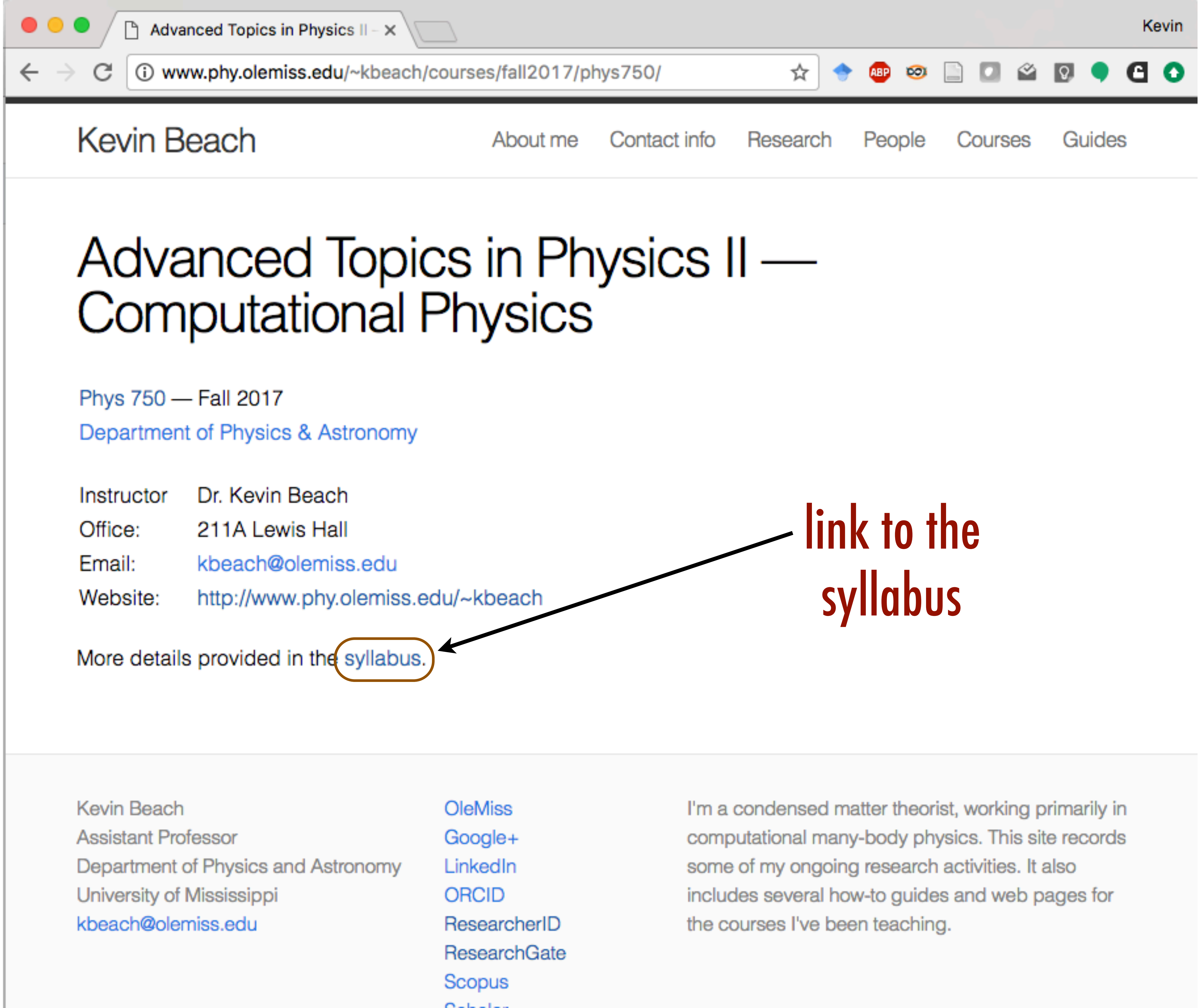
Advanced Topics In Physics II – Computational Physics

Phys 750 Lecture 1



Advanced Topics in Physics II –

www.phy.olemiss.edu/~kbeach/courses/fall2017/phys750/

Kevin

Kevin Beach

About me Contact info Research People Courses Guides

Advanced Topics in Physics II — Computational Physics

Phys 750 — Fall 2017
Department of Physics & Astronomy

Instructor	Dr. Kevin Beach
Office:	211A Lewis Hall
Email:	kbeach@olemiss.edu
Website:	http://www.phy.olemiss.edu/~kbeach

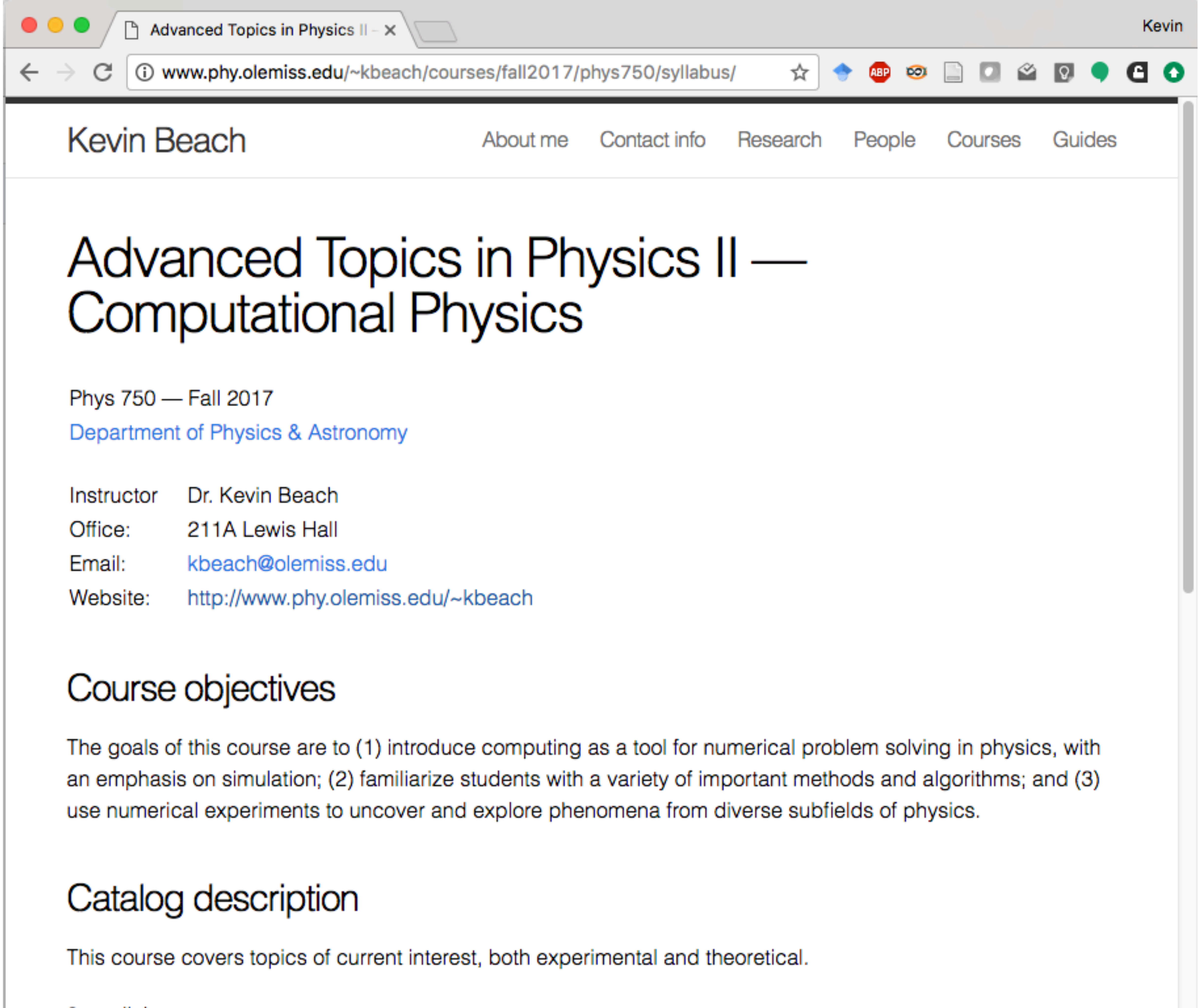
More details provided in the syllabus.

link to the syllabus

Kevin Beach
Assistant Professor
Department of Physics and Astronomy
University of Mississippi
kbeach@olemiss.edu

OleMiss
Google+
LinkedIn
ORCID
ResearcherID
ResearchGate
Scopus

I'm a condensed matter theorist, working primarily in computational many-body physics. This site records some of my ongoing research activities. It also includes several how-to guides and web pages for the courses I've been teaching.



Advanced Topics in Physics II – ×

www.phy.olemiss.edu/~kbeach/courses/fall2017/phys750/syllabus/

Kevin

Kevin Beach

About me Contact info Research People Courses Guides

Advanced Topics in Physics II — Computational Physics

Phys 750 — Fall 2017

Department of Physics & Astronomy

Instructor	Dr. Kevin Beach
Office:	211A Lewis Hall
Email:	kbeach@olemiss.edu
Website:	http://www.phy.olemiss.edu/~kbeach

Course objectives

The goals of this course are to (1) introduce computing as a tool for numerical problem solving in physics, with an emphasis on simulation; (2) familiarize students with a variety of important methods and algorithms; and (3) use numerical experiments to uncover and explore phenomena from diverse subfields of physics.

Catalog description

This course covers topics of current interest, both experimental and theoretical.

Tools and skills

- Basic programming in C++14, using the gcc or llvm compilers
- Standard, freely available tools (e.g., make, gnuplot, LaTeX, ...)
- Skeleton codes designed for UNIX or MacOS
- Assignment submission via Bitbucket: please sign up for a free academic account at bitbucket.org using your olemiss.edu email address

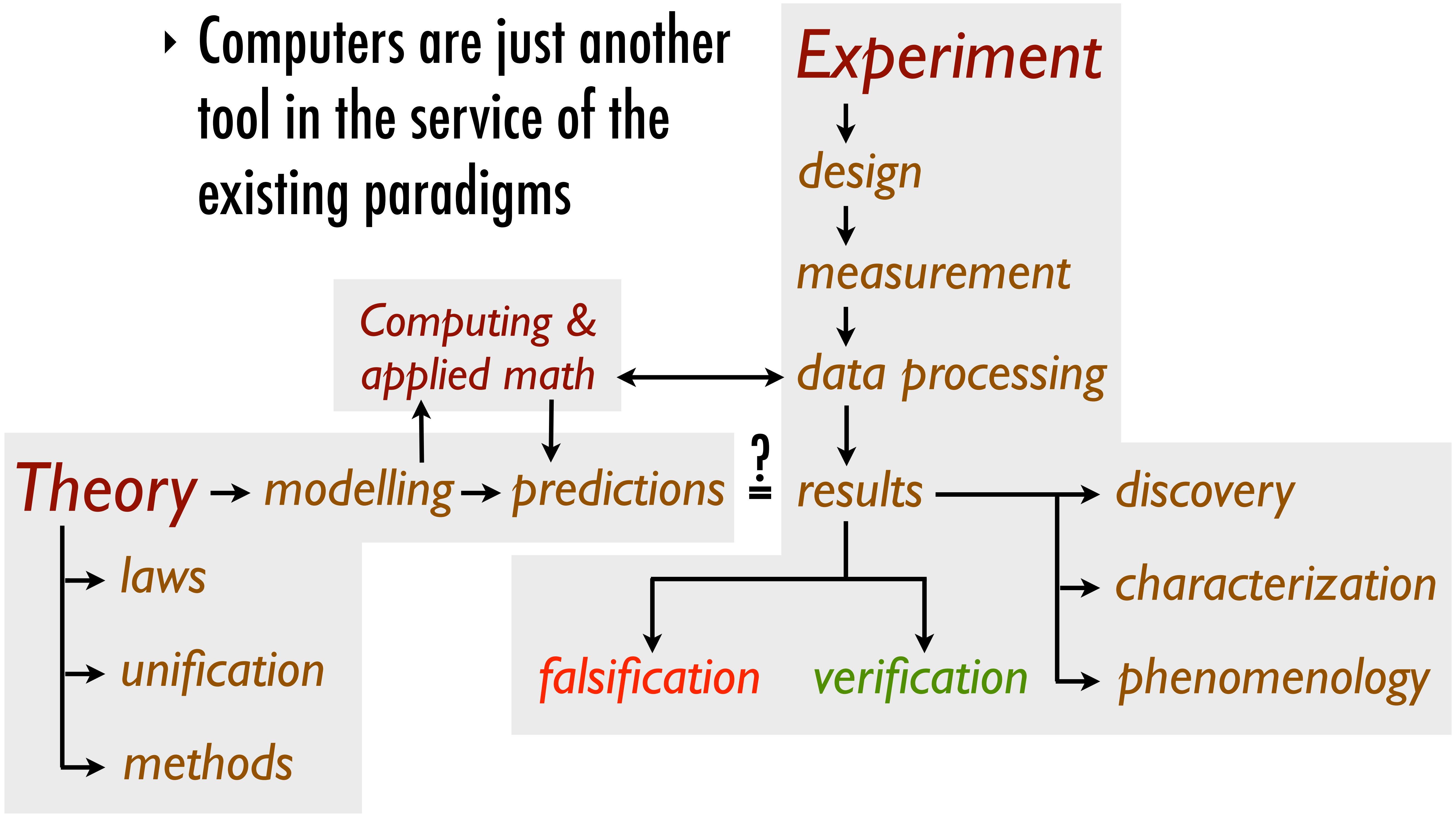
Computational physics

What is computational physics?

- A branch of physics in its own right and an important bridge between theory and experiment
- Concerned with the development and implementation of numerical algorithms that can simulate complex physical behaviour
- Employed extensively in almost every physics discipline

The purely instrumental view

- Computers are just another tool in the service of the existing paradigms



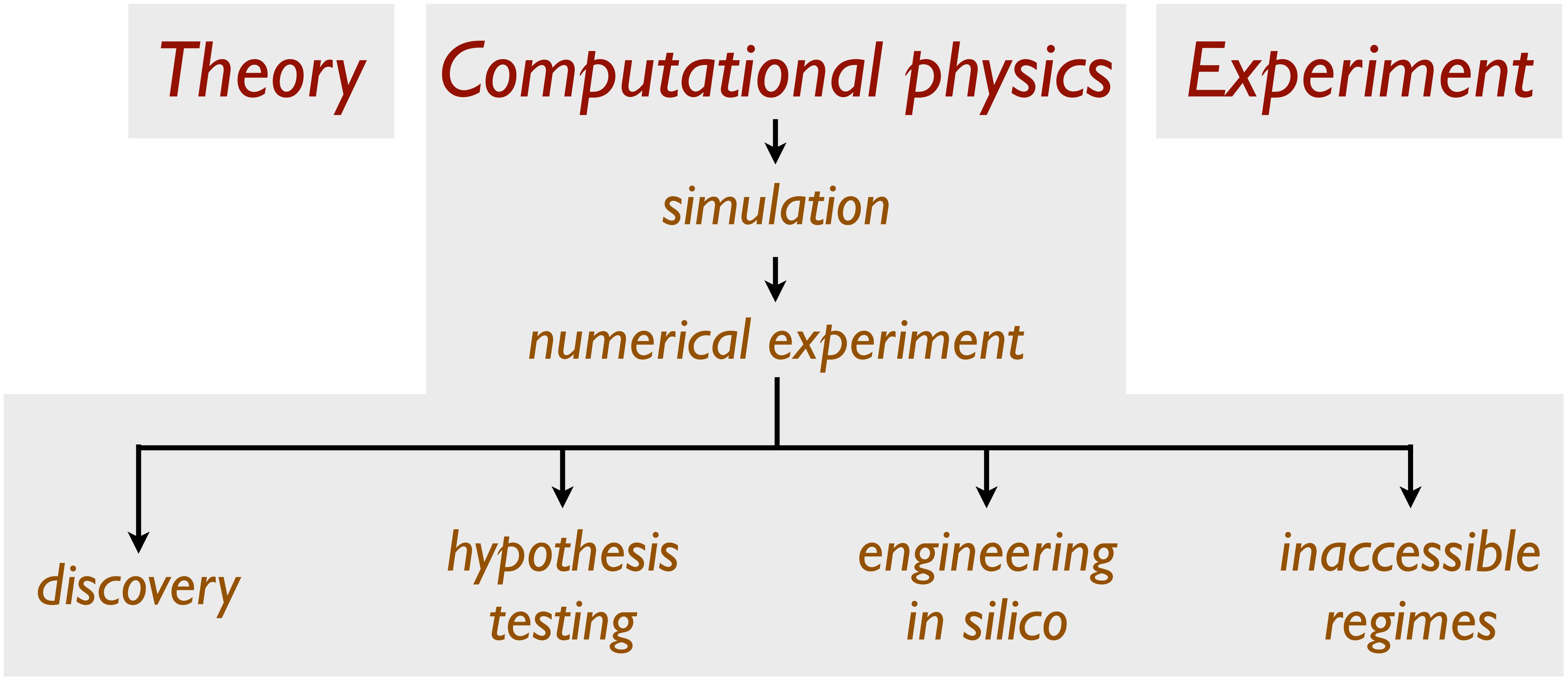
The purely instrumental view

- Contact between theory and experiment relies on our making quantitative comparisons
- The computer plays a supporting role:
  - treat by numerical methods models that cannot be solved analytically

    Wait! Can we solve any models by hand?
  - manipulate experimental data (background subtraction, Fourier analysis, etc.)

Is it always necessary (or even possible) to perform an experiment?

A richer view



e.g., chaos, fractals, strange attractors

e.g., quark confinement in QCD

e.g., designer materials like graphane

e.g., metallic hydrogen at ultra-high pressure

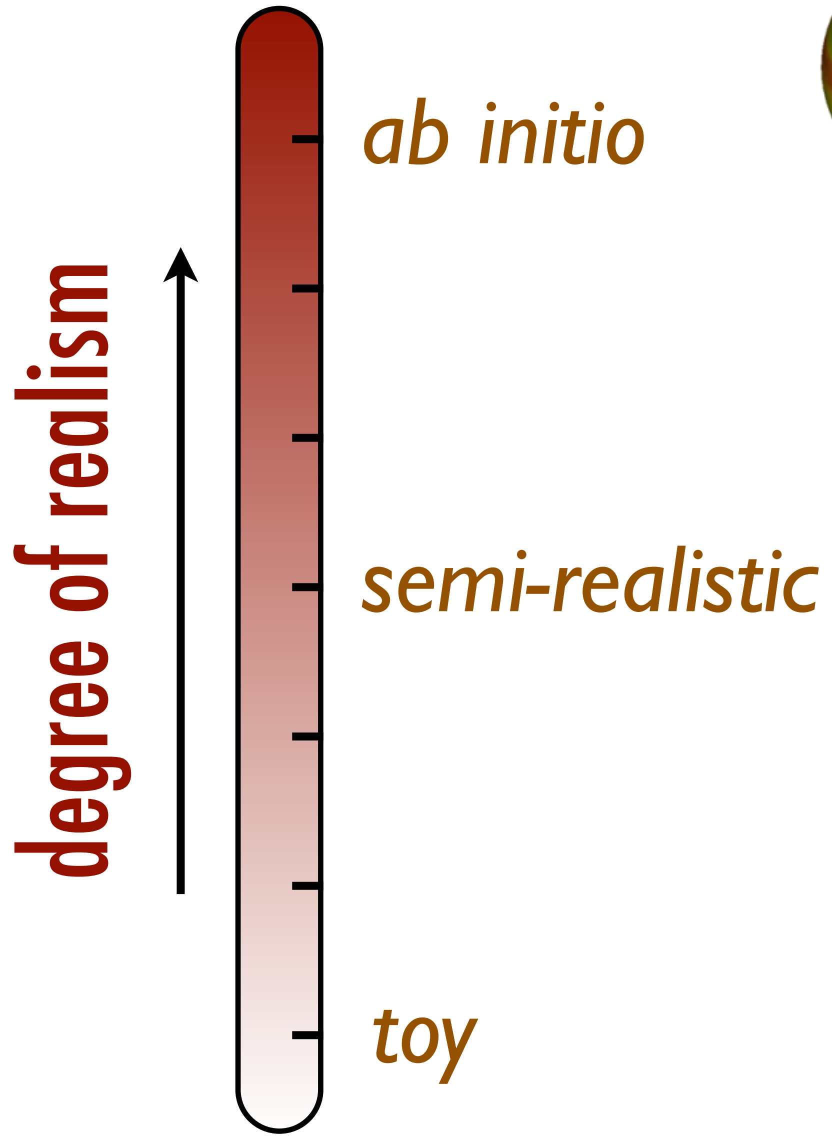
A richer view

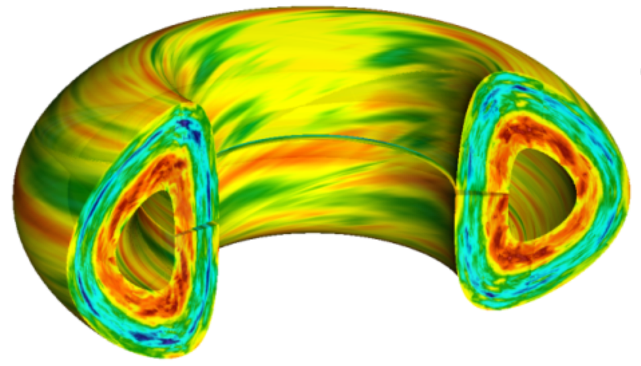
- Computational physics seen as a co-equal branch of physics, complementary to theory and experiment
- Draws on lessons from computer science and applied mathematics but applies insights from physics
- Many problems can be addressed by no other means: Wigner crystals, plasma near fusion ignition, stellar interiors, colliding black holes, galaxy formation, undiscovered elements, climate models, ...

Important issues

- How do we make physics come alive in the machine?
- What degree of realism is necessary in our modelling?
- How do we design algorithms?
- What are the key considerations for efficiency of storage and execution time?
- How do we apply insights from physics?

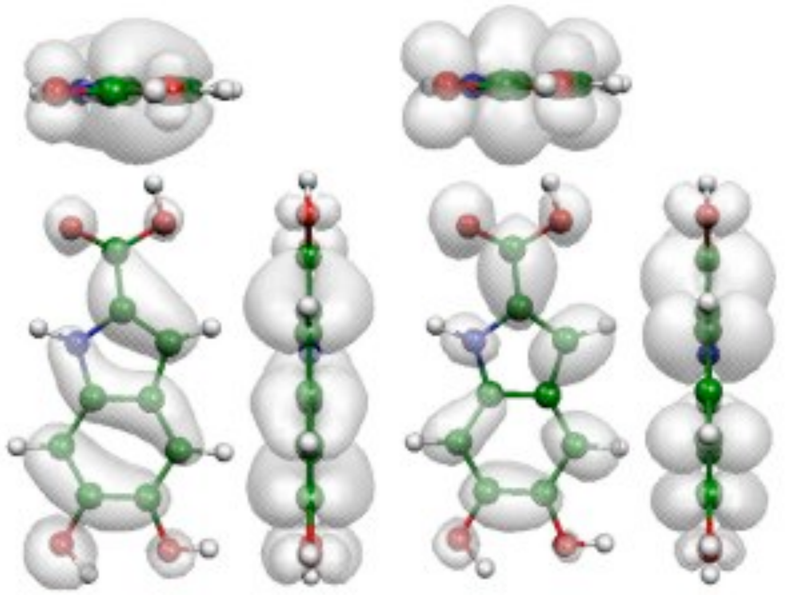
Degree of realism



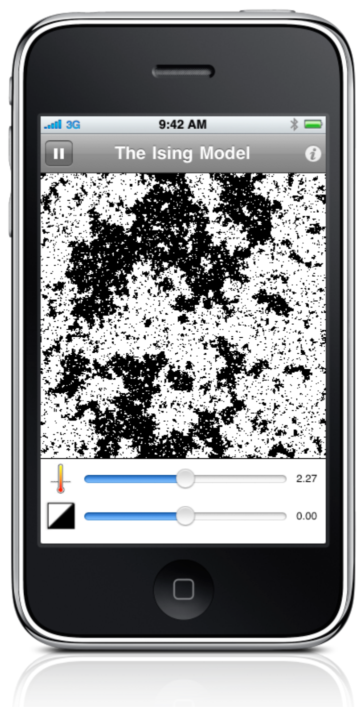


e.g., tokamak plasma

e.g., DFT quantum chemistry



computational complexity



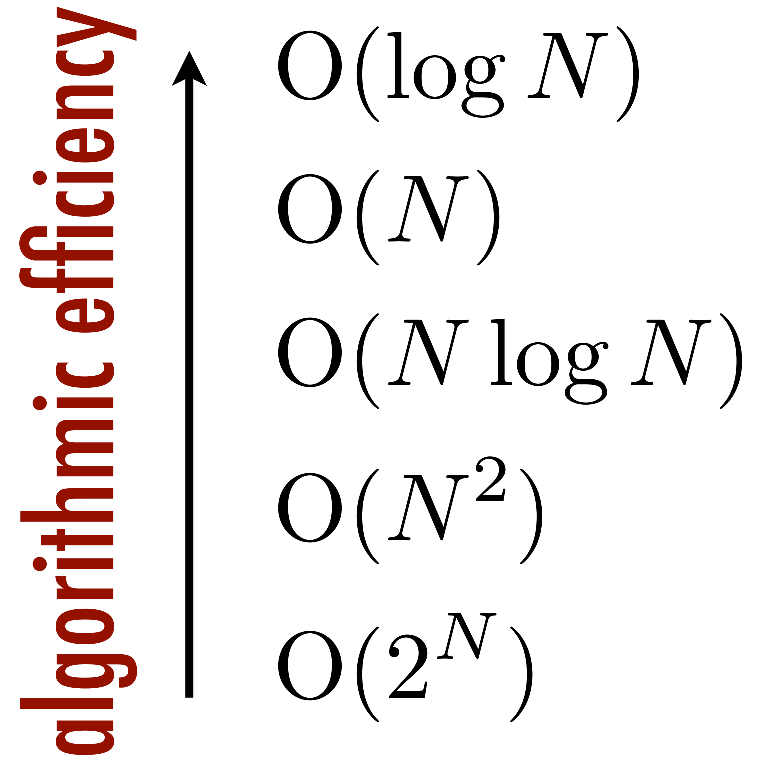
e.g., Ising model

Algorithm families

- Time evolution (of particles or fields):
  - molecular dynamics, N-body simulation, computational fluid dynamics
  - forward integration of the equations of motion
- Stochastic sampling
  - Monte Carlo methods
  - ensemble averages for systems in thermal equilibrium
- Normal mode analysis
  - linearized dynamics, spectral methods, matrix mechanics

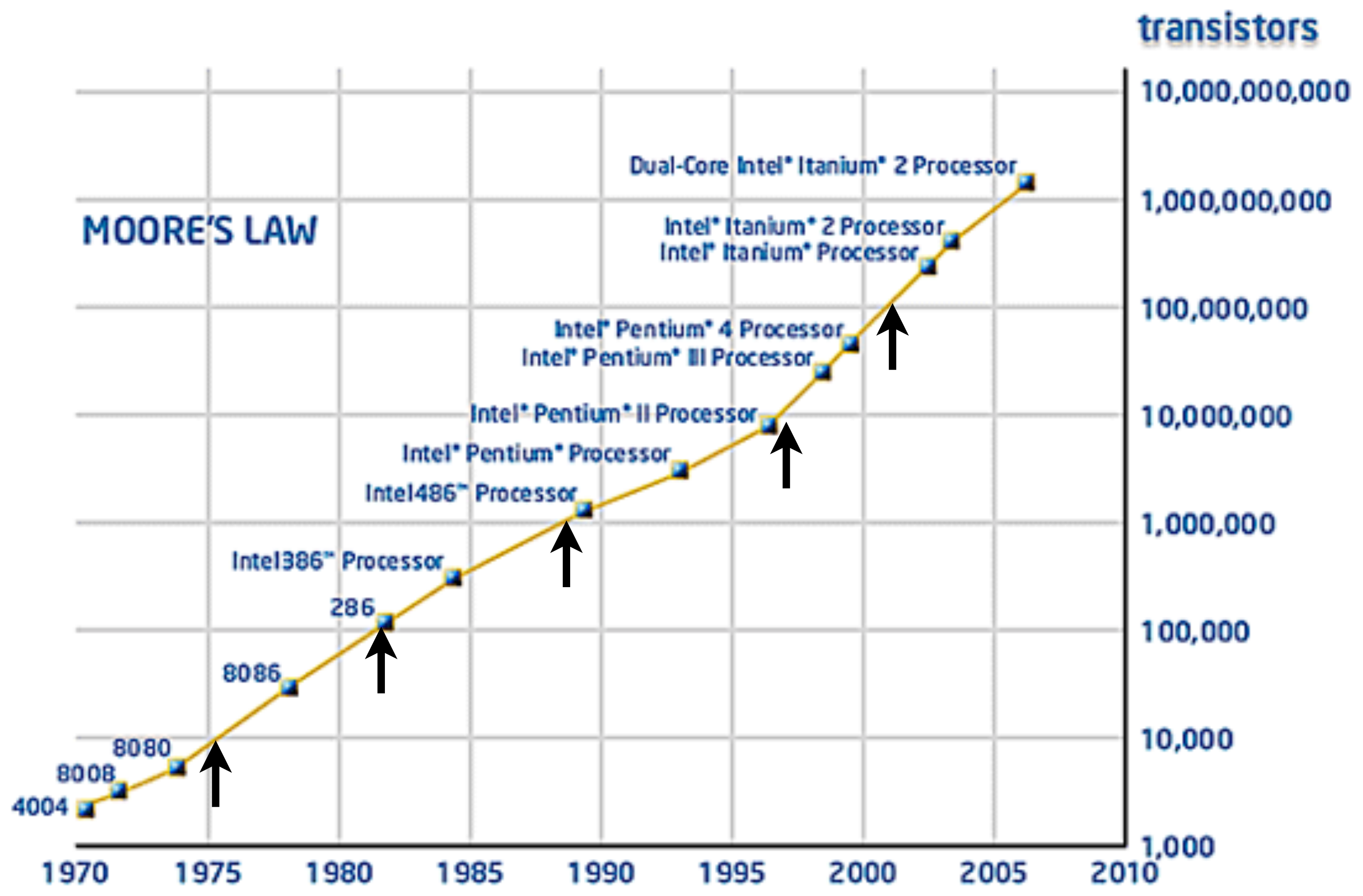
Algorithmic scaling

- How do the storage requirements and execution time scale as the size of the simulation grows?
- For N
  - particles ...
  - time steps ...
  - lattice spacings ...



Algorithmic scaling

good $O(N \log N)$	N=10	N=80	N=667	N=5720	N=50000
bad $O(N^2)$	N=10	N=32	N=100	N=316	N=1000
ugly $O(2^N)$	N=10	N=12.5	N=15	N=17.5	N=20



Algorithmic scaling

- e.g., simulation of a gravitating system; execution time limited by the calculation of all pairwise forces

```
const int N = 8; // number planets

// simple loop with N^2 operations
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
      if (i != j)
         acc[i] += force(i,j)/m[i];
```

Algorithmic scaling

- e.g., simulation of a gravitating system; execution time limited by the calculation of all pairwise forces

```
const int N = 8; // number planets
// simple loop with N^2 operations
for (int i = 0; i < N; ++i)
   for (int j = 0; j < N; ++j)
      if (i != j)
         acc[i] += force(i,j)/m[i];
```

```
// More efficient but op count
// N*(N-1)/2 is still O(N^2)
for (int i = 0; i < N; ++i)
   for (int j = i+1; j < N; ++j)
   {
      const double tmp = force(i,j);
      acc[i] += tmp/m[i];
      acc[j] -= tmp/m[j];
   }
```

Algorithmic scaling

- e.g., find the median from an array of energy values; a naive implementation that scales as $O(N^2)$

```
double median_naive(double* E, int N) {
     for (int i = 0; i <= N/2; ++i)
     {
          double smallest = E[i];
          for (int j = N-1; j > i; --j)
               if (E[j] < smallest) swap(E[i],E[j]);
     }
     return E[N/2];
}
double E[100] = { -3.98, 14.72, ..., 0.0892 };
double m = median_naive(E,100);
```

Algorithmic scaling

- median can be found in $O(N \log N)$ guaranteed

```
double E[100] = { -3.98, 14.72, ..., 0.0892 };
sort(E,E+100);
double m = E[50];
```

- or in $O(N)$ on average and $O(N^2)$ worst case with Hoare's selection algorithm (QuickSelect)

C++ language review

Edit-compile-run cycle

prompt

- From the UNIX terminal:

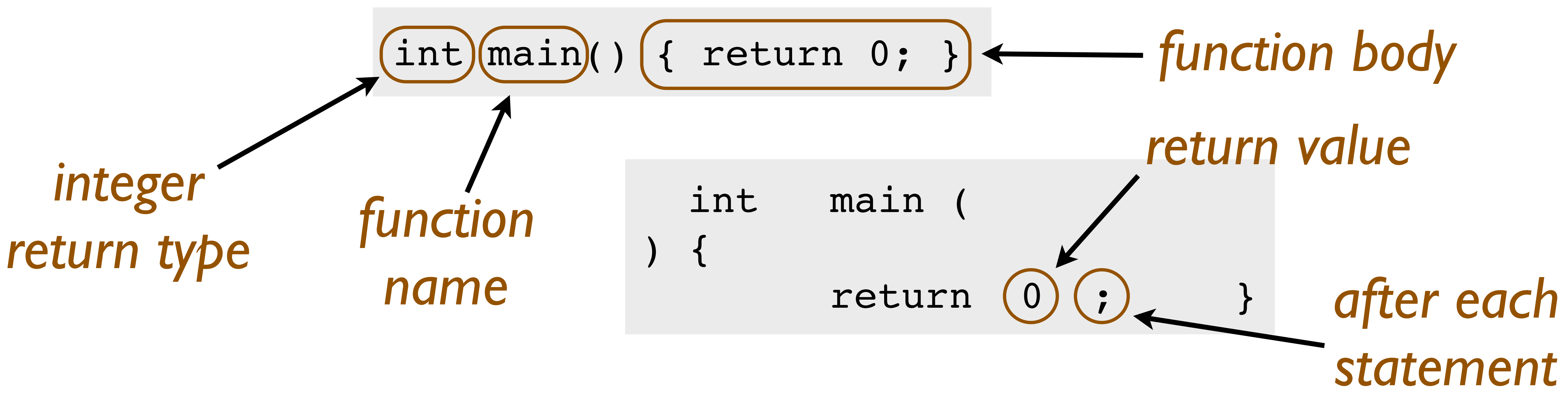
```
$ emacs myprog.cpp &
$ g++ -o myprog myprog.cpp \
      -ansi -pedantic \
      -lm -DNDEBUG
$ ls -F
myprog.cpp   myprog*
$ ./myprog
             0      1.41421
     0.0314159      1.41404
     0.0628319      1.41352
     0.0942478      1.41264
             |            |
       3.07876    0.0444215
       3.11018    0.0222135
```

user-created program file

```cpp
#include <cmath>
using std::sqrt;
using std::cos;

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

#include <iomanip>
using std::setw;

int main()
{
   for (int n = 0; n < 100; ++n)
   {
      const double x = M_PI*n/100.0;
      cout << setw(20) << x
           << setw(20) << sqrt(1+cos(x))
           << endl;
   }
   return 0;
}
```

Code formatting

- C++ code is freeform:
  - code structure is indicated by semicolons and braces
  - white space has no meaning
- These are equivalent restatements of the **null program**:

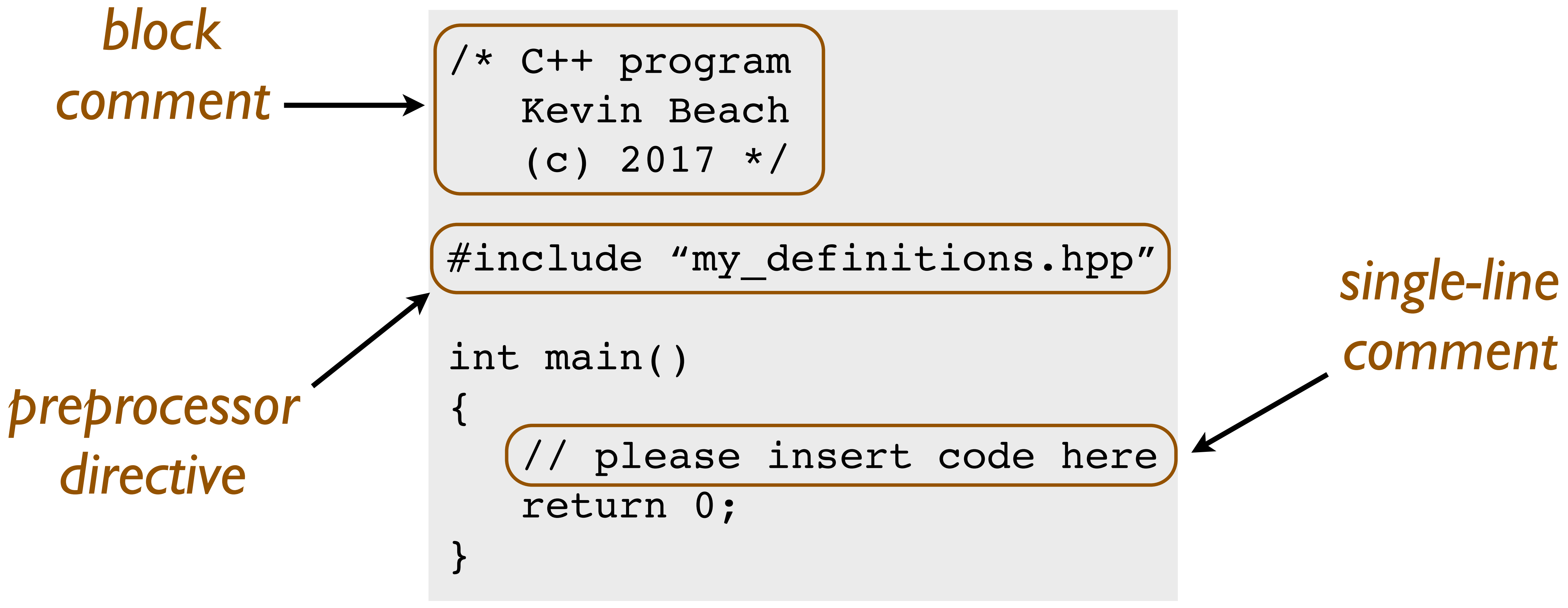


```
int main() { return 0; }
```

```
   int    main (
) {
         return  0  ;      }
```

Code formatting

- Single-line and block comments are supported
- Hash-prefixed commands are preprocessor directives



```
/* C++ program
   Kevin Beach
   (c) 2017 */

#include “my_definitions.hpp”

int main()
{
   // please insert code here
   return 0;
}
```

Properties

- The type, mutability, scope, and duration and of every object must be specified before it is used:
    - type and mutability are set by (prefix) keywords
    - scope and duration are controlled by where in the code an object is declared
- Every function acts on and returns objects of definite type

Types and their modifiers

- The integer and double-precision floating point types accept type modifiers
- Objects of any type may be flagged as immutable

type modifiers

`signed, unsigned`

`short, long`

`long`

type name	*literal*
`void`	
`bool`	`true, false`
`char`	`'a'`
`int`	`-81`
`float`	`3.14F`
`double`	`3.14`

constant modifier

`const`

Declarations and prototypes

- Type declarations and function prototypes

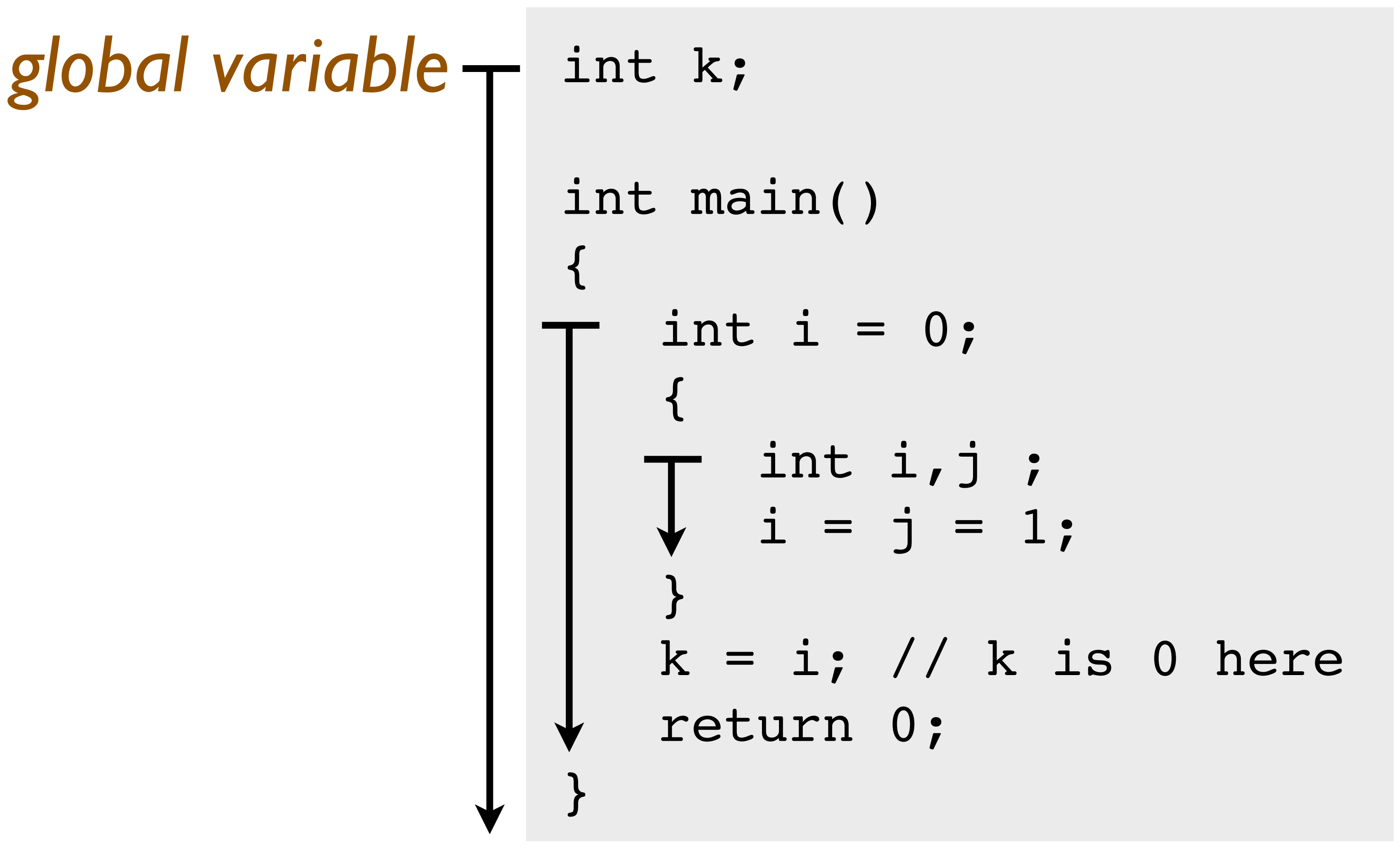
```
int sum(int x, int y) { return x+y; }
int prod(int, int);

const int A = 5;

int main()
{
    int a;
    int b = 6;
    a = sum(b,A); // give a the value 11
    return 0;
}
```

function prototype

integer variable declaration

Declarations and prototypes

- Objects survive until their current code block terminates
- A variable name may be temporarily obscured



```
int k;

int main()
{
    int i = 0;
    {
        int i,j ;
        i = j = 1;
    }
    k = i; // k is 0 here
    return 0;
}
```

Algebraic operators

modify in place

pre- and postfix "side-effects"

functions from the math library

```
int a = (5+7*3)/2; // a is 13
a = a - 2;         //      11
a += 4;            //      15
a /= 3;            //      5
int b = a%4;       // b is 1
int c = ++a/2;     // a is 6, c is 3
int d = a--/3;     // a is 5, d is 2

#include <cmath>
using std::pow; using std::sqrt;

double x0 = 2.0*2.0;
double x1 = pow(2.0,2.5);
double x2 = x0*2.0;
double eps = sqrt(x0*x2)-x1;
```

Boolean (logical) operators

assignment operator

test for equality

```
#include <cassert>

int a = 5;
int b = 7;

assert(a != b);
assert(!( a == b));

bool test1 = a < 2*b+5 and a != b;
bool test2 = a*b >= 3 or a*b <= -3;
bool test3 = b > a > 3;

assert(test1);
assert(test2);
assert(!test3);
```

run-time checks

legal but misleading

Bitwise operators

enumerated type

```
enum directions { N = 1, E = 2, S = 4, W = 8 };
const uint8_t opt1 = 020;    //  2*8 == 16
const uint8_t opt2 = 0x20;   // 2*16 == 32

unsigned char flags = N | W;
assert( (flags & N) and (flags & W) );

flags |= S | E;
assert( flags == N | S | E | W );

flags &= ~S;
assert( flags == N | E | W );

flags ^= N | E | opt1;
assert( flags == W | opt1 );
flags ^= opt1 | opt2;
assert( !(flags & opt1) and (flags & opt2) );
```

octal and hex

test bits

set, clear, and toggle bits

Control structures

- C++ provides standard looping and branching constructs:

```
int x = 1, y = 1;
int n = 5;
while (n > 0) { x *= 2; --n; }
do { y *= 2; } while (y < 32);
assert(x == y and y == 32);

const int M = (x < 0 ? -x : x);

int div5 = 0;
for (int m = 0; m < M; ++m)
   if (m%5 == 0)
      cout << ++div5 << endl;
   else
      do_something();
assert(div5 == 7);
```

looping

branching

Function arguments

- Functions arguments are passed by value, which prevents side effects
- Changes can be made to propagate outside the function by passing a reference instead

reference operator

```
int thrice(int x) { return 3*x; }
void triple(int &x) { x *= 3; }

int x = 3;
const int y = thrice(x); // y is 9
                         // x is still 3
triple(x); // x is now 9
```

Command-line arguments

```
#include <cstdlib>
using std::atoi; using std::atof;
using std::exit;
#include <iostream>
using std::cerr; using std::cout;
using std::endl;

int main(int argc, char* argv[])
{
   if (argc != 3)
   { cerr << “Two arguments required” << endl; exit(1); }

   int i = atoi(argv[1]); // argv[0] is “myprog”
   double x = atof(argv[2]);

   cout << “Int: ” << i << “ FP: ” << x << endl;
   return 0;
}
```

```
$ emacs myprog.cpp &
$ g++ -o myprog myprog.cpp \
      -ansi -pedantic \
      -lm -DNDEBUG
$ ./myprog
Two arguments required
$ ./myprog 5 -3.88
Int: 5 FP: -3.88
```

Writing to the terminal

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

#include <iomanip>
using std::setw;

#include <cmath>

main()
{
   cout.setf(std::ios::scientific);
   cout.precision(8);
   cout << setw(16) << M_PI << endl;
   return 0;
}
```

Composite objects

- arrays are groups of identically typed objects stored contiguously in memory
- structures and classes bundle objects of arbitrary type

```
bool active[3];
active[0] = active[1] = active[2] = true;

struct particle { double m, x, vx; };
particle gas[100]; particle test;

gas[0].m = 1.0;
test.vx = -5.5;
```

member operator

zero-based array indexing

Composite objects

- Classes may also have methods associated with them
- Methods are functions belonging to a class that act on its internal data

constructor / initialization list

```
class particle
{
public:
    double m, x, vx;
    particle(double m_, double x_, double vx_) :
       m(m_), x(x_), vx(vx_) {}
    double energy(void) { return 0.5*vx*vx/m; }
};

particle p(2.0,0.0,3.0);
const double E = p.energy();
```

method

Passing composite objects

- Arrays are passed as pointers to the first element
- Structures and classes should be passed by reference
- Only a memory address rather than the data is copied

```
double dot_product(double u[], double v[], int N);
double dot_product(double* u, double* v, int N);

double momentum(const particle &p)
{ return p.m*p.vx; }

void evolve_no_accel(particle &p, double dt)
{ p.x += p.vx*dt; }
```

pointer dereferencing

reference to a constant object

Passing composite objects

- arrays don’t know their own size and are thus dangerous

```
#include <cassert>
double sum_squares(const double x[], int N)
{
   if (N < 1)
      assert(false);
   double sum = 0.0;
   for (int i = 0; i < N; ++i)
      sum += x[i]*x[i];
   return sum;
}
const double v[3] = { 1.0, 2.0, -3.0 };
double norm_v = sum_squares(v,5);
```

forces a run-time error

memory error: “segmentation fault”

Templates

- Templates provide a mechanism for adding compile-time pattern matching to classes and functions:

```
template <typename T>
T abs_value(T x)
{
   if (x < 0) return -x;
   else return x;
}

template <int y>
void increment_by(int &x) { x += y; }

int a = -5;
int b = abs_value(a); // b is 5
increment_by<3>(b);   //      8
```

STL container classes

- The C++ standard library provides a variety of dynamically-allocated data structures:

```
#include <vector>
using std::vector;

double sum_squares(const vector<double> &v)
{
   assert(v.size()>0);
   double sum = 0.0;
   for (int i = 0; i < v.size(); ++i)
      sum += x[i]*x[i];
   return sum;
}

vector<double> u; double u0 = 0.5;
while(u.size() < 10) u.push_back(u0*=2.0);
const double N = sum_squares(u);
```

STL container classes

- sequence containers
  - $O(1)$ element access
  - $O(N)$ insertion
- associative containers
  - $O(N \log N)$ lookup
  - $O(1)$ insertion

```
#include <vector>
using std::vector

#include <deque>
using std::deque
```

```
#include <set>
using std::set

#include <map>
using std::map
```

```
list, slist, multiset, multimap, stack, queue, ...
```

Pointers and iterators

- Pointers point to data at a particular location in memory
- Iterators are pointer-like abstractions that are provided by the C++ container classes

```
double a[5] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
vector<double> v(a,a+5);

for (int i = 0; i < v.size(); ++i)
   assert(a[i] == v[i]);

for (double* step = a; step < a+5; ++step)
   do_work_on(*step);

for (vector<double>::iterator step = v.begin();
     step != v.end(); ++step)
   do_work_on(*step);
```

Generic programming

```
#include <vector>
using std::vector;

template <class Iter>
double sum_squares(Iter begin, Iter end)
{
   assert(begin != end);
   Iter p = begin;
   double sum = (*p)*(*p);
   while (++p < end) sum += (*p)*(*p);
   return sum;
}

const double A[] = { 2.7, -5.5, 100.1 };
const int B[] = { 1, 2, 3, 4, 5 };
vector<double> v(A,A+3);

sum_squares(v.begin(),v.end());
sum_squares(B,B+5);
```

File output and input

```
#include <iostream>
using std::endl;
#include <fstream>
using std::ofstream; using std::ifstream;
#include <cassert>

main()
{
   ofstream fout(“myfile.txt”);
   fout << “1 2 3 4 5” << endl;
   fout << “6 7 8 9 10” << endl;
   fout.close();
   ifstream fin(“myfile.txt”);
   vector<int> v;
   while(fin) { int i; fin >> i;
                v.push_back(i); }
   fin.close();
   assert(v.size() == 10);
   return 0;
}
```

Our overall approach

- Programming style will be more procedural than OO:
  - Treat C++ as a syntactically cleaner version of C
  - Make use of templates, generic programming, and STL data structures
- Things we’ll largely ignore:
  - class inheritance and polymorphism
  - virtual functions
  - exception handling