# Computational Economics and Econometrics

A. Ronald Gallant
Penn State University

# Course Website

http://www.aronaldg.org/courses/compecon

Go to website and discuss

- Preassignment
- Course plan (briefly now, in more detail a few slides later)
- Source code
- libscl
- Lectures
- Homework
- Projects

# Course Objective – Intro

Introduce modern methods of computation and numerical analysis to enable students to solve computationally intensive problems in economics, econometrics, and finance. The key concepts to be mastered are the following:

- The object oriented programming style.
- The use of standard data structures.
- Implementation and use of a matrix class.
- Implementation and use of numerical algorithms.
- Practical applications.
- Parallel processing.

# Details follow

# Object Oriented Programming

Object oriented programming is a style of programming developed to support modern computing projects. Much of the development was in the commercial sector. The features of interest to us are the following:

- The computer code closely resembles the way we think about a problem.
- The computer code is compartmentalized into objects that perform clearly specified tasks. Importantly, this allows one to work on one part of the code without having to remember how the other parts work: selective ignorance
- One can use inheritance and virtual functions both to describe the project design as interfaces to its objects and to permit polymorphism. Interfaces cause the compiler to enforce our design, relieving us of the chore. Polymorphism allows us to easily swap in and out objects so we can try different models, different algorithms, etc.
- The structures used to implement objects are much more flexible than the minimalist types of non-object oriented language structures such as subroutines, functions, and static common storage.

# Learning Object Oriented Programming

I believe that object oriented programming can only be learned by example. However, the examples must be similar to one's own projects for two reasons.

1. It is hard to sustain interest in other people's problems at the level of detail required to learn.

2. It is easier to transfer ideas, code, and designs between similar projects.

One case study will implement MCMC estimation of a habit economy. This example encompasses all essential techniques: matrix class, inheritance, polymorphism, vector class, associative maps, parallelization by MPI, etc. Another will estimate a dynamic game using particle filters with paralleliztion by threads.

# Data Structures

Data structures are standard ways of organizing information on a computer.

In a computer science course data structures are usually conceptualized at the level of detail of their implementation as trees, heaps, linked lists, etc. We are interested in them at a higher level of abstraction, primarily:

- The C++ vector class which allows one to store any type of object in a container that is indexed by integers. We will have a need to store GMM objective functions each containing moments from different data in a vector.
- The C++ associative map which allows one to store any type of object in a container that is indexed by any object that can be ordered. We will have need to store matrices in an associative map indexed by matrices.

# Matrix Class

The matrix class that we shall use is libary `libscl`, which is available by going to www.aronaldg.org and clicking on “Browse webfiles”.

You will learn how to compile this library for your own use.

You can see how such libraries are implemented from the source code. But be warned, there are many old fashioned C language tricks in it designed to combat poorly implemented compilers.

This library makes C++ programming as easy as Gauss, Matlab, etc., but your programs can be much more complex and will run much faster.

# Numerical Algorithms

These two references are the best places to look first:

- Press, William H., Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling (1993), Numerical Recipes in C, The Art of Scientific Computing, Second Edition, Cambridge University Press.

- Miranda, Maria J., and Paul L. Fackler (2002), Applied Computational Economics and Finance, MIT Press.

The actual code in Press at. al. is of low quality. You will usually have to rewrite it to get something usable. Search the web instead. But their descriptions of the numerical analysis ideas are concise and readable. Most other numerical analysis references are neither

# Numerical Algorithms

GNU has implemented a scientific library for use with C and C++ This is a freeware competitor to the proprietary NaG and IMSL libraries. See the web site

`http://www.gnu.org/software/gsl/manual/html_node`

for details.

There is an example of the use of a BFGS algorithm from the GSL library in the C++ implementation of SNP. The production version of SNP uses a BFGS algorithm from libscl.

# Numerical Algorithms

Boost is a repository for free peer-reviewed portable C++ source libraries. The emphasis is on libraries which work well with the C++ standard library. See

`http://www.boost.org`

for details.

Mostly augments C++ data structures but does have matrix classes, random number generators, etc. The random number generators are high quality.

# Practical Application

Our case studies will implement the ideas in these papers:

- Gallant, A. Ronald, and George Tauchen (2009), "SNP: A Program for Nonparametric Time Series Analysis, User's Guide," `http://www.aronaldg.org/webfiles/snp`
- Gallant, A. Ronald, and George Tauchen (2009), "EMM: A Program for Efficient Method of Moments Estimation, User's Guide," `http://www.aronaldg.org/webfiles/emm`
- Chernozhukov, Victor, and Han Hong (2003), "An MCMC Approach to Classical Estimation," *Journal of Econometrics* 115, 293–346.
- Aldrich, Eric M., and A. Ronald Gallant (2010), "Habit, Long Run Risks, Prospect? A Statistical Inquiry," *Journal of Financial Econometrics*, 9, 589–618. `http://www.aronaldg.org/webfiles/papers/tm.pdf`
- Gallant, A. Ronald, Han Hong, and Ahmed Khwaja (2008), "Bayesian Estimation of a Dynamic Game with Endogenous, Partially Observed, Serially Correlated State," `http://www.aronaldg.org/webfiles/papers/socc.pdf`
- Vertenstein, Mariana, Tony Craig, Adrianne Middleton, Diane Feddema, and Chris Fischer (2010) "CESM1.0 User's Guide," `http://www.cesm.ucar.edu/models/cesm1.0/cesm/cesm_doc/book1.html`

# Parallel Processing – Clusters

The Message Passing Interface (MPI) is a standardized parallelization technique and the most widely used. The following are the best general and specific references, respectively:

- Foster, Ian, Designing and Building Parallel Programs (1995), Addison-Wesley, New York. ISBN 0-0201-57594-9. Online at http://www.mcs.anl.gov/dbpp.

- Pacheco, Peter S., A User's Guide to MPI (1995), Manuscript, Department of Mathematics, University of San Francisco. Online at course website or from math.usfca.edu in directory pub/MPI.

# Parallel Processing – Multiple CPU Machines

Although the Message Passing Interface (MPI) can be used on multiple CPU machines (cores counts as CPUs) pthreads provide more flexibility. The following is the best general reference:

- Blaise Barney, POSIX Threads Programming, Livermore Computing. http://www.llnl.gov/computing/tutorials/pthreads
- Additional references at the end of the article.

# Parallel Processing – Multiple CPU Machines

Rather than threads, on a multiple CPU machine, one can use OpenMP. Similar to threads but less disruptive to code logic.

A common strategy on super-computers, which are clusters of multiple CPU machines, is to use OpenMP on each node and MPI across nodes.

We will examine the code of a climate model that uses this strategy to run on an IBM Power 575: 128 nodes, 32 CPUs per node, each CPU is 4.7GHz, 4096 CPUs in total, weight 33,000 lbs., not counting the circulating water cooling equipment.

The following is a good reference:

- OpenMP Specifications
  http://www.openMP.org

# Parallel Processing – Video Cards

Video cards are massively parallel devices, 300+ GPU's are common. They are very cheap, e.g. $600 for 240 GPU's. They can be installed in a desktop machine; your machine, e.g. a Mac, may already have one installed. They are easy to program using an NVIDIA C compiler that has extensions for parallelization. Can mix host CPU and GPU instructions. The bad news is that copying from host memory to device memory is slow. The following are the best references:

- http://www.nvidia.com/object/cuda_home.html
  - ▷ CUDA_Programming_Guide_2.2.1.pdf
  - ▷ CUDA_Reference_Manual_2.2.pdf
  - ▷ CUBLAS_Library_2.1.pdf

# Parallel Processing – Video Cards

Rather than CUDA on a video card one can use OpenCL. OpenCL has broader applicability. It can be used with other vendor's cards, e.g. AMD, and can be used on a Mac. CUDA quit working on a Max with the Snow Leopard OS. OpenCL comes with Snow Leopard and later releases of Mac OS. OpenCL can be used to drive any supported compute unit on a host, including the CPU. The following are the best references:

- http://www.khronos.org/opencl/registry/cl
  - ▷ OpenCL_1.2_Specification.pdf
  - ▷ OpenCL_1.1_C++_Bindings_Specification.pdf

- Scarpino, Matthew, (2011) *OpenCL in Action*, Manning Publications, Shelter Island, NY. (`http://www.manning.com`)

# Course Objective – Summary

Introduce modern methods of computation and numerical analysis to enable students to solve computationally intensive problems in economics, econometrics, and finance. The key concepts to be mastered are the following:

- The object oriented programming style.
- The use of standard data structures.
- Implementation and use of a matrix class.
- Implementation and use of numerical algorithms.
- Practical applications.
- Parallel processing.

Go to course website and go through Course Plan in detail

# Downloading Source Code

Go to the course web site and click on “Source Code”

To view a text file such as ch00/makefile, left click on ch00. and then left click on makefile. To download the file, right click on makefile. and select ”Download Linked File” (Safari) or ”Save Link As...” (Firefox).

# Computing

- If you use a Microsoft Windows machine, install Cygwin from http://www.cygwin.com
  - ▷ An alternative is MinGW (the Minimalist GNU for Windows) from http://www.mingw.org
- If you use a Mac, install Xcode from the second of the two CDs that came with the machine or, for new machines, from https://developer.apple.com/xcode
- Later in the course you will be given an account on a machine with 16 cores for assignments involving parallel computing.

# Course Performance Evaluation

- Homework 50%.
  Homework assignments are posted on the website.

- Midterm 20%.
  Covers the C++ language. Many questions are similar to homework.

- Project 30%.
  Topic chosen by the student with instructor's approval. Suggestions are posted on the website.

# Running Code Locally

- Cygwin: Follow the documentation.

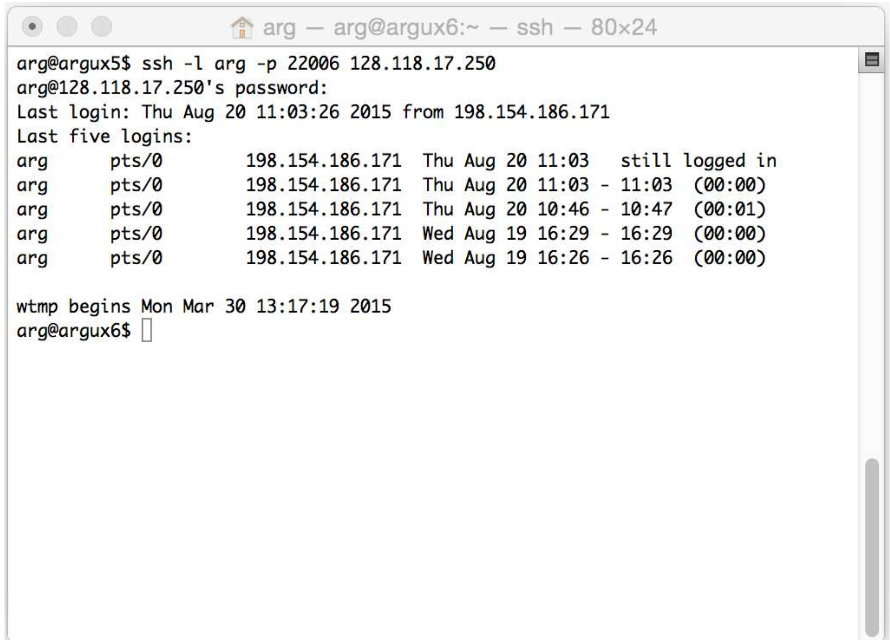
- MinGW: Follow the documentation.

  ▷ Or see the excellent Appendix C of Scarpino, Matthew, (2011) *OpenCL in Action*, Manning Publications, Shelter Island, NY. (`http://www.manning.com`)

- Mac: Use terminal.

- Thereafter, except for the next slide, everything looks much the same as what follows.

# Logging on to a Remote Machine




Above, instead of username `arg`, use your username, e.g.,

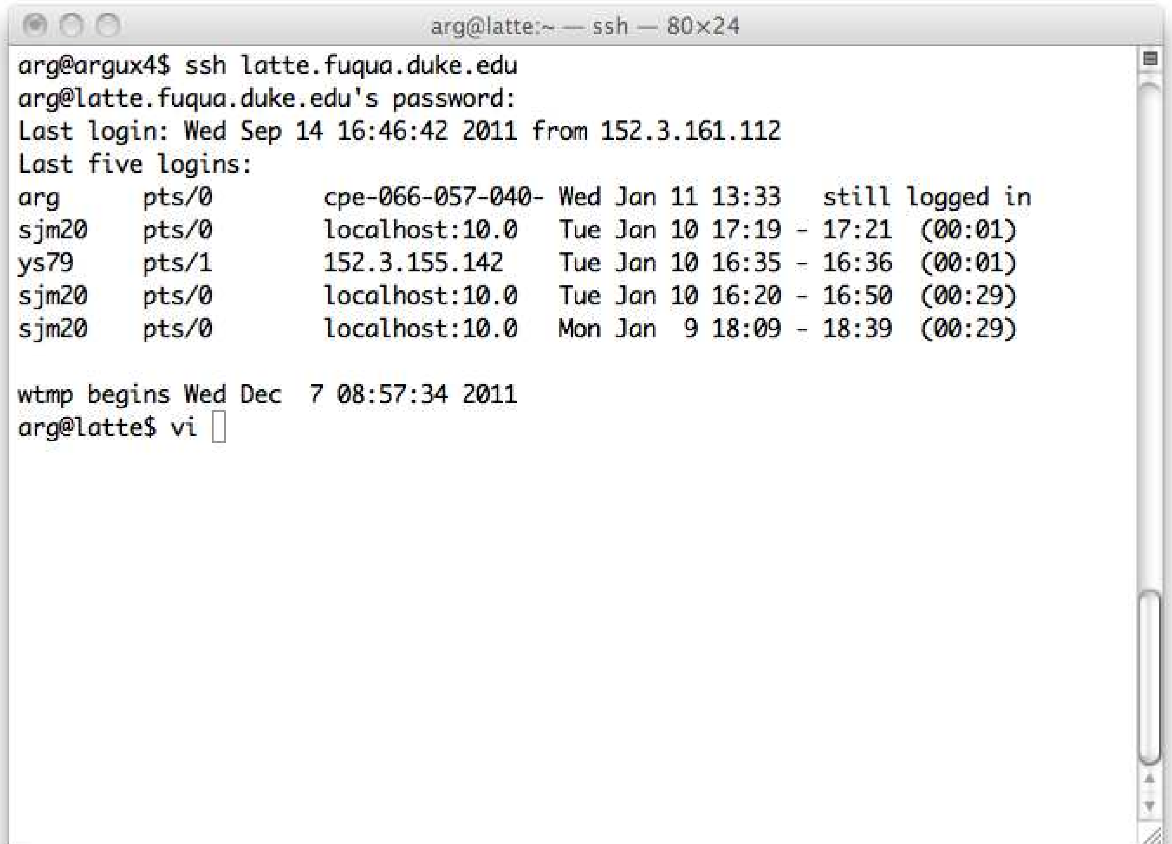
**ssh -l xyz125 -p 22006 128.118.17.250**

On a PC use F-Secure SSH Client or Tera Term SSH or something similar. On a Mac use Terminal, as shown, found in /Applications/Utilities.

# Editors

- I use vi, which is available on any Unix machine.
  - ▷ A Mac is a Unix machine.
  - ▷ If `vi filename` doesn't work then `vim filename` will.
  - ▷ If both work, use `vim` rather than `vi`
  - ▷ See `http://vimdoc.sourceforge.net` for documentation.
- Any editor that will write ASCII text files will work.
- The editor with the least learning cost is nano, which is on the Mac and which can be downloaded for a PC and many flavors of Unix at `http://www.nano-editor.org`

# Built-In Documentation for vi

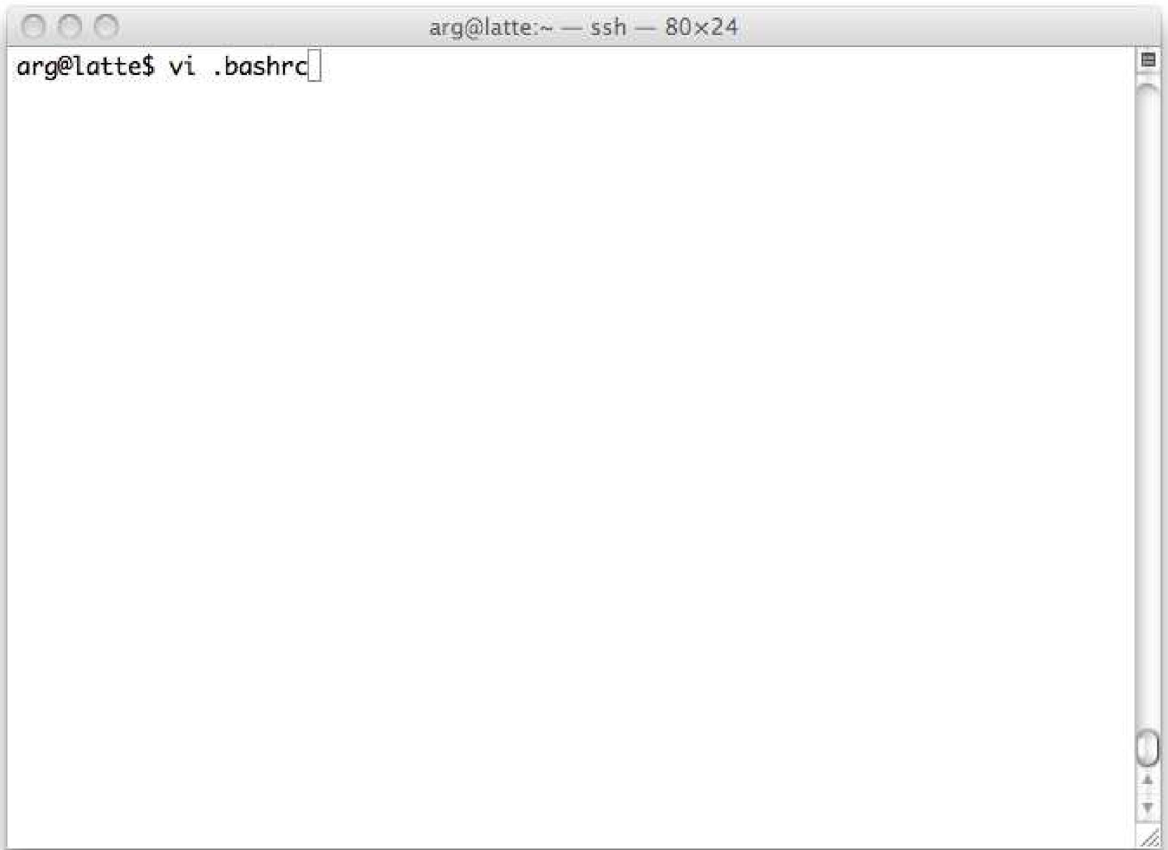



Also see page 85 of Sams Teach Yourself Unix in 10 Minutes. Most important: **i** to edit text, **Esc** to stop editing, **:w** to save, **:q** to quit.

# Built-In Documentation for vi


Warning: Backspace key may not work. Use Ctrl-H if this happens. Can be fixed by edits to .vimrc. I'll discuss if this happens and drives you nuts.

# Customizing Your Environment




Note the dot: vi .bashrc. The leading dot makes the file invisible. To see all files type 'ls '-lag. It is possible that you are using .tcshrc or .cshrc; in which case the ideas here and on the next slide are similar.

# Customizing Your Environment


```
# .bashrc

if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

PS1='\u@\h\$ '

alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
alias dir='ls -la'
alias copy='cp -i'
alias del='rm -i'
alias delete='rm -i'
alias erase='rm -i'
alias home='cd ~/'
alias list='cat'
alias md='mkdir'
alias move='mv -i'
alias rd='rmdir'
alias rename='mv -i'
~
".bashrc" 22L, 341C                                           22,1          All
```

NEVER EVER CHANGE THE FIRST LINES OF THIS FILE. Be extremely careful editing it. Best is to edit a copy called .bashrc.new, test it with source .bashrc.new, and if it works install it with cp .bashrc.new .bashrc.

# Chapter 0. Getting started

First main point:

- Running the ”Hello World” program.

# Running prog01 under Linux

- Assume code is in this directory:
  $HOME/compecon/src/ch00
  — equivalently —
  ~/compecon/src/ch00

- Which has these files within it:
  makefile
  prog01.cpp

- Open a window

- Enter these commands within it:
  cd ~/compecon/src/ch00
  make
  prog01

# Files within ~/compecon/src/ch00

## makefile

```
CXX      = g++
SDIR     = .
CXXFLAGS = -O2 -Wall -c
LDFLAGS  = -lm

prog01 : prog01.o
	$(CXX) -o prog01 prog01.o $(LDFLAGS)

prog01.o : $(SDIR)/prog01.cpp
	$(CXX) $(CXXFLAGS) $(SDIR)/prog01.cpp

clean :
	rm -f *.o
	rm -f core core.*

veryclean :
	rm -f *.o
	rm -f core core.*
	rm -f prog01
```

Warning: The indentation is with tabs, not blanks.

The variable names `CXX`, `CXXFLAGS`, `LDFLAGS` are GNU's conventions for C++; most people have gotten into the habit of using them.

# GNU's Makefile Variable Name Conventions

| | | |
|---|---|---|
| CC | C compiler | e.g. CC = gcc |
| CFLAGS | C compiler flags | e.g. CFLAGS = -O2 -c |
| FC | Fortran 77 compiler | e.g. FC = g77 |
| FFLAGS | Fortran 77 compiler flags | e.g. FFLAGS = -O2 -c |
| CXX | C++ compiler | e.g. CXX = g++ |
| CXXFLAGS | C++ compiler flags | often use CFLAGS instead |
| F90 | Fortran 90 compiler | e.g. F90 = gfortran |
| F90FLAGS | Fortran 90 compiler flags | usually use FFLAGS instead |
| LDFLAGS | Loader flags | e.g. LDFLAGS = -lm |

This list is not exhaustive.

Sometimes these variable names are put into the environment and the definitions omitted from the makefile.

When not in a mixed compiler environment, many just use CC, CFLAGS, FC, FFLAGS, and LDFLAGS for whatever flavors of C, C++, or Fortran are being used.

# Files within ~/compecon/src/ch00

## prog01

```
#include <iostream>

int main()
{

  std::cout << "Hello, world" << '\n';

  std::cout << "Goodbye\n";

  return 0;

}
```

# Some Makefile Variables

| | |
|---|---|
| `$@` | The name of the rule's target |
| `$^` | The dependency names, separated by spaces |
| `$<` | The first dependency |
| `$?` | Dependencies more current than the target |

This list is not exhaustive.

In addition, one often adds the lines

```
.PHONY: clean
.PHONY: veryclean
```

to tell `make` that `clean` and `veryclean` are not filenames.

# Files within ~/compecon/src/ch00

## makefile.alt

```
CXX      = g++
SDIR     = .
CXXFLAGS = -O -Wall -c
LDFLAGS  = -lm

prog01 : prog01.o
	$(CXX) -o $@ $^ $(LDFLAGS)

prog01.o : $(SDIR)/prog01.cpp
	$(CXX) $(CXXFLAGS) $^

.PHONY: clean
.PHONY: veryclean

clean :
	rm -f *.o
	rm -f core core.*

veryclean :
	rm -f *.o
	rm -f core core.*
	rm -f prog01
```

Warning: The indentation is with tabs, not blanks.

# Class Demo

Run prog01 on laptop and course machine

Put CXX, CXXFLAGS, LDFLAGS in the enviornment, delete from makefile, rerun.

# Chapter 0. Getting started

Remaining main points:

- comments: either //... to end of line or /* ... */ free form
- include: #include <iostream>
- main: int main() { ...
- return: ... return 0; }
- expression: tmp = exp(15) + 77;
- left associative, right associative
- scope

# Chapter 0 Main Points

Illustrate main points with prog02.

# Chapter 1. Working with strings

Main points:

- variable, object, type
- definition, scope, local variable
- interface, which consists of constructors (string(), string(n,'c'), string("text")), operators ([ ], +, <<, =), member functions (size, c_str), etc.
- overloading, e.g. string(), string(n,'c'), string("text")
- const
- cin, cout

# Variable, Object, Type

- Type defines both a data structure and the collection of operations that can be performed on it. For instance, a string is a type and some operations that are permitted are `str1+str2`, `cout<<str`, `str.c_str()`. An operation that is forbidden is `string('a')`. But `string("a")` is permitted.

- An object is part of a computer's memory that has a type. For instance, the statement `cout<<string("How now brown cow.");` will create a string object somewhere in memory but it will not have a name (that is available to us).

- A variable is an object that has a name. For instance, the definition `string cow = string("How now brown cow.");` gives the object above the name `cow`.

# Definition, Declaration

- A declaration associates a name to a type. For instance, `string str;`. A name must be declared before it can be used.
- A definition provides enough information that the compiler can allocate space. A statement can be both a declaration and a definition. For simple types, most declarations are also definitions. In Chapter 3 examples will make the distinction clear.
- Only one definition is allowed. There can be numerous declarations. Header files such as `<string>` are usually collections of declarations. The corresponding definitions are usually in source code that has been compiled and placed in libraries.
- The `extern` specification forces a statement to be a declaration and not a definition: `extern string str;`.

# const

Defining a variable to be `const` is a promise to the compiler never to change it.

A `const` variable can only be defined, not declared then defined, because first declaring then defining would violate the promise not to change the variable.

# cin, cout

```
std::string s1 = "... something ... ";
std::cout << s1;   // writes the entire string including blanks

std::string s2, s3;
std::cin >> s2;   // reads one word from an input line
getline(cin,s3);  // reads an entire input line
```

## prog01

```
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char** argp, char** envp)  // See Section 10.4
{
  // Here are three equivalant ways to define a std::string from
  // a string literal according to the language standard.

  string str01("The quick brown fox jumped over the lazy dogs");

  string str02 = "The quick brown fox jumped over the lazy dogs";

  string str03 = string("The quick brown fox jumped over the lazy dogs");

  // The same constructior is called in each instance. The second
  // and third statements are idiomatic because the assignment
  // operator is not called.

  cout << str01 << '\n';      // This does not flush cout and is
                              // therefore more efficient than
  cout << str02 << endl;      // this line, which flushes cout.

  cout << str03 << '\n';

  cout.flush();               // Explicitly flushes cout.
```

## prog01 (continued)

```
// The following are roughly equivalent to each other but not to the
// above.  Here the strings are first initialized as null strings by
// the default constructor and then the assignment operator destroys
// the string and replaces it with another.

{ // Scope: Because of these braces ..

  string str04, str05, str06;

  str04 = str01;

  str05 = string("The quick brown fox jumped over the lazy dogs");

  str06 = "The quick brown fox jumped over the lazy dogs";

  cout << str04 << '\n' << str05 << '\n' << str06 << '\n';

} // ... the variables str04, str05, str06 have been deleted
  // and can no longer be used without re-definition.

string str06;    // str06 is re-defined here

// Strings can be added

str06 = str03 + string("\n") + string("How now brown cow?") + "\n";

cout << str06;
```

## prog01 (continued)

```
  //A string of given length can be constructed with a fill character.

  string str07(40,'*');

  // Elements can be changed, but BE WARNED, the first element is
  // str07[0] and the last is str08[39].  An out of range access
  // will cause the program to abort.

  str07[0] = 'H'; str07[1] = 'o'; str07[2] = 'w';
  str07[4] = 'a'; str07[5] = 'r'; str07[6] = 'e';
  str07[8] = 'y'; str07[9] = 'o'; str07[10] = 'u'; str07[11] = '?';

  cout << str07 << '\n';

  return 0;
}
```

## Class Demo

Run prog01.

## Warning: Forward Reference

The discussion and examples for Chapter 2 will also use material from Sections 10.1 through 10.4 of the book.

# Chapter 2. Looping and counting

Main points:

- control statements: while, for, do...while, switch, if...else
- boolean, true and false
- operators: logical and arithmetic
- precedence, parenthesis

# Built in types

There are many built-in types. They come in five groups:

1. Boolean: bool (0 is false, anything else is true)

2. Character types: char, signed char, unsigned char, wchar_t

3. Integer types: comes in three sizes – short int, int, long int – and three forms — signed int, int, unsigned int.

4. Floating-point types: float, double, and long double.

5. No type information available: void.

# Built in types

One can abbreviate short int by short and long int by long. A short, int, or long is signed, a char may or may not be signed.

For numerical work on an Intel box, you will use only bool, char, int, double, and void. On some machines, e.g. DEC, int and double are not big enough for numerical work and one needs to use long int and long double instead.

What we shall do a bit later to get portable code is use a typedef to make REAL and INTEGER types that are synonyms for whatever is correct for the machine. They are defined once and for all in a header.

# Operators

The types can be related by a bewildering number of operators with elaborate rules of precedence listed on page 32. Many you will never ever use and you will probably never be able to trust your memory of precedence.

Don't worry about precedence, arithmetic operators obey the rules of algebra i.e. a*x+y*z means (a*x)+(y*z). For anything else, use parenthesis to make sure the machine does what you want. E.g., for

```
(r == 0 || r == rows - 1 || c == 0 || c == cols - 1)
```

on page 23 write

```
( (r == 0) || (r == (rows-1)) || (c == 0) || (c == (cols-1)) )
```

and there will be no doubt that the machine will do what you want it to do.

## Operators

Always use

| | | |
|---|---|---|
| a += b | instead of | a = a + b |
| a -= b | instead of | a = a - b |
| a *= b | instead of | a = a*b |
| a /= b | instead of | a = a/b |

because they execute much faster.

# Operators

When this executes

```
int a;
int b = 1;
a = ++b;
```

both a and b will equal 2. When this executes

```
int a;
int b = 1;
a = b++;
```

a will equal 1 and b will equal 2.

Stated differently: ++b returns b+1 and b++ returns b.

# Operators

What the following will do is unpredictable

```
f(n++,n);
```

because the order in which arguments are evaluated by a function is not specified by the language standard and there is variation both between and within implmentations.

What will happen is either equivalent to

```
f(n,n);
++n;
```

or to

```
f(n,n+1);
++n;
```

depending on which argument gets evaluated first.

Similarly for `f(++n,n)`.

## if statement

```
if (a < b) {
  //do something
}

if (a < b) {
  //do something
}
else {
  //do something else
}
```

NEVER EVER, EVER LEAVE OFF THESE BRACES despite what the book says.

# while statement

```
i = 0;     // initialize
while (i < 10) {
  //do something
  ++i;     // increment
}
```

The variable i has the value 10 at the end of the loop

NEVER EVER, EVER LEAVE OFF THESE BRACES despite what the book says.

# do ... while statement

```
i = 0;     // initialize
do {
  //do something
  ++i;     // increment
} while (i < 10);
```

The variable i has the value 10 at the end of the loop. Regardless of how i is initialized, the statements in braces will be executed at least once.

NEVER EVER, EVER LEAVE OFF THESE BRACES despite what the book says.

# for statement

```
for (int i=0; i<10; ++i) {
  //do something
}
```

The variable i is not available end of the loop.

```
int i;
for (i=0; i<10; ++i) {
  //do something
}
```

The variable i is available end of the loop.

NEVER EVER, EVER LEAVE OFF THESE BRACES despite what the book says.

# switch statement

```
char c = 0;
cin >> c;
switch (c) {
  case 'a':
    // do something
    break;
  case 'b':
    // do something
    break;
  // etc
  default :
    // do something
    break;
}
```

# switch statement

```
int i = 0;
cin >> i;
switch (i) {
  case 1 :
    // do something
    break;
  case 2 :
    // do something
    break;
  // etc
  default :
    // do something
    break;
}
```

# Brief Introduction to Arrays and Pointers

See Sections 10.1, 10.2, and 10.3.

```
double a[3];  //array of three doubles a[0], a[1], a[2]

double* aptr; //a pointer to a double

aptr = &a[0]; //& means take the address of a[0]
              //aptr now points to a[0]

cout << *aptr;  //* means dereference aptr;
                //the value of a[0] gets printed

aptr++;         //aptr now points to a[1]

char* cptr = "Hello world.";

cout << *cptr++; // H gets printed, *cptr is now e.
```

# Counting

Computer scientists count from zero. Container classes (strings, vectors, maps), arrays of built-in types, etc. are indexed from zero:

```
double x[n];
for (int i=0; i<n; ++i) {
  x[i]=something;
}
double* xi = &x[0];
double* top = xi + n;
while (xi < top) {
  *xi++ = something;
}
char* str = "The quick brown fox jumped.";
char copy[256];
char* s = str;   char* c = &copy[0];
while (*c++=*s++);
```

Economists, numerical analysts, statisticians, etc. count from one. One just has to get used to being careful:

```
realmat y(n,1);
for (int i=1; i<=n; ++i) {
  y[i]=x[i-1];
}
```

# Invariants

An invariant is basically the state of an object with state meaning much the same thing as its meaning in economics.

Computer scientists attach a lot of importance to invariants and to making sure that the invariants are correct; i.e. that the state of the object is good.

Real world programmers understand the concept but usually do not do a lot of invariant creation and checking in production code because it slows execution too much.

However, they often have such code in the program controlled by a compiler directive to check the state while debugging code:

```
#define DEBUG
// ...
#if defined DEBUG
   // code to check state
#endif
// ...
#undef DEBUG
```

# More on Arrays and Pointers

```
double b[6] = {0.0,1.0,2.0,3.0,4.0,5.0}; // initialization of an array
double* bptr = &b[0];                    // pointer to first element

*(bptr+2) = 1.0; bptr[2] = 1.0; // equivalent, all change 2.0 to 1.0
*(b+2) = 1.0; b[2] = 1.0;       // remember, counting starts at 0

char  d[13] = {'H','e','l','l','o',' ','W','o','r','l','d','.','\0'};
char* dptr = &d[0];

*(dptr+11) = '!';  d[11] = '!';  // both legal, both change '.' to '!'

char* cptr = "Hello world.";     // cptr points to a string literal
                                 // of 13 chars, a trailing '\0'
                                 // gets stuck in automatically

*(cptr+11) = '!'; // illegal, program will abort with a segmentation
                  // fault because a string literal is stored in
                  // static memory and cannot be changed.

// Moral: It is much safer to use the C++ string class than to fool
// with C style strings.
```

## prog01

```
#include <iostream>
using std::cout;

int main()
{
  double b[6] = {0.0,1.0,2.0,3.0,4.0,5.0};   // b is actually
                                              // a pointer
  double* bptr = &b[0];

  for (int i=0; i<6; ++i) cout << b[i] << "  "; cout << '\n';

  *(bptr+2) = 1.0;
  for (int i=0; i<6; ++i) cout << b[i] << "  "; cout << '\n';

  bptr[2] = 2.0;
  for (int i=0; i<6; ++i) cout << b[i] << "  "; cout << '\n';

  *(b+2) = 3.0;
  for (int i=0; i<6; ++i) cout << b[i] << "  "; cout << '\n';

  b[2] = 4.0;
  for (int i=0; i<6; ++i) cout << b[i] << "  "; cout << '\n';

  return 0;
}
```

## prog02

```
#include <iostream>
#include <fstream>
#include <string>
using std::ofstream;
using std::string;
using namespace std;

int main(int argc, char** argp, char** envp)  // See Section 10.4
{
  char* cstr = "This is a C-style string.";
  cout << cstr << '\n';

  string sstr(cstr);                  // How to convert a C-string
  cout << sstr << '\n';               // to a C++ string

  const char* copy = sstr.c_str(); // How to convert a C++ string
  cout << copy << '\n';               // to a C-string

  ofstream fout("arg.txt"); // How to open a file for writing

  if (!fout) {              // How to error check an fstream
    cerr << "File open failed\n";
    return 1;
  }
```

## prog02 (continued)

```
char** ptr;     // Pointer to a pointer to a C-style string
char*  str;     // Pointer to a char.

        // Remark: A C-style string is an array of char whose
        // last element is '\0'.  Most functions that have a
        // a pointer to char as an argument assume that it
        // points to the first element of a C-style string.

ptr = argp;     // Don't want to change argp so assign to ptr

for (int i=0; i<argc; ++i) {
  str = *ptr++;             // Derefernce, assign, increment
  fout << str << '\n';      // Print str
}
             // Remark: When operator<<'s argument is
             // a pointer to char it is assumed to be
             // pointer to a C-style string, which is
             // a '\0' terminated array of char.  The
             // array is printed, not the pointer value.
```

# prog02(continued)

```
ptr = argp;

for (int i=0; i<argc; ++i) {
  str = *ptr++;           // Dereference ptr, assign to str, increment
  while(*str) {           // C-strings terminated by '\0', i.e. false
    fout << *str++;       // Print one character, increment
  }
  fout << '\n';
}

for (int i=0; i<argc; ++i) {      // Equivalent, [] dereferences
  fout << argp[i] << '\n';        // pointers
}

for (int i=0; i<argc; ++i) {     // Equivalent, [][] dereferences
  int j=0;                       // pointers
  while(argp[i][j]) fout << argp[i][j++];
  fout << '\n';
}
```

## prog02(continued)

```
  ofstream envout("env.txt");
   ptr = envp;

  while(*ptr) {        // Last pointer in envp is null, i.e., 0
    envout << *ptr++ << '\n';
  }

  return 0;
}
```

# prog03 A better way

```
#include "libscl.h"

using namespace std;
using namespace scl;

int main(int argc, char** argp, char** envp)
{
 // A better way, vectors are discussed in Chapter 3.

  vector<string> arguments;

  char** ptr = argp;
  char** top = argp + argc;
  while (ptr < top) arguments.push_back(*ptr++);

  vector<string> environment;

  ptr = envp;
  while (*ptr) environment.push_back(*ptr++);
```

## prog03 (continued)

```
  ofstream fout;
  fout.open("arg.txt");
  if (!fout) error("Error, cannot open fout");

  vector<string>::size_type i = 0;
  while (i != arguments.size()) fout << arguments[i++] << ' ';
  fout << '\n';

  fout.close(); fout.clear();

  fout.open("env.txt");
  if (!fout) error("Error, cannot open fout");

  i = 0;
  while (i != environment.size()) fout << environment[i++] << '\n';

  fout.close(); fout.clear();

  return 0;
}
```

## Class Demo

Run prog01 and prog03.

# Arrays and Pointers

The use of C-style arrays and strings is confusing and error prone.

The use of C-style arrays and strings can be avoided in C++ and should be.

C-style strings are avoided by using the string class from the STL, as we have seen.

C-style arrays can be avoided using a matrix class. This is our next topic.

# Installation and use of a library

We will install libscl, which is a statistical computing library built upon a matrix class.

The matrix class is very similar to the string class in concept and usage.

It will give you the ease of doing matrix algebra in C++ that you have in Gauss, Matlab, Splus, R, etc.

But it will run much faster!

# Installing libscl

- Go to the course website

  www.aronaldg.org/courses/compecon

- Click on libscl

- Download file `libscl.tar`

- Assume for the remaining slides that the file `libscl.tar` is downloaded to `$HOME`, which is your home directory.

# Installing libscl

```
arg@argux0$ cd $HOME
arg@argux0$ mkdir lib
arg@argux0$ cd lib
arg@argux0$ tar -xf ../libscl.tar
arg@argux0$ ls -l
total 4
drwxrwxr-x 7 arg arg 4096 Dec 29 11:11 libscl
arg@argux0$ cd libscl
arg@argux0$ ls -l
total 32
drwxrwxr-x 2 arg arg  4096 Dec 29 11:11 cblas
-rw-rw-r-- 1 arg arg  1083 Dec 29 11:11 copyrite
drwxrwxr-x 2 arg arg  4096 Dec 29 11:11 gpp
drwxrwxr-x 2 arg arg  4096 Dec 29 11:11 ms
drwxrwxr-x 2 arg arg  4096 Dec 29 11:11 src
drwxrwxr-x 2 arg arg 12288 Dec 29 11:11 test
arg@argux0$ cd gpp
arg@argux0$ make
arg@argux0$ cd ../cblas
arg@argux0$ make
```

# Using libscl

- Look at makefiles.

- Hello world – painless version of Chapter 2's starbox.

- Regression – easy as Matlab or Gauss.

- Data construction – illustrates random number generation.

- Detailed study of headers that define the library – scltypes.h, sclerror.h, intvec.h, realmat.h, libscl.h.

## makefile

```
CXX      = g++
SDIR     = .
ISCL     = $(HOME)/lib/libscl/gpp
LSCL     = $(HOME)/lib/libscl/gpp
CXXFLAGS = -O2 -Wall -c -I$(SDIR) -I$(ISCL)
LDFLAGS  = -lm -L$(LSCL) -lscl

PROGRAMS = data hello regr

all: $(PROGRAMS)

data : data.o
	$(CXX) -o data data.o $(LDFLAGS)

data.o : $(SDIR)/data.cpp
	$(CXX) $(CXXFLAGS) $(SDIR)/data.cpp

hello : hello.o
	$(CXX) -o hello hello.o $(LDFLAGS)

hello.o : $(SDIR)/hello.cpp
	$(CXX) $(CXXFLAGS) $(SDIR)/hello.cpp
```

# makefile (continued)

```
regr : regr.o
	$(CXX) -o regr regr.o $(LDFLAGS)

regr.o : $(SDIR)/regr.cpp
	$(CXX) $(CXXFLAGS) $(SDIR)/regr.cpp

clean :
	rm -f *.o
	rm -f core core.*

veryclean :
	rm -f *.o
	rm -f core core.*
	rm -f $(PROGRAMS)
```

# hello.cpp

```
#include "scltypes.h"
#include "sclerror.h"
#include "sclfuncs.h"
#include "intvec.h"
#include "realmat.h"
#include "kronprd.h"    // libscl.h is the only one actually needed
#include "libscl.h"     // because each includes its predecessor.

using namespace scl;

using std::cout;
using std::string;

int main(int argc, char** argp, char** envp)
{
  cout << starbox("/Hello world//");

  cout << starbox("/How now /brown cow//");

  string msg("\nSecond argument changes delimiter\n");
  msg += string("'/' is no longer a delimiter\n\n");

  cout << starbox(msg.c_str(),'\n');

  return 0;
}
```

## hello > hello.out

```
**********************************************************************
*                                                                    *
*                            Hello world                             *
*                                                                    *
**********************************************************************


**********************************************************************
*                                                                    *
*                              How now                               *
*                             brown cow                              *
*                                                                    *
**********************************************************************


**********************************************************************
*                                                                    *
*                 Second argument changes delimiter                  *
*                    '/' is no longer a delimiter                    *
*                                                                    *
**********************************************************************
```

## regr.cpp

```
#include "libscl.h"
using namespace scl;
using std::cout;

int main(int argc, char** argp, char** envp)
{
  realmat data;
  vecread("regr.dat",data);

  realmat y = data("",1);
  realmat X = data("",seq(2,data.ncol()));

  realmat b = inv(T(X)*X)*T(X)*y;

  realmat sse = T(y - X*b)*(y - X*b);

  realmat V = sse[1]*inv(T(X)*X)/(y.nrow()-X.ncol());

  cout << starbox("/Estimate of b and its variance//");

  cout << b << V;

  return 0;
}
```

regr > regr.out

```
*********************************************************************
*                                                                   *
*                   Estimate of b and its variance                  *
*                                                                   *
*********************************************************************


                                    Col   1

                         Row   1      0.72910
                         Row   2      1.50053
                         Row   3      0.97680


                         Col   1        Col   2        Col   3

             Row   1  0.00875663   -0.013017 -0.00046027
             Row   2   -0.013017    0.025357 -0.00014200
             Row   3 -0.00046027 -0.00014200  0.00199555
```

## data.cpp

```
#include "libscl.h"
using namespace scl;

int main(int argc, char** argp, char** envp)
{
  INTEGER n=100;
  INTEGER p=4;
  INT_32BIT seed = 100542;

  realmat data(n,p);

  for (INTEGER i=1; i<=n; ++i) {
    data(i,2) = 1.0;
    data(i,3) = ran(seed);
    data(i,4) = unsk(seed);
    data(i,1) = data(i,2)+data(i,3)+data(i,4)+0.5*unsk(seed);
  }

  vecwrite("regr.dat",data);

  return 0;
}
```

# Headers

- scltypes.h – contains typedefs and includes headers used frequently.
  - ▷ Go to http://www.cplusplus.com/reference/cfloat before looking at scltypes.h
- sclerror.h – the error handler.
- sclfuncs.h – routines and classes that don't use class realmat.
- intvec.h – integer vectors used as a helper class for realmat.
- realmat.h – declares the matrix class realmat.
- kronprd.h – declares the matrix class kronprd.
- libscl.h – routines and classes that use class realmat.

# Extensive libscl example

Discuss headers.

Discuss tstrm.cpp.

# The BLAS

The blas, or more relevant to us the cblas, is a set of functions for matrix algebra that are optimized to be as fast as possible for a given machine.

There are two main sources of the blas: the manufacturer of the CPU, and ATLAS (Automatrically Tuned Linear Algebra Software), http://math.atlas.sourceforge.net. The one routine that provides a huge performance improvement is matrix multiply: cblas_dgemm.

The performance advantage can be 100 to 1 for large matrices.

`libsclcb` uses the `cblas`. `libsclcb` is included with the `libscl` distribution.

# cblas_dgemm

Computes: $C \leftarrow \alpha AB + \beta C$

Syntax:

```
cblas_degmm(Order, TransA, TransB,
  M, N, K,
  alpha, A, lda, B, ldb,
  beta, C, ldc)
```

Used in: `dgmprd.cpp` and `realmat.cpp`

Example:

```
//The following call computes C = A*B;

realmat C(A.nrow(),B.ncol());
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
  A.nrow(), B.ncol(), A.ncol(),
  1.0, A.begin(), A.nrow(), B.begin(), B.nrow(),
  0.0, C.begin(), C.nrow());
```

# General description of cblas_dgemm

More generally: $C \leftarrow \alpha op(A) op(B) + \beta C$

If TransA $==$ CblasNoTrans, then $op(A) = A$.
If TransA $==$ CblasTrans, then $op(A) = A'$
M and K are the number of rows and columns of $op(A)$, respectively

If TransB $==$ CblasNoTrans, then $op(B) = B$.
If TransB $==$ CblasTrans, then $op(B) = B'$
K and N are the number of rows and columns of $op(B)$, respectively

M and N are the number of rows and columns of $C$, respectively.

If Order $==$ CblasColMajor then matrices are stored columnwise and lda is the number of rows that $A$ was dimensioned with, ldb is the same for $B$, and ldc is the same for $C$.

If Order $==$ CblasRowMajor then matrices are stored by rows and lda is the number of columns that $A$ was dimensioned with, ldb is the same for $B$, and ldc is the same for $C$.

A, B, and C are pointers to the first elements of $A$, $B$, and $C$, respectively, which are all arrays of double.

alpha and beta are double.

# cblas_dgemm

More generally: $C \leftarrow \alpha op(A) op(B) + \beta C$

Syntax:

```
cblas_degmm(Order, TransA, TransB,
  M, N, K,
  alpha, A, lda, B, ldb,
  beta, C, ldc)
```

Example:

```
//The following call computes C = A'*B;

INTEGER M = A.ncol();
INTEGER N = B.ncol();
INTEGER K = A.nrow();
realmat C(M,N);

cblas_dgemm(CblasColMajor, CblasTrans, CblasNoTrans,
  M, N, K,
  1.0, A.begin(), A.nrow(), B.begin(), B.nrow(),
  0.0, C.begin(), C.nrow());
```

# How dgemm Works

It localizes the computation to ensure that all variables are in the CPU's cache. The goal is to minimize waiting for reading and writing to memory.

Look at /proc/cpuinfo on latte and other machines to see cache size.

Loop unrolling (or loop unwinding) is a device to get the cache filled with the correct elements. Unrolling examples follow.

The `cblas` also reorders instructions to exploit pipelining.

# Loop Unrolling

```
/*
Compute C=A*B where C is MxN, A is MxK, B is KxN, and
cache holds at least 12 elements
*/
INTEGER MN=M*N;
for (INTEGER i=1; i<=MN; ++i) C[i] = 0.0;
INTEGER M0=4*(M/4); INTEGER M1=M-M%4+1;
for (INTEGER k=1; k<=K; ++k) {
  for (INTEGER j=1; j<=N; ++j) {
    for (INTEGER i=0; i<M0; i+=4) {
      C(i+1,j) += A(i+1,k)*B(k,j);
      C(i+2,j) += A(i+2,k)*B(k,j);
      C(i+3,j) += A(i+3,k)*B(k,j);
      C(i+4,j) += A(i+4,k)*B(k,j);
    }
    for (INTEGER i=M1;i<=M;++i) C(i,j) += A(i,k)*B(k,j);
  }
}
```

Remark: The GNU g++ compiler will automatically unroll loops under -O3.

# Duff's Device

```
INTEGER MN=M*N;
for (INTEGER i=1; i<=MN; ++i) C[i] = 0.0;
for (INTEGER k=1; k<=K; ++k) {
  for (INTEGER j=1; j<=N; ++j) {
    INTEGER i = 0;
    INTEGER M0 = (M + 7) / 8;
    switch (M % 8) {
      case 0: do {
              C(++i,j) += A(i,k)*B(k,j);
      case 7: C(++i,j) += A(i,k)*B(k,j);
      case 6: C(++i,j) += A(i,k)*B(k,j);
      case 5: C(++i,j) += A(i,k)*B(k,j);
      case 4: C(++i,j) += A(i,k)*B(k,j);
      case 3: C(++i,j) += A(i,k)*B(k,j);
      case 2: C(++i,j) += A(i,k)*B(k,j);
      case 1: C(++i,j) += A(i,k)*B(k,j);
              } while (--M0 > 0) ;
    }
  }
}
```

# Some Timings

Program `src/cblas/mult`, Xeon Quad Core 3.16GHz 6144KB cache,
M, N, K = 16005 1000 100

```
Do not depend on optimization flag:
Using libscl's C = A*B, dgmprd    clock = 4.78, 4.61
Using libsclcb's C = A*B, dgmprd  clock = 0.74, 0.5
Using cblas_dgemm                 clock = 0.5

Optimization flag O0:
Using loop unrolling              clock = 26.92
Using Duff's device               clock = 26.76
Using no acceleration techniques  clock = 27.29

Optimization flag O1:
Using loop unrolling              clock = 6.94
Using Duff's device               clock = 6.91
Using no acceleration techniques  clock = 7.14

Optimization flag O2:
Using loop unrolling              clock = 5.68
Using Duff's device               clock = 6.12
Using no acceleration techniques  clock = 5.76

Optimization flag O3:
Using loop unrolling              clock = 5.71
Using Duff's device               clock = 6.13
Using no acceleration techniques  clock = 5.76
```

# Code

- Look at `makefiles` and `mult.cpp` in `src/cblas`.
  - Mention `stopwatch` in libscl.

- Look at `dgmprd` in `libscl`.

# Chapter 3. Working with batches of data

Main points:

- introduces the first and simplest of the container classes in the STL, the vector class
- shows how to apply an algorithm from <algorithm> to a container class, namely sort
- describes the logic behind using "while (cin >> x) { ... }" to test for end of file
- describes how to change and restore the precision of an ostream
  - Can use `fmt` in `libscl` instead

# Container classes

The most useful in computational economics:

1. string, used to store and manipulate character data

2. vector, used to store any type that is usefully indexed by an int

3. list, like a vector, but used when there will be a lot of insertions and both ends will be growing

4. associative maps, used to store any type and index it by any type for which comparison can be defined; i.e. realmats indexed by realmats

# Begin and end

The text showed how to use the member functions begin and end of the vector class as inputs to the sort algorithm. You may have noticed that sort(b,e) doesn't have to be told much. All it needs is to have the ends marked and to have the container hold a type for which the operator $<$ is defined. What kind of container it is doesn't matter. That is, sort works for maps, lists, etc.

As will be seen in Chapter 5, the members begin, end, and iterator are the main tools used to gain this generality for algorithms.

They can also make your own programs more general. The container you use can be changed without having to rewrite code.

# Begin and end – continued

When we looked at pointers we studied this idiom:

```
int n;
double x[n];
//...
double* xi = &x[0];
double* top = xi + n;
while (xi < top) {
  *xi++ = something;
}
```

Begin and end are manifestations of this idiom. Begin corresponds to `&x[0]` and end corresponds to `top`. In fact, if you did this: `sort(&x[0],top)`, the array `x` would get sorted.

# End of file

The usage

```
ifstream stream("filename");
while(stream >> x) { // do something }
```

works because the expression (stream $>>$ x) returns the stream for which is was called. What is missing from the discussion is the point that there is a function called a conversion operator that takes an istream as its argument and returns a bool and that the compiler automatically calls this conversion operator.

Another approach is

```
ifstream stream("filename");
while( (stream >> x).good() ) { //... }
```

# Conversions

Whenever one type is expected as an argument and another is given, the compiler will try to find either a constructor or conversion operator to convert from the given type to the expected type. If a conversion operator or constructor exists, it can be used explicitly. For instance, `bool a=bool(cin);`. Another example is `intvec idx=intvec("1:15");` which converts a C-style string to an `intvec`. Here is an instance where you actually need to use explicit conversion to get the desired result:

```
realmat X(r,c);
for {INTEGER i=1; i<=r; i++) {
  for {INTEGER j=1; j<=c; j++) {
    X(i,j) = REAL(i)/REAL(j);
  }
}
```

## prog01

```
int main(int argc, char** argp, char** envp)
{
  // If you know how big a vector will be ahead of time, you can declare
  // its length and use it like an array rather than use push_back.

  INT_32BIT seed = 12345;

  typedef vector<REAL>::size_type rv_int;

  rv_int len_rv = 9;

  vector<REAL> rv(len_rv);

  for (rv_int i=0; i<len_rv; i++) {    // Note: counting from zero
    rv[i] = scl::unsk(seed);           // unsk is normal(0,1)
  }

  cout << starbox("/Contents of rv//") << '\n';
  for (rv_int i=0; i<rv.size(); i++) { // Note: counting from zero
    cout << "     rv["<<i<<"] = " << rv[i] << '\n';
  }
```

## prog01 (continued)

```
// This usage has some of the pitfalls of arrays.  The use of
// push_back as in the book is safer.  If you want to make it
// more efficient, do this.

rv_int est_len_rmv = 1000;
vector<realmat> rmv;
rmv.reserve(est_len_rmv);

INTEGER j = 1;
while (j < 9) {
  realmat rm(1,j);
  for (INTEGER i=1; i<=j; ++i) {  // Note: counting from one
    rm[i] = scl::ran(seed);       // ran is uniform on (0,1)
  }
  j = scl::iran(seed,9)+1;        // iran is uniform on 0,..,9
  rmv.push_back(rm);
}
```

## prog01 (continued)

```
  // Regardless of the amount of space reserved, this works
  // the same as in the text and size reports what was actually
  // pushed, not what you reserved.

  cout << starbox("/Contents of rmv//");

  typedef vector<realmat>::size_type rmv_int;
  for (rmv_int i=0; i<rmv.size(); ++i) { // Note: counting from zero.
    cout << rmv[i];
  }
  cout << "\n     Unused capacity = "<< rmv.capacity()-rmv.size()<<'\n';

  // As with vectors, realmats can be grown as needed.

  realmat A(2,1,REAL_MAX);
  realmat B(1,2,-REAL_MAX);
  for (INTEGER i=1; i<=iran(seed,15)+1; ++i) { // Note: from one
    realmat new_col(2,1,unsk(seed));
    A = cbind(A,new_col);
    realmat new_row(1,2,unsk(seed));
    B = rbind(B,new_row);
  }

  cout << starbox("/Contents of A and B//") << A << B;

  return 0;
}
```

## prog01 output

```
*********************************************************************
*                                                                   *
*                           Contents of rv                          *
*                                                                   *
*********************************************************************

rv[0] = 0.190261
rv[1] = 0.532602
rv[2] = -1.57
rv[3] = 1.87929
rv[4] = -2.14323
rv[5] = -0.577296
rv[6] = -0.489348
rv[7] = 0.725152
rv[8] = -1.01215
```

## prog01 output (continued)

```
**********************************************************************
*                                                                    *
*                          Contents of rmv                           *
*                                                                    *
**********************************************************************


                                      Col   1

                          Row   1     0.91065


                    Col   1       Col   2       Col   3       Col   4

        Row   1     0.98221       0.94942       0.93087      0.051522

Unused capacity = 998
```

# prog01 output (continued)

```
**********************************************************************
*                                                                    *
*                         Contents of A and B                        *
*                                                                    *
**********************************************************************


                    Col   1      Col   2      Col   3      Col   4

        Row   1 1.7977e+308    -1.89638     0.17084     0.44763
        Row   2 1.7977e+308    -1.89638     0.17084     0.44763


                                 Col   1      Col   2

                    Row   1-1.7977e+308-1.7977e+308
                    Row   2     0.20556     0.20556
                    Row   3     0.25277     0.25277
                    Row   4     1.18597     1.18597
```

# Output Precision

The text p. 43 suggests

```
streamsize prec = cout.precision();
cout << "Your final grade is " << setprecision(3)
     << 0.2 + midterm + 0.4*final + 0.4*sum/count
     << setprecision(prec) << endl;
```

My preference is to use a manipulator from libscl

```
cout << "Your final grade is "
     << fmt('f',5,3,0.2 + midterm + 0.4*final + 0.4*sum/count)
     << endl;
```

Discuss `libscl/src/fmt.cpp`

# Chapter 4. Organizing programs and data

Main points:

- call by value, call by reference, call using pointers, `const`
- overloading
- states of an iostream, clever method of returning iostream as a reference
- error handling with exceptions, try and catch
- header files (.h files) and source files (.cpp files aka .c, .C, and .cc)
- structs
- algorithms and predicates (compare function)
- compiler directives (#ifndef, #define, etc.)

## Call by Value: value.cpp

```
#include "libscl.h"
using namespace scl;  using namespace std;

REAL f(realmat b) {b(1,1)=5.0; return b(1,1);}

int main()
{
  realmat a(5,5,0.0);
  REAL r = f(a);
  cout << "a(1,1) = " << a(1,1) << '\n';
  cout << "r = " << r  << '\n';
  return 0;
}

Output: a(1,1) = 0
        r = 5
```

The function `f` cannot change `a`; `b` is a copy of `a`. Inefficient if `a` is large and a copy is not actually needed.

## Call by Reference: refer.cpp

```
#include "libscl.h"
using namespace scl;  using namespace std;

REAL f(realmat& b) {b(1,1)=5.0; return b(1,1);}

int main()
{
  realmat a(5,5,0.0);
  REAL r = f(a);
  cout << "a(1,1) = " << a(1,1) << '\n';
  cout << "r = " << r  << '\n';
  return 0;
}

Output: a(1,1) = 5
        r = 5
```

The function `f` changes `a`; `b` is a reference to `f`. Efficient. A reference is usually implemented as a pointer that is automatically dereferenced at every use. All variables in Fortran are implemented this way: pointers that are dereferenced at every use.

# Call Using a Pointer: ptr.cpp

```
#include "libscl.h"
using namespace scl;  using namespace std;

REAL f(realmat* bptr) {(*bptr)(1,1)=5.0; return (*bptr)(1,1);}

int main()
{
  realmat a(5,5,0.0);
  REAL r = f(&a);
  cout << "a(1,1) = " << a(1,1) << '\n';
  cout << "r = " << r  << '\n';
  return 0;
}
Output: a(1,1) = 5
        r = 5
```

Efficient but error prone. The function `f` cannot change `bptr` because `bptr` is passed by value. But `f` can change the thing pointed to, which is `a`. Calls using pointers are often necessary to use code intended to be used with both C and C++. The MPI parallel processing library is an example. This is how you call a Fortran subroutine from C++.

# Call Using const Reference: crefer.cpp

```
#include "libscl.h"
using namespace scl;  using namespace std;

//REAL f(const realmat& b) {b(1,1)=1.0; return b(1,1);}  //This is an error.
REAL f(const realmat& b) {realmat a=b; a(1,1)=1.0; return a(1,1);} //This is OK.

int main()
{
  realmat a(5,5,0.0);
  REAL r = f(a);
  cout << "a(1,1) = " << a(1,1) << '\n';
  cout << "r = " << r  << '\n';
  return 0;
}
Output: a(1,1) = 0
        r = 1
```

`b(1,1)=1.0;` violates the const promise but taking a copy `realmat a=b;` honors the const promise.

# Call Using const Pointer: cptr.cpp

```
#include "libscl.h"
using namespace scl;  using namespace std;

//REAL f(const realmat* bptr) {realmat* a=bptr; return (*a)(1,1);}   //Error.
//REAL f(const realmat* bptr) {const realmat* a=bptr; return (*a)(1,1);}//OK.
REAL f(const realmat* bptr) {realmat a=*bptr; a(1,1)=1; return a(1,1);} //OK.

 int main()
{
  realmat a(5,5,0.0);
  REAL r = f(&a);
  cout << "a(1,1) = " << a(1,1) << '\n';
  cout << "r = " << r  << '\n';
  return 0;
}

Output: a(1,1) = 0
        r = 1
```

`realmat* a=bptr;` violates the const promise. `const realmat* a=bptr;` honors the const promise. Taking a copy `realmat a=*bptr;` honors the const promise.

# Overloading: over.cpp

```
#include "libscl.h"
using namespace scl;  using namespace std;

REAL f(realmat b) {cout << "#1 "; return b[1];}
REAL f(realmat* bptr) {cout << "#2 "; return (*bptr)[1];}
REAL f(REAL b) {cout << "#3 "; return b;}
REAL f(INTEGER b) {cout << "#4 "; return b;}

int main()
{
  realmat a(5,5,1.5);
  cout << f(a) << '\n';
  cout << f(&a) << '\n';
  cout << f(a[1]) << '\n';
  cout << f(INTEGER(a[1])) << '\n';
  return 0;
}

Output: #1 1.5
        #2 1.5
        #3 1.5
        #4 1
```

The compiler figures out which function is meant by the function's arguments.

## Clever

```
istream& read(istream& is, Student_info& s)
{
  // read and store the student's name and
  // midterm and final exam grades
  is >> s.name >> s.midterm >> s.final;

  read_hw(is, s.homework);  // read and store all the
  return is;                // student's homework grades
}
```

One doesn't have to do a lot of error checking. Just return the `istream` and let the calling program figure out if an error occurred.

# Disaster

```
istream& read(Student_info& s)
{
  ifstream is("grades.txt");
  is >> s.name >> s.midterm >> s.final;
  read_hw(is, s.homework);
  return is;
}
```

`is` is destroyed on exit from `read` — remember scope rules — and any attempt to use `is` by the calling program will most likely cause termination with a segmentation fault. But there are no guarantees other than that something awful will happen.

## Exceptions

The text explains the use of error handling by means of exceptions: `try` ... `catch(object)`. Library `libscl` uses an older style of error handling that is usually better for numerical analysis, where the error is most likely a coding error and no recovery is possible. It is easy to modify `libscl` error handling. To see how to modify it to use the exceptions style, look at the instructions in `sclerror.h` and examples in `tstrm.cpp`.

Sometimes one must modify `libscl` error handling. Parallel processing is an example because MPI implementations have their own error handlers that should be used. We will see an example of this in the case study: `habit_main_mpi.cpp`.

# Headers and Separate Compilation

## funcs.h

```
#ifndef __FILE_FUNCS_H_SEEN__
#define __FILE_FUNCS_H_SEEN__

#include "libscl.h"

extern REAL f(scl::realmat b);          // Note the usage scl::realmat
extern REAL f(scl::realmat* bptr);      // It is unacceptable to have
extern REAL f(REAL b);                  // a using statement in a header
extern REAL f(INTEGER b);

#endif
```

# Headers and Separate Compilation

## funcs.cpp

```
#include "funcs.h"
using std::cout;

REAL f(realmat b) {cout << "#1 "; return b[1];}
REAL f(realmat* bptr) {cout << "#2 "; return (*bptr)[1];}
REAL f(REAL b) {cout << "#3 "; return b;}
REAL f(INTEGER b) {cout << "#4 "; return b;}
```

# Headers and Separate Compilation

## fmain.cpp

```
#include "libscl.h"
#include "funcs.h"
using namespace scl;  using namespace std;

int main()
{
  realmat a(5,5,1.5);
  cout << f(a) << '\n';
  cout << f(&a) << '\n';
  cout << f(a[1]) << '\n';
  cout << f(INTEGER(a[1])) << '\n';
  return 0;
}
```

# Headers and Separate Compilation

## makefile

```
CC       = g++
SDIR     = .
ISCL     = $(HOME)/lib/libscl/gpp
LSCL     = $(HOME)/lib/libscl/gpp
CFLAGS   = -O -Wall -c -I$(SDIR) -I$(ISCL)
LFLAGS   = -lm -L$(LSCL) -lscl

fmain : fmain.o funcs.o
        $(CC) -o fmain fmain.o funcs.o $(LFLAGS)

fmain.o : $(SDIR)/fmain.cpp $(SDIR)/funcs.h
        $(CC) $(CFLAGS) $(SDIR)/fmain.cpp

funcs.o : $(SDIR)/funcs.cpp $(SDIR)/funcs.h
        $(CC) $(CFLAGS) $(SDIR)/funcs.cpp
```

# Structs

```
struct Student_info {
  string name;
  double midterm;
  double final;
  vector<double> homework;
};  // note the semicolon -- it's required
//...
vector<Student_info> students;
students[0].name = "Paul";
//...
Student_info si;
si.name = "Paul";
//...
Student_info* si_ptr = &si;
(*si_ptr).name = "Paul";
si_ptr->name = "Paul";
//...
```

A struct is a way to collect together a group of different types. It is itself a type and can be used as a type. How to access member student is illustrated.

# Demo

Compile and run value, refer, ptr, crefer, cptr, over, and fmain.

# Warning.

Read slides for Chapter 6 before reading Chapter 6 to save yourself unnecessary reading because we shall skip much of Chapter 6.

# Chapter 5. Using sequential containers and analyzing strings.

Main points:

- iterators
- iterator types, iterator operations
- erasing, inserting, and iterator invalidation
- lists
- string manipulation
- details in Section 5.9

# Indexing with pointers

This is the fastest way to compute a sum of the elements of an array and is instinctive with experienced C programmers:

```
int n=5000;
double a[n];
\\ fill a with something
double* t = a;
double* top = a + n;
double sum = 0.0;
while(t<top) {
  sum += *t++;
}
```

## Things to Notice

- One begins at a pointer to the first element, which is `a`, and uses a pointer to one past the last element, which is `top`, to stop the iterations.

- Iterators are the natural abstraction of this usage.

# Iterators and Pointers

In fact, the notion of a pointer and iterator are so similar that a pointer is an example of an iterator. The following code actually does work.

```
int n=5000;
double a[n];
\\ fill a with something
double* t = a;
double* top = a + n;
sort(t,top);
while(t<top) {
  cout << *t++ << '\n';
}
```

# Iterators and Pointers

Recall, page 44, that the sort function was suggested for use with the vector container class. But, as just seen, it works for any container type that

1. permits random access
2. has a pointer to the first element,
3. has a pointer to one past the last element,
4. and for which the operation `*a < *b` is defined.

# Example of Iterator Usage

```
vector<realmat> v;

for(INTEGER i=1; i<=5; i++) {
  realmat x(1,5);
  for (INTEGER j=1; j<=5; ++j) {
    x[j] = unsk(seed);
  }
  v.push_back(x);
}

vector<realmat>::const_iterator v_iter;

cout << starbox("/before sort//");

for (v_iter = v.begin(); v_iter < v.end(); ++v_iter) {
  cout << *v_iter;
}

sort(v.begin(),v.end(),realmat_cmp());

cout << starbox("/after sort//");

for (v_iter = v.begin(); v_iter < v.end(); ++v_iter) {
  cout << *v_iter;
}
```

# Example of Iterator Usage

```
**********************************************************************
*                            before sort                             *
**********************************************************************

              Col   1     Col   2     Col   3     Col   4     Col   5
  Row   1    -2.52605     0.46897     1.12823     1.22625     0.66402
  Row   1    -0.39204   -0.048188     1.82471   -0.020153   -0.055393
  Row   1    -0.52677    -1.44619     0.66122     1.01763     0.43759
  Row   1     1.16838    0.078569    -0.50887    -0.84152     1.73040
  Row   1    -0.65305    -0.28374     0.41111    0.097130    -0.80161

**********************************************************************
*                             after sort                             *
**********************************************************************

              Col   1     Col   2     Col   3     Col   4     Col   5
  Row   1    -2.52605     0.46897     1.12823     1.22625     0.66402
  Row   1    -0.65305    -0.28374     0.41111    0.097130    -0.80161
  Row   1    -0.52677    -1.44619     0.66122     1.01763     0.43759
  Row   1    -0.39204   -0.048188     1.82471   -0.020153   -0.055393
  Row   1     1.16838    0.078569    -0.50887    -0.84152     1.73040
```

# Pitfalls

There are two easily made mistakes that can be made with pointers to a container:

1. Forgetting to preserve a copy of a pointer to the first element before incrementing. If you do not have some way to retrieve a pointer to the first element, then that container is lost. E.g. `void f(double* a)` followed by a random bunch of `a++` in the body of `f`; the container `a` is now lost within the body of function `f`.

2. Forgetting that some operations, like erasure, can cause a pointer to point to something you do not want it to point to. The discussion of this in the text is good.

# cutstr

`cutstr` in `libscl` is a generalization of the book's `split`

```
std::vector<string> cutstr(const std::string& str, char delim);
```

1. The delimiter is white space other than a tab.
   Returns the delimited words with white space stripped.

2. The delimiter is a tab.
   The tab is stripped. White space is not.

3. The delimiter is a comma.
   Cuts the string into words using Microsoft Excel CSV rules.
   The comma is stripped. White space is not.

4. The delimiter is other than the above.
   The delimiter is stripped. White space is not.

# Illustration and Homework

- Discuss and run program `prog01`.

- Discuss homework and illustrate with `time prog02`, `time prog03`.

- Go through header and source for `intvec` to introduce ideas of classes and how container classes are coded.

  ▷ Just briefly touch on the ideas in the code because the code is discussed in detail in the slides for Chapter 11.

# Chapter 6. Using library algorithms

Main points:

- generic algorithm, declared in <algorithm>, – find, find_if, search, copy, remove_copy, partition, etc.
- iterator adapter, declared in <iterator>, – back_inserters, front_inserter, inserter, . . .
- prefix `x=++a;`, same as `a++; x=a;`, and postfix `x=a++;`, same as `x=a; a++;`. Warning: do not rely on this behavior except for built-in variables and iterators.
- Functions that return `void` and how to exit them with `return;` or falling off the end.
- The ideas of this chapter are useful for text and list processing but not of much use in numerical analysis with the exception of `find`, `copy`, `accumulate`, and `transform`. Skim chapter; read Section 6.5 carefully.
- Discuss `accumulate` and `transform` in class. Illustrate with `prog08` in `src/ch05`.
- Warning: Never use the `static` qualifier in code that might be parallelized.

# Chapter 7. Using associative containers

Main points:

- associative container or `map`, declared in <map> – very useful in numerical work
- pairs – how one recovers the key-value pair from a `map` iterator
- How to use `find` to make sure a map element exists.
- If the key is not already in the map, then accessing the map adds the key and initializes the value using its default constructor.
- How to declare a function type and how to default initialize the last argument(s) of a function.
- Recursion – a function can call itself.

# Maps, pairs, recursion, and find

The ideas of maps, pairs, and recursion will be illustrated by an example: computing a multivariate polynomial.

The example does not illustrate the use of `find`. Briefly, the problem is this: If a map `m` (defined as `map<K,V> m;`) is accessed using `m[k]`, where `k` has type `K`, and the element `m[k]` does not exist, then a sort of automatic push_back occurs and `m[k]` is created. To avoid this behavior, use `m_itr=m.find(k)`, where `m_itr` has type `map<K,V>::iterator` to access `m`. If `m[k]` is not in the map, then `m_itr==m.end()`. The discussion of the book is good. Another way to deal with this is to check `v`, where `v=m[k]`, for a special value that occurs only when `v` is default initialized. Examples of the latter occur in the case study.

# Function type

In the type declarations

```
int f(double);
int (*g)(double);
```

the identifier `f` has type "function returning int" and the identifier `g` has type "pointer to function returning int". The book suggests the usage

```
int h(double x, int p(double)) {return p(x);}
// ...
i=h(x,f);
```

# Function type

My readings suggest that the following is what actually happens:

```
int h(double x, int (*ptr)(double))
{
  return (*ptr)(x);
}
//...
i=h(x,&f);
```

My readings suggest that this automatic pointer and address generation for function calls is in both the C++ and ANSI C language standards. Moreover, the type `int f(double)` no longer actually exists. It now gets interpreted as `int (*f)(double)`. My readings also suggest that some think what the book does is the preferred style.

# Function type

Most experienced programmers would write the code this way:

```
int f(double x) {// do something;}
// ...
typedef int (*F_PTR)(double x);
// ...
int h(double x, F_PTR ptr)) {return (*ptr)(x);}
// ...
i=h(x,&f);
```

This is my style.

# Multivariate Polynomials

We shall use maps and recursions to show how to implement a multivariate polynomial. First some background.

A multi-index is a vector with elements that are either zero or positive. Here is one

$$\lambda = (0, 3, 5).$$

A monomial can be represented as a vector raised to a multi-index power. For $x = (5.0,\ 6.1,\ 2.3)$, here it is

$$x^\lambda = x_1^{\lambda_1} x_2^{\lambda_2} x_3^{\lambda_3} = (5.0)^0 (6.1)^3 (2.3)^5 = 14609.3$$

An intvec can represent $\lambda$ and a realmat can represent $x$.

# Multivariate Polynomials

Index let $a_\lambda$ be the coefficient of the monomial $x^\lambda$ in a multivariate polynomial and let

$$|\lambda| = \sum_{i=1}^{d} \lambda_i,$$

where $d$ is the dimension of $x$; recall that the elements of $\lambda$ must be zero or positive. A multivariate polynomial of degree $K$ can be written as

$$\mathcal{P}_K(x) = \sum_{|\lambda| \le K} a_\lambda x^\lambda.$$

This notation is standard.

# Multivariate Polynomials

The minimal object that we need to compute in order to implement a multivariate polynomial is a column vector

$$b(x) = \left( x^{\lambda_{(1)}}, x^{\lambda_{(2)}}, \ldots, x^{\lambda_{(n)}} \right)'$$

where $\lambda_{(1)}$, $\lambda_{(2)}$ $\ldots$, $\lambda_{(n)}$ is some ordering of the multi-indexes that satisfy $|\lambda| \le K$.

It would also be nice to know the value of $n$, to have a vector containing the ordered multi-indexes, and to have the Jacobian of $b(x)$, which is the $n \times d$ matrix

$$\left( \frac{\partial}{\partial x'} \right) b(x) = \left[ \left( \frac{\partial}{\partial x} \right) x^{\lambda_{(1)}}, \ldots, \left( \frac{\partial}{\partial x} \right) x^{\lambda_{(n)}} \right]'$$

It would be frosting on the cake to exclude interactions of large degree. An interaction is a multi-index with more than one element not zero.

# Example, deg_main = 2, deg_inter = 2

```
x transpose =  (1, 2, 3)

basis     multi        jacobian
 1.0      0 0 0      0.0  0.0  0.0
 3.0      0 0 1      0.0  0.0  1.0
 9.0      0 0 2      0.0  0.0  6.0
 2.0      0 1 0      0.0  1.0  0.0
 6.0      0 1 1      0.0  3.0  2.0
 4.0      0 2 0      0.0  4.0  0.0
 1.0      1 0 0      1.0  0.0  0.0
 3.0      1 0 1      3.0  0.0  1.0
 2.0      1 1 0      2.0  1.0  0.0
 1.0      2 0 0      2.0  0.0  0.0
```

# Example, deg_main = 3, deg_inter = 2

```
x transpose =  (1, 2, 3)

basis     multi        jacobian
 1.0      0 0 0       0.0   0.0   0.0
 3.0      0 0 1       0.0   0.0   1.0
 9.0      0 0 2       0.0   0.0   6.0
27.0      0 0 3       0.0   0.0  27.0
 2.0      0 1 0       0.0   1.0   0.0
 6.0      0 1 1       0.0   3.0   2.0
 4.0      0 2 0       0.0   4.0   0.0
 8.0      0 3 0       0.0  12.0   0.0
 1.0      1 0 0       1.0   0.0   0.0
 3.0      1 0 1       3.0   0.0   1.0
 2.0      1 1 0       2.0   1.0   0.0
 1.0      2 0 0       2.0   0.0   0.0
 1.0      3 0 0       3.0   0.0   0.0
```

# Multivariate Polynomials

If the dimension `d` were known at compile time one could generate the monomials using `d` nested for loops:

```
INTEGER d, deg;        // dimension, degree
intvec multi(d);       // a multi index
REAL y;                // value of the monomial
for (INTEGER i=0; i<=deg; ++i) {
  multi[1] = i;
  y = pow(x[1],i);
  for (INTEGER j=0; j<=deg-i; ++j) {
    multi[2] = j;
    y *= pow(x[2],j);
    //...
        for (INTEGER l=0; l<=deg-i-j-...-k; ++l) {
          multi[d] = l;
          cout << y*pow(x[d],l) << multi << '\n';
        }
    //...
  }
}
```

When `d` is not known at compile one can use recursion to generate the `d` nested for loops. Details follow.

# Multivariate Polynomials
# – declaration (in libscl.h)

```
class poly {
private:
  char    type_poly;        // 'r' regular, 'h' Hermite, 'l' Laguerre
  realmat x;                // current value of x
  INTEGER dim_x;            // dimension of x
  INTEGER deg_main;         // degree of main effects
  INTEGER deg_inter;        // degree of interactions
  INTEGER len_basis;        // number of basis functions
  realmat powers;           // table lookup for pow(x[i],j)
  realmat derivs;           // table lookup for (d/dx)pow(x[i],j)
  void make_monomials(INTEGER d, REAL y, intvec midx);
  std::map<intvec,REAL,scl::intvec_cmp> monomials;
public:
  poly(char type, const realmat& x_init, INTEGER d_main, INTEGER d_inter);
  void    set_x(const realmat& x_new);
  INTEGER get_len() const { return len_basis; }
  void    get_basis(realmat& basis) const;
  void    get_multi(std::vector<intvec>& multi) const;
  void    get_multi(std::vector<std::string>& multi,char c) const;
  void    get_jacobian(realmat& jacobian) const;
};
```

## Using Class Poly

Illustrate with prog00.cpp in src/ch07

# Multivariate Polynomials – constructor (in poly.cpp)

```
poly::poly
  (char type, const realmat& x_init,
    INTEGER d_main, INTEGER d_inter)
  : type_poly(type), dim_x(x_init.size()),
    deg_main(d_main), deg_inter(d_inter)
{
  switch (type_poly) {
    case 'r':
    case 'h':
    case 'l':
      break;
    default :
      error ("Error, poly, type must be 'r','h' or 'l'");
      break;
  }

  set_x(x_init);

  len_basis = monomials.size();
}
```

# Multivariate Polynomials
# – set_x (in poly.cpp)

```
void poly::set_x(const realmat& x_new)
{
  if (x_new.ncol() != 1 && x_new.nrow() != 1) {
      error("Error, poly, x must be either a row or column vector");
  }
  if (x_new.nrow() != dim_x) error("Error, poly, dim_x cannot change");

  x = x_new;

  powers.resize(dim_x,deg_main+1);
  derivs.resize(dim_x,deg_main+1);

  switch (type_poly) {

    case 'r':
      for (INTEGER i=1; i<=dim_x; i++) {
        powers(i,1) = 1.0;                              // code requires
        derivs(i,1) = 0.0;                              // that the first
        for (INTEGER j=1; j<=deg_main; j++) {           // be 1.0, don't
          powers(i,j+1) = powers(i,j)*x[i];             // add a type poly
          derivs(i,j+1) = REAL(j)*powers(i,j);          // for which false
        }
      }
      break;
```

# Multivariate Polynomials – set_x (in poly.cpp)

```
case 'h':                                     // orthogonal wrt MVN
  for (INTEGER i=1; i<=dim_x; i++) {
    powers(i,1) = 1.0;                                    // deg 0
    derivs(i,1) = 0.0;
    powers(i,2) = x[i];                                   // deg 1
    derivs(i,2) = 1.0;
    REAL r0 = 1.0;
    for (INTEGER j=2; j<=deg_main; j++) {
      REAL r1 = sqrt(REAL(j));
      powers(i,j+1) = (powers(i,j)*x[i]
                       - r0*powers(i,j-1))/r1;    // deg j
      derivs(i,j+1) = r1*powers(i,j);
      r0 = r1;
    }
  }
  break;
```

# Multivariate Polynomials
# – set_x (in poly.cpp)

```
    case 'l':                              // orthogonal wrt exp
      for (INTEGER i=1; i<=dim_x; i++) {
        powers(i,1) = 1.0;                             // deg 0
        derivs(i,1) = 0.0;
        powers(i,2) = 1.0 - x[i];                      // deg 1
        derivs(i,2) = -1.0;
        REAL r0 = 1.0;
        for (INTEGER j=2; j<=deg_main; j++) {
          REAL r1 = REAL(j);
          REAL p=(powers(i,j)*(2.0*r1-1.0-x[i])-r0*powers(i,j-1))/r1;
          REAL d=(derivs(i,j)*(2.0*r1-1.0-x[i])-r0*derivs(i,j-1))/r1;
          d -= powers(i,j)/r1;
          powers(i,j+1) = p;
          derivs(i,j+1) = d;
          r0 = r1;
        }
      }
      break;
  }
  make_monomials(dim_x, REAL(), intvec());
}
```

The real work is done by make_monomials. Member functions like set_x are often called methods.

# Multivariate Polynomials – make_monomials (in poly.cpp)

```
void poly::make_monomials(INTEGER d, REAL y, intvec midx)
{
  if (d == dim_x) {     // external call must have d == dim_x

    midx.resize(dim_x,0);

    for (INTEGER j=0; j<=deg_main; j++) {
      midx[1] = j;
      REAL y_new = powers(1,j+1);
      if (d == 1) {
        monomials[midx] = y_new;
      }
      else {
        make_monomials(d-1,y_new,midx);
      }
    }

  }
```

# Multivariate Polynomials
# – make_monomials (in poly.cpp)

```
else {                  // recursive calls have d < dim_x

  INTEGER sum = 0;
  for (INTEGER i=1; i<=dim_x-d; i++) {
    sum += midx[i];
  }

  if (sum == 0) {       // main effect for x[dim_x-d+1]

    for (INTEGER j=0; j<=deg_main; j++) {
      midx[dim_x-d+1] = j;
      REAL y_new = y * powers(dim_x-d+1,j+1);
      if (d == 1) {
        monomials[midx] = y_new;
      }
      else {
        make_monomials(d-1,y_new,midx);
      }
    }

  }
```

# Multivariate Polynomials – make_monomials (in poly.cpp)

```
else if (sum < deg_inter) {
     // interaction term involving x[dim_x-d+1]

  for (INTEGER j=0; j<=deg_inter-sum; j++) {
    midx[dim_x-d+1] = j;
    REAL y_new = y * powers(dim_x-d+1,j+1);
    if (d == 1) {
      monomials[midx] = y_new;
    }
    else {
      make_monomials(d-1,y_new,midx);
    }
  }

}
```

# Multivariate Polynomials – make_monomials (in poly.cpp)

```
    else {                // x[dim_x-d+1] not involved

      midx[dim_x-d+1] = 0;
      if (d == 1) {
        monomials[midx] = y;
      }
      else {
        make_monomials(d-1,y,midx);
      }

    }
  }
}
```

# Multivariate Polynomials
# – get_basis (in poly.cpp)

```
void poly::get_basis(realmat& basis) const
{
  basis.resize(len_basis,1);

  std::map<intvec,REAL,intvec_cmp>::const_iterator
        itr = monomials.begin();

  for (INTEGER i=1; i<=len_basis; i++) {
    basis[i] = itr->second;
    ++itr;
  }
}
```

# Multivariate Polynomials – get_multi (in poly.cpp)

```
void poly::get_multi(std::vector<intvec>& multi) const
{
  typedef std::vector<intvec>::size_type v_int;

  v_int v_len = len_basis;

  multi.resize(v_len);

  std::map<intvec,REAL,intvec_cmp>::const_iterator
           itr = monomials.begin();

  for (v_int i=0; i < v_len; i++) {
    multi[i] = itr->first;
    ++itr;
  }
}
```

# Multivariate Polynomials – get_jacobian (in poly.cpp)

```
void poly::get_jacobian(realmat& jacobian) const
{
  jacobian.resize(len_basis,dim_x);

  intvec multi;
  REAL basis;

  std::map<intvec,REAL,intvec_cmp>::const_iterator
         itr = monomials.begin();
```

# Multivariate Polynomials – get_jacobian (in poly.cpp)

```
  for (INTEGER i=1; i<=len_basis; i++) {

    multi = itr->first;
    basis = itr->second;
    ++itr;

    for (INTEGER j=1; j<=dim_x; j++) {

      if (multi[j] == 0) {
        jacobian(i,j) = 0.0;
      }
      else if (powers(j,multi[j]+1) != 0.0) {
        REAL ratio = basis/powers(j,multi[j]+1);
        jacobian(i,j) = ratio*derivs(j,multi[j]+1);
      }
      else {
        REAL ratio = 1.0;
        for (INTEGER k=1; k<=dim_x; k++) {
          if (k != j) ratio *= powers(k,multi[k]+1);
        }
        jacobian(i,j) = ratio*derivs(j,multi[j]+1);
      }
    }
  }
}
```

## Example

Illustrate the use of class poly with the epa example in compecon/src/ch07.

# Chapter 8. Writing generic functions

Main points:

- generic function, aka template functions
- typename
- instantiation occurs at compile time if function called, generic function must be in header, cannot be compiled separately
- types of iterators, just be aware that a hierarchy exists, don't waste time trying to learn it
- off the end values

# Template Functions

```
template <class S, class T>
return_type function_name(S sval, T tval, double x) {//...}
//...
double s,t,x;
function_name(s,t,x);
```

If the types do not appear in the argument list, they must be supplied at the call

```
template <class S, class T>
T function_name(S sval) {//...}
//...
double s,t;
t=function_name<double,double>(s);
```

May use `template <typename S, typename T>` instead of `template <class S, class T>`.

# typename

Within the body of the template function, the typename keyword must be used to qualify declarations that use types that are defined as member types of a template parameter. For example,

```
typename T::size_type identifier;
```

declares `identifier` to have type `T::size_type`. Note that `size_type` must be defined as a member type within class `T`'s definition.

# Instantiation

The entire template function source code must be available within the source code that calls the function. As a practical matter, this means putting the template function code in a header which is included.

The template function must actually be called for complete syntax checking and compilation to occur. Otherwise syntax errors in the template function may not be detected.

At compile time actual parameters are substituted for the template parameters and the resulting code is compiled. The use of the template function for one type can result in error messages that would not occur with another type; e.g. illegal conversions.

# Example: Simple Statistics for a Container Class

```
#ifndef __FILE_SIMPLE_H_SEEN__
#define __FILE_SIMPLE_H_SEEN__

#include "libscl.h"

struct stats {
  REAL mean;
  REAL sdev;
  REAL var;
  REAL skew;
  REAL kurt;
  INTEGER nobs;
};
```

```
template <class P>
stats simple(const P begin, const P end)
{
  stats ret;
  ret.mean = 0.0;
  ret.nobs = 0;

  P x = begin;
  while (x != end) {
    ret.mean += *x++;
    ++ret.nobs;
  }

  if (ret.nobs<2)
    scl::error("Error, simple, not enough data");

  ret.mean /= REAL(ret.nobs);
```

```
  REAL adev = 0.0;
  REAL r,p;
  ret.var = ret.skew = ret.kurt = 0.0;

  x = begin;
  while (x != end) {
    adev += (r = *x++ - ret.mean) > 0 ? r : -r;
    ret.var  += (p = r*r);
    ret.skew += (p *= r);
    ret.kurt += (p *= r);
  }

  adev /= REAL(ret.nobs);
  ret.var  /= REAL(ret.nobs-1);
  ret.sdev = sqrt(ret.var);
  if (ret.var) {
    ret.skew /= (REAL(ret.nobs) * ret.var * ret.sdev);
    ret.kurt = ret.kurt/(REAL(ret.nobs) * ret.var * ret.var) - 3.0;
  }
  else {
    ret.skew = ret.kurt = REAL_MAX;
  }

  return ret;
}

#endif
```

# Main: Filename from command line

```
#include "libscl.h"
#include "simple.h"

using namespace scl; using namespace std;

int main(int argc, char *argp[], char *envp[])
{
  string msg = string("Error, ") + argp[0] + ", ";

  if (argc < 2) error(msg+"specify a filename on command line");

  ifstream is(argp[1]);
  if (!is) error(msg+"bad filename");
```

## Main: Reading a file whose length is unknown efficiently

```
realmat x;
REAL r;
while (is >> r) x.push_back(r);
```

The `realmat` member function `void push_back(const REAL r);` does the buffering automatically.

# Main: Call to simple

```
  stats s = simple(rm.begin(), rm.end());

  cout << '\n'
    << "  mean     = " << s.mean << '\n'
    << "  std dev  = " << s.sdev << '\n'
    << "  variance = " << s.var  << '\n'
    << "  skewness = " << s.skew << '\n'
    << "  kurtosis = " << s.kurt << '\n'
    << "  no.obs.  = " << s.nobs << '\n' ;

  vector<REAL> vc(rm.size());
  copy(rm.begin(), rm.end(), vc.begin());

  s = simple(vc.begin(),vc.end());

  cout << '\n'
    << "  mean     = " << s.mean << '\n'
    << "  std dev  = " << s.sdev << '\n'
    << "  variance = " << s.var  << '\n'
    << "  skewness = " << s.skew << '\n'
    << "  kurtosis = " << s.kurt << '\n'
    << "  no.obs.  = " << s.nobs << '\n' ;

  return 0;
}
```

## Example

Illustrate the use of simple with the T-Bill data in compecon/src/ch08.

# Chapter 9. Defining new types

Main points:

- structs, classes
- private, public
- member functions
- constructors

# structs, classes public, private, constructors

- We have already covered the ideas in this chapter.

- Review the use of constructors with realmat.h in libscl.

  ▷ Just briefly touch on the ideas in the code because the ideas are discussed in detail in the slides for Chapter 11.

# Chapter 10. Managing memory and low level data structures

Main points:

- pointers and arrays
- pointers to functions
- string literals, array initialization
- arguments to main
- reading and writing files
- built-in memory management

## pointers, arrays
## main
## input-output
## memory management

We have already covered the ideas in this chapter.

Give full story on command line arguments (slides below).

Review the use of the new operator (slide below) and with real-mat.cpp in libscl.

# Command Line Arguments, Full Story

C and C++ language standard:

```
  int main()
  int main(void)
  int main(int argc, char** argv)
  int main(int argc, char* argv[])

  // Function getenv in stdlib.h gives access to env
  // argv[argc] is guaranteed to be a null pointer
```

Unix (POSIX excepted), Microsoft Windows:

```
  int main(int argc, char** argv, char** envp)
  int main(int argc, char* argv[], char* envp[])

  // Discussed in previous lectures and examples
  // env[size] is guaranteed to be a null pointer
```

Mac OS X and Darwin:

```
   int main(int argc, char** argv, char** envp, char** apple)
   int main(int argc, char*  argv[], char* envp[], char* apple[])

   // apple contains arbitrary OS-supplied information, apple[0]
   // is the filename, including path, of the executing binary
```

# new operator

| | |
|---|---|
| `T* p = new T` | Allocates and default-initializes an object of type `T` and returns a pointer. |
| `T* p = new T(args)` | Allocates and initializes an object of type `T` using `arg` to initialize `T`. |
| `delete p` | Destroys the object to which `p` points. |
| `T* q = new T[n]` | Allocates and default-initializes an array of size `n` of type `T` and returns a pointer. |
| `delete [] q` | Destroys the object to which `q` points. |

# Chapter 11. Defining abstract data types

Main points:

- constructors
- destructors
- copy
- assign

# constructors, destructors
# copy, assign

Every class whose constructor allocates memory dynamically must have these items because the compiler will supply them if the programmer does not. What the compiler does is usually not what ought to be done.

The book's discussion of these points is excellent.

The book describes memory management using the facilities of the `std` library rather than the `new` operator. I will illustrate with `new` which is adequate for containers of built in types such as `intvec` and `realmat`.

One should use `std` library memory management for containers of class type to avoid unnecessary calls to the default constructor.

# constructors, destructors, assignment

```
class intvec {
private:
  INTEGER         len;
  INTEGER*        ix;
                  intvec(INTEGER lgth, INTEGER* iptr)
                    : len(lgth), ix(iptr) { }
public:
                  intvec() : len(0), ix(0) { }

  explicit        intvec(INTEGER length);
                  intvec(INTEGER length, INTEGER fill_value);

  explicit        intvec(const char* str);

                  intvec(const intvec& ivec);

                  ~intvec() { delete [] ix; }

  intvec&         operator=(const intvec& ivec);
```

Note: Differs from libscl because the private variable stor is omitted.

# explicit

Without the explicit declarators, the compiler would accept the following statements and would attempt execution.

```
int main
{
  intvec ivec = "Hello world";
  intvec jvec = 5;
  return 0;
}
```

A constructor defines a conversion operator, which may have unintended side effects. With the explicit declarator, the programmer must actually code

```
int main
{
  intvec ivec("Hello world");
  intvec jvec(5);
  return 0;
}
```

to get the same outcome.

# default constructor

The constructor

```
intvec() : len(0), ix(0) { }
```

is called in these instances

```
REAL f(const intvec& ivec);
REAL g(intvec& ivec);
int main()
{
  intvec ivec;
  REAL x = f(intvec());
  REAL y = g(intvec());   //Error, won't compile.
  return 0;
}
```

# constructor, new, destructor

```
  ~intvec() { delete [] ix; }
```

Whenever the operator **`new`** is used in a constructor, a destructor must be provided, otherwise there will be a memory leak.

```
intvec::intvec(INTEGER length)
{
  if (length<=0) error("Error, intvec, intvec, length not positive");
  len = length;
  ix = new(nothrow) INTEGER[len];
  if (ix == 0) {
    len = 0;
    error("Error, intvec, intvec, operator new failed");
  }
}
```

In the following, the destructor is called when **`jvec`** goes out of scope; **`len`** gets deleted automatically, only **`ix`** that had space assigned to it by operator **`new`** needs explicit deletion.

```
int main()
{
  {intvec jvec(5);}  //destructor called on exit from this scope
  return 0;
}
```

# copy constructor

```
intvec::intvec(const intvec& ivec)
{
  len = ivec.len;
  if (len > 0) {
    ix = new(nothrow) INTEGER[len];
    if (ix == 0) {
      len = 0;
      error("Error, intvec, intvec, operator new failed.");
    }
    INTEGER* top = ix + len;
    INTEGER* t = ix;
    const INTEGER* u = ivec.ix;
    while (t < top) *t++ = *u++;
  }
  else {
    len = 0; ix = 0;
  }
}
```

## Typical usage

```
intvec f(intvec ivec) //copy constructor invoked when f called
{
  intvec jvec=ivec;   //copy constructor invoked
  intvec kvec(ivec);  //equivalent to previous line
  return jvec;        //copy constructor invoked when f returns
}
```

# assignment operator

```
intvec& intvec::operator=(const intvec& ivec)
{
  if (this != &ivec) {
    if (ivec.len > 0) {
      INTEGER* newix = new(nothrow) INTEGER[ivec.len];
      if (newix == 0)
        error("Error, intvec, operator =, operator new failed.");
      delete [] ix; // Applying delete to 0 has no effect.
      ix = newix;
      len = ivec.len;
      INTEGER* top = ix + len;
      INTEGER* t = ix;
      const INTEGER* u = ivec.ix;
      while (t < top) *t++ = *u++;
    }
    else {
      delete [] ix; // Applying delete to 0 has no effect.
      len = 0; ix = 0;
    }
  }
  return *this;  //So that chaining works; i.e. ivec=jvec=kvec.
}
```

# Use of assignment operator

```
int main()
{
  intvec ivec(5);      //explicit constructor invoked
  intvec jvec = ivec; //copy constructor invoked
  intvec kvec;         //default constructor invoked
  kvec = jvec;         //assignment operator called
  return 0;
}
```

# constructor in return

A constructor in a return statement, as in the following, will keep the copy constructor from being called in some C++ implementations. It is good style regardless.

```
intvec operator+(const intvec& ivec, const intvec& jvec)
{
  if (ivec.len != jvec.len)
    error("Error, intvec, operator +, vectors not conformable.");
  if (ivec.len == 0)
    error("Error, intvec, operator +, null matrix.");
  INTEGER* newix = new(nothrow) INTEGER[ivec.len];
  if (newix == 0)
    error("Error, intvec, operator +, operator new failed.");
  INTEGER* ri = newix;
  INTEGER* rtop = ri + ivec.len;
  const INTEGER* ai = ivec.ix;
  const INTEGER* bi = jvec.ix;
  while (ri < rtop) *ri++ = *ai++ + *bi++;
  return intvec(ivec.len,newix);
}
```

# private constructor

The constructor

```
intvec(INTEGER lgth, INTEGER* iptr) : len(lgth), ix(iptr) { }
```

is in the **private** part of class **intvec** to prevent the following usage, which would crash at execution.

```
int main()
{
  INTEGER ix[5] = {1, 2, 3, 4, 5};
  {
    intvec ivec(5,ix);
  }  //a crash will occur here when the destructor is called
     //because it was not operator new that allocated the space
     //to pointer ix
  return 0;
}
```

# Chapter 12. Making class objects act like values

Main points:

- overloading the index operator [ ]
- automatic conversions
- conversion operators
- friends
- assignment operators
- binary operators

# Overloading operator[ ]

```
class container {
private:
  INTEGER len;
  REAL *  x;
public:
  container() : len(0), x(0) {}
  container(INTEGER l) : len(l) {x = new(nothrow) REAL[len];}
 ~container() { delete [] x; }
  INTEGER size() { return len; }

//This will compile:
  const REAL& operator[] (INTEGER i) const { return x[i]; };
  REAL&       operator[] (INTEGER i) { return x[i]; };

//This will compile:
//REAL  operator[] (INTEGER i) const { return x[i]; };
//REAL& operator[] (INTEGER i) { return x[i]; };

//This will not compile:
//const REAL& operator[] (INTEGER i) { return x[i]; };
//REAL&       operator[] (INTEGER i) { return x[i]; };

//This will not compile:
//REAL  operator[] (INTEGER i) { return x[i]; };
//REAL& operator[] (INTEGER i) { return x[i]; };
};
```

# Conversions

- A conversion operator is used by the compiler to automatically convert one type to another.
- Only explicit arguments can be converted.

Consider:

```
class whatever {
public:
   double member_function(double arg1, int arg2);
          whatever(double arg3);
          whatever();
};
int main()
{
   double arg1 = 1;
   double arg2 = 2;
   double arg3 = 3;
   whatever arg0(arg3);
   double arg4 = arg0.member_function(arg1, arg2);
```

arg0 cannot be converted; arg1, arg2, arg3, and arg4 can be converted.

# Conversions

- A constructor that has a single argument is also a conversion operator.
- The explicit declarator prevents a constructor with a single argument from being a conversion operator.
- A conversion operator from an owned class to a non-owned class can be defined within the owned class.
- Conversions can be costly: functions with explicit arguments may be required for performance.
  - ▷ Even so, the g++ compiler still might convert. Why the g++ compiler converts unecessarily is a mystery (to me).
  - ▷ To stop this from hapenning, the `realmat(const trealmat&)` constructor is `explicit` in class `realmat`.

# Conversions

```
struct den_val {
  bool      positive;
  REAL      log_den;
            den_val() : positive(false), log_den(-REAL_MAX) { }
            den_val(bool p, REAL l) : positive(p), log_den(l) { }

// Constructs a den_val from REAL and converts REAL to den_val
            den_val(REAL ld) : positive(true), log_den(ld) {}

// Converts den_val to REAL
            operator REAL() const { return this->log_den; }

// Constructs a den_val from INTEGER but does not convert
  explicit den_val(INTEGER i) : positive(true), log_den(REAL(i)) {}

  den_val operator+=(den_val f)
  {
    positive = positive && f.positive;
    if (positive) log_den += f.log_den;
    else  log_den = -REAL_MAX;
    return *this;
  }
};
```

# Friends

- `friend` declarations within a class allow the friend to access private members of the class.

- Binary operators such as `+`, `-`, `*`, `/`, `==`, `!=` are usually implemented as friends to allow both arguments to be treated symmetrically and to allow both to be converted.

- The assignment operator and assignment versions of binary operators such as `+=`, `-=`, `*=`, `/=` are intrinsically asymmetric and are usually implemented as member functions.

- Illustrate with `intvec.h` and `intvec.cpp`.

# Member, Static, Friend

Stroustrup (1997, p. 278): An ordinary member function declaration specifies three logically distinct things:

1. The function can access the private part of the class declaration, and

2. the function is in the scope of the class, and

3. the function must be invoked on an object (has a `this` pointer).

By declaring a member function static, we can give it the first two properties only. By declaring a function a `friend`, we can give it the first property only.

# Chapter 13. Using inheritance and dynamic binding

This is one of the most important chapters in the book because it describes the OOP concepts that are most useful in scientific computing.

The chapter is very dense, it will require a careful reading.

This is the last chapter we shall cover in class.

You should skim the remaining chapters so you will understand the more sophisticated OOP concepts found in industrial strength code.

# Chapter 13. Using inheritance and dynamic binding

Main points:

- inheritance
- virtual functions
- polymorphism
- dynamic and static binding
- handle classes

# Inheritance and Virtual Functions

A (derived) class may build upon a (base) class, keeping features of the of the base class and adding features of its own. This is called inheritance.

If a member function of the base class is declared virtual, the derived class can redefine it. If declared pure virtual, the derived class must redefine it. A base class that has a virtual function must have a virtual destructor.

A base class may have three parts: private, protected, public. The derived class has access to the protected and public parts of a base class. Everyone has access to the public parts of any class, including a base class, but does not have access to the protected part.

# Inheritance and Virtual Functions

Two features of inheritance allow polymorphism, which means that the identity of a class does not need to be known at compile time, only at run time. They are:

A pointer to the base class may also point to a derived class.

A function argument that is a reference to the base class can accept derived classes for that argument when called.

## Inheritance and Virtual Functions

First we will look at a simple example, which is taken from homework Assignment 7.

## Homework 7, den_val, aka. denval

```
// This struct is in libscl; specifically it is in sclfuncs.h

struct den_val {
  bool    positive;
  REAL    log_den;
          den_val() : positive(false), log_den(-REAL_MAX) { }
          den_val(bool p, REAL l) : positive(p), log_den(l) { }
  den_val operator+=(den_val f)
  {
    positive = positive && f.positive;
    if (positive) log_den += f.log_den;
    else  log_den = -REAL_MAX;
    return *this;
  }
};
```

# Homework 7, base and derived classes

```
class density_base {
public:
  virtual den_val operator() (REAL) = 0;
  virtual ~density_base() { }
};

class uniform : public density_base { // public here means that what's
public:                               // public in the base class is
  den_val operator() (REAL x)         // public in the dervied class
  {
    if (0.0<=x && x<=1.0) return den_val(true,0.0);
    return den_val();
  }
};

class exponential : public density_base {
public:
  den_val operator() (REAL x)
  {
    if (0.0<=x) return den_val(true,-x);
    return den_val();
  }
};
```

# Homework 7 prob with references

```
REAL prob(density_base& f, REAL a, REAL b, INTEGER n)
{ //Compute probability with trapezoid rule
  REAL sum = 0.0;
  REAL x = a;
  REAL inc = (b - a)/REAL(n);
  if (f(a).positive) sum += exp(f(a).log_den)/2.0;
  for (INTEGER i=1; i<n; ++i) {
    x += inc;
    if (f(x).positive) sum += exp(f(x).log_den);
  }
  if (f(b).positive) sum += exp(f(b).log_den)/2.0;
  return sum*inc;
}
```

# Homework 7 main with references

```
int main(int argc, char** argp, char** envp)
{
  uniform u;
  exponential e;
  REAL a=0.0;
  REAL b=1.0;
  INTEGER g=100;
  switch (argc) {
    case 4: g=atoi(argp[3]);
    case 3: a=atof(argp[1]); b=atof(argp[2]); break;
    default: error(string("Usage: ")+argp[0]+" a b [g] "); break;
  }
  cout << prob(u,a,b,g) << '\n';
  cout << prob(e,a,b,g) << '\n';
  return 0;
}
```

# Homework 7 prob with pointers

```
REAL prob(density_base* f, REAL a, REAL b, INTEGER n)
{ //Compute probability with trapezoid rule
  REAL sum = 0.0;
  REAL x = a;
  REAL inc = (b - a)/REAL(n);
  if ((*f)(a).positive) sum += exp((*f)(a).log_den)/2.0;
  for (INTEGER i=1; i<n; ++i) {
    x += inc;
    if ((*f)(x).positive) sum += exp((*f)(x).log_den);
  }
  if ((*f)(b).positive) sum += exp((*f)(b).log_den)/2.0;
  return sum*inc;
}
```

# Homework 7 main with pointers

```
int main(int argc, char** argp, char** envp)
{
  uniform u;
  exponential e;
  REAL a=0.0;
  REAL b=1.0;
  INTEGER g=100;
  switch (argc) {
    case 4: g=atoi(argp[3]);
    case 3: a=atof(argp[1]); b=atof(argp[2]); break;
    default: error(string("Usage: ")+argp[0]+" a b [g] "); break;
  }
  density_base* f;
  f = &u;
  cout << prob(f,a,b,g) << '\n';
  f = &e;
  cout << prob(f,a,b,g) << '\n';
  return 0;
}
```

# Homework 7 instantiation at runtime

```
int main(int argc, char** argp, char** envp)
{
  char d='n';
  REAL a=0.0;
  REAL b=1.0;
  INTEGER g=100;
  switch (argc) {
    case 5: g=atoi(argp[4]);
    case 4: d=argp[1][0], a=atof(argp[2]); b=atof(argp[3]); break;
    default: error(string("Usage: ")+argp[0]+" d a b [g] "); break;
  }
  density_base* f = 0;
  switch(d) {
    case 'e': f = new exponential; break;
    case 'u': f = new uniform; break;
    case 'n': f = new normal; break;
    default: error("Error, d must be e or n or u");
  }
  cout << prob(f,a,b,g) << '\n';
  delete f;  // Never forget to delete the pointer
  return 0;
}
```

# Handle Classes

The use of pointers and operator new is inherently error prone and a common source of memory leaks. A handle class can get around this. A handle class encapsulates memory allocation and deallocation. The programmer is spared the details: the scope rules and rules for automatic will automatically take care that no memory leaks or other errors occur. The classes `realmat` and `intvec` are examples of this.

A handle class must have a copy constructor, an assignment operator, and a destructor to work properly. The programmer will have to write them because the compiler generated defaults will not work correctly for pointers to space allocated by new.

# Object Oriented Programming

- Motivate the ideas of OOP using Project Suggestion 11, which simulates the game of craps.
  - ▷ Play the game at `http://www.crapdice.com/crapgame.html`
  - ▷ Go through the project statement at course website
  - ▷ Go through the project design in `craps_base.h`.
- The code uses inheritance, virtual functions, and pure virtual functions to lay out and enforce a project design and to allow interchangeable components through polymorphism.
  - ▷ Bets are polymorphic so that they can be placed in lists of bets.
  - ▷ Strategies are polymorphic so that players at the same game can have different strategies.
  - ▷ A template function is used.
  - ▷ The list and vector container classes are used.

# MCMC Case Study

- Present MCMC case study slides.

- When finished continue with these slides and then

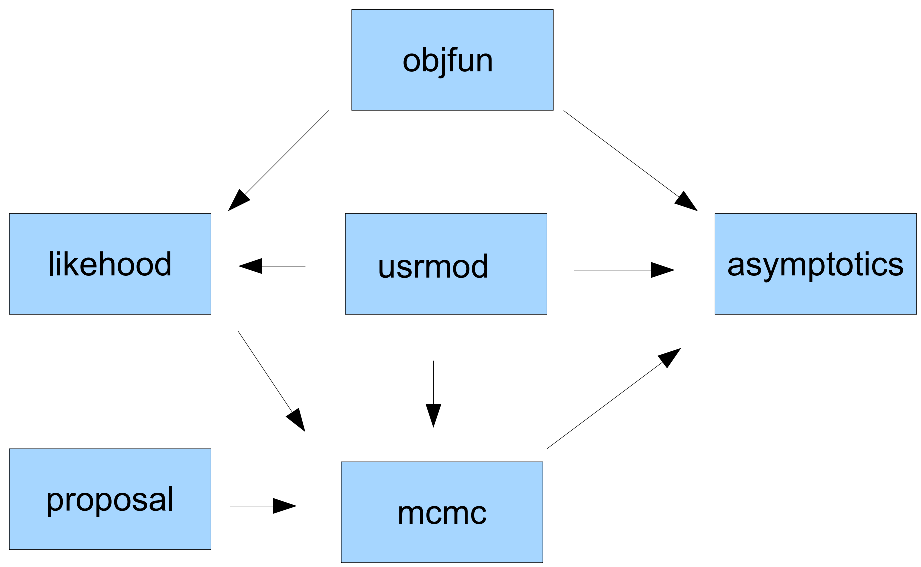
- Discuss the code in `habit_main.h`, `habit_main.h`, `asymptotics.h`.

# Inheritance and Virtual Functions

The case study uses inheritance, virtual functions, and pure virtual functions to lay out and enforce a project design and to allow interchangeable components through polymorphism.

URL: http://wwwr.aronaldg.org/webfiles/compecon/src/case/libmcmc/src/
Files: libmcmc_base.h mcmc.h

## Fig 1. MCMC Case Design




Arrows show which classes depend on which. The next three slides, titled `libmcmc_base.h`, present the intefaces that enforce the design. The fourth slide in the sequence, titled `mcmc.h`, presents `mcmc`, which inherits from the interface `mcmc_base`.

# libmcmc_base.h

```
class objfun_base {
private:
  void required() const
    { scl::error("Error, objfun_base, derived class must overide"); }
public:
  virtual void set_parms(const realmat& data) { required(); }
  virtual REAL operator()(const realmat& sim) const {required();return 0;}
  virtual     ~objfun_base() {};
  virtual objfun_base* new_objfun() { required(); return 0; }
  virtual void         delete_objfun(objfun_base*) { required(); }
};
```

# libmcmc_base.h

```
class usrmod_base {
public:
  virtual INTEGER len_theta() = 0;
  virtual INTEGER len_stats() = 0;
  virtual bool    gen_sim(realmat& sim) = 0;       //Same seed every call
  virtual bool    gen_sim(realmat& sim, realmat& stats) = 0;
  virtual bool    gen_bootstrap(realmat& sim) = 0; //Different seed
  virtual void    get_theta(realmat& theta) = 0;
  virtual void    set_theta(const realmat& theta) = 0;
  virtual bool    support(const realmat& theta) = 0;
  virtual den_val prior(const realmat& theta, const realmat& stats) = 0;
  virtual         ~usrmod_base() {};
};

class proposal_base {
public:
  virtual den_val operator()
                 (const realmat& th_old, const realmat& th_new)=0;
  virtual void    draw
                 (INT_32BIT& seed,const realmat& th_old,realmat& th_new)=0;
  virtual INTEGER len_theta()=0;
  virtual         ~proposal_base() {};
};
```

# libmcmc_base.h

```
class likehood_base {
public:
  virtual den_val operator()(const realmat& theta, realmat& stats) = 0;
  virtual INTEGER len_theta() = 0;
  virtual INTEGER len_stats() = 0;
  virtual         ~likehood_base() {};
};

class asymptotics_base {
public:
  virtual bool set_asymptotics(const realmat& chain) = 0;
  virtual void get_asymptotics(realmat& theta_hat, realmat& V_hat) = 0;
  virtual         ~asymptotics_base() {};
};

class mcmc_base {
public:
  virtual REAL    draw(INT_32BIT& seed, realmat& theta_start,
                    realmat& theta_chain, realmat& stats_chain,
                    realmat& pi_chain) = 0;
  virtual void    set_simulation_size(INTEGER n) = 0;
  virtual void    set_temp(REAL temperature) = 0;
  virtual REAL    get_temp() = 0;
  virtual realmat get_mode() = 0;
  virtual         ~mcmc_base() {};
};
```

# mcmc.h

```
class mcmc : public mcmc_base {
private:
  proposal_base& T;
  likehood_base& L;
  usrmod_base& U;
  INTEGER simulation_size;
  REAL    temp;
  realmat mode;
public:
          mcmc(proposal_base& T_fn, likehood_base& L_fn, usrmod_base& U_mod)
            : T(T_fn), L(L_fn), U(U_mod), simulation_size(1), temp(1.0) { }
  REAL    draw(INT_32BIT& seed, realmat& theta_start,
            realmat& theta_chain, realmat& stats_chain,
            realmat& pi_chain);
  void    set_simulation_size(INTEGER n);
  void    set_temp(REAL temperature);
  REAL    get_temp() { return temp; }
  realmat get_mode() { return mode; }
};
```

# Proposal

The proposal is a group move normal. A group is a subset of the parameters. They are moved together as a group according to a multivariate normal density. The vector `prop_def` contains classes that define these normals and is used to construct the group move proposal.

The proposal randomly selects one element of the vector and then draws from the normal described therein. If all groups have one element, then the proposal is actually a single move normal.

The details are not important other than to understand the reason for the way proposal in `habit_main.cpp` is constructed.

## habit_main.h

- A primitive user interface.
- Everything is known at compile time.
- Polymorphism is implemented with `typedefs`.
- Go through code.

# habit_main.cpp

- Constructs objects in the following order.
  - ▷ prop_def: def (default constructed, then initialized)
  - ▷ objfun: criterion (default constructed, then initialized from data)
  - ▷ usrmod: economy (default constructed)
  - ▷ likehood: generic_likelihood (constructed from economy, criterion)
  - ▷ proposal: selectable_proposal (constructed from def)
  - ▷ mcmc: metrop_hast (constucted from economy, selectable_proposal, generic_likehood followed by initializations)
  - ▷ asymptotics: summary (constructed from data, economy, criterion, metrop_hast).
- Runs MCMC chain, calls asymptotics, which is an accumulator for $\hat{\theta}$, $\mathcal{I}$, $\mathcal{J}$, and writes chain's output to files.
- Go through code.

# asymptotics.cpp

- Initializes a vector of objective functions, each constructed from a bootstrap sample.

- Numerically differentiates the vector of objective functions to get score vectors.

  ▷ Discuss numerical differentiation (next slide) here.

- Accumulates: mean $\theta$, $\mathcal{J}$, $\mathcal{I}$.

- Go through code.

# Two Sided Numerical Differentiation

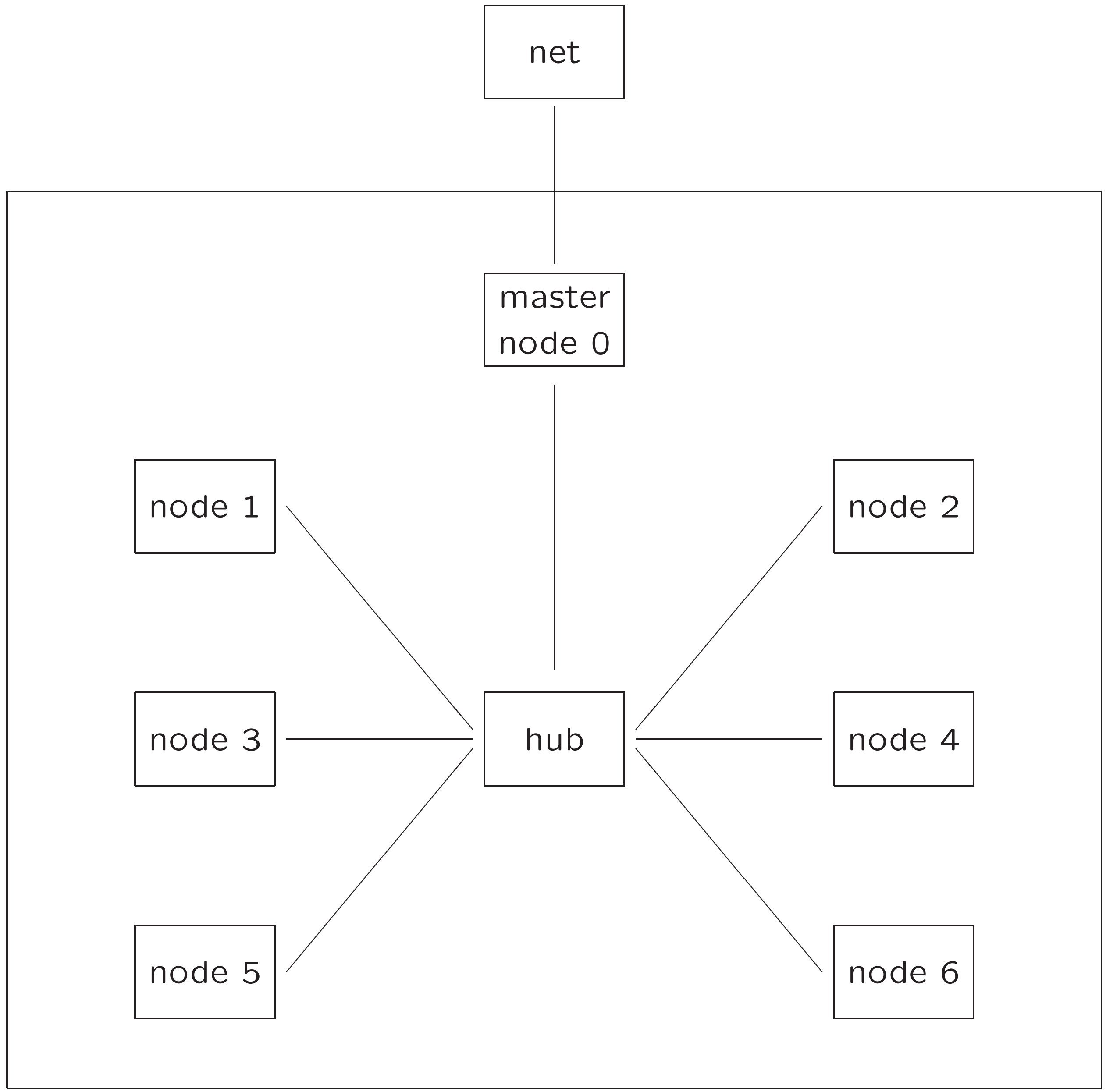
Reference: Press, William H., Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling (1993), *Numerical Recipes in C, The Art of Scientific Computing, Second Edition*, Cambridge University Press. See for one-sided and other formulas.

$$
\begin{aligned}
h &= \text{REAL\_EPSILON}^{(1.0/3.0)} \max(|x|, 1) \\
lo &= x - h \\
hi &= x + h \\
hh &= hi - lo \\
f'(x) &= \frac{f(hi) - f(lo)}{hh}
\end{aligned}
$$

$h$ minimizes the sum of bounds on roundoff and Taylor series truncation error. The odd way of computing $2h$ as $hh$ is to make sure that roundoff does not add additional errors. If REAL is double, then accuracy is about 11 digits. If $f$ is a noisy function, accuracy might not even be 1 digit. There are special methods for noisy functions.

# Parallel Computing: Overview

- Clusters.
  - ▷ Memory is not shared among all CPUs.
  - ▷ Communication is via message passing.
    - ◇ MPI is the industry standard and is portable.
  - ▷ Most common coding strategy is master/slave (aka. administrator/worker or leader/team) branches within a single program.
- Symmetric Multi-Processor (SMP) machine.
  - ▷ Memory is shared among all CPUs; cores count as CPUs.
  - ▷ MPI can be used.
  - ▷ Threads and OpenMP can be used.
- Graphics devices.
  - ▷ A graphics device is a massively parallel SMP machine.
  - ▷ Uses threads that are automatically launched by the device.

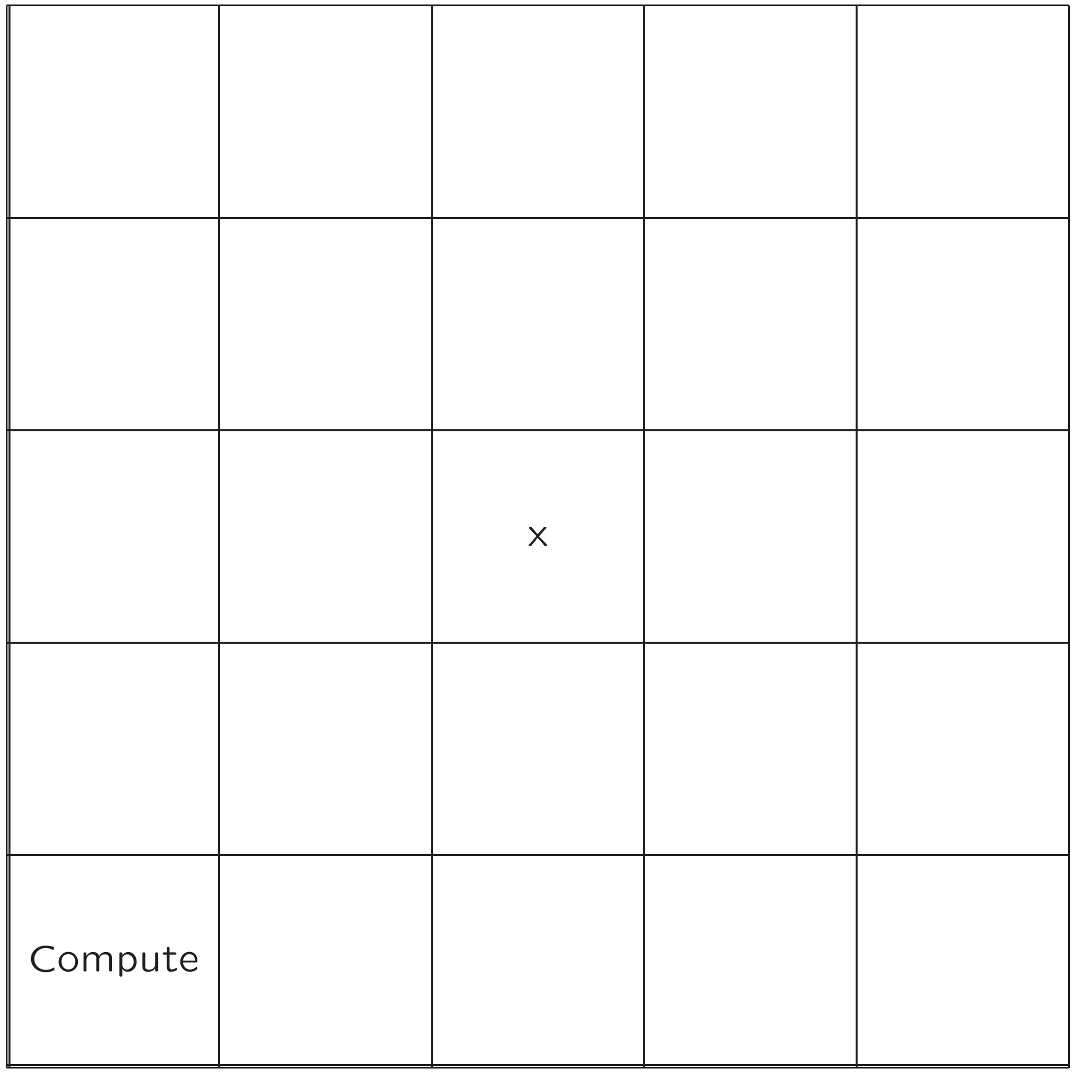



Typical small cluster configuration

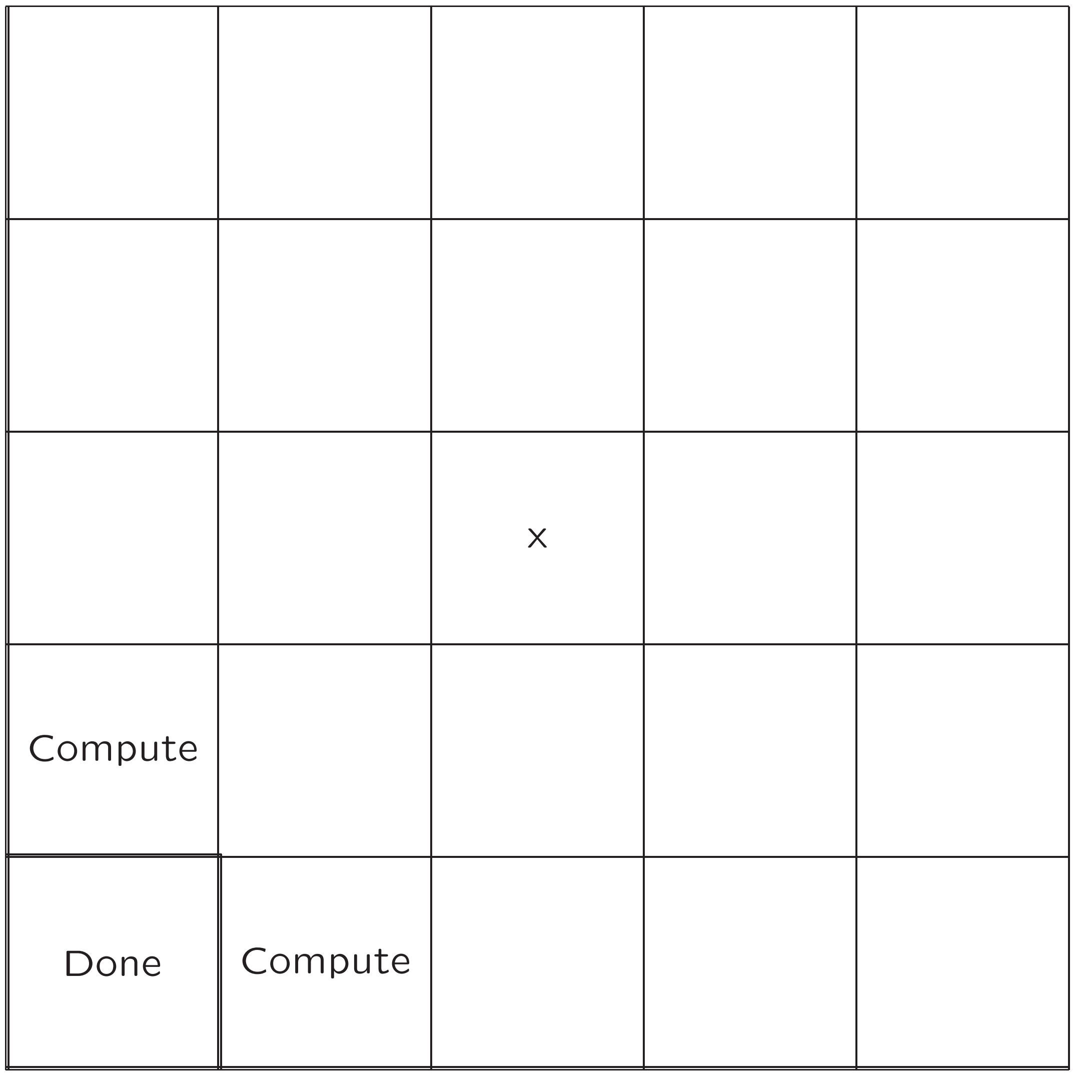

A serious cluster: NCAR's bluefire
IBM Power 575: 128 nodes, 32 CPUs per node
Each CPU is 4.7GHz, 4096 CPUs in total

# Classic Example

Solving a differential equation by dividing up the boundaries.



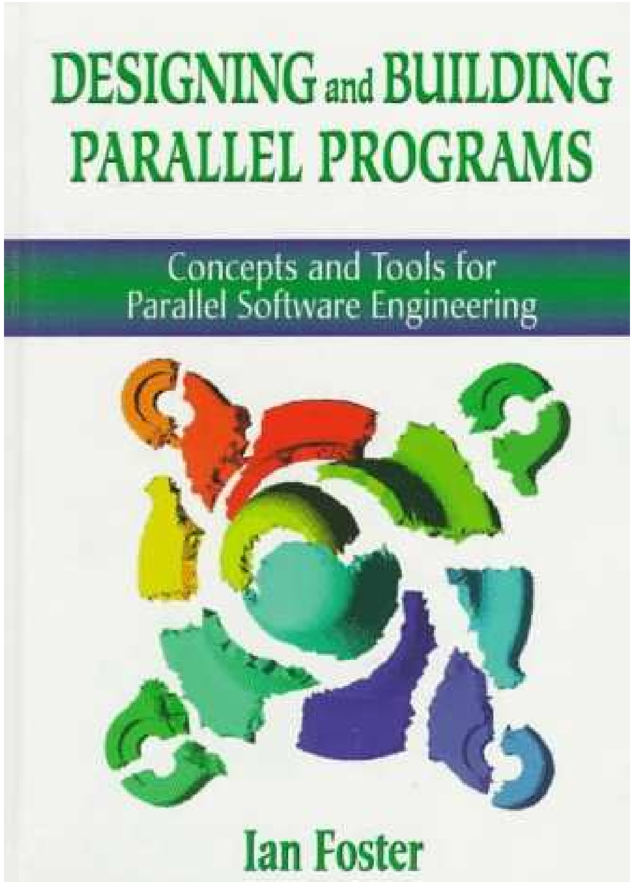

Step 1




Step 2

| | | | | |
|---|---|---|---|---|
| | | | | |
| | | | | |
| Compute | | x | | |
| Done | Compute | | | |
| Done | Done | Compute | | |

Step 3




Copyright: 1995
Publisher: Addison-Wesley Pub Co.
ISBN: 0-201-57594-9
Online at: http:://www.mcs.anl.gov/dbpp

# Coding Strategies

- *Shell Scripts.* Some programs, such as nonlinear optimizers that use multiple, random starts, are so embarrassingly parallelizable, that parallelization can be done with shell scripts alone.

- *Message Passing Interface (MPI).* The industry-standard protocol for implementing parallel processing. PVM is similar. Allows communication among processes running on different processors. Architecture independent: Code written for a cluster will run on multiple-processor, shared-memory machines. Mildly disruptive to serial code logic.
  - ▷ http://www.mpi-forum.org MPI reference
  - ▷ http://www.open-mpi.org software
  - ▷ http://ladon.iqfr.csic.es/docs/MPI_ug_in_FORTRAN.pdf Fortran
  - ▷ ftp://math.usfca.edu/pub/MPI/mpi.guide.ps C & C++

- *POSIX Threads (Pthreads).* Allows functions with the same name but different instances of the same argument to be run simultaneously. All functions have full access to memory and other machine resources. Can be disruptive to serial code logic and may require care to avoid simultaneous use of the same memory locations or other resources.
  - ▷ http://www.llnl.gov/computing/tutorials/pthreads

# Coding Strategies (Continued)

- *Parallelized Libraries.* Allows sequential code to have some of the benefits of parallelism. Works best on SMP machines. Can actually impede performance if coupled with MPI.
  - http://www.nag.co.uk/numberic/fd/FDdescription.asp
  - http://www.goguewave.com/products/imsl-numerical-libraries/c-library.aspx
- *High Performance Fortran.* A sort of hybrid of the strategies above, allows both threads and message passing. Worked poorly for us.
  - http://hpff.rice.edu
- *Open Multi-Processing (OpenMP)* Implements multiprocessing programming in C/C++ and Fortran on SMP machines. It is a set of compiler directives, library routines, and environment variables that influence run-time behavior. Least disruptive to existing serial code. Similar to threads; easier to code. Most compilers have it.
  - https://computing.llnl.gov/tutorials/openMP
  - http://www.openmp.org/mp-documents/spec30.pdf

# Coding Strategies (Continued)

- *Open Computing Language (OpenCL).* A language for programming GPU devices. Can be seriously disruptive to serial logic; especially when it forces dependencies among objects that would otherwise be independent. CUDA is similar and simpler but only works for Nvidia cards.
  - ▷ http://www.khronos.org
- *ViennaCL.* A scientific computing library that encapsulates OpenCL in the style of the C++ Standard Template Library. Hides all OpenCL unpleasantness from the user. Not disruptive to serial logic. By far the easiest way to use graphics devices.
  - ▷ http://viennacl.sourceforge.net
- *OpenACC.* Similar to OpenMP but for use with GPU devices. Uses compiler directives. Somewhat disruptive to serial logic due to having to code to compensate for slow GPU↔CPU memory copy. Not yet widely available; semi Nvidia proprietary.
  - ▷ http://http://www.openacc.org

# Introduction to MPI

- Pacheco, Peter S., A User's Guide to MPI (1995), Manuscript, Department of Mathematics, University of San Francisco.

  Online at course website.

- http://www.open-mpi.org/doc

  Online at course website.

# A Simple MPMD Coded as SPMD – 1

```
#include "libscl.h"
#include "mpi.h"

using namespace std;
using namespace scl;

namespace {

  int my_rank;          // Rank of process

  void mpi_error (string msg) {
    cout << msg << endl; MPI_Abort(MPI_COMM_WORLD, my_rank);
  }
  void mpi_warn (string msg) {
    cout << msg << endl; MPI_Abort(MPI_COMM_WORLD, my_rank);
  }

  const int buflen = 100; // These lines here so fits on two slides.
  char buffer[buflen];    // Buffer for messages
  int no_procs;           // Number of processes
  int tag = 50;           // Tag for messages
}
```

# A Simple MPMD Coded as SPMD – 2

```
int main(int argc, char *argp[], char *envp[])
{
  MPI_Init(&argc, &argp);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &no_procs);

  LIB_ERROR_HANDLER_PTR previous_error=set_lib_error_handler(&mpi_error);
  LIB_WARN_HANDLER_PTR previous_warn=set_lib_warn_handler(&mpi_warn);

  if (my_rank != 0) { // Slave
    sprintf(buffer, "\tGreetings from process %d \n", my_rank);
    int dest = 0;
    MPI_Send (buffer, buflen, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }
  else { // Master
    for (int source = 1; source < no_procs; ++source) {
      MPI_Status status;
      MPI_Recv(buffer,buflen,MPI_CHAR,source,tag,MPI_COMM_WORLD,&status);
      cout << buffer;
    }
  }

  MPI_Finalize();

  previous_error = set_lib_error_handler(previous_error);
  previous_warn = set_lib_warn_handler(previous_warn);
}
```

# MPI Makefile

```
CXX      = /usr/lib64/openmpi/1.4-gcc/bin/mpic++
SDIR     = ./
IMPI     = /usr/lib64/openmpi/1.4-gcc/include
LMPI     = /usr/lib64/openmpi/1.4-gcc/lib
ISCL     = $(HOME)/lib/libscl/gpp
LSCL     = $(HOME)/lib/libscl/gpp
IDIRS    = -I. -I$(SDIR) -I$(IMPI) -I$(ISCL)
LDIRS    = -L$(LMPI) -L$(LSCL)
CXXFLAGS = -O2 -Wall -c  $(IDIRS)
LDFLAGS  = $(LDIRS)  -lscl -lm

hello : hello.o
$(CXX) -o hello hello.o $(LDFLAGS)

hello.o : $(SDIR)/hello.cpp
$(CXX) $(CXXFLAGS) $(SDIR)/hello.cpp

clean :
rm -f *.o

veryclean :
rm -f *.o
rm -f  hello
```

# MPI Shell Script

```
#! /bin/sh

# Some 64 bit machines need this:
export PATH="$PATH:/usr/lib64/openmpi/1.4-gcc/bin/"
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/lib64/openmpi/1.4-gcc/lib"
export C_INCLUDE_PATH="$C_INCLUDE_PATH:/usr/lib64/openmpi/1.4-gcc/include"

echo "localhost cpu=48" > OpenMPIhosts

test -f hello.err  && mv -f hello.err  hello.err.bak
test -f hello.out  && mv -f hello.out  hello.out.bak

make -f makefile.mpi.OpenMPI_1.4 >hello.out 2>&1 && \
  mpirun --hostfile OpenMPIhosts ${PWD}/hello >>hello.out 2>hello.err

RC=$?

case $RC in
  0) exit 0 ;;
  esac
exit 1;
```

# Examples

- Compile and run hello on argux6

- Discuss case study: habit_main_mpi.h and habit_main_mpi.cpp

# MPI_Init and MPI_Finalize

```
#include "mpi.h"

main(int argc, char** argp)
{
  //...

  //No MPI functions called before this
  //Passing these args allows system setup
  MPI_Init(&argc, &argp);

  //...

  MPI_Finalize();
  //No MPI functions called after this

  //...
}
```

# MPI_Comm_rank and MPI_Comm_size

```
int MPI_Comm_rank(MPI_Comm comm, int* rank);
int MPI_Comm_size(MPI_Comm comm, int* size);
```

comm, a communicator or collection of processes that can send messages to each other. For simple programs the predefined MPI_COMM_WORLD is enough. Usually a library would have its own communicator.

rank is the processor's number: 0 is master node.
size is the number of processors in a communicator

## MPI_Send and MPI_Receive

```
int MPI_Send(void* buffer, int count,
       MPI_Datatype datatype,
       int dest, int tag, MPI_Comm comm)
int MPI_Receive(void* buffer, int count,
       MPI_Datatype datatype,
       int source, int tag, MPI_Comm comm,
       MPI_Status* status)
```

message is a buffer of length count of type MPI_datatype. datatypes are MPI_CHAR (signed char), MPI_INT (signed int), MPI_DOUBLE (double), and some others. dest is the rank of the recipient, source is the rank of the sender (MPI_ANY_SOURCE allowed), tag is a user specified message identifier (MPI_ANY_TAG allowed), comm is the communicator; status.source gives rank of sender; status.tag gives tag of message to allow resolution of wildcards MPI_ANY_SOURCE and MPI_ANY_TAG.

## MPI_Bcast

```
int MPI_Bcast(void* buffer, int count,
       MPI_Datatype datatype,
       int root, MPI_Comm comm)
```

message is a buffer of length count of type MPI_datatype. root is the rank of the sender, comm is the communicator. Sends the same message to every process in communicator comm. All processes in comm must call MPI_Bcast and must have root and comm the same. MPI_Recv cannot be used to receive a broadcast message.

# Input/Output

The only process that can read or write a file is the root (the process with rank=0). What happens if another process tries to is indeterminate.

# Posix Threads (Pthreads)

A threaded process has one or more processes within it that can run independently and simultaneously.

A thread is one of these (sub) processes.

Reference: POSIX Threads Programming
http://www.llnl.gov/computing/tutorials/pthreads/

# Pthread Warning

The material on pthreads is for an older C++ language standard such as the C++ compiler on argux6. For the C++11 standard, such as clang++ on a Mac, the code must be modified. See, e.g., compecon/src/hello_pthread_clang. For the C++11 encapsulation of pthreads in a thread class see

https://solarianprogrammer.com/2011/12/16/cpp-11-thread-tutorial

# Thread Properties

- The main program is a thread.

- A thread exists within the process that creates it and uses that process's resources.

- A thread has its own independent flow of control.

- A thread has its own stack and registers.

- A thread shares memory and files with the process that creates it and with all other threads.

# Consequences of Thread Properties

- Changes made by one thread to shared resources, such as closing a file, will be seen by all other threads.

- Two pointers having the same value point to the same data.

- Reading and writing to the same memory locations is possible and therefore requires explicit synchronization by the programmer.

# A Simple Threaded Program – 1 of 2

```
// Compile with g++ -pthread in LDFLAGS

#include "libscl.h"
#include <pthread.h>   // Header for pthread
#include <unistd.h>    // Header for sysconf

namespace {

  struct arg_type {
    INTEGER threadid;
    std::string message;
  };

  void* write_arg(void* arg_ptr)
  {
    arg_type* arg = (arg_type*)(arg_ptr);
    arg->message += scl::fmt('d',2,arg->threadid)() + "\n";
    std::cout << arg->message;
    pthread_exit(NULL);
  }

}
```

# A Simple Threaded Program – 2 of 2

```
int main(int argc, char** argp, char** envp)
{
  #if defined _SC_NPROCESSORS_ONLN
    INTEGER num_threads = sysconf(_SC_NPROCESSORS_ONLN);
  #else
    INTEGER num_threads = 2;
    std::cerr << "The variable _SC_NPROCESSORS_ONLN is not defined, using "
              << num_threads << " threads instead\n";
  #endif

  pthread_t threads[num_threads];
  arg_type  args[num_threads];
  int rc, t;
  for(t=0; t<num_threads; t++){
     args[t].threadid = t;
     args[t].message = "Hello from thread number ";
     rc = pthread_create(&threads[t], NULL, write_arg, (void*)(&args[t]));
     if (rc) scl::error("Cannot create thread");
  }
  pthread_exit(NULL);
}
```

# Makefile

```
CC       = g++
SDIR     = .
IDIR     = $(HOME)/lib/libscl/gpp
LDIR     = $(HOME)/lib/libscl/gpp
CFLAGS   = -O2 -Wall -c -I$(SDIR) -I$(IDIR)
LFLAGS   = -pthread -lm -L$(LDIR) -lscl

hello : hello.o
$(CC) -o hello hello.o $(LFLAGS)

hello.o : $(SDIR)/hello.cpp
$(CC) $(CFLAGS) $(SDIR)/hello.cpp

clean :
rm -f *.o
rm -f core core.*

veryclean :
rm -f *.o
rm -f core core.*
rm -f hello
```

# Function Naming Conventions

| Function Prefix | Functionality |
| --- | --- |
| `pthread_` | Threads themselves and thread management |
| `pthread_attr_` | Thread attributes objects |
| `pthread_mutex_` | Thread synchronization, **mut**ual **exc**lusion |
| `pthread_mutexattr_` | Mutex attributes objects |
| `pthread_cond_` | Condition variables |
| `pthread_condattr_` | Condition attributes objects |
| `pthread_key_` | Thread data keys |

# pthread_create

```
int  pthread_create(pthread_t* thread,
       const pthread_attr_t* attr,
       void* (*start_routine)(void*), void* arg)
```

`pthread_create` creates a new thread that executes `start_routine`, which is a pointer to a function that takes a single `void*` argument and returns a `void*`. `arg` is the argument passed to `pthread_create`. To pass more than one argument, one must define a `struct`, convert it to `void*`, and then convert it back to the `struct` within `start_routine`. `NULL` may be used if no argument is to be passed. `attr` is an object that defines attributes of the thread such as stacksize; `NULL` may be used to specify the default attributes. `thread` points to where the thread id is stored on return. `pthread_create` returns zero on success.

# pthread_exit

```
void pthread_exit (void* value_ptr)
```

`pthread_exit` terminates a thread's execution. The main program is a thread that must also be terminated with `pthread_exit`. If one fails to do this, all threads will be terminated when the main terminates even if their work is not finished. `value_ptr` is a termination status for use by a thread that joins the calling thread; `NULL` may be used. `pthread_exit` does not destroy files created by the thread. Any file opened and used only by the thread must be explicitly destroyed before calling `pthread_exit`.

# Managing Attributes

```
int pthread_attr_init(pthread_attr_t* attr)
int pthread_attr_destroy(pthread_attr_t* attr)
```

`pthread_attr_init` allocates space for an attribute object, initializes it to default values, and puts its address in the location pointed to by attr. `pthread_attr_destroy` destroys the object and releases the space assigned to it. Both return zero on success. Other routines described later are used to query the attributes and set them. `pthread_attr_destroy` has no effect on the threads previously created with the destroyed object.

# Joining Threads – 1 of 2

```
int pthread_attr_setdetachstate
    (pthread_attr_t* attr, int detachstate)
int pthread_attr_getdetachstate
    (const pthread_attr_t* attr, int* detachstate)
```

Joining is one way to set up a master/slave relationship among threads. Joining makes the joined thread wait on the termination of other threads. A thread created with detach state set to `PTHREAD_CREATE_DETACHED` cannot be waited upon. A thread created with detach state set to `PTHREAD_CREATE_JOINABLE` can be waited upon. The latter is supposed to be the default but not all implementations conform. Set it yourself to be safe.

# Joining Threads – 2 of 2

```
int pthread_join(pthread_t thread, void** status)
int pthread_detach(pthread_t thread)
```

The first argument is the thread to wait for, which is called the target thread. The `pthread_join` function makes the thread that calls `pthread_join` wait until the target thread terminates. The target thread's termination status is returned in the status parameter. If the target thread is already terminated, but not yet detached, `pthread_join` returns immediately. It is impossible to join a detached thread, even if it is not yet terminated. The target thread is automatically detached after all joined threads have been woken up. `pthread_detach` can be used to explicitly detach a thread even though it was created as joinable. There is no converse routine. Both functions return zero on success.

# Pages and Caches

- Due to the way memory is managed by the operating system (paging) and the CPU (caching), one should write programs so that data being accessed are in contiguous areas of memory to the greatest extent possible.
- In particular one should access the elements of an array in the order in which they are stored.
- The elements of a `realmat` are ordered columnwise in memory; i.e. the vec of the matrix is stored. Therefore, in a double `for` loop accessing elements `A(i,j)` of `realmat A`, the index `i` should be fastest moving and the index `j` slowest moving.
- The followng matrix multiplication example ($R = A*B$) showing how to join threads reflects these memory management principles.

# A Joined Threads Program – 1 of 3

```
#include "libscl.h"
#include <pthread.h>   // Header for pthread

namespace {

  struct arg_type {
    INTEGER col;
    realmat* r_ptr;
    const realmat* a_ptr;
    const realmat* b_ptr;
    arg_type() { }
    arg_type(INTEGER j,realmat* rp,const realmat* ap,const realmat* bp)
      : col(j), r_ptr(rp), a_ptr(ap), b_ptr(bp) { }
  };

  void* mult(void* arg_ptr)  {
    arg_type* arg = (arg_type*)(arg_ptr);
    INTEGER j = arg->col;
    for (INTEGER k=1; k<=(*(arg->a_ptr)).ncol(); ++k) {
      REAL b_kj = (*(arg->b_ptr))(k,j);
      for (INTEGER i=1; i<=(*(arg->a_ptr)).nrow(); ++i) {
        (*(arg->r_ptr))(i,j) += b_kj * (*(arg->a_ptr))(i,k);
      }
    }
    pthread_exit((void*) 0);
  }
}
```

# A Joined Threads Program – 2 of 3

```
int main (int argc, char** argp)
{
  realmat a,b;
  if(!vecread("a.dat",a) || !vecread("b.dat",b)) error("Read failed");
  if (a.ncol() != b.nrow()) error("Not conformable");

  realmat r(a.nrow(),b.ncol(),0.0);

  INTEGER num_threads = b.ncol();

  pthread_t threads[num_threads];
  arg_type  args[num_threads];

  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);

  for (INTEGER t=0; t<num_threads; ++t) {
    args[t] = arg_type(t+1,&r,&a,&b);
    int rc = pthread_create(&threads[t],&attr,&mult,(void*)(&args[t]));
    if (rc) error("Cannot create threads");
  }
```

# A Joined Threads Program – 3 of 3

```
  pthread_attr_destroy(&attr);

  void* status;
  for (INTEGER t=0; t<num_threads; ++t) {
    int rc = pthread_join(threads[t], &status);
    if (rc) error("Cannot join threads");
  }

  std::cout << a << b << r << '\n';

  pthread_exit(NULL);
}
```

# Stack Management

```
int pthread_attr_getstacksize
      (const pthread_attr_t* attr, size_t* stacksize)
int pthread_attr_setstacksize
      (pthread_attr_t *attr, size_t stacksize)
```

Stacksize is implementation dependent and exceeding the default stack limit is easy to do. Exceeding it causes program termination and/or corrupted data. Safe and portable programs do not depend upon the default stack limit, but instead, explicitly allocate enough stack for each thread by using the `pthread_attr_setstacksize` routine. Both functions return zero on success. It is hard to determine how much stack is needed. If the thread has an array like `double a[N*M]` then the stack should be increased from the default by at least `sizeof(double)*N*M`. Allocating on the heap instead of the stack is a way around this.

# Miscellaneous Routines

```
pthread_t pthread_self(void)
int pthread_equal(pthread_t t1, pthread_t t2)
int pthread_once
      (pthread_once_t* once_control, void (*init_routine)(void))
void pthread_yield(void)
```

`pthread_self` returns the ID of the calling thread. `pthread_equal` returns a non-zero value if t1 and t2 refer to the same thread, or zero if they do not. `pthread_once` executes `init_routine` once. The first call by any thread executes `init_routine`, without parameters. Any subsequent call will have no effect. `init_routine` is typically an initialization routine. The `once_control` parameter is a synchronization control structure that requires initialization prior to calling `pthread_once`: `pthread_once_t once_control = PTHREAD_ONCE_INIT;`. It determines whether the associated initialization routine has been called. The function `pthread_yield` forces the calling thread to relinquish use of its processor, and to wait in the run queue before it is scheduled again.

# Mutex Variables

Mutex is an abbreviation for "mutual exclusion". Mutex variables are a another way to implement thread synchronization and a way to protect shared data when multiple writes occur. Usually the action performed by a thread owning a mutex is the updating of global variables.

When several threads compete for a mutex, the losers block at that call – an unblocking call is available with "trylock" instead of the "lock" call.

When protecting shared data, it is the programmer's responsibility to make sure every thread that needs to use a mutex does so. For example, if four threads are updating the same data, but only one uses a mutex, the data can still be corrupted.

# Mutex Coding

A typical sequence in the use of a mutex is as follows:

- Create and initialize a mutex variable
- Several threads attempt to lock the mutex
- Only one succeeds and that thread owns the mutex
- The owner thread performs some set of actions
- The owner unlocks the mutex
- Another thread acquires the mutex and repeats the process
- Finally the mutex is destroyed

# Creating and Destroying Mutexes

```
int pthread_mutex_init
      (pthread_mutex_t* mutex, const pthread_mutexattr_t* attr)
int pthread_mutex_destroy(pthread_mutex_t* mutex)
int pthread_mutexattr_init(pthread_mutexattr_t* attr)
int pthread_mutexattr_destroy(pthread_mutexattr_t* attr)
```

Mutex variables must be declared with type `pthread_mutex_t` and must be initialized before they can be used. There are two ways to initialize a mutex variable: statically, when it is declared, e.g. `pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;` or dynamically, using `pthread_mutex_init()` to set attributes as specified by the mutex attibutes object `attr`, which may specified as `NULL` to accept defaults. The mutex is initially unlocked.

`pthread_mutexattr_init` and `pthread_mutexattr_destroy` are used to create and destroy mutex attribute objects. There are three mutex attributes: (1) Protocol: Specifies the protocol used to prevent priority inversions for a mutex. (2) Prioceiling: Specifies the priority ceiling of a mutex. (3) Process-shared: Specifies the process sharing of a mutex. (4) Type: Enables deadlock detection. The defaults are usually adequate so how to set them is not discussed and not all implementations provide all mutex attributes anyway.

# Locking and Unlocking Mutexes

```
int pthread_mutex_lock(pthread_mutex_t* mutex)
int pthread_mutex_trylock(pthread_mutex_t* mutex)
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

`pthread_mutex_lock` locks the mutex. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex in the locked state with the calling thread as its owner. If a thread attempts to relock a mutex that it has already locked a disaster called deadlock occurs.

`pthread_mutex_trylock` tries to lock the specified mutex. If the mutex is already locked, an error is returned. Otherwise, this operation returns with the mutex in the locked state with the calling thread as its owner.

`pthread_mutex_unlock` attempts to unlock the specified mutex. If there are threads blocked on the mutex object when `pthread_mutex_unlock` is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread acquires the mutex.

# A Mutex Program – 1 of 3

```
#include "libscl.h"
#include <pthread.h>   // Header for pthread
#include <unistd.h>    // Header for sysconf
using namespace scl;

namespace {

  realmat a;
  REAL fnorm;
  pthread_mutex_t mutexsum;

  void* mult(void* arg)
  {
     INTEGER* jptr = (INTEGER*)(arg);
     INTEGER n = a.nrow();
     REAL* t = &a(1,*jptr);
     REAL* top = t + n;
     REAL sum = 0.0;
     while (t<top) sum += pow(*t++,2);
     pthread_mutex_lock(&mutexsum);
     fnorm += sum;
     pthread_mutex_unlock(&mutexsum);
     pthread_exit(NULL);
  }
}
```

# A Mutex Program – 2 of 3

```
int main (int argc, char** argp)
{
  if(!vecread("a.dat",a)) error("Read failed");

  INTEGER num_threads = a.ncol();

  pthread_t threads[num_threads];

  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);

  pthread_mutex_init(&mutexsum, NULL);

  INTEGER arg[num_threads];

  fnorm = 0.0;

  for (INTEGER t=0; t<num_threads; ++t) {
    arg[t] = t+1;
    int rc = pthread_create(&threads[t], &attr, &mult, (void*)(&arg[t]));
    if (rc) error("Cannot create threads");
  }

  pthread_attr_destroy(&attr);
```

# A Mutex Program – 3 of 3

```
  void* status;
  for (INTEGER t=0; t<num_threads; ++t) {
    int rc = pthread_join(threads[t], &status);
    if (rc) error("Cannot join threads");
  }

  fnorm = sqrt(fnorm);

  std::cout<<"The Frobenius norm of"<< a <<"\nis "<< fnorm <<'\n';

  pthread_mutex_destroy(&mutexsum);
  pthread_exit(NULL);
}
```

# Condition Variables

Condition variables provide yet another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.

Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.

A condition variable is always used in conjunction with a mutex lock.

# Condition Variable Coding – 1 of 4

Main Thread

- Declare and initialize global data/variables which require synchronization (such as "count")

- Declare and initialize a condition variable object

- Declare and initialize an associated mutex

- Create threads A and B to do work

# Condition Variable Coding – 2 of 4

Thread A

- Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
- Lock associated mutex and check value of a global variable
- Call `pthread_cond_wait` to perform a blocking wait for signal from Thread B. Note that a call to `pthread_cond_wait` automatically and atomically unlocks the associated mutex variable so that it can be used by Thread B.
- When signalled, wake up. Mutex is automatically and atomically locked.
- Explicitly unlock mutex
- Continue

# Condition Variable Coding – 3 of 4

Thread A

- Do work
- Lock associated mutex
- Change the value of the global variable that Thread A is waiting upon.
- Check value of the global Thread A wait variable. If it fulfills the desired condition, signal Thread A.
- Unlock mutex.
- Continue

# Condition Variable Coding – 4 of 4

Main Thread

- Join
- Continue

# Creating and Destroying Condition Variables

```
int pthread_cond_init
      (pthread_cond_t* cond, const pthread_condattr_t* attr)
int pthread_cond_destroy(pthread_cond_t* cond)
int pthread_condattr_init(pthread_condattr_t* attr)
int pthread_condattr_destroy(pthread_condattr_t* attr)
```

Condition variables must be declared with type `pthread_cond_t`, and must be initialized before they can be used. There are two ways to initialize a condition variable: Statically, when it is declared; i.e. `pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;` Dynamically, with `pthread_cond_init`. The ID of the created condition variable is returned to the calling thread through the condition parameter. This method permits setting condition variable object attributes, `attr`.

The optional `attr` object is used to set condition variable attributes. There is only one attribute defined for condition variables: `process-shared`, which allows the condition variable to be seen by threads in other processes. The attribute object, if used, must be of type `pthread_condattr_t`, which may be specified as `NULL` to accept defaults.

Note that not all implementations may provide the `process-shared` attribute.

`pthread_condattr_init` and `pthread_condattr_destroy` create and destroy condition variable attribute objects.

# Waiting and Signaling on Condition Variables

```
int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex)
int pthread_cond_signal(pthread_cond_t* cond)
int pthread_cond_broadcast(pthread_cond_t* cond)
```

`pthread_cond_wait` blocks the calling thread until the specified condition is signalled. This routine should be called while mutex is locked, and it will automatically release the mutex while it waits. After signal is received and thread is awakened, mutex will be automatically locked for use by the thread. The programmer is then responsible for unlocking mutex when the thread is finished with it.

`pthread_cond_signal` signals (or wakes up) another thread which is waiting on the condition variable. It should be called after mutex is locked, and must unlock mutex in order for `pthread_cond_wait` routine to complete.

`pthread_cond_broadcast` should be used instead of `pthread_cond_signal` if more than one thread is in a blocking wait state.

It is a logical error to call `pthread_cond_signal` before calling `pthread_cond_wait`.

Proper locking and unlocking of the associated mutex variable is essential when using these routines. For example: Failing to lock the mutex before calling `pthread_cond_wait` may cause it noT to block. Failing to unlock the mutex after calling `pthread_cond_signal` may not allow a matching `pthread_cond_wait` routine to complete (it will remain blocked).

# OpenMP

The references for the slides that follow are

- http://en.wikipedia.org/wiki/OpenMP
- https://computing.llnl.gov/tutorials/openMP
- https://computing.llnl.gov/tutorials/openMP/exercise.html
- http://www.openmp.org/mp-documents/spec30.pdf
- http://www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf

# OpenMP

- SPMD multithreading master/slave parallelization for SMP machines.

- The code that runs in parallel is marked with a preprocessor directive, i.e. a pragma.

- Pragmas are controlled by data sharing, synchronization, and scheduling clauses.

- Library functions provide environment information

- After the execution of the parallelized code, the threads "join" back into the master thread.

# Plan

- We will look at these pragmas.
  - pragma omp parallel
  - pragma omp for
  - pragma omp critical
- these clauses
  - private
  - shared
  - default
  - schedule
- and these functions
  - omp_get_thread_num()
  - omp_get_num_threads()
  - omp_in_parallel()
- There are others.

# First Some Examples

We will look at the three examples we used for pthreads

- Hello world
- Matrix multiply
- Frobenius norm

# Makefile

```
CXX      = g++
SDIR     = .
ISCl     = $(HOME)/lib/libscl/gpp
LSCl     = $(HOME)/lib/libscl/gpp
CXXFLAGS = -fopenmp -O2 -Wall -c -I$(SDIR) -I$(ISCl)
LDFLAGS  = -fopenmp -lm -L$(LSCl) -lscl

all : hello mult frobnorm

hello : hello.o
	$(CXX) -o hello hello.o $(LDFLAGS)

hello.o : $(SDIR)/hello.cpp
	$(CXX) $(CXXFLAGS) $(SDIR)/hello.cpp

mult : mult.o
	$(CXX) -o mult mult.o $(LDFLAGS)

mult.o : $(SDIR)/mult.cpp
	$(CXX) $(CXXFLAGS) $(SDIR)/mult.cpp

frobnorm : frobnorm.o
	$(CXX) -o frobnorm frobnorm.o $(LDFLAGS)

frobnorm.o : $(SDIR)/frobnorm.cpp
	$(CXX) $(CXXFLAGS) $(SDIR)/frobnorm.cpp
```

# Hello World

```
// Compile with g++ -fopenmp flag in both CXXFLAGS & LDFLAGS
#include "libscl.h"
#include <omp.h>
using namespace std; using namespace scl;

int main(int argc, char** argp, char** envp)
{
  INTEGER tid, nthreads;
  string msg;
  #pragma omp parallel private(nthreads, tid, msg)
  {
    tid = omp_get_thread_num();
    msg = "Hello from thread =" + fmt('d',3,tid)() + "\n";
    cout << msg;
    if (tid == 0) {
      bool inpar = omp_in_parallel();
      msg = "\n";
      if (inpar) msg += "Code block running in parallel\n";
      else msg += "Code block running serial\n";
      nthreads = omp_get_num_threads();
      msg += "Number of threads =" + fmt('d',3,nthreads)();
      msg += "\n\n";
      cout << msg;
    }
  }
  return 0;
}
```

# Matrix Multiply – 1

```
#include "libscl.h"
#include <omp.h>

using namespace std;
using namespace scl;

int main(int argc, char** argp, char** envp)
{
  realmat a, b;
  if(!vecread("a.dat",a) || !vecread("b.dat",b)) error("Read failed");
  if (a.ncol() != b.nrow()) error("Not conformable");

  const INTEGER arows = a.nrow();
  const INTEGER acols = b.nrow();
  const INTEGER bcols = b.ncol();
```

# Matrix Multiply – 2

```
  realmat r(arows,bcols,0.0);

  INTEGER chunk = 2;
  INTEGER tid;

  string msg;

  #pragma omp parallel shared(chunk) private(tid, msg)
  {
    tid = omp_get_thread_num();
    #pragma omp for schedule(static, chunk)
    for (INTEGER j=1; j<=bcols; ++j) {
      msg = "Thread" + fmt('d',3,tid)();
      msg += " did col" + fmt('d',3,j)() + "\n";
      cout << msg;
      for (INTEGER k=1; k<=acols; ++k) {
        for (INTEGER i=1; i<=arows; ++i) {
          r(i,j) += a(i,k)*b(k,j);
        }
      }
    }
  }

  std::cout << a << b << r << '\n';

  return 0;
}
```

# Frobenius Norm – 1

```
#include "libscl.h"
#include <omp.h>

using namespace scl;
using namespace std;

int main (int argc, char** argp, char** envp)
{
  realmat a;
  if(!vecread("a.dat",a)) error("Read failed");

  a = T(a);

  cout << a << '\n';

  INTEGER arows = a.nrow();
  INTEGER acols = a.ncol();

  string msg;
```

# Frobenius Norm – 2

```
  INTEGER chunk = 2;
  INTEGER tid;

  REAL fnorm = 0.0;
  REAL sum;

  #pragma omp parallel shared(chunk, fnorm) private(tid, msg, sum)
  {
    tid = omp_get_thread_num();
    #pragma omp for schedule (static, chunk)
    for (INTEGER j=1; j<=acols; ++j) {
      msg = "Thread" + fmt('d',3,tid)();
      msg += " did col" + fmt('d',3,j)() + "\n";
      cout << msg;
      sum = 0.0;
      for (INTEGER i=1; i<=arows; ++i) {
        sum += pow(a(i,j),2);
      }
      #pragma omp critical
      { fnorm += sum; }
    }
  }
  fnorm = sqrt(fnorm);
  cout << "\nFrobenius norm = " << fnorm << '\n';
  return 0;
}
```

# Parallel Pragma

```
#pragma parallel [clause[ [, ]clause] ...] new-line
structured-block
clause: private(list)
        shared(list)
        default(shared | none)
```

The parallel pragma forms a team of threads and starts parallel execution of a structured-block, which is a block of code enclosed in braces. No jumps into or out of a structured block are allowed. Within a parallel region, thread numbers uniquely identify each thread. There is an implied barrier at the end of a parallel region. After the end of a parallel region, only the master thread of the team resumes execution. If execution of a thread terminates while inside a parallel region, execution of all threads in all teams terminates. Variables in the private (comma separated) list are local to each thread. Variables in the shared list are global to all threads. Private and shared clauses override a default clause. Using `default(none)` forces explicit declaration of all variables in the structured-block as private or shared. A backslash is used to continue a pragma line.

# For Pragma

```
#pragma omp for [clause[ [, ]clause] ...] new-line
structured-block
clause: shared(list)
        private(list)
        schedule(kind[, chunk_size])
```

The for pragma must appear within the structured block of a parallel pragma. It specifies that the iterations of one or more associated loops are distributed across threads that already exist in the team executing the parallel pragma. The static form of the schedule clause is `schedule(static,chunk)` where `chunk` is an integer. `static` specifies allocation of iterations to threads prior to execution of threads in a team, is the default, and may be all that is available with some compilers. `chunk` specifies how many iterations are allocated to each thread. It must be smaller than the limit of the outermost loop index or one thread will do all iterations. All loop limits must be loop invariant. No default clause; private and shared as above. If available, use `schedule(dynamic)`

# Critical Pragma

```
#pragma omp critical [(name)] new-line
structured-block
```

The critical construct restricts execution of the associated structured block to a single thread at a time, where name can optionally be used to identify the critical region. Identifiers naming a critical region have external linkage and occupy a namespace distinct from that used by ordinary identifiers.

A thread waits at the start of a critical region identified by a given name until no other thread in the program is executing a critical region with that same name. Critical sections not specifically named by omp critical directive invocation are mapped to the same unspecified name.

A typical example is

```
#pragma omp critical
{ sum += x; }
```

which makes sure that only one thread at a time updates `sum`.

# Advantages of OpenMP

- Simple: Need not deal with message passing as MPI does
- Data layout and decomposition is handled automatically by directives.
- Incremental parallelism: Can work on one portion of the program at a time, no dramatic change to code is needed.
- Unified code for both serial and parallel applications: OpenMP constructs are treated as comments when sequential compilers are used.
- Original (serial) code statements need not, in general, be modified when parallelized with OpenMP. This reduces the chance of inadvertently introducing bugs.

# Disadvantages of OpenMP

- Risk of introducing difficult to debug synchronization bugs and race conditions
  - ▷ A race condition is two or more threads trying to write to the same memory location simultaneously
- Only runs in shared-memory multiprocessor platforms
- Requires a compiler that supports OpenMP.
- Scalability is limited by memory architecture.
- Reliable error handling is missing.
- Can't be used on GPUs

# Other Directives, Clauses, etc.

Review

http://www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf

# Graphics Devices


## Tesla C1060 Computing Processor

NVIDIA® Tesla™ C1060 Computing Processor enables the transition to energy efficient parallel computing power by bringing the performance of a small cluster to a workstation.

Now Available!

- Buy now online or from a system builder
- Buy Tesla Personal Supercomputer

Print page

| | |
|---|---|
| Form Factor | 10.5" x 4.376", Dual Slot |
| # of Tesla GPUs | 1 |
| # of Streaming Processor Cores | 240 |
| Frequency of processor cores | 1.3 GHz |
| Single Precision floating point performance (peak) | 933 |
| Double Precision floating point performance (peak) | 78 |
| Floating Point Precision | IEEE 754 single & double |
| Total Dedicated Memory | 4 GB GDDR3 |
| Memory Speed | 800MHz |
| Memory Interface | 512-bit |
| Memory Bandwidth | 102 GB/sec |
| Max Power Consumption | 187.8 W |
| System Interface | PCIe x16 |
| Auxiliary Power Connectors | 6-pin & 8-pin |
| Thermal Solution | Active fan sink |
| Software Development Tools | C-based CUDA Toolkit |

## Nvidia is the Dominant Firm

- CUDA (Compute Unified Device Architecture) is Nvidia's name for their software support of the device.

- The CUDA runtime library and the CUBLAS are the most useful to us.
  - ▷ CUDA runtime functions are prefixed with cuda.
  - ▷ CUBLAS functions are prefixed with cublas.

- The CUDA Driver API gives finer control of the device but is harder to use.
  - ▷ Driver API functions are prefixed with cu.

- OpenCL is supported by Nvidia, Apple, AMD, etc. and is similar to CUDA. We consider it after CUDA.

# How to Determine Device Characteristics – 1

```
CPUCC    = g++
GPUCC    = nvcc
SDIR     = .
ISCL     = ../libscl_float/gpp
LSCL     = ../libscl_float/gpp
ICUDA    = /usr/local/cuda/include
LCUDA    = /usr/local/cuda/lib
CPUFLAGS = -O2 -Wall -c -I$(ICUDA) -I$(SDIR) -I$(ISCL)
GPUFLAGS = -O -c -I$(ICUDA) -I$(SDIR)
LFLAGS   = -L$(LCUDA) -lcuda -lcudart -lcublas -L$(LSCL) -lscl -lm

PROGRAMS = device

LIBRARIES = libscl.a

all : $(LIBRARIES) $(PROGRAMS)

libscl.a :
	make -C $(LSCL)

device : device.o $(LIBRARIES)
	$(CPUCC) -o device device.o $(OBJECTS) $(LFLAGS)

device.o : $(SDIR)/device.cpp
	$(CPUCC) $(CPUFLAGS) $(SDIR)/device.cpp
```

# How to Determine Device Characteristics – 2

```
#include "libscl.h"
#include "cuda.h"
#include "cuda_runtime.h"
#include "cublas.h"

using namespace std;

int main(int argc, char** argp, char** envp)
{
  cudaError_t err;

  int deviceCount;
  cudaGetDeviceCount(&deviceCount);

  int device;
  cudaDeviceProp deviceProp;

  for (device = 0; device < deviceCount; ++device) {
    cout << '\n';
    cout << "Properties for device " << device << '\n';
    err = cudaGetDeviceProperties(&deviceProp, device);
    if (err == cudaSuccess) {
      cout << "Device characteristics" << '\n';
      cout << "\tname = " << deviceProp.name << '\n';
```

# How to Determine Device Characteristics – 3

```
      cout << "\ttotalGlobalMem = " << deviceProp.totalGlobalMem << '\n';
      cout << "\tsharedMemPerBlock = " << deviceProp.sharedMemPerBlock << '\n';
      cout << "\tregsPerBlock = " << deviceProp.regsPerBlock << '\n';
      cout << "\twarpSize = " << deviceProp.warpSize << '\n';
      cout << "\tmemPitch = " << deviceProp.memPitch << '\n';
      cout << "\tmaxThreadsPerBlock = " << deviceProp.maxThreadsPerBlock<<'\n';
      cout << "\tmaxThreadsDim = " << deviceProp.maxThreadsDim[0] << ' '
           << deviceProp.maxThreadsDim[1] <<' '<< deviceProp.maxThreadsDim[2] << '\n';
      cout << "\tmaxGridSize = " << deviceProp.maxGridSize[0] << ' '
           << deviceProp.maxGridSize[1] <<' '<< deviceProp.maxGridSize[2] << '\n';
      cout << "\ttotalConstMem = " << deviceProp.totalConstMem << '\n';
      cout << "\tdeviceProp.major = " << deviceProp.major << '\n';
      cout << "\tdeviceProp.minor = " << deviceProp.minor << '\n';
      cout << "\tclockRate = " << deviceProp.clockRate << '\n';
      cout << "\ttextureAlignment = " << deviceProp.textureAlignment << '\n';
      cout << "\tdeviceOverlap = " << deviceProp.deviceOverlap << '\n';
      cout << "\tmultiProcessorCount = " << deviceProp.multiProcessorCount << '\n';
      cout << "\tkernelExecTimeoutEnabled = "
           << deviceProp.kernelExecTimeoutEnabled << '\n';
      cout << "\tintegrated = " << deviceProp.integrated << '\n';
      cout << "\tcanMapHostMemory = " << deviceProp.canMapHostMemory << '\n';
      cout << "\tcomputeMode = " << deviceProp.computeMode << '\n';
    }
  }
  return 0;
}
```

# Dell T7400 Device Characteristics – 1

```
Properties for device 0

Device characteristics
        name = Tesla C1060
        totalGlobalMem = 4294705152
        sharedMemPerBlock = 16384
        regsPerBlock = 16384
        warpSize = 32
        memPitch = 262144
        maxThreadsPerBlock = 512
        maxThreadsDim = 512 512 64
        maxGridSize = 65535 65535 1
        totalConstMem = 65536
        deviceProp.major = 1
        deviceProp.minor = 3
        clockRate = 1296000
        textureAlignment = 256
        deviceOverlap = 1
        multiProcessorCount = 30
        kernelExecTimeoutEnabled = 0
        integrated = 0
        canMapHostMemory = 1
        computeMode = 0
```

# Dell T7400 Device Characteristics – 2

```
Properties for device 1

Device characteristics
        name = Quadro NVS 290
        totalGlobalMem = 267714560
        sharedMemPerBlock = 16384
        regsPerBlock = 8192
        warpSize = 32
        memPitch = 262144
        maxThreadsPerBlock = 512
        maxThreadsDim = 512 512 64
        maxGridSize = 65535 65535 1
        totalConstMem = 65536
        deviceProp.major = 1
        deviceProp.minor = 1
        clockRate = 918000
        textureAlignment = 256
        deviceOverlap = 1
        multiProcessorCount = 2
        kernelExecTimeoutEnabled = 1
        integrated = 0
        canMapHostMemory = 0
        computeMode = 0
```

# MacBookPro Device Characteristics

```
Properties for device 0

Device characteristics
        name = GeForce 8600M GT
        totalGlobalMem = 536674304
        sharedMemPerBlock = 16384
        regsPerBlock = 8192
        warpSize = 32
        memPitch = 262144
        maxThreadsPerBlock = 512
        maxThreadsDim = 512 512 64
        maxGridSize = 65535 65535 1
        totalConstMem = 65536
        deviceProp.major = 1
        deviceProp.minor = 1
        clockRate = 933330
        textureAlignment = 256
        deviceOverlap = 0
        multiProcessorCount = 4
        kernelExecTimeoutEnabled = 1
        integrated = 0
        canMapHostMemory = 0
        computeMode = 0
```

# Simple CUDA Code from the Guide

A kernel is defined using the __global__ declaration specifier and the number of CUDA threads for each call is specified using <<<  >>> syntax: Each of the threads that execute a kernel is given a unique thread ID that is accessible within the kernel through the built-in threadIdx variable. As an illustration, the following sample code adds two vectors A and B of size N and stores the result into vector C:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Kernel invocation
    VecAdd<<<1, N>>>(A, B, C);
}
```

Each of the threads that execute VecAdd() performs one pair-wise addition.

# CUBLAS

- It is possible to use the CUBLAS directly from C++ code in a style similar to the the CBLAS.

- This involves device memory allocation and host-to-device and device-to-host copies that cannot be avoided.

- CUDA functions using the kernel are asymmetric: threads are still executing after the function returns.

  ▷ One must explicitly synchronize after a call.

- Go through the example at src/cuda/cublas.

# THRUST

- Thrust is a template library patterned after the standard template library that implements many template library algorithms to execute on a GPU
  - ▷ http://code.google.com/p/thrust
  - ▷ Same design philosophy as boost: www.boost.org
- Mixes easily with CUDA.
  - ▷ Especially useful for handling memory transfers for use with CUDA.
  - ▷ Alleviates many synchronization issues.

# The Limit on the Number of Threads - 1

- For performance reasons the number of threads that can be executed simultaneously is limited.
  - ▷ This number is called the Block dimension.
  - ▷ To get enough threads, one divides the number of threads needed by the Block dimension to get a multiple called the Grid.
  - ▷ One passes the Grid and Block sizes in `<<<dimGrid,dimBlock>>>` syntax.
- The dim can be a one, two, or three dimensional concept.

# The Limit on the Number of Threads - 2

- I find the use of more than one dimension confusing and use column major indexing, which is also that of `realmat` and the CUBLAS.

- Column major indexing, where X has r rows, c columns, and size s=r*c. i is the row index; j is the column index, and k is the location in the vec of the matrix

- i=0,...,r-1; j=0,...,c-1; k=0,...,s-1;

  ▷ Locations in a C array: k = r*j + i; j = k/r; i = k%r

- i=1,...,r; j=1,...,c; k=1,...,s;

  ▷ Locations: k = r*(j-1) + i - 1; j = k/r + 1; i = (k-1)%r

- Go through the example at src/cuda/corrX.

# OpenCL

OpenCL is similar to CUDA; some things are harder than CUDA, some are easier. What does make more sense is memory management. The scientific template library ViennaCL contains a BLAS, link below.

The references for the OpenCL slides that follow are

- http://www.khronos.org/opencl/registry/cl
  - ▷ OpenCL_1.2_Specification.pdf
  - ▷ OpenCL_1.1_C++_Bindings_Specification.pdf

- https://www.marcusbannerman.co.uk/index.php/research/teaching-resources

- http://developer.amd.com/sdks/AMDAPPSDK/documentation/Pages/devault.aspx

- http://http://viennacl.sourceforge.net

- Scarpino, Matthew, (2011) *OpenCL in Action*, Manning Publications, Shelter Island, NY.

# How to Determine Device Characteristics – 1

```
CXX      = g++
SDIR     = .
OCLDIR   = /System/Library/Frameworks/OpenCL.framework/Versions/Current
IOCL     = $(OCLDIR)/Headers
LOCL     = $(OCLDIR)/Libraries
ISCL     = ../libscl_float/gpp
LSCL     = ../libscl_float/gpp
IVCL     = ../viennacl
CXXFLAGS = -O2 -Wall -c -shared -I$(SDIR) -I$(ISCL) -I$(IOCL) -I$(IVCL)
LDFLAGS  = -lm -L$(LSCL) -lscl -L$(LOCL) -framework OpenCL

devices : devices.o $(OBJECTS)
        $(CXX) -o devices devices.o $(OBJECTS) $(LDFLAGS)

devices.o : $(SDIR)/devices.cpp
        $(CXX) $(CXXFLAGS) $(SDIR)/devices.cpp
```

Above is for Max OS 10.7.3, for Linux use

```
CXX      = g++
SDIR     = .
ISCL     = ../libscl_float/gpp
LSCL     = ../libscl_float/gpp
IVCL     = ../viennacl
CXXFLAGS = -O2 -Wall -c -shared -I$(SDIR) -I$(ISCL) -I$(IVCL)
LDFLAGS  = -lm -L$(LSCL) -lscl -lOpenCL
```

# How to Determine Device Characteristics – 2

```
#include "scltypes.h"
#include "sclerror.h"
#include "cl.hpp"

int main()
{
  cl_int err;

  std::vector<cl::Platform> platformList;
  err = cl::Platform::get(&platformList);
  if (err != CL_SUCCESS) scl::error("Error, cl::Platform::get failed");

  std::vector<cl::Platform>::const_iterator plitr;
  for (plitr=platformList.begin(); plitr!=platformList.end(); ++plitr) {

    std::string platformVendor;
    err = plitr->getInfo((cl_platform_info)CL_PLATFORM_VENDOR, &platformVendor);
    if (err != CL_SUCCESS) scl::error("Error, cl::Platform::getInfo failed");
    std::cout << platformVendor << '\n';

    std::vector<cl::Device> deviceList;
    err = plitr->getDevices((cl_device_type)CL_DEVICE_TYPE_ALL, &deviceList);
    if (err != CL_SUCCESS) scl::error("Error, cl::Platform::getDevices failed");
```

# How to Determine Device Characteristics – 3

```
std::vector<cl::Device>::const_iterator dlitr;
for (dlitr=deviceList.begin(); dlitr!=deviceList.end(); ++dlitr) {

  std::string deviceName;
  err = dlitr->getInfo((cl_device_info)CL_DEVICE_NAME, &deviceName);
  if (err != CL_SUCCESS) scl::error("Error, cl::Device::getInfo failed");
  std::cout << deviceName << '\n';

  std::vector<size_t> deviceSizes;
  cl_device_info work_item_sizes = CL_DEVICE_MAX_WORK_ITEM_SIZES;
  err = dlitr->getInfo(work_item_sizes, &deviceSizes);
  if (err != CL_SUCCESS) scl::error("Error, cl::Device::getInfo failed");

  std::vector<size_t>::const_iterator dsitr;
  std::cout << "  Max work item sizes: " << '\n';;
  for(dsitr=deviceSizes.begin(); dsitr!=deviceSizes.end(); ++dsitr) {
    std::cout << "    " << *dsitr << '\n';
  }

  cl_device_info work_group_size = CL_DEVICE_MAX_WORK_GROUP_SIZE;
  size_t groupSize;
  err = dlitr->getInfo(work_group_size, &groupSize);
  if (err != CL_SUCCESS) scl::error("Error, cl::Device::getInfo failed");
  std::cout << "  Max work group size = " << groupSize << '\n';
```

# How to Determine Device Characteristics – 4

```
      cl_device_info compute_units = CL_DEVICE_MAX_COMPUTE_UNITS;
      cl_uint computeSize;
      err = dlitr->getInfo(compute_units, &computeSize);
      if (err != CL_SUCCESS) scl::error("Error, cl::Device::getInfo failed");
      std::cout << "  Max compute units = " << computeSize << '\n';

      cl_device_info double_info = CL_DEVICE_DOUBLE_FP_CONFIG;
      cl_device_fp_config double_conf;
      err = dlitr->getInfo(double_info, &double_conf);
      if (err != CL_SUCCESS) scl::error("Error, cl::Device::getInfo failed");
      std::cout << std::boolalpha;
      std::cout << "  Has double = " << bool(double_conf) << '\n';
    }
  }

  return 0;
}
```

# How to Determine Device Characteristics – 5

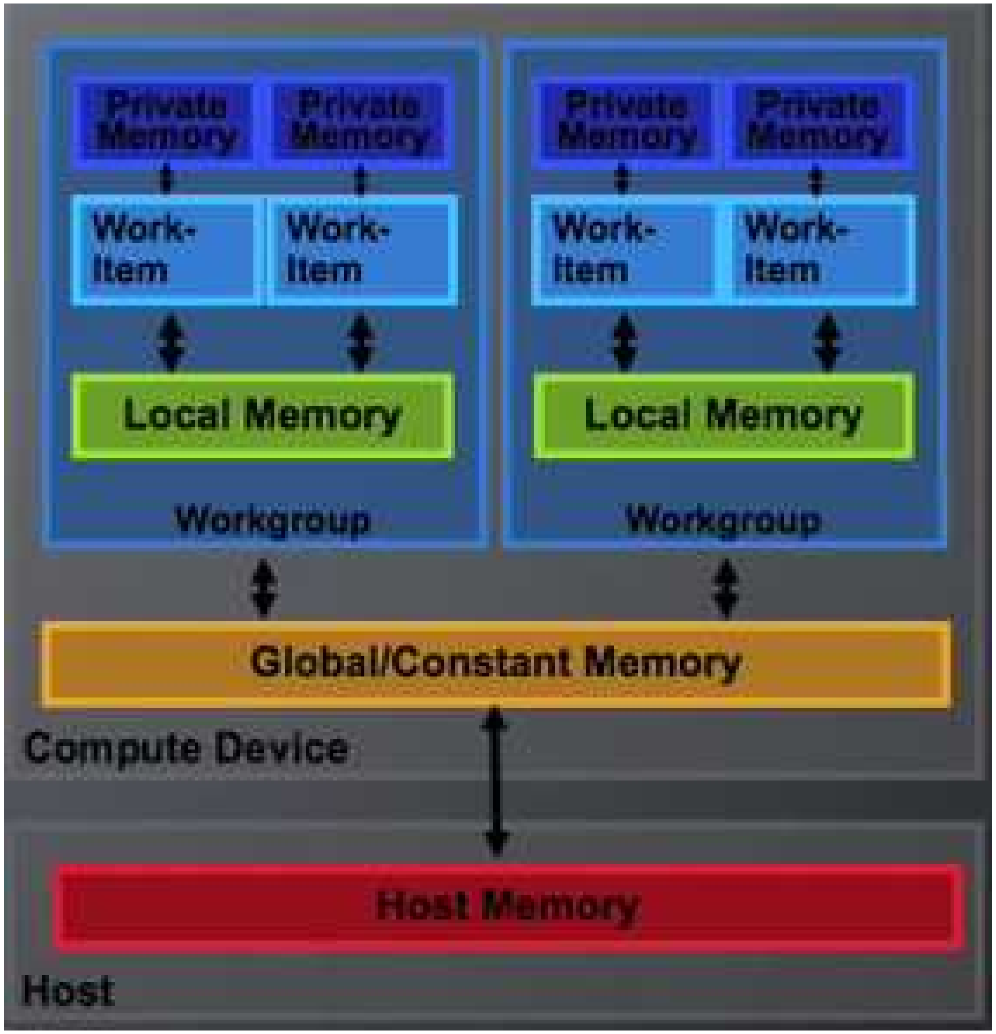
```
Apple

Intel(R) Core(TM) i7-2860QM CPU @ 2.50GHz
  Device type = 2
  Max work item sizes:
    1024
    1
    1
  Max work group size = 1024
  Max compute units = 8
  Has double = true

ATI Radeon HD 6770M
  Device type = 4
  Max work item sizes:
    1024
    1024
    1024
  Max work group size = 1024
  Max compute units = 6
  Has double = false
```

## A Kernel

```
__kernel void squareArray(__global float* input, __global float* output)"
  {
    output[get_global_id(0)] = input[get_global_id(0)]*input[get_global_id(0)];
  };
```

Go through the rest of the **`hello.cpp`** example to illustrate context, building the kernel, running the kernel, copying memory, etc.

Fig 2. Work Items, Work Groups, and Memory



# OpenCL Addresses – 1

Indexes, called work-items, index PEs and can also index memory. There can be up to three such indexes: $(x, y, z)$. I will use two, $x$ and $y$, for illustration.

There are two indexing schemes, global and group:

**Global Indexing:** There are $G_x \times G_y$ PEs available. PEs are indexed by *global-ids* $g_x = 0, \cdots, G_x - 1$ and $g_y = 0, \cdots, G_y - 1$.

**Group Indexing:** The PEs are divided into work-groups of sizes $S_x$ and $S_y$. PEs within a work-group are indexed by *local-ids* $s_x = 0, \cdots, S_x - 1$ and $s_y = 0, \cdots, S_y - 1$.

There are, therefore, $W_x \times W_y = G_x/S_x \times G_y/S_y$ work-groups. Work-groups are indexed by *group-ids* $w_x = 0, \cdots, W_x - 1$ and $w_y = 0, \cdots, W_y - 1$.

# OpenCL Addresses – 2

The relationship between global and group indexing is

$$\begin{aligned} g_x &= w_x * S_x + s_x \\ g_y &= w_y * S_x + s_y \end{aligned}$$

If there are offsets $F_x$ and $F_y$ then

$$\begin{aligned} g_x &= w_x * S_x + s_x + F_x \\ g_y &= w_y * S_x + s_y + F_y \end{aligned}$$

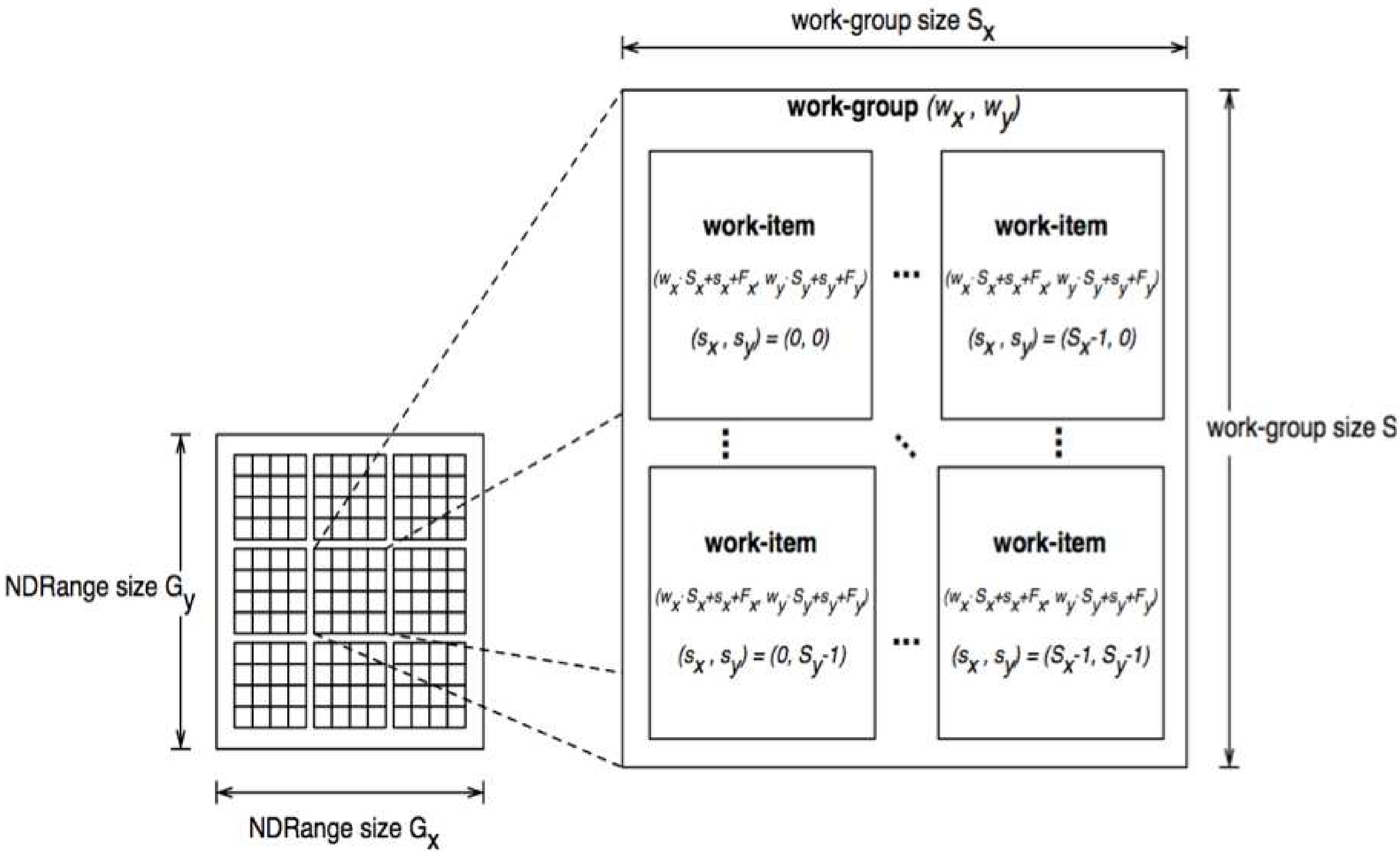
Within a kernel, these calls provide the indexes

$$\begin{aligned} g_x &= \text{get\_global\_id(0)} \\ g_y &= \text{get\_global\_id(1)} \\ w_x &= \text{get\_group\_id(0)} \\ w_y &= \text{get\_group\_id(1)} \\ s_x &= \text{get\_local\_id(0)} \\ s_y &= \text{get\_local\_id(1)} \end{aligned}$$

with similar calls to get $G_x$, $W_x$, $S_x$, $F_x$ etc.

Typically one uses *global-ids* to index global memory and *local-ids* to index local memory.

# Fig 3. OpenCL Addresses – 3



# OpenCL Speed

The speed of OpenCL code is governed by the same rules we have already discussed. Moreover, there is no optimizing compiler to help you. The rules are

- Access memory sequentially.

- Localize computations so that all fetchs are from the cache.

  There is no cache on a graphics card so one has to make one's own from local memory. Local memory is fast; global memory is slow.

- Avoid if statements.

  If you must use them, arrange code so that they evaluate to true more frequently than false because pipelines usually make that assumption.

# Warps

An instruction is executed on a contiguous set of PEs simultaneously. Nvidia calls this set a warp; ATI a wavefront. An Nvidia warp has 32 PEs in it; an ATI wavefront has 64. Work-group sizes $S_x$, $S_y$ should be a multiple of the warp or wavefront.

If a computation only uses, e.g., 16 PEs with a warp of 32, then 32 will be launched, 16 of which will compute nothing.

This makes branching especially deadly:

```
if (get_local_id(0) < 16) {
  // Do something
  // Work-items 16 through 31 are disabled but still run.
}
else {
  // Do something
  // Work-items 0 through 15 are disabled but still run.
}
```

If branch you must, branch on warp boundaries.

# OpenCL Examples

We will next examine some code from `matrixvecmult.cpp` and `matrixvecmult.cl` to illustrate addressing and memory management.

- The kernels we will look at compute

  $$W = MV$$

  where $M$ is a matrix and $V$ is a vector.

- $M$ is stored row-wise.
  - ▷ To access memory sequentially,
  - ▷ increment the column index.

# A CPU Implementation

```
void MatrixVectorMul(const float* M, const float* V, float* W,
  uint width, uint height )
{
  for (uint y = 0; y < height; ++y) {  // y is the row index

    const float* row = M + y * width;  // row points to row y

    float dotProduct = 0;
    for (uint x = 0; x < width; ++x) { // x is the column index
      dotProduct += row[x] * V[x];
    }

    W[y] = dotProduct;
  }
}
```

# The GPU Analog

```
__kernel void MatrixVectorMul1(const __global float* M,
                               const __global float* V,
                               __global float* W,
                               uint width, uint height)
{
  // Each work-item computes one element of W
  uint y = get_global_id(0);
  if (y >= height) return;

  const __global float* row = M + y * width;

  float dotProduct = 0;
  for (uint x = 0; x < width; ++x)
    dotProduct += row[x] * V[x];

  W[y] = dotProduct;
}
```

Problem: `height` could exceed the global size limit.

# GPU Analog with Size Problem Fixed

```
__kernel void MatrixVectorMul2(const __global float* M,
                               const __global float* V,
                               __global float* W,
                               uint width, uint height)
{
  // Each work-item computes multiple elements of W
  for (uint y = get_global_id(0); y < height; y += get_global_size(0))
    {
      const __global float* row = M + y * width;

      float dotProduct = 0;
      for (uint x = 0; x < width; ++x)
        dotProduct += row[x] * V[x];

      W[y] = dotProduct;
    }
}
```

Problem: We are jumping though memory. Each work-item read is separated by `width`. We are not using local memory at all.

# Jump Problem Fix – 1

```
__kernel void MatrixVectorMul3(const __global float* M,
                               const __global float* V,
                               __global float* W,
                               uint width, uint height,
                               __local float* partialDotProduct)
{
  // Each work-group computes multiple elements of W
  for (uint y = get_group_id(0); y < height; y += get_num_groups(0))
    {
      const __global float* row = M + y * width;

      float sum = 0;
      for (uint x = get_local_id(0); x < width; x += get_local_size(0))
        sum += row[x] * V[x];

      partialDotProduct[get_local_id(0)] = sum;

      barrier(CLK_LOCAL_MEM_FENCE);
```

The barrier guarantees that `y` cannot be incremented until all reads from global memory at `row[x]` and `V[x]` and writes to local memory at `partialDotProduc[x]` have finished. Indexing along `x` is contiguous because of execution by warps.

# Jump Problem Fix – 2

```
    if (get_local_id(0) == 0)
      {
        float dotProduct = 0;
        for (uint t = 0; t < get_local_size(0); ++t)
          dotProduct += partialDotProduct[t];
        W[y] = dotProduct;
      }

    barrier(CLK_LOCAL_MEM_FENCE);
  }
}
```

Problem: The whole work-group must wait for the partial dot products to be accumulated.

# Accumulation Problem Fix – 1

```
__kernel void MatrixVectorMul4(const __global float* M,
                               const __global float* V,
                               __global float* W,
                               uint width, uint height,
                               __local float* partialDotProduct)
{
  // Each work-group computes multiple elements of W
  for (uint y = get_group_id(0); y < height; y += get_num_groups(0))
    {
      const __global float* row = M + y * width;

      float sum = 0;
      for (uint x = get_local_id(0); x < width; x += get_local_size(0))
        sum += row[x] * V[x];

      partialDotProduct[get_local_id(0)] = sum;
```

Up to here is the same as before except the barrier statement is moved to within the loop on the next slide.

Note that `partialDotProduct` is local memory.

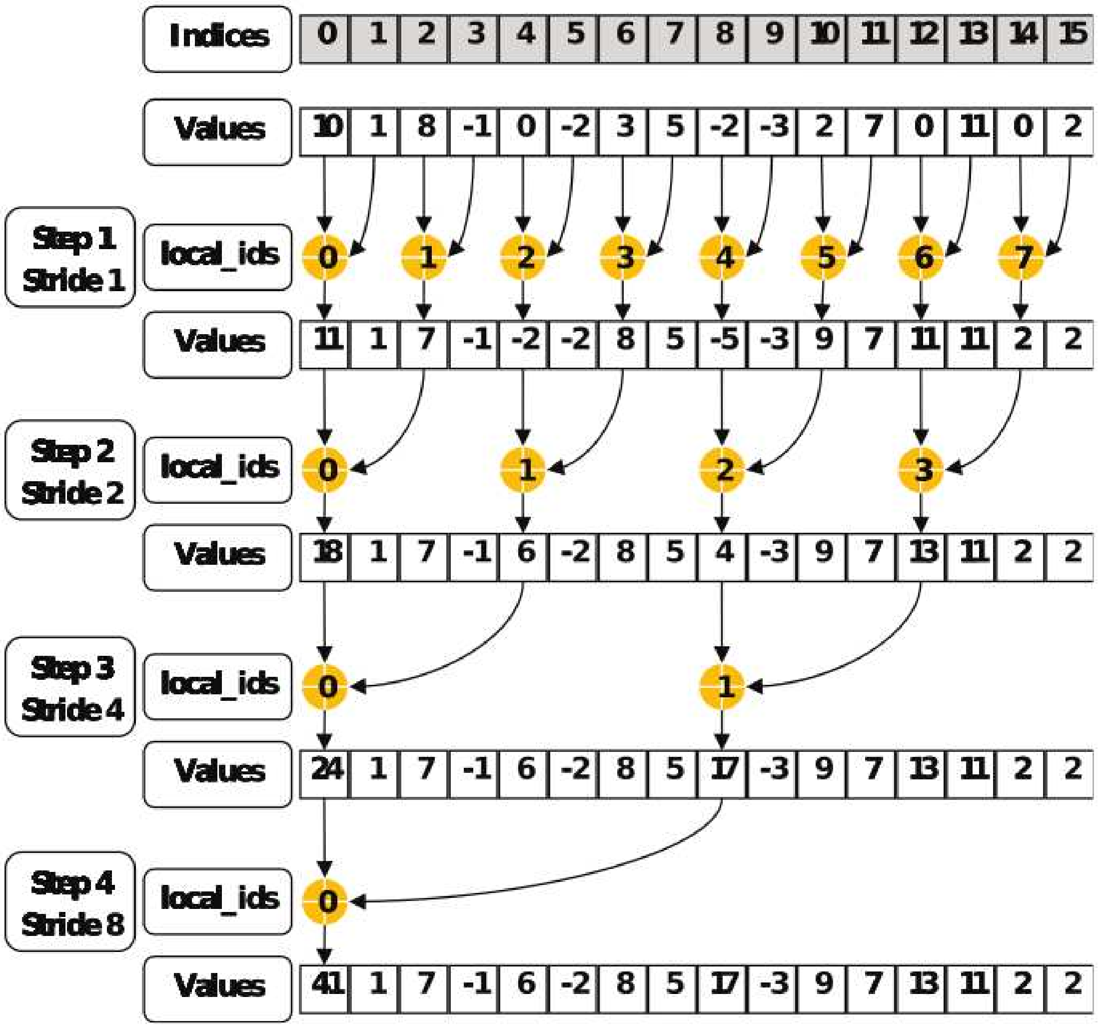
# Accumulation Problem Fix – 2

```
      for (uint stride = 1; stride < get_local_size(0); stride *= 2)
        {
          barrier(CLK_LOCAL_MEM_FENCE);
          uint index = 2 * stride * get_local_id(0);

          if (index < get_local_size(0))
            partialDotProduct[index] +=
              partialDotProduct[index + stride];
        }

      if (get_local_id(0) == 0)
        W[y] = partialDotProduct[0];

      barrier(CLK_LOCAL_MEM_FENCE);
    }
}
```

On entry to the loop, the barrier blocks execution until all work-items in the group have finished updating local memory. After that, `stride` cannot increment until all work-items have once again updated local memory.

The next slide shows what the `stride` loop does.

Fig 4. Stride Loop



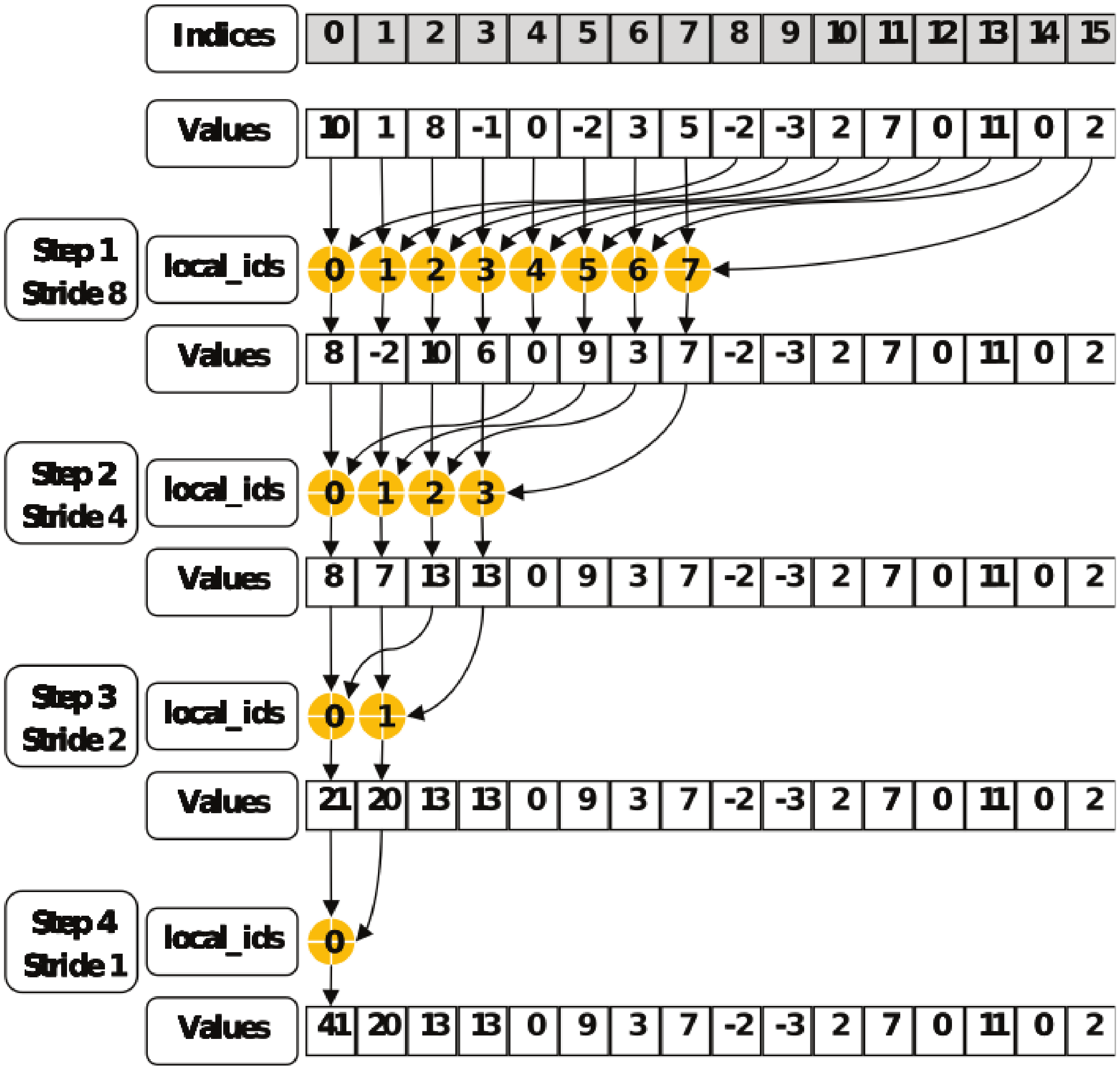
# A Better Accumulation Loop

```
      for (uint stride = get_local_size(0)/2; stride > 0; stride /= 2)
        {
          barrier(CLK_LOCAL_MEM_FENCE);

          if (get_local_id(0) < stride)
            partialDotProduct[get_local_id(0)] +=
              partialDotProduct[get_local_id(0) + stride];
        }

      if (get_local_id(0) == 0) W[y] = partialDotProduct[0];

      barrier(CLK_LOCAL_MEM_FENCE);
    }
}
```

This loop accesses local memory in separated chunks, which is a bit faster.

The next slide shows what the `stride` loop does.

# Fig 5. Better Loop



# Telsa C1060 Timing

```
Tesla C1060

CPU took 0.168037 sec

Testing MatrixVectorMul1
  WorkGroupSize = 64 GlobalSize 100032
  Average kernel execution time 0.137472

Testing MatrixVectorMul2
  WorkGroupSize = 64 GlobalSize 3840
  Average kernel execution time 0.132226

Testing MatrixVectorMul3
  WorkGroupSize = 64 GlobalSize 3840
  Average kernel execution time 0.0278443

Testing MatrixVectorMul4
  WorkGroupSize = 64 GlobalSize 3840
  Average kernel execution time 0.0219479

Testing MatrixVectorMul5
 WorkGroupSize = 64 GlobalSize 3840
  Average kernel execution time 0.0206547
```

# OpenCL Demo

Compile and run `matrixvecmult` on `argux2`'s Telsa C1060 and on the laptop's ATI Radeon HD 6770M.

# ViennaCL

- A scientific computing library
- Includes a BLAS
- Exceptionally easy to use.
  - http://http://viennacl.sourceforge.net

# ViennaCL $C = AB$ Example – 1

- To interface with a `realmat` use a `vclmat` in `libscl`

```
class vclmat {
public:
  vclmat();                              \\ default constructor
  vclmat(scl::realmat& a);               \\ construct vclmat wrapper for a
  vclmat(const vclmat& cpu_a);           \\ copy constructor, not wrapper
  vclmat& operator=(const vclmat& cpu_a); \\ assignment, lhs not wrapper
  operator scl::realmat() const;         \\ conversion operator
  bool is(const scl::realmat& a) const;  \\ test if vclmat is a wrapper
  unsigned int size() const;             \\ viennacl interface method
  unsigned int  size1() const;           \\ viennacl interface method
  unsigned int  size2() const;           \\ viennacl interface method
  REAL* begin();                         \\ viennacl interface method
  REAL* end();                           \\ viennacl interface method
  REAL operator()(uint i, uint j);       \\ viennacl interface method
  REAL& operator()(uint i, uint j);      \\ viennacl interface method
};
```

- To define a ViennaCL matrices from `realmat A, B, C`

```
  viennacl::matrix<float,viennacl::column_major> gpu_A(A.nrow(),A.ncol());
  viennacl::matrix<float,viennacl::column_major> gpu_B(B.nrow(),B.ncol());
  viennacl::matrix<float,viennacl::column_major> gpu_C(A.nrow(),B.ncol());
```

# ViennaCL $C = AB$ Example – 2

- To copy `realmat A, B` from CPU to GPU

```
vclmat cpu_A(A);
vclmat cpu_B(B);
viennacl::copy(cpu_A,gpu_A);
viennacl::copy(cpu_B,gpu_B);
```

- To compute `C = A * B` on the GPU

```
gpu_C = viennacl::linalg::prod(gpu_A, gpu_B);
```

- To copy from GPU to CPU to `realmat C`

```
realmat C(A.nrow(),B.ncol());
vclmat cpu_C(C);
viennacl::copy(gpu_C,cpu_C);
if (!cpu_C.is(C)) C = cpu_C;
```

# ViennaCL $C = AB$ Timing, Linux

```
Arows = 1000
Acols = 10000
Bcols = 1000;

Linux 2.6.18, Intel Xeon 3.16GHz 6144 KB cache (unified),
Telsa C1060, libscl
libscl_float mult time = 14.9863      <----
viennacl A & B copy time = 0.661133i         compare
viennacl mult time = 0.001064         <----
viennacl C copy time = 0.632763
viennacl total time = 1.29496
GPU/CPU time = 8.64098 per cent

Linux 2.6.18, Intel Xeon 3.16GHz 6144 KB cache (unified),
Telsa C1060, libsclcb
libscl_float mult time = 1.68685      <----
viennacl A & B copy time = 0.660871          compare
viennacl mult time = 0.001028         <----
viennacl C copy time = 0.633325
viennacl total time = 1.29522
GPU/CPU time = 76.7837 per cent
```

Moral: Minimize copies between CPU and GPU

Better: Use pthreads to do something simultaneously

# ViennaCL $C = AB$ Timing, Apple

```
Arows = 1000
Acols = 10000
Bcols = 1000;

Mac OS X 10.6.8, Intel i7 2.66GHz 256 KB L2 (per core), 4MB L3
GeForce GT 330M libscl
libscl_float mult time = 10.8579      <----
viennacl A & B copy time = 0.68326           compare
viennacl mult time = 0.063123         <----
viennacl C copy time = 3.37502
viennacl total time = 4.1214
GPU/CPU time = 37.9576 per cent

Mac OS X 10.6.8, Intel i7 2.66GHz 256 KB L2 (per core), 4MB L3
GeForce GT 330M libsclcb
libscl_float mult time = 3.85385      <----
viennacl A & B copy time = 0.643956          compare
viennacl mult time = 0.064301         <----
viennacl C copy time = 3.39043
viennacl total time = 4.09868
GPU/CPU time = 106.353 per cent
```

Moral: Minimize CPU$\leftrightarrow$GPU copies, esp. GPU$\rightarrow$CPU

Better: Use pthreads to do something simultaneously

# ViennaCL Regression Example – 1

- using libscl

```
realmat X(n,p);
realmat y(n,1);
realmat C = T(X)*X;
realmat b = invpsd(C)*(T(X)*y);
```

- memory layout is same as realmat if tag = column_major

```
viennacl::matrix<float,viennacl::column_major> gpu_X(n,p);
viennacl::matrix<float,viennacl::column_major> gpu_C(p,p);
viennacl::vector<float> gpu_y(n);
viennacl::vector<float> gpu_b(p);
```

# ViennaCL Regression Example – 2

- fast_copy

```
viennacl::fast_copy(X.begin(), X.end(), gpu_X);
viennacl::fast_copy(y.begin(), y.end(), gpu_y.begin());
```

- compute `b = invpsd(T(X)*X)*(T(X)*y)` on the GPU

```
gpu_C = viennacl::linalg::prod(trans(gpu_X), gpu_X);
gpu_b = viennacl::linalg::prod(trans(gpu_X), gpu_y);
viennacl::linalg::lu_factorize(gpu_C);
viennacl::linalg::lu_substitute(gpu_C, gpu_b);
```

- copy `b` from GPU to CPU

```
viennacl::fast_copy(gpu_b.begin(),gpu_b.end(),b.begin());
```

Notice that there is no need for a wrapper when fast_copy is used

# ViennaCL Timing, Linux, libscl

```
const INTEGER p = 30;
const INTEGER n = 100000;

Linux 2.6.18, Intel Xeon 3.16GHz 6144 KB cache (unified),
Telsa C1060, libscl

regr is not using fast_copy

ilibscl least squares time = 0.207739
viennacl X & y copy time = 0.095376
viennacl least squares time = 0.001063
viennacl b copy time = 0.085818    <-- GPU to CPU expensive
viennacl total time = 0.182257
GPU/CPU total time          =  87.7337 per cent
GPU/CPU least squares time  =   0.5117 per cent

regr is using fast_copy

libscl least squares time = 0.205742
viennacl X & y copy time = 0.013628
viennacl least squares time = 0.001215
viennacl b copy time = 0.085871    <-- GPU to CPU expensive
viennacl total time = 0.100714
GPU/CPU total time          =  48.9516 per cent
GPU/CPU least squares time  =   0.5905 per cent
```

# ViennaCL Timing, Linux, libsclcb

```
const INTEGER p = 30;
const INTEGER n = 100000;

Linux 2.6.18, Intel Xeon 3.16GHz 6144 KB cache (unified),
Telsa C1060, libsclcb

regr is not using fast_copy

libscl least squares time = 0.04383
viennacl X & y copy time = 0.095083
viennacl least squares time = 0.001313
viennacl b copy time = 0.085606    <-- GPU to CPU expensive
viennacl total time = 0.182002
GPU/CPU total time          = 415.2453 per cent
GPU/CPU least squares time  =   2.9957 per cent

regr is using fast_copy

libscl least squares time = 0.043693
viennacl X & y copy time = 0.013632
viennacl least squares time = 0.001202
viennacl b copy time = 0.08596     <-- GPU to CPU expensive
viennacl total time = 0.100794
GPU/CPU total time          = 230.6868 per cent
GPU/CPU least squares time  =   2.7510 per cent
```

# ViennaCL Summary

- ViennaCL is exceptionally easy to use.
  - ▷ http://http://viennacl.sourceforge.net

- No need to use the OpenCL classes
  1. `Platform`,
  2. `Device`,
  3. `Context`,
  4. `Program`,
  5. `CommandQueue`,
  6. `Buffer`,
  7. `KernelFunctor`,

- But you can to get more control or to run your own `kernels`
  - ▷ A ViennaCL device can be an SMP machine's CPU's
  - ▷ There can be more than one device

## We Are Finished!

We are finished with parallel computing.

We'll move on to optimization.

# Nonlinear Optimization and Equation Solving

- For many statistical objective functions we have seen that MCMC can be used as an optimizer.

- We will now study methods that can be used for smooth functions.

- We will be discussing the ideas behind four classes in libscl:
  - ▷ nleqns_base – Interface that defines a nonlinear function
  - ▷ nlsolve – Implements Newton's method with line search
  - ▷ nlopt – Implements the BFGS quasi Newton method
  - ▷ linesrch – Implements Fletcher's line search method

- Go through declaration in `libscl.h`

# Optimization of Smooth Functions

Smooth functions $f(x)$, $x \in \mathcal{R}^n$, either have analytic derivatives or have derivatives that can be well approximated numerically.

We are trying to find $x^*$ that minimizes $f(x)$ ...

... by constructing a sequence $x^{(0)}$, $x^{(1)}$, ... that converges to $x^*$, where we must choose $x^{(0)}$.

Reference: Fletcher, Roger (1987), *Practical Methods of Optimization, Second Edition* Wiley, New York, ISBN 0 471 91547 5.

# Basic Strategy

If at $x^{(k)}$, approximate $f(x)$ by a quadratic

$$f(x^{(k)} + \delta) \doteq f(x^{(k)}) + g^{(k)T}\delta + \frac{1}{2}\delta^T G^{(k)}\delta$$

The value of $\delta$ that minimizes $f(x^{(k)} + \delta)$ is

$$s^{(k)} = -\left(G^{(k)}\right)^{-1} g^{(k)}$$

Rather than accept $x^{(k)} + s^{(k)}$ as our next approximation we only accept $s^{(k)}$ as a search direction: We seek to find $\alpha^{(k)}$ that approximately minimizes

$$f(\alpha) = f(x^{(k)} + \alpha s^{(k)})$$

and put $x^{(k+1)} = x^{(k)} + \alpha^{(k)} s^{(k)}$

# Basic Strategy

In the approximation, $g^{(k)}$ is usually the gradient $(\partial/\partial x)f(x)$.

Numerical schemes that are cheap to compute and guarantee positive definiteness are used to approximate $H^{(k)} = \left(G^{(k)}\right)^{-1}$ because the exact Hessian is often a poor choice for $G^{(k)}$.

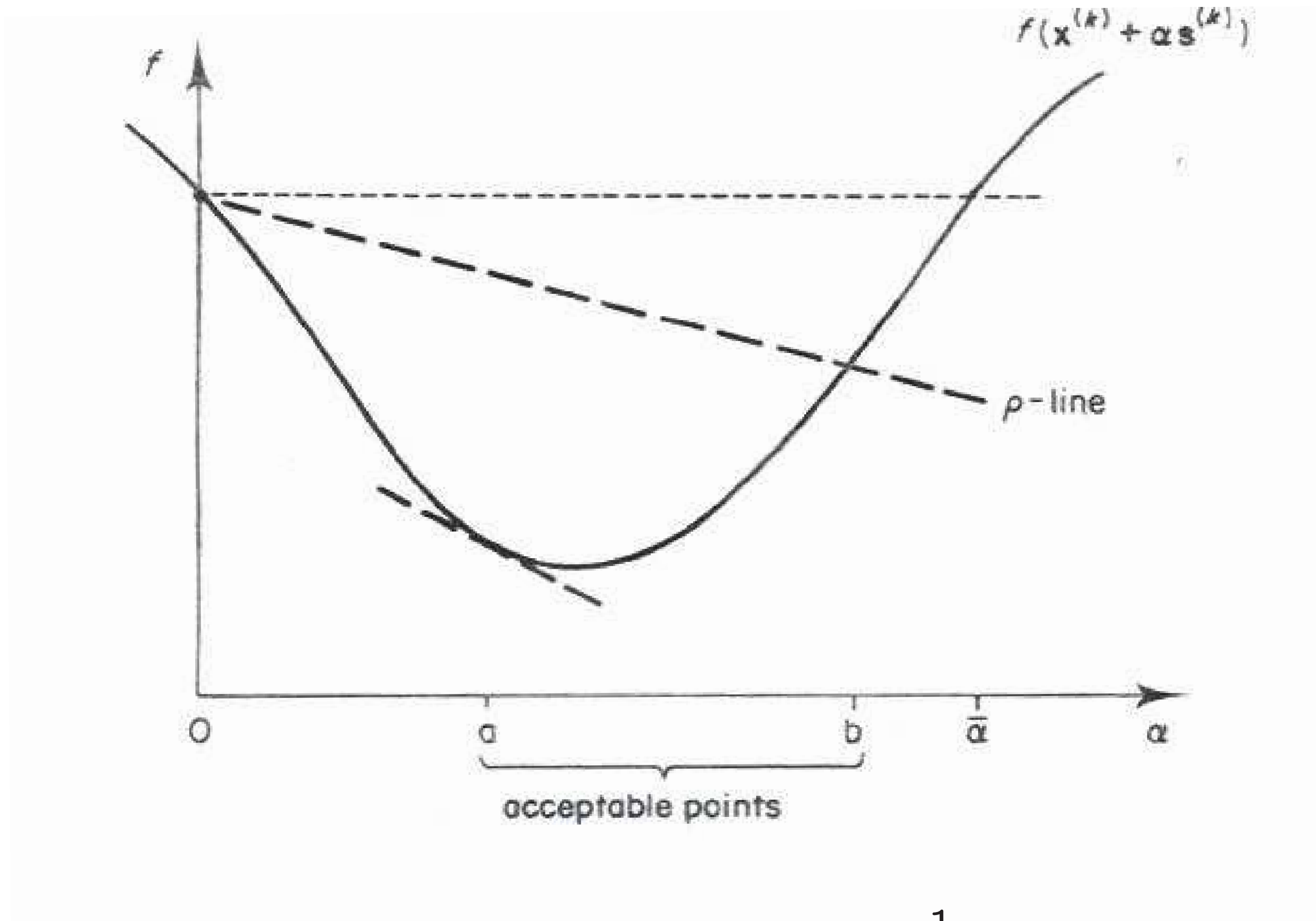
What is most important to do well is to find $\alpha^{(k)}$ that approximately minimizes

$$f(\alpha) = f(x^{(k)} + \alpha s^{(k)})$$

or at least find values for $\alpha^{(k)}$ that do not tend to zero as iterations progress and that satisfy

$$f(x^{(k)} + \alpha^{(k)} s^{(k)}) < f(x^{(k)})$$

# Wolfe-Powell Conditions




$$f(\alpha) \leq f(0) + \alpha \rho f'(0) \quad \rho \doteq \frac{1}{2}$$

$$f'(\alpha) \geq \sigma f'(0) \quad \sigma \in (\rho, 1)$$

# Fletcher's Line Search Strategy

Best method I've ever used!

First it brackets, trying to find an interval $[a_i, b_i]$ that contains the interval [a, b] of acceptable points.

Next it sections to get a sequence of brackets $[a_i, b_i]$ containing $a^{(k)}$ whose length tends to zero.

What follows describes my implementation (class linesrch in libscl) and corrects two errors in Fletcher's book.

## Definitions and Tuning Parameters

Terminate means terminate line search and accept current $\alpha_i$ as $\alpha^{(k)}$.

Terminate $B$ means terminate bracketing and accept current $[a_i,\, b_i]$.

Intervals written $[a_i,\, b_i]$ can have $a_i > b_i$ and must be checked before use.

$\bar{f}$ is a lower bound; use 0 for a minimum norm problem; use current value less an optimistic decrease for other problems;

$\mu = [\bar{f} - f(0)]/[\rho f'(0)]$ $\quad$ $\mu$ is where the $\rho$-line equals $\bar{f}$

Initialize $\alpha_0$ at 0 and $\alpha_1$ at either $\min(\mu, 1)$ or a good guess at $\alpha^{(k)}$

$\sigma = 0.1$ for an accurate estimate of $\alpha^{(k)}$; use $\sigma = 0.9$ for speed.

$\tau_1 = 9.0$; $\tau_2 = 0.1$; $\tau_3 = 0.5$; $\rho = 0.01$; but $\tau_2 \leq \sigma$; $\rho < \sigma$

Use a cubic polynomial to interpolate the four points given.

Actual code has defenses against NaNs, Infs, etc. and provides diagnostics.

To use the code only $\bar{f}$ must be specified; $\alpha_1$ and $\sigma$ may be specified

# Bracketing

for $\quad i := 1, 2, \ldots$ do

begin $\quad$ evaluate $f(\alpha_i)$

if $f(\alpha_i) \leq \bar{f}$ then terminate

if $f(\alpha_i) > f(0) + \rho \alpha_i f^{'}(0)$ or $f(\alpha_i) \geq f(\alpha_{i-1})$

then begin $a_i := \alpha_{i-1}$; $b_i := \alpha_i$; terminate $B$; end;

evaluate $f^{'}(\alpha_i)$;

if $|f^{'}(\alpha_i)| \leq -\sigma f^{'}(0)$ then terminate;

if $f^{'}(\alpha_i) \geq 0$

then begin $a_i := \alpha_i$; $b_i := \alpha_{i-1}$; terminate $B$; end;

if $\mu \leq 2\alpha_i - \alpha_{i-1}$

then $a_{i+1} := \mu$;

else use $f(\alpha_{i-1}), f^{'}(\alpha_{i-1}), f(\alpha_i), f^{'}(\alpha_i)$ to interpolate

best $\alpha_{i+1} \in [2\alpha_i - \alpha_{i-1}, \min(\mu, \alpha_i + \tau_1(\alpha_i - \alpha_{i-1}))]$

end;

# State at Terminate $B$

On exiting the bracketing algorithm, the following are true:

1. $a_i$ is the current best trial point (least $f$ ) that satisfies $f(\alpha) \leq f(0) + \alpha\rho f'(0)$.

2. $f'(a_i)$ has been evaluated and satisfies $(b_i - a_i)f'(a_i) < 0$ but not $|f'(a_i)| \leq -\sigma f'(0)$.

3. $b_i$ satisfies either $f(b_i) > f(0) + \alpha\rho f'(0)$ or $f(b_i) > f(a_i)$ or both.

Fletcher proves that a bracket that satisfies these conditions contains an interval of acceptable points.

# Sectioning

for $\quad j := i, i+1, \ldots$ do
begin $\quad$ use $f(\alpha_{i-1}), f^{'}(\alpha_{i-1}), f(\alpha_i), f^{'}(\alpha_i)$ to interpolate
best $\alpha_j \in [a_j + \tau_2(b_j - a_j),\, b_j - \tau_3(b_j - a_j)]$;
evaluate $f(\alpha_j)$;
if $|(a_j - \alpha_j)f^{'}(a_j)| <$ solution_tolerance then terminate;
if $f(\alpha_j) > f(0) + \rho\alpha_j f^{'}(0)$ or $f(\alpha_j) \geq f(a_j)$;
then begin $a_{j+1} := a_j$; $b_{j+1} := \alpha_j$; end;
else begin
evaluate $f^{'}(\alpha_j)$;
if $|f^{'}(\alpha_j)| \leq -\sigma f^{'}(0)$ or $f(\alpha_j) \leq \bar{f}$ then terminate;
$a_{j+1} := \alpha_j$;
if $(b_j - a_j)f^{'}(\alpha_j) \geq 0$ then $b_{j+1} := a_j$ else $b_{j+1} := b_j$;
end;
end;

# Recall the Basic Strategy

If at $x^{(k)}$, approximate $f(x)$ by a quadratic

$$f(x^{(k)}+\delta) \doteq f(x^{(k)}) + g^{(k)T}\delta + \frac{1}{2}\delta^T G^{(k)}\delta$$

The value of $\delta$ that minimizes $f(x^{(k)}+\delta)$ is

$$s^{(k)} = -\left(G^{(k)}\right)^{-1} g^{(k)}$$

Find $\alpha^{(k)}$ that approximately minimizes

$$f(\alpha) = f(x^{(k)} + \alpha s^{(k)})$$

and put $x^{(k+1)} = x^{(k)} + \alpha^{(k)} s^{(k)}$

# The Linear Term

The best choice of $g^{(k)}$ is $(\partial/\partial x)f(x^{(k)})$ computed analytically.

Avoid numerical approximation if at all possible. If forced to use numerical approximation, at least use two sided differences (see class nleqns in libscl). You may have to use higher order polynomials to get sufficient accuracy.

# The Quadratic Term

The best choice for $H^{(k)} = \left(G^{(k)}\right)^{-1}$ is the BFGS (Broyden, Fletcher, Goldfarb, Shanno) formula

$$H^{(k+1)} = H + \left(1 + \frac{\gamma' H \gamma}{\gamma' \delta}\right) \frac{\delta \delta'}{\gamma' \delta} - \left(\frac{\delta \gamma' H + H \gamma \delta'}{\gamma' \delta}\right)$$

where $H = H^{(k)}$, $\delta = x^{(k)} - x^{(k-1)}$, $\gamma = g^{(k)} - g^{(k-1)}$, $H^{(0)} = I$

The theory supporting this formula is tedious. The main idea is that one wants an update formula of the form $H^{(k+1)} = H + aa' + bb'$ that satisfies the *quasi Newton condition* $H^{(k+1)}\gamma = \delta$, which says that Taylor's theorem $\gamma = G\delta + o(\|\delta\|)$ holds exactly. $H^{(k+1)}$ converges to the Hessian at the optimum.

# Example

Illustrate with code in `compecon/src/nlopt`

# Solving Smooth Systems

Smooth functions $f(x)$, $x \in \mathcal{R}^n$, either have analytic derivatives or have derivatives that can be well approximated numerically.

We are trying to find $x^*$ that solves $f(x) = 0$.

# Basic Strategy

If at $x^{(k)}$, approximate $f(x)$ by a linear system

$$f(x^{(k)} + \delta) \doteq f(x^{(k)}) + F^{(k)}\delta$$

The value of $\delta$ that solves $f(x^{(k)} + \delta)$ is

$$s^{(k)} = -\left(F^{(k)}\right)^{-1} f(x^{(k)})$$

Rather than accept $x^{(k)} + s^{(k)}$ as our next approximation we only accept $s^{(k)}$ as a search direction: We line search using our previous algorithm to find $\alpha^{(k)}$ that approximately minimizes

$$f(\alpha) = \|f(x^{(k)} + \alpha s^{(k)})\|$$

and put $x^{(k+1)} = x^{(k)} + \alpha^{(k)} s^{(k)}$

Note that $s^{(k)}$ is the steepest descent direction for minimizing $\|f(x)\|$. Method is called Newton's method with line search.

# Examples

- Illustrate interface and numerical differentiation with abstract function object nleqns_base in libscl.h and nleqns.cpp.

- Illustrate BFGS with examples in src/nlopt src/logistic/mle.

- Illustrate Newton's method with example in src/nlsolve.

- Illustrate BFGS from libscl with SNP.

- Illustrate BFGS from GSL (General Scientific Library) with SNP.

# Timing

We have studied various methods to time code to learn what methods are more efficient and to find where the most time is spent.

The next topic, profiling, is another way to find where time is being spent.

# Profiling

- Add -pg flag to compiler flags and linker flags.
- Do this for libraries that are to be included in the profile.
- Make the libraries, make the program, run the program, and then type: `gprof progname gmon.out > profile.out`
- Profile.out is a self annotated file with timing information.
- Illustrate with `nlreg` profile example.
- Recompile everything without -pg flag when finished because code compiled with the -pg runs about 30% longer.

# Particle Filters

In time series models that have latent variables (i.e. have unobservable variables), particle filters are used to

- Estimate the path of the latent variables using all the data; called smoothing.

- Estimate the path of the latent variables up to time $t$ using only information available at time $t$; called filtering.

- Integrate the unobservable variables out of the joint density of observable and unobservable variables to get the marginal density of the observable variables, which is the likelihood.

# Particle Filters – Sources


- Douced, Arnaud, Nando de Freitas, Neil Gordon, Editors (2001), *Sequential Monte Carlo Methods in Practice,* Springer, New York, ISBN 0 387 95146 6.

- Durham, Garland B. (2006), "Monte Carlo Methods for Estimating, Smoothing, and Filtering One and Two-Factor Stochastic Volatility Models". *Journal of Econometrics* 133, 273–305.

- Gallant, A. Ronald, Han Hong, Ahmed Khwaga (2011), "Bayesian Estimation of a Dynamic Game with Endogenous, Partially Observed, Serially Correlated State," Link on course web site.

# Particle Filters – Plan

- Introduce ideas with the simplest stochastic volatility model
  - ▷ Introduce algorithms
  - ▷ Sample code
- Present the general theory

# Stochastic Volatility Model – Shephard form

$$
\begin{aligned}
x_t &= \phi x_{t-1} + \sigma e_t \\
y_t &= \beta \exp(x_{t-1})\, u_t \\
e_t &\sim N(0,1) \\
u_t &\sim N(0,1) \\
x_0 &\sim N[0, \sigma^2/(1-\phi^2)] \\
(e_t, u_t) &\sim \text{iid} \\
\mathcal{E}(e_t u_t) &= \rho
\end{aligned}
$$

- Compatible with Euler discretization of continuous time models but does not agree with the notational conventions of the particle filter literature.

- Claimed to be better empirically by Yu, Jun, (2005), "On Leverage in a Stochastic Volatility Model," *Journal of Econometrics* 127, 165–178.

- In my view Yu's evidence is not persuasive.

# Stochastic Volatility Model – Polson form

$$\begin{aligned} x_t &= \phi x_{t-1} + \sigma e_t \\ y_t &= \beta \exp(x_t)\, u_t \\ e_t &\sim N(0,1) \\ u_t &\sim N(0,1) \\ x_0 &\sim N[0, \sigma^2/(1-\phi^2)] \\ (e_t, u_t) &\sim \text{iid} \\ \mathcal{E}(e_t u_t) &= \rho \end{aligned}$$

- Agrees with the notational conventions of the particle filter literature.
- When $\rho = 0$, the case of no leverage, it makes no difference whether one uses the Shephard or Polson forms.
- I will assume $\rho = 0$ for simplicity, which is a reasonable assumption for exchange rate data.

# Stochastic Volatility Model – Densities

$$
\begin{aligned}
x_0 &\sim n[0,\, \sigma^2/(1-\phi^2)] \\
x_t &\sim n[\phi x_{t-1},\, \sigma^2] \\
y_t &\sim n\{0,\, [\beta \exp(x_t)]^2\} \\
\theta &= (\phi, \sigma, \beta)
\end{aligned}
$$

# The Abstraction

| | |
|---|---|
| $x_0 \sim p(x_0 \mid \theta)$ | marginal density |
| $x_t \sim p(x_t \mid x_{t-1}, \theta)$ | transition density |
| $y_t \sim p(y_t \mid x_t, \theta)$ | measurement density |
| $y_{1:t} = \{y_1, \ldots, y_t\}$ | observed data |
| $x_{0:t} = \{x_0, \ldots, x_t\}$ | unobsered variables |

In many of the slides that follow we will suppress $\theta$ to save space.

# Generic Particle Filter Problem

$$\begin{aligned}
x_0 &\sim p(x_0) \\
x_t &\sim p(x_t|x_{t-1}) \\
y_t &\sim p(y_t|x_t) \\
y_{1:t} &= \{y_1, \ldots, y_t\} \\
x_{0:t} &= \{x_0, \ldots, x_t\}
\end{aligned}$$

Goals:

- Estimate the posterior $p(x_{0:t}|y_{1:t})$ recursively.
- Estimate the filtering distribution $p(x_t|y_{1:t})$ recursively.
- Approximate integrals of the form $\int p(y_{1:t}|x_{0:t})\, f_t(x_{0:t})\, dx_{0:t}$

# Particle Filter Algorithm

1. Initialization, $t = 0$.
   (a) For $i = 1, \ldots, N$ sample $x_0^{(i)}$ from $p(x_0)$ and set $t$ to 1.

2. Importance sampling step.
   (a) For $i = 1, \ldots, N$ sample $\tilde{x}_t^{(i)}$ from $p(x_t | x_{t-1}^{(i)})$ and set $\tilde{x}_{0:t}^{(i)} = (x_{0:t-1}^{(i)}, \tilde{x}_t^{(i)})$
   (b) For $i = 1, \ldots, N$ compute weights $\tilde{w}_t^{(i)} = p(y_t | \tilde{x}_t^{(i)})$
   (c) Normalize the weights.

3. Selection step
   (a) For $i = 1, \ldots, N$ sample with replacement the particles $x_{0:t}^{(i)}$ from the set $\{\tilde{x}_{0:t}^{(i)}\}$ according to the weights.
   (b) Increment $t$ and go to step 2

# Filtering and Smoothing

Smoothing conditions on all the data $y_{1:n}$. In smoothing one plots, integrates, etc. using draws $x_{1:t}$ from $p(x_{1:t}|y_{1:n})$.

Filtering conditions only on the data $y_{1:t}$ available at time $t$. In filtering, one plots, integrates, etc. using draws $x_{1:t}$ from $p(x_{1:t}|y_{1:t})$.

Sometimes one needs draws $x_{0:t}$ from $p(x_{0:t}|y_{1:t-1})$.

> Because $p(x_t|y_{1:t-1}) = \int p(x_t|x_{t-1})p(x_{t-1}|y_{1:t-1})\, dx_{t-1}$, one can draw $x_{0:t-1}$ from $p(x_{0:t-1}|y_{1:t-1})$ and then draw $\tilde{x}_t$ from $p(x_t|x_{t-1})$ to get a draw $\tilde{x}_{0:t} = (x_{0:t-1},\, \tilde{x}_t)$ from $p(x_{0:t}|y_{1:t-1})$.

Note that one has this draw at step 2a of the algorithm.

# Pointwise Likelihood Approximation

$$\begin{aligned}
p(y_{1:t}|\theta) &= \prod_{t=2}^{n} p(y_t|y_{1:t-1}, \theta) \\
&= \prod_{t=2}^{n} \int p(y_t|x_t, \theta)\, p(x_t|y_{1:t-1}, \theta)\, dx_t \\
&\doteq \prod_{t=2}^{n} \frac{1}{N} \sum_{i=1}^{N} p(y_t|\tilde{x}_t^{(i)}, \theta) \\
&= \prod_{t=2}^{n} \frac{1}{N} \sum_{i=1}^{N} \tilde{w}_t^{(i)}
\end{aligned}$$

where the particles are $\{\tilde{x}_{1:t}^{(i)}\}$ from step 2a and the weights are from step 2b.

The conditioning is on $y_{1:t-1}$, not $y_{1:t}$, see previous slide.

# Local Likelihood Approximation

$$
\begin{aligned}
p(y_{1:t}|\theta) &= \prod_{t=2}^{n} p(y_t|y_{1:t-1},\theta) \\
&= \prod_{t=2}^{n} \int p(y_t|x_t,\theta)\frac{p(x_t|y_{1:t-1},\theta)}{p(x_t|y_{1:t-1},\theta^o)}p(x_t|y_{1:t-1},\theta^o)\,dx_t \\
&\doteq \prod_{t=2}^{n} \frac{1}{N}\sum_{i=1}^{N} p(y_t|\tilde{x}_t^{(i)},\theta)\,\frac{p(\tilde{x}_t^{(i)}|y_{1:t-1},\theta)}{p(\tilde{x}_t^{(i)}|y_{1:t-1},\theta^o)} \\
&\doteq \prod_{t=2}^{n} \frac{1}{N}\sum_{i=1}^{N} p(y_t|\tilde{x}_t^{(i)},\theta) \quad \text{hopefully}
\end{aligned}
$$

where the particles $\{\tilde{x}_{1:t}^{(i)}\}$ are from $p(x_{1:t}|y_{1:t-1},\theta^o)$; i.e., step 2a.

To compute a mle, cycle between hill climbing with respect to $\theta$ and shifting $\theta^o$ closer to the current putative optimum.

# SV Particle Filter – 1 of 8

```
#ifndef __FILE_SVMOD_H_SEEN__
#define __FILE_SVMOD_H_SEEN__
#include "libscl.h"

struct sample {
  REAL x0;
  scl::realmat x;
  scl::realmat y;
  sample(INTEGER n) : x0(0.0), x(1,n), y(1,n) {}
};

class svmod {
private:
  REAL phi;
  REAL sigma;
  REAL beta;
public:
  svmod() : phi(0.9),sigma(0.5),beta(0.01) { }
  void set_parms(const scl::realmat& theta);
  scl::realmat get_parms() const;
  REAL draw_x0(INT_32BIT& seed) const;
  REAL draw_xt(REAL xlag, INT_32BIT& seed) const;
  REAL prob_yt(REAL yt, REAL xt) const;
  sample draw_sample(INTEGER n, INT_32BIT& seed) const;
};

#endif
```

# SV Particle Filter – 2 of 8

```
#include "libscl.h"
#include "svmod.h"

using namespace std;
using namespace scl;

REAL svmod::draw_x0(INT_32BIT& seed) const
{
  return (sigma/sqrt(1.0-phi*phi))*unsk(seed);
}

REAL svmod::draw_xt(REAL xlag, INT_32BIT& seed) const
{
  return phi*xlag + sigma*unsk(seed);
}

REAL svmod::prob_yt(REAL yt, REAL xt) const
{
  const REAL roottwopi = sqrt(6.283195307179587);
  REAL sd = beta*exp(xt);
  REAL z = yt/sd;
  return exp(-0.5*z*z)/(roottwopi*sd);
}
```

# SV Particle Filter – 3 of 8

```
sample svmod::draw_sample(INTEGER n, INT_32BIT& seed) const
{
  sample s(n);
  s.x0 = draw_x0(seed);
  REAL xlag = s.x0;
  for(INTEGER t=1; t<=n; ++t) {
    s.x[t] = draw_xt(xlag, seed);
    s.y[t] = beta*exp(s.x[t])*unsk(seed);
    xlag = s.x[t];
  }
  return s;
}
```

# SV Particle Filter – 4 of 8

```
#include "libscl.h"
#include "svmod.h"
using namespace scl;
using namespace std;

int main(int argc, char** argp, char** envp)
{
  INTEGER n = 100;
  INTEGER N = 5000;
  INT_32BIT seed = 780695;

  svmod m;

  sample s = m.draw_sample(n,seed);
```

# SV Particle Filter – 5 of 8

```
vector<realmat> smooth(N);
vector<realmat> filter(N);
vector<realmat> draws(N);
REAL weights[N];
REAL log_likelihood = 0.0;
```

# SV Particle Filter – 6 of 8

```
// Initialization

realmat y(n+1,1);
y[1] = 0.0;
for (INTEGER t=2; t<=n+1; t++) {
  y[t] = s.y[t-1];
}

REAL sum = 0.0;
realmat x(n+1,1);
for (INTEGER i=0; i<N; ++i) {
  x[1] = m.draw_x0(seed);
  smooth[i] = filter[i] = draws[i] = x;
}
```

# SV Particle Filter – 7 of 8

```
for (INTEGER t=2; t<=n+1; ++t) {

  // Importance sampling step

  sum = 0.0;
  for (INTEGER i=0; i<N; ++i) {
    draws[i][t] = m.draw_xt(draws[i][t-1],seed);
    sum += weights[i] = m.prob_yt(y[t],draws[i][t]);
  }
  log_likelihood += log(sum/REAL(N));

  for (INTEGER i=1; i<N; ++i)  weights[i] += weights[i-1];
  for (INTEGER i=0; i<N; ++i)  weights[i] /= sum;

  // Selection step

  for (INTEGER i=0; i<N; ++i) {
    REAL u = ran(seed);
    INTEGER j = 0;  while(weights[j] <= u) ++j;
    smooth[i] = draws[j];
    filter[i][t] = draws[j][t];
  }
  draws = smooth;
}
```

# Comment on Previous Slide

The assignment

```
draws = smooth
```

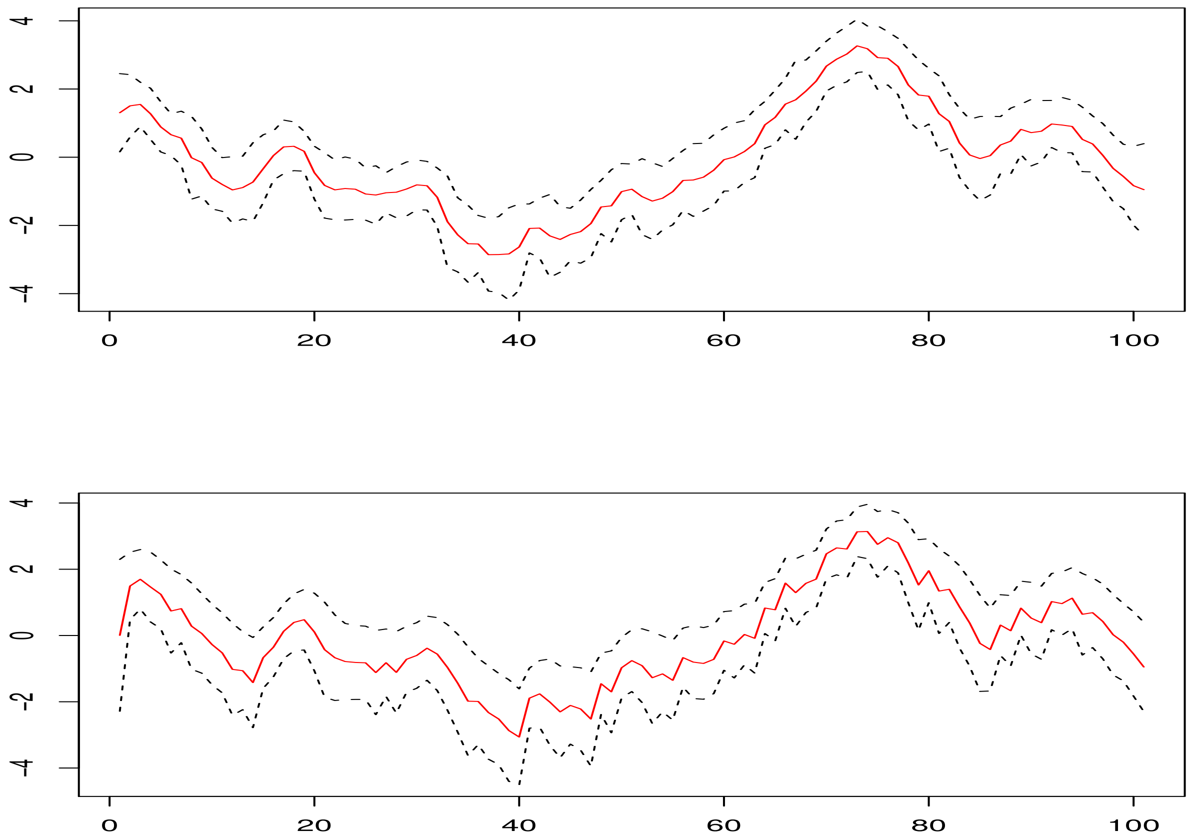
in the last line of the previous slide is costly and can be avoided using pointers.

See `particle_fast.cpp` at the course website.

# SV Particle Filter – 8 of 8

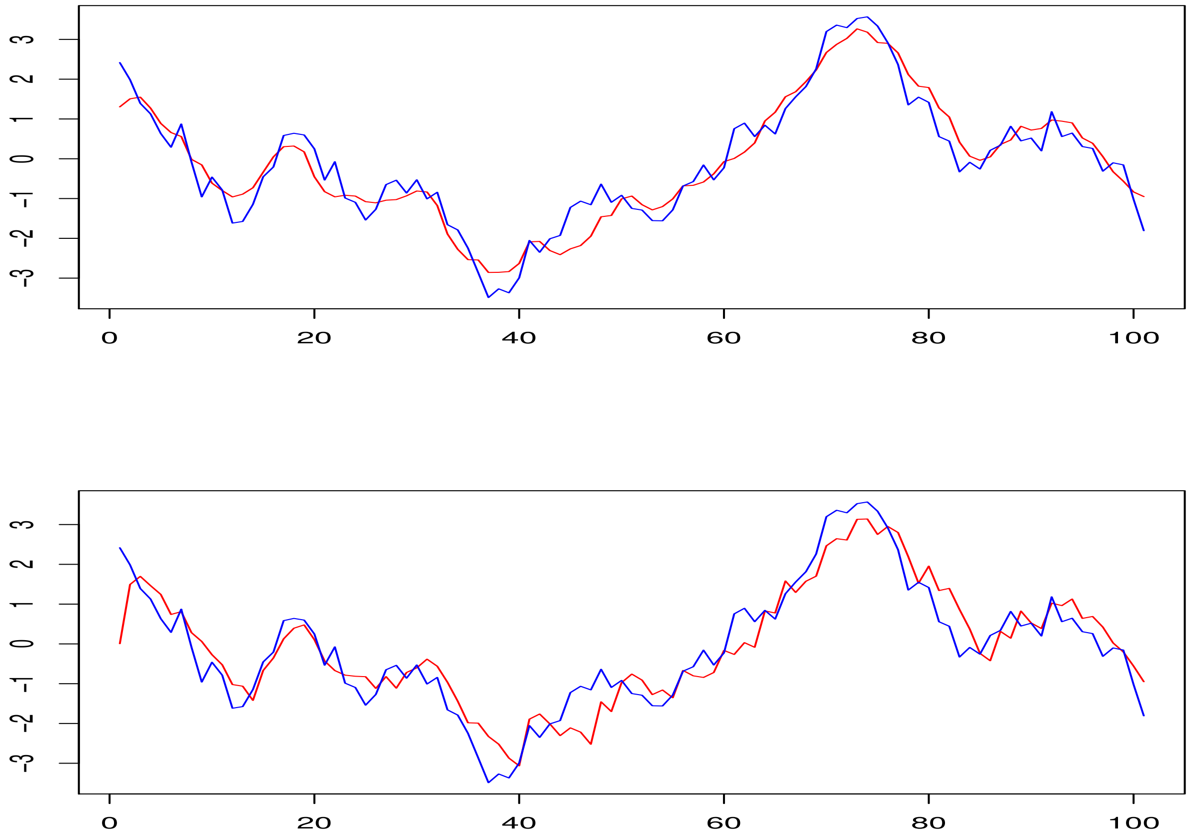
```
  realmat mean(n+1,1,0.0);
  for (INTEGER i=0; i<N; ++i) {
    mean += smooth[i];
  }
  mean = mean/N;

  realmat sdev(n+1,1,0.0);
  for (INTEGER i=0; i<N; ++i) {
    realmat z = smooth[i] - mean;
    for (INTEGER t=1; t<=n+1; ++t) sdev[t] += z[t]*z[t];
  }
  for (INTEGER t=1; t<=n+1; ++t) sdev[t] = sqrt(sdev[t]/REAL(N-1));

  ofstream fout("smooth.csv");
  if (!fout) error("Error, particle, cannot open fout");

  fout << "mean, sdev, x, y" << '\n';
  fout << mean[1] <<','<< sdev[1] <<','<< s.x0 <<','<< 0 <<'\n';
  for (INTEGER t=2; t<=n+1; ++t) {
    fout<<mean[t]<<','<<sdev[t]<<','<<s.x[t-1]<<','<<s.y[t-1]<<'\n';
  }

\\ similar code for filter

  cout << "The log likelihood is " << log_likelihood << '\n';
  return 0;
}
```

# SV Particle Filter Results – 1 of 2




The top panel is a smooth, the bottom a filter. The solid red line is the mean of the volatility particles. The dashed black lines are plus and minus two standard deviations of the particles.

# SV Particle Filter Results – 2 of 2




The top panel is a smooth, the bottom a filter. The red line is the mean of the volatility particles. The blue line is the true volatility path.

# Why Does This Work?

- It is an application of importance sampling applied sequentially with a particular choice of importance function that simplifies the algebra.

- The selection step corrects for a defect in importance sampling that arises when it is applied sequentially.

- Details follow.

- In what follows, the hidden Markov state $x$ can be either scalar or vector and the same for the observation $y$.

# Importance Sampling

We want to compute the integral $\int f(x)p(x)\,dx$, where $p(x)$ is a density function with support $\mathcal{X} \subset \Re^d$ .

Suppose we can find a density function $\pi(x)$ whose support includes $\mathcal{X}$ from which we can draw a sample $x^{(1)}, \ldots, x^{(N)}$.

Then

$$\int f(x)p(x)\,dx = \int f(x)w(x)\pi(x)\,dx$$

where

$$w(x) = \frac{p(x)}{\pi(x)}$$

whence

$$\int f(x)p(x)\,dx \doteq \frac{1}{N}\sum_{i=1}^{N} f(x^{(i)})w(x^{(i)})$$

## For Importance Sampling to Work Well

- The variance with respect to $\pi(x)$ of $f(x)w(x)$ must be small.
  - ▷ To satisfy this requirement for general $f(x)$ one tries to make the variance of $w(x)$ small.
  - ▷ Making the variance of $w(x)$ small is usually accomplished by making sure that $\pi(x)$ has fatter tails than $p(x)$ .
  - ▷ For example, if $p(x)$ has exponential tails like $\exp^{-x'\Sigma^{-1}x/2}$, then choose $\pi(x)$ to be a density with polynomial tails like the multivariate $t$-distribution.
- Importance sampling is hard to get to work well when the dimension of $x$ is large because it is hard to draw a sample that lands where $f(x)w(x)$ is large.
  - ▷ This is also the reason that move-one-at-a-time MCMC often works better than independence MCMC or Gibbs or group-move.

# Normalized Weights

Let

$$\widehat{w}^{(i)} = \frac{w(x^{(i)})}{\sum_{t=i}^{N} w(x^{(i)})}$$

then

$$\begin{aligned}
\int f(x)\, p(x)\, dx &= \frac{\int f(x) w(x) \pi(x)\, dx}{\int w(x)\pi(x)\, dx} \\
&\doteq \frac{\frac{1}{N}\sum_{i=1}^{N} f(x^{(i)}) w(x^{(i)})}{\frac{1}{N}\sum_{i=1}^{N} w(x^{(i)})} \\
&= \sum_{i=1}^{N} \widehat{w}^{(i)} f(x^{(i)})
\end{aligned}$$

# Why are Normalized Weights Relevant?

$$\sum_{i=1}^{N} \widehat{w}^{(i)} = 1 \ \& \ \int f(x)\, p(x)\, dx \doteq \sum_{i=1}^{N} \widehat{w}^{(i)} f(x^{(i)}),$$

imply that we are integrating $f(x)$ with respect to a discrete distribution that puts probability $\widehat{w}^{(i)}$ on the point $x^{(i)}$.

That means we could alternatively generate a random sample $\{\widehat{x}^{(i)}\}$ from this discrete distribution and use the formula

$$\int f(x) p(x)\, dx \doteq \frac{1}{N} \sum_{i=1}^{N} f(\widehat{x}^{(i)})$$

A random sample can be generated by sampling the points $\{\widehat{x}^{(i)}\}$ with replacement with probability $\widehat{w}^{(i)}$.

# What Are We Trying to Compute?

For the density

$$p(x_{0:t}|y_{1:t}) = \frac{p(y_{1:t}|x_{0:t})p(x_{0:t})}{\int p(y_{1:t}|x_{0:t})p(x_{0:t})\, dx_{0:t}}$$

we want to compute importance weights $\tilde{w}_t^{(i)}$ and particles $\{\tilde{x}_{0:t}^{(i)}\}$ recursively.

Recursively means that one has available at time $t$ the weights $w_{t-1}^{(i)}$ (usually $= 1/N$ due to resampling) and particles $\{x_{0:t-1}^{(i)}\}$ from time $t-1$ and the observed $y_t$ and uses cheap updating formulas to compute the time $t$ weights $\tilde{w}_t^{(i)}$ and particles $\{\tilde{x}_{0:t}^{(i)}\}$.

# Recursive Importance Sampling Theorem

If the importance function $\pi(x_{0:t}|y_{1:t})$ for $p(x_{0:t}|y_{1:t})$ factors as

$$\pi(x_{0:t}|y_{1:t}) = \pi(x_t|x_{0:t-1}, y_{1:t})\pi(x_{0:t-1}|y_{1:t-1})$$

then one can draw $x_t^{(i)}$ from $\pi(x_t|x_{0:t-1}^{(i)}, y_{1:t})$, put

$$x_{1:t}^{(i)} = (x_{0:t-1}^{(i)}, x_t^{(i)}),$$

and use weights

$$\tilde{w}_t^{(i)} \propto \tilde{w}_{t-1}^{(i)} \frac{p(y_t|x_t^{(i)})p(x_t^{(i)}|x_{t-1}^{(i)})}{\pi(x_t^{(i)}|x_{0:t-1}^{(i)}, y_{1:t})}$$

A special case of this is the “blind sampler”

$$\pi(x_{0:t}|y_{1:t}) = p(x_{0:t}) = p(x_0) \prod_{k=1}^{t} p(x_k|x_{k-1}),$$

which is the choice in the particle filter algorithm.

# Why Follow It with a Selection Step?

When importance sampling is done recursively what happens as $t$ increases is that a few of the weights $\tilde{w}_t^{(i)}$ increasingly dominate so that most particles effectively die out.

What resampling does is eliminate some of the particles that have negligible weight, replace them with particles that have larger weight, and adjust the weights. The result is more equal weights.

We described what is called bootstrap or multinomial selection. After completing a bootstrap selection, each particle has weight $1/N$.

Other resampling schemes such as stratified sampling or systematic sampling are sometimes used.

# Recursion Verification – 1 of 3

Factorization of the importance function as

$$\pi(x_{0:t}|y_{1:t}) = \pi(x_t|x_{0:t-1}, y_{1:t})\pi(x_{0:t-1}|y_{1:t-1})$$

implies the recursion

$$\pi(x_{0:t}|y_{1:t}) = \pi(x_0) \prod_{k=1}^{t} \pi(x_k|x_{0:k-1}, y_{1:k})$$

and conversely. Factorization (or the recursion) implies that one can draw $x_t^{(i)}$ from $\pi(x_t|x_{0:t-1}^{(i)}, y_{1:t})$ and put $x_{1:t}^{(i)} = (x_{0:t-1}^{(i)}, x_t^{(i)})$.

The following fact, verified next slide, is needed to establish the recursion for $\tilde{w}^{(i)}$

$$p(x_{0:t}|y_{1:t}) \;=\; p(x_{0:t-1}|y_{1:t-1})\frac{p(y_t|x_t)p(x_t|x_{t-1})}{p(y_t|y_{1:t-1})}$$

# Recursion Verification – 2 of 3

$$
\begin{aligned}
p(x_{0:t}|y_{1:t}) &= \frac{p(x_{0:t-1}, y_{1:t-1})p(x_t, y_t|x_{0:t-1}, y_{1:t-1})}{p(y_{1:t})} \\
&= \frac{p(x_{0:t-1}, y_{1:t-1})p(x_t, y_t|x_{0:t-1})}{p(y_{1:t})} \\
&= \frac{p(x_{0:t-1}, y_{1:t-1})p(x_{0:t}, y_t)}{p(y_{1:t})p(x_{0:t-1})} \\
&= \frac{p(x_{0:t-1}, y_{1:t-1})p(x_{0:t}, y_t)p(y_{1:t-1})p(x_{0:t})}{p(y_{1:t})p(x_{0:t-1})p(y_{1:t-1})p(x_{0:t})} \\
&= \frac{p(x_{0:t-1}|y_{1:t-1})p(y_t|x_{0:t})p(x_t|x_{0:t-1})}{p(y_t|y_{1:t-1})} \\
&= p(x_{0:t-1}|y_{1:t-1})\frac{p(y_t|x_t)p(x_t|x_{t-1})}{p(y_t|y_{1:t-1})}
\end{aligned}
$$

The second step is because the distribution of $(y_t, x_t)$ depends only on $x_t$. The last is because $x_t$ is Markovian.

# Recursion Verification – 3 of 3

$$
\begin{aligned}
\tilde{w}_t &= \frac{p(x_{0:t}|y_{1:t})}{\pi(x_{0:t}|y_{1:t})} \\
&= \frac{p(x_{0:t}|y_{1:t})}{\pi(x_{0:t-1}|y_{1:t-1})\pi(x_t|x_{0:t-1}, y_{1:t})} \\
&= \tilde{w}_{t-1}\frac{p(x_{0:t}|y_{1:t})}{p(x_{0:t-1}|y_{1:t-1})\pi(x_t|x_{0:t-1}, y_{1:t})} \\
&= \tilde{w}_{t-1}\frac{p(x_{0:t-1}|y_{1:t-1})p(y_t|x_t)p(x_t|x_{t-1})}{p(x_{0:t-1}|y_{1:t-1})\pi(x_t|x_{0:t-1}, y_{1:t})p(y_t|y_{1:t-1})} \\
&= \tilde{w}_{t-1}\frac{p(y_t|x_t)p(x_t|x_{t-1})}{\pi(x_t|x_{0:t-1}, y_{1:t})p(y_t|y_{1:t-1})}
\end{aligned}
$$

The second step uses the factorization assumption. The fourth step uses the result from the previous slide. The term $p(y_t|y_{1:t-1})$ in the denominator of the last equation does not involve $x$ and therefore drops out when the weights are normalized.

# The Particle Filter Algorithm Once Again

1. Initialization, $t = 0$.
   - For $i = 1, \ldots, N$ sample $x_0^{(i)}$ from $p(x_0)$ and set $t$ to 1.

2. Importance sampling step.
   - For $i = 1, \ldots, N$ sample $\tilde{x}_t^{(i)}$ from $p(x_t | x_{t-1}^{(i)})$ and set $\tilde{x}_{0:t}^{(i)} = (x_{0:t-1}^{(i)}, \tilde{x}_t^{(i)})$ .
   - For $i = 1, \ldots, N$ compute weights $\tilde{w}_t^{(i)} = p(y_t | \tilde{x}_t^{(i)})$ .
   - Normalize the weights.

3. Selection step
   - For $i = 1, \ldots, N$ sample with replacement the particles $x_{0:t}^{(i)}$ from the set $\{\tilde{x}_{0:t}^{(i)}\}$ according to the weights.
   - Increment $t$ and go to step 2

# Two Factor SV Model with Leverage

A two factor Shephard timing SV model with leverage.

$$
\begin{aligned}
y_t &= \alpha_0 + \exp(v_{1,t-1} + v_{2,t-1})(r_{31}e_{1t} + r_{32}e_{2t} + r_{33}e_{3t}) \\
v_{1t} &= \beta_0 + \beta_{11}v_{1,t-1} + r_{11}e_{1t} \\
v_{2t} &= \beta_0 + \beta_{22}v_{2,t-1} + r_{21}e_{1t} + r_{22}e_{2t}
\end{aligned}
$$

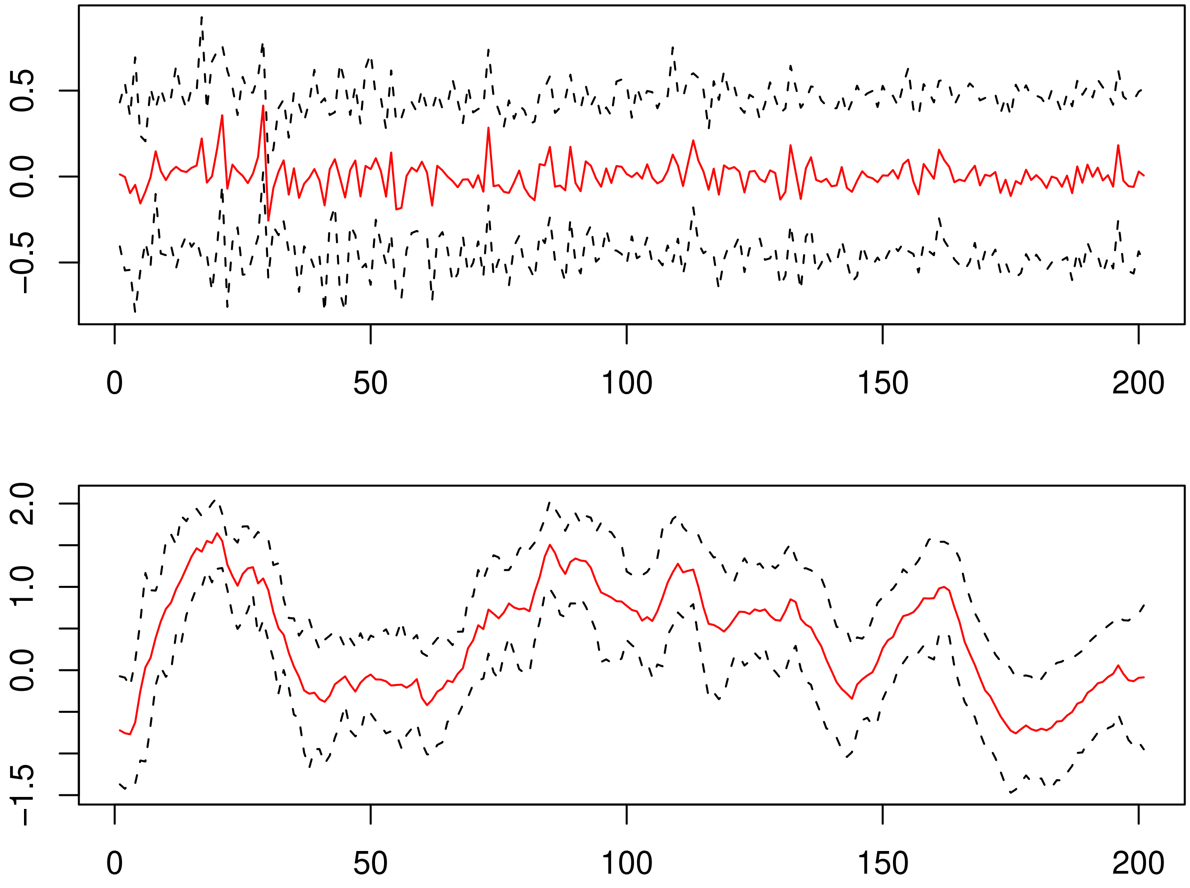
# Two Factor SV Model in State Space Form

This is the Shephard timing two factor SV model with leverage put into the form required for a particle filter.

$$
\begin{aligned}
x_{1t} &= \beta_0 + \beta_{11} x_{1,t-1} + r_{11} e_{1t} \\
x_{2t} &= \beta_0 + \beta_{22} x_{2,t-1} + r_{21} e_{1t} + r_{22} e_{2t} \\
x_{3t} &= x_{1,t-1} \\
x_{4t} &= x_{2,t-1} \\
y_t &= \alpha_0 + \exp(x_{3t} + x_{4t})(r_{31} e_{1t} + r_{32} e_{2t} + r_{33} e_{3t})
\end{aligned}
$$

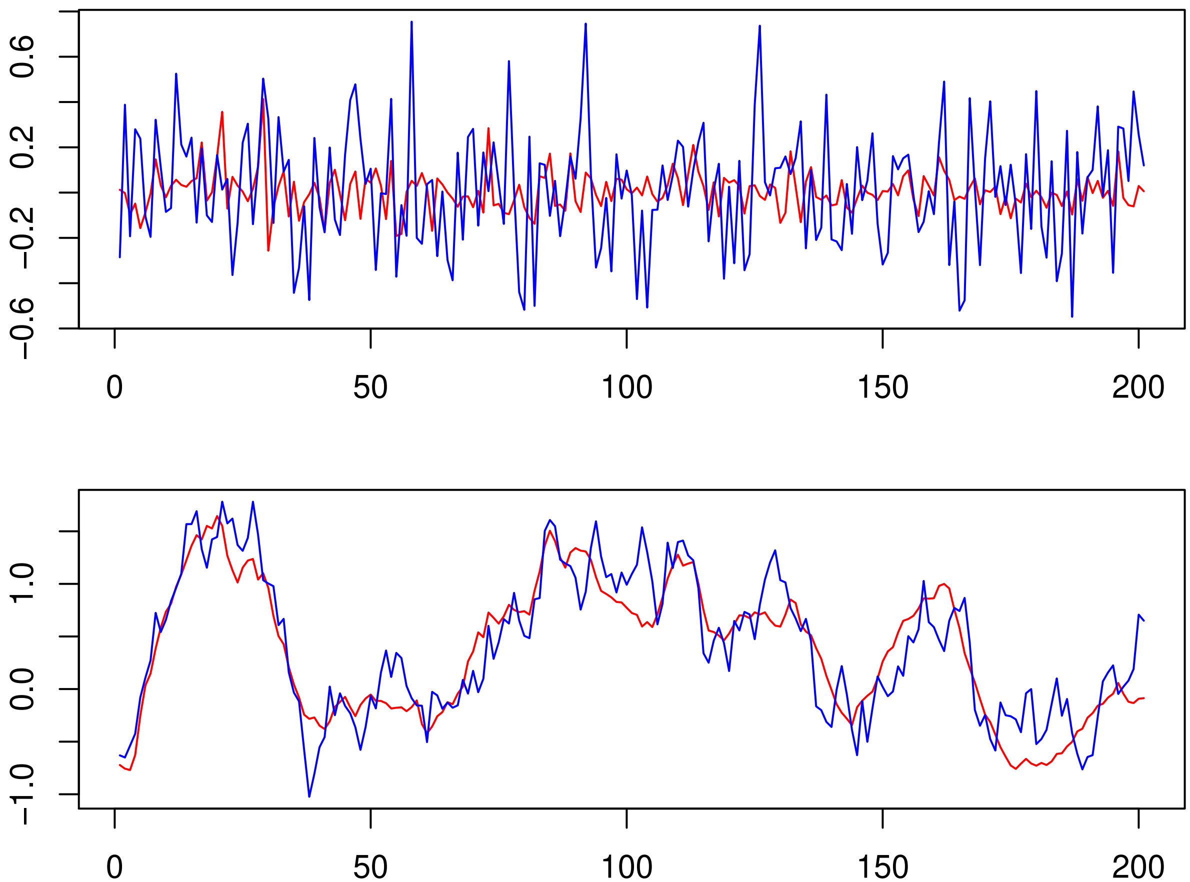
Run example in `src/sv2fac`.

# 2F SV Particle Filter Results – 1 of 2




The solid red line is the mean of the volatility particles. The dashed black lines are plus and minus two standard deviations of the particles.

# 2F SV Particle Filter Results – 2 of 2




The red line is the mean of the volatility particles. The blue line is the true volatility path.

# Generalizations and Extensions

Durham, Garland B. (2006) "Monte Carlo Methods for Estimating, Smoothing, and Filtering One and Two-Factor Stochastic Volatility Models". *Journal of Econometrics* 133, 273–305.

Easy to read general discussion of how to estimate stochastic volatility models that are far more general than discussed here.

Also includes a discussion of how to implement particle filters for problems that do not fit easily within the "cannonical form" of the problem as discussed here.

A preprint is at the course website. This corrects an error on page 17.

$$w^{(s)} \propto p(x_{t+1} | v_{t+1}^{(s)}, v_t^{(s)}, \mathcal{F}_t)$$

# One Lag, Two Factor SV Model

A one lag, two factor, Shephard timing SV model with leverage.

$$
\begin{aligned}
y_t &= \alpha_0 + \alpha_1 y_{t-1} \\
&\quad + \exp(v_{1,t-1} + v_{2,t-1})(r_{31}e_{1t} + r_{32}e_{2t} + r_{33}e_{3t}) \\
v_{1t} &= \beta_0 + \beta_{11} v_{1,t-1} + r_{11}e_{1t} \\
v_{2t} &= \beta_0 + \beta_{22} v_{2,t-1} + r_{21}e_{1t} + r_{22}e_{2t}
\end{aligned}
$$

The Durham version of particle filtering does not presume state space form. Lags are explicitly expressed in the densities instead. Specifically, for this model, $v_t = (v_{1t}, v_{2t})$, $v_t$ depends on $v_{t-1}$, and $y_t$ depends on $(y_{t-1}, v_t, v_{t-1})$. Sample code is in `svlag2fac`.

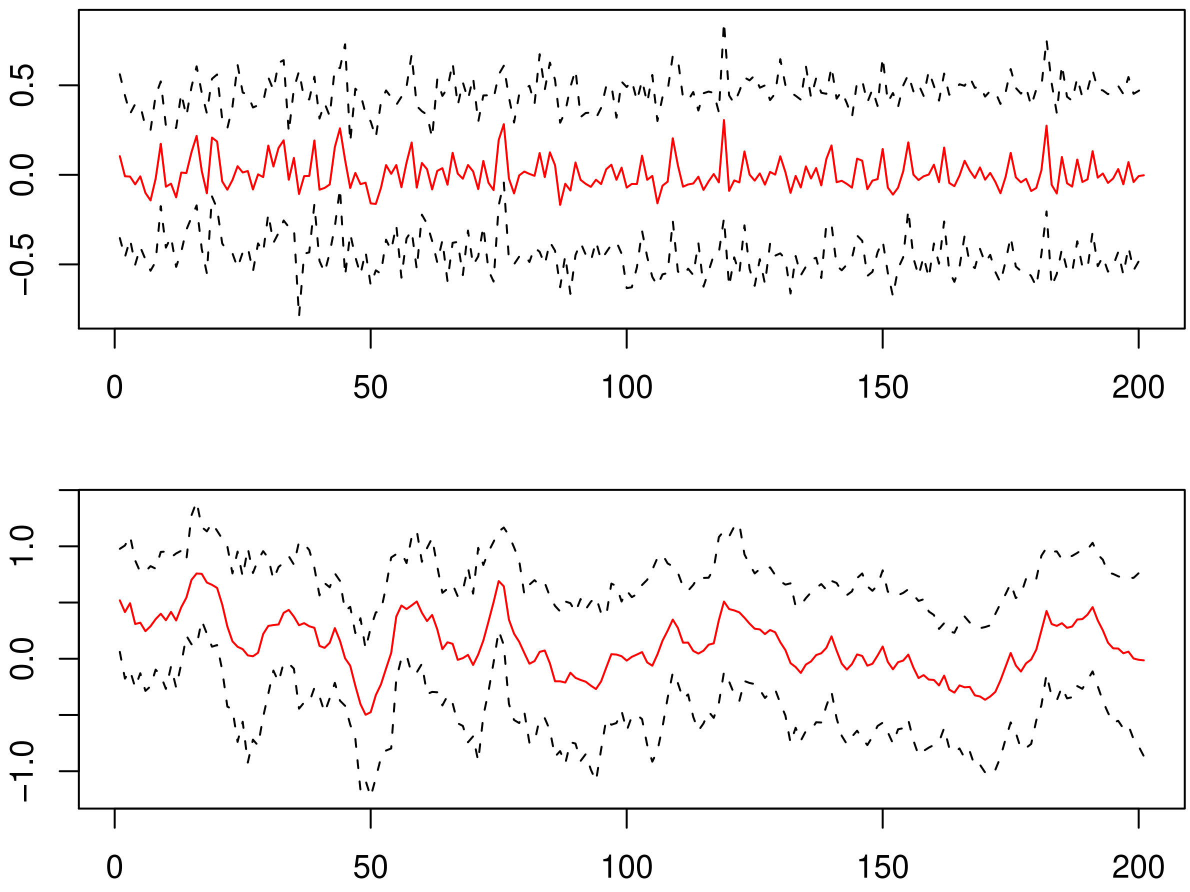
## Algorithm for One Lag, Two Factor SV Model

1. Initialization, $t = 0$.
   - For $i = 1, \ldots, N$ sample $(y_0^{(i)}, v_1^{(i)}, v_0^{(i)})$ and set $t$ to 1.

2. Importance sampling step.
   - For $i = 1, \ldots, N$ sample $\tilde{v}_t^{(i)}$ from $p(v_t | v_{t-1}^{(i)})$ and set $\tilde{v}_{0:t}^{(i)} = (v_{0:t-1}^{(i)}, \tilde{v}_t^{(i)})$ .
   - For $i = 1, \ldots, N$ compute weights $\tilde{w}_t^{(i)} = p(y_t | y_{t-1}, \tilde{v}_{t-1}^{(i)})$.
   - Normalize the weights.

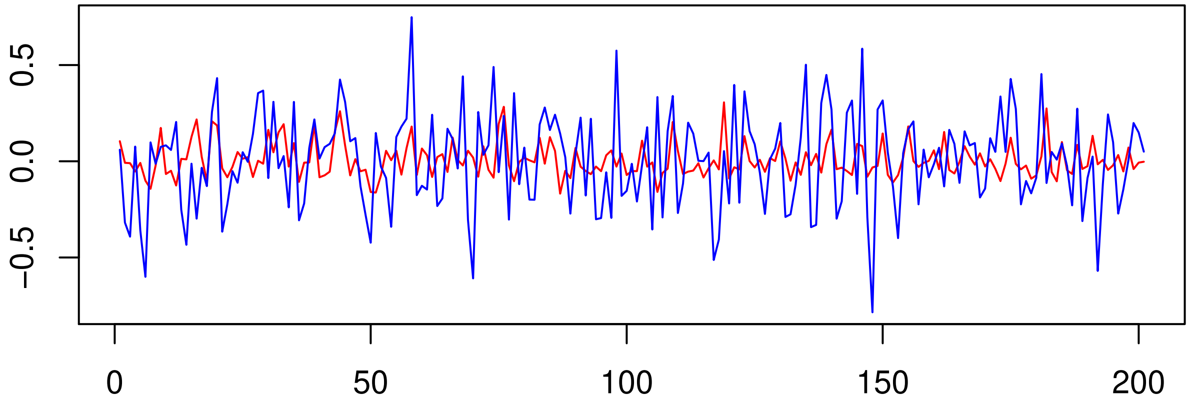
3. Selection step
   - For $i = 1, \ldots, N$ sample with replacement the particles $v_{0:t}^{(i)}$ from the set $\{\tilde{v}_{0:t}^{(i)}\}$ according to the weights.
   - Increment $t$ and go to step 2

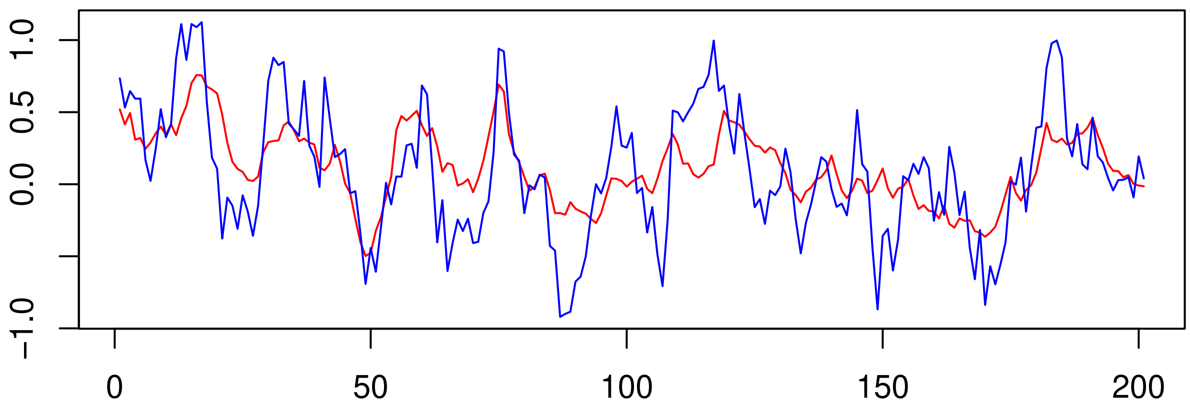
# 2F 1L SV Particle Filter Results – 1 of 2




The solid red line is the mean of the volatility particles. The dashed black lines are plus and minus two standard deviations of the particles.

# 2F 1L SV Particle Filter Results – 2 of 2




The red line is the mean of the volatility particles. The blue line is the true volatility path.

# Debugging

Illustrate use of `cerr` in debugging.