Fourier Series

Given a periodic function, *x(t), (-T <t <T)*:

$$x(t) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty}\left(a_n \cos\frac{n\pi t}{T} + b_n \sin\frac{n\pi t}{T}\right)$$

$$a_n = \frac{1}{T}\int_{-T}^{T} x(t)\cos\frac{n\pi t}{T}\ dt$$

$$b_n = \frac{1}{T}\int_{-T}^{T} x(t)\sin\frac{n\pi t}{T}\ dt$$

Complex Version

$$x(t) = \sum_{k=-\infty}^{\infty} X_k \, e^{i\frac{k\pi t}{T}}$$

$$X_k = \frac{1}{2T}\int_{-T}^{T} x(t)e^{-i\frac{k\pi t}{T}} \, dt$$

Discrete Fourier Transform

x_n is complex series in time domain; we have N samples at $\Delta t = T_r / (N-1)$.

X_k is representation in Fourier space

$$X_k = \sum_{n=0}^{N-1} x_n \, e^{-2i\pi \frac{kn}{N}}$$ for each $k = 0, 1, .., N\text{-}1$

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \; e^{2i\pi \frac{kn}{N}}$$ for each $n = 0, 1, .., \text{N-1}$

Fourier Transform Pair

$$\hat{f}(\lambda) = \int_{-\infty}^{\infty} f(y)\, e^{-2\pi i \lambda y} dy$$

$$f(y) = \int_{-\infty}^{\infty} f(\lambda)\, e^{2\pi i \lambda y} d\lambda$$

Solve PDE with Fourier transforms

$$\hat{f}_y(\lambda) = \int_{-\infty}^{\infty} \frac{\partial f(y)}{\partial y}\, e^{-2\pi i \lambda y} dy$$

$$\hat{f}_y(\lambda) = 2\pi i \lambda \hat{f}(\lambda)$$

$$\hat{f}_{yy}(\lambda) \quad = \quad -4\pi^2 \lambda\, \hat{f}(\lambda)$$

Fourier Transforms and FFT

Mild-slope equation:

$$\nabla_h \cdot \left(CC_g \nabla_h \phi\right) + \sigma^2 \frac{C_g}{C}\, \phi(x,y) = 0$$

Assume shore parallel beach (no derivatives of depth in y). After Fourier transform in *y*:

$$\Big(CC_g\ \phi(\lambda, x)\Big)_{xx} + (k^2 - \lambda^2)\, CC_g\, \phi = 0$$

Now, only need to integrate in the *x* direction. At the end, inverse transform back into the real domain.

Fast Fourier Transform

Methodology to do discrete Fourier transform form with fewer operations

DFT is O(N^2) while the Cooley-Tukey (1965) algorithm is O($N \log_2 N$)

For N = 1024,   102 times faster

cuFFT

http://docs.nvidia.com/cuda/cufft/#introduction

cuFFT Library

Algorithms highly optimized for input sizes that can be written in the form:

$N = 2^a \times 3^b \times 5^c \times 7^d$

In general the smaller the prime factor, the better the performance, i.e., powers of two are fastest.

An O(n log n) algorithm for every input data size

Single-precision (32-bit floating point) and double-precision (64-bit floating point). Single-precision transforms have higher performance than double-precision transforms.

Complex and real-valued input and output. Real valued input or output require less computations and data than complex values and often have faster time to solution. Types supported are:

- C2C - Complex input to complex output
- R2C - Real input to complex output
- C2R - Symmetric complex input to real output

1D, 2D and 3D transforms

Read more at: http://docs.nvidia.com/cuda/cufft/index.html

cuFFT Plan

cufftPlan1D(), cufftPlan2D(), or cufftPlan3D()

Create a simple plan for a 1D/2D/3D transform respectively.

cufftPlanMany() - Creates a plan supporting batched input and strided data layouts.

With the plan, cuFFT derives the internal steps that need to be taken. These steps may include multiple kernel launches, memory copies, and so on. Also allocates buffer space.

Plan can be used over and over again.

Basic FFT

```
#define NX 256
#define BATCH 10
#define RANK 1
 ...
{ cufftHandle plan;
cufftComplex *data;
 ...
cudaMalloc((void**)&data, sizeof(cufftComplex)*NX*BATCH);
cufftPlanMany(&plan, RANK, NX, &iembed, istride, idist, &oembed, ostride,
                         odist, CUFFT_C2C, BATCH);
 ...
cufftExecC2C(plan, data, data, CUFFT_FORWARD);
cudaDeviceSynchronize();
 ...
cufftDestroy(plan);
 cudaFree(data); }

# include <cufft.h>
nvcc [options] filename.cu -I/usr/local/cuda/inc -L/usr/local/cuda/lib -lcufft
```

cuFFT commands

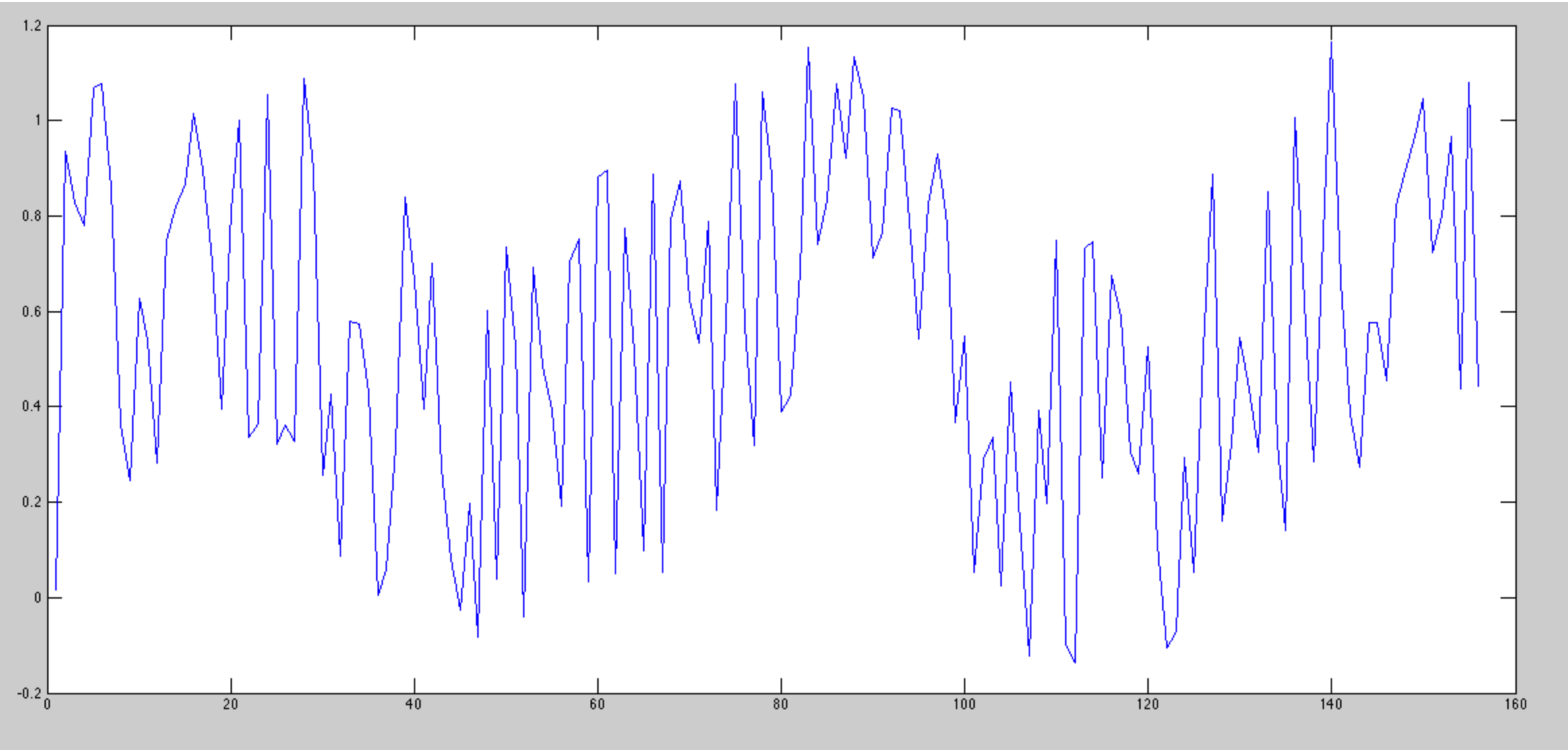
```
cufftExecC2C(cufftHandle plan, cufftComplex *idata,
                                    cufftComplex *odata, int direction);

cufftExecZ2Z(cufftHandle plan, cufftDoubleComplex *idata,
                                    cufftDoubleComplex *odata, int direction);
```

for single and double complex data. Note that the data must be on the device.

direction is forward or inverse.

Determine Power Spectrum

Signal (2048 seconds long) = T_r



Sample of 160 s
(not yet detrended)

Data

$\Delta t = 1.0$ s, $\Delta f = 1/(N\, T_r)$, $N = 2048$, $T_r = N\, \Delta t$

Compute spectrum given spectrumCLASS.cu

variance of data = sum of squares of X_k

Parseval's Theorem

OpenGL™ and CUDA

OpenGL started by Silicon Graphics in 1992 to provide 3D graphics

API (application programming interface) to specify:

- primitives: points, lines and polygons
- properties: colors, lighting, textures, etc.
- view: camera position and perspective

OpenGL can display results from GPU either indirectly or directly

OpenGL (most of main())

```
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
    glutInitWindowSize(800, 600);
    viewport[0] = 0;
    viewport[1] = 0;
    viewport[2] = 800;
    viewport[3] = 600;
    glutCreateWindow("GPUSPH:  Hit Space Bar to start!");
    initGL();
    initMenus();

    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMouseFunc(mouse);
    glutMotionFunc(motion);
    glutKeyboardFunc(key);
    glutIdleFunc(idle);

    glutMainLoop();
}
```

functions for OpenGL

```
void mouse(int button, int state, int x, int y)
{
    if (state == GLUT_UP) {
        buttonState = 0;
        reset_target();
        glutPostRedisplay();
        return;
    }
    if (button == 3) {
        zoom(-1.0/16);
        return;
    }
    if (button == 4) {
        zoom(1.0/16);
        return;
    }
    buttonState |= 1<<button;

}
```

display

```
void display()
{
     timingInfo = psystem->PredcorrTimeStep(true);

     // render
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
          psystem->drawParts(show_boundary, show_floating,
                                          show_vertex, view_field);
          glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
          problem->draw_boundary(psystem->getTime());
          problem->draw_axis();
          glutSwapBuffers();
     }


     ……
```

OpenGL

The general way to use OpenGL is to draw everything you need to draw, then show this image with a buffer swapping command.

If you need to update the image, you draw everything again, even if you only need to update part of the image.

If you want to animate objects moving on the screen, you need a loop that constantly clears and redraws the screen.

Callbacks

Can change conditions by the use of updates to data

In GPUSPH we have mb_callback and gravity_callbacks.

OpenGL.cpp

```
void Draw() {
	glClear(GL_COLOR_BUFFER_BIT);
	glColor3f(1.0, 1.0, 1.0);
	glBegin(GL_LINES);
		glVertex3f(0.25, 0.25, 0.0);
		glVertex3f(0.75, 0.75, 0.0);
	glEnd();
	glFlush();
}
void Initialize() {
	glClearColor(0.0, 0.0, 0.0, 0.0);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}

int main(int iArgc, char** cppArgv) {
	glutInit(&iArgc, cppArgv);
	glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
	glutInitWindowSize(250, 250);
	glutInitWindowPosition(200, 200);
	glutCreateWindow("FirstOpenGLWindow");
	Initialize();
	glutDisplayFunc(Draw);
	glutMainLoop();
	return 0;
}
```