

PHY 411-506 Computational Physics II
Chapter 12: Interdisciplinary Topics
Lecture 9

Friday April 25, 2008

Lecture Outline

| | |
|---|----------|
| Two dimensional sandpile automaton | 3 |
| Model and local update rule | 3 |
| A typical steady state | 6 |
| Self-organized criticality | 6 |
| Two-dimensional sandpile automaton simulation | 8 |

Two dimensional sandpile automaton

- The one-dimensional sandpile automaton has a very simple and rather uninteresting steady state behavior: the pile assumes the critical slope and any grain added to the pile slides off the edge without disturbing the pile.
- P. Bak, C. Tang and K. Wiesenfeld, *Phys. Rev. Lett.* **59**, 381 (1987) introduced a very simple two-dimensional sandpile automaton model which has complex and interesting behavior.
- A realistic two-dimensional sandpile model would have a two-dimensional array of columns of sand grains. At each column, one would define a *vector* slope with components in the x and y directions.

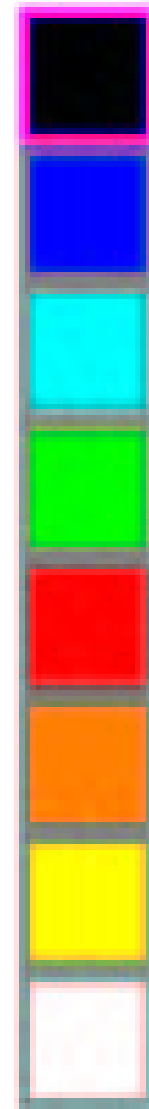
Model and local update rule

- The Bak-Tang-Wiesenfeld model has a single *scalar* slope, which can take 8 discrete values

$$s_{i,j} = 0, 1, 2, 3, 4, 5, 6, 7,$$

at each column position i, j . The column i, j is *unstable* if $s_{i,j} > 3$. The sandpile is unstable if any column is unstable.

- The sandpile is updated with the following local rule:
 - ◇ if $s_{i,j} > 3$, then remove 4 units of slope from column i, j and add one unit of slope to each of the 4 neighboring columns.
 - ◇
- This rule is applied synchronously to all columns in the pile, and repeated until a steady state is reached.
- To make the model well defined, *boundary conditions* need to be specified at the edges of of the sandpile.
 - ◇ One simple possibility is to use *open* boundary conditions: if a grain (strictly a unit of slope) falls off the edge of the pile, it disappears.
 - ◇ Other possibilites are: *periodic* boundary conditions—if a grain falls off one end of the pile it appears at the opposite end; and *flow* boundary conditions—grains are injected into the pile at each time step at one or more boundary points.



A typical steady state

- The figure shows the steady state of a pile with 200×200 columns starting from an initial unstable state with $s_{i,j} = 7$.

Self-organized criticality

- From the figure, it is clear that a steady state of this model can be extremely complex and intricate.
- The state is *critical*, which means that if a single grain of sand is added to it, an avalanche of toppling columns is triggered, and the system settles down to a different critical state.
- These events are so complicated that it is impossible to predict what will happen without actually running the simulation.
- Let n_t be the number of topplings in an avalanche. By generating a large number of avalanches, one can measure the distribution of numbers $N(n_t)$ of avalanches as a function of avalanche size n_t .

- For the sandpile model, the distribution has a *power law* behavior:

$$N(n_t) \sim \frac{1}{n_t^b},$$

where b is the *exponent* of the power law.

- ◇ Power-law behavior implies that the avalanches have no natural scale or size: events of all sizes can occur in the distribution.
 - If the exponent b is very small, then events of all sizes are equally probable.
 - If $b > 0$ then larger events are less likely than smaller events.
- ◇ The special characteristic of a power law is that the *ratio* of events which differ in magnitude by a fixed multiplicative factor is *independent* of the size of the events. If the factor is 10, for example, then

$$\frac{N(10n_t)}{N(n_t)} = \frac{1}{10^b}.$$

- For the sandpile models the exponent is found to be of order one: $b \simeq 1$.
- Bak, Tang and Wiesenfeld called this behavior *Self-Organized Criticality*.

The sandpile evolves or *organizes itself* without any external influence into a complex *critical* state where events of all sizes can occur.

Two-dimensional sandpile automaton simulation

- The following program implements the Bak-Tang-Wiesenfeld model using OpenGL graphics.
- It is modeled on Mike Creutz's xsand.c

sand2.cpp

```
/*  
    Two-dimensional sandpile automaton  
*/  
  
#include <cstdlib>  
#include <iostream>  
using namespace std;  
  
#ifdef __APPLE__  
# include <GLUT/glut.h>  
#else  
# include <GL/gl.h>  
# include <GL/glu.h>
```

```
# include <GL/glut.h>
#endif

int L = 200;           // number of columns in x and y
int **slope;          // slope at each column
bool **stable;        // is the column stable?
GLubyte *image;       // bitmap image of pile
int selectedSlope = 4; // initial slope
bool addGrain;        // add grain to steady state?
```

- Note the arrays above. A two-dimensional integer array will represent the pile, and a two-dimensional boolean array records whether a column is stable or not.
- We will use a bitmapped image to draw the pile on the screen.
- The following function allocates memory for the arrays.

sand2.cpp

```
void allocate() {
    static int oldL = 0;
    if (oldL != L) {
        if (slope) {
            for (int i = 0; i < oldL; i++) {
```

```
        delete [] slope[i];
        delete [] stable[i];
    }
    delete [] slope;
    delete [] stable;
    delete [] image;
}
slope = new int* [L];
stable = new bool* [L];
image = new GLubyte [9 * L * L];
for (int i = 0; i < L; i++) {
    slope[i] = new int [L];
    stable[i] = new bool [L];
}
oldL = L;
}
}

void initialize() {
    allocate();
    for (int i = 0; i < L; i++) {
        for (int j = 0; j < L; j++) {
            slope[i][j] = selectedSlope;
        }
    }
}
```

}

- The function above initializes the pile to the slope selected by the user.
- The two functions below test whether the pile is stable by checking the slope at each column, and implements the toppling rule.

sand2.cpp

```
bool pileIsStable() {
    bool isStable = true;
    for (int i = 0; i < L; i++)
        for (int j = 0; j < L; j++) {
            if (slope[i][j] > 3)
                isStable = stable[i][j] = false;
            else
                stable[i][j] = true;
        }
    return isStable;
}
```

```
void topple(int i, int j) {
    slope[i][j] -= 4;
    if (i + 1 < L)
        ++slope[i + 1][j];
}
```

```
    if (i - 1 >= 0)
        ++slope[i - 1][j];
    if (j + 1 < L)
        ++slope[i][j + 1];
    if (j - 1 >= 0)
        ++slope[i][j - 1];
}
```

- The code below implements OpenGL graphics for this sandpile system.

sand2.cpp

```
int mainWindow, sandpileWindow, colorsWindow;
int xSize, ySize, margin, colorSize;

void takeStep() {
    if (pileIsStable()) {
        if (addGrain) {
            int i = int((rand() / double(RAND_MAX)) * L);
            int j = int((rand() / double(RAND_MAX)) * L);
            ++slope[i][j];
        } else {
            glutIdleFunc(NULL);
        }
    } else {
        for (int i = 0; i < L; i++)
```

```
        for (int j = 0; j < L; j++)
            if (!stable[i][j])
                topple(i, j);
    }
    glutSetWindow(sandpileWindow);
    glutPostRedisplay();
}

GLubyte colors[8][3] = { {255, 255, 255}, {255, 255, 0},
                        {255, 127, 0},    {255, 0, 0},
                        {0, 255, 0},     {0, 255, 255},
                        {0, 0, 255},     {0, 0, 0}    };

void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    glutSwapBuffers();
    glutSetWindow(sandpileWindow);
    glutPostRedisplay();
}

void displaySandpile() {
    glClear(GL_COLOR_BUFFER_BIT);
    for (int i = 0; i < L; i++)
        for (int j = 0; j < L; j++)
            for (int k = 0; k < 3; k++)
                image[3 * (L * i + j) + k] = colors[slope[i][j]][k];
}
```

```
    glRasterPos2i(0, 0);
    glDrawPixels(L, L, GL_RGB, GL_UNSIGNED_BYTE, image);
    glutSwapBuffers();
}

void displayColors() {
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3ub(255, 0, 255);
    glRectd(0, selectedSlope * colorSize, colorSize,
            (selectedSlope + 1) * colorSize);
    for (int i = 0; i < 8; i++) {
        glColor3ubv(colors[i]);
        glRectd(2, i * colorSize + 2, colorSize - 2, (i + 1) * colorSize - 2);
    }
    glutSwapBuffers();
}

void redraw() {
    glutSetWindow(sandpileWindow);
    glutPostRedisplay();
    glutSetWindow(colorsWindow);
    glutPostRedisplay();
}

void reshape(int w, int h) {
```

```
    glVertex(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, w, 0, h);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

bool running;
bool scribble;

void reset(int menuItem) {
    switch (menuItem) {
        case 1:                // user selects "Reset" from popup
            initialize();
            redraw();
            break;
        case 2:                // user selects "Scribble" from popup
            scribble = true;
            glutSetCursor(GLUT_CURSOR_SPRAY);
            break;
        default:
            break;
    }
}
```

```
void mouseSandpile (int button, int state, int x, int y) {
    switch (button) {
    case GLUT_LEFT_BUTTON:
        if (state == GLUT_DOWN && !scribble) {
            if (running) {
                glutIdleFunc(NULL);
                running = false;
            } else {
                glutIdleFunc(takeStep);
                running = true;
            }
        } else if (state == GLUT_UP && scribble) {
            scribble = false;
            glutSetCursor(GLUT_CURSOR_INHERIT);
        }
        break;
    default:
        break;
    }
}
```

```
void motionSandpile(int x, int y) {
    if (scribble) {
        slope[L - y][x] = selectedSlope;
    }
}
```

```
        glutPostRedisplay();
    }
}

void mouseColors (int button, int state, int x, int y) {
    switch (button) {
    case GLUT_LEFT_BUTTON:
        if (state == GLUT_DOWN) {
            selectedSlope = int((1 - y / double(L)) * 8);
            redraw();
        }
        break;
    default:
        break;
    }
}

void makeSandpileWindow() {
    sandpileWindow = glutCreateSubWindow(mainWindow, margin, margin, L, L);
    glClearColor(1.0, 1.0, 1.0, 0.0);
    glShadeModel(GL_FLAT);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glutDisplayFunc(displaySandpile);
    glutReshapeFunc(reshape);
    glutMouseFunc(mouseSandpile);
}
```

```
    glutMotionFunc(motionSandpile);
    glutCreateMenu(reset);           // create a popup menu
    glutAddMenuEntry("Reset", 1);
    glutAddMenuEntry("Scribble", 2);
    glutAttachMenu(GLUT_RIGHT_BUTTON); // attach it to right mouse button
}
```

```
void makeColorsWindow() {
    colorsWindow = glutCreateSubWindow(mainWindow, 2 * margin + L, margin,
                                       colorSize, L);

    glClearColor(0.5, 0.5, 0.5, 0.0);
    glutReshapeFunc(reshape);
    glutDisplayFunc(displayColors);
    glutMouseFunc(mouseColors);
}
```

```
int main(int argc, char *argv[]) {
    glutInit(&argc, argv);
    if (argc > 1)
        L = atoi(argv[1]);
    initialize();
    margin = 20;
    colorSize = L / 8;
    xSize = L + colorSize + 3 * margin;
    ySize = L + 2 * margin;
```

```
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(xSize, ySize);
    glutInitWindowPosition(100, 100);
    mainWindow = glutCreateWindow("Two-dimensional sandpile automaton");
    glClearColor(1.0, 1.0, 1.0, 0.0);
    glShadeModel(GL_FLAT);
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    makeColorsWindow();
    makeSandpileWindow();
    glutMainLoop();
}
```