# SIMULACIONES

**IV + IoT + DL**

**Industria 5.0**


Diego L. Aristizábal Ramírez
Profesor, Facultad de Ciencias, Departamento de Física
Universidad Nacional de Colombia
Medellín

# ARITMÉTICA COMPUTACIONAL

# 3

| Temas |
| --- |
| ✓ INTRODUCCIÓN |
| ✓ NÚMEROS MÁQUINA: NÚMEROS ENTEROS EN JAVA |
| ✓ NÚMEROS MÁQUINA: NUMEROS REALES EN JAVA TIPO float |
| ✓ NÚMEROS MÁQUINA: NUMEROS REALES EN JAVA TIPO double |
| ✓ EXACTITUD Y PRECISIÓN |
| ✓ ERRORES EN LA REALIZACIÓN DE OPERACIONES |
| ✓ MI SEGUNDA APP: ALGUNOS CÁLCULOS ARITMÉTICOS |
| ✓ REFERENCIAS |
| ✓ ANEXOS |

## INTRODUCCIÓN

La ciencia y la tecnología describen los fenómenos reales mediante modelos fisicomatemáticos. El estudio de estos modelos permite un conocimiento más profundo del fenómeno, así como de su evolución futura. La solución de estos modelos no siempre es posible lograrlo aplicando métodos analíticos clásicos por diferentes razones:

- ✓ No se adecuan al modelo concreto.
- ✓ Su aplicación resulta excesivamente compleja.
- ✓ La solución formal es tan complicada que hace imposible cualquier interpretación posterior.
- ✓ Simplemente no existen métodos analíticos capaces de proporcionar soluciones al problema.

En estos casos son útiles las técnicas numéricas, que mediante una labor de cálculo más o menos intensa, conducen a soluciones aproximadas que son siempre numéricas. El importante esfuerzo de cálculo que implica la mayoría de estos métodos hace que su uso esté íntimamente ligado al empleo de computadores.

## NÚMEROS MÁQUINA: NÚMEROS ENTEROS EN JAVA

Los computadores tienen gran potencia de cálculo, constituida por su gran capacidad de guardar información y la velocidad de cálculo. Sin embargo, sólo pueden representar un número finito de números, a los cuales se les denomina **números máquina**. La cantidad de estos dependerá del número de bits del procesador (32 bits, 64 bits, …), por lo que al realizar cálculos deberá estar redondeando los resultados a **números máquina**, así mismo deberá estar truncando las cifras significativas debido a la finitud de su memoria. Esto lo debe estar supervisando el usuario para no obtener resultados a veces fatales en sus cálculos.



### Números enteros en Java

El lenguaje empleado en este curso es Java, y su máquina virtual para representar los enteros emplea el sistema numérico con complemento a 2 (ver anexo 1), con el cual puede representar enteros en el rango $[-2^{n-1}, 2^{n-1}-1]$, donde n es el número de bits disponibles. Un bit se utiliza para representar el signo y los n - 1 bits restantes representan la magnitud. Si el número a representar se sale de ese rango, aparece lo que se denomina overflow (desbordamiento), Figura 1.

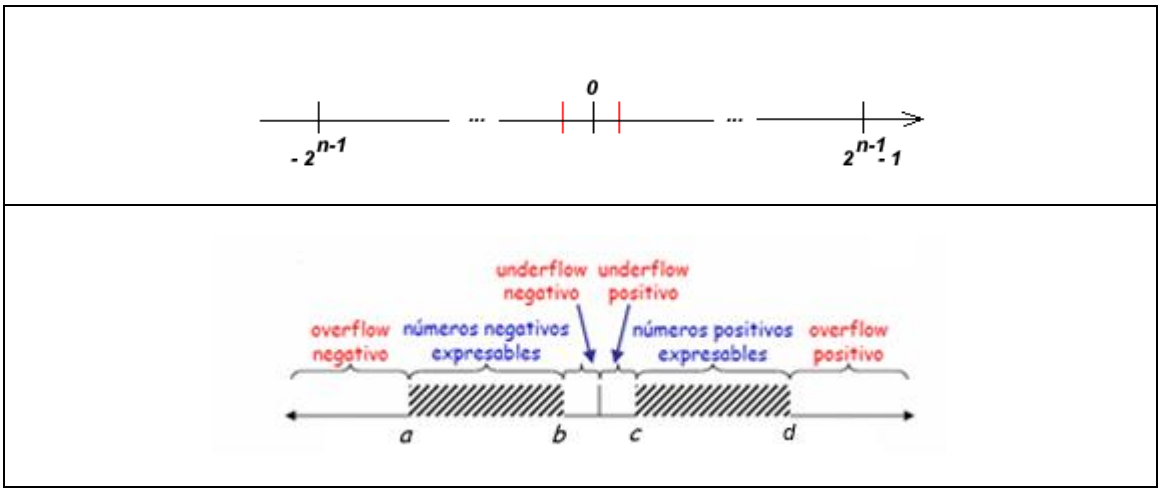


**Figura 1**

Java cuenta con cuatro tipos de números enteros: **`byte`**, **`short`**, **`int`**, **`long`**, para los cuales se emplean respectivamente para representarlos, ocho bits (1 byte), 16 bits (2 bytes), 32 bits (4 bytes), 64 bits (8 bytes).

El cuadro de la Tabla 1, da información respecto a estos enteros con base en la recta numérica ilustrada en la Figura 1.

**Tabla 1: Tipos enteros en Java**

| Tipo | n (bits) | bytes | Cantidad de Números | a = - $2^{n-1}$ | b | c | d = $2^{n-1}$ - 1 |
|---|---|---|---|---|---|---|---|
| byte | 8 | 1 | $2^8$ | -128 | -1 | +1 | +127 |
| short | 16 | 2 | $2^{16}$ | -32768 | -1 | +1 | +32767 |
| int | 32 | 4 | $2^{32}$ | -2147483648 | -1 | +1 | +2147483647 |
| long | 64 | 8 | $2^{64}$ | -9223372036854775808 | -1 | +1 | +9223372036854775807 |



Son características de la representación de los enteros las siguientes:

- ✓ No presentan underflow. Es decir, no hay números alrededor de CERO que no se puedan representar con el computador.
- ✓ Presentan overflow (desbordamiento) positivo y negativo, Figura 1.
- ✓ Densidad constante. Es decir, en segmentos iguales de la recta numérica hay la misma cantidad de números enteros

## NUMEROS MÁQUINA: NUMEROS REALES EN JAVA TIPO `float`

Java usa dos números de punto flotante (ver anexo 2 para más detalles): precisión simple (`float`) y precisión doble (`double`).

Los `float`, se representan siguiendo la distribución de bits para el signo, el exponente y la mantisa que se ilustra en la Figura 2. Se emplean 32 bits (precisión simple): 1 bit para el signo, 8 bits para el exponente y 23 bits para la mantisa.

Los 8 bits disponibles para almacenar el exponente, en principio, daría un rango para E comprendido entre 0 y 255 (es decir $2^8$=256 posibilidades). Sin embargo, El 0 y el 255 se deben descartar, ya que correspondería a que todos los bits del exponente estuvieran en ceros o todos en unos, los cuales están reservados en la norma IEEE754 a representaciones especiales. Los valores normalizados de los números en representación en punto flotante para la norma IEEE754 se deben escribir así:

$$N = \pm(1 + M)\cdot 2^{E - 127}$$

tal que $0 < E < 255$, $2^8$ - 2 posibilidades. Así que, E se mueve en el intervalo 1 a 254 (ambos inclusive), y al restarle 127 queda el exponente dentro de del rango entre $2^{-126}$ y $2^{127}$. Con base en esto se calculará a continuación los números máquina (valores representables en el

computador) máximo y mínimo con la norma IEEE 754 y del tipo `float` de Java. Es decir se hallarán los valores a,b,c y d ilustrados en la Figura 2.

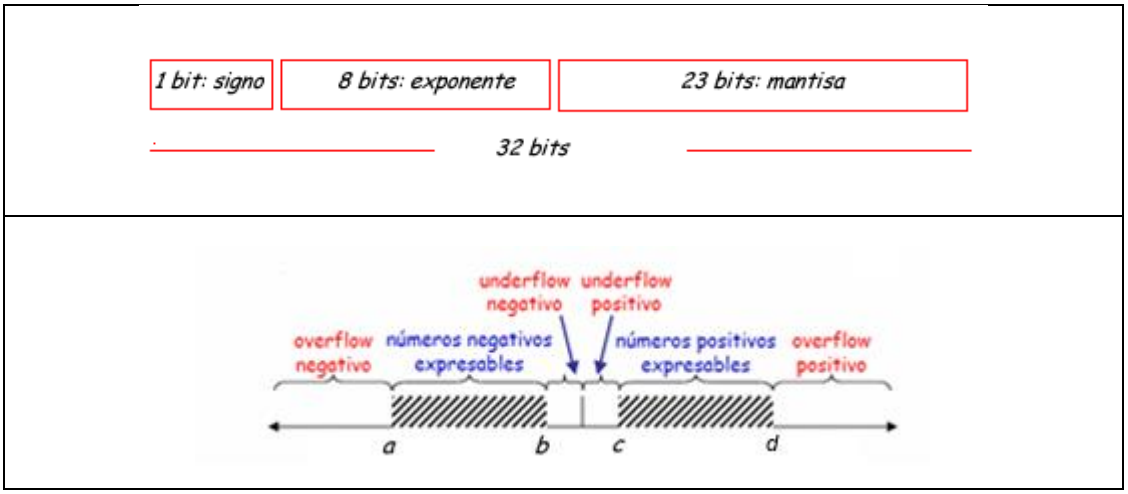


**Figura 2**

En primera instancia las mayores y menores mantisas, y los mayores y menores exponentes son:

- Mayor mantisa (23 bits en 1):

$$M = 11111111111111111111111 = 2^{-1} + 2^{-2} + \ldots + 2^{-23} = 1 - 2^{-23}$$

- Menor mantisa (no cero):

$$M = 00000000000000000000001 = 2^{-23}$$

- Mayor exponente (No todos los bits en 1):

$$E = 11111110_2 = 254_{10}$$

$$2^{E-C} = 2^{254-127} = 2^{127}$$

- Menor exponente ( No todos los bits en cero):

$$E = 00000001_2 = 1_{10}$$

$$2^{E-C} = 2^{1-127} = 2^{-126}$$

Por lo tanto,

- $a = -(1 + M) \bullet 2^{127} = -(1 + 1 - 2^{-23}) \bullet 2^{127} = -(2 - 2^{-23}) \bullet 2^{127} = -3.40282347 \text{ x } 10^{+38}$
- $b = -(1 + 2^{-23}) \bullet 2^{-126} = -(1 + 2^{-23}) \bullet 2^{-126} = -1.1754945 \text{ x } 10^{-38}$
- $c = +(1 + 2^{-23}) \bullet 2^{-126} = +(1 + 2^{-23}) \bullet 2^{-126} = +1.1754945 \text{ x } 10^{-38}$
- $d = +(1 + M) \bullet 2^{127} = +(1 + 1 - 2^{-23}) \bullet 2^{127} = +(2 - 2^{-23}) \bullet 2^{127} = +3.40282347 \text{ x } 10^{+38}$

*Observación*: En los textos de programación en Java reportan los valores de $b$ y $c$ sin normalizar la mantisa:



$b = -(2^{-23}) \bullet 2^{-126} = -2^{-149} = -1.4012984 \text{ x } 10^{-45}$ (Para calcular **underflow**).

$c = +(2^{-23}) \bullet 2^{-126} = 2^{-149} = 1.4012984 \text{ x } 10^{-45}$ (Para calcular **underflow**).

Es necesario anotar que la representación de los números en precisión simple (`float`):

- Presentan *underflow*.
- Presentan *overflow*.
- Mayor densidad de números cerca del cero.
- Cantidad de números representables: $2^{32}$
- Épsilon (por redondeo)[1]:

$$\text{eps} = 0.5 \text{ x } \left[2 \text{ x } 2^{-\text{numero\_bits\_mantisa}}\right] = 0.5 \text{ x } 2^{1-\text{numero\_bits\_mantisa}} = 0.5 \text{ x } 2^{1-23}$$

$$\text{eps} = 1{,}1920929 \; x \; 10^{-7}$$

- Número de cifras decimales[2] : 6 o 7, ya que $2^{-23} \sim 10^{-7}$

## NÚMEROS MÁQUINA: NÚMEROS REALES EN JAVA TIPO `double`

La representación sigue la distribución de bits para el signo, el exponente y la mantisa que se ilustra en la Figura 3. Se emplean 64 bits (precisión doble): 1 bit para el signo, 11 bits para el exponente y 52 bits para la mantisa.

---

[1] El épsilon de una máquina (computador) es el número decimal más pequeño que, sumado a 1, la máquina da como resultado un valor diferente de 1, es decir, que no es redondeado.

[2] Emplear $\log_2(2^{-23}) = \log_2(10)^x$ y la fórmula de cambio de base en logaritmos $\log_a x = \frac{\log_b x}{\log_b a}$

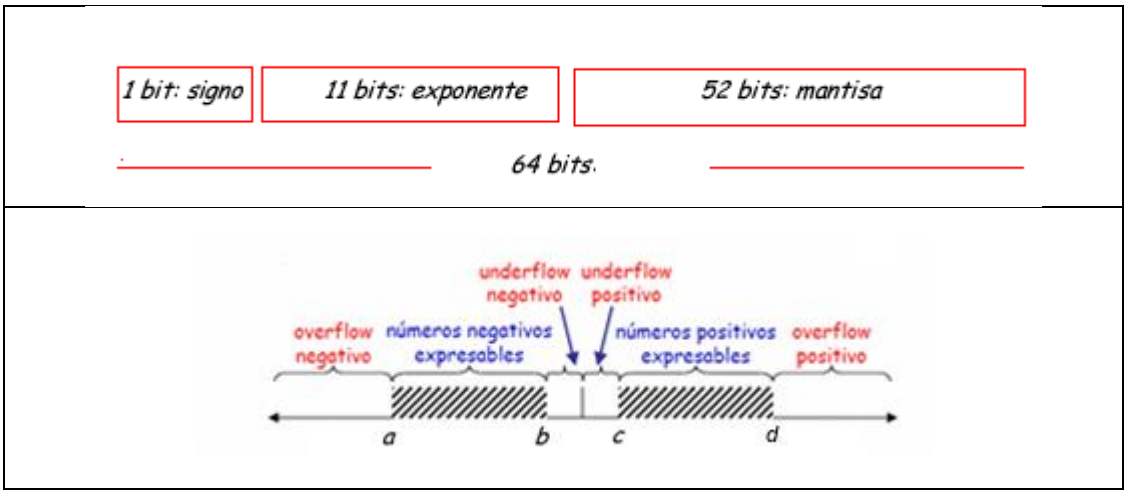




**Figura 3**

Los 11 bits disponibles para almacenar el exponente, en principio, daría un rango para E comprendido entre 0 y 2047 (es decir $2^{11}$=2048 posibilidades). Sin embargo, El 0 y el 2047 se deben descartar, ya que correspondería a que todos los bits del exponente estuvieran en ceros o todos en unos, los cuales están reservados en la norma IEEE754 a representaciones especiales. Por lo valores normalizados de los números en representación en punto flotante para la norma IEEE754 se deben escribir así:

$$N = \pm(1 + M)\cdot 2^{E - 1023}$$

tal que $0 < E < 2047$, $2^8$ - 2 posibilidades. Así que, E se mueve en el intervalo 1 a 2046 (ambos inclusive), y al restarle 1023 queda el exponente dentro de del rango entre $2^{-1022}$ y $2^{1023}$. Análogamente a la representación de los números máquina para precisión simple se puede proceder para precisión doble y así obtener los valores a, b,c y d ilustrados en la Figura 3, obteniéndose:

- $a = -(1 + M) \bullet 2^{1023} = -(1 + 1 - 2^{-52}) \bullet 2^{1023} = -(2 - 2^{-52}) \bullet 2^{1023}$

  $a = -1.7976931348623157 \text{ x } 10^{+308}$
- $b = -(1 + 2^{-52}) \bullet 2^{-1022} = -(1 + 2^{-52}) \bullet 2^{-1022}$

  $b = -2.225073858507202 \text{ x } 10^{-308}$
- $c = +(1 + 2^{-52}) \bullet 2^{-1022} = +(1 + 2^{-52}) \bullet 2^{-1022}$

  $c = 2.225073858507202 \text{ x } 10^{-308}$
- $d = (1 + M) \bullet 2^{1023} = (1 + 1 - 2^{-52}) \bullet 2^{1023} = (2 - 2^{-52}) \bullet 2^{1023}$

  $d = +1.7976931348623157 \text{ x } 10^{+308}$

*Observación*: En los textos de programación en Java reportan los valores de $b$ y $c$ sin normalizar la mantisa:

$b = -(0 + 2^{-52}) \bullet 2^{-1022} = -4.9406556458412465 \text{ x } 10^{-324}$ (Para calcular **underflow**).

$c = (0 + 2^{-52}) \bullet 2^{-1022} = 4.9406556458412465 \text{ x } 10^{-324}$ (Para calcular **underflow**).

Es necesario anotar que la representación de los números en precisión doble (double):

- Presentan *underflow*.
- Presentan *overflow* .
- Mayor densidad de números cerca del cero.
- Cantidad de números representables: $2^{64}$.
- Épsilon (por redondeo)

$$\text{eps} = 0.5 \text{ x } [2\text{x } 2^{-\text{numero\_bits\_mantisa}}] = 0.5 \text{ x } 2^{1-\text{numero\_bits\_mantisa}} = 0.5 \text{ x } 2^{1-52}$$

$$\text{eps} = 2{,}2204046 \; x \; 10^{-16}$$

- Número de cifras decimales[3]: 16 o 17 ya que $2^{-52} \sim 10^{-17}$

**Densidad de los números máquina**

Es interesante notar una propiedad de estos números de especial importancia en los cálculos numéricos y que hace referencia a su densidad en la recta de los números reales. Supóngase que el número de bits de la mantisa es 24. En el intervalo $[1,2)$ es posible representar $2^{24}$ números igualmente espaciados y separados por una distancia $\frac{1}{2^{24}}$. De modo análogo, en cualquier intervalo $[2^f, 2^{f+1})$ hay $2^{24}$ números equiespaciados[4], pero su densidad en este caso es $\frac{2^f}{2^{24}}$ Por ejemplo, entre $2^{20} = 1048576$ y $2^{21} = 2097152$ hay $2^{24} = 16777216$ números, pero el espaciado entre dos números sucesivos es de sólo $\frac{1}{16}$.

## EXACTITUD Y PRECISIÓN

Para los valores en punto flotante se define:

[3] Emplear $\log_2(2^{-52}) = \log_2(10)^x$ y la fórmula de cambio de base en logaritmos $\log_a x = \frac{\log_b x}{\log_b a}$

[4] En el intervalo [1,2), f = 0

- ✓ **Precisión**: representa la cantidad de dígitos significativos que tiene su mantisa ("*significand*"). En Java los valores `double` son más precisos que los `float`.
- ✓ **Exactitud**: representa qué tan próximo se encuentra un valor en punto flotante, respecto de su valor correcto.

Ejemplo

1/3= 0.3333333f (más exacto)

1/3=0.34444444444444444d (más preciso)



Los errores asociados con los cálculos y medidas se pueden caracterizar observando su exactitud y precisión. La *exactitud* se refiere a qué tan cerca o no del valor verdadero está el valor calculado o el medido. La *precisión* se refiere a qué tan cerca está un valor individual medido o calculado con respecto a los otros (repetibilidad).
Los métodos numéricos deben ser lo suficientemente exactos para que cumplan los requisitos de un problema particular de ingeniería. También deben ser lo suficientemente precisos para el diseño en la ingeniería

## ERRORES EN LA REALIZACIÓN DE OPERACIONES

Estos errores son debidos a:

- ➢ Acumulación de errores de redondeo
- ➢ Pérdida de dígitos significativos.

**Acumulación de errores de redondeo**

Sean x e y dos números máquina, y $\otimes$ el indicador de cualesquiera de las cuatro operaciones aritméticas básicas, $+,\ -,\ \times,\ \div$ Se tiene que,

$$(\text{float})(x \otimes y) = [x \otimes y]\ (1+\delta)\ \ ;\ |\delta| \leq \text{eps}$$

En el caso de operaciones compuestas

$$\begin{aligned}(\text{float})[x\ (y+z)] &= [x\ (\text{float})(y+z)](1+\delta_1) && |\delta_1|<2^{-22}\\ &= [x\ (y+z)](1+\delta_2)(1+\delta_1) && |\delta_2|<2^{-22}\\ &= x\ (y+z)(1+\delta_3) && |\delta_3|<2^{-21}\end{aligned}$$

El error de redondeo se multiplica por 2 (la precisión disminuye). Y aunque el error de una operación puede ser pequeño, el error puede aumentar si se encadenan operaciones.

**Pérdida de digitos significativos**

Al trabajar con operaciones matemáticas se debe evitar las sustracciones de cantidades casi iguales, ya que esta es una fuente de pérdida de precisión.

Por ejemplo, la resta de $x = 0.3721478693$, $y = 0.3720230572$ y se posee una máquina hipotética que en punto flotante despliegue sólo hasta 5 decimales, ¿cuál es el error relativo?



$$(\text{float})(x) = 0.37215$$

$$(\text{float})(y) = 0.37202$$

$$(\text{float})(x) - (\text{float})(y) = 0.00013$$

$$\left|\frac{(x - y) \; - \; [(\text{float})(x) - (\text{float})(y)]}{x \; - \; y}\right| \cong 4\%$$

En muchos casos estos errores se pueden evitar, como en la evaluación de la siguiente expresión, la cual implica una sustracción y las pérdidas de cifras significativas si $x \to 0$ :

$$y = \sqrt{x^2 + 1} - 1$$

La pérdida de precisión se puede evitar haciendo el siguiente artificio,

$$y = \left(\sqrt{x^2 + 1} - 1\right)\left(\frac{\sqrt{x^2 + 1} + 1}{\sqrt{x^2 + 1} + 1}\right) = \frac{x^2}{\sqrt{x^2 + 1} + 1}$$

## MI SEGUNDA APP: ALGUNOS CÁLCULOS ARITMÉTICOS

Usando el siguiente código crear una aplicación denominada **MiSegundaApp**. Seguir los mismos pasos de la generación de **MiPrimeraApp** del módulo # 2.

NOTA: Cuando pega el código desde este documento a **Android Studio** éste puede quedar muy desorganizado. Para arreglar esto, seleccionar toda la clase y hacer **CTRL + ALT + L** y listo.

```
public class ActividadPrincipalMiSegundaApp extends Activity {

    private TextView resultados;
    private String cadena;


    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);


        /*llamada al método para crear los elementos de la interfaz gráfica de usuario
(GUI)*/
        crearElementosGui();


        /*para informar cómo se debe adaptar la GUI a la pantalla del dispositivo*/
        ViewGroup.LayoutParams parametro_layout_principal = new
ViewGroup.LayoutParams(ViewGroup.LayoutParams.MATCH_PARENT,
ViewGroup.LayoutParams.MATCH_PARENT);


        /*pegar al contenedor la GUI: en el argumento se está llamando al método
crearGui()*/
        this.setContentView(crearGui(), parametro_layout_principal);


       /*llamada al metodo para calcular overflow de byte*/
        calculoOverflowByte();

        /*llamada al metodo para calcular overflow de short*/
        calculoOverflowShort();


        /*llamada al metodo para calcular overflow de int*/
        calculoOverflowInt();

        /*llamada al metodo para calcular overflow de long*/
        calculoOverflowLong();

        /*llamada al metodo para calcular overflow de float*/
        calculoOverflowFloat();

        /*llamada al metodo para calcular overflow de double*/
        calculoOverflowDouble();

        /*llamada al metodo para calcular underflow de float*/
        calculoUnderflowFloat();

        /*llamada al metodo para calcular underflow de double*/
        calculoUnderflowDouble();
```

```
        /*llamada al metodo para calcular epsilón máquina en float*/
        calculoEpsilonFloat();

        /*llamada al metodo para calcular epsilón máquina en double*/
        calculoEpsilonDouble();

        /*llamar el método para desplegar el resultado*/
        desplegarResultado();



        //forzar pantalla landscape
        this.setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);

    }

    //crear los objetos de la interfaz gráfica de usuario (GUI)
    private void crearElementosGui(){

        resultados=new TextView(this);
        resultados.setTextSize(TypedValue.COMPLEX_UNIT_SP, 12);
        resultados.setTextColor(Color.YELLOW);
        resultados.setBackgroundColor(Color.BLACK);

    }


    //organizar la distribución de los objetos de de la GUI usando administradores de
diseño
    private LinearLayout crearGui(){

        //administrador de diseño
        LinearLayout linear_principal = new LinearLayout(this);
        linear_principal.setOrientation(LinearLayout.VERTICAL);
        linear_principal.setGravity(Gravity.CENTER_HORIZONTAL);
        linear_principal.setGravity(Gravity.FILL);
        linear_principal.setBackgroundColor(Color.rgb(250,150,50));


        /*pegar el objeto cadena (es tipo TextView)*/
        //parámetro de pegada
        LinearLayout.LayoutParams parametrosPegada= new
LinearLayout.LayoutParams(android.view.ViewGroup.LayoutParams.MATCH_PARENT,0);
        parametrosPegada.setMargins(50, 50, 50, 50);
        parametrosPegada.weight = 1.0f;
        //pegar
        linear_principal.addView(resultados, parametrosPegada);


        return linear_principal;
```

```
}


/*Método para mostrar overflow en bytes*/
private void  calculoOverflowByte(){

    byte x=127;
    byte y=1;
    byte z=(byte)(x+y); //se hizo casting

    //cadena resultado
    String resultadoOverflowByte="   Overflow en byte: "+x +" + "+ y + "= "+z;
    cadena= "   RESULTADOS\n"+resultadoOverflowByte;

}

/*Método para mostrar overflow en short*/
private void  calculoOverflowShort(){

    short x=32767;
    short y=1;
    short z=(short)(x+y); //se hizo casting

    //cadena resultado
    String resultadoOverflowShort="   Overflow en short: "+x +" + "+ y + "= "+z;
    cadena= cadena+"\n"+resultadoOverflowShort;

}


/*Método para mostrar overflow en int*/
private void  calculoOverflowInt(){

    int x=2147483647;
    int y=1;
    int z=x+y; //se hizo casting

    //cadena resultado
    String resultadoOverflowInt="   Overflow en int: "+x +" + "+ y + "= "+z;
    cadena= cadena+"\n"+resultadoOverflowInt;

}

/*Método para mostrar overflow en long*/
private void  calculoOverflowLong(){

    long x=(long)(Math.pow(2,63)-1);//se hizo casting
    long y=1;
    long z=x+y;

    //cadena resultado
```

```
        String resultadoOverflowLong="   Overflow en long: "+x +" + "+ y + "= "+z;
        cadena= cadena+"\n"+resultadoOverflowLong;

    }

    /*Método para mostrar overflow en float*/
    private void  calculoOverflowFloat(){

        float x=(float)(3.40282347*Math.pow(10,38));//se hizo casting
        //aguanta hasta 7 decimales
        float y=(float)(0.0000001*x);//se hizo casting
        float z=x+y;

        //cadena resultado
        String resultadoOverflowFloat="   Overflow en float: "+x + " + "+y + "= "+z;
        cadena= cadena+"\n"+resultadoOverflowFloat;

    }

    /*Método para mostrar overflow en double*/
    private void  calculoOverflowDouble(){

        double x=1.7976931348623157*Math.pow(10,308);//se hizo casting
        //aguanta hasta 16 decimales
        double y=0.0000000000000001*x;
        double z=x+y;

        //cadena resultado
        String resultadoOverflowDouble="   Overflow en double: "+x + " + "+y + "= "+z;
        cadena= cadena+"\n"+resultadoOverflowDouble;

           }

    /*Método para mostrar underflow en float*/
    private void  calculoUnderflowFloat(){

        float x=(float)Math.pow(2,-149);
        float y=2f;
        float z=x/y;

        //cadena resultado
        String resultadoUnderflowFloat="   Underflow en float: "+x + " / "+y + "= "+z;
        cadena= cadena+"\n"+resultadoUnderflowFloat;

    }

    /*Método para mostrar underflow en double*/
    private void  calculoUnderflowDouble(){

        double x=Math.pow(2,-1074);
        double y=2d;
```

```
        double z=x/y;

        //cadena resultado
        String resultadoUnderflowDouble="   Underflow en double: "+x + " / "+y + "= "+z;
        cadena= cadena+"\n"+resultadoUnderflowDouble;

    }


    /*calcula el epsilon para float*/
    private void calculoEpsilonFloat(){

        //Cálculo teórico del épsilon
        float epsilonTeoricoFloat=0.5f*(float)(Math.pow(2,1-23));//eps por redondeo

        //Cálculo computacional del épsilon
        float epsilonCalculadoFloat=1f;

        do {
            epsilonCalculadoFloat=epsilonCalculadoFloat/2;

        } while(epsilonCalculadoFloat+1.0f>1.0f);

        /* Se debe multiplicar por 2 por que el epsilon se calcula con el primer
        suceso que cumpla que epsilon+1.0=1.0 y el algoritmo  lo sobrepasa.*/

        epsilonCalculadoFloat=2*epsilonCalculadoFloat;


        cadena= cadena+"\n"+"   Epsilon float teorico  "+epsilonTeoricoFloat+ ",
Epsilon float computacional  "+ epsilonCalculadoFloat;

    }


    /*calcula el epsilon para double*/
    private void calculoEpsilonDouble(){

        //Cálculo teórico del épsilon
        float epsilonTeoricoDouble=0.5f*(float)(Math.pow(2,1-52));//eps por redondeo

        //Cálculo computacional del épsilon
        float epsilonCalculadoDouble=1f;

        do {
            epsilonCalculadoDouble=epsilonCalculadoDouble/2;

        } while(epsilonCalculadoDouble+1.0d>1.0d);

        /* Se debe multiplicar por 2 por que el epsilon se calcula con el primer
        suceso que cumpla que epsilon+1.0=1.0 y el algoritmo  lo sobrepasa.*/
```

```
        epsilonCalculadoDouble=2*epsilonCalculadoDouble;


        cadena= cadena+"\n"+"   Epsilon double teorico  "+epsilonTeoricoDouble+ ",
Epsilon double computacional  "+ epsilonCalculadoDouble;

    }


    private void desplegarResultado(){

        resultados.setText(cadena);
    }

}
```



Pasar a construir el archivo manifiesto. Hacer clic en el archivo **app.manifests> AndroidManifest.xml**. En el código desplegado agregar el código de tal forma que quede como el que se presenta a continuación.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.curso_simulaciones.misegundaapp">

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.MiSegundaApp"
        tools:targetApi="31" >

        <activity
android:name="com.curso_simulaciones.misegundaapp.ActividadPrincipalMiSegundaApp"
            android:exported="true">

            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

    </application>

</manifest>
```

Su compilación y ejecución debe desplegar la pantalla ilustrada en la Figura 4.

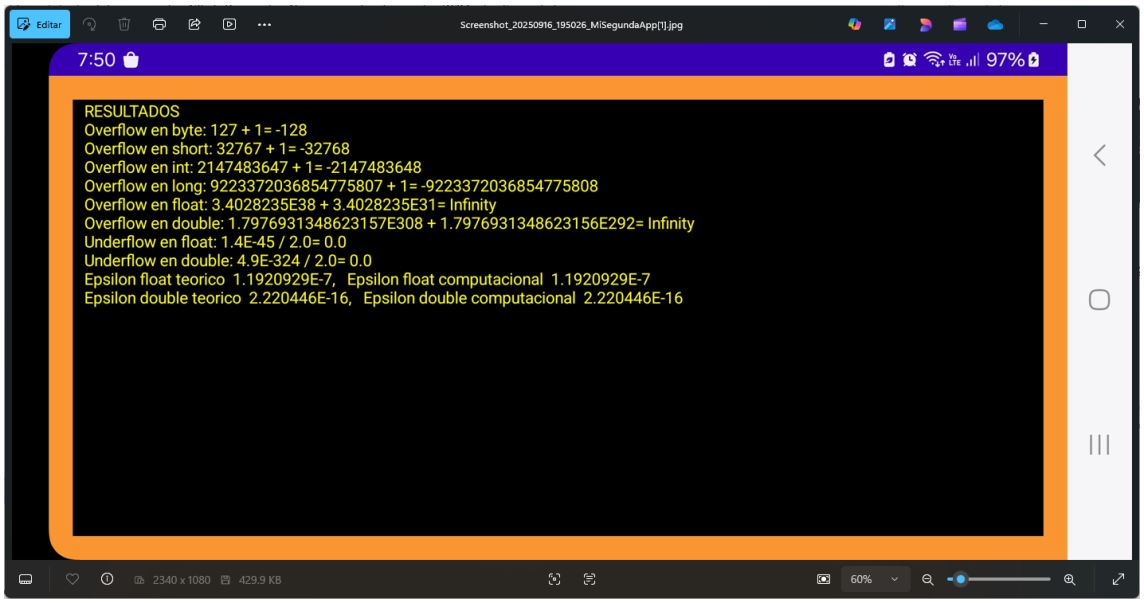
Editar
Screenshot_20250916_195026_MiSegundaApp[1].jpg
7:50
97%
RESULTADOS
Overflow en byte: 127 + 1= -128
Overflow en short: 32767 + 1= -32768
Overflow en int: 2147483647 + 1= -2147483648
Overflow en long: 9223372036854775807 + 1= -9223372036854775808
Overflow en float: 3.4028235E38 + 3.4028235E31= Infinity
Overflow en double: 1.7976931348623157E308 + 1.7976931348623156E292= Infinity
Underflow en float: 1.4E-45 / 2.0= 0.0
Underflow en double: 4.9E-324 / 2.0= 0.0
Epsilon float teorico 1.1920929E-7, Epsilon float computacional 1.1920929E-7
Epsilon double teorico 2.220446E-16, Epsilon double computacional 2.220446E-16
2340 x 1080
429.9 KB
60%

**Figura 4**

## REFERENCIAS


- ✓ Franco G. A (2000)., **Curso de Lenguaje Java, Universidad** del País Vasco. http://www.sc.ehu.es/sbweb/fisica/curso.htm.
  Consultada [Enero de 2023].

# ANEXO 1

## REPRESENTACIÓN DE NÚMEROS ENTEROS

Tomando los dígitos 0 y 1 como unidad mínima de memoria y que se denomina **bit** (1 byte equivale a 8 bits. El kilobyte equivale a 1024 bytes.), se tendrá que con $n$ bits se pueden representar $2^n$ enteros. El rango de estos dependerá del método que se opte para representarlos: binario puro sin signo, binario puro con signo, complemento a 2 o por exceso Z.



Complemento a 2

El lenguaje empleado en este curso es Java, y su máquina virtual para representar los enteros emplea el sistema numérico con complemento a 2, con el cual puede representar enteros en el rango $\left[-2^{n-1}, 2^{n-1}-1\right]$, donde $n$ es el número de bits disponibles. Un bit se utiliza para representar el signo y los $n-1$ bits restantes representan la magnitud. Si el número a representar se sale de ese rango, aparece lo que se denomina overflow (desbordamiento).

¿Qué es el complemento a dos?

El **complemento a dos** de un número $N$ que, expresado en el sistema binario está compuesto por $n$ dígitos, se define como:

$$C_2^N = 2^N - N$$

Por ejemplo el número, $N = 45$ expresado en binario es $N = 101101_2$, tiene 6 dígitos, $n = 6$, por lo tanto su complemento a dos será:

$$C_2^{45} = 2^6 - 45 = 64 - 45 = 19_{10} = 010011_2$$

Puede parecer engorroso, pero es muy fácil obtener el complemento a dos de un número a partir de su complemento a uno, debido a que el complemento a dos de un número binario es una unidad mayor que su complemento a uno, es decir:

$$C_2^N = C_1^N + 1$$

¿Para qué sirve el complemento a 2?

Su utilidad principal se encuentra en las operaciones matemáticas con números binarios. En particular, la resta de números binarios se facilita enormemente utilizando el complemento a dos: la resta de dos números binarios puede obtenerse sumando al minuendo el complemento a dos del sustraendo .

# ANEXO 2
# REPRESENTACIÓN DE NÚMEROS REALES

## Representación de los números reales

La limitación más significativa se encuentra en el almacenaje y manipulación de cantidades fraccionarias. Este será el tema de esta sección.

Notación científica normalizada



Todo número decimal $x$ se puede representar en notación científica como:

$$x = \pm r \times 10^{n}$$

donde $\frac{1}{10} < r < 1$ y $n$ un entero. Como ejemplo,

$$952.5054 = 0.9525054 \times 10^{3}$$

$$-0.006614 = -0.6614 \times 10^{-2}$$

De la misma forma se puede utilizar la notación científica en el caso binario:

$$x = \pm q \times 2^{m}$$

donde $\frac{1}{2} < q < 1$ (si $x \neq 0$), y $m$ un entero. El número $q$ se llama mantisa y el entero $m$ exponente, los cuales se representarán en binarios.

**Representación de los números reales con Java**

Java para representar los números reales emplea la notación de punto flotante bajo la norma IEEE 754, en la cual un número de punto flotante se representa cumpliendo:

- Signo explícito
- Representación del exponente en exceso.
- Mantisa normalizada con un 1 implícito (1.M)
- Son representaciones especiales las siguientes:
  - `0 00000000 00000000000000000000000 = +0`
  - `1 00000000 00000000000000000000000 = -0`

- 0 11111111 00000000000000000000000 = +Infinito
- 1 11111111 00000000000000000000000 = -Infinito

Resumiendo: los números reales en punto flotante para la norma IEEE 754 se representan así:

$$N = (\text{signo})(1.d_1d_2\ldots d_k) \bullet 2^{E-C}$$



donde $d_1d_2...d_k$ es la mantisa ($M$), $E$ recibe el nombre de *exponente almacenado*, $k$ es un número fijo que da la precisión (el número de dígitos de la mantisa). El número $C$ es el *desplazamiento* y sirve para que se puedan representar exponentes con valores positivos y negativos. La diferencia $E - C$ es el exponente. La parte entera vale 1 y no es necesaria almacenarla. Es de hecho el primer dígito significativo del número, que forzosamente debe valer 1. La representación anterior es equivalente a la siguiente:

$$N = \pm(1 + M) \bullet 2^{E-C}$$

En la Tabla 1 se ilustran las precisiones para números de punto flotante que permite la norma IEEE 754.

**Tabla 1**

| Precisión | # bits | # bits Exponente | # bits Mantisa |
|---|---|---|---|
| Simple | 32 | 8 | 23 |
| Simple extendida | ≥ 43 | ≥ 11 | ≥ 31 |
| Doble | 64 | 11 | 52 |
| Doble extendida | ≥ 80 | ≥ 15 | ≥ 63 |
| Cuádruple | 128 | 15 | 112 |

**Números máquina aproximados**Se trata de estimar el error en que se incurre al aproximar un número real positivo $x$ mediante un número de máquina. Si se representa el número:

$$x = (a_1 a_2 ... a_k a_{k+1} a_{k+2} ...)_2 \ x \ 2^m$$

en donde cada $a_i$ es 0 ó 1 y el bit principal es $a_1 = 1$. Un **número de máquina** se puede obtener de dos formas:

- ✓ **Truncamiento**: descartando todos los bits excedentes $a_{k+1} a_{k+2} \ldots$. El número resultante, $x'$ es siempre menor que $x$ (se encuentra a la izquierda de $x$ en la recta real).



$$x' = (a_1 a_2 ... a_k)_2 \ x \ 2^m \quad ; \quad x' < x$$

- ✓ **Redondeo por exceso**: aumentando en una unidad el último bit remanente $a_k$ y después se elimina el exceso de bits como en el caso anterior.

$$x = \left[ (a_1 a_2 ... a_k)_2 + 2^{-k} \right] x \ 2^m \quad ; \quad x'' > x$$

En ambos casos el error relativo está acotado,

$$\left| \frac{x - x^*}{x} \right| \leq 2^{-k}$$

donde $x^*$ es el número máquina más cercano a $x$ (puede ser $x'$ o $x''$. Si se define

$$\delta = \frac{x - x^*}{x}$$

se podrá escribir,

$$x^* = x (1+\delta)$$

con $|\delta| < 2^{-k}$.

**Épsilon de la máquina (eps)**

Si se denota a $f_L(x)$ como el número máquina punto flotante $x^*$ más cercano a $x$, se tendrá,

$$f_L(x) = x\,(1+\delta), \quad |\delta| \leq \mathbf{eps}$$

donde **eps** = $\frac{2^{1-n}}{2}$ en caso de redondeo, y **eps** = $2^{1-n}$ en el caso de truncamiento, siendo n el número de bits empleado para almacenar la mantisa. El número **eps** representa el error de redondeo unitario y depende de la arquitectura del computador.