LENGUAJES DE PROGRAMACIÓN

Solución al Trabajo Práctico - Septiembre de 2019

EJERCICIO 1

Escriba un programa en C++ que calcule y escriba en la consola los autovalores de una matriz 2×2, la cual ha sido declarada en el programa como un array bidimensional constante de componentes reales.

Sea $\mathbf{A}$ la matriz anterior. El cálculo de los autovalores de la matriz $\mathbf{A}$ debe realizarse calculando los ceros del polinomio característico $\det(\mathbf{A} - \lambda \cdot \mathbf{I})$.

El programa debe escribir los autovalores en la consola en formato científico, con 6 dígitos detrás del punto decimal.

Mostrar en la memoria el resultado obtenido de ejecutar el programa para cada una de las dos matrices siguientes.

Caso de Prueba 1: $\mathbf{A} = \begin{pmatrix} -1 & 3 \\ -2 & 4 \end{pmatrix}$

Caso de Prueba 2: $\mathbf{A} = \begin{pmatrix} -0.5 & 3 \\ -2 & -0.5 \end{pmatrix}$

Solución al Ejercicio 1

En el programa se calculan los autovalores de dos maneras: usando sólo números reales y empleando la cabecera <complex>.

```
#include <iostream>
#include <iomanip>
#include <complex>

const double A[][2] = { {-1, 3}, {-2, 4} };
//const double A[][2] = { {-0.5, 3}, {-2, -0.5} };

int main() {
    double b  = -(A[0][0]+A[1][1]);
    double c  = A[0][0]*A[1][1] - A[0][1]*A[1][0];
    double b2_4ac  = b*b-4*c;
    // ---------------------------------------
    // Opción 1: Cálculo usando reales
    // ---------------------------------------
    double auxR = std::sqrt(std::abs(b2_4ac));
    if ( b2_4ac  < 0 ) {
        std::cout << std::setprecision(6) << std::scientific
              << "Autovalores:\n"
              << -0.5*b << " + i " << 0.5*auxR << "\n"
              << -0.5*b << " - i " << 0.5*auxR << std::endl;
    } else {
        std::cout << std::setprecision(6) << std::scientific
              << "Autovalores:\n"
              << 0.5*(-b+auxR) << "\n"
              << 0.5*(-b-auxR) << std::endl;
    }
    // ---------------------------------------------------
    // Opción 2: Cálculo usando números complejos
    // ---------------------------------------------------
    std::complex<double> auxC =
                      std::sqrt((std::complex<double>)(b2_4ac));
    std::cout << std::setprecision(6) << std::scientific
          << "Autovalores:\n" << 0.5*(-b+auxC)
          << "\n" << 0.5*(-b-auxC) << std::endl;
    return 0;
}
```

EJERCICIO 2

Escriba un programa en C++ para la gestión de un calendario de eventos. El programa debe permitir que el usuario, mediante comandos introducidos por consola, planifique nuevos eventos, extraiga el evento más inminente, borre el calendario, escriba en la consola contenido del calendario, y finalice el programa.

Cada evento está especificado mediante dos números enteros no negativos. El primero de ellos, al que nos referiremos como IDENT, es un identificador del evento. El segundo, al que nos referiremos como TIME, indica el instante de tiempo para el cual está planificado que suceda el evento.

El programa debe leer y ejecutar los comandos que el usuario va introduciendo por consola. La sintaxis de los comandos se muestra a continuación.

Comando	**Significado**
`list`	Listado de todos los eventos planificados en el calendario.
`pop`	Escribe en la consola y elimina del calendario el evento más inminente. En caso de que haya más de un evento planificado para el instante más inminente, se escribe en consola y elimina aquel que fue introducido primero en el calendario.
`clear`	Borra todo el contenido del calendario.
IDENT@TIME	Comprueba si ya existe en el calendario un evento con ese mismo IDENT y TIME. En caso afirmativo, no realiza ninguna acción. En caso negativo, inserta en el calendario este nuevo evento.
`end`	Muestra en la consola el mensaje “Programa finalizado” y termina el programa.

Al arrancar el programa, el calendario se encuentra vacío. El programa debe escribir en la consola el símbolo > a fin de indicar al usuario que está listo para que éste introduzca un comando. Cuando el usuario introduce un comando, el programa lo lee y, o bien lo ejecuta, o bien muestra un mensaje de error en caso de que no lo reconozca.

La ejecución de un comando puede conllevar una modificación del calendario, la escritura en consola y la finalización del programa. Salvo en este último caso, una vez ejecutado el comando, el programa escribe nuevamente en la consola el símbolo >, para indicar al usuario que está listo para que éste introduzca un comando.

A continuación se muestra un posible secuencia de comandos y el texto escrito en cada caso por el programa en la consola.

```
> list
Calendario vacio

> 77@12
> 28@7
> 19@12
> 100@0
> 19@18
> list
IDENT 100, TIME 0
IDENT 28, TIME 7
IDENT 77, TIME 12
IDENT 19, TIME 12
IDENT 19, TIME 18

> 77@12
> list
IDENT 100, TIME 0
IDENT 28, TIME 7
IDENT 77, TIME 12
IDENT 19, TIME 12
IDENT 19, TIME 18

> pop
IDENT 100, TIME 0

> list
IDENT 28, TIME 7
IDENT 77, TIME 12
IDENT 19, TIME 12
IDENT 19, TIME 18

> 12@ 1
Comando desconocido
> list
IDENT 28, TIME 7
IDENT 77, TIME 12
IDENT 19, TIME 12
IDENT 19, TIME 18

> clear
> list
Calendario vacio

> end
Programa finalizado
```

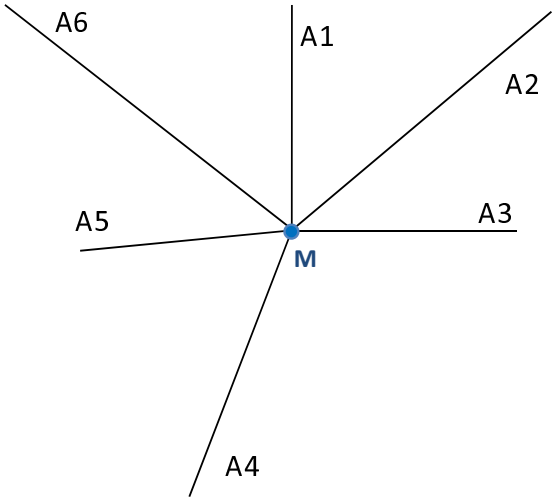
Solución al Ejercicio 2

```
#include <iostream>
#include <string>
#include <stdexcept>
#include <stdlib.h>
#include <list>
#include <sstream>

struct Evento {
    int ident;
    int time;
};

Evento isEvento(std::string comando)
              throw (std::invalid_argument){
    unsigned int posAt = comando.find('@');
    if (posAt<1 || posAt>comando.size()-2)
        throw std::invalid_argument ("");
    std::string ident = comando.substr(0,posAt);
    std::string time  = comando.substr(posAt+1,comando.size());
    if ( ident.find_first_not_of("0123456789") != std::string::npos ||
         time.find_first_not_of("0123456789") != std::string::npos  )
        throw std::invalid_argument ("");
    Evento ev ={atoi(ident.c_str()),atoi(time.c_str())};
    return ev;
}

std::string listCalendario(std::list<Evento> &calendario){
    std::stringstream ss;
    std::list<Evento>::iterator p = calendario.begin();
    while ( p != calendario.end() ){
        ss    << "IDENT " << p->ident << ", "
          << "TIME "  << p->time  << "\n";
        p++;
    }
    if (ss.str().size() == 0)
        ss << "Calendario vacio\n";
    return ss.str();
}

std::string listPrimerElementoCalendario(std::list<Evento>
&calendario){
    std::stringstream ss;
    std::list<Evento>::iterator p = calendario.begin();
    if ( p != calendario.end() ){
        ss    << "IDENT " << p->ident << ", "
          << "TIME "  << p->time  << "\n";
    } else {
        ss << "Calendario vacio\n";
    }
    return ss.str();
}
```

```
void insertaEvento(std::list<Evento> &calendario,Evento &ev){
    std::list<Evento>::iterator p = calendario.begin();
    while ( p != calendario.end() ){
        if ( p->ident == ev.ident && p->time == ev.time )
            return;
        if ( p->time > ev.time ){
            calendario.insert(p,ev);
            return;
        }
        p++;
    }
    calendario.push_back(ev);
    return;
}

int main(){
    std::list<Evento> calendario;
    do{
        std::cout << "> ";
        std::string comando;
        // Lectura de una línea por consola
        std::getline(std::cin,comando);
        // Elimina espacio en blanco al comienzo y final del comando
        const char* t = " \t\n";
        comando.erase(0,comando.find_first_not_of(t));
        comando.erase(comando.find_last_not_of(t) + 1);
        // Ejecución del comando
        if ( comando == "list" ){
            std::cout << listCalendario(calendario) << std::endl;
        } else if ( comando == "pop" ){
            std::cout << listPrimerElementoCalendario(calendario)
                      << std::endl;
            if (calendario.size()>0)
                calendario.pop_front();
        } else if ( comando == "clear" ){
            calendario.clear();
        } else if ( comando == "end" ){
            std::cout << "Programa finalizado" << std::endl;
            break;
        } else {
            try {
                Evento ev = isEvento(comando);
                insertaEvento(calendario,ev);
            } catch ( std::invalid_argument &exc){
                std::cout << "Comando desconocido\n";
            }
        }
    } while (true);
    return 0;
}
```

EJERCICIO 3

Consideremos un sistema radial de carreteras como el mostrado en la figura, que está compuesto por seis carreteras, nombradas A1, A2, . . . , A6, que parten del punto M. La señalización de los puntos kilométricos se realiza en cada una de las carreteras suponiendo que el punto M está situado en el kilómetro cero.



En un fichero de texto llamado *puntosRecarga.txt* se encuentra información sobre la ubicación de los puntos de recarga para vehículos eléctricos que hay en las seis carreteras. Cada fila del fichero describe un punto de recarga:

- En la primera columna está escrita una cadena de caracteres compuesta por letras y números, que identifica el punto de recarga.
- En la segunda y tercera columnas están escritas el nombre de la carretera (A1, A2, . . . , A6) y el punto kilométrico de dicha carretera en el cual está situado el punto de recarga, respectivamente. El punto kilométrico es un valor entero mayor que cero.

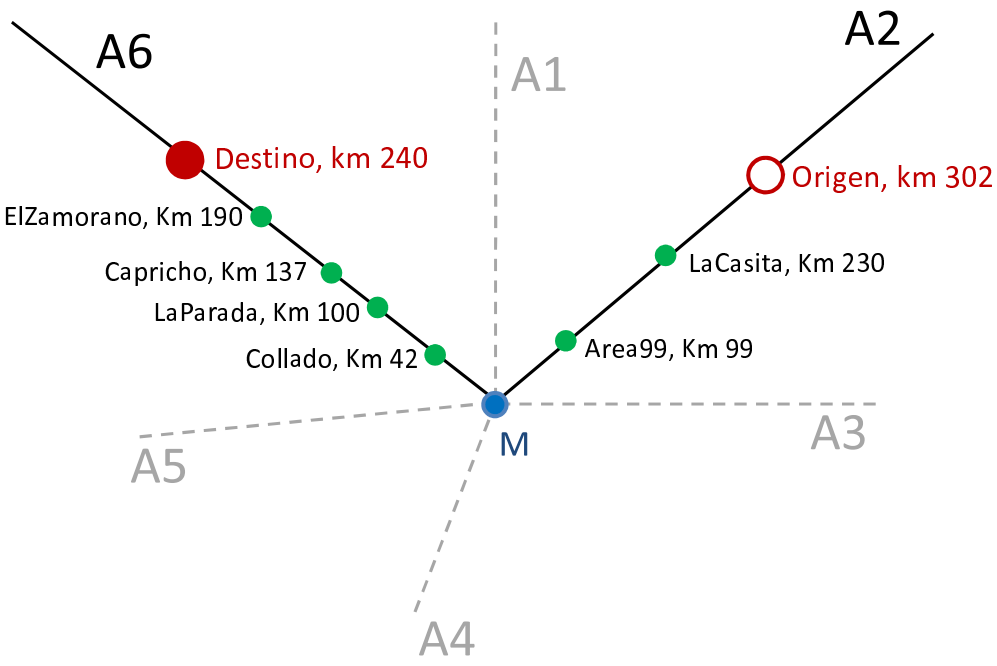
Las primeras filas de *puntosRecarga.txt* podrían ser, por ejemplo:

```
Area99      A2   99
Collado     A6   42
LaOrensana  A6  505
Desfiladero A4  265
Tarancon    A3   86
```

Escriba un programa en C++ que, para un determinado punto origen y un determinado punto destino especificados por el usuario a través de la consola, y

haciendo uso de la información almacenada en el fichero *puntosRecarga.txt*, escriba en un fichero llamado *misPuntosRecarga.txt* la información de los puntos de recarga situados en la ruta que une el punto origen con el punto destino, añadiendo asimismo una cuarta columna en la cual se especifique la distancia en kilómetros entre el punto origen de la ruta y el correspondiente punto de recarga. Los puntos de recarga deben aparecer en el fichero *misPuntosRecarga.txt* en orden creciente de distancia al punto origen de la ruta. Además, en la primera fila del fichero debe especificarse el origen de la ruta y en la última fila el destino.

A continuación se muestra un ejemplo. Supongamos que el usuario especifica que el punto origen de su ruta es el punto kilométrico 302 de la carretera A2, y el destino es el punto kilométrico 240 de la carretera A6. Los puntos de recarga situados en la ruta entre el punto origen y el punto destino están señalados mediante puntos verdes en la figura (esta información debe ser extraída por el programa del fichero *puntosRecarga.txt*).



El fichero *misPuntosRecarga.txt* generado por el programa en este caso debería ser:

```
Origen  A2  302  0
LaCasita  A2  230  72
Area99  A2  99  203
Collado  A6  42  344
LaParada  A6  100  402
Capricho  A6  137  439
ElZamorano  A6  190  492
Destino  A6  240  542
```

Solución al Ejercicio 3

```
#include <iostream>
#include <string>
#include <fstream>
#include <list>
#include <sstream>
#include <algorithm>
#include <stdexcept>
#include <cmath>

const std::string puntosFich    = "puntosRecarga.txt";
const std::string misPuntosFich = "misPuntosRecarga.txt";

struct PuntoRecarga {
    std::string ident;
    std::string carretera;
    int km;
    int kmAlOrigen;
};

struct Ruta {
    std::string carreteraOrigen;
    int kmOrigen;
    std::string carreteraDestino;
    int kmDestino;
};

bool esCarretera(std::string nombreCarretera){
    return   nombreCarretera.size()==2  &&
           nombreCarretera[0]=='A'    &&
           nombreCarretera[1]>='1' && nombreCarretera[1]<='6';
}

Ruta entradaRuta(){
    Ruta ruta;
    do {
       std::cout << "Introduzca origen\nCarretera: ";
       std::cin >> ruta.carreteraOrigen;
    } while (!esCarretera(ruta.carreteraOrigen));
    do {
       std::cout << "Km: ";
       std::cin >> ruta.kmOrigen;
    } while (ruta.kmOrigen<0);
    do {
       std::cout << "Introduzca destino\nCarretera: ";
       std::cin >> ruta.carreteraDestino;
    } while (!esCarretera(ruta.carreteraDestino));
    do {
       std::cout << "Km: ";
       std::cin >> ruta.kmDestino;
    } while (ruta.kmDestino<0);
    return ruta;
}
```

```
std::list<PuntoRecarga> getPuntosRecarga(std::string carretera,
        int kmOrigen, int kmDestino) throw (std::invalid_argument){
// Esta función devuelve en una lista los puntos de recarga existes entre el
// kmOrigen y el kmDestino de la misma carretera (la pasada como argumento)
// La lista está ordenada en orden creciente de distancia al kmOrigen

    // Apertura y lectura de fichero
    std::ifstream inFich(puntosFich.c_str(), std::ios::in);
    if (!inFich)
        throw std::invalid_argument ("ERROR al abrir " + puntosFich);
    // Inserta en la lista los puntos de recarga entre origen y destino
    std::list<PuntoRecarga> listaPuntosRecarga;
    while ( !inFich.eof() ) {
        PuntoRecarga punto;
        inFich >> punto.ident;
        if (inFich.eof()) break;
        inFich >> punto.carretera >> punto.km;
        if ( punto.carretera == carretera &&
             punto.km < std::max(kmOrigen,kmDestino) &&
             punto.km > std::min(kmOrigen,kmDestino) ) {
            punto.kmAlOrigen = std::abs(punto.km - kmOrigen);
            std::list<PuntoRecarga>::iterator p =
                                listaPuntosRecarga.begin();
            while ( p != listaPuntosRecarga.end() ) {
                if ( p->kmAlOrigen > punto.kmAlOrigen ) {
                    listaPuntosRecarga.insert(p,punto);
                    break;
                }
                p++;
            }
            if (p == listaPuntosRecarga.end())
                listaPuntosRecarga.push_back(punto);
        }
    }
    inFich.close();
    return listaPuntosRecarga;
}

void escrituraFichero(std::list<PuntoRecarga> listaPuntosRecarga)
            throw (std::invalid_argument){
    std::ofstream file_out(misPuntosFich.c_str(),
                        std::ios::out | std::ios::trunc);
    if (!file_out)
        throw std::invalid_argument ("ERROR al escribir en " +
                        misPuntosFich);
    std::list<PuntoRecarga>::iterator p = listaPuntosRecarga.begin();
    std::stringstream ss;
    while ( p != listaPuntosRecarga.end() ) {
        ss    << p->ident       << "  "
          << p->carretera   << "  "
          << p->km          << "  "
          << p->kmAlOrigen  << "\n";
        p++;
    }
    file_out << ss.str();
    file_out.close();
    return;
}
```

```
int main() {
    // Entrada por consola de puntos origen y destino
    Ruta ruta = entradaRuta();
    PuntoRecarga origen =
                    {"Origen", ruta.carreteraOrigen, ruta.kmOrigen, 0};
    try {
        if (ruta.carreteraOrigen == ruta.carreteraDestino) {
            std::list<PuntoRecarga> listaTramo =
              getPuntosRecarga(ruta.carreteraOrigen, ruta.kmOrigen,
                  ruta.kmDestino);
            PuntoRecarga destino = {"Destino",
                ruta.carreteraDestino, ruta.kmDestino,
                std::abs(ruta.kmOrigen-ruta.kmDestino)};
            listaTramo.push_front(origen);
            listaTramo.push_back(destino);
            escrituraFichero(listaTramo);
        } else {
            std::list<PuntoRecarga> listaTramo1 =
              getPuntosRecarga(ruta.carreteraOrigen, ruta.kmOrigen, 0);
            listaTramo1.push_front(origen);
            std::list<PuntoRecarga> listaTramo2 =
              getPuntosRecarga(ruta.carreteraDestino, 0, ruta.kmDestino);
            PuntoRecarga destino = {"Destino", ruta.carreteraDestino,
              ruta.kmDestino, ruta.kmDestino};
            listaTramo2.push_back(destino);
            std::list<PuntoRecarga>::iterator p = listaTramo2.begin();
            while ( p != listaTramo2.end() ) {
               p->kmAlOrigen += ruta.kmOrigen;
               p++;
            }
            listaTramo1.insert(listaTramo1.end(),
                                listaTramo2.begin(), listaTramo2.end());
            escrituraFichero( listaTramo1 );
        }
    } catch ( std::invalid_argument &exc ) {
        std::cout << exc.what() << std::endl;
    }
    return 0;
}
```

EJERCICIO 4

Supongamos que se desea realizar una ruta entre un punto origen y un punto destino, empleando un coche eléctrico cuya autonomía son N kilómetros. Que la autonomía del coche sean N kilómetros significa que el coche puede recorrer N kilómetros sin necesidad de recargar.

El coche inicia la ruta con la batería completamente cargada.

Si la longitud de la ruta es menor o igual a N, el coche podrá realizar la ruta completa sin recargar su batería.

Si la longitud de la ruta es mayor que N, será preciso realizar una o varias paradas de recarga de la batería.

Escriba un programa en C++ que realice las acciones siguientes.

1. Solicitar por consola al usuario que éste introduzca por consola la autonomía (N) del vehículo, la cual es un número entero mayor que cero.
2. Leer el fichero *misPuntosRecarga.txt* correspondiente a la ruta a realizar. Este fichero, que se supone que está disponible, tiene el formato descrito en el Ejercicio 3. Si se produce error en la lectura del fichero, el programa debe mostrar un mensaje en la consola indicándolo y terminar.
3. De los datos anteriores, escribir en la consola si es posible realizar la ruta y, en caso afirmativo, cuál es el número mínimo de recargas necesario para ello.
4. Terminar.

Solución al Ejercicio 4

```
#include <iostream>
#include <string>
#include <fstream>
#include <vector>
#include <stdexcept>

const std::string misPuntosFich = "misPuntosRecarga.txt";

struct PuntoRecarga {
    std::string ident;
    std::string carretera;
    int km;
    int kmAlOrigen;
};

std::vector<PuntoRecarga> leeMisPuntosRecarga()
                          throw (std::invalid_argument) {
    std::ifstream inFich(misPuntosFich.c_str(), std::ios::in);
    if (!inFich)
        throw std::invalid_argument ("ERROR al abrir " +
                                misPuntosFich);
    std::vector<PuntoRecarga> misPuntosRecarga;
    while ( !inFich.eof() ) {
        PuntoRecarga punto;
        inFich >> punto.ident;
        if (inFich.eof()) break;
        inFich >> punto.carretera >> punto.km >> punto.kmAlOrigen;
        misPuntosRecarga.push_back(punto);
    }
    inFich.close();
    return misPuntosRecarga;
}
```

```
int main() {
    int N;
    do {
        std::cout << "Autonomia (km): ";
        std::cin >> N;
    } while (N <= 0);
    std::vector<PuntoRecarga> misPuntosRecarga;
    try {
        misPuntosRecarga = leeMisPuntosRecarga();
    } catch (std::invalid_argument &exc) {
        std::cout << exc.what() << std::endl;
        return 0;
    }
    // Es posible hacer la ruta?
    for (unsigned int i=0; i<misPuntosRecarga.size()-1; i++) {
        int distancia =
            misPuntosRecarga[i+1].kmAlOrigen - misPuntosRecarga[i].kmAlOrigen;
        if ( distancia > N ) {
            std::cout << "No es posible realizar la ruta\n"
                  << "La distancia entre "
                  << misPuntosRecarga[i].ident
                  << " y " << misPuntosRecarga[i+1].ident <<
                  " son " << distancia << " km" << std::endl;
            return 0;
        }
    }
    // Número mínimo de recargas
    int kmUltimaRecarga = 0;
    int numeroRecargas = 0;
    for (unsigned int i=0; i<misPuntosRecarga.size(); i++) {
        if (misPuntosRecarga[i].kmAlOrigen - kmUltimaRecarga > N) {
            kmUltimaRecarga = misPuntosRecarga[i-1].kmAlOrigen;
            numeroRecargas++;
        }
    }
    std::cout << "Es posible realizar la ruta.\n"
          << "Numero minimo de recargas: " << numeroRecargas
          << std::endl;
    return 0;
}
```