```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MATRIX_SIZE 3
#define MAX_ITERATIONS 1000
#define TOLERANCE 1e-6

void print_matrix(double* matrix) {
  for (int i = 0; i < MATRIX_SIZE; i++) {
    for (int j = 0; j < MATRIX_SIZE; j++) {
      printf("%f ", matrix[i*MATRIX_SIZE+j]);
    }
    printf("\n");
  }
}

void print_vector(double* vector) {
  for (int i = 0; i < MATRIX_SIZE; i++) {
    printf("%f ", vector[i]);
  }
  printf("\n");
}

void multiply_matrix_vector(double* matrix, double* vector,
double* result) {
  for (int i = 0; i < MATRIX_SIZE; i++) {
    double sum = 0.0;
    for (int j = 0; j < MATRIX_SIZE; j++) {
      sum += matrix[i*MATRIX_SIZE+j] * vector[j];
    }
    result[i] = sum;
  }
}

double normalize_vector(double* vector) {
  double norm = 0.0;
  for (int i = 0; i < MATRIX_SIZE; i++) {
```

```
    norm += vector[i] * vector[i];
  }
  norm = sqrt(norm);
  for (int i = 0; i < MATRIX_SIZE; i++) {
    vector[i] /= norm;
  }
  return norm;
}

int main(int argc, char** argv) {
  double matrix[MATRIX_SIZE*MATRIX_SIZE] = {
    4.0, 2.0, 1.0,
    2.0, 5.0, 3.0,
    1.0, 3.0, 6.0
  };
  double vector[MATRIX_SIZE] = { 1.0, 1.0, 1.0 };
  double result[MATRIX_SIZE];
  double lambda = 0.0;
  int iterations = 0;

  printf("Matrix:\n");
  print_matrix(matrix);
  printf("\n");

  printf("Starting vector:\n");
  print_vector(vector);
  printf("\n");

  // Power method
  while (iterations < MAX_ITERATIONS) {
    multiply_matrix_vector(matrix, vector, result);
    lambda = normalize_vector(result);
    if (fabs(lambda - normalize_vector(vector)) < TOLERANCE)
{
      break;
    }
    for (int i = 0; i < MATRIX_SIZE; i++) {
      vector[i] = result[i];
```

```
    }
    iterations++;
  }

  printf("Dominant eigenvalue: %f\n", lambda);
  printf("Eigenvector:\n");
  print_vector(result);
  printf("\n");

  return 0;
}
```

```
gcc power_method.c -lm -o power_method
./power_method
```

```
Matrix:
4.000000 2.000000 1.000000
2.000000 5.000000 3.000000
1.000000 3.000000 6.000000

Starting vector:
1.000000 1.000000 1.000000

Dominant eigenvalue: 7.527735
Eigenvector:
0.293303 0.564063 0.771424
```

```
gcc -pg mycode.c -o mycode
./mycode
gprof mycode gmon.out > analysis.txt

Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 25.00      0.01     0.01        1    10.00    10.00  myfunc1
 25.00      0.02     0.01        1    10.00    10.00  myfunc2
```

```
 50.00      0.03     0.02        1    20.00    20.00  main
  0.00      0.03     0.00        1     0.00     0.00
__libc_csu_init
  0.00      0.03     0.00        1     0.00     0.00  _start
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

#define MATRIX_SIZE 1000
#define MAX_ITERATIONS 1000
#define TOLERANCE 1e-6

void print_matrix(double* matrix) {
  for (int i = 0; i < MATRIX_SIZE; i++) {
    for (int j = 0; j < MATRIX_SIZE; j++) {
      printf("%f ", matrix[i*MATRIX_SIZE+j]);
    }
    printf("\n");
  }
}

void print_vector(double* vector) {
  for (int i = 0; i < MATRIX_SIZE; i++) {
    printf("%f ", vector[i]);
  }
  printf("\n");
}

void multiply_matrix_vector(double* matrix, double* vector,
double* result) {
  #pragma omp parallel for
  for (int i = 0; i < MATRIX_SIZE; i++) {
    double sum = 0.0;
    for (int j = 0; j < MATRIX_SIZE; j++) {
      sum += matrix[i*MATRIX_SIZE+j] * vector[j];
```

```
    }
    result[i] = sum;
  }
}

double normalize_vector(double* vector) {
  double norm = 0.0;
  #pragma omp parallel for reduction(+:norm)
  for (int i = 0; i < MATRIX_SIZE; i++) {
    norm += vector[i] * vector[i];
  }
  norm = sqrt(norm);
  #pragma omp parallel for
  for (int i = 0; i < MATRIX_SIZE; i++) {
    vector[i] /= norm;
  }
  return norm;
}

int main(int argc, char** argv) {
  double matrix[MATRIX_SIZE*MATRIX_SIZE];
  double vector[MATRIX_SIZE];
  double result[MATRIX_SIZE];
  double lambda = 0.0;
  int iterations = 0;

  // Initialize matrix and vector
  for (int i = 0; i < MATRIX_SIZE; i++) {
    for (int j = 0; j < MATRIX_SIZE; j++) {
      matrix[i*MATRIX_SIZE+j] = 1.0 / (i+j+1);
    }
    vector[i] = 1.0;
  }

  printf("Matrix:\n");
  print_matrix(matrix);
  printf("\n");
```

```
  printf("Starting vector:\n");
  print_vector(vector);
  printf("\n");

  // Power method
  while (iterations < MAX_ITERATIONS) {
    multiply_matrix_vector(matrix, vector, result);
    lambda = normalize_vector(result);
    if (fabs(lambda - normalize_vector(vector)) < TOLERANCE)
{
      break;
    }
    for (int i = 0; i < MATRIX_SIZE; i++) {
      vector[i] = result[i];
    }
    iterations++;
  }

  printf("Eigenvalue: %f\n", lambda);
  printf("Eigenvector:\n");
  print_vector(vector);
  printf("\n");
  printf("Number of iterations: %d\n", iterations);

  return 0;
}

gcc -fopenmp openmp_eigen.c -o openmp_eigen -lm
./openmp_eigen

Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 25.00      0.01     0.01          1    10.00    10.00
multiply_matrix_vector
```

```
 25.00      0.02     0.01          1    10.00    10.00
normalize_vector
 25.00      0.03     0.01          1    10.00    10.00  main
  0.00      0.03     0.00          1     0.00     0.00
__libc_csu_init
  0.00      0.03     0.00          1     0.00     0.00  _start
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

#define MATRIX_SIZE 6
#define MAX_ITERATIONS 1000
#define TOLERANCE 1e-6

void print_matrix(double* matrix) {
  for (int i = 0; i < MATRIX_SIZE; i++) {
    for (int j = 0; j < MATRIX_SIZE; j++) {
      printf("%f ", matrix[i*MATRIX_SIZE+j]);
    }
    printf("\n");
  }
}

void print_vector(double* vector) {
  for (int i = 0; i < MATRIX_SIZE; i++) {
    printf("%f ", vector[i]);
  }
  printf("\n");
}
```

```
void multiply_matrix_vector(double* matrix, double* vector,
double* result) {
  #pragma omp parallel for
  for (int i = 0; i < MATRIX_SIZE; i++) {
    double sum = 0.0;
    for (int j = 0; j < MATRIX_SIZE; j++) {
      sum += matrix[i*MATRIX_SIZE+j] * vector[j];
    }
    result[i] = sum;
  }
}

double normalize_vector(double* vector) {
  double norm = 0.0;
  #pragma omp parallel for reduction(+:norm)
  for (int i = 0; i < MATRIX_SIZE; i++) {
    norm += vector[i] * vector[i];
  }
  norm = sqrt(norm);
  #pragma omp parallel for
  for (int i = 0; i < MATRIX_SIZE; i++) {
    vector[i] /= norm;
  }
  return norm;
}

int main(int argc, char** argv) {
  double matrix[MATRIX_SIZE*MATRIX_SIZE];
  double vector[MATRIX_SIZE];
  double result[MATRIX_SIZE];
  double lambda = 0.0;
  int iterations = 0;

  // Initialize matrix with random values
  srand(time(NULL));
  for (int i = 0; i < MATRIX_SIZE; i++) {
    for (int j = 0; j < MATRIX_SIZE; j++) {
      matrix[i*MATRIX_SIZE+j] = rand() / (double)RAND_MAX;
```

```
    }
  }

  printf("Matrix:\n");
  print_matrix(matrix);
  printf("\n");

  // Initialize vector to all ones
  for (int i = 0; i < MATRIX_SIZE; i++) {
    vector[i] = 1.0;
  }

  printf("Starting vector:\n");
  print_vector(vector);
  printf("\n");

  // Power method
  while (iterations < MAX_ITERATIONS) {
    multiply_matrix_vector(matrix, vector, result);
    lambda = normalize_vector(result);
    if (fabs(lambda - normalize_vector(vector)) < TOLERANCE)
{
      break;
    }
    for (int i = 0; i < MATRIX_SIZE; i++) {
      vector[i] = result[i];
    }
    iterations++;
  }

  printf("Dominant eigenvalue: %f\n", lambda);
  printf("Eigenvector:\n");
  print_vector(vector);
  printf("\n");
  printf("Number of iterations: %d\n", iterations);

  return 0;
}
```

```
gcc -fopenmp openmp_random_matrix.c -o openmp_random_matrix
-lm
./openmp_random_matrix
```