Physics 750: Lecture 14 Handout

Tuesday, October 31, 2017

Performing linear algebra calculations in C/C++ is awkward. Unlike conventional mathematical notation, which uses one-based indexing, C/C++ array elements are counted from zero. Moreover, there is no completely standard (or even convenient) way to set up matrices. You have several choices:

1. You can use two-dimensional C arrays.

   ```
   const size_t N = 100, M = 150;
   double A[N][M];
   ```

   The code snippet above creates storage elements

   $$\begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,M} \\ A_{2,1} & \ddots & & \\ \vdots & & \ddots & \\ A_{N,1} & & & A_{N,M} \end{pmatrix} = \begin{pmatrix} \texttt{A[0][0]} & \texttt{A[0][1]} & \cdots & \texttt{A[0][M-1]} \\ \texttt{A[1][0]} & \ddots & & \\ \vdots & & \ddots & \\ \texttt{A[N-1][0]} & & & \texttt{A[N-1][M-1]} \end{pmatrix}.$$

   Except for the indices being offset by one, this is conceptually close to the mathematical notation of an $N \times M$ matrix $A_{i,j}$ with $i = 1, 2, \ldots, N$ and $j = 1, 2, \ldots, M$. In reality, the elements are laid out linearly in memory in row-major order. Each term `A[n]` is a pointer to the first element of the `n`th row.

   $$\underset{\substack{\uparrow \\ \texttt{A[0]}}}{\boxed{\texttt{A[0][0]}}} \quad \boxed{\texttt{A[0][1]}} \quad \cdots \quad \boxed{\texttt{A[0][M-1]}} \quad \underset{\substack{\uparrow \\ \texttt{A[1]}}}{\boxed{\texttt{A[1][0]}}} \quad \boxed{\texttt{A[1][1]}} \quad \cdots \quad \boxed{\texttt{A[1][M-1]}}\cdots$$

   A serious disadvantage to this scheme is that multi-dimension arrays cannot be passed to functions in the way you would expect. (Their dimensions must be known at compile time. If that is not the case, you need to pass a pointer to a dynamically allocated block of memory and integer values describing the matrix shape.)

2. You can use one-dimensional C arrays and explicitly implement the row-major or column-major ordering.

   ```
   double B[N*M];
   double C[N*M];

   for (size_t n = 0; n < N; ++n)
      for (size_t m = 0; m < M; ++m)
         B[n+m*N] = C[n*M+m] = A[n][m]; // B by row and C by column
   ```

   An important consideration is that LAPACK,[†] the most important numerical library of linear algebra routines, is written in FORTRAN. By convention, FORTRAN stores matrices ordered by column.

3. The C++ language standard does not yet[‡] provide a standard matrix class. But you can create your own or use one the many freely-available class definitions.

[†] http://www.netlib.org/lapack/

[‡] http://www.boost.org/doc/libs/1_65_1/libs/numeric/ublas/doc/index.html

```
class matrix
{
private:
   size_t N, M;
   vector<double> data;
public:
   matrix() : N(0), M(0), data() {}
   matrix(size_t N_, size_t M_) : N(N_), M(M_), data(N*M) {}
   void resize(size_t N_, size_t M_) { N=N_; M=M_; data.resize(N*M); }
   size_t rows(void) const { return N; }
   size_t cols(void) const { return M; }
   double& operator()(size_t n, size_t m) { return data[n*M+m]; }
   const double& operator()(size_t n, size_t m) const { return data[n*M+m]; }
   void operator=(double x) { for (size_t p = 0; p < data.size(); ++p) data[p] = x; }
};

matrix operator+(const matrix &A, const matrix &B)
{
   assert(A.rows()==B.rows() and A.cols()==B.cols());
   matrix tmp(A.rows(),A.cols());
   for (size_t n = 0; n < A.rows(); ++n)
      for (size_t m = 0; m < A.cols(); ++m)
         tmp(n,m) = A(n,m) + B(n,m);
   return tmp;
}
```

In the code snippet above, `operator+` implements matrix addition (but inefficiently—note the hidden temporary). In practice, it's often better to use the low-level matrix and vector operations provided by BLAS (Basic Linear Algebra Subprograms). Many computer vendors ship hand-tuned versions of these routines that are very fast. The LAPACK (Linear Algebra PACKage) library is build on top of BLAS. It can compute matrix inversion, single-value decomposition, and eigenvectors using a variety of methods.

For maximum compatibility with older code, LAPACK is written in FORTRAN and uses FORTRAN77 bindings. It assumes that matrices use column-major order and one-based indexing. The library functions are named according to the convention TXXYYY, where T denotes the working type, XX the storage format of the matrices, and YYY the nature of the computation. LAPACK uses conventional storage for two-dimensional arrays and packed storage for symmetric, Hermitian, or triangular matrices.

`s`	`float`
`d`	`double`
`c`	`complex<float>`
`z`	`complet<double>`

`di`	diagonal
`ge`	general
`gb`	general band
`gt`	general tridiagonal
`sy`	symmetric
`sp`	symmetric (packed)
`he`	hermitian
⋮	etc.

For example, the upper triangular (`'U'`) part of an $N \times N$ symmetric matrix can be packed into a linear array of size $N(N+1)/2$.

$$\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{pmatrix}$$

For elements $1 \leq i \leq j \leq N$, the matrix is reorganized according to $i, j \to i + \frac{1}{2}j(j-1)$.

$$\boxed{A_{1,1}}\ \underbrace{\boxed{A_{1,2}}\boxed{A_{2,2}}}\ \underbrace{\boxed{A_{1,3}}\boxed{A_{2,3}}\boxed{A_{3,3}}}\ \underbrace{\boxed{A_{1,4}}\boxed{A_{2,4}}\boxed{A_{3,4}}\boxed{A_{4,4}}}$$

In the following code, we use the equivalent zero-indexed rule, `i,j` $\to$ `i+j*(j+1)/2` with $0 \leq$ `i` $\leq$ `j` $<$ `N`, to build a symmetric matrix. The matrix is then put into the `evd` eigenvalue solver. In this case, additional integer and double-precision work space is required.

```
int eigensolve(vector<double> &H, vector<double> &Eval, vector<double> &Evec)
{
   // Solve the eigenvalue problem with LAPACK's dsepvd routine
   int N = Eval.size();
   assert(H.size() == N*(N+1)/2);
   assert(Evec.size() == N*N);

   int info;
   char jobz='V';
   char uplo='U';
   vector<double> work(1+6*N+N*N);
   int lwork = work.size();
   vector<int> iwork(3+5*N);
   int liwork = iwork.size();

   dspevd_(&jobz,&uplo,&N,&(H[0]),&(Eval[0]),&(Evec[0]),
           &M,&(work[0]),&lwork,&(iwork[0]),&liwork,&info);

   return info;
}

vector<double> H(55);
vector<double> Eval(10);
vector<double> Evec(100);

for (size_t j = 0; j < 10; ++j)
   for (size_t i = 0; i <= j; ++i)
      H[i+j*(j+1)/2] = some_function(i,j);

if (eigensolve(H,Eval,Evec)) { ... }
else
{
   cerr << "Solver failed!" << endl;
   exit(1);
}
```

A more modern alternative is Eigen,[§] a C++ template library for linear algebra that provides matrices, vectors, numerical solvers, and related algorithms.

[§] http://eigen.tuxfamily.org/index.php?title=Main_Page