ME-438: Individual Project

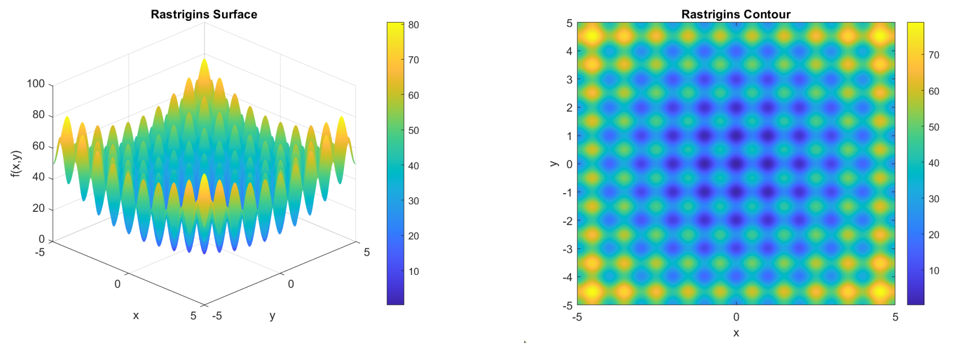
Department of Mechanical Engineering

École Polytechnique Fédérale de Lausanne

Name:	Yo-Shiun Cheng
Student Number:	386249

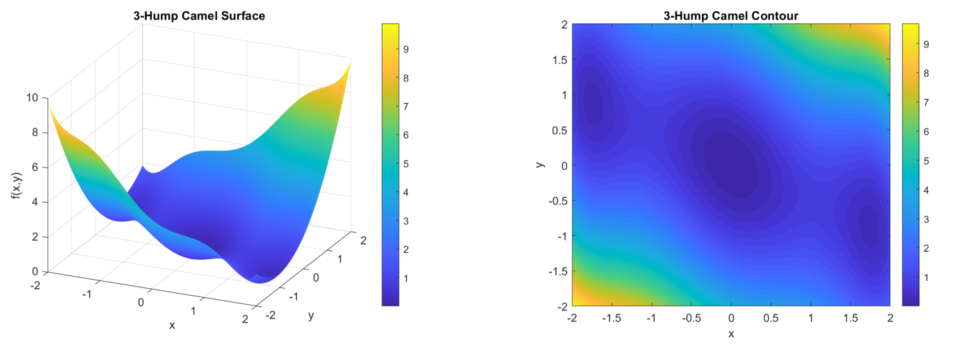
1 Black-box Optimization

In this study, we compare Genetic Algorithm (GA) and Bayesian Optimization (BO) on two 2-D test functions: the *Rastrigin function* (see Figure 1) and the smoother *three-hump camel function* (see Figure 2). For GA, we sweep the mutation rate $\mu \in \{0.01, 0.05, 0.10, 0.20, 0.30\}$ (population size fixed at 50); for BO, we vary the exploration ratio $\kappa \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$ (six seed points, maximum 30 evaluations). Each configuration is run for 10 independent trials, and we report mean and standard deviation over trials to illustrate how optimal GA and BO settings change between rugged (Rastrigin) and smooth (3-hump camel) landscapes.



(a) Surface (b) Contour

Figure 1: Surface and contour of the Rastrigins function.



(a) Surface (b) Contour

Figure 2: Closer look at the surface and contour of the three-hump camel function.

1.1 Benchmark Functions

Rastrigin

$$f_{\text{Rastrigin}}(x_1, x_2) \;=\; 20 \;+\; \sum_{i=1}^{2} \big[x_i^2 - 10\cos(2\pi x_i)\big], \quad x_i \in [-5, 5].$$

Three-Hump Camel

$$f_{\text{Camel}}(x_1, x_2) \;=\; 2x_1^2 - 1.05x_1^4 + \frac{x_1^6}{6} + x_1 x_2 + x_2^2, \quad x_i \in [-5, 5].$$

1.2 Parameter Sweeps and Protocol

For each algorithm–function pair, we sweep the key hyperparameter (μ for GA, κ for BO) over its respective grid. At each setting:

1. Run $N = 10$ independent trials with different random seeds.
2. Record:
   - Best objective value $f_{\min}$ after all evaluations.
   - Convergence time t.
3. Compute $\overline{f}_{\min} \pm \sigma_f$ and $\bar{t}_{\text{conv}}$ over 10 trials.

1.3 Results

Bayesian Optimization Figures 3a and 3b show the BO exploration-ratio sweep on the Rastrigin and 3-Hump Camel functions, respectively. In each plot, the left (blue) axis reports the *mean best objective value* after 30 function evaluations (with vertical bars indicating one standard deviation across 10 trials), while the right (orange) axis shows the *mean elapsed time* (in seconds).

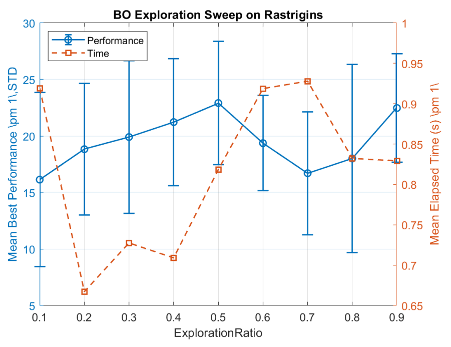
Rastrigin. As κ increases from 0.1 to 0.5, the mean best-found Rastrigin value rises from approximately 16 to 22 (worsening performance), indicating that BO queries become overly exploratory and neglect exploitation of promising regions. Beyond $\kappa = 0.5$, performance improves slightly, reaching a local minimum around $\kappa = 0.6$–0.7 (mean best value ≈ 18), before degrading again at $\kappa = 0.9$. In parallel, elapsed time is lowest at $\kappa = 0.2$ (≈ 0.65s) and increases to about 0.95s at $\kappa = 0.7$, reflecting the extra overhead of wider exploration in the GP acquisition step.

3-Hump Camel. The three-hump camel sweep shows less variability across trials (smaller STD bars). The best mean-fitness occurs at $\kappa = 0.6$ (≈ 1). Low exploration ($\kappa = 0.1$) sometimes converges rapidly (mean elapsed time ≈ 0.65s) but risks getting stuck in a local hump, leading to a higher mean (≈ 2.0). At $\kappa = 0.6$, performance grows (mean ≈ 1.0) and time drops to ≈ 0.70s, suggesting BO occasionally succeed to balance exploration and exploitation in this mid-range.

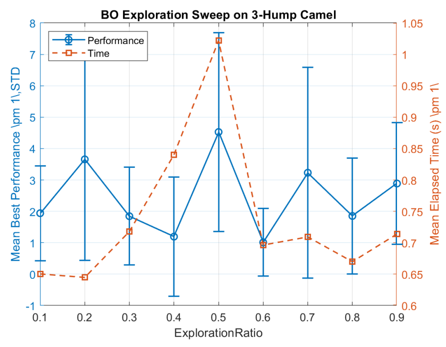
Genetic Algorithm Figures 4a and 4b display the GA mutation-rate sweep.

Rastrigin. As the mutation rate μ increases from 0.01 to 0.10, the mean best Rastrigin value drops sharply from ≈ 3.7 to ≈ 1.1, indicating that small increases in mutation help the GA escape local traps. Beyond $\mu = 0.10$, performance plateaus: $\mu = 0.20$ yields ≈ 1.2, and $\mu = 0.30$ further reduces the mean to ≈ 0.3 (closest to the global minimum of 0). However, elapsed time rises from $\approx$ 7s at $\mu = 0.01$ to ≈ 13.5s at $\mu = 0.10$, reflecting that higher mutation slows convergence by disrupting exploitation. At $\mu = 0.30$, time is ≈ 12.9s, suggesting a slight efficiency gain once mutation is large enough to find the basin quickly.

3-Hump Camel. On the smoother camel function, even a small mutation rate ($\mu = 0.01$) yields mean best fitness ≈ 0.29, but increasing μ to 0.05 improves the mean to ≈ 0.17. Further increases ($\mu = 0.10, 0.20$) produce modest gains (≈ 0.13–0.12), while $\mu = 0.30$ achieves the lowest mean ≈ 0.07. The trend is more gradual than on Rastrigin, reflecting the simpler landscape. Elapsed time climbs



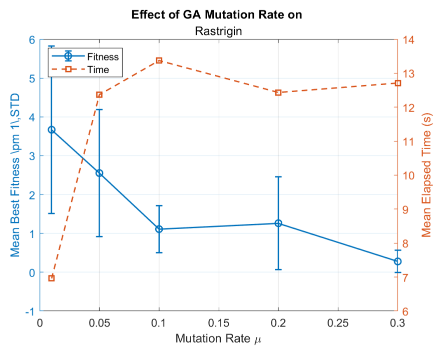
(a) Rastrigins



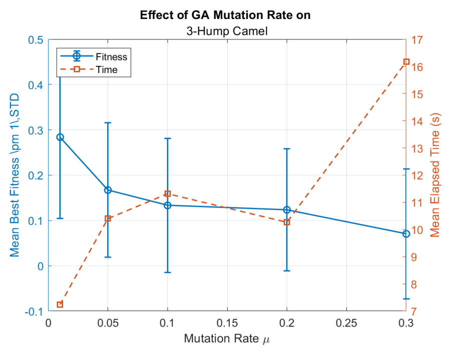
(b) Three-hump Camel

Figure 3: BO Exploration Ratio Sweep on Rastrigins and Three-hump Camel Functions.

from ≈ 7.2s at $\mu = 0.01$ up to ≈ 16.2s at $\mu = 0.30$, indicating that larger mutation delays convergence even when the valley is smooth.



(a) Restrigins



(b) Three-hump Camel

Figure 4: GA Mutation Rate Sweep on Rastrigins and Three-hump Camel Functions.

1.4 Discussion

GA vs. BO: Sample Efficiency and Robustness

- On the **smooth 3-Hump Camel**, BO (with $\kappa = 0.50$) is *more sample-efficient* than GA: BO often finds a value near zero within 30 evaluations, whereas GA requires way more evaluations. As a result, BO's runtime (≈ 0.65s) is significantly less than GA's runtime ($\approx$ 7–12s). However, BO's performance variance is large (STD up to ≈ 3.5), while GA's outcomes at $\mu = 0.05$–0.10 are more consistent (STD ≈ 0.28–0.32), making GA more *robust* against unlucky initial seeds in this case.

- On the **rugged Rastrigin**, neither method fully escapes all local minima at low hyperparameter values. GA with high mutation ($\mu = 0.30$) yields near-zero mean ($\approx 0.30 \pm 0.44$) but at large time cost (≈ 12.9s). BO's best setting ($\kappa = 0.10$) attains mean $\approx 16 \pm 5$ in only ≈ 0.94s, but performance is much worse than GA's. Thus, GA is *more robust and effective* at thoroughly exploring a rugged landscape, while BO trades off accuracy for speed.

2 PCA on Data-set

In this section, we apply PCA to the classic Iris dataset (150 samples, four measurements) to investigate whether a lower-dimensional representation captures most of the variability and separates the three species. First, we standardize each feature and compute the principal components via eigen value decomposition. We then visualize the data projected onto the first two principal components (PCs), color-coded by species, and plot the percentage of variance explained by each PC.

2.1 Data and Preprocessing.

We load MATLAB's built-in `fisheriris` dataset (150 samples, four features: sepal length, sepal width, petal length, petal width) and standardize each feature to zero mean and unit variance.

2.2 PCA Computation.

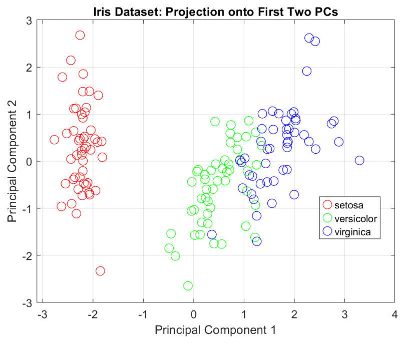
We apply MATLAB's built-in `pca` function to the 150×4 standardized matrix to obtain:

`coeff`: a 4×4 matrix whose columns are the principal directions (eigenvectors).

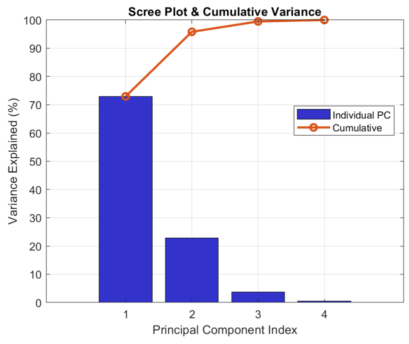
`score`: a 150×4 matrix of projected coordinates in PC space.

`explained`: a 4×1 vector giving the percentage of variance explained by each PC.

We then plot the first two PCs (PC1 vs. PC2), coloring each point by its species label (Figure 5a). Finally, we generate a "scree plot" of individual `explained` percentages and a cumulative-variance curve to decide how many PCs suffice (Figure 5b).



(a)



(b)

Figure 5: Projection of the First Two Principle Components and the Scree Plot.

2.3 Effectiveness of PCA.

PCA reduces the original four-dimensional Iris measurements to a two-dimensional representation while retaining over 95% of the total variance, as shown in the scree plot (Figure 5b). In the PC1–PC2 scatter (Figure 5a), Setosa forms a distinct cluster along PC1, reflecting its markedly different petal dimensions, whereas Versicolor and Virginica overlap somewhat but are still more distinguishable than in the raw feature space. Because PC1 + PC2 capture roughly 96% of the variance, two components suffice for accurate low-dimensional embeddings, providing both efficient reconstruction and clear class separation. If one requires more variance retention, a third principal component can be included (cumulative $\approx$ 97.5%), but the gain is small.

3 Perform K-means and Identify the Optimal Number of Clusters

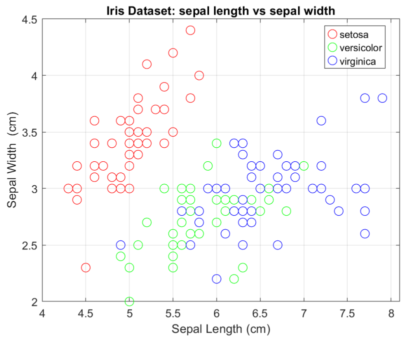
In this section, we apply k-means clustering to the Iris dataset using only sepal length and sepal width (the first two features). Our goal is to identify the natural grouping of the 150 flowers and compare it to the known species labels. We use the average silhouette score to determine the optimal number of clusters k, then plot the resulting clusters and centroids in the sepal-feature plane. Finally, we map each cluster to the majority species within it and compute clustering accuracy against the ground truth.

3.1 Methods.

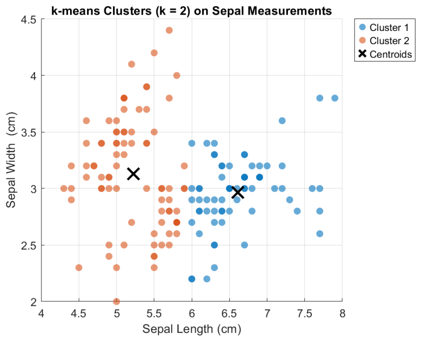
We extract only sepal length and sepal width from `fisheriris` (150×2 matrix). To select the optimal number of clusters k, we run k-means (Euclidean distance, 10 replicates) for $k = 2$ through 6, computing the average silhouette score for each. The k that maximizes the silhouette score is chosen. Finally, we re-run k-means with this optimal k, plot the 2-D clusters and centroids, and assign each cluster to its majority species to compute clustering accuracy against the true labels.

3.2 Results.

Figure 7a shows average silhouette scores versus k; the maximum occurs at $k = 2$. Figure 6b displays the final 2-cluster partition: each point is colored by cluster, with black '×' markers denoting centroids. Figure 7b reports how clusters align with {*setosa, versicolor, virginica*}, yielding an overall accuracy of only 62%.

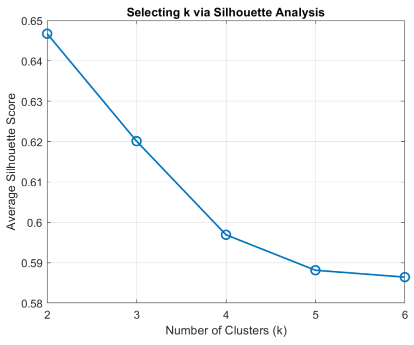


(a)

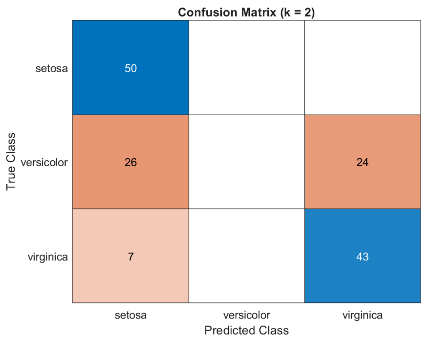


(b)

Figure 6: Comparison of the ground truth clustering (left) and the k-means clustering with optimal k (right).



(a) Average silhouette score of k from 2 to 6.


Confusion Matrix (k = 2)
True Class
Predicted Class

	setosa	versicolor	virginica
setosa	50		
versicolor	26		24
virginica	7		43

(b) Confusion matrix of $k = 2$

Figure 7

4 Review of Data-Driven Design Papers

RoboGrammar: Graph Grammar for Terrain-Optimized Robot Design

Summary Zhao et al. (2020) present *RoboGrammar*, a pipeline that automatically co-designs robot morphology and control for a given terrain by representing each candidate as a graph generated from a recursive grammar—ensuring designs are both buildable and symmetric. Structural rules add body segments and mirrored limbs, while component rules assign actual link and joint types with their limits. Each generated design is evaluated in simulation using a sampling-based MPC (MPPI) to find a viable gait. To explore the large design space efficiently, they introduce *Graph Heuristic Search* (GHS), where a Graph Neural Network (GNN) learns online to estimate the best achievable reward from any partial graph, guiding the search and pruning unlikely branches. On flat, low-friction, ridged, and gapped terrains, RoboGrammar discovers novel, high-performing morphologies—such as long-legged walkers or adaptive bipeds—that outperform Monte Carlo Tree Search and random sampling while using far fewer expensive simulations. This demonstrates that a compact grammar combined with a learned GNN heuristic can efficiently yield robust, terrain-optimized robot designs.

Comments They chose a grammar-based representation because it handles the enormous number of possible robot shapes while ensuring every design can actually be built. By using bio-inspired rules they avoid wasting time on absurd or impossible morphologies. Instead of sampling designs at random or using vanilla Monte Carlo, they introduced Graph Heuristic Search (GHS). GHS trains a graph neural network to predict how well any partial design might perform, which lets the search focus on good candidates and drastically cuts down on costly MPC simulations. In their tests across various terrains, GHS reached high-reward designs in about 2,000 evaluations, compared to roughly 5,000 for both MCTS and random search. That said, their current grammar still assumes symmetrical limbs and a fixed library of joint and link types; it can't yet handle continuous parameters or asymmetrical shapes . Also, while MPPI works well for finding stable gaits, more advanced controllers—such as reinforcement learning agents—might produce faster or better behaviors. Finally, all of their results remain in simulation, so building and testing these designs in the real world is a key next step.

Improving Robotic Cooking Using Batch Bayesian Optimization

Summary Junge et al. (2020) propose a practical method for tuning an automated omelette-making robot based on real human taste tests. They use a UR5 arm fitted with a custom two-fingered gripper and standard kitchen tools, and define simple parameters—salt, pepper, whisking time, and pan heating time—to control the cooking process. Human participants score each omelette on a 0–10 scale for flavor, appearance, and texture. Because taste testing is both costly and subjective, they pit traditional Sequential Bayesian Optimization against a "Batch" approach. In Batch BO, ten recipes are chosen up front, participants taste and can re-evaluate all ten samples, and only then is a Gaussian Process model fitted to predict the single best combination. Across eight tasters, this batch strategy consistently outperforms the sequential method—delivering larger quality gains from the same number of trials—by reducing cognitive bias (since tasters compare samples directly) and ensuring broader exploration of the recipe space. In short, Batch BO lets the robot learn a superior omelette recipe in as few as ten trials.

Comments The authors chose Bayesian Optimization because each omelette trial takes significant time and ingredient costs, and human taste evaluations are constrained by appetite and memory. By using Batch BO, they select all sampling points in advance—ensuring greater variation in the recipe parameters and allowing tasters to re-evaluate earlier samples—which helps reduce noise in the quality function and addresses grounding biases. In their experiments, Sequential BO often converges prematurely under low exploration (κ) and fails to adequately explore the control space, resulting in noisy and unreliable improvements. In contrast, Batch BO's uniform sampling produces a broader dataset and a more accurate Gaussian Process model of taste preferences. However, Batch BO can miss narrowly optimal spice ratios if the initial grid is not sufficiently dense; one improvement would be to follow a coarse batch with focused sequential refinements. Additionally, the current pipeline only measures flavor, appearance, and texture on integer scales—adding continuous aroma metrics (e.g., via olfactory sensors) or computer vision for color and texture could enhance model fidelity. Finally, because salt, pepper, and whisking ranges are fixed, a truly personalized recipe might require adaptive parameter bounds or a hierarchical, multi-stage optimization (e.g., coarse spice levels followed by fine-tuned adjustments) when dealing with more complex dishes.

References

Junge, K., Hughes, J., Thuruthel, T. G., & Iida, F. 2020, IEEE Robotics and Automation Letters, 5, 760

Zhao, A., Xu, J., Konaković-Luković, M., et al. 2020, ACM Transactions on Graphics (TOG), 39, 1

A Bayesian Optimization Exploration Rate Sweep Code

```matlab
clc; clear; close all;

%% Settings
X1 = optimizableVariable('x',[-5 5]);
X2 = optimizableVariable('y',[-5 5]);
vars = [X1,X2];
explorationRatios = 0.1 : 0.1 : 0.9;
nExperiments = 10;
objFuncs = {@rastriginsfcn, @threeHumpCamel};
funcNames = {'Rastrigins', '3-Hump Camel'};

%% parallel pool
if isempty(gcp('nocreate'))
    parpool('AttachedFiles','bo_sweep.m');
end

%% Loop over each objFuncs
for fIdx = 1 : numel(objFuncs)
    objFunc = objFuncs{fIdx};
    fname = funcNames{fIdx};

    % plot the function surface and contour
    [xg, yg] = meshgrid(linspace(-5,5,200), linspace(-5,5,200));
    Z = arrayfun(@(xx,yy) objFunc(struct('x',xx,'y',yy)), xg, yg);

    figure('Name', [fname ' Surface']);
    surf(xg, yg, Z, 'EdgeColor','none'); shading interp;
    title([fname ' Surface']); xlabel('x'); ylabel('y'); zlabel('f(x,y)');
    view(45,30); colorbar;

    figure('Name', [fname ' Contour']);
    contourf(xg, yg, Z, 50, 'LineColor','none');
    title([fname ' Contour']); xlabel('x'); ylabel('y'); colorbar;

    % results array
    avgTime = zeros(size(explorationRatios));
    avgPerf = zeros(size(explorationRatios));

    % sweep exploration ratio
    for idx = 1 : numel(explorationRatios)
        k = explorationRatios(idx);
        timevec = zeros(1, nExperiments);
        perfvec = zeros(1, nExperiments);

        for j = 1 : nExperiments

            % Perform Bayesian Optimization
            results = bayesopt(objFunc, vars, ...
                'AcquisitionFunctionName','expected-improvement-plus', ...
                'ExplorationRatio',k, ...
                'NumSeedPoints',6, ...
                'UseParallel', true, ...
                'PlotFcn',{});  % @plotAcquisitionFunction,
                    @plotObjectiveModel,@plotMinObjective
            timevec(j) = results.TotalElapsedTime;
            perfvec(j) = results.MinEstimatedObjective;
        end

        avgTime(idx) = mean(timevec);
        avgPerf(idx) = mean(perfvec);
        perfStd(idx) = std(perfvec);
    end

    % save and display results
    T = table(explorationRatios', avgTime', avgPerf', perfStd', ...
        'VariableNames', {'ExplorationRatio', 'AvgTime', 'AvgPerformance', '
            StdPerformance'});
    filename = sprintf('bo_avg_results_%s.csv',fname);
    writetable(T, filename);
    fprintf('Results saved to %s\n', filename);
    disp(T);
end

%% Plot aggregated BO results per function (with  STD )
for fIdx = 1:numel(objFuncs)
    fname = funcNames{fIdx};
    % Read in the CSV you just wrote
    T = readtable(sprintf('bo_avg_results_%s.csv', fname));

    figure('Name', fname, 'NumberTitle','off');
    % Left axis: performance  1  STD
    yyaxis left
    errorbar(T.ExplorationRatio, T.AvgPerformance, T.StdPerformance, ...
        '-o','LineWidth',1.2, 'CapSize',10);
    ylabel('Mean Best Performance \pm 1\,STD');
    xlabel('ExplorationRatio');

    % Right axis: timing  1  STD (optional errorbars)
    yyaxis right
    plot(T.ExplorationRatio, T.AvgTime, ...
        '--s','LineWidth',1.2);
    ylabel('Mean Elapsed Time (s) \pm 1\');

    title(sprintf('BO Exploration Sweep on %s', fname));
    grid on;
    legend('Performance','Time','Location','northwest');
end


%% plot the function surface and contour
[xg, yg] = meshgrid(linspace(-2,2,200), linspace(-2,2,200));
Z = arrayfun(@(xx,yy) threeHumpCamel(struct('x',xx,'y',yy)), xg, yg);

figure('Name', [fname ' Surface']);
surf(xg, yg, Z, 'EdgeColor','none'); shading interp;
title(['3-Hump Camel Surface']); xlabel('x'); ylabel('y'); zlabel('f(x,y)');
view(45,30); colorbar;

figure('Name', [fname ' Contour']);
contourf(xg, yg, Z, 50, 'LineColor','none');
title(['3-Hump Camel Contour']); xlabel('x'); ylabel('y'); colorbar;

% objective functions
function f = rastriginsfcn(X)
    xx = [X.x, X.y];
    f = 10*numel(xx) + sum(xx.^2 - 10*cos(2*pi*xx));
```

```
end

function f = threeHumpCamel(X)   % 3-hump camel function
    x = X.x;
    y = X.y;
    f  = (2*x^2 - 1.05*x^4 + (x^6)/6 + x*y + y^2);
end
```

B Genetic Algorithm Mutaiton Rate Sweep Code

```
clc; clear; close all;

% setting
nvar = 2;

nTrials = 10;   % repeats per rate
popSize = 50;
mutationRates = [0.01, 0.05, 0.1, 0.2, 0.3];

objFuncs = {@rastriginsfcn, @camel3humpfcn};
funcNames = {'Rastrigin', '3-Hump Camel'};
nFunc = numel(objFuncs);

% storage
results = table('Size', [nFunc * numel(mutationRates), 6], ...
    'VariableTypes', {'string', 'double', 'double', 'double', 'double', '
       double'}, ...
    'VariableNames', {'Function', 'MutationRate', 'MeanFval', 'StdFval', '
       MeanTime', 'MeanGens'});
row = 1;

% parallel pool
if isempty(gcp('nocreate'))
    parpool('AttachedFiles','ga_sweep.m');
end


% sweep over objective functions
for fIdx = 1 : nFunc
    func = objFuncs{fIdx};
    fname = funcNames{fIdx};
    fprintf('Processing function: %s\n', fname);

    % loop over mutation rates
    for mu = mutationRates
        fprintf('Testing mutation rate    = %.3f\n', mu);
        fvals  = zeros(nTrials, 1);
        times  = zeros(nTrials, 1);
        gens  = zeros(nTrials, 1);

        for t = 1 : nTrials
            rng(t);
            options = optimoptions('ga', ...
                'UseParallel',true, ...
                'PlotFcn',{'gaplotbestf','gaplotdistance','gaplotstopping'},
                    ...
                'PopulationSize', popSize, ...
                'MutationFcn', {@mutationuniform, mu});

            % run GA and time it
            tic;
            [x, fval, exitflag, output] = ga(func,nvar,[],[],[],[],[],[],[],
               options);
            times(t) = toc;
            fvals(t) = fval;
            gens(t) = output.generations;
        end

        results.Function(row) = fname;
        results.MutationRate(row) = mu;
        results.MeanFval(row) = mean(fvals);
        results.StdFval(row) = std(fvals);
        results.MeanTime(row) = mean(times);
        results.MeanGens(row) = mean(gens);
        row = row + 1;
    end
end

% Display / save
disp(results);
writetable(results, 'ga_sweep_results.csv');

% plot
for fIdx = 1 : nFunc
    fname = funcNames{fIdx};
    subT = results(strcmp(results.Function, fname),  :);

    fig = figure;
    yyaxis left
    errorbar(subT.MutationRate, subT.MeanFval, subT.StdFval,'-o','LineWidth'
       ,1.2);
    ylabel('Mean Best Fitness \pm 1\,STD');
    xlabel('Mutation Rate \mu');
    yyaxis right
    plot(subT.MutationRate, subT.MeanTime,'--s','LineWidth',1.2);
    ylabel('Mean Elapsed Time (s)');

    title('Effect of GA Mutation Rate on ', fname);
    grid on;
    legend('Fitness','Time','Location','northwest');
end

% objective functions
function f = rastriginsfcn(X)
    xx = [X(1), X(2)];
    f  = 10*numel(xx) + sum(xx.^2 - 10*cos(2*pi*xx));
end

function f = camel3humpfcn(X)
    x = X(1);
    y = X(2);
    f = 2*x^2 - 1.05*x^4 + (x^6)/6 + x*y + y^2;
end
```

C PCA on Data-set Code

```
clc; clear; close all;

load fisheriris.mat;

X = meas;
 mu = mean(X, 1);
 sigma = std(X, [], 1);
 Z = (X - mu) ./ sigma;

[coeff, score, latent, ~, explained] = pca(Z);

cumExplained = cumsum(explained);

figure('Name','Iris PCA: PC1 vs PC2','NumberTitle','off');
gscatter(score(:,1), score(:,2), species, 'rgb', 'o', 8);
xlabel('Principal Component 1');
ylabel('Principal Component 2');
title('Iris Dataset: Projection onto First Two PCs');
legend({'setosa','versicolor','virginica'}, 'Location','best');
grid on;

figure('Name','Iris PCA: Explained Variance','NumberTitle','off');
bar(explained, 'FaceColor',[.2 .2 .8]);
hold on;
plot(cumExplained, '-o', 'LineWidth', 2, 'Color',[.85 .33 .10]);
xlabel('Principal Component Index');
ylabel('Variance Explained (%)');
title('Scree Plot & Cumulative Variance');
xticks(1:4);
legend({'Individual PC','Cumulative'}, 'Location','best');
grid on;
hold off;
```

D K-means Code

```
clc; clear; close all;

load fisheriris
X_full = meas(:,1:2);
labels = species;

figure('Name','sepal length vs sepal width','NumberTitle','off');
gscatter(X_full(:,1), X_full(:,2), labels, 'rgb', 'o', 8);
xlabel('Sepal Length (cm)');
ylabel('Sepal Width  (cm)');
title('Iris Dataset: sepal length vs sepal width');
legend({'setosa','versicolor','virginica'}, 'Location','best');
grid on;

maxK = 6;  % test k = 2 to 6
avgSil = zeros(maxK,1);
allSil = cell(maxK,1);
opts = statset('UseParallel',false);

for k = 2:maxK
    rng(1);
    [idx, C] = kmeans(X_full, k, ...
```

```matlab
                     'Replicates', 10, ...
                     'Options', opts, ...
                     'Distance', 'sqeuclidean');
    sil_vals = silhouette(X_full, idx, 'sqeuclidean');
    avgSil(k) = mean(sil_vals);
    allSil{k} = sil_vals;
end

figure('Name','Silhouette Scores for k = 2 to 6','NumberTitle','off');
kRange = 2:maxK;
plot(kRange, avgSil(kRange), '-o', 'LineWidth', 1.5, 'MarkerSize', 8);
xlabel('Number of Clusters (k)');
ylabel('Average Silhouette Score');
title('Selecting k via Silhouette Analysis');
xticks(kRange);
grid on;

[~, bestK] = max(avgSil(2:maxK));
bestK = bestK + 1;  % because index 1 corresponds to k=2

fprintf('Optimal k by silhouette: %d\n', bestK);

rng(1);
[idx_final, centroids] = kmeans(X_full, bestK, ...
                                'Replicates', 10, ...
                                'Options', opts, ...
                                'Distance', 'sqeuclidean');

colors = lines(bestK);
figure('Name',sprintf('k-means Clustering (k = %d)', bestK),'NumberTitle','
   off');
hold on;
for c = 1:bestK
    scatter(X_full(idx_final==c,1), X_full(idx_final==c,2), ...
            40, colors(c,:), 'filled', 'MarkerFaceAlpha',0.6);
end
plot(centroids(:,1), centroids(:,2), 'kx', 'MarkerSize', 15, 'LineWidth', 2)
   ;
hold off;
xlabel('Sepal Length (cm)');
ylabel('Sepal Width  (cm)');
title(sprintf('k-means Clusters (k = %d) on Sepal Measurements', bestK));
legendStrings = arrayfun(@(x) sprintf('Cluster %d',x), 1:bestK, '
   UniformOutput',false);
legend([legendStrings, {'Centroids'}], 'Location','bestoutside');
grid on;

uniqueSpecies = unique(labels);
clusterLabels = strings(bestK,1);
accuracyCount = 0;
for c = 1:bestK
    members = labels(idx_final==c);
    if isempty(members)
        clusterLabels(c) = "N/A";
        continue
    end
    counts = cellfun(@(s) sum(strcmp(members,s)), uniqueSpecies);
    [~, idxMax] = max(counts);
    clusterLabels(c) = uniqueSpecies{idxMax};
```

```
end

assignedLabels = strings(150,1);
for i = 1:150
    assignedLabels(i) = clusterLabels(idx_final(i));
end

trueLabels = string(labels);
accuracy = mean(assignedLabels == trueLabels);
fprintf('Clustering accuracy (k = %d): %.2f%%\n', bestK, accuracy*100);
figure('Name','Confusion Matrix: Cluster vs True Species','NumberTitle','off');
confusionchart(trueLabels, assignedLabels);
title(sprintf('Confusion Matrix (k = %d)', bestK));
```