Converting MATLAB code into C++ using Armadillo

Luke Taranowski

Abstract

This paper will delve into the process I used to convert pre-existing MATLAB code into C++. The code in question uses pseudospectral methods to solve the eigenvalue problem of a Helium particle in an extreme magnetic field, such as those in a neutron star. The results of these calculations can tell us what a particle looks like and how its electrons are changing state, and whether they are gaining or losing energy.

1. Introduction

MATLAB is well-known for its ease of use and powerful computation. However, for more advanced computations C++ can offer performance improvements. The translation between MATLAB and C++ will be facilitated with the Armadillo library, which greatly simplifies the process.

2. Translating MATLAB into C++

2.1 Setting up the Build Environment.

The first step in setting up the environment was to decide upon a library fit for our purposes. I decided on Armadillo which is a very fast linear algebra library for C++; acting as a wrapper for other linear algebra libraries. IntelMKL is one such library that I initially setup with the project, as it is well suited for the Intel processor on my Windows machine. However, this setup did not include support for non-symmetric matrix eigenvalue/eigenvector calculations around a sigma value, as Armadillo requires the ARPACK and SuperLU libraries for that, which are difficult to install onto a Windows machine. This issue was circumvented by using a virtual Ubuntu machine (VirtualBox), as it had packages readily available for OpenBLAS, ARPACK, and SuperLU.

2.2 Translation

Armadillo provides deliberately similar syntax to Matlab, which makes the actual translation easier. However there were a few key differences of note. Namely, Matlab is a

statically typed language, and C++ is dynamically typed, and indexing in Matlab starts at 1 as opposed to 0 as in C++. Armadillo also does not provide support for 4d matrices. The solution to this was to use a Field(similar to std::vector) of Cubes(3d matrix).

2.3 The Electron Data Structure

This structure acts as a representation of the electrons in our Helium atom. It contains their respective energy level, Neumann conditions, eigenvalue and eigenvector (once calculated). We use two electrons for the configuration of our atom, 1s0 and 2p0.

3. The Hydrogenic Problem

To solve the helium atom in a strong magnetic field requires solving a partial differential equation (PDE). The code begins by defining the parameters of the problem such as the nuclear charge, configuration of the atom, as well as the strength of the magnetic field. The first step involves creating a Chebyshev grid that clusters points near the edges (-1, +1). Using this grid increases the accuracy and speed of the calculations and is a core part of pseudospectral methods. We use the function 'cheb(N)' to accomplish this (Trefethen, 2008). This function facilitates the creation of the grid and also calculates the differentiation matrix. Next, we create the operator coefficients for the PDE. The boundary conditions for the problem are then specified. Dirichlet conditions are imposed on the +1 boundaries, where the wave function must vanish, so we can remove those corresponding rows. On the -1 boundaries the Neumann conditions are imposed, where the derivative must be zero. Next we construct the left-hand side operator, convert our matrix into a sparse matrix, and then solve the eigenvalue problem, storing the eigenvalues and eigenvectors in their respective electron data structures.

4. Plotting of Eigenvalues/vectors

Using the calculated eigenvectors, we can create a contour plot of the spatial probability distribution of the electron. We can see that as the magnetic field strength increases the hydrogen state becomes elongated.

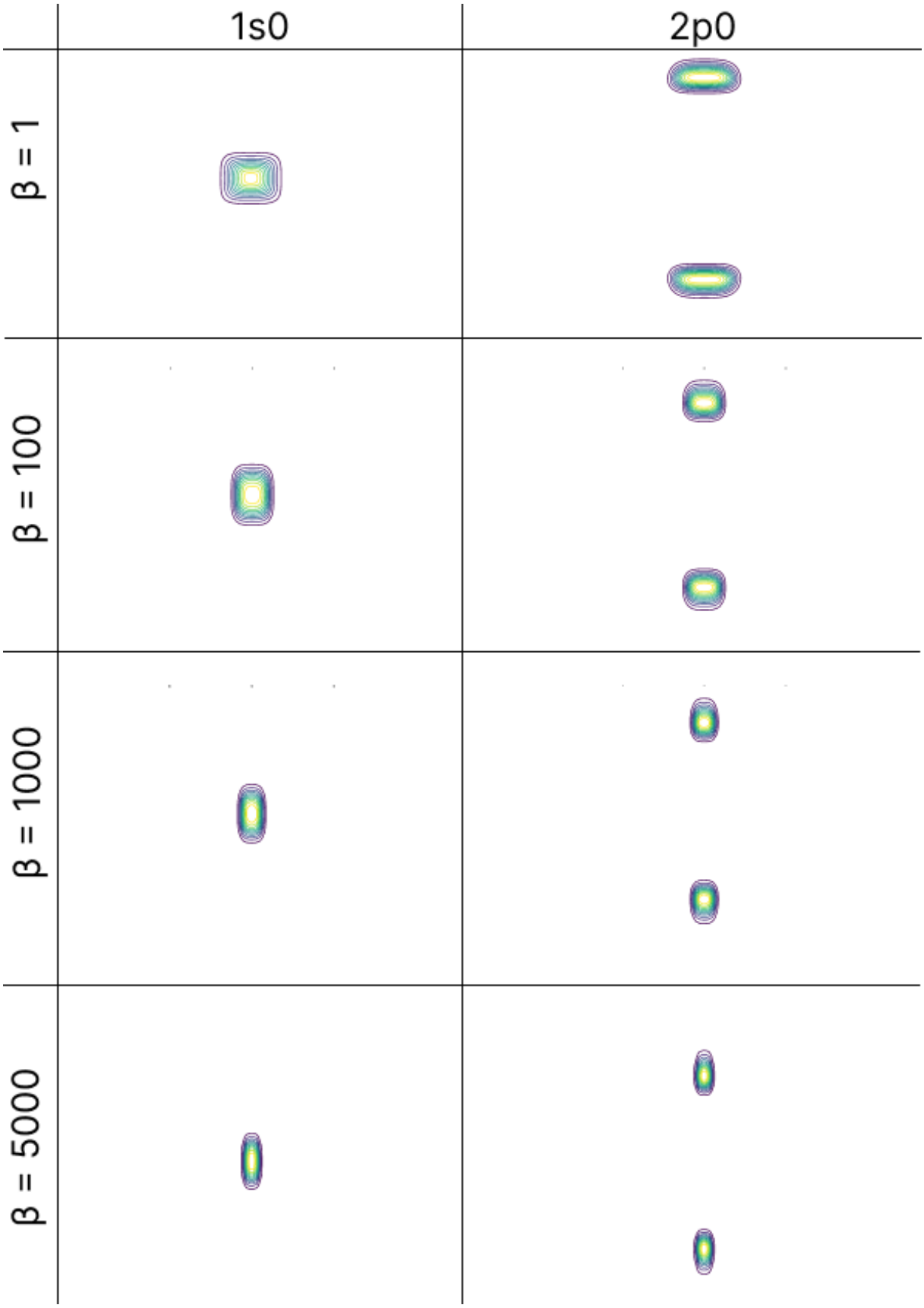

Figure 1: Contour Plot of Spatial Probability Distribution

5. Program Speedup / Runtime Implications

All testing was done inside the virtual machine. The results were obtained by running the calculations five times and taking the average. The calculation took 23.62 seconds in Matlab, as compared to 21.29 seconds in the C++ program, implying a minor speedup of about 11%. One thing to note is that the Matlab code is multi-threaded, and the C++ code is not. With the addition of multi-threading using OpenMP, the gains to performance could be more substantial. Moving the project outside of the virtual machine may also have yielded more performance improvements.

6. Conclusion

Overall the translation from MATLAB to C++ was a success. The outputs match, and the speed of the program was marginally improved. Future work on this project could include further analysis of the eigenvalues and eigenvectors, yielding even more information about the state of the atom, and the changing energy levels of its electrons. Multi-threading the code would also be a good avenue for improvement to increase the performance gains.

Author Contributions

Anand Thirmulai: Supervision, Original Code Author, Professor.

References

Trefethen, L. N. (2008). *Spectral methods in MATLAB (p.54)*. SIAM.

Appendix

A. Source Code

```
#include <string>
#include <vector>
#define _USE_MATH_DEFINES
#include <math.h>
#include <armadillo>
#include <fstream>
#include <iomanip>

#define tiny 1e-14
#define LARGE 1e+14
```

```
using namespace arma;

// The first quantum number represents the general energy level.
enum Term
{
   _1s0 = 0, // 1s0 == ground state. 1st state = 2s0 and so on.

   _2s0,
   _2p0,
   _2pMin1,

   _3pMin1,
   _3p0,
   _3dMin2,
   _3dMin1,

   _4fMin2,
   _4dMin1,

   _5fMin2,
};

struct Electron
{
   Electron(int m_, int parity_, mat Neumann_,
            int excitation_, mat term_, mat direct_Neumann_)
   {
      m = m_;
      parity = parity_;
      Neumann = Neumann_;
      excitation = excitation_;
      term = term_;
      direct_Neumann = direct_Neumann_;
   }

   int m;
   int parity;
   mat Neumann;
   int excitation;
   mat term;
   mat direct_Neumann;
   double eigval;
```

```
   vec eigvec;
   double Ehyd;
   vec eigval_ehyd;
};

void cheb(mat &D, mat &x, double N)
{
   if (N == 0)
   {
      D = {0};
      x = 1;
      return;
   }

   // x = cos(pi*(0:N)/N)'; //
   // Create column vector: [0, 1, 2, ... , N]
   colvec indices = linspace(0, N, N + 1);
   // Scale each element in indices by PI, then divide by N to normalize.
   colvec scaled_indices = M_PI * indices / N;
   // Apply the cosine function to each element.
   x = arma::cos(scaled_indices);

   //  c = [2; ones(N-1,1); 2].*(-1).^(0:N)';
   // Create column vector, where the size is N-1: [1, 1, 1, ... , 1]
   colvec c(N + 1, fill::ones);
   // Set endpoints to 2.
   c(0) = 2;
   c(N) = 2;

   // Create a column vector, size is N+1: [-1, -1, -1, ... , -1]
   colvec base(N + 1, fill::value(-1));
   // [0, 1, 2, ... , N] (Note: length is equal to N + 1)
   colvec powers = regspace(0, N);
   // Multiply everything together.
   colvec res = pow(base, powers);
   c = c % res;

   // X = repmat(x,1,N+1); //
   // Repeat the x vector N + 1 number of times to make a 2d matrix.
   mat X = repmat(x, 1, N + 1);

   // Make the differentiation matrix.
   mat dX = X - X.t();
```

```
   // D  = (c*(1./c)')./(dX+(eye(N+1))); //
   // N+1 by N+1 identity matrix.
   mat identity_mat = eye(N + 1, N + 1);

   // Calculate the inverse of c (each elements inverse)
   // and transpose it so that it's a row vector.
   vec c_inv = (1.0 / c);
   // Create a matrix out of c and its inverse.
   mat c_mat = c * c_inv.t();

   mat dX_identity = dX + identity_mat;
   D = c_mat / dX_identity;

   // D  = D - diag(sum(D')); //
   // Create a diagonal matrix, with each element
   // corresponding to the sum of a row.
   // (Note: Matlab only calculates the sums of columns, but Armadillo
   // lets us sum up the rows.
   // So we don't need to tranpose like in the original code.
   D = D - arma::diagmat(arma::sum(D, 1));
}

int main(void)
{
   const int Znucl = 2;

   mat config_term = {Term::_1s0, Term::_2p0};
   std::vector<double> config_term_cell;
   config_term_cell.push_back(Term::_1s0);
   config_term_cell.push_back(Term::_2p0);

   mat betaZar = {100};

   // Get length of BetaZar array.
   for (int betaZcount = 0; betaZcount < betaZar.size(); betaZcount++)
   {
      double betaZ = betaZar(betaZcount);
      mat sigma_guess_ar = {-10};
      double sigma_guess = sigma_guess_ar(betaZcount);

      mat rhomax_scaling_ar = {2};
      mat N_ar = {51};
```

```
for (int rhomax_count = 0; rhomax_count < rhomax_scaling_ar.size();
 rhomax_count++)
{
   for (int Ncount = 0; Ncount < N_ar.size(); Ncount++)
   {
      double N = N_ar(Ncount);

      std::vector<Electron> electrons;

      for (int i = 0; i < Znucl; i++)
      {
         if (config_term_cell[i] == Term::_1s0)
         {
            Electron new_electron(0, 1, {{1, 1}}, 0, config_term.col(i),
                {{1, 1}});
            electrons.emplace_back(new_electron);
         }
         else if (config_term_cell[i] == Term::_2s0)
         {
            Electron new_electron(0, 1, {{1, 1}}, 1, config_term.col(i),
                {{1, 1}});
            electrons.push_back(new_electron);
         }
         else if (config_term_cell[i] == Term::_2pMin1)
         {
            Electron new_electron(-1, 1, {{0, 1}}, 0, config_term.col(i),
                {{1, 1}});
            electrons.push_back(new_electron);
         }
         else if (config_term_cell[i] == Term::_3pMin1)
         {
            Electron new_electron(-1, 1, {{0, 1}}, 1, config_term.col(i),
                {{1, 1}});
            electrons.push_back(new_electron);
         }
         else if (config_term_cell[i] == Term::_3dMin2)
         {
            Electron new_electron(-2, 1, {{0, 1}}, 0, config_term.col(i),
                {{1, 1}});
            electrons.push_back(new_electron);
         }
         else if (config_term_cell[i] == Term::_2p0)
```

```
    {
        Electron new_electron(0, -1, {{1, 0}}, 0, config_term.col(i),
            {{1, 1}});
        electrons.push_back(new_electron);
    }
    else if (config_term_cell[i] == Term::_3p0)
    {
        Electron new_electron(0, -1, {{1, 0}}, 1, config_term.col(i),
            {{1, 1}});
        electrons.push_back(new_electron);
    }
    else if (config_term_cell[i] == Term::_3dMin1)
    {
        Electron new_electron(-1, -1, {{1, 0}}, 0, config_term.col(i),
            {{1, 1}});
        electrons.push_back(new_electron);
    }
    else if (config_term_cell[i] == Term::_4dMin1)
    {
        Electron new_electron(-1, -1, {{1, 0}}, 1, config_term.col(i),
            {{1, 1}});
        electrons.push_back(new_electron);
    }
    else if (config_term_cell[i] == Term::_4fMin2)
    {
        Electron new_electron(-2, -1, {{1, 0}}, 0, config_term.col(i),
            {{1, 1}});
        electrons.push_back(new_electron);
    }
    else if (config_term_cell[i] == Term::_5fMin2)
    {
        Electron new_electron(-2, -1, {{1, 0}}, 1, config_term.col(i),
            {{1, 1}});
        electrons.push_back(new_electron);
    }
}

// Armadillo doesn't have built-in 4d arrays.
// (Idea) Use a field of Cubes(3d array)
// exchange_Neumann.zeros(1, 2, Znucl, Znucl);
field<cube> exchange_Neumann(Znucl);
for (int i = 0; i < Znucl; i++)
{
```

```
   exchange_Neumann(i) = arma::cube(1, 2, Znucl, fill::zeros);
}

double betaZ2 = betaZ * betaZ;
mat msq = zeros(Znucl, 1);
mat delta_m_sq = zeros(Znucl, Znucl);

for (int i = 0; i < Znucl; i++)
   msq(i) = electrons[i].m * electrons[i].m;

for (int i = 0; i < Znucl; i++)
{
   for (int j = 0; j < Znucl; j++)
   {
      if (j == i)
         delta_m_sq(i, j) = (electrons[i].m - electrons[j].m) * (
             electrons[i].m - electrons[j].m);
   }
}

for (int i = 0; i < Znucl; i++)
{
   for (int j = 0; j < Znucl; j++)
   {
      if (i != j && delta_m_sq(i, j) != 0)
      {
         exchange_Neumann(j).slice(i) = {0, 0};
      }
      else if (i != j && delta_m_sq(i, j) == 0)
      {
         exchange_Neumann(j).slice(i) = {0, 1};
      }
   }
}

mat D;
mat x;
cheb(D, x, N);
mat D2 = D * D;

// D = D(2:N + 1, 2 : N + 1); //
// Remove the first row and first column.
D = D(span(1, N), span(1, N));
```

```
// D2 = D2(2:N + 1, 2 : N + 1); //
D2 = D2(span(1, N), span(1, N));

// x(end) = x(end) + tiny; //
x(N) += tiny;
// x = x(2:N + 1); //
x = x.rows(1, N);
colvec y = x;

uword rhomax_scaling = rhomax_scaling_ar(rhomax_count);
double rhomax = 100 * rhomax_scaling / (1 + log10(betaZ));
double alpha = 99 / rhomax;

//[xx, yy] = meshgrid(x, y); //
colvec xx = vectorise(repmat(x, 1, N).t(), 0);
colvec yy = vectorise(repmat(y, 1, N), 0);

// rho=(1/alpha)*(10.^(xx+1)-1); //
vec powers(xx.size(), fill::value(10.0));
vec rho = (1.0 / alpha) * (arma::pow(powers, xx + 1.0) - 1.0);
vec rhoinv = 1.0 / rho;
vec rho2 = square(rho);
vec rho2inv = 1.0 / rho2;
vec z = (1.0 / alpha) * (arma::pow(powers, yy + 1.0) - 1.0);
vec z2 = square(z);

vec a = -alpha * pow((1 / log(10)), 2) * rhoinv;
a = a % (arma::pow(powers, -xx - 1) - arma::pow(powers, -2 * xx - 2))
    ;
vec b = -alpha * (1 / log(10)) * rhoinv;
b = b % (arma::pow(powers, -2 * xx - 2));
vec c = -pow(alpha, 2) * pow((1 / log(10)), 2) * arma::pow(powers, -2
     * yy - 2);
vec d = pow(alpha, 2) * (1 / log(10)) * arma::pow(powers, -2 * yy -
    2);

mat ecoeff = zeros(N * N, Znucl);
for (int i = 0; i < Znucl; i++)
{
   // ecoeff(:,i)=msq(i)*rho2inv + betaZ2*rho2 -2./sqrt(rho2+z2);
   ecoeff.col(i) = msq(i) * rho2inv + betaZ2 * rho2 - 2 / arma::sqrt(
       rho2 + z2);
}
```

```
arma::cube e_exch = zeros(N * N, Znucl, Znucl);
for (int i = 0; i < Znucl; i++)
{
   for (int j = 0; j < Znucl; j++)
   {
      if (j != i)
      {
         e_exch.slice(j).col(i) = delta_m_sq(i, j) * rho2inv;
      }
   }
}

mat I = eye(N, N);
mat Dx2kr = kron(D2, I);
mat Dxkr = kron(D, I);
mat Dy2kr = kron(I, D2);
mat Dykr = kron(I, D);

mat aDx2 = diagmat(a) * Dx2kr;
mat bDx = diagmat(b) * Dxkr;
mat cDy2 = diagmat(c) * Dy2kr;
mat dDy = diagmat(d) * Dykr;

mat Lcore = aDx2 + bDx + cDy2 + dDy;
uvec rows_to_delete = regspace<uvec>(N - 1, N, N * (N - 1) - 1);
Lcore.shed_rows(rows_to_delete);
ecoeff.shed_rows(rows_to_delete);
for (uword i = N - 1; i < N * (N - 1); i += N)
   e_exch.shed_row(i);

uword Nsquared = (N - 1) * (N - 1); // (N-1)^2

// Define the C matrix
mat Cmtrx = zeros(Nsquared, N - 1);
for (int i = N - 1; i < N * (N - 1); i += N)
{
   Cmtrx.col(i / N) = Lcore.rows(0, Nsquared - 1).col(i);
}

// Next remove every Nth column in the first N* (N - 1) columns,
    included.
// Lcore(:, N : N : N * (N - 1)) = [];
```

```
Lcore.shed_cols(rows_to_delete);

// Define the different L matrices
cube Lmtrx = zeros(Nsquared + N, Nsquared + N, Znucl);
cube Lpmtrx = zeros(Nsquared + N, Nsquared + N, Znucl);
field<cube> Lexchng(Znucl);
for (int i = 0; i < Znucl; i++)
   Lexchng(i) = arma::cube(Nsquared + N, Nsquared + N, Znucl, fill::
       zeros);

for (int i = 0; i < Znucl; i++)
{
   Lmtrx.slice(i) = Lcore + diagmat(ecoeff.col(i));
   Lpmtrx.slice(i) = Lmtrx.slice(i) + 2 * betaZ * (electrons[i].m -
       1) * eye(Nsquared + N, Nsquared + N);

   for (int j = 0; j < Znucl; j++)
   {
      if (j != i)
      {
         Lexchng(j).slice(i) = -Lcore - diagmat(e_exch.slice(j).col(i)
             );
      }
   }
}

mat Ldirect = -Lcore;

// Define the matrix E
cube Emtrx = zeros(Nsquared, Nsquared, Znucl);
cube Epmtrx = zeros(Nsquared, Nsquared, Znucl);
field<cube> Eexchng(Znucl);
for (int i = 0; i < Znucl; i++)
   Eexchng(i) = arma::cube(Nsquared, Nsquared, Znucl, fill::zeros);

Emtrx.subcube(span(0, Nsquared - 1), span(0, Nsquared - 1), span::all
    ) = Lmtrx.subcube(span(0, Nsquared - 1), span(0, Nsquared - 1),
    span::all);
mat Edirect = Ldirect.rows(0, Nsquared - 1).cols(0, Nsquared - 1);
// For each cube in the 4d matrix
for (int i = 0; i < Eexchng.size(); i++)
   Eexchng(i).subcube(span(0, Nsquared - 1), span(0, Nsquared - 1),
       span::all) = Lexchng(i).subcube(span(0, Nsquared - 1), span(0,
```

```
      Nsquared - 1), span::all);
Epmtrx.subcube(span(0, Nsquared - 1), span(0, Nsquared - 1), span::
    all) = Lpmtrx.subcube(span(0, Nsquared - 1), span(0, Nsquared - 1)
    , span::all);

// Define the matrix En
mat ENmtrx = zeros(Nsquared, N);
mat ENtilde = zeros(Nsquared, N - 1);

// ENmtrx.submat(span(0,Nsquared-1),span(0,N-1)) = Lmtrx.subcube(span
    (0,Nsquared-1),span(Nsquared, Lmtrx.n_cols), span(0,0));
mat smat = Lmtrx.slice(0).rows(0, Nsquared - 1).cols(Nsquared, Lmtrx.
    n_cols - 1);
ENmtrx.cols(0, N - 1) = smat;
ENtilde = ENmtrx.cols(0, N - 2);
mat ENdirect = Ldirect.rows(0, Nsquared - 1).cols(Nsquared, Ldirect.
    n_cols - 1);
mat ENdirect_tilde = ENdirect.cols(0, N - 2);

// Set up boundary condition matrices
uword Nterm = N * (N - 1); // N * (N-1)
mat B = kron(D, I);
mat B0 = B.rows(Nterm, B.n_rows - 1).cols(0, Nterm - 1);
mat B2 = zeros(N, N - 1);
for (int i = N - 1; i < Nterm; i += N)
   B2.col(i / N) = B0.col(i);
uvec cols_to_shed = regspace<uvec>(N - 1, N, Nterm - 1);
B0.shed_cols(cols_to_shed);
mat BN = B.rows(Nterm, B.n_rows - 1).cols(Nterm, B.n_cols - 1);
mat BN_tilde = BN.submat(span(0, N - 2), span(0, N - 2));
mat B1 = B0;
mat B1_tilde = B1.submat(span(0, N - 2), span::all);

// Next set up boundary condition matrix in the y-direction
mat By = kron(I, D);
mat Bycopy = By;

// Define the H matrix
mat H = zeros(N - 1, N - 1);
for (int i = N - 1; i < Nterm; i += N)
   for (int j = N - 1; j < Nterm; j += N)
      H(i / N, j / N) = By(i, j);
```

```
Bycopy.shed_cols(cols_to_shed);

// Define the J matrices
mat Jfull = Bycopy.submat(span(0, Nterm - 1), span(Nterm - N + 1,
    Bycopy.n_cols - 1));
mat J = zeros(N - 1, N);
for (int i = N - 1; i < Nterm; i += N)
   J.row(i / N) = Jfull.row(i);

// Prune bycopy array -- get rid of the Jfull part
Bycopy = Bycopy.submat(span(0, Nterm - 1), span(0, Nterm - N));

// Define the G matrix
mat G = zeros(N - 1, Nterm - N + 1);
for (int i = N - 1; i < Nterm; i += N)
   G.row(i / N) = Bycopy.row(i);

// Define inverse matrices to be used
mat BNinv = inv(BN);
mat BN_tildeinv = inv(BN_tilde);
mat Hinv = inv(H);
mat HJBinv = inv(H - J * BNinv * B2);
mat GJB = G - J * BNinv * B1;
mat CHG = Cmtrx * Hinv * G;
mat EBHG = ENmtrx * (BNinv * (B1 - B2 * Hinv * G));
mat EBHGdir = ENdirect * (BNinv * (B1 - B2 * Hinv * G));
mat EBBtilde = ENtilde * BN_tildeinv * B1_tilde;
mat EBBtildedir = ENdirect_tilde * BN_tildeinv * B1_tilde;

// Setup and solve the eigenvalue problem
cube Mmtrx = zeros(Nsquared, Nsquared, Znucl);
cube Mdirect = zeros(Nsquared, Nsquared, Znucl);
field<cube> Mexchng(Znucl);
for (int i = 0; i < Znucl; i++)
   Mexchng(i) = arma::cube(Nsquared, Nsquared, Znucl, fill::zeros);

for (int i = 0; i < Znucl; i++)
{
   if (electrons[i].parity == 1)
   {
      Mmtrx.slice(i) = Emtrx.slice(i) - electrons[i].Neumann(0) * EBHG
           - electrons[i].Neumann(1) * (CHG);
```

```
      Mdirect.slice(i) = Edirect - electrons[i].direct_Neumann(0) *
          EBHGdir - electrons[i].direct_Neumann(1) * (-CHG);
   }
   else
   {
      Mmtrx.slice(i) = Emtrx.slice(i) - electrons[i].Neumann(0) *
          EBBtilde;
      Mdirect.slice(i) = Edirect - electrons[i].direct_Neumann(0) *
          EBBtildedir - electrons[i].direct_Neumann(1) * (-CHG);
   }
}

for (int i = 0; i < Znucl; i++)
{
   for (int j = 0; j < Znucl; j++)
   {
      if (j != i)
      {
         if (electrons[i].parity == 1 && electrons[j].parity == 1)
         {
            Mexchng(j).slice(i) = Eexchng(j).slice(i) -
                exchange_Neumann(j)(0, 0, i) * EBHGdir -
                exchange_Neumann(j)(0, 1, i) * (-CHG);
         }
         else
         {

            Mexchng(j).slice(i) = Eexchng(j).slice(i) -
                exchange_Neumann(j)(0, 0, i) * EBBtildedir -
                exchange_Neumann(j)(0, 1, i) * (-CHG);
         }
      }
   }
}

cube Mdinv = zeros(Nsquared, Nsquared, Znucl);
field<cube> Mexinv(Znucl);
for (int i = 0; i < Znucl; i++)
   Mexinv(i) = arma::cube(Nsquared, Nsquared, Znucl, fill::zeros);

for (int i = 0; i < Znucl; i++)
{
   Mdinv.slice(i) = inv(Mdirect.slice(i));
```

```
   for (int j = 0; j < Znucl; j++)
   {
      if (j != i)
         Mexinv(j).slice(i) = inv(Mexchng(j).slice(i));
   }
}

eigs_opts opts;
opts.maxiter = 550;
opts.tol = 0.00000001; // 1e-8;
uword neigvecs = 250;

cx_cube V(Nsquared, neigvecs, Znucl, fill::zeros);
cx_cube Lam(neigvecs, neigvecs, Znucl, fill::zeros);

arma::cx_vec eigval;
arma::cx_mat eigvec;

for (int i = 0; i < Znucl; i++)
{
   //[V(:, : , i), Lam(:, : , i)] = eigs(sparse(Mmtrx(:, : , i)),
       neigvecs, sigma_guess, opts);
   sp_mat spMi = sp_mat(Mmtrx.slice(i));
   bool status = eigs_gen(eigval, eigvec, spMi, neigvecs, sigma_guess
       , opts);
   if (!status)
   {
      std::cerr << "eig_gen failed for slice " << i << std::endl;
      continue;
   }

   V.slice(i) = eigvec;
   Lam.slice(i) = diagmat(eigval);
}

cx_vec Lamsort(neigvecs * Znucl, fill::zeros);
for (int i = 0; i < Znucl; i++)
{
   cx_vec diagonal = Lam.slice(i).diag();
   Lamsort = sort(diagonal);
   uvec ii = sort_index(diagonal);

   cx_vec llam = Lamsort + 2 * betaZ * (electrons[i].m - 1);
```

```
                uvec b = find(real(llam) < 0);

                vec newlam = real(llam(b));

                // V1=V(:,ii(b),i);
                mat V1 = real(V.slice(i).cols(ii.elem(b)));

                // bimag=find(imag(newlam)~=0);
                uvec bimag = find(imag(newlam) != 0);

                // V1(:,bimag)=[];
                V1.shed_cols(bimag);
                newlam.shed_rows(bimag);

                electrons[i].eigval = real(llam(electrons[i].excitation));

                if (i == 0)
                   electrons[i].eigvec = sort(V1.col(electrons[i].excitation));
                else
                   electrons[i].eigvec = sort(-V1.col(electrons[i].excitation));

                electrons[i].Ehyd = real(llam(electrons[i].excitation));
                electrons[i].eigval_ehyd = sort(V1.col(electrons[i].excitation));
             }
          }
       }
    }
}
```