total_cost_max,
        "mape_dyn_%": mape_dyn,
        "mape_max_%": mape_max,
        "hold_s": hold
    })
    print(hold)
    print(total_cost_max)
    print(total_cost_dyn)
    # print(iter_idx % plot_interval)

    # --- Plot & SAVE every plot_interval iters ---
    if (iter_idx % plot_interval) == 0:
        # 2×2 plot
        fig, axs = plt.subplots(2,2,figsize=(12,10))

        im0 = axs[0,0].imshow(Z_dyn_pred, origin='lower',
                              extent=(0,domain_size,0,domain_size),
                              vmin=z_min_dyn, vmax=z_max_dyn, cmap='plasma')
        axs[0,0].set_title(f"Iter {iter_idx}: GP Pred (Dynamic)")
        axs[0,0].scatter(all_x_dyn, all_y_dyn, c=np.arange(len(all_x_dyn)),
                         cmap='jet', s=5)
        plt.colorbar(im0, ax=axs[0,0], shrink=0.8)

        # Compute UE first, then plot (use to_np!)
        UE_dyn_map = to_np(UE(gp_dyn, test_x_torch)).reshape(res, res)
        im1 = axs[0,1].imshow(UE_dyn_map, origin='lower',
                              extent=(0,domain_size,0,domain_size),
                              cmap='plasma')
        axs[0,1].set_title("GP Uncertainty (Dynamic)")
        if len(visited_dyn) > 0:
            dx, dy = zip(*visited_dyn)
```

```
        axs[0,1].scatter(dx, dy, c=np.arange(len(visited_dyn)),
                         cmap='jet', s=50)
    plt.colorbar(im1, ax=axs[0,1], shrink=0.8)

    idx = np.arange(len(all_z_dyn))
    z_gt  = np.array(all_z_max)  # using max-hold as proxy GT
    z_max_series = np.array(all_z_max)
    z_dyn_series = np.array(all_z_dyn)

    axs[1,0].plot(idx, z_gt,  'k-',  label='Ground Truth (max)')
    axs[1,0].plot(idx, z_max_series, 'r--', label='Max Hold')
    axs[1,0].plot(idx, z_dyn_series,'b-.', label='Dynamic Hold')
    axs[1,0].set_title("Measured vs. GT")
    axs[1,0].set_xlabel("Measurement #")
    axs[1,0].set_ylabel("Hardness")
    axs[1,0].legend(loc='best')
    axs[1,0].grid(True, linestyle='--', alpha=0.6)

    steps_arr = np.arange(1, len(cost_dyn)+1)
    axs[1,1].plot(steps_arr, cost_dyn, label='Dynamic Hold')
    axs[1,1].plot(steps_arr, cost_max, '--', label='Always 80s Hold')
    axs[1,1].set_title(f"Cumulative Cost at Iter {iter_idx}")
    axs[1,1].set_xlabel("Iteration")
    axs[1,1].set_ylabel("Cost (s)")
    axs[1,1].legend(loc='best')
    axs[1,1].grid(True, linestyle='--', alpha=0.6)

    plt.tight_layout()
    fig.savefig(os.path.join(OUTDIR, f"iter_{iter_idx:04d}_dynamic_2x2.png"),
                dpi=200, bbox_inches='tight')
    plt.close(fig)

    # max-hold GP alone
    fig2, (ax0, ax1) = plt.subplots(1,2,figsize=(12,5))
    im2 = ax0.imshow(Z_max_pred, origin='lower',
                     extent=(0,domain_size,0,domain_size),
                     vmin=z_min_max, vmax=z_max_max, cmap='plasma')
    ax0.set_title(f"Iter {iter_idx}: GP Pred (Max Hold)")
    ax0.scatter(all_x_max, all_y_max,
                c=np.arange(len(all_x_max)), cmap='jet', s=5)
    plt.colorbar(im2, ax=ax0, shrink=0.8)

    UE_max_map = to_np(UE(gp_max, test_x_torch)).reshape(res, res)
    im3 = ax1.imshow(UE_max_map, origin='lower',
                     extent=(0,domain_size,0,domain_size),
                     cmap='plasma')
    ax1.set_title("GP Uncertainty (Max Hold)")
    if len(visited_max) > 0:
        vxm, vym = zip(*visited_max)
        ax1.scatter(vxm, vym, c=np.arange(len(visited_max)),
                     cmap='jet', s=50)
    plt.colorbar(im3, ax=ax1, shrink=0.8)

    plt.tight_layout()
    fig2.savefig(os.path.join(OUTDIR, f"iter_{iter_idx:04d}_max_only.png"),
                dpi=200, bbox_inches='tight')
```

```
        plt.close(fig2)

        # Difference map (%)
        diff_map = 100 * (Z_max_pred - Z_dyn_pred) / np.maximum(np.abs(Z_max_pred),
        fig3, axd = plt.subplots(1,1,figsize=(6.5,5))
        imd = axd.imshow(diff_map, origin='lower',
                         extent=(0,domain_size,0,domain_size),
                         cmap='plasma')
        title = (f"Iter {iter_idx}: Difference (%)\n"
                 f"MAPE_dyn={mape_dyn:.2f}%, MAPE_max={mape_max:.2f}%")
        axd.set_title(title)
        plt.colorbar(imd, ax=axd, label='% difference')
        plt.tight_layout()
        fig3.savefig(os.path.join(OUTDIR, f"iter_{iter_idx:04d}_diff_map.png"),
                     dpi=200, bbox_inches='tight')
        plt.close(fig3)


# === AFTER LOOP: save metrics table and cost-vs-MAPE plot ===  # NEW
df_metrics = pd.DataFrame(metrics, columns=["iteration","cost_dyn_s","cost_max_s","
csv_path = os.path.join(OUTDIR, "metrics_mape_cost_per_iter.csv")
df_metrics.to_csv(csv_path, index=False)

# Cost vs MAPE figure
figc, axc = plt.subplots(figsize=(7.5,5.2))
axc.plot(df_metrics["cost_dyn_s"], df_metrics["mape_dyn_%"], marker='o', label="Dyn
axc.plot(df_metrics["cost_max_s"], df_metrics["mape_max_%"], marker='^', linestyle=
axc.set_xlabel("Cumulative cost (s)")
axc.set_ylabel("MAPE (%) vs Z_true")
axc.set_title("Cost vs MAPE (evaluated each iteration)")
axc.grid(True, linestyle='--', alpha=0.6)
axc.legend()
plt.tight_layout()
figc.savefig(os.path.join(OUTDIR, "cost_vs_mape.png"), dpi=200, bbox_inches='tight'
plt.close(figc)

print(f"[Saved] Table: {csv_path}")
print(f"[Saved] Cost-vs-MAPE plot: {os.path.join(OUTDIR, 'cost_vs_mape.png')}")
print(f"[Saved] Iteration figures every {plot_interval} iters in {OUTDIR}/")
```

```
c is 3495.6988028673877
c is 3495.6988028673877
c is 3495.6988028673877
c is 3495.6988028673877
c is 3495.6988028673877
5
21214.192817204326
19339.192817204326
```

High drift case

In [ ]:

```
import os  # NEW
import numpy as np
import torch
import matplotlib.pyplot as plt
from gpytorch.means import ConstantMean
```

```
from gpytorch.kernels import ScaleKernel, RBFKernel
import random
from scipy.ndimage import zoom
import pandas as pd  # NEW

# === Placeholder imports (implement these) ===
# from your_module import viGP, UE, measure_from_Hgrid, evaluate_sample_cost_remain
# from your_module import movement_time, t_engage, default_t_lift

SEED = 42
np.random.seed(SEED)
torch.manual_seed(SEED)
random.seed(SEED)

# === Output dir ===
OUTDIR = "outputs_dynamic_vs_max"  # NEW
os.makedirs(OUTDIR, exist_ok=True) # NEW

def to_np(x):
    """Return a NumPy array regardless of whether x is torch.Tensor or np.ndarray/l
    if isinstance(x, torch.Tensor):
        return x.detach().cpu().numpy()
    return np.asarray(x)


# === Configuration ===
domain_size           = 1000
default_grid_size     = 5
spacing               = 5
t_setup               = 20
t_move                = 20
default_t_drift       = 300
default_t_measurement = 40   # base measurement time
t_comeback            = 35
n_initial             = 5
n_steps               = 100

# Dynamic measurement hold times
min_hold      = 0
med_hold      = 5
max_hold      = 80
eval_interval = 5
plot_interval = 5
user_threshold_pct = 3.0  # % of mean hardness

# Drift-event parameters
t_event    = 7000.0
t_event_end= 145000.0
decay_tau  = 30000.0
bias_amp5  = 1   # one-sided bias std for 5 s hold
bias_amp0  = 0.5   # one-sided bias std for 0 s hold
```

```
# H_grid: your pandas DataFrame of shape (1000, 1000)
Z_full = H_grid.values                          # → (1000, 1000)
# res should match the resolution you use for your GP plots
res = 20                                       # for example
scale = res / Z_full.shape[0]                   # = 20 / 1000 = 0.02
# use scipy.ndimage.zoom to interpolate/down-sample
Z_true = zoom(Z_full, zoom=scale, order=1)     # → (res, res)

# Build grid centers
max_offset = (default_grid_size - 1) * spacing
grid_centers_x = np.linspace(0, domain_size - max_offset, 100)
grid_centers_y = np.linspace(0, domain_size - max_offset, 100)
available_centers = [(x, y) for x in grid_centers_x for y in grid_centers_y]

# Precompute local offsets for a grid
ix, iy = np.meshgrid(np.arange(default_grid_size),
                     np.arange(default_grid_size))
local_offsets = np.stack([ix.ravel(), iy.ravel()], axis=1) * spacing

# Storage for dynamic vs max-hold branches
all_x_dyn = []; all_y_dyn = []; all_z_dyn = []
all_x_max = []; all_y_max = []; all_z_max = []
visited_dyn = []; visited_max = []
cost_dyn = []; cost_max = []
total_cost_dyn = 0.0
total_cost_max = 0.0

rng = np.random.RandomState(0)

# === Metrics storage (for CSV + plots) ===  # NEW
metrics = []  # rows: dict(iter, cost_dyn, cost_max, mape_dyn, mape_max)

# === Helper: safe MAPE ===  # NEW
def mape_percent(y_true, y_pred, eps=1e-8):
    denom = np.maximum(np.abs(y_true), eps)
    return float(np.mean(np.abs((y_true - y_pred) / denom)) * 100.0)

# === Prepare a fixed test grid for GP predictions ===  # NEW
xt = np.linspace(0, domain_size, res)
yt = np.linspace(0, domain_size, res)
XX, YY = np.meshgrid(xt, yt)
test_pts = np.column_stack([XX.ravel(), YY.ravel()])
test_x_torch = torch.tensor(test_pts / domain_size, dtype=torch.float32)

# === 1) Initial warm-up sampling (use max_hold for both) ===
init_idxs = np.random.choice(len(available_centers),
                             n_initial, replace=False)

for idx in init_idxs:
    gx, gy = available_centers[idx]
    pos    = local_offsets + np.array([gx, gy])
    xs, ys = pos[:,0], pos[:,1]

    # simulate raw measurement
    zs0 = [measure_from_Hgrid(int(x), int(y), noise_std=1.4)
           for x,y in zip(xs, ys)]
```

```
    # dynamic branch: no bias on first round
    zs_dyn = np.array(zs0)
    zs_max = np.array(zs0)

    # record
    all_x_dyn.extend(xs); all_y_dyn.extend(ys); all_z_dyn.extend(zs_dyn)
    all_x_max.extend(xs); all_y_max.extend(ys); all_z_max.extend(zs_max)
    visited_dyn.append((gx,gy)); visited_max.append((gx,gy))

    # both use max_hold initially
    t_meas = default_t_measurement + max_hold
    c = evaluate_sample_cost_remaining(xs, ys,
                t_drift=default_t_drift,
                t_measurement=t_meas,
                t_comeback=t_comeback)[0]
    print(f"c is {c}")

    total_cost_dyn += t_setup + t_move + c
    total_cost_max += t_setup + t_move + c
    cost_dyn.append(total_cost_dyn)
    cost_max.append(total_cost_max)

# remove initial from availability
available_centers = [c for i,c in enumerate(available_centers)
                     if i not in init_idxs]

prev_hold = max_hold

# === 2) Active-learning loop with dynamic holds & bias ===
for step in range(n_steps):
    iter_idx = step + 1  # 1-based
    # print(f"1:{step}")

    # --- Fit GP on dynamic data ---
    X_dyn   = np.column_stack([all_x_dyn, all_y_dyn]) / domain_size
    z_dyn   = np.array(all_z_dyn)
    z_min_dyn, z_max_dyn = z_dyn.min(), z_dyn.max()
    train_x = torch.tensor(X_dyn, dtype=torch.float32)
    # guard if all equal
    denom_dyn = (z_max_dyn - z_min_dyn) if (z_max_dyn > z_min_dyn) else 1.0
    train_y = torch.tensor((z_dyn - z_min_dyn)/denom_dyn, dtype=torch.float32)
    inducing = train_x[:min(500, train_x.size(0))]
    gp_dyn = viGP(inducing,
                  mean_module=ConstantMean(),
                  kernel_module=ScaleKernel(RBFKernel()))
    gp_dyn.fit(train_x, train_y, training_iter=200)

    # --- Fit GP on max-hold data ---
    X_max   = np.column_stack([all_x_max, all_y_max]) / domain_size
    z_max_a = np.array(all_z_max)
    z_min_max, z_max_max = z_max_a.min(), z_max_a.max()
    train_xm = torch.tensor(X_max, dtype=torch.float32)
    denom_max = (z_max_max - z_min_max) if (z_max_max > z_min_max) else 1.0
    train_ym = torch.tensor((z_max_a - z_min_max)/denom_max, dtype=torch.float32)
    inducing_m = train_xm[:min(500, train_xm.size(0))]
```

```
    gp_max = viGP(inducing_m,
                  mean_module=ConstantMean(),
                  kernel_module=ScaleKernel(RBFKernel()))
    gp_max.fit(train_xm, train_ym, training_iter=200)
    # print(f"2:{step}")

    # --- Acquisition using prev_hold ---
    t_meas_prev = default_t_measurement + prev_hold
    acq_vals = []
    for gx, gy in available_centers:
        pos    = local_offsets + np.array([gx, gy])
        X_test = torch.tensor(pos / domain_size, dtype=torch.float32)
        unc    = UE(gp_dyn, X_test).mean().item()
        c0     = evaluate_sample_cost_remaining(pos[:,0], pos[:,1],
                          t_drift=default_t_drift,
                          t_measurement=t_meas_prev,
                          t_comeback=t_comeback)[0]
        acq_vals.append(unc / (t_setup + t_move + c0))
    best_idx = int(np.argmax(acq_vals))
    gx, gy = available_centers.pop(best_idx)
    visited_dyn.append((gx,gy))
    visited_max.append((gx,gy))

    # --- Determine hold for this block ---
    curr_t = total_cost_dyn
    in_event = (curr_t >= t_event) and (curr_t <= t_event_end)
    err_5    = 5.5 * np.exp(-(curr_t - t_event)/decay_tau) if in_event else 0.0

    if prev_hold == 0:
        hold = med_hold if (iter_idx) % eval_interval == 0 else min_hold
    else:
        pos      = local_offsets + np.array([gx, gy])
        hard_vals= np.array([measure_from_Hgrid(int(x), int(y),noise_std=1.4)
                             for x,y in zip(pos[:,0], pos[:,1])])
        T_var =  np.std(hard_vals)
        T_var = -1  # your logic
        T_big = max(user_threshold_pct*np.mean(hard_vals)/100,
                    2*T_var)
        if err_5 <= T_var:
            hold = min_hold
        elif err_5 < T_big:
            hold = med_hold
        else:
            hold = max_hold

    prev_hold = hold
    # print(f"3:{step}")

    # --- Final measurement with dynamic bias + max hold ---
    pos    = local_offsets + np.array([gx, gy])
    xs, ys = pos[:,0], pos[:,1]
    zs0    = np.array([measure_from_Hgrid(int(x), int(y),
                           noise_std=1.4)
                       for x,y in zip(xs, ys)])

    # compute event bias std
```

```
    in_event = (curr_t >= t_event) and (curr_t <= t_event_end)
    std5 = bias_amp5 * np.exp(-(curr_t - t_event)/decay_tau) if in_event else 0.0
    std0 = bias_amp0 if in_event else 0.0

    if hold == med_hold:
        biases = rng.uniform(-std5, std5, size=zs0.shape)
    elif hold == min_hold:
        biases = rng.uniform(-std0, std0, size=zs0.shape)
    else:
        biases = np.zeros_like(zs0)

    zs_dyn_new = zs0 + biases
    zs_max_new = zs0.copy()  # always no bias, σ=0.1

    # append
    all_x_dyn.extend(xs); all_y_dyn.extend(ys); all_z_dyn.extend(zs_dyn_new)
    all_x_max.extend(xs); all_y_max.extend(ys); all_z_max.extend(zs_max_new)

    # update dynamic cost
    t_meas_u = default_t_measurement + hold
    c_u      = evaluate_sample_cost_remaining(xs, ys,
                 t_drift=default_t_drift,
                 t_measurement=t_meas_u,
                 t_comeback=t_comeback)[0]
    total_cost_dyn += t_setup + t_move + c_u
    cost_dyn.append(total_cost_dyn)

    # update max cost
    t_meas_m = default_t_measurement + max_hold
    c_m      = evaluate_sample_cost_remaining(xs, ys,
                 t_drift=default_t_drift,
                 t_measurement=t_meas_m,
                 t_comeback=t_comeback)[0]
    total_cost_max += t_setup + t_move + c_m
    cost_max.append(total_cost_max)

    # --- ALWAYS compute GP predictions + MAPE this iteration (save to metrics) ---
    # --- ALWAYS compute GP predictions + MAPE this iteration (save to metrics) ---
    with torch.no_grad():
        Z_dyn_pred = to_np(gp_dyn.predict(test_x_torch)).reshape(res, res)
        Z_dyn_pred = (Z_dyn_pred * denom_dyn + z_min_dyn)

        Z_max_pred = to_np(gp_max.predict(test_x_torch)).reshape(res, res)
        Z_max_pred = (Z_max_pred * denom_max + z_min_max)


    mape_dyn = mape_percent(Z_true, Z_dyn_pred)
    mape_max = mape_percent(Z_true, Z_max_pred)

    metrics.append({
        "iteration": iter_idx,
        "cost_dyn_s": total_cost_dyn,
        "cost_max_s": total_cost_max,
        "mape_dyn_%": mape_dyn,
        "mape_max_%": mape_max,
        "hold_s": hold
```

```
    })
    print(hold)
    print(total_cost_max)
    print(total_cost_dyn)
    # print(iter_idx % plot_interval)

    # --- Plot & SAVE every plot_interval iters ---
    if (iter_idx % plot_interval) == 0:
        # 2×2 plot
        fig, axs = plt.subplots(2,2,figsize=(12,10))

        im0 = axs[0,0].imshow(Z_dyn_pred, origin='lower',
                              extent=(0,domain_size,0,domain_size),
                              vmin=z_min_dyn, vmax=z_max_dyn, cmap='plasma')
        axs[0,0].set_title(f"Iter {iter_idx}: GP Pred (Dynamic)")
        axs[0,0].scatter(all_x_dyn, all_y_dyn, c=np.arange(len(all_x_dyn)),
                         cmap='jet', s=5)
        plt.colorbar(im0, ax=axs[0,0], shrink=0.8)

        # Compute UE first, then plot (use to_np!)
        UE_dyn_map = to_np(UE(gp_dyn, test_x_torch)).reshape(res, res)
        im1 = axs[0,1].imshow(UE_dyn_map, origin='lower',
                              extent=(0,domain_size,0,domain_size),
                              cmap='plasma')
        axs[0,1].set_title("GP Uncertainty (Dynamic)")
        if len(visited_dyn) > 0:
            dx, dy = zip(*visited_dyn)
            axs[0,1].scatter(dx, dy, c=np.arange(len(visited_dyn)),
                             cmap='jet', s=50)
        plt.colorbar(im1, ax=axs[0,1], shrink=0.8)

        idx = np.arange(len(all_z_dyn))
        z_gt  = np.array(all_z_max)  # using max-hold as proxy GT
        z_max_series = np.array(all_z_max)
        z_dyn_series = np.array(all_z_dyn)

        axs[1,0].plot(idx, z_gt,  'k-',  label='Ground Truth (max)')
        axs[1,0].plot(idx, z_max_series, 'r--', label='Max Hold')
        axs[1,0].plot(idx, z_dyn_series,'b-.', label='Dynamic Hold')
        axs[1,0].set_title("Measured vs. GT")
        axs[1,0].set_xlabel("Measurement #")
        axs[1,0].set_ylabel("Hardness")
        axs[1,0].legend(loc='best')
        axs[1,0].grid(True, linestyle='--', alpha=0.6)

        steps_arr = np.arange(1, len(cost_dyn)+1)
        axs[1,1].plot(steps_arr, cost_dyn, label='Dynamic Hold')
        axs[1,1].plot(steps_arr, cost_max, '--', label='Always 80s Hold')
        axs[1,1].set_title(f"Cumulative Cost at Iter {iter_idx}")
        axs[1,1].set_xlabel("Iteration")
        axs[1,1].set_ylabel("Cost (s)")
        axs[1,1].legend(loc='best')
        axs[1,1].grid(True, linestyle='--', alpha=0.6)

        plt.tight_layout()
        fig.savefig(os.path.join(OUTDIR, f"iter_{iter_idx:04d}_dynamic_2x2.png"),
```

```
                    dpi=200, bbox_inches='tight')
        plt.close(fig)

        # max-hold GP alone
        fig2, (ax0, ax1) = plt.subplots(1,2,figsize=(12,5))
        im2 = ax0.imshow(Z_max_pred, origin='lower',
                         extent=(0,domain_size,0,domain_size),
                         vmin=z_min_max, vmax=z_max_max, cmap='plasma')
        ax0.set_title(f"Iter {iter_idx}: GP Pred (Max Hold)")
        ax0.scatter(all_x_max, all_y_max,
                    c=np.arange(len(all_x_max)), cmap='jet', s=5)
        plt.colorbar(im2, ax=ax0, shrink=0.8)

        UE_max_map = to_np(UE(gp_max, test_x_torch)).reshape(res, res)
        im3 = ax1.imshow(UE_max_map, origin='lower',
                         extent=(0,domain_size,0,domain_size),
                         cmap='plasma')
        ax1.set_title("GP Uncertainty (Max Hold)")
        if len(visited_max) > 0:
            vxm, vym = zip(*visited_max)
            ax1.scatter(vxm, vym, c=np.arange(len(visited_max)),
                         cmap='jet', s=50)
        plt.colorbar(im3, ax=ax1, shrink=0.8)

        plt.tight_layout()
        fig2.savefig(os.path.join(OUTDIR, f"iter_{iter_idx:04d}_max_only.png"),
                    dpi=200, bbox_inches='tight')
        plt.close(fig2)

        # Difference map (%)
        diff_map = 100 * (Z_max_pred - Z_dyn_pred) / np.maximum(np.abs(Z_max_pred),
        fig3, axd = plt.subplots(1,1,figsize=(6.5,5))
        imd = axd.imshow(diff_map, origin='lower',
                         extent=(0,domain_size,0,domain_size),
                         cmap='plasma')
        title = (f"Iter {iter_idx}: Difference (%)\n"
                f"MAPE_dyn={mape_dyn:.2f}%, MAPE_max={mape_max:.2f}%")
        axd.set_title(title)
        plt.colorbar(imd, ax=axd, label='% difference')
        plt.tight_layout()
        fig3.savefig(os.path.join(OUTDIR, f"iter_{iter_idx:04d}_diff_map.png"),
                    dpi=200, bbox_inches='tight')
        plt.close(fig3)


# === AFTER LOOP: save metrics table and cost-vs-MAPE plot ===  # NEW
df_metrics = pd.DataFrame(metrics, columns=["iteration","cost_dyn_s","cost_max_s","
csv_path = os.path.join(OUTDIR, "metrics_mape_cost_per_iter.csv")
df_metrics.to_csv(csv_path, index=False)

# Cost vs MAPE figure
figc, axc = plt.subplots(figsize=(7.5,5.2))
axc.plot(df_metrics["cost_dyn_s"], df_metrics["mape_dyn_%"], marker='o', label="Dyn
axc.plot(df_metrics["cost_max_s"], df_metrics["mape_max_%"], marker='^', linestyle=
axc.set_xlabel("Cumulative cost (s)")
axc.set_ylabel("MAPE (%) vs Z_true")
```

```
axc.set_title("Cost vs MAPE (evaluated each iteration)")
axc.grid(True, linestyle='--', alpha=0.6)
axc.legend()
plt.tight_layout()
figc.savefig(os.path.join(OUTDIR, "cost_vs_mape.png"), dpi=200, bbox_inches='tight'
plt.close(figc)

print(f"[Saved] Table: {csv_path}")
print(f"[Saved] Cost-vs-MAPE plot: {os.path.join(OUTDIR, 'cost_vs_mape.png')}")
print(f"[Saved] Iteration figures every {plot_interval} iters in {OUTDIR}/")
```

Heteroskedastic GP fro noise awareness

In [7]:

```
import numpy as np
import torch
import matplotlib.pyplot as plt
import gpytorch
from gpytorch.models import ApproximateGP
from gpytorch.variational import VariationalStrategy, CholeskyVariationalDistributi
from gpytorch.kernels import ScaleKernel, RBFKernel
from gpytorch.means import ConstantMean
from torch.distributions import Normal

# 1) Heteroscedastic GP definition
class GPModel(ApproximateGP):
    def __init__(self, inducing_points):
        variational_dist = CholeskyVariationalDistribution(inducing_points.size(0))
        variational_strategy = VariationalStrategy(
            self, inducing_points, variational_dist, learn_inducing_locations=True
        )
        super().__init__(variational_strategy)
        self.mean_module = ConstantMean()
        self.covar_module = ScaleKernel(RBFKernel())

    def forward(self, x):
        return gpytorch.distributions.MultivariateNormal(
            self.mean_module(x), self.covar_module(x)
        )

def heteroscedastic_elbo(mean_gp, noise_gp, train_x, train_y, num_samples=50):
    q_f        = mean_gp(train_x)
    q_log_noise= noise_gp(train_x)
    kl_f       = mean_gp.variational_strategy.kl_divergence()
    kl_noise   = noise_gp.variational_strategy.kl_divergence()
    # Monte Carlo estimate of E_q[log p(y|f,σ)]
    log_liks = []
    for _ in range(num_samples):
        f_samp      = q_f.rsample()
        log_noise_s = q_log_noise.rsample()
        noise_var   = torch.exp(log_noise_s) + 1e-6
        dist        = Normal(f_samp, noise_var.sqrt())
        log_liks.append(dist.log_prob(train_y).sum())
    expected_ll = torch.stack(log_liks).mean()
    elbo        = expected_ll - kl_f - kl_noise
```

```
    return -elbo

def train_hetero(train_x, train_y, mean_gp, noise_gp, lr=0.01, iters=300):
    mean_gp.train(); noise_gp.train()
    optimizer = torch.optim.Adam(
        list(mean_gp.parameters()) + list(noise_gp.parameters()), lr=lr
    )
    for i in range(iters):
        optimizer.zero_grad()
        loss = heteroscedastic_elbo(mean_gp, noise_gp, train_x, train_y)
        loss.backward()
        optimizer.step()

def predict_hetero(mean_gp, noise_gp, test_x):
    mean_gp.eval(); noise_gp.eval()
    with torch.no_grad():
        f_dist   = mean_gp(test_x)
        n_dist   = noise_gp(test_x)
        mean     = f_dist.mean.cpu().numpy()
        noise    = torch.exp(n_dist.mean).cpu().numpy()
        var      = f_dist.variance.cpu().numpy()
    return mean, noise, var
```

```
c:\Users\vchawla\OneDrive - University of Tennessee\Documents\DOCUMENTS\Python\Pytorch_ML\pytor2\Lib\site-packages\tqdm\auto.py:21: TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

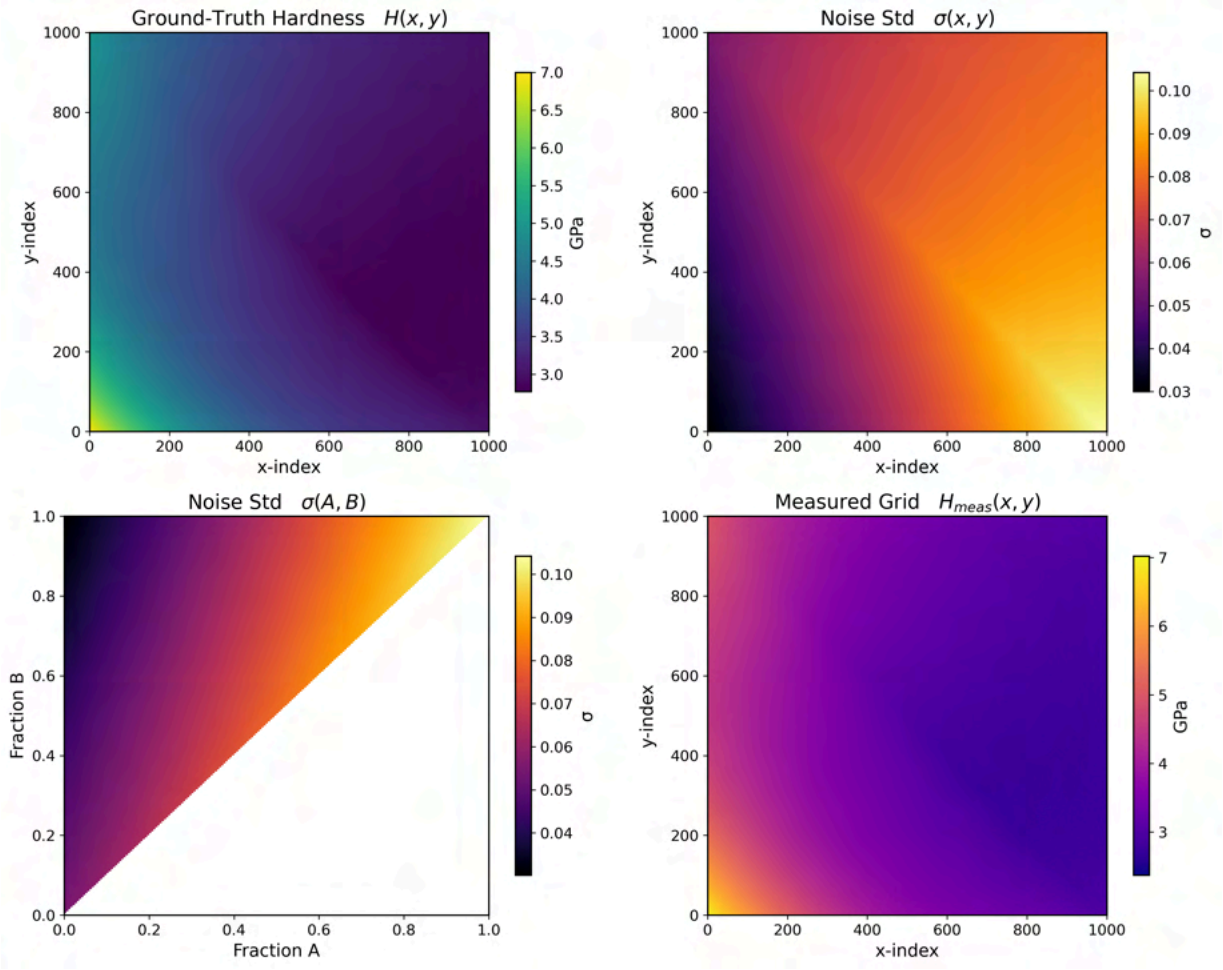
Noisy Ground Truth

In [8]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from  scipy.stats import binned_statistic_2d

# ------------------------------------------------------------------
# 1.  Build 1000 × 1000 ternary-composition grid + ground-truth hardness
# ------------------------------------------------------------------
nx = ny = 1000
x_coords, y_coords = np.meshgrid(np.linspace(0, 1, nx),
                                 np.linspace(0, 1, ny))
A = x_coords
B = y_coords
C = np.maximum(1.0 - A - B, 0.01)          # keep C ≥ 0
total = A + B + C                          # renormalise so A+B+C = 1
A /= total ;  B /= total ;  C /= total

def hardness_model(a,b,c):
    base         = 3*a + 5*b + 7*c                    # rule-of-mixtures
    ss_strength  = -4*a*b - 3*b*c - 2*c*a             # solid-solution term
    saturation   = -6*c**2 * (1 - c)                  # saturation penalty
    return base + ss_strength + saturation
```

```
H_grid = hardness_model(A,B,C)                          # ground-truth hardness

# ------------------------------------------------------------------
# 2.  Composition-dependent noise:  σ(A,B,C)
# ------------------------------------------------------------------
σ0, αA, αB, αC = 0.005, 0.1, 0.05, 0.0250
noise_std_grid = σ0 + αA*A + αB*B + αC*C                # same shape as H_grid

# ------------------------------------------------------------------
# 3.  One synthetic "measured" grid  (H + Gaussian noise)
# ------------------------------------------------------------------
rng = np.random.default_rng(123)
H_meas = H_grid + rng.normal(0, noise_std_grid)         # element-wise σ

# ------------------------------------------------------------------
# 4.  Re-bin σ onto a regular (A,B) composition grid for plotting
# ------------------------------------------------------------------
nbins = 200                                             # resolution in A–B space
σ_AB, A_edges, B_edges, _ = binned_statistic_2d(
        A.ravel(), B.ravel(), noise_std_grid.ravel(),
        statistic='mean', bins=nbins, range=[[0,1],[0,1]])
σ_AB = np.flipud(σ_AB.T)                                # put origin bottom-left

# ------------------------------------------------------------------
# 5.  Plot:  2 × 2   (all imshow, no 3-D)
# ------------------------------------------------------------------
fig, axs = plt.subplots(2, 2, figsize=(14, 11))

# (0,0) ground-truth hardness in x–y space
im0 = axs[0,0].imshow(H_grid, origin='lower', cmap='viridis',
                      extent=(0, nx, 0, ny))
axs[0,0].set_title("Ground-Truth Hardness   $H(x,y)$")
axs[0,0].set_xlabel("x-index") ; axs[0,0].set_ylabel("y-index")
fig.colorbar(im0, ax=axs[0,0], shrink=0.8, label="GPa")

# (0,1) noise σ in x–y space
im1 = axs[0,1].imshow(noise_std_grid, origin='lower', cmap='inferno',
                      extent=(0, nx, 0, ny))
axs[0,1].set_title("Noise Std   $\\sigma(x,y)$")
axs[0,1].set_xlabel("x-index") ; axs[0,1].set_ylabel("y-index")
fig.colorbar(im1, ax=axs[0,1], shrink=0.8, label="σ")

# (1,0) noise σ on regular composition grid (Fraction A vs Fraction B)
im2 = axs[1,0].imshow(σ_AB, origin='lower', cmap='inferno',
                      extent=(0,1,0,1), aspect='auto')
axs[1,0].set_title(r"Noise Std   $\sigma(A,B)$")
axs[1,0].set_xlabel("Fraction A") ; axs[1,0].set_ylabel("Fraction B")
fig.colorbar(im2, ax=axs[1,0], shrink=0.8, label="σ")

# (1,1) one synthetic measured grid  (H + noise)
im3 = axs[1,1].imshow(H_meas, origin='lower', cmap='plasma',
                      extent=(0, nx, 0, ny))
axs[1,1].set_title("Measured Grid   $H_{meas}(x,y)$")
axs[1,1].set_xlabel("x-index") ; axs[1,1].set_ylabel("y-index")
fig.colorbar(im3, ax=axs[1,1], shrink=0.8, label="GPa")
```

```
plt.tight_layout()
plt.show()
```



Adaptive Grid based on Target SEM

In [9]:
```
##############################################################################
#  ADAPTIVE-GRID  +  DYNAMIC-HOLD  +  HETEROSCEDASTIC-GP  SAMPLER  (SAVING)
##############################################################################
import os
import numpy as np, torch, random, matplotlib.pyplot as plt
from scipy.ndimage import zoom
import pandas as pd

# ———————— reproducibility ————————————————————————————————————
SEED = 42
np.random.seed(SEED); torch.manual_seed(SEED); random.seed(SEED)

# ———————— output ———————————————————————————————————————————
OUTDIR = "outputs_adaptive_grid_dynamic_hetero"
OUTDIR = "outputs_adaptive_grid_dynamic_hetero"
os.makedirs(OUTDIR, exist_ok=True)

# ———————— fixed parameters ——————————————————————————————————
domain_size           = 1000          # µm (for normalising GP inputs)
t_setup, t_move       = 20, 20        # s
```

```
default_t_drift       = 300          # s
default_t_measurement = 40           # s
t_comeback            = 35           # s

# grid adaptation
g_min, g_max          = 2, 5         # min / max square-grid dimension
target_sem            = 0.05         # GPa desired SEM
n_initial, n_steps    = 5, 100       # ← 100 iterations as requested
plot_interval         = 10           # save plots every N blocks

# dynamic hold thresholds
min_hold, med_hold, max_hold = 0, 5, 80
eval_interval         = 5            # force a 5-s check every N blocks
user_threshold_pct    = 4.0          # % of μ to define T_big

# drift-event (simulated one-sided bias)
t_event, t_event_end  = 7_000., 45_000.
decay_tau             = 1_000.
bias_amp5, bias_amp0  = 0.3, 0.5

# ————————— helper: heteroscedastic noise & measurement ————————————————
σ0, αA, αB, αC = 0.005, 0.1, 0.05, 0.0250
ny, nx = H_grid.shape

def noise_std_comp(xi:int, yi:int) -> float:
    xi = int(np.clip(xi, 0, nx-1))
    yi = int(np.clip(yi, 0, ny-1))
    return σ0 + αA*A[yi,xi] + αB*B[yi,xi] + αC*C[yi,xi]

def measure_from_Hgrid_hetero(xi:int, yi:int) -> float:
    xi = int(np.clip(xi, 0, nx-1))
    yi = int(np.clip(yi, 0, ny-1))
    H  = H_grid[yi, xi]
    σ  = noise_std_comp(xi, yi)
    return H + np.random.normal(0, σ)

# ————————— reference ground-truth image (for diagnostics) ————————————————
res     = 25
Z_true  = zoom(H_grid, zoom=res/ny, order=1)          # 25×25 GT map

# ————————— catalogue of possible  Block-centres  (5-μm lattice) ————————————
spacing      = 5
max_offset   = (g_max-1)*spacing
centres      = [(x, y) for x in np.linspace(0, domain_size-max_offset, 100)
                       for y in np.linspace(0, domain_size-max_offset, 100)]

# 5×5 offset template (slice [:g**2] for smaller grids)
ix, iy        = np.meshgrid(np.arange(g_max), np.arange(g_max))
offsets_5x5   = np.stack([ix.ravel(), iy.ravel()], axis=1)*spacing
offsets_ref   = offsets_5x5.copy()                    # always 25 pts

# ————————— storage arrays ————————————————————————————————————————————
all_x, all_y, all_z = [], [], []           # dynamic branch data
cost_dyn, cost_max  = [], []
tot_cost_dyn = tot_cost_max = 0.0
visited = []
```

```
# ————————— metrics table ————————————————————————————————————————
metrics = []  # rows per block: iteration, grid_size, n_pts, hold, costs, MAPE, MAE

def mape_percent(y_true, y_pred, eps=1e-8):
    denom = np.maximum(np.abs(y_true), eps)
    return float(np.mean(np.abs((y_true - y_pred) / denom)) * 100.0)

def mae(y_true, y_pred):
    return float(np.mean(np.abs(y_true - y_pred)))

# ————————— 1) initial 5×5 blocks (80-s hold) ————————————————————————
init_idxs = np.random.choice(len(centres), n_initial, replace=False)
for idx in init_idxs:
    cx,cy   = centres[idx]
    blk_xy  = offsets_5x5 + np.array([cx,cy])
    xs,ys   = blk_xy[:,0], blk_xy[:,1]
    zs      = [measure_from_Hgrid_hetero(x,y) for x,y in zip(xs,ys)]

    all_x.extend(xs); all_y.extend(ys); all_z.extend(zs); visited.append((cx,cy))

    t_meas  = default_t_measurement + max_hold
    move_cost  = evaluate_sample_cost_remaining(xs,ys,
                  t_drift=default_t_drift,
                  t_measurement=t_meas,
                  t_comeback=t_comeback)[0]
    add_c   = t_setup + t_move + move_cost
    tot_cost_dyn += add_c; tot_cost_max += add_c
    cost_dyn.append(tot_cost_dyn); cost_max.append(tot_cost_max)

centres   = [c for i,c in enumerate(centres) if i not in init_idxs]
prev_hold = max_hold
g_curr    = g_max                                   # start 5×5

# ————————— optional: baseline “iter 0” diagnostics ————————————————————
def diagnose_and_save(iter_idx, gp_m, gp_s, suffix=""):
    grid = np.linspace(0,domain_size,res)
    XX,YY = np.meshgrid(grid,grid)
    test  = torch.tensor(np.column_stack([XX.ravel(),YY.ravel()])/domain_size,
                         dtype=torch.float32)
    Zμ, Zσ, _ = predict_hetero(gp_m, gp_s, test)
    Zμ = Zμ.reshape(res,res); Zσ = Zσ.reshape(res,res)
    mape = mape_percent(Z_true, Zμ); mae_v = mae(Z_true, Zμ)

    fig,axs = plt.subplots(2,2,figsize=(13,10))
    im0 = axs[0,0].imshow(Zμ,origin='lower',
                          extent=(0,domain_size,0,domain_size),cmap='plasma')
    axs[0,0].set_title(f'Iter {iter_idx}: GP mean');     fig.colorbar(im0,ax=axs[0,
    im1 = axs[0,1].imshow(Zσ,origin='lower',
                          extent=(0,domain_size,0,domain_size),cmap='plasma')
    axs[0,1].set_title('Predicted σ');                   fig.colorbar(im1,ax=axs[0,

    axs[1,0].plot(np.arange(len(all_z)),all_z,'.',ms=2)
    axs[1,0].set_title('Dynamic measurements'); axs[1,0].set_xlabel('indent #')
    axs[1,0].set_ylabel('Hardness (GPa)')
```

```
    axs[1,1].plot(cost_dyn,label='Dynamic')
    axs[1,1].plot(cost_max,'--',label='Fixed 5×5 / 80 s')
    axs[1,1].set_title(f'Cumulative cost   MAPE={mape:.2f}% | MAE={mae_v:.4f}')
    axs[1,1].set_xlabel('block'); axs[1,1].set_ylabel('time (s)')
    axs[1,1].legend(); axs[1,1].grid(ls='--',alpha=.5)

    plt.tight_layout()
    fig.savefig(os.path.join(OUTDIR, f"iter_{iter_idx:04d}{suffix}_diagnostics.png"
                dpi=200, bbox_inches='tight')
    plt.close(fig)
    return mape, mae_v

# Warm-up fit just to produce iter 0000 figure
if len(all_x) > 0:
    X0 = torch.tensor(np.column_stack([all_x,all_y])/domain_size, dtype=torch.float
    y0 = torch.tensor(np.array(all_z), dtype=torch.float32)
    M0 = min(300, X0.size(0))
    inducing0 = X0[torch.randperm(X0.size(0))[:M0]]
    gp0_m, gp0_s = GPModel(inducing0), GPModel(inducing0)
    train_hetero(X0, y0, gp0_m, gp0_s, iters=200, lr=1e-2)
    diagnose_and_save(0, gp0_m, gp0_s, suffix="_warmup")

# ———————— 2) active-learning loop ————————————————————————————
for step in range(n_steps):
    iter_idx = step + 1

    # – fit heteroscedastic GP on current dynamic data –
    X = torch.tensor(np.column_stack([all_x,all_y])/domain_size, dtype=torch.float3
    y = torch.tensor(np.array(all_z), dtype=torch.float32)
    M = min(300, X.size(0))
    inducing = X[torch.randperm(X.size(0))[:M]]
    gp_m, gp_s = GPModel(inducing), GPModel(inducing)
    train_hetero(X, y, gp_m, gp_s, iters=300, lr=1e-2)

    # – acquisition: UE / cost for every candidate centre –
    acq_vals = []
    for cx,cy in centres:
        blk_xy  = offsets_5x5[:g_curr**2] + np.array([cx,cy])
        Xtest   = torch.tensor(blk_xy/domain_size, dtype=torch.float32)
        μ, σ, var  = predict_hetero(gp_m, gp_s, Xtest)
        unc     = (σ+var).mean()

        est_tmeas = default_t_measurement + prev_hold
        est_cost  = evaluate_sample_cost_remaining(
                        blk_xy[:,0], blk_xy[:,1],
                        t_drift=default_t_drift,
                        t_measurement=est_tmeas,
                        t_comeback=t_comeback)[0]
        acq_vals.append(unc / (t_setup+t_move+est_cost))

    # pick best centre
    best   = int(np.argmax(acq_vals))
    cx,cy  = centres.pop(best)
    visited.append((cx,cy))

    # – block-specific σ → grid size  (adaptive grid) –
```

```
    blk_xy   = offsets_5x5 + np.array([cx,cy])          # 25 candidate pts
    X_blk    = torch.tensor(blk_xy/domain_size, dtype=torch.float32)
    _, σ_blk, _ = predict_hetero(gp_m, gp_s, X_blk)
    σ_est    = float(σ_blk.mean())
    n_req    = int(np.ceil((σ_est/target_sem)**2))
    g_curr   = min(max(int(np.ceil(np.sqrt(n_req))), g_min), g_max)

    # slice offsets for actual measurement
    blk_xy   = offsets_5x5[:g_curr**2] + np.array([cx,cy])

    # — drift-hold decision (one-sided) —
    t_now   = tot_cost_dyn
    in_evt  = t_event <= t_now <= t_event_end
    err_5   = 0.5*np.exp(-(t_now - t_event)/decay_tau) if in_evt else 0.0

    T_var   = 0.0                               # can swap to σ_est if desired
    μ_blk   = gp_m(X_blk).mean.mean().item()    # model mean (assumes raw units)
    T_big   = max(user_threshold_pct*μ_blk/100., 2*T_var)

    if prev_hold==0:
        hold = med_hold if (iter_idx)%eval_interval==0 else min_hold
    else:
        hold = min_hold if err_5<=T_var else med_hold if err_5<T_big else max_hold
    prev_hold = hold

    # — measurement & bias injection —
    xs,ys   = blk_xy[:,0], blk_xy[:,1]
    base_z  = np.array([measure_from_Hgrid_hetero(x,y) for x,y in zip(xs,ys)])
    if in_evt:
        σ5 = bias_amp5*np.exp(-(t_now-t_event)/decay_tau); σ0=bias_amp0
    else:
        σ5 = σ0 = 0.0
    bias = np.random.uniform(0,σ5,base_z.shape) if hold==med_hold else \
           np.random.uniform(0,σ0,base_z.shape) if hold==min_hold else \
           np.zeros_like(base_z)
    vals_dyn = base_z - bias

    all_x.extend(xs); all_y.extend(ys); all_z.extend(vals_dyn)

    # — cost bookkeeping (dynamic vs fixed 5×5/80 s) —
    c_dyn = evaluate_sample_cost_remaining(
              xs,ys, t_drift=default_t_drift,
              t_measurement=default_t_measurement+hold,
              t_comeback=t_comeback)[0]

    ref_xy = offsets_ref + np.array([cx,cy])
    c_bas  = evaluate_sample_cost_remaining(
              ref_xy[:,0], ref_xy[:,1],
              t_drift=default_t_drift,
              t_measurement=default_t_measurement+max_hold,
              t_comeback=t_comeback)[0]

    tot_cost_dyn += t_setup+t_move+c_dyn
    tot_cost_max += t_setup+t_move+c_bas
    cost_dyn.append(tot_cost_dyn); cost_max.append(tot_cost_max)
```

```
    # — prediction & metrics every iteration —
    grid = np.linspace(0,domain_size,res)
    XX,YY = np.meshgrid(grid,grid)
    test  = torch.tensor(np.column_stack([XX.ravel(),YY.ravel()])/domain_size,
                         dtype=torch.float32)
    Zμ, Zσ, _ = predict_hetero(gp_m, gp_s, test)
    Zμ = Zμ.reshape(res,res)
    mape_it = mape_percent(Z_true, Zμ); mae_it = mae(Z_true, Zμ)

    metrics.append({
        "iteration": iter_idx,
        "grid_size": g_curr,
        "n_points": g_curr**2,
        "hold_s": hold,
        "cost_dyn_s": tot_cost_dyn,
        "cost_max_s": tot_cost_max,
        "mape_%": mape_it,
        "mae": mae_it
    })

    # — visualisation every plot_interval —
    if (iter_idx)%plot_interval==0:
        fig,axs = plt.subplots(2,2,figsize=(13,10))

        im0 = axs[0,0].imshow(Zμ,origin='lower',
                              extent=(0,domain_size,0,domain_size),cmap='plasma')
        axs[0,0].set_title(f'Iter {iter_idx}: GP mean');     fig.colorbar(im0,ax=ax

        Zσ_img = Zσ.reshape(res,res)
        im1 = axs[0,1].imshow(Zσ_img,origin='lower',
                              extent=(0,domain_size,0,domain_size),cmap='plasma')
        axs[0,1].set_title('Predicted σ');                   fig.colorbar(im1,ax=ax

        axs[1,0].plot(np.arange(len(all_z)),all_z,'.',ms=2)
        axs[1,0].set_title('Dynamic measurements'); axs[1,0].set_xlabel('indent #')
        axs[1,0].set_ylabel('Hardness (GPa)')

        axs[1,1].plot(cost_dyn,label='Dynamic')
        axs[1,1].plot(cost_max,'--',label='Fixed 5×5 / 80 s')
        axs[1,1].set_title(f'Cumulative cost   MAPE={mape_it:.2f}% | MAE={mae_it:.4
        axs[1,1].set_xlabel('block'); axs[1,1].set_ylabel('time (s)')
        axs[1,1].legend(); axs[1,1].grid(ls='--',alpha=.5)

        plt.tight_layout()
        fig.savefig(os.path.join(OUTDIR, f"iter_{iter_idx:04d}_diagnostics.png"),
                    dpi=200, bbox_inches='tight')
        plt.close(fig)

# ———————— SAVE: metrics CSV & summary plots ————————————————————————
df = pd.DataFrame(metrics, columns=[
    "iteration","grid_size","n_points","hold_s",
    "cost_dyn_s","cost_max_s","mape_%","mae"
])
csv_path = os.path.join(OUTDIR, "metrics_adaptive_grid_dynamic_hetero.csv")
df.to_csv(csv_path, index=False)
```

```
# Cost vs MAPE (dynamic vs baseline)
plt.figure(figsize=(7.8,5.0))
plt.plot(df["cost_dyn_s"], df["mape_%"], marker='o', label="Dynamic (adaptive grid)
plt.plot(df["cost_max_s"], df["mape_%"], marker='^', linestyle='--', label="Baselin
plt.xlabel("Cumulative cost (s)")
plt.ylabel("MAPE (%) vs Z_true")
plt.title("Cost vs MAPE")
plt.grid(True, ls='--', alpha=.6)
plt.legend()
plt.tight_layout()
plt.savefig(os.path.join(OUTDIR, "cost_vs_mape.png"), dpi=200, bbox_inches='tight')
plt.close()

# Iteration vs MAPE
plt.figure(figsize=(7.8,5.0))
plt.plot(df["iteration"], df["mape_%"], marker='o')
plt.xlabel("Iteration")
plt.ylabel("MAPE (%)")
plt.title("Iteration vs MAPE")
plt.grid(True, ls='--', alpha=.6)
plt.tight_layout()
plt.savefig(os.path.join(OUTDIR, "iteration_vs_mape.png"), dpi=200, bbox_inches='ti
plt.close()

# Time vs Grid Size
plt.figure(figsize=(7.8,5.0))
plt.step(df["cost_dyn_s"], df["grid_size"], where='post')
plt.xlabel("Cumulative cost (s)")
plt.ylabel("Grid size (g × g)")
plt.title("Adaptive grid size vs time")
plt.grid(True, ls='--', alpha=.6)
plt.tight_layout()
plt.savefig(os.path.join(OUTDIR, "time_vs_grid_size.png"), dpi=200, bbox_inches='ti
plt.close()

# Time vs Hold (0/5/80)
plt.figure(figsize=(7.8,5.0))
plt.step(df["cost_dyn_s"], df["hold_s"], where='post')
plt.xlabel("Cumulative cost (s)")
plt.ylabel("Hold time used (s)")
plt.title("Dynamic hold vs time")
plt.grid(True, ls='--', alpha=.6)
plt.tight_layout()
plt.savefig(os.path.join(OUTDIR, "time_vs_hold.png"), dpi=200, bbox_inches='tight')
plt.close()

print(f"[Saved] CSV: {csv_path}")
print(f"[Saved] Figures every {plot_interval} iters in {OUTDIR}/")
print(f"[Saved] Summary: cost_vs_mape.png, iteration_vs_mape.png, time_vs_grid_size
```

```
[Saved] CSV: outputs_adaptive_grid_dynamic_hetero\metrics_adaptive_grid_dynamic_hete
ro.csv
[Saved] Figures every 10 iters in outputs_adaptive_grid_dynamic_hetero/
[Saved] Summary: cost_vs_mape.png, iteration_vs_mape.png, time_vs_grid_size.png, tim
e_vs_hold.png
```

Target SEM =0.5

In [11]:
```
################################################################################
#  ADAPTIVE-GRID  +  DYNAMIC-HOLD  +  HETEROSCEDASTIC-GP  SAMPLER  (SAVING)
################################################################################
import os
import numpy as np, torch, random, matplotlib.pyplot as plt
from scipy.ndimage import zoom
import pandas as pd

# ————— reproducibility ———————————————————————————————————————
SEED = 42
np.random.seed(SEED); torch.manual_seed(SEED); random.seed(SEED)

# ————— output ————————————————————————————————————————————————
OUTDIR = "outputs_adaptive_grid_dynamic_hetero_semp1"
OUTDIR = "outputs_adaptive_grid_dynamic_hetero_semp1"
os.makedirs(OUTDIR, exist_ok=True)

# ————— fixed parameters ———————————————————————————————————————
domain_size           = 1000          # µm (for normalising GP inputs)
t_setup, t_move       = 20, 20        # s
default_t_drift       = 300           # s
default_t_measurement = 40            # s
t_comeback            = 35            # s

# grid adaptation
g_min, g_max          = 2, 5          # min / max square-grid dimension
target_sem            = 0.5          # GPa desired SEM
n_initial, n_steps    = 5, 100        # ← 100 iterations as requested
plot_interval         = 10            # save plots every N blocks

# dynamic hold thresholds
min_hold, med_hold, max_hold = 0, 5, 80
eval_interval         = 5             # force a 5-s check every N blocks
user_threshold_pct    = 4.0           # % of µ to define T_big

# drift-event (simulated one-sided bias)
t_event, t_event_end  = 7_000., 45_000.
decay_tau             = 1_000.
bias_amp5, bias_amp0  = 0.3, 0.5

# ————— helper: heteroscedastic noise & measurement ——————————————
σ0, αA, αB, αC = 0.005, 0.1, 0.05, 0.0250
ny, nx = H_grid.shape

def noise_std_comp(xi:int, yi:int) -> float:
    xi = int(np.clip(xi, 0, nx-1))
    yi = int(np.clip(yi, 0, ny-1))
    return σ0 + αA*A[yi,xi] + αB*B[yi,xi] + αC*C[yi,xi]

def measure_from_Hgrid_hetero(xi:int, yi:int) -> float:
    xi = int(np.clip(xi, 0, nx-1))
```

```
    yi = int(np.clip(yi, 0, ny-1))
    H  = H_grid[yi, xi]
    σ  = noise_std_comp(xi, yi)
    return H + np.random.normal(0, σ)

# ————————— reference ground-truth image (for diagnostics) ———————————————
res     = 25
Z_true  = zoom(H_grid, zoom=res/ny, order=1)          # 25×25 GT map

# ————————— catalogue of possible  Block-centres  (5-µm lattice) ————————————
spacing      = 5
max_offset   = (g_max-1)*spacing
centres      = [(x, y) for x in np.linspace(0, domain_size-max_offset, 100)
                       for y in np.linspace(0, domain_size-max_offset, 100)]

# 5×5 offset template (slice [:g**2] for smaller grids)
ix, iy        = np.meshgrid(np.arange(g_max), np.arange(g_max))
offsets_5x5   = np.stack([ix.ravel(), iy.ravel()], axis=1)*spacing
offsets_ref   = offsets_5x5.copy()                    # always 25 pts

# ————————— storage arrays ——————————————————————————————————————————
all_x, all_y, all_z = [], [], []           # dynamic branch data
cost_dyn, cost_max  = [], []
tot_cost_dyn = tot_cost_max = 0.0
visited = []

# ————————— metrics table ————————————————————————————————————————————
metrics = []  # rows per block: iteration, grid_size, n_pts, hold, costs, MAPE, MAE

def mape_percent(y_true, y_pred, eps=1e-8):
    denom = np.maximum(np.abs(y_true), eps)
    return float(np.mean(np.abs((y_true - y_pred) / denom)) * 100.0)

def mae(y_true, y_pred):
    return float(np.mean(np.abs(y_true - y_pred)))

# ————————— 1) initial 5×5 blocks (80-s hold) ———————————————————————————
init_idxs = np.random.choice(len(centres), n_initial, replace=False)
for idx in init_idxs:
    cx,cy   = centres[idx]
    blk_xy  = offsets_5x5 + np.array([cx,cy])
    xs,ys   = blk_xy[:,0], blk_xy[:,1]
    zs      = [measure_from_Hgrid_hetero(x,y) for x,y in zip(xs,ys)]

    all_x.extend(xs); all_y.extend(ys); all_z.extend(zs); visited.append((cx,cy))

    t_meas  = default_t_measurement + max_hold
    move_cost  = evaluate_sample_cost_remaining(xs,ys,
                    t_drift=default_t_drift,
                    t_measurement=t_meas,
                    t_comeback=t_comeback)[0]
    add_c   = t_setup + t_move + move_cost
    tot_cost_dyn += add_c; tot_cost_max += add_c
    cost_dyn.append(tot_cost_dyn); cost_max.append(tot_cost_max)

centres   = [c for i,c in enumerate(centres) if i not in init_idxs]
```

```
prev_hold = max_hold
g_curr    = g_max                                  # start 5×5

# ———————— optional: baseline "iter 0" diagnostics ————————————————————
def diagnose_and_save(iter_idx, gp_m, gp_s, suffix=""):
    grid = np.linspace(0,domain_size,res)
    XX,YY = np.meshgrid(grid,grid)
    test  = torch.tensor(np.column_stack([XX.ravel(),YY.ravel()])/domain_size,
                         dtype=torch.float32)
    Zμ, Zσ, _ = predict_hetero(gp_m, gp_s, test)
    Zμ = Zμ.reshape(res,res); Zσ = Zσ.reshape(res,res)
    mape = mape_percent(Z_true, Zμ); mae_v = mae(Z_true, Zμ)

    fig,axs = plt.subplots(2,2,figsize=(13,10))
    im0 = axs[0,0].imshow(Zμ,origin='lower',
                          extent=(0,domain_size,0,domain_size),cmap='plasma')
    axs[0,0].set_title(f'Iter {iter_idx}: GP mean');     fig.colorbar(im0,ax=axs[0,
    im1 = axs[0,1].imshow(Zσ,origin='lower',
                          extent=(0,domain_size,0,domain_size),cmap='plasma')
    axs[0,1].set_title('Predicted σ');                   fig.colorbar(im1,ax=axs[0,

    axs[1,0].plot(np.arange(len(all_z)),all_z,'.',ms=2)
    axs[1,0].set_title('Dynamic measurements'); axs[1,0].set_xlabel('indent #')
    axs[1,0].set_ylabel('Hardness (GPa)')

    axs[1,1].plot(cost_dyn,label='Dynamic')
    axs[1,1].plot(cost_max,'--',label='Fixed 5×5 / 80 s')
    axs[1,1].set_title(f'Cumulative cost   MAPE={mape:.2f}% | MAE={mae_v:.4f}')
    axs[1,1].set_xlabel('block'); axs[1,1].set_ylabel('time (s)')
    axs[1,1].legend(); axs[1,1].grid(ls='--',alpha=.5)

    plt.tight_layout()
    fig.savefig(os.path.join(OUTDIR, f"iter_{iter_idx:04d}{suffix}_diagnostics.png"
                dpi=200, bbox_inches='tight')
    plt.close(fig)
    return mape, mae_v

# Warm-up fit just to produce iter 0000 figure
if len(all_x) > 0:
    X0 = torch.tensor(np.column_stack([all_x,all_y])/domain_size, dtype=torch.float
    y0 = torch.tensor(np.array(all_z), dtype=torch.float32)
    M0 = min(300, X0.size(0))
    inducing0 = X0[torch.randperm(X0.size(0))[:M0]]
    gp0_m, gp0_s = GPModel(inducing0), GPModel(inducing0)
    train_hetero(X0, y0, gp0_m, gp0_s, iters=200, lr=1e-2)
    diagnose_and_save(0, gp0_m, gp0_s, suffix="_warmup")

# ———————— 2) active-learning loop ————————————————————————————————
for step in range(n_steps):
    iter_idx = step + 1

    # — fit heteroscedastic GP on current dynamic data —
    X = torch.tensor(np.column_stack([all_x,all_y])/domain_size, dtype=torch.float3
    y = torch.tensor(np.array(all_z), dtype=torch.float32)
    M = min(300, X.size(0))
    inducing = X[torch.randperm(X.size(0))[:M]]
```

```
    gp_m, gp_s = GPModel(inducing), GPModel(inducing)
    train_hetero(X, y, gp_m, gp_s, iters=300, lr=1e-2)

    # — acquisition: UE / cost for every candidate centre —
    acq_vals = []
    for cx,cy in centres:
        blk_xy  = offsets_5x5[:g_curr**2] + np.array([cx,cy])
        Xtest   = torch.tensor(blk_xy/domain_size, dtype=torch.float32)
        μ, σ, var  = predict_hetero(gp_m, gp_s, Xtest)
        unc     = (σ+var).mean()

        est_tmeas = default_t_measurement + prev_hold
        est_cost  = evaluate_sample_cost_remaining(
                        blk_xy[:,0], blk_xy[:,1],
                        t_drift=default_t_drift,
                        t_measurement=est_tmeas,
                        t_comeback=t_comeback)[0]
        acq_vals.append(unc / (t_setup+t_move+est_cost))

    # pick best centre
    best   = int(np.argmax(acq_vals))
    cx,cy  = centres.pop(best)
    visited.append((cx,cy))

    # — block-specific σ → grid size  (adaptive grid) —
    blk_xy   = offsets_5x5 + np.array([cx,cy])          # 25 candidate pts
    X_blk    = torch.tensor(blk_xy/domain_size, dtype=torch.float32)
    _, σ_blk, _ = predict_hetero(gp_m, gp_s, X_blk)
    σ_est    = float(σ_blk.mean())
    n_req    = int(np.ceil((σ_est/target_sem)**2))
    g_curr   = min(max(int(np.ceil(np.sqrt(n_req))), g_min), g_max)

    # slice offsets for actual measurement
    blk_xy   = offsets_5x5[:g_curr**2] + np.array([cx,cy])

    # — drift-hold decision (one-sided) —
    t_now   = tot_cost_dyn
    in_evt  = t_event <= t_now <= t_event_end
    err_5   = 0.5*np.exp(-(t_now - t_event)/decay_tau) if in_evt else 0.0

    T_var   = 0.0                               # can swap to σ_est if desired
    μ_blk   = gp_m(X_blk).mean.mean().item()    # model mean (assumes raw units)
    T_big   = max(user_threshold_pct*μ_blk/100., 2*T_var)

    if prev_hold==0:
        hold = med_hold if (iter_idx)%eval_interval==0 else min_hold
    else:
        hold = min_hold if err_5<=T_var else med_hold if err_5<T_big else max_hold
    prev_hold = hold

    # — measurement & bias injection —
    xs,ys   = blk_xy[:,0], blk_xy[:,1]
    base_z  = np.array([measure_from_Hgrid_hetero(x,y) for x,y in zip(xs,ys)])
    if in_evt:
        σ5 = bias_amp5*np.exp(-(t_now-t_event)/decay_tau); σ0=bias_amp0
    else:
```

```
    σ5 = σ0 = 0.0
bias = np.random.uniform(0,σ5,base_z.shape) if hold==med_hold else \
       np.random.uniform(0,σ0,base_z.shape) if hold==min_hold else \
       np.zeros_like(base_z)
vals_dyn = base_z - bias

all_x.extend(xs); all_y.extend(ys); all_z.extend(vals_dyn)

# — cost bookkeeping (dynamic vs fixed 5×5/80 s) —
c_dyn = evaluate_sample_cost_remaining(
          xs,ys, t_drift=default_t_drift,
          t_measurement=default_t_measurement+hold,
          t_comeback=t_comeback)[0]

ref_xy = offsets_ref + np.array([cx,cy])
c_bas  = evaluate_sample_cost_remaining(
          ref_xy[:,0], ref_xy[:,1],
          t_drift=default_t_drift,
          t_measurement=default_t_measurement+max_hold,
          t_comeback=t_comeback)[0]

tot_cost_dyn += t_setup+t_move+c_dyn
tot_cost_max += t_setup+t_move+c_bas
cost_dyn.append(tot_cost_dyn); cost_max.append(tot_cost_max)

# — prediction & metrics every iteration —
grid = np.linspace(0,domain_size,res)
XX,YY = np.meshgrid(grid,grid)
test  = torch.tensor(np.column_stack([XX.ravel(),YY.ravel()])/domain_size,
                     dtype=torch.float32)
Zμ, Zσ, _ = predict_hetero(gp_m, gp_s, test)
Zμ = Zμ.reshape(res,res)
mape_it = mape_percent(Z_true, Zμ); mae_it = mae(Z_true, Zμ)

metrics.append({
    "iteration": iter_idx,
    "grid_size": g_curr,
    "n_points": g_curr**2,
    "hold_s": hold,
    "cost_dyn_s": tot_cost_dyn,
    "cost_max_s": tot_cost_max,
    "mape_%": mape_it,
    "mae": mae_it
})

# — visualisation every plot_interval —
if (iter_idx)%plot_interval==0:
    fig,axs = plt.subplots(2,2,figsize=(13,10))

    im0 = axs[0,0].imshow(Zμ,origin='lower',
                          extent=(0,domain_size,0,domain_size),cmap='plasma')
    axs[0,0].set_title(f'Iter {iter_idx}: GP mean');     fig.colorbar(im0,ax=ax

    Zσ_img = Zσ.reshape(res,res)
    im1 = axs[0,1].imshow(Zσ_img,origin='lower',
                          extent=(0,domain_size,0,domain_size),cmap='plasma')
```

```
        axs[0,1].set_title('Predicted σ');                  fig.colorbar(im1,ax=ax

        axs[1,0].plot(np.arange(len(all_z)),all_z,'.',ms=2)
        axs[1,0].set_title('Dynamic measurements'); axs[1,0].set_xlabel('indent #')
        axs[1,0].set_ylabel('Hardness (GPa)')

        axs[1,1].plot(cost_dyn,label='Dynamic')
        axs[1,1].plot(cost_max,'--',label='Fixed 5×5 / 80 s')
        axs[1,1].set_title(f'Cumulative cost   MAPE={mape_it:.2f}% | MAE={mae_it:.4
        axs[1,1].set_xlabel('block'); axs[1,1].set_ylabel('time (s)')
        axs[1,1].legend(); axs[1,1].grid(ls='--',alpha=.5)

        plt.tight_layout()
        fig.savefig(os.path.join(OUTDIR, f"iter_{iter_idx:04d}_diagnostics.png"),
                    dpi=200, bbox_inches='tight')
        plt.close(fig)

# ———————— SAVE: metrics CSV & summary plots ————————————————————————
df = pd.DataFrame(metrics, columns=[
    "iteration","grid_size","n_points","hold_s",
    "cost_dyn_s","cost_max_s","mape_%","mae"
])
csv_path = os.path.join(OUTDIR, "metrics_adaptive_grid_dynamic_hetero.csv")
df.to_csv(csv_path, index=False)

# Cost vs MAPE (dynamic vs baseline)
plt.figure(figsize=(7.8,5.0))
plt.plot(df["cost_dyn_s"], df["mape_%"], marker='o', label="Dynamic (adaptive grid)
plt.plot(df["cost_max_s"], df["mape_%"], marker='^', linestyle='--', label="Baselin
plt.xlabel("Cumulative cost (s)")
plt.ylabel("MAPE (%) vs Z_true")
plt.title("Cost vs MAPE")
plt.grid(True, ls='--', alpha=.6)
plt.legend()
plt.tight_layout()
plt.savefig(os.path.join(OUTDIR, "cost_vs_mape.png"), dpi=200, bbox_inches='tight')
plt.close()

# Iteration vs MAPE
plt.figure(figsize=(7.8,5.0))
plt.plot(df["iteration"], df["mape_%"], marker='o')
plt.xlabel("Iteration")
plt.ylabel("MAPE (%)")
plt.title("Iteration vs MAPE")
plt.grid(True, ls='--', alpha=.6)
plt.tight_layout()
plt.savefig(os.path.join(OUTDIR, "iteration_vs_mape.png"), dpi=200, bbox_inches='ti
plt.close()

# Time vs Grid Size
plt.figure(figsize=(7.8,5.0))
plt.step(df["cost_dyn_s"], df["grid_size"], where='post')
plt.xlabel("Cumulative cost (s)")
plt.ylabel("Grid size (g × g)")
plt.title("Adaptive grid size vs time")
plt.grid(True, ls='--', alpha=.6)
```

```
plt.tight_layout()
plt.savefig(os.path.join(OUTDIR, "time_vs_grid_size.png"), dpi=200, bbox_inches='ti
plt.close()

# Time vs Hold (0/5/80)
plt.figure(figsize=(7.8,5.0))
plt.step(df["cost_dyn_s"], df["hold_s"], where='post')
plt.xlabel("Cumulative cost (s)")
plt.ylabel("Hold time used (s)")
plt.title("Dynamic hold vs time")
plt.grid(True, ls='--', alpha=.6)
plt.tight_layout()
plt.savefig(os.path.join(OUTDIR, "time_vs_hold.png"), dpi=200, bbox_inches='tight')
plt.close()

print(f"[Saved] CSV: {csv_path}")
print(f"[Saved] Figures every {plot_interval} iters in {OUTDIR}/")
print(f"[Saved] Summary: cost_vs_mape.png, iteration_vs_mape.png, time_vs_grid_size
```

```
[Saved] CSV: outputs_adaptive_grid_dynamic_hetero_semp1\metrics_adaptive_grid_dynami
c_hetero.csv
[Saved] Figures every 10 iters in outputs_adaptive_grid_dynamic_hetero_semp1/
[Saved] Summary: cost_vs_mape.png, iteration_vs_mape.png, time_vs_grid_size.png, tim
e_vs_hold.png
```

Defining local "safe zones for indentation"

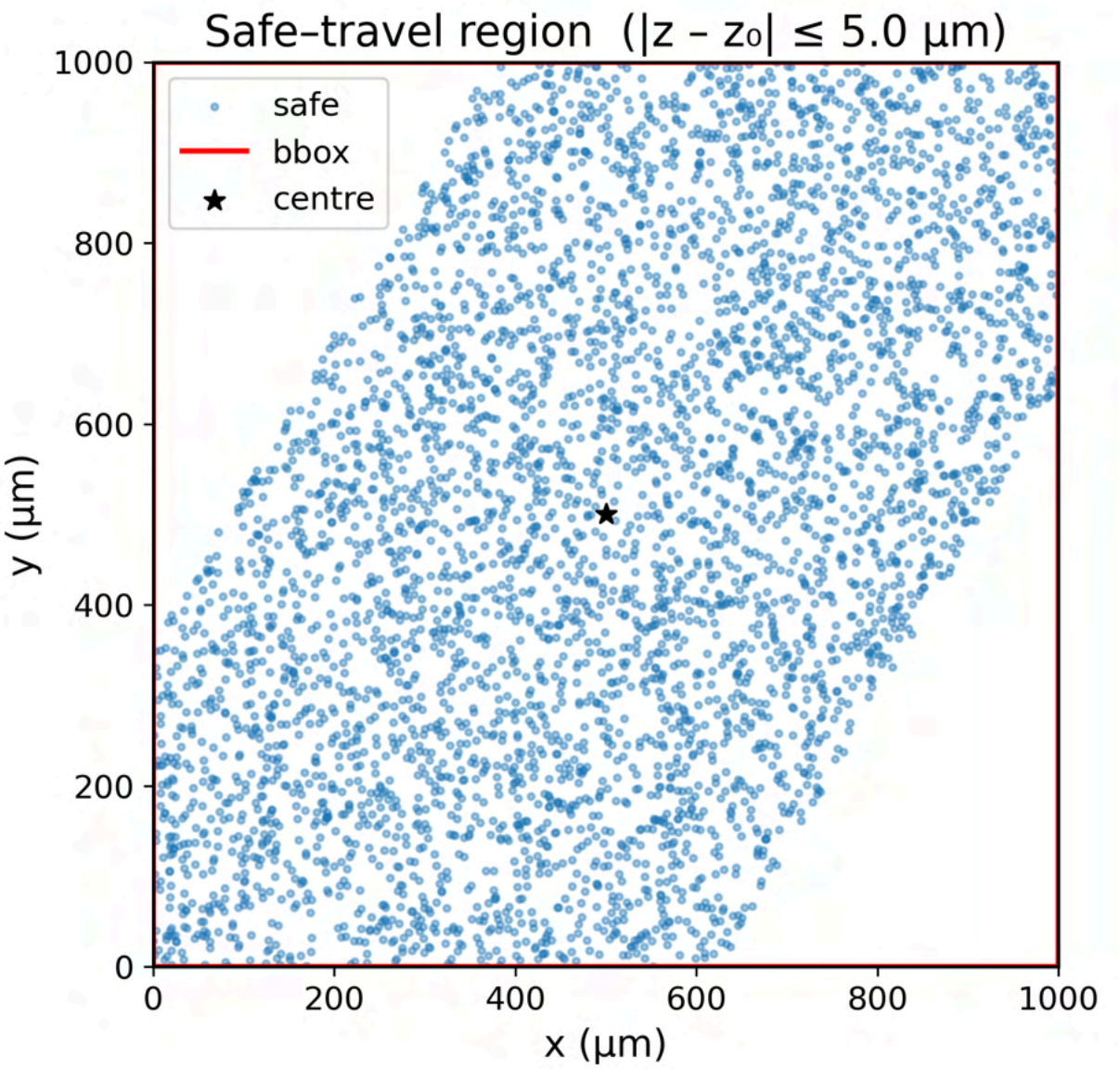
In [55]:

```
import numpy as np

def fit_plane_xy_to_z(points):
    """
    Fit z ≈ a*x + b*y + c from 3 (or more) (x,y,z) points.
    Returns (a, b, c, predict_fn).
    """
    P = np.asarray(points, dtype=float)
    if P.ndim != 2 or P.shape[1] != 3 or P.shape[0] < 3:
        raise ValueError("Provide an array-like of shape (N,3) with N>=3.")
    X = np.c_[P[:,0], P[:,1], np.ones(len(P))]   # [x y 1]
    z = P[:,2]
    a, b, c = np.linalg.lstsq(X, z, rcond=None)[0]
    return a, b, c, (lambda x, y: a*x + b*y + c)

# Example:
pts = [(X_bl,Y_bl, Z_bl), (X_tr,Y_tr, Z_tr), (X_br,Y_br, Z_br)]
a, b, c, f = fit_plane_xy_to_z(pts)
print(f"z = {a:.6f}*x + {b:.6f}*y + {c:.6f}")
print("z(0.5, 0.5) =", f(0.5, 0.5))
```

```
z = -0.000000*x + -0.000002*y + 9.671757
z(0.5, 0.5) = 9.6717562231465
```

Now lets decide boundaries

In [56]:
```
import numpy as np, matplotlib.pyplot as plt

# ── plane & safety spec ──────────────────────────────────────────
a,b,c = 0.012, -0.007, 3.0      # z = ax + by + c
x0,y0 = 500., 500.
z0    = a*x0 + b*y0 + c
dz_safe = 10.0/2                # 5 µm

# ── dense cloud of points ────────────────────────────────────────
N = 60_00
xs = np.random.uniform(0,1000,N)
ys = np.random.uniform(0,1000,N)
zs = a*xs + b*ys + c
mask = np.abs(zs - z0) <= dz_safe
xs_s, ys_s = xs[mask], ys[mask]

# bounding box
xmin,xmax = xs_s.min(), xs_s.max()
ymin,ymax = ys_s.min(), ys_s.max()

# ── plot ─────────────────────────────────────────────────────────
fig,ax = plt.subplots(figsize=(6,6))
ax.scatter(xs_s, ys_s, s=6, c='tab:blue', alpha=.4, label='safe')
ax.plot([xmin,xmax,xmax,xmin,xmin],
        [ymin,ymin,ymax,ymax,ymin], 'r-', lw=2, label='bbox')
ax.scatter([x0],[y0], c='k', s=60, marker='*', label='centre')

ax.set_aspect('equal'); ax.set_xlim(0,1000); ax.set_ylim(0,1000)
ax.set_xlabel('x (µm)'); ax.set_ylabel('y (µm)')
ax.set_title(fr"Safe-travel region  (|z − z₀| ≤ {dz_safe:.1f} µm)")
ax.legend()
plt.tight_layout(); plt.show()
```

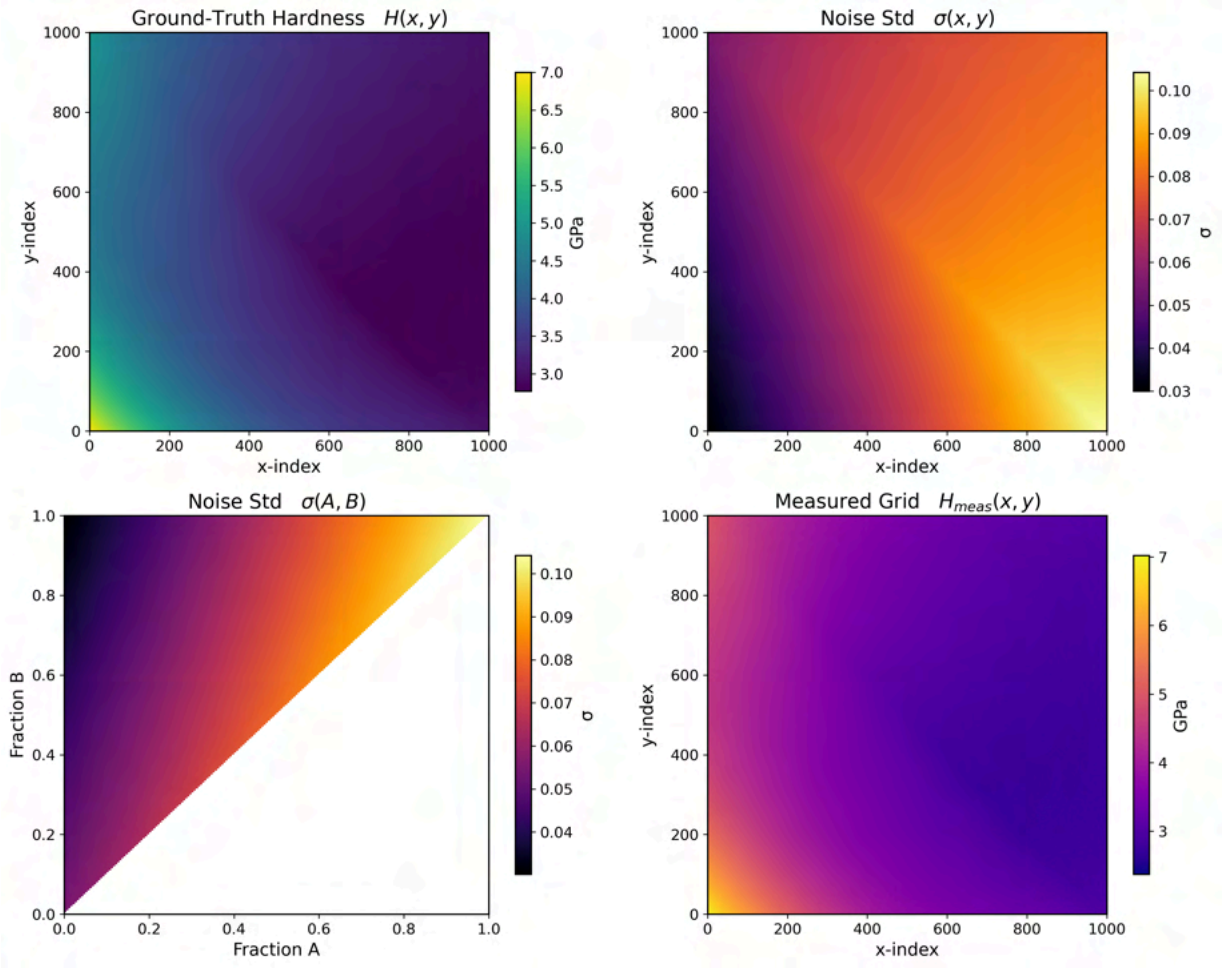


Ground truth

In [57]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from  scipy.stats import binned_statistic_2d

# ---------------------------------------------------------------------
# 1.  Build 1000 × 1000 ternary-composition grid + ground-truth hardness
# ---------------------------------------------------------------------
nx = ny = 1000
x_coords, y_coords = np.meshgrid(np.linspace(0, 1, nx),
                                 np.linspace(0, 1, ny))
A = x_coords
B = y_coords
C = np.maximum(1.0 - A - B, 0.01)          # keep C ≥ 0
total = A + B + C                          # renormalise so A+B+C = 1
A /= total ;  B /= total ;  C /= total

def hardness_model(a,b,c):
```

```
    base         = 3*a + 5*b + 7*c                    # rule-of-mixtures
    ss_strength  = -4*a*b - 3*b*c - 2*c*a             # solid-solution term
    saturation   = -6*c**2 * (1 - c)                  # saturation penalty
    return base + ss_strength + saturation

H_grid = hardness_model(A,B,C)                        # ground-truth hardness

# ------------------------------------------------------------------
# 2.  Composition-dependent noise:  σ(A,B,C)
# ------------------------------------------------------------------
σ0, αA, αB, αC = 0.005, 0.1, 0.05, 0.0250
noise_std_grid = σ0 + αA*A + αB*B + αC*C              # same shape as H_grid

# ------------------------------------------------------------------
# 3.  One synthetic "measured" grid  (H + Gaussian noise)
# ------------------------------------------------------------------
rng = np.random.default_rng(123)
H_meas = H_grid + rng.normal(0, noise_std_grid)       # element-wise σ

# ------------------------------------------------------------------
# 4.  Re-bin σ onto a regular (A,B) composition grid for plotting
# ------------------------------------------------------------------
nbins = 200                                           # resolution in A–B space
σ_AB, A_edges, B_edges, _ = binned_statistic_2d(
        A.ravel(), B.ravel(), noise_std_grid.ravel(),
        statistic='mean', bins=nbins, range=[[0,1],[0,1]])
σ_AB = np.flipud(σ_AB.T)                              # put origin bottom-left

# ------------------------------------------------------------------
# 5.  Plot:  2 × 2   (all imshow, no 3-D)
# ------------------------------------------------------------------
fig, axs = plt.subplots(2, 2, figsize=(14, 11))

# (0,0) ground-truth hardness in x–y space
im0 = axs[0,0].imshow(H_grid, origin='lower', cmap='viridis',
                      extent=(0, nx, 0, ny))
axs[0,0].set_title("Ground-Truth Hardness   $H(x,y)$")
axs[0,0].set_xlabel("x-index") ; axs[0,0].set_ylabel("y-index")
fig.colorbar(im0, ax=axs[0,0], shrink=0.8, label="GPa")

# (0,1) noise σ in x–y space
im1 = axs[0,1].imshow(noise_std_grid, origin='lower', cmap='inferno',
                      extent=(0, nx, 0, ny))
axs[0,1].set_title("Noise Std   $\\sigma(x,y)$")
axs[0,1].set_xlabel("x-index") ; axs[0,1].set_ylabel("y-index")
fig.colorbar(im1, ax=axs[0,1], shrink=0.8, label="σ")

# (1,0) noise σ on regular composition grid (Fraction A vs Fraction B)
im2 = axs[1,0].imshow(σ_AB, origin='lower', cmap='inferno',
                      extent=(0,1,0,1), aspect='auto')
axs[1,0].set_title(r"Noise Std   $\sigma(A,B)$")
axs[1,0].set_xlabel("Fraction A") ; axs[1,0].set_ylabel("Fraction B")
fig.colorbar(im2, ax=axs[1,0], shrink=0.8, label="σ")

# (1,1) one synthetic measured grid  (H + noise)
im3 = axs[1,1].imshow(H_meas, origin='lower', cmap='plasma',
```

```
                      extent=(0, nx, 0, ny))
axs[1,1].set_title("Measured Grid   $H_{meas}(x,y)$")
axs[1,1].set_xlabel("x-index") ; axs[1,1].set_ylabel("y-index")
fig.colorbar(im3, ax=axs[1,1], shrink=0.8, label="GPa")

plt.tight_layout()
plt.show()
```



In [58]:
```
σ0, αA, αB, αC = 0.005, 0.1, 0.05, 0.0250
def noise_std_comp(xi:int, yi:int) -> float:
    """σ(A,B,C) at integer pixel (xi,yi)."""
    xi = int(np.clip(xi, 0, nx-1))
    yi = int(np.clip(yi, 0, ny-1))
    return σ0 + αA*A[yi,xi] + αB*B[yi,xi] + αC*C[yi,xi]

def measure_from_Hgrid_hetero(xi:int, yi:int) -> float:
    """Hardness + Gaussian noise whose std depends on local composition."""
    xi = int(np.clip(xi, 0, nx-1))
    yi = int(np.clip(yi, 0, ny-1))
    H  = H_grid[yi, xi]
    σ  = noise_std_comp(xi, yi)
    return H + np.random.normal(0, σ)
```

In [59]:
```
# ———————————————— PLOT HELPER (v3) ————————————————
def plot_state(iter_no:int,
               gp_m, gp_s,                    # trained GPs
```

```
            Z_true:np.ndarray,              # coarse GT map  (res×res)
            rings:dict,                     # {ring:[(x,y)…]}
            a:float, b:float, c:float,      # plane coeffs
            dz_safe:float,
            field_lim:float,                # ±µm plot window
            g_centre:tuple,                 # (xcentre,ycentre) this stay
            global_history:list,            # all global centres so far
            local_centres_done:list,        # centres visited this stay
            local_indents:list,             # every (x,y) indent this stay
            cost_dyn:list, cost_base:list): # cumulative-time traces
    """
    Four-panel status figure:
        (0,0) safe band & meta-grid
        (0,1) GP mean
        (1,0) TOTAL predictive σ  (model + noise)
        (1,1) cumulative time cost
    """
    # ——— 1. predictive maps on same coarse grid as Z_true ———
    res  = Z_true.shape[0]
    grid = np.linspace(0, domain_size, res)
    XX, YY = np.meshgrid(grid, grid)
    test = torch.tensor(np.column_stack([XX.ravel(), YY.ravel()]) / domain_size,
                        dtype=torch.float32)
    Zμ, Zσ_model, Zvar_noise = predict_hetero(gp_m, gp_s, test)
    Zμ         = Zμ.reshape(res, res)
    # Zσ_total   = np.sqrt(Zσ_model**2 + Zvar_noise).reshape(res, res)
    # total predictive σ  (model + noise)
    Zσ_total = np.sqrt(Zσ_model**2 + Zvar_noise).reshape(res, res)


    mape = np.mean(np.abs((Z_true - Zμ) / Z_true)) * 100

    fig, axs = plt.subplots(2, 2, figsize=(13, 10))

    # ── (0,0) SAFE BAND + META GRID ──────────────────────────
    ax   = axs[0, 0]
    xc, yc = g_centre
    center = np.array([xc, yc])

    # safe-zone lines
    xx = np.linspace(-field_lim, field_lim, 2)
    upper = (( dz_safe) - a * xx - c) / b
    lower = ((-dz_safe) - a * xx - c) / b
    ax.plot(xx, upper, 'r--'); ax.plot(xx, lower, 'r--')
    ax.fill_between(xx, upper, lower, color='orange', alpha=.15,
                    label='safe zone')

    # ring rectangles centred on current global point
    grad_norm = np.hypot(a, b)
    e_perp = np.array([a, b]) / grad_norm
    e_par  = np.array([b, -a]) / grad_norm
    STEP = 450

    for k in range(1, len(rings)):        # skip k=0 (single point)
        r = k * STEP
        rect_xy = np.array([[xc - r, yc - r],
```

```
                            [xc + r, yc - r],
                            [xc + r, yc + r],
                            [xc - r, yc + r],
                            [xc - r, yc - r]])
        ax.plot(rect_xy[:,0], rect_xy[:,1], 'g-', lw=.8, alpha=.6)


    # global centres (black)
    if global_history:
        gxy = np.asarray(global_history)
        ax.scatter(gxy[:,0], gxy[:,1], c='k', s=28, marker='o',
                   label='global centres', zorder=5)

    # visited local centres (blue line)
    if local_centres_done:
        lxy = np.asarray(local_centres_done)
        ax.plot(lxy[:,0], lxy[:,1], 'b-o', ms=4, lw=1.0,
                label='local centres', zorder=6)

    # indents this stay (dots)
    if local_indents is not None and len(local_indents):
        pts = np.asarray(local_indents)
        ax.scatter(pts[:,0], pts[:,1], c='royalblue', s=8, alpha=.6,
                   label='indents', zorder=4)

    ax.set_aspect('equal')
    ax.set_xlim(-field_lim, field_lim); ax.set_ylim(-field_lim, field_lim)
    ax.set_xlabel('x (µm)'); ax.set_ylabel('y (µm)')
    ax.set_title(f'Iter {iter_no}: safe band & meta-grid')
    ax.legend(fontsize=8, loc='upper left')

    # ── (0,1) GP mean ──────────────────────────────────────
    im0 = axs[0,1].imshow(Zμ, origin='lower', extent=(0,1,0,1), cmap='viridis')
    axs[0,1].set_title('GP mean');  fig.colorbar(im0, ax=axs[0,1])

    # ── (1,0) TOTAL predictive σ (model + noise) ──────────
    im1 = axs[1,0].imshow(Zσ_total, origin='lower', extent=(0,1,0,1),
                          cmap='inferno')
    axs[1,0].set_title('Total predictive σ'); fig.colorbar(im1, ax=axs[1,0])

    # ── (1,1) cumulative time cost ────────────────────────
    axs[1,1].plot(cost_dyn,      label='dynamic')
    axs[1,1].plot(cost_base, '--', label='baseline 5×5 / 80 s')
    axs[1,1].set_xlabel('block'); axs[1,1].set_ylabel('time (s)')
    axs[1,1].set_title(f'Cumulative cost   MAPE {mape:.2f}%')
    axs[1,1].grid(ls='--', alpha=.5); axs[1,1].legend()

    plt.tight_layout(); plt.show()
```

In [60]:

```
# -# ──────────────────── PLOT HELPER (v5) ────────────────────
def plot_state(iter_no:int,
               gp_m, gp_s,
               Z_true, rings,
               a,b,c, dz_safe,
               field_lim,
               g_centre,
```

```
            global_history,
            local_centres_done,
            local_indents,
            cost_dyn, cost_base):
    """
    6-panel figure:
      (0,0) safe band & meta-grid
      (0,1) GP mean
      (0,2) model σ   (epistemic)
      (1,0) noise σ   (aleatoric)
      (1,1) empty (spacer)
      (1,2) cumulative cost
    """
    # ── predictive maps ──────────────────────────────────
    res  = Z_true.shape[0]
    grid = np.linspace(0, domain_size, res)
    XX, YY = np.meshgrid(grid, grid)
    test = torch.tensor(np.column_stack([XX.ravel(), YY.ravel()]) / domain_size,
                        dtype=torch.float32)

    μ, noise_var, var_lat = predict_hetero(gp_m, gp_s, test)
    μ        = μ.reshape(res, res)
    σ_model  = np.sqrt(var_lat).reshape(res, res)
    σ_noise  = np.sqrt(noise_var).reshape(res, res)

    mape = np.mean(np.abs((Z_true - μ) / Z_true)) * 100

    fig, axs = plt.subplots(2, 3, figsize=(17, 10))
    axs[1,1].axis('off')                    # spacer cell

    # ── (0,0) safe band & grid ──────────────────────────
    ax = axs[0,0]
    xc, yc = g_centre

    z0   = a*xc + b*yc + c
    xx = np.linspace(-field_lim, field_lim, 2)
    upper = (( z0+dz_safe) - a*xx - c) / b
    lower = ((z0-dz_safe) - a*xx - c) / b
    ax.plot(xx, upper, 'r--'); ax.plot(xx, lower, 'r--')
    ax.fill_between(xx, upper, lower, color='orange', alpha=.15, label='safe zone')

    # green ring rectangles centred *exactly* on (xc, yc)
    STEP = 450
    for k in range(1, len(rings)):
        r = k * STEP
        rect = np.array([[xc-r, yc-r],
                         [xc+r, yc-r],
                         [xc+r, yc+r],
                         [xc-r, yc+r],
                         [xc-r, yc-r]])
        ax.plot(rect[:,0], rect[:,1], 'g-', lw=.8, alpha=.6)

    # centres & indents
    if global_history:
        gxy = np.asarray(global_history)
        ax.scatter(gxy[:,0], gxy[:,1], c='k', s=28, label='global centres', zorder=
```

```
    if local_centres_done:
        lxy = np.asarray(local_centres_done)
        ax.plot(lxy[:,0], lxy[:,1], 'b-o', ms=4, lw=1., label='local centres', zord
    if local_indents is not None and len(local_indents):
        pts = np.asarray(local_indents)
        ax.scatter(pts[:,0], pts[:,1], c='royalblue', s=8, alpha=.6,
                   label='indents', zorder=4)

    ax.set_aspect('equal')
    ax.set_xlim(-field_lim, field_lim); ax.set_ylim(-field_lim, field_lim)
    ax.set_xlabel('x (µm)'); ax.set_ylabel('y (µm)')
    ax.set_title(f'Iter {iter_no}: safe band & meta-grid')
    ax.legend(fontsize=8, loc='upper left')

    # ── (0,1) GP mean ──────────────────────────────────────
    im = axs[0,1].imshow(µ, origin='lower', extent=(0,1,0,1), cmap='plasma')
    axs[0,1].set_title('GP mean'); fig.colorbar(im, ax=axs[0,1])

    # ── (0,2) model σ (epistemic) ──────────────────────────
    im = axs[0,2].imshow(σ_model, origin='lower', extent=(0,1,0,1), cmap='inferno')
    axs[0,2].set_title('Model σ'); fig.colorbar(im, ax=axs[0,2])

    # ── (1,0) noise σ (aleatoric) ──────────────────────────
    im = axs[1,0].imshow(σ_noise, origin='lower', extent=(0,1,0,1), cmap='magma')
    axs[1,0].set_title('Noise σ'); fig.colorbar(im, ax=axs[1,0])

    # ── (1,2) cumulative cost ──────────────────────────────
    axs[1,2].plot(cost_dyn, label='dynamic')
    axs[1,2].plot(cost_base, '--', label='baseline 5×5 / 80 s')
    axs[1,2].set_xlabel('block'); axs[1,2].set_ylabel('time (s)')
    axs[1,2].set_title(f'Cumulative cost   MAPE {mape:.2f}%')
    axs[1,2].grid(ls='--', alpha=.5); axs[1,2].legend()

    plt.tight_layout(); plt.show()
```

Meta Grid + Adaptive Grid sizing + Adaptive hold for Drift + Cost aware decision making

In [62]:

```
import numpy as np
import torch
import random
import matplotlib.pyplot as plt
from scipy.ndimage import zoom
from math import sqrt

# ─── ASSUMED PREDEFINED in your notebook: ───
#   • GPModel, train_hetero, predict_hetero
#   • evaluate_sample_cost_remaining
#   • measure_from_Hgrid_hetero
#   • H_grid, A, B, C  (your ternary-model arrays)
# ──────────────────────────────────────────
```

```
SEED = 42
np.random.seed(SEED)
torch.manual_seed(SEED)
random.seed(SEED)

# ——————————————————————————————————————————————————————————————————————
# 1) SAFE-BAND PLANE + RING OFFSETS (at origin)
# ——————————————————————————————————————————————————————————————————————
a, b, c       = 0.02, 0.03, 0.00
dz_safe       = 5.0
grad_norm     = sqrt(a*a + b*b)
BAND_HALF     = dz_safe / grad_norm

e_perp = np.array([ a,  b]) / grad_norm
e_par  = np.array([ b, -a]) / grad_norm

STEP_RING = 45
rings = {}
ring = 0
while True:
    r = ring * STEP_RING
    if ring == 0:
        pts = [(0,0)]
    else:
        us = np.arange(-r, r+STEP_RING, STEP_RING)
        vs = np.arange(-r+STEP_RING, r, STEP_RING)
        pts = [(u, r) for u in us] + [(u, -r) for u in us] \
            + [(r, v) for v in vs] + [(-r, v) for v in vs]
    rings[ring] = pts
    ring += 1
    if r > BAND_HALF + 1e-6:
        break

def point_in_band_shifted(xc, yc, z0):
    return abs(a*xc + b*yc + c - z0) <= dz_safe + 1e-9

# ——————————————————————————————————————————————————————————————————————
# 2) GLOBAL GRID: 0–10 mm at 50 µm pitch
# ——————————————————————————————————————————————————————————————————————
domain_size = 10_00
STEP_GLOBAL = 10
xs2 = np.arange(0, domain_size+1, STEP_GLOBAL)
ys2 = np.arange(0, domain_size+1, STEP_GLOBAL)
global_pool = [(x,y) for x in xs2 for y in ys2]

# ——————————————————————————————————————————————————————————————————————
# 3) CONSTANTS, OFFSETS & GROUND-TRUTH
# ——————————————————————————————————————————————————————————————————————
t_setup, t_move       = 20, 20
default_t_drift       = 300
default_t_measurement = 40
t_comeback            = 35

g_min, g_max          = 2, 5
target_sem            = 0.5        # forces g_curr = g_max if zero
```

```
n_initial, n_steps    = 2, 40
plot_interval         = 1

min_hold, med_hold, max_hold = 0, 5, 80
eval_interval                = 5
user_threshold_pct           = 4.0

t_event, t_event_end = 7_000., 45_000.
decay_tau            = 1_000.
bias_amp5, bias_amp0 = 0.30, 0.50

num_center = 1  # how many local grids per block

spacing     = 5
ix, iy      = np.meshgrid(np.arange(g_max), np.arange(g_max))
offsets_5x5 = np.stack([ix.ravel(), iy.ravel()], axis=1) * spacing
offsets_ref = offsets_5x5.copy()

ny, nx = H_grid.shape
res    = 25
Z_true = zoom(H_grid, zoom=res/ny, order=1)

# ──────────────────────────────────────────────────────────────────────────
# 4) PLOT HELPER (6 panels)
# ──────────────────────────────────────────────────────────────────────────
def plot_state(step, gp_m, gp_s, Z_true,
               rings, a,b,c, dz_safe, field_lim, g_centre,
               visited_globals, drilled_centres, drilled_points,
               cost_dyn, cost_base, title_suffix=""):

    grid = np.linspace(0, field_lim, res)
    XX, YY = np.meshgrid(grid, grid)
    test = torch.tensor(
        np.column_stack([XX.ravel(), YY.ravel()]) / field_lim,
        dtype=torch.float32
    )
    μ_map, noise_map, var_lat_map = predict_hetero(gp_m, gp_s, test)
    μ_map       = μ_map.reshape(res,res)
    σ_model_map = np.sqrt(var_lat_map).reshape(res,res)
    σ_noise_map = noise_map.reshape(res,res)
    mape        = np.mean(np.abs((Z_true - μ_map)/Z_true))*100

    fig, axs = plt.subplots(2,3, figsize=(18,10))
    axs[1,1].axis('off')

    gx, gy = g_centre
    z0 = a*gx + b*gy + c
    xx = np.linspace(-field_lim, field_lim, 2)
    upper = ((z0+dz_safe)-a*xx-c)/b
    lower = ((z0-dz_safe)-a*xx-c)/b

    ax = axs[0,0]
    ax.plot(xx,upper,'r--'); ax.plot(xx,lower,'r--')
    ax.fill_between(xx,upper,lower,color='orange',alpha=0.2)
    for k, uv_list in rings.items():
        if k==0: continue
```

```
        r = k*STEP_RING
        corners = [(-r,-r),(r,-r),(r,r),(-r,r),(-r,-r)]
        rect = np.array([
            (gx + u*e_par[0] + v*e_perp[0],
             gy + u*e_par[1] + v*e_perp[1])
            for u,v in corners])
        ax.plot(rect[:,0], rect[:,1],'g-',lw=0.8,alpha=0.6)

    if visited_globals:
        vg = np.array(visited_globals)
        ax.scatter(vg[:,0],vg[:,1],c='k',s=30, label='global centres')
    if drilled_centres:
        dc = np.array(drilled_centres)
        ax.plot(dc[:,0],dc[:,1],'b-o',ms=4,label='local centres')
    if drilled_points:
        dp = np.array(drilled_points)
        ax.scatter(dp[:,0],dp[:,1],c='royalblue',s=8,alpha=0.6,label='indents')

    ax.set_aspect('equal')
    ax.set_title(f"Iter {step}{title_suffix}")
    ax.legend(fontsize=8, loc='upper left')

    im = axs[0,1].imshow(μ_map,origin='lower',extent=(0,1,0,1),cmap='plasma')
    axs[0,1].set_title('GP mean');     fig.colorbar(im,ax=axs[0,1])

    im = axs[0,2].imshow(σ_model_map,origin='lower',extent=(0,1,0,1),cmap='inferno'
    axs[0,2].set_title('Model σ');    fig.colorbar(im,ax=axs[0,2])

    im = axs[1,0].imshow(σ_noise_map,origin='lower',extent=(0,1,0,1),cmap='magma')
    axs[1,0].set_title('Noise σ');    fig.colorbar(im,ax=axs[1,0])

    axs[1,2].plot(cost_dyn,      label='dynamic')
    axs[1,2].plot(cost_base,'--',label='baseline 5×5/80s')
    axs[1,2].set_title(f'Cumulative cost   MAPE {mape:.2f}%')
    axs[1,2].set_xlabel('block'); axs[1,2].set_ylabel('time (s)')
    axs[1,2].grid(ls='--',alpha=0.5); axs[1,2].legend()

    plt.tight_layout()
    plt.show()


# ──────────────────────────────────────────────────────────────────────────
# 5) INITIAL GLOBAL DRILLS (no safe-band)
# ──────────────────────────────────────────────────────────────────────────
all_x, all_y, all_z = [], [], []
cost_dyn, cost_base = [], []
tot_cost_dyn = tot_cost_base = 0.0
visited_globals = []

rng = np.random.default_rng(SEED)
init_idxs = rng.choice(len(global_pool), size=n_initial, replace=False)
for idx in sorted(init_idxs, reverse=True):
    gx, gy = global_pool.pop(idx)
    visited_globals.append((gx,gy))
    print(f"[Init] drilling global centre = ({gx},{gy}) @ 5×5, hold={max_hold}s")
```

```
    blk = offsets_5x5 + np.array([gx,gy])
    xs, ys = blk[:,0], blk[:,1]
    zs = np.array([measure_from_Hgrid_hetero(x,y) for x,y in zip(xs,ys)],
                  dtype=np.float32)

    all_x.extend(xs); all_y.extend(ys); all_z.extend(zs.tolist())

    tm = default_t_measurement + max_hold
    cm = evaluate_sample_cost_remaining(xs, ys,
           t_drift       = default_t_drift,
           t_measurement = tm,
           t_comeback    = t_comeback)[0]
    step_cost = t_setup + t_move + cm

    tot_cost_dyn  += step_cost
    tot_cost_base += step_cost
    cost_dyn.append(tot_cost_dyn)
    cost_base.append(tot_cost_base)

g_curr    = g_max
prev_hold = max_hold

# ──────────────────────────────────────────────────────────────────────
# 6) MAIN ACTIVE-LEARNING LOOP
# ──────────────────────────────────────────────────────────────────────
for step in range(1, n_steps+1):
    print(f"\n===== ITER {step} =====")
    # ── 6.1) Fit hetero-GP ──────────────────────────────────────
    train_inputs  = torch.tensor(
        np.column_stack([all_x, all_y]) / domain_size,
        dtype=torch.float32
    )
    train_targets = torch.tensor(
        np.array(all_z, dtype=np.float32),
        dtype=torch.float32
    )
    M = min(300, train_inputs.size(0))
    inducing = train_inputs[torch.randperm(train_inputs.size(0))[:M]] \
            + 1e-3*torch.randn(M,2)
    gp_m, gp_s = GPModel(inducing), GPModel(inducing)
    train_hetero(train_inputs, train_targets, gp_m, gp_s, iters=250, lr=1e-2)
    print(f"→ GP fit on {train_targets.shape[0]} points")

    # ── 6.2) GLOBAL acquisition & debug ─────────────────────────
    acqs = []
    for (cx, cy) in global_pool:
        block = offsets_5x5[:g_curr**2] + np.array([cx, cy])
        Xb    = torch.tensor(block / domain_size, dtype=torch.float32)
        _, _, var_lat = predict_hetero(gp_m, gp_s, Xb)
        sigma_block  = np.sqrt(var_lat).mean()
        est_cost     = evaluate_sample_cost_remaining(
                          block[:,0], block[:,1],
                          t_drift       = default_t_drift,
                          t_measurement = default_t_measurement + prev_hold,
                          t_comeback    = t_comeback
                       )[0]
```

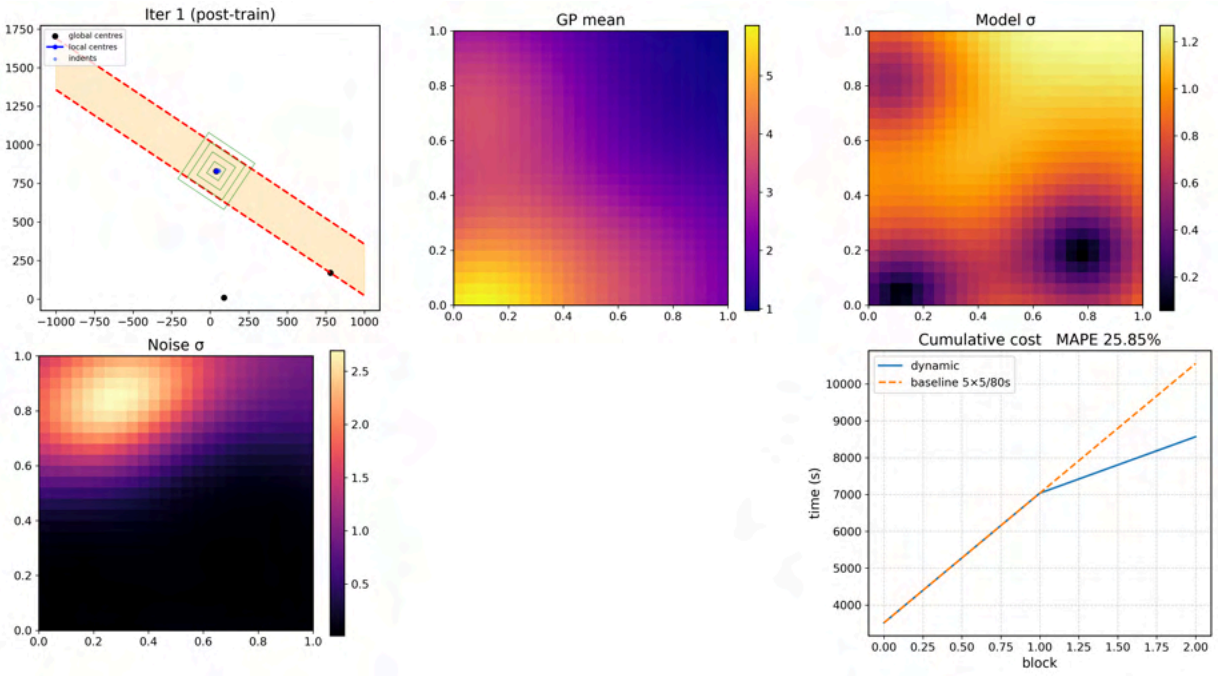
```
        acqs.append(sigma_block / (t_setup + t_move + est_cost))

    acqs = np.array(acqs)

    # debug top-4
    ranked4 = sorted(zip(global_pool, acqs),
                     key=lambda iv: iv[1], reverse=True)[:4]
    print(f"[Iter {step}] Top-4 globals (cx,cy → UE/cost):")
    for (cx, cy), score in ranked4:
        print(f"   ({cx:6.1f},{cy:6.1f}) → {score:.6f}")

    # pick best
    best_idx = int(np.argmax(acqs))
    gx, gy   = global_pool.pop(best_idx)
    visited_globals.append((gx, gy))
    print(f"→ Selected global = ({gx:.1f},{gy:.1f}), score = {acqs[best_idx]:.6f}")


    # 6.3) grid-size adapt via block σ
    blk25 = offsets_5x5 + np.array([gx,gy])
    X25   = torch.tensor(blk25/domain_size,dtype=torch.float32)
    _, noise25, var25 = predict_hetero(gp_m, gp_s, X25)
    σ_blk = np.sqrt(var25).mean().item()
    if target_sem>0:
        n_req = int(np.ceil((σ_blk/target_sem)**2))
        g_curr = min(max(int(np.ceil(np.sqrt(n_req))),g_min),g_max)
    else:
        g_curr = g_max
    print(f"→ block σ={σ_blk:.4f} → g_curr={g_curr}")

    # 6.4) find local centres in band
    z0 = a*gx + b*gy + c
    local_centres = []
    for k, uv_list in rings.items():
        for u,v in uv_list:
            xc = gx + u*e_par[0] + v*e_perp[0]
            yc = gy + u*e_par[1] + v*e_perp[1]
            if point_in_band_shifted(xc,yc,z0):
                local_centres.append((xc,yc))
                if len(local_centres)>=num_center: break
        if len(local_centres)>=num_center: break
    print(f"→ Local centres: {local_centres}")

    # 6.5) compute cost_vec
    all_loc = np.vstack([offsets_5x5[:g_curr**2]+np.array(c)
                         for c in local_centres])
    cost_vec = evaluate_sample_cost_remaining(
        all_loc[:,0], all_loc[:,1],
        t_drift       = default_t_drift,
        t_measurement = default_t_measurement+prev_hold,
        t_comeback    = t_comeback
    )
    print(f"→ cost_vec length {len(cost_vec)}")

        # … after you've built local_centres and precomputed cost_vec …
```

```
    # 6.6) LOCAL loop with "stay-vs-go" decision
    drilled_centres, drilled_points = [], []
    local_i = 0
    gpts    = g_curr**2

    while local_i < len(local_centres):
        lx, ly = local_centres[local_i]
        print(f"    → drilling local centre #{local_i} = ({lx:.1f},{ly:.1f})")
        blk_local = offsets_5x5[:gpts] + np.array([lx, ly])
        xs_loc, ys_loc = blk_local[:,0], blk_local[:,1]
        zs_loc = np.array([measure_from_Hgrid_hetero(x,y)
                           for x,y in zip(xs_loc,ys_loc)], dtype=np.float32)

        # record the drill
        drilled_centres.append((lx,ly))
        drilled_points.extend(blk_local.tolist())
        all_x.extend(xs_loc); all_y.extend(ys_loc); all_z.extend(zs_loc.tolist())

        # ❶ compute Δ-cost of drilling this one local grid
        i0 = local_i * gpts
        i1 = i0 + gpts
        c0 = cost_vec[i0]
        c1 = cost_vec[i1] if i1 < len(cost_vec) else 0.0
        Δc = c0 - c1
        tot_cost_dyn += Δc
        cost_dyn.append(tot_cost_dyn)
        print(f"       Δ-cost = {Δc:.2f} s, tot_cost_dyn = {tot_cost_dyn:.2f} s")

        # quick GP update with new local measurements

        # build tensors for the new local block
        X_loc = torch.tensor(blk_local / domain_size, dtype=torch.float32)  # shape
        y_loc = torch.tensor(zs_loc,            dtype=torch.float32)        # shape

        # concatenate into your training set
        train_inputs  = torch.cat([train_inputs,  X_loc], dim=0)
        train_targets = torch.cat([train_targets, y_loc], dim=0)

        print("  → After concat: inputs:", train_inputs.shape,
            " targets:", train_targets.shape)

        # refit or continue training
        train_hetero(train_inputs, train_targets, gp_m, gp_s, iters=120, lr=5e-3)


        # ❶ recompute best possible GLOBAL gain (UE/cost) over remaining global_poo
        best_global = -np.inf
        for (cxg, cyg) in global_pool:
            cand = offsets_5x5[:gpts] + np.array([cxg, cyg])
            _, _, var_lat = predict_hetero(
                gp_m, gp_s,
                torch.tensor(cand/domain_size, dtype=torch.float32)
            )
            ug = np.sqrt(var_lat).mean()
            cm = evaluate_sample_cost_remaining(
                cand[:,0], cand[:,1],
```

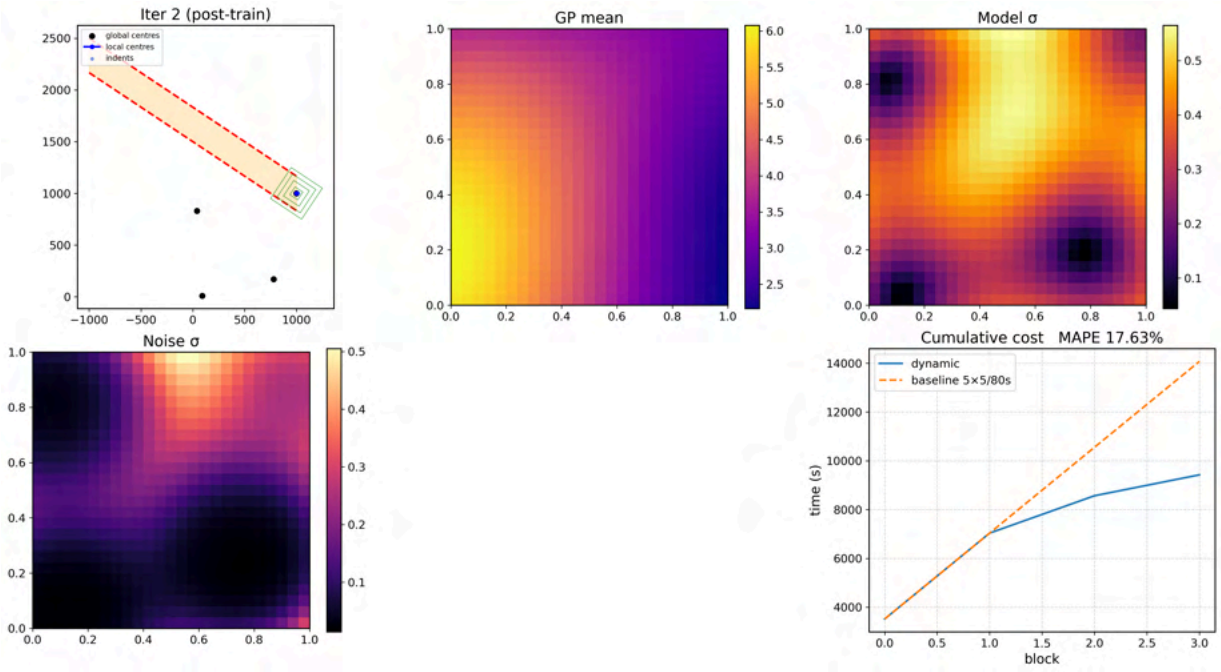
```
                t_drift       = default_t_drift,
                t_measurement = default_t_measurement + prev_hold,
                t_comeback    = t_comeback
            )[0]
            best_global = max(best_global, ug / (t_setup + t_move + cm))

        # ❷ compute the GAIN of drilling **one more** local grid
        if local_i+1 < len(local_centres):
            start = (local_i+1)*gpts
            nxt_block = all_loc[start : start + gpts]
            _, _, var_nxt = predict_hetero(
                gp_m, gp_s,
                torch.tensor(nxt_block/domain_size, dtype=torch.float32)
            )
            ug_next = np.sqrt(var_nxt).mean()
            # Δ-cost for that next grid:
            c0n = cost_vec[(local_i+1)*gpts]
            c1n = cost_vec[(local_i+2)*gpts] if (local_i+2)*gpts < len(cost_vec) el
            delta_next = c0n - c1n
            gain_loc = ug_next / max(delta_next, 1e-6)
        else:
            gain_loc = -np.inf

        print(f"       gain_loc = {gain_loc:.6f}, best_global = {best_global:.6f}")

        # ❸ the "stay-vs-go" test
        if gain_loc >= best_global:
            print("       ⇒ STAY local for one more grid")
            local_i += 1
            continue
        else:
            print("       ⇒ GO to next global centre")
            break


    # 6.7) drift-hold decision
    t_now = tot_cost_dyn
    in_evt = (t_event<=t_now<=t_event_end)
    err_5  = (0.5*np.exp(-(t_now-t_event)/decay_tau)) if in_evt else 0.0

    μ_blk = gp_m(X25).mean.mean().item()
    # μ_blk = μ_blk*(y.max()-y.min()) + y.min()
    y_arr = np.array(all_z, dtype=np.float32)
    y_range = y_arr.max() - y_arr.min()
    μ_blk = μ_blk * y_range + y_arr.min()
    T_var = 0.0
    T_var = -1 ######################### deciding it cannot be 0
    T_big = max(user_threshold_pct*μ_blk/100.,2*T_var)
    if prev_hold==0:
        hold = med_hold if (step%eval_interval)==0 else min_hold
    else:
        hold = (min_hold if err_5<=T_var
                else med_hold if err_5<T_big
                else max_hold)
    prev_hold = hold
    print(f"→ hold={hold}s  (err_5={err_5:.4f}, T_big={T_big:.4f})")
```

```
    # 6.8) pad baseline
    ref_xy = offsets_ref + np.array([gx,gy])
    cb     = evaluate_sample_cost_remaining(
        ref_xy[:,0], ref_xy[:,1],
        t_drift       = default_t_drift,
        t_measurement = default_t_measurement+max_hold,
        t_comeback    = t_comeback)[0]
    one5x5 = t_setup + t_move + cb
    while len(cost_base)<len(cost_dyn):
        tot_cost_base += one5x5
        cost_base.append(tot_cost_base)
    print(f"→ baseline padded to {len(cost_base)} steps")

    # 6.9) plot
    if step%plot_interval==0:
        plot_state(step, gp_m, gp_s, Z_true,
                   rings, a,b,c, dz_safe, domain_size,
                   (gx,gy), visited_globals,
                   drilled_centres, drilled_points,
                   cost_dyn, cost_base,
                   title_suffix=" (post-train)")
```

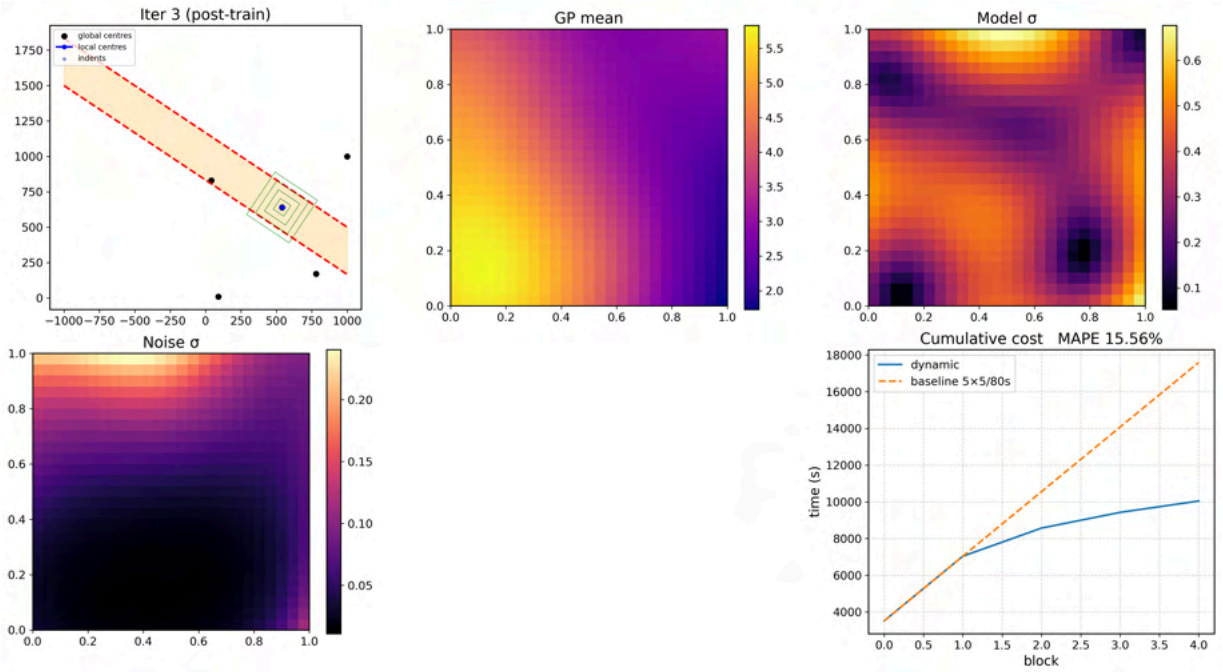
```
[Init] drilling global centre = (780,170) @ 5×5, hold=80s
[Init] drilling global centre = (90,10) @ 5×5, hold=80s

===== ITER 1 =====
→ GP fit on 50 points
[Iter 1] Top-4 globals (cx,cy → UE/cost):
   (  40.0, 830.0) → 0.000360
   (  50.0, 830.0) → 0.000360
   (  30.0, 830.0) → 0.000360
   (  40.0, 820.0) → 0.000360
→ Selected global = (40.0,830.0), score = 0.000360
→ block σ=1.2655 → g_curr=3
→ Local centres: [(40.0, 830.0)]
→ cost_vec length 9
    → drilling local centre #0 = (40.0,830.0)
       Δ-cost = 1531.89 s, tot_cost_dyn = 8567.89 s
  → After concat: inputs: torch.Size([59, 2])  targets: torch.Size([59])
       gain_loc = -inf, best_global = 0.000811
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.1042, T_big=0.5537)
→ baseline padded to 3 steps
```



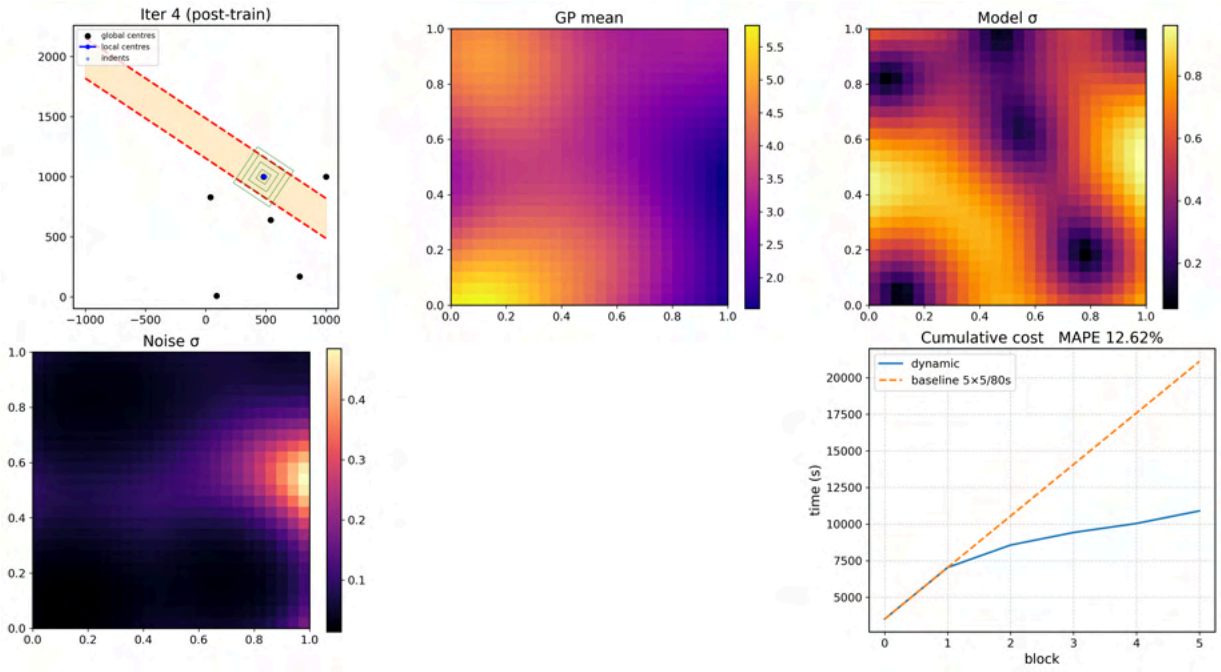

```
===== ITER 2 =====
→ GP fit on 59 points
[Iter 2] Top-4 globals (cx,cy → UE/cost):
   (1000.0,1000.0) → 0.001247
   ( 990.0,1000.0) → 0.001245
   (1000.0, 990.0) → 0.001242
   ( 980.0,1000.0) → 0.001242
→ Selected global = (1000.0,1000.0), score = 0.001247
→ block σ=1.1150 → g_curr=3
→ Local centres: [(1000.0, 1000.0)]
→ cost_vec length 9
    → drilling local centre #0 = (1000.0,1000.0)
       Δ-cost = 856.89 s, tot_cost_dyn = 9424.77 s
  → After concat: inputs: torch.Size([68, 2])  targets: torch.Size([68])
       gain_loc = -inf, best_global = 0.000635
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0442, T_big=0.4971)
→ baseline padded to 4 steps
```



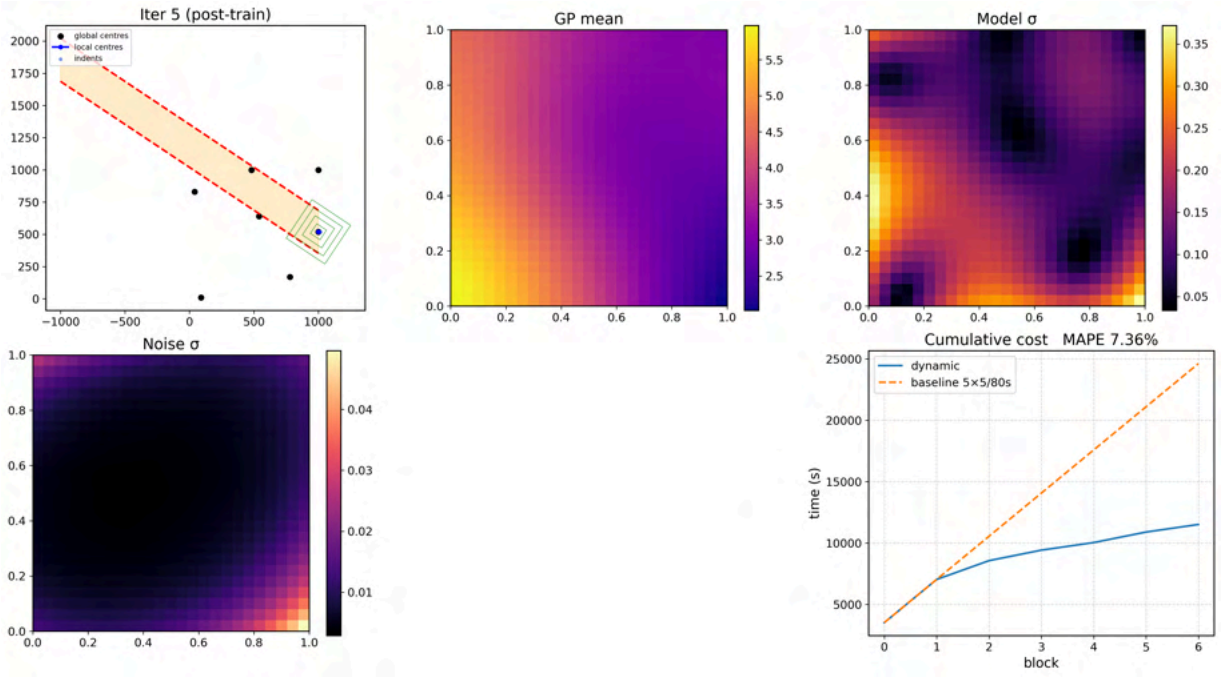

```
===== ITER 3 =====
→ GP fit on 68 points
[Iter 3] Top-4 globals (cx,cy → UE/cost):
   ( 540.0, 640.0) → 0.000899
   ( 540.0, 630.0) → 0.000899
   ( 530.0, 630.0) → 0.000899
   ( 530.0, 640.0) → 0.000899
→ Selected global = (540.0,640.0), score = 0.000899
→ block σ=0.8005 → g_curr=2
→ Local centres: [(540.0, 640.0)]
→ cost_vec length 4
    → drilling local centre #0 = (540.0,640.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 10042.96 s
  → After concat: inputs: torch.Size([72, 2])  targets: torch.Size([72])
       gain_loc = -inf, best_global = 0.001032
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0238, T_big=0.5280)
→ baseline padded to 5 steps
```



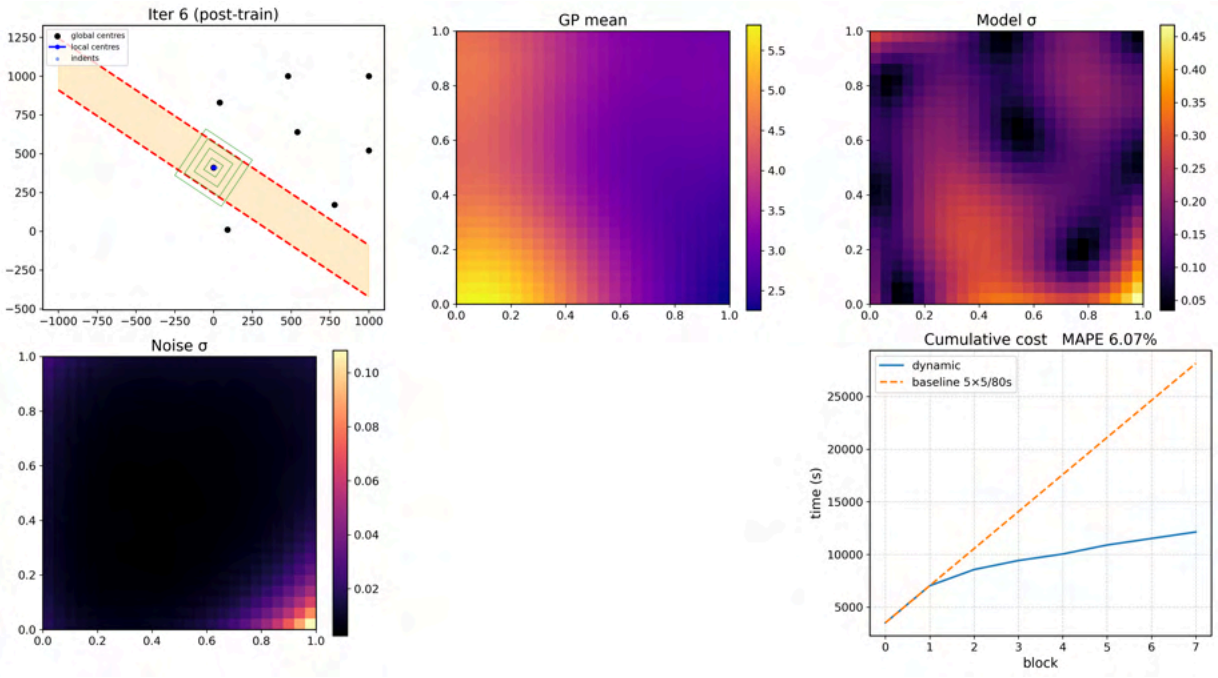

```
===== ITER 4 =====
→ GP fit on 72 points
[Iter 4] Top-4 globals (cx,cy → UE/cost):
   ( 480.0,1000.0) → 0.001593
   ( 490.0,1000.0) → 0.001592
   ( 470.0,1000.0) → 0.001592
   ( 500.0,1000.0) → 0.001591
→ Selected global = (480.0,1000.0), score = 0.001593
→ block σ=1.0595 → g_curr=3
→ Local centres: [(480.0, 1000.0)]
→ cost_vec length 9
    → drilling local centre #0 = (480.0,1000.0)
       Δ-cost = 856.89 s, tot_cost_dyn = 10899.85 s
  → After concat: inputs: torch.Size([81, 2])  targets: torch.Size([81])
       gain_loc = -inf, best_global = 0.001090
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0101, T_big=0.5579)
→ baseline padded to 6 steps
```



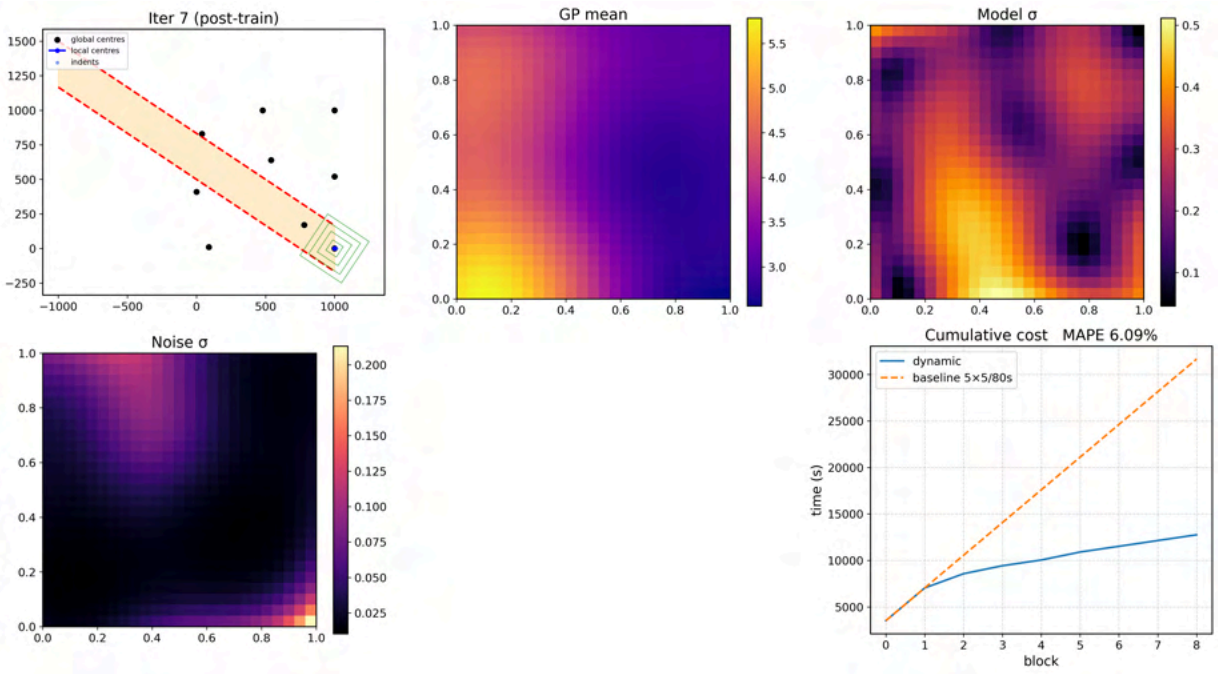

```
===== ITER 5 =====
→ GP fit on 81 points
[Iter 5] Top-4 globals (cx,cy → UE/cost):
   (1000.0, 520.0) → 0.000531
   (1000.0, 510.0) → 0.000531
   (1000.0, 530.0) → 0.000531
   (1000.0, 500.0) → 0.000531
→ Selected global = (1000.0,520.0), score = 0.000531
→ block σ=0.4742 → g_curr=2
→ Local centres: [(1000.0, 520.0)]
→ cost_vec length 4
    → drilling local centre #0 = (1000.0,520.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 11518.04 s
  → After concat: inputs: torch.Size([85, 2])  targets: torch.Size([85])
       gain_loc = -inf, best_global = 0.000578
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0055, T_big=0.4786)
→ baseline padded to 7 steps
```



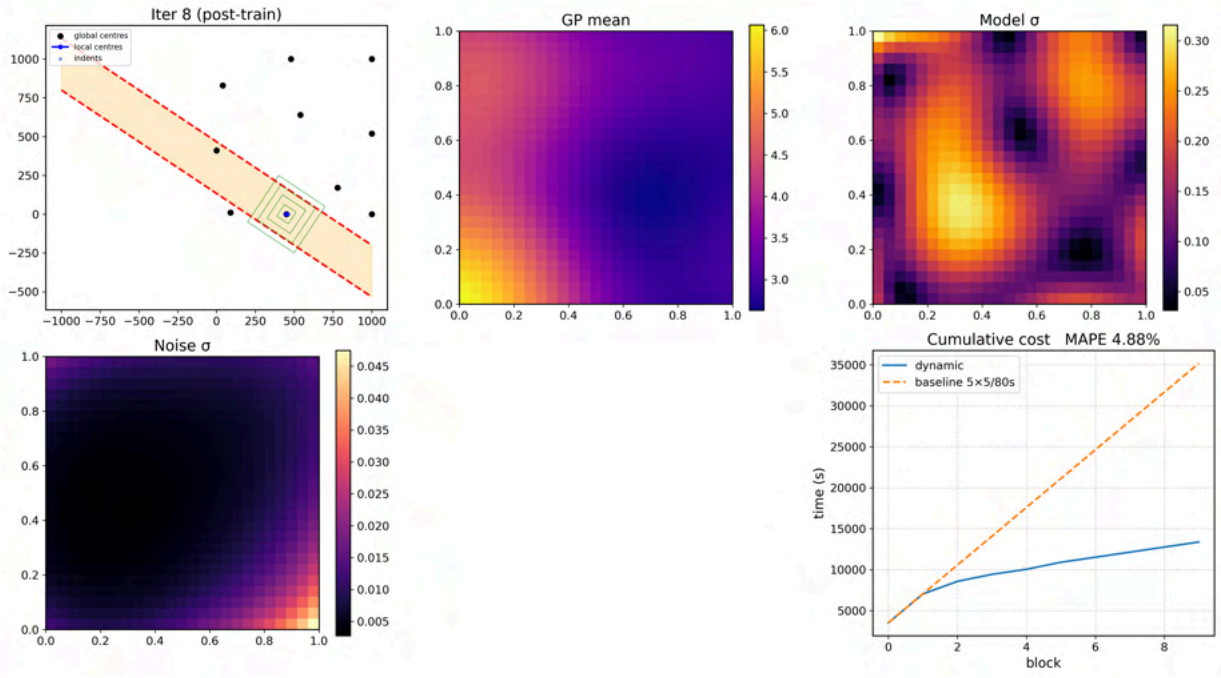

```
===== ITER 6 =====
→ GP fit on 85 points
[Iter 6] Top-4 globals (cx,cy → UE/cost):
   (   0.0, 410.0) → 0.000722
   (1000.0,   0.0) → 0.000722
   (   0.0, 420.0) → 0.000721
   (   0.0, 400.0) → 0.000721
→ Selected global = (0.0,410.0), score = 0.000722
→ block σ=0.4713 → g_curr=2
→ Local centres: [(0.0, 410.0)]
→ cost_vec length 4
    → drilling local centre #0 = (0.0,410.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 12136.23 s
  → After concat: inputs: torch.Size([89, 2])  targets: torch.Size([89])
       gain_loc = -inf, best_global = 0.000727
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0029, T_big=0.7073)
→ baseline padded to 8 steps
```



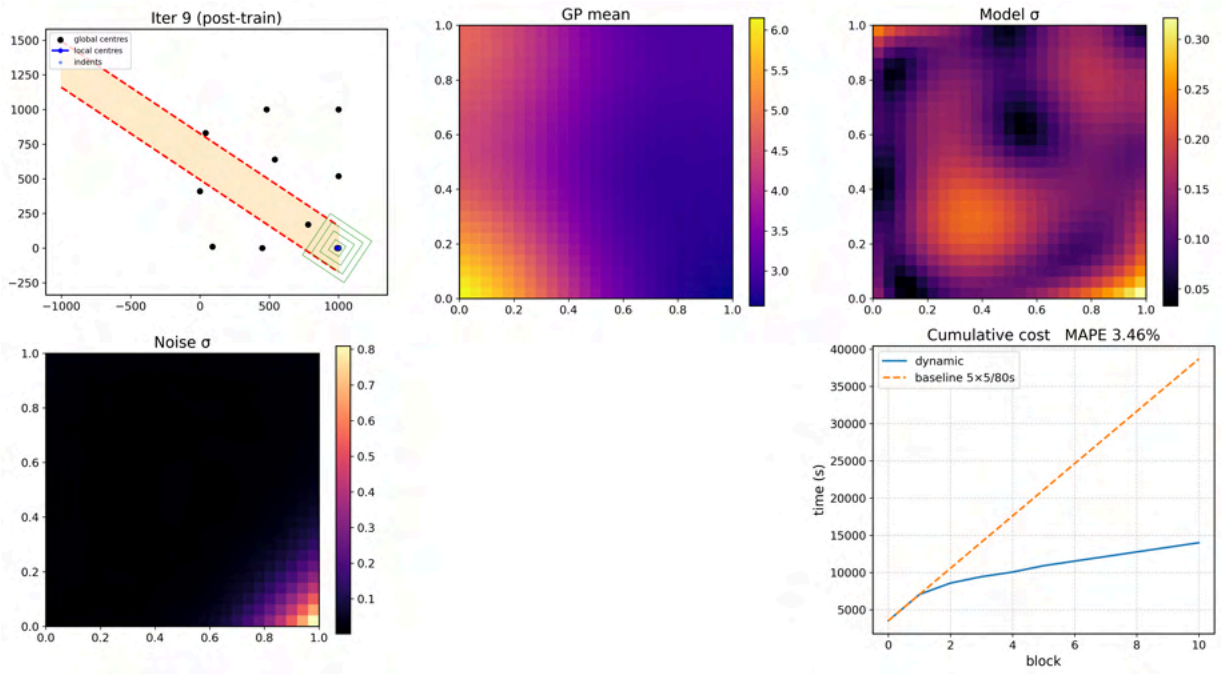

```
===== ITER 7 =====
→ GP fit on 89 points
[Iter 7] Top-4 globals (cx,cy → UE/cost):
   (1000.0,   0.0) → 0.001136
   (1000.0,  10.0) → 0.001114
   ( 990.0,   0.0) → 0.001114
   (1000.0,  20.0) → 0.001092
→ Selected global = (1000.0,0.0), score = 0.001136
→ block σ=0.7346 → g_curr=2
→ Local centres: [(1000.0, 0.0)]
→ cost_vec length 4
    → drilling local centre #0 = (1000.0,0.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 12754.42 s
  → After concat: inputs: torch.Size([93, 2])  targets: torch.Size([93])
       gain_loc = -inf, best_global = 0.000779
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0016, T_big=0.4460)
→ baseline padded to 9 steps
```



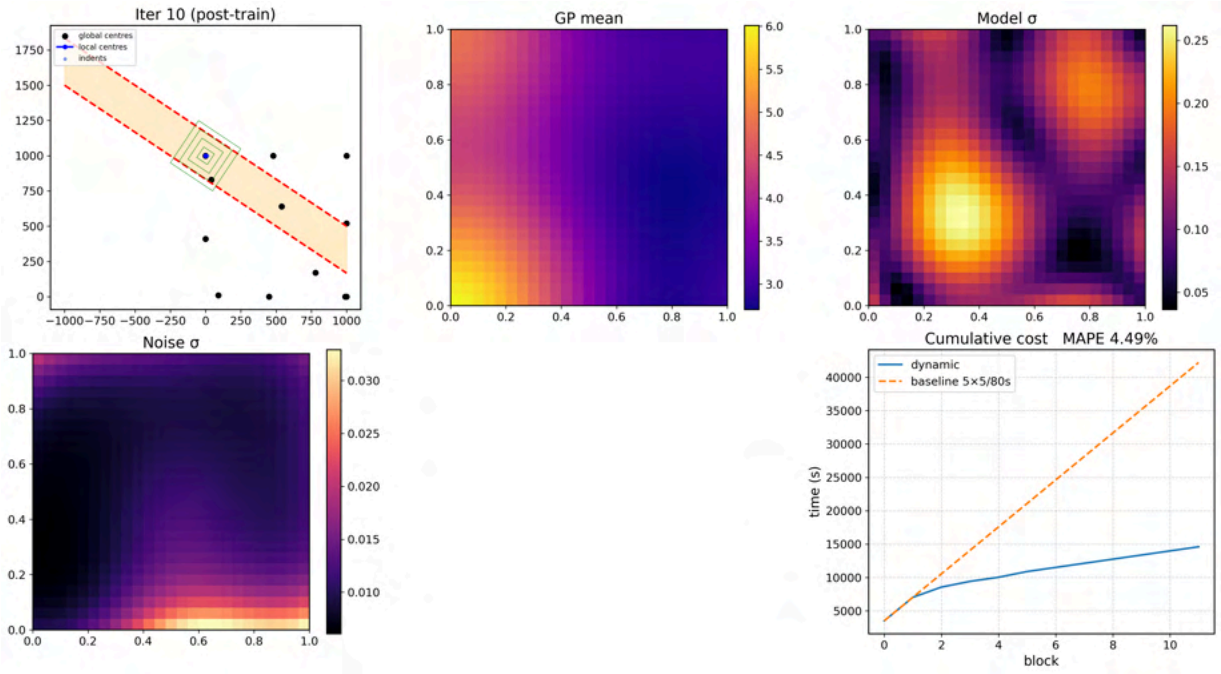

```
===== ITER 8 =====
→ GP fit on 93 points
[Iter 8] Top-4 globals (cx,cy → UE/cost):
   ( 450.0,   0.0) → 0.000696
   ( 440.0,   0.0) → 0.000696
   ( 460.0,   0.0) → 0.000695
   ( 430.0,   0.0) → 0.000695
→ Selected global = (450.0,0.0), score = 0.000696
→ block σ=0.4508 → g_curr=2
→ Local centres: [(450.0, 0.0)]
→ cost_vec length 4
    → drilling local centre #0 = (450.0,0.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 13372.61 s
  → After concat: inputs: torch.Size([97, 2])  targets: torch.Size([97])
       gain_loc = -inf, best_global = 0.000472
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0009, T_big=0.6087)
→ baseline padded to 10 steps
```



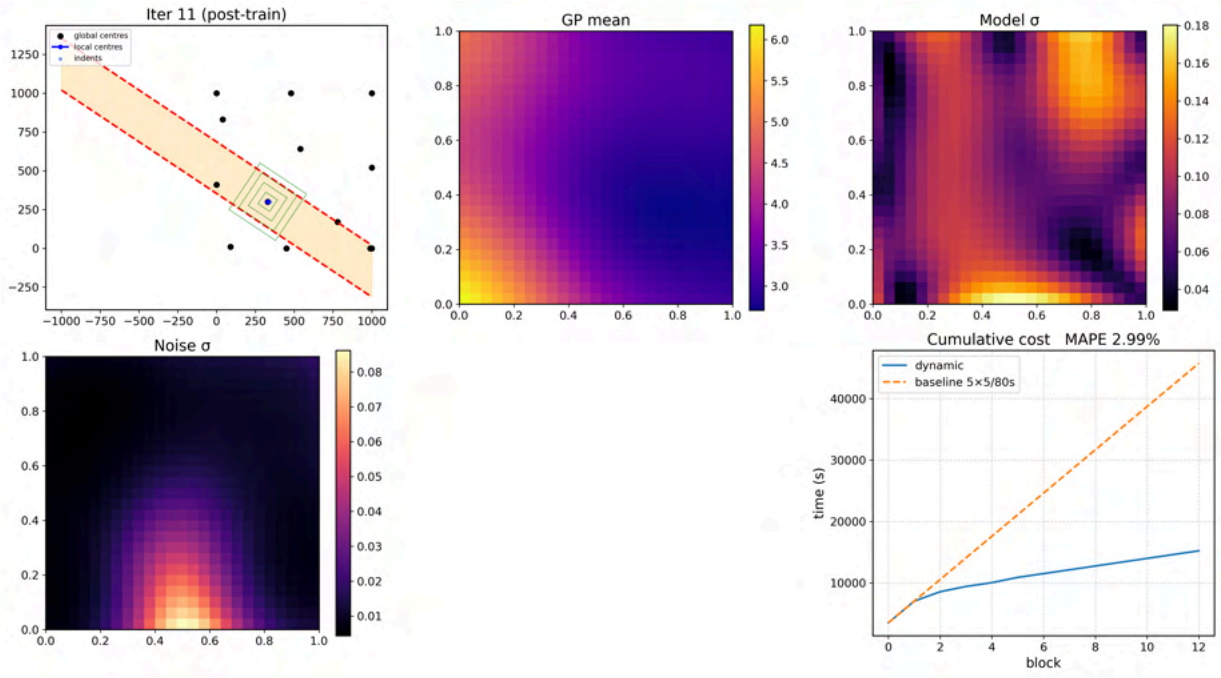

```
===== ITER 9 =====
→ GP fit on 97 points
[Iter 9] Top-4 globals (cx,cy → UE/cost):
   ( 990.0,   0.0) → 0.000676
   (1000.0,  10.0) → 0.000672
   ( 980.0,   0.0) → 0.000666
   ( 990.0,  10.0) → 0.000661
→ Selected global = (990.0,0.0), score = 0.000676
→ block σ=0.4357 → g_curr=2
→ Local centres: [(990.0, 0.0)]
→ cost_vec length 4
    → drilling local centre #0 = (990.0,0.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 13990.81 s
  → After concat: inputs: torch.Size([101, 2])  targets: torch.Size([101])
       gain_loc = -inf, best_global = 0.000481
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0005, T_big=0.4457)
→ baseline padded to 11 steps
```



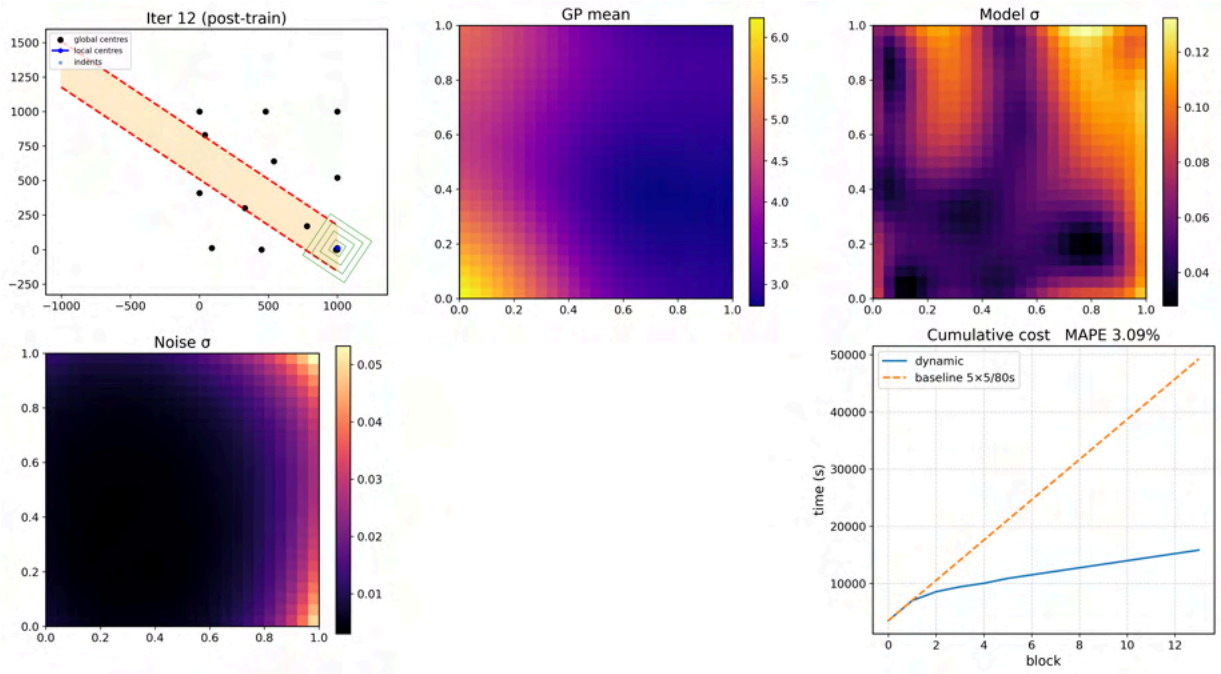

```
===== ITER 10 =====
→ GP fit on 101 points
[Iter 10] Top-4 globals (cx,cy → UE/cost):
   (   0.0,1000.0) → 0.000490
   ( 330.0, 310.0) → 0.000488
   ( 320.0, 310.0) → 0.000488
   ( 330.0, 300.0) → 0.000488
→ Selected global = (0.0,1000.0), score = 0.000490
→ block σ=0.3367 → g_curr=2
→ Local centres: [(0.0, 1000.0)]
→ cost_vec length 4
    → drilling local centre #0 = (0.0,1000.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 14609.00 s
  → After concat: inputs: torch.Size([105, 2])  targets: torch.Size([105])
       gain_loc = -inf, best_global = 0.000400
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0002, T_big=0.7474)
→ baseline padded to 12 steps
```



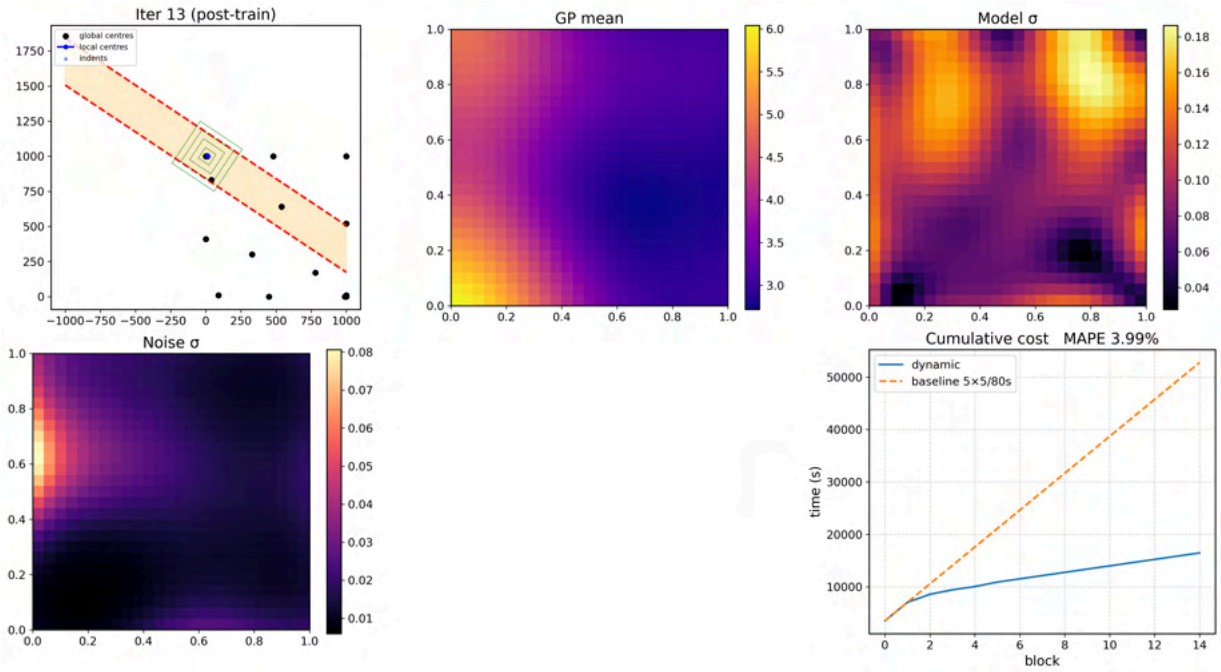

```
===== ITER 11 =====
→ GP fit on 105 points
[Iter 11] Top-4 globals (cx,cy → UE/cost):
   ( 330.0, 300.0) → 0.000382
   ( 340.0, 300.0) → 0.000382
   ( 330.0, 290.0) → 0.000382
   ( 340.0, 290.0) → 0.000382
→ Selected global = (330.0,300.0), score = 0.000382
→ block σ=0.2506 → g_curr=2
→ Local centres: [(330.0, 300.0)]
→ cost_vec length 4
    → drilling local centre #0 = (330.0,300.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 15227.19 s
  → After concat: inputs: torch.Size([109, 2])  targets: torch.Size([109])
       gain_loc = -inf, best_global = 0.000275
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0001, T_big=0.5791)
→ baseline padded to 13 steps
```



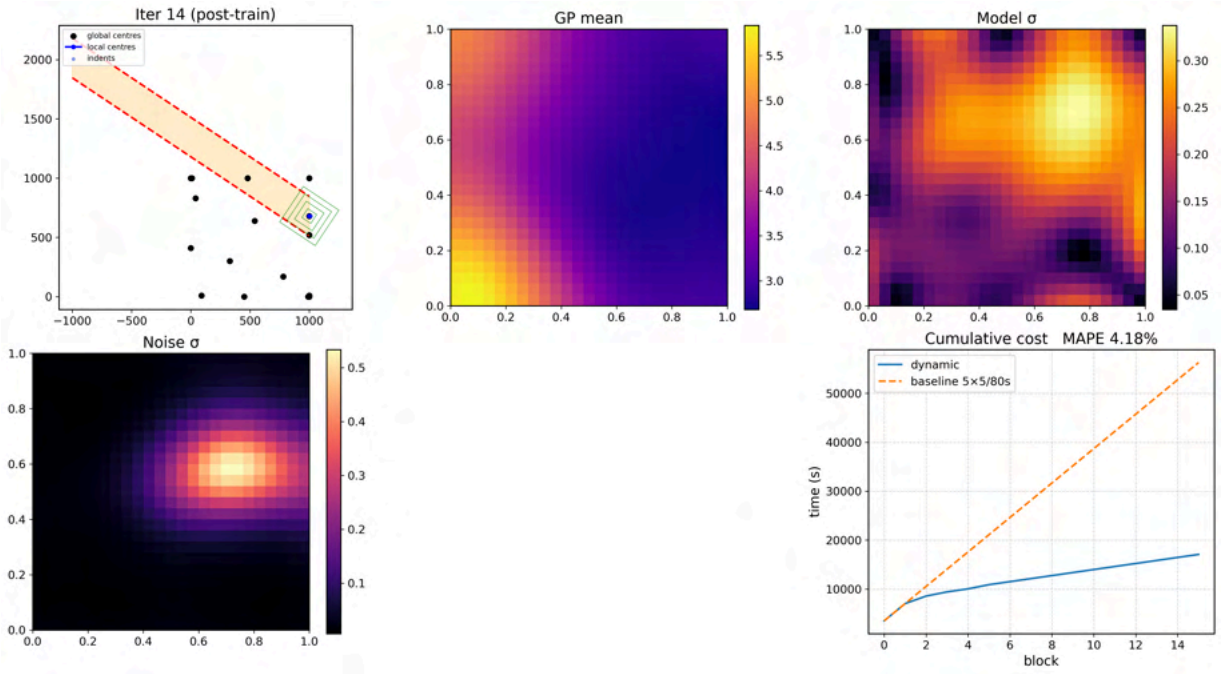

```
===== ITER 12 =====
→ GP fit on 109 points
[Iter 12] Top-4 globals (cx,cy → UE/cost):
   (1000.0,  10.0) → 0.000335
   (1000.0,  20.0) → 0.000327
   ( 990.0,  10.0) → 0.000325
   ( 980.0,   0.0) → 0.000324
→ Selected global = (1000.0,10.0), score = 0.000335
→ block σ=0.2166 → g_curr=2
→ Local centres: [(1000.0, 10.0)]
→ cost_vec length 4
    → drilling local centre #0 = (1000.0,10.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 15845.38 s
  → After concat: inputs: torch.Size([113, 2])  targets: torch.Size([113])
       gain_loc = -inf, best_global = 0.000202
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0001, T_big=0.4835)
→ baseline padded to 14 steps
```



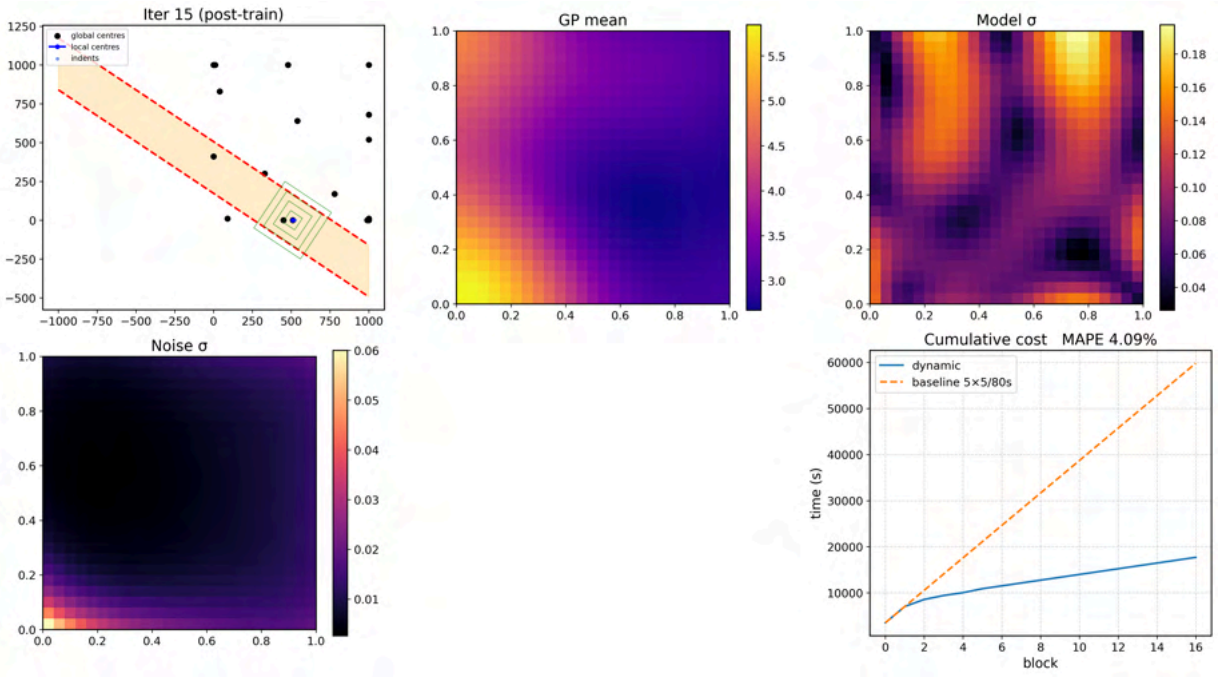

```
===== ITER 13 =====
→ GP fit on 113 points
[Iter 13] Top-4 globals (cx,cy → UE/cost):
   (  10.0,1000.0) → 0.000448
   (  20.0,1000.0) → 0.000445
   (  30.0,1000.0) → 0.000442
   (  40.0,1000.0) → 0.000439
→ Selected global = (10.0,1000.0), score = 0.000448
→ block σ=0.3035 → g_curr=2
→ Local centres: [(10.0, 1000.0)]
→ cost_vec length 4
    → drilling local centre #0 = (10.0,1000.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 16463.57 s
  → After concat: inputs: torch.Size([117, 2])  targets: torch.Size([117])
       gain_loc = -inf, best_global = 0.000285
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.7479)
→ baseline padded to 15 steps
```



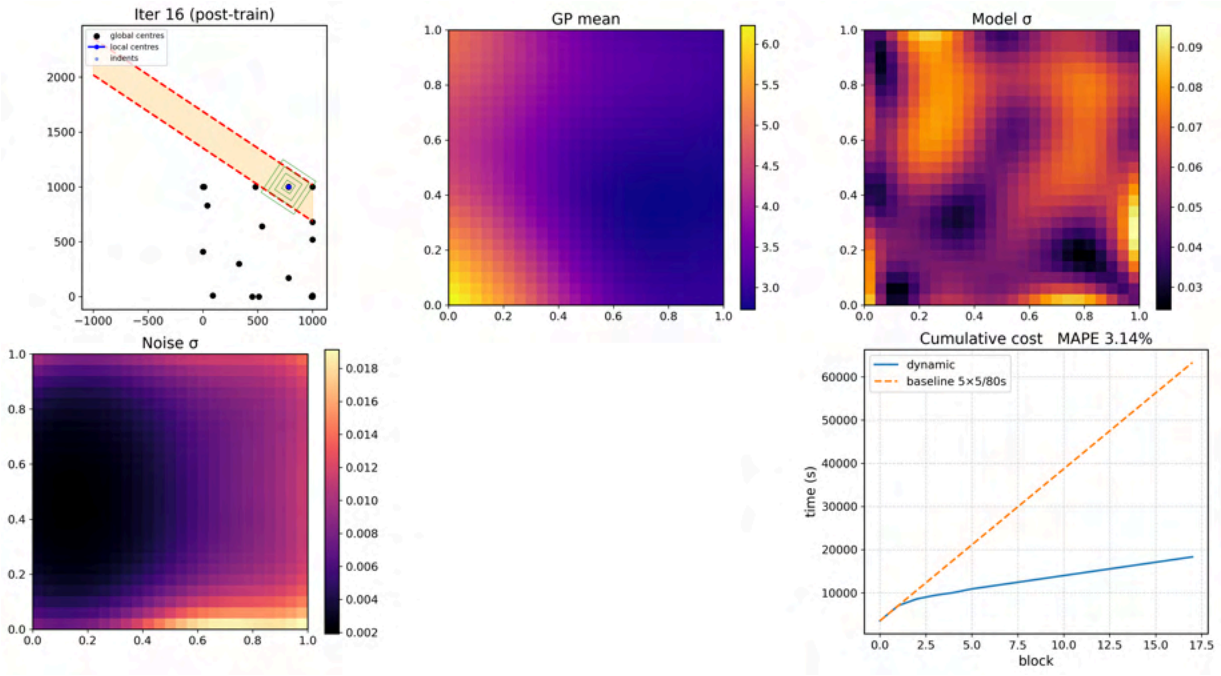

```
===== ITER 14 =====
→ GP fit on 117 points
[Iter 14] Top-4 globals (cx,cy → UE/cost):
   (1000.0, 680.0) → 0.000734
   (1000.0, 690.0) → 0.000733
   (1000.0, 670.0) → 0.000733
   (1000.0, 700.0) → 0.000732
→ Selected global = (1000.0,680.0), score = 0.000734
→ block σ=0.4818 → g_curr=2
→ Local centres: [(1000.0, 680.0)]
→ cost_vec length 4
    → drilling local centre #0 = (1000.0,680.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 17081.76 s
  → After concat: inputs: torch.Size([121, 2])  targets: torch.Size([121])
       gain_loc = -inf, best_global = 0.000515
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.4620)
→ baseline padded to 16 steps
```



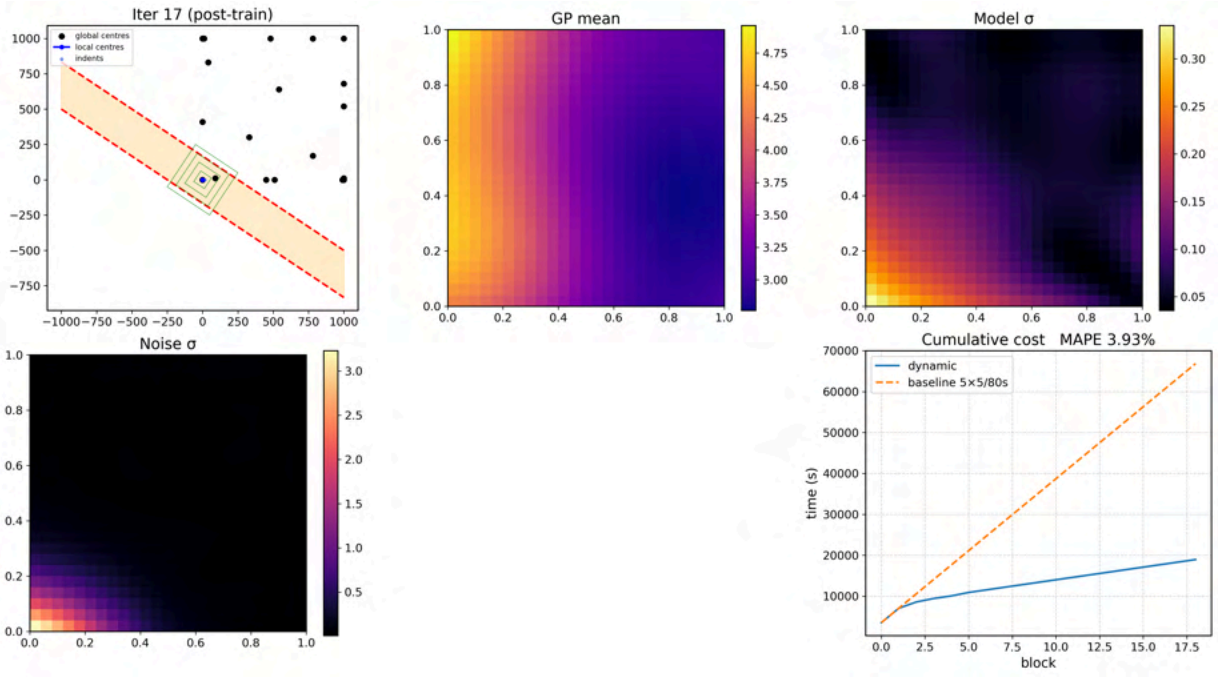

```
===== ITER 15 =====
→ GP fit on 121 points
[Iter 15] Top-4 globals (cx,cy → UE/cost):
   ( 510.0,   0.0) → 0.000469
   ( 500.0,   0.0) → 0.000469
   ( 520.0,   0.0) → 0.000469
   ( 490.0,   0.0) → 0.000469
→ Selected global = (510.0,0.0), score = 0.000469
→ block σ=0.2967 → g_curr=2
→ Local centres: [(510.0, 0.0)]
→ cost_vec length 4
    → drilling local centre #0 = (510.0,0.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 17699.95 s
  → After concat: inputs: torch.Size([125, 2])  targets: torch.Size([125])
       gain_loc = -inf, best_global = 0.000303
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.5802)
→ baseline padded to 17 steps
```



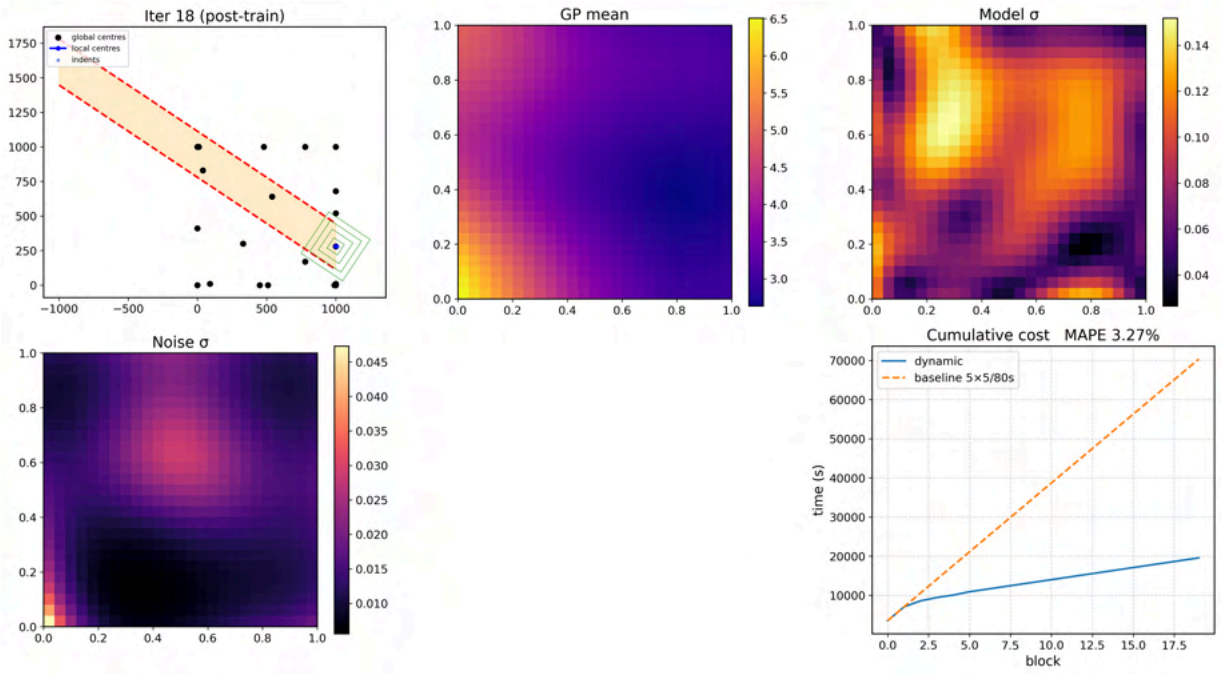

```
===== ITER 16 =====
→ GP fit on 125 points
[Iter 16] Top-4 globals (cx,cy → UE/cost):
   ( 780.0,1000.0) → 0.000238
   ( 770.0,1000.0) → 0.000238
   ( 790.0,1000.0) → 0.000237
   ( 760.0,1000.0) → 0.000237
→ Selected global = (780.0,1000.0), score = 0.000238
→ block σ=0.1586 → g_curr=2
→ Local centres: [(780.0, 1000.0)]
→ cost_vec length 4
    → drilling local centre #0 = (780.0,1000.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 18318.15 s
  → After concat: inputs: torch.Size([129, 2])  targets: torch.Size([129])
       gain_loc = -inf, best_global = 0.000152
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.5217)
→ baseline padded to 18 steps
```



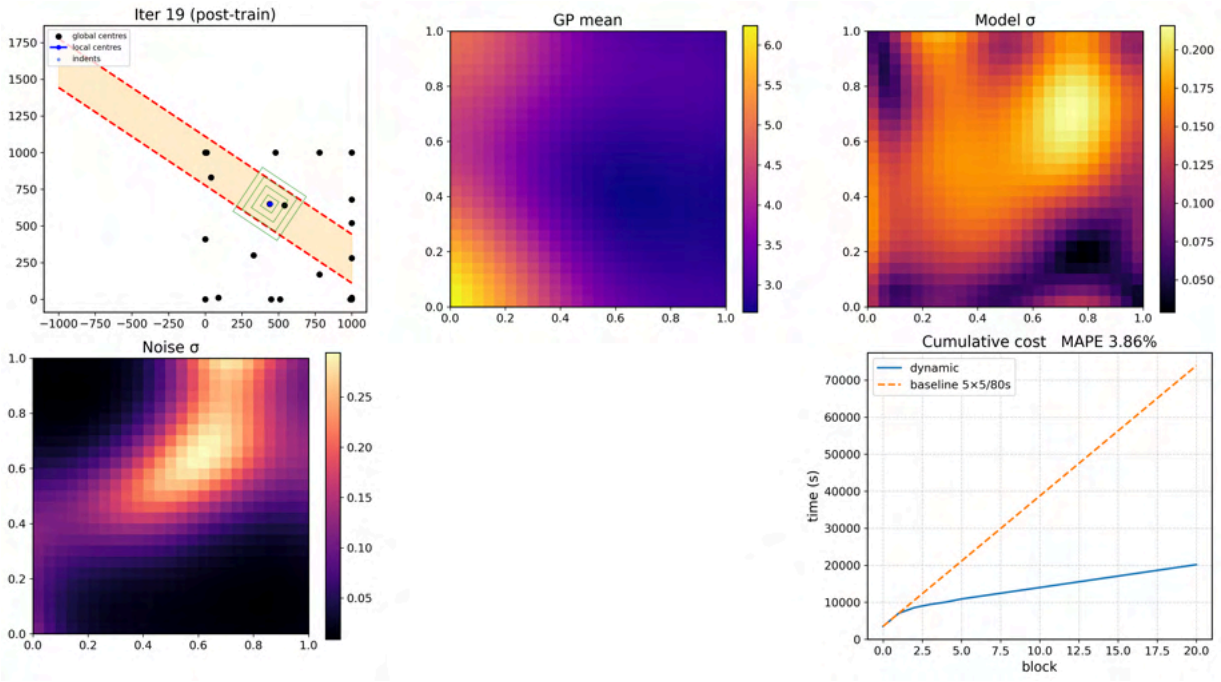

```
===== ITER 17 =====
→ GP fit on 129 points
[Iter 17] Top-4 globals (cx,cy → UE/cost):
   (   0.0,   0.0) → 0.000751
   (  10.0,   0.0) → 0.000742
   (   0.0,  10.0) → 0.000742
   (  20.0,   0.0) → 0.000733
→ Selected global = (0.0,0.0), score = 0.000751
→ block σ=0.4851 → g_curr=2
→ Local centres: [(0.0, 0.0)]
→ cost_vec length 4
    → drilling local centre #0 = (0.0,0.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 18936.34 s
  → After concat: inputs: torch.Size([133, 2])  targets: torch.Size([133])
       gain_loc = -inf, best_global = 0.000498
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.8610)
→ baseline padded to 19 steps
```



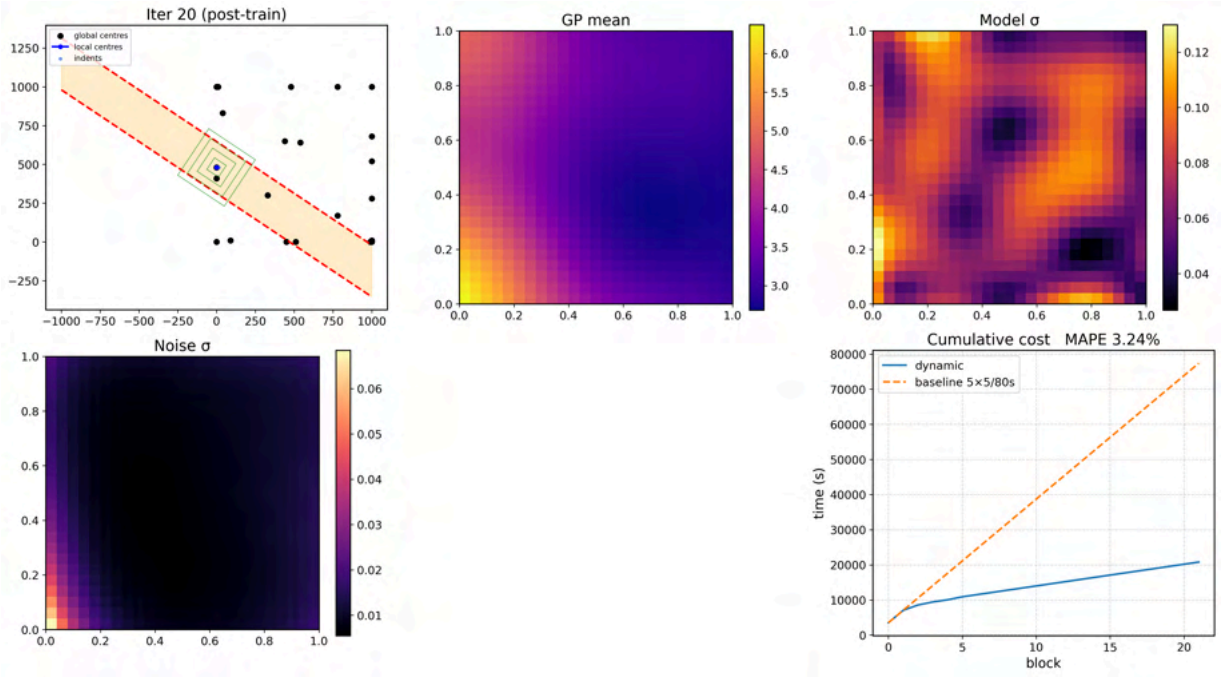

```
===== ITER 18 =====
→ GP fit on 133 points
[Iter 18] Top-4 globals (cx,cy → UE/cost):
   (1000.0, 280.0) → 0.000319
   (1000.0, 270.0) → 0.000319
   (1000.0, 290.0) → 0.000318
   ( 220.0,1000.0) → 0.000318
→ Selected global = (1000.0,280.0), score = 0.000319
→ block σ=0.2114 → g_curr=2
→ Local centres: [(1000.0, 280.0)]
→ cost_vec length 4
    → drilling local centre #0 = (1000.0,280.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 19554.53 s
  → After concat: inputs: torch.Size([137, 2])  targets: torch.Size([137])
       gain_loc = -inf, best_global = 0.000232
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.5858)
→ baseline padded to 20 steps
```



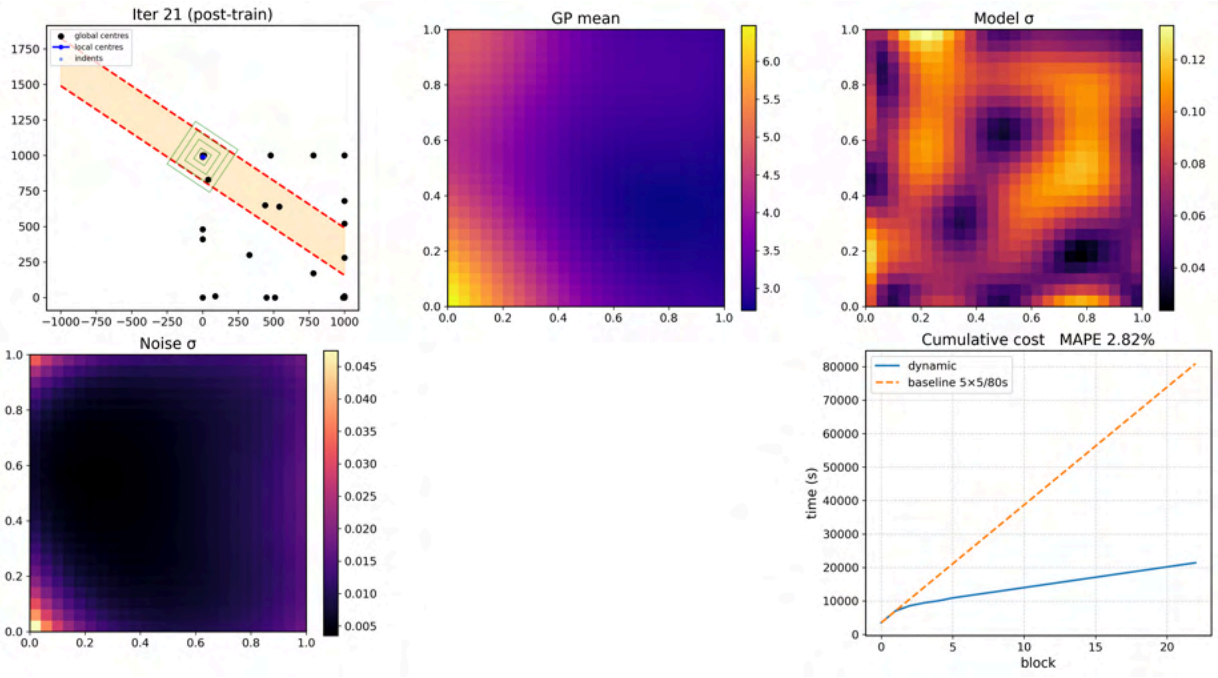

```
===== ITER 19 =====
→ GP fit on 137 points
[Iter 19] Top-4 globals (cx,cy → UE/cost):
   ( 440.0, 650.0) → 0.000553
   ( 450.0, 650.0) → 0.000553
   ( 440.0, 640.0) → 0.000553
   ( 450.0, 640.0) → 0.000553
→ Selected global = (440.0,650.0), score = 0.000553
→ block σ=0.3628 → g_curr=2
→ Local centres: [(440.0, 650.0)]
→ cost_vec length 4
    → drilling local centre #0 = (440.0,650.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 20172.72 s
  → After concat: inputs: torch.Size([141, 2])  targets: torch.Size([141])
       gain_loc = -inf, best_global = 0.000329
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.6529)
→ baseline padded to 21 steps
```



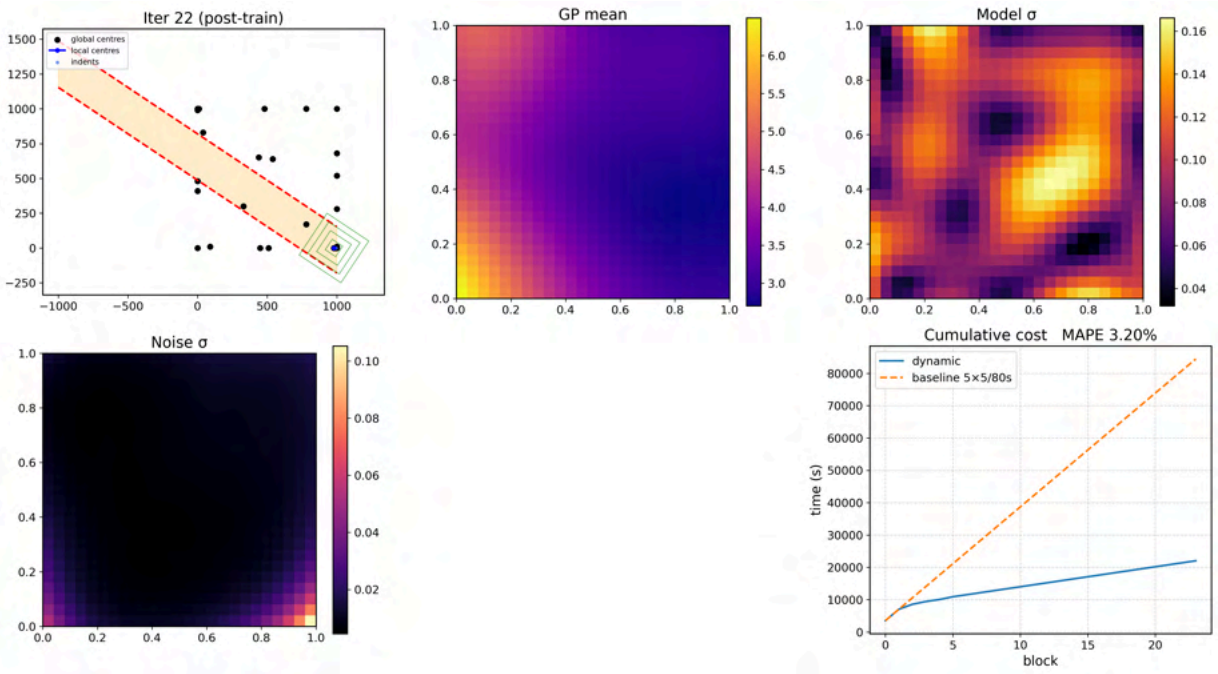

```
===== ITER 20 =====
→ GP fit on 141 points
[Iter 20] Top-4 globals (cx,cy → UE/cost):
   (   0.0, 480.0) → 0.000322
   (   0.0, 490.0) → 0.000322
   (   0.0, 470.0) → 0.000322
   (   0.0, 460.0) → 0.000322
→ Selected global = (0.0,480.0), score = 0.000322
→ block σ=0.2100 → g_curr=2
→ Local centres: [(0.0, 480.0)]
→ cost_vec length 4
    → drilling local centre #0 = (0.0,480.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 20790.91 s
  → After concat: inputs: torch.Size([145, 2])  targets: torch.Size([145])
       gain_loc = -inf, best_global = 0.000192
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.8694)
→ baseline padded to 22 steps
```



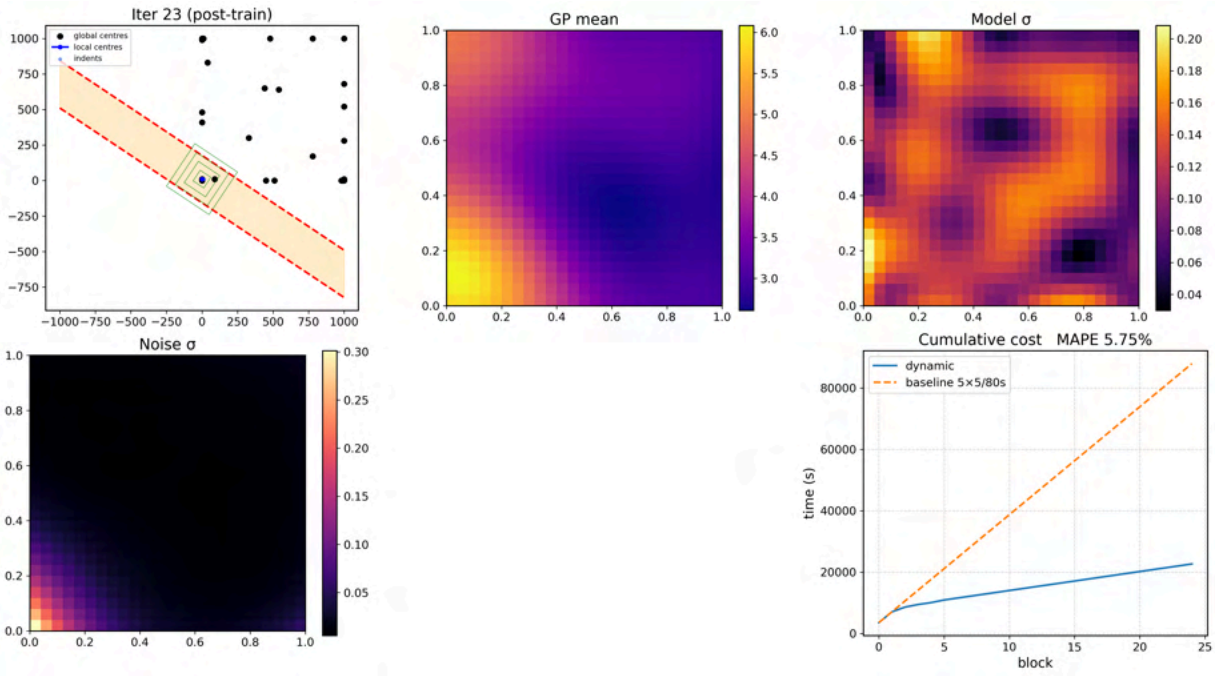

```
===== ITER 21 =====
→ GP fit on 145 points
[Iter 21] Top-4 globals (cx,cy → UE/cost):
   (   0.0, 990.0) → 0.000369
   (  20.0,1000.0) → 0.000367
   (  10.0, 990.0) → 0.000360
   (  30.0,1000.0) → 0.000360
→ Selected global = (0.0,990.0), score = 0.000369
→ block σ=0.2504 → g_curr=2
→ Local centres: [(0.0, 990.0)]
→ cost_vec length 4
    → drilling local centre #0 = (0.0,990.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 21409.10 s
  → After concat: inputs: torch.Size([149, 2])  targets: torch.Size([149])
       gain_loc = -inf, best_global = 0.000203
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.9534)
→ baseline padded to 23 steps
```



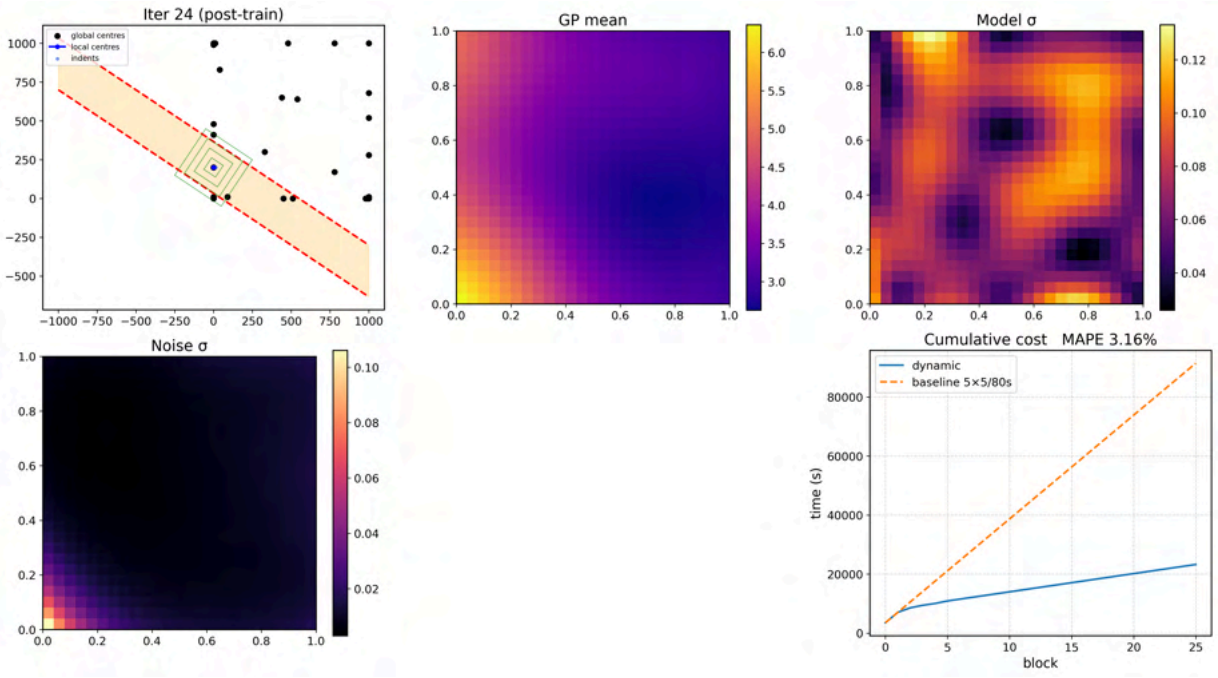

```
===== ITER 22 =====
→ GP fit on 149 points
[Iter 22] Top-4 globals (cx,cy → UE/cost):
   ( 980.0,   0.0) → 0.000439
   (1000.0,  20.0) → 0.000439
   ( 990.0,  10.0) → 0.000438
   ( 970.0,   0.0) → 0.000432
→ Selected global = (980.0,0.0), score = 0.000439
→ block σ=0.2830 → g_curr=2
→ Local centres: [(980.0, 0.0)]
→ cost_vec length 4
    → drilling local centre #0 = (980.0,0.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 22027.29 s
  → After concat: inputs: torch.Size([153, 2])  targets: torch.Size([153])
       gain_loc = -inf, best_global = 0.000254
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.6219)
→ baseline padded to 24 steps
```



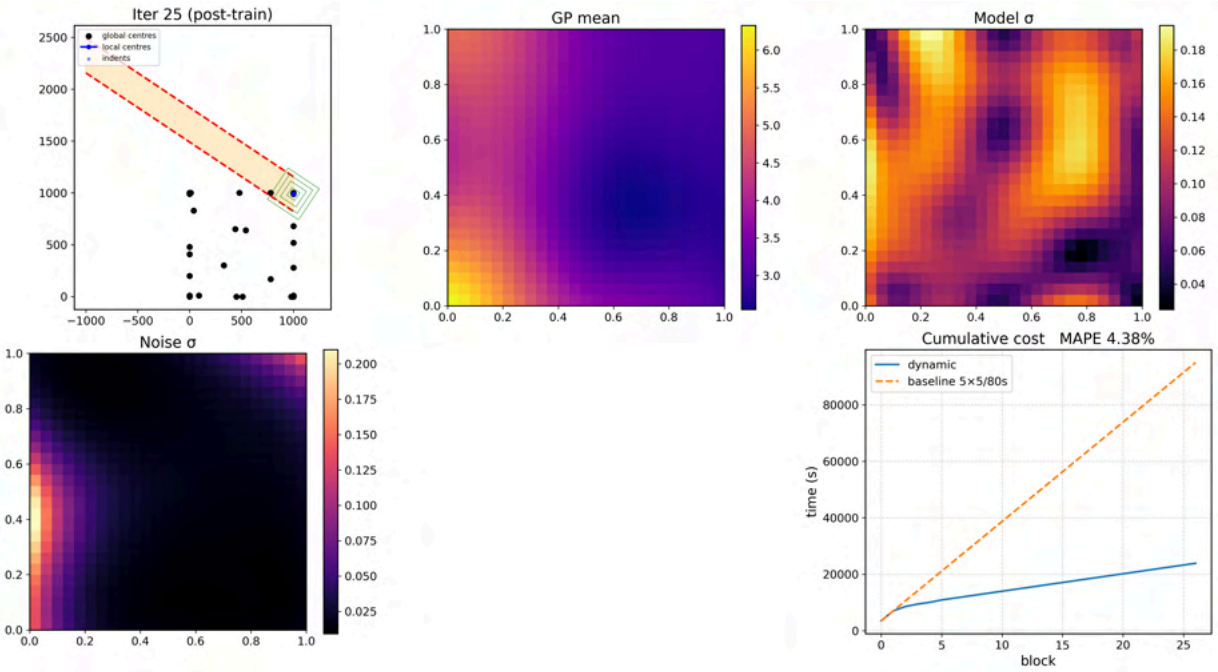

```
===== ITER 23 =====
→ GP fit on 153 points
[Iter 23] Top-4 globals (cx,cy → UE/cost):
   (   0.0,  10.0) → 0.000640
   (   0.0,  20.0) → 0.000638
   (   0.0,  30.0) → 0.000636
   (   0.0,  40.0) → 0.000634
→ Selected global = (0.0,10.0), score = 0.000640
→ block σ=0.4161 → g_curr=2
→ Local centres: [(0.0, 10.0)]
→ cost_vec length 4
    → drilling local centre #0 = (0.0,10.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 22645.49 s
  → After concat: inputs: torch.Size([157, 2])  targets: torch.Size([157])
       gain_loc = -inf, best_global = 0.000309
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=1.1422)
→ baseline padded to 25 steps
```



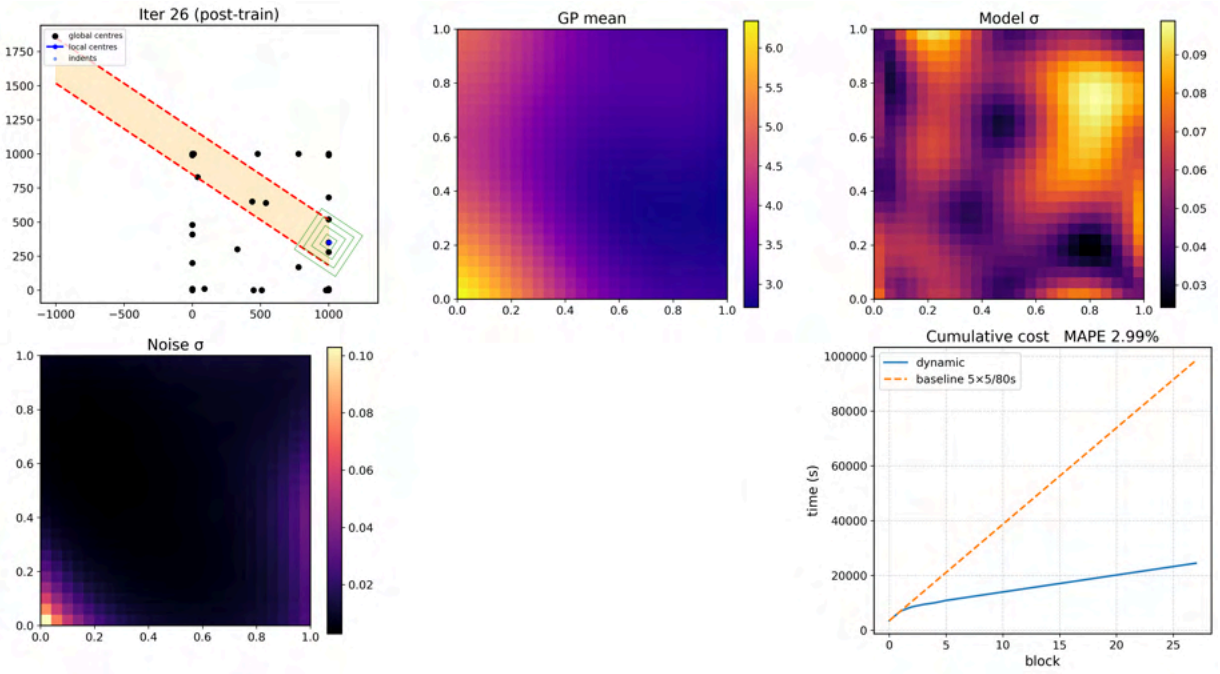

```
===== ITER 24 =====
→ GP fit on 157 points
[Iter 24] Top-4 globals (cx,cy → UE/cost):
   (   0.0, 200.0) → 0.000278
   (   0.0, 210.0) → 0.000277
   (   0.0, 190.0) → 0.000277
   (   0.0, 220.0) → 0.000276
→ Selected global = (0.0,200.0), score = 0.000278
→ block σ=0.1804 → g_curr=2
→ Local centres: [(0.0, 200.0)]
→ cost_vec length 4
    → drilling local centre #0 = (0.0,200.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 23263.68 s
  → After concat: inputs: torch.Size([161, 2])  targets: torch.Size([161])
       gain_loc = -inf, best_global = 0.000203
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=1.0364)
→ baseline padded to 26 steps
```



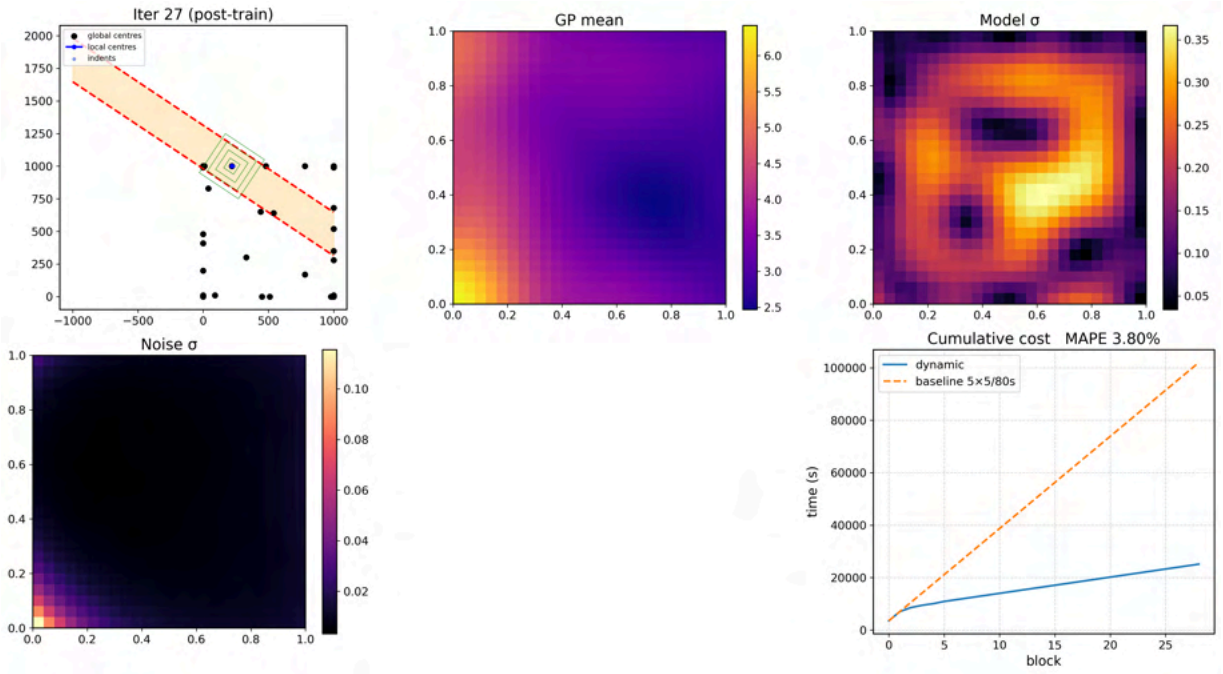

```
===== ITER 25 =====
→ GP fit on 161 points
[Iter 25] Top-4 globals (cx,cy → UE/cost):
   (1000.0, 990.0) → 0.000464
   ( 990.0,1000.0) → 0.000459
   (1000.0, 980.0) → 0.000456
   ( 990.0, 990.0) → 0.000450
→ Selected global = (1000.0,990.0), score = 0.000464
→ block σ=0.3132 → g_curr=2
→ Local centres: [(1000.0, 990.0)]
→ cost_vec length 4
    → drilling local centre #0 = (1000.0,990.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 23881.87 s
  → After concat: inputs: torch.Size([165, 2])  targets: torch.Size([165])
       gain_loc = -inf, best_global = 0.000298
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.6306)
→ baseline padded to 27 steps
```



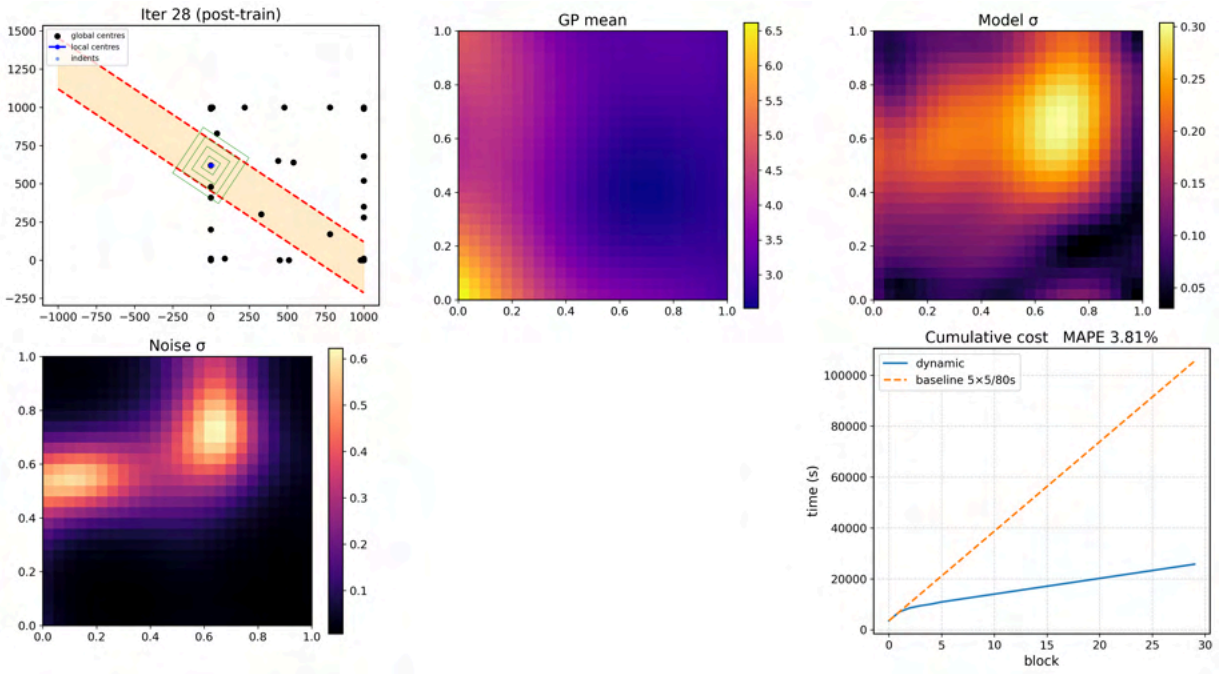

```
===== ITER 26 =====
→ GP fit on 165 points
[Iter 26] Top-4 globals (cx,cy → UE/cost):
   (1000.0, 350.0) → 0.000253
   (1000.0, 340.0) → 0.000253
   (1000.0, 360.0) → 0.000253
   (1000.0, 330.0) → 0.000252
→ Selected global = (1000.0,350.0), score = 0.000253
→ block σ=0.1677 → g_curr=2
→ Local centres: [(1000.0, 350.0)]
→ cost_vec length 4
    → drilling local centre #0 = (1000.0,350.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 24500.06 s
  → After concat: inputs: torch.Size([169, 2])  targets: torch.Size([169])
       gain_loc = -inf, best_global = 0.000151
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.5738)
→ baseline padded to 28 steps
```



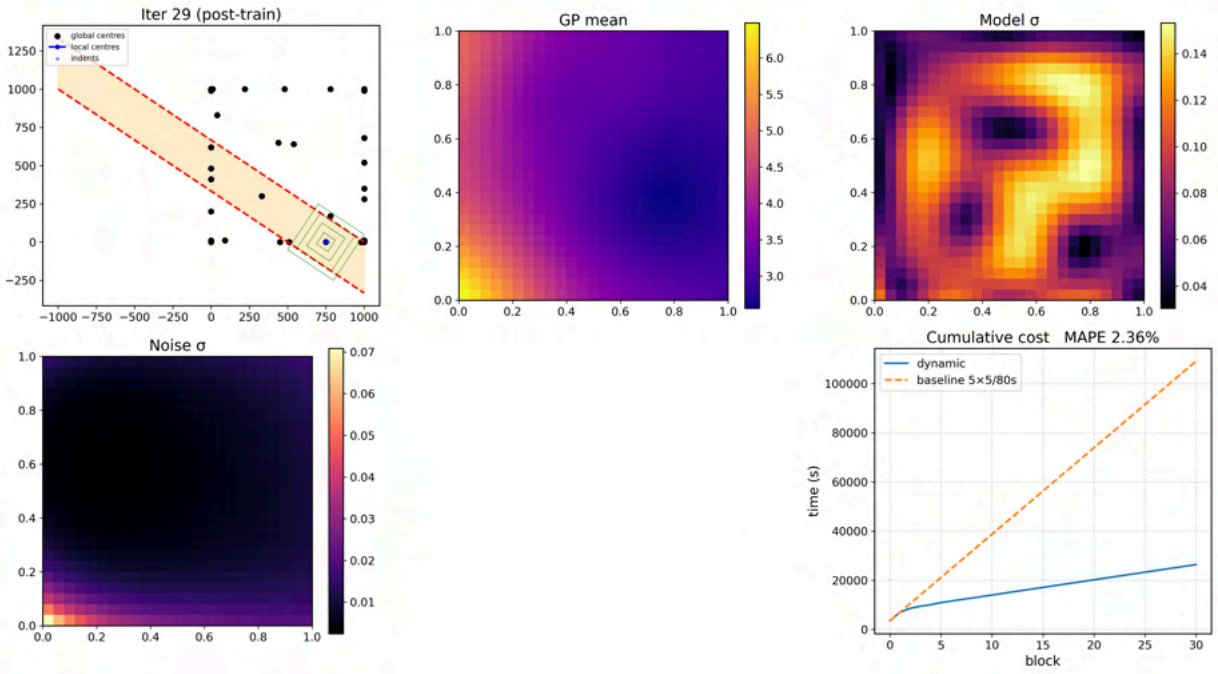

```
===== ITER 27 =====
→ GP fit on 169 points
[Iter 27] Top-4 globals (cx,cy → UE/cost):
   ( 220.0,1000.0) → 0.000734
   ( 210.0,1000.0) → 0.000734
   ( 230.0,1000.0) → 0.000732
   ( 200.0,1000.0) → 0.000730
→ Selected global = (220.0,1000.0), score = 0.000734
→ block σ=0.4897 → g_curr=2
→ Local centres: [(220.0, 1000.0)]
→ cost_vec length 4
    → drilling local centre #0 = (220.0,1000.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 25118.25 s
  → After concat: inputs: torch.Size([173, 2])  targets: torch.Size([173])
       gain_loc = -inf, best_global = 0.000563
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.7951)
→ baseline padded to 29 steps
```



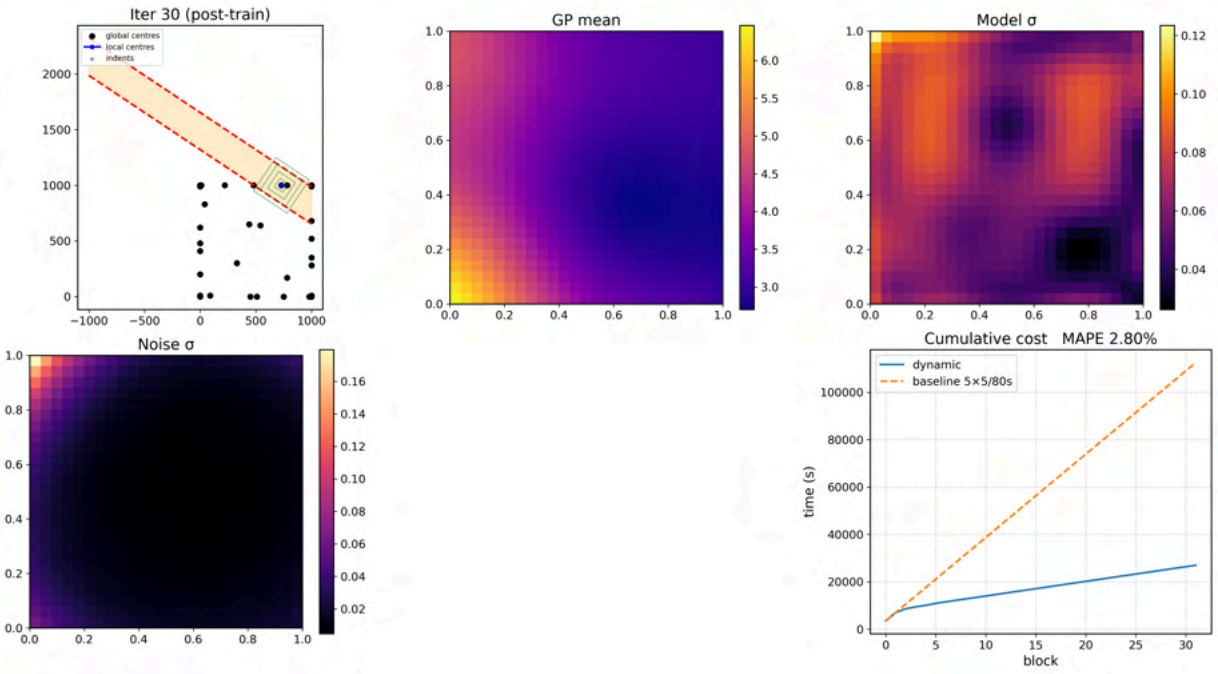

```
===== ITER 28 =====
→ GP fit on 173 points
[Iter 28] Top-4 globals (cx,cy → UE/cost):
   (   0.0, 620.0) → 0.000689
   (   0.0, 610.0) → 0.000689
   (   0.0, 630.0) → 0.000688
   (   0.0, 600.0) → 0.000688
→ Selected global = (0.0,620.0), score = 0.000689
→ block σ=0.4497 → g_curr=2
→ Local centres: [(0.0, 620.0)]
→ cost_vec length 4
    → drilling local centre #0 = (0.0,620.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 25736.44 s
  → After concat: inputs: torch.Size([177, 2])  targets: torch.Size([177])
       gain_loc = -inf, best_global = 0.000463
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.8425)
→ baseline padded to 30 steps
```



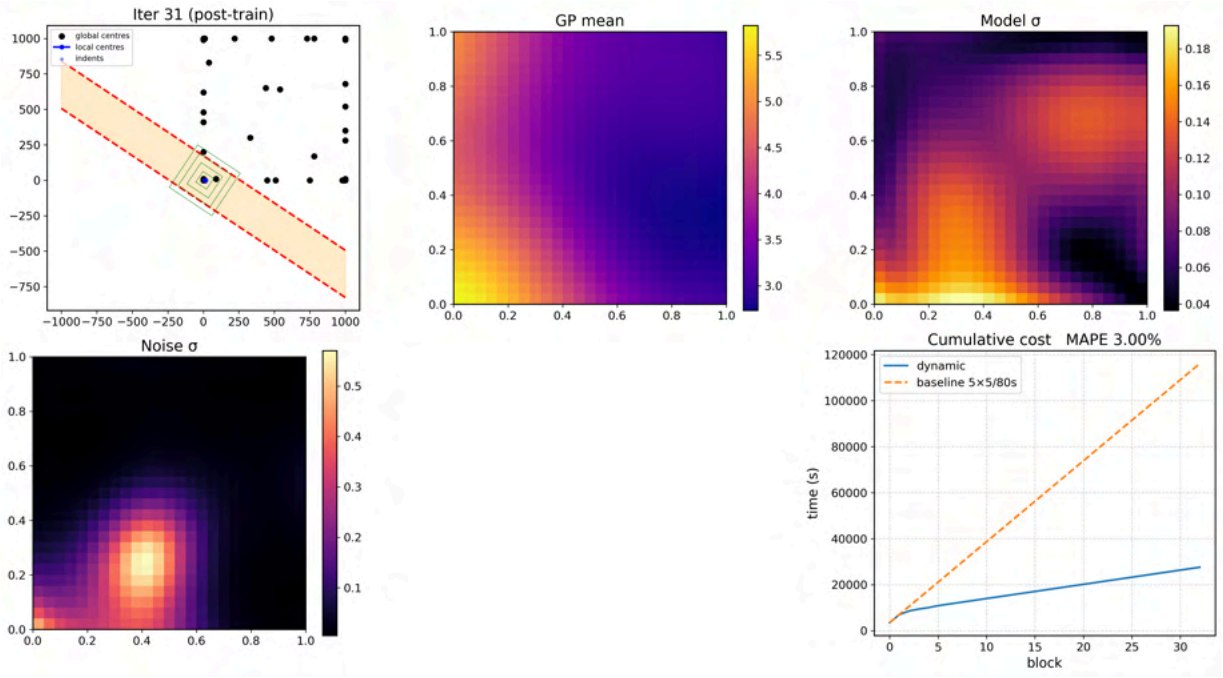

```
===== ITER 29 =====
→ GP fit on 177 points
[Iter 29] Top-4 globals (cx,cy → UE/cost):
   ( 750.0,   0.0) → 0.000278
   ( 760.0,   0.0) → 0.000278
   ( 740.0,   0.0) → 0.000277
   ( 770.0,   0.0) → 0.000277
→ Selected global = (750.0,0.0), score = 0.000278
→ block σ=0.1718 → g_curr=2
→ Local centres: [(750.0, 0.0)]
→ cost_vec length 4
    → drilling local centre #0 = (750.0,0.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 26354.63 s
  → After concat: inputs: torch.Size([181, 2])  targets: torch.Size([181])
       gain_loc = -inf, best_global = 0.000235
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.6712)
→ baseline padded to 31 steps
```



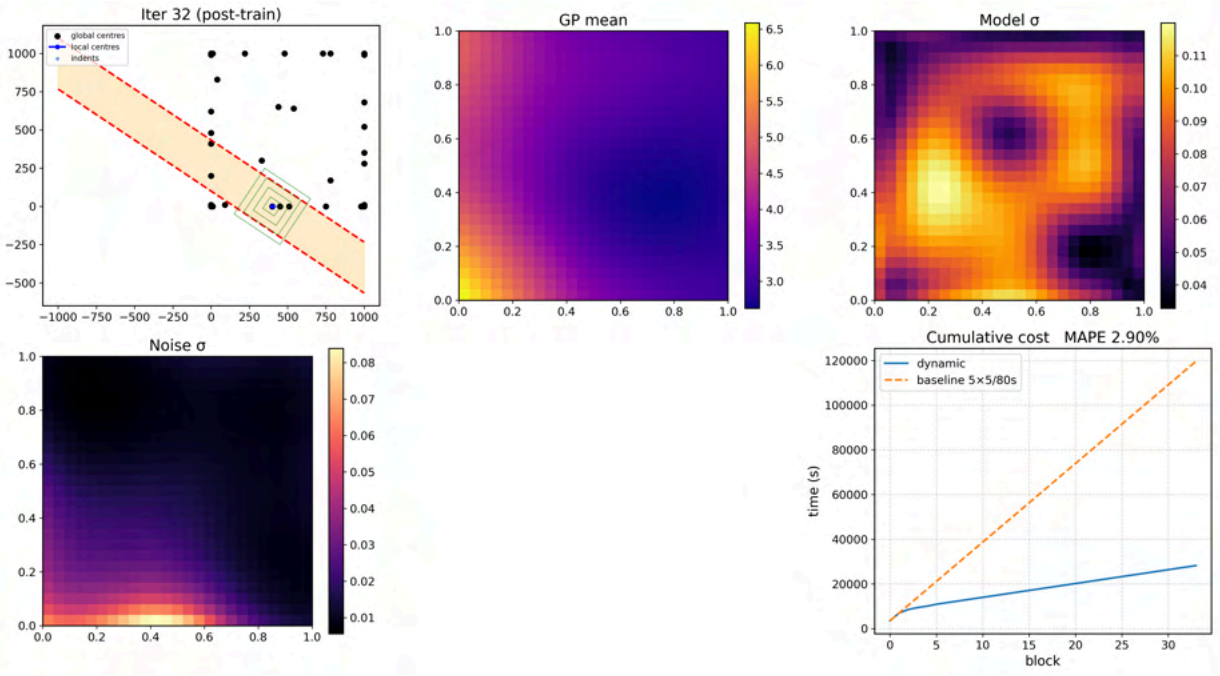

```
===== ITER 30 =====
→ GP fit on 181 points
[Iter 30] Top-4 globals (cx,cy → UE/cost):
   ( 730.0,1000.0) → 0.000392
   ( 720.0,1000.0) → 0.000392
   ( 710.0,1000.0) → 0.000392
   ( 740.0,1000.0) → 0.000392
→ Selected global = (730.0,1000.0), score = 0.000392
→ block σ=0.2631 → g_curr=2
→ Local centres: [(730.0, 1000.0)]
→ cost_vec length 4
    → drilling local centre #0 = (730.0,1000.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 26972.82 s
  → After concat: inputs: torch.Size([185, 2])  targets: torch.Size([185])
       gain_loc = -inf, best_global = 0.000170
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.6566)
→ baseline padded to 32 steps
```



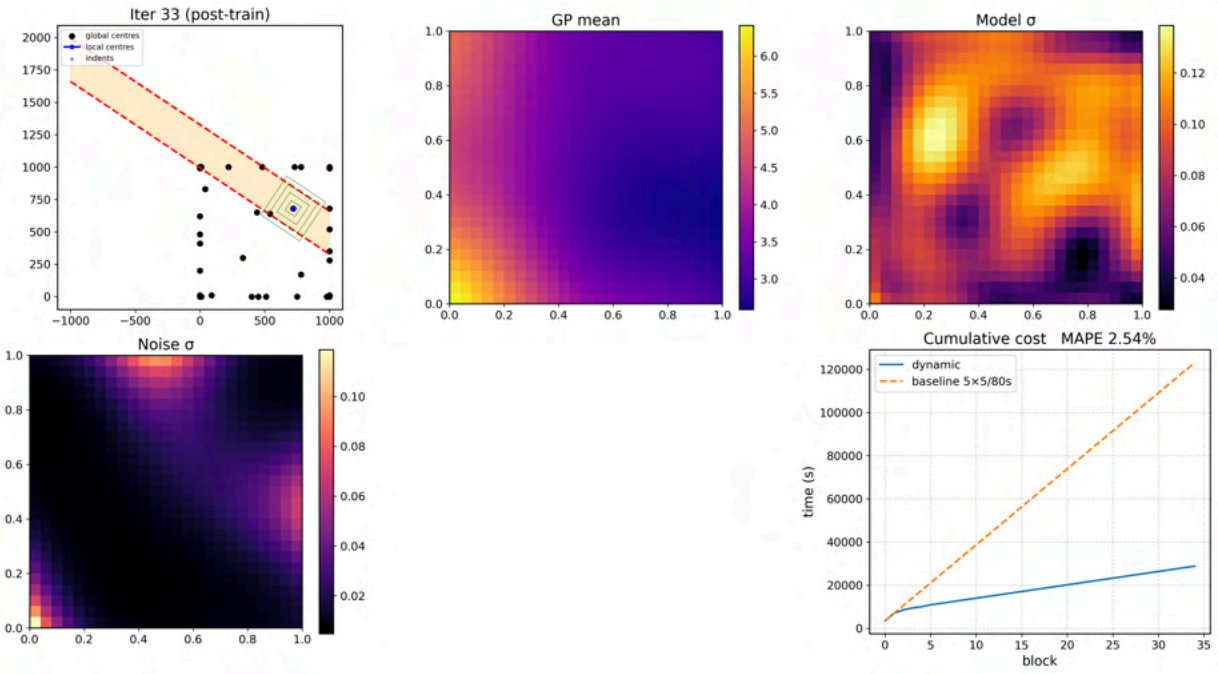

```
===== ITER 31 =====
→ GP fit on 185 points
[Iter 31] Top-4 globals (cx,cy → UE/cost):
   (  10.0,   0.0) → 0.000377
   (  20.0,   0.0) → 0.000374
   (  30.0,   0.0) → 0.000371
   (  40.0,   0.0) → 0.000369
→ Selected global = (10.0,0.0), score = 0.000377
→ block σ=0.2412 → g_curr=2
→ Local centres: [(10.0, 0.0)]
→ cost_vec length 4
    → drilling local centre #0 = (10.0,0.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 27591.02 s
  → After concat: inputs: torch.Size([189, 2])  targets: torch.Size([189])
       gain_loc = -inf, best_global = 0.000294
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=1.1100)
→ baseline padded to 33 steps
```



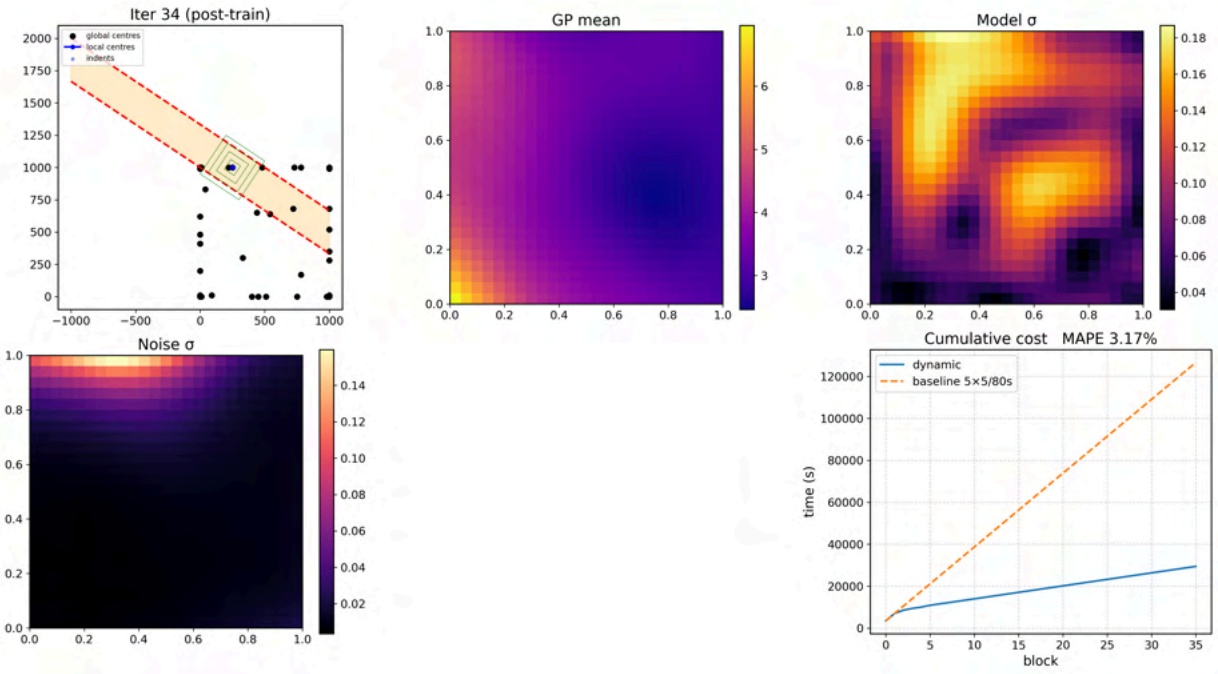

```
===== ITER 32 =====
→ GP fit on 189 points
[Iter 32] Top-4 globals (cx,cy → UE/cost):
   ( 400.0,   0.0) → 0.000301
   ( 410.0,   0.0) → 0.000301
   ( 390.0,   0.0) → 0.000301
   ( 420.0,   0.0) → 0.000300
→ Selected global = (400.0,0.0), score = 0.000301
→ block σ=0.1927 → g_curr=2
→ Local centres: [(400.0, 0.0)]
→ cost_vec length 4
    → drilling local centre #0 = (400.0,0.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 28209.21 s
  → After concat: inputs: torch.Size([193, 2])  targets: torch.Size([193])
       gain_loc = -inf, best_global = 0.000183
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.8018)
→ baseline padded to 34 steps
```



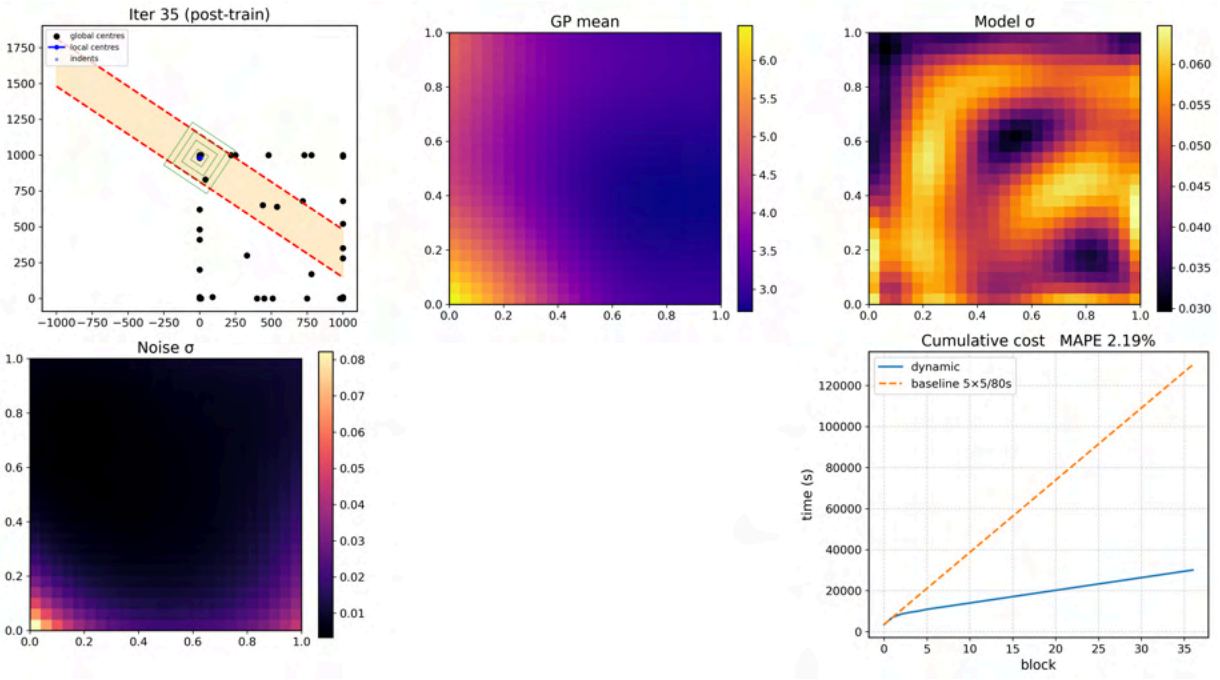

```
===== ITER 33 =====
→ GP fit on 193 points
[Iter 33] Top-4 globals (cx,cy → UE/cost):
   ( 720.0, 680.0) → 0.000341
   ( 730.0, 680.0) → 0.000341
   ( 720.0, 670.0) → 0.000341
   ( 730.0, 690.0) → 0.000341
→ Selected global = (720.0,680.0), score = 0.000341
→ block σ=0.2235 → g_curr=2
→ Local centres: [(720.0, 680.0)]
→ cost_vec length 4
    → drilling local centre #0 = (720.0,680.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 28827.40 s
  → After concat: inputs: torch.Size([197, 2])  targets: torch.Size([197])
       gain_loc = -inf, best_global = 0.000212
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.6314)
→ baseline padded to 35 steps
```



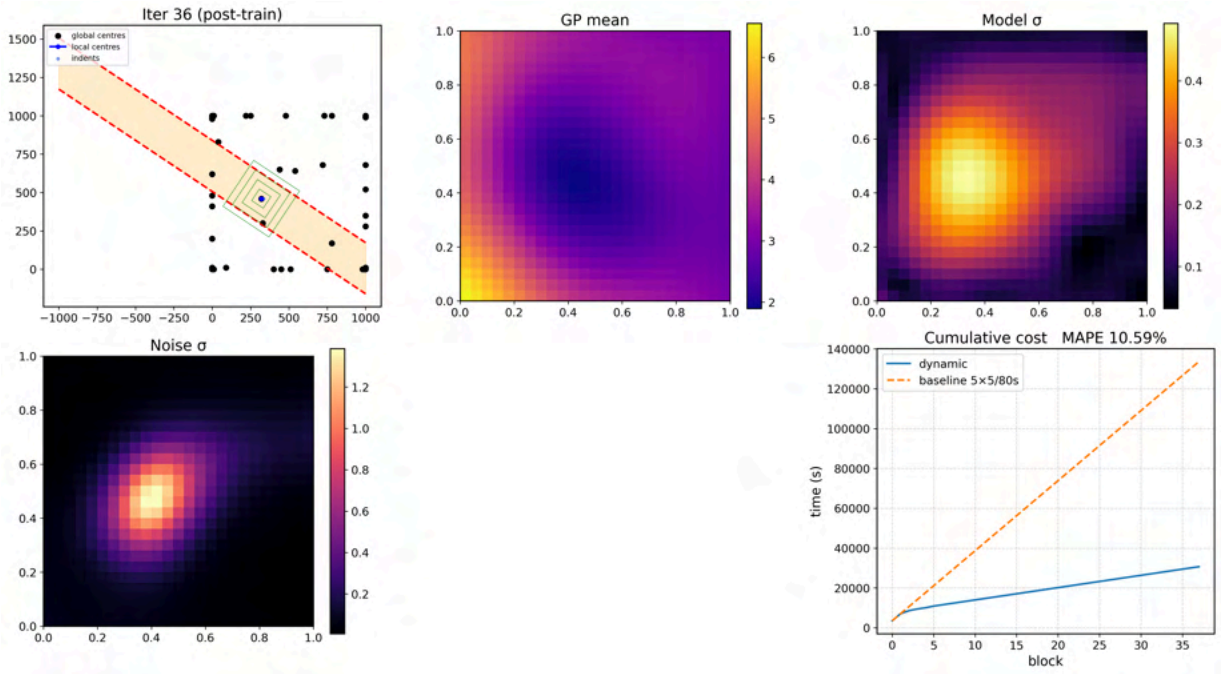

```
===== ITER 34 =====
→ GP fit on 197 points
[Iter 34] Top-4 globals (cx,cy → UE/cost):
   ( 250.0,1000.0) → 0.000481
   ( 240.0,1000.0) → 0.000480
   ( 260.0,1000.0) → 0.000480
   ( 230.0,1000.0) → 0.000480
→ Selected global = (250.0,1000.0), score = 0.000481
→ block σ=0.3240 → g_curr=2
→ Local centres: [(250.0, 1000.0)]
→ cost_vec length 4
    → drilling local centre #0 = (250.0,1000.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 29445.59 s
  → After concat: inputs: torch.Size([201, 2])  targets: torch.Size([201])
       gain_loc = -inf, best_global = 0.000285
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.7394)
→ baseline padded to 36 steps
```



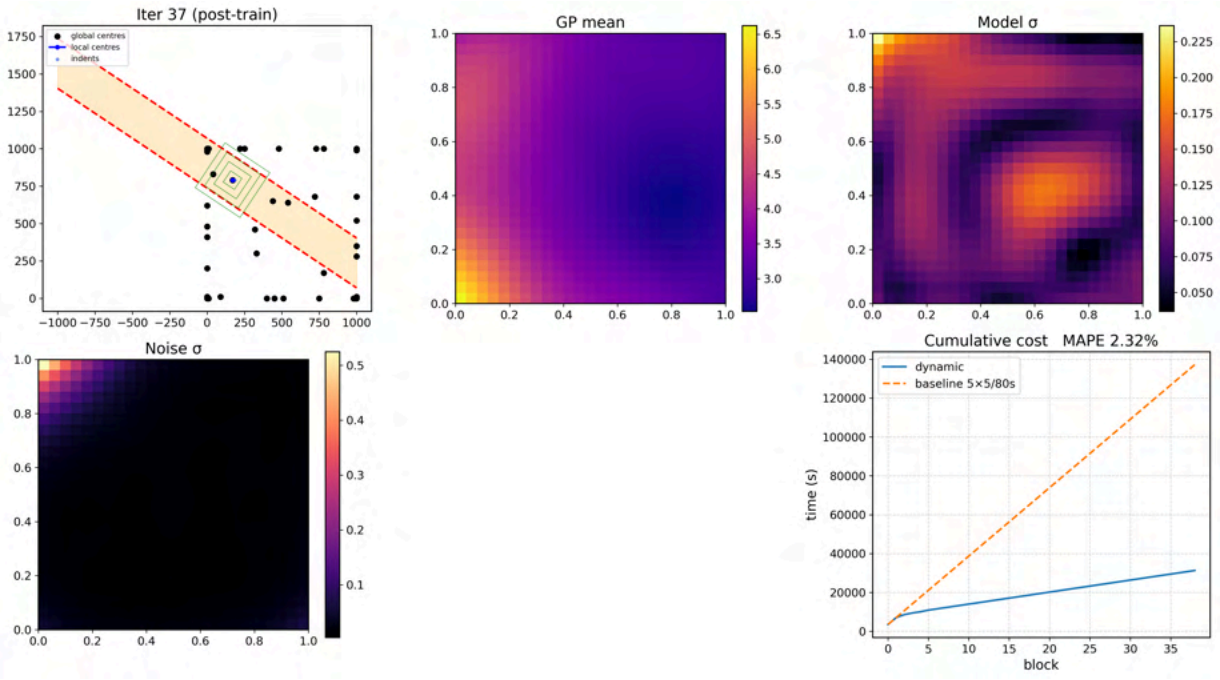

```
===== ITER 35 =====
→ GP fit on 201 points
[Iter 35] Top-4 globals (cx,cy → UE/cost):
   (   0.0, 980.0) → 0.000253
   (  10.0, 990.0) → 0.000251
   (   0.0, 970.0) → 0.000251
   (  20.0,1000.0) → 0.000250
→ Selected global = (0.0,980.0), score = 0.000253
→ block σ=0.1669 → g_curr=2
→ Local centres: [(0.0, 980.0)]
→ cost_vec length 4
    → drilling local centre #0 = (0.0,980.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 30063.78 s
  → After concat: inputs: torch.Size([205, 2])  targets: torch.Size([205])
       gain_loc = -inf, best_global = 0.000101
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.9591)
→ baseline padded to 37 steps
```



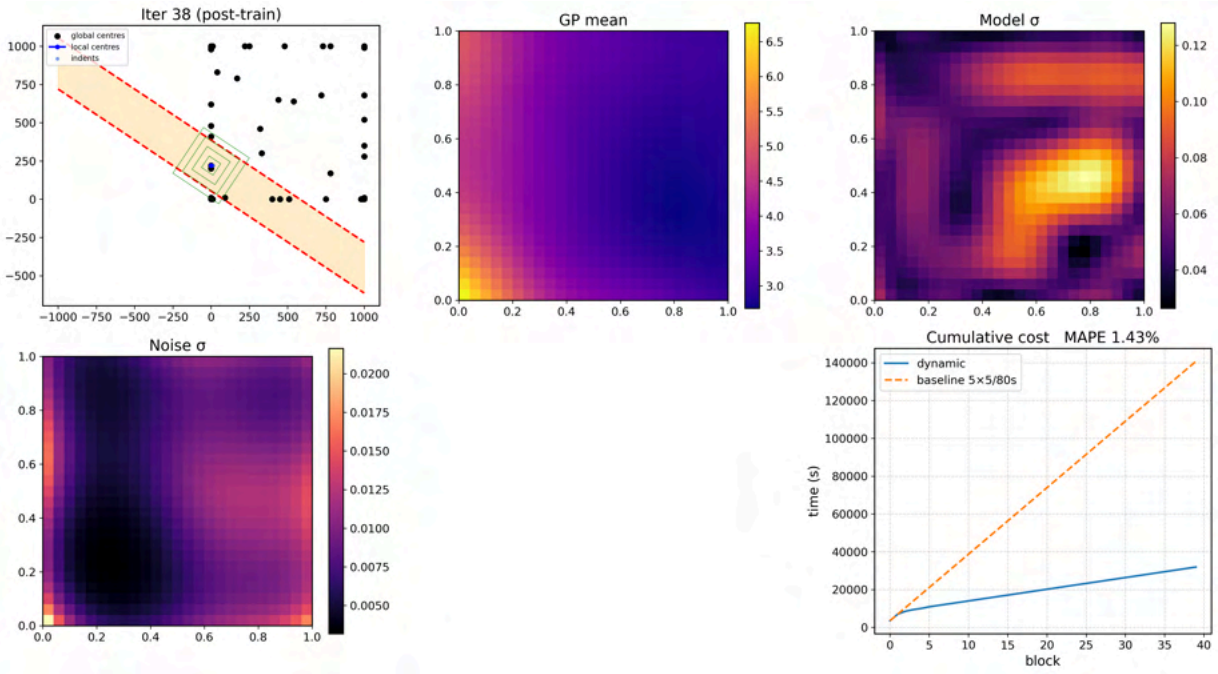

```
===== ITER 36 =====
→ GP fit on 205 points
[Iter 36] Top-4 globals (cx,cy → UE/cost):
   ( 320.0, 460.0) → 0.001115
   ( 320.0, 450.0) → 0.001114
   ( 330.0, 460.0) → 0.001114
   ( 310.0, 460.0) → 0.001114
→ Selected global = (320.0,460.0), score = 0.001115
→ block σ=0.7305 → g_curr=2
→ Local centres: [(320.0, 460.0)]
→ cost_vec length 4
    → drilling local centre #0 = (320.0,460.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 30681.97 s
  → After concat: inputs: torch.Size([209, 2])  targets: torch.Size([209])
       gain_loc = -inf, best_global = 0.000751
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.4638)
→ baseline padded to 38 steps
```



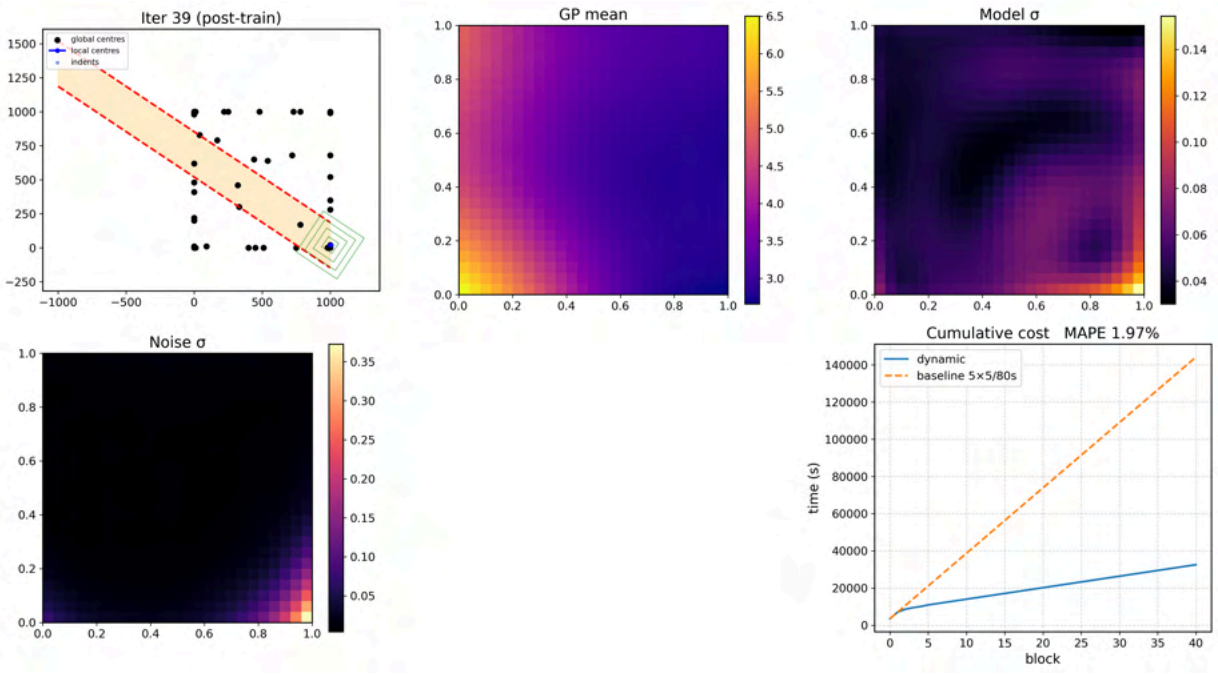

```
===== ITER 37 =====
→ GP fit on 209 points
[Iter 37] Top-4 globals (cx,cy → UE/cost):
   ( 170.0, 790.0) → 0.000343
   ( 170.0, 800.0) → 0.000343
   ( 160.0, 790.0) → 0.000343
   ( 180.0, 800.0) → 0.000343
→ Selected global = (170.0,790.0), score = 0.000343
→ block σ=0.2251 → g_curr=2
→ Local centres: [(170.0, 790.0)]
→ cost_vec length 4
    → drilling local centre #0 = (170.0,790.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 31300.16 s
  → After concat: inputs: torch.Size([213, 2])  targets: torch.Size([213])
       gain_loc = -inf, best_global = 0.000330
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.8480)
→ baseline padded to 39 steps
```



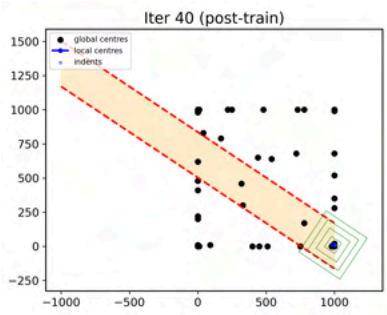

```
===== ITER 38 =====
→ GP fit on 213 points
[Iter 38] Top-4 globals (cx,cy → UE/cost):
   (   0.0, 220.0) → 0.000265
   (   0.0, 230.0) → 0.000265
   (   0.0, 210.0) → 0.000265
   (   0.0, 240.0) → 0.000265
→ Selected global = (0.0,220.0), score = 0.000265
→ block σ=0.1732 → g_curr=2
→ Local centres: [(0.0, 220.0)]
→ cost_vec length 4
    → drilling local centre #0 = (0.0,220.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 31918.36 s
  → After concat: inputs: torch.Size([217, 2])  targets: torch.Size([217])
       gain_loc = -inf, best_global = 0.000195
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.9958)
→ baseline padded to 40 steps
```

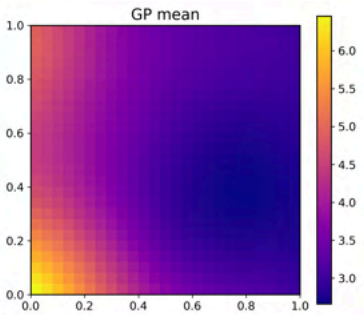


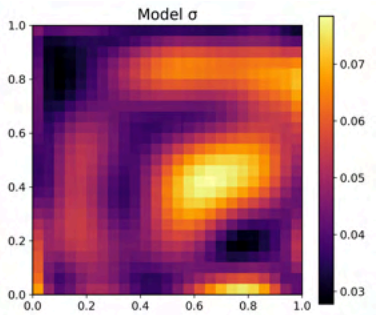

```
===== ITER 39 =====
→ GP fit on 217 points
[Iter 39] Top-4 globals (cx,cy → UE/cost):
   (1000.0,  20.0) → 0.000376
   ( 990.0,  10.0) → 0.000373
   (1000.0,  30.0) → 0.000368
   ( 990.0,  20.0) → 0.000364
→ Selected global = (1000.0,20.0), score = 0.000376
→ block σ=0.2433 → g_curr=2
→ Local centres: [(1000.0, 20.0)]
→ cost_vec length 4
    → drilling local centre #0 = (1000.0,20.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 32536.55 s
  → After concat: inputs: torch.Size([221, 2])  targets: torch.Size([221])
       gain_loc = -inf, best_global = 0.000225
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.5750)
→ baseline padded to 41 steps
```

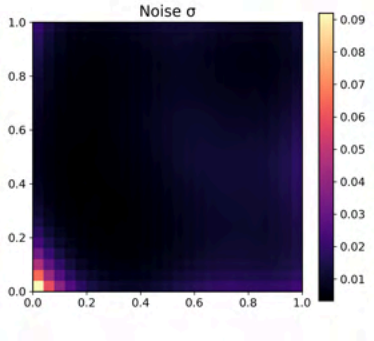


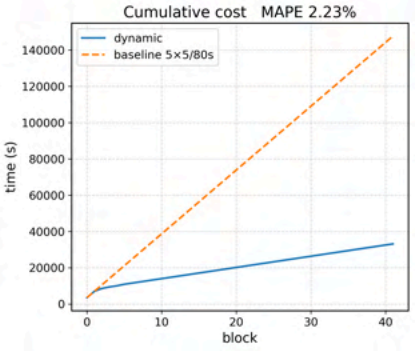

```
===== ITER 40 =====
→ GP fit on 221 points
[Iter 40] Top-4 globals (cx,cy → UE/cost):
   ( 990.0,  10.0) → 0.000160
   (1000.0, 760.0) → 0.000160
   (1000.0, 770.0) → 0.000160
   (1000.0, 750.0) → 0.000160
→ Selected global = (990.0,10.0), score = 0.000160
→ block σ=0.1022 → g_curr=2
→ Local centres: [(990.0, 10.0)]
→ cost_vec length 4
    → drilling local centre #0 = (990.0,10.0)
       Δ-cost = 618.19 s, tot_cost_dyn = 33154.74 s
  → After concat: inputs: torch.Size([225, 2])  targets: torch.Size([225])
       gain_loc = -inf, best_global = 0.000120
       ⇒ GO to next global centre
→ hold=5s  (err_5=0.0000, T_big=0.6242)
→ baseline padded to 42 steps
```


Iter 40 (post-train)
global centres
local centres
indents


GP mean


Model σ


Noise σ


Cumulative cost MAPE 2.23%
dynamic
baseline 5×5/80s
time (s)
block