

Program for wave packets in the Schrödinger equation

We would like to solve the time-dependent Schrödinger equation

$$i\hbar \frac{\partial}{\partial t} \psi(x, t) = -\frac{\hbar^2}{2m} \frac{\partial^2}{\partial x^2} \psi + V(x) \psi .$$

using the Crank-Nicolson discretization formula

$$\Psi^{n+1} = \left(\mathbf{I} + \frac{i\tau}{2\hbar} \mathbf{H} \right)^{-1} \left(\mathbf{I} - \frac{i\tau}{2\hbar} \mathbf{H} \right) \Psi^n ,$$

which is stable, conserves probability, and is $\mathcal{O}(\tau^3)$ accurate in time. This matrix formula can be simplified by noting that

$$\left(\mathbf{I} + \frac{i\tau}{2\hbar} \mathbf{H} \right)^{-1} \left(\mathbf{I} - \frac{i\tau}{2\hbar} \mathbf{H} \right) = \mathbf{Q}^{-1} - \mathbf{I} ,$$

where

$$\mathbf{Q} = \frac{1}{2} \left(\mathbf{I} + \frac{i\tau}{2\hbar} \mathbf{H} \right)$$

has the same tridiagonal (or cyclic tridiagonal) form as $\mathbf{H}$. The discretized Schrödinger equation can thus be solved in two steps: First solve the linear equation

$$\mathbf{Q}\chi = \Psi^n , \quad \chi = \mathbf{Q}^{-1} \Psi^n ,$$

for χ from the known Ψ^n , and then compute

$$\Psi^{n+1} = \chi - \Psi^n .$$

Solving a Tridiagonal System of Equations

The inverse of a tridiagonal matrix is *not* tridiagonal in form! For example,

$$\begin{pmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{pmatrix}^{-1} = -\frac{1}{6} \begin{pmatrix} 5 & 4 & 3 & 2 & 1 \\ 4 & 8 & 6 & 4 & 2 \\ 3 & 6 & 9 & 6 & 3 \\ 2 & 4 & 6 & 8 & 4 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}.$$

This implies that it would require $\mathcal{O}(N^2)$ operations to compute

$$\mathbf{Q}^{-1}\Psi^n,$$

if $\mathbf{Q}^{-1}$ were known explicitly. In addition, storing the value of $\mathbf{Q}^{-1}$ requires $\mathcal{O}(N^2)$ memory locations compared with only $\mathcal{O}(N)$ to store the non-zero elements of $\mathbf{Q}$. The *Thomas algorithm* solves the tridiagonal system *without* constructing $\mathbf{Q}^{-1}$. The tridiagonal matrix elements of $\mathbf{Q}$ in the Schrödinger problem have the following values:

$$\beta_j = \frac{1}{2} + \frac{i\tau\hbar}{4mh^2} + \frac{i\tau}{4\hbar}V_j, \quad j = 1, \dots, N,$$

$$\alpha_j = \gamma_j = -\frac{i\tau\hbar}{8mh^2}, \quad j = 1, \dots, N-1.$$

The Thomas algorithm applied to this problem can be summarized as follows:

- Set $\beta'_1 = \beta_1$ and $b'_1 = \Psi_1^n$.
- Perform the forward elimination iteration

$$\left. \begin{aligned} \beta'_j &= \beta_j - \frac{\alpha_{j-1}}{\beta'_{j-1}}\gamma_{j-1} \\ b'_j &= \Psi_j^n - \frac{\alpha_{j-1}}{\beta'_{j-1}}b'_{j-1} \end{aligned} \right\} \quad j = 2, \dots, N.$$

- Set $\chi_N = b'_N/\beta'_N$.
- Perform the backsubstitution iteration

$$\chi_j = \frac{b'_j - \gamma_j \chi_{j+1}}{\beta'_j}, \quad j = N, \dots, 1.$$

Periodic Boundary Conditions: Sherman-Morrison Formula

It requires a little more work to solve the equation

$$\mathbf{Q}\chi = \Psi^n,$$

when $\mathbf{Q}$ is cyclic tridiagonal,

$$\begin{pmatrix} \beta_1 & \gamma_1 & 0 & \dots & 0 & 0 & \alpha_N \\ \alpha_1 & \beta_2 & \gamma_2 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & \alpha_{N-2} & \beta_{N-1} & \gamma_{N-1} \\ \gamma_N & 0 & 0 & \dots & 0 & \alpha_{N-1} & \beta_N \end{pmatrix}, \quad \alpha_N = \gamma_N = -\frac{i\tau\hbar}{8mh^2}.$$

This matrix can be simplified by defining two vectors $\vec{u}$ and $\vec{v}$ and their *outer product* $\vec{u} \otimes \vec{v}$

$$\vec{u} = \begin{pmatrix} \beta \\ 0 \\ \vdots \\ 0 \\ \gamma_N \end{pmatrix}, \quad \vec{v} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ \alpha_N/\beta \end{pmatrix}, \quad \vec{u} \otimes \vec{v} = \begin{pmatrix} \beta & 0 & 0 & \dots & 0 & 0 & \alpha_N \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ \gamma_N & 0 & 0 & \dots & 0 & 0 & \alpha_N\gamma_N/\beta \end{pmatrix},$$

where $\beta \neq 0$ and will be chosen later. We can then decompose

$$\mathbf{Q} = \mathbf{A} + \vec{u} \otimes \vec{v}.$$

The matrix $\mathbf{A}$ is tridiagonal. The *Sherman-Morrison formula* allows us to solve for $\chi = \mathbf{Q}^{-1}\Psi^n$ by first solving the two equations

$$\mathbf{A}\vec{y} = \Psi^n, \quad \mathbf{A}\vec{z} = \vec{u},$$

for $\vec{y}$ and $\vec{z}$ using the Thomas algorithm, and then computing

$$\chi = \vec{y} - \left[\frac{\vec{v} \cdot \vec{y}}{1 + (\vec{v} \cdot \vec{z})} \right] \vec{z}.$$

Numerical Recipes recommends choosing $\beta = -\beta_1$ to avoid potential loss of precision in applying the Thomas algorithm to the matrix $\mathbf{A}$. For a derivation of the Sherman-Morrison formula and discussion of cyclic tridiagonal matrices, see Section 2.7 of *Numerical Recipes*.

Program for wave packets in the Schroedinger equation

```
// Program for wave packets in the Schroedinger equation

#include <cmath>
#include <complex>
#include <cstdlib>
#include <iostream>

using namespace std;

#include <GL/glut.h>

const double pi = 4*atan(1.0); // pi
int N = 200;                  // number of grid points
double L = 100;                // system extends from -L/2 to L/2
double h = L / (N - 1);       // grid size
```

```
double h_bar = 1;           // natural units
double mass = 1;            // natural units
double tau = 1;             // time step
double *x;                  // coordinates of grid points
bool periodic = true;       // false = oo potential, true = periodic

// Potential V(x)
double V0 = 1.0;            // height of potential well
double Vwidth = 10;         // width of potential well
double Vcenter = L / 4;     // center of potential well

double V (double x) {
    double halfWidth = fabs(0.5 * Vwidth);
    if (abs(x - Vcenter) <= halfWidth)
        return V0;
    else
        return 0;
}

// vectors to represent the matrix Q for sparse matrix routines
complex<double> *alpha, *beta, *gamma, *chi;

complex<double> *psi;        // complex wavefunction

// Initial wave packet
double x0 = - L / 4;        // location of center
double velocity = 1.0;       // average velocity
double k0;                  // average wavenumber
double sigma0 = L / 10;     // width of wave packet
```

```
double Norm_psi;                // norm of psi

void getInput() {
    cout << " Wave packet motion in the Schroedinger equation\n"
         << " -----\n";
    cout << " Enter size L of the 1-D region:  ";
    cin >> L;
    cout << " Enter number of grid points:  ";
    cin >> N;
    cout << " Enter integration time step:  ";
    cin >> tau;
    cout << " Enter position of center of potential step:  ";
    cin >> Vcenter;
    cout << " Enter height of potential step:  ";
    cin >> V0;
    cout << " Enter width of potential step:  ";
    cin >> Vwidth;
    cout << " Enter position of wave packet:  ";
    cin >> x0;
    cout << " Enter velocity of wave packer:  ";
    cin >> velocity;
    cout << " Enter width of wave packet:  ";
    cin >> sigma0;
}

double t = 0;                    // time

void initialize () {
    static int oldN = 0;
```

```
if (oldN != N) {
    if (oldN != 0) {
        delete [] x;
        delete [] psi;
        delete [] alpha;
        delete [] beta;
        delete [] gama;
    }
    // reallocate arrays
    x = new double[N+1];
    psi = new complex<double>[N+1];
    alpha = new complex<double>[N+1];
    beta = new complex<double>[N+1];
    gama = new complex<double>[N+1];
    oldN = N;
}
// reset the lattice
h = L / (N - 1);
for (int i = 1; i <= N; i++)
    x[i] = h * (i - 1) - L / 2;
// initialize the packet
k0 = mass * velocity / h_bar;
Norm_psi = 1 / sqrt(sigma0 * sqrt(pi));
for (int i = 1; i <= N; i++) {
    double expFactor = exp(-(x[i] - x0) * (x[i] - x0)
                           / (2 * sigma0 * sigma0));
    psi[i] = complex<double>(Norm_psi * cos(k0 * x[i]) * expFactor,
                           Norm_psi * sin(k0 * x[i]) * expFactor);
}
```

```

// elements of tridiagonal matrix Q = (1/2)(1 + i tau H / (2 hbar))
for (int j = 1; j <= N; j++) {
    complex<double> i(0.0, 1.0);
    beta[j] = 0.5 + i * tau / (4 * h_bar) *
        (V(x[j]) + h_bar * h_bar / (mass * h * h));
    alpha[j] = gama[j] = - i * tau * h_bar / (8 * mass * h * h);
}
t = 0;
}

```

```

void tri_ge (complex<double> *alpha, complex<double> *beta,
             complex<double> *gama, complex<double> *b,
             complex<double> *x, int N) {
    complex<double> *betaPrime = new complex<double>[N+1];
    complex<double> *bPrime = new complex<double>[N+1];
    // Perform forward elimination
    betaPrime[1] = beta[1];
    bPrime[1] = b[1];
    for (int i = 2; i <= N; i++) {
        complex<double> coeff = alpha[i-1] / betaPrime[i-1];
        betaPrime[i] = beta[i] - coeff * gama[i-1];
        bPrime[i] = b[i] - coeff * bPrime[i-1];
    }
    // Perform back substitution
    x[N] = bPrime[N] / betaPrime[N];
    for (int i = N-1; i > 0; i--)
        x[i] = (bPrime[i] - gama[i] * x[i+1]) / betaPrime[i];
    delete [] betaPrime;
    delete [] bPrime;
}

```

```
}

void cyclic (complex<double> *alpha, complex<double> *beta,
            complex<double> *gama, complex<double> *b,
            complex<double> *x, int N) {
    complex<double> *betaPrime = new complex<double>[N+1];
    complex<double> *u = new complex<double>[N+1];
    complex<double> *z = new complex<double>[N+1];
    complex<double> c = - beta[1];
    betaPrime[1] = beta[1] - c;
    for (int i = 2; i < N; i++)
        betaPrime[i] = beta[i];
    betaPrime[N] = beta[N] - alpha[N] * gama[N] / c;
    tri_ge(alpha, betaPrime, gama, b, x, N);
    u[1] = c;
    complex<double> zero(0.0, 0.0);
    for (int i = 2; i < N; i++)
        u[i] = zero;
    u[N] = alpha[N];
    tri_ge(alpha, betaPrime, gama, u, z, N);
    complex<double> fact = (x[1] + alpha[N] * x[N] / c)
        / (1.0 + z[1] + alpha[N] * z[N] / c);
    for (int i = 1; i <= N; i++)
        x[i] -= fact * z[i];
    delete [] betaPrime;
    delete [] u;
    delete [] z;
}
```

```
void takeStep () { // time step using sparse matrix routines
    chi = new complex<double>[N+1];
    if (periodic)
        cyclic(alpha, beta, gama, psi, chi, N);
    else
        tri_ge(alpha, beta, gama, psi, chi, N);
    for (int j = 1; j <= N; j++)
        psi[j] = chi[j] - psi[j];
    delete [] chi;
    t += tau;

    glutPostRedisplay();
}

void display() {
    glClear(GL_COLOR_BUFFER_BIT);

    // draw real part of psi
    glColor3ub(0, 0, 255);
    glBegin(GL_LINE_STRIP);
        for (int j = 2; j <= N; j++) {
            glVertex2d(x[j-1], psi[j-1].real());
            glVertex2d(x[j], psi[j].real());
        }
    glEnd();

    // draw imaginary part of psi
    glColor3ub(0, 255, 0);
    glBegin(GL_LINE_STRIP);
```

```
    for (int j = 2; j <= N; j++) {
        glVertex2d(x[j-1], psi[j-1].imag());
        glVertex2d(x[j], psi[j].imag());
    }
glEnd();

// draw probability
glColor3ub(255, 0, 0);
glBegin(GL_LINE_STRIP);
    double pOld = psi[1].real() * psi[1].real() +
        psi[1].imag() * psi[1].imag();
    for (int j = 2; j <= N; j++) {
        double p = psi[j].real() * psi[j].real() +
            psi[j].imag() * psi[j].imag();
        glVertex2d(x[j-1], 4 * pOld);
        glVertex2d(x[j], 4 * p);
        pOld = p;
    }
glEnd();

// draw potential well
glColor3ub(255, 0, 255);
glBegin(GL_LINE_STRIP);
    double Vold = V(x[1]);
    for (int j = 2; j <= N; j++) {
        double Vnew = V(x[j]);
        glVertex2d(x[j-1], 0.2 * Vold);
        glVertex2d(x[j], 0.2 * Vnew);
        Vold = Vnew;
    }
glEnd();
```

```
    }
    glEnd();

    glutSwapBuffers();
}

void reshape(int w, int h) {
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-L / 2, L / 2, -0.3, 0.3);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

bool running;

void mouse(int button, int state, int x, int y) {
    switch (button) {
    case GLUT_LEFT_BUTTON:
        if (state == GLUT_DOWN) {
            if (running) {
                glutIdleFunc(NULL);
                running = false;
            } else {
                glutIdleFunc(takeStep);
                running = true;
            }
        }
    }
}
```

```
        break;
    default:
        break;
    }
}

int main(int argc, char *argv[]) {
    glutInit(&argc, argv);
    getInput();
    initialize();
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(600, 400);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Wave packet motion in Schroedinger's equation");
    glClearColor(1.0, 1.0, 1.0, 0.0);
    glShadeModel(GL_FLAT);
    glutDisplayFunc(display);
    glutMouseFunc(mouse);
    glutReshapeFunc(reshape);
    glutMainLoop();
}
```