PHYS580 Homework 6

Yicheng Feng
PUID: 0030193826

December 4, 2019

Workflow: I use the Linux system, and code C++ in terminals. For visualization, I use the C++ package – ROOT (made by CERN) to make plots. At the end, the reports are written in LaTeX.

The codes for this lab are written as the following files:

- `square_well_1d.h` and `square_well_1d.cxx` for the class `SquareWell1D` to calculate the bound states with various wall potentials for the square wells.
- `match_square_well_1d.h` and `match_square_well_1d.cxx` for the class `MatchSquareWell1D` to calculate various wave functions of the double wells.
- `qm_pow4.h` and `qm_pow4.cxx` for the class `QMPow4` to calculate the ground state of the anharmonic potential.
- `lennard_jones_1d.h` and `lennard_jones_1d.cxx` for the class `LennardJones1D` to calculate the wave function in the Lennard Jones potential with deterministic variational method and match method.
- `hw6.cxx` for the main function to make plots.

To each problem of this lab report, I will attach the relevant parts of the code. If you want to check the validation of my code, you need to download the whole code from the link `https://github.com/YichengFeng/phys580/tree/master/hw6`.

(1) *[Problem 10.5 (p.322)]*

Using the shooting method to study how the wave function for a particle-in-a-box depends on the magnitude of the potential outside the box V_0. Examine the variation of ψ beyond the walls of the box and show that it decays exponentially with distance in this region. Study the decay length as a function of V_0 and compare the results for different energy levels. As the energy of the level approaches V_0 the decay length should become larger.

Physics explanation:

The square well is set to locate at $[-1, 1]$ with various wall potential $V_0 = 10, 50, 100$. For the boundary condition, we require the wavefunctions to be zero at $x = \pm 3$. We use the shooting method to find all the bound states of various V_0 with various initial energy E_i, where the final energy is close to. The error of energy is controlled under 10^{-10} with binary search.

If one state is bound, it should decay exponentially outside the well, so we fit the tail $(1 < x < 3)$ with a exponential function $p_0 e^{-p_1 x}$. The decay length λ can be calculated from p_1: $\lambda = 1/p_1$.

The results of $V_0 = 10$ is shown in Fig. 1, and there is only 1 bound state. The results of $V_0 = 50$ is shown in Fig. 2, and there are 7 bound states. The results of $V_0 = 100$ is shown in Fig. 3, and there are 9 bound states.

The wall potential (V_0), energy E, fitting parameter p_1, and decay length λ are listed in the following table.

wall potential (V_0)	E	p_1	λ
10	0.8197	4.286	0.2333
50	1.0189	9.894	0.1011
	4.0716	9.561	0.1046
	9.1211	9.041	0.1106
	16.141	8.220	0.1217
	24.985	7.073	0.1414
	35.505	5.479	0.1825
	46.840	2.523	0.3963
100	1.0759	14.03	0.07128
	4.3044	13.75	0.07273
	9.6636	13.43	0.07446
	17.164	12.88	0.07764
	26.723	12.10	0.08264
	38.378	11.05	0.09050
	51.927	9.805	0.10120
	67.386	8.071	0.12390
	84.180	5.633	0.17753

We can see that in one specific wall potential V_0, the decay length λ of bound states increases along the increasing energy. For one specific energy $E < V_0$, when the wall potential V_0 increases, the difference $V_0 - E$ increases. As a result, the decay length λ increases with V_0 for this specific energy.

Plots:

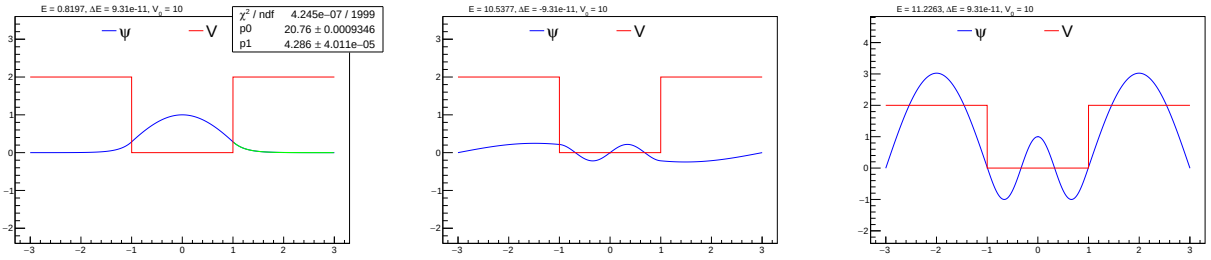

Figure 1: The wavefunction depends on various energy in the 1D square well with the wall potential $V_0 = 10$. (The first one is the only bound state.)

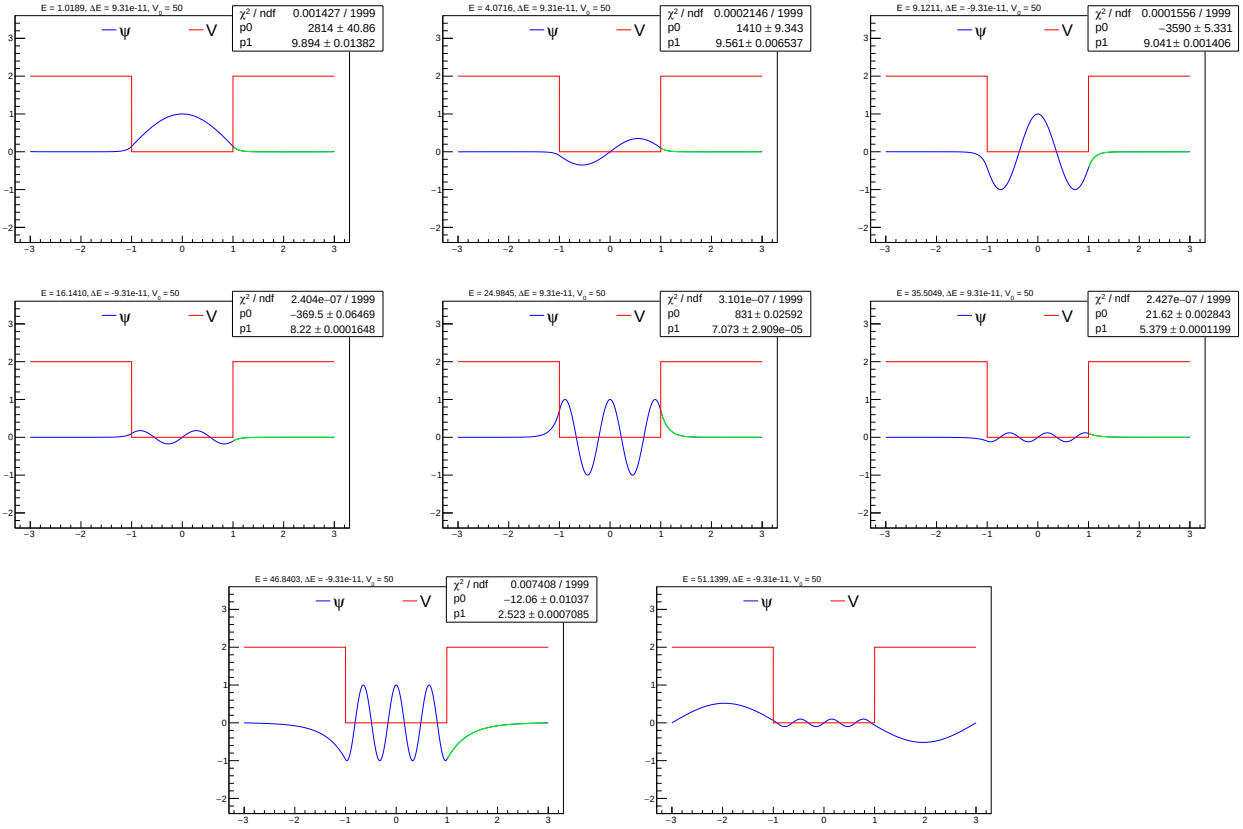

Figure 2: The wavefunction depends on various energy in the 1D square well with the wall potential $V_0 = 50$. (The last one is already unbound.)

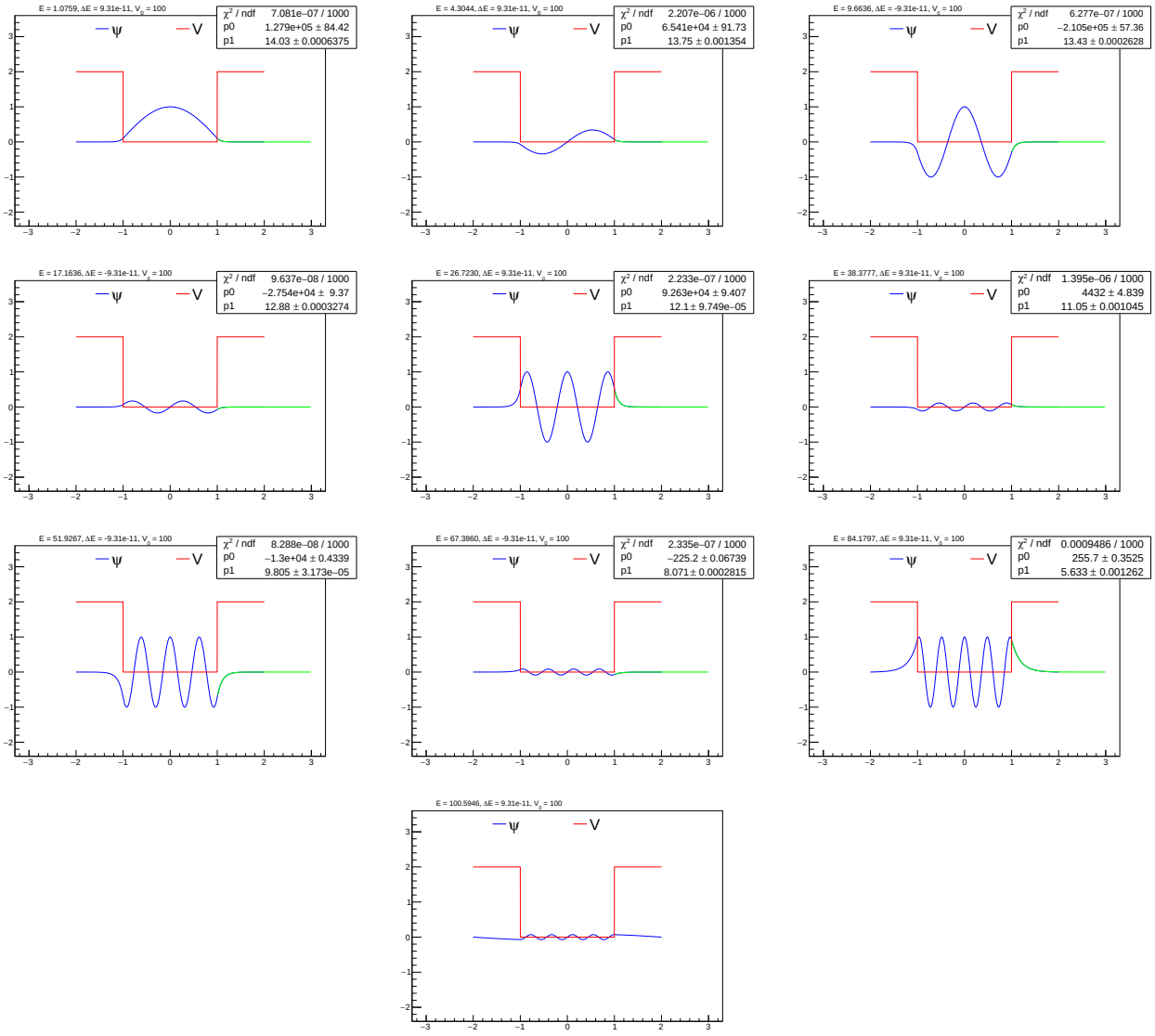

Figure 3: The wavefunction depends on various energy in the 1D square well with the wall potential $V_0 = 100$. (The last one is already unbound.)

Relevant code:

For the shooting method

```
//-------------------------------------------------------------------------//

void SquareWell1D::cal_once() {

        if(!check()) return;

        _xL.clear();
        _xR.clear();
        _psiL.clear();
        _psiR.clear();
        _x.clear();
        _psi.clear();

        if(_parity==1) {
                if(_a!=0) {
                        _psiR.push_back(0.1);
                        _psiR.push_back(0.1);
                        _psiL.push_back(0.1);
                        _psiL.push_back(0.1);
                } else {
                        _psiR.push_back(1);
                        _psiR.push_back(1);
                        _psiL.push_back(1);
                        _psiL.push_back(1);
                }
        } else {
                _psiR.push_back(-_dx);
                _psiR.push_back(0);
                _psiL.push_back(-_dx);
                _psiL.push_back(0);
        }
        _xR.push_back(-_dx);
        _xR.push_back(0);
        _xL.push_back(_dx);
        _xL.push_back(0);

        int i = 0;
        while(true) {
                i++;
                double x = (i-1)*_dx;
                double tmpR = 2*_psiR[i] - _psiR[i-1] - 2*(_E - potential(x))*_dx*_dx*_psiR[i];
                _xR.push_back(i*_dx);
                _xL.push_back(-i*_dx);
                _psiR.push_back(tmpR);
                _psiL.push_back(_parity*tmpR);

                //if(fabs(_psiR[i+1])>_b) break;
                if(fabs(_xR[i+1])>_boundary*_L) break;
        }

        int n = _psiL.size();
        for(int i=0; i<n; i++) {
                _x.push_back(_xL[n-1-i]);
                _psi.push_back(_psiL[n-1-i]);
        }
        for(int i=2; i<n; i++) {
                _x.push_back(_xR[i]);
```

```
58                  _psi.push_back(_psiR[i]);
59          }
60  }
61
62  //--------------------------------------------------------------------------//
63
64  void SquareWell1D::adjust_once() {
65
66          cal_once();
67          double psiAt1Now = _psiR[floor(_boundary*_L/_dx+1)];
68          if(psiAt1Now==0) {
69                  _dE = 0;
70          } else if(psiAt1Now*_psiAt1<0) {
71                  _dE = -0.5*_dE;
72          } else if(psiAt1Now>0 && psiAt1Now>_psiAt1) {
73                  _dE = -_dE;
74          } else if(psiAt1Now<0 && psiAt1Now<_psiAt1) {
75                  _dE = -_dE;
76          } else {
77                  _dE = _dE;
78          }
79
80          _E += _dE;
81          _psiAt1 = psiAt1Now;
82  }
83
84  //--------------------------------------------------------------------------//
85
86  void SquareWell1D::adjust_until(double ee) {
87
88          int n = 0;
89          while(fabs(_dE)>ee) {
90                  adjust_once();
91                  if(n == _iter) break;
92                  n ++;
93          }
94  }
95
96  //--------------------------------------------------------------------------//
```

(2) *[Problem 10.12 (p.333)]*
Employ the variational Monte Carlo method to calculate the ground-state energy and wave function of the anharmonic oscillator whose potential is given by $V(x) = x^4$.

Physics explanation:

We use the variational Monte Carlo method to calculate the ground-state energy and wave function of the anharmonic oscillator whose potential is given by $V(x) = x^4$.

The initial wave function is a uniform at the range $[-1, 1]$. We set the temperature $T = 0$, which means the simulation only goes in the direction with energy decreasing. After 10000 attempts, we get the result shown in Fig. 4. The wave function is in blue, and the energy is 0.7054 for the ground state.

Plots:

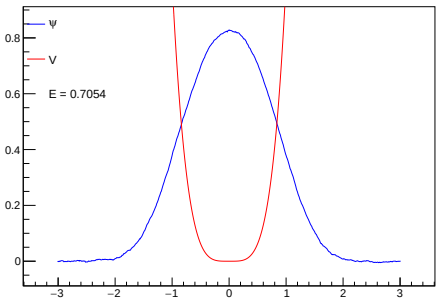

Figure 4: The wave function (blue) is calculated by the variational Monte Carlo method with the anharmonic potential $V(x) = x^4$ (red) after 10000 attempts.

Relevant code:

For the variational Monte Carlo method

```
1  //------------------------------------------------------------------------//
2
3  bool QMPow4::cal_var_once() {
4
5          int n = 1 + floor(1.0*rand()/RAND_MAX*(_iend-2));
6          int m = n + floor(1.0*rand()/RAND_MAX*(_iend-1-n));
7          double p = 1.0*rand()/RAND_MAX;
8          for(int i=n; i<=m; i++) {
9                  _psi_old[i] = _psi[i];
10                 _psi[i] += 2*(p-0.5)*_dpsi;
11         }
12         _newE = var_energy();
13         double deltaE = _newE - _varE;
14         bool keep;
15         if(_newE >= _varE) {
16                 keep = false;
17                 if(_T > 0) {
18                         p = 1.0*rand()/RAND_MAX;
19                         if(p <= exp(-deltaE/_T)) keep = true;
20                 }
21         } else {
22                 keep = true;
23         }
24
```

```
25          if(keep) {
26                  _varE = _newE;
27                  var_psi_normalize();
28          } else {
29                  for(int i=n; i<=m; i++) {
30                          _psi[i] = _psi_old[i];
31                  }
32          }
33
34          return keep;
35  }
36
37  //------------------------------------------------------------------------//
38
39  int QMPow4::cal_var_attempts(int n_attempts) {
40
41          int n_move = 0;
42          for(int i_attempts=0; i_attempts<n_attempts; i_attempts++) {
43                  if(cal_var_once()) n_move ++;
44          }
45          _n_total += n_attempts;
46          _n_moves += n_move;
47
48          return n_move;
49  }
50
51  //------------------------------------------------------------------------//
```

(3) Write a matching method program to study the coupling between two one-dimensional quantum mechanical systems in neighboring square wells that are separated by a small, square barrier (cf. Figs. 10.11 and 10.12 of the textbook). In particular, observe how identical unperturbed states in each well get mixed due to being coupled through the finite barrier. Demonstrate numerically, for at at least two different examples (such as the two ground states and then two excited states), that the initially equal energy levels split up. Namely, the parity even mixture moves down in energy, while the parity odd one moves up. This phenomenon is discussed in Chapter 10 of the book (p.318-320).

Physics explanation:

For the original infinitely deep square well with $L = 0.5$, the energy of the states should be

$$E_n = \frac{\hbar^2 n^2 \pi^2}{8mL^2} = \frac{n^2\pi^2}{2}. \tag{1}$$

To see how the those wave functions of the single well would be in the double wells, we just input the corresponding energy E_n and match the left and right wave functions (ψ_L, ψ_R) in values but not in the derivatives.

To see the actual wave functions of the double wells, we then match ψ_L, ψ_R in values and derivatives after enough iterations around the energy E_n. For each E_n, there should be two states with even and odd parity, respectively.

Figure 5 shows the original ground states (left pad) of the single well, and the actual two states of the double wells around the ground energy (E_1) (even: middle pad; odd: right pad). We get $E_1 = \pi^2/2 \approx 5.0348$ and $E_{1e} = 5.1076$, $E_{1o} = 5.2967$. Both states of the double wells are higher than the energy of the single well state, and the odd state has energy higher than the even.

Figure 6 shows the original states (left pad) of the single well, and the actual two states of the double wells around the energy (E_2) (even: middle pad; odd: right pad). We get $E_2 = 2\pi^2 \approx 19.8392$ and $E_{1e} = 20.2928$, $E_{1o} = 21.1554$. Both states of the double wells are higher than the energy of the single well state, and the odd state has energy higher than the even.

Plots:

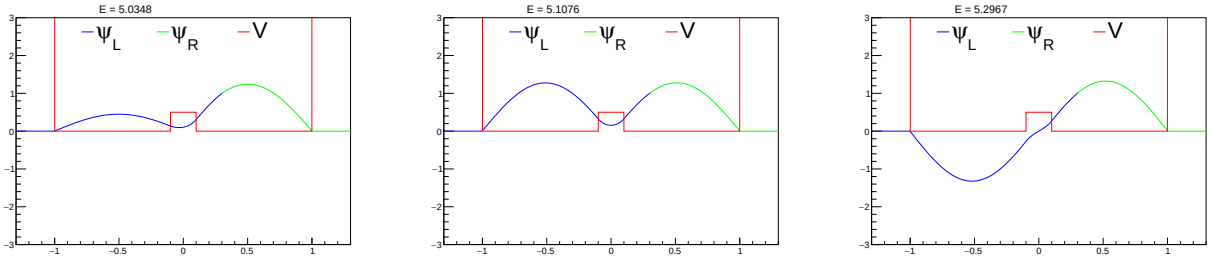

Figure 5: The energy of two ground states (left pad) of the infinitely deep square well splits into two cases (even: middle pad; odd: right pad) in the neighboring square wells.

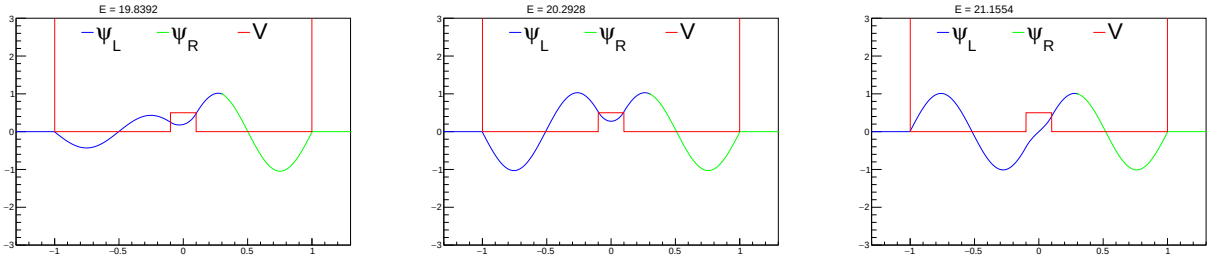

Figure 6: The energy of two excited states (left pad) of the infinitely deep square well splits into two cases (even: middle pad; odd: right pad) in the neighboring square wells.

Relevant code:

For the match method

```
1  //------------------------------------------------------------------------//
2
3  void MatchSquareWell1D::cal_match_once() {
4
5          if(!check()) return;
6
7          _xL.clear();
8          _xR.clear();
9          _psiL.clear();
10         _psiR.clear();
11         _psi.clear();
12
13         _psiL.push_back(0);
14         _psiL.push_back(1e-2*_dx);
15         _psiR.push_back(0);
16         _psiR.push_back(1e-2*_dx);
17
18         _xL.push_back(_x_left-_dx);
19         _xL.push_back(_x_left);
20         _xR.push_back(_x_right+_dx);
21         _xR.push_back(_x_right);
22
23         double psimatch;
24
25         for(int i=1; i<_matchL+16; i++) {
```

```
                double x = _x_left + (i-1)*_dx;
                double tmpL = 2*_psiL[i] - _psiL[i-1] - 2*(_E - potential(x))*_dx*_dx*_psiL[i];
                _psiL.push_back(tmpL);
                _xL.push_back(x+_dx);
        }
        psimatch = _psiL[_matchL+1];
        for(int i=0; i<_psiL.size(); i++) {
                _psiL[i] /= psimatch;
        }

        for(int i=1; i<_matchR+16; i++) {
                double x = _x_right - (i-1)*_dx;
                double tmpR = 2*_psiR[i] - _psiR[i-1] - 2*(_E - potential(x))*_dx*_dx*_psiR[i];
                _psiR.push_back(tmpR);
                _xR.push_back(x-_dx);
        }
        psimatch = _psiR[_matchR+1];
        for(int i=0; i<_psiR.size(); i++) {
                _psiR[i] /= psimatch;
        }
}

//------------------------------------------------------------------------//

void MatchSquareWell1D::adjust_match_once() {

        cal_match_once();

        double devL = (_psiL[_matchL+2] - _psiL[_matchL])/(2*_dx);
        double devR = (_psiR[_matchR] - _psiR[_matchR+2])/(2*_dx);
        double tmpdiff = devL - devR;

        if(tmpdiff==0) {
                _dE = 0;
        } else if(tmpdiff*_devdiff<0) {
                _dE = -0.5*_dE;
        } else if(tmpdiff>0 && tmpdiff>_devdiff) {
                _dE = -_dE;
        } else if(tmpdiff<0 && tmpdiff<_devdiff) {
                _dE = -_dE;
        } else {
                _dE = _dE;
        }

        _E += _dE;
        _devdiff = tmpdiff;
}

//------------------------------------------------------------------------//

void MatchSquareWell1D::adjust_match_until(double ee) {

        int n = 0;
        while(fabs(_dE)>ee) {
                if(n==_Nmax) break;
                adjust_match_once();
                n ++;
        }
}
```

(4) Obtain a deterministic variational estimate for the ground state energy and wave function of a particle in a 1D Lennard-Jones potential with $\epsilon = 10, \sigma = 1$ (remember, $m = 1, \hbar = 1$ units are used). Compare the obtained energy value and the wave function to what you have found in Lab 13, part 2. The complexity of the calculation, of course, very much depends on how good your variational basis is, so my suggestion is that you use

$$\psi_n(x) = \begin{cases} \text{const} \times \left(x - \frac{3}{4}\right)^n e^{-\alpha x}, & \text{if } x > \frac{3}{4} \\ 0, & \text{otherwise} \end{cases} \qquad (a > 0) \tag{2}$$

Set $n = 3$, i.e., keep only one basis function ψ_3, and optimize a such that the estimated ground state energy is minimized.

[Optional: why did I suggest keeping ψ_3, and not, ψ_2 or ψ_4, for example?]

Physics explanation:

We use the Monte Carlo simulation for the value of α. We add a random number $0 < \Delta\alpha < 1$ to α. If the energy gets lower, we keep the new value of α. Otherwise, we go back to the previous α value.

After 4000 attempts, we get the results shown (black curve) in Fig. 7. The deterministic variational method gives $\alpha = 4.9395$ and energy $E = -1.8256$.

We also repeat the previous match method here to get the more presice results (blue and green curves) and the energy is $E = -1.8905$.

Plots:

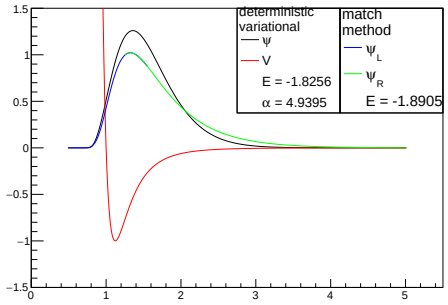

Figure 7: The wave functions are calculated from the deterministic variational method (black) and the match method (blue and green). The latter is the presice one.

Relevant code:

For the deterministic variational method

```
1 //------------------------------------------------------------------------//
2
3 bool LennardJones1D::cal_var_once() {
4
5         int n = 0;
6         int m = _iend;
7         double p = 1.0*rand()/RAND_MAX;
8         double oldA = _varA;
9         if(_varA!=0) {
```

```
                _varA += (p-0.5)*fabs(_varA);
        } else {
                _varA = p-0.5;
        }
        for(int i=n; i<=m; i++) {
                _psi_old[i] = _psi[i];
                if(_x[i]>3.0/4.0) {
                        _psi[i] = pow(_x[i]-3.0/4.0, _varN)*exp(-_varA*_x[i]);
                } else {
                        _psi[i] = 0;
                }
        }
        _newE = var_energy();
        double deltaE = _newE - _varE;
        bool keep;
        if(_newE >= _varE) {
                keep = false;
                if(_T > 0) {
                        p = 1.0*rand()/RAND_MAX;
                        if(p <= exp(-deltaE/_T)) keep = true;
                }
        } else {
                keep = true;
        }

        if(keep) {
                _varE = _newE;
                var_psi_normalize();
        } else {
                for(int i=n; i<=m; i++) {
                        _psi[i] = _psi_old[i];
                }
                _varA = oldA;
                var_psi_normalize();
        }

        return keep;
}

//------------------------------------------------------------------------//

int LennardJones1D::cal_var_attempts(int n_attempts) {

        int n_move = 0;
        for(int i_attempts=0; i_attempts<n_attempts; i_attempts++) {
                if(cal_var_once()) n_move ++;
        }
        _n_total += n_attempts;
        _n_moves += n_move;

        return n_move;
}

//------------------------------------------------------------------------//
```

For the match method

```
//------------------------------------------------------------------------//

void LennardJones1D::cal_match_once() {

```

```
5         if(!check()) return;
6
7         _xL.clear();
8         _xR.clear();
9         _psiL.clear();
10        _psiR.clear();
11        _psi.clear();
12
13        _psiL.push_back(0);
14        _psiL.push_back(1e-2*_dx);
15        _psiR.push_back(0);
16        _psiR.push_back(1e-2*_dx);
17
18        _xL.push_back(_x_left-_dx);
19        _xL.push_back(_x_left);
20        _xR.push_back(_x_right+_dx);
21        _xR.push_back(_x_right);
22
23        double psimatch;
24
25        for(int i=1; i<_matchL+16; i++) {
26                double x = _x_left + (i-1)*_dx;
27                double tmpL = 2*_psiL[i] - _psiL[i-1] - 2*(_E - potential(x))*_dx*_dx*_psiL[i];
28                _psiL.push_back(tmpL);
29                _xL.push_back(x+_dx);
30        }
31        psimatch = _psiL[_matchL+1];
32        for(int i=0; i<_psiL.size(); i++) {
33                _psiL[i] /= psimatch;
34        }
35
36        for(int i=1; i<_matchR+16; i++) {
37                double x = _x_right - (i-1)*_dx;
38                double tmpR = 2*_psiR[i] - _psiR[i-1] - 2*(_E - potential(x))*_dx*_dx*_psiR[i];
39                _psiR.push_back(tmpR);
40                _xR.push_back(x-_dx);
41        }
42        psimatch = _psiR[_matchR+1];
43        for(int i=0; i<_psiR.size(); i++) {
44                _psiR[i] /= psimatch;
45        }
46 }
47
48 //------------------------------------------------------------------------//
49
50 void LennardJones1D::adjust_match_once() {
51
52        cal_match_once();
53
54        double devL = (_psiL[_matchL+2] - _psiL[_matchL])/(2*_dx);
55        double devR = (_psiR[_matchR] - _psiR[_matchR+2])/(2*_dx);
56        double tmpdiff = devL - devR;
57
58        if(tmpdiff==0) {
59                _dE = 0;
60        } else if(tmpdiff*_devdiff<0) {
61                _dE = -0.5*_dE;
62        } else if(tmpdiff>0 && tmpdiff>_devdiff) {
63                _dE = -_dE;
64        } else if(tmpdiff<0 && tmpdiff<_devdiff) {
```

```
65                  _dE = -_dE;
66          } else {
67                  _dE = _dE;
68          }
69
70          _E += _dE;
71          _devdiff = tmpdiff;
72  }
73
74  //-------------------------------------------------------------------------//
75
76  void LennardJones1D::adjust_match_until(double ee) {
77
78          while(fabs(_dE)>ee) {
79                  adjust_match_once();
80          }
81  }
82
83  //-------------------------------------------------------------------------//
```