1º trabalho de Física Computacional

Fernando Barao (Dep. Física, IST)

Instruções

A resolução dos exercícios deve ser feita:

- na zona de trabalho *svn* de cada grupo, criando para tal o directório **FC/2014/<groupname>/Trab1**

```
cd FC/2014/<groupname>
svn mkdir Trab1
svn ci -m "directorio Trab1 criado"
```

- deve ser acompanhada de uma memória descritiva escrita num ficheiro latex e convertida para pdf a sumariar o raciocínio e a resolução de cada alínea, a inserir no mesmo directório. Este documento não deve ser extenso

```
ficheiro tex: groupXXTrab1.tex
ficheiro pdf: groupXXTrab1.pdf
```

- o código implementado deve conter os comentários necessários e suficientes à sua compreensão

Este trabalho deve ser entregue, isto é, *svn commited*, até às 09H00 do dia 27 de Outubro de 2014.

[Exercício 1.]

Um polinómio de uma só variável x e grau n é uma função que se pode expressar da seguinte forma:

$$P(x) = \sum_{i=0}^{n} a_i \, x^i$$

Defina uma classe C++ **Pol** que trate polinómios de grau n qualquer e onde estejam definidas as operações adição/subtracção de polinómios, multiplicação de polinómios e multiplicação de um polinómio por um escalar. Defina:

a) os *constructores da classe* que julgar necessários de forma a que os objectos da classe se possam instanciar das seguintes formas:

```
//criar polinómio de grau zero e coeffs nulos
 Pol P;

//criar polinómio de grau 5 e coeffs nulos
 Pol P5("P5",5);

 //criar o polinómio de grau 9: 2+x+5x^3+5x^5+2x^4+4x^6+9x^7+8x^8+x^9
 int a[] = {...}; //define os coefs do polinomio num array de int
 Pol Q("Q",9,a);

 //criar polinomio de grau 3 colocando os coefs num vector
 vector<int> b(4);
 Pol V("V",3,b);
```

b) os seguintes métodos da classe necessários de forma a que o utlizador no seu programa possa definir o polinómio sem recorrer ao construtor:

```
void SetPolynomial(int, int*)
void SetCoefficient(int, int)
```

c) os métodos necessários na classe de forma a que seja possível realizar as seguintes operações com os objectos polinómio:

```
 int a[] = {...};
 Pol P("P",7,a); //criar polinómio de grau 7 e nome P
 Pol Q(P); //nome da copia: _P

 Pol *T = new pol(P);
 Pol U(T);

 Pol V;
 V = Q;
```

```
Pol A(P), C(P);
A += Q;
Pol B = V+Q;
C -= Q;
Pol D = V-Q;

V.Scale(4); //4*V
B *= Q; //B=B*Q
```

Nota: tenha em conta que o nome da cópia do polinómio deve incluir o nome original com um *underscore* (_) prévio.

d) métodos que permitam imprimir os polinómios de acordo com o seguinte código exemplo:

```
//criar polinómio
 Pol P("P",7,a);
 p.Print();  // este metodo imprime o seguinte:
             // polynomial:  P[7] coeffs = ...
 p.PrintPol();
            // este metodo imprime o seguinte:
            // polynomial:  P[7]  = 2*X^0 +3*X^1 + 5*X^2 -5X^4 +...
```

e) Tendo em conta as definições dos polinómios,

$\boldsymbol{P_1(x) = 2x^2 + 4x^3 + x^5 + x^8}$

$\boldsymbol{P_2(x) = x^4 - 2x + 4}$,

determine os resultados para as seguintes operações:

1. $\boldsymbol{P(x) = 4P_1(x)}$
2. $\boldsymbol{P(x) = P_1(x) + P_2(x)}$
3. $\boldsymbol{P(x) = P_1(x) * P_2(x)}$

Escreva um programa principal **TPolop.C** onde estas operações sejam realizadas e os resultados mostrados de forma compreensível no monitor do computador.

f) Defina os métodos da classe **Evaluate, Derivative** que permitam calcular o valor do polinómio e da sua derivada, para um valor de $\boldsymbol{x}$ qualquer.

```
double Evaluate(double);
const Pol Derivative();
```

g) Escreva um programa principal **TPol8.C** onde se defina o polinómio de grau 8,

$$\boldsymbol{P(x) = x^8 - 8x^7 + 28x^6 - 56x^5 + 70x^4 - 56x^3 + 28x^2 - 8x + 1}$$

e no qual se calcule o valor do polinómio, usando o método *Evaluate*, no intervalo $[\mathbf{0.99996, 1.0001}]$ com um passo de $\mathbf{10^{-5}}$. Calcule a diferença entre o valor que encontra e o valor de referência que é dado pela função *static* **P8** da classe **cTrab1** existente na biblioteca de XX=32 ou 64 bits, **FC/LIBs/libFC_x86_XX.a** para linux ou **FC/LIBs/libFC_mac.a** para mac:

```
class cTrab1 {
  public:
    static double P8(x);
    ...
};
```

Será possível optimizar a codificação do polinómio (e do método *Evaluate*, portanto) de forma a obter-se ume maior precisão no seu resultado?

h) Descreva quais as instruções que usaria para construir a biblioteca **lib2014<groupame>.a**, com base na classe **Pol** desenvolvida neste exercício.

Sumário:

```
No final deste exercício, os seguintes códigos devem existir
no directório FC/2014/<groupname>/Trab1:
-> Pol.h, Pol.C: classe Pol
-> TPolop.C    : programa para realizar operações sobre os polinómios
-> TPol8.C     : programa que realiza as operações da alínea g)
```

[Exercício 2.]

a) Neste exercício pretende-se desenvolver uma classe C++ cujo nome será **MPol** que permita gerir uma lista de polinómios de forma dinâmica, usando a classe **Pol** desenvolvida no exercício anterior. A classe não deve armazenar as expressões dos polinómios; somente os objectos *Pol*.

A classe deve cumprir os seguintes requisitos:

Ler as expressões dos polinómios a partir de um ficheiro

```
vector<string> ReadFile(string);
```

Descodificar os valores dos coeficientes e das potências do polinómio

```
Pol Convert(string);
```

Adicionar o polinómio ao fim da lista existente

```
void AddPol(Pol&);
```

Remover o polinómio da lista

```
void DelPol(string);
```

Encontrar um dado polinómio

```
Pol FindPol(string);
```

Listar o conteúdo da pilha de polinómios

```
vector<Pol> List();
```

Devem ser utilizados os polinómios existentes no ficheiro **Pol.txt**:

```
P = 4*X^4 -5*X^6 +2*X^0 -X^1 +2*X^3
Q = 2*X^5 -3*X^9 +4*X^2 +X^1
R = 2*X^5 +1*X^0 +6*X^1
S = 4*X^4 -5*X^6 +2*X^0 -X^1 +2*X^3
T = X^2 +2*X^1 +2*X^0
U = -2*X^2 +4*X^1 -5*X^4 +8*X^9 -12*X^7 +7*X^6
```

b) Desenvolva um programa principal cujo nome será **ReadPol.C** que execute as seguintes tarefas:

- Construir uma lista de polinómios, através da leitura do ficheiro *Pol.txt* e usando a classe desenvolvida **MPol**.
  Liste os polinómios existentes na pilha, imprimindo as suas expressões no sentido crescente das potências e usando o mesmo tipo de formato do ficheiro *Pol.txt*.
- Realize agora operações sobre três polinómios. Recorra à função **GetPolOperations** existente na biblioteca de 32 ou 64 bits (XX=32 ou 64) **FC/LIBs/libFC_x86_XX.a** para linux ou **FC/LIBs/libFC_mac.a** para mac, para conhecer as operações que deverá realizar.

  Neste exemplo, mostra-se como obter as operações a realizar nos polinómios para o grupo A01.

```
cTrab1 C; //instanciar a classe cTrab1
//exemplo: a string s contem as operações a realizar pelo grupo A01
string s = C.GetPolOperations(A01);
```

  Adicione à lista de polinómios, os polinómios resultantes das operações entre o $\mathbf{1}^{\underline{o}}$ e $\mathbf{2}^{\underline{o}}$ polinómios da operação, entre o $\mathbf{2}^{\underline{o}}$ e $\mathbf{3}^{\underline{o}}$ polinómios da operação e ainda o resultado final. Liste os polinómios existentes na pilha após estas operações (da mesma forma que na 1a alínea).
- Remova da lista os polinómios Q e T.
  Liste os polinómios existentes na pilha após esta operação (da mesma forma que na 1a alínea).

Sumário:

```
No final deste exercício, os seguintes códigos devem existir
no directório FC/2014/<groupname>/Trab1:
-> MPol.h, MPol.C: classe MPol
-> ReadPol.C     : programa
```

[Exercício 3.]

As matrizes são muito utilizadas em métodos numéricos, por exemplo na resolução de um sistema de equações. A ideia deste exercício é desenvolver classes que permitam a operação de matrizes bi-dimensionais que armazenam números de precisão *double*.
Uma matriz bi-dimensional é caracterizada por um certo número de linhas, n e de colunas, m. Tomemos as seguintes matrizes:

$$A = \begin{bmatrix} 2 & 0 & 5 & -0 \\ 0 & -7 & 0 & -2 \\ 1 & 0 & 3 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 10 & 5 & 0 \\ 5 & 0 & 8 & -2 \\ 1 & -5 & 0 & +6 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 3 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 1.0 & 7.0 & 5.0 & 3.0 & -3.0 \\ 5.0 & 2.0 & 8.0 & -2.0 & 4.0 \\ 1.0 & -5.0 & -4.0 & 6.0 & 7.6 \\ 0.0 & -5.0 & 3.0 & +3.2 & 3.3 \\ 1.0 & 7.0 & 2.0 & 2.1 & 1.2 \end{bmatrix}$$

a) A definição de uma matriz é mais facilmente implementável usando uma classe que armazene os elementos lineares da matriz, linha ou coluna.
Vamos por isso começar por definir uma classe **Vec** cujos *data members* são,

```
class Vec {
public:
 ...
private:
  int N; //number of elements
  double *entries; // pointer to array of doubles
};
```

Os construtores desta classe devem ser tais que nos permitam a construção dos vectores usando as seguintes formas:

```
#include "Vec.h"
...
Vec v1(10); //array with 10 values set to zero
Vec v2(10,5.); //array with 10 values set to 5.

double a[]={1.2, 3.0, 5.4, 5.2, 1.0};
Vec v1(5,a); //array with 5 values given by "a" pointer

Vec v2(v1); //define a vector by using another one
```

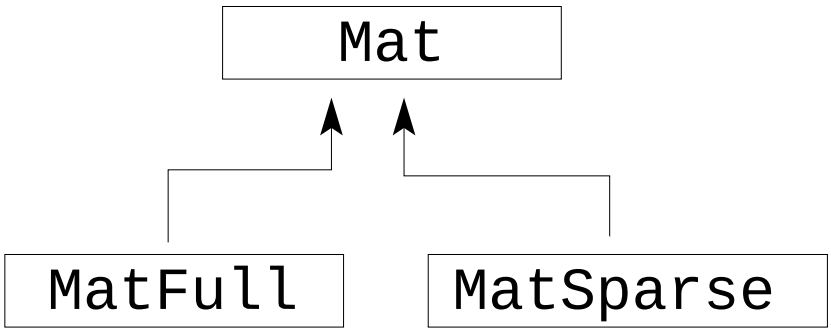
Deve ser definido ainda o **overloading de operadores** de forma que possamos:

- igualar dois vectores (=)
- somar dois vectores (+=, +)
- subtrair dois vectores (-=, -)
- aceder a um elemento i do vector através de $v[i]$
- poder fazer o negativo (-) ou o positivo (+) do vector
- multiplicar dois vectores (a[i] = b[i]*c[i])

Devem ser também definidos os métodos **Size, Scale, Dot** que permitirão respectivamente:

- obter a dimensão do vector
- multiplicar o vector por um escalar
- fazer o produto interno com outro vector

b) Após a definição da classe **Vec** podemos agora definir as matrizes. A ideia que aqui se propõe é a de definir uma classe base **Mat** e implementar os aspectos particulares em classes que herdem desta. Em particular, e sem ser exaustivo, as matrizes podem ser definidas através de duas classes: a classe **MatFull**, que armazenará todos os elementos da matriz, e a classe **MatSparse**, onde de forma optimizada somente os elementos não nulos da matriz são armazenados.



De forma a simplificar o problema proponho a seguinte classe de base onde se definem como puramente virtuais todos os métodos.

```
class Mat {
public:
  virtual Vec  operator*(const Vec&) const = 0; //multiply matrix by vector
  virtual Vec& GetRow(int) const = 0; //get row vector
  virtual Vec& GetCol(int) const = 0; //get column vector
  virtual void Print() const = 0; //print formatted matrix
  virtual Mat  operator*(const Mat&) const = 0; //multiply matrix by matrix
protected:
  int nrows; //nb of rows
};
```

A classe **MatFull** terá os seguintes *data members*:

```
private:
  int ncols; //nb of columns
  Vec *mvec; //pointer to an array of Vec objects
```

e devem ser definidos os constructores necessários para que esta possa ser instanciada das seguintes formas:

```
// M1 is a matrix with 4 rows and 5 columns and all elements set to zero
MatFull M1(4,5,0.);

// M2 is a matrix (4,5) and a is now a pointer to an array of objects Vec
MatFull M2(4,5,a);

// M3 is created and made similar to M2
MatFull M3(M2);
```

A ideia do armazenamento *sparse* da matriz passa pelo facto de se evitar o armazenamento dos elementos nulos da matriz. Há várias formas de armazenamento propondo-se aqui o chamado *compressed sparse row format*. Os elementos não nulos da matriz, por hipótese em número NN, são armazenados em três *array* 's uni-dimensionais: um *array* **a** de dimensão NN que armazena os elementos não nulos linha a linha (da esquerda para a direita e de cima para baixo); um *array* **b** de dimensão NN que armazene os índices das colunas dos elementos armazenados em **a**, ou seja, o elemento a[i] encontra-se na coluna b[i]; e um último *array* **c** de dimensão (nrows+1) que armazena para cada linha i no elemento c[i] a posição no *array* **a** do primeiro elemento não nulo dessa linha. O último elemento do array é o número de elementos não nulos, NN. Tenha atenção na definição do *array* **c**, os casos em que pode existir uma linha inteira de zeros na matriz.

A classe **MatSparse** terá os seguintes *data members*:

```
private:
  int ncols;
  int NN; //number of nonzero entries of the matrix
  double *a;
  int *b;
  int *c;
```

Devem ser definidos os constructores necessários para que esta classe possa ser instanciada das seguintes formas:

```
// M1 is a matrix with 4 rows and 5 columns and set all elements
double *a;
int *b, *c;
MatSparse M1(4,5,a,b,c);

// M2 is created and made similar to M1
MatSparse M2(M1);

// assignment
M2 = M1;

// from matrix MatFull Ma
```

```
MatSparse M3(Ma);
```

c) Construa um programa **Tmatrix.C** onde se realizem as seguintes operações:

- obter a matriz A no formato *sparse*, partindo do formato *full* e usando o constructor respectivo (sparse)
- imprimir os elementos da matriz sparse A
- obter a matriz B no formato *sparse*, partindo do formato *full* e usando o constructor respectivo (sparse)
- imprimir os elementos da matriz sparse B
- multiplicar as matrizes $\boldsymbol{C} \times \boldsymbol{B}$ e imprimir o resultado

Sumário:

```
No final deste exercício, os seguintes códigos devem existir
no directório FC/2014/<groupname>/Trab1:
-> Vec.h, Vec.C                                          : classe Vec
-> Mat.h, MatFull.h, MatFull.C, MatSparse.h, MatSparse.C: classes matriz
-> Tmatrix.C                                             : programa
```

Fernando Barao
Dep. Fisica IST
20 de Outubro de 2014