Exercise 5

Power Method, BLAS/LAPACK, OpenMP

Solutions review & feedback

Question 1a: Power method

```
#include <random>

// alpha: Diagonal scaling factor
// A:     N x N Matrix
// N:     Matrix dimension
void initialize_matrix(const double alpha,
                       double* const A, const int N)
{
   // random generator with seed 0
   std::default_random_engine g(0);
   // uniform distribution in [0, 1]
   std::uniform_real_distribution<double> u;

    // matrix initialization
    for (int i = 0; i < N; i++) {
        for (int j = i+1; j < N; j++) {
            const double rand = u(g);
            A[i*N + j] = rand;
            A[j*N + i] = rand;
        }
        A[i*N + i] = (i+1.0) * alpha;
    }
}
```

1. Symmetric matrix, A:
   A[i,j] = A[j,i]

Question 1a: Power method

```
...
auto tstart = std::chrono::steady_clock::now();
// Main algorithm:
// You can re-structure the Power Method algorithm such that you only
// perform one GEMV operation in the iteration loop.  By doing so, you have
// to use a pointer swap at the end of the loop (which is very cheap to
// do).
_gemv(N, N, A, q0, q1);
while (true)
{
    _norm(N, q1);
    _gemv(N, N, A, q1, q0);
    lambda1 = 0.0;
    for (int i = 0; i < N; ++i)
        // estimate eigenvalue with Rayleigh quotient
        lambda1 += q0[i] * q1[i];
    ++iter;
    if (std::abs(lambda1 - lambda0) < tol)
        break;
    lambda0 = lambda1;
    std::swap(q0, q1);
}
// end of algorithm
auto tend = std::chrono::steady_clock::now();
auto time = std::chrono::duration_cast<std::chrono::milliseconds>(tend -
    tstart).count();
...
```

1. Timing should include first `gemv` since it is part of the algorithm
2. Single `gemv` in iteration loop
3. No need to repeat `gemv` calculation to compute eigenvalue!

$$\mathbf{q}^{(k+1)} = \frac{A\mathbf{q}^{(k)}}{\|A\mathbf{q}^{(k)}\|_2}$$

$$\lambda = \frac{\mathbf{q}^T A\mathbf{q}}{\mathbf{q}^T\mathbf{q}}$$

4. Use pointer swap to update eigenvectors (or equivalent design)

Question 1b+c: Power method (CBLAS, LAPACK)

CBLAS:

```
...
auto tstart = std::chrono::steady_clock::now();
// Main algorithm:
// You can re-structure the Power Method algorithm such that you only
// perform one GEMV operation in the iteration loop.  By doing so, you have
// to use a pointer swap at the end of the loop (which is very cheap to
// do).
cblas_dsymv(CblasRowMajor, CblasUpper , N, 1.0, A, N, q0, 1, 0.0, q1, 1);
while (true)
{
    const double norm = cblas_dnrm2(N, q1, 1);
    cblas_dscal(N, 1./norm, q1, 1);
    cblas_dsymv(CblasRowMajor, CblasUpper , N, 1.0, A, N, q1, 1, 0.0, q0, 1);
    lambda1 = cblas_ddot(N, q0, 1, q1, 1);
    ++iter;
    if (std::abs(lambda1 - lambda0) < tol)
        break;
    lambda0 = lambda1;
    std::swap(q0, q1);
}
// end of algorithm
auto tend = std::chrono::steady_clock::now();
auto time = std::chrono::duration_cast<std::chrono::milliseconds>(tend -
    tstart).count();
...
```

1. Same matrix-vector and vector operations, but replaced by CBLAS function calls
2. Lecture slides: usage example for CBLAS function calls
3. Use of `std::swap(…)` instead of `cblas_dcopy(…)`

LAPACK:

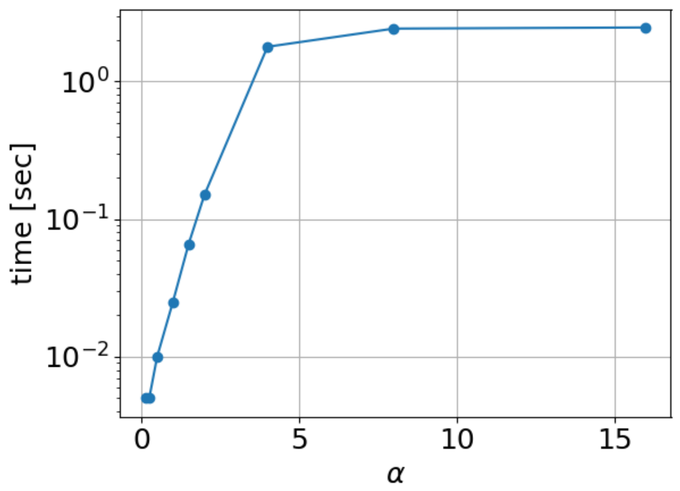
```
...

auto tstart = std::chrono::steady_clock::now();
int info = LAPACKE_dsyev(LAPACK_ROW_MAJOR, jobz, uplo, N, A, N, lambdas);
auto tend = std::chrono::steady_clock::now();
auto time = std::chrono::duration_cast<std::chrono::milliseconds>(tend -
    tstart).count();
...
```

Single function call

Question 1b+c: Power method (CBLAS, LAPACK)

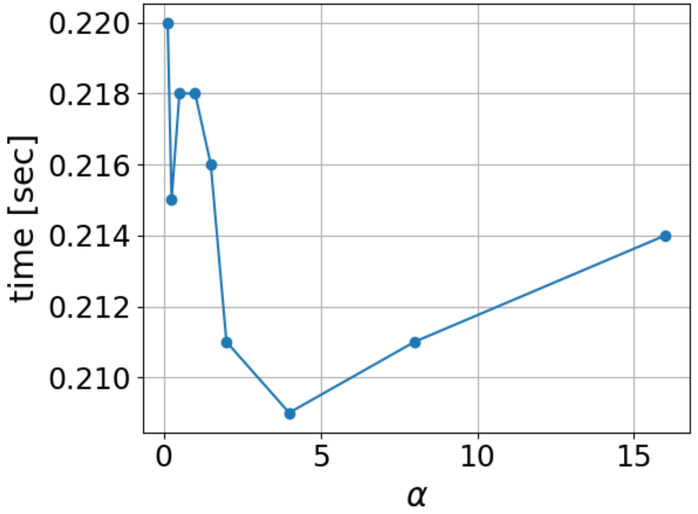
CBLAS

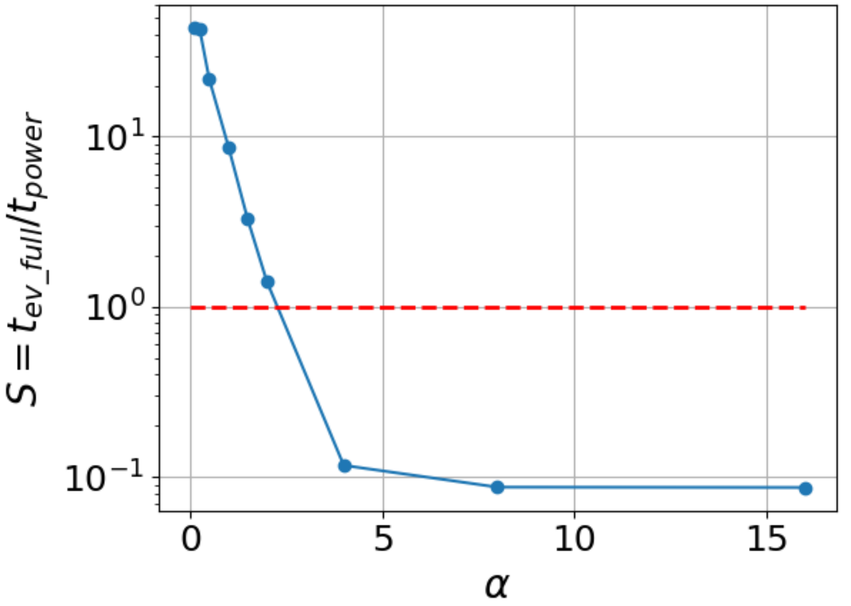
difference in time-to-solution spans *2 orders of magnitude*



LAPACK

time-to-solution *invariant of α*

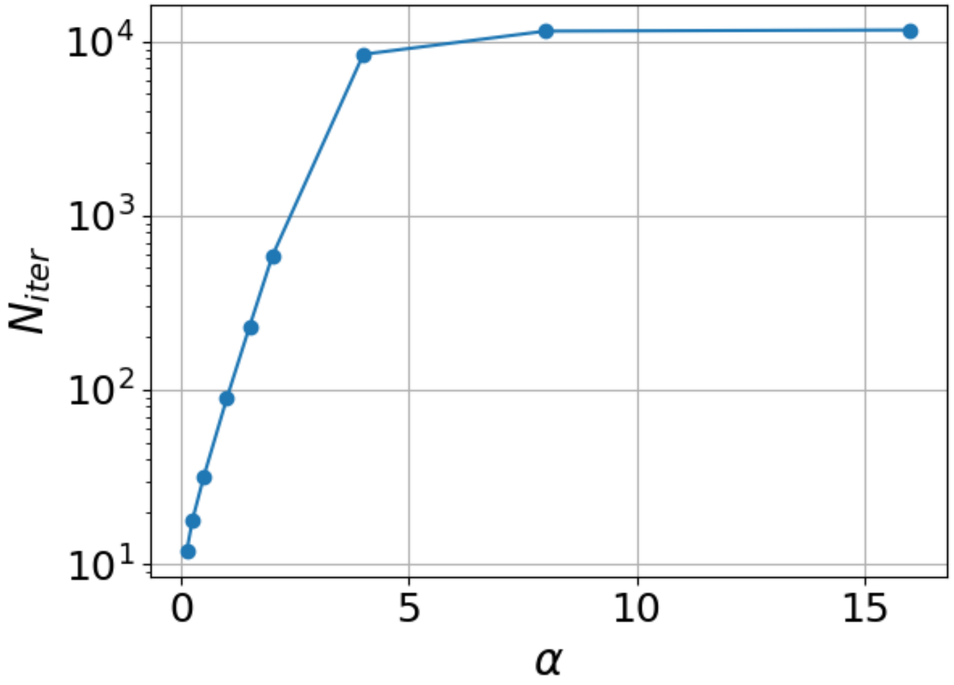




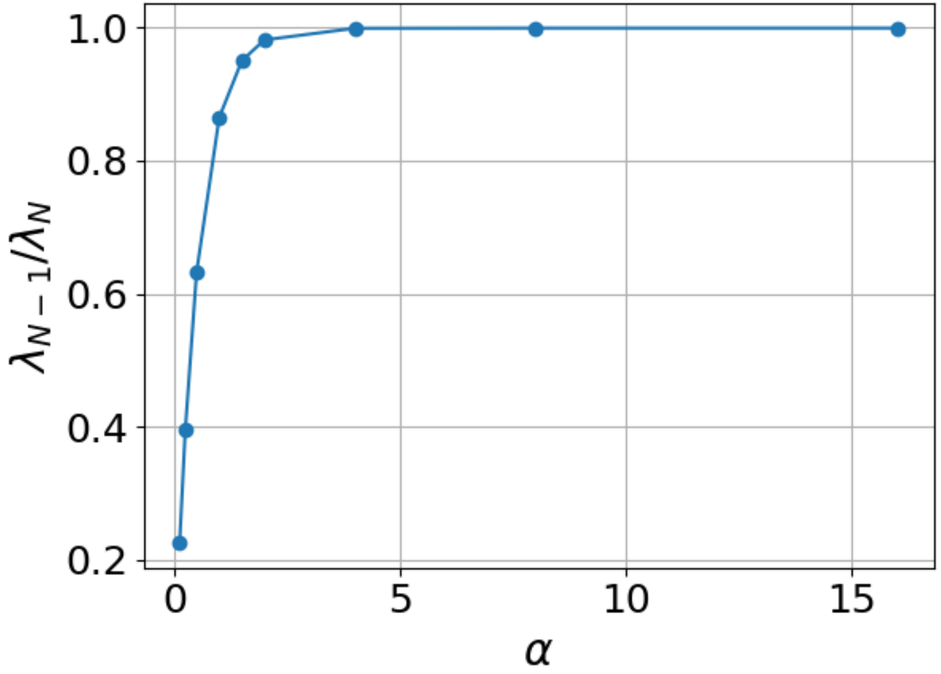
- Power method: iterative algorithm
- Might not always be the best choice - this depends on the properties of the matrix we are interested in!

Question 1d: Convergence of the power method

Number of iterations



Eigenvalue ratio



Question 2: OpenMP bug hunts

```
...
10  #pragma omp parallel
11  {
12
13      for (int step=0; step<100; step++)
14      {
15          #pragma omp parallel for nowait
16          for (int i=1; i<n; i++) {
17              b[i-1] = (a[i]+a[i-1])/2.;
18              c[i-1] += a[i];
19          }
20
21          #pragma omp for
22          for (int i=0; i<m; i++)
23              z[i] = sqrt(b[i]+c[i]);
24
25          #pragma omp for reduction(+:sum)
26          for (int i=0; i<m; i++)
27              sum = sum + z[i];
28
29          #pragma omp critical
30          {
31              do_work(t, sum);
32          }
33
34          #pragma omp single
35          {
36              t = new_value(step);
37          }
38      }
39  }
...
```

Remove both! ALL

Same iteration space! Combination in single for loop to save the overhead of one `#pragma omp for`

`critical` is unnecessary. Remove and add a barrier after function `do_work`.

Solution 1

Solution 2

Explicitly state the assumption that `do_work` can be executed by a single thread since it's arguments are constant and the function computes a total sum. In this case `critical` should be replaced with `single`.

See solution pdf for full solution of Question 2

General remarks

- Avoid copy-pasting
- Do submit your codes along with pdf solutions and results
- Collect results (text, figures, theoretical proofs in a pdf file)
- If in doubt, google or ask on piazza!

e.g.

power_manual.cpp

power_cblas.cpp

eigenv_lapack.cpp

```cpp
#include "common.h"

...

// matrix initialization
double* A = new double[N*N];
initialize_matrix(alpha, A, N);
```

common.h

```cpp
void initialize_matrix(const double alpha,
                       double* const A,
                       const int N);
```