1 Atomic physics – hydrogen-like ions

- Here: focus on solving Schrodinger equation for single-electron wavefunction (e.g., hydrogen)
- Useful, even for more complex atoms, since we use single-electron wavefunctions to build multi-electron wavefunctions

1.1 Hydrogenlike ions: quick review

Very quick review here. For more details, see, e.g., textbook [W. R. Johnson, *Atomic Structure Theory* (2007)], available as pdf from library.

1.1.1 Angular separation

In general, we have the Schrodinger equation:

$$\hat{H}\psi = \varepsilon\psi, \tag{1}$$

where

$$\hat{H}(\boldsymbol{r}) = \frac{\boldsymbol{p}^2}{2m} + V(\boldsymbol{r}) \tag{2}$$

with $\boldsymbol{p} = -i\hbar\boldsymbol{\nabla}$.

For single-electron atoms, the potential is entirely spherically symmetric, meaning:

$$V(\boldsymbol{r}) = V(r) = -\frac{Ze^2}{4\pi\epsilon_0\, r}. \tag{3}$$

Therefore, it is simplest to work in spherical coordinates, in which

$$\nabla^2 = \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial}{\partial r}\right) + \frac{1}{r^2\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial}{\partial\theta}\right) + \frac{1}{r^2\sin^2\theta}\sin\theta\frac{\partial^2}{\partial\phi^2}.$$

Motivated by the spherical symmetry, we use the separation of variables for ψ:

$$\psi(\boldsymbol{r}) = R(r)Y(\theta,\phi) = \frac{P(r)}{r}Y(\theta,\phi). \tag{4}$$

Putting this back into the Hamiltonian equation, leads to two equations:

$$\frac{1}{\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial Y}{\partial\theta}\right) + \frac{1}{\sin^2\theta}\frac{\partial^2 Y}{\partial\phi^2} + \lambda Y = 0, \tag{5}$$

and

$$\frac{\partial^2 P}{\partial r^2} + \frac{2m}{\hbar^2}\left(\varepsilon - V(r) - \frac{\lambda\hbar^2}{2mr^2}\right)P = 0, \tag{6}$$

where λ is a separation constant.

Notice that Eq. (5) is independent of the potential V and the energy ε; therefore the solutions are always the same. The solutions are the spherical harmonics

$$Y = Y_{lm} \tag{7}$$

for integer values of $\lambda \equiv l(l+1)$ and m. l takes values $0, 1, 2, 3, ...$, and m takes the values $-l, ..., 0, ..., l$. This is not proved here, but is done in any quantum mechanics textbook.

The spherical harmonics are orbital momentum eigenstates according to:

$$L^2|lm\rangle = \hbar^2 l(l+1)|lm\rangle \tag{8}$$
$$L_z|lm\rangle = \hbar m|lm\rangle \tag{9}$$

where $\boldsymbol{L} = \boldsymbol{r} \times \boldsymbol{p}$ is the operator of (orbital) angular momentum, and using Dirac notation so that $|lm\rangle$ represents spherical harmonic Y_{lm}. Therefore, we recognise l as the total (orbital) angular momentum, and m as its projection onto z quantisation axis. These are normalised according to

$$\int Y^*_{l'm'} Y_{lm} \, \mathrm{d}\Omega = \delta_{ll'}\delta_{mm'}, \tag{10}$$

($\mathrm{d}\Omega = \sin\theta \mathrm{d}\phi \mathrm{d}\theta$).

Thus, the problem reduces to solving the radial equation (6), which depends on the potential V, which we can write as:

$$\underbrace{\left[\frac{-\hbar^2}{2m}\frac{\partial^2}{\partial r^2} + V(r) + \frac{l(l+1)\hbar^2}{2mr^2}\right]}_{H_r} P(r) = \varepsilon P(r). \tag{11}$$

We may therefore define the "radial Schrodinger equation", $H_r P = \varepsilon P$. For a given l, the equation has an infinite number of solutions – we label the bound-state solutions with n (principal quantum number) P_{nl}, $n = 1, 2, 3, ...$, with $n > l$. They are normalised as

$$\int P_{n'l} P_{nl} \, \mathrm{d}r = \delta_{nn'}. \tag{12}$$

For Hydrogenlike ions, $V(r) = -\frac{Ze^2}{4\pi\epsilon_0 \, r}$, and the solutions for the energies are

$$\varepsilon_n = -\frac{Z^2 R_y}{n^2}, \tag{13}$$

where $R_y = \frac{m_e e^4}{8(\pi\epsilon_0\hbar)^2} \approx 13.6\,\mathrm{eV}$.

1.1.2 Bound-state solutions

Note that Eq. (1) has an infinite number of solutions for any given ε. However, we are interested in the bound state solutions, which have $\psi \to 0$ as $r \to \infty$, and ψ normalisable (and thus ψ everywhere finite).

- $\psi(\boldsymbol{r}) \to 0$ as $|r| \to \infty$
- $\psi(\boldsymbol{r})$ finite as $|r| \to 0$ – and continuous

The boundary conditions become simpler in terms of P:

- $P(r) \to 0$ as $|r| \to \infty$
- $P(r) \to 0$ finite as $|r| \to 0$

More explicitly

- $P_{nl}(r) \sim r^{l+1}$ as $|r| \to 0$
- $P_{nl}(r) \sim \exp(-\sqrt{2|\varepsilon|})$ finite as $|r| \to \infty$

Interestingly, these conditions hold for any atom, even multi-electron atoms. (The reason is that, for multi-electron atoms we still have $V(r) \sim -Z/r$ for small r, $V(r) \sim -1/r$ for large r.)

For bound states. we have $\varepsilon < 0$. There are also unbound (continuum) states, which have $\varepsilon > 0$, and $P(r) \sim sin(\omega r)$ for large r – though we will not be directly concerned with these. The full set of solutions (including continuum) form a closed, complete, orthogonal set.

1.1.3 Matrix elements, expectation values

Typically, in any quantum mechanics problem, we are interested in calculating amplitudes (known as matrix elements):

$$\langle a|\hat{h}|b\rangle = \int \psi^*_{n_a l_a m_a} \hat{h}\, \psi_{n_b l_b m_b}\, \mathrm{d}V. \tag{14}$$

Here, we will consider only the simplest case: radial operators $\hat{h} = h(r)$ (sometimes called scalar operators).

In this case, we have

$$\int \psi^*_{n_a l_a m_a} \hat{h}\, \psi_{n_b l_b m_b}\, \mathrm{d}V = \left(\int P_{n_a l_a} h(r)\, P_{n_b l_b}\, \mathrm{d}r\right)\left(\int Y^*_{l_a m_a} Y_{l_b m_b}\, \mathrm{d}\Omega\right) \tag{15}$$

$$= \left(\int P_{n_a l_a} h(r)\, P_{n_b l_b}\, \mathrm{d}r\right) \delta_{l_a l_b}\delta_{m_a m_b}. \tag{16}$$

Therefore, matrix elements for such operators only depend on radial P functions; and are non-zero only transitions in which the angular quantum numbers do not change.

1.1.4 Atomic units

- Measure mass in units of m_e
- Spin in units of $\hbar$
- Length in units of Bohr radius $a_B = \frac{4\pi\epsilon_0\hbar^2}{m_e e^2}$
- Charge in units of $|e|$, with $4\pi\epsilon_0 = 1$
- Energy in Hartree units, $2R_y = \frac{m_e e^4}{4(\pi\epsilon_0\hbar)^2} \approx 27.2\,\mathrm{eV}$

In these units, important constants take the values: $m_e = |e| = 4\pi\epsilon_0 = \hbar = a_B = 1$, and speed of light $c = 1/\alpha$, where $\alpha = \frac{e^2}{4\pi\epsilon_0\hbar c} \approx 1/137$ is the fine structure constant.

In these units, the radial Schrodinger equation takes the form:

$$\left[\frac{-1}{2}\frac{\partial^2}{\partial r^2} + V(r) + \frac{l(l+1)\hbar^2}{2r^2}\right] P(r) = \varepsilon P(r), \tag{17}$$

For Hydrogenlike ions, we again have:

$$V(r) = -\frac{Z}{r} \tag{18}$$

and

$$\varepsilon_n = -\frac{Z^2}{2n^2}. \tag{19}$$

1.2 Numerical solution to radial equation

$$\left[\frac{-1}{2}\frac{\partial^2}{\partial r^2} + V(r) + \frac{l(l+1)\hbar^2}{2r^2}\right] P(r) = \varepsilon P(r), \tag{20}$$

For hydrogenlike ions, this can be solved exactly. But for more general potentials, it cannot be. This will be the starting point for more complex systems - only $V(r)$ will change. The same techniques apply for general $V(r)$ potentials.

There are many methods available to solve the equation; here I will outline a few briefly.

1.2.1 Linear multi-step method

Define

$$Q(r) \equiv \frac{\mathrm{d}P}{\mathrm{d}r}, \qquad y(r) \equiv \begin{pmatrix} P(r) \\ Q(r) \end{pmatrix}. \tag{21}$$

Then, the radial Schrodinger equation can now be expressed

$$\frac{\mathrm{d}y}{\mathrm{d}r} = \begin{pmatrix} Q(r) \\ -2\left(\varepsilon - V - \frac{l(l+1)}{2r^2}\right) P(r) \end{pmatrix} \tag{22}$$

$$= D(r)y(r), \tag{23}$$

where D is the 2x2 matrix (that depends on r)

$$D = \begin{pmatrix} 0 & 1 \\ -2\left(\varepsilon - V - \frac{l(l+1)}{2r^2}\right) & 0 \end{pmatrix} \tag{24}$$

This has form of first-order ODE for y, but is a matrix equation.

Assume we know $y(r)$, and want to find $y(r + \Delta r)$. Since we know the derivative, we can project forwards:

$$y(r + \Delta r) \approx y(r) + \frac{\mathrm{d}y}{\mathrm{d}r}\Delta r, \tag{25}$$

or more precisely:

$$y(r + \Delta r) \approx y(r) + \int_r^{r+\Delta r} D(r')y(r')\,\mathrm{d}r'. \tag{26}$$

Using an $(N + 1)$-step numerical integration method (more next lecture), with $\Delta r = N\delta r$, we have

$$y(r + \Delta r) \approx y(r + (N-1)\delta r) + \delta r \sum_{i=0}^{N} b_i D(r + i\delta r) y(r + i\delta r). \tag{27}$$

The b_i are numerical integration "weights" – will be discussed next lecture. We move the $i = N$ term to the left, and solve for $y(r + \Delta r)$:

$$y(r + \Delta r) \approx [1 - \delta r b_N D(r + N\delta r)]^{-1} \left(y(r + (N-1)\delta r) + \delta r \sum_{i=0}^{N-1} D(r + i\delta r) y(r + i\delta r). \right) \tag{28}$$

Therefore, we can approximate $y(r + \Delta r)$ so long as N previous values $y(r)$, $y(r + \delta r)$, $y(r + 2\delta r)$,... are known. Note that it involves finding the inverse of a 2x2 matrix.

- Very accurate
- Need N initial points: e.g., Use low-r expansion

- Derivative operator depends on ε

Since the derivative operator depends on ε, we cannot solve for P and ε at the same time. Instead, we have to guess ε, and then solve the equation. Most likely, this guess will not be correct – this means the solution will not have the correct boundary condition. Keep making small adjustments to the guessed ε until the boundary conditions match – then we have successfully solved for the bound state.

1.2.2 Diagonalise over basis

Use some (finite) set of basis functions $\{S_i(r)\}$, and write:

$$P(r) = \sum_i c_i S_i(r). \tag{29}$$

In general, this is approximate, since the basis set is finite.

Sub this in to the radial Schrodinger equation:

$$HP = \varepsilon P.$$

Using Dirac notation, this gives:

$$\sum_i H|S_i\rangle c_i = \varepsilon \sum_j |S_j\rangle c_j \tag{30}$$

$$\sum_i \langle S_k|H|S_i\rangle c_i = \varepsilon \sum_j \langle S_k|S_j\rangle c_j, \tag{31}$$

where we multiplied on the left by $\langle S_k|$ (and integrated). The result is just a matrix equation:

$$\sum_i H_{ki} c_i = \varepsilon \sum_j S_{kj} c_j \tag{32}$$

$$\implies H\boldsymbol{c} = \varepsilon S\boldsymbol{c}. \tag{33}$$

This is a generalised eigenvalue matrix equation, which can be solved to yield a set of eigenvalues ε, and eigen vectors $\boldsymbol{c}$. Each eigenvector is the set of c_i expansion coefficients. Note that if the basis set if orthonormal, the S matrix $S_{ij} = \langle S_i|S_j\rangle$ is just the identity; however, this is not true in general.

In theory, any basis set will do (e.g., polynomials). In practice, since we can only use a finite set, choosing a good basis set is important. Good choices are Gaussian basis states, B-splines, or hydrogen-like wavefunctions.

1.2.3 Cast derivative operator to finite-element matrix (finite difference method)

- Not most accurate, but simplest
- Use for assignment!

We can numerically approximate the first- and second-order derivative as:

$$\frac{\mathrm{d}f}{\mathrm{d}r} \approx \frac{f(x+\delta/2) - f(x-\delta/2)}{\delta} \tag{34}$$

$$\frac{\mathrm{d}^2 f}{\mathrm{d}r^2} \approx \frac{f(x-\delta) - 2f(x) + f(x+\delta)}{\delta^2}. \tag{35}$$

If we choose a finite spacing $\{..., x_1, x_2, x_3, ..., x_n, ...\}$, where each point is spaced δ apart: $x_{n+1} = x_n + \delta$, we may write the function f as a vector in this finite space:

$$f = \begin{pmatrix} \vdots \\ f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_n) \\ \vdots \end{pmatrix}. \tag{36}$$

Then, we may cast the derivative as a matrix multiplication:

$$\frac{\mathrm{d}^2 f}{\mathrm{d}r^2} \approx \frac{1}{\delta^2} \begin{pmatrix} & & \ddots & \vdots & & & \\ \dots & 0 & 1 & -2 & 1 & 0 & \dots \\ & & & \vdots & \ddots & & \end{pmatrix} \begin{pmatrix} \vdots \\ f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \end{pmatrix} = \frac{1}{\delta^2} \begin{pmatrix} \vdots \\ f(x_0) - 2f(x_1) + f(x_2) \\ f(x_1) - 2f(x_2) + f(x_3) \\ f(x_2) - 2f(x_3) + f(x_4) \\ \vdots \end{pmatrix}. \tag{37}$$

Take a moment to ensure this makes sense – you have seen the same thing in the previous lectures.

Of course, we cannot compute a matrix of infinite size. Therefore, we must choose a finite spacing δ. Further, we must truncate the vector somewhere. We can do this by assuming $f(x) = 0$ for $x < x_{\rm min}$ and $x > x_{\rm max}$. This is called the *hard boundary condition.*

In this case, we have a finite-dimensional matrix equation for the derivative:

$$\frac{\mathrm{d}^2 f}{\mathrm{d}r^2} \approx \frac{1}{\delta^2} \begin{pmatrix} -2 & 1 & 0 & \dots & & & \\ 1 & -2 & 1 & 0 & \dots & & \\ 0 & 1 & -2 & 1 & 0 & \dots & \\ & & & & \ddots & & \\ & & \dots & 0 & 1 & -2 & \end{pmatrix} \begin{pmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_n) \end{pmatrix} = \frac{1}{\delta^2} \begin{pmatrix} -2f(x_1) + f(x_2) \\ f(x_1) - 2f(x_2) + f(x_3) \\ f(x_2) - 2f(x_3) + f(x_4) \\ \vdots \\ f(x_{n-1}) - 2f(x_n) \end{pmatrix}. \tag{38}$$

We use n equally spaced points from x_1 to x_n, so that δ,

$$\delta = \frac{x_n - x_1}{n - 1}. \tag{39}$$

Using this, we may re-cast the radial Schrodiner equation into a matrix equation:

$$\left[\frac{-1}{2} D_2 + V(r) + \frac{l(l+1)}{2r^2} \right] P = \varepsilon P, \tag{40}$$

where D_2 in the $n \times n$ second-order derivative matrix. This is a simple eigenvalue problem, which can be solved using usual techniques. Notice that, in order for $V(r) * f(r)$ to work, $V(r)$ (and any other function of r) can be cast simply to a diagonal matrix:

$$\begin{pmatrix} V(x_1) & & & & 0 \\ & V(x_2) & & & \\ & & V(x_3) & & \\ & & & \ddots & \\ 0 & & & & V(x_n) \end{pmatrix}. \tag{41}$$

Since the finite-element Hamiltonian matrix

$$H = \left[\frac{-1}{2}D_2 + V(r) + \frac{l(l+1)}{2r^2}\right]$$

is real and symmetric, we can use the LAPACK routine DSYEV.

The output will be a set of n eigenvalues, and n eigenvectors. The eigenvalues are the energies, and the eigenvectors are the wavefunctions (evaluated at the discrete x_i points).

Example 1.1: DSYEV Parameters. Documentation: http://www.netlib.org/lapack/explore-html/index.html.

```
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.
);
```

- Output eigenvalues are sorted, and eigenvectors are normalised to 1 (via inner product)
- 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 (on macos, you may also need the -lblas flag)

Note that due the the "hard boundary condition", we have $P(r) = 0$ for $r > r_{\max}$. However, the correct boundary condition should be $P(r_n) \sim \exp(-\sqrt{2|\varepsilon|}r_n)$. This is not a problem, so long as all the wavefunctions we are directly interested in effectively go to zero well inside the maximum radius r_n. i.e., we must choose r_n to be much larger than the typical radius of the electron orbitals we are interested in. The expectation value for r scales as

$$\langle r\rangle \sim \frac{n^2}{Z}a_B.$$

Note that by solving the equation in this way, we have generated a full set of wavefunctions (eigenvectors). We are typically only directly interested in the first few. Of course, since we approximated the derivative using finite-elements, they are not exact, however, we have roughly a full set of orthonormal wavefunctions. This is important, since we often need to use a complete set of states to sum over in higher-orders of perturbation theory. Recall for a perturbative expansion $H \to H + \delta h$, $\psi \to \psi_0 + \delta\psi$, $\varepsilon \to \varepsilon + \delta\varepsilon$, we have:

$$|\delta\psi\rangle = \sum_n \frac{|n\rangle\langle n|\delta h|\psi_0\rangle}{\varepsilon_0 - \varepsilon_n}. \tag{42}$$

1.3 Example: finite difference method: solve DE as matrix eigenvalue problem

Here, I present some example code to interface with DSYEV. I first write a very simple class to store the matrix; I use std::vector to store the data, and write a .at() function to read/write to the matrix elements. Of course, you do not need to do it this way, but I present this just as an example. Then, I show an example function for writing the derivative operator as a matrix.

Then, I show an example function, which uses the above matrix class, that interfaces with DSYEV to find eigenvalues and eigenvectors. I also define a simple struct 'Eigen' that holds both the vector of eigenvalues, and the matrix of eigenvectors.

This and more examples are on the blackboard site, under the week 6 workshop.

Example 1.2: Very basic class to hold square matrix (don't forget header guards, excluded here for brevity).

```
// matrix.hpp
#include <vector>

class SquareMatrix {
public:
  int dimension;
  std::vector<double> vec; // store data in std::vector called 'vec'

  // Constructore: Initialise a SquareMatrix of dimension*dimension, with 0s
  SquareMatrix(int in_dimension)
      : dimension(in_dimension), vec(in_dimension * in_dimension) {}

  // Access the (i,j)th element by reference (so can edit)
  double &at(int i, int j) { return vec.at(i * dimension + j); }
};
```

Example 1.3: We may write a simple function that creates second-order derivative operator matrix. This can be easily modified to return the entire Hamiltonian operator as a matrix (in general, you'll need to pass more information to the function, like $r_{\rm min}$ and/or $r_{\rm max}$).

```
SquareMatrix form_d2(int dimension, double dr) {
  // Create empty square matrix, called d2
  // Note: All elements set to zero by default, see SquareMatrix
  SquareMatrix d2(dimension);
  const auto dr2 = dr * dr;
  // Fill all non-zero elements (matrix set all)
  for (int i = 0; i < dimension; ++i) {
    // Diagonal parts:
    d2.at(i, i) = -2.0 / dr2;
    // off-diagonal parts (note: careful not to go outside matrix bounds!)
    if (i - 1 >= 0) {
      d2.at(i, i - 1) = 1.0 / dr2;
    }
    if (i + 1 < dimension) {
      d2.at(i, i + 1) = 1.0 / dr2;
    }
  }
  return d2;
}
```

Example 1.4: Basic wrapper function that interfaces with DSYEV.

```cpp
// eigensystems.hpp
#include "matrix.hpp"
// Very simple class; stores eigen values and eigen vectors (put in header file)
struct Eigen {
  std::vector<double> values;
  SquareMatrix vectors;
  // constructor:
  Eigen(int dimension) : values(dimension), vectors(dimension) {}
};

Eigen RealSymmetric(SquareMatrix matrix);
//---------------------------------------------------------------------------
// eigensystems.cpp
#include "eigensystems.hpp"
#include <iostream>

// dsyev_ is a symbol in the LAPACK library files
// Documentation: http://www.netlib.org/lapack/explore-html/index.html
extern "C" {
extern int dsyev_(char *, char *, int *, double *, int *, double *, double *,
                  int *, int *);
}

// Finds eigen values and eigen vectors of real, symmetric matrix
Eigen RealSymmetric(SquareMatrix matrix) {
  // Create empty 'Eigen' object, where we store the result
  Eigen result(matrix.dimension);

  // Data required by dsyev. See documentation for description of each option:
  char jobz{'V'};
  char uplo{'U'};
  int dimension = matrix.dimension;
  int lwork = 6 * dimension;
  std::vector<double> work(lwork);
  int info;

  // calculate eigenvalues using the DSYEV lapack subroutine
  dsyev_(&jobz, &uplo, &dimension, matrix.vec.data(), &dimension,
         result.values.data(), work.data(), &lwork, &info);

  // 'matrix' contained the matrix on input, and contains a matrix of
  // eigenvectors on output.. We copy these over into the 'result' matrix (use
  // move to avoid copy is optional)
  result.vectors.vec = std::move(matrix.vec);

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

  return result;
}
```