Alguns códigos lineares com o μ–pad

Notas de apoio para Teoria de Códigos

Pedro Patrício*

Maio de 2006

Conteúdo

*Departamento de Matemática da Universidade do Minho; pedro@math.uminho.pt

1 Documentação

O *MuPAD* está disponível em

http://www.mupad.de

para *download*. As versões com licença para estudante (gratuita) são a 2.5.x para Windows (*light*) e 3.x para Linux. Uma licença de estudante pode ser pedida aqui:

https://www.mupad.org/muptan/muptan.php

Espera-se que de forma temporária, estas licenças não estão actualmente a ser distribuidas. Aconselha-se vivamente a visita e leitura de

http://fuchssteiner.info/

No *site* do *MuPAD* também pode encontrar alguma documentação. Por exemplo, a referente à versão 3.1 pode ser consultada aqui:

http://research.mupad.de/doc/31/eng/index.html

Por exemplo, se pretender saber como construir um corpo de Galois e manipular os seus elementos, pode usar a documentação online disponível em

http://research.mupad.de/doc/31/eng/Dom_GaloisF.html

O mesmo pode ser feito em relação à versão 2.5:

http://research.mupad.de/doc/25/eng/index.shtml

Na página da disciplina, pode também encontrar um guia muito rápido, da autoria de W. Oevel e J. Gerhard.

Pode ainda usar os mirrors e respectiva documentação disponibilizados pelo Departamento de Informática da Universidade do Minho

http://ftp.di.uminho.pt/pub/apps/math/MuPAD/

2 Aquecendo os motores

Ao invés do que sucede em linguagem matemática (pelo menos na sua forma mais vulgar), tanto no mupad como noutras ferramentas computacionais urge distinguir de uma forma clara o símbolo que atribui um valor a uma variável do que inquirir sobre a igualdade entre duas expressões. O primeiro, que mais nos interessa neste momento, é escrito como `:=`.

```
>> x:=3;
```

3

Se se pretender que o valor de x não seja exibido, basta terminar a instrução com `:` em vez de `;`

```
>> x:=3:
>>
```

O valor de x pode ser mostrado digitando-o:

```
>> x;
```

3

Para deixar x sem mácula, apague-se a atribuição:

```
>> delete(x);
>> x
```

$$x$$

Uma função é definida do modo seguinte:

```
>> f:=x->x^8-1;
```

```
x -> x^8 - 1
```

Torna-se simples calcular a imagem de 2 por f:

```
>> f(2);
```

```
255
```

Em alternativa,

```
>> evalp(f(x),x=2);
```

255

Atente agora nas diferenças:

```
>> f;
```

```
x -> x^8 - 1
```

```
>> f(x);
```

```
x^8 - 1
```

Os zeros de f são calculados usando

```
>> solve(f(x));
```

Caso alguma dúvida surja ao utilizador, pode usar a ajuda imersa no mupad:

```
>> ?solve
```

A factorização de polinómios:

```
>> factor(f(x));
```

```
(x - 1) (x + 1) (x^2 + 1) (x^4 + 1)
```

O *mupad* também indica o número de factores:

```
>> nops(factor(f(x)))
```

Surpreendido? O factor i é indicado usando `factor(f(x))[`i`]`. Faça então um ciclo `for` de forma a serem apresentados todos os factores (use `print`, em, a título de exemplo, `print("factor",`i`, "e' ",factor(f(x))[`i`])`).

De referir que os comandos do mupad são *case sensitive*. Por exemplo,

```
>> FaCtOr(f(x));
```

```
FaCtOr(x^8 - 1)
```

É claro que o polinómio f não é irredutível já que tem factores, na sua decomposição, que são não constantes (e não nulos, diga-se de passagem, embora tal seja óbvio).

```
>> irreducible(f(x));
```

```
FALSE
```

3 Um pouco de aritmética (a tabuada revisitada)

Assuma que se pretende testar o código ISBN de um livro recebido. Antes de mais, o código ISBN-10 (investigue o que se passa com o ISBN-13) é uma sequência de 10 símbolos $x_1x_2\ldots x_9x_{10}$ onde $x_i = 0..9$, se $i = 1..9$, e $x_{10} \in \{0..9\} \cup \{X\}$. É indiferente a colocação dos hífenes: estes servem apenas para separar a codificação de certo tipo de informação – país ou língua de edição, número da editora, número do livro. O símbolo x_{10} é de facto o responsável pela detecção de erros – o símbolo de verificação. Tomando $X = 10$ no que se segue, estes têm que satisfazer

$$\sum_{i=1}^{10}(11-i)x_i \equiv 0 \mod 11.$$

Ou doutra forma, $(x_1, x_2, \ldots, x_{10}) \cdot (10, 9, 8, 7, 6, 5, 4, 3, 2, 1) \equiv 0 \mod 11$.

Por exemplo, e para simplificar, assuma recebido o suposto código 989-616072-4 (de que livro? pelo menos a língua deve tentar saber; http://www.isbn-international.org/en/identifiers/allidentifiers.html), dado como uma tabela:

```
>> cod:=table(1=9,2=8,3=9,4=6,5=1,6=6,7=0,8=7,9=2,10=4)
```

Faça `?table` para entender a sintaxe. Se preferir, pode usar o comando `array`:

```
>> cod:=array(1..10,[9,8,9,6,1,6,0,7,2,4]);
```

Podemos definir $\mathbb{Z}_{11}$ como

```
>> Z11:=Dom::IntegerMod(11);
```

Falta-nos agora verificar se o somatório é multiplo de 11, ou seja, se $\sum_{i=1}^{10} ix_i \equiv 0 \mod 11$. Antes é preciso calcular a soma:

```
>> sum((11-i)*cod[i],i=1..10)
```

Repare que `cod[i]` indica o elemento `i` da tabela `cod`. Falta agora estudar a congruência $\mod 11$

```
>> sum((11-i)*cod[i],i=1..10) mod 11
```

ou

```
>> Z11(sum((11-i)*cod[i],i=1..10))
```

Se não tivéssemos à disposição o comando `sum` ou um similar, então ter-se-ia optado, por exemplo, por um ciclo `for`. Veja a sintaxe com `?for`.

```
>> soma:=0:
>> for i from 1 downto 10 do
soma:=soma+(11-i)*cod[i]
end_for:
>> print(Z11(soma));
```

As mudanças de linha são feitas por `shift+return`, se estiver a usar o *x-mupad* em linux, ou o *mupad* no Windows.

É escusado referir a importância que ciclos como

```
for i from inicio to fim <step tamanhodosalto> do ..  end_for
repeat ..  until condição end_repeat
while condição do ..  end_while
```

ou instruções de decisão como

```
if condição
then ..
<elif condição then ..>
<elif condição then ..>
<...>
<else ..>
end_if
```

têm importância quase vital na construção de códigos.

4 Matrizes e afins

O comando `matrix` cria uma matriz de ordem pré-definida. A sua sintaxe é a seguinte:

```
matrix(m, n, [[a11, a12, ...], [a21, a22, ...], ...])
```

O vectores coluna e linha são definidos atribuindo, respectivamente, os valores $n = 1$ e $m = 1$.

```
>> A:=matrix(2,2,[[1,2],[2,4]]);
```

De facto, pode-se abdicar da informação `2,2` já que a matriz tem as entradas definidas explicitamente:

```
>> A:=matrix([[1,2],[2,4]]);
```

É relevante aqui referir-se a livraria *linalg* de ferramentes de apoio à álgebra linear. Por exemplo, permite-nos obter a factorização LU de A:

```
>> linalg::factorLU(A);
```

Os vários comandos desta livraria estão assim disponíveis:

```
>> ?linalg
```

Sendo tedioso a escrita repetitiva de `linalg::comando`, pode-se optar por *exportar* a livraria em bloco

```
>> export(linalg);
```

ficando assim todos os comandos disponíveis de uma forma mais expedita.

```
>> nullspace(A);
```

Certamente que o leitor pôde prever o resultado ao rever a factorização LU obtida mais acima. É desnecessário o cálculo do determinante, mas se for incrédulo tente

```
>> det(A);
```

Atente que a livraria foi exportada para o *kernel*, caso contrário teria que digitar

```
>> linalg::det(A);
```

As entradas da matriz são acedidas assim:

```
>> A[1,2];
```

2

Considere agora os vectores-linha

```
>> a:=matrix([1,0,3,4]);
>> b:=matrix([1,2,3,4]);
```

Vamos criar um "array" `c` que *apenas* tem como propósito armazenar informação:

```
>> c:=array(1..4);
```

Como resposta, o utilizador é informado que as entradas não estão especificadas. Nada de útil se processa no ciclo seguinte:

```
>> i:=0:
while i<4 do
        i:=i+1:
        c[i]:=a[i]*b[i]:
end_while:
eval(c);
```

O comando `eval` tem como objectivo mostrar o conteúdo de `c`. As mudanças de linha são feitas por `shift+return`, se estiver a usar o *x-mupad* em linux, ou o *mupad* no Windows. Em vez de um ciclo `while` poder-se-ia ter optando por um `for`:

```
>> for i from 1 to 4 do
        c[i]:=a[i]*b[i]:
end_for:
eval(c);
```

`Dom::Matrix(R)` cria o domínio das matrizes cujas componentes estão em `R`. Por exemplo,

```
>> MatZ2:=Dom::Matrix(Dom::IntegerMod(2));
```

cria o domínio (ou categoria) das matrizes sobre $\mathbb{Z}_2$, que denominámos por `MatZ2`. É óbvio que a denominação fica ao critério do utilizador. Não se esqueça de alguns pontos-chave, como a clareza (se começar a denominar os domínios de forma aleatória, será uma veradeira confusão encontrar, posteriormente, o que pretende), nem usar termos proprietários do mupad (uma forma de contornar este problema é usar algumas letras maiúsculas – recorde que é *case-sensitive*).

Por exemplo, suponha que pretende definir a matriz $A = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}$ enquanto matriz sobre $\mathbb{Z}_2$. Nada mais simples:

```
>> G:=MatZ2([[1,0,1,0,1],[0,1,0,1,1]]);
```

Suponha agora que pretende calcular o espaço nulo, `nG`, desta matriz, e respectiva dimensão:

```
>> nG:=linalg::nullspace(G);
```

```
>> nops(nG)
```
O vector i dessa base pode ser chamado usando `nG[i]`.

Uma instrução de grande utilidade é a de concatenação de matrizes, quer seja a de linhas ou a de colunas. Suponhamos que se prentende definir uma nova matriz $A = \left[\begin{array}{c|c} I_2 & G \end{array}\right]$.

```
>> I2:=matrix::identity(2);
>> I2:=coerce(I2,MatZ2);
```
A segunda instrução tem como propósito transformar a matriz identidade 2×2, por defeito real, numa sobre $\mathbb{Z}_2$. Façamos agora a concatenação horizontalmente:

```
>> A:=linalg::concatMatrix(I2,G);
```
Se se pretendesse concatenar verticalmente, far-se-ia uso de `linalg::stackMatrix`. Explore a *ajuda* sempre que necessário!

5 Procedimentos

Um procedimento é uma junção de instruções que, associadas, produzem uma tarefa, e que, estando agrupadas sob uma denominação, estão sujeitas a serem usadas por outras sob essa denominação. Um exemplo:

```
>> meumin := proc(x :  Type::Real, y :  Type::Real)
      begin
        if x < y then
            return(x);
         end_if;
         y;
      end_proc:
```
Em certos casos, é necessário a utilização de outras variáveis apenas a ser utililizadas no procedimento. Neste caso, é usado o comando `local` antes de `begin`. A instrução `return` devolve o valor obtido no procedimento.

Eis a sintaxe:

```
>>proc(
x1 <= default1> <:  type1>,
x2 <= default2> <:  type2>, ...
)  <:  returntype>
<name pname;>
<option option1, option2, ...;>
<local local1, local2, ...;>
<save global1, global2, ...;>
begin
body
end_proc
```

Tendo criado um conjunto relevante de procedimentos, torna-se imperioso a existência de uma ferramenta que permita *ler* os procedimentos em sessões futuras. Ou seja, é provavelmente uma boa ideia criar ficheiros com procedimentos que executem certas tarefas para depois, numa sessão de trabalho, esse procedimento ser invocado de uma forma expedita sem que tenha que ser de novo

introduzido pelo utilizador. No caso do procedimento `meumax` apresentado atrás, o autor destas linhas procedeu da forma seguinte:

(i) a sessão contendo apenas o procedimento foi gravada em
`/home/pedro/Aulas/2004-2005/Codigos/meumax.mupad`

(ii) aberta uma nova sessão, o procedimento `meumax` foi carregado usando
`>> read("/home/pedro/Aulas/2004-2005/Codigos/meumax.mupad")`

Apresentamos, de seguida, dois conjuntos de procedimentos que implementam os códigos ISBN e EAN-13.

- ISBN

  COMO INSTALAR

  Copie, para uma directoria, os ficheiros
  verISBN.txt
  menu.txt
  testaisbn.txt
  constroiisbn.txt

  No MUPAD, indique o caminho para essa directoria, usando

  ```
  READPATH:="directoria"
  ```

  Por exemplo, estando os ficheiros em

  ```
  d:\users\pedro\My Documents
  usou-se
  \begin{verbatim}READPATH:="d:\\users\\pedro\\My Documents"
  ```

  Ainda no MuPAD, faça

  ```
  read("verISBN.txt")
  ```

  Optimizado para Mupad, em linux, num terminal. Testado no XMUPAD pro em linux e no MuPADlight, em Windows.

  verISBN.txt

  ```
  /* ************************************ */
  /*         verISBN v0.01 beta1          */
  /****************************************/
  /* copyleft, pedro patricio, 2006       */
  /****************************************/


  read("menu.txt"):
  ```

menu.txt

```
/* ************************************ */
/*          M E N U  das opcoes         */
/****************************************/

prima:=0:
read("testaisbn.txt"):
read("constroiisbn.txt"):
repeat
   input("Prima 1 para testar o ISBN-10; 2 para calcular o digito de controlo;
 3 sair " , prima):
if prima=1 then
input("Indique o ISBN da forma [a,b,c,...] ",isbn):
testaisbn(isbn)
end_if:
if prima=2 then
input("Indique o ISBN da forma [a,b,c,...] ",isbn):
constroiisbn(isbn)
end_if /*menu das opcoes */
until prima=3 end_repeat:
```

Seguem-se os procedimentos propriamente ditos.

constroiisbn.txt

```
/* ************************************ */
/* Procedim  que  constroi o dig controlo*/
/****************************************/


constroiisbn:=proc(isbn)

local i, teste, checksum;

begin
isbnArray:=array(1..9,isbn);
teste:=0;
for i from 9 downto 1 do
teste:=teste+(11-i)*isbnArray[i]
end_for:
checksum:=-teste mod 11: /* checksum e' o digito de controlo */
if checksum=10 then /* caso em que o caracter e' X=10 */
```

```
print("O digito de controlo e' X")
else print(" O digito de controlo e' ",checksum):
end_if:
end_proc:
```

testaisbn.txt

```
/* ************************************ */
/* Procedimento que testa o ISBN-10      */
/****************************************/

testaisbn:=proc(isbn)

local i, isbnArray, teste;

begin
isbnArray:=array(1..10,isbn); /* criar o array com o isbn */
if (isbnArray[10]=X or isbnArray[10]=x) then
isbnArray[10]:=10
end_if: /* o caso em que o controlo e' X=10 */
teste:=0;
for i from 10 downto 1 do
teste:=teste+(11-i)*isbnArray[i]
end_for:
if (teste mod 11 = 0) then print("O ISBN esta' correcto")
else print(" O ISBN esta' incorrecto"):
end_if:
end_proc:
```

- EAN-13

  Aqui esta' um exemplo de um conjunto de procedimentos que testam e controem o digito de controlo de um codigo EAN-13

  verEAN v0.01 beta1

  COMO INSTALAR

  Copie, para uma directoria, os ficheiros
  verEAN.txt
  menu.txt
  testaean.txt

constroiean.txt

No MUPAD, indique o caminho para essa directoria, usando

```
READPATH:="directoria"
```

::LINUX::
Por exemplo, estando os ficheiros em

```
/home/pedro/Desktop
```

usou-se READPATH:=”/home/pedro/Desktop”

::WINDOWS:: Por exemplo, estando os ficheiros em

```
d:\users\pedro\My Documents
```

usou-se

```
READPATH:="d:\\users\\pedro\\My Documents"
```

Ainda no MuPAD, faça

```
read("verEAN.txt")
```

Optimizado para Mupad, em linux, num terminal. Testado no XMUPAD pro em linux, e no MUPAD-light em windows.

verEAN.txt

```
/* ************************************ */
/*         verEAN v0.01 beta1           */
/****************************************/
/* copyleft, pedro patricio, 2006       */
/****************************************/


read("menu.txt"):


```

menu.txt

```
/* ************************************ */
/*          M E N U  das opcoes         */
/****************************************/

prima:=0:
```

```
read("testaean.txt"):
read("constroiean.txt"): /* leitura dos sub-procedimentos */
repeat
input("Prima 1 para testar o EAN-13; 2 para calcular o digito de controlo;
 3 sair :" , prima):
if prima=1 then
input("Indique o EAN da forma [a,b,c,...] ",ean): /*entrada do ean pelo utilizador */
testaean(ean)
end_if:
if prima=2 then
input("Indique o EAN da forma [a,b,c,...] ",ean): /*entrada do ean pelo utilizador */
constroiean(ean)
end_if /*menu das opcoes */
until prima=3 end_repeat:
```

constroiean.txt

```
/* ************************************ */
/* Procedim  que  constroi o dig controlo*/
/****************************************/


constroiean:=proc(ean)

local i, isbn, testePAR, testeIM, checksum, eanArray;

begin
eanArray:=array(1..12,ean);
        testeIM:=0;
        for i from 1 to 6 do
                testeIM:=testeIM+eanArray[2*i-1]
        end_for:
        testePAR:=0;
        for i from 1 to 6 do
                testePAR:=testePAR+3*eanArray[2*i]
        end_for:
        checksum:=-(testeIM+testePAR) mod 10:
  print(" O digito de controlo e' ",checksum):
end_proc:
```

testaean.txt

```
/* ************************************ */
```

```
/* Procedimento que testa o EAN-13       */
/****************************************/


testaean:=proc(ean)

local i,  testeIM, testePAR, teste, eanArray;

begin
eanArray:=array(1..13,ean); /* criar o array com o ean */
testeIM:=0;
for i from 1 to 7 do
testeIM:=testeIM+eanArray[2*i-1]
end_for:
testePAR:=0;
for i from 1 to 6 do
testePAR:=testePAR+3*eanArray[2*i]
end_for:
teste:=testeIM+testePAR:
if (teste mod 10 = 0) then print("O EAN esta' correcto")
else print(" O EAN esta' incorrecto"):
end_if:
end_proc:
```

Como exercício, construa um procedimento

`hamming:=proc(numero:Type::PosInt)`

onde n é um argumento de entrada, número inteiro positivo, e cujo resultado final é uma matriz $n \times (2^n - 1)$, cuja coluna j é a representação do número natural j na sua escrita binária. A matriz resultante chama-se *matriz de Hamming*. Por exemplo,

```
>> hamming(3);
```

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Para tal, use a instrução `mod` que a um par de naturais faz corresponder o resto da divisão inteira do primeiro pelo segundo.

Caso ache conveniente, pode dividir o problema em dois, em que um consiste em transformar a escrita decimal de um natural na sua representação binária, e o outro constrói a matriz.

6 Polinómios

Antes de mais, façamos um *reset* do *kernel*:

```
>> reset();
```

O conjunto (mais do que isso, o anel) dos polinómios com tantas variáveis quantas se queira é denotado por `Dom::Polynomial`.

`Dom::Polynomial(R, ..)` cria o domínio dos polinómios com coeficientes no anel comutativo R, podendo a representação ser exibida de uma forma pré-definida.

Sintaxe:

`Dom::Polynomial( <R <, Order>>)`

`Order` pode tomar o valor `LexOrder`, `DegreeOrder`, ou `DegInvLexOrder` ou um elemento de `Dom::MonomOrdering`. *Default:* `LexOrder`.

Por exemplo, suponha o leitor que se quer estudar polinómios com uma variável, em que os coeficientes são elementos de $\mathbb{Z}_3$. Ou seja, o anel sob estudo é $\mathbb{Z}_3[x]$. Mais concretamente, pretende-se estudar o polinómio dado por

$$f(x) = x^2 + x + 2.$$

Defina-se, no μ–pad, o domínio $\mathbb{Z}_3[x]$. Usaremos a sigla `Polin3` para representar essa entidade:

```
>> Polin3:=Dom::Polynomial(Dom::IntegerMod(3));
```

Para mais informações, digite

```
>> ?Dom::Polynomial
```

Para definir f, nada mais fácil:

```
>> f:=Polin3(x^2+x+5);
```

$$(1 \bmod 3)\ x^2 + (1 \bmod 3)\ x + (2 \bmod 3)$$

O polinómio é irredutível:

```
>> irreducible(f);
```

```
TRUE
```

Para se calcular $f(1)$,

```
>> evalp(f,x=1);
```

```
1 mod 3
```

Temos, então, um polinómio irredutível, e portanto, $\mathbb{Z}_3[x]/(f(x))$ é um corpo, no caso um corpo de Galois. Será um corpo gerado pelo polinómio x? Uma forma expedita de se verificar se $f(x)$ é primitivo é escrever os elementos de $\mathbb{Z}_3[x]/(f(x))$ enquanto potências de x. Por exemplo, $x^3 \mod f(x)$ é calculado da forma seguinte:

```
>> powermod(x,3,f);
```

```
2 x + 2
```

Para verificarmos se $f(x)$ é primitivo, ou de forma equivalente, se o corpo $\mathbb{Z}_3[x]/(f(x))$ é gerado por x, basta percorrer as potências de x e verificar se todas, até certo expoente, são diferentes de 1. Pode ser ainda de interesse apresentar todas as potências modulo $f(x)$ para cálculos futuros:

```
>> for i from 1 to 8 do
t:=powermod(x,i,f);
print(x^i,"elemento do corpo:",t);
end_for
```

O resultado é:

```
    x, "elemento do corpo:", x
  x^2 , "elemento do corpo:", 2 x + 1
  x^3 , "elemento do corpo:", 2 x + 2
    x^4 , "elemento do corpo:", 2
   x^5 , "elemento do corpo:", 2 x
  x^6 , "elemento do corpo:", x + 2
  x^7 , "elemento do corpo:", x + 1
    x^8 , "elemento do corpo:", 1
```

Poderá ser conveniente (e certamente sê-lo-á) criar um procedimento que tem como entrada um polinómio (para simplificar, em $\mathbb{Z}_2[x]$) primitivo $p(x)$ e como saída uma tabela com a correspondência entre as potências de x (polinómio gerador do grupo cíclico) e os polinómios de $\mathbb{Z}_2/(p(x))$. Antes de passarmos à resolução do problema, e portanto construção do procedimento), vejamos qual o metodologia a ser usada na implementação. Até agora, tem-se usado o MuPad não só como interpretador, mas também como editor (colocado ponto de vista de uma forma simplista). Faremos neste caso uso de um editor de texto externo, ficando o MuPad com a função de interpretador. Num editor de texto, crie um ficheiro, por exemplo, TabPrim.txt[1]. Comente as instruções de forma a que mais tarde seja fácil a si ou outrem entender o caminho que vai sendo traçado.

```
TabPrim:=proc(f)

   local grau,tabela, i;

   begin
        grau:=degree(f):
        tabela:=array(1..2^grau-1):
        for i from 1 to 2^grau-1 do
            tabela[i]:=powermod(x,i,f)
        end_for:
        return(tabela):
   end_proc:
```

Agora, e já no MuPad, o procedimento construido externamente é carregado com

```
>> read("TabPrim.txt")
```

Tenha atenção ao caminho (vulgo PATH) onde o seu ficheiro está. Temos então um procedimento no *kernel* do sistema que, dado um polinómio irredutível, constrói uma tabela das potências:

```
>> Polin2:=Dom::Polynomial(Dom::IntegerMod(2))
```

[1] É óbvio que lhe pode chamar o que quiser. A extensão .txt pode parecer despropositada, mas pode servir de alguma ajuda em alguns SO's no reconhecimento do tipo de ficheiro.

```
                Dom::Polynomial(Dom::IntegerMod(2), LexOrder)
>> f:=Polin2(x^4+x+1);

                                4
                  (1 mod 2) x  + (1 mod 2) x + (1 mod 2)
>> TabPrim(f);
```

Se se quiser saber como se escreve x^{10} basta fazer

```
>> TabPrim(f)[10]

                                  2
                                 x  + x + 1
```

Mais adiante, será crucial encontrar raizes de polinómios cujos coeficientes estão num certo corpo de Galois. Consulte `Dom::GaloisField` no MuPad.

Considere $\pi(x) = x^4 + x + 1 \in \mathbb{Z}_2[x]$. Este polinómio é primitivo, pelo que $\mathbb{F} = \mathbb{Z}_2[x]/(\pi(x))$ é um corpo e $\mathbb{F}^* = \langle x \rangle$ Pretende-se saber que elementos de $\mathbb{F}$ anulam $m(x) = x^2 + x + 1 \in \mathbb{Z}_2[x]$.

```
>> // Criar um corpo de Galois usando Dom::GaloisField
>> p:=a^4+a+1:    // polinomio primitivo p
>> F:=Dom::GaloisField(2,4,p):    // corpo de Galois criado!
>> PolF:=Dom::Polynomial(F);   // anel dos polinomios dobre F
>> m:=PolF(x^2+x+1)   // o polinomio m sobre Z2:  calcula-se as raizes sobre F
>> solve(m=0,x);   // as raizes de m
>> F(a^5);
```

A última instrução revela que a^5 é uma das raízes. Como se escreve a outra raiz como potência de a?

7 Códigos de Hamming

Os códigos de Hamming serão os primeiros com direito a implementação. No esquema de correcção de rros, será necessário escrever um número apresentado no sistema binário no decimal. De facto, o número será dado à custa de um vector (coluna) sobre $\mathbb{Z}_2$. Por exemplo, o vector $[1, 0, 1, 1]^T$ indica o número $(1011)_2$. Pretende-se encontrar a representação decimal. Para tal, recordamos como se faz tal correspondência: a representação decimal de $(v_1 v_2 \cdots v_k)_2$ é

$$\sum_{i=1}^{k} v_i 2^{k-i}.$$

No exemplo anterior, temos $1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$, que no caso é 11. Este algoritmo é facilmente implementado.

No entanto, o vector v fornecido pode ser elemento de $\mathbb{Z}_2$.

```
>> Z2:=Dom::IntegerMod(2):
```

```
>> M2:=Dom::Matrix(Z2):
>> delete(i):
>> v:=M2([1,0,1,1]):
>> sum(v[i]*2^(4-i),i=1..4);
```

```
1 mod 2
```

Aqui surge um problema, já que o MUPAD efectua os cálculos em $\mathbb{Z}_2$.

Para contornar esta questão, transforma-se, à medida do necessário, as entradas $v[i]$ de v dado em elementos de $\mathbb{Z}$, fazendo uso de uma variável auxiliar:

```
>> aux:=coerce(v[i],Dom::Integer);
```

Assuma que usa um ciclo `for`. A cada passo do ciclo, a soma vai sendo calculada:

```
>> soma:=soma+aux*2^(n-i);
```

Ficamos, para este caso, com o ciclo seguinte:

```
>> soma:=0
```

```
0
```

```
>> for i from 1 to 4 do
 aux:=coerce(v[i],Dom::Integer);
 soma:=soma+aux*2^(4-i);
 end_for: print(soma);
```

```
11
```

Se estiver a construir um procedimento, não se esqueça de enviar o resultado para o programa principal:

```
return(soma);
```

Esta secção tem como finalidade fornecer algumas ideias para a implementação de um código de Hamming. Recorde que a correcção de erros de r é feita à custa do síndrome de r, ou seja, à custa do vector coluna Hr^T, onde H é a matriz de Hamming, matriz de paridade do código. Mais, um código de Hamming é um código perfeito corrector erros singulares. De facto, a posição do erro em r é dada pela representação decimal de Hr^T. Por exemplo, suponha que r recebido tem síndrome $[1,0,1,1]^T$. Pelo que foi visto nas aulas teóricas, a posição de r com erro é $(1011)_2$, que corresponde a 11 na base 10. Note ainda que sendo H a matriz de paridade do código de Hamming, com 4 linhas, então tem $2^4 - 1 = 15$ colunas. Ou seja, r é um vector linha com 14 bits, sendo que o 11° está incorrecto (segundo o esquema usado na correcção pelo vizinho mais próximo, fazendo uso da distância de Hamming).

É ainda necessário, de forma a se poder implementar um código de Hamming, construir um procedimento `hamming:=proc(numero:Type::PosInt)` onde n é um argumento de entrada, número inteiro positivo, e cujo resultado final é uma matriz $n \times (2^n - 1)$, cuja coluna j é a representação do número natural j na sua escrita binária. Ou seja, uma forma fácil de construir uma matriz

de Hamming. Para tal, concentramo-nos num algoritmo que, dado um número $(k)_{10}$, encontre um vector coluna sobre $\mathbb{Z}_2$, com dimensão fixa à partida, cujas entradas são os dígitos (0 ou 1) correspondentes à representação binária de n. Por exemplo, que a 5, fixando em 4 o número de linhas do vector, tenha como resultado $[0,1,0,1]^T$.

Suponha, então, que se pretende construir um vector coluna v, com `dimen` linhas, cujas entradas são os símbolos da representação binária de q, onde `dimen` $\geq \lfloor \log_2 q \rfloor + 1$. Aplica-se o algoritmo da divisão:

$$\begin{aligned} q = q_0 &= 2q_1 + r_1, \\ q_1 &= 2q_2 + r_2, \\ &\vdots \\ q_{n-2} &= 2q_{n-1} + r_{n-1}, \\ q_{n-1} &= 2 * 0 + r_n, \end{aligned}$$

onde $0 \leq r_i < 2$. O vector pretendido será $[0, \ldots, 0, r_n, r_{n-1}, \ldots, r_2, r_1]^T$.

No que se segue, atribua 4 a `dimen` e 5 a q.

```
>> for i from dimen downto 1 do
                v[i]:=Dom::IntegerMod(2)(q):
                q:=q div 2:
   end_for:
>> v;
                                +-         -+
                                |  0 mod 2  |
                                |           |
                                |  1 mod 2  |
                                |           |
                                |  0 mod 2  |
                                |           |
                                |  1 mod 2  |
                                +-         -+
```

O que lhe resta fazer? Construir cada coluna da matriz de Hamming, concatenando-as. Por exemplo:

```
export(linalg):
dimen:=3:
q:=1: v:=M2(dimen,1):
for i from dimen downto 1 do
                v[i]:=Dom::IntegerMod(2)(q):
                q:=q div 2:
   end_for:
H:=v:
for q2 from 2 to 2^dimen-1 do
q:=q2:for i from dimen downto 1 do
```

```
            v[i]:=Dom::IntegerMod(2)(q):
            q:=q div 2:
   end_for:
H:=concatMatrix(H,v)
end_for: print(H);
```

Suponha que foi recebido por uma canal simétrico binário

$$r = \begin{pmatrix} 1 \bmod 2 & 1 \bmod 2 & 1 \bmod 2 & 1 \bmod 2 & 1 \bmod 2 & 1 \bmod 2 & 1 \bmod 2 \end{pmatrix}$$

onde o esquema de codificação usado foi Hamming [7,4]. Seja H a matriz de Hamming com 3 linhas, que pode ser obtida pelas instruções atrás definidas. Basta-nos calcular o síndrome de r, isto é, Hr^T:

```
>> H*transpose(r)
                                  +-         -+
                                  |  0 mod 2  |
                                  |           |
                                  |  0 mod 2  |
                                  |           |
                                  |  0 mod 2  |
                                  +-         -+
```

Assuma que previamente foi exportada a livraria *linalg*. Como o síndrome é o vector nulo, segue que r é palavra-código.

Suponha agora que foi recebido

$$r = \begin{pmatrix} 1 \bmod 2 & 1 \bmod 2 & 1 \bmod 2 & 0 \bmod 2 & 0 \bmod 2 & 0 \bmod 2 & 1 \bmod 2 \end{pmatrix}$$

Calcula-se o síndrome de r

```
>> s:=H*transpose(r)
                                  +-         -+
                                  |  1 mod 2  |
                                  |           |
                                  |  1 mod 2  |
                                  |           |
                                  |  1 mod 2  |
                                  +-         -+
```

para se concluir que não é palavra-código. Sendo o código de Hamming, o processo de correcção de erros (neste caso de um erro) fornece um algoritmo para a correcção: o síndrome é a representação binária da posição de r que tem o erro. Faça-se uso do que inicialmente se tratou nesta secção.

```
>> soma:=0:
for i from 1 to 3 do
 aux:=coerce(s[i],Dom::Integer);
 soma:=soma+aux*2^(3-i);
end_for:
print(soma);
```

Ou seja, o elemento $r[7]$ está errado. Corrijamo-lo:

```
>> r[soma]:=r[soma]+Z2(1);
                                    0 mod 2
```

8 Códigos lineares

Os códigos de Hamming são códigos lineares em que a matriz de paridade pode ser dada por uma matriz de Hamming. Estes têm a vantagem de serem perfeitos, embora sejam apenas correctores de erros singulares e de os parâmetros seguirem regras (mais) fixas.

O esquema que se irá seguir de correcção de erros de códigos lineares gerais recorre aos síndromes. Para tal, é preciso calcular o número de erros do código linear à custa de, por exemplo, uma matriz geradora ou de uma matriz de paridade. Listam-se então os síndromes dos líderes de cada classe. Recebido um vector, este é corrigível se o seu síndrome igualar um único síndrome de um líderes, líder esse que é subtraído ao vector recebido por forma a se efectuar a correcção.

Suponha que A é matriz geradora de um código C, onde

$$A = \begin{pmatrix} 1 \bmod 2 & 1 \bmod 2 & 1 \bmod 2 & 1 \bmod 2 & 0 \bmod 2 \\ 0 \bmod 2 & 1 \bmod 2 & 1 \bmod 2 & 0 \bmod 2 & 1 \bmod 2 \end{pmatrix}$$

Esta matriz tem característica 2:

```
>> rank(A)
                                        2
```

Como o código é linear, a distância mínima do código é igual ao menor peso de Hamming das palavras-código não nulas. Ou seja, $d(C) = \min_{c \in C \setminus \{0\}} w(c)$. As palavras código não nulas são as combinações lineares (não nulas) das linhas de A. No nosso caso, `row(A,1); row(A,2); row(A,1)+row(A,2)`. Segue que a distância mínima é 3, e portanto é corrector de erros singulares.

Para se aplicar o esquema de correcção de erros, é ainda necessário calcular a matriz de paridade. Ou seja, uma matriz com característica 3 tal que $AH^T = 0$. Ou ainda, uma matriz cujas colunas da transposta formem uma base do espaço nulo de A.

```
>> nullspace(A)
             -- +-          -+  +-          -+  +-          -+ --
             |  |  0 mod 2  |  |  1 mod 2  |  |  1 mod 2  |  |
             |  |           |  |           |  |           |  |
             |  |  1 mod 2  |  |  0 mod 2  |  |  1 mod 2  |  |
             |  |           |  |           |  |           |  |
             |  |  1 mod 2  |, |  0 mod 2  |, |  0 mod 2  |  |
             |  |           |  |           |  |           |  |
             |  |  0 mod 2  |  |  1 mod 2  |  |  0 mod 2  |  |
             |  |           |  |           |  |           |  |
             |  |  0 mod 2  |  |  0 mod 2  |  |  1 mod 2  |  |
             -- +-          -+  +-          -+  +-          -+ --
```

Neste caso,

```
>> H:=concatMatrix(nullspace(A)[1],nullspace(A)[2],nullspace(A)[3])
```

```
+-                           -+
|  0 mod 2, 1 mod 2, 1 mod 2  |
|                             |
|  1 mod 2, 0 mod 2, 1 mod 2  |
|                             |
|  1 mod 2, 0 mod 2, 0 mod 2  |
|                             |
|  0 mod 2, 1 mod 2, 0 mod 2  |
|                             |
|  0 mod 2, 0 mod 2, 1 mod 2  |
+-                           -+
```

Em casos mais complexos, será preciso criar um ciclo de forma a se construir a concatenação iterativamente. Repare que a matriz de paridade é a transposta da matriz apresentada em cima. Portanto,

```
>> H:=transpose(H)
```

```
+-                                                 -+
|  0 mod 2, 1 mod 2, 1 mod 2, 0 mod 2, 0 mod 2  |
|                                                   |
|  1 mod 2, 0 mod 2, 0 mod 2, 1 mod 2, 0 mod 2  |
|                                                   |
|  1 mod 2, 1 mod 2, 0 mod 2, 0 mod 2, 1 mod 2  |
+-                                                 -+
```

Os líderes são, visto o código corrigir 1 erro, as linhas da matriz identidade I_5, e portanto os síndromes dos líderes não são mais que as colunas de H.

Suponha que se recebeu por uma canal simétrico binário com ruído o vector

$$r = \begin{pmatrix} 1 \bmod 2 & 0 \bmod 2 & 0 \bmod 2 & 1 \bmod 2 & 0 \bmod 2 \end{pmatrix}$$

Calcula-se o síndrome Hr^T de r:

```
>> H*transpose(r)
```

```
+-         -+
|  0 mod 2  |
|           |
|  0 mod 2  |
|           |
|  1 mod 2  |
+-         -+
```

Como o síndrome não é nulo, segue que $r \notin C$. Mas o síndrome de r é igular à coluna 5 de H, que por sua vez é o síndrome da linha 5 de matriz identidade I_5. Portanto, a correcção é feita como

```
>> I5:=M2(matrix::identity(5)):
```

```
>> c:=r+row(I5,5)
              +-                                                 -+
              | 1 mod 2, 0 mod 2, 0 mod 2, 1 mod 2, 1 mod 2 |
              +-                                                 -+
```

9 BCH

O código BCH é um código linear cíclico, construído à custa de polinómios sobre $\mathbb{Z}_2$. O papel da primitividade de certos polinómios irredutíveis é crucial nesta teoria. Para tal, definamos o domínios dos polinómios sobre $\mathbb{Z}_2$ e teste-se a primitividade de $p(x) = x^4 + x + 1$:

```
>> Pol2:=Dom::Polynomial(Dom::IntegerMod(2));
               Dom::Polynomial(Dom::IntegerMod(2), LexOrder)
>> q:=Pol2(x^4+x+1)
                              4
                  (1 mod 2) x  + (1 mod 2) x + (1 mod 2)
>> fs:=2^(degree(q))

                                    16
>> for i from 1 to fs-1 do
     field[i]:=powermod(x,i,q):
   end_for:

>> field[fs]:=0:
>> ftable:=table():
>> for i from 1 to fs-1 do
     ftable[field[i]]:=x^i:
   end_for:
>> ftable[field[fs]]:=0:
>> eval(ftable);

                          table(
                            0 = 0,
                                 15
                            1 = x  ,
                             3        14
                            x  + 1 = x  ,
                             3    2        13
                            x  + x  + 1 = x  ,
                             3    2            12
                            x  + x  + x + 1 = x  ,
                             3    2        11
                            x  + x  + x = x  ,
                             2            10
                            x  + x + 1 = x  ,
```

```
 3        9
x  + x = x ,
 2        8
x  + 1 = x ,
 3           7
x  + x + 1 = x ,
 3    2    6
x  + x  = x ,
 2        5
x  + x = x ,
          4
x + 1 = x ,
 3    3
x  = x ,
 2    2
x  = x ,
x = x
)
```

Vejamos como se pode escrever $x^3 + 1$ como elemento do grupo:

```
>> ftable[x^3+1]
```

```
                                  14
                                 x
```

Façamos agora uso do domínio dos corpos de Galois implementado no MuPAD:

```
p:=a^4+a+1:
F:=Dom::GaloisField(2,4,p):
PolF:=Dom::Polynomial(F)
```

Assuma $f(x) = x^{2^4-1} - 1 = x^{15} - 1$, e calcule-se a respectiva factorização em polinómios irredutíveis:

```
>> f:=Pol2(x^15-1)
```

```
                                  15
                        (1 mod 2) x   + (1 mod 2)
```

```
>> for i from 1 to (nops(factor(f))-1)/2 do
     print(factor(f)[2*i]);
   end_for;
```

```
                         (1 mod 2) x + (1 mod 2)
```

```
                            2
                  (1 mod 2) x  + (1 mod 2) x + (1 mod 2)
```

```
                  4              3
(1 mod 2) x  + (1 mod 2) x  + (1 mod 2)

                  4
(1 mod 2) x  + (1 mod 2) x + (1 mod 2)

                  4              3
(1 mod 2) x  + (1 mod 2) x  + (1 mod 2)
```

Construa-se uma tabela contendo os factores irredutíveis de $f(x)$:

```
>> for i from 1 to (nops(factor(f))-1)/2 do
     TabFact[i]:=factor(f)[2*i]
   end_for:
```

Destes factores, vejamos de que potência de α são minimais aniquiladores:

```
for k from 1 to nops(TabFact) do  // vamos percorrer cada factor irredutivel
m:=TabFact[k]:
print("Polinomio: ",m,"  k=",k);  // mensagem de controlo
mF:=PolF(0):  // aqui comeca o algoritmo que transforma o polinomio em PolF
for i from 0 to degree(m) do
    mF:=mF+PolF(coeff(m,i)*x^i):
end_for:
for j from 1 to nops(solve(mF=0,x)) do
    print("solucoes: ",solve(mF=0,x)[j])  // agora temos as raizes sobre F
end_for
end_for
```

Uma rápida análise aos resultados, obtemos

```
>> m1:=TabFact[4]: m2:= m1: m4:=m1: m3:=TabFact[5]: m6:=m3: m5:=TabFact[2]:
```

onde `m?` indica o polinómio m_i minimal aniquilador de α^i.

O caso menos evidente será a obtenção de m_5. No entanto, fazendo

```
>> F(a^5)
                                  2
                                 a  + a
```

chega-se à conclusão que $x^5 \equiv x^2 + x$, que por sua vez anula `TabFact[2]`.

O polinómio do código BCH considerando as primeiras 6 potências de α é $g(x) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$:

```
>> g:=lcm(m1,m2,m3,m4,m5,m6)
          10              8              5              4
(1 mod 2) x   + (1 mod 2) x  + (1 mod 2) x  + (1 mod 2) x  +

             2
   (1 mod 2) x  + (1 mod 2) x + (1 mod 2)
```

A inserção de g pode ser feita directamente:

```
>> g:=Pol2(x^10+x^8+x^5+x^4+x^2+x+1);
          10              8              5              4
(1 mod 2) x   + (1 mod 2) x  + (1 mod 2) x  + (1 mod 2) x  +
              2
   (1 mod 2) x  + (1 mod 2) x + (1 mod 2)
```

A codificação de um polinómio $h(x)$ é feita fazendo $h(x)g(x)$. Por exemplo, para $h(x) = x^3 + x + 1$,

```
>> h:=Pol2(x^3+x+1)

                            3
                  (1 mod 2) x  + (1 mod 2) x + (1 mod 2)
```

```
>> Pol2(h*g)

          13              10              9              7
(1 mod 2) x   + (1 mod 2) x   + (1 mod 2) x  + (1 mod 2) x  +

             6              5
   (1 mod 2) x  + (1 mod 2) x  + (1 mod 2)
```

Assuma agora que foi recebido o polinómio $r(x) = 1 + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^{10}$:

```
>> r:=Pol2(1+x^2+x^3+x^4+x^5+x^6+x^7+x^10)

          10              7              6              5
(1 mod 2) x   + (1 mod 2) x  + (1 mod 2) x  + (1 mod 2) x  +

             4              3              2
   (1 mod 2) x  + (1 mod 2) x  + (1 mod 2) x  + (1 mod 2)
```

O polinómio recebido não é palavra código, já que g não divide r:

```
>> Pol2(divide(r,g,Rem))

            8              7              6              3
  (1 mod 2) x  + (1 mod 2) x  + (1 mod 2) x  + (1 mod 2) x  + (1 mod 2) x
```

O passo seguinte envolve o cálculo dos síndromes de $r(x)$. Para tal, iremos transformar r num polinómio em $\mathbb{F}[x]$.

```
>> rF:=PolF(0):
   for i from 0 to degree(r) do
       rF:=rF+PolF(coeff(r,i)*x^i):
```

```
   end_for:
>> print(rF);
```

```
                  10    7    6    5    4    3    2
                 x   + x  + x  + x  + x  + x  + x  + 1
```

Assumimos que r tem 3 erros. A tabela dos síndromes é calculada assim:

```
t:=3; syn:=array(1..2*t):
for i from 1 to 2*t do
    syn[i]:=evalp(rF,x=F(a^i)):
end_for:
print(syn):
```

e obtemos

```
  +- 3   3    2   3    2   3    2          2          3    2        -+
  | a , a  + a , a  + a , a  + a  + a + 1, a  + a + 1, a  + a  + a + 1 |
  +-                                                                  -+
```

O sistema de equações lineares que se obtem (v. aulas teóricas), que determina os coeficientes do polinómio localizador de erros $E(z)$ tem como matriz

$$\begin{pmatrix} a^3 & a^3+a^2 & a^3+a^2 \\ a^3+a^2 & a^3+a^2 & a^3+a^2+a+1 \\ a^3+a^2 & a^3+a^2+a+1 & a^2+a+1 \end{pmatrix}$$

que resulta de

```
MatF:=Dom::Matrix(F)
V:=MatF([[syn[1],syn[2],syn[3]],[syn[2],syn[3],syn[4]],[syn[3],syn[4],syn[5]]])
```

O termo independente é

```
>> b:=MatF([[syn[4]],[syn[5]],[syn[6]]])
```

```
                  +-               -+
                  |   3    2        |
                  |  a  + a  + a + 1|
                  |                 |
                  |      2          |
                  |     a  + a + 1  |
                  |                 |
                  |   3    2        |
                  |  a  + a  + a + 1|
                  +-               -+
```

Verifique-se que a matriz é não-singular:

```
>> det(V)
```

$$a^3 + a^2 + a + 1$$

A única solucão é dada por $x = V^{-1}b$:

```
>> inverseLU(V)
```

$$\left[\begin{array}{cccc} a^3 + a^2 + a, & a^3 + a^2 + a + 1, & a^3 + a^2 + 1 \\ a^3 + a^2 + a + 1, & a + 1, & a^3 + 1 \\ a^3 + a^2 + 1, & a^3 + 1, & a^3 + a^2 + a \end{array}\right]$$

```
>> solucao:=inverseLU(V)*b
```

$$\left[\begin{array}{c} a^2 \\ a \\ a^3 + a^2 + 1 \\ a^3 \end{array}\right]$$

Vejamos a que potência de α corresponde a segunda entrada do vector solução:

```
>> ftable[x^3+x^2+1]
```

$$x^{13}$$

Temos então os síndromes, e portanto temos em nosso poder o polinómio localizador de erros ELP:

```
sig2:=a^13
sig3:=a^3
```

```
sig1:=a^2
ELP:=x->x^3+sig3*x^2+sig2*x+sig
```

Por exaustão, encontram-se os zeros deste polinómio, e faz-se a correcção dos erros:

```
>> Erro:=Pol2(0):
>> for i from 0 to 14 do
       if F(ELP(a^i))=F(0) then
          print("erro na posicao ",i):
          Erro:=Erro+Pol2(x^i);
       end_if:
   end_for: print("e(x)=",Erro):

                          "erro na posicao ", 0

                          "erro na posicao ", 5

                         "erro na posicao ", 12

                                    12              5
             "e(x)=", (1 mod 2) x   + (1 mod 2) x  + (1 mod 2)
>> palavracod:=Pol2(r+Erro);

           12              10              7              6
(1 mod 2) x   + (1 mod 2) x   + (1 mod 2) x  + (1 mod 2) x  +

              4              3              2
   (1 mod 2) x  + (1 mod 2) x  + (1 mod 2) x
```

Referências

[1] , W.C. Huffman, V. Pless, *Fundamentals of Erros-Correcting Codes*, Cambridge UP, 2003.

[2] R. Klima, N. Sigmon, E. Stitzinger, *Applications of Abstract Algebra with MAPLE*, CRC Press, 2000.

[3] J. H. van Lint, *Introduction to coding theory*, Springer, 1999.

[4] M. Majewski, *MuPAD Pro Computing Essentials*, Springer, 2004.