Infinite Quantum Well: Shooting Methods

Boundary-value and eigenvalue problems

Second order ordinary differential equations of the general form

$$\frac{d^2y}{dx^2} = f(y, y'; x) \,, \tag{1}$$

where $y'(x) = dy/dx$ are commonly encountered in many applications. Here, the independent variable x represents a domain with fixed boundaries $a \leq x \leq b$. With a simple change of variable, the domain can be mapped to $[0, 1]$. The solution of the differential equation is fixed by specifying the behavior of the dependent variable $y(x)$ and/or its derivative $y'(x)$ at the two boundaries.

- Dirichlet boundary conditions specify $y(0)$ and $y(1)$.
- Neumann boundary condition specify $y'(0)$ and $y'(1)$.
- Periodic boundary conditions set $y(0) = y(1)$ and $y'(0) = y'(1)$.
- Mixed boundary conditions which involve combinations of Dirichlet and Neumann are also possible.

Numerical methods to solve boundary values problems are discussed in Chapter 17 of Numerical Recipes[1].

An *eigenvalue problem* results when the equation depends on a parameter λ

$$\frac{d^2y}{dx^2} = f(y, y'; x; \lambda) \,, \tag{2}$$

and the boundary conditions are such that solutions exist only for particular values of λ called eigenvalues. This generally happens when the boundary conditions are *homogeneous*, i.e., of the form $a_0y(0) + b_0y'(0) = 0$ and $a_1y(1) + b_1y'(1) = 0$, where a_i, b_i are constants.

The one-dimensional time-independent Schrödinger equation

$$\frac{d^2\psi}{dx^2} = -\frac{2m}{\hbar^2}\left[E - V(x)\right]\psi(x) \,, \tag{3}$$

where $V(x)$ is a given potential function, has bound state solutions only for particular eigenvalues of the energy E.

Shooting Methods and Relaxation Methods

Because $y(0)$ and $y'(0)$ are not specified simultaneously, it is not possible to use a *marching algorithm* such as Euler's or Runge-Kutta to start the solution at $x = 0$ and march it step by step to $x = 1$. A more complex iterative procedure must be used to generate the solution.

A *shooting procedure* starts with a guess for the unknown initial value parameter. For example, if $y(0) = y_0$ is specified, guess a value for $y'(0) = \delta$. Generate a trial solution using a marching algorithm. This solution $y_\delta(x)$ will depend on δ. The difference

$$F(\delta) = y_\delta(1) - y_1 \tag{4}$$

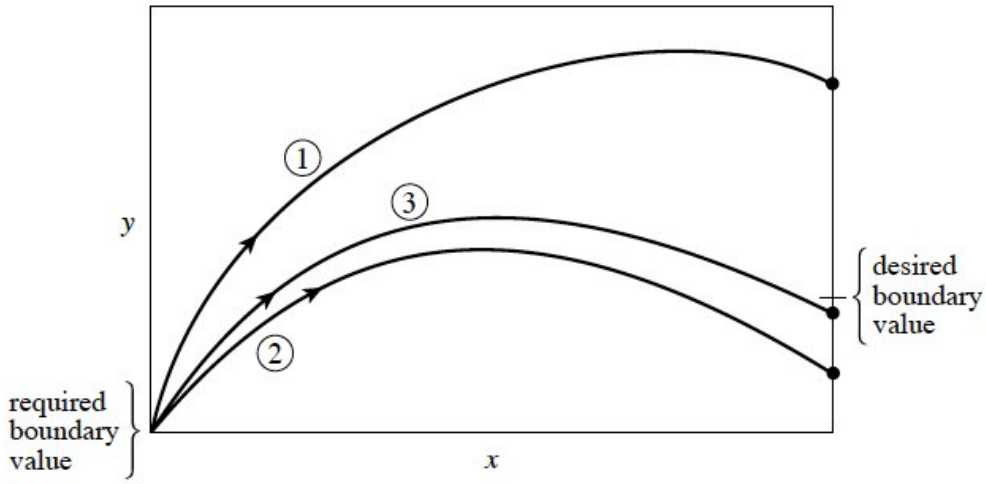

Figure 1: From Numerical Recipes[1]. Trial solutions that satisfy the required boundary condition at the left boundary are generated. The discrepancy from the desired boundary condition at the right boundary is reduced iteratively to zero.

from the desired boundary value $y(1) = y_1$ at the other boundary will also depend on δ. The correct solution will satisfy $F(\delta) = 0$, i.e., it is the *root* of the function F. Fig. 1 illustrates the general concept.

A *relaxation procedure* starts with a guess for the solution $y_{\rm g}(x)$ at all values of x, which also satisfies the boundary conditions. This function will in general *not* satisfy the differential equation. The discrepancy

$$G(x) = \frac{d^2 y_{\rm g}}{dx^2} - f\left(y_{\rm g}, y'_{\rm g}; x\right) \ , \tag{5}$$

is computed at interior points $0 < x < 1$, usually on a lattice using a finite difference approximation. Iterative adjustments are made to the guess at each interior point such that the discrepancy function tends to zero. The guessed solution thus *relaxes* to the exact solution.

Root Finding

A root of a function $f(x)$ is a value of its argument x at which the value of the function equals zero.

Finding the roots of a function numerically can be problematic:

- Some functions, such as $f(x) = 1 + |x|$, may not have a root at all!
- Some functions, such as $1 + x^2$ and $\cosh(x)$, do have roots, but not for real values of x.
- There are functions, such as $\sin(1/x)$, which have a large or infinite number of roots in a finite interval of x, and one is then faced with the problem of deciding exactly which of these roots is to be found.

Thus to find a root of a function, make sure the root exists, and have at least a rough idea about where the root is located in order to compute its value efficiently.

It pays to make a rough plot of the function, either using some simple analytic approximation, or by the brute-force method of evaluating the function at a large number of points and plotting the values graphically.

Once the root has been located approximately, there are efficient numerical algorithms to locate it as precisely as required. Chapter 9 of Numerical Recipes[2] describes many such algorithms.

Bisection Search

This algorithm assumes that

1. the function has a root at which it changes sign,
2. that the root can be *bracketed* by two values x_0 and x_1 such that $x_0 <$ the root $< x_1$, and
3. that there are no other roots in the interval $[x_0, x_1]$.

The algorithm proceeds by repeatedly bisecting the interval:

- Let $x_{\frac{1}{2}} = (x_0 + x_1)/2$ be the bisection point.
- Compute the product $f(x_0) \times f(x_{\frac{1}{2}})$.
  - If this product is positive, then x_0 and $x_{\frac{1}{2}}$ are on the same side of the root, and $x_{\frac{1}{2}}$ is obviously closer to it. Therefore replace $x_0 \to x_{\frac{1}{2}}$.
  - If the product is not positive, then x_0 and $x_{\frac{1}{2}}$ are on opposite sides of the root, and x_1 and $x_{\frac{1}{2}}$ must therefore be on the same side of the root with $x_{\frac{1}{2}}$ closer to it. Therefore replace $x_1 \to x_{\frac{1}{2}}$.
- If $|x_1 - x_0|$ is smaller than some desired accuracy ϵ, or if $f(x_{\frac{1}{2}})$ happens to equal zero, then the root has been located with sufficient precision, and the search can be halted. If not, then steps 1–3 are repeated.

Bound States in an Infinitely Deep Quantum Well

Consider the potential

$$V(x) = \begin{cases} 0 & \text{for } 0 \le x \le 1 \\ \infty & \text{otherwise} \end{cases} , \tag{6}$$

and choose units such that $\hbar^2/(2m) = 1$ and the well has unit width. Then Schrödinger's equation is

$$\frac{d^2\psi}{dx^2} = [V(x) - E]\,\psi \,, \qquad \psi(0) = \psi(1) = 0 \,. \tag{7}$$

C++ Infinite Well Program

The following C++ code is organized using several small functions:

- `double V(double x)` calculates the potential $V(x)$.
- `vector<double> flow(const vector<double>& y, const double x)` computes the derivative vector

$$\frac{d}{dx}\begin{pmatrix}\psi(x)\\ \psi'(x)\end{pmatrix} = \begin{pmatrix}\psi'(x)\\ \frac{-2m}{\hbar^2}\,[E - V(x)]\end{pmatrix} \tag{8}$$

  for a given E, not necessarily an eigenvalue.

- `double F(double E)` computes the search function whose roots are the energy eigenvalues. Starting with inItial values $\psi(0) = 1$ and $\psi'(0) = 1$, it integrates the Schrödinger equation using an adaptive fourth-order Runge-Kutta method, and returns $\psi(1)$.
- `double eigenvalue(double E0, double E1, bool print_steps)` implements the bisection search method to find an eigenvalue given bracketing guesses.
- `void print(int step, double x, double dx)` is an convenience function used by `eigenvalue`.

Program 1: `http://www.physics.buffalo.edu/phy410-505/topic5/infwell.cpp`

```
#include <cmath>
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

#include "odeint.hpp"           // to use 4th order Runge-Kutta
using namespace cpl;

double accuracy;                // for integration and root finding

double V(                       // potential function
  double x
) {
  if (x >= 0 && x <= 1)
    return 0;
  else return 1e10;             // large value
}

double E;                       // current value of E for root finding

vector<double> flow(            // flow vector for Runge-Kutta integration
  const vector<double>& y,
  const double x
) {
  double psi = y[0], psi_prime = y[1];
  vector<double> flow(2);
  flow[0] = psi_prime;
  flow[1] = (V(x) - E) * psi;
  return flow;
}

double F(                       // search function for root finding
  double E
) {
  ::E = E;                      // set global variable needed by flow
  RK4 rk4;
  rk4.set_step_size(0.001);
  rk4.set_accuracy(accuracy);
  vector<double> y(2);          // (psi(x), psi'(x))
  y[0] = 0;                     // psi(0)
```

```
  y[1] = 1;                     // arbitrary value of slope
  rk4.integrate(flow, y, 0, 1); // integrate Schroedinger equation
  return y[0];                  // looking for psi(1) = 0
}

void print(int step, double x, double dx) {
  int precision = cout.precision();
  cout.setf(ios::right, ios::adjustfield);
  cout << " " << setw(4) << step << "    ";
  cout.precision(15);
  cout.setf(ios::left, ios::adjustfield);
  cout.setf(ios::showpoint | ios::fixed);
  cout << setw(20) << x << "  " << setw(20) << dx << '\n';
  cout.precision(precision);
}

double eigenvalue(              // find eigenvalue using bisection search
  double E0,                    // guess for lower bracketing energy
  double E1,                    // guess for higher bracketing energy
  bool print_steps              // print steps if true
) {

  if (print_steps) {
    cout << "\n Using Bisection Search ...\n"
         << " Step            E                    dE\n"
         << " ----    ------------------    ------------------\n";
  }

  int step = 0;
  double E_half = (E0 + E1) / 2;
  double dE = E1 - E0;
  if (print_steps)
    print(step, E_half, dE);
  double F0 = F(E0);

  // bisection search
  while (abs(dE) > abs(accuracy)) {
    double F_half = F(E_half);
    if (F_half == 0) {
      dE = 0;
    } else {
      if (F0 * F_half > 0) {
        E0 = E_half;
        F0 = F_half;
      } else {
        E1 = E_half;
      }
      E_half = (E0 + E1) / 2;
      dE = E1 - E0;
    }
    ++step;
```

```
    if (print_steps)
      print(step, E_half, dE);
  }
  if (print_steps) {
    cout << " ------------------------------------------------\n";
  }

  return E_half;
}

int main() {

  cout << " Bound state energies of infinitely deep potential well\n"
       << " ------------------------------------------------------\n"
       << " Enter bracketing guesses E_0, E_1, and desired accuracy: ";
  double E0, E1;
  cin >> E0 >> E1 >> accuracy;

  cout << " Eigenvalue E = " << eigenvalue(E0, E1, true) << endl;

}
```

Homework Problem

Add code to infwell.cpp to compute the eigenfunction $\psi_n(x)$ after it has found an eigenvalue E_n. Make plots of the normalized eigenfunctions for a few of the lowest eigenstates and compare with the corresponding analytic solutions from your quantum mechanics textbook.

References

[1] W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, "Numerical Recipes in C" (Cambridge University Press 1992), §17.0 Two Point Boundary Value Problems, `http://www.nrbook.com/a/bookcpdf/c17-0.pdf`.

[2] W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, "Numerical Recipes in C" (Cambridge University Press 1992), §9.0 Root Finding and Nonlinear Sets of Equations, `http://www.nrbook.com/a/bookcpdf/c9-0.pdf`.