# Ejercicio Numero 1

a. El algoritmo quadratic1.c computa las raíces de esta ecuación empleando los tipos de datos float y double. Compile y ejecute el código. ¿Qué diferencia nota en el resultado?

La diferencia se encuentra en la precision del resultado, el double es un tipo de dato con mayor bits de precision que el float.

b. El algoritmo quadratic2.c computa las raíces de esta ecuación, pero en forma repetida. Compile y ejecute el código variando la constante TIMES. ¿Qué diferencia nota en la ejecución?

```
QUAD2 TIMES 50
 CLUSTER
   SIN OPTIMIZAR
     Tiempo requerido solucion Double: 19.441738
     Tiempo requerido solucion Float: 18.744871
   OPTIMIZADO
     Tiempo requerido solucion Double: 2.693236
     Tiempo requerido solucion Float: 2.421828

 LOCAL
   SIN OPTIMIZAR
     Tiempo requerido solucion Double: 12.147080
     Tiempo requerido solucion Float: 13.480428
   OPTIMIZADO
     Tiempo requerido solucion Double: 0.708603
     Tiempo requerido solucion Float: 0.627858

QUAD2 TIMES 100
 CLUSTER
   SIN OPTIMIZAR
     Tiempo requerido solucion Double: 39.253563
     Tiempo requerido solucion Float: 37.839327
   OPTIMIZADO
     Tiempo requerido solucion Double: 6.817833
     Tiempo requerido solucion Float: 4.702941
 LOCAL
   SIN OPTIMIZAR
     Tiempo requerido solucion Double: 23.427781
     Tiempo requerido solucion Float: 26.285536
   OPTIMIZADO
     Tiempo requerido solucion Double: 1.416524
     Tiempo requerido solucion Float: 1.187522
```

La mayor diferencia que se encuentra es que en el CLUSTER tarda mas el double que el float

pero en Local pasa lo contrario. Esto se debe a que ciertas arquitecturas de hardware benefician mas a float y otras benefician mas a los double.

c. El algoritmo quadratic3.c computa las raíces de esta ecuación, pero en forma repetida. Compile y ejecute el código variando la constante TIMES. ¿Qué diferencia nota en la ejecución? ¿Qué diferencias puede observar en el código con respecto a quadratic2.c?

```
QUAD3 TIMES 50
 CLUSTER
   SIN OPTIMIZAR
     Tiempo requerido solucion Double: 19.441738
     Tiempo requerido solucion Float: 18.744871
   OPTIMIZADO
     Tiempo requerido solucion Double: 3.149339
     Tiempo requerido solucion Float: 1.870756
 LOCAL
   SIN OPTIMIZAR
     Tiempo requerido solucion Double: 12.090996
     Tiempo requerido solucion Float: 8.014069
 OPTIMIZADO
     Tiempo requerido solucion Double: 0.661828
     Tiempo requerido solucion Float: 0.409379

QUAD3 TIMES 100
 CLUSTER
   SIN OPTIMIZAR
     Tiempo requerido solucion Double: 38.889745
     Tiempo requerido solucion Float: 55.777811
   OPTIMIZADO
     Tiempo requerido solucion Double: 4.657589
     Tiempo requerido solucion Float: 3.162098
 LOCAL
   SIN OPTIMIZAR
     Tiempo requerido solucion Double: 23.522227
     Tiempo requerido solucion Float: 15.528005
   OPTIMIZADO
     Tiempo requerido solucion Double: 1.303832
     Tiempo requerido solucion Float: 0.787573
```

Tarda menos respecto al punto B. En este caso se nota una tendencia a que las resoluciones con float tarden menos, esto se debe a que se utilizan constantes en Float en vez de “castearlas” a Float desde Double y en que se utilizan funciones especificas del tipo Float como powf() y sqrtf()

# Ejercicio Numero 2

Empezaremos lo mas explicito y simple posible sin esforzarnos demasiado en encontrar ninguna optimizacion, de esta forma podriamos documentar las optimizaciones y sus cambios en el tiempo

de ejecucion.

Como excepcion a esto, comenzamos con una extraccion de una multiplicacion `(RA + RB) -> R(A + B)` , sencillamente porque nos resulto mas sencillo contruirlo directamente de esta forma.

A continuacion los specs de la PC hogareña:

- Procesador: Intel Core i7-10700 2.90 GHz
- Ram: 16,0 GB

Escribimos un pequeño test bash para asegurar la correctitud del programa a medida que vamos agregando optimizaciones.

```
Resultados del algoritmo base en PC:
  Tiempo en segundos 0.5465590954 para 512
  Tiempo en segundos 5.0782840252 para 1024
  Tiempo en segundos 114.8600111008 para 2048
```

### Almacenamiento de valores sobre su recalculamiento - Optimizacion V2

La primera optimizacion que implementamos fue la de almacenar valores en variables auxiliares en vez de realizar el mismo calculo varias veces.

Esto lo realizamos con el calculo del indice y con el acceso a elementos de las matrices, pero no vimos mejora y hasta en ciertos casos empeoraba por lo que descatamos esta optimizacion.

```
Resultados del algoritmo V2 en PC:
  Tiempo en segundos 0.5131499767 para 512
  Tiempo en segundos 9.0782840252 para 1024
```

## Mejora a la Optimizacion V2 - Optimizacion V2.2

Eliminamos el almacenamiento del calculo de los indices de las matrices pero mantuvimos el almacenamiento de los accesos a la matrices durante el calculo de R donde si vimos una mejora significativa.

```
Resultados del algoritmo V2.2 en PC:
  Tiempo en segundos 0.4950668812 para 512
  Tiempo en segundos 4.0553719997 para 1024
  Tiempo en segundos 62.7206149101 para 2048
```

## Matriz resultado en vez de reutilizar R - Optimizacion V3.

Probamos usar una matriz para almacenar el resultado en vez de usar R mismo. No presento mejoras, y hasta cierta desmejora.

```
Resultados del algoritmo V3 en PC:
  Tiempo en segundos 0.5233290195  para 512
  Tiempo en segundos 4.0354280472  para 1024
  Tiempo en segundos 66.0288639069 para 2048
```

## Decremento del indice como condicion en el for - Optimizacion V4

Para realizar esta optimizacion tomamos ventaja de que en C el 0 se considera falso para combinar el decremento y la condicion del for en una sola expresion. Para esto tuvimos que invertir todos los for, se observo una mejora.

```
Resultados del algoritmo V4 en PC:
  Tiempo en segundos 0.4419710636 para 512
  Tiempo en segundos 3.7746620178 para 1024
  Tiempo en segundos 52.0080058575 para 2048
```

## Acceso por columnas del operando derecho en la multiplicacion de matrices - Optimizacion V5

Finalmente, accedimos por columna en los operandos derechos de las multiplicacion de matrices para asi realizarlas mas eficientemente. Esto se debe a que para el calculo de un elemento del resultado de una multiplicaion se debe recorrer la fila de la matriz izquierda y la columna de la matriz derecha, por lo que acceder a la matriz derecha por filas es extremadamente ineficiente. La mejora fue tan significativa que nos dejo satisfechos y dimos por finalizado el proceso de optimizacion.

Para esto debimos incluir una nueva matriz resultado pero aun asi se mantuvo la mejora.

```
Resultados del algoritmo V5 en PC:
  Tiempo en segundos 0.2881238461 512
  Tiempo en segundos 2.2192440033 1024
  Tiempo en segundos 18.6206161976 2048
```

```
Resultados del algoritmo V5 optimizado con O3 en PC:
  Tiempo en segundos 0.1262190342 en 512
  Tiempo en segundos 1.0131981373 en 1024
  Tiempo en segundos 8.9814372063 en 2048
```

```
  Tiempo en segundos 71.7591030598 en 4096


Resultados del algoritmo V5 en CLUSTER:
  Tiempo en segundos 0.9209411144 para 512
  Tiempo en segundos 7.8719360828 para 1024
  Tiempo en segundos 66.6019508839 para 1024


Resultados del algoritmo V5 optimizado con O3 en CLUSTER:
  Tiempo en segundos 0.2335169315 para 512
  Tiempo en segundos 3.9887001514 para 1024
  Tiempo en segundos 33.6858358383 para 2048
  Tiempo en segundos 289.3322269917 para 4096
```