MSc Statistical Programming 2023 Assessed Practical Assignment

P151

December 18, 2023

1 British House Prices

1.1 Exploratory Data Analysis

Figure 1 below displays the evolution of House Prices in England over time.

Figure 1: Average House Price in England Over Time

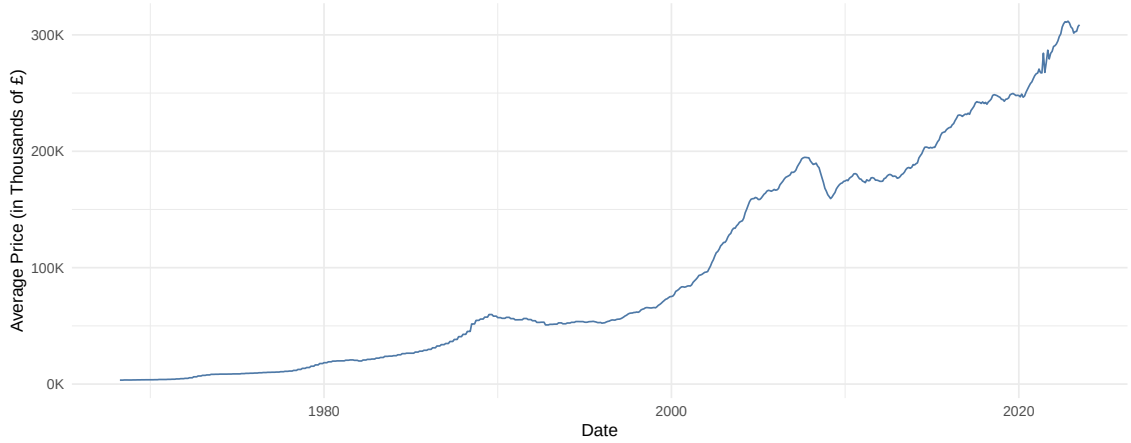


Figure 2 below additionally displays the evolution of House Prices in the Oxford Regions as well as England as a whole over time.

Table 1 below shows that of the ten regions with the highest median of the ratio $\frac{regional\ average\ house\ price}{average\ house\ price\ in\ England}$. Perhaps unsurprisingly, we can see that the ten most expensive regions by this metric are geographically concentrated in London and its suburbs, and that no Oxford Regions are included.

Figure 3 below compares the month-on-month percentage increase in House Prices in England to monthly inflation, defined as the percentage increase in the monthly Consumer Price Index over the period in which data for both is available. We can see that like in most countries, the increase in the average price of a house has drastically outpaced the general cost-of-living increase since the year 2000.

Figure 2: Comparison of Average House Prices between England and Oxford Regions

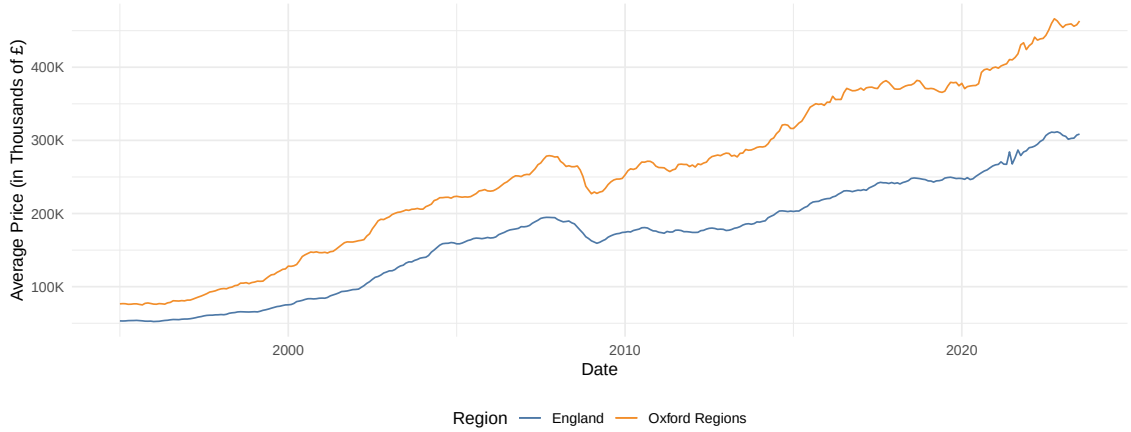


Table 1: Ten Highest Median Ratios of Average House Prices

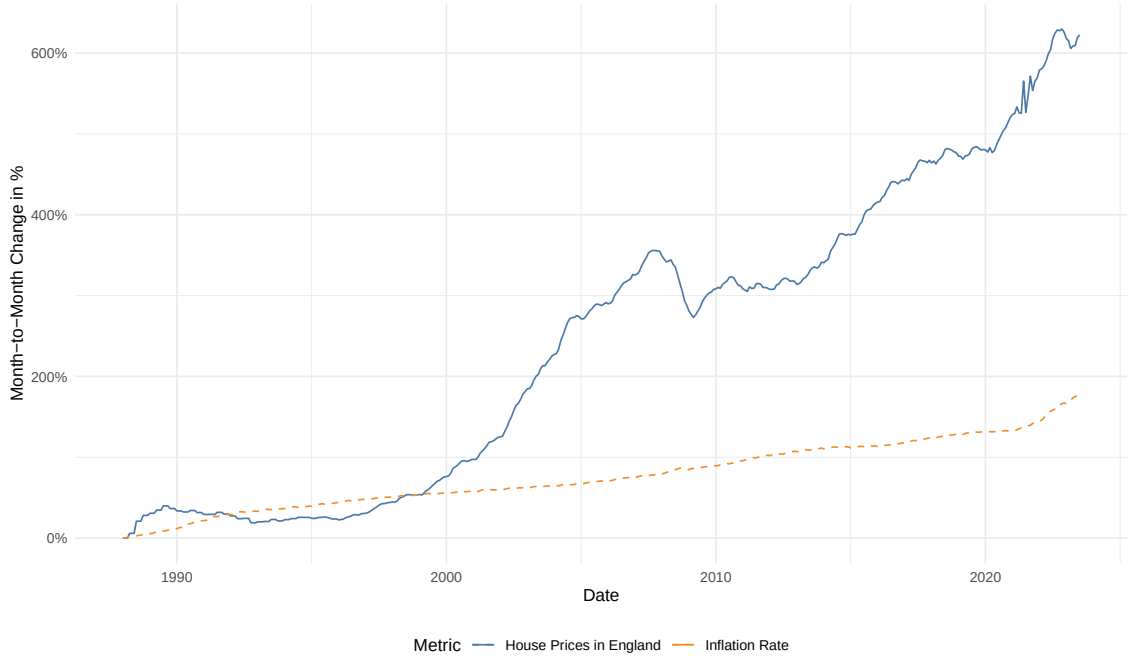
	Region Name	Median Ratio	Initial Price	Final Price	% Increase	Oxford?
1	Kensington and Chelsea	4.59	182694.83	1344669.00	636.02	FALSE
2	City of Westminster	3.24	133025.28	954356.00	617.42	FALSE
3	Camden	2.92	120932.89	828389.00	585.00	FALSE
4	Hammersmith and Fulham	2.79	124902.86	798537.00	539.33	FALSE
5	City of London	2.56	91448.98	905489.00	890.16	FALSE
6	Richmond upon Thames	2.51	109326.12	770107.00	604.41	FALSE
7	Elmbridge	2.34	106524.00	678279.00	536.74	FALSE
8	Islington	2.32	92516.49	710181.00	667.63	FALSE
9	Wandsworth	2.15	88559.04	632895.00	614.66	FALSE
10	Mole Valley	1.98	101899.76	563083.00	452.59	FALSE

2 Chromosome Painting

2.1 Implementation of `forward` Algorithm

```r
# Function to calculate emission probability (eq. 3)
emission_probability <- function(observed, reference, error = 0.1) {
  return ((1 - error)^(observed == reference) * error^(observed != reference))
}

# matrices alpha and beta , both with K rows and T columns
forward <- function(haps, hap, error = 0.1) {
  K <- nrow(haps)
  T <- ncol(haps)

  # Initialize alpha matrix
  alpha <- matrix(0, nrow = K, ncol = T)

  # Compute initial and emission probabilities
  pi <- 1 / K  # (eq. 1)
  for (k in 1:K) { # (eq. 4)
    alpha[k, 1] <- pi * emission_probability(hap[1], haps[k, 1], error)
```

Figure 3: Percentage Increase in House Prices compared to Inflation Rate




```
  }

  # Induction step to compute (eq. 5)
  for (t in 2:T) {
    for (k in 1:K) {
      transition_sum <- 0
      for (i in 1:K) {
  A_ik <- ifelse(i == k, (1 - 0.999) / K + 0.999, (1 - 0.999) / K) # (eq. 2)
  transition_sum <- transition_sum + alpha[i, t - 1] * A_ik
  }
      b_kt <- emission_probability(hap[t], haps[k, t], error)
      alpha[k, t] <- transition_sum * b_kt
    }
  }

  return(alpha)
}
```

2.2 Unit Test for `forward` Algorithm Implementation

```
test_that("alpha matrix has expected form when haps and hap always match", {
  K <- 10  # Number of rows in haps
  T <- 15  # Number of columns in haps = length of hap
  e <- 0.1  # Error rate (= default value)
```

```
  # Create haps and hap such that they always match (both all 0)
  haps <- matrix(0, nrow = K, ncol = T)
  hap <- rep(0, T)

  # Run the forward function
  alpha <- forward(haps, hap, error = e)

  # Check first column
  expected_first_column <- rep((1 - e) / K, K)
  expect_equal(alpha[, 1], expected_first_column)

  # Check all other columns
  for (t in 2:T) {
    expect_equal(alpha[, t], alpha[, t - 1] * (1 - e), tolerance = 1e-5)
  }
})

## Test passed
```

2.3 Implementation of `backward` Algorithm

```
backward <- function(haps, hap, error = 0.1) {
  K <- nrow(haps)
  T <- ncol(haps)

  # Initialize beta matrix (eq. 6)
  beta <- matrix(0, nrow = K, ncol = T)
  beta[, T] <- 1  # Set last column to 1

  # Induction step (eq. 7)
  for (t in (T-1):1) {
      for (k in 1:K) {
        sum_transition <- 0
        for (i in 1:K) { # (eq. 2)
          A_ki <- ifelse(k == i, (1 - 0.999) / K + 0.999, (1 - 0.999) / K)
          b_i_t_plus_1 <- emission_probability(hap[t+1], haps[i, t+1], error)
          sum_transition <- sum_transition +
              A_ki * b_i_t_plus_1 * beta[i, t+1]
        }
        beta[k, t] <- sum_transition
      }
    }

  return(beta)
}
```

2.4 Implementation of gamma Function

```
gamma <- function(haps, hap, error = 0.1) {
  K <- nrow(haps)
  T <- ncol(haps)

  # Compute alpha and beta matrices
  alpha <- forward(haps, hap, error)
  beta <- backward(haps, hap, error)

  # Compute normalization factor (denominator)
  norm_factor <- sum(alpha[, T])

  # Initialize gamma matrix
  gamma_matrix <- matrix(0, nrow = K, ncol = T)

  # Update gamma values (eq. 8)
  for (t in 1:T) {
    for (k in 1:K) {
      gamma_matrix[k, t] <- (alpha[k, t] * beta[k, t]) / norm_factor
    }
  }
  return(gamma_matrix)
}
```

2.5 Unit Test for gamma Function Implementation

```
test_that("column sums of gamma matrix all equal 1", {
  set.seed(42)  # For reproducibility
  K <- 10  # Number of rows in haps
  T <- 15  # Number of columns in haps = length of hap
  e <- 0.1  # Error rate (= default value)

  # Create random haps and hap with 0 or 1 entries
  haps <- matrix(sample(0:1, K * T, replace = TRUE), nrow = K, ncol = T)
  hap <- sample(0:1, T, replace = TRUE)

  # Run the gamma function
  gamma_matrix <- gamma(haps, hap, error = e)

  # Check that each column sum is close to 1
  for (t in 1:T) {
    expect_equal(sum(gamma_matrix[, t]), 1, tolerance = 1e-5)
  }
})

## Test passed
```

2.6 Computational Complexity of `forward` and `backward` Algorithms

1. The `forward` algorithm has a time complexity of $\mathcal{O}(K^2 \cdot T)$ as for each time step t, it iterates over K states, and within each state again iterates over K states to compute the transition probabilities.

2. The `backward` algorithm has an identical time complexity of $\mathcal{O}(K^2 \cdot T)$ as it performs the same iterations, just in a different oder.

Figures 4 and 5 below show benchmark results for both the forward and backward algorithms, executed 10 times each for all $K \times T$ with $K, L = 5, \ldots, 20$. The fitted lines for the median execution time for each value of K and T, respectively, look quadratic in K and linear in T.

Figure 4: Benchmark Results for `forward` Algorithm

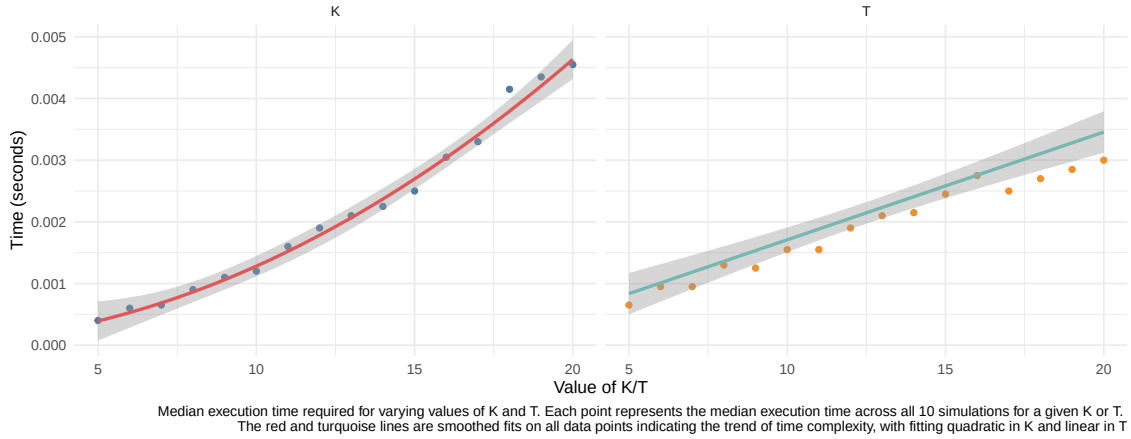
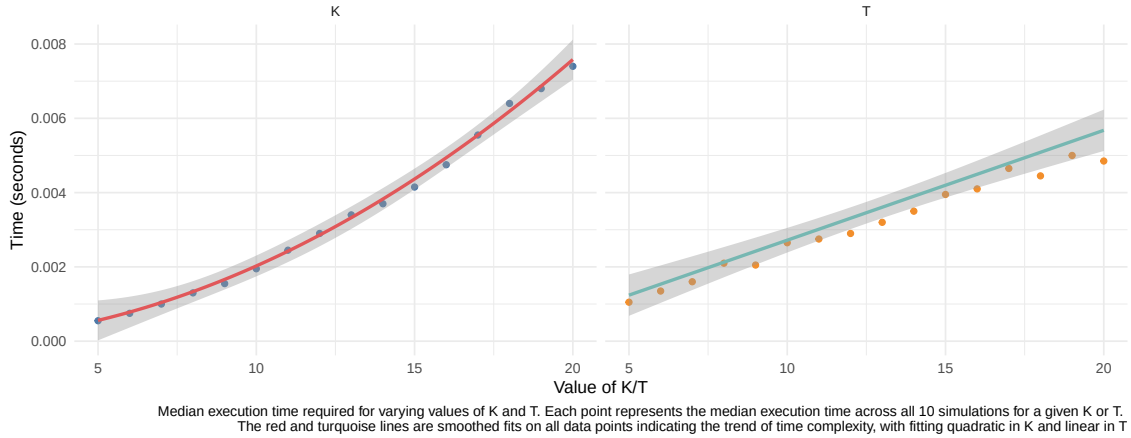


Figure 5: Benchmark Results for `backward` Algorithm



2.7 Implementation of `forward2` Algorithm

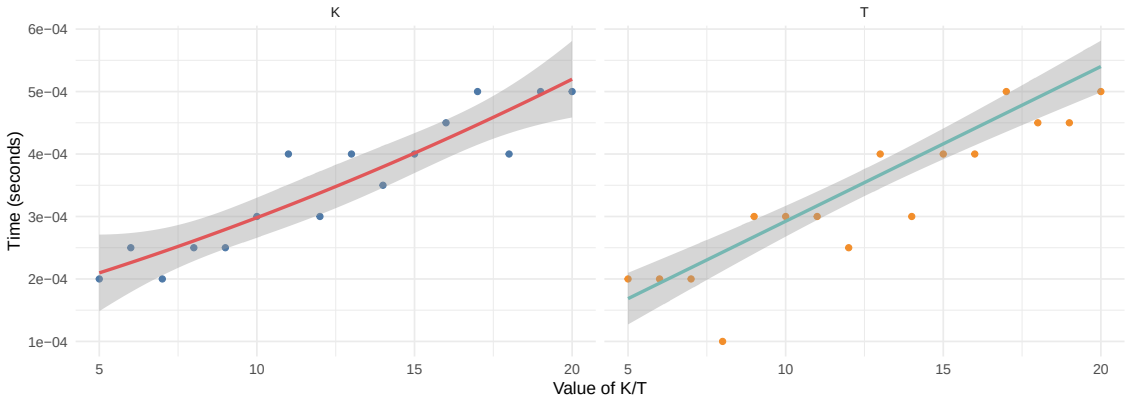
```r
forward2 <- function(haps, hap, error = 0.1) {
  K <- nrow(haps)
  T <- ncol(haps)
  alpha <- matrix(0, nrow = K, ncol = T)
  pi <- 1 / K

  for (k in 1:K) {
    alpha[k, 1] <- pi * emission_probability(hap[1], haps[k, 1], error)
  }

  for (t in 2:T) {
    phi <- sum(alpha[, t - 1]) * (1 - 0.999) / K
    for (k in 1:K) {
      A_ik <- ifelse(k == k, (1 - 0.999) / K + 0.999, (1 - 0.999) / K)
      b_kt <- emission_probability(hap[t], haps[k, t], error)
      alpha[k, t] <- (phi + 0.999 * alpha[k, t - 1]) * b_kt
    }
  }

  return(alpha)
}
```

```r
test_that("forward and forward2 produce the same outputs", {
  set.seed(42)  # For reproducibility
  K <- 10  # Number of rows in haps
  T <- 15  # Number of columns in haps = length of hap
  e <- 0.1  # Error rate (= default value)

  # Generate example data
  haps <- matrix(sample(0:1, K * T, replace = TRUE), nrow = K)
  hap <- sample(0:1, T, replace = TRUE)

  # Run both functions
  alpha_forward <- forward(haps, hap, e)
  alpha_forward2 <- forward2(haps, hap, e)

  # Check if the outputs are equal
  expect_equal(alpha_forward, alpha_forward2, tolerance = 1e-5)
})

## Test passed
```

The `forward2` algorithm has a time complexity of $\mathcal{O}(K \cdot T)$ as it computes the initial probabilities ($\mathcal{O}(K)$) and then for each time step t, it sums over all K states ($\mathcal{O}(K)$) and computes equation (14) ($\mathcal{O}(1)$). Since the main loop runs for each time step T, and within each time step, two $\mathcal{O}(K)$ operations are performed (one for summing alpha and one for

the loop over K), the overall complexity of the forward2 function is $\mathcal{O}(K \cdot T)$. Figure 6 below displays this improved performance.

Figure 6: Benchmark Results for `forward2` Algorithm



Median execution time required for varying values of K and T. Each point represents the median execution time across all 10 simulations for a given K or T. The red and turquoise lines are smoothed fits on all data points indicating the trend of time complexity, with fitting quadratic in K and linear in T

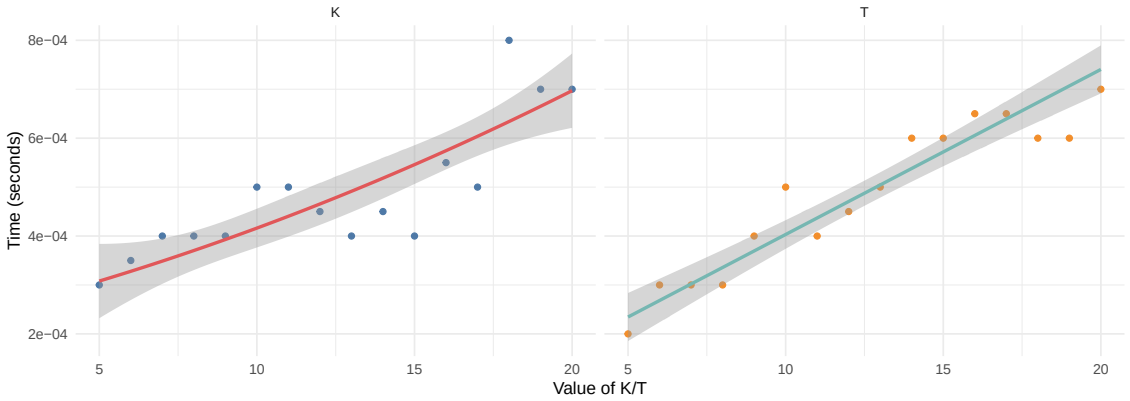
2.8 Implementation of `backward2` Algorithm

```r
backward2 <- function(haps, hap, error = 0.1) {
  K <- nrow(haps)
  T <- ncol(haps)

  # Initialize beta matrix (eq. 6)
  beta <- matrix(0, nrow = K, ncol = T)
  beta[, T] <- 1  # Set last column to 1

  # Induction step (eq. 14)
  for (t in (T-1):1) {
    # Calculate the phi term which is constant for all k (eq. 13)
    phi <- sum(beta[, t+1] *
                 sapply(1:K, function(i) emission_probability(hap[t+1],
                                haps[i, t+1], error))) * (1 - 0.999) / K

    for (k in 1:K) { # (eq. 14)
      beta[k, t] <- phi + 0.999 * beta[k, t+1] *
        emission_probability(hap[t+1], haps[k, t+1], error)
    }
  }

  return(beta)
}
```

```r
test_that("backward and backward2 produce the same outputs", {
  set.seed(42)  # For reproducibility
  K <- 10  # Number of rows in haps
  T <- 15  # Number of columns in haps = length of hap
  e <- 0.1  # Error rate (= default value)

  # Generate example data
  haps <- matrix(sample(0:1, K * T, replace = TRUE), nrow = K)
  hap <- sample(0:1, T, replace = TRUE)

  # Run both functions
  beta_backward <- backward(haps, hap, e)
  beta_backward2 <- backward2(haps, hap, e)

  # Check if the outputs are equal
  expect_equal(beta_backward, beta_backward2, tolerance = 1e-5)
})

## Test passed
```

The `backward2` algorithm has a time complexity of $\mathcal{O}(K \cdot T)$ as it computes the initial probabilities ($\mathcal{O}(K)$) and then for each time step t, it sums over all K states ($\mathcal{O}(K)$) and computes equation (14) ($\mathcal{O}(1)$). Since the main loop runs for each time step T, and within each time step, two $\mathcal{O}(K)$ operations are performed (one for summing alpha and one for the loop over K), the overall complexity of the forward2 function is $\mathcal{O}(K \cdot T)$. Figure 7 below displays this improved performance.

Figure 7: Benchmark Results for `backward2` Algorithm



Median execution time required for varying values of K and T. Each point represents the median execution time across all 10 simulations for a given K or T. The red and turquoise lines are smoothed fits on all data points indicating the trend of time complexity, with fitting quadratic in K and linear in T

2.9 Implementation of `gamma2` Function

```r
gamma2 <- function(haps, hap, error = 0.1) {
  K <- nrow(haps)
  T <- ncol(haps)
```

```r
  # Compute alpha and beta matrices using forward2 and backward2
  alpha <- forward2(haps, hap, error)
  beta <- backward2(haps, hap, error)

  # Compute normalization factor (denominator)
  norm_factor <- sum(alpha[, T])

  # Initialize gamma matrix
  gamma_matrix <- matrix(0, nrow = K, ncol = T)

  # Update gamma values (eq. 8)
  for (t in 1:T) {
    for (k in 1:K) {
      gamma_matrix[k, t] <- (alpha[k, t] * beta[k, t]) / norm_factor
    }
  }

  return(gamma_matrix)
}
```

```r
test_that("gamma and gamma2 produce the same outputs", {
  set.seed(42)  # For reproducibility
  K <- 10  # Number of rows in haps
  T <- 15  # Number of columns in haps = length of hap
  e <- 0.1  # Error rate (= default value)

  # Generate example data
  haps <- matrix(sample(0:1, K * T, replace = TRUE), nrow = K)
  hap <- sample(0:1, T, replace = TRUE)

  # Run both functions
  gamma_matrix <- gamma(haps, hap, e)
  gamma_matrix2 <- gamma2(haps, hap, e)

  # Check if the outputs are equal
  expect_equal(gamma_matrix, gamma_matrix2, tolerance = 1e-5)
})

## Test passed
```

2.10 Computational Complexity of `gamma` and `gamma2`

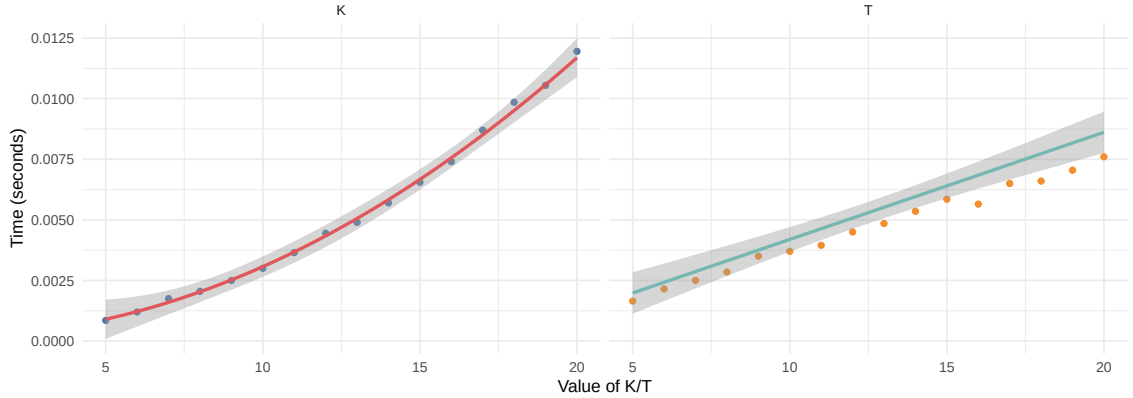
1. Based on the previous discussion, executing `forward2` and `backward2` each costs $\mathcal{O}(K \cdot T)$. Computing the normalization factor involves summing over one column of the `alpha` matrix, which is $\mathcal{O}(K)$. The nested loop structure for updating the `gamma` matrix involves iterating over T and each K. Within the inner loop, the operation to calculate each entry is $\mathcal{O}(1)$. Therefore, the complexity of updating the gamma

matrix is $\mathcal{O}(K \cdot T)$. With 3 operations in $\mathcal{O}(K \cdot T)$, the overall time complexity of `gamma2` is $\mathcal{O}(K \cdot T)$.

2. By the same logic, executing `forward` and `backward` each costs $\mathcal{O}(K^2 \cdot T)$, which dominates the other terms that contribute to the time complexity of `gamma`, which as an overall time complexity of $\mathcal{O}(K^2 \cdot T)$

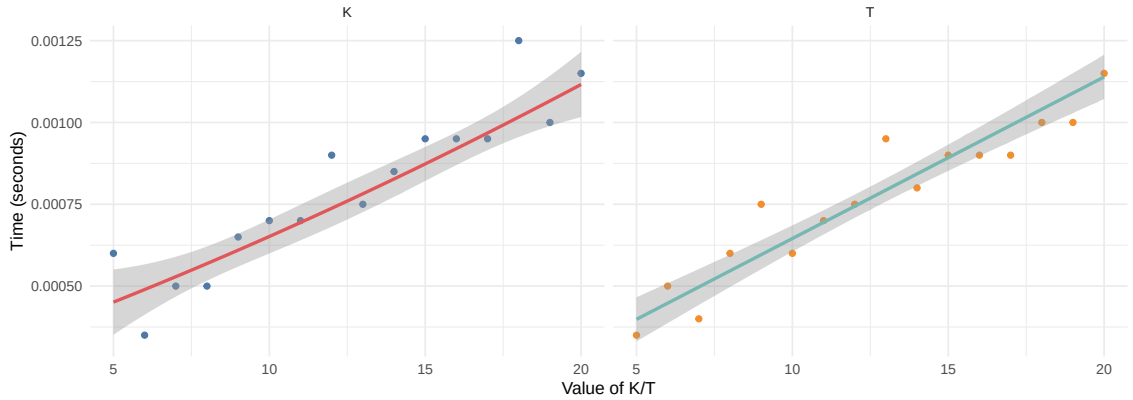
Figures 8 and 9 below show that the empirical time complexity of `gamma` and `gamma2` indeed matches the reasoning laid out above.

Figure 8: Benchmark Results for `gamma` Function



Median execution time required for varying values of K and T. Each point represents the median execution time across all 10 simulations for a given K or T. The red and turquoise lines are smoothed fits on all data points indicating the trend of time complexity, with fitting quadratic in K and linear in T

Figure 9: Benchmark Results for `gamma2` Function



Median execution time required for varying values of K and T. Each point represents the median execution time across all 10 simulations for a given K or T. The red and turquoise lines are smoothed fits on all data points indicating the trend of time complexity, with fitting quadratic in K and linear in T

```r
# Extract the first 5 target haplotypes
first_5_haps <- samples_raw[1:5, ]

paint_and_sum <- function(target_hap, ref_panel) {
  gamma_results <- gamma2(ref_panel, target_hap, error = 0.1)

  # Sum contributions from YRI and CEU haplotypes
  yri_sum <- rowSums(gamma_results[grepl("YRI", rownames(ref_panel)), ])
  ceu_sum <- rowSums(gamma_results[grepl("CEU", rownames(ref_panel)), ])

  return(list(YRI = yri_sum, CEU = ceu_sum))
}

# Apply function to first 5 target haplotypes (parallelized)
first_5_painted <- mclapply(1:nrow(first_5_haps), function(i) {
  paint_and_sum(first_5_haps[i, ], refpanel_raw)
}, mc.cores = detectCores())

compile_results_matrix <- function(painted_results) {
  results_matrix <- matrix(nrow = length(painted_results), ncol = 3)
  colnames(results_matrix) <- c("Sum_YRI", "Sum_CEU", "Percentage_YRI")
  rownames(results_matrix) <- rownames(first_5_haps)
  # Loop through each list of results and calculate the sum of YRI, CEU,
  # percentage of YRI
  for (i in seq_along(painted_results)) {
    sum_yri <- sum(painted_results[[i]]$YRI)
    sum_ceu <- sum(painted_results[[i]]$CEU)
    total_sum <- sum_yri + sum_ceu
    percentage_yri <- (sum_yri / total_sum) * 100

    results_matrix[i, ] <- c(sum_yri, sum_ceu, percentage_yri)
  }

  colnames(results_matrix) <- c("Sum YRI", "Sum CEU",
                                "Percentage of YRI in Total")
  return(results_matrix)
}

painted <- compile_results_matrix(first_5_painted)
```

Table 2 below shows that among the first 5 target haplotypes, haplotypes 2 and 5 show a high contribution of YRI haplotypes. Target haplotype 5 suggests an entirely African genetic background over the investigated chromosome.

Table 2: Sums and Proportion of YRI and CEU Ancestry in the first 5 Target Haplotypes

	Sum YRI	Sum CEU	Percentage of YRI in Total
ASW-NA19818	46.12	70.88	39.42
ASW-NA19819	98.58	18.42	84.26
ASW-NA19834	30.18	86.82	25.80
ASW-NA19835	71.69	45.31	61.27
ASW-NA19900	116.46	0.54	99.54