Scientific Computing (Phys 2109/Ast 3100H)
I. Scientfic Software Development

SciNet HPC Consortium

University of Toronto

Winter 2013

SciNet compute • calcul CANADA

Part I

Introduction to Software Development

Lecture 4

Homework 1: Common issues

Testing and Modularity

- Testing
- Modularity

Course Project

- Project description
- Homework 2

SciNet compute • calcul CANADA

Homework 1: Common issues

SciNet
compute • calcul
CANADA

Flags

- What was up with this *-I$(GSLINC)* and *-L$(GSLLIB)*?
- These were added as an example of how to tell the compiler to look for headers (-I) and libraries (-L) in non-standard locations.
- Both -I and -L require a directory as an argument.
- In the example, these arguments were the environment variables GSLINC and GSLLIB.
- You will not have these, nor need these at this point.
- Nor do you need -lgsl or -lgslcblas.

- Good flag: -Wall : add to the CXXFLAGS variable
- Good flag: -O3 for optimization: add to CXXFLAGS variable
- Good flag: -g for debug info: in CXXFLAGS and LDFLAGS

Nice Makefile:

```
CXXFLAGS=-Wall -g -O3
LDFLAGS=-g
LDLIBS=
CXX=g++

all:  main

main.o:  main.cc
    $(CXX) -c $(CXXFLAGS) -o main.o main.cc

main:  main.o
    $(CXX) $(LDFLAGS) -o main main.o $(LDLIBS)

clean:
    rm -f main.o
```

– Most of you got the dependency thing.

– Some were missing a 'clean' target and/or an 'all' target.

Git

- The idea is to keep recording versions of your code that work.
- And while going so, keep comments of what you changed.
- Make comments meaningfull

```
$ git init
$ git add <files>
$ git commit -m '1st version of gaussian generating code'
$ ...
$ git commit -a -m 'Split it up in several files'
$ ...
```

SciNet compute • calcul CANADA

Git

```
$ git log
commit 1d82b1e257456ee66fa1143f13742f1e6e88d895
Author:  Ramses van Zon <rzon@scinet.utoronto.ca>
Date:  Thu Jan 24 11:02:55 2013 -0500

    Split it up in several files

commit 9d0dbb7fe6a3f02dc6b1b1d3bb55a77f3ab21ca9
Author:  Ramses van Zon <rzon@scinet.utoronto.ca>
Date:  Thu Jan 24 11:02:26 2013 -0500

    1st version of gaussian generating code
```

Cheat sheet from Git Tower:
www.git-tower.com/files/cheatsheet/Git_Cheat_Sheet_grey.pdf

SciNet compute • calcul CANADA

Header files

We'll look at those in depth today...

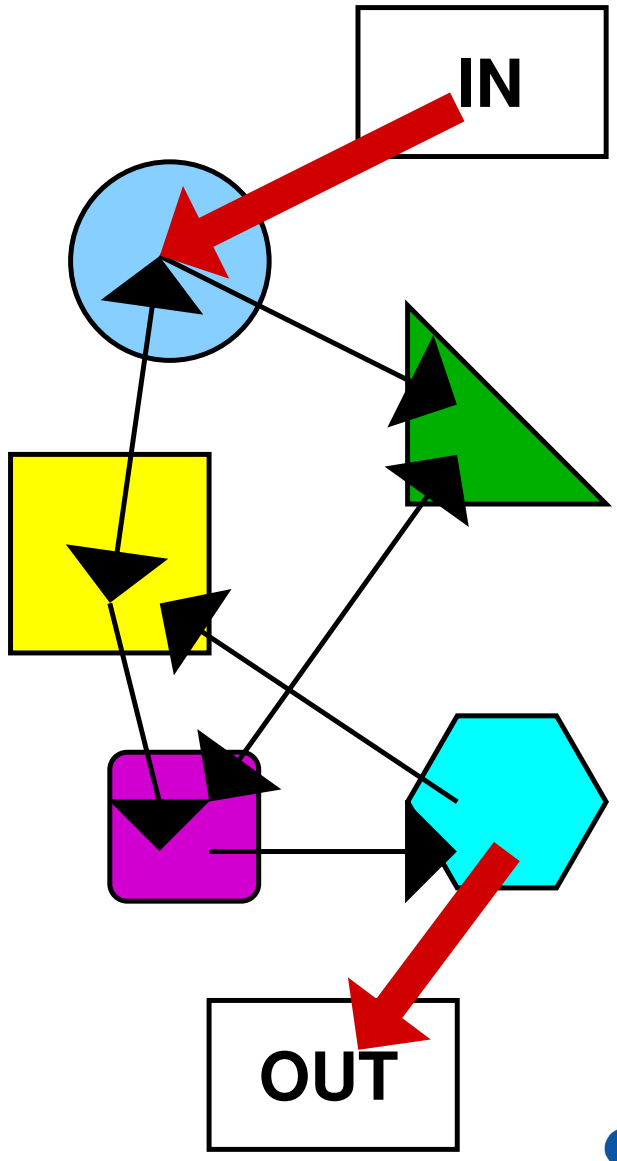
Testing

Why Testing?

- Crashes are easy to find (although sometimes harder to find root cause of)
- Wrong answers are harder.
- Slightly wrong answers are hardest of all (but most dangerous!)
- Before doing a production run, should do some small runs as a sanity check: testing.

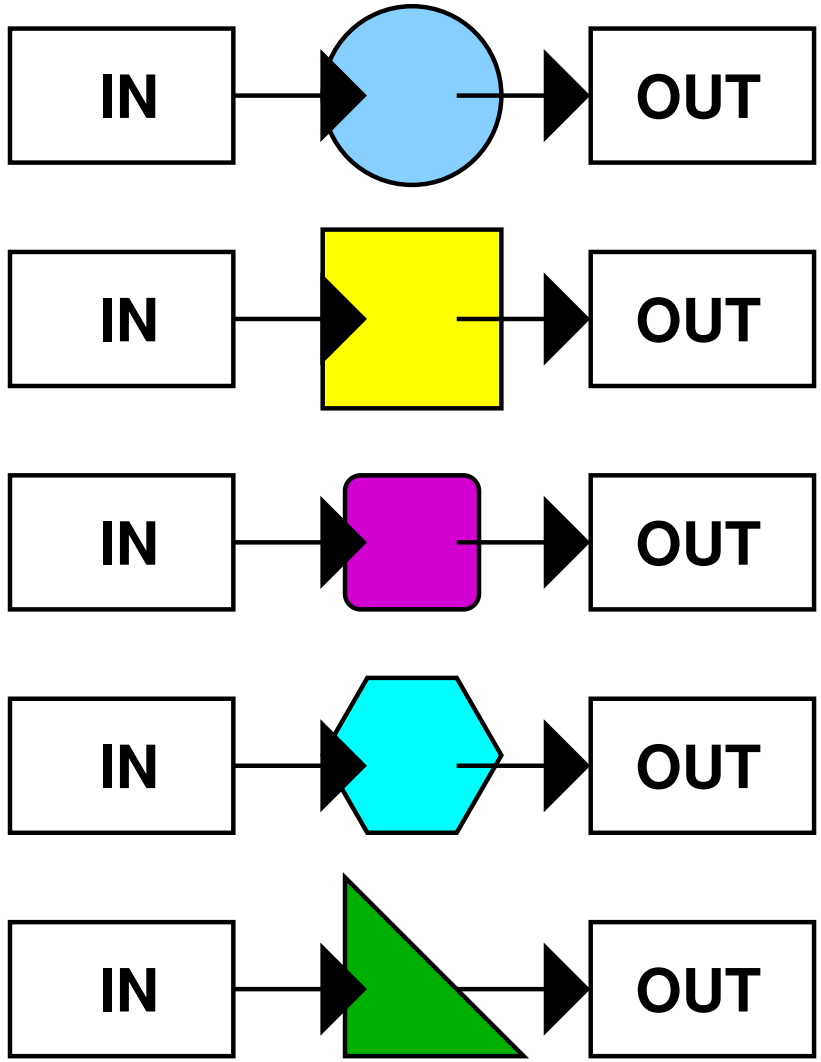
Integrated Testing

- Complicated piece of software, with many interacting parts
- Difficult to tell where a problem begins in a final answer
- Integrated testing



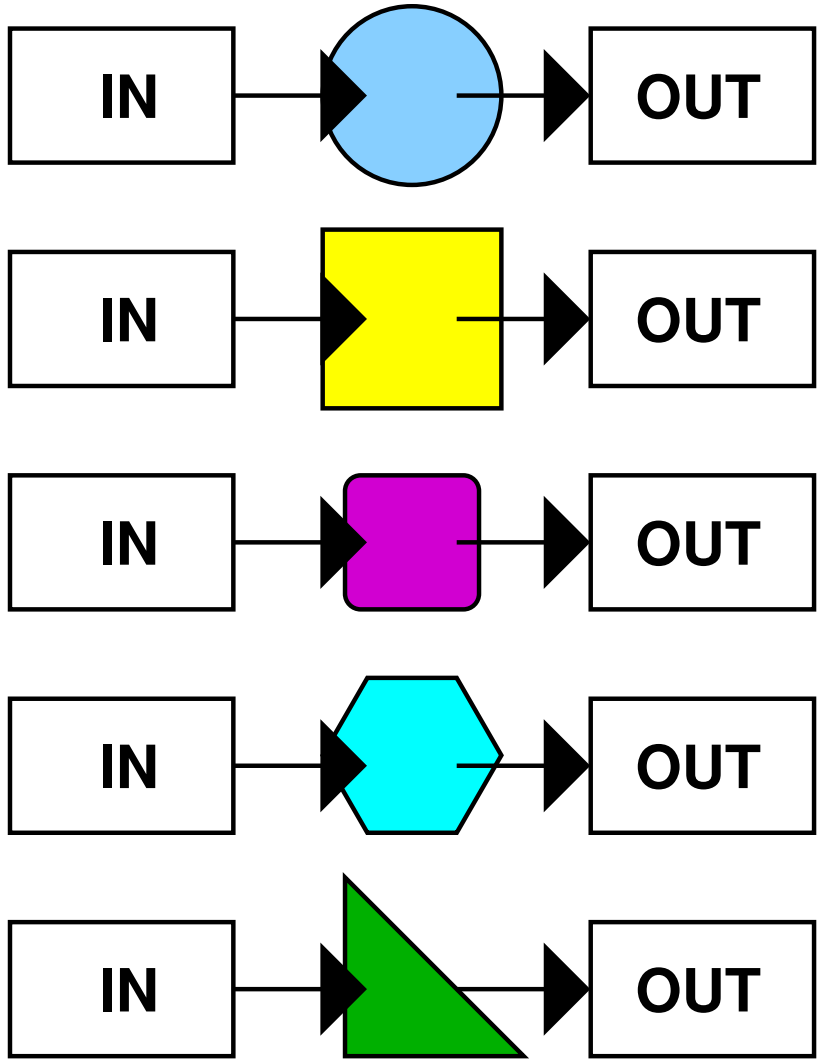
Unit Testing

- Testing major pieces of the code individually
- Comparing easy solutions, typical solutions, wierd edge cases
- Enormously speeds up, simplifies, finding problems when introduced



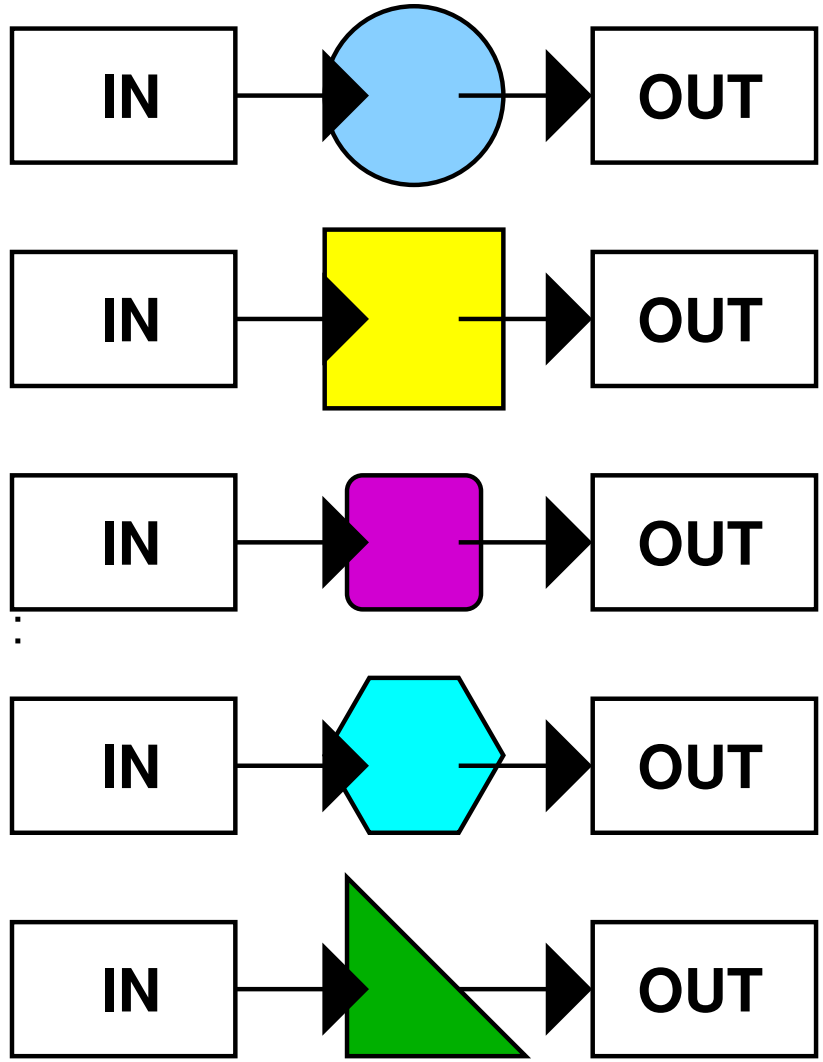
Unit Testing

- Faced with a complex piece of software which does not have testing done on it regularly (integrated and unit)?
- Just save yourself a lot of time and assume it is wrong.



Testing and Modularity

- Modular software is needed for unit testing
- Have to have separable, independent units
- Also answers the question "How much should be in module?": *What would be good independent tests?*
- Module = testable unit of functionality



Testing Frameworks

- There are lots of excellent testing frameworks that you can use – Google Tests (C++), xUnits, Check (C), Node (python), JUnit (Java).
- They're great but they have a big learing curve.
- You do not need anything that elaborate to get started with unit testing.

Simple Test Example

```
int diffusionOperator(
   float ** rho,          // density field
   int n, int m,          // size of interior grid
   float dx, float D,     // spacing, diffusion constant
   float ** Dd2rhodx2) // output
{
   // Code goes here...
   return 0;
}

bool passTestDiffusionOperatorConstant() {
   // Try diffusionOperator with a constant rho
   // If answer correct, return true, else false
}

bool passTestDiffusionOperatorGradient() {
   // Try diffusionOperator with a linearly gradient
   // If answer correct, return true, else false
}
```

Simple Test Example

```
// diffusionOperatorTests.cc
#include <iostream>

bool passTestDiffusionOperatorConstant();
bool passTestDiffusionOperatorGradient();

int main()
{
   if (not passTestDiffusionOperatorConstant()) {
      std::cerr << "diffusionOperatorConstant"
                << " FAILED" << std::endl;
      return 1;
   }
   if (not passTestDiffusionOperatorGradient()) {
      std::cerr << "diffusionOperatorGradient"
                << " FAILED" << std::endl;
      return 1;
   }
}
```

Interface v. Implementation

- The implementation – actual code – goes in the .cc file.
- The interface – what the calling code needs to know about – goes in the .h or .hh header file.
- This distinction is crucial for writing modular code.

```
// diffusionOperator.h
int diffusionOperatorConstant(float ** rho, int n,
   int m, float dx, float D, float ** Dd2rhodx2);
bool passTestDiffusionOperatorConstant();
bool passTestDiffusionOperatorGradient();
```

What does main.cc need at compile time?

Imagine we're writing a matrix in binary and in ascii:

```
// outputarray.h
void toBin(const char* n, double** x, int r, int c);
void toAsc(const char* n, double** x, int r, int c);
```

```
// main.cc

#include "outputarray.h"

int main()
{
   // ....
   toBin("data.bin", data, nrows, ncols);
   // ....
   toAsc("data.txt", data, nrows, ncols);
   // ....
}
```

```
g++ -Wall -O3 main.c -c -o main.o
```

Interface v. Implementation

- When main.cc is being compiled to a .o file, it needs to know that there exists out there somewhere functions of the form

```
void toBin(const char* n, double** x, int r, int c)
void toAsc(const char* n, double** x, int r, int c)
```

- This allows it to check the number and type of arguments and return type (interface).
- It does not need to know the implementation details (source of routine).
- **Neither does programmer of main.cc .**

Guards against multiple inclusion

- Header files can include other header files.
- Hard to figure out what header files are included already.
- Including a header file twice can lead to doubly defined entities: compiler error.
- Common antidote is a 'preprocessor guard' in every header

```
// outputarray.h
#ifndef OUTPUTARRAYH
#define OUTPUTARRAYH
void toBin(const char* n, double** x, int r, int c);
void toAsc(const char* n, double** x, int r, int c);
#endif
```

Compiling v. Linking

- main.o cannot be executed - it is missing the routines for toAsc, toBin, ...
- At link time, the .o's (or libraries) must be linked in to the executable that satisfy all those routines that the code needs.
- If you leave out one of the needed .o's, fatal error: 'symbol not found'.

What goes into the interface (.h)?

- At the very least, the function prototypes.
- There may also be constants that calling function and routine need to agree on (e.g., error codes) or definitions of data structures, classes, etc..
- Often a desciption of the module and its functions in comments.
- Usually there is one .cc/.h pair per module — often more than one routine.
- Not necessarily every function prototype.
- Internal routines should not get exposed in the .h.

What goes into the implementation (.cc)?

- Everything defined in .h which requires code that is not in the .h.
- Internal routines that are used by the .h routine.
- To ensure consistency, include the corresponding .h file at the top of the file.
- Needs to be compiled, and linked to code that uses the .h.

Why does modularity matter?

- Scientific software can be large, complex, subtle
- If each section uses internal details from each other section, have to understand the whole code at once.
- Interactions grow as (Lines of Code)^2
- This is also why global variables are bad
- **Have** to enforce boundaries between sections of code — self-contained modules of functionality.
- Makes testing easier.

More work up front

- Think about what you want the pieces of functionality to be.
- How are you going to use these routines?
- Think about everything you might want to use these routines for; *then* design the interface.
- May change a bit in the early stages, but if it changes a lot you should rethink things — you're not using the functionality the way you thought.
- Like documentation, etc. — more work upfront, much more productivity in the long run.

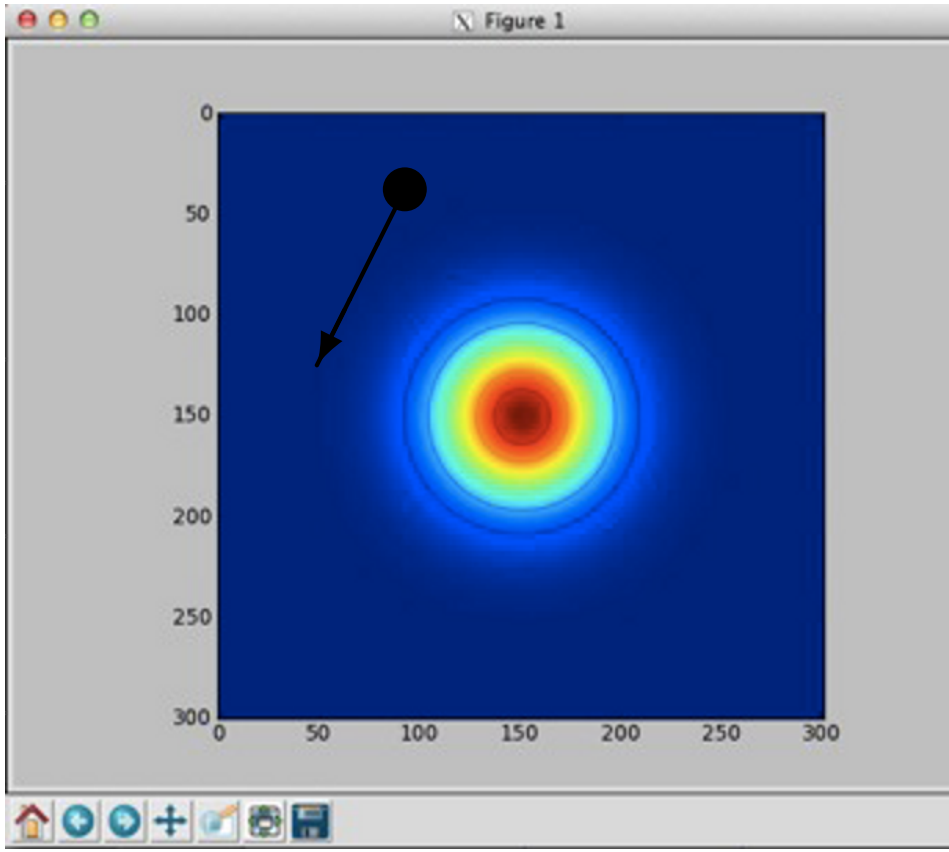
Module design

- Keep purpose of module clear.
- As simple as possible (for your own sanity).
- As general as makes sense.
- Must be testable.

Course Project

Course Project

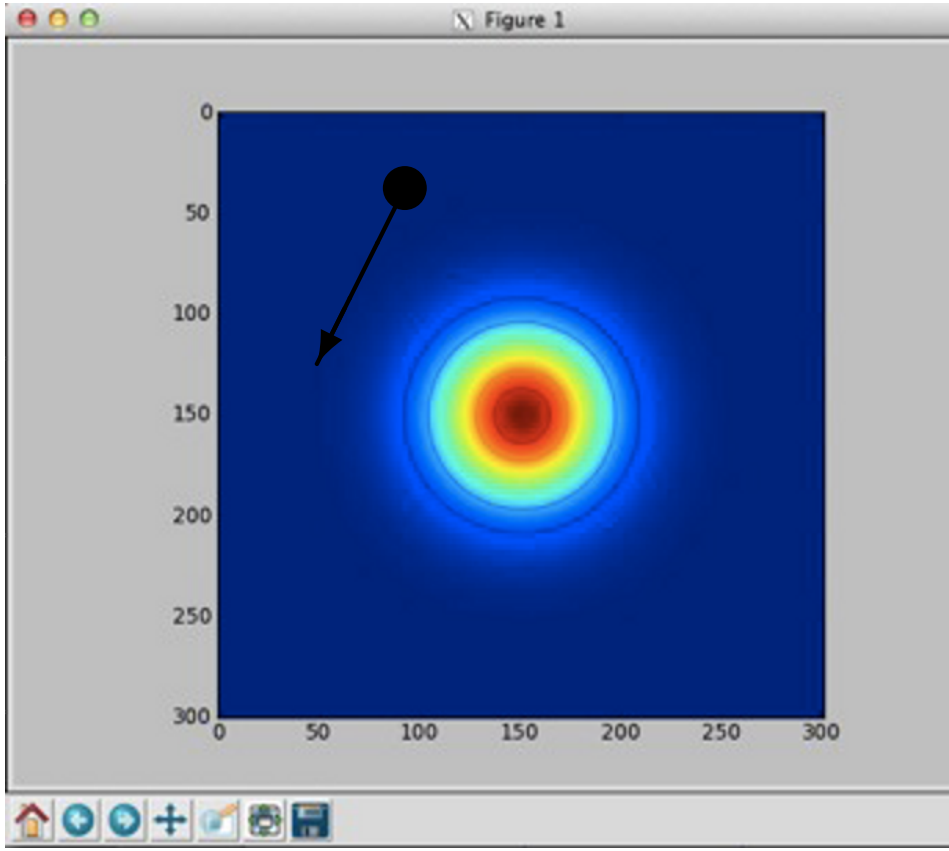
- Will be working on for next 3 weeks
- Charged tracer particle moving in a diffusive environment
- Colloidal transport in fluid medium
- Couple kinds of physics, couple kinds of data structures (grid, particle)



SciNet
compute • calcul
CANADA

Course Project

- Get source code: http://wiki.scinethpc.ca/wiki/images/f/f0/diffuse.cc
- Setup: Supervisor has this old code for diffusive background, "works fine," wants you to add tracer particle to it.
- Python script to plot diffusing field as a 'movie'. http://wiki.scinethpc.ca/wiki/images/f/f0/plotdata.py



Diffusion

- Program is supposed to be solving the PDE

$$\frac{\partial \rho}{\partial t} = D\left(\frac{\partial^2 \rho}{\partial x^2} + \frac{\partial^2 \rho}{\partial y^2}\right).$$

- Does so by discretizing space on a 2d grid and time on a 1d grid.

- Time stepping rather trivial:

$$\rho(t + \Delta t) = \rho(t) + \Delta t \times \text{RHS}.$$

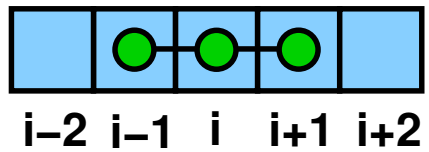
  (First order in t)

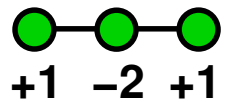
- RHS computed using discretized Laplacian.

Discretizing Derivatives

- Done by finite differencing the discretized values
- Implicitly or explicitly involves interpolating the data and taking the derivative of the interpolant.
- More accuracy — larger stencils.

$$\left.\frac{d^2\rho}{dx^2}\right|_i \approx \frac{\rho_{i+1} - 2\rho_i + \rho_{i-1}}{\Delta x^2}$$

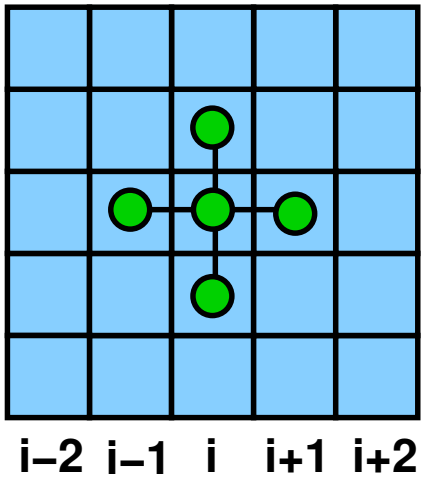


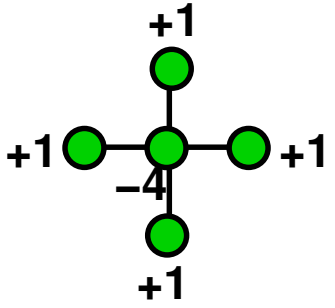


Discretizing Derivatives

- Done by finite differencing the discretized values
- Implicitly or explicitly involves interpolating the data and taking the derivative of the interpolant.
- More accuracy — larger stencils.

$$\left(\frac{\partial^2\rho}{\partial x^2}+\frac{\partial^2\rho}{\partial y^2}\right)_i \approx \frac{\rho_{i,j+1}+\rho_{i+1,j}-4\rho_{i,j}+\rho_{i-1,j}-\rho_{i,j-1}}{\Delta x^2}$$





Course Project

- Code isn't a disaster as these things go.
- Even has comments! That are still true!
- But one monolithic routine. Difficult to follow (even in this simple 193-line case)

```
//
// diffuse.cc
//
// Monolythic C++ version of 2d diffusion with output for graphics.
//
// Compile with e.g. "g++ diffuse.cc -Wall -O3 -o diffuse".
//

#include <iostream>
#include <fstream>
#include <cmath>
#include <string>
#include <algorithm>

// Simulation parameters
const float x1 = -12;
const float x2 = 12;
const float D = 1;
const int numPoints = 128;
const int numSteps = 4800;
const int plotEvery = 150;

// Parameters of the initial density
const float a0 = 0.5/M_PI;
const float sigma0 = 1;

// Parameter for the theoretical prediction
const int numImages = 2;

// Output files
const std::string dataFilename = "data.npy";
const std::string theoryFilename = "theory.npy";

// Header data for npy files
const int  headersize = 5*16;
const int  cheadersize = 61;
const char cheader[cheadersize+1]
= "\x93NUMPY\1\0\x46\0{'descr': '<f4', 'fortran_order': False, 'shape': (";
const std::string header( cheader, cheader+cheadersize );


int main( int argc, char *argv[] )
{
    // Data structures
    float *x;
    float ***rho;
    float time;
    float dt, dx;
    float error;
    float rhoint;
    int theory, before, active;

    // Compute derived parameters
    dx = (x2 - x1)/(numPoints - 1);
    dt = dx*dx*D/5;
```

Course Project

- You're almost always better off in these situations spending some time cleaning these things up some first
- For your own sanity
- But need to make sure your changes don't change answers
- So lets start setting up decent development environment, baseline

```
//
// diffuse.cc
//
// Monolythic C++ version of 2d diffusion with output for graphics.
//
// Compile with e.g. "g++ diffuse.cc -Wall -O3 -o diffuse".
//

#include <iostream>
#include <fstream>
#include <cmath>
#include <string>
#include <algorithm>

// Simulation parameters
const float x1 = -12;
const float x2 = 12;
const float D = 1;
const int numPoints = 128;
const int numSteps = 4800;
const int plotEvery = 150;

// Parameters of the initial density
const float a0 = 0.5/M_PI;
const float sigma0 = 1;

// Parameter for the theoretical prediction
const int numImages = 2;

// Output files
const std::string dataFilename = "data.npy";
const std::string theoryFilename = "theory.npy";

// Header data for npy files
const int  headersize = 5*16;
const int  cheadersize = 61;
const char cheader[cheadersize+1]
= "\x93NUMPY\1\0\x46\0{'descr': '<f4', 'fortran_order': False, 'shape': (";
const std::string header( cheader, cheader+cheadersize );


int main( int argc, char *argv[] )
{
    // Data structures
    float *x;
    float ***rho;
    float time;
    float dt, dx;
    float error;
    float rhoint;
    int theory, before, active;

    // Compute derived parameters
    dx = (x2 - x1)/(numPoints - 1);
    dt = dx*dx*D/5;
```

Homework 2

HW2

- Start a git repository for this project, and add the two files.
- Create a Makefile and add it to the repository.
- Since we have no tests, run the program and save its output as a baseline integrated test (add to repository). Then write a 'test' target in your makefile that:
  - Runs 'diffuse' with output to a new file.
  - Compares the file with the baseline test file.
    (hint: the unix command diff compares files).
- Move the global variables into the main routine.
- *Chorus: Test your modified code, and commit.*
- Extract a diffusion operator routine, that gets called from main.

  Chorus
- Create a .cc/.h module for the diffusion operator.

  Chorus

HW2

- Add two tests for the diffusion operator: for a constant and for a linear input field (`rho[i][j]=a*i+b*j`).
  Add these to the test target in the makefile.

  Chorus
- Extract three more .cc/.h modules:
  - for output (should not contain hardcoded filenames)
  - computation of the theory
  - and for the array allocation stuff.

  Chorus
- Describe, but don't implement in the .h and .cc, what would be appropriate unit tests for these.

  Email in your source code and the git log file of all your commits as a .zip or .tar file by email to rzon@scinethpc.ca and ljdursi@scinethpc.ca by next Thursday at 9:00 am.

SciNet compute • calcul CANADA