

PHY 410-505 Computational Physics I

Chapter 1: A First Numerical Problem

Lecture 5

Friday September 5, 2008

Lecture Outline

Improved Radioactive Decay Programs	3
Pseudocode	3
C++ and Mathematica Codes	4
Top level code	4
Objects (data structures) in C++	5
Global variables	6
Function Declarations	7
Utility Functions	8
Function to get simulation parameters	12
Do the Computation	14
Plot the Results	17

Improved Radioactive Decay Programs

Pseudocode

Top Level: Broad outline of computational goals

- Repeat
 - Get physical constants n_0 , τ and t_{\max}
 - Do computation for various time steps Δt and store results
 - Plot results to compare with theory

Until user says done

Do computation: for various time steps Δt and store results

- Repeat
 - Get algorithmic constant Δt
 - Initialize variables n , t , data structure to store results
 - Loop until $t \geq t_{\max}$
 - Take Euler step
 - Store t and $n(t)$

Until user says done

C++ and Mathematica Codes

These concepts are illustrated in the codes decay2.cpp and decay2.m

Top level code

C++ code: implemented in the main function

```
int main() {
    // loop to perform several computations
    do {
        getConstants();
        doComputation();
        plotResults();
    } while (getYesNo("Another computation?") == true);
}
```

- Each top level pseudocode instruction implemented by C++ function

Mathematica code: implemented in loop following all definitions

```
While[ True,
    getConstants[];
    doComputation[];
```

```
    plotResults[];  
    If[getYesNo["Another computation?"], Continue[], Break[]]  
]
```

- Note use of While, True, If, Continue, Break

Objects (data structures) in C++

C++ code: Define convenient data structures

```
struct DataPoint {  
    double t;           // time  
    double n;           // N(t)  
};  
  
struct DataSet {  
    double dt;           // time step Delta t  
    vector<DataPoint> points; // vector of data points  
};
```

- A struct defines an object whose members are public by default
- Use the C++ Standard Template Library (STL) vector object

Mathematica: has numerous built-in data structures, including lists which can be used to store tables of data

Global variables

These are variables visible everywhere in the program following their declarations. Global variables are useful for sharing information between program parts.

C++ code: parameters and results of a computation

```
double n0 = 1000;           // number of nuclei at time t = 0
double tau = 1;             // mean lifetime
double tMax = 5;           // time to stop simulation
double dt = 0.2;           // time step Delta t for Euler algorithm

vector<DataSet> results;    // all results of a computation
```

- In C++ global variables are default initialized
 - int's and double's to 0
 - std::vector's to an empty vector

Mathematica code: parameters and results of a computation

```
n0 = 1000          (* number of nuclei at time t = 0      *)
tau = 1           (* mean lifetime                               *)
tMax = 5          (* time to stop simulation                       *)
dt = 0.2          (* time step Delta t for Euler algorithm        *)
dataSets = { }    (* list of computational results                *)
dtValues = { }    (* list of dt values in dataSets                *)
```

- uninitialized variables are treated as symbols
- { } represents an empty list

Function Declarations

C++ code: Functions must be declared OR defined before they can be used

```
void getConstants();          // user inputs N(0), tau, t_max
void doComputation();        // solve equation for several values of dt
void plotResults();          // use Gnuplot to plot all results

double getDouble(string prompt); // utility function to enter real number
```

```
bool getYesNo(string prompt); // utility function to get Yes/No response
string convert(int i);       // converts an integer to a string
```

- the *signature* of a function consists of its
 1. return value type
 2. name
 3. sequence of argument types
- argument names are optional
- declarations also useful in commenting your program

Mathematica: declarations are not required

- undefined functions are treated as symbols
- properties of functions can be changed dynamically, unlike C++

Utility Functions

Define some simple functions to help with parameter input, checking for input errors

C++ code: prompt user to input a real number or say Yes/No

```
double getDouble(string prompt) {
    bool done = false;
    double value;
    while (!done) {
        cout << prompt << ": " << flush;
        cin >> value;
        if (cin.fail()) {           // conversion to double has failed
            cin.clear();           // reset cin
            string badResponse;
            cin >> badResponse;    // eat the illegal entry string
            cout << "Bad response ... ";
        } else done = true;
    }
    return value;
}
```

```
bool getYesNo(string prompt) {
    bool done = false, yes;
    while (!done) {
        cout << prompt << " Enter Y[es] or N[o]: " << flush;
        string response;
        cin >> response;
        switch (response[0]) {
            case 'y':
```

```
        case 'Y':
            yes = done = true;
            break;
        case 'n':
        case 'N':
            yes = false;
            done = true;
            break;
        default:
            cout << "Bad response ... ";
            break;
    }
}
return yes;
}
```

- note examples of while and switch structures

Mathematica code: similar to C++ versions

```
getYesNo[prompt_] := (* function to get Yes/No response *)
Module[ { done = False, response, yes },
    While[ !done,
```

```

response = InputString[prompt " Enter Y[es] or N[o]:"];
Switch[ StringTake[ToUpperCase[response], 1],
  "Y", done = yes = True,
  "N", done = True; yes = False,
  _ , done = False; Beep[]
]
];
yes (* value returned by function *)
]

getDouble[prompt_] := (* function to get a real number *)
Module[ { d },
  While[ True,
    d = Input[prompt];
    If[ NumberQ[d],
      Break[],
      Beep[]
    ]
  ];
d (* value returned by function *)
]

```

- note underscore in function argument
- note use of Module to define the function body

- first argument of Module is a list of local variables, which can be initialized
- second argument of Module is a sequence of expressions separated by semicolons
- the last expression is returned as the value of the function
- note use of While and Switch structures

Function to get simulation parameters

C++ code: uses the getYesNo and getDouble functions

```
void getConstants() {  
  
    // print old values of physical constants which are global variables  
    cout << endl  
        << "Simulation of Radioactive Decay\n"  
        << "-----\n"  
        << "Current values of physical constants: \n"  
        << "N(0) = " << n0 << ", tau = " << tau  
        << ", t_max = " << tMax << endl;  
  
    if (getYesNo("Change these values?") == false)
```

```

    return;

    n0 = getDouble("Enter new value of N(0)");
    tau = getDouble("Enter new value of tau");
    tMax = getDouble("Enter new value of t_max");
}

```

- function has no arguments and void (no) return value
- it modifies on the global variables

Mathematica code: uses the `getYesNo` and `getDouble` functions

```

getConstants[_:Null] :=          (* user inputs N(0), tau, t_max          *)
Module[ { },
  Print["Simulation of Radioactive Decay"];
  Print["-----"];
  Print["Current values of physical constants:"];
  Print["N(0) = ", n0, " tau = ", tau, " t_max = ", tMax];
  If[ getYesNo["Change these values?"],
    n0 = getDouble["Enter new value of N(0): "];
    tau = getDouble["Enter new value of tau: "];
    tMax = getDouble["Enter new value of t_max: "]
  ]
]

```

- underscore in function argument represents any value
- :Null in function argument says that if no value is entered, then the default argument value is Null (something like void in C++)
- we could also define getConstants as a global symbol, without the []

Do the Computation

for given $N(0)$, τ and t_{\max} , and several valued of Δt

C++ code: this does the real work of the calculation

```
void doComputation() {

    results.clear();        // empty the results vector

    // loop to generate data for several values of time step dt
    do {
        // get new time step
        cout << "Current value of dt = " << dt << endl;
        dt = getDouble("Enter new value of dt");

        // initialize computational variables and data set
        double t = 0, n = n0;
```

```
    DataSet set;                // new empty data set
    set.dt = dt;
    DataPoint p;                // new data point
    p.t = t;
    p.n = n0;
    set.points.push_back(p);    // add the point to the set

    int iterations = int(tMax/dt);
    for (int i = 0; i < iterations; i++) {

        // one time step using Euler's algorithm
        p.n = n -= n / tau * dt;
        p.t = t += dt;

        set.points.push_back(p); // add the new point to the set
    }

    results.push_back(set);     // add the new set to results vector
} while (getYesNo("Another time step?") == true);
}
```

- note use of member (.) operator to access members of objects
- `results.clear()` calls the member function `clear` of the STL vector

object

- note use of `push_back()` member function of vector to add a new element at the end of a vector

Mathematica code: similar to C++ version

```
doComputation[_:Null] :=      (* solve equation for several dt values *)
  Module[ { t, n, set, iterations, i },
    dataSets = { };
    dtValues = { };
    While[ True,
      dt = getDouble["Old dt = " <> ToString[dt] <> " enter new dt: "];
      t = 0;
      n = n0;
      set = {{t, n}};
      iterations = IntegerPart[tMax / dt];

      For[ i = 0, i < tMax/dt, i++,
        n -= n / tau * dt;
        t += dt;
        AppendTo[set, {t, n}]
      ];
    ];
```

```

AppendTo[dataSets, set];
AppendTo[dtValues, dt];

    If[getYesNo["Another time step?"], Continue[], Break[]]
]
]

```

- lists are used to store generated data
- the AppendTo function adds an element to the end of list

Plot the Results

C++ code: calls Gnuplot interactively

```

void plotResults() {

    // use Gnuplot to plot data sets stored in global variable results
    // write a Gnuplot script file, setting title, etc.
    ofstream scriptFile("script");
    scriptFile << "set title \"N(0) = " << n0 << ", tau = " << tau << "\"\n"
        << "set xlabel \"Time t (sec)\"\n"
        << "set ylabel \"Number of nuclei N(t)\"\n"
        << "f(x)=" << n0 << "*exp(-x/" << tau << ")\n"
        << "plot f(x) title \"Exact\"";
}

```

```
// write the data sets in individual files
for (int i = 0; i < results.size(); i++) {

    // create an individual file name
    string dataFileName("data_");
    dataFileName += convert(i);

    // open the file and write the data
    ofstream dataFile(dataFileName.c_str());
    for (int j = 0; j < results[i].points.size(); j++) {
        dataFile << results[i].points[j].t << '\t'
                << results[i].points[j].n << '\n';
    }
    dataFile.close();

    // add plotting information for this set to the script file
    scriptFile << ", \"" << dataFileName << "\" "
                << " title \"dt = \" << results[i].dt << "\"";
}

scriptFile << "\npause mouse\n";
scriptFile.close();

// call Gnuplot to execute the script file
```

```

string gnuplot("\"C:\\Program Files\\gnuplot\\bin\\pgnuplot\");
string command = gnuplot + " script";
system(command.c_str());

}

```

- data sets are stored in separate files
- a script file is created with Gnuplot commands
- the system function of C++ executes its argument just as if you had typed it in a terminal window

Mathematica code:

```

plotResults[_:Null] := (* plot all results on a single graph *)
Module[ { t, i, dt },
  exact = Plot[n0 Exp[-t/tau], {t, 0, tMax}, DisplayFunction->Identity];
  showCommand = "Show[exact, ";
  plots = { };

  For[i = 1, i <= Length[dataSets], i++,
    plot = ListPlot[dataSets[[i]], DisplayFunction->Identity];
    AppendTo[plots, plot];
    showCommand = showCommand <> "plots[[" <> ToString[i] <> "]], "
  ];

```

```
        showCommand = showCommand <> "DisplayFunction->$DisplayFunction]";  
        ToExpression[showCommand]  
    ]
```

- store each plot as the element of a list
- create a string with a Show command
- convert the string to an expression and evaluate it using ToExpression
- a Mathematica program can construct and execute program code on the fly!!