IST- MEFT - Física Computacional - 27/10/2014
Prof. Fernando Barão
**Grupo B09**

Pedro Pereira: 78889
João Alves: 79006

**1º Trabalho Física Computacional - Memória Descritiva**

**Resumo**

Neste trabalho, os principais problemas foram derivados de memory leaks e alocações de memórias incorretas.

# 1 Polinómios

Este exercício familiarizou-nos com a definição de classes e dos seus operadores e métodos. Além disso levou-nos a refletir sobre métodos numéricos e formas de minimização de erro na definição da função evaluate. Para construir a biblioteca referida na alínea h) deste exercício, procederíamos da seguinte forma:
g++ -c Pol.C
ar ruv lib2014B09.a Pol.o
ranlib lib2014B09.a

## 1.1 Construtores

Foi necessário definir um default constructor, que permitisse inicializar um polinómio zero sem que fosse necessária a inserção de polinómios. Para além deste, foram definidos outros construtores que permitem a inicialização de um polinómio pretendido.

## 1.2 Métodos

O principal cuidado que se teve na execução dos métodos desta classe foi o de comparar o grau dos polinómios intervenientes, garantindo sempre a aplicação do método para valores alocados de memória. Isto é, quando se adicionam dois polinómios, tem-se o cuidado de não adicionar coeficientes de maior grau que o do mais pequeno, de modo a não ler valores inválidos de memória.

Na alinea g, estudou-se o método evaluate para o polinómio dado, tendo os valores por ele determinados um erro relativo bastante elevado. Isso deriva da existência de um cancelamento subtrativo perto do valor onde se pretende calcular a função. É possível otimizar pois, o número de condição do problema é dado por $P(f,x) = \frac{x.f'(x)}{f(x)}$, sendo que $f(x) = (x-1)^8$, o limite quando x tende para 1 é $8<\infty$, logo está bem condicionado. Para resolver este problema, coloca-se o polinómio na forma $P(x) = (x-z)^k * R(x)$, onde R(x) não se anula para z. Percebemos isto, foi verificámos que o polinómio dado era convertível em $P(x) = (x-1)^8$, calculando o valor usando essa expressão, obtém-se a solução numérica igual á que nos foi pedida para comparar. Assim, reduz-se o erro relativo do valor calculado. Para encontrar R(x), aplica-se o método de Ruffini sucessivamente a P(x). Infelizmente, por questões de tempo não foi possível a execução total deste método. À esquerda do zero conseguimos obter excelentes aproximações, contudo á direita continua a ter alguns problemas.

# 2 Lista de Polinómios

## 2.1 Construtor

Foi necessário definir um construtor default, de forma a criar um objeto genérico pelo qual se pode aceder aos métodos e às variáveis privadas.

## 2.2 Métodos

Definir os métodos exigidos foi relativamente simples, e a sua explicação nos comentários é suficiente para a compreensão, excepto o caso do método Convert. O

trabalho nesta secção consistiu essencialmente numa pesquisa das funções da biblioteca fstream e cstdlib que nos fossem úteis. No método convert,inicialmente procurou-se o x. Guardou-se a posiçao do x num vetor, percorreu-se cada x e andou-se pra direita até aparecer um número, continuando até aparecer um espaço. Seguidamente, guardou-se os carateres entre ambos e converteu-se para número, isso era o grau. Fez-se o mesmo para a esquerda e isso era o coeficiente. Teve de se considerar os casos em que estava só um + ou só um - ou nada, depois ordenou-se, e preencheu-se com 0 os graus que não tinham coeficiente. No caso do ReadPol, foi usada a função ReadFile e um convert consecutivo a cada string i do vetor de strings retornado por ReadFile. Seguidamente, procedemos segundo o enunciado e obtivémos que a operação a realizar entre polinómios era: +R - S + P. Realizaram-se essas operações, utilizando as operações definidas em Pol, adicionando os valores parciais ao objeto de MPol genérico ao qual vai ser aplicado o método List de forma a imprimi-los. Removeu-se os polinómios pedidos usando a função DelPol definida em MPol e criou-se outro vetor de Pols, ao qual se aplicou List novamente.

# 3 Vetores e Matrizes

## 3.1 Vetores

Passou por 2 fases a realização desta secção.

### 3.1.1 Construtores

Em relação aos construtores e aos métodos Size,Scale e Dot, foi relativamente simples defini-los.

### 3.1.2 Métodos

Na fase inicial, a maior dificuldade foi definir os operadores +/- e +=/-= duma forma eficiente. Encontrou-se a solução de definir cuidadosamente +=/-= e definir +/- à custa destes. Como se pode verificar abaixo para os casos da soma.

```
const Vec& Vec::operator+=(const Vec& v) {
//teste de dimensão aqui
  for (int i=0;i<N; ++i)
    entries[i]+=v.entries[i];

return *this;   }
```

```
const Vec Vec::operator+( const Vec& v) {
 Vec w(*this);
 w+=v;
 return w;   }
```

A 2ª fase consistiu em criar funções que facilitassem a manipulação/alocação de memória dos Vecs para facilitar-nos a vida nas classes derivadas de Mat. Foi nesta fase que surgiram as funções SetN, Setentries, GetN e Getentries. Foi também necessário definir um construtor default Vec, de forma a tornar possível a construção de arrays do tipo Vec.

## 3.2 MatFull e MatSparse

### 3.2.1 Construtores

Assumiu-se que cada vetor era uma linha da matriz, sendo a matriz definida por um array de Vecs. Definir os construtores desta classe deu, comparativamente à anterior, mais problemas. Tal como já dito, foi necessário construir-se funções extra em Vec devido a problemas de alocação de memória que surgiam a instanciar os objetos MatFull.

Por exemplo, para o construtor do tipo M(nrows,ncols,a) com 'a' a ser um ponteiro para um array de vetores, inicialmente alocava-se memória para um número de vetores, i, igual ao número de linhas da matriz, e igualava-se cada mvec[i] a[i]. Surgiam problemas porque não se dava indicação do número de entradas que cada vetor teria. Foi daí que surgiu a necessidade de criar as funções SetN e Setentries, que corrigiram o problema. O mesmo problema teria surgido na MatSparse, contudo já iamos preparados.

### 3.2.2 Métodos

Neste caso existiram duas fases distintas na definiçao dos métodos. Inicialmente, fizeram-se os métodos da

MatFull, sendo que o facto de a matriz estar completamente preenchida facilitou bastante este processo. É apenas importante referir que foi utilizado o produto interno para multiplicar matrizes, uma vez que cada entrada de uma matriz corresponde ao produto de uma linha da primeira por uma coluna da segunda. Posteriormente, realizaram-se os métodos da MatSparse. Tendo sido definida uma função SparseToFull, que partindo da MatSparse define um vetor double** a partir do qual se constrói a correspondente MatFull. Assim, reutilizam-se os métodos definidos para a MatFull, facilitando bastante a execução do programa. Um método mais eficiente para a multiplicação usando MatSparse seria, possivelmente, utilizar o vetor com as entradas não nulas da matriz e verificar com o vetor com os indíces quais as entradas que multiplicam, guardando os resultados noutro vetor, sendo todos as outras entradas zero. Mais uma vez não implementámos este método por falta de tempo, mas ao cogitar esta solução demonstra que entedemos o conceito por trás da MatSparse. No final de tudo, surgiu um problema a fazer print das matrizes no enunciado do exercício. Sempre que a última entrada da matriz era 0, no caso da matriz A e B, dava um erro de memória. A solução encontrada foi definir esse valor como $1 * 10^{-200}$, que, é uma aproximação aceitável, sendo assim o erro do produto das matrizes muito pequeno. Também é de acrescentar que tentámos fazer o exigido pela classe Mat, isto é, pôr o $operador*$ a retornar $Mat\&$, dando assim uso ao polimorfismo do $C++$, mas devido a termos implementado o operador multiplicação antes disso, quando fomos verificar se ainda estava tudo ok, percebemos que teríamos que redefinir toda a função e repensar parte do nosso raciocínio. Então, por falta de tempo, limitámo-nos a retornar MatFull e MatSparse.

## Referências


[1] *Stroustrup, Bjarne. The C++ Programming Language. Reading, MA: Addison-Wesley, 2013.*

[2] *Barão, Fernando. Slides das Aulas Teóricas. 2014.*