PHYS4070 Worksheet. Week 2: Matrices and Eigenvalues

We could store a matrix using old C-style 2D array:

```
static const int dim = 2; //'static const': must be compile-time constant!
double x[dim][dim] = {0.5, 1.5, 2.5, 3.5};
for(int i=0; i<dim; i++){
    for(int j=0; j<dim; j++){
        std::cout << x[i][j] << ' ';
    }
    std::cout << '\n';
}
```

But, this requires array size to be known at compile time.
It also has all the pitfalls of old C-style arrays (raw pointers, no range checks etc.)
It's possibly to also use dynamically-sized C-style arrays, but there be dragons.

```
// Exactly equivalent to:
double y[dim*dim] = {0.5, 1.5, 2.5, 3.5};
for(int i=0; i<dim; i++){
    for(int j=0; j<dim; j++){
        std::cout << y[i*dim + j] << ' ';
    }
    std::cout << '\n';
}
```

- The 2D array is actually stored as a single chunk of memory (i.e. in 1 dimension), and `[i][j]` is just short-hand for (i*dim+j)
- Can do this using dynamic memory allocation too, but we will not; in c++ there are nicer ways
- In modern c++, we would not use a basic c-array, but instead use a class (data structure from a library). Examples below.

Using regular std::vector, wrap in a class

- We *can* however use an std::vector to store a 2D matrix, we just need to access the elements using the i*dim+j
- Example: consider following matrix:

$$\begin{pmatrix} 0.5 & 1.5 \\ 2.5 & 3.5 \end{pmatrix}$$

```
std::vector<double> v{0.5, 1.5, 2.5, 3.5};
int dim = 2; // for a 2x2 matrix
for(int i=0; i<dim; i++){
    for(int j=0; j<dim; j++){
        std::cout << v.at(i*dim + j)<<" ";
    }
    std::cout<<'\n';
}
```

- In order to make things easier, we will make our own class that holds a matrix using std::vector to store data
- In real-world code, many such classes exist already, and we would use one of these matrix classes (e.g., from the great 'Eigen' library)
- But here, we will "re-invent the wheel", since it is a good learning exercise, and will greatly help you understand classes in c++
- Also: what we need for the assignment is very simple, we can code it in just a few lines

---

Worksheet tasks: part A

- I will give you solutions to this part at the end of Wednesday workshop; you may use your or my solution to continue on with part B for Thursday

1. Write a class to store a 2D square matrix

- Use std::vector to hold data
- The constructor should take the dimension N as input, and create a vector of correct length (N*N)
- Provide a member function to return .data() from vector, so we can access the c-style array (needed to interface with lapack)
  - `double* data() { return v.data(); }`
- Provide a function that allows us to read *and edit* the i,j element
  - To edit, this must return a *reference*, e.g.,
  - `double & at(int j, int j) { return v.at(i*dimension + j); }`
  - This allows, e.g., `x = matrix.at(i,j);` *and* `matrix.at(i,j) = x;`
- Provide operator overload of '+', that allows us to add two matrices together
  - This must be a function that takes two matrices, and returns one matrix (function signature)

2. Write a function that takes in a matrix (class you created above) and find the eigenvalues and eigenvectors

- Assume the matrix is real and symmetric, so use LAPACK function DSYEV
- Documentation: http://www.netlib.org/lapack/explore-html/index.html
- In/out parameters listed below:

```
int dsyev_(
  char * jobz,  // 'V' = compute e. values and vectors. 'N' = values only
  char * uplo,  // 'U' = upper triangle of matrix is stored, 'L' = lower
  int * n,      // dimension of matrix a
  double * a,   // c-style array for matrix a (ptr to array, pointer to a[0])
                // On output, a contains matrix of eigenvectors
  int * lda,    // For us, lda=n
  double * w,   // array of dimension n - will hold eigenvalues
  double * work,// 'workspace': array of dimension lwork
  int * lwork,  // dimension of workspace: ~ 6*n works well
  int * info    // error code: 0=worked.
);
```

- This function should return the eigen values *and* vectors [eigenvectors are stored in a matrix (2D array)]
- Eigenvalues are sorted, and eigenvectors are normalised to 1 (via inner product)
- Normally, functions in c++ return only one thing. We have two options:
- A: Pass in/out parameters to function by reference like this:
  - `void solveEigenSystem(Matrix matrix, Matrix &eigenvectors, Vector &eigenvalues);`
  - This is typically frowned upon, since it makes code difficult to read (which value is input, which is output?)
- B: Define a class/struct that holds a matrix of eigenvectors and a vector of eigenvalues, (e.g., called *MatrixAndVector*), and returns this
  - `MatrixAndVector solveEigenSystem(Matrix matrix);`
- Note: FORTRAN (language LAPACK is written in) uses column-major ordering to access 2D arrays, wile c and c++ use row-major.
  This means `m[i][j]` in c++ is `m[j][i]` in FORTRAN.. so we often need to transpose the matrix before sending to LAPACK
  - Our matrix is symmetric, so this doesn't matter, except for 'uplo'
  - 'uplo': 'U' means upper triangle *in FORTRAN* is stored -- so lower in c++ [we can just fill entire matrix though]
  - For other LAPACK functions, you can often just tell them the matrix is a transpose, so we don't need to waste time transposing it ourselves
- Don't forget `extern "C"`, and to declare the `dsyev_` function. and the -llapack linker (compile) flag (you may also need the -lblas flag)
- Example:

```
// std::vector<double> matrix = ...
// Assume this std::vector contains our matrix

char jobz{'V'};
char uplo{'U'};
int dimension = ...; // N if we have NxN matrix
int lwork = 6 * dimension;
std::vector<double> work(lwork);
int info = 0; // will hold potential error message

// create a blank vector to store calculated eigenvalues:
std::vector<double> evals(dimension);

dsyev_(&jobz, &uplo, &dimension, matrix.data(), &dimension, result.vector.data(), work.data(),
// note: on INPUT 'matrix' is the input matrix. After dsyev_ runs, 'matrix' will now contain a

// check for errors
if (info != 0) {
    std::cout << "DSYEV returned error code: " << info << '\n';
}
```

3. Use your code to calculate eigen values/vectors of simple 2x2 matrix

$$m_{ij} = \frac{1.0}{i + j + 1.0}$$

- i,j range from 0 to 1
- Expected eigenvalues should be: {1.26759, 0.0657415}
- With corresponding eigenvectors: {{1.86852, 1.}, {-0.535184, 1.}}
    - Note: Normalisation will be different with LAPACK

4. Quantum simple harmonic oscillator (optional)

- The Hamiltonian of 1D QSHO, in simplest units case, is

$$H = \frac{\hat{p}^2}{2} + \frac{x^2}{2}$$

- Use finite-difference method to solve Schrodinger eq over x=[-5,5] by casting problem to matrix eigenvalue problem
    - Encode derivative operator as a matrix: (... 1, -2, 1, 0, ...) / dx^2, dx = (xmax - xmin)/Nsteps
    - Form full symmetric Hamiltonian matrix
    - Use hard boundary condition
    - Probably need at least a few hundred steps

- Compare eigenvalues to known energies: En = (n + 1/2)
- We also have a full set of orthogonal wavefunctions (eigenvectors). These are not yet properly normalised: check that the first two wavefunctions (eigenvectors) are indeed orthogonal
- Plot First 3 wavefunctions - do the look how you expect?

Worksheet tasks: part B: Hydrogen

In the assignment, you are tasked with Solving Schodinger equation for a many-electron atom; here we will practise the procedure for the simplest case of hydrogen.

- The radial Hamiltonian for Hydrogen atom is:

$$H_r = \frac{-1}{2}\frac{\partial^2}{\partial r^2} - \frac{Z}{r} + \frac{l(l+1)}{2r^2}, \tag{1}$$

- We will use a new very powerful method to solve the Schrodinger equation, by expanding the solutions over a basis of B-spline (basis) functions, b. (Use provided code to calculate B-splines)

$$P(r) = \sum_j^{N_b} c_j b_j(r), \tag{2}$$

- Solve the Schrodinger equation for Hydrogen by solving the eigenvalue problem using DSYGV:

$$\sum_j \langle i|\hat{H}_r|j\rangle c_j = \varepsilon \sum_j \langle i|j\rangle c_j \tag{3}$$

$$\implies H_r\,\vec{c} = \varepsilon\, B\,\vec{c}. \tag{4}$$

$$H_{ij} = \langle i|\hat{H}_r|j\rangle = \int b_i(r)\hat{H}_r b_j(r)\,dr\,, \qquad B_{ij} = \langle i|j\rangle = \int b_i(r)b_j(r)\,dr, \tag{5}$$

You can use any integration scheme for these integrals - it will be much easier if you store the values of the B-splines in an array *before* trying to do the integrals.

For the general case of

$$H = -\frac{1}{2}\frac{\partial^2}{\partial r^2} + V(r),$$

we have

$$H_{ij} = -\frac{1}{2}\int b_i(r) b_j''(r)\, dr + \int b_i(r) V(r) b_j(r)\, dr \tag{6}$$
$$= +\frac{1}{2}\int b_i'(r) b_j'(r)\, dr + \int b_i(r) V(r) b_j(r)\, dr \tag{7}$$

(using integration by parts).
Integration by parts has two benefites: simpler and more stable to calculate first-derivatives, and H becomes manifestly symmetric (it would be symmetric anyway, except for numerical errors).

As described in lectures, discard the first two (index=0 and 1) B-splines, and the last one (index=n-1) to enforce the boundary conditions.

Use ~30-60 Bsplines of order k=7. You will have to choose good r0 and rmax.

1. Compare energies for s and p states to expected
   - Note: Biggest source of error likely comes from integration grid, r0, rmax, and num_stepd
   - Since the H and B matrix sizes depend on number of B-splines used, NOT number of integration points, we can increase number of points without slowing down code very much!
2. Use expansion coefficients and B-splines to construct wavefunctions; check that they are properly normalised (they should already be)
3. Plot wavefunctions for 1s, 2s, and 2p
4. Think about simple extension to this needed for assignment.

DSYGV parameters (very similar to DSYEV, can adapt previous code)

```
extern "C"
int dsygv_(
    int *ITYPE,   // =1 for problems of type Av=eBv
    char *JOBZ,   // ='V' means calculate eigenvectors
    char *UPLO,   // 'U': upper triangle of matrix is stored, 'L': lower
    int *N,       // dimension of matrix A
    double *A,    // c-style array for matrix A (ptr to array, pointer to a[0])
                  // On output, A contains matrix of eigenvectors
    int *LDA,     // For us, LDA=N
    double *B,    // c-style array for matrix B [Av=eBv]
    int *LDB,     // For us, LDB =N
    double *W,    // Array of dimension N - will hold eigenvalues
    double *WORK, // 'workspace': array of dimension LWORK
    int *LWORK,   // dimension of workspace: ~ 6*N works well
    int *INFO     // error code: 0=worked.
);
```

Example for using the provided B-spline code

```cpp
#include "bspline.hpp"
#include <iostream>
int main(){
  double r0 = 1.0e-3;
  double rmax = 50.0;
  int k_spine = 7;   // order of B-splines
  int n_spline = 60;

  // Initialise the B-spline object
  BSpline bspl(k_spine, n_spline, r0, rmax);

  // Value of the 1st (index=0) B-spline at r=0
  std::cout << bspl.b(0, 0.0) << "\n";
  // Value of the 6th (index=5) B-spline at r=1.5 au
  std::cout << bspl.b(5, 1.5) << "\n";
  // Value of the last (index=N-1) B-spline at r=rmax
  std::cout << bspl.b(n_spline - 1, rmax) << "\n";
}
```