22-3-2024

Vectores con RPC GEN

Práctica 2

Julián Blanco González
UNIVERSIDAD DE EXTREMADURA

Índice

Objetivo de la práctica

El objetivo de la práctica es trabajar con los vectores usando el protocolo RPC. Para ello, hay que realizar 5 funciones en la parte del servidor:

- **Inicializar vector:** inicializa un vector dependiendo del tamaño y rango de aleatorios introducido.
- **Sumar 2 Vectores:** suma 2 vectores pasados por parámetro
- **Producto escalar:** hace el producto escalar de dos vectores pasados por parámetro
- **Producto vectorial:** realiza el producto vectorial de dos vectores pasados por parámetro
- **Saxpy:** hacer la operación de álgebra sobre los dos vectores. La fórmula es:

$$SAXPY(\alpha, X, Y) = \alpha * X + Y$$

En el código del cliente, lo primero es pedir el tamaño del vector y el rango de números aleatorios a crear. Luego, es ir llamando a las funciones de 1 en 1 de manera secuencial, e ir comprobando algunos casos especiales, que luego explicaré en el apartado del cliente.

Vectores con RPC

A la hora de trabajar con vectores, RPC genera por defecto en el archivo de cabecera (".h"), una estructura del vector donde tiene los campos de longitud y valor, para poder luego acceder desde el servidor o el cliente. También, se crea un fichero con extensión ("xdr"), que sirve para validar los datos del vector.

En el lado del servidor, al trabajar con vectores (espacio en memoria), hay que hacer 3 cosas antes de inicializarlos:

- Liberar espacio que estuviera de antes cogido en memoria
- Declarar el vector (tamaño)
- Reservar memoria para el tamaño escogido del vector

Se pueden crear vectores de tamaño fijo como variable.

Tamaño fijo

```
- Typedef float vector <tamaño>;
- Struct vector {
      Float v [tamaño]
  }
```

Tamaño variable

- `Typedef float vector <>;`
- `Struct vector {`

  `Float v <>`

  `}`

Fichero de interfaz (vectores.x)

Este es el fichero .x que he creado para hacer la práctica:

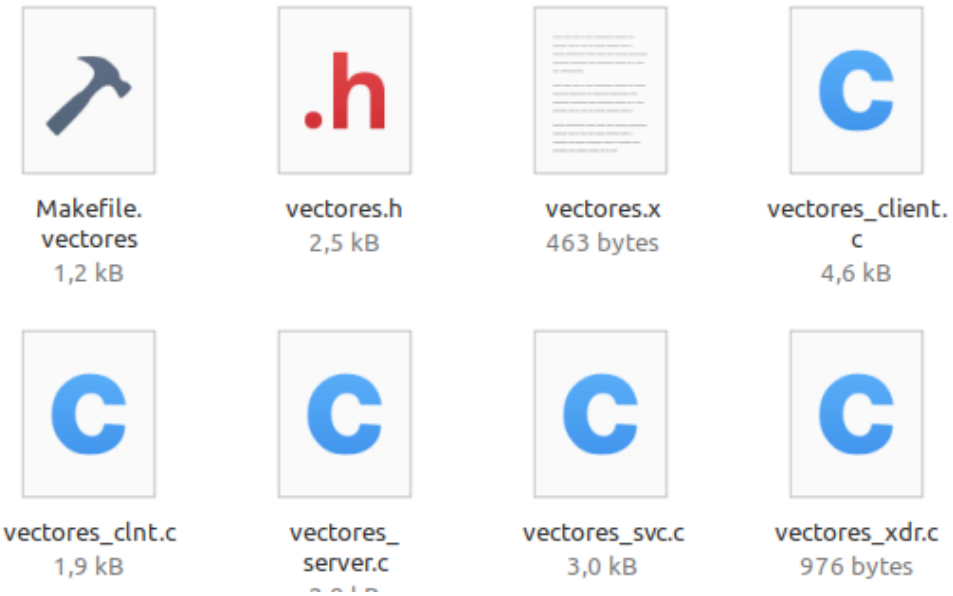
```
typedef float vec<>;

struct vectores {
    vec v1;
    vec v2;
};
struct datosInit {
    int limiteV;
    int rango;
};
struct vectoresSaxpy {
    float numero;
    vec v1;
    vec v2;
};

program PROGRAMA {
    version VERSION {
        vectores initVector(datosInit)=1;
        vec sumaVector(vectores)=2;
        float productoEscalar(vectores)=3;
        vec productoVectorial(vectores)=4;
        vec saxpyOperacion(vectoresSaxpy)=5;
    }=1;
}=0x20000013;
```

1. **Estructuras de datos**:
    - **vec** es un vector de números de punto flotante de tamaño variable.
    - **vectores** es una estructura que contiene dos vectores vec.
    - **datosInit** es una estructura con dos campos enteros: limiteV (tamaño del vector) y rango (límite superior de los aleatorios).
    - **vectoresSaxpy** es una estructura con un número de punto flotante y dos vectores vec para poder hacer la función saxpy.
2. **Programa**:
    - El programa se llama “PROGRAMA” y tiene una versión llamada “VERSION”.
    - La versión define varias funciones:
        - **initVector** recibe la estructura datosInit y devuelve 2 vectores.
        - **sumaVector** recibe los 2 vectores y devuelve un vector.
        - **productoEscalar** recibe los 2 vectores y devuelve un float.
        - **productoVectorial** recibe los 2 vectores y devuelve un vector.
        - **saxpyOperacion** recibe la estructura vectoresSaxpy (escalar y los 2 vectores) y devuelve un vector.
3. **ID del programa**:
    - La ID del programa es 0x20000013.

Para generar todos los archivos, se utiliza el comando rpcgen:

```
julian@julian-VirtualBox:~/Escritorio/Distribuidos/Vectores_S2$ rpcgen -a vectores.x
```

Se generan estos archivos:



Voy a nombrar lo nuevo respecto de la sesión anterior:

Vectores.h: está declarado la estructura del vector y se le asigna automáticamente los campos de longitud y valor.

```
typedef struct {
    u_int vec_len;
    float *vec_val;
} vec;
```

Vectores_xdr.c: función de validación. Hay que ver el nombre de la función para poder llamarla en el servidor a la hora de liberar memoria:

```
bool_t
xdr_vec (XDR *xdrs, vec *objp)
{
    register int32_t *buf;

    if (!xdr_array (xdrs, (char **)&objp->vec_val, (u_int *) &objp->vec_len, ~0,
       sizeof (float), (xdrproc_t) xdr_float))
        return FALSE;
    return TRUE;
}
```

Servidor.c

Código del servidor. Aquí están las definiciones de cada función.

Inicializar vector

```
vectores *
initvector_1_svc(datosInit *argp, struct svc_req *rqstp)
{
    static vectores  result;

    xdr_free((xdrproc_t)xdr_vec,(char *)&result);
    result.v1.vec_len = argp->limiteV;
    result.v1.vec_val = malloc(result.v1.vec_len*sizeof(float));
    result.v2.vec_len = argp->limiteV;
    result.v2.vec_val = malloc(result.v2.vec_len*sizeof(float));

    printf("\nVector1 con tamaño: %d \n", result.v1.vec_len);
    //Iniciar con valores aleatorios de 0 a rango (usuario)
    for (int i = 0; i < result.v1.vec_len; i++)
    {
        result.v1.vec_val[i] = (float)rand() / RAND_MAX * argp->rango;
        printf("Elemento %d: %.3f\n", i, result.v1.vec_val[i]);
    }

    printf("\nVector2 con tamaño: %d \n", result.v2.vec_len);
    //Iniciar con valores aleatorios de 0 a rango (usuario)
    for (int i = 0; i < result.v2.vec_len; i++)
    {
        result.v2.vec_val[i] = (float)rand() / RAND_MAX * argp->rango;
        printf("Elemento %d: %.3f\n", i, result.v2.vec_val[i]);
    }
    return &result;
}
```

Libero la memoria anterior que hubiera (llamada a la función del archivo xdr). Luego, le asigno a cada vector el tamaño pasado por parámetro (lo introduce el usuario en el cliente). Justo después, reservo la memoria suficiente para guardar los vectores y poder utilizarlos luego. Y, por último, genero los aleatorios (float) de cada vector.

*Para acceder a cada campo de cada vector, primero accedo a la estructura vectores, y luego a la estructura de cada vector:

result.v1.vec_len: v1 (struct vectores) y vec_len (campo del vector individual)

Sumar 2 vectores

```
vec *
sumavector_1_svc(vectores *argp, struct svc_req *rqstp)
{
    static vec  result;

    result.vec_len = argp->v1.vec_len;
    result.vec_val = malloc(result.vec_len * sizeof(float));

    for (int i = 0; i < result.vec_len; i++) {
        result.vec_val[i] = argp->v1.vec_val[i] + argp->v2.ve
        //printf("Posición [%d]: %.3f + %.3f = %.3f\n", (i+1)
    }

    return &result;
}
```

Le doy el tamaño del vector1 (podría ser el 2) y reservo la memoria para ello. Para hacer la suma de los vectores, simplemente voy sumando los valores dependiendo de la posición y guardándolos en el vector resultado.

Producto escalar

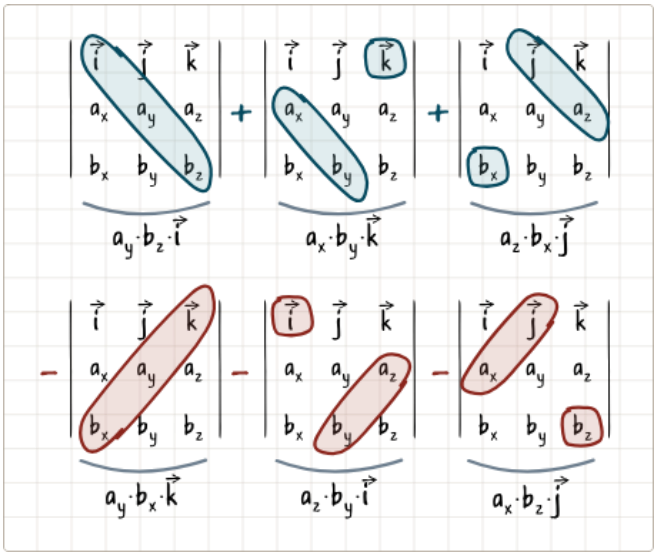
```
float *
productoescalar_1_svc(vectores *argp, struct svc_req *rqstp)
{
    static float  result;
    result = 0.0f;

    for (int i = 0; i < argp->v1.vec_len; i++)
    {
        result = result + (argp->v1.vec_val[i] * argp->v2.vec_val[i]);
    }

    return &result;
}
```

Para hacer el producto escalar, tengo que hacer el producto por posición de ambos vectores e irlos sumando. Devuelvo el resultado, que es un float.

Producto vectorial

```
vec *
productovectorial_1_svc(vectores *argp, struct svc_req *rqstp)
{
    static vec  result;

    //Solo me va a llamar a esta función, cuando en el cliente, ambos vectores sean de 3 dimensiones
    result.vec_len = 3;
    result.vec_val = malloc(3 * sizeof(float));

    result.vec_val[0] = argp->v1.vec_val[1] * argp->v2.vec_val[2] - argp->v1.vec_val[2] * argp->v2.vec_val[1];
    result.vec_val[1] = argp->v1.vec_val[2] * argp->v2.vec_val[0] - argp->v1.vec_val[0] * argp->v2.vec_val[2];
    result.vec_val[2] = argp->v1.vec_val[0] * argp->v2.vec_val[1] - argp->v1.vec_val[1] * argp->v2.vec_val[0];

    return &result;
}
```

Reservo 3 lugares de memoria, ya que el producto vectorial solo se puede hacer para vectores tridimensionales. Luego para hacer el producto vectorial, hay que aplicar la regla de Sarrus:



Saxpy

```
vec *
saxpyoperacion_1_svc(vectoresSaxpy *argp, struct svc_req *rqstp)
{
    static vec  result;

    result.vec_len = argp->v1.vec_len;
    result.vec_val = malloc(result.vec_len * sizeof(float));

    for (int i = 0; i < result.vec_len; i++)
    {
        result.vec_val[i] = argp->numero * argp->v1.vec_val[i] + argp->v2.vec_val[i];
    }

    return &result;
}
```

Le doy el tamaño del vector1 (podría ser el 2) y reservo la memoria. Para hacer saxpy, simplemente hay que aplicar la fórmula que puse al principio, es decir, multiplicar el escalar por cada coordenada del primer vector y sumarle la coordenada del segundo (se hace por posición).

Cliente.c

Variables

```
CLIENT *clnt;
vectores  *result_1;
datosInit  initvector_1_arg;
vec  *result_2;
vectores  sumavector_1_arg;
float  *result_3;
vectores  productoescalar_1_arg;
vec  *result_4;
vectores  productovectorial_1_arg;
vec  *result_5;
vectoresSaxpy  saxpyoperacion_1_arg;
int tamano, rango;
float escalar;
```

*Las variables enteras “tamano” y “rango” son para la función de inicializar vector.

*La variable float “escalar” es para la operación saxpy

Llamadas a las funciones

```
/*result_1 = initvector_1(&initvector_1_arg, clnt);
if (result_1 == (vectores *) NULL) {
    clnt_perror (clnt, "call failed");
}
result_2 = sumavector_1(&sumavector_1_arg, clnt);
if (result_2 == (vec *) NULL) {
    clnt_perror (clnt, "call failed");
}
result_3 = productoescalar_1(&productoescalar_1_arg, clnt);
if (result_3 == (float *) NULL) {
    clnt_perror (clnt, "call failed");
}
result_4 = productovectorial_1(&productovectorial_1_arg, clnt);
if (result_4 == (vec *) NULL) {
    clnt_perror (clnt, "call failed");
}
result_5 = saxpyoperacion_1(&saxpyoperacion_1_arg, clnt);
if (result_5 == (vec *) NULL) {
    clnt_perror (clnt, "call failed");
}*/
```

Funciones

A continuación, voy a mostrar como llamar a las funciones del servidor en el cliente.

Inicializar Vector

```
printf("1. Introducir el tamaño y el rango de ambos vectores\n");
printf("\nTamaño: ");
scanf("%d", &tamano);
printf("Rango de los números aleatorios: ");
scanf("%d", &rango);

initvector_1_arg.limiteV = tamano;
initvector_1_arg.rango = rango;

result_1 = initvector_1(&initvector_1_arg, clnt);
if (result_1 == (vectores *) NULL) {
    clnt_perror (clnt, "call failed");
}
else{
    printf("\nVector 1: ");
    mostrar_vector(&result_1->v1);
    printf("\nVector 2: ");
    mostrar_vector(&result_1->v2);

}
```

Primero, pido por teclado, tanto el tamaño como el rango límite de los aleatorios. Luego, le asigno ambos valores a los campos de la estructura de datos de entrada (datosInit). Después, copio la llamada a la función “initVector_1”. Si no da error, recibo en la estructura result_1 (vectores) los vectores inicializados con el tamaño y el rango de aleatorios.

Suma de vectores

```
printf("2. Sumar ambos vectores: \n");
result_2 = sumavector_1(result_1, clnt);
if (result_2 == (vec *)NULL)
{
    clnt_perror(clnt, "call failed");
}
else
{
    printf("\nResultado de la suma: ");
    mostrar_vector(result_2);
}
```

Copio la llamada a la función de suma del servidor, pero le cambio el parámetro de entrada por mi estructura de vectores.

Producto escalar

```
printf("3. Producto escalar de ambos vectores\n");
result_3 = productoescalar_1(result_1, clnt);
if (result_3 == (float *) NULL) {

    clnt_perror (clnt, "call failed");
}
else
{
    printf("\nProducto escalar: %.4f\n", *result_3);
}
```

Igual que la suma, copio la llamada y sustituyo el parámetro por la estructura de los vectores. Luego muestro el resultado por pantalla con un puntero a float.

Producto vectorial

```
printf("4. Producto vectorial de ambos vectores\n");
//Si los vectores no son de tamaño 3, no puede hacerse
if(result_1->v1.vec_len != 3 || result_1->v2.vec_len != 3)
{
    printf("\nLos vectores deben ser tridimensionales para calcular el producto vectorial.\n");
}
else
{
    result_4 = productovectorial_1(result_1, clnt);
    if (result_4 == (vec *)NULL)
    {
        clnt_perror(clnt, "call failed");
    }
    else
    {
        printf("\nResultado del producto vectorial:\n");
        mostrar_vector(result_4);
    }
}
```

Igual que los dos anteriores, solo que realizo la llamada al producto vectorial siempre y cuando, la longitud de los vectores sea 3, ya que el producto vectorial solo se puede hacer con vectores de 3 dimensiones.

Saxpy

```
printf("5. SAXPY(α,X,Y)=α*X+Y\n");
printf("\nIntroduce el escalar del saxpy: ");
scanf("%f", &escalar);

saxpyoperacion_1_arg.numero = escalar;
saxpyoperacion_1_arg.v1 = result_1->v1;
saxpyoperacion_1_arg.v2 = result_1->v2;

result_5 = saxpyoperacion_1(&saxpyoperacion_1_arg, clnt);
if (result_5 == (vec *) NULL) {
    clnt_perror (clnt, "call failed");
}
else
{
    printf("\nResultado de la operación SAXPY:\n");
    mostrar_vector(result_5);
}
```

Primero, pido que el usuario introduzca el float escalar. Una vez introducido, le asigno a cada campo de la estructura que me he creado para la operación saxpy, su variable correspondiente.

MostrarVector (auxiliar)

```
void mostrar_vector(vec *vector) {
    printf("[ ");
    for (int i = 0; i < vector->vec_len; i++) {
        printf("%.3f", vector->vec_val[i]);
        if (i < vector->vec_len - 1) {
            printf(", ");
        }
    }
    printf(" ]\n");
}
```

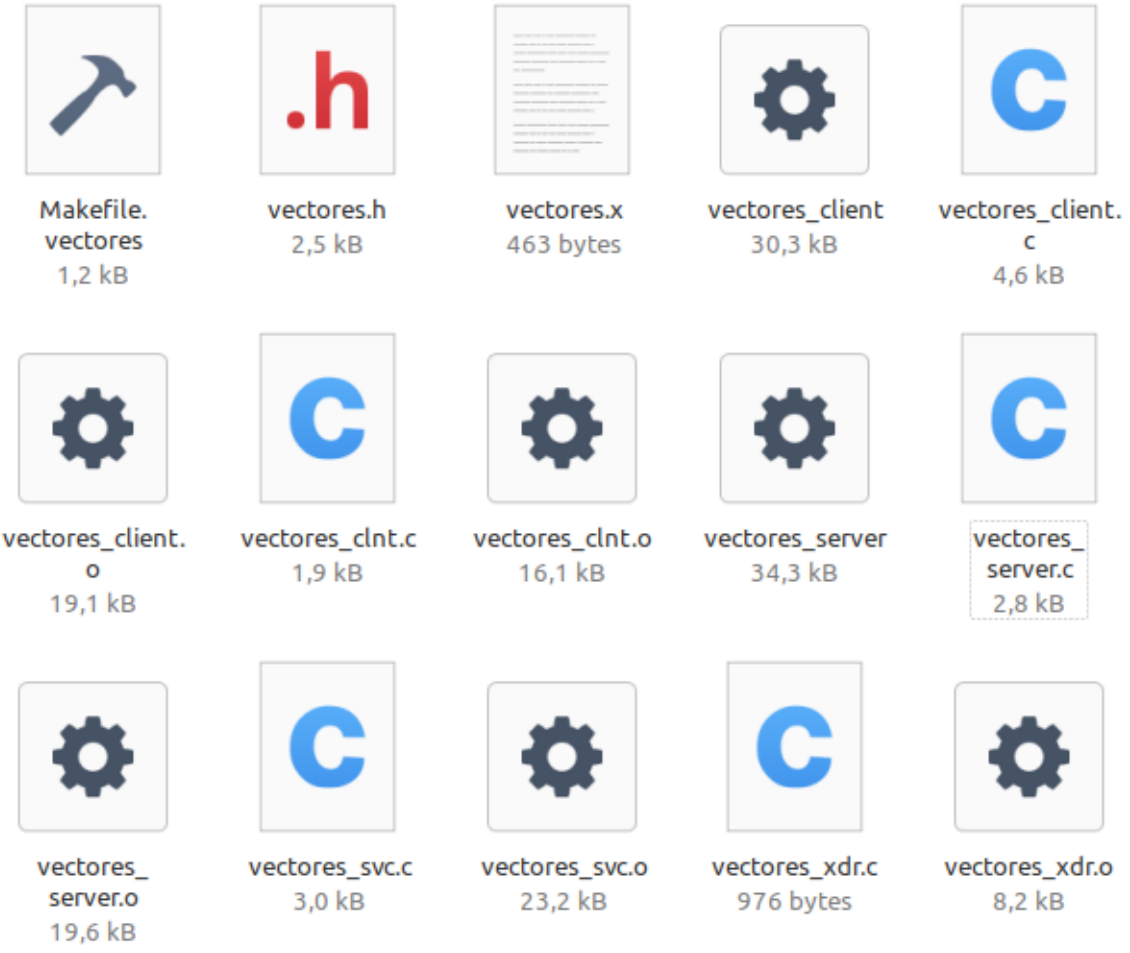
Para no tener que estar haciendo un for cada vez que quiera mostrar un vector, he hecho esta función básica para mostrar un vector pasado por parámetro.

Compilar con MakeFile

Una vez acabado el código del servidor y el cliente, compilo todos los archivos con el comando MakeFile (he añadido las 3 líneas al archivo Makefile para que no dé fallos al compilar). El comando es el siguiente:

```
julian@julian-VirtualBox:~/Escritorio/Distribuidos/Vectores_S2$ make -f Makefile.vectores
```

Y genera todos archivos ".o":



Ejecución de la práctica

Para ejecutar la práctica, hay que ejecutar 2 comandos:

Lanzar el server:

```
julian@julian-VirtualBox:~/Escritorio/Distribuidos/S2$ ./vectores_server
```

Lanzar el cliente:

```
julian@julian-VirtualBox:~/Escritorio/Distribuidos/S2$ ./vectores_client localhost
```

Voy a poner capturas de una página de internet, que hace de calculadora online, para validar los resultados obtenidos.

Inicializar vectores

```
Práctica 2. Trabajando con vectores
------------------------------------------------------------
1. Introducir el tamaño y el rango de ambos vectores

Tamaño: 3
Rango de los números aleatorios: 10

Vector 1: [ 8.402, 3.944, 7.831 ]

Vector 2: [ 7.984, 9.116, 1.976 ]
------------------------------------------------------------
```

Suma de ambos vectores

```
------------------------------------------------
2. Sumar ambos vectores:

Resultado de la suma: [ 16.386, 13.060, 9.807 ]
------------------------------------------------
```

Verificación:

Calculemos la suma (la resta) de los vectores:

$$\overline{a} + \overline{b} = \{a_x + b_x ; a_y + b_y ; a_z + b_z\} = \{8.402 + 7.984 ; 3.944 + 9.116 ; 7.831 + 1.976\} =$$

$$= \{16.386 ; 13.06 ; 9.807\}$$

Producto escalar

```
----------------------------------------
3. Producto escalar de ambos vectores

Producto escalar: 118.5080
----------------------------------------
```

Verificación:

Calculemos el producto escalar de los vectores:

$$\overline{a} \cdot \overline{b} = a_x \cdot b_x + a_y \cdot b_y + a_z \cdot b_z = 8.402 \cdot 7.984 + 3.944 \cdot 9.116 + 7.831 \cdot 1.976 =$$

$$= 67.081568 + 35.953504 + 15.474056 = 118.509128$$

Producto Vectorial

```
4. Producto vectorial de ambos vectores

Resultado del producto vectorial:
[ -63.600, 45.928, 45.106 ]
```

Verificación:

Solución:

$$\overline{a}\times\overline{b}=\begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}=\begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ 8.402 & 3.944 & 7.831 \\ 7.984 & 9.116 & 1.976 \end{vmatrix}=$$

$$=\boldsymbol{i}(3.944\cdot 1.976 - 7.831\cdot 9.116) - \boldsymbol{j}(8.402\cdot 1.976 - 7.831\cdot 7.984) + \boldsymbol{k}(8.402\cdot 9.116 - 3.944\cdot 7.984)=$$

$$=\boldsymbol{i}(7.793344 - 71.387396) - \boldsymbol{j}(16.602352 - 62.522704) + \boldsymbol{k}(76.592632 - 31.488896)=$$

$$=\{-63.594052;\ 45.920352;\ 45.103736\}$$

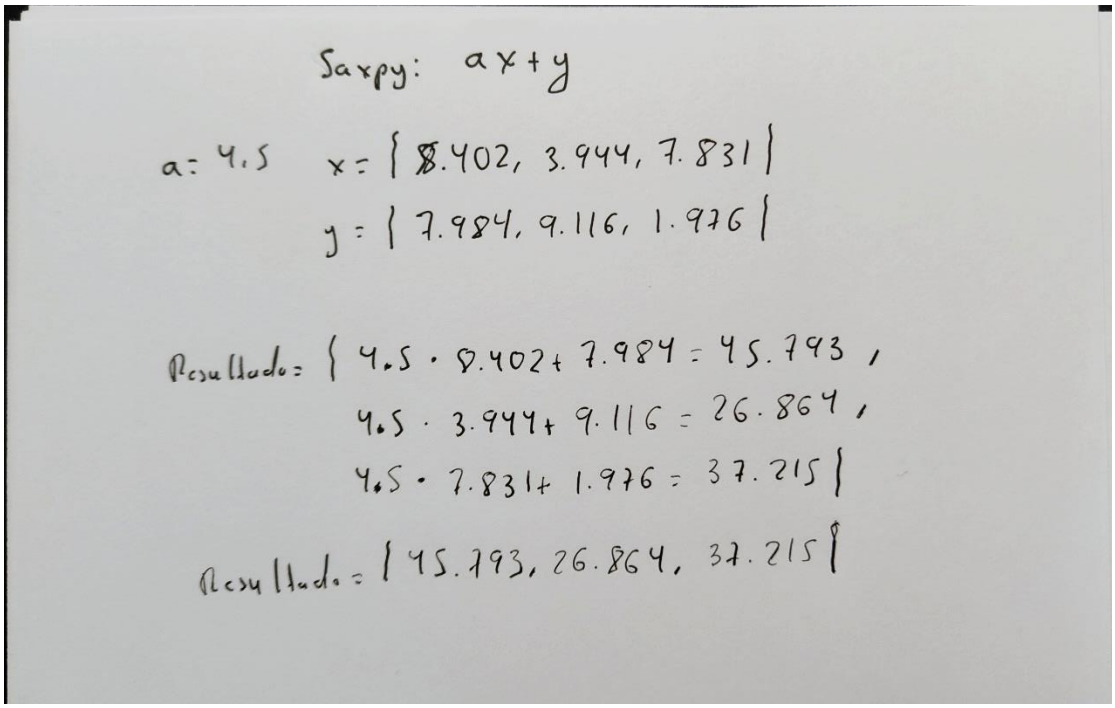
Ejecución cuando los vectores no son de 3 dimensiones:

```
Práctica 2. Trabajando con vectores
------------------------------------------------------------------------
1. Introducir el tamaño y el rango de ambos vectores

Tamaño: 5
Rango de los números aleatorios: 98

Vector 1: [ 32.852, 75.286, 27.222, 54.289, 46.785 ]

Vector 2: [ 61.629, 35.749, 50.313, 93.319, 89.787 ]
------------------------------------------------------------------------
2. Sumar ambos vectores:

Resultado de la suma: [ 94.481, 111.035, 77.535, 147.608, 136.572 ]
------------------------------------------------------------------------
3. Producto escalar de ambos vectores

Producto escalar: 15352.5264
------------------------------------------------------------------------
4. Producto vectorial de ambos vectores

Los vectores deben ser tridimensionales para calcular el producto vectorial.
------------------------------------------------------------------------
```

Operación Saxpy

```
5. SAXPY(α,X,Y)=α*X+Y

Introduce el escalar del saxpy: 4.5

Resultado de la operación SAXPY:
[ 45.793, 26.864, 37.215 ]
----------------------------------------
```

Verificación:



Problemas encontrados

A la hora de inicializar el segundo vector, los números aleatorios, eran los mismos que los del primer vector, ya que al iniciar la semilla en el server y las dos llamadas de aleatorios es una después de otra. Lo que hago entonces, es generar la semilla una sola vez, y lo hago en el cliente.

También, otro problema ha sido como guardar memoria para ambos vectores, ya que sin la estructura que he creado, si llamaba 2 veces a la función de initVector, con dos vectores diferentes, se me sobrescribía el primero con el segundo, y el resultado era que quedaban 2 vectores con los mismos valores. Por ello, he hecho la estructura con dos vectores.

Bibliografía

He seguido para la creación de los vectores:

PDF de la práctica: S2.pdf

Y para poder hacer la práctica y mejorar la documentación, he seguido estos enlaces:

Saxpy

https://medium.com/@alexisvaquero/como-implementar-saxpy-en-c-8b77e3df0a7d

Calculadora online

https://es.onlinemschool.com/math/assistance/vector/