PHYS 234: Introductory Computational Physics

Doug Gingrich

Introduction

- The need for computers in science.
- What is computational physics?
  - Provides a means to solve complex numerical problems.
  - Enables attacking problems which otherwise might not be solvable.
  - One important aspect is modeling large complex systems.
  - Some modeling/solution techniques are impossible without computers.
  - Nonlinear problems are difficult without computers.

Basics

- Hardware:
  - CPU, memory, disk, I/O.
- Software:
  - Operating systems (OS):
    - Software that communicates with the CPU.
    - Lets you interact with the system.
  - Applications and languages.

Desktop environment (DE)

- Bundled programs running on top of an operating system, which share a common graphical user interface (GUI).
  - Provides access to some of the features of the underlying operating system.
  - Examples: web browser, terminal application, file manager, log in/out, etc.
- Command-line interface (CLI)
  - Provides full control over operating system
- Usually get to the CLI by clicking on the terminal icon in the DE.

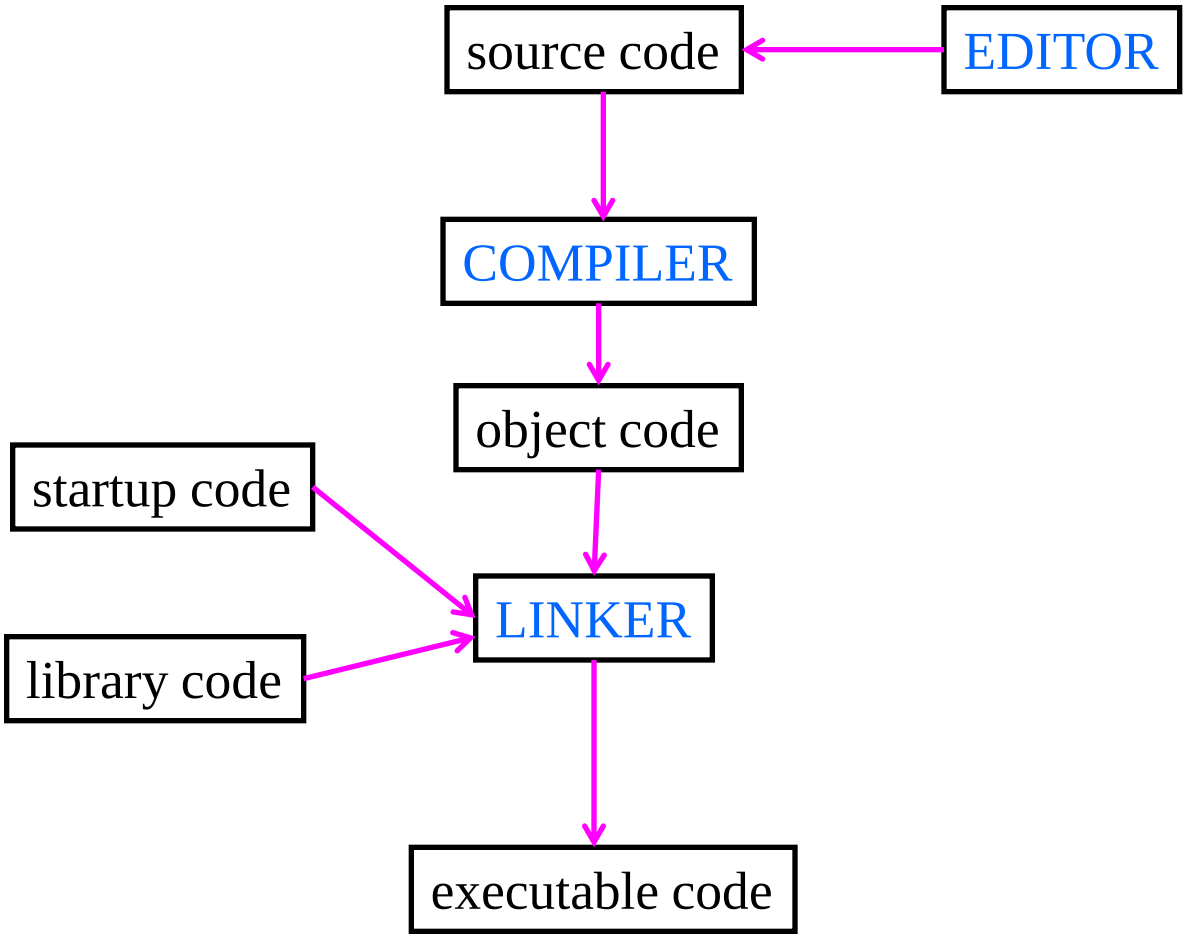
PHYS 234: Lecture 1

Doug Gingrich

Linux, shell, and C++

- **Linux**
  - UNIX is common operating system in science and companies with large computing needs.
  - Linux is a free version of UNIX that runs on many different types of computers.
- **Shell**
  - A UNIX shell is a command-line interpreter or shell that provides a traditional user interface for the UNIX operating system.
    - By default, you will use the bash shell.
- **C++**
  - FORTRAN use to be the most widely used scientific programing language.
  - UNIX is written in C.
  - C++ extends C's capabilities.

Programing steps



IDE

- Integrated development environment.
  - Manages all steps of program development, including editing, from one master program.

Text editors

- emacs
  - Extensibility.
- vim
  - Vi Improved.
- nano
  - Emulates Pico (Pine composer).
- gedit
  - Editor of GNOME desktop environment.
- nedit
  - Nirvan editor for X windows system.

Compilers and makefiles

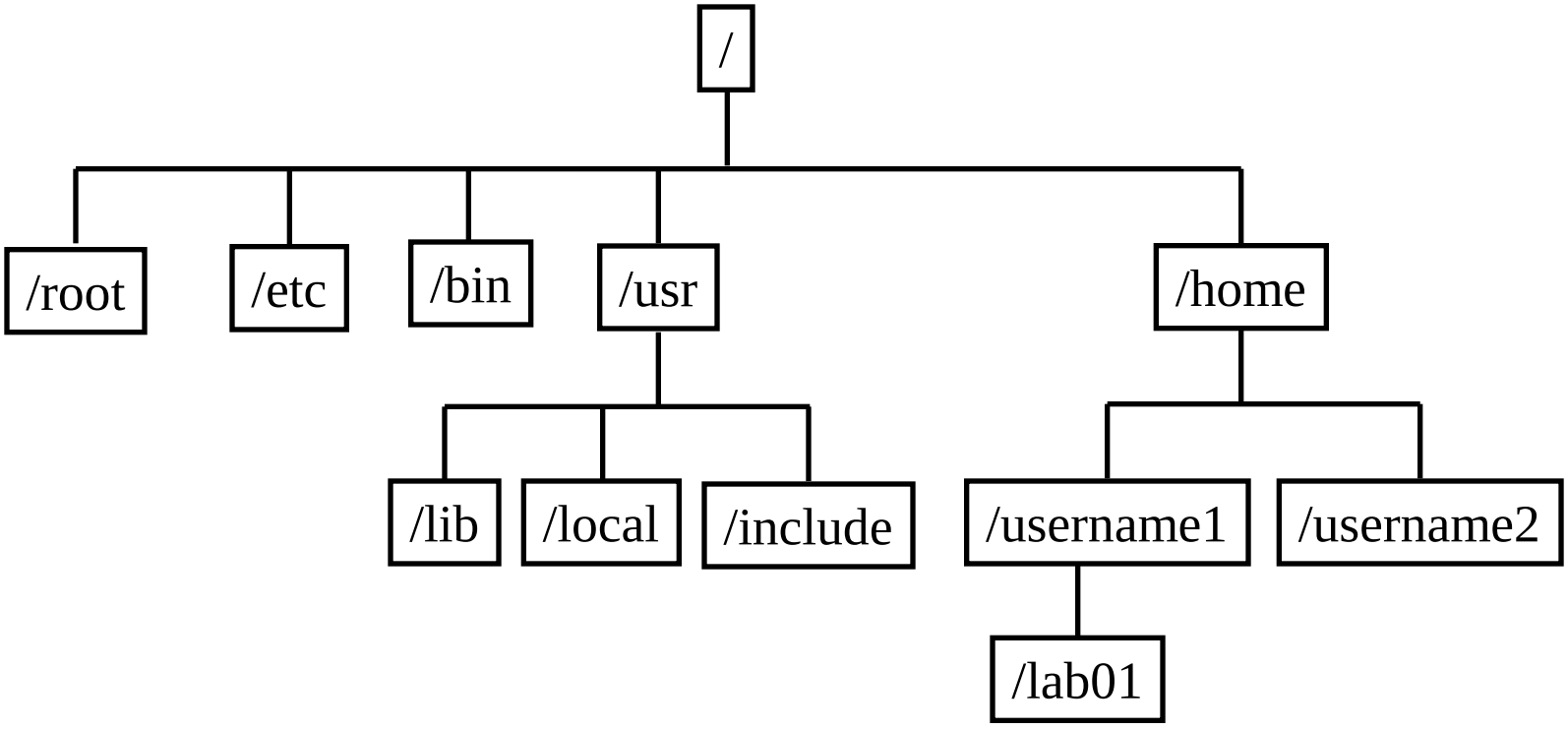
- **g++**
  - A compiler is a computer program (or set of programs) that transforms source code written in a programing language (the source language) into another computer language (the target language often having a binary form known as object code).
  - The most common reason for converting source code is to create an executable program.
- **Make**
  - In software development, Make is a utility that automatically builds executable programs and libraries from source code by reading files called makefiles which specify how to derive the target program.

Introduction to Linux

- **Logging in**
  - Linux is a multi-user system.
  - Linux is case sensitive.
  - When logged in you will see the prompt $.
- **Getting help**
  - Do not type "help", type "man" for manual pages.
  - man sucks for a beginner, use the web.

The file system

This is just an example, it may not be what you see.



/u/g/i/gingrich

File naming

`filename.type`

- C++ source code: `filename.cpp` (`.C`, `.cxx`, `.cc`, `.c++`)
- C++ header files: `filename.h` (`.hpp`)
- Object code: `filename.o`
- Executable code: `a.out` (usually explicitly named to `filename`)
- Makfile: `makefile`
- gnuplot instructions: `filename.gp`

The file system

- Moving around the system.
  - By default you land in `/home/username` when logging in.
  - `pwd` : print working directory.
  - `cd` *`directoryname`* : move to directory *`directoryname`*.
- Listing directories.
  - `ls` : list directory content (`ls -lFh` is a good choice of options).
- Creating files.
  - Use an editor.

PHYS 234: Lecture 2

Doug Gingrich

Why C++?

- C++ "real-world" language.
- May use C++ again outside this course.
- Consider this course also a C++ course.
- Transition to specialized languages will be easy.
- C++ computer science type features:
  - Procedural language (represented by C).
  - Object-oriented language (class enhancements).
  - Generic program (supported by templates).

Gnuplot and OpenGL

- We need some graphics package.
- **Gnuplot**
  - Portable command-line driven graphing utility.
  - Source code is free.
  - Created to allow scientists and students to visualize mathematical functions and data interactively.
- **OpenGL**
  - Open Graphics Library is a cross-language multi-platform application programing interface (API) for rendering 2D and 3D vector graphics.

Homework

- Read chapters 1-3 in K&G
- Read chapters 1-2 in Prata
- Browse the web to get an introduction

Error, accuracy, and stability

- Computers store numbers not with infinite precision but rather in some approximation that can be packed into a fixed number of bits.
- A number in integer representation is exact.
- Arithmetic between numbers in integer representation is also exact, with the provisos:
  - The answer is not outside the range of integers that can be represented.
  - That division is interpreted as producing an integer result, throwing away any integer remainder.

Round-off error

- Arithmetic among numbers in floating-point representation is not exact, even if the operands happened to be exactly represented.
- Round-off error (rounding error) is the difference between the calculated approximation of a number and its exact mathematical value.
  - This is a form or quantization error.
- Round-off error is a characteristic of computer hardware.
- When a sequence of calculations subject to rounding error is made, errors may accumulate, sometimes dominating the calculation.
  - In ill-conditioned problems, significant error may accumulate.

Truncation error

- Many numerical algorithms compute discrete approximations to some desired continuous quantity.
- The discrepancy between the true answer and the answer obtained in a practical calculation is called the truncation error.
- A characteristic of the program or algorithm used, independent of the hardware.
- Clever minimization of truncation error is the content of the field of numerical analysis.

Stability

- A method is unstable if any round-off error that becomes mixed into the calculation at an early stage is successively magnified until it comes to swamp the true answer.

Machine representation

- We are familiar with experimental uncertainty.
  - There exist also numerical uncertainty.
- Every computer has a limit on how small or large a number can be.
- Computers represent numbers in binary.
  - 8-bit example: `00001111 = 15`
- For a negative number, use sign bit.
  - `10000000 = -127`
- Also, `00000000 = 0`

Integer range

- Negative numbers represented by 2s compliment.
  - `00001111 = +15`
  - `11110000` 1s complement
  - `11110001 = -15` 2s complement (1s complement +1)
- For n-bit integer word range:
  - $[-2^{n-1}, 2^{n-1}-1]$

Floating-point representation

- In a floating-point representation, a number is represented internally by a sign bit S (interpreted as plus or minus), an exact integer exponent E, and an exactly represented binary mantissa M.

$$S \times M \times b^{E-e}$$

where b is the base of the representation (b=2 almost always), and e is the bias of the exponent, a fixed integer constant for any given machine and representation.

- Several floating-point bit patterns can in principle represent the same number.
  - Bit patterns that are as left-shifted as they can be are termed normalized.

Floating-point representation

- Most processors us the IEEE standard:
  - 32-bit float: E=8-bits (e=127), M=23-bits
  - 64-bit double: E=11-bits (e=1023), M=52 bits.
- IEEE standard also includes:
  - Positive and negative infinity
  - Positive and negative zero (treated computationally equivalent)
  - `NaN` (not a number)

Real range and precision

- Consider 16-bit word:
  - 0 1000 10000000000 = 0.5
  - (sign bit) (4-bit exponent) (11-bit mantissa)
  - Float = $(-1)^s$ x mantissa x $2^{exp-bias}$
  - Exponent contains a bias to allow negative values: 7
    - Range of exponent [0-7=-7,15-7=8]
- Precisions of machine:
  - $1/2^{15} = 3.1 \times 10^{-5}$

Programs

```
int main (void)
{
  // This is a comment.
  statements
  return 0;
}
```

Hello World! program

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

int main()
{
   cout << "Hello World!" << endl;
   return 0;
}
```

Good programing

- Think before you type.

PHYS 234: Lecture 3

Doug Gingrich

iostream and namespace

- To use input and output you need

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

- `#include` is a preprocessor directive that adds the contents of file `iostream`
- Allows the use of `cin` and `cout` facilities.
- `using namespace std;` makes all the `std` namespace available. To make just particular names available:

```
using std::cout;
using std::endl;
using std::cin;
```

Output with cout

```
int protons = 8;
cout << protons << endl;
```

- `cout` knows how to display a variety of data types.
- `endl` causes screen cursor to move to the beginning of the next line.

- Formatting is possible. For example,

```
#include <iomanip>
std::setw;


cout << setw(6) << protons << endl;
```

Concatenating with cout

```
cout << "This element has " << proton << " protons" << endl;

cout << "This element has "
     << proton
     << " protons"
     << endl;

cout << "This element has ";
cout << proton;
cout << " protons";
cout << endl;
```

assert

```
#include <cassert>
assert (a == 0);
```

- If the argument is `false` a message is written to the standard error device and `abort` is called, terminating the program execution.

C++ data types

- Built-in C++ types.
  - Fundamental types:
    - Integer types
    - Floating-point type
  - Compound types:
    - Arrays, strings, points, structures, etc.
- Create your own data types (OOP paradigm).

Simple variables

- Variables store information:
  - Where the information is stored.
  - What value is kept there.
  - What kind of information is stored.

```
int protons;
protons = 5;
```

- Variable names:
  - Use capitalization or underscores to separate words in a variable name.
  - Not too long, and not too short.
  - Be consistent.

Bits and bytes

- **Bit** is fundamental unit of memory: binary state.
- **Byte** is typically 8 bits.
- **Word** is a number of bytes (or bits).
  - Examples: 32-bit word, 64-bit word, etc.

Integer types

- Numbers with no fraction.
- Infinite number of integers.
- On computer, can only represent a subset of integers.
- Several choices: `bool`, `char`, `short`, `int`, `long`, `long long`
- Each choice has a different width (number of bits).
  - Exact width not fixed by C++ definition:
    - `short` at least 16 bit
    - `int` at least as big as `short`
    - `long` at least 32 bit, and at least as big as `int`
    - `long long` at least 64 bit, and at least as big at `long`

`signed` and `unsigned` integers

- Several choices: `char`, `short`, `int`, `long`, `long long`
- They come in `signed` and `unsigned` versions.
- `Unsigned` types can not hold negative numbers.
- If data type is 8 bits, say:
  - `unsigned`: all 8 bits available to hold positive integer.
  - `signed`: only 7 bits available to hold positive or negative integer.
    - the other bit holds the sign information.

Integer declarations and literals

```
short var1;
int var2;
long var3;
long long var4;
unsigned int var5;
```

- `int` generally sized to natural integer size of computer.
- An integer literal, or constant, is one you write out explicitly.

Integer initialization

- Combines assignment and declaration.

```
int protons = 5;
int electrons;
electrons = 5;
int photons = electrons;
int neutrons = protons + 2;
```

- If a variable is not initialized, its value is undetermined.

PHYS 234: Lecture 4

Doug Gingrich

`char` type

- Can store all the letters, digits, punctuation, and the like.
- Internally a number code is used for letters.
  - `char` is really an integer type (often a byte).
- For `char` literals, use single quotes for `char`, example 'G'
  - double quotes for strings.
- Some `char` can not be entered from keyboard:
  - Escape sequences: '\n' for newline, '\t' for tab, etc.
- `char` is unsigned by default.

char representation

- Internally a number code is used for letters.
  - Example: code 65 is A, code 77 is M, etc.
  - You input letters and the code outputs letters, but in memory number codes are stored.

```
char ch = 'M';
int i = ch;
cout << ch << " is " << i << endl;
```

M is 77

```
ch = ch + 1;
i = ch;
cout << ch << " is " << i << endl;
```

N is 78

`bool` type

- Two predefined literals: `true` and `false`
  - `true` represented as 1
  - `false` represented as 0
- Example,

```
bool is_proton = true;
```

const qualifier

- Read-only variable

```
const type name = value;
```

- Is necessary to initialize

```
const int protons = 2;
```

- `protons` can not be changed in subsequent code.

Floating-point numbers

- Used to represent numbers with fractional parts.
- The computer stores these numbers in two parts:
  - value and scale.
- Can write these in decimal-point notation or exponential notation: example, `2.34` or `8.3e-3`
- Floating-point types:
  - `float`: at least 32-bits
  - `double` : at least 48-bits and no smaller than `float`
  - `long double` : at least as big as `double`

Exponential represent

```
2.52e+8     // can use E or e, + is optional.
8.33E-4     // exponent can be negative.
7E5         // same as 7.0E+05.
```

- Optional + or − sign.
- Decimal point is optional.
- You can use e or E.
- No spaces between E+ or E−.
- Sign in exponential can be + or −, or omitted.

Type conversion

- C++ converts values when you assign a value of one arithmetic type to a variable of another arithmetic type.
- C++ converts values when you combine mixed types in expressions.
- C++ converts values when you pass arguments to functions.
- Type casts:
  - Can force type conversion explicitly.
  - Two syntaxs: `(typeName)value` and `typeName(value)`
  - New syntax: `static_cast<typeName> (value)`

Conversion on initialization and assignment

- Assigning a type to a greater range usually poses no problems:
  - Example `short` to `long`.
- Assigning a type to a lesser range can pose a problem:
  - `double` to `float`.
  - Floating-point type to integer type.
  - Bigger integer type to smaller integer type.
- A zero assigned to a `bool` variable converts to `false`.
- A non-zero value assigned to a `bool` converts to `true`.

Conversions in expressions

- What happens when you combing two different arithmetic types in on expression?
- C++ makes two types of automatic conversions:
  - Some types are automatically converted whenever they occur.
  - They are called integral promotions.

```
short protons = 20;
short neutrons = 35;
short nucleons = protons + neutrons;
```

  - Also `unsigned short` converted to `int`
- Some types are converted when they are combined with other types in an expression. For example, adding an `int` to a `float`.
  - The smaller is converted to the larger.

Conversion in passing arguments

- Normally, C++ function prototyping controls type conversions for the passing of arguments.
- It is possible, although usually unwise, to waive prototype control for argument passing.
- In this case:
  - Promotes `char` and `short` (`signed` and `unsigned`) to `int` type.
  - `float` promoted to `double`.

C++ arithmetic operators

```
int nucleons = protons + neutrons;
```

- `protons` and `neutrons` are the operands.
- `+` is the operator.
- `protons + neutrons` is the expression.
- Five arithmetic operators: +, -, *, /, %
- Division / with integers gives integer part of quotient, with fractional part being discarded.
- Modulus % is remainder of dividing first operand by second operand.
  - Only works on integer data types.
- Order of operation (operator precedence and associativity).
  - Can use parentheses to enforce your own priorities.
  - *, /, % have equal precedence.
  - +, - have equal but lower precedence then *, /, %

PHYS 234: Lecture 5

Doug Gingrich

Increment/decement operators

- pre-increment: `++a`
- pre-decrement: `--a`
- post-increment: `a++`
- post-decrement: `a--`
- For pre-increment/decrement, the increment/decrement is done before the rest of the operation.
- For post-increment/decrement, the increment/decrement is done after the rest of the operation.

`cout << a++ << endl;` and

`cout << ++a << endl;` give different output.

Combination assignment operators

`i = i + j;` can be written as

`i += j;`

Also

`i = i - j;` can be written as `i -= j;`

`i = i * j;` can be written as `i *= j;`

`i = i / j;` can be written as `i /= j;`

`i = i % j;` can be written as `i %= j;`

Evaluation of polynomials

- For a polynomial of degree `N`, with coefficients `c[i]` with `i = 0,…,N`.
  - Should not evaluate using:

```
p = c[0] + c[1]*x + c[2]*x*x + c[3]*x*x*x + c[4]*x*x*x*x;
```

  - Should not evaluate using `pow` function:

```
p = c[0] + c[1]*x + c[2]*pow(x,2.0) + c[3]*pow(x,3.0)+ c[4]*pow(x,4.0);
```

  - This is the preferred approach (that does not use a loop):

```
p = c[0] + x*(c[1] + x*(c[2] + x*(c[3] + x*c[4])));
```

Homework

- Read chapters 3 in Prata
- Browse the web to learn about simple data types

Compound types: Arrays

- Arrays hold several values of one type.
- Computer stores all the elements of an array consecutively in memory.
- Array declaration:
  - Type of value to be stored in each element.
  - Name of array.
  - Number of elements in array.

```
typeName arrayName[arraySize];
```

- `arraySize` must be known at compile time.
- Each element of the array can be treated as a simple variable.
- `sizeof(arrayName)` gives size of array in bytes.

PHYS 234: Lecture 6

Doug Gingrich

Compound types: Arrays

- Arrays hold several values of one type.
- Computer stores all the elements of an array consecutively in memory.
- Array declaration:
  - Type of value to be stored in each element.
  - Name of array.
  - Number of elements in array.

```
typeName arrayName[arraySize];
```

- `arraySize` must be known at compile time.
- Each element of the array can be treated as a simple variable.
- `sizeof(arrayName)` gives size of array in bytes.

Arrays

- Accessing an array element:
  - `array[0], …, array[arraySize-1];`
- The index starts at 0.
- Compiler does not check for a valid array index.
- Array initialization: `int protons[3] = {1, 10, 5};`
- To initialize all elements to 0: `int protons[3] = {0};`
- Assignment to a single element: `protons[2] = 5;`
- Arithmetic: `protons[2] = protons[0] + protons[1];`

  `total = protons[0] + protons[1] + protons[2];`
- `sizeof(protons[0])` is `4` bytes (for 32-bit `int`)
- `sizeof(protons)` is `12` bytes (for 32-bit `int`)
- `protons[-1]` or `protons[3]` are invalid.

Arrays initialization

```
int protons[2] = {5, 7}; // allowed
protons[2] = {5,7};      // not allowed
int protons[2] = {5};    // allowed, set remaining
                         // elements to 0
int electrons[2]; electrons = protons; // not allowed

int protons[] = {5, 7, 4};
int num_elements = sizeof (protons) / sizeof (int);
```

PHYS 234: Lecture 7

Doug Gingrich

Strings

- `string` is a series of characters stored in consecutive bytes of memory (array of `char`).
- The last character of every string is the null `char`: `\0`

```
char pion[] = {'p','i','o','n'}; \\ not string

char pion[] = {'p','i','o','n','\0'}; \\ string
```

- Better initialization: `\0` character is implied.

```
char hadron[] = "pion";

char hadron[5] = "pion";
```

- When determining the array size, must include `\0` in count.
- `'G'` is `char` literal, `"G"` is `string` literal consisting of two characters `G` and `\0`.

Strings and other things

- Whitespace: spaces, tabs, newlines.
- `String` concatenation:

```
cout << "this is a proton" << endl;

cout << "this is a" " proton" << endl;

cout << "this is a"
" proton" << endl;
```

- Can use a symbolic constant for the array size.

```
const int Size = 15;

char protons[Size];

char electrons[Size];
```

String input

- Reads input up to first whitespace character.

```
char protons[20];

cin >> protons;
```

- Can not input an entire line into `protons` array.
- If input is longer then `19` characters there could be problems.
- Often want to read string input a line at a time.

Line-oriented input with `getline()`

- Reads input up to newline character and discards newline character.

```
char protons[20];

cin.getline(protons,20);   // reads 19 char

cin.getline(protons,5);    // reads 4 char

cin.getline(protons,30);   // reads 19 char
```

- Replaces newline character with null character

Line-oriented input with `get()`

- Reads input up to newline character and leaves it in the input queue.

```
cin.get(array1,size1);
cin.get(array2,size2); // a problem


cin.get(array1,size1); // read first line
cin.get();             // read newline character
cin.get(array2,size2); // read second line
```

- Function `get()` is overloaded.
- `getline()` is easier to use, `get()` makes error checking easier.

PHYS 234: Lecture 8-9

Doug Gingrich

for loop

```
for (initialisation; test-expresssion; update-expression) {
  statements
}
```

```
for (int i=0; i<10; i++) {
  statements
}
```

- `i++` **increments** `i` **by** `1` `(i.e. i -> i + 1)`

while loop

```
while (test-condition)
{
   statements
}
```

- The *statements* are executed if `test-condition` is true.
- The cycle of testing and executions is continued.

do while loop

```
do {
   statements
} while (test-condition);
```

- Is an exit-condition loop.
- The `statements` are executed first time regardless of `test-condition`
- The *`statements`* are executed again if `test-condition` is true.
- The cycle of execution and testing is continued.

if statement

```
if (ch == 'A') {
  a_grade++;
}
else if (ch == 'B') {
  b_grade++;
}
else {
  soso++;
}
```

- Can have many `else if` cases.
- Can have no `else if` cases.
- Do no have to have an `else` cases.

Logical expressions

- C++ provides three logical operators to combine or modify existing expressions.
  - Logical OR : || (or `or`)
  - Logical AND : && (or `and`)
  - Logical NOT : ! (or `not`)
- Use parentheses to avoid worrying about operator precedence.

The ? : operator

- Called the conditional operator (requires 3 operands).
- Short form of `if else` statement.

`expression1 ? experssion2 : expresssion3`

- If `expression1 = true,` entire expression is `expression2`
- If `expression1 = false,` entire expression is `expression3`

```
int c;
if (a > b)
  c = a;
else
  c = b;
```

→ `int c = a > b ? a : b;`

switch statement

```
switch (integer-expression)
{
   case label1 :
      statements
      break;
   case label2:
      statements
      break;
   default:
      statements
}
```

- Program jumps to `case` of `integer-expression = labeln`
- `default` catches all other cases.

`break` and `continue` statements

- Allows skipping over parts of code.
- `break` : use in `switch` and in any loop.
  - Causes program to jump to next statement outside of `switch` or loop
- `continue` : used in loops.
  - Causes program to skip body of loop and start a new loop cycle.

PHYS 234: Lecture 10

Doug Gingrich

string class

- Allows using `string` variable (object) rather than a `char` array.

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

// Using char array
char hadron[20] = "meson";
char star[20];
cout << hadron;
cin >> star;

// Using string
string shadron = "pion";
string sstar;
cout << shadron;
cin >> sstar;
```

string class operations

```
string str1;
string str2 = "protons";
str1 = str2;                // object assignment OK
string str3;
str3 = str1 + str2;         // assign str3 the join strings
str1 += str2;               // add str2 to end of str2

int len1 = str1.size(); // obtain length of str1

getline(cin,str1);          // no length specified
```

Structures

- Structures hold items of different data types.
- It is a user definable type.
- After you define the type, you can create a variable of that type.
- Structure declaration:

```
struct inflatable // inflatable is name of new data type
{
  char name[20];
  float volume;
  double price;
};

inflatable hat; // creates variable hat of type inflatable
```

- The items in the `struct` are the members.
- Access member like `hat.price, hat.volume, hat.name[3]`

Other structures properties

- Structure declaration can be placed:
  - above `main()` : accessible to all functions.
  - inside `main()` : accessible only to `main()`
- Initialization:

```
inflatable hat = {"sports style", 1.88, 29.99};
```

- You can pass structures as arguments in function.
- A function can return a structure.
- Can assign (=) one structure to another.
- Can create arrays of structures.

Enumerations

- `enum` facility is alternative to `const`
- Allows creation of symbolic constants.

```
enum colour {red, blue, green};
```

  - By default `red=0, blue=1, green=2`

`colour spectrum;` Spectrum is a variable of type colour.

- Only the assignment operator is defined for enumerations.

Array alternatives

- `vector` template class
  - Dynamic array: `vector<typeName> vt(n_elem);`
  - `n_elem` can be an a constant or a variable.

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

  - Can set size of vector object during runtime and can append new data to the end or insert new data in the middle.
- `array` template class
  - An improved version of built-in array type.
- These are instances of a more general container class.

Pointers and free storage

- Pointer variables store addresses of values, rather then the values themselves.
- If `proton` is a variable, `&proton` is its address.
  - `&` is address operator.
- If `electron` is a pointer, `*electron` is its value.
  - `*` is indirect value or dereferencing operator.
- For a data type, the named quantity is the value, and the address of that value is the derived quantity.
- For a pointer, the named quantity is the address, and the value at that address is the derived quantity.

C++ philosophy and dangers

- Pointers and free storage enable runtime decisions.
  - Runtime: while the program is running.
  - Compile time: when the program is compiled.
- Pointers are not integer types.
- Dangers of pointers:
  - Creating a pointer does not allocate memory to hold the data to which the address points.
  - Always initialize a point to a definite address before applying the dereferencing operator (*) to it.

Declaring and initializing pointers

- Declaring a pointer: `int *ptr;`
  - `*ptr` is an `int`, `ptr` is a pointer to an `int`
- Space around `*` does not matter: (2 choices):

```
int *ptr;
```

  - Emphasizes that the combination `*ptr` is type `int`

```
int* ptr;
```

  - Emphasizes that `int*` is type pointer-to-`int`
- A pointer is always a pointer to a specific type.
- Can initialize a pointer in its declaration statement:

```
int higgens = 5;
int * pt = &higgens;
```

Allocating memory with new

- Can use pointers to allocate unnamed memory during runtime.

```
typeName * pointer_name = new typeName;
```

  - At compile time:

```
int higgens;
int * pt = &higgens;
```

  - At runtime:

```
int * pn = new int;
```

Freeing memory with `delete`

- `delete` enables returning memory to the free pool when you are done with it.

```
int * ps = new int; // allocate memory
…                  // use the memory
delete ps;         // free memory
```

- Always balance the use of `new` with a `delete`.
- Only use `delete` to free memory that was created by `new`.

Dynamic arrays and pointer arithmetic

- Typically use `new` with arrays, strings and structures.

```
int * pt = new int [10];
```

- The `new` operator returns the address of the first element in the block.
- To free the memory use

```
delete [] pt;
```

- Using a dynamic array:

```
pt[3] = 4;
pt = pt + 1; // pt now points to second element
```

- Adding 1 to an integer variable increase its value by 1.
- Adding 1 to a pointer increases its value by the number of bytes of the type to which it points.

Homework

- Read chapters 4 in Prata
- Browse the web to learn about compound data types

Command-line processing

- Command-line arguments are arguments that appear on the command line when you type a command.
- Use the alternative function heading for `main()`

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

`argc` is the number of arguments on the command line.

`argv` is a pointer to a pointer to a `char`

- An array of pointers to pointers to the command-line arguments.

`argv[0]` a pointer to first character of a string holding the command name.

`argv[1]` a pointer to first character of a string holding the first command-line argument, and so on.

PHYS 234: Lecture 11

Doug Gingrich

Functions

- Modules from which C++ programs are built.
- To use a C++ function:
  - Provide a function definition.
    - Tells the calling function what is returned.
  - Provide a function prototype
    - Tells the calling function what to expect.
  - Call the function.

void functions

- Some functions return nothing or return `void`.
- Typically used to perform some sort of action.

```
void functionName(parameterList)
{
  statements
  return; // optional
}

void cheers(int n)
{
  for (int i=0; i<n; i++) {
    std::cout << "Cheers! ";
  }
  std::cout << std::end;
}
```

Functions with return values

- An argument is information sent to the function, and the return value is a value sent back from the function.
- Think of the return value as what is substituted for the function call in the statement after the function finishes its job.
- The returned value can be anything except for an array.

```
type functionname (argumentlist)
{
  statements
}

double calculation (int a, int b)
{
  double y;
  y = a * b + a /b;
  return y;
}
```

Multiple return statements

- Function terminates after executing a return statement.

```
int bigger(int a, int b)
{
  if (a > b)
    return a; // if a > b function terminates here
  else
    return b; // otherwise, function terminates here
}
```

Functions in a program

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

// function prototypes
void simon(int);
double taxes(double);

// function 1
int main ()
{
  …
  return 0;
}
```

```
// function 2
void simon(int n)
{
  …
}

//function 3
double taxes(double t)
{
  …
  return 2 * t;
}
```

Passing by value

```
double cube (double x);

int main()
{
  double side=5;
  double volume = cube(side);
}

double cube(double x)
{
  return x * x * x;
}
```

Passing by reference

```
double cube (double &x);

int main()
{
  double side=5;
  double volume = cube(side);
}

double cube(double &x)
{
  return x * x * x;
}
```

Function prototypes

- A C++ program should provide a prototype for each function used in the program (describes function interface).
- A function prototype does for functions what a variable declaration does for variables.

```
// Function protype
double sqrt(double); // or double sqrt(double y);

double x;
x = sqrt(6.25);
```

- The prototype does not include the code for the function.
- Function prototype must appear before where function used.
- Function prototype does not require you have a name for the variable.
  - Variable name is just a place holder, does not have to match names in function definition.

What prototypes do for you

- The compiler correctly handles the function return value.
- The compiler checks that you use the correct number of function arguments.
- The compiler checks that you use the correct type of arguments.
  - If you don't, it converts the arguments to the correct type, if possible.
- Prototyping takes place during compile time and is termed *static checking*.

Functions and arrays

```
int sum_arr(int arr[], int n) // arr array name, n = size
```

- `arr[]` is actually a pointer

Default arguments

- A default argument is a value that's used automatically if you omit the corresponding actual argument from a function call.
- Default value given in function prototype.

```
int harpo (int n, int m = 4, int j = 5);
beeps = harpo(2);          // same as harpo(2,4,5)
beeps = harpo(1,8);        // same as harpo(1,8,5)
beeps = harpo(8,7,6);
```

- Must add defaults from the left to the right.

Function overloading

- Can attach many functions to the same name.
  - Different types of arguments.

```
void print(const char * str, int width);
void print(double d, int width);
void print(long l, int width);
void print(int I, int width);
void print(const char *str);
```

Command-line processing

- Command-line arguments are arguments that appear on the command line when you type a command.
- Use the alternative function heading for `main()`

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

`argc` is the number of arguments on the command line.

`argv` is a pointer to a pointer to a `char`

- An array of pointers to pointers to the command-line arguments.

`argv[0]` a pointer to first character of a string holding the command name.

`argv[1]` a pointer to first character of a string holding the first command-line argument, and so on

Recursion

- A C++ function can call itself.
- Need something to terminate the chain of infinite calls.

```
void recurs(argumentlist)
{
  statement1
  if (test)
    recurs(arguments)
  statements2
}
```

- When recursion terminates the stack is unwound.
- Each recursive call creates its own set of variables.

Homework

- Read chapters 7 in Prata
- Browse the web to understand functions

PHYS 234: Computational Physics

Lecture 12: Wednesday, 11 February 2015

1 Histograms

A histogram is a graphical representation of the distribution of data. It is an estimate of the probability distribution of a continuous variable. To construct a histogram, the first step is to "bin" the range of values – that is, divide the entire range of values into a series of smaller intervals – and then count how many values fall into each interval. A rectangle is drawn with height proportional to the count and width equal to the bin size, so that the rectangles abut each other. A histogram may be normalized displaying the relative frequencies. It then shows the proportion of cases that fall into each of several categories, with the sum of heights equal to 1.

Histograms give a rough sense of the density of the data, and often for density estimation: estimating the probability density function of the underlying variable. The total area of a histogram used for probability density is always normalzied to 1.

Histograms are often confused with bar charts.

2 Monte Carlo simulation

Stochastic: can not predict from the observation of one event how the next event, with same initial starting conditions, will come out.

2.1 Uniform random number generation

There is no true computer generated random number. Random number will eventually repeat. This is called the period of generator. You should check the generator. Random numbers should also be uniform in interval $[0, 1)$.

We use the `mtrand` class in the labs. Declare

```
mtrand R;
```

then can use `R()` as random number in interval $[0, 1)$.

Linear transformations of uniform random numbers

$$\begin{array}{lcl} [0,1) & \rightarrow & R() \\ [0,C) & \rightarrow & C * R() \\ [a,b) & \rightarrow & (b-a) * R() + a \end{array} \tag{1}$$

2.2 Random number for nonuniform distributions

eg. Gaussian, interval is not $[0, 1)$.

2.2.1 Inversion method

Method in general Normal uniform distribution x available in interval $[0, 1)$. Probability to find number in interval dx around x is

$$\begin{aligned} p(x)dx &= dx \quad \text{for } 0 \leq x < 1\,, \\ &= 0 \quad \text{otherwise}\,. \end{aligned} \tag{2}$$

$p(x)dx$ is the probability and $p(x)$ is the probability density.
Normalization gives

$$\int_{-\infty}^{\infty} p(x)dx = 1\,. \tag{3}$$

Suppose that we have a uniform distribution in x. We want distribution in x that is not uniform, $y = y(x)$. The probability distribution of y, denoted by $y(x)dy$, is determined by the fundamental transformation law of probabilities, which is simply

$$|p(y)dy| = |p(x)dx| \tag{4}$$

or

$$p(y) = p(x)\left|\frac{dx}{dy}\right| \qquad\qquad \frac{dn}{dy} = \frac{dn}{dx}\left|\frac{dx}{dy}\right|\,. \tag{5}$$

Since x is uniformly distributed, $p(x) = dn/dx = 1$.

Let's see what is involved in using the above transformation method to generate some arbitrary desired distribution of y's , say one with $p(y) = f(y)$ for some positive function f whose integral is unity. We need to solve the differential equation

$$\frac{dx}{dy} = f(y)\,. \tag{6}$$

But the solution of this is just $x = F(y)$, where $F(y)$ is the indefinite integral of $f(y)$.

$$x = F(y) = \int^{x} f(y)dy\,. \tag{7}$$

The desired transformation that takes a uniform deviate into one distribution as $f(y)$ is therefore

$$f(x) = F^{-1}(x) \tag{8}$$

where F^{-1} is the inverse function to F.

Example We want to generate a random variable t with pdf

$$f(t) = \lambda e^{-\lambda t} \text{ for } t \geq 0\,. \tag{9}$$

Cumulative distribution

$$F(t) = \int_0^t \lambda e^{-\lambda\tau} d\tau = 1 - e^{-\lambda t} \text{ for } t \geq 0\,. \tag{10}$$

If r is uniform random variable

$$r = F(t) = 1 - e^{-\lambda t} \tag{11}$$

$$t = -\frac{\ln(1-r)}{\lambda} \tag{12}$$

If r is uniform in $[0, 1)$ than $1 - r$ is uniform in $(0, 1]$.

$$\Rightarrow t = -\frac{\ln r}{\lambda}\,. \tag{13}$$

2.2.2 Accept and rejection method

This method is easiest but not the fastest. Consider generating numbers following a certain distribution $f(x)$ with $x \in [a, b)$. Generating two random numbers would fill the plan in (x, y). Reject those points outside of $f(x)$.

Generate two random numbers:

1. one between a and b,
2. second between 0 and $f^{\max}$, assuming $f(x) > 0$,
3. reject all points (x, y) where $y > f(x)$.

2.3 Monte Carlo integration

Monte Carlo methods can be used to integrate complicated functions. Good for multi-dimensional integrals. Put the regions you want to integrate into a regions you know the integral of. Use accept and reject to determine area.

Example is calculating π. Area of circle is πr^2. For unit radius, area is π. Generate random numbers in one quarter of plan (positive (x, y)). Look for points less than $\sqrt{1 - x^2}$

How to estimate the accuracy and error of the Monte Carlo integration? The distribution about the true value often has a Gaussian shape, which is the case for a random process. Standard deviation can be written as

$$\sigma = \sqrt{\frac{\frac{1}{N}\sum f^2(x_i) - (\frac{1}{N}\sum f(x_i))^2}{N}}\,, \tag{14}$$

which shows how $\sigma \propto 1/\sqrt{N}$. In order to reduce the dispersion of our integral by 2 we need to throw 4 times as many points.

"Introductory Computational Physics", A. Klein & A. Godunov, Cambridge University Press

PHYS 234: Computational Physics

Lecture 13: Monday 23 and Wednesday 25 February 2015

1 Solution of nonlinear equations

Consider the case when $f(x)$ is a function of a single real variable x. We want to solve $f(x) = 0$. This is equivalent to looking for the real root in the interval $[a, b]$.

For a polynomial, the roots and one more condition determine the polynomial. Suppose the polynomial has roots at c_1, c_2, and c_3. To within numerical factor A the polynomial is

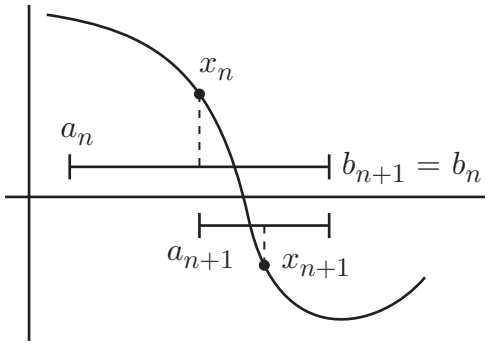
$$p(x) = A(x - c_1)(x - c_2)(x - c_3) \tag{1}$$

A condition like $p(0) = B$ will allow the determination of A.

I suggest you first attempt a graphical analysis of the function as much as possible. You need to bracket a root (know that a function changes sign in an identified interval) before you can look for it.

1.1 Bisection method

Bisection is the simplist but most robust method. The method never fails. $f(x)$ changes sign between $x = a$ and $x = b$. i.e. $f(a)f(b) < 0 \Rightarrow$ at least one real root in $[a, b]$.



1. Divide $[a, b]$ into 2 equal halves with a middle point at $x_1 = (a + b)/2$.

$$f(a)f(x_1) \begin{cases} < 0 & \text{there is root in } [a, x_1] \\ > 0 & \text{there is root in } [x_1, b] \\ = 0 & \text{then } x_1 \text{ is the root} \end{cases} \tag{2}$$

2. Repeat in interval that has root.

3. After n steps the interval is $\epsilon = (b-a)/2^n$.

4. Stop when $\epsilon = (b-a)/2^n = \epsilon_0/2^n$ is of desired tolerance.

The number of iterations to reach a desire tolerance is

$$n = \log_2 \frac{\epsilon_0}{\epsilon} \tag{3}$$

The convergence of a method can be view as

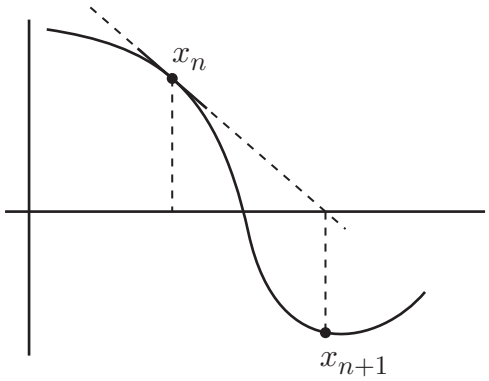
$$\epsilon_{n+1} = \text{constant} \times (\epsilon_n)^m \tag{4}$$

If $m = 1$, as in the bisection method, the convergence is linear. If $m > 1$ the convergence is superlinear.

While the analytic function may be zero at the root, it is probably never exactly zero, for any floating-point argument. It is best to use an absolute tolerance when considering convergence as a relative tolerance has problems near zero. An example would be to finish in 50 bisections, with $2^{-50} \approx 10^{-15}$.

1.2 Newton's (or Newton and Raphson) method

This method uses $f'(x)$ to accelerate the convergences. Graphically the method consists of extending the tangent line at a current point x_i until it crosses zero, then setting the next guess x_{i+1} to the abscissa of that zero crossing.



Taylor series

$$f(x) = f(x_0) + (x-x_0)f'(x_0) + (x-x_0)^2\frac{f''(x_0)}{2!} + (x-x_0)^3\frac{f'''(x_0)}{3!} + \ldots \tag{5}$$

If $f(x) = 0$, keep first 2 terms.

$$f(x) = 0 = f(x_0) + (x-x_0)f'(x_0) \tag{6}$$

$$x = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{7}$$

The next iteration

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{8}$$

Problems:

1. Very slow approach to solution when $f'(x) \to 0$ is around the root.
2. Difficult with local minima, leading to next iteration value x_{k+1} far away. In this case the higher-order terms in the series are important.
3. lack of convergence for asymmetric functions $f(a+x) = -f(a-x)$.

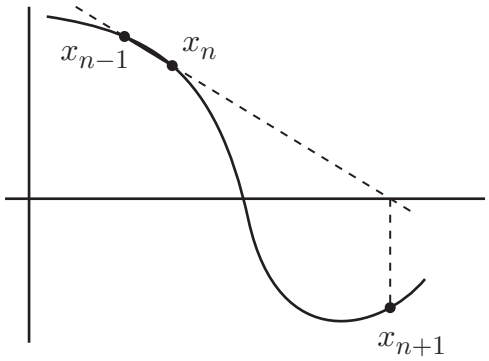
The advantage of the Newton-Raphson method is its rate of convergence.

$$\epsilon_{i+1} = -\epsilon_i^2 \frac{f''(x)}{2f'(x)}\,, \tag{9}$$

so the convergence is quadratic. Near a root the number of significant digits approximately doubles with each step. Method of choice for any function whose derivative can be evaluated efficiently, and whose derivative is continuous and nonzero in the neighbourhood of a root.

The Newton-Raphson method is not restricted to one dimensions. It readily generalises to multiple dimensions.

1.3 Method of secants



Variation of Newton's method when the evaluation of derivatives is difficult. The derivative f' of the continuous function $f(x)$ at the point x.

$$f'(x_0) = \frac{f(x_0) - f(x_1)}{x_0 - x_1} \tag{10}$$

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \Rightarrow x_{k+1} = x_k - \frac{(x_k - x_{k-1})f(x_k)}{f_k - f_{k-1}} \tag{11}$$

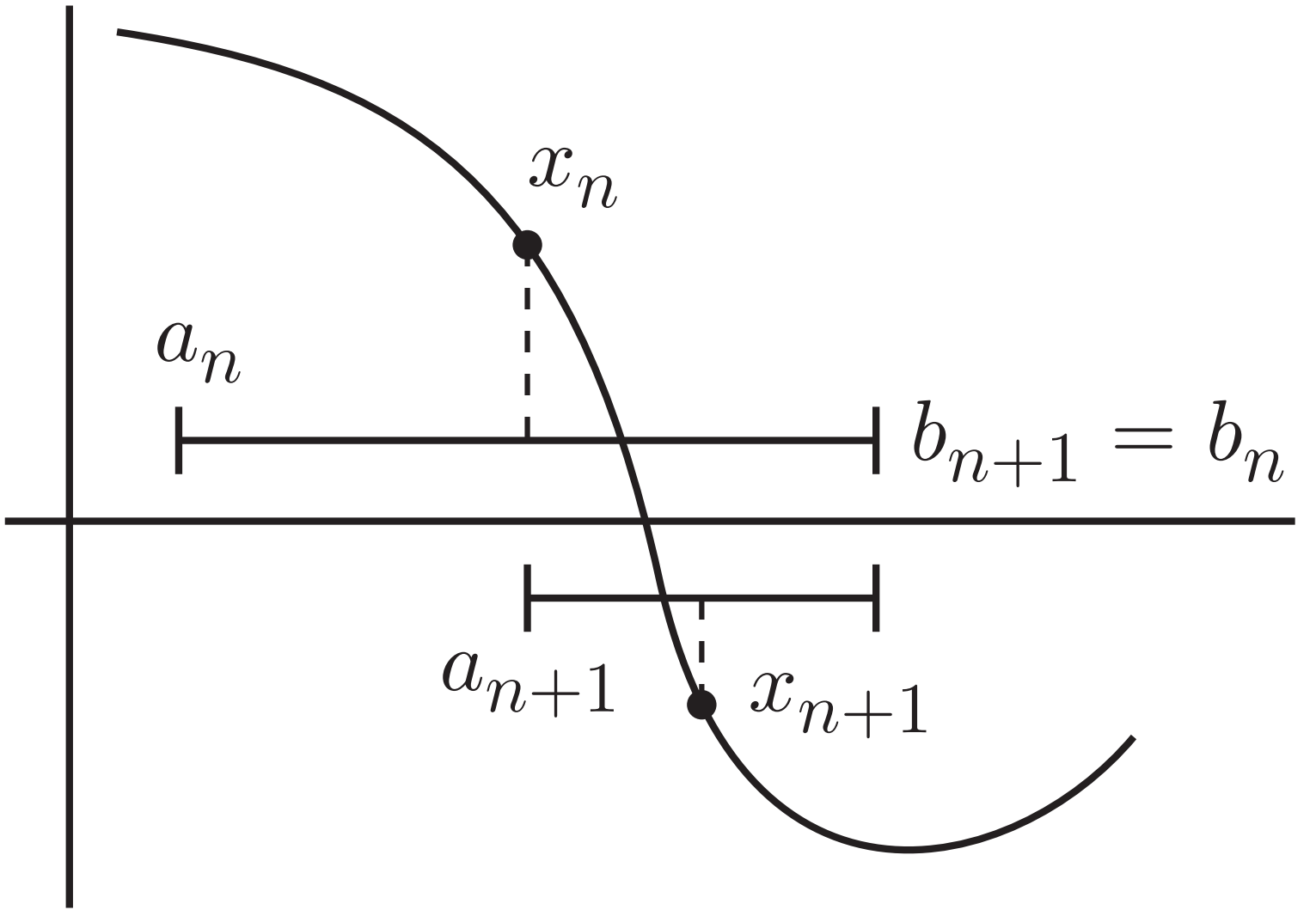
The method assumes the function to be approximately linear in the local region of interest, and the next improvement in the root is taken as the point where the approximating line crosses the axis. After each iteration, one of the previous boundary points is discarded in favour of the latest estimate of the root. The secant retains the most recent of the prior estimates (this requires an arbitrary choice on the first iteration).

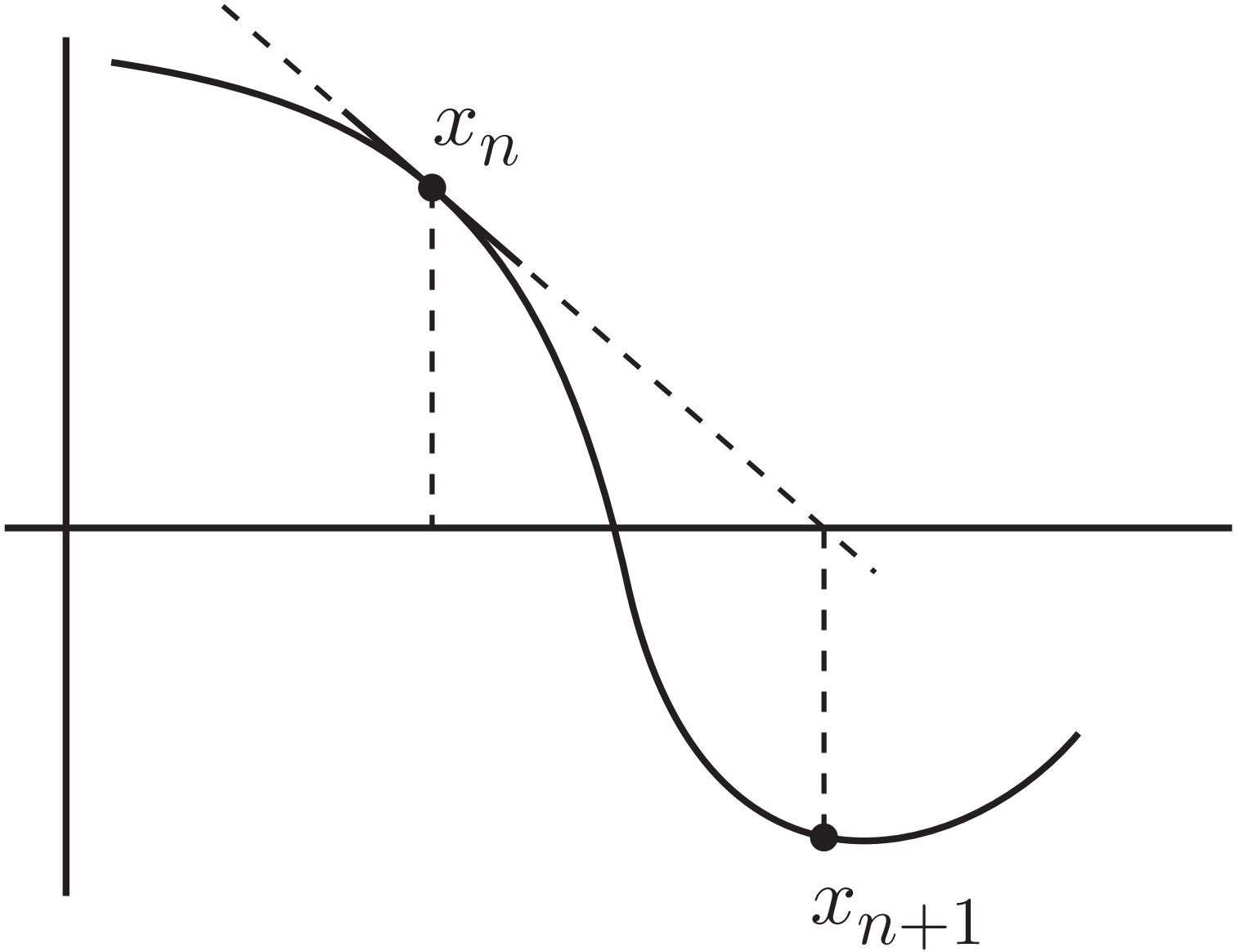
Need to select 2 initial points to start. The root does not necessarily remain bracketed. The algorithm can not be guaranteed to converge.

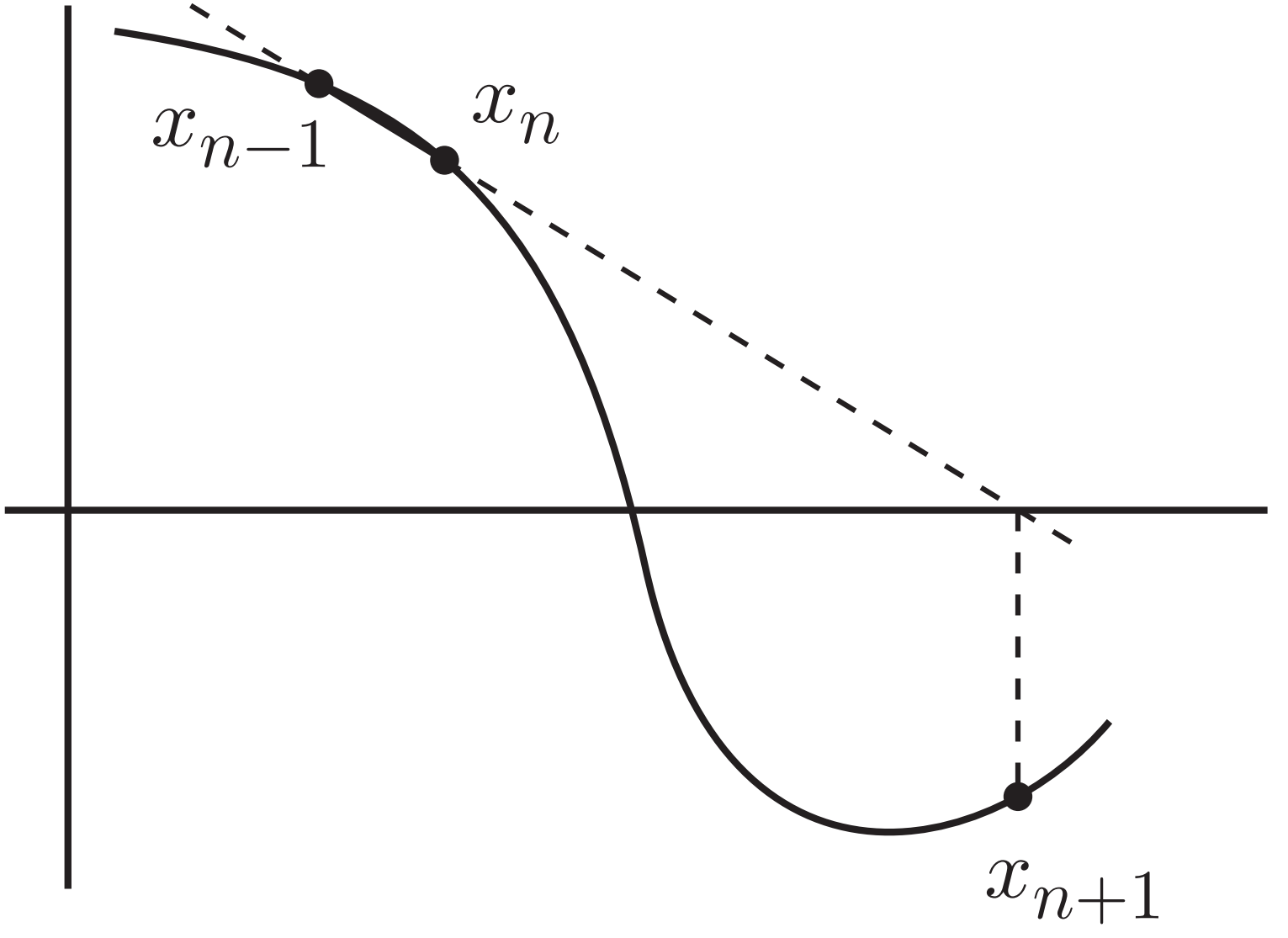
The convergence of secant method between Newton's and bisection method.

$$\lim_{k\to\infty} |\epsilon_{k+1}| \approx \text{ const } \times |\epsilon_k|^{1.618} \tag{12}$$

"Introductory Computational Physics", A. Klein & A. Godunov, Cambridge University Press
"Numerical Recipes: The Art of Scientific Computing", W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Cambridge University Press


x_n
a_n
$b_{n+1} = b_n$
a_{n+1}
x_{n+1}


x_n
x_{n+1}


x_{n-1}
x_n
x_{n+1}

PHYS 234: Computational Physics

Lecture 14: Wednesday 25 and Friday 27 February 2015

1 Interpolation

Measurements usually yield a discrete set of points (x_i, y_i). We want to know y for arbitrary x. We might be only interested in a small local region.

Interpolation: select function $g(x)$, such that $g(x) = f_i$ for each data point i. $g(x)$ good approximation for other x lying between original data points. Like drawing a smooth curve through the data points.

Infinite number of functions may interpolate data. If spacing too big, interpolation may be totally off. Using polynomials is the most common function of choice.

1.1 Lagrange interpolation

For any two points, there is a unique line. For any three points, there is a unique quadratic. In finite interval $[a, b]$ function $f(x)$ can always be represented by a polynomial $P(x)$. Find $P(x)$ from set of points $(x_i, f(x_i))$.

Linear interpolation between two points.

$$b = \frac{y_2(x_2) - y_1(x_1)}{x_2 - x_1} \tag{1}$$

$$a = y_1(x_1) - x_1 \frac{y_2(x_2) - y_1(x_1)}{x_2 - x_1} \tag{2}$$

$$y(x) = a + bx \tag{3}$$

Therefore

$$y(x) = \frac{x - x_2}{x_1 - x_2} y_1(x_1) + \frac{x - x_1}{x_2 - x_1} y_2(x_2) \tag{4}$$

$$= \frac{1}{x_1 - x_2}[(x - x_2)y_1(x_1) - (x - x_1)y_2(x_2)] \tag{5}$$

Is equation for line through $(x_1, y_1(x_1))$ and $(x_2, y_2(x_2))$.

Can improve result using second degree polynomial and employ quadratic interpolation. Next use 4 points and 3'rd degree polynomial. In general

$$P(x) = \sum_{k=1}^{N} \lambda_k(x) f(x_k)\,, \tag{6}$$

where

$$\lambda_k(x) = \frac{\prod_{l=1\neq k}^{N}(x - x_l)}{\prod_{l=1\neq k}^{N}(x_k - x_l)} \tag{7}$$

which is the Lagrange interpolation formula.

Lagrange method give no error estimations and is awkward to program. The Neville algorithm is much better.

1.2 Neville's algorithm

Linear interpolations between successive iterations. Degree polynomial is number of iterations.

1. First determine linear polynomial P_{ij} between neighboring points.
2. Next, interpolation between previously determined intermediate points. This interpolation is now polynomial of degree two.
3. Continue until only 1 point left.

By including more points in interpolation, increase degree of polynomial.

As an example, consider 5 measurements: $p_1(x_1), p_2(x_2), p_3(x_3), p_4(x_4), p_5(x_5)$. We want to interpolate the value at x. The first iteration is to determine the linear polynomial p_{ij} between neighbouring points.

$$p_{12} = \frac{1}{x_1 - x_2}[(x - x_2)p(x_1) - (x - x_1)p(x_2)] \tag{8}$$

$$p_{23} = \frac{1}{x_2 - x_3}[(x - x_3)p(x_2) - (x - x_2)p(x_3)] \tag{9}$$

$$p_{34} = \frac{1}{x_3 - x_4}[(x - x_4)p(x_3) - (x - x_3)p(x_4)] \tag{10}$$

$$p_{45} = \frac{1}{x_4 - x_5}[(x - x_5)p(x_4) - (x - x_4)p(x_5)] \tag{11}$$

We now have polynomials of degree 1. The second iteration is to determine the linear polynomial p_{ijk} between x values determined above.

$$p_{123} = \frac{1}{x_1 - x_3}[(x - x_3)p_{12} - (x - x_1)p_{23}] \tag{12}$$

$$p_{234} = \frac{1}{x_2 - x_4}[(x - x_4)p_{23} - (x - x_2)p_{34}] \tag{13}$$

$$p_{345} = \frac{1}{x_3 - x_5}[(x - x_5)p_{34} - (x - x_3)p_{45}] \tag{14}$$

We now have polynomials of degree 2. The third iteration is to determine the linear polynomial p_{ijkl} between x values determined above.

$$p_{1234} = \frac{1}{x_1 - x_4}[(x - x_4)p_{123} - (x - x_1)p_{234}] \tag{15}$$

$$p_{2345} = \frac{1}{x_2 - x_5}[(x - x_5)p_{234} - (x - x_2)p_{345}] \tag{16}$$

We now have polynomials of degree 3. The fourth iteration is to determine the linear polynomial p_{ijklm} between x values determined above.

$$p_{12345} = \frac{1}{x_1 - x_5}[(x - x_5)p_{1234} - (x - x_1)p_{2345}] \tag{17}$$

We now have a polynomial of degree 4.

1.3 Linear interpolation

Approximate data at a point x by a straight line passing through two data points x_j and x_{j+1} closest to x:

$$g(x) = a_0 + a_1 x \tag{18}$$

where a_0 and a_1 are coefficients of the linear function. To determine the coefficients

$$g(x_j) = f_j = a_0 + a_1 x_j \tag{19}$$

$$g(x_{j+1}) = f_{j+1} = a_0 + a_1 x_{j+1} \tag{20}$$

Solving

$$g(x) = f_j + \frac{x - x_j}{x_{j+1} - x_j}(f_{j+1} - f_j) = f_j \frac{x - x_{j+1}}{x_j - x_{j+1}} + f_{j+1} \frac{x - x_j}{x_{j+1} - x_j} \tag{21}$$

on the interval $[x_j, x_{j+1}]$.

Works well for very smooth function when 2'nd and higher derivatives are small. Improve quality by increasing number of data points on interval.

1.4 Polynomial interpolation

Remember Horner's rule.

$$P(x) = a_1 x^n + a_2 x^{n-1} + \ldots + a_{n+1} \tag{22}$$

requires $n(n+1)/2$ multiplications and n additions.

$$P(x) = a_{n+1} + x(a_n + x(a_{n-1} + x(\ldots(a_2 + a_1 x)\ldots))) \tag{23}$$

requires n multiplications and n additions.

Simple method

$$g(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \tag{24}$$

Condition that polynomial pass through points

$$f_j(x_j) = f(x_j) = a_0 + a_1 x_j + a_2 x_j^2 + \ldots + x_n x_j^n \tag{25}$$

A system of $n+1$ linear equations to determine coefficients a_j. Number of data points -1 used in interpolation defines order of interpolation. Linear (or two-point) interpolation is first order interpolation.

We need to solve a system of linear equations. Solving this gives back Lagrange's formula for polynomial interpolation.

Higher order polynomial improves interpolation. Too high can cause wild oscillations.

1.5 Cubic spline

Polynomial interpolation gives discontinuity of derivatives at data points. A spline is a polynomial between each pair of points, but one whose coefficients are determined slightly nonlocally. Coefficients for spline interpolations use all data points. i.e. nonlocal information, to get global smoothness up to some order of derivatives. Cubic splines are most popular. Cubic splines produce interpolated function that is continuous through the second derivative.

Interpolated function on interval (x_j, x_{j+1}).

$$g(x) = f_i + b_i(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3 \tag{26}$$

- 3 parameters b_j, c_j, d_j for each interval.
- $(n-1)$ intervals $\Rightarrow 3n-3$ equations for deriving coefficients.
- $g_j(x_j) = f_j(x_j)$ imposes $(n-1)$ equations.
- Require continuous 1st and 2nd derivatives at each $n-2$ points. $g'_{j-1}x_j = g'_j x_j$; $g''_{j-1}x_j = g''_j x_j$ These impose $2(n-2)$ equations.
- $n - 1 + 2(n-2) = 3n-5$ equations for coefficients $\Rightarrow$ 2 conditions are missing.

There are a few possibilities to fix the two conditions. Natural splines $\rightarrow$ 2nd order derivatives zero at boundaries. The natural cubic spline is the smoothest function which interpolates the data.

Generally, spline does not have advantages over polynomial interpolation when used for smooth, well behaved data, or when data points are close on the x axis. Splines good for sparse data.

1.6 Rational function interpolation

Ratio of two polynomials.

$$g(x) = \frac{a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n}{b_0 + b_1 x + b_2 x^2 + \ldots + b_m x^m} \tag{27}$$

Good for interpolation of function with poles. Two steps:

1. Need to choose powers for numerator and denominator n, m.
2. System of equations can be determined in n, m not too high relative to number of data points.

Can fix one parameter (e.g. b_0) as only ratios make sense.

"Introductory Computational Physics", A. Klein & A. Godunov, Cambridge University Press

PHYS 234: Computational Physics

Lecture 15: Wednesday, 4 March 2015

1 Derivatives

Avoid calculating derivatives on a computer if you can For a function $f(x)$, the derivative with respect to x is defined as

$$\frac{df(x)}{dx} = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h} \,. \tag{1}$$

Problem $f(x+h) - f(x)$ is small, and $h \to 0$ is small.

1.1 Forward difference

Taylor series around $x + h$ is

$$f(x+h) = f(x) + \frac{h}{1!}f'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \ldots \tag{2}$$

Solve for $f'(x)$

$$\begin{aligned} f'(x) &= \frac{1}{h}\left[f(x+h) - f(x) - \frac{h^2}{2!}f''(x) - \frac{h^3}{3!}f'''(x) - \ldots\right] &(3)\\ &= \frac{f(x+h) - f(x)}{h} - \left[\frac{h}{2!}f''(x) + \frac{h^2}{3!}f'''(x) + \ldots\right] &(4)\end{aligned}$$

Write forward difference as

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{5}$$

The result is almost like the definition of a derivative. The error is of order h

1.2 Central difference and higher order methods

Consider also Taylor series around $x - h$

$$f(x+h) = f(x) + \frac{h}{1!}f'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \ldots \tag{6}$$

$$f(x-h) = f(x) - \frac{h}{1!}f'(x) + \frac{h^2}{2!}f''(x) - \frac{h^3}{3!}f'''(x) + \ldots \tag{7}$$

Subtracting the two equations gives

$$f(x+h) - f(x-h) = \frac{2h}{1!} f'(x) + \frac{2h^3}{3!} f'''(x) + \text{odd terms} \tag{8}$$

which fives

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2) \tag{9}$$

We have increased the accuracy by one order in h.

This a 3-point formula: $x-h$, h, $x+h$. Can make a 5-point formula ($x-2h$, $x-h$, h, $x+h$, $x+2h$), etc.

$$f'(x) = \frac{1}{12h} \left[f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h) \right] + \mathcal{O}(h^4) \tag{10}$$

The above is written intuitively. There is a numerical advantage to group the equation as follows.

$$\begin{aligned} f_1 &= f(x-2h) + 8f(x+h) && (11) \\ f_2 &= 8f(x-h) + f(x+2h) && (12) \\ f'(x) &= \frac{1}{12h}(f_1 - f_2) && (13) \end{aligned}$$

1.3 Higher order derivatives

Adding, rather than subtracting, the two equations gives

$$f'' = \frac{f(x+h) + f(x-h) - 2f(x)}{h^2} + \mathcal{O}(h^2) \tag{14}$$

which is a three point formula.

"Introductory Computational Physics", A. Klein & A. Godunov, Cambridge University Press

PHYS 234: Computational Physics

Lecture 16: Friday, 6 March 2015

1 Numerical Integration

$$\int_a^b f(x)dx = \lim_{\Delta x \to 0} \sum_{i=0}^{n-1} f(x_i)\Delta x_i \tag{1}$$

1.1 Simple integration methods

1.1.1 Rectangular and trapezoid integration

Divide interval $[a, b]$ into N intervals, equally spaced $h = x_i - x_{i-1}$. Approximate the area by strips of width h and average height

$$\overline{f_i(x)} = \frac{f_i(x) + f_{i-1}(x)}{2} \tag{2}$$

or $\overline{f_i(x)} \approx f(x_{i-1/2})$ at $x_i - h/2$ implies

$$\int_a^b f(x)dx = h\sum_{i=1}^{N} f_{i-1/2} \tag{3}$$

which is the rectangular method.

Can determine area using trapezoids. 4-points $(x_i, 0), (x_i, f(x_i)), (x_{i+1}, f(x_{i+1})), (x_{i+1}, 0)$. Area of trapezoid

$$I_{[x_i,x_{i+1}]} = h\frac{f(x_i) + f(x_{i+1})}{2} = \frac{h}{2}[f(x_i) + f(x_{i+1})] \tag{4}$$

The endpoints are used only once. The other points are used twice: lower and upper edges.

$$\int_a^b f(x)dx = h\left[\frac{1}{2}f(x_0 = a) + f(x_0 + h) + \ldots + \frac{1}{2}f(x_N = b)\right] \tag{5}$$

$$= \frac{h}{2}\left[f(x_0 = a) + 2\sum_{i=1}^{N-1} f(x_0 + ih) + f(x_N = b)\right] \tag{6}$$

The error resulting from approximating the true area in a strip under the curve $f(x)$ between x_i and x_{i+1} by the area of trapezoid is

$$E_{T_i} = -\frac{1}{12}f''(\xi)h^3 \quad x_i < \xi < x_{i+1} \tag{7}$$

1.1.2 Simpson method

Approximate function in each interval by parabola. Taylor series at midpoint in $[x_{i-1}, x_{i+1}]$

$$\begin{aligned}\int_{x_{i-1}}^{x_{i+1}} f(x)dx &= f(x_i)\int_{x_{i-1}}^{x_{i+1}} dx + \frac{f'(x_i)}{1!}\int_{x_{i-1}}^{x_{i+1}}(x-x_i)dx + \frac{f''(x_i)}{2!}\int_{x_{i-1}}^{x_{i+1}}(x-x_i)^2dx \quad (8)\\ &+\frac{f'''(x_i)}{3!}\int_{x_{i-1}}^{x_{i+1}}(x-x_i)^3dx + \ldots + \frac{f^{(n)}(x_i)}{n!}\int_{x_{i-1}}^{x_{i+1}}(x-x_i)^n dx \quad (9)\end{aligned}$$

$x_{i+1} - x_i = h$ and $x_i - x_{i-1} = h$. Because expansion is symmetrical around x_i all terms with odd derivatives vanish (integral 0)

$$\int_{x_{i-1}}^{x_{i+1}} f(x)dx = f(x_i)\cdot 2h + f''(x_i)\cdot\frac{2h^3}{3\cdot 2!} + \mathcal{O}(h^5 f^{(4)}(x_i)) \quad (10)$$

Replace 2nd derivative is

$$f''(x) = \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} \quad (11)$$

we have

$$\begin{aligned}\int_{x_{i-1}}^{x_{i+1}} f(x)dx &= 2hf(x_i) + \frac{h}{3}\left[f(x_{i-1}) - 2f(x_i) + f(x_{i+1})\right] + \mathcal{O}(h^5 f^{(4)}(x_i)) \quad (12)\\ &= h\left[\frac{1}{3}f(x_{i-1}) + \frac{4}{3}f(x_i) + \frac{1}{3}f(x_{i+1})\right] + \mathcal{O}(h^5 f^{(4)}(x_i)) \quad (13)\end{aligned}$$

For entire interval

$$\begin{aligned}\int_a^b f(x)dx &= \frac{h}{3}\left[f(a) + 4f(a+h) + 2f(a+2h) + 4f(a+3h) + \ldots + f(b)\right] \quad (14)\\ &= \frac{h}{3}\left[f(x) + 4\sum_{i=1,3,5,\ldots}^{N-1} f(a+ih) + 2\sum_{i=2,4,6,\ldots}^{N} f(a+ih) + f(b)\right] \quad (15)\end{aligned}$$

Need even number of intervals, odd number of points.

The error resulting from approximating the true area in two strips under the curve $f(x)$ between x_{i-1} and x_{i+1} by the area under a second-degree parabola is

$$E_{T_i} = -\frac{1}{90}f^{(4)}(\xi)h^5 \quad x_{i-1} < \xi < x_{i+1} \quad (16)$$

1.2 Considerations

In general

$$\int_a^b f(x)dx = \sum_{i=1}^{N} f(x_i)w_i \tag{17}$$

where w_i are weights. For trapezoidal method $(h/2, h/2)$ weights. For Simpson method $(h/3, 4h/3, h/3)$ weights. In these methods the points equally spaced. The methods are known as quadrature. They approximate the integral by a polynomial in each slice.

More advanced methods approximate the integral by a polynomial over the entire interval.

An adaptive quadrature routine is a numerical quadrature algorithm which uses one or two basic quadrature rules and which automatically determines the subinterval sizes so that the computed result meets some prescribed accuracy requirement.

Bear in mind that when computing the value of a definite integral by means of a quadrature formula you are really replacing the given integrand by a polynomial and integrating this polynomial over the given interval of integration. The accuracy of the result will depend upon how well the polynomial represents the integrand over this interval; or, geometrically, on how will the graph of the polynomial coincides with the graph of the integrand. Before beginning the computation of an integral by a quadrature formula the computer should ascertain the nature and behaviour of the integrand over the interval of integration. In some instances it may be necessary to construct an accurate graph of the integrand. The computation can then be planned with reference to the nature and behaviour of the function to be integrated.

"Introductory Computational Physics", A. Klein & A. Godunov, Cambridge University Press

PHYS 234: Computational Physics

Lecture 17: Monday, 9 March 2015

1 Steepest Descent

1.1 Least-squares fit to an arbitrary function

When fitting data y_i with a function $y(x)$ which is not linear in the parameters, this is called the method of non-linear least squares. Consider the function $y(x, a_j)$ with parameters a_j.

We can define a measure of goodness of fit χ^2

$$\chi^2 = \sum \left[\frac{1}{\sigma_i^2} \left[y_i - y(x_i) \right]^2 \right] \tag{1}$$

where σ_i are the uncertainties in the data points y_i.

There are three sources of error that contribute to the size of χ^2:

1. The data y_i are a random sample from the parent population with expected values $\langle y_i \rangle$ given by the parent distribution. The fluctuations of the y_i about the expected values $\langle y_i \rangle$ may be statistically greater or less than the expected uncertainty σ_i.

2. χ^2 is a continuous function of all parameters a_j.

3. The choice of the functional behaviour of the analytical function $y(x)$ as an approximation to the "true" function $\langle f(x) \rangle$ will influence the range of possible values for χ^2.

Nothing can be done about the contributions from 1. without repeating the experiment. The optimum values for the parameters a_j can be estimated with the least-squares method by minimising the contributions from 2. The resulting value of χ^2 for several different functions $y(x)$ can be compared to determine the most probable functional form for $y(x)$ as in 3.

1.2 Method of least squares

According to the method of least squares, the optimum values of the parameters a_j are obtained by minimising χ^2 with respect to each of the parameters simultaneously.

$$\frac{\partial}{\partial a_j} \chi^2 = \frac{\partial}{\partial a_j} \sum \left[\frac{1}{\sigma_i^2} \left[y_i - y(x_i) \right]^2 \right] = 0 \tag{2}$$

It is generally not convenient to derive an analytical expression for calculating the parameters of a non-linear function $y(x)$. Instead, χ^2 must be considered a continuous function of the

n parameters a_j describing a hypersurface in the n-dimensional space. The space must be searched for the appropriate minimum value of χ^2.

One of the difficulties of such a search is that for an arbitrary function there may be more than one local minimum for χ^2 within a reasonable range of values for the parameters a_j.

In the simplest mapping procedure, the permissible range for each parameter a_j is divided into n equal increments Δa_j so that the n-parameter space is divided into $\prod_{j=1}^{n} n_j$ hypercubes. The value of χ^2 is then evaluated at each of the vertices of these hypercubes. This procedure yields a course map of the behaviour of χ^2 as a function of all the parameters a_j.

1.3 Grid search

If the variation of χ^2 with each parameters a_j is fairly independent of how well optimised the other parameters are, then the optimum values can be determined most simply by the minimising the χ^2 with respect to each parameter separately. With successive iterations of locating the local minimum for each parameter in turn, the absolute minimum may be located with any desired precision. The main disadvantage is that if the variation of χ^2 with various parameters are not independent, then this method may converge very slowly. The simplicity of a grid search can sometimes compensate for this inefficiency.

The procedure of the grid search is as follows:

1. One parameter a_j is incremented by a quantity Δa_j, where the magnitude of this quantity is specified and the sign is chosen such that χ^2 decreases.

2. The parameter a_j is repeatedly incremented by the same amount Δa_j until χ^2 starts to increase.

3. Assuming the variation of χ^2 near the minimum can be described in terms of a parabolic function of the parameter a_j, we can use the values of χ^2 for the last three values of a_j to determine the minimum of the parabola. $a_j(3) = a_j(2) + \Delta a_j = a_j(1) + 2\Delta a_j$, $\chi^2(3) > \chi^2(2) \leq \chi^2(1)$.

4. The minimum of the parabola is given by

$$a_j(\min) = a_j(3) - \Delta a_j \left[\frac{\chi^2(3) - \chi^2(2)}{\chi^2(3) - 2\chi^2(2) + \chi^2(1)} + \frac{1}{2} \right] \tag{3}$$

5. χ^2 is minimised for each parameter in turn.

6. The above procedure is repeated until the last iteration yields a negligibly small decrease in χ^2.

1.4 Gradient search

In the gradient-search method of least squares, all the parameters a_j are incremented simultaneously, with the relative magnitudes adjusted so that the resultant direction of travel is parameter space is along the gradient (or direction of maximum variation) of χ^2.

The gradient $\vec{\nabla}\chi^2$, or direction in which χ^2 increases most rapidly, is a vector whose components are equal to the rate at which χ^2 increase in that direction

$$\vec{\nabla}\chi^2 = \sum_{j=1}^{n} \left[\frac{\partial \chi^2}{\partial a_j} \hat{a}_j \right] \tag{4}$$

In order to determine the gradient, the variation of χ^2 in the neighbourhood of the starting point is sampled independently for each parameter to yield an approximate value for the first derivative

$$(\vec{\nabla}\chi^2)_j = \frac{\partial \chi^2}{\partial a_j} \approx \frac{\chi^2(a_j + f\Delta a_j) - \chi^2(a_j)}{f\Delta a_j} \tag{5}$$

The amount by which a_j is changed in order to determine this derivative should be smaller than the step size Δa_j. The fraction f should be on the order of 10% ($f = 0.1$).

With this definition, the gradient has both magnitude and dimensions. In fact, if the dimensions of the various parameters a_j are not all the same, the components of the gradient do not even have the same dimensions. Let us define the dimensionless parameter b_j by normalising each of the parameters a_j to a size constant Δa_j which characterises the variation of χ^2 with a_j rather roughly

$$b_j \equiv \frac{a_j}{\Delta a_j} \tag{6}$$

We then define a dimensionless gradient γ with a magnitude of unity.

$$\gamma_j = \frac{\partial \chi^2 / \partial b_j}{\sqrt{\sum_{k=1}^{n} \left(\frac{\partial \chi^2}{\partial b_k} \right)^2}} \quad \frac{\partial \chi^2}{\partial b_j} = \frac{\partial \chi^2}{\partial a_j} \Delta a_j \tag{7}$$

The direction which the gradient-search method follows is the direction of steepest descent, which is the opposite direction from the gradient γ. The search begins by incrementing all the parameters simultaneously by an amount δa_j whose relative value is given by the corresponding components of the dimensionless gradient γ, and whose absolute magnitude is given by the size constant Δa_j.

$$\delta a_j = -\gamma_j \Delta a_j \tag{8}$$

The minus sign ensures that the value of χ^2 decreases. The size constant Δa_j must be the same in both equations.

There are several choices for methods of continuing the gradient search.

A reasonable perturbation on the method is to search along the direction of the original gradient in small steps, calculating only the value of χ^2 until the value of χ^2 begins to raise again. At this point, the gradient is recomputed and the search begins again in the new direction. Whenever the search straddles a minimum, a parabolic interpolation of χ^2 will improve the precision of locating the bottom of the minimum.

A more sophisticated approach is to use second partial derivatives of χ^2 calculated with finite difference to determine modifications to the gradient along the search path

$$\left.\frac{\partial\chi^2}{\partial a_j}\right|_{a_j+\delta a_j} \sim \left.\frac{\partial\chi^2}{\partial a_j}\right|_{a_j} + \sum_{k=1}^{n}\left(\frac{\partial^2\chi^2}{\partial a_j \partial a_k}\delta a_k\right) \tag{9}$$

If the search is already fairly near the minimum, this method does decrease the number of steps needed, but at the expense of more elaborate computation. If the search is note near enough to the minimum, this method can actually increase the number of steps required if first-order perturbations on the gradient are not valid. For this reason we will ignore this type of approach.

The gradient search suffers markedly as the search approaches the minimum because the evaluation of the derivatives according to the method consists of taking differences between nearly equal numbers. In fact, at the minimum of χ^2, these differences should vanish.

"Data Reduction and Error Analysis for the Physical Sciences", P.R. Bevington, McGraw-Hill

PHYS 234: Computational Physics

Lecture 18: Wednesday 11 and Friday 13 and Monday 16 March 2015

1 Differential equations

An equation has a number as a solution. A differential equation has a function as a solution. Finding the general solution is called "integration" of the equation. Differential equations are defined and classified accordingly.

- ODE = ordinary differential equation
- PDE = partial differential equation
- linear and non-linear
- homogeneous and non-homogeneous

We make use of the fact that any nth order linear differential equation can be reduced to n coupled linear differential equation.

$$y^{(n)} = f(x, y, y', y'', \ldots, y^{(n-1)}) \tag{1}$$

can be the following substitution

$$\begin{aligned} u_1 &= y' && (2)\\ u_2 &= y'' && (3)\\ &\vdots \\ u_{n-1} &= y^{(n-1)} && (4) \end{aligned}$$

by cast into a system of n linear equations

$$\begin{aligned} y' &= u_1 && (5)\\ u_1' &= u_2 && (6)\\ u_2' &= u_3 && (7)\\ &\vdots \\ u_{n-2}' &= u_{n-1} && (8)\\ u_{n-1}' &= f(x, y, u_1, u_2, \ldots, u_{n-1}) && (9) \end{aligned}$$

The approach we shall be concerned with involves the step-by-step methods (also called difference methods or discrete variable methods). A sequence of discrete points $t_0, t_1, t_2, \ldots$ is generated. At each time t_n, the solution $y(t_n)$ is approximated by a number y_n which is computed form earlier values. Starting with the initial values, the solutions are constructed by short steps ahead for equal intervals $\Delta x = h$ of x, each step usually being checked by some method before proceeding to the next step.

1.1 Simple Euler method

Consider the approximate derivative

$$y'(x_0) \approx \frac{y(x_0 + h) - y(x_0)}{(x_0 + h) - x_0} \tag{10}$$

We want $y(x_0 + h)$

$$y(x_0 + h) \approx y(x_0) + hy'(x_0) \tag{11}$$

This is the Euler forward integration equation. It is the same as the first 2 terms of the Taylor series.

Example: $xy' + y = 0 \Rightarrow y' = -\frac{y}{x}$, analytic solution $y = C/x$.

Using the above equation

$$y(x_0 + h) = y(x_0) - h\frac{y(x_0)}{x_0} \tag{12}$$

1. start from point x_0, divide interval between x_0 and x_f into N equidistant steps of size h

2. calculate y at $x_0, x_0 + h, x_0 + 2h, \ldots, x_0 + Nh$.

This results in the true curve begin approximated by a series of straight-line segments. This is consider a self-starting method since it requires a value of th dependent variable at only one point to start the procedure.

The truncation error is

$$E_{t_i} \approx \frac{y_i'' h^2}{2} \tag{13}$$

1.2 Modified Euler method

Previous method use derivative at beginning of interval. We can can use midpoint of integral between x and $x + h$.

Example: $xy' + y = 0$.

$$y(x_0 + h) = y(x_0) + hy'(x_0 + h/2) \tag{14}$$

At midpoint

$$y\left(x_0+\frac{h}{2}\right)=y(x_0)+\frac{h}{2}y'(x_0) \tag{15}$$

For example $y'=-y/x$.

$$y'(x_0+h/2)=-\frac{y(x_0+h/2)}{x_0+h/2} \tag{16}$$

$$y(x_0+h)=y(x_0)-h\frac{y(x_0+h/2)}{x_0+h/2} \tag{17}$$

1.3 Modified Euler method; predictor-corrector

Consider a first-order differential equation of the form

$$y'=f(x,y) \tag{18}$$

where y is known when $x=0$. If we substitute the known initial value of y into the above equation, we obtain the value of y' at $x=0$. Next, a predicted value of y at $x=\Delta x$ is found by using Euler's equation

$$P(y_1)=y_0+y_0'h\,, \tag{19}$$

where $P(y_1)$ is the predicted value of y_1 and $h=\Delta x$. In obtaining the predicted value of y_1, the term $y_0'h$ is the rectangular area. This area is larger than the true area under the given curve, so the predicted value of y_1 obtained is too large by some small amount. If the predicted value of y_1 is substituted into the given differential equation along with $x=h$, and approximate value of y_1', $P(y_1)$, we will use the notation $P(y_1')$ to represent it. Using the trapezoidal area as the approximation of the true area under the y' curve, a corrected value of y_1, which we will denote by $C(y_1)$, can be determined

$$C(y_1)=y_0+\left[\frac{y_0'+P(y_1')}{2}\right]h\,. \tag{20}$$

This is known as the corrector equation. The corrected value of y_1 is substituted to obtain a corrected value of y_1'. The later value is denoted by $C(y_1')$. The iteration process continues by using $C(y_1')$ in place of $P(y_1')$ to obtain a still better value of y_1, and using this improved value to get a further improved value of y_1'. The process is repeated until successive values of y_1 differ by less than some prescribed epsilon value selected to specify the accuracy desired.

With the desired value of y_1 obtained, we are ready to move ahead one step to determine the value of y_2. This begins by using the predictor equation

$$P(y_2)=y_1+y_1'h\,, \tag{21}$$

where $P(y_2)$ is the first predicted value of y_2, and y_1 and y_1' are the most accurate values obtained fro these quantities in the preceding iteration. The iterative process described for determining an accurate value of y_1 is then repeated, and so on.

The general form for application at any step is

$$\begin{aligned} (\text{predictor})P(y_{y+i}) &= y_i + y_i' h \quad (22) \\ (\text{corrector})C(y_{y+i}) &= y_i + \left[\frac{y_i' + P(y_{i+1}')}{2}\right] h \quad (23) \end{aligned}$$

Since the corrector equation is based on using a trapezoidal area, the per-step truncation error of this method is

$$E_{TC_i} = -\frac{1}{12} y_i''' h^3 \quad (24)$$

A more advance method is Runge-Kutta Method (4th order). The Rung-Kutta method is designed to approximate Taylor series method without requiring explicit differentiation of, or evaluation of, derivatives beyond the first. The approximation is obtained at the expense of several evaluations of the function.

1.4 Error in the numerical solutions

The use of numerical methods for solving differential equations generally yields solutions which differ from the true solutions. The difference between the numerical solution and the true solution, at any given step is known as the total error at that step. If the numerical solution is to be of practical value, the total error must be kept within reasonable limits over the desired solution interval. The primary purpose of error analysis is to provide a means of controlling this error. The total error at any step results from the following conditions:

1. A roundoff error is introduced in the integration process at a given step by performing the arithmetic operations of that step with numerical values having limited number of significant digits. This is knowns as the per-step roundoff error. Roundoff error is generally present to some degree in each step of a numerical-integration process. Since its magnitude in each step depends primarily upon the digital capacity of the computer begin used, the per-step roundoff error is essentially independent of the step. The total error due to roundoff increases with the number of steps.

2. A truncation error is introduced in the integration process at a given step by the use of approximate formulas in the calculations of that step. This is known as per-step truncation error. Truncation error also appears in each step of the numerical process. Unlike roundoff error, the per-step truncation error is a function of the step size, since it varies as the order of error, h^n, of the method begin used. Thus the only means of controlling the overall per-step error of a method at a given step is by controlling the truncation error at that step.

3. An error is present at a given step because of errors introduced in proceeding steps.

Since the total error in a numerical-integration process depends on the step size used, the analyst is faced with the problem of selecting a step size to maintain the total error less than some specified tolerance. On the other hand, there is no point in making the solution more accurate than the data justified by choosing too small a step size, since this unnecessarily uses machine resources. Furthermore, there is a limit to the increase in accuracy which can be obtained by decreasing the step size, owing to the presence of roundoff error. The total error is decreased as the per-step truncation error is decreased up to the point where the increasing number of steps and the presence of roundoff error causes an increase in the total error. Thus the total error can be kept to a minimum by maintaining the per-step truncation error within some appropriate tolerance.

Other tricks that can be played, is to dynamically change the step size to keep a constant error (allowed tolerance). Or, we can use a fixed step size but repeat the calculation reducing h by one-half each time until desired tolerance is reached.

When the range of integration is relatively short, relatively small step sizes can be used, and there is little reason to analyse the truncation error, or complicate the solution by using an advanced method.

In physics (as opposed to pure math) we can invoke physical principles like conservation of energy to check the results.

"Introductory Computational Physics", A. Klein & A. Godunov, Cambridge University Press
"Applied Numerical Methods for Digital Computation With FORTRAM and CSMP", M.L. James, G.M. Smith, J.C. Wolford, IEP - A Dun-Donnelley Publisher

PHYS 234: Computational Physics

Lecture 19: Friday 20 and Monday 23 March 2015

1 Solution of simultaneous algebraic equations

We will discuss two cases

1. linear (normal case) systems of equations
2. eigenvalue problem

On a computer, matrices are used for both cases.

1.1 Linear systems of equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \quad (1)\\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \quad (2)\\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots &\ldots \ldots\\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &= b_m \quad (3) \end{aligned}$$

$x_j, j = 1, \ldots, n$ are a set of unknowns, and $b_i, i = 1, \ldots, m$ and a_{ij} are coefficients.

1. $m > n$, number equations greater than unknowns: over determined
2. $m < n$, number equations less than unknowns: under determined (can not be solved)
3. $m = n$, the case we will consider

In matrix notation

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \cdots \\ b_n \end{pmatrix} \quad (4)$$

or

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \quad (5)$$

A is called the coefficient matrix. If the constants on the right-hand side of the equations are included as a column of elements in the position shown below, the resulting matrix is called the augmented matrix.

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{pmatrix} \tag{6}$$

1. nonhomogeneous equations: if $b_i \neq 0$ unique solution when $|\mathbf{A}| \neq 0$ (all equations independent)

2. homogeneous equations: if all $b_i = 0$, nontrivial solution iff $|\mathbf{A}| = 0$ (all equations not independent)

Cramer's rule using determinants is often used. This method consumes too much computer resources and is not practical for large systems of equations. $(n-1)(n+1)!$ multiplications; $n = 10 \Rightarrow\approx 3 \times 10^8$ multiplications. Another possibility is to write the system of equations as a matrix equation and solve for the inverse matrix. The inverse requires a lot of arithmetic and produces less accurate results. We will consider an elimination method. The type of method is sometimes referred to as an elimination method, a reduction method, or a direct method.

1.2 Gaussian elimination

The basic idea is to add or subtract linear combinations of the given equations until each equation contains only one of the unknowns, thus giving an immediate solutions.

Using Gauss's method a set of n equations in n unknowns is reduced to an equivalent triangular set (an equivalent set is a set having identical solution values), which is then easily solved by back substitution. Consider 3 linear equation, for example,

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1 \tag{7}$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2 \tag{8}$$
$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \tag{9}$$

$$(8) - a_{21}/a_{11} \times (7) \Rightarrow 0 + a'_{22}x_2 + a'_{23}x_3 = b'_2 \tag{10}$$

$$(9) - a_{31}/a_{11} \times (7) \Rightarrow 0 + a'_{32}x_2 + a'_{33}x_3 = b'_3 \tag{11}$$

In general

$$a'_{ij} = a_{ij} - a_{1j}a_{i1}/a_{11} \quad b'_i = b_i - b_1 a_{i1}/a_{11} \tag{12}$$

for $i = 2, \ldots, n$ and $j = 1, \ldots, n$. The equation (7) used to eliminate the unknowns in the equations is called the pivot equation. The element we divided by a_{11} is called the pivot element, or pivot.

The last 2 equations form a system of 2 equations and 2 unknowns. Repeating

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 && (13)\\ a'_{22}x_2 + a'_{23}x_3 &= b'_2 && (14)\\ a''_{33}x_3 &= b''_3 && (15) \end{aligned}$$

$$a''_{ij} = a'_{ij} - a'_{2j}a'_{i2}/a'_{22} \quad b''_i = b'_i - b'_2a'_{i2}/a'_{22} \tag{16}$$

for $i = 3, \ldots, n$ and $j = 2, \ldots, n$. The pivot in this case is a_{22}. The result is now a triangle set of equation.

Repeating procedure $n-1$ times to obtain $x_3 = b''_3/a''_{33}$. This is forward elimination.

In general, the elements of the reduced matrix $\mathbf{A}'$ can be written directly from the original matrix $\mathbf{A}$, using

$$a^k_{ij} = a^{k-1}_{ij} - \frac{a^{k-1}_{ik}}{a^{k-1}_{kk}}a_{kj} \quad k+1 \leq j \leq m \quad k+1 \leq i \leq n \tag{17}$$

where

i is the row number of the matrices
j is the column number of the matrices
k is the number of the identifying pivot row
n is the number of rows of the matrices
m is the number of columns of the matrices

Verify with $k = 1$, then $k = 2$, etc.

1.2.1 Back substitution

Doing back substitution, we find x_2 then x_1.

$$x_2 = \frac{1}{a'_{22}}(b'_2 - x_3a'_{23}) \tag{18}$$

In general, $x_n = a_{nm}/a_{mm}$ and

$$x_i = \frac{1}{a'_{ii}}\left(a'_{im} - \sum_{j=i+1}^{n} a'_{ij}x_j\right) \tag{19}$$

for $i = n-1, n-2, \ldots, 1$. The combination of Gaussian elimination and back substitution yield a solution to the set of equation.

1.2.2 Considerations and solution on computer

Zero diagonal elements If one diagonal element is zero, the procedure fails. Can interchange rows. Pushing elements which are zero off the diagonal is called partial pivoting. Interchanging the rows is called partial pivoting. Complete pivoting consists of interchanging columns as well as rows. One criteria for complete pivoting is to reorder equations so that $a_{11} > a_{22} > a_{33} > \ldots > a_{nn}$ can increases efficiency.

Roundoff errors Calculating the fractions involved in the method results in loss of precisions. Limits the number of equations that can be solved. Roundoff error will accumulate and be high for large n.

Ill-conditioned systems ($|A| \approx 0$). If a small relative change in one or more of the coefficients of a system of equations results in a small relative change in the solution, the system of equations is called a well-conditioned system. If the coefficients are sparse more equations can be solved before roundoff errors become significant.

Application to computer Since the computer can handle only numerical data, the used of matrices is required in programming. The augmented matrix is usually used. The pivot equation becomes the pivot row in the matrix. The pivot coefficient becomes the pivot element.

As the number of equations in the system increases, the computation time grows nonlinearly. The most powerful method to solve most systems of linear equations is the use of standard libraries. Should check solution by direct substitution of $x_1, x_2, \ldots, x_n$ into original system.

1.3 Gaussian elimination used in determinant evaluation

Cramer's rule is not recommended for solution of linear simultaneous equations. But sometimes it is necessary to evaluate a determinant. For large determinants, the used of Gaussian elimination is much more efficient than expansion by minors. From the definition of a determinant and the rules of expansion by minors, we have

1. A determinant is changed only in sign by interchanging any two of its rows (or columns).
2. A determinant is not changed in value if we add (or subtract) a constant multiple of the elements of a row (or column) to the corresponding elements of another row (or column).

We can apply the method of Gaussian elimination to a determinant without changing its value. Therefore can reduce the determinant to an upper triangular form without changing its value. The value of the determinant is then simply the product of the diagonal elements. Partial pivoting may be used but the resulting sign changes must be taken into account.

1.4 Gaussian-Jordan elimination method

This procedure varies from the Gaussian method in that, when an unknown is eliminated, it is eliminated from all the other equations, that is, from those preceding the pivot equation as well as those following it. This eliminates the necessity of using the back substitution process employed in Gauss's method.

It should be noted that, if it were desired to solve a second set of simultaneous equations which differed from a first set only in the constant terms which appeared, both sets would be solved at the same time by representing each set of constants as a separate column in augmenting the coefficient matrix.

1.5 Matrix inversion

When it is necessary to solve a large number of different sets of simultaneous equations which differ only by the constant values appearing in the respective equations, the matrix-inversion method may be used to advantage in reducing the number of operations required.

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{20}$$

where $\mathbf{A}$ is the coefficient matrix, $\mathbf{x}$ the column matrix of unknowns and $\mathbf{b}$ the column matrix of constants. The inverse $\mathbf{A}^{-1}$ is given by

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I} \tag{21}$$

This is equivalent to n sets of simultaneous equations. Previously, it was stated that the Gauss-Jordan elimination method could be utilised to solve simultaneously n sets of simultaneous equations, associated with each set of equations on the right side of the coefficient matrix in the augmented matrix.

In this case we can write the augmented matrix as

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} & 1 & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & a_{2n} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & 0 & 0 & \cdots & 1 \end{pmatrix} \tag{22}$$

and solve for the inverse matrix coefficients by using Gauss-Jordan elimination.

1.6 Eigenvalue problem

$$\mathbf{A} \cdot \mathbf{x} = \lambda \mathbf{x} \tag{23}$$

Write system of equations out

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = \lambda x_1 \tag{24}$$
$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = \lambda x_2 \tag{25}$$

$$\ldots\ldots\ldots\ldots\ldots \quad \ldots \tag{26}$$

$$a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = \lambda x_n \tag{27}$$

The equations above look like a linear system of equations. However, there is a substantial difference between the eigenvalue problem and the linear system of equations. For the eigenvalue problem the coefficients λ are unknown and solution for system exists only for specific values of λ. These values are called the eigenvalues.

Regrouping gives

$$(a_{11} - \lambda)x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = 0 \tag{28}$$

$$a_{21}x_1 + (a_{22} - \lambda)x_2 + \ldots + a_{2n}x_n = 0 \tag{29}$$

$$\ldots\ldots\ldots\ldots\ldots \quad \ldots \tag{30}$$

$$a_{n1}x_1 + a_{n2}x_2 + \ldots + (a_{nn} - \lambda)x_n = 0 \tag{31}$$

Can write as

$$(\mathbf{A} - \lambda\mathbf{I}) \cdot \mathbf{x} = 0 \tag{32}$$

Solution iff

$$\det|\vec{A} - \lambda\vec{I}| = 0 \tag{33}$$

Solutions is polynomial in λ of degree n

$$c_n\lambda^n + c_{n-1}\lambda^{n-1} + \ldots + c_1\lambda + c_0 = 0 \tag{34}$$

This is the characteristic equation of $\mathbf{A}$.

Roots give eigenvalues.

For symmetric $a_{ij} = a_{ji}$ or hermitian $a_{ij} = a_{ij}^* \Rightarrow$ eigenvalues real.

Once λ are solved for, can solve for $\mathbf{x}$ (eigenvectors).

When the order of the matrix becomes large, computer methods must be adapted to efficiently solve the problem. Unstable as size of matrix increases.

"Introductory Computational Physics", A. Klein & A. Godunov, Cambridge University Press
"Applied Numerical Methods for Digital Computation With FORTRAM and CSMP", M.L. James, G.M. Smith, J.C. Wolford, IEP - A Dun-Donnelley Publisher

PHYS 234: Computational Physics

Lecture 20: Monday 23 and Wednesday 25 March 2015

1 Linear regression

Chi-squared for a straight line

$$\chi^2 = \sum \left[\frac{1}{\sigma_i^2} (y_i - a - bx_i)^2 \right] \tag{1}$$

1.1 Constant errors

Minimum value of chi-squared with $\sigma_i = \sigma$

$$\begin{aligned} \frac{\partial}{\partial a}\chi^2 &= \frac{\partial}{\partial a}\left[\frac{1}{\sigma^2} \sum (y_i - a - bx_i)^2 \right] & (2) \\ &= \frac{-2}{\sigma^2} \sum (y_i - a - bx_i) = 0 & (3) \end{aligned}$$

$$\begin{aligned} \frac{\partial}{\partial b}\chi^2 &= \frac{\partial}{\partial b}\left[\frac{1}{\sigma^2} \sum (y_i - a - bx_i)^2 \right] & (4) \\ &= \frac{-2}{\sigma^2} \sum [x_i(y_i - a - bx_i)] = 0 & (5) \end{aligned}$$

These equations can be rearranged to yield a pair of simultaneous equations

$$\begin{aligned} \sum y_i &= \sum a + \sum bx_i = aN + b\sum x_i & (6) \\ \sum x_i y_i &= \sum ax_i + \sum bx_i^2 = a\sum x_i + b\sum x_i^2 & (7) \end{aligned}$$

We solve the equations for the coefficients a and b. This will give us the values of th coefficients for which χ^2 is a minimum. Let's do it using determinants.

$$\begin{aligned} a &= \frac{1}{\Delta} \begin{vmatrix} \sum y_i & \sum x_i \\ \sum x_i y_i & \sum x_i^2 \end{vmatrix} = \frac{1}{\Delta}(\sum x_i^2 \sum y_i - \sum x_i \sum x_i y_i) & (8) \\ b &= \frac{1}{\Delta} \begin{vmatrix} N & \sum y_i \\ \sum x_i & \sum x_i y_i \end{vmatrix} = \frac{1}{\Delta}(N\sum x_i y_i - \sum x_i \sum y_i) & (9) \\ \Delta &= \begin{vmatrix} N & \sum x_i \\ \sum x_i & \sum x_i^2 \end{vmatrix} = N\sum x_i^2 - (\sum x_i)^2 & (10) \end{aligned}$$

1.2 Weighting the fit

If the uncertainties are not equal throughout, it is necessary to reintroduce the standard deviation as a weighting factor. Minimising the χ^2 gives

$$\frac{\partial}{\partial a}\chi^2 = \frac{\partial}{\partial a}\sum\left[\frac{1}{\sigma_i^2}(y_i - a - bx_i)^2\right] \tag{11}$$

$$= -2\sum\left[\frac{1}{\sigma_i^2}\sum(y_i - a - bx_i)\right] = 0 \tag{12}$$

$$\frac{\partial}{\partial b}\chi^2 = \frac{\partial}{\partial b}\sum\left[\frac{1}{\sigma_i^2}(y_i - a - bx_i)^2\right] \tag{13}$$

$$= -2\sum\left[\frac{x_i}{\sigma_i^2}(y_i - a - bx_i)\right] = 0 \tag{14}$$

These equations can be rearranged to yield a pair of simultaneous equations

$$\sum\frac{y_i}{\sigma_i^2} = a\sum\frac{1}{\sigma_i^2} + b\sum\frac{x_i}{\sigma_i^2} \tag{15}$$

$$\sum\frac{x_i y_i}{\sigma_i^2} = a\sum\frac{x_i}{\sigma_i^2} + b\sum\frac{x_i^2}{\sigma_i^2} \tag{16}$$

The solutions are

$$a = \frac{1}{\Delta}\begin{vmatrix}\sum\frac{y_i}{\sigma_i^2} & \sum\frac{x_i}{\sigma_i^2}\\ \sum\frac{x_i y_i}{\sigma_i^2} & \sum\frac{x_i^2}{\sigma_i^2}\end{vmatrix} = \frac{1}{\Delta}\left(\sum\frac{x_i^2}{\sigma_i^2}\sum\frac{y_i}{\sigma_i^2} - \sum\frac{x_i}{\sigma_i^2}\sum\frac{x_i y_i}{\sigma_i^2}\right) \tag{17}$$

$$b = \frac{1}{\Delta}\begin{vmatrix}\sum\frac{1}{\sigma_i^2} & \sum\frac{y_i}{\sigma_i^2}\\ \sum\frac{x_i}{\sigma_i^2} & \sum\frac{x_i y_i}{\sigma_i^2}\end{vmatrix} = \frac{1}{\Delta}\left(\sum\frac{1}{\sigma_i^2}\sum\frac{x_i y_i}{\sigma_i^2} - \sum\frac{x_i}{\sigma_i^2}\sum\frac{y_i}{\sigma_i^2}\right) \tag{18}$$

$$\Delta = \begin{vmatrix}\sum\frac{1}{\sigma_i^2} & \sum\frac{x_i}{\sigma_i^2}\\ \sum\frac{x_i}{\sigma_i^2} & \sum\frac{x_i^2}{\sigma_i^2}\end{vmatrix} = \sum\frac{1}{\sigma_i^2}\sum\frac{x_i^2}{\sigma_i^2} - \left(\sum\frac{x_i}{\sigma_i^2}\right)^2 \tag{19}$$

1.3 Estimate of σ

If the fluctuations in the measurements y_i are statistical, we can estimate analytically what the standard deviation corresponding to each observation is, without having to determine it experimentally. If we were to make the same measurement repeatedly, we would find that the observed values were distributed about their mean in a Poisson distribution instead of a Gaussian distribution. One advantage of the Poisson distribution is that the standard deviation is automatically determined

$$\sigma = \sqrt{y} \tag{20}$$

We make the following two assumptions to simplify the calculation:

1. The shapes of the individual Poisson distributions governing the fluctuations in the observed y_i are nearly Gaussian. (χ^2 is the best goodness of fit for a Gaussian distribution.)

2. Within the errors of the experiment, the uncertainties σ_i in the observations y_i my be approximated by $\sigma_i^2 \approx y_i$.

With the approximations, we have

$$a = \frac{1}{\Delta}\begin{vmatrix} N & \sum \frac{x_i}{yi} \\ \sum x_i & \sum \frac{x_i^2}{yi} \end{vmatrix} = \frac{1}{\Delta}\left(N\sum \frac{x_i^2}{y_i} - \sum x_i \sum \frac{x_i}{y_i}\right) \tag{21}$$

$$b = \frac{1}{\Delta}\begin{vmatrix} \sum \frac{1}{y_i} & N \\ \sum \frac{x_i}{y_i} & \sum x_i \end{vmatrix} = \frac{1}{\Delta}\left(\sum x_i \sum \frac{1}{y_i} - N\sum \frac{x_i}{y_i}\right) \tag{22}$$

$$\Delta = \begin{vmatrix} \sum \frac{1}{y_i} & \sum \frac{x_i}{y_i} \\ \sum \frac{x_i}{y_i} & \sum \frac{x_i^2}{y_i} \end{vmatrix} = \sum \frac{1}{y_i} \sum \frac{x_i^2}{y_i} - \left(\sum \frac{x_i}{y_i}\right)^2 \tag{23}$$

"Data Reduction and Error Analysis for the Physical Sciences", P.R. Bevington, McGraw-Hill

PHYS 234: Computational Physics

Lecture 21: Wednesday, 25 March 2015

1 Evaluation of continued functions

Can be powerful way of evaluating functions.

$$f(x) = b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cfrac{a_4}{b_4 + \cfrac{a_5}{b_5 + \cdots}}}}} \tag{1}$$

Compact notation

$$f(x) = b_0 + \frac{a_1}{b_1+}\ \frac{a_2}{b_2+}\ \frac{a_3}{b_3+}\ \frac{a_4}{b_4+}\ \frac{a_5}{b_5+}\ \cdots \tag{2}$$

where a, b can be functions of x (usually constant times x or times x^2)
Example:

$$\tan x = \frac{x}{1-}\ \frac{x^2}{3-}\ \frac{x^2}{5-}\ \frac{x^2}{7-}\ \cdots \tag{3}$$

Continued fractions frequently converge more rapidly than power series expansions.

How do you tell how far to go when evaluating a continued fraction? Need to evaluate right to left. Algorithm for left to right evaluation.

$$f_n = \frac{A_n}{B_n} \quad A_{-1} \equiv 1, \quad B_{-1} \equiv 0, \quad A_0 \equiv b_0, \quad B_0 \equiv 1 \tag{4}$$

$$A_j = b_j A_{j-1} + a_j A_{j-2} \quad B_j = b_j B_{j-1} + a_j B_{j-2} \quad j = 1, 2, \ldots, n \tag{5}$$

Evaluate until f_j and f_{j-1} are close enough.

"Numerical Recipes: The Art of Scientific Computing", W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Cambridge University Press

PHYS 234: Computational Physics

Lecture 22: Wednesday, 1 April and Wednesday, 8 April 2015

1 Partial differential equations

We now consider the numerical solution, $u = u(x, y)$, of partial differential equations of the general form

$$a(x,y)\frac{\partial^2 u}{\partial x^2} + b(x,y)\frac{\partial^2 u}{\partial x \partial y} + c(x,y)\frac{\partial^2 u}{\partial y^2} = f(x, y, u, \partial u/\partial x, \partial u/\partial y)\,. \tag{1}$$

We discuss the methods of solution according to the classification as either elliptical, parabolic, or hyperbolic.

1.1 Elliptical partial differential equations

If the coefficients in Eq. (1) satisfy $b^2 - 4ac < 0$, the equation is an elliptical PDE. These types of equations result from equilibrium-type boundary-value problems. Typical examples are Laplace's equation and Poisson's equation.

A closed solution domain is characteristic of elliptical partial differential equations. The function $u(x, y)$ must satisfy both the differential equation on the closed domain and the boundary conditions on the close boundary of the domain. We will illustrate the solution method using Laplace's equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0\,. \tag{2}$$

Let us superimpose a two-dimensional rectangular grid with spacing Δx by Δy upon the domain; indexed by j and i, respectively. The numerical solution will consist of determining the u's at the finitely spaced grid points. We can write the central difference approximate for the second partial derivatives of u's with respect to distance, in the x and y directions, as:

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\Delta x)^2}\,, \tag{3}$$

$$\frac{\partial^2 u}{\partial y^2} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta y)^2}\,. \tag{4}$$

Plugging these two differentials into Laplace's equations gives

$$\frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\Delta x)^2} + \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta y)^2} = 0\,. \tag{5}$$

If we use a square grid so that $\Delta x = \Delta y$, we obtain

$$u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j} - 4u_{i,j} = 0\,. \tag{6}$$

This is referred to as the Laplacian difference equation. This equation must hold at every interior grid point in the domain. We can also write this equations as

$$u_{i,j} = \frac{1}{4}(u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j})\,. \tag{7}$$

and notice the value of u at a point (i, j) is the average of the values of u of all its nearest neighbours.

For the boundary conditions, either the values are known at the boundaries or the derivatives are zero at the boundaries. For insulated boundaries (no heat flow from boundary)

$$\begin{aligned}
u_{i+1,j} + 2u_{i,j+1} + u_{i-1,j} - 4u_{i,j} &= 0, \text{ left} \\
u_{i+1,j} + 2u_{i,j-1} + u_{i-1,j} - 4u_{i,j} &= 0, \text{ right} \\
u_{i,j-1} + 2u_{i-1,j} + u_{i,j+1} - 4u_{i,j} &= 0, \text{ upper} \\
u_{i,j-1} + 2u_{i+1,j} + u_{i,j+1} - 4u_{i,j} &= 0, \text{ lower}
\end{aligned}$$

These give four equations at the boundaries. The problem has been reduced to obtaining solving a simultaneous set of linear algebraic equations in the unknown u's at the grid-points. The total number of equations depends on the number of grid points.

1.1.1 Method of iteration

Since the boundary values of the desired function are assumed to be known, we denote them by a's. The values of the required function at the interior of the grid are unknown, but in

order to start the iteration process we assign initial values on the basis of the best estimate available.

We start the iteration process by computing an improved value of u_1 by using the difference equation, the new or improved value of u_1 being denoted by u_1'. Then we proceed to improve u_2 in the same manner, and so on with all the other interior grid points. The traversal proceeds over the grid in order in which the points are numbered. Thus we have

$$\begin{aligned}
u_1' &= \frac{1}{4}(u_2 + a_2 + a_{24} + u_7) \\
u_2' &= \frac{1}{4}(u_3 + a_3 + u_1' + u_8) \\
\cdots &\quad \cdots \quad \cdots\cdots\cdots\cdots\cdots\cdots \\
u_7' &= \frac{1}{4}(u_8 + u_1' + a_{23} + u_{13}) \\
u_8' &= \frac{1}{4}(u_9 + u_2' + u_7' + u_{14}) \\
\cdots &\quad \cdots \quad \cdots\cdots\cdots\cdots\cdots\cdots \\
u_{24}' &= \frac{1}{4}(a_{12} + u_{18}' + u_{23}' + a_{14})
\end{aligned} \tag{8}$$

In approximating the first u's the initial values of the other u's are used. However, as soon as an approximate u at a grid point is calculated, this calculated value supersedes the

estimated valued and is used as the u at the point until it is, in term superseded by a new calculated value. Thus the latest calculated values of the u's we are seeking are always used in calculating newer and better values. The applicable equations are used at points on the domain which are selected in some systematic manner, usually either by rows or by columns.

The process outlined above is repeated as long as it improves the u's. For the second traverse we would start with

$$u_1'' = \frac{1}{4}(u_2' + a_2 + a_{24} + u_7')\,, \tag{9}$$

etc.

In solving difference equations by the iteration method, it might be beneficial to start with a course grid (large value of h). Then when iteration gives no further improvement in the u's, the whole process could be repeated with a finer grid (smaller value of h) and the iteration continued until the u's do not change to within the desired tolerance.

1.2 Parabolic partial differential equations

If the coefficients in Eq. (1) satisfy $b^2 - 4ac = 0$, the equation is a parabolic PDE. These types of equations result from propagation-type boundary-value problems. The solution advances indefinitely from known initial values, and thus the solution domain is open-ended. A typical example is the heat-flow equation in one dimension, x,

$$\frac{\partial u}{\partial t} = \frac{k}{c\rho}\frac{\partial^2 u}{\partial x^2}\,, \tag{10}$$

where u is the temperature, k is the thermal conductivity, and c is the specific heat capacity and ρ is the mass density of the material. For our purposes, $k/(c\rho)$ is just a constant.

The solution $u(x,t)$ propagates with time in the space-time plane. If we consider a space-time grid $((t,x) \to (i,j))$, a solution will consists of determining the temperature u at each grid point, and also satisfy an initial condition and boundary condition.

This time we put the equation in finite-difference form by substituting a forward-difference approximation for the first partial derivative:

$$\frac{\partial u}{\partial t} = \frac{u_{i+1,j} - u_{i,j}}{\Delta t} \tag{11}$$

and a central-difference approximation for the second partial derivative:

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\Delta x)^2}\,. \tag{12}$$

Substituting these difference equations into the heat equations gives

$$u_{i+1,j} = u_{i,j} + \frac{k}{c\rho}\frac{\Delta t}{(\Delta x)^2}(u_{i,j+1} - 2u_{i,j} + u_{i,j-1})\,. \tag{13}$$

We can obtain the temperature at a particular grid point with coordinates x and $(t+\Delta t)$ in terms of the temperatures at adjacent grid points $(x - \Delta x)$, x and $(x + \Delta x)$ at time t. Known initial temperatures and boundary temperatures provide the values necessary to start the calculations which then proceed row by row, and the end points of each row satisfying the given boundary conditions, until some final temperature state (with time) is approximately satisfied as the solution approaches a steady state. This row-by-row progression with time, which continues indefinitely, illustrates the open-ended nature of the solution domain of a parabolic-type partial differential equation.

The stability of the solution is important. It can be shown that the solution will be stable and non-oscillatory for boundary conditions that remain constant with time if

$$\frac{k}{c\rho}\frac{\Delta t}{(\Delta x)^2} \leq 0.25\,. \tag{14}$$

The solution will be stable if

$$\frac{k}{c\rho}\frac{\Delta t}{(\Delta x)^2} \leq 0.50\,. \tag{15}$$

1.3 Hyperbolic partial differential equations

If the coefficients in Eq. (1) satisfy $b^2 - 4ac > 0$, the equation is a hyperbolic PDE. These types of equations result from propagation-type boundary-value problems. The solution advances indefinitely from known initial values, and thus the solution domain is open-ended. A typical example is the one-dimensional, x, wave equation:

$$\frac{\partial^2 u}{\partial t^2} = a^2\frac{\partial^2 u}{\partial x^2}\,. \tag{16}$$

A physics example is the wave equation for a vibrating string

$$\frac{\partial^2 y}{\partial t^2} = \frac{T}{\mu}\frac{\partial^2 y}{\partial x^2}\,, \tag{17}$$

where y is the vertical displacement of the string, T is the tension of the string, and μ is the linear density of the string.

The solution exists in a general space-time plane. A solution procedure begins by superimposing a rectangular grid over the solution domain. The value of the dependent variable y is then determined at all grid points over the desired time interval. The numerical procedure itself begins with the use of the known initial-condition values and proceeds from there in a row-by-row progression with time, always satisfying the specified bounary conditions as the solution progresses.

The differential equation may be expressed algebraically by using the central-difference approximations for the two second partial derivatives:

$$\frac{\partial^2 y}{\partial t^2} = \frac{y_{i+1,j} - 2y_{i,j} + y_{i-1,j}}{(\Delta t)^2} \tag{18}$$

and

$$\frac{\partial^2 y}{\partial x^2} = \frac{y_{i,j+1} - 2y_{i,j} + y_{i,j-1}}{(\Delta x)^2} \,. \tag{19}$$

Substituting the two difference equations into the wave equation gives

$$\frac{y_{i+1,j} - 2y_{i,j} + y_{i-1,j}}{(\Delta t)^2} = \frac{T}{\mu}\left(\frac{y_{i,j+1} - 2y_{i,j} + y_{i,j-1}}{(\Delta x)^2}\right) \tag{20}$$

or

$$y_{i+1,j} = 2y_{i,j} - y_{i-1,j} + C\left(y_{i,j+1} - 2y_{i,j} + y_{i,j-1}\right) \,, \tag{21}$$

where $C = (T/\mu)(\Delta t)^2/(\Delta x)^2$. The value of C is important in considering the numerical solution of a partial differential equation. For example, $C > 1$ is unstable, $C \leq 1$ is stable, and $C = 1$ corresponds to a theoretically correct solution. The accuracy decreases as the value of C decreases further from $C = 1$. Try picking Δx and Δt such that $C = 1$.

"Applied Numerical Methods for Digital Computation With FORTRAM and CSMP", M.L. James, G.M. Smith, J.C. Wolford, IEP - A Dun-Donnelley Publisher
"Numerical Mathematical Analysis", J.B. Scarborough, The Johns Hopkins Press

PHYS 234: Computational Physics

Lecture 23: Wednesday, 8 April 2015

1 Boundary value problems of ordinary differential equations

In boundary-value problems, conditions are specified at both ends of the interval over which the integration is to be performed. Since a minimum of two boundary conditions are specified, we only consider differential equations of second order, or higher.

1.1 Shooting method

Both linear and nonlinear differential equations of second order, or higher, can be solved using a trial and error method, often called the shooting method. The technique uses the initial value method, discussed earlier, but replaces the known initial condition in that case with an approximation or guess. For example, consider the following general second-order differential equation:

$$y'' = f(x, y, y') \tag{1}$$

with the boundary conditions

$$x = x_1 \Big|_{y'=?}^{y=y_1} \qquad x = x_N \Big|^{y=y_N} \ . \tag{2}$$

An estimate is made for $y'(x_1)$ and a trial solution is calculated using usual initial value methods. After the trial solution is obtained, the known boundary value y_N at the end of the integration interval is compared with the corresponding value obtained by the trial solution. If the trail solution value does not agree with the known boundary value, a new initial condition is approximated and another trial solution calculated. The procedure is continued until a trail solution is obtained which satisfies the known final boundary value to the desired tolerance.

This trial and error approach is usually only applied to obtain solutions in which just one initial value is unknown. If it is necessary to approximate several different initial values, the number of trails could become large.

1.2 System of equations method

As usual, the differential equation to be solved is first expressed as an algebraic difference equation. A grid is imposed on the region in which the integration is to be performed. Difference equations are used to approximate the given differential equation which must be

satisfied at each of the points on the grid. If N grid points are used, we obtain a set of $N-2$ algebraic equations which can be solved simultaneously to obtain the desired solution.

This method is usual used for solving linear differential equations only. Nonlinear differential equations will result in nonlinear algebraic equations which are more difficult to solve as a system of simultaneous equations.

"Applied Numerical Methods for Digital Computation With FORTRAM and CSMP", M.L. James, G.M. Smith, J.C. Wolford, IEP - A Dun-Donnelley Publisher