An Introduction to C Programming
Review of Exercises

Francesco Safai Tehrani (INFN Rome)
francesco.safaitehrani@roma1.infn.it

How I did it: compilation and the like

- gcc compiler: “pure C” (with GCC extensions)
- all programs compiled with
  - gcc -Wall -pedantic -o <prg_name> prg_name.c
    - -Wall = -W(arning) all
    - -pedantic = activates all checks for pure C conformance
- no command line argument management
  - I basically ignore all the command line processing
  - arguments are ‘passed’ as #define-s
- NO GLOBAL VARIABLES
  - unless they’re absolutely needed ;)
    - yup, I decide when they’re absolutely needed... YMMV :)

Some more observations

- The techniques I expect you to have mastered:
  - C program structure
    - structure and organize your code in functional units
  - Basic debugging
    - printf-s and the like
  - Basic algorithmic thinking
    - take an algorithm described in text and turn it into code
  - Basic memory management
    - variables, arrays and dynamic memory handling
- We'll review how you fared in these areas at the end of this presentation.
  - Yup, not too bad. No worries there.

Language, language, language...

- Yes: C != C++
  - even though they look (kinda) alike
- There was more than a bit of confusion between C and C++ (intentional and not)
  - This might be due to background (e.g. learning C++ but not ‘proper’ C)
- C++-isms that I’ve consistently found in the code:
  - C++ style comments: // comment (C: /* comment */)
  - C++ namespaces (“using namespace std;”)
  - C++ stream output (iostream, cout, endl, ...)
  - arrays with variable size
    - this is not C++, but ISO C99, accepted by gcc as an extension for ISO C90 and C++
    - C arrays must have a constant size
  - definition of variables throughout the code
    - C wants variable definition strictly at the beginning of a function body
- It’s fine to mix, as long as you know what you’re doing. Do you?

Terminology (from C to C++)

- A term I will be using often:
  - function interface (or simply interface) of signature
    - the set of arguments that a function accepts
    - int sum(int a, int b)
      - “int a, int b” is the function signature/interface
- A header file is a library’s interface
  - or maybe a collection of header files...
- Interfaces are going to be very important in C++
  - and in Object Oriented programming in general ...
- (much) more on this later

Ex.1 - Factorial

Rewrite both the iterative and recursive factorial functions using forward counting (from 1 to num).

Not much to say about this exercise. The factorial is a simple iterative algorithm that can easily be implemented iteratively or recursively.

This time the algorithm was turned around to use forward counting. This caused the recursive version to require a second argument, thus creating the unpleasant need to change the function interface, from:

```
factorial(n)
```

to

```
factorial(n, weird_parameter_that_does_not_make_much_sense_for_the_end_user)
```

Does it matter?
Is it logical?
So how do you solve the issue?

Principle of Least Surprise (a form of...)

```c
#include <stdio.h>

int recursive_forward_factorial(int num, int index) {
  if(index==num) return num;
  return index * recursive_forward_factorial(num, index+1);
}

inline int recursive_factorial(int num) {
  return recursive_forward_factorial(num, 1);
}

int iterative_factorial(int num) {
    int i, result=1;
    for (i=1; i<=num; i++) result *= i;
    return result;
}

int main (int argc, const char * argv[]) {
    printf("Iterative 6! :%d\n", iterative_factorial(6));
    printf("Recursive 6! :%d\n", recursive_factorial(6));
    return 0;
}
```

Utility function: a simple function used to hide the complexity and publish the expected interface.
Inline: a suggestion to the compiler that this function call could be replaced by its code.
The compiler is free to accept the suggestion or ignore it.

Ex.2 - Fibonacci Numbers

The formula to calculate the n-th Fibonacci number F_n is:

$F_n = F_n-1 + F_n-2$
$F_0 = 0$
$F_1 = 1$

Write a program containing two functions (one iterative and one recursive) which take n as argument and return F_n.

Another standard exercise: the calculation of the Fibonacci number is just a little bit more complex than the factorial, since you need to keep track of the last two results.

The recursive implementation of Fibonacci instead is an exercise in complexity (in the computing sense):

each function invocation is transformed into two, so it would look like the algorithmic complexity is 2^n. The real one is $\sim 1.6^n$. Hence the calculation of F49 takes 49 sums in the iterative implementation and $\sim 10^{10}$ in the recursive implementation.

A (virtual) demo should clarify this issue...

```c
#include <stdio.h>

#define FIBNUM 49 /* the highest Fn found on the Wikipedia page */

long recursive_fibonacci(int num) {
  if (num<=1) return num;
  return recursive_fibonacci(num-1)+recursive_fibonacci(num-2);
}

long iterative_fibonacci(int num) {
    int i;
    long f0, f1, tmp;
    if (num<=1) return num;

    f0 = 0;
    f1 = 1;
    for (i=2; i<=num; i++) {
      tmp = f0;
      f0 = f1;
      f1 = f0+tmp;
    }
    return f1;
}

int main (int argc, const char * argv[]) {
    printf("The %d Fibonacci number is (iteratively) :%ld\n", FIBNUM, iterative_fibonacci(FIBNUM));
    printf("The %d Fibonacci number is (recursively) :%ld\n", FIBNUM, recursive_fibonacci(FIBNUM));
    return 0;
}
```

The culprit: double recursion...

Now... can we still salvage it somehow?
Let's say we *HAVE* to do it recursively...

```c
#include <stdio.h>

#define MAXFNS 100
long fns[MAXFNS];

#define FIBNUM 49 /* the last Fn found on the Wikipedia page */

long recursive_fibonacci(long num) {
  if (num<=1) return num;
  if (fns[num] == -1) {
    fns[num] = recursive_fibonacci(num-1)+recursive_fibonacci(num-2);
  }
  return fns[num];
}

int main (int argc, const char * argv[]) {
    int idx;
    for(idx=0; idx<MAXFNS; idx++) fns[idx] = -1;
    printf("The %d Fibonacci number is (recursively) :%ld\n", FIBNUM, recursive_fibonacci(FIBNUM));
    return 0;
}
```

This technique is called memoization.

It implies storing the results of an expensive calculation for later use. It is different from a Look-Up Table (which we'll describe later) in that only actual calculation results are stored while a Look-Up Table is usually pre-calculated.

To memoize the temporary results we use a global variable
Global variables are evil, but in this case the use is justified (naturally, you might feel differently).

ex2-fibonacci-memo.c

```c
#include<stdio.h>

typedef struct fibpair {
    long val, prevval;
} fibpair;

fibpair recfib(long num) {
    float sum;
    fibpair tmp = { 1, 0 };
    if (num == 1) return tmp;
    if (num == 0) { tmp.val = 0; return tmp; }
    tmp = recfib( num-1 );
    sum = tmp.val + tmp.prevval;
    tmp.prevval = tmp.val;
    tmp.val = sum;
    return tmp;
}

long recursive_fib(long num) {
    fibpair tmp = recfib( num );
    return tmp.val;
}

int main() {
    int i;
    printf("The %2ith Fib. number is = %ld\n", 49, recursive_fib(49));
    return 0;
}
```

A different technique, suggested by Erkcan Özcan, uses a C struct to hold two values (the current and previous Fibonacci number) and returns them at every step of the iteration. Using a (slightly) more complex data structure we avoid the need of the double recursion while at the same time retaining the speed and memory efficiency of the iterative approach.

ex2-fibonacci-struct.c

Ex.2 - Fibonacci Numbers - demo

[Unix note: time logs the amount of time that a command takes to execute (man time)]

Now we execute ex2-fibonacci:

```
~/> time ./ex2-fibonacci
The 49 Fibonacci number is (iteratively) :7778742049  <-- this line appears immediately
The 49 Fibonacci number is (recursively) :7778742049  <-- and this one ~3mins later
./ex2-fibonacci  186.60s user 0.17s system 99% cpu 3:07.92 total
```

Let's try the memoization version:

```
~/> time ./ex2-fibonacci-memo
The 49 Fibonacci number is (iteratively) :7778742049
The 49 Fibonacci number is (recursively) :7778742049
./ex2-fibonacci2  0.00s user 0.00s system 5% cpu 0.037 total
```

And last but not least the struct-based version:

```
~/> time ./ex2-fibonacci-struct
The 49th Fib. number is = 7778741248
./ex2-fibonacci-oe  0.00s user 0.00s system 19% cpu 0.009 total
```

It can be easily seen from the output of time that the two 'smarter' approaches to the recursive calculation of the Fibonacci numbers are much more efficient both from the wall-clock standpoint and from the CPU standpoint.

Ex.3 - Unit Conversion Library

For this exercise you will implement a few unit conversion libraries. You can find the conversion factors and algorithms online.

Start with a library to convert centimeters to inches, meters to feet and vice versa, then add miles to kilometers. Add as many as you want.

Now create another library to convert weights, and implement the conversion between kilograms and pounds. Add as many as you want.

Now create a library to convert between different temperature scales, Celsius to Fahrenheit and vice versa, Celsius to Kelvin, Kelvin to Fahrenheit and so on.

Create a test program to use these libraries and print various conversions. Check that the result are correct.
Now, unless you've done some design in advance, you will find yourself with a lot of functions which do exactly the same thing (more or less).

Would it be possible to rewrite your conversion libraries to minimize code repetition, maybe by implementing some utility functions in a special dedicated library? (Utility functions are functions which solve a specific problem in a more general way).

Rewrite your libraries to maximize code reuse. Is it simpler now to add new conversions? Discuss your solution.

This is by and large the exercise you've been most creative about. What I had in mind was a lot simpler but most of you created rather complex structures...

```c
#include <stdio.h>
#include "weight.h"
#include "temperature.h"
#include "length.h"

int main (int argc, const char * argv[]) {
  printf("1 cm in inches: %7.3f\n", cm_to_in(1.));
  printf("1 inch in  cms: %7.3f\n", in_to_cm(1.));
  printf("1 pound in kgs: %7.3f\n", lb_to_kg(1.));
  printf("1 kg in pounds: %7.3f\n", kg_to_lb(1.));
  printf("60F in Celsius: %7.3f\n",   C_to_F(60));
  printf("60Celsius in F: %7.3f\n",   F_to_C(60));
  return 0;
}
```

ex3-conversion_library.c

```
#ifndef LENGTH_H
#define LENGTH_H

double cm_to_in(double);
double in_to_cm(double);

#endif
```

```
#include "length.h"

double cm_to_in(double cms) {
  return cms/2.54;
}

double in_to_cm(double ins) {
  return ins*2.54;
}
```

```
#ifndef TEMPERATURE_H
#define TEMPERATURE_H

double C_to_F(double);
double F_to_C(double);

#endif
```

```
#include "temperature.h"

double C_to_F(double cdeg) {
  return cdeg*9./5.+32.;
}

double F_to_C(double fdeg) {
  return (fdeg-32)*5./9.;
}
```

```
#ifndef WEIGHT_H
#define WEIGHT_H

double kg_to_lb(double);
double lb_to_kg(double);

#endif
```

```
#include "weight.h"

double kg_to_lb(double kgs) {
  return kgs/0.45359237;
}

double lb_to_kg(double lbs) {
  return lbs*0.45359237;
}
```

```c
#ifndef LENGTH_H
#define LENGTH_H

#include "utility.h"

double cm_to_in(double);
double in_to_cm(double);

#endif
```

```c
#include "length.h"

double cm_to_in(double cms) {
  return reverse_conversion(cms, 2.54, 0.);
}

double in_to_cm(double ins) {
  return direct_conversion(ins, 2.54, 0.);
}
```

```c
#ifndef TEMPERATURE_H
#define TEMPERATURE_H

#include "utility.h"

double C_to_F(double);
double F_to_C(double);

#endif
```

```c
#include "temperature.h"

double C_to_F(double cdeg) {
  return direct_conversion(cdeg, 9./5., 32.);
}

double F_to_C(double fdeg) {
  return reverse_conversion(fdeg, 9./5., 32.);
}
```

```c
#ifndef WEIGHT_H
#define WEIGHT_H

#include "utility.h"

double kg_to_lb(double);
double lb_to_kg(double);

#endif
```

```c
#include "weight.h"

double kg_to_lb(double kgs) {
  return reverse_conversion(kgs, 0.45359237, 0.);
}

double lb_to_kg(double lbs) {
  return direct_conversion(lbs, 0.45359237, 0.);
}
```

ex3 library [v2]

```c
#ifndef UTILITY_H
#define UTILITY_H

double direct_conversion(double, double, double);
double reverse_conversion(double, double, double);

#endif
```

```c
#include "utility.h"

double direct_conversion(double value, double m_factor, double a_factor) {
  return value*m_factor + a_factor;
}

double reverse_conversion(double value, double m_factor, double a_factor) {
  return (value-a_factor)/m_factor;
}
```

`utility.h/c`

Ex.4 - Crash the Stack

Write a program to crash the stack.
As a bonus point, add a counter to check the stack depth.

```
#include<stdio.h>

void crash_stack(int index) {
    printf("%d ... ", index);
    crash_stack(index+1);
}

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

Surprisingly this is one of the exercises you all managed to get right... ;)

Not much to say about it. This works as expected, layering new function calls one on top of the other until the stack is exhausted and the program crashes.

Ex.5 - Return multiple values

Write a function that accepts two positive numbers, and returns their sum, their difference and their mean value. Also make it so that the function returns something indicating an error if one of the arguments is negative.
Write a program to use this function and print its results.

This exercise had two possible kinds of solution:
- write a function with a longer signature that contains three dummy arguments passed by pointer that the function can then use to store the calculated values
- write a function that returns an array containing the results of the calculation

Both approaches were used, but most of you did not manage correctly (= as requested by the text of the exercise) the error condition.
In general printf-ing an error message is not really useful, especially if your function is used in a library by somebody else. It would be preferable for your functions to produce meaningful (=documented) error values.

Using the second approach, returning an error condition is much more complex, hence I favour (and implement) the first approach.

```c
#include <stdio.h>

int calc_data(int op1, int op2, int* sum, int* diff, float* mean) {
  if((op1<0) || (op2<0)) { return -1; }
  *sum = op1+op2;
  *diff = op1-op2;
  *mean = *sum / 2.0;
  return 0;
}

int main (int argc, const char * argv[]) {
    int op1 = 40;
    int op2 = 2;

    int sum, diff;
    float mean;

    if(calc_data(op1, op2, &sum, &diff, &mean) == -1) {
        printf("Whoops! One of the operands of calc_data is negative!\n");
    } else {
        printf("Calc data results: \n");
        printf("              %5d + %5d = %5d\n", op1, op2, sum);
        printf("              %5d - %5d = %5d\n", op1, op2, diff);
        printf("Mean value of %5d,  %5d = %8.2f\n", op1, op2, mean);
    }
    return 0;
}
```

Dummy arguments

Handling the error condition

The printf formatting works like this:
%[-c]n[.m]X
The - specifies that the field is left justified
The [c] is a padding character: %05d, 42 = 00042
The n is the field length in characters
The [.m] only for float/double fields, indicates the number of digits after the '.'
X is the format specifier [d = integers, f = floats/doubles, ...]

ex5-multiple_values.c

Ex.6 - Numerical integration

Write a program to calculate a numeric integral using the the composite trapezoidal rule (http://en.wikipedia.org/wiki/Trapezoidal_rule).
The program should define a function that accepts an array containing the values of the function to integrate and any other relevant parameter: float integrate(float values[], ...)

The main part of the program should fill the values array, with values calculated from the function to be integrated. Ideally this function should also be stored in a function (okay, the mathematical function to be integrated should be stored in a C function).

This logical separation allows you to write the integration code and reuse it as needed, while making it also possible to easily implement other integration algorithms and reuse the same mathematical functions.

You should be careful when defining the integration interval, the integration steps and all the relevant parameters. You might also want to define some utility function to map the integer indexes of the values array onto the integration step.
The math.h header contains a number of mathematical functions which might be useful.

Now... this produced some interesting results. It's a slightly more complex algorithm and caused some problems...

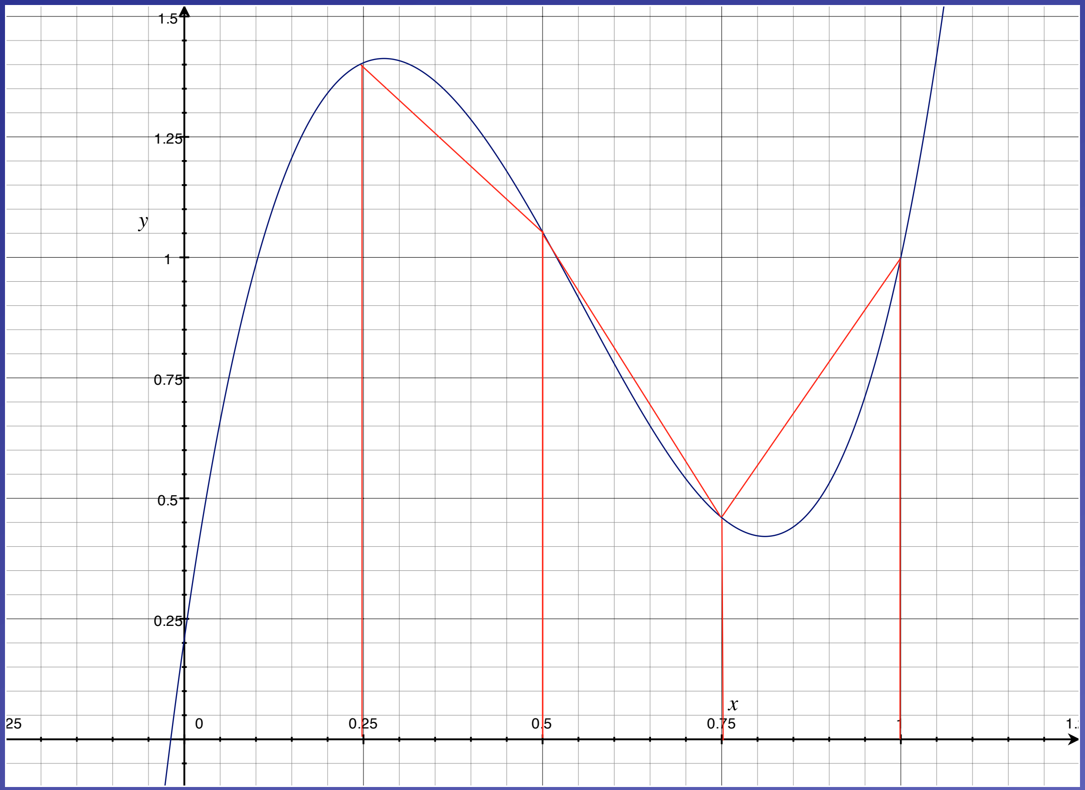
Ex.6 - Numerical integration (2)

Take f(x), an interval [a, b] and divide it in N subintervals of length step_len = (b-a)/N

$x_1 = a$
...
$x_i = a + i*step\_len$
...
$x_N = b$

Now the area of the integral can be approximated as:

$Area = \sum ½(f(x_i)+f(x_{i+1}))*step\_len$



A few observations from the code I've seen:
- it's best to analyze the algorithm and implement it faithfully (premature optimization is the root of all evil)
- it's also best to analyze the algorithm and implement it correctly

In this case there's a constant operation (multiplication by ½step_len) that can be factored out of the summation. Also a bit of simple manipulation shows that the formula can be rewritten as:

$$Area = [\ ½(f(a)+f(b)) + \sum f(x_i)\ ]*step\_len$$

```c
#include <stdio.h>
#include <math.h>

#define STEPS 1000

double function(double x) {
    return exp(x)*pow(x,2);
}

void calculate_function(double a, double b, int steps, double* values) {
  int idx;
  double step_len = (b-a)/steps;
  for(idx=0; idx<=steps; idx++) {
    values[idx] = function(a+idx*step_len);
  }
}

double integrate_function(double a, double b, int steps, double* values) {
  int idx;
  double step_len = (b-a)/steps;
  double result = 0;

  for(idx=1; idx<steps; idx++)
    result += values[idx];

  result = (result + (values[0]+values[steps])/2.0) * step_len;
  return result;
}

int main (int argc, const char * argv[]) {
    double a, b;
    double result;
    double values[STEPS+1];

    a = 1;
    b = 2;

    calculate_function(a, b, STEPS, values);
    result = integrate_function(a, b, STEPS, values);

    printf("Integration of e^x*x^2 between %7.2f and %7.2f yields %7.2f\n", a, b, result);
    return 0;
}
```

ex6-integration.c

Ex.7 - Endianness

Figure out the endianness of the computer you are using using a C program.

```
#include<stdio.h>

int main() {
    short a = 1;
    char* x = (char*) &a;
    if (x[0] == 1 ) { printf("Little endian\n"); }
    else            { printf("Big endian\n");    }
    return 0;
}
```

And this is the other exercises most of you managed to get right... ;)
[Maybe it's because you like to break things apart and peek inside? Perfect scientific mindset, if I may say so...]

I chose a short because it only contains two bytes so that:

```
0x1 in little endian would look like 01 00
0x1 in big endian would look like    00 01
```

If you read the material suggested for the exercise, you should also know a little bit more about how C represents integer numbers internally.

Ex.8 - Monte Carlo integration

Write a program to implement the 1D Monte Carlo integration.

The Monte Carlo integration is a numerical integration algorithm that uses random numbers.

The algorithm works as follows:
1. inscribe your function in a rectangle whose left and right sides are the same as the integration limits, and whose lower side lays on the x axis
2. generate a random point within this rectangular area
3. if the point is under the curve, increment a counter

Repeat 2. and 3. N times. With N large enough, the integral of the curve is
~= (counter/N)*rectangle_area

See: http://en.wikipedia.org/wiki/Monte_Carlo_integration and man rand for information about random number generation in C.

Monte Carlo methods are members of the family of random algorithms. Random algorithms use random number to tackle complex problems unapproachable by deterministic methods. MC algorithms are time bound: they end in a fixed amount of time, with a small error. Las Vegas algorithms are not time bound, but always find the correct answer (altough the algorithm not terminating does not give indication regarding the existence of such an answer).

http://en.wikipedia.org/wiki/Random_algorithm

The complexity of this exercise was similar to that of the integration one. In fact some code and logic could be reused.

Most of your solutions took two rather major simplification hypotheses:

1. monotonic function: f(a) and f(b) are absolute minimum/maximum for the function
2. f(x) > 0 for x∈[a,b]

This is not a problem, but it would been nice to have it explicitly noted in the code, so that I know you know what you're doing.

This is always true: you need to document what you are doing so that the end user of your code/library knows what (s)he's getting (always the Principle of Least Surprise).

Another detail that affects deeply the MC methods in general is the quality of the pseudo-random number generator (PRNG). I suggested the use of rand, even though it's well known to be a poor generator.

It would be very useful for those of you who don't know why rand is a poor PRNG to take a few minutes to read this page:
http://en.wikipedia.org/wiki/Pseudorandom_number_generator

My code looks for the function max and min values, and uses them like this:
if min < 0 the integration rectangle y-axis goes between [min, max]
otherwise it goes between [0, max]
That's equivalent to translating the x axis to min.
If max<0, the program returns 1 (which in Unix-speak usually indicates an error)

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#define STEPS 1000
#define SAMPLES 10000000

double function(double x) { return exp(x)*pow(x,2); }

void min_max(double a, double b, int steps, double* min, double* max) {
  int idx;
  double value;
  double step_len = (b-a)/steps;

  *min = function(a);
  *max = function(a);

  for(idx=0; idx<=steps; idx++) {
    value = function(a+idx*step_len);
    if(value > *max) *max = value;
    if(value < *min) *min = value;
  }
}

double mc_integrate_function(double a, double b, double min, double max, int samples) {
  int idx, counter = 0;
  double xr, yr;
  double lenH = b-a;
  double lenV = (max - min);

  srand(time(0));

  for(idx=0; idx<samples; idx++) {
    xr = a   + lenH * (rand()/(double)RAND_MAX);
    yr = min + lenV * (rand()/(double)RAND_MAX);

    if(function(xr) > yr) counter++;
  }
  return lenH*lenV*counter/samples;
}

int main (int argc, const char * argv[]) {
    double a, b, min, max, result;

    a = 1;
    b = 2;

    min_max(a, b, STEPS, &min, &max);
    if(min > 0) min = 0;
    if(max < 0) return 1;

    result = mc_integrate_function(a, b, min, max, SAMPLES);

    printf("Integration of e^x*x^2 between %7.2f and %7.2f yields %7.2f\n", a, b, result);
    return 0;
}
```

ex8-monte_carlo_integration.c

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#define STEPS 1000
#define SAMPLES 10000000

double function(double x) { return exp(x)*pow(x,2); }

void min_max(double a, double b, int steps, double* min, double* max) {
  int idx;
  double value;
  double step_len = (b-a)/steps;

  *min = function(a);
  *max = function(a);

  for(idx=0; idx<=steps; idx++) {
    value = function(a+idx*step_len);
    if(value > *max) *max = value;
    if(value < *min) *min = value;
  }
```

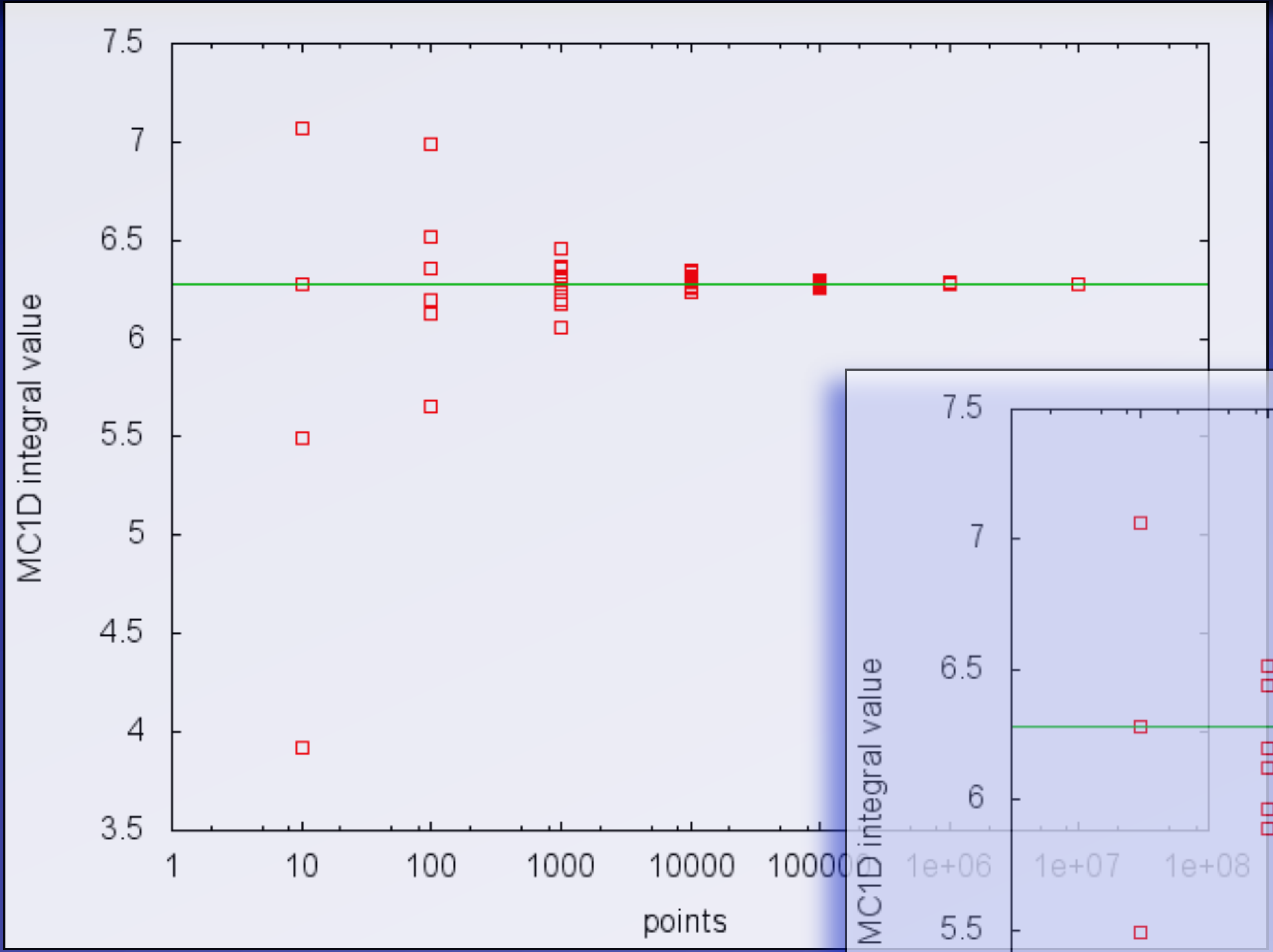
The min_max function uses again the trick of dummy arguments passed by pointers to return multiple values... ugly, but necessary.

This would need to be explicitly stated in your library documentation.

```c
double mc_integrate_function(double a, double b, double min, double max, int samples) {
  int idx, counter = 0;
  double xr, yr;
  double lenH = b-a;
  double lenV = (max - min);

  srand(time(0));

  for(idx=0; idx<samples; idx++) {
    xr = a   + lenH * (rand()/(double)RAND_MAX);
    yr = min + lenV * (rand()/(double)RAND_MAX);

    if(function(xr) > yr) counter++;
  }
  return lenH*lenV*counter/samples;
}

int main (int argc, const char * argv[]) {
    double a, b, min, max, result;

    a = 1;
    b = 2;

    min_max(a, b, STEPS, &min, &max);
    if(min > 0) min = 0;
    if(max < 0) return 1;

    result = mc_integrate_function(a, b, min, max, SAMPLES);

    printf("Integration of e^x*x^2 between %7.2f and %7.2f yields %7.2f\n", a, b, result);
    return 0;
}
```

The srand(time(0)) is a standard trick to reinitialize the RNG to a new seed. With rand once the seed is known the entire sequence is known.

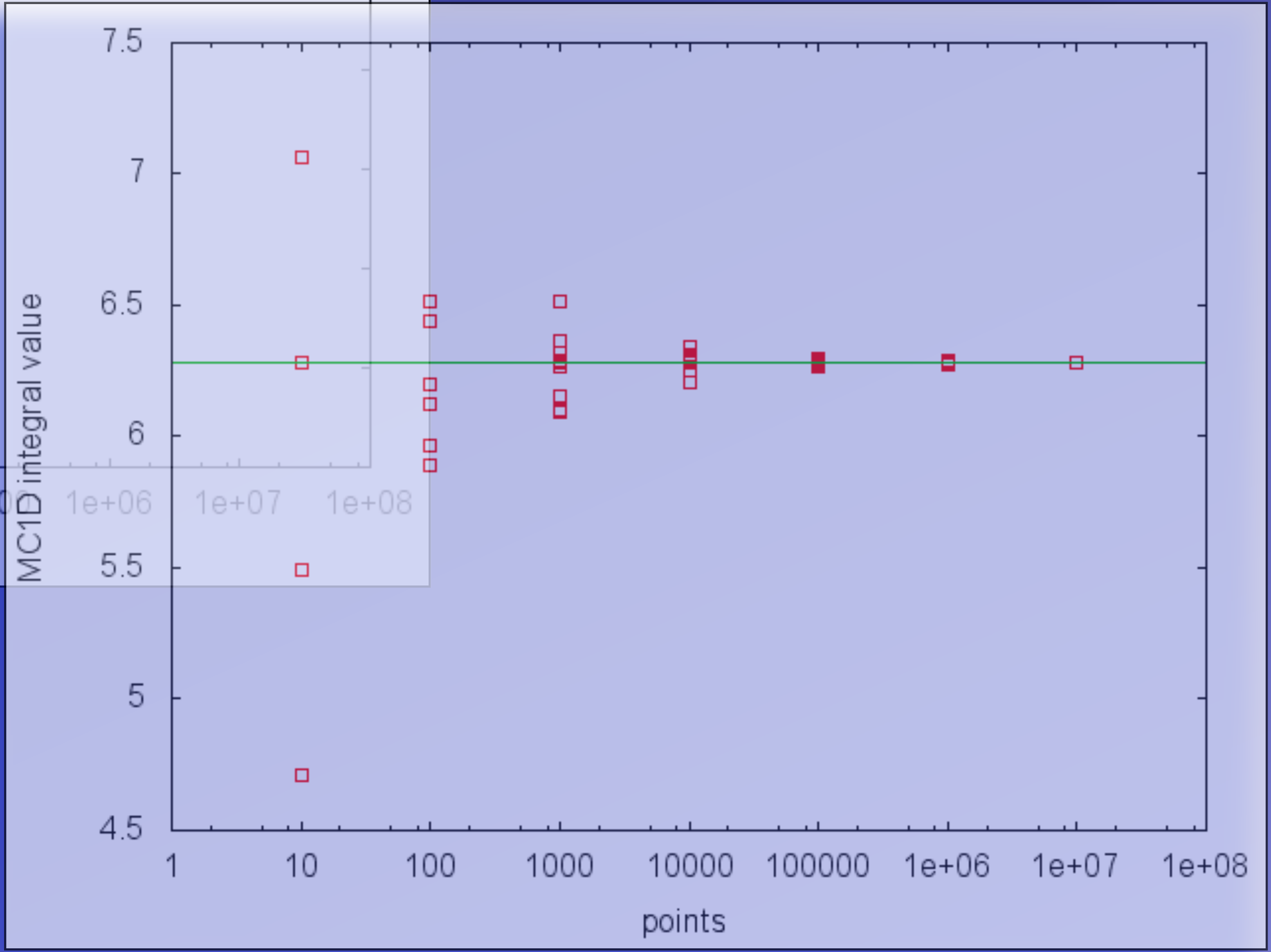
So... how does the 1D Monte Carlo method for integration behave?
I've run a test calculating the same integral with a variable number of points (from 10 to 10^7). Each calculation was repeated 10 times. Just for fun I also redid the same test with a program written in Python which uses a much better random number generator (the Mersenne Twister)



<- this is the C version

The green line is the real value of the integral

and this is the Python version ->



Ex.9 - 1D Cellular Automata

Implement a program to generate 1D Cellular Automata. A Cellular Automaton is a discrete model of a system which evolves over time.

A 1D Cellular Automaton consists of ‘cells’ living on a line. Each cell can be dead (0) or alive (1).

Each cell has two neighbors and its evolution is determined by the state of the two neighbors, its own state and a set of evolution rules, which determine the new state at the next iteration.

Write a program that evolves a cellular automaton given an initial configuration and a set of evolution rules.

See: http://en.wikipedia.org/wiki/Cellular_automaton and http://www.stilldreamer.com/mathematics/1d_cellular_automaton/

Probably the most complex exercise in the set. This required quite a bit of algorithmic thinking compared to the previous ones.

Your solutions have been ... creative. Which is good.

Some of you have chosen to specialize the program to a specific rule writing ad-hoc code, rather than attempting a generic implementation.

1D CA

I will use this example in the C++ lecture, hence the longer description.

Let's give some nomenclature.

A cellular automaton lives on a playground of N spaces.
The playground is circular (its leftmost element and its rightmost element are next to each other).
Each 'space' in the playground is called a cell.
A cell can be dead (0) or alive (1).
Time flows in discrete steps. At each given step the cell state can change or remain the same.
The cell state changes depending on its own state and the state of its first neighbors (usually known as a neighborhood).

Each automaton is completely defined by its initial state and its evolution rules.

Since there are eight possible configurations for a neighborhood, each with a possible outcome of 0 or 1 in the next state, there is a total of 256 possible evolution rules.

State	111	110	101	100	011	010	001	000
Outcome	0	0	1	1	0	1	0	0

As an example, this is rule 52 (2^2 + 2^4 + 2^5).

```c
#include <stdio.h>
#include <stdlib.h>

#define PLAYGROUND_SIZE 80
#define GENERATIONS     1
#define RULE            30

void init_rules(int rule, int* evolutionTable) {
  int idx;
  for(idx=0; idx<8; idx++) {
    evolutionTable[idx] = (rule >> idx) & 1;
  }
}

void init_playground(int size, int* playground) {
  int idx;
  for(idx=0;idx<size; idx++) { playground[idx] = 0; }
  /* impulse */
  playground[size/2] = 1;
}

void print_playground(int size, int* playground) {
  int idx;
  for(idx=0; idx<size; idx++){
    if(playground[idx]==1) { printf("o"); }
    else                   { printf(" "); }
  }
  printf("\n");
}
```

```c
void evolve_automaton(int size, int generations, int* evolutionTable, int* playground) {
  int* tmp;
  int idx, cell_idx;
  int c0, c1;
  int* tmp_playground = malloc(PLAYGROUND_SIZE*sizeof(int));

  print_playground(size, playground);

  for(idx=0; idx<generations; idx++) {
    for(cell_idx=0; cell_idx<size; cell_idx++) {
      c0 = cell_idx==0?size-1:cell_idx-1;
      c1 = playground[c0] * 4 + playground[cell_idx] * 2 + playground[(cell_idx+1)%size];
      tmp_playground[cell_idx] = evolutionTable[c1];
    }

    tmp = tmp_playground;
    tmp_playground = playground;
    playground = tmp;

    print_playground(size, playground);
  }
  free(tmp_playground);
}

int main (int argc, const char* argv[]) {
  int evolutionTable[8];
  int* playground = malloc(PLAYGROUND_SIZE*sizeof(int));

  init_rules(RULE, evolutionTable);

  init_playground(PLAYGROUND_SIZE, playground);
  evolve_automaton(PLAYGROUND_SIZE, GENERATIONS, evolutionTable, playground);

  free(playground);
  return 0;
}
```

Ex.10 - The Sieve of Eratosthenes

The Sieve of Eratosthenes is a simple iterative algorithm to generate a table of prime numbers.

Take the list of the first 100 numbers, and start by removing the multiples of 2. Then proceed to remove the multiples of 3. Then the multiples of 5 (4 has been removed when we removed the multiples of 2) and so on. At the end of this process, what's left are only the prime numbers between 1 and 100.

Write a program to implement this algorithm and use it to calculate the prime factors of an integer number.

See:
http://en.wikipedia.org/wiki/Prime_factor
http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

This is the exercise where I saw the most 'creativity' in the algorithm interpretation. Rather than implementing the Sieve most of you implemented a direct visit method where rather than decimating the array removing the multiples "directly", you selected the prime and then iterated against every outstanding element of the array and checked (via the modulo operator) if it was a multiple before removing it.

Same result but different algorithm... I'm kind of curious as to why most of you used this particular approach.

My implementation of the algorithm follows the standard route, but adds a little bit of fun but performing a zero suppression on the list of primes.

I generate the list of primes 'sieving' the array. This leaves me with an array containing the prime numbers and a lot ot zeroes.

To produce a compact list I perform a zero-suppression creating a new list and copying the non-zero numbers in it. The only 'innovative' trick is that rather than doing it iteratively, I do it recursively using a technique which is typical of functional programming where a list is split into a first element (which is operated upon) and a list containing the rest of the elements which is then recursively fed to the function itself. Recursion ends when the function receives the empty list.

The only additional trick is that to support my technique I've used a simple self-describing array, that is an array which carries informations about itself (its length in this particular case, which is stored in its first element). A better approach would be to use a struct that holds the array itself and an integer containing its lenght. This would also work for non-integer arrays.

I've split the program in functional blocks to make it a little more readable (even though they're not in the same order you would find in the actual program).

Don't worry if the zero-suppression code looks a bit unfamiliar.

```c
int main (int argc, const char * argv[]) {
  int* primeNumbers;
  int* factors;
  int idx;

  primeNumbers = produce_sieve(SIEVE_SIZE);
  for(idx=1; idx<primeNumbers[0]; idx++) printf("%d ", primeNumbers[idx]);
  printf("\n");

  factors = prime_factors(NUMBER, primeNumbers);

  if(!factors[0]) {
    printf("It wasn't possible to fully calculate the prime factors of %d with ", NUMBER);
    printf("a table containing %d primes.\n", primeNumbers[0]);
    printf("This is the best I could do:\n");
    printf("%d ~= ", NUMBER);
  } else {
    printf("%d = ", NUMBER);
  }

  for(idx=1; idx<primeNumbers[0]; idx++) {
    if(factors[idx]!=0) {
      printf("%d^%d ", primeNumbers[idx], factors[idx]);
    }
  }
  printf("\n");

  free(factors);
  return 0;
}
```

```
/* The Sieve of Eratosthenes */

int* produce_sieve(int size) {
    int* data = (int*)malloc(size*sizeof(int));
    int idx, done, base;

    for(idx=0; idx<size; idx++) data[idx] = idx;

    done = 0;
    idx = 2; /* skip 0 and 1 */
    data[1] = 0; /* remove 1 from the sieve */
    do {
      if(data[idx] != 0) {
        for(base=2*data[idx]; base < size; base += data[idx]) data[base] = 0;
      }
      if(++idx>size) { done=1; }
    } while(!done);

    return zero_suppress(data, size);
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define SIEVE_SIZE 100
#define NUMBER     53139008 /* 13^2 * 17^3 * 2^6 */
/* #define NUMBER     101 */

/* zero suppression */
int* append(int* list, int datum) {
  list[list[0]++] = datum;
  return list;
}

int* zero_suppress_me(int* data, int size, int* outcome) {
  if(size==0) return outcome;
  if(data[0]) return zero_suppress_me(++data, --size, append(outcome, data[0]));
  return             zero_suppress_me(++data, --size, outcome);
}

int* zero_suppress(int* data, int size) {
  int* outcome = (int*)malloc(size);
  outcome[0] = 1;
  outcome = zero_suppress_me(data, size, outcome);
  outcome = realloc(outcome, outcome[0]);
  return outcome;
}
```

```
/* Brute force prime factors calculation */

int divideAll(int value, int factor) {
  if(value%factor==0) return 1+divideAll(value/factor, factor);
  return 0;
}

int check_factorization(int value, int* primes, int* factors) {
  int cvalue, idx;
  for(idx=1; idx<primes[0]; idx++) {
    cvalue *= (int)pow(primes[idx], factors[idx]);
  }
  if(cvalue==value) return 1;
  else              return 0;
}

int* prime_factors(int value, int* primes) {
  int table_size = primes[0];
  int* factors   = (int*)malloc(table_size*sizeof(int));
  int idx;

  for(idx=1; idx<table_size; idx++) {
    factors[idx] = divideAll(value, primes[idx]);
  }

  if(check_factorization(value, primes, factors)) factors[0] = 0;
  else                                            factors[0] = 1;

  return factors;
}
```

In the end...

- The exercises:
  - I’ve received relatively few sets of exercises
    - You can still send yours in and ‘benefit’ from some personalized comments on your code ;)
    - ALL exercise blocks will be corrected. Be patient.
      - ... but don’t wait too long...
- Thanks for the time and effort you’ve put toward your implementations
  - For some of you, these exercises were evidently way too simple... sorry :)
    - ‘Yet another Fibonacci calculator...’
  - For some were way too difficult... sorry :)
- All in all, it was expected...
- ... but! If you’ve found these exercises difficult, you *NEED* to work on practicising your programming skills and algorithmic thinking.
  - It doesn’t get any easier
  - If you have questions, ask them...

In the end (this time for real)

- A programming language is just a tool to an end.
- There's not a "right" or a "wrong" tool, just a lot of different tools.
  - We stole a number of techniques (memoization, catamorphisms, recursion, ...) from functional programming...
    - They're tools.
  - The larger your set of tools and your skill in using them, the higher your expressive power and ability to tackle complex problems.
- And remember:

A language that doesn't affect the way you think about programming, is not worth knowing.

(Alan J. Perlis, Epigrams on Programming)