02635 Fall 2016 — Module 9

Homework

- Read section 12.8 (pp. 170-178) in "Writing Scientific Software"
- Catch up on unfinished exercises from previous weeks
- Optional: read Appendix A in "Writing Scientific Software" if you need to brush up your linear algebra knowledge

Introduction

Linear algebra is an essential tool in computational mathematics, and software for numerical linear algebra is a key component in many mathematical software packages. Matrix Laboratory, or MATLAB, is a well-known software package that makes it easy for the user to manimulate matrices and vectors, and it provides a comprehensive set of numerical linear algebra routines for various matrix decompositions and computations. MATLAB is an interpreted language, and it is great for fast prototyping and research. However, MATLAB is proprietary and has strict hardware requirements which means that it is not well-suited for all applications. However, implementing a set of high-quality routines for numerical linear that one can use in a stand-alone program is time-consuming and by all means nontrivial, but luckily there are a number of free libraries that a both free and of high quality.

In this week's exercises, we will work with two standard external libraries that provide a set of numerical linear algebra routines, namely the BLAS (an abbreviation of *Basic Linear Algebra Subroutines*) and LAPACK (an abbreviation of *Linear Algebra PACKage* and pronounced *L-A-PACK*). The BLAS and LAPACK libraries were developed in late 1970s in FORTRAN, a programming language that was very popular at the time (and still is today in some communities). The reference implementations, which are available at netlib.org, are still FORTRAN code, but their technical specification has been standardized, so today there exists a number optimized implementations such as MKL (Intel's *Math Kernel Library*), ACML (*AMD Core Math Library*), ATLAB (*Automatically Tuned Linear Algebra Software*), and OpenBLAS. While these libraries may not be implemented in FORTRAN, they all adhere to the BLAS specification, and the compiled libraries can be used in many different programming languages simply by linking against the compiled library.

The BLAS provides a number of routines that are specified in the BLAS Technical Forum Standard. As described in "Writing Scientific Software", the BLAS routines are divided into three categories:

- BLAS level 1 routines implement basic vector operations (scaling a vector, adding two vectors, etc.)
- BLAS level 2 routines implement basic matrix-vector operations (matrix-vector multiplication, solving triangular systems of equations, etc.)
- BLAS level 3 routines implement matrix-matrix operations (matrix-matrix multiplication, etc.).

The BLAS routines have mnemonic names. For example, the routine for adding a scalar multiple of a vector to another vector is called **daxpy** which is a mnemonic for remembering *double (precision) a x plus y*. The same operation for single precision also exists, and it has the name **saxpy** (*single (precision) a x plus y*). You can find a complete list of the routines in the BLAS library on the BLAS website.

The LAPACK library provides a number of matrix factorization routines (LU, Cholesky, QR, etc.), routines for computing eigenvalues and singular values, routines for solving linear equations, and routines for solving linear least-squares problems. Many routines in the LAPACK library make use of routines from the BLAS library, and hence both libraries are necessary when using LAPACK. The LAPACK routines also have mnemonic names. For example, the **dgels** routine solves a least-squares problem (LS) with a unstructured or general (GE) coefficient matrix with input in double precision (D).

Exercises

1. Download the ZIP file `week9.zip` from CampusNet. It includes two examples, `example_blas.c` and `example_cblas.c`, that demonstrate how to scale an array of doubles using the BLAS routine `dscal`. The first example, `example_blas.c`, includes the `dscal` prototype explicitly in the source file:

   ```
   /* DSCAL (scale array)                              */
   void dscal_(
     const int * n,          /* length of array      */
     const double * a,       /* scalar a             */
     double * x,             /* array x              */
     const int * incx        /* array x, stride      */
   );
   ```

   The `dscal` routine scales n elements of an array by a scalar a (i.e., corresponding to the first two arguments). The n elements that will be scaled are `x[0]`, `x[*incx]`, `x[2*(*incx)`, ..., `x[(n-1)*(*incx)`. In other words, the first n elements of the array `x` will be scaled if `*incx` is equal to 1, every other element (starting with `x[0]`) will be scaled if `*incx` is equal to 2, and every kth element will be scaled if `*incx` is equal to k. Finally, notice that with the exception of the third argument `x`, all other arguments are specified using the `const` type qualifier. This tells you that the the routine is only allowed to modify `x`.

The second example, `example_cblas.c`, uses CBLAS which is a C interface to the BLAS library. CBLAS provides a header file, `cblas.h`, that includes a prototype for each routine in the CBLAS library. The CBLAS equivalent of the BLAS routine `dscal` is `cblas_dscal` which has the following prototype:

```
/* CBLAS_DSCAL (scale array)                        */
cblas_dscal(
    const int n,           /* length of array     */
    const double a,        /* scalar a            */
    double * x,            /* array x             */
    const int incx         /* array x, stride     */
);
```

Notice that unlike the BLAS prototype, most of the arguments in the CBLAS prototype are not pointers. If you would like to use the CBLAS library, you may include the following preprocessor directives

```
#if defined(__APPLE__) && defined(__MACH__)
#include <Accelerate/Accelerate.h>
#else
#include <cblas.h>
#endif
```

to include the correct header file in your code.

A remark for Windows users: The BLAS and LAPACK libraries are included on many Unix/Linux systems, including the DTU Unix system and Mac OS X. Unfortunately this is not the case with Windows. The ZIP file `week9.zip` contains a 64-bit version of OpenBLAS for Windows which contains both the BLAS and LAPACK libraries (`libopenblas.a`). Ask the instructor or a TA for help if you encounter any problems, or do the exercises on the DTU Unix system via ThinLinc.

Compiling the examples on the DTU Unix system

The first example, `example_blas.c`, may be compiled using the following command:

```
$ gcc example_blas.c -Wall -L/usr/lib64/atlas -lf77blas -o example_blas
```

The `-L` directive tells the linker where to look for the BLAS library (i.e., in the directory `/usr/lib64/atlas`) and the `-l` directive tells the linker to link against `f77blas` which is the name of the BLAS library included in ATLAS.

The second example, `example_cblas.c` may be compiled using the following command:

```
$ gcc example_cblas.c -Wall -L/usr/lib64/atlas -lcblas -o example_cblas
```

Notice that the `-l` directive is now follwed by `cblas` which tells the linker to link against the CBLAS library.

Mac OS X

The first example, `example_blas.c`, may be compiled using the following command:

```
$ cc example_blas.c -Wall -lblas -o example_blas
```

The second example, `example_cblas.c` may be compiled using the following command:

```
$ cc example_cblas.c -Wall -lcblas -o example_cblas
```

Windows

Assuming that OpenBLAS library (`libopenblas.a`) is in the same directory as the source, the first example, `example_blas.c`, may be compiled using the following command:

```
$ gcc example_blas.c -Wall -L. -lopenblas -o example_blas
```

The directive `-L` is followed by a period. This tells the linker to look for the BLAS library in the current directory.

The second example, `example_cblas.c` may be compiled using the following command:

```
$ gcc example_cblas.c -Wall -L. -lopenblas -o example_cblas
```

Makefiles

The ZIP file also contains makefiles which can be used to compile the examples. This is how you can compile the first example on the DTU Unix system:

```
$ make -f Makefile_gbar_blas
```

2. Use the `dscal` routine from the BLAS library (or `cblas_dscal` from the CBLAS library) to scale

   - a row of a two-dimensional array,
   - a column of a two-dimensional array,
   - the diagonal of a two-dimensional square array.

   Start by creating a small two-dimensional array, say, of size 5×5, and set all elements to 1.0. Then do the following:

   1. scale the second row by 2.5 using `dscal`
   2. scale the third column by -0.2 using `dscal`
   3. scale the diagonal elements by 2.0 using `dscal`
   4. print the two-dimensional array to check that correct elements were scaled.

3. The BLAS routine `daxpy` can be used for simple vector operations and for row and column manipulations in a matrix. The `daxpy` prototypes is as follows:

```
/* DAXPY (double a x plus y)                          */
void daxpy_(
    const int * n,          /* length of arrays x and y */
    const double * a,       /* scalar a                 */
    const double * x,       /* array x                  */
    const int * incx,       /* array x, stride          */
    double * y,             /* array y                  */
    const int * incy        /* array y, stride          */
);
```

   Recall that you need to include this in your source file. Alternatively, if you prefer to use CBLAS, you may include the `cblas.h` header file which specifies the `cblas_daxpy` prototype:

```
/* CBLAS_DAXPY (double a x plus y)                    */
void cblas_daxpy (
    const int n,            /* length of arrays x and y */
    const double a,         /* scalar a                 */
    const double *x,        /* array x                  */
    const int incx,         /* array x, stride          */
    double *y,              /* array y                  */
    const int incy          /* array y, stride          */
);
```

   Write a program that creates a two-dimensional array

```
double A[4][3] = {{1,2,3},{4,5,6},{7,8,9},{10,11,12}};
```

and uses `daxpy` or `cblas_daxpy` to perform the following operations:

- subtract 4.0 times the first row of the array `A` from its second row
- subtract 7.0 times the first row of the array `A` from its third row
- subtract 10.0 times the first row of the array `A` from its fourth row
- subtract the second column of `A` from its third column.

Print the result to verify that you get the correct result.

4. The BLAS level 2 routine `dgemv` is used for "general matrix-vector" multiplication and may perform one of the following operations:

$$y := \alpha Ax + \beta y \quad \text{or} \quad y := \alpha A^T x + \beta y$$

where α and β are scalar constants and A is a matrix of size $m \times n$. The `dgemv` prototype is as follows:

```
/* DGEMV (general matrix-vector product)                    */
void dgemv_(
    const char * trans,    /* 'N' or 'T' if A is transposed  */
    const int * m,         /* rows in A                      */
    const int * n,         /* columns in A                   */
    const double * alpha,  /* scalar alpha                   */
    const double * A,      /* array A                        */
    const int * lda,       /* array A, leading dimension     */
    const double * x,      /* array x                        */
    const int * incx,      /* array x, stride                */
    const double * beta,   /* scalar beta                    */
    double * y,            /* array y                        */
    const int * incy       /* array y, stride                */
);
```

The first input argument `trans` must be a pointer to a `char` and determines whether to multiply with A (if `trans` points to the character `N`) or its transpose A^T (if `trans` points to the character `T`).

Routines from the BLAS library require that matrices are stored as one-dimensional arrays with the column-major ordering of the elements. Recall that the internal layout of two-dimensional arrays in C use row-major ordering of the elements, so if `B` is a pointer to a two-dimensional array, then `*B` is a pointer to the first element of the first row. It is easy to verify that the exact same internal layout

corresponds to a column-major representation of the transpose of the array—the rows of a matrix B are the columns of B^T .

The input argument `lda` is the *leading dimension* of the array `A` . For a column-major array, this should be equal to the number of rows in the array.

The following example demonstrates how you use `dgemv` for the following matrix-vector multiplication

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} := \alpha \begin{bmatrix} 1 & 2 & 3 & 4 \\ -2 & 1 & -3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \beta \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

with $\alpha = 2.0$, $\beta = -1.0$, $x = (1, 0, 0, 0)$ and $y = (1, 1, 1)$:

```c
#include <stdio.h>

void dgemv_(
    const char * trans,    /* 'N' or 'T' if A is transposed  */
    const int * m,         /* rows in A                      */
    const int * n,         /* columns in A                   */
    const double * alpha,  /* scalar alpha                   */
    const double * A,      /* array A                        */
    const int * lda,       /* array A, leading dimension     */
    const double * x,      /* array x                        */
    const int * incx,      /* array x, stride                */
    const double * beta,   /* scalar beta                    */
    double * y,            /* array y                        */
    const int * incy       /* array y, stride                */
);

int main(void) {

  double A[3][4] = {{1,2,3,4},{-2,1,-3,4},{5,6,7,8}};
  double x[4] = {1,0,0,0};
  double y[3] = {1,1,1};
  double alpha=2.0, beta=-1.0;
  int i, lda=4, incx=1, incy=1, m=3, n=4;
  char transN='N', transT='T';

   /*       trans,  m,  n,  alpha,  A,  lda, x,  incx,  beta, y,  incy  */
   dgemv_(&transT, &n, &m, &alpha, *A, &lda, x, &incx, &beta, y, &incy);

   /* Print result */
   for (i=0;i<3;i++) printf("y[%d] = % g\n",i,y[i]);

   return 0;
}
```

Notice that the first input is a pointer to the character `T`, and the dimensions m and n have been swapped. This is to account for the fact that matrix A is stored in row-major format, but the `dgemv` routine will interpret it as column-major.

The matrix-vector multiplication can also be done with the CBLAS interface as follows:

```c
#include <stdio.h>
#if defined(__APPLE__) && defined(__MACH__)
#include <Accelerate/Accelerate.h>
#else
#include <cblas.h>
#endif

int main(void) {

  double A[3][4] = {{1,2,3,4},{-2,1,-3,4},{5,6,7,8}};
  double x[4] = {1,0,0,0};
  double y[3] = {1,1,1};
  double alpha=2.0, beta=-1.0;
  int i, lda=4, incx=1, incy=1, m=3, n=4;

  cblas_dgemv(CblasRowMajor, CblasNoTrans, m, n,
              alpha, *A, lda, x, incx, beta, y, incy);
  /* Print result */
  for (i=0;i<3;i++) printf("y[%d] = % g\n",i,y[i]);

  return 0;
}
```

Notice that unlike the direct BLAS call in the previous example, the CBLAS call has an extra argument (the first argument `CblasRowMajor`) that specifies that the array `A` is row-major, and hence we can use `CblasNoTrans` (corresponding to no transposition) and do not have to swap m and n. Thus, the CBLAS routine `cblas_dgemv` is often more convenient than the `dgemv` routine when programming in C.

Use the BLAS routine `dgemv` (or the CBLAS routine `cblas_dgemv`) to perform the following operations:

- compute $x = A^T y$ using the matrix A from the example and $y = (1, -1, 1)$
- compute the sum of the rows in A
- compute the sum of the last three columns of A
- add two times the sum of the first two rows of A to the last row of A.

Print the results to verify your code.