Take Home Exam SSY316

Matthew Newson, Erik Norlin

January 2023

Q1: Principal Component Analysis (PCA) of Genomes

1.

Performing singular value decomposition of the mean centered data set X with individuals as rows and features as columns yields $X = U\Sigma V^T$. The principal components are $U\Sigma \in \mathbb{R}^{995\times 995}$, the principal directions $V \in \mathbb{R}^{10101\times 995}$, and the variances along the diagonal of $\frac{1}{n-1}\Sigma^2 \in \mathbb{R}^{995\times 995}$.

2.

Figure 1 shows the first two principal components of the data set.

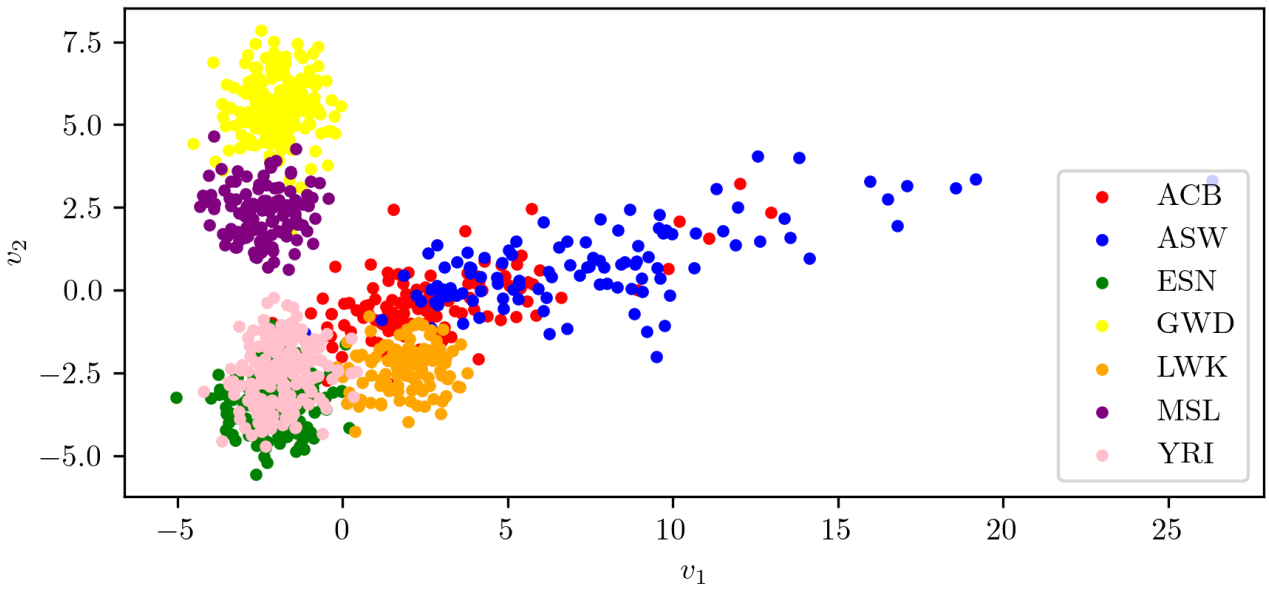


Figure 1: The first two principal components of the data set with labels African Caribbean in Barbados (ACB), African Ancestry in Southwest US (ASW), Esan in Nigeria (ESN), Gambian in Western Division (GWD), Luhya in Webuye (LWK), Mende in Sierra Leone (MSL), and Yoruba in Ibadan (YRI).

3.

The figure tells us that populations located in Nigeria (ESN and YRI) overlap and the populations located in America (ACB and ASW) overlap somewhat. Populations that do not share the same regions, such as the population in

Gambia and in Sierra Leone do not overlap as much.

An interpretation of the first two principal components could be that they capture genetic similarities and differences in the populations because populations living in the same regions tend to cluster together.

4.

Figure 2 shows the first and third principal components of the data set.

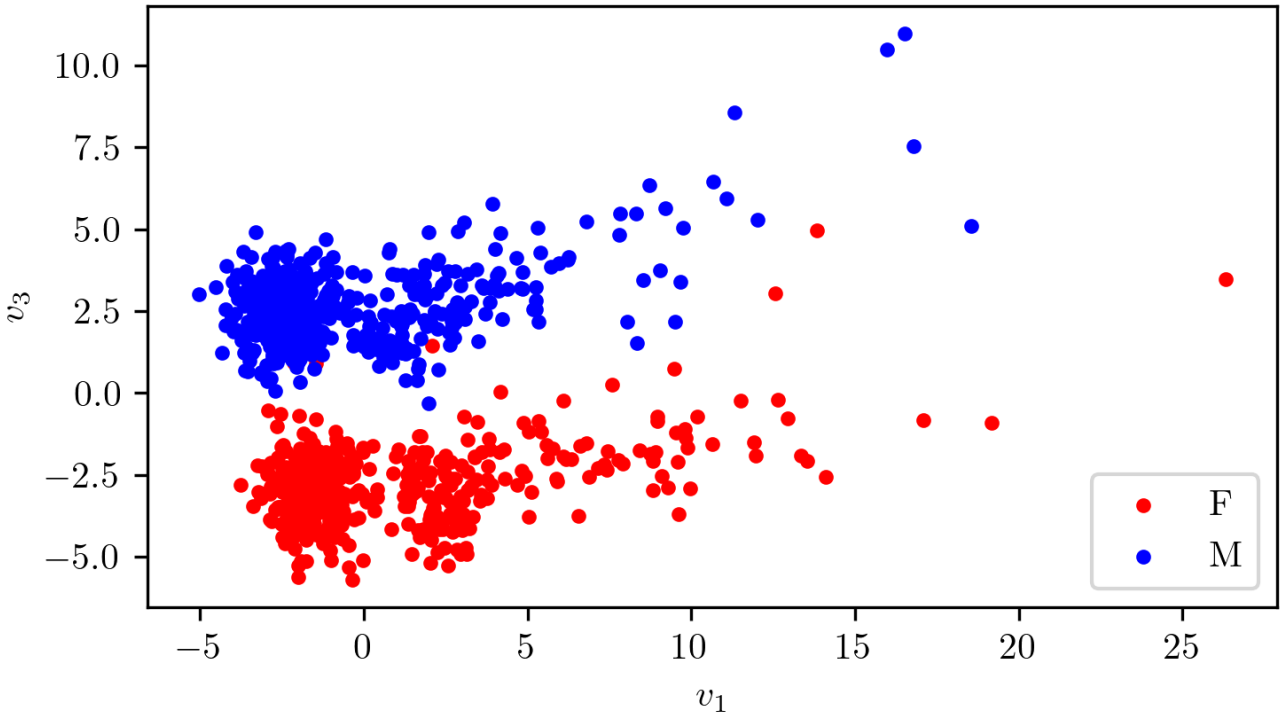


Figure 2: The first and third principal components of the data set with labels males (M) and females (F).

5.

From the figure we can see that the first and third principal components capture genetic differences in males and females.

6.

From inspection in Figure 3 it is noticeable that the third principal direction dominantly points in the direction of the dimensions representing nucleobase indices larger than 9500, which could indicate that these nucleobases are responsible for male and female differences.

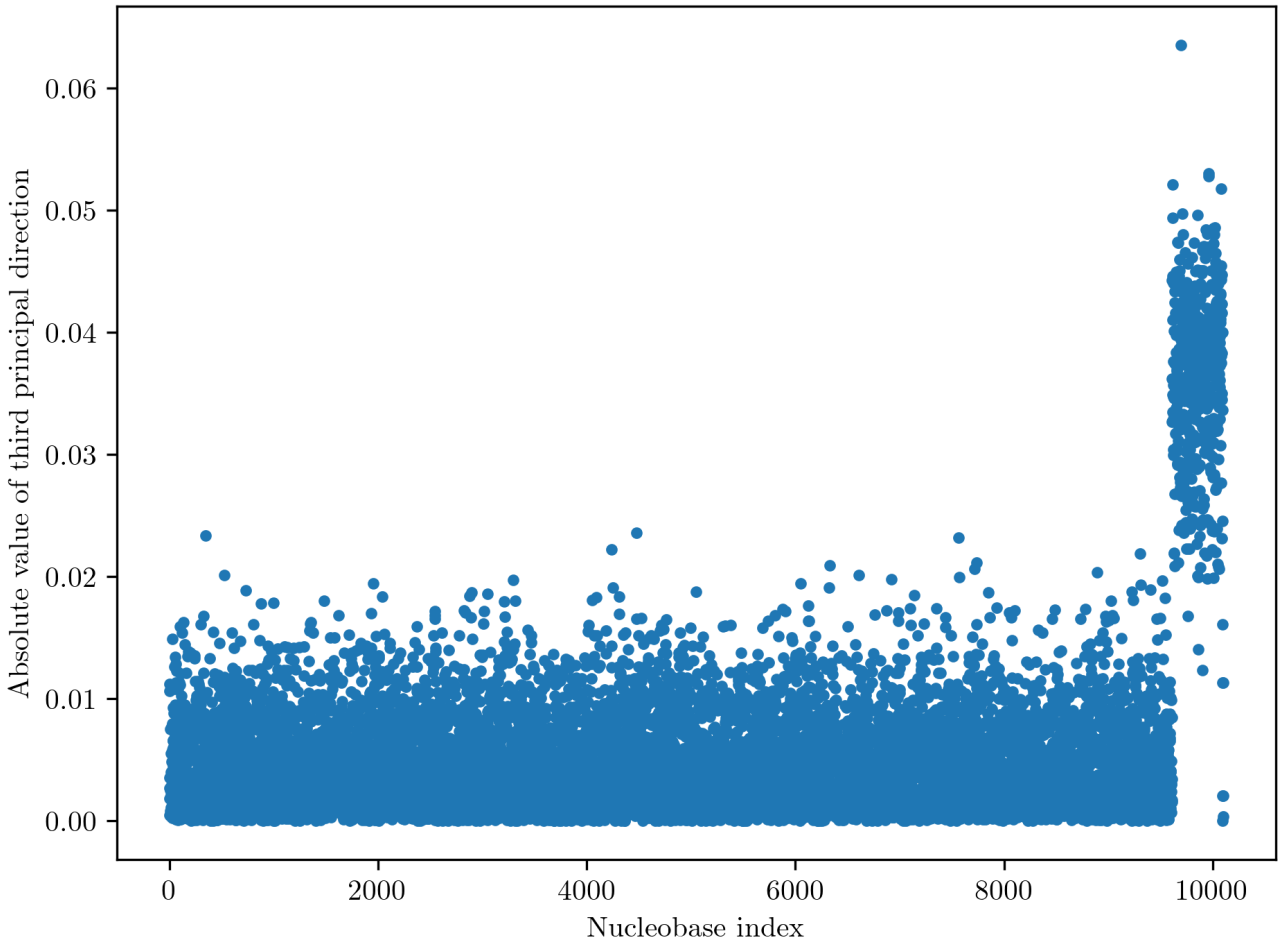


Figure 3: The absolute value of the third principal direction against the nucleobase index.

Q2: Markov Chain Monte Carlo

1.

The full joint likelihood is

$$p(\mathbf{y}, \boldsymbol{\theta}, \boldsymbol{\eta}) = p(\boldsymbol{\eta})p(\boldsymbol{\theta} \mid \boldsymbol{\eta})p(\mathbf{y} \mid \boldsymbol{\theta})$$

$$= p(\mu_{att})p(\mu_{def})p(\tau_{att})p(\tau_{def})p(home) \prod_{t=0}^{T=19} p(att_t \mid \mu_{att}, \tau_{att})p(def_t \mid \mu_{def}, \tau_{def}) \prod_{g=0}^{G=379} \prod_{j=0}^{J=1} p(\mathbf{y}_{gj} \mid \boldsymbol{\theta}_{gj})$$

$$= \mathcal{N}(\mu_{att} \mid 0, \tau_1^{-1})\mathcal{N}(\mu_{def} \mid 0, \tau_1^{-1})\mathrm{Gam}(\tau_{att} \mid \alpha, \beta)\mathrm{Gam}(\tau_{def} \mid \alpha, \beta)\mathcal{N}(home \mid 0, \tau_0^{-1})$$
$$\cdot \prod_{t=0}^{T=19} \mathcal{N}(att_t \mid \mu_{att}, \tau_{att}^{-1})\mathcal{N}(def_t \mid \mu_{def}, \tau_{def}^{-1})$$
$$\cdot \prod_{g=0}^{G=379} \mathrm{Po}(y_{g0} \mid home + att_{h(g)} - def_{a(g)})\mathrm{Po}(y_{g1} \mid att_{a(g)} - def_{h(g)})$$

2.

The Metropolis-Hastings algorithm for sampling from the posterior $p(\boldsymbol{\theta}, \boldsymbol{\eta} \mid \mathbf{y})$ is shown in Algorithm 1.

Algorithm 1 Metropolis-Hastings

Initialize $\mathbf{x}_s$
$i = 0$
while *number of samples* $<$ *number of samples to sample* **do**
  Sample hyper-priors and priors $\mathbf{x}_{s+1}$ from the proposal distribution $\mathcal{N}(\mathbf{x}_s, \sigma^2 I)$
  Compute the probabilities $p(\mathbf{x}_s)$ and $p(\mathbf{x}_{s+1})$ given the observed data
  Compute the acceptance probability $p_A = p(\mathbf{x}_{s+1})/p(\mathbf{x}_s)$
  Accept the new sample $\mathbf{x}_s \leftarrow \mathbf{x}_{s+1}$ **if** $u \sim \mathcal{U}(0, 1) < p_A$
  Save $\mathbf{x}_s$ every t:th iteration and **if** $i >$ burn in
  Add one to i
end

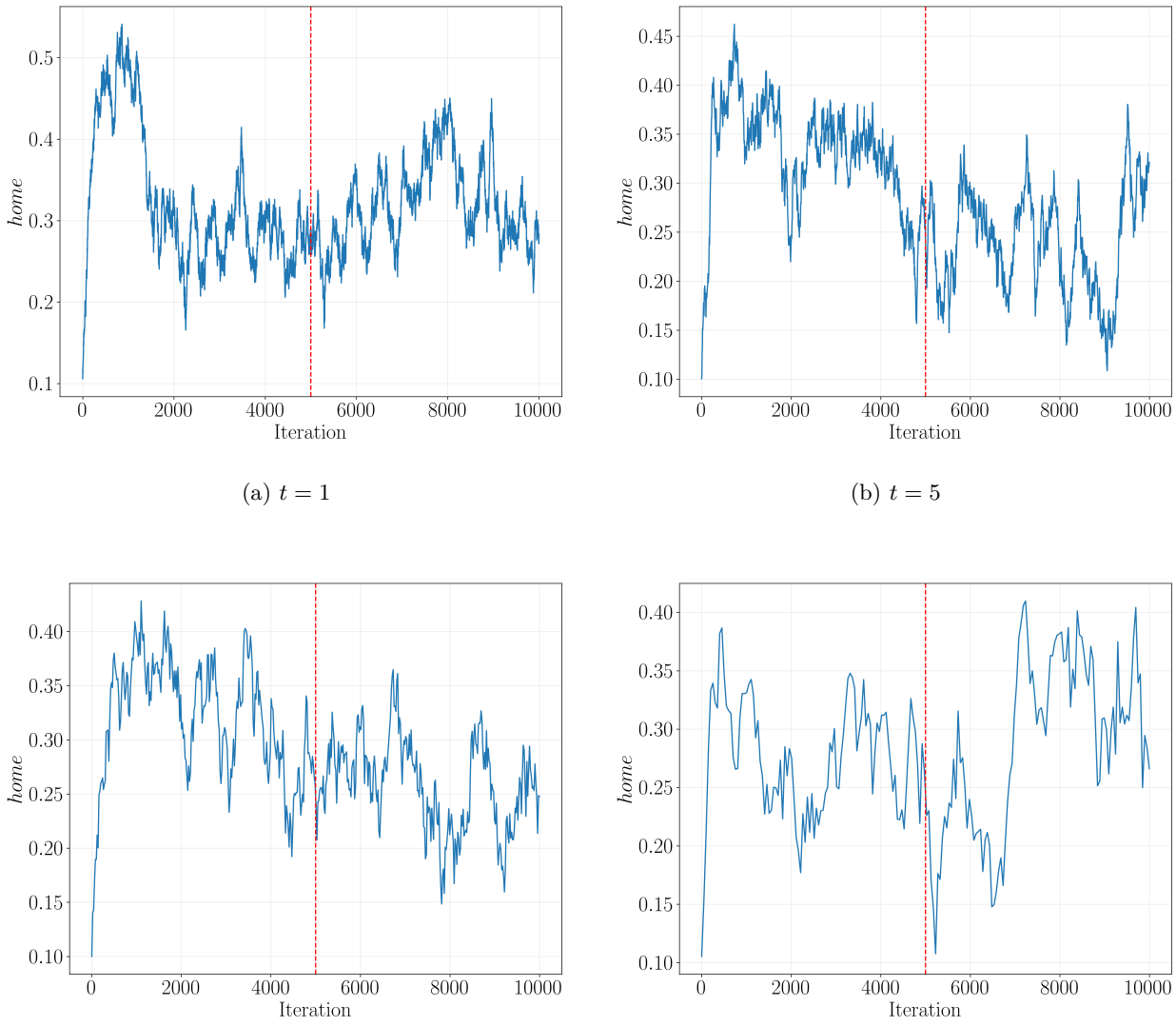
3.

The Metropolis-Hastings algorithm was run for the parameter sets $\sigma = 0.005, 0.05, 0.5$ and $t = 1, 5, 20, 50$ with a burn-in phase of 5000 steps. Hence, the total number of iterations for each parameter set were $5000 + 5000t$. Figures 4, 5 and 6 show trace plots of the *home* variable against iterations. *home* is only plotted against the first 10000 iterations for every parameter set to make a more fair comparison between the trace plots. Also, the rejection ratios for the 5000 samples for each parameter set are presented in Table 1.

The rejection rate for $\sigma = 0.005$ is low, around 10% for every t, causing the chain to move around quickly which is good, but it does not cover a wide range of the distribution evenly. This makes sense because the deviation of the proposal distribution is so small that proposed samples are always close to current samples.

For $\sigma = 0.5$ we have the opposite case where the deviation of proposed samples from current samples is so large that they are rejected too often, around 100% of the time for each t. The chain covers a wide range of the distribution, but fails to move around and gets stuck on some values for long periods of time.

Lastly, for $\sigma = 0.05$ we have a relatively high rejection ratio, around 80% for each t. In this case however, the chain moves around reasonably quickly and covers a wide range of the distribution. $t = 5$ yields the smallest rejection rate for this standard deviation, and we can see from the trace plots that this chain looks like be the be the most stable one. Thus, the parameter set that seems the most optimal is $\sigma = 0.05$ and $t = 5$.



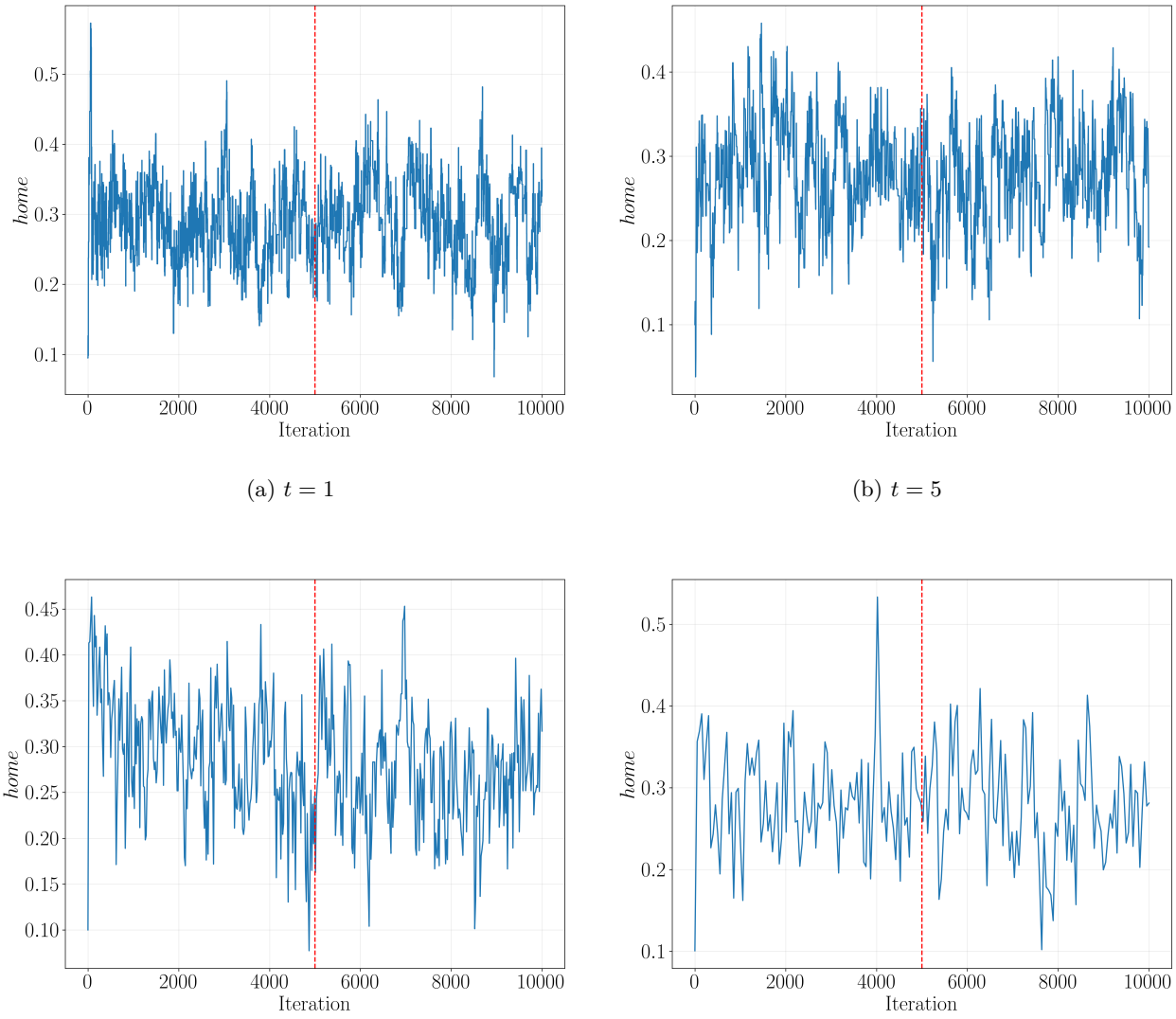
(a) $t = 1$

(b) $t = 5$

(c) $t = 20$

(d) $t = 50$

Figure 4: Trace plots of the *home* variable against the 10000 iterations for $\sigma = 0.005$ and $t = 1, 5, 20, 50$. The red dashed line marks the end of the burn-in period.



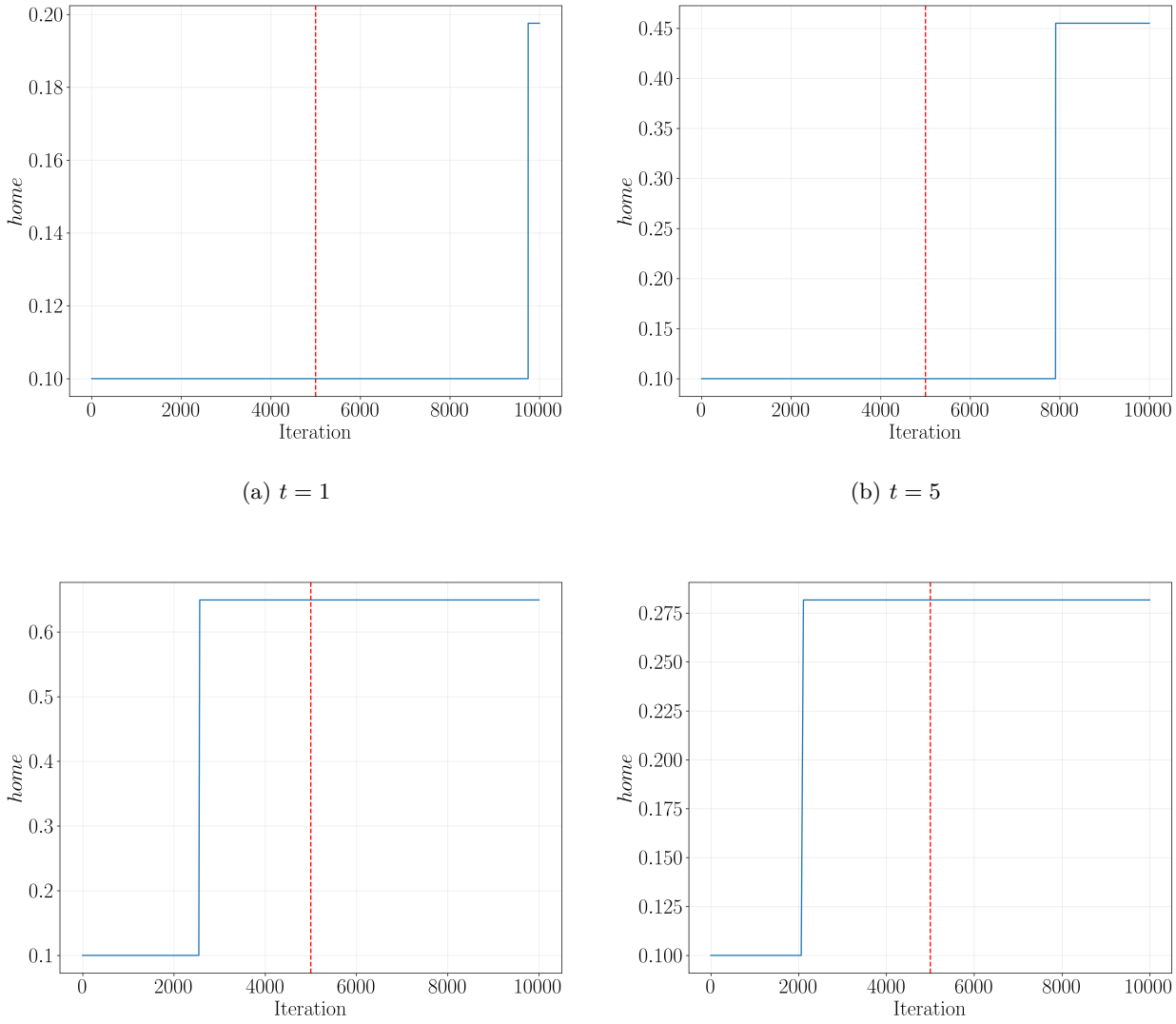
(a) $t = 1$

(b) $t = 5$

(c) $t = 20$

(d) $t = 50$

Figure 5: Trace plots of the *home* variable against the first 10000 iterations for $\sigma = 0.05$ and $t = 1, 5, 20, 50$. The red dashed line marks the end of the burn-in period.



(a) $t = 1$

(b) $t = 5$

(c) $t = 20$

(d) $t = 50$

Figure 6: Trace plots of the *home* variable against the first 10000 iterations for $\sigma = 0.5$ and $t = 1, 5, 20, 50$. The red dashed line marks the end of the burn-in period.

Table 1: Rejection ratios for 5000 samples for each parameter set of σ and t.

$t \setminus \sigma$	0.005	0.05	0.5
1	0.1080	0.8196	0.9998
5	0.0995	0.8158	0.9999
20	0.1039	0.8270	1.0000
50	0.1038	0.8299	1.0000

Figure 7 shows the posterior histogram of 5000 MCMC samples of the *home* variable for the optimal parameter set. We can see that it closely follows the PDF of the proposal distribution.

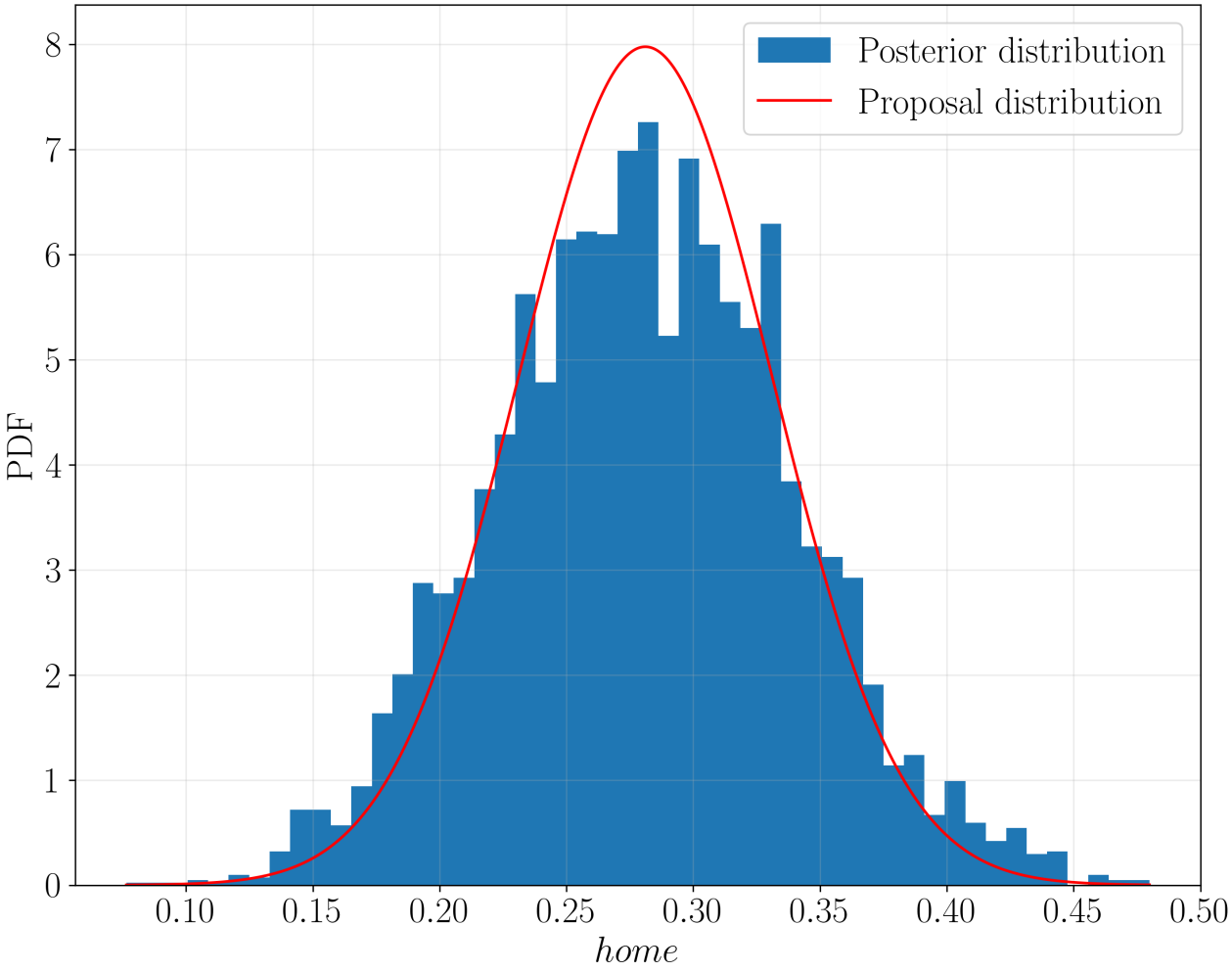


Figure 7: Posterior histogram of 5000 MCMC samples of the *home* variable plotted side by side with the proposal distribution for the optimal parameter set $\sigma = 0.5$ and $t = 5$.

The expected attack against the expected defense of each team in Premier League 2013-2014 is presented in Figure 8.

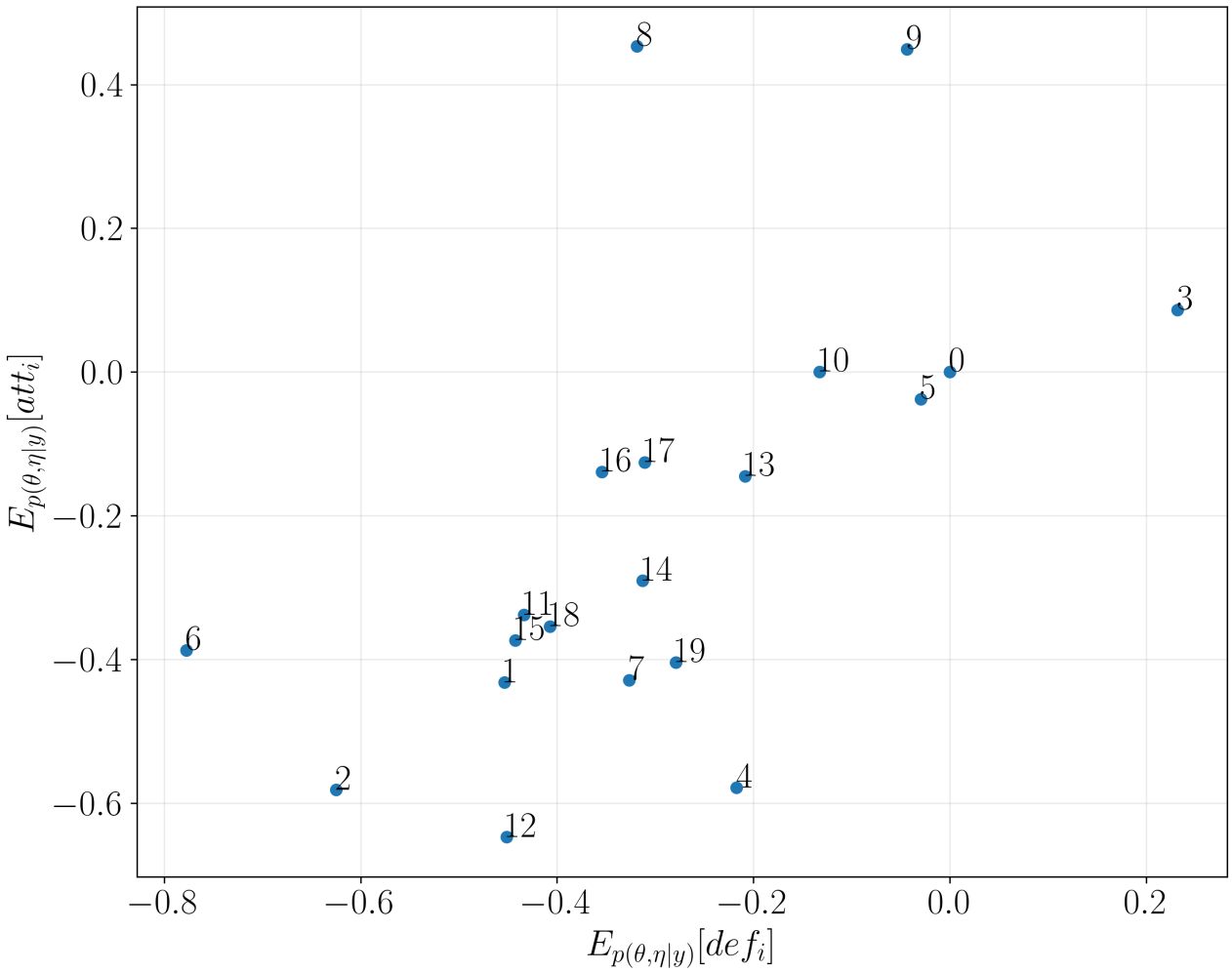


Figure 8: The expected attack against the expected defense of each team in Premier League 2013-2014. The indices of the points maps to the team indices of the data set.

Q3: Variational Inference (VI)

A Review of Vanilla LDA

1.

False, the LDA involves posterior inference with the Dirichlet distribution as a conjugate prior to the Multinomial distribution.

2.

True, non-convex optimization is required.

3.

The model parameters are $\boldsymbol{\alpha} \in \mathbb{R}^K$ and $\boldsymbol{\eta} \in \mathbb{R}^V$. Thus, there are $K + V$ number of model parameters.

4.

The computational complexity for one document in mean-field VI for LDA is $\mathcal{O}(N^2K)$, where N is the number of words in the document and K is the number of topics.

5.

The memory complexity for one document in mean-field VI for LDA is $\mathcal{O}(K(N + V + 1))$, where N is the number of words in the document, K the number of topics and V the number of unique words.

6.

False, it is used for extracting topics from text corpora.

More HMMs

1.

Observed variables
w and z.

Hidden variables
δ, p, and θ.

Model parameters to be estimated
α, β, γ, and T.

Q4: QWOP

3.

Three methods were implemented to optimize QWOP. The first being uniform random search, the second being gradient descent (GD) with fixed step rate and the last being an evolutionary algorithm (EA). Random search was the first solution that came to mind and was straight forward to implement; in a for loop, pick a random sequence and simulate the plan. The best plan obtained using random search was obtained after 40000 iterations and resulted in a total distance of 6 m. GD on the other hand did not perform as well and topped at 3 m, and seemingly got stuck every time. Since the objective function consists of 40 variables and is assumably highly non-convex, it makes sense that any given instance of GD will most likely quickly converge to a local minimum resulting in poor performance in the game. For this reason, an EA was considered. EAs are suitable for optimization problems where the objective function is highly non-convex. It consists of a population of randomly initialized individuals (game plans in this case) and the ones with higher fitness (better performance in the game) reproduce with each other, often leading to offsprings with even higher fitness. Over time the fitness of the individuals gets better and better resulting in continuously better performance in the game. The EA that was implemented was initialized with 500 individuals and ran for 100 generations. The best performing individual got to almost 8 m, beating random search by roughly 2 m. The corresponding plan is presented in Listing 1.

```
plan = [-1, -1, -1, 1, -1, -1, -1, 1, -1, -1, -1, -1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1, -1, -1,
    -1, -1, 1, 1, -1, 1, -1, -1, -1, -1, -1, -1, 1, 1, 1, -1]
```

Listing 1: Best plan obtained for QWOP by the EA. Yields a total distance of 7.84 m.

Furthermore, the initial plan in GD was experimented with in hope to improve GD's performance. Starting GD with the best plans obtained from random search and the EA respectively did however not yield any better performance. GD still converged quickly at around 3 m every time.

Appendix

```
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.decomposition import PCA


def get_most_appearing_char(column):
    return column.value_counts().idxmax()

df = pd.read_csv('p4dataset2023.txt', sep=' ', header=None)
most_appearing_chars = df.apply(get_most_appearing_char, axis=0)
X_arr = df.iloc[:,3:].eq(most_appearing_chars[3:]).astype(int)
X_arr_centered = X_arr - X_arr.mean(axis=0)

U, S, VT = la.svd(X_arr_centered, full_matrices=False)
print('U shape:', U.shape, 'S shape:', S.shape, 'VT shape:', VT.shape)

pca = PCA(n_components=3)
X_pca = pca.fit(X_arr_centered)
XV = X_pca.transform(X_arr_centered)

labels_data = np.array(df.iloc[:,2])
labels = np.unique(labels_data)
colors = ['red', 'blue', 'green', 'yellow','orange', 'purple', 'pink', 'brown']
cdict = {keyword: colors[index] for index, keyword in enumerate(labels)}

fig = plt.figure(figsize=(6,3))
ax = plt.axes()
for label in labels:
    idx = np.where(label == labels_data)
    ax.scatter(XV[idx,0], XV[idx,1], marker='.', c=cdict[label], label=label)
ax.set_xlabel('$v_1$')
ax.set_ylabel('$v_2$')
ax.set_aspect('equal')
ax.legend(loc="lower right")
fig.tight_layout()
plt.show()

labels_data = np.array(df.iloc[:,1])
labels = np.unique(labels_data)
colors = ['red', 'blue', 'green', 'yellow','orange', 'purple', 'pink', 'brown']
cdict = {keyword: colors[index] for index, keyword in enumerate(labels)}

fig = plt.figure(figsize=(5,3))
ax = plt.axes()
for label in labels:
    idx = np.where(label == labels_data)
    ax.scatter(XV[idx,0], XV[idx,2], marker='.', c=cdict[label], label=label)
ax.set_xlabel('$v_1$')
ax.set_ylabel('$v_3$')
ax.set_aspect('equal')
ax.legend(loc="lower right")
fig.tight_layout()
plt.show()

fig = plt.figure()
ax = plt.axes()
ax.plot(np.abs(X_pca.components_[2]), '.')
ax.set_xlabel('Nucleobase index')
ax.set_ylabel('Absolute value of third principal direction')
fig.tight_layout()
```

```
plt.show()
```

Listing 2: Python script for Q1.

```
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt
import pandas as pd
import scipy.stats as st


def enforce_corner_constraint(theta, n_teams):
    theta[0] = theta[n_teams] = 0
    return theta

def propose_samples(current, sigma):
    I = np.eye(current.shape[0])
    proposal = np.random.multivariate_normal(current, cov=I*sigma**2)
    return proposal

def compute_hyper_prior_probabilities(eta):
    tau1 = 0.0001
    alpha = beta = 0.1

    mu_attack_logprob = st.norm(0, 1/np.sqrt(tau1)).logpdf(eta[0])
    mu_defense_logprob = st.norm(0, 1/np.sqrt(tau1)).logpdf(eta[1])
    tau_attack_logprob = st.gamma(alpha, scale=1/beta).logpdf(eta[2])
    tau_defense_logprob = st.gamma(alpha, scale=1/beta).logpdf(eta[3])

    eta_logprob = mu_attack_logprob + mu_defense_logprob + tau_attack_logprob +
    tau_defense_logprob
    return eta_logprob

def compute_prior_probabilities(eta, theta, n_teams):
    tau0 = 0.0001

    mu_attack = eta[0]
    mu_defense = eta[1]
    tau_attack = eta[2]
    tau_defense = eta[3]

    home = theta[-1]
    attack = theta[:n_teams]
    defense = theta[n_teams:-1]

    home_logprob = st.norm(0, 1/np.sqrt(tau0)).logpdf(home)
    attack_logprob = st.norm(mu_attack, 1/np.sqrt(tau_attack)).logpdf(attack)
    defense_logprob = st.norm(mu_defense, 1/np.sqrt(tau_defense)).logpdf(defense)

    theta_prob = home_logprob + np.sum(attack_logprob + defense_logprob)
    return theta_prob

def compute_likelihood(theta, n_teams, dataf):

    goals_home = dataf['goals_home'].values.astype(int)
    goals_away = dataf['goals_away'].values.astype(int)
    home_team = dataf['home_team'].values.astype(int)
    away_team = dataf['away_team'].values.astype(int)

    home = theta[-1]
    attack = theta[:n_teams]
    defense = theta[n_teams:-1]

    theta_home = np.exp(home + attack[home_team] - defense[away_team])
    theta_away = np.exp(attack[away_team] - defense[home_team])
```

```
61
62      loglikelihood_home = st.poisson(theta_home).logpmf(goals_home)
63      loglikelihood_away = st.poisson(theta_away).logpmf(goals_away)
64
65      loglikelihood = np.sum(loglikelihood_home + loglikelihood_away)
66      return loglikelihood
67
68  def compute_probabilities(eta, theta, n_teams, dataf):
69      eta_logprob = compute_hyper_prior_probabilities(eta)
70      theta_logprob = compute_prior_probabilities(eta, theta, n_teams)
71      loglikelihood = compute_likelihood(theta, n_teams, dataf)
72      logprob = eta_logprob + theta_logprob + loglikelihood
73      return logprob
74
75  def metropolis_hastings(n_samples, n_teams, sigma, dataf, thinning, burn_in, sample_burn_in=False)
        :
76
77      starting_point = 0.1
78      eta_current = np.full(4, starting_point)
79      theta_current = enforce_corner_constraint(np.full(1+n_teams*2, starting_point), n_teams)
80      samples = []
81      rejection = []
82      t = 0
83
84      if sample_burn_in:
85          n_samples = n_samples+(burn_in/thinning)
86
87      while n_samples > len(samples):
88
89          eta_proposal = propose_samples(eta_current, sigma)
90          theta_proposal = enforce_corner_constraint(propose_samples(theta_current, sigma), n_teams)
91
92          current_logprob = compute_probabilities(eta_current, theta_current, n_teams, dataf)
93          proposal_logprob = compute_probabilities(eta_proposal, theta_proposal, n_teams, dataf)
94          acceptance_logprob = proposal_logprob - current_logprob
95
96          u = np.random.uniform()
97          if np.log(u) < acceptance_logprob:
98              eta_current = eta_proposal.copy()
99              theta_current = theta_proposal.copy()
100             rejection.append(0)
101         else:
102             rejection.append(1)
103
104         if t % thinning == 0:
105             if sample_burn_in:
106                 samples.append(theta_current.copy())
107             elif t > burn_in:
108                 samples.append(theta_current.copy())
109
110         t += 1
111         print(f'\r{len(samples)} / {n_samples} samples', end='')
112
113     return np.array(samples), np.array(rejection)
114
115
116 dataf = pd.read_csv('premier_league_2013_2014.dat', sep=',', header=None)
117 dataf.columns = ['goals_home', 'goals_away', 'home_team', 'away_team']
118
119
120 n_samples = 5000
121 n_teams = 20
122 sigmas = [0.005, 0.05, 0.5]
123 thinning = [1, 5, 20, 50]
124 burn_in = 5000
```

```
home_batches = []
rejection_batches = []

for i in range(len(sigmas)):
    for j in range(len(thinning)):

        print(f'\nRunning sigma={sigmas[i]} and thinning={thinning[j]}')
        samples, rejection = metropolis_hastings(n_samples, n_teams, sigmas[i], dataf, thinning[j], burn_in, sample_burn_in=True)
        homes = samples[:,-1]
        home_batches.append(homes)
        rejection_batches.append(rejection)


plot_its = burn_in + n_samples
t = 0
for i in range(len(sigmas)):
    for j in range(len(thinning)):

        rejection = rejection_batches[t]
        rejection = rejection[burn_in:]
        print(f'Rejection rate: sigma={sigmas[i]}, thinning={thinning[j]}:', np.round(np.mean(rejection), 4))

        home = home_batches[t]
        plot_samples = plot_its//thinning[j]
        its = np.array(np.linspace(0,plot_its,plot_samples))
        n_homes = home.shape[0]

        fig = plt.figure()
        ax = plt.axes()
        ax.plot(its, home[:plot_samples], '-')
        # ax.plot(np.array(np.linspace(0,n_homes*thinning[j],n_homes)), home, '-')
        ax.axvline(x=burn_in, color='red', linestyle='--', label='Burn-in')
        ax.set_xlabel('Iteration')
        ax.set_ylabel('$home$')
        ax.grid(True, alpha=0.25)
        fig.tight_layout()
        plt.show()

        t += 1


samples, _ = metropolis_hastings(n_samples, n_teams, sigmas[1], dataf, thinning[1], burn_in, sample_burn_in=False)
homes = samples[:,-1]
proposal_x = np.linspace(np.min(homes), np.max(homes), 1000)
proposal_y = st.norm(np.mean(homes), sigmas[1]).pdf(proposal_x)

fig = plt.figure()
plt.hist(homes, bins=50, density=True, label='Posterior distribution')
plt.plot(proposal_x, proposal_y, color='red', label='Proposal distribution')
plt.xlabel('$home$')
plt.ylabel('PDF')
plt.grid(True, alpha=0.25)
plt.legend()
fig.tight_layout()
plt.show()


expected_attack = np.mean(samples[:,:n_teams], 0)
expected_defense = np.mean(samples[:,n_teams:-1], 0)

fig = plt.figure()
ax = plt.axes()
```

```
ax.plot(expected_defense, expected_attack, 'o')
for i in range(expected_defense.shape[0]):
    ax.text(expected_defense[i], expected_attack[i], str(i))
ax.set_xlabel(r'$E_{p(\theta,\eta\mid y)}[def_i]$')
ax.set_ylabel(r'$E_{p(\theta,\eta\mid y)}[att_i]$')
ax.grid(True, alpha=0.25)
fig.tight_layout()
plt.show()
```

Listing 3: Python script for Q2.

```
def tournament_select(fitness_list, tournament_probability, tournament_size):
    most_fit = 0
    fitness_index = 0
    individual_index = 1

    population_size = len(fitness_list)
    tour_fitness_and_individual = np.zeros((tournament_size, 2))

    for i in range(tournament_size):
        i_tmp = np.random.randint(0, population_size)
        tour_fitness_and_individual[i, fitness_index] = fitness_list[i_tmp]
        tour_fitness_and_individual[i, individual_index] = i_tmp

    tour_fitness_and_individual = tour_fitness_and_individual[tour_fitness_and_individual[:,
    fitness_index].argsort()[::-1]]

    while True:
        r = np.random.rand()
        if r < tournament_probability:
            selected_individual_index = int(tour_fitness_and_individual[most_fit, individual_index
    ])
            return selected_individual_index
        else:
            remaining_participants = tour_fitness_and_individual.shape[0]
            if remaining_participants > 1:
                tour_fitness_and_individual = np.delete(tour_fitness_and_individual, most_fit,
    axis=0)
            else:
                selected_individual_index = int(tour_fitness_and_individual[most_fit,
    individual_index])
                return selected_individual_index


def crossover(individual1, individual2):
    n_genes = len(individual1)
    crossover_point = np.random.randint(1, n_genes)
    new_individuals = np.zeros((2, n_genes))

    for j in range(n_genes):
        if j < crossover_point:
            new_individuals[0, j] = individual1[j]
            new_individuals[1, j] = individual2[j]
        else:
            new_individuals[0, j] = individual2[j]
            new_individuals[1, j] = individual1[j]

    return new_individuals


def decode_chromosome(chromosome, number_of_variables, maximum_variable_value):
    x = np.zeros(number_of_variables)
    n_genes = len(chromosome)
    variable_length = int(np.floor(n_genes / number_of_variables))
    i_gene = 0
```

```
    maximum_variable_value = np.abs(maximum_variable_value)

    for i in range(number_of_variables):
        x[i] = 0.0
        for j in range(0, variable_length):
            x[i] += 2**-j * chromosome[i_gene]
            i_gene += 1

        # x[i] = -maximum_variable_value + ((2 * maximum_variable_value * x[i]) / (1 - 2**(-
    variable_length)))
        x[i] = -maximum_variable_value + (2 * maximum_variable_value * x[i]) / (2**variable_length
     - 1)

    return x


def evaluate_individual(x): # objective funtion
    fitness = sim(x)
    return fitness


def initialize_population(population_size, number_of_genes):
    population = np.zeros((population_size, number_of_genes), dtype=int)

    for i in range(population_size):
        for j in range(number_of_genes):
            r = np.random.rand()
            if r < 0.5:
                population[i, j] = 0
            else:
                population[i, j] = 1

    return population


def mutate(individual, mutation_probability):
    n_genes = len(individual)
    mutated_individual = np.copy(individual)

    for i in range(n_genes):
        r = np.random.rand()
        if r < mutation_probability:
            mutated_individual[i] = 1 - individual[i]

    return mutated_individual


def run_function_optimization(population_size, number_of_genes, number_of_variables,
    maximum_variable_value,
                              tournament_size, tournament_probability, crossover_probability,
    mutation_probability,
                              number_of_generations):
    maximum_fitness_best = 0
    best_variable_values_best = np.zeros(number_of_variables)
    population = initialize_population(population_size, number_of_genes)

    for generation in range(number_of_generations):
        maximum_fitness = 0.0
        fitness_list = np.zeros(population_size)
        for i in range(population_size):
            chromosome = population[i, :]
            variable_values = decode_chromosome(chromosome, number_of_variables,
    maximum_variable_value)
            fitness_list[i] = evaluate_individual(variable_values)
            if fitness_list[i] > maximum_fitness:
```

```
                maximum_fitness = fitness_list[i]
                i_best_individual = i
                best_variable_values = variable_values

            if fitness_list[i] > maximum_fitness_best:
                maximum_fitness_best = fitness_list[i]
                best_variable_values_best = variable_values

        temporary_population = population
        for i in range(0, population_size, 2):
            i1 = tournament_select(fitness_list, tournament_probability, tournament_size)
            i2 = tournament_select(fitness_list, tournament_probability, tournament_size)
            r = np.random.rand()
            if r < crossover_probability:
                individual1 = population[i1, :]
                individual2 = population[i2, :]
                new_individual_pair = crossover(individual1, individual2)
                temporary_population[i, :] = new_individual_pair[0, :]
                temporary_population[i+1, :] = new_individual_pair[1, :]
            else:
                temporary_population[i, :] = population[i1, :]
                temporary_population[i+1, :] = population[i2, :]

        temporary_population[0, :] = population[i_best_individual, :]
        for i in range(1, population_size):
            temp_individual = mutate(temporary_population[i, :], mutation_probability)
            temporary_population[i, :] = temp_individual

        population = temporary_population
        print('generation:', generation, 'maximum_fitness:', maximum_fitness_best, end='\r')

    return maximum_fitness_best, best_variable_values_best


def approx_grad(plan, eta):
    grad = (sim(plan + eta) - sim(plan)) / eta
    return grad


def grad_descent(plan, eta, n_its):
    tot_dists = [0]
    best_plan = plan.copy()

    for t in range(n_its):

        grad = approx_grad(plan, eta)
        alpha = eta
        plan = plan - alpha*grad
        dist = sim(plan)

        if tot_dists[-1] < dist:
            tot_dists.append(dist)
            best_plan = plan.copy()
        else:
            tot_dists.append(tot_dists[-1])

        print('t =', t, 'max dist:', np.max(tot_dists), end='\r') if t%10 == 0 else None

    return best_plan, tot_dists


def random_search(n_its):
    tot_dists = [0]
    best_plan = None

```

```
    for t in range(n_its):

        plan = np.random.uniform(-1, 1, 40)
        dist = sim(plan)

        if tot_dists[-1] < dist:
            tot_dists.append(dist)
            best_plan = plan.copy()
        else:
            tot_dists.append(tot_dists[-1])

        print('t =', t, 'max dist:', np.max(tot_dists), end='\r') if t%10 == 0 else None

    return best_plan, tot_dists


if __name__ == "__main__":

    n_its = 1000
    eta = 0.1
    samples = np.linspace(0, n_its, n_its+1)
    plan = np.random.uniform(-1, 1, 40)

    best_plan, tot_dists = random_search(n_its)
    print('\nRandom search: \nbest dist:', tot_dists[-1], '\nbest plan:', best_plan)
    best_plan, tot_dists = grad_descent(plan, eta, n_its)
    print('\nGradient descent: \nbest dist:', tot_dists[-1], '\nbest plan:', best_plan)

    population_size = 500
    maximum_variable_value = 1
    number_of_genes = 50
    number_of_variables = 40

    tournament_size = 3
    tournament_probability = 0.785
    crossover_probability = 0.8
    mutation_probability = 0.02
    number_of_generations = 100

    maximum_fitness, best_plan = run_function_optimization(
        population_size, number_of_genes, number_of_variables, maximum_variable_value,
        tournament_size, tournament_probability, crossover_probability, mutation_probability,
        number_of_generations
    )
    print('\nEvolutionary algorithm: \nbest dist:', maximum_fitness, '\nbest plan:', best_plan)

    best_plan = [-1, -1, -1, 1, -1, -1, -1, 1, -1, -1, -1, -1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1,
     -1, -1, -1, -1, 1, 1, -1, 1, -1, -1, -1, -1, -1, -1, 1, 1, 1, -1]

    dist = sim(best_plan)
    print('Best dist;', dist)
    print('Best plan:', best_plan)
```

Listing 4: The implemented methods for optimizing QWOP in Python for Q4.