MŰEGYETEM 1782

Faculty of Electrical Engineering and Informatics

Basics of Programming 1

Integration C program

Developers doc

Student
Klevis Imeri

Lecturer
Gabor Horvath

Budapest, november, 2022

Contents

main.c

Functions:

- int *main*()
- void *interval_integrator*()

int main()

1. Initializes the settings with default values:

```
double step = 0.001, start = -3.14, end = 3.14;
```

The step is the partition, the start is the lower bound and the end is the upper bound. These variable names are used because they make the code cleaner without losing the meaning.
When it calls the menu from:

```
start_menu(&step, &start, &end);
```

link

The menu returns the new edited settings.
It checks if the upper bound is lower than the lower bound. If yes it interchanges their values:

```
if(start > end){
  double temp = end;
  end = start;
  start = temp;
}
```

The lower bound should be smaller than the upper bound for the program to work.
It calls the function to take the expression from the user:

```
size_t size_expression;
char *expression = take_valid_expression(&size_expression);
```

link

It puts the expression char array into a pointer. The expression array is dynamically allocated from the function. Also the function returns the size of the array through the pointer.
It gets the number of nodes and creates a dynamically allocated array to store the nodes:

```
    size_t number_of_nodes =
    number_of_nodes_in_expression(expression, size_expression);
    Node *nodes = (Node *)malloc(sizeof(Node)*number_of_nodes);
```

Calls the parser to parse the nodes:

```
    parser(expression, nodes, &start, 0);
```

link

Calls integrator to find the Riemann Sum and Definite Integral:

```
interval_integrator(nodes, number_of_nodes, &start, end, step);
```

link

Creates the image (bitmap):

```
create_bitmap(nodes, number_of_nodes, &start, end, step, 2000, 2000,
600, 6);
```

link

Notes:
You can use the r.sh with the bash command to compile the c files with gcc if you are using linux.

void interval_integrator()

Syntax:

```
double interval_integrator(Node *nodes, size_t number_of_nodes,
double *start, double end, double step);
```

```
Parameters:
   -  Node *nodes      | is the array where the nodes are stored
   -  number_of_nodes  | number of nodes in the array
   -  start            | the lower bound as pointer (because it
                       | connected to the nodes of the expression as
                       | a pointer. It is the x variable.)
   -  end              | the upper bound
   -  step             | the partition

Return:
   -  Prints: The Riemann Sum
   -  Prints: The Definite_integral

Usage:
It integrates and finds the Riemann Sum and the Definite Integral
Formula used in a for loop;

     riemann_sum += fabs(eval_Nodes(nodes, number_of_nodes))*step;
     definite_integral += eval_Nodes(nodes, number_of_nodes)*step;
```

menu.c

Functions:

- void *start_menu*()

void *start_menu*()

Syntax:

```
void start_menu(double *partition, double *lower_bound, double
*upper_bound);
```

Parameters:

They should be initialized.

- partition	the partition variable pointer
- lower_bound	the lower bound variable pointer
- upper_bound	the upper bound variable pointer

Return:

- partition	value (pointer)
- lower_bound	value (pointer)
- upper_bound	value (pointer)

Usage:

It starts the menu and it keeps it running. According to the user inputs, this function changes the partition, the lower bound and the upper bound values. The user inputs are checked if they are valid. The startup values of the partition, the lower bound and the upper bound values are set before imputed to the function.

How the menu looks:

```
Partition = 0.001     Lower bound = -3.14   Upper bound = 3.14
   0.Change the partition.
   1.Change Lower bound.
   2.Change Upper bound.
   3.Continue.
Choose option:
```

chararray.c

Functions:

- void *print_char_array*()
- void *copyarr*()
- char *take_expression()
- char *take_valid_expression()

void *print_char_array*()

Syntax:

```
void print_char_array(char *array, size_t size);
```

Parameters:

```
   -  array      | the array you want to print
   -  size       | the size of the array
```

Return: void

Usage:

It prints the array in the console. The format:

```
     { elm1, elm2,...., elmN }
```

void *copyarr*()

Syntax:

```
void copyarr(char* destination, char* source, size_t size_source);
```

Parameters:

```
   -  destination      | is a char array. The new characters will be
                       | copied into it.
   -  source           | is a char array. The new characters will be
                       | copied from here to the destination array.
   -  size_source      | the size of the source char array
```

Returns:

```
   -  destination      | char array (pointer)
```

Usage:

It copies the source array into the destination array.
The destination array should be bigger.

char *take_expression()

Syntax:

```
char *take_expression(size_t *size);
```

Parameters:

```
   -  size  | is the pointer to the size of the array
```

Returns:

```
   -  expression | imputed by the user as a dynamically allocated
                 | char array
   -  size       | of the char array (pointer)
```

Usage:

Asks the user to input the mathematical expression and dynamically allocates the char array for the expression. Returns the pointer to the array. The format of the array created: {'(' expression ')'}.

int is_expression_valid()

Syntax:

```
int is_expression_valid(char *expression, size_t size)
```

Parameters:

```
 - expression    | the array of the mathematical expression
 - size          | size of the array
```

Returns:

```
 - int           | 1 if the user entered a valid mathematical
                 | expression, else 0
```

Usage:

def'n: # - number of

It takes the mathematical expression and checks if the user has entered a valid math expression that our program supports. it checks for:

```
 - size>2        | if the size if 2 that means that the char array is
                 | empty. It only has '(',')'. If the array is empty
                 | it returns 0 and prints to the screen:
                 |"ERROR! You wrote nothing!"
 - #'(' != #')'  | if the number of open brackets is not equal to the
                 | the number of closed bracket than it return 0
                 | and prints:
                 |"ERROR! You didn't close the bracket!"
 - x*+9          | if 2 binary operators are closed to each other.
                 | ERROR! You have two binary operations one after
                 | the other!
- valid f(x)     | if the user entered a valid function.
                 | ERROR! You may have spelling mistakes or inputted
                 | a word which is not a function
-(1) => (1.0)    | (<num?) is a node identifier.
-3x => (3x)      | not a valid expression
-#nodes != 0     | the number of nodes should not be zero
```

char *take_valid_expression()

Syntax:

```
char *take_valid_expression(size_t *size);
```

Parameters:

```
     - size     |is the pointer to the size of the array
```

Returns:

```
   -  expression | imputed by the user as a dynamically allocated
                 | char array
```

- size | of the char array (pointer)

Usage:
Uses the take_expression() function to take expression form the user and then uses the *is_expression_valid()* function to check if the user imputed a valid input. If not it loops until the user enters a valid expression. When the user enters a valid input it returns the dynamically allocated array of the expression.

cursorlib.c

Functions:

- typedef enum Direction{}
- char *cursor()

typedef enum Direction{}

Syntax:

```
typedef enum Direction{
    RIGHT,
    LEFT,
    FUNCTION
} Direction;
```

Usage:

It is used to describe the behavior of the cursor.

Note: it is located in cursorlib.h.

char *cursor()

Syntax:

```
char *cursor(char *cursor, char c1, char c2, char c3, Direction mode)
```

Parameters:

- cursor	is is the pointer to the beginning of the search
- c1	is the first character
- c2	is the second character
- c3	is the third character
- mode	is the mode of the cursor

Return:

Depends on the mode:

Notes: ')' is a stopping char for cursor.

- RIGHT | it traverses toward the right. Which char
  | it hits first (c1,c2,c3) it returns its address

- LEFT | it traverses toward the left. Which char
  | it hits first (c1,c2,c3) it returns its address

- FUNCTION | it traverses toward the right. Which char
  | letter it hits fist except 'x' it returns its
  | address. Uses *is_small_letter()*

Usage:

Traversing in the char array given the starting address.
Checks if the given character exists according to mode.
Gives back the address of that char. If the char doesn’t exist
it returns the ‘)’ address.

charlib.c

Functions:

```
      - int what_function(char *cursor)
      - int is_binary_operator(char c)
      - int is_small_letter(char c, char omit)
      - int is_letter(char c, char omit)
      - int is_num(char c);
      - int char_to_int(char c);
      - int string_to_int(char *start);
      - double string_to_decimal(char *start);
      - double convert_to_num(char *address, Direction DIR);
      - int is_variable_multiplicaton(char *cursor);
```

Note: For the function with color there is no documentation here.

int *char_to_int*()

Syntax:

```
int char_to_int(char c);
```

Parameters:

```
   -  c                        | is the input char value
```

Returns:

```
   -  one digit int [0,9]    | the integers from the char [‘0’,‘9’]
```

Usage:

It takes the value of the char and converts it into corresponding int. It converts the characters that represent the numbers from 1 to 9 to integer numbers.

Example:

‘1’(char) -> 1(int)

‘8’(char) -> 8(int)

int *string_to_int*()

Syntax:

```
int string_to_int(char *start);
```

```
Parameters:
   -  start      | the pointer to the starting position of the
                 | integer in the expression. One before the
                 | starting digit.
Returns:
   -  int        | the number that the string represent
   -  ‘\0’       | the string that represented the number
                 | becomes all ‘\0’.
```

Usage:
It takes the starting position of the string that represents the int number. The starting position is one before the first digit starts. I tranverses to the left until it hits a char that is not a number. It returns the int. If we have a decimal number we use this function to find the integer part of that number.

Example:
c is the starting char (not the first digit 0):

```
{...‘+’,‘2’, ‘3’, ‘2’, ‘1’, ‘0’, ‘c’,...}
                                  ↑
Traversing to the left:
{...‘+’,‘2’, ‘3’, ‘2’, ‘1’, ‘0’, ‘c’,...}
                                 ←↑
{...‘+’,‘2’, ‘3’, ‘2’, ‘1’, ‘0’, ‘c’,...}    | int = 0*10^0 = 0
                            ←↑
{...‘+’,‘2’, ‘3’, ‘2’, ‘1’, ‘\0’, ‘c’,...}   | int = 0 + 1*10^1 = 10
                       ←↑
                   . . .
{...‘+’,‘2’, ‘\0’, ‘\0’, ‘\0’, ‘\0’, ‘c’,...}| int = 3210 + 2*10^4
```

```
        ←↑
{...‘+’,‘\0’, ‘\0’, ‘\0’, ‘\0’, ‘\0’, ‘c’,...}   | int = 23210
      ↑
```

Not a number (in our case the +) it stops.
It returns the number 23210. If we have a decimal number we use this function to find the integer part of that number. In this case the starting char is the dot.
Example:

```
{...‘-’,‘2’, ‘3’, ‘.’, ‘1’, ‘0’, ‘c,...}
                    ↑
```

‘.’ is the starting char.
It will transverse until it hits the ‘-’ and it will return 23.

int *string_to_decimal*();

Syntax:

```
double string_to_decimal(char *start);
```

Parameters:

```
   -  start      | the pointer to the starting position of the
                 | Decimal part in the expression. Usually is the
                 | ‘.’ but can be any char.
```

Returns:

```
   -  the “decimal”    | the numbers after ‘.’ in the decimal
                       | representation
   -  ‘\0’             | the string that represented the number
                       | becomes all ‘\0’.
```

Usage:
Usually we take the position of the dot and compute the decimal number to the right until it hits a char that is not a number. It returns the value after the ‘.’. Leaves back a trail of

Example:
The dot is the starting position:

```
{...‘-’,‘2’, ‘3’, ‘.’, ‘1’, ‘6’, ‘c,...}
                    ↑
```

Transverses to the right:

```
{...‘-’,‘2’, ‘3’, ‘.’, ‘1’, ‘6’, ‘c,...}
                        ↑→
```

Evaluates:

```
{...‘-’,‘2’, ‘3’, ‘.’, ‘1’, ‘6’, ‘c,...}    | dec = 1*10^(-1) = 0.1
                        ↑→
{...‘-’,‘2’, ‘3’, ‘.’, ‘\0’, ‘6’, ‘c,...}   | dec = 0.1 + 6*10^(-2)
                              ↑→
```

It hits a char that is not a number:

```
{...‘-’,‘2’, ‘3’, ‘.’, ‘\0’, ‘\0’, ‘c,...} | dec = 0.16
                                    ↑→
```

It returns the 0.16 decimal number.

int *convert_to_num*()

Syntax:

```
double convert_to_num(char *address, Direction DIR);
```

Parameters:

```
     - address  | address of the number in expression
     - DIR      | right or left of the node
```

Returns:

```
     - number   | the decimal number form the char array
```

Usage:

Every time you need to convert a number that is a string of char into a double. It uses the string_to_decimal() and string_to_int().

parserlib.c

def'n: '(expressions ')' is a block

Functions:

- void *print_block*(char *block)
- int block_parser()
- int parser()

int *block_parser*()

Syntax:

```
int block_parser(char *block, Node *nodes, double *x, int
index_nodes);
```

Parameters:

- block | block of expressions without '('')' inside
- nodes | the array of nodes
- x | the pointer to the variable
- index_nodes | keep track of the number of nodes created

Returns:

- index_nodes | the number of nodes created
- nodes | the array filled with the nodes (pointer)

Usage:

You give it the block of expressions and the variable x and the array where to put the nodes. It fills the array with all the nodes in the block of expressions. It searches the nodes using hierarchy. It uses the create_node() function to create and put the nodes in the array of nodes.

Hierarchy:

- functions
- ^
- * / <num>x (x)
- + -

```
Example:
(x+2/x^2)  | searching for functions
↑→
(x+2/x^2)  | did not find functions
        ↑
(x+2/x^2)  | searching for '^'
↑→
(x+2/x^2)  | if found '^'. create_node(↑)
      ↑
(x+2/[])   | searching for '^'
     ↑→
(x+2/[])   | did not find '^'
       ↑
(x+2/[])   | searching for '*' '/' <num>x (x)
↑→
(x+2/[])   | if found '/'. create_node(↑)
    ↑
(x+[])     | did not find '*' '/' <num>x (x)
     ↑
(x+2/[])   | searching for '+' '-'
↑→
(x+2/[])   | it found '+'. create_node(↑)
  ↑
([])       | remove '(' and ')'
[]         | returns the number of nodes created = index_nodes + 3
```

int parser()

```
Syntax:
int parser(char *block, Node *nodes, double *x, int index_nodes);

Parameters:
      - block           | block of expressions without '('')' inside
      - nodes           | the array of nodes
      - x               | the pointer to the variable
      - index_nodes     | keep track of the number of nodes created
```

Returns:

- index_nodes | the number of nodes created
- nodes | the array filled with the nodes (pointer)

Note: it uses block_parser().

Example:
p1 - pareser 1
p2 - parser 2
'(' - Searching for '('
✓ - Found the '('
x - Did not found '('
bp - calls the function block_parser()
[] - parsed the nodes by block_parser()
✓p3 - p3 is finished

```
           p1->'('                                  p1->✓
(x * (x^(2+2)) - sin(x/2))  =>  (x * (x^(2+2)) - sin(x/2)) =>
↑→                                    ↑

  p2->'('      p2->✓      p3->'('    p3->x                     ✓p3
(x^(2+2)) => (x^(2+2)) => (2+2)  => (2+2) => p3bp => (x * (x^[]) - sin(x/2)) =>
↑→               ↑          ↑→              ↑

p2->'('       p2->x                   ✓p2                    p1->'('
(x^[]) => (x^[]) => p2bp => (x * [] - sin(x/2)) => (x * [] - sin(x/2)) =>
↑→              ↑                                  ↑→

     p1->✓                 p2->'('    p2->x                 ✓p2
(x * [] - sin(x/2)) => (x/2)  =>  (x/2) => p2bp = (x * [] - sin [] ) =>
             ↑         ↑→               ↑
   p1->'('                   p1->x
(x * [] - sin [] ) =>  (x * [] - sin [] ) => p1pb => [] => expression parsed
↑→                                        ↑
```

nodelib.c

Functions:

- typedef struct Node{}
- size_t *number_of_nodes_in_expression*()
- void *print_Node*(Node node)
- void *print_Node_array*(Node *array, size_t size)
- void *create_Node*()
- double *eval_Nodes*()

typedef struct Node{}

Syntax:

```
typedef struct Node{
    char type; //is it + or - or...
    int value; //every function has an int value assign to it
    double x; //the left num
    double y; //the right num
    double *px; //point to the left result or num
    double *py; //point to the right result or num
    double result;
    int index;
} Node;
```

Usage:

It is a node in a binary tree. The pointers px and py are connected to the results of the other nodes. Every node is stored in the array of nodes. x and y are used to store numbers. px and py can point to then. px and py values at the addresses get evaluated when called the eval_Nods() function.

void create_Node()

Syntax:

```
void create_Node(char *address, Node *nodes, double *variable, int
index_nodes)
```

Parameters:

```
      - address        | address of the node
      - nodes          | the array of nodes
      - variable       | the pointer to the variable
      - index_nodes    | keep track of the number of nodes created
```

Returns:

```
      - nodes          | the array filled with the new node (pointer)
```

Usage:

You give it the position of the node and it:

1. Creates the node
2. Deletes it from the expression
3. Creates the node identifier for that node

Types of nodes and entering the node:

```
 - Binary operation: ^,*,/,+,-         | at binary operator
 - Special: <num>x, (x)                | at x
 - Functions: sin, cos, abs, log, exp  | at first letter of function
```

Explanation:

1.Entering a binary nodes:

```
x*3.45     | it sets the type to '*'(char) and value to '*'(int)
 ↑
x*3.45     | it searches to the right and it find a number
  ↑→
x*3.45     | calls convert_to_num(RIGHT) and puts it on y and *py
  ↑
x*3.45     | it resets and it searches to the left
←↑
x*'\0'     | it find a variable puts it on *px
↑
 []        | were the node was  it creates a node identifier
```

Notes: the node identifier is a char value telling the index of the node in the node array.

```
Entering <num>x:
3x         | is sets the type t0 '*'(char) and value to '*'(int)
 ↑
3x         | puts the cursor-- so it can transverse to the right
↑
3x         | finds x. Makes *py point to variable
 ↑
3x         | transverses left. Find a num. convert_to_num(LEFT)
↑          | sets x = 3 *px =3
[]         | creates the node identifier
Entering a func:
sin []     | calls the what_function() to check what function it is
↑
sin []     | makes the type 'f' and value '1' (read what_function())
↑
sin []     | set the pointer to the end of the function
   ↑
sin []     | transverse right find []. Makes py point to the result
     ↑     | of the node that is identifies with [](node identifier)
sin []     | px = py
↑
[]         | creates the node identifier for the function
```

double eval_Nodes()

Syntax:

```
double eval_Nodes(Node *nodes, size_t number_of_nodes)
```

Parameters:

```
     - nodes                | the array of nodes
     - number_of_nodes      | the size of the array of nodes
```

Returns:

```
     - double    | the value of evaluating all the nodes. Or the
                 | value the mathematical expression gives.
```

Usage:

Evaluates the array of nodes and finds the result of the expression correlating to the variable. Returns it. Values at py and px are evaluated not x and y.

bitmap.c

Functions:

- typedef struct rgb_data{}
- void *save_bitmap*()
- void *create_bitmap*()

typedef struct rgb_data{}

Syntax:

```
typedef struct rgb_data{
        float r, g, b;
} rgb_data;
```

Usage:

It is used to store a RGB color.

void save_bitmap()

Syntax:

```
void save_bitmap(const char *file_name, int width, int height, int
dpi, rgb_data *pixel_data)
```

Parameters:

- filename	the bmp image filename
- width height dpi	pixel parameters
- pixel_dat	the array of all pixels

Returns:

- functions.bmp	the bmp image

Usage:

It takes all the settings and the array of pixels. It creates the heder of the bpm and attaches the array of pixels. It create the bmp.

void create_bitmap()

Syntax:

```
void create_bitmap(Node *nodes, size_t size, double *start, double
end, double step, int width, int height, int dpi, int
numberline_length)
```

Parameters:

```
 - nodes               | the array of nodes (expression)
 - size, start, step   | partition, lower, upper bound
 - width, height , dpi | pixel data
```

Returns:

```
 - functions.bmp       | the bmp image
```

Usage:

It takes the expressions and the integration parameters. Creates the array of pixels. Create the number lines, the function, the rectangles. Calls the save_bitmap() to save the bmp.

Testing

The program is tested:

a. in normal conditions

b. in bound conditions

c. not continuously behaved functions

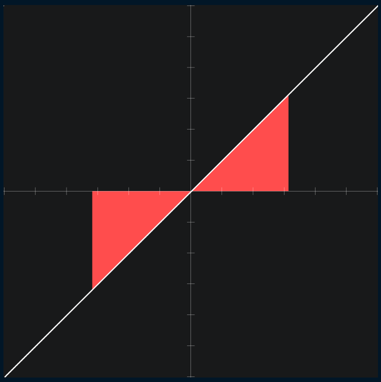
d. User invalid expressions

Normal conditions

The functions the program is tested with in normal conditions:

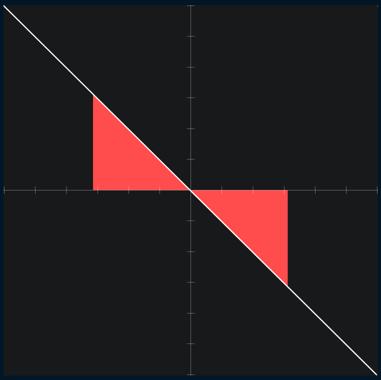
```
Partition = 0.001     Lower bound = -3.14   Upper bound = 3.14
////////////////////////////////////////////////////////////////////
Enter the function: x

The Riemann Sum:        9.862740
The Definite Integral:  -0.000000
```



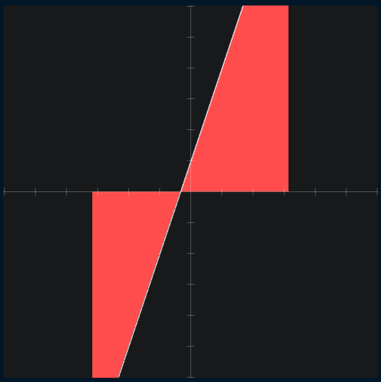
```
////////////////////////////////////////////////////////////////////
Enter the function: -x

The Riemann Sum:        9.862740
The Definite Integral:  0.000000
```



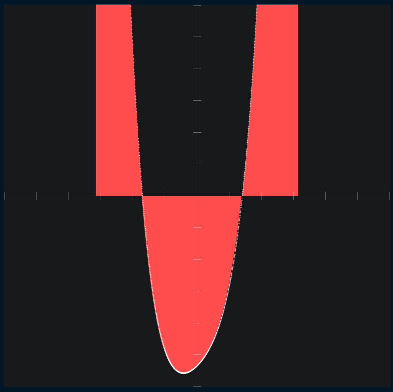
```
////////////////////////////////////////////////////////////////////
Enter the function: (3x)+1

The Riemann Sum:        29.921554
The Definite Integral:  6.281000
```



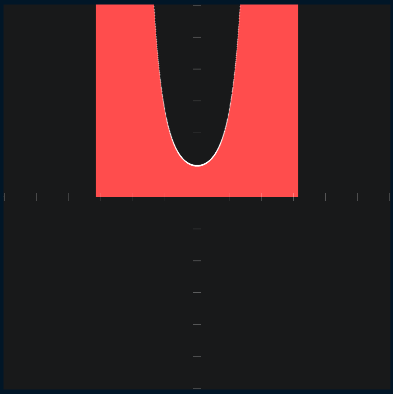
```
////////////////////////////////////////////////////////////////////
Enter the function: 1/2 *x^4 + x^2- 5.32 +x

The Riemann Sum:        73.226252
The Definite Integral:  48.331941
```



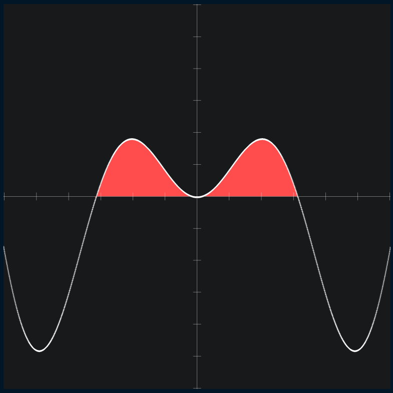
```
////////////////////////////////////////////////////////////////////
Enter the function: exp(x^2)

The Riemann Sum:        6492.158769
The Definite Integral:  6492.158769
```



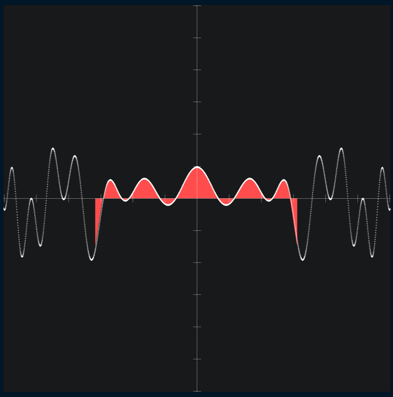
```
////////////////////////////////////////////////////////////////////
Enter the function: x*sin(x)

The Riemann Sum:        6.283182
The Definite Integral:  6.283182
```

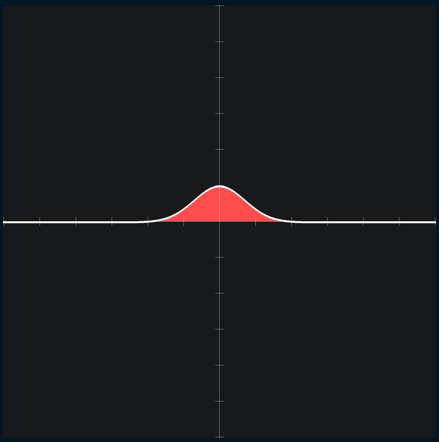


```
////////////////////////////////////////////////////////////////////
Enter the function: cos(3x) + sin(x^2)

The Riemann Sum:        2.415905
The Definite Integral:  1.548423
```

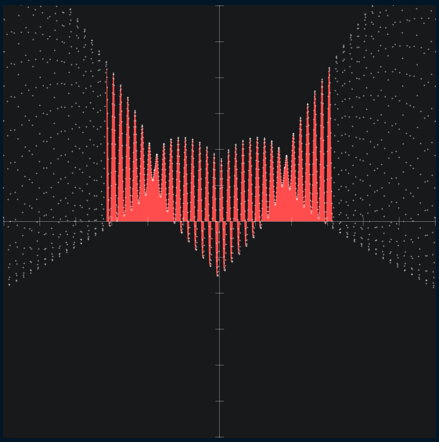


```
////////////////////////////////////////////////////////////////////
Enter the function: exp(-x^2)
```



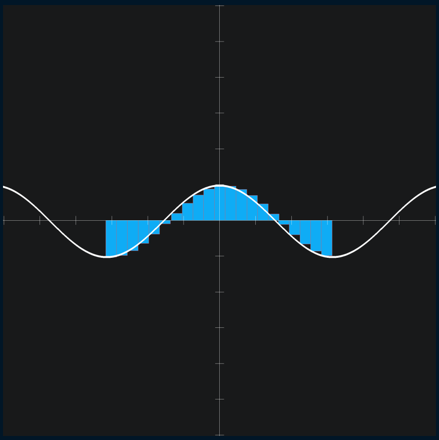
```
The Riemann Sum:        1.772438
The Definite Integral:  1.772438
```

```
////////////////////////////////////////////////////////////////////
Enter the function: x^(2/3)+0.9*(3.3-x^2)^(1/2)*sin(10*3.14*x)
```



```
The Riemann Sum:        9.070347
The Definite Integral:  8.081880
```

```
///////////////////////////////////////////////////////////////////
Partition = 0.3  Lower bound = -3.14   Upper bound = 3.14
Enter the function: cos(x)
```

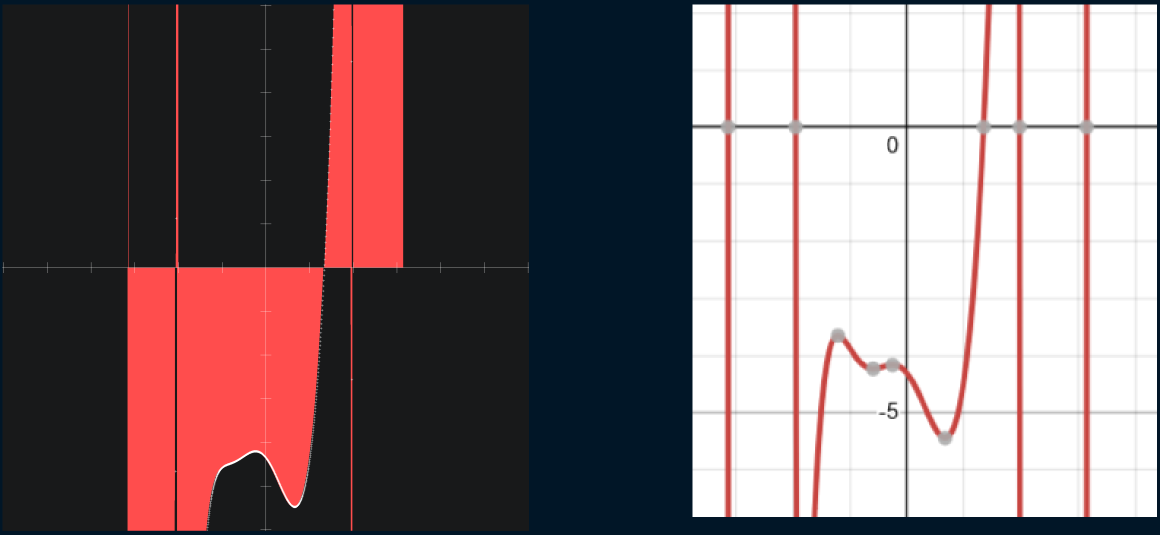


```
The Riemann Sum:         4.021494
The Definite Integral:  -0.016713
```

```
///////////////////////////////////////////////////////////////////
```

Bound conditions

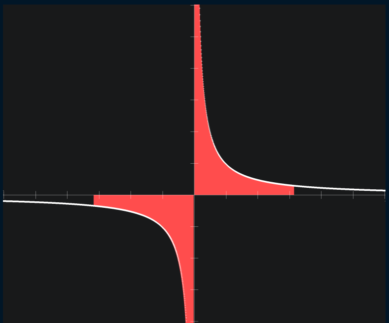
The conditions where things are very small or very large:
1. partitions smaller than 0.001 do not have any effects on the result. They just are unreasonable and slow the program.
2. upper and lower bounds greater or smaller than +-100,000 slow the program down even though it still works.
3. cos(3x) + sin(x^2)+(x^3-x)/(x^2-4)+1/2 *x^4 + x^2- 5.32 +x - 1/(-x) - (3x)+1-exp(x)/(x)+exp(x^2)*sin(x). This is an expression you can enter in the program and the graph will be the same as in desmos.



The graphs of the functions. a) program, b) desmos

Not continuously behaved functions

The previous function does not fail to graphs but it fails to find the area precisely because the function is very steep and it has limits that approach infinity.
A simpler example:
//
Enter the function: 1/x



```
The Riemann Sum:         4256584713.949382
The Definite Integral:  -4256584696.690656
///////////////////////////////////////////////////////////////////
```

Whenever the functions are divided by zero form the program gives the area of a big number and not infinite. Remember the partitions do not approach zero; they are fixed. But anyway, results of this kind should not be taken into consideration.

User invalid expressions

In this section we will try to enter weird things to fail the program.

```
///////////////////////////////////////////////////////////////////
Partition = 0.001     Lower bound = -3.14   Upper bound = 3.14
   0.Change the partition.
   1.Change Lower bound.
   2.Change Upper bound.
   3.Continue.
Choose option: asdofpgklks}{}}|^[~~~ewd*&*&%$#$$@!$%**&*(
///////////////////////////////////////////////////////////////////
```

The program doesn't break.

The program doesn't break no matter what you enter in this stage.

```
///////////////////////////////////////////////////////////////////
Enter the function: 03249759237r9dhfoihgdi0u32yghwsr0-g89hsdouhf
ERROR! You may have spelling mistakes or inputted a which is not a
function
Press Enter to continue...
///////////////////////////////////////////////////////////////////
Enter the function: sin
ERROR! You may have spelling mistakes or inputted a which is not a
function
Press Enter to continue...
///////////////////////////////////////////////////////////////////
Enter the function: sin(X)+cos(32x)
```

```
ERROR! You may have spelling mistakes or inputted a which is not a function
Press Enter to continue...
////////////////////////////////////////////////////////////////////
Enter the function: sin(X)+cos(32x)
ERROR! You may have spelling mistakes or inputted a which is not a function
Press Enter to continue...
```

It recognized the big X

```
////////////////////////////////////////////////////////////////////
Enter the function: x*+y*4^%4
ERROR! You have two binary operations one after the other!
Press Enter to continue...
////////////////////////////////////////////////////////////////////
Enter the function: sin(x
ERROR! You didn't close a bracket!
Press Enter to continue...
////////////////////////////////////////////////////////////////////
Enter the function: cos(3x) + sin(x^2)+(x^3-x)/(x^2-4)+1/2 *x^4 + x^2- 5.32 +x - 1/(-x*sin(x) + exp(x)/(x)
ERROR! You didn't close a bracket!
Press Enter to continue...
```

Can you find which one?

```
////////////////////////////////////////////////////////////////////
```

*User may find cases where it can make the program crash but they are not easy to find