1. Como es bien sabido, la representación numérica binaria de tipo signo/exponente/mantisa, depende de la creatividad o precisión que desee el programador. Analicemos qué sucede con un sistema de numeración binario a $n > 0$ (entero) bits, para esto elige un valor par entre 10 y 20, crea el sistema de numeración entero similar al estudiado en clase, (la mantisa debe ser, al menos, la mitad del número elegido). Describe tu sistema de numeración, ¿Cuál es el menor y mayor de los números representados? Presenta un número que no se pueda representar en tu sistema seleccionado. Describe tu proceso. (4 pts) NOTA: punto extra si además se describe un sistema fraccionario que tenga sentido.

Demostración. Usaremos 16 bits en total. 1 para el signo, 3 para el exponente, 12 para la mantisa. Esto nos genera números en el rango

$$[-1 * 2^7 * (2^{12} - 1), 2^7 * (2^{12} - 1)] = [-524160, 524160]$$

Listing 1: Generación de números posibles con el sistema elegido. Fraccionario y entero.

```c
#include <stdio.h>
#define lsb 0.000244140625

void print_binary(FILE *file, int n) {
  for (int i = 11; i >= 0; i--) { // 6 bits are enough for numbers up
      to 63
    int bit = (n >> i) & 1;
    fprintf(file, "%d", bit);
  }
  fprintf(file, ",");
}

/*

16 bit machine:
1 bit for sign
3 bits for exponent
12 bits for mantissa

sgn * 2^exp * mantissa
max. number represented:
0111111111111111 = (2^(12) - 1) * 2^7 = 4095 * 2^7 = 524160

min. number represented:
111111111111111 = -524160


impossible to represent the number 2^12 + 1 = 4097

*/

/*

print_representation() writes to a csv file "binary.csv"
all possible number representations in decimal format.
we then confirm that 4097 is not present.
*/
void print_representation() {
```

```c
  int curr = 0x0;
  FILE *file;
  file = fopen("binary.csv", "w+");
  /*
    header for the csv, representing
    the exponent of 2 which the mantissa
    is being multiplied by
   */
  fprintf(file, "binary,
      2^{0},2^{1},2^{2},2^{3},2^{4},2^{5},2^{6},2^{7}\n");

  for (int i = 1; i < 1 << 12;
       i++) { // loop from i = 1 (we ignore 0), up to 2^12 - 1
    print_binary(file, i);
    for (int j = 0; j < 1 << 3; j++) { // loop from j = 0 up to 2^3 -
       1
      if (j == 7) {
        fprintf(file, "%d\n", i * (1 << j));
      } else {
        fprintf(file, "%d,", i * (1 << j));
      }
    }
  }
}

void print_fractions() {
  FILE *file;
  file = fopen("fractions.csv", "w+");
  fprintf(file, "binary,
      2^{0},2^{1},2^{2},2^{3},2^{4},2^{5},2^{6},2^{7}\n");
  int curr;
  double i;

  for (i = lsb, curr = 0; i <= 1; i += lsb, curr++) {
    print_binary(file, curr);
    for (int j = 0; j < 1 << 3; j++) {

      if (j == 7) {
        fprintf(file, "%f\n", i * (1 << j));
      } else {
        fprintf(file, "%f,", i * (1 << j));
      }
    }
  }
}

int main() {
  print_representation();
  print_fractions();
  return 0;
}
```

el código anterior genera un csv con todos los valores (positivos) posibles de la representación elegida. En la siguiente tabla, presentamos algunos valores. La columna representa el exponente por que se multiplica el binario.

binary	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7
000000000001	1	2	4	8	16	32	64	128
000000000010	2	4	8	16	32	64	128	256
000000000011	3	6	12	24	48	96	192	384
000000000100	4	8	16	32	64	128	256	512
000000000101	5	10	20	40	80	160	320	640
000000000110	6	12	24	48	96	192	384	768
000000000111	7	14	28	56	112	224	448	896
000000001000	8	16	32	64	128	256	512	1024
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
111111111111	4095	8190	16380	32760	65520	131040	262080	524160

Para el sistema fraccionario, definimos el bit implicito como

$$\begin{cases} \mathtt{b}_{imp} = 1 & \mathtt{if}\ 0 < \mathtt{exp} < 7 \\ \mathtt{b}_{imp} = 0 & \mathtt{if}\ \mathtt{exp} = 0\ \mathtt{and}\ \mathtt{matissa} \neq 0 \end{cases}$$

y

$$\begin{cases} \mathtt{exp} = \mathtt{exp} - 3 & \mathtt{if}\ 0 < \mathtt{exp} < 7 \\ \mathtt{exp} = 2 & \mathtt{if}\ \mathtt{exp} = 0 \end{cases}$$

de esta manera, obtenemos los valores

binary	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7
000000000000	0.000244	0.000488	0.000977	0.001953	0.003906	0.007812	0.015625	0.031250
000000000001	0.000488	0.000977	0.001953	0.003906	0.007812	0.015625	0.031250	0.062500
000000000010	0.000732	0.001465	0.002930	0.005859	0.011719	0.023437	0.046875	0.093750
000000000011	0.000977	0.001953	0.003906	0.007812	0.015625	0.031250	0.062500	0.125000
000000000100	0.001221	0.002441	0.004883	0.009766	0.019531	0.039062	0.078125	0.156250
000000000101	0.001465	0.002930	0.005859	0.011719	0.023437	0.046875	0.093750	0.187500
000000000110	0.001709	0.003418	0.006836	0.013672	0.027344	0.054687	0.109375	0.218750
000000000111	0.001953	0.003906	0.007812	0.015625	0.031250	0.062500	0.125000	0.250000
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
111111111111	1.000000	2.000000	4.000000	8.000000	15.000000	32.000000	64.000000	128.000000

$\square$

2. Consideremos la siguiente sucesión $\{x_n\}_{n\in\mathbb{N}} \subset \mathbb{R}$ definida de manera recurrente como: (4 pts)

$$x_2 = 2,$$
$$x_{n+1} = 2^{n-\frac{1}{2}}\sqrt{1-\sqrt{1-4^{1-n}x_n^2}}$$

   (a) Escribe un programa para encontrar los primeros n términos de la sucesión.
   (b) Aplica tu programa para encontrar los términos de la sucesión cuando

   $$n = 6, 7, 8, 9, 10, 12, 18, 20$$

   , ¿a qué valor tiende la sucesión? (ésta sucesión tiende a un valor distinto de cero).
   (c) Encuentra los términos 50 y 100, ¿qué sucede con la sucesión? Grafica los términos de la sucesión desde 2 hasta 100. Explica detalladamente el por qué del comportamiento que tiene la computadora al calcular estos términos.

Demostración. Primero mostraremos que utilizar directamente la secuencia resulta en los términos divergiendo a infinito debido a los errores de precisión de punto flotante. Podemos notar que la secuencia puede ser reescrita de tal manera que la cantidad de restas (2) se puede reducir a 1.

$$\begin{aligned}
x_{n+1} &= 2^{n-1/2}\sqrt{1-\sqrt{1-4^{1-n}x_n^2}} \\
&= 2^{n-1/2}\sqrt{1-\sqrt{1-4^{1-n}x_n^2}} \cdot \frac{\sqrt{1-\sqrt{1+4^{1-n}x_n^2}}}{\sqrt{1+\sqrt{1-4^{1-n}x_n^2}}} \\
&= 2^{n-1/2}\frac{\sqrt{1-(1-4^{1-n}x_n^2)}}{\sqrt{1-\sqrt{1+4^{1-n}x_n^2}}} \\
&= 2^{n-1/2}\frac{2^{1-n}x_n}{\sqrt{1-\sqrt{1+4^{1-n}x_n^2}}} \\
&= \frac{\sqrt{2}x_n}{\sqrt{1-\sqrt{1+4^{1-n}x_n^2}}}.
\end{aligned}$$

Listing 2: Calcular secuencia de números que se acerca a π

```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define SQRT2 1.4142135624

/*

this implementation rewrites the formula into

x_{n+1} = \frac{\sqrt{2}x_{n}}{\sqrt{1+\sqrt{1-4^{1-n}x^{2}_{n}}}}

although using division, this reduces the number of subtractions used,
which leads to less floating point precision errors
accumulating, resulting in the sequence staying at the
convergence point for longer.

 */

double find_x(int n) {
  FILE *file;
  file = fopen("p2.csv", "w+");
  double x = 2.0; // first term
  double four_pow_one_minus_n = 1.0; // the term 4^{1-n}, divided by 4
      each iteration
  for (int i = 3; i < n + 2; i++) {
    x = SQRT2 * x / (sqrt(1 + sqrt(1 - (four_pow_one_minus_n /= 4)* x
       * x)));
    fprintf(file, "x_%d, %.16lf\n", i, x);
  }
  return x;
}
```

```c
/*

Naive implementation: this implementation uses the sequence formula

x_{n+1} = 2^{n-1/2} \sqrt{1-\sqrt{1-4^{1-n}x_{n}^{2}}}

This leads to the sequence diverging due to floating point precision
at the term 29, which then spirals into infinity.

 */
double xn(int n) {
  double x = 2.0; // first term
  double two_pow_n_minus_one = SQRT2; // term 2^{n-1/2}, multiplied by
      2 each iteration
  double four_pow_one_minus_n = 1.0; // term 4^{1-n}, divided by 4
     each iteration
  FILE *file;
  file = fopen("naive_p2.csv", "w+");
  fprintf(file, "x_2, %.16lf\n", x);
  for (int i = 3; i < n + 2; i++) {
    x = (two_pow_n_minus_one *= 2) *
        sqrt(1 - sqrt(1 - (four_pow_one_minus_n /= 4) * x * x));
    // Note that the assignation of two_pow_n_minus_one is done before
        being used by x
    // Similarly with four_pow_one_minus_n
    fprintf(file, "x_%d, %.16lf\n", i, x);
  }
  return x;
}

int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "Invalid arguments. Usage: ./p2 (terms)");
    exit(1);
  }
  int terms = atoi(argv[1]);
  find_x(terms);
  xn(terms);
  return 0;
}
```

Figure 1: Secuencia usando algoritmo ingenuo

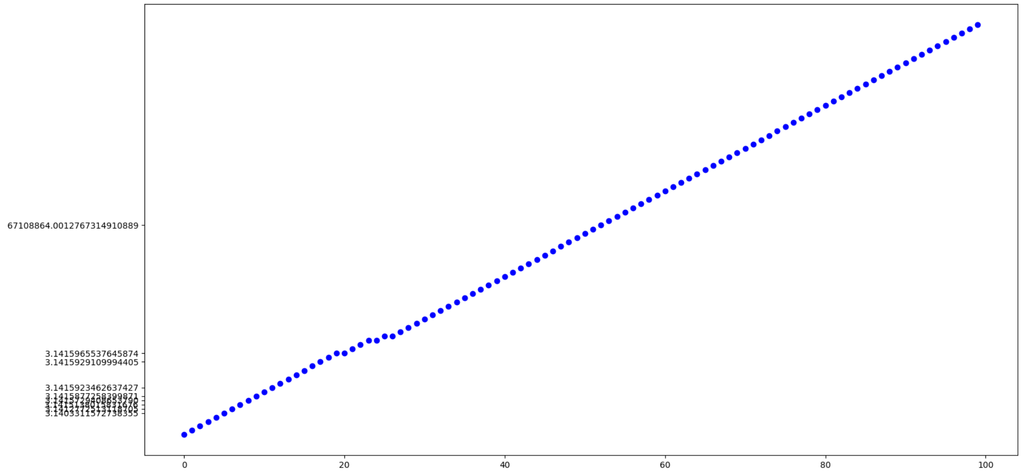
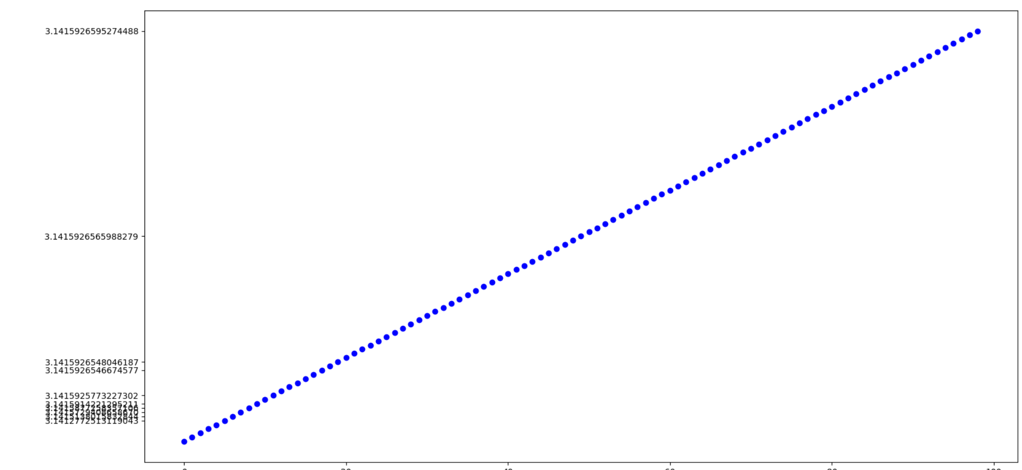


Figure 2: Secuencia usando algoritmo reduciendo restas



Como podemos observar, el segundo algoritmo (que reduce el numero de restas), se mantiene cerca de π, mientras que el ingenuo explota y diverge a infinito. $\square$

3. Al número representable inmediatamente después de la unidad se le conoce como el epsilón de la computadora machine el cual nos permite calcular el número real representable inmediatamente posterior; ésto se obtiene al multiplicar el epsilón por el real y sumarlo a ese real. Utilizando el siguiente algoritmo, crea un código que calcule el machine de tu computadora, sólo hace falta indicar el tipo de variable que es "epsilon" (float o doble). El último valor impreso corresponde al epsilon de la computadora. (2 pts)

Demostración. La lista impresa es

1.0000000000000000000000
0.5000000000000000000000
0.2500000000000000000000
0.1250000000000000000000
0.0625000000000000000000
0.0312500000000000000000
0.0156250000000000000000
0.0078125000000000000000
0.0039062500000000000000
0.0019531250000000000000
0.0009765625000000000000
0.0004882812500000000000
0.0002441406250000000000
0.0001220703125000000000
0.0000610351562500000000
0.0000305175781250000000
0.0000152587890625000000
0.0000076293945312500000
0.0000038146972656250000
0.0000019073486328125000
0.0000009536743164062500
0.0000004768371582031250
0.0000002384185791015625
0.0000001192092895507812
0.0000000596046447753906
0.0000000298023223876953
0.0000000149011611938477
0.0000000074505805969238
0.0000000037252902984619
0.0000000018626451492310
0.0000000009313225746155
0.0000000004656612873077
0.0000000002328306436539
0.0000000001164153218269
0.0000000000582076609135
0.0000000000291038304567
0.0000000000145519152284
0.0000000000072759576142
0.0000000000036379788071
0.0000000000018189894035
0.0000000000009094947018
0.0000000000004547473509
0.0000000000002273736754
0.0000000000001136868377
0.0000000000000568434189
0.0000000000000284217094
0.0000000000000142108547
0.0000000000000071054274
0.0000000000000035527137
0.0000000000000017763568
0.0000000000000008881784

$$0.0000000000000004440892$$
$$0.0000000000000002220446$$

Después de 52 iteraciones, obtenemos que

$$\epsilon = 0.0000000000000002220446 = 2.22046 \times 10^{-16} \approx 2^{-52}$$

.

Listing 3: Calcular epsilon de la máquina

```c
#include <stdio.h>

int main() {
  double epsilon = 1;
  while (1+epsilon > 1) {
    printf("%.22f\n", epsilon);
    epsilon /= 2;
  }
}
```

$\square$