



F. Wermelinger
Office: Pierce Hall 211

Lab 5

Eigenvalues, power method and external libraries

Issued: March 27, 2023
Due: April 7, 2023 11:59pm

In this lab we are computing eigenvalues using the power method¹. We are further interested in comparing the performance of our power method implementation with CBLAS and Eigen² reference kernels.

Submission instructions: please see the lab submission section in the syllabus.^a

^a<https://harvard-iacs.github.io/2023-CS205/pages/syllabus.html#lab-submission>

Task 1: Power Method

The power method is an iterative technique for locating the dominant (largest) eigenvalue of a matrix. In addition, the power method also computes the associated eigenvector.

Consider the symmetric $N \times N$ matrix A , where the diagonal elements are given by $a_{ii} = \alpha i$ for $i = 1, 2, \dots, N$ and the off-diagonal elements are random numbers drawn from a uniform distribution $\mathcal{U}_{[0,1]}$ where $a_{ij} = a_{ji}$ for all $i \neq j$. We consider the following values for $\alpha \in \{\frac{1}{8}, \frac{1}{4}, \frac{1}{2}, 1, \frac{3}{2}, 2, 4, 8, 16\}$ and $N = 1024$. Additionally, all results should be computed in double precision.

The power method produces a sequence of column vectors $\mathbf{q}^{(k)} \in \mathbb{R}^{N \times 1}$ given by

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

If $\mathbf{q}^{(0)}$ is not deficient and the largest eigenvalue of A is unique, then $\mathbf{q}^{(k)}$ will converge to an eigenvector with eigenvalue $\lambda^{(k)}$. The eigenvalue can be obtained using the Rayleigh quotient³

$$\lambda^{(k)} = \frac{(\mathbf{q}^{(k)})^\top A \mathbf{q}^{(k)}}{(\mathbf{q}^{(k)})^\top \mathbf{q}^{(k)}}. \quad (2)$$

¹https://en.wikipedia.org/wiki/Power_iteration

²<https://eigen.tuxfamily.org/index.php>

³https://en.wikipedia.org/wiki/Rayleigh_quotient

Note that the eigenvalues you compute in the following sub-tasks must always be the same, you just employ different approaches to compute them.

- a) Follow the TODO labels in the file `code/t1/src/power_manual.cpp` and implement your own matrix multiplication program to calculate the dominant eigenvalue of the matrix A using the power method. Stop at the k -th iteration if the condition $|\lambda^{(k)} - \lambda^{(k-1)}| < 10^{-12}$ is satisfied. Use $\mathbf{q}^{(0)} = [1, 0, 0, \dots]^T$ for the initial guess.

The code will generate a file `manual.dat` which you can use for plotting. Create a plot `eigenvalues.png` for the computed eigenvalues $\lambda^{(k)}$ on the ordinate and the values α on the abscissa. Create a second plot `iterations.png` for the number of iterations on the ordinate (use a logarithmic scale for this axis) and α on the abscissa. Use $\alpha \in \{\frac{1}{8}, \frac{1}{4}, \frac{1}{2}, 1, \frac{3}{2}, 2, 4, 8, 16\}$ and $N = 1024$ for all your experiments. Include these two plots in your final submission.

You can build your code with the provided `Makefile` by running

```
$ make power_manual
```

after you have loaded a recent compiler, for example

```
$ module load gcc/12.1.0-fasrc01
```

You can execute your code by specifying the dimension N and one or more values for α . For example, a problem size of $N = 1024$ and two values $\alpha = \frac{1}{8}$ and $\alpha = \frac{1}{4}$:

```
$ ./power_manual 1024 0.125 0.25
```

- b) Manual implementation of common compute kernels like the matrix-vector multiplication in the power method is often a bad idea because it is very likely that optimized implementations exist in external libraries. Such common kernels for linear algebra operations are provided in the BLAS⁴ library. The goal in this part of the lab is to implement another version of the power method using the C API of the BLAS library. Follow the TODO labels in the file `code/t1/src/power_cblas.cpp` and implement the power method using CBLAS routines. Use the same termination criterion and initial guess as in the previous part.

Since we are working with a *symmetric* matrix, you want to use a CBLAS routine for a symmetric matrix-vector product. Please read the “What functions are there and how can I call them from my C code?” section in the OpenBLAS FAQ:

<https://github.com/xianyi/OpenBLAS/wiki/Faq#what-functions-are-there-and-how-can-i-call-them-from-my-c-code>

You will need routines from both level 1 and level 2 BLAS, follow the NetLib⁵ link in the FAQ link above (you will find the DAXPY, SGEMV and DGEMM routines from Problem 2 in Homework 2 there as well). In order to use the CBLAS routines, you must include the `cblas.h` header in your code. The following routines may be useful for this task:

⁴https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms

⁵https://netlib.org/blas/index.html#_blas_routines

- `cblas_dsymv`
- `cblas_ddot`
- `cblas_dnrm2`
- `cblas_dscal`

Note that the CBLAS routines need additional arguments that define the memory layout used since C/C++ uses row-major by default but BLAS routines are implemented in Fortran which is column-major by default. You can find the CBLAS function declarations in the file “`${OPENBLAS_INCLUDE}/cblas.h`” after you have loaded the OpenBLAS module on the cluster (see below). A BLAS quick reference guide can be found at this link: <http://www.netlib.org/blas/blasqr.pdf>.

The code will generate a file `cblas.dat` which you can use for plotting. Add this data to the plot you have created in part a.) and label it with BLAS implementation.

To build your code with BLAS, you must load the OpenBLAS library before you compile

```
$ module load gcc/12.1.0-fasrc01 OpenBLAS/0.3.7-fasrc01
```

You can then build your code with the provided Makefile by running

```
$ make power_cblas
```

- c) In this last part we implement a third version of the power method using the C++ *header only* Eigen⁶ library. The benefit of Eigen is operator overloading available in C++ that allows you to write code in an object oriented fashion, similar to Python.

Follow the TODO labels in the file `code/t1/src/power_eigen.cpp` and implement the power method using the already defined Eigen instances for A and q . Use the same termination criterion and initial guess as in the previous part. See this documentation to learn more about matrix and vector arithmetic in Eigen

https://eigen.tuxfamily.org/dox/group__TutorialMatrixArithmetic.html

The necessary header is already included for you. You may find the following methods implemented in Eigen matrix/vector types useful (see the documentation for their full description):

- `normalize()`
- `dot()`

The code will generate a file `eigen.dat` which you can use for plotting. Add this data to the plot you have created in part a.) and label it with Eigen implementation.

To build your code with Eigen, you must load the library before you compile

⁶<https://eigen.tuxfamily.org/index.php>

```
$ module load eigen/3.3.7-fasrc01
```

You can then build your code with the provided Makefile by running

```
$ make power_eigen
```

- d) If you ran your three implementations from the previous tasks and carefully checked the wall time it takes to execute them, you may have observed only a small or moderate improvement compared to your naive implementation in `power_manual.cpp` (assuming you did not spend effort for optimization).

The BLAS library can be compiled in different ways and optimized targets can be built for different architectures. Compiling against optimized machine code is transparent of your code. Since you have written your code using the CBLAS API by including the `cblas.h` header, what happens is that your code will be *linked* to a different, optimized object code for these functions. Your code will automatically benefit from this if you compile and link it to a different BLAS implementation (the same is happening with different MPI implementations for example).

The lab handout includes a job submission script that you can use to run two jobs, each using a different OpenBLAS implementation (one is optimized for the Intel Broadwell CPUs we are using). Study the parts in the submission script where the library modules are loaded and submit the following jobs (you should do this on a compute node since the login nodes are not fully reliable for job submission):

```
$ sbatch submit_intel_broadwell.sh  
$ sbatch submit_intel_broadwell.sh optimized
```

When your jobs are done, study the output files and look for differences in the wall time of the four jobs. Briefly document your observations in a file `observations.md` (if any). Submit your observation notes along with the job output files.