2011-03-28 作業解答

1. 完成矩陣的實作：建立 mat.cc 檔，配合 mat.hh 及 mat_exp.cc（存在 CP SSH 伺服器的/usr/local/src/hw5 目錄中）編譯及連結出可執行檔。（輸出結果如前）

There are actually many points to note in the implementation of the Matrix class. We have constructors that are used to initialize objects of the class. In these constructors we need to take care of the proper allocation of memory for the matrix elements and fill in their initial values. This block of memory is to be disposed by the destructor of the class.

We use a one dimension array for the storage of the two dimensional matrix. Thus, we have an element access function that linearizes two dimensional indexes into one. For a complete class, there actually should be an overloading “const” version of the “elem()” member function allowing element access to a constant value of Matrix.

The “static” member functions of a class are pertinent to the class but do not associate with any particular object of the class. Practically, they are just like “friend” functions that can access the non-public part of a class but are enclosed in the class's namespace.

External overloading functions are for syntactical conveniences. They can not or are not proper to be the members of the class that they are dealing with. They can try to fulfill their roles through member functions of the class. But if that's insufficient or not feasible, we can just make them friends of the class.

C++:

```
// mat.cc
#include <mat.hh>
Matrix::Matrix(size_t nr, size_t nc) :
      n_row(nr),
      n_col(nc)
{
      vals = new double[n_row * n_col];
}

Matrix::Matrix(const Matrix & m) : // copy constructor
      n_row(m.n_row),
      n_col(m.n_col)
{
      vals = new double[n_row * n_col];
      for (size_t i = n_row * n_col; i --;) vals[i] = m.vals[i];
}

Matrix::Matrix(size_t nr, size_t nc, double const * v) :
      n_row(nr),
      n_col(nc)
{
      vals = new double[n_row * n_col];
      for (size_t i = n_row * n_col; i --;) vals[i] = v[i];
}

Matrix::~Matrix()
{
      delete [] vals;
}

double & Matrix::elem(size_t ri, size_t ci) // element access
{
      return vals[ri * n_col + ci];
```

```
}

// overloading operators
Matrix Matrix::operator+(const Matrix & m) const
{
	if (n_col != m.n_col || n_row != m.n_row) throw;
	Matrix r = * this;
	for (size_t i = n_row * n_col; i --;) r.vals[i] += m.vals[i];
	return r;
}

Matrix Matrix::operator-(const Matrix & m) const
{
	if (n_col != m.n_col || n_row != m.n_row) throw;
	Matrix r = * this;
	for (size_t i = n_row * n_col; i --;) r.vals[i] -= m.vals[i];
	return r;
}

Matrix Matrix::operator*(const Matrix & m) const
{
	if (n_col != m.n_row) throw;
	Matrix r(n_row, m.n_col);
	for (size_t i = 0; i < n_row; i ++)
	 for (size_t j = 0; j < m.n_col; j ++) {
		double v = 0;
		for (size_t k = 0; k < n_col; k ++)
			v += vals[i * n_col + k] * m.vals[k * n_col + j];
		r.vals[i * r.n_col + j] = v;
	}
	return r;
}

Matrix Matrix::operator*(double v) const
{
	Matrix r = * this;
	for (size_t i = n_row * n_col; i --;) r.vals[i] *= v;
	return r;
}

// making unit Matrix
Matrix Matrix::unit(size_t sz)
{
	Matrix mm(sz + 1, sz + 1);
	Matrix m(sz, sz);
	for (size_t i = 0; i < sz; i ++)
	  for (size_t j = 0; j < sz; j ++) m.elem(i, j) = (i == j ? 1 : 0);
	return m;
}

void Matrix::stream_out(std::ostream & o) const
{
	for (size_t ri = 0; ri < n_row; ri ++) {
		double * row = vals + ri * n_col;
		o << * row;
		for (size_t ci = 1; ci < n_col; ci ++) {
			o << "\t" << row[ci];
		}
		o << '\n';
	}
}
```

```
Matrix operator*(double v, const Matrix & m)
{
	return m * v;
}

std::ostream & operator<<(std::ostream & o, const Matrix & m)
{
	m.stream_out(o);
	return o;
}
```

2. 行星軌道：給定下列方程的初始條件

$$\frac{d^2x}{dt^2}=\frac{x}{(x^2+y^2)^{3/2}} \qquad \frac{d^2y}{dt^2}=\frac{y}{(x^2+y^2)^{3/2}}$$

以及差分時間 τ，寫一程式計算 x 及 y 在 $0<t<T$ 間的軌跡。將 $x(0)=2$, $y(0)=0$, $v_x(0)=v_y(0)=0.2$ 及 $\tau=0.02, 0.01, 0.001$; $T=100$ 的運動軌跡作成圖形檔。

After transforming the second order ordinary differential equations into a system of first order difference equations, it remains a matter of translating the algorithm into your favorite programming language.

C++:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
	double dt, x, y, vx, vy, T;
	cout << "# Input: time_step x0 y0 vx0 vy0 T\n";
	cin >> dt >> x >> y >> vx >> vy >> T;
	cout << "# Got: " << dt << ' ' << x << ' ' << y << ' '
	     << vx << ' ' << vy << ' ' << T << '\n';
	double t = 0;
	while (t < T) {
		double r = x * x + y * y;
		r = pow(r, 1.5);
		double ax = - x / r;
		double ay = - y / r;
		x += dt * vx;
		y += dt * vy;
		vx += dt * ax;
		vy += dt * ay;
		t += dt;
		cout << t << '\t' << x << '\t' << y << '\n';
	}
	return 0;
}
```

While the lecture described the Euler's method of integration, the demo plot in the slides was actually mistakenly generated from the Leapfrog method. Using the Euler's method as implemented above, the orbit barely closes for a timestep τ as small as 0.0001. It runs away before completing one revolution for any τ greater than 0.001. This is mainly due to the errors in Euler's method tend to be one sided.