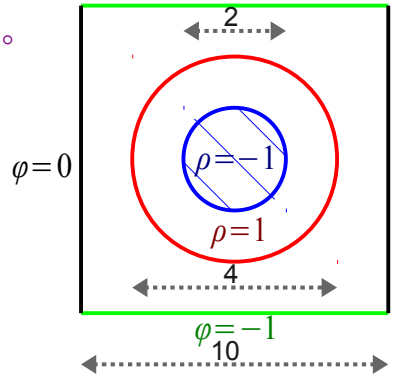
2011-05-09 作業解答

1. 用鬆弛法解 Poisson 方程式，邊界及電荷分佈如右。制作三維的電位圖。



The relaxation method outlined in the lecture can be implemented with the code fragments provided. However, here we are doing it a bit differently.

C++:

```
#include <iostream>
#include <cmath>
class PoissonRelax
{
	double xmin, xmax, ymin, ymax;
	size_t szx, szy;
	double * phi, * rho;
	// auxilary
	size_t sz;
	double ax, ay, * nph;
public:
```

The computation is cast into a general class for doing Poisson relaxation in a rectangle area. We also allow the mesh size to be different in the x and y directions.

```
	PoissonRelax(double xn, double xm, double yn, double ym,
			size_t sx, size_t sy) :
		xmin(xn), xmax(xm), ymin(yn), ymax(ym), szx(sx), szy(sy)
	{
		sz = (szx + 1) * (szy + 1);
		ax = (xmax - xmin) / szx;
		ay = (ymax - ymin) / szy;
		phi = new double [sz];
		rho = new double [sz];
		nph = new double [sz];
	}
	~PoissonRelax()
	{
		delete [] phi;
		delete [] rho;
		delete [] nph;
	}
```

A charge distribution is passed to the class as a pointer to the 2D function. Below, we only consider constant boundary values that can be specified independently for the four boundaries.

```
	void set_charge(double (*func)(double, double))
	{
		for (size_t i = 0; i < sz; i ++) {
			phi[i] = 0;
			nph[i] = 0;
			double x = xmin + ax * (i / (szy + 1));
			double y = ymin + ay * (i % (szy + 1));
			rho[i] = (* func)(x, y);
		}
	}
	void set_boundary(double top, double bot, double lef, double rit)
	{
		for (size_t ix = 1; ix < szx; ix ++) {
			size_t i = ix * (szy + 1);
			nph[i] = phi[i] = top;
```

```
                i += szy;
                nph[i] = phi[i] = bot;
            }
            for (size_t iy = 1; iy < szy; iy ++) {
                size_t i = iy;
                nph[i] = phi[i] = lef;
                i += szx * (szy + 1);
                nph[i] = phi[i] = rit;
            }
        }
        void dump(std::ostream & out)
        {
            for (size_t i = 0; i < sz; i ++) {
                int ix = i / (szy + 1);
                int iy = i % (szy + 1);
                double x = xmin + ax * ix;
                double y = ymin + ay * iy;
                out << x << '\t' << y << '\t' << phi[i] << '\n';
                if (iy == szy && ix != szx) out << '\n';
            }
        }
        double relax()
        {
            double err = 0;
            double ax2 = ax * ax, ay2 = ay * ay;
            double aa = ax2 * ay2, a2a = 2 * (ax2 + ay2);
            double bx = ax2 / a2a, by = ay2 / a2a, br = aa / a2a;
```

The complicated calculation here is to allow anisotropic lattice points. It reduces to the case presented in lecture when we have ax == ay.

```
            for (size_t ix = 1; ix < szx; ix ++)
            for (size_t iy = 1; iy < szy; iy ++) {
                size_t i = ix * (szy + 1) + iy;
                nph[i] = bx * (phi[i - 1] + phi[i + 1])
                       + by * (phi[i - szy - 1]
                       + phi[i + szy + 1]) + br * rho[i];
                double chg = fabs(nph[i] - phi[i]);
                if (chg > err) err = chg;
            }
            double * t = phi;
            phi = nph;
            nph = t;
            return err;
        }
};
double potential(double x, double y)
{
        double r = sqrt(x * x + y * y);
        if (r < 2) return - 1;
        if (r < 4) return 1;
        return 0;
}
int main()
{
        PoissonRelax pr(- 5, 5, - 5, 5, 63, 63);
        pr.set_charge(& potential);
        pr.set_boundary(- 1, - 1, 0, 0);
        double error = 2e-10;
        double lerr;
        size_t icnt = 0;
        do {
            lerr = error;
```

```
            error = pr.relax();
            icnt ++;
        } while (error > 1e-12 || error < lerr);
```

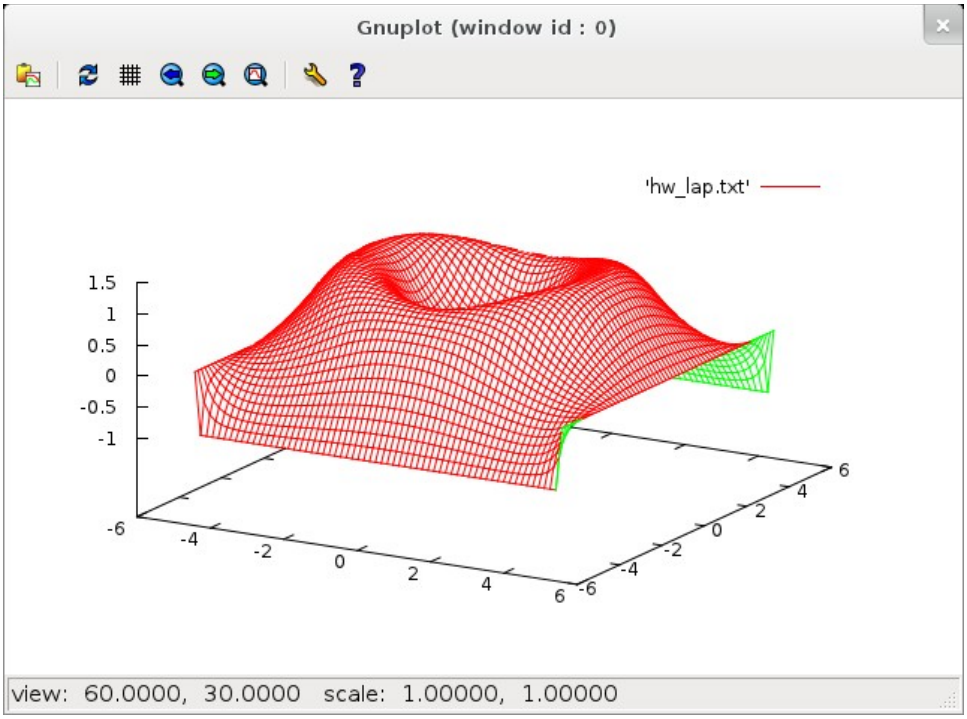
We actually iterate until the error can not be reduced further.

```
        pr.dump(std::cout);
        std::cerr << "icnt = " << icnt << '\n';
        return 0;
    }
```

We send the output to a file and plot it with gnuplot:
Terminal:

```
cjj@solid:~$ g++ hw_lap.cc
cjj@solid:~$ g++ hw_lap.cc -o hw_lap
cjj@solid:~$ ./hw_lap > hw_lap.txt
icnt = 20205
cjj@solid:~$ gnuplot

        G N U P L O T
        Version 4.4 patchlevel 0
        last modified March 2010
        System: Linux 2.6.39-1-amd64

        Copyright (C) 1986-1993, 1998, 2004, 2007-2010
        Thomas Williams, Colin Kelley and many others

        gnuplot home:     http://www.gnuplot.info
        faq, bugs, etc:   type "help seeking-assistance"
        immediate help:   type "help"
        plot window:      hit 'h'

Terminal type set to 'wxt'
gnuplot> set hidden
gnuplot> set st da lines
gnuplot> splot 'hw_lap.txt'
gnuplot>
```

The resulting plot is as follows:



2. 用游蛇法計算自避漫步在步數 $n = 100, 200, 400, 800, 1600$ 時的平均頭尾距離。

As suggested in the lecture, we can keep both a 2D array (maybe linearized) for sites occupied by the snake as well as a 1D array for the position of each segment of the snake. While these information are redundant, it greatly speed up the time in checking the overlap condition. To avoid the snake slithering out of bound, we adopt period boundary for the 2D lattice. However, this complicates the calculation of the head-to-tail displacement vector. Instead of taking the direct difference between the head and tail positions, we need to sum up the segmental displacement along the entire snake. But, the alternative approach adopted here is to keep an up-to-date displacement vector during the slithering of the snake to avoid the integration. Depending on how often the displacement vector is measured, this might not be the better approach. The implementation (also written into a class) is as follows:

C++:

```
	#include "ran_nr.hh"
	#include <iostream>
	class Snake
	{
		RanNR rng;
		size_t * sites;
		bool * arena;
		size_t hd;
		size_t len;
		int dispx;
		int dispy;
		// utilities
		int len_p;
		int len_o;
		int delta_x(size_t i, size_t j)
		{return len_o + (j % len + len_p - i % len) % len;}
		int delta_y(size_t i, size_t j)
		{return len_o + (j / len + len_p - i / len) % len;}
```

The above functions are used to find out the displacement vector between near by sites.

```
	public:
		Snake(size_t l) :
			rng(123), len(l)
		{
			sites = new size_t [len];
			arena = new bool [len * len];
			hd = 0;
			// lay down the snake in a line
			for (size_t i = 0; i < len; i ++) {
				sites[i] = i;
				arena[i] = true;
			}
			for (size_t i = len; i < len * len; i ++) arena[i] = false;
			dispx = len - 1;
			dispy = 0;
			len_p = len + len / 2;
			len_o = - len / 2;
		}
		~Snake()
		{
			delete [] sites;
			delete [] arena;
		}
		enum Dir {DIR_LEFT, DIR_RIGHT, DIR_UP, DIR_DOWN, DIR_NUM};
		size_t neighbor(size_t base, Dir dir)
		{
```

```
                switch (dir) {
                case DIR_LEFT:
                        return base % len ? base - 1 : base + len - 1;
                case DIR_RIGHT:
                        base += 1;
                        if (base % len) return base;
                        return base - len;
                case DIR_UP:
                        if (base < len) return base + (len - 1) * len;
                        return base - len;
                case DIR_DOWN:
                        return (base + len) % (len * len);
                }
        }
        void slither()
        {
                size_t old_head = sites[hd];
                size_t old_tail = sites[hd ? hd - 1 : len - 1];
                size_t new_head, new_tail;
                Dir d = Dir(rng.uniform() * DIR_NUM);
                if (rng.uniform() < 0.5) {
                        new_tail = neighbor(old_tail, d);
                        arena[old_head] = false; // remove head site
                        if (arena[new_tail]) { // overlap
                                arena[old_head] = true;
                                return;
                        }
                        sites[hd] = new_tail;
                        arena[new_tail] = true;
                        hd = (hd + 1) % len;
                        new_head = sites[hd];
                }
                else {
                        new_head = neighbor(old_head, d);
                        arena[old_tail] = false; // remove tail site
                        if (arena[new_head]) { // overlap
                                arena[old_tail] = true;
                                return;
                        }
                        sites[hd ? hd - 1 : len - 1] = new_head;
                        arena[new_head] = true;
                        hd = hd ? hd - 1 : len - 1;
                        new_tail = sites[hd ? hd - 1 : len - 1];
                }
                // update displacement
                dispx += delta_x(old_tail, new_tail)
                         - delta_x(old_head, new_head);
                dispy += delta_y(old_tail, new_tail)
                         - delta_y(old_head, new_head);
        }
        double disp2()
        {
                return dispx * dispx + dispy * dispy;
        }
};
int main()
{
        for (size_t l = 100; l <= 1600; l *= 2) {
                Snake s(l);
```

Since the slithering process is equal likely to go forward and back, it is similar to a random walk that the time it takes to have all segments of the snake be moved at least once grows

quadratically with the length of the snake. Below, we set the warm-up time and the sampling time of the process accordingly.

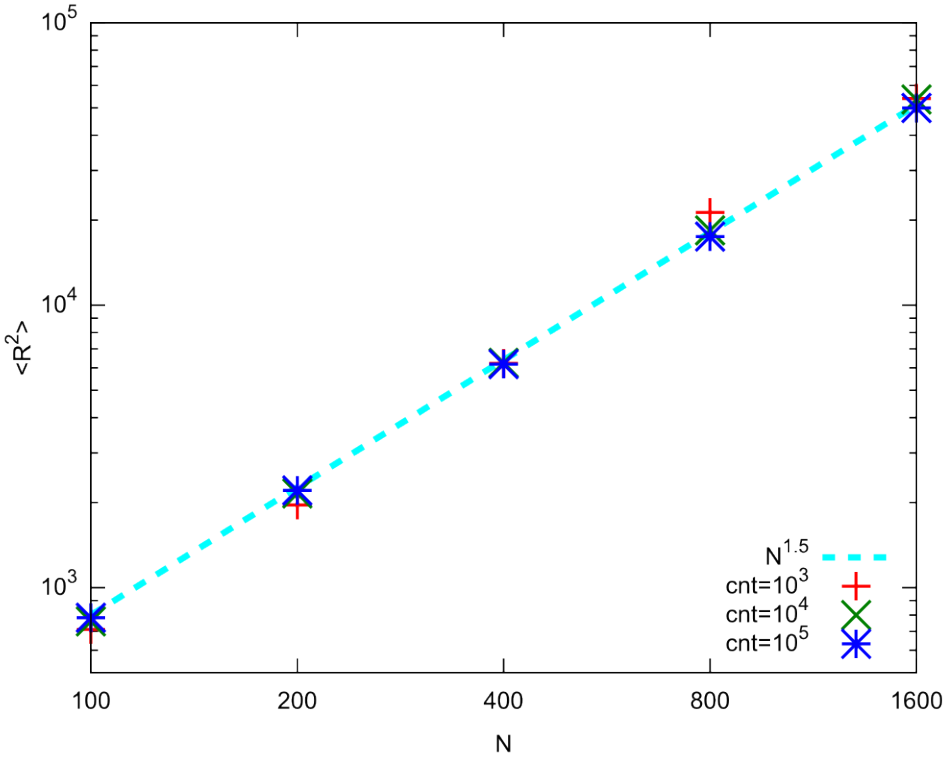
```
                for (size_t i = 0; i < l * l; i ++) s.slither(); // warm up
                size_t cnt = 0;
                double dis = 0;
                while (cnt < 1000) {
                        for (size_t i = 0; i < l * l / 100; i ++) s.slither();
                        cnt ++;
                        dis += s.disp2();
                }
                std::cout << l << '\t' << dis / cnt << std::endl;
        }
        return 0;
}
```

To get a good statistics, the calculations can take sometime to complete. Following are the results for different limits on the cnt variable. The calculations are done on a 2GHz Intel Celeron (E1400) desktop.

cnt	1000	10000	100000
len: 100	711.06	759.583	782.64
len: 200	1961.27	2155.93	2211.35
len: 400	6228.41	6248.29	6181.57
len: 800	21309	18407	17489.1
len: 1600	53902.8	53464.8	49944.3
Used Time	0m7.104s	1m5.096s	10m46.092s

3. 分析 2.所得數據，求 $\langle R^2 \rangle_n \sim n^{2\nu}$ 中的 ν 值。

The results are plotted below.



Judging by eye, the results are pretty close to the theoretical value of $2\nu = 1.5$. Here, we

simply use the “fit” command of gnuplot to do a linear regression on the double-log scale. Gnuplot:

```
gnuplot> f(x)=m*x+b
gnuplot> fit f(x) 'snake0.txt' using (log($1)):(log($2)) via m,b


 Iteration 0
 WSSR        : 0.0216825         delta(WSSR)/WSSR   : 0
 delta(WSSR) : 0                 limit for stopping : 1e-05
 lambda      : 4.35078

initial set of free parameter values

m               = 1.59301
b               = -0.795098
*******************
After 1 iterations the fit converged.
final sum of squares of residuals : 0.0216825
rel. change during last iteration : 0

degrees of freedom    (FIT_NDF)                        : 3
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0850147
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.0072275

Final set of parameters            Asymptotic Standard Error
=======================            ==========================

m               = 1.59301          +/- 0.03879      (2.435%)
b               = -0.795098        +/- 0.2355       (29.62%)


correlation matrix of the fit parameters:

               m      b
m               1.000
b              -0.987  1.000
gnuplot>
```

Such fit performed by gnuplot is similar to the minimization of the mean-square error that we have done previously. The scaling exponent 2ν is highlighted above. We do this for the three sets of data and can see it converges to the theoretical value nicely:

cnt	1000	10000	100000
2ν	1.59301	1.53683	1.49751
ν	.796505	.768415	.748755