2011-04-25 作業解答

1. 寫程式以 RK4 數值方法積分一維 Schrödinger 方程式。假定位能井 $V(x) = -44\,\theta(1-|x|)$，用射擊法在 $x \in [-4, 4]$的區間中找尋束縛態的本徵能量及本徵函數（約有五組，函數畫在一張圖即可）。

［隨意題］與理論值比較。

Outside the square well potential, the bound state ($E < 0$) solution of Schrödinger equation should be of exponential growth on the left and of exponential decay on the right. Theoretically, the exponential growth determines the ratio between the value and slope condition at the $x = -4$ boundary. However, any inaccurate choice will only incur an exponentially decaying error that can be ignored for our purpose. Similarly, the accuracy for the “target” at $x = 4$ boundary that we should adopt in the shooting method is not crucial to the determination of bound state energies. We can thus safely set the target to 0 without significant impact to the resulting energy levels.
C++:

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <iomanip>
#include <vector>
#include <cmath>
using namespace std;
typedef void (*SlopeFunc)(double t, double * y, double * f);
void runge_kutta(double t, double * y, double h, SlopeFunc sf, int ydim)
{
      double k0[ydim], k1[ydim], k2[ydim], k3[ydim], yy[ydim];
            (*sf)(t, y, k0);
      for (int i = 0; i < ydim; i ++) yy[i] = y[i] + k0[i] * h / 2;
      (*sf)(t + h / 2, yy, k1);
      for (int i = 0; i < ydim; i ++) yy[i] = y[i] + k1[i] * h / 2;
      (*sf)(t + h / 2, yy, k2);
      for (int i = 0; i < ydim; i ++) yy[i] = y[i] + k2[i] * h;
      (*sf)(t + h, yy, k3);
      for (int i = 0; i < ydim; i ++) {
            y[i] += (k0[i] + 2 * (k1[i] + k2[i]) + k3[i]) * h / 6;
      }
}
```

The above is a generic implementation of Runge-Kutta. It integrates any given slope function sf on any number ydim of variables y over a small interval t.

```
double potential(double x)
{
      if (x > - 1 && x < 1) return - 44;
      return 0;
}
double xmin = - 4;
double xmax = 4;
int npts = 2000; // number of points
double energy;
```

These parameters determine the range and resolution of the integral at given energy in the Schrödinger equation, which is turned into two-variable first-order differential equations below. The signature of the function is identical to the SlopeFunc type accepted by runge_kutta so it can be passed as the integrand.

```
void schrodinger(double x, double * y, double * f)
{ // y[0]: \varphi, y[1]: \varphi'
```

```
      f[0] = y[1];
      f[1] = y[0] * (potential(x) - energy);
}
int nnode; // number of nodes
vector<double> trace; // trace of wave function
```

During the integration, we keep track the number of nodes (zero crossings) of the wave function encountered as well as the entire trajectory of the wave function.

```
double integrate(double e)
{
      energy = e;
      double y[2] = {0.01, 0};
      nnode = 0;
      double y0 = y[0];
      double dx = (xmax - xmin) / npts;
      trace.clear();
      trace.push_back(y0);
      for (int i = 0; i < npts; i ++) {
            double x = xmin + i * dx;
            runge_kutta(x, y, dx, &schrodinger, 2);
            if (y0 < 0 && y[0] >= 0 || y0 > 0 && y[0] <= 0) nnode ++;
            y0 = y[0];
            trace.push_back(y0);
      }
      return y0;
}
```

The eigen functions can be indexed by the number of nodes they have. Below, we normalize and save the entire trajectory of a function to a data file wave_n.dat named with the index n.

```
void save_trace(int n)
{
      ostringstream fns;
            fns << "wave_" << n << ".dat";
      ofstream f(fns.str().c_str());
      double dx = (xmax - xmin) / npts;
      double norm = 0;
      for (size_t i = 0; i < trace.size(); i ++) {
            norm += trace[i] * trace[i];
      }
      norm = sqrt(norm * dx);
      f << setprecision(15);
      for (size_t i = 0; i < trace.size(); i ++) {
            f << xmin + i * dx << '\t' << trace[i] / norm << '\n';
      }
}
vector<double> eigenE;
```

Here, I am going do the shooting method a little differently. Instead of looking for energies where the value of the wave function hits zero at the $x = 4$ boundary, I am pinpointing the locations in energy where the number of nodes in the wave function increases. This binary search is done recursively in find_eigen below and the results are stored in the dynamic array eigenE defined above. The recursion is down to when the size of the search interval can no longer be decreased. Because of inherent inaccuracy in floating point arithmetics, we can not count on the size of this interval to shrink to zero.

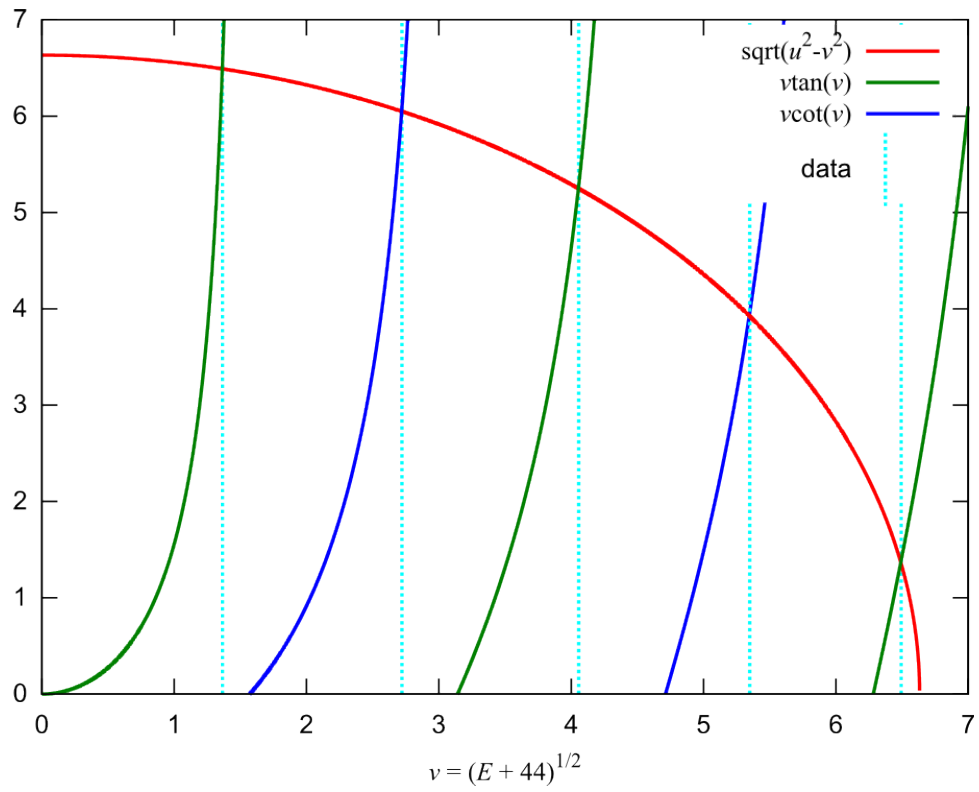
```
void find_eigen(double a, double b, int na, int nb)
{ // assume a < b, na & nb are number of nodes at a & b
      double c = (a + b) / 2;
      if (c <= a || c >= b) { // |b-a| small enough
            eigenE.push_back(c);
            save_trace(na);
            return;
      }
      integrate(c);
      int nc = nnode;
      if (na < nc) find_eigen(a, c, na, nc);
      if (nc < nb) find_eigen(c, b, nc, nb);
}
int main()
{
      integrate(0); // upper bound
      int nb = nnode;
      double e_low = - 10;
      do { // find low bound of energy
            e_low *= 2;
            integrate(e_low);
      } while (nnode > 0);
      find_eigen(e_low, 0, 0, nb);
      cout << setprecision(15);
      for (size_t i = 0; i < eigenE.size(); i ++) cout << eigenE[i] << '\n';
      return 0;
}
```

The eigen energies obtained by the above code are in the second column of the following table.

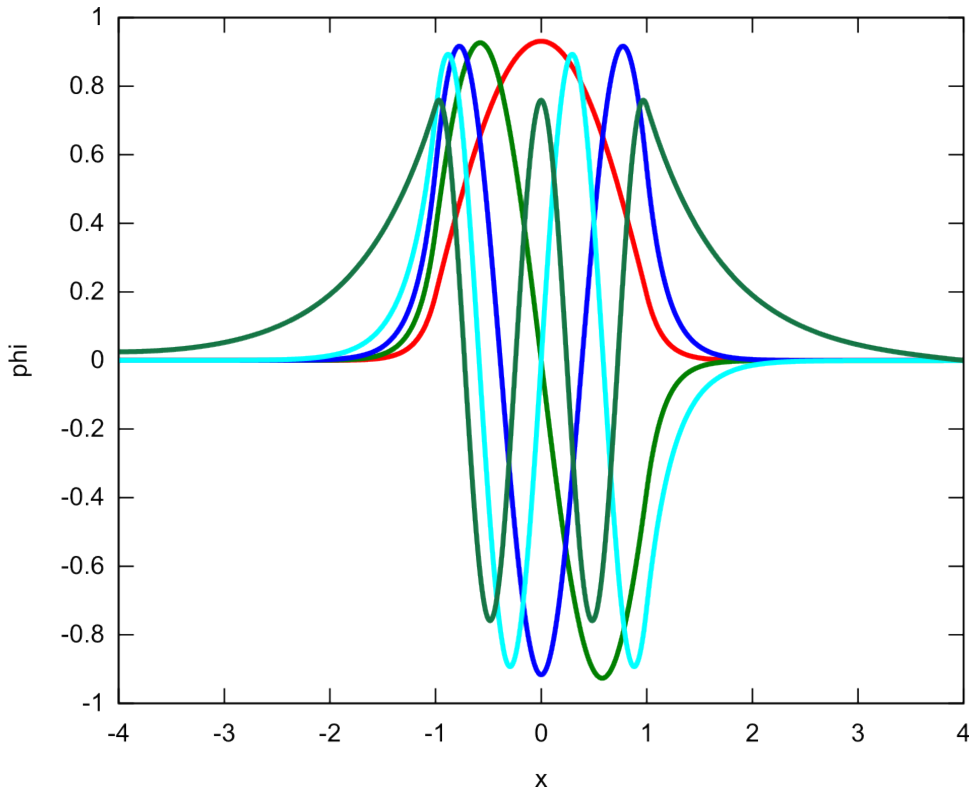
	calculated results	exact solution
E_0	-42.1380998222273	-42.1402410887795
E_1	-36.597470875891	-36.6059044639124
E_2	-27.5408672757842	-27.55923142531
E_3	-15.39061847916	-15.4209316356095
E_4	-1.83712409012774	-1.86949880536433

Comparing to the “exact” solution, we see the errors of our calculations are within 2% of the exact values. These errors can be from our approximate choices of boundary conditions, the finite range of the integration, and the error of numerical integration.

The theoretical values of the eigen energies in a square well are given by the intersects of the function sqrt(u^2-v^2) with $v\tan(v)$ for the symmetric states and with $v\cot(v)$ for the antisymmetric states, where u^2 is the depth of the well and v^2 is the particle energy relative to the bottom of the well. While they can be calculated to arbitrary precision, there are no close form analytical expression for these values. The values listed in the above table are obtained with a simple binary-search root finder. For our particular case, the parameters are $u^2 = 44$, $v^2 = E + 44$. We plot these relevant curves below and mark the calculated data with vertical dash lines.



As expected from the table of energy values, the vertical lines match well with the intersects. The normalized wave functions for the corresponding eigen states are plotted below. As mentioned the numbers of zero-crossings give the indexes of these eigen states.



2. 實作 Hoshen-Kopelman 演算法：寫程式輸入二維點位滲流(site percolation)的點格大小(lattice size)及點位佔有機率 p，用 1000 個 seed 的亂數序列來求平均最大簇集(cluster)大小。［如無法用 H-K 法，可用多次掃描來標定 site 所屬的 cluster。］

The Hoshen-Kopelman algorithm makes use of a two-step label system to avert the need of updating the labels for all cluster sites upon the merger of clusters. However, we do need to traverse the referencing links to find the actual cluster label to which a given site belongs. The number of steps in such a traversal should be much limited. An implementation of the Hoshen-Kopelman algorithm is listed below.
C++:
```
#include "ran_nr.hh"
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;
size_t szx, szy;
size_t sz2;
bool * sites;
int * lab;  // labels for sites
vector<int> nlab; // labels for labels
int cluster_lab(size_t i) // find cluster label
{
      int l = lab[i];
      while (nlab[l] <= 0) l = - nlab[l];
      return l;
}
```
The above function is for the traversal of two-step label system to find the actual cluster of a given site i.
```
int main()
{
      double p; // percolation prob.
      unsigned long seed;
      cout << "# input szx szy p seed:\n";
      cin >> szx >> szy >> p >> seed;
      cout << "# size " << szx << 'x' << szy << ", p="
           << p << ", seed " << seed << '\n';
      RanNR rng(seed);
      sz2 = szx * szy;
      // allocate memory
      sites = new bool [sz2]; lab = new int [sz2];
      for (size_t i = 0; i < sz2; i ++) { // generate the system
            sites[i] = rng.uniform() < p;
      }
      for (size_t i = 0; i < sz2; i ++) if (sites[i]) {
            if ((i % szx) && sites[i - 1]) {
```
The (i % szx) condition above makes sure the x coordinate is none zero and i - 1 gives the left neighbor of site i.
```
                  int l = cluster_lab(i - 1);
                  lab[i] = l; nlab[l] ++;
                  if (i < szx || ! sites[i - szx]) continue;
```
The condition i < szx means it is on the first row and i - szx gives the top neighbor of site i.
```
                  // merge
                  int ll = cluster_lab(i - szx);
                  if (ll < l) {
                        nlab[ll] += nlab[l]; nlab[l] = - ll;
                  }
```

```
                        else if (ll > l) {
                              nlab[l] += nlab[ll]; nlab[ll] = - l;
                        }
                        continue;
                  }
                  if (i >= szx && sites[i - szx]) {
                        int l = cluster_lab(i - szx);
                        lab[i] = l; nlab[l] ++;
                        continue;
                  }
                  // new cluster
                  lab[i] = nlab.size(); nlab.push_back(1);
            }
```

It is not required. But, we do a relabeling below so we have a continuous labeling rlab starting from 1. We also find out the size of the largest cluster bb at the same time.

```
            int bb = 0, cc = 0;
            vector<int> rlab(nlab.size());
            // count and relabel
            for (int i = 0; i < nlab.size(); i ++) if (nlab[i] > 0) {
                  if (bb < nlab[i]) bb = nlab[i];
                  rlab[i] = ++ cc;
            }
            for (size_t y = 0; y < szy; y ++) {
                  for (size_t x = 0; x < szx; x ++) {
                        size_t i = y * szx + x;
                        if (sites[i]) {
                              int l = cluster_lab(i);
                              cout << setw(4) << rlab[l];
                        }
                        else cout << "    ";
                  }
                  cout << '\n';
            }
            cout << "# found " << cc << " cluters.\n";
            cout << "# largest cluster has " << bb << " sites.\n";
            return 0;
}
```

The Hoshen-Kopelman method described above is an efficient one-pass algorithm. But, the two-step label system could be a bit tricky to implement. Following code does the same thing using an inefficient algorithm that tries to synchroize the labels between neighboring sites iteratively. It scans through the system many times until all neighboring sites have the same labels.

C++:

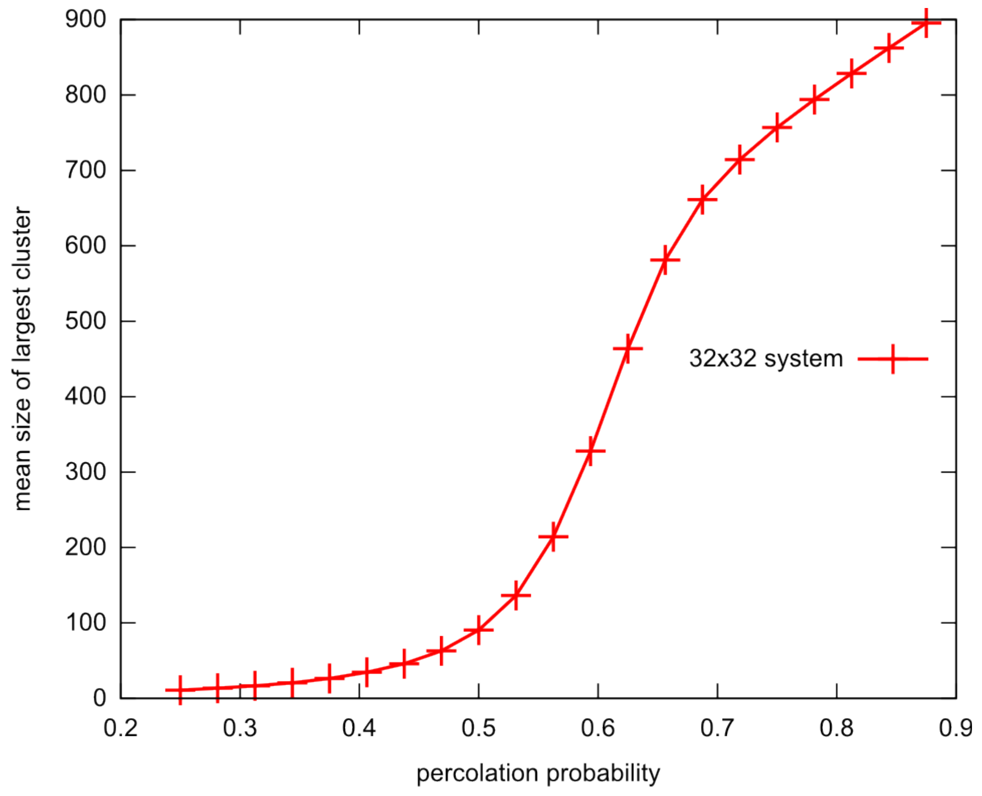
```
#include "ran_nr.hh"
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;
size_t szx, szy;
size_t sz2;
bool * sites;
int * lab;  // labels for sites
int main()
{
      double p; // percolation prob.
      unsigned long seed;
      cout << "# input szx szy p seed:\n";
      cin >> szx >> szy >> p >> seed;
      cout << "# size " << szx << 'x' << szy << ", p=" << p << ", seed " << seed
<< '\n';
      RanNR rng(seed);
      sz2 = szx * szy;
      // allocate memory
      sites = new bool [sz2]; lab = new int [sz2];
      for (size_t i = 0; i < sz2; i ++) { // generate the system
            sites[i] = rng.uniform() < p;
      }
      int lcnt = 0;
      for (size_t i = 0; i < sz2; i ++) if (sites[i]) lab[i] = lcnt ++;
      int change;
      do { // iteratively synchronize neighboring sites
            change = 0;
            for (size_t i = 0; i < sz2; i ++) if (sites[i]) {
                  if ((i % szx) && sites[i - 1] && lab[i] != lab[i - 1]) {
                        if (lab[i] < lab[i - 1]) lab[i - 1] = lab[i];
                        else lab[i] = lab[i - 1];
                        change ++;
                  }
                  if (i >= szx && sites[i - szx] && lab[i] != lab[i - szx]) {
                        if (lab[i] < lab[i - szx]) lab[i - szx] = lab[i];
                        else lab[i] = lab[i - szx];
                        change ++;
                  }
            }
      } while (change);
      vector<int> ccnt;
      // relabel and count the size of the clusters
      for (size_t i = 0; i < sz2; i ++) if (sites[i]) {
            int l = lab[i];
            if (l >= ccnt.size()) {
                  ccnt.resize(l + 1, 0); clab.resize(l + 1, 0);
            }
            if (ccnt[l] == 0) clab[l] = ++ lcnt;
            ccnt[lab[i]] ++;
      }
      // find size of the biggest
      int bb = 0;
      for (int i = 0; i < ccnt.size(); i ++) if (bb < ccnt[i]) bb = ccnt[i];
      for (size_t i = 0; i < sz2; i ++) {
            if (sites[i]) cout << setw(4) << clab[lab[i]];
            else cout << "    ";
            if (i % szx == szx - 1) cout << '\n';
      }
      cout << "# found " << lcnt << " cluters.\n";
      cout << "# largest cluster has " << bb << " sites.\n";
      return 0;
}
```

With the Hoshen-Kopelman program, say “hk”, in place, we can use a BASH script to do the ensemble average over different random seeds. This saves us from doing further programming in C++ when it is not desirable or even possible. (Think about the situation where we do not have the source code to a binary program.)
BASH:

```
#!/bin/bash
p=0.25
while [ `bc -l<<<"${p}<0.9"` == 1 ];do
      s=0
      tot=0
      while ((s<1000));do
```

```
                csz=`./hk<<<"32 ${p} ${s}"|grep largest|awk '{print $5}'`
                tot=$((tot+csz))
                s=$((s+1))
        done
        echo -e "${p}\t"`bc -l<<<"${tot}/1000"`
        p=`bc -l<<<"${p}+0.03125"`
done
```

Running the script and redirecting the output to a data file, we can plot the resultant mean size of largest cluster in a 32x32 system as a function of the percolation probability.


mean size of largest cluster
percolation probability
32x32 system