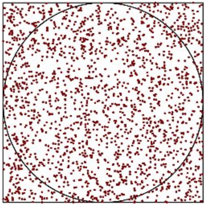
Outline

Outline

Let's Estimate π with MC sampling

Write a program to estimate the area C of the unit circle using MC sampling.

$$C = \iint_{x^2+y^2\leq 1} dxdy$$



- Let's consider a quarter of the area $0 \leq x \leq 1, 0 \leq y \leq 1$
- extract N points in there (`N`)
- count how many of them fall in (`inside`)
- 4 `inside` / `N` gives an estimate of π

Let's Estimate π with MC sampling

hints ...

- ask user for **N**
- Repeat the following steps **N** times:
  1. assign x and y random numbers in the range $[0, 1)$
  2. If $(x^2 + y^2 \leq 1)$, increment *inside*
- print your estimate of π

- Try many different values of **N** and check MC error
- check range values for variable types to handle **N**
- Use `rand()` and `RAND_MAX` from `stdlib.h`

```
const double rand_norm = 1.0/RAND_MAX; // avoid integer division
  ...
  x = rand_norm * rand();
```

Outline

Finding Roots With Bisection

Write a program that implements root finding with bisection and apply it to a known function (E.g. one from `math.h`).

- Bisection method works if we are able to confine a root of $f(x)$ in an interval between a and b, so that $f(a)f(b) < 0$.
- Bisection follows an iterative search:
  1. find the middle point c of a, b
  2. evaluate $p = f(a)f(c)$
  3. if $p = 0$, you are really lucky!
     1. c is the root
  4. if $p > 0$, root is in the interval c, b
     1. set $a = c$
  5. if $p < 0$, root is in the interval a, c
     1. set $b = c$
  6. repeat from 1 until $|b - a| < \epsilon$, where ϵ is a threshold

Finding Roots With Bisection

Use the following elements:

- `while` and `if/else` controls
- `fabs()`

Remember to make your program robust:

- choose appropriate ϵ to reflect the precision of used C types
- handle errors and exit in a controlled way

Try it with:

- a known function from `math.h` in your main code
- an external library `libmysterious.a` you compile, providing an include file to use as follows: `double mysteriousf(double x);`

Outline

Outline

Compute Prime Numbers

Write a simple program that:

- asks the user for an integer number **N**
- finds and prints out all prime numbers up to **N**

A *prime number* is a natural number which has exactly *two distinct* natural number divisors: 1 and itself

program outline:

- get upper limit N from user
- for each number $2 < n < N$
    - check if an exact divisor $b < n$ of n exists
    - if no b is found, than n is prime

Compute Prime Numbers (II)

- Use the following elements:
  - `printf()` and `scanf()`
  - `for` construct
  - `while` construct on $b < n$ and
  - `if` construct on n
- Remember to make your program robust:
  - check for proper input from the user ($N < 0$??)
  - check type limits
  - handle errors and exit in a controlled way

Outline

Function Integration

Write a simple program that computes the integral from 0 to 1 of the function $f(x) = \frac{4}{(1+x^2)}$

Use the Riemann definition of an integral, that is

$$\int_a^b f(x)dx = \lim_{N\to\infty} \sum_{i=1}^{N} f(x_i)\Delta x \ , \ with\ \Delta x = \frac{b-a}{N}$$

Program outline:

- Split $[a, b]$ into N subintervals of Δx width
- compute the function $f(x)$ in the middle point x_i of each interval and multiply for Δx
- sum up all contributions
- print the result and find out if it is correct

Outline

Outline

1. Consolidate your C basics
2. More C Basics
3. Working with Arrays
   - Build An Histogram
   - Array Transformation
   - Smoothing
4. Arrays and Structures
5. Working with pointers
6. Working with Strings and File I/O

Let's Build An Histogram

- Is **`rand`** as uniform as they say? Let's test...
- Write a program that:
  - Generates random numbers in the range 0, 1
  - Builds an histogram and computes their average
- Use **`rand()`** and **`RAND_MAX`** from **`stdlib.h`**
- Initialize to 0 an array of *ninterv* **`int`**s that holds the histogram; then, at each iteration:
  - Generate a random number
  - Find out the bin it belongs to (i.e. its index in the array)
  - Increment the corresponding array element and accumulate a sum to compute the average

Outline

Array Transformation

Write a program that computes the difference between each element of an array and its successive element.

$$A[x_i] \rightarrow A[x_i] - A[x_{i+1}]$$

- start with an array A[20] initialized as **`A[i] = i`**
- assume periodic boundary conditions
- use `%` operator in indexing expressions to implement periodic boundary conditions

Outline

Array Smoothing

Write a program that takes an array and applies N times a moving average smoothing transformation.

A moving average smoothing is a substitution:

$$A[x_i] \rightarrow \frac{1}{2k+1} \sum_{j=i-k}^{i+k} A[x_j]$$

- assume periodic boundary conditions on data
- iterate for $N = 1, 2, 5, 10$ times
- check your program with $k = 1, 2, 16$

hints

- start with an array A[20] initialized as `A[i] = i`
- check results at each iteration printing smoothed array elements
- Use `%` operator in indexing expressions to implement periodic boundary conditions

Array Smoothing (hints)

- Write a function that makes a single smoothing pass
  - taking as arguments both the data array and the array to store smoothed values into

Array Smoothing (hints)

- Write a function that makes a single smoothing pass
  - taking as arguments both the data array and the array to store smoothed values into
- and iterate smoothing passes

Array Smoothing (hints)

- Write a function that makes a single smoothing pass
  - taking as arguments both the data array and the array to store smoothed values into
- and iterate smoothing passes
- print smoothing array at each iteration to check results

Array Smoothing (hints)

- Write a function that makes a single smoothing pass
    - taking as arguments both the data array and the array to store smoothed values into
- and iterate smoothing passes
- print smoothing array at each iteration to check results
- for periodic boundary conditions:
    - use a wider working array, duplicating leading and trailing data
    - or use `%` operator in indexing expressions to implement periodic boundary conditions

Outline

1 Consolidate your C basics

2 More C Basics

3 Working with Arrays

4 Arrays and Structures
- Matrices

5 Working with pointers

6 Working with Strings and File I/O

7 I/O In Action

SuperComputing Applications and Innovation

Outline

Working with Matrices

Write functions using VLA to compute:

- matrix-vector product
- matrix-matrix product

- collect your functions in the source file **`linear_algebra_vla.c`**
- initialize matrix **`A[i][j]=i*j, B[i][j]=i+j, V[i]=i`**
- start with small squared 3x3 matrix to check results
- write **`printMatrix()`** and **`printVector()`** functions to check results
- try with a non-square matrix input too

Outline

Outline

Using Pointers to Functions

Use **`qsort`**, and **`comparedouble`** (from Module 7) to sort an array of n **`double`** random numbers in the range 0, 10

But **`qsort`** can be more powerful!
Initialize an array of **`vec3d`** variables with random numbers, and sort them by their first component

Using Pointers to Functions II

Revise your previous exercises to take a generic function as input:

- rewrite a general bisection function
- rewrite a general Riemann integration function

Always check compiler warning for function signature and type mismatches

Outline

Working with Matrices (II)

Rewrite your linear_algebra functions for matrix-vector and matrix-matrix product without VLA, using pointers to **`double`**

- collect your functions in **`linear_algebra.c`**
- remember to cast function arguments appropriately
- check results against your previous version

SuperComputing Applications and Innovation

Outline

BLAS Library

The BLAS library (Basic Linear Algebra Subprograms) contains routines for basic vector and matrix operations.

- Quick Reference:
  `http://www.netlib.org/blas/index.html`
- BLAS are divided into 3 levels:
  - Level 1: vector-vector operations
  - Level 2: matrix-vector operations
  - Level 3: matrix-matrix operations
- widely used in scientific software
- Often provided as a part of architecture optimized Math Libraries:
  ACML(AMD), ESSL(IBM), GotoBLAS, MKL(Intel), Sun Performance Library, etc

BLAS C Interface

- originally written for Fortran77
- BLAS provides a standard C interface
  - ... but include file name is not!
- function names are all lowercase and of the form: **`cblas_xname(...)`**
  - *x* denotes the data type:
    s for **`float`**,
    d for **`double`**,
    c for **`float complex`**,
    z for **`double complex`**

An Example Involving Vectors

BLAS Level 1: $op: \quad y \leftarrow \alpha x + y$

`cblas_saxpy (n, α, x, incx, y, incy)`

- the name says it all!
- **`n`** is the size of vectors **`x`** and **`y`**
- α is the vector **`x`** multiplyer
- **`incx, incy`** are increments to select vector elements

An Example With Vectors And Matrices

BLAS Level 2: $op: \quad y \leftarrow \alpha A x + \beta y$

cblas_dgemv (CblasRowMajor, CblasNoTrans, m, n, α, A, lda, x, incx, β, y, incy)

- this is a general matrix vector multiply and add
- **CblasRowMajor** selects memory layout of data
  ***enum* CBLAS_ORDER {CblasRowMajor, CblasColMajor}**
- **CblasNoTrans** is used to transpose matrix
  ***enum* CBLAS_TRANSPOSE {CblasNoTrans, CblasTrans, CblasConjTrans}**
- **m, n** are dimensions of matrix A
- **lda** is the leading dimension of array A
- **incx, incy** are increments to select vector elements
- Matrices and vectors are passed as pointers (cast as appropriate!)

Level 2 BLAS In Action

Multiply a matrix A[20][10] by a vector x[10] and put results into vector y[10]

```
int i, m = 20, n = 10;
double A[m][n], x[n], y[n];
double alpha = 1.0, beta = 0.0;
int lda = n, incx = incy = 1;
double dx = 0.05;

for (i=1; i<=m; i++) {
  for (j=1; j<=n; j++) {
    A[i][j] = (double) i * j + 0.5;
  }
}
for (i=0; i<n; i++) {
  x[i] = cos((double) i*dx);
}

cblas_dgemv (CblasRowMajor, CblasNoTrans,
  m, n, alpha, (double *) A, lda,
  x, incx, beta, y, incy);
```

Selecting Elements

From matrix A[20][10], Let's extract a submatrix subA[8][5],
And let's multiply it by even elements of vector x[10]

```
int i, m = 20, n = 10;
int subm = 8, subn = 5;
double A[m][n], x[n], y[subm];
double alpha = 1.0, beta = 0.0;
int lda = n, incx = 2, incy = 1;
double dx = 0.05;

for (i=1; i<=m; i++) {
  for (j=1; j<=n; j++) {
    A[i][j] = (double) i * j + 0.5;
  }
}
for (i=0; i<n; i++) {
  x[i] = cos((double) i*dx);
}

cblas_dgemv (CblasRowMajor, CblasNoTrans,
  subm, subn, alpha, (double *) A, lda,
  x, incx, beta, y, incy);
```

Yet Another BLAS Example: Matrices

BLAS Level 3: $op: \quad C \leftarrow \alpha AB + \beta C$

`cblas_zhemm (CblasRight, CblasUpper,`
`m, n,` α**`, A, lda, B, ldb,`** β**`, C, ldc)`**

- this is an hermitian matrix matrix multiply and add
- **`CblasRight`** and **`CblasUpper`** select matrix representation in memory (half the elements is enough for hermitian ones)
- **`m, n`** are sizes of matrix **`A`**, **`B`**, **`C`**
- **`lda, ldb, ldc`** are leading dimensions of array **`A`**, **`B`**, and **`C`**
- Matrices and vectors are passed as pointers (cast as appropriate!)

Using BLAS library

Use BLAS library to compute matrix-matrix and matrix-vector product

The BLAS functions you need:

- DGEMV: Double precision GEneral Matrix-Vector product (BLAS lev2)
- DGEMM: Double precision GEneral Matrix-Matrix product (BLAS lev3)

- Use GSL (GNU Scientific Library) library **`libgslcblas.a`** in **`lib/`**
- include header file **`include/gsl_cblas.h`**
- do cast your arrays to proper BLAS function parameters

Timing Your Function

Use `clock()` function from `time.h` for timing your version of matrix-matrix product function against BLAS GEMM for square matrices of sizes 100, 200, 500, 1000

- `clock_t clock(void);` returns the processor clock time used since the beginning of the program
- divide the returned value by `CLOCKS_PER_SEC` to get the number of seconds
- adapt the following code to measure your functions

```
#include <time.h>
  clock_t start, stop;
  double t = 0.0;

  // Start timer
  start = clock();
  // call your function
  // Stop timer
  stop = clock();
  t = (double) (stop-start)/CLOCKS_PER_SEC;
```

Outline

1. Consolidate your C basics
2. More C Basics
3. Working with Arrays
4. Arrays and Structures
5. Working with pointers
6. **Working with Strings and File I/O**
   - Command Line Parsing
   - Parse ASCII file
7. I/O In Action

Outline

1. Consolidate your C basics
2. More C Basics
3. Working with Arrays
4. Arrays and Structures
5. Working with pointers
6. Working with Strings and File I/O
   - Command Line Parsing
   - Parse ASCII file
7. I/O In Action

Command Line Parsing

Write a program that parses command line and:

- accepts stand alone options (on/off switch)
- accepts option with a single argument (int, double and string)
- outputs a report of what it parsed

- use `int argc, char *argv[]` parameters of `main()`
- use `switch` control
- use `strto...()`, `strcmp()`

Outline

1. Consolidate your C basics
2. More C Basics
3. Working with Arrays
4. Arrays and Structures
5. Working with pointers
6. Working with Strings and File I/O
   - Command Line Parsing
   - Parse ASCII file
7. I/O In Action

Parsing a file

Write a function that accepts a string containing a file name as its argument, and parses the file, storing retrieved values into global variables; the file has the format:

- `keyword value`
- empty lines or starting with a # should be ignored

Use main program in `simpleparsermain.c` and the input file `param.dat` to test your function.
Recognized parameters:

- `nx` (int) number of points in the x direction;
- `ny` (int) number of points in the y direction;
- `tol` (double) some kind of tolerance or threshold;

Of course feel free to add more keys if you want! Hints:

- Use `fgets()` to retrieve input lines, and test its return value;
- Parse input lines with `sscanf()`;

Parsing a file II

After implementing and testing a very simple version:

- Find a way to check that values found in the file have the expected type and correct domain range (nx < 0 is nonsense)
- check for multiple parameter definitions
- Return an error message and exit if you find an unrecognized key;
- What if a line starts with white spaces? Did you already handle this?

Consider using `strtok()` or similar to implement more flexible and complicated parsing.

Parse a structured ASCII file

Write a program that reads protein coordinates from a .pdb file format

- .pdb file format files are wide used in bioinformatic
- coordinates begins with **ATOM** tag

```
ATOM 1 N PRO A 1 8.316 21.206 21.530 1.00 17.44 N ATOM 2 CA PRO A 1 7.608 20.729 20.33
1.00 17.44 C ATOM 3 C PRO A 1 8.487 20.707 19.092 1.00 17.44 C ATOM 4 O PRO A 1 9.466
21.457 19.005 1.00 17.44 O
```

Seek in your file

Use seek I/O functionality to collect elements from a binary file

- open the **`binaryfile.dat`** file from course package
- read elements using the following steps
  - 8 bytes 128 bytes bakward from end of file, set bookmark **`A`** here
  - 8 bytes 32 bytes from the beginnig, set bookmark **`B`** here
  - 4 bytes 32 bytes from bookmark **`A`**
  - 16 bytes 64 bytes from bookmark **`B`**
- convert read elements to chars and print them

Outline

1. Consolidate your C basics
2. More C Basics
3. Working with Arrays
4. Arrays and Structures
5. Working with pointers
6. Working with Strings and File I/O
7. I/O In Action
   - ASCII vs Binary

SuperComputing Applications and Innovation

Outline

Let's Make Comparisons

- Write a program that:
  - Initializes an array of 900000 `doubles` with random numbers in the range 0, 1
  - Writes the array 10 times to an ASCII file.
  - Uses `time` and `difftime` from `time.h` to time the writing operation
- Remember to print enough decimal digits, to recover exact binary form of your data
- Try writing array elements on one line with no white spaces in between
- Try writing one element per line
- How big is your output file?

Let's Make Comparisons

- Modify your program so that it writes to a binary file with `fwrite`
- Try different solutions:
  - Write one element at a time: pointer arithmetic will help;
  - Write the array in chunks of 1000 elements;
  - Write the whole array with a single call to `fwrite`.
  - In any case, check the value returned by `fwrite`.
- What about output file dimensions?
- And what about time?

Walking Around In Our File

- Our output file now contains 10 copies of our array
- Let's read the first element of each copy in the file
- Let's print it together with its position
    - Let's use `fseek` to reach the right position
    - And `ftell` to have the current position returned
    - Remember to `fopen` the file with all the necessary modes

Outline

1. Consolidate your C basics
2. More C Basics
3. Working with Arrays
4. Arrays and Structures
5. Working with pointers
6. Working with Strings and File I/O
7. I/O In Action
8. Working with Memory

Outline

Big Array Transformation

Rewrite your Array Transformation program so that the transformation is performed by a function which takes the array to be processed as an argument.

- use VLA array in your first version
- check you program with $size = 1000, 10000, 100000$
- does this work with automatic array declarations?
- after you checked, use **`malloc`** to dynamically allocate the array
- remember to use **`free`** on dynamically allocated variables

Array Transformation With Unknown Dimensions

Rewrite your array transformation program so that it reads input data from the file

- write a file containing a floating point number on each line
- let the first line contains the number of subsequent lines in the file (int)
- Once you read the first line, you can call `calloc` to allocate enough space to hold data
- Don't forget to check if `calloc` succeeded

Working with Matrices

Rewrite your linear algebra functions so to allocate your matrices with **`malloc`**.

- remove printing functions
- use BLAS library to check results
- cast your argument appropriately

Rights & Credits

Slides and examples were authored by:

- Michela Botti
- Federico Massaioli
- Luca Ferraro
- Stefano Tagliaventi