GAP, un sistema per l'algebra computazionale

Introduzione e applicazioni alla teoria dei codici correttori

7 Gennaio 2016

Contents

1 Nozioni preliminari **3**
1.1 Accedere e lasciare GAP 3
1.2 GAP come calcolatrice 3
1.3 Errori e Break Loop 4
1.4 Variabili e assegnamento 5
1.5 Costruzione e gestione di oggetti elementari, o di interesse . . 7
1.6 Confronti e operazioni booleane 11
1.7 Iterazioni e condizionali 11
1.8 Lettura di file 13

2 Gruppi e campi finiti in GAP **15**
2.1 Gruppi di permutazioni 15
2.2 Campi finiti e polinomi 18

3 Teoria dei codici correttori **23**
3.1 Creazione e gestione di codici lineari 23
3.2 Comunicazione secondo il modello BSC 30
3.3 Codici ciclici 34

Introduzione

La presente vuole essere una breve e basilare introduzione all'utilizzo del pacchetto software "guava" per la teoria dei codici correttori. Si andranno ad esaminare i metodi principali di tale pacchetto, con esempi che rendano concreto lo studio.

In una prima fase si illustreranno le caratteristiche e la sintassi fondamentali del linguaggio GAP (*Groups, Algorithms and Programming*) strettamente necessarie per la comprensione degli argomenti trattati.

Per i risultati teorici si fa riferimento a [1, 3]. Per quanto riguarda il linguaggio e i metodi di GAP, ogni particolare illustrato viene completamente descritto nel manuale disponibile in rete, si faccia riferimento a [2].

Chapter 1

Nozioni preliminari

GAP è composto di un pacchetto di software sviluppato in parte in C, in parte nel linguaggio di Programmazione GAP stesso. Procediamo dunque ad osservare la semantica e la sintassi di base di quest'ultimo che ci saranno necessarie per il seguito.

1.1 Accedere e lasciare GAP

L'accesso a GAP avviene in modi diversi a seconda dell'OS operante sulla macchina in utilizzo. Per quanto riguarda distribuzioni UNIX, dovremo accedere al terminale e spostarci nella directory contenente il codice, dunque lanciare un programma shell con una sintassi del tipo bin/gap.sh.

Per Windows basterà raggiungere la cartella in cui abbiamo creato il collegamento al programma al momento dell'installazione e cliccare sull'icona.

Se dovessero presentarsi particolari difficoltà si faccia riferimento a [2].

Una volta avviato il programma sarà stampato un breve banner e il prompt *gap>*, al termine del quale si andranno ad inserire i comandi.

Per lasciare il programma è sufficiente inserire il comando *quit;*.

1.2 GAP come calcolatrice

Per cominciare, osserviamo che GAP può essere utilizzato come una semplice calcolatrice di linea, in grado però di lavorare con interi di lunghezza "arbitraria" e razionali, oltre ovviamente che con i numeri in virgola mobile.

Per eseguire calcoli di base, coinvolgenti unicamente le quattro operazioni elementari e l'esponenziale, è sufficiente inserire l'espressione da esaminare nella linea di comando e far seguire al tutto un punto e virgola (;).

Si noti che la divisione operata da GAP è in generale l'operazione esatta, pertanto un comando del tipo 2/3; restituirà in maniera tautologica 2/3. Per ottenere il risultato atteso 0.66667 è necessario inserire un'espressione

del tipo 2/3.0; indicando a GAP di voler lavorare con numeri in virgola mobile.

Ancora, è possibile calcolare il resto della divisione intera per mezzo del comando *mod*, con la seguente sintassi

$$dividendo\ mod\ divisore;$$

dove ovviamente dividendo e divisore sono numeri interi.

Per effettuare operazioni diverse da quelle sopra indicate, GAP si affida a delle funzioni. Ogni funzione in GAP ha un nome che indica chiaramente il suo scopo; tale identificazione inoltre inizia sempre con una lettera maiuscola. La chiamata ad una funzione presenta la seguente sintassi:

$$FunctionName(arg_1, ..., arg_n);$$

Oltre alle varie funzioni elementari dell'analisi, le quali devono essere sempre chiamate con numeri in virgola mobile per argomento, GAP comprende numerosi metodi specifici, ad esempio dal campo della teoria dei numeri, quali la funzione φ che, applicata ad un intero m, restituisce il numero di interi positivi minori e coprimi con esso.

Altra funzione di interesse può essere **IsPrime**, che effettua un test di primalità sul suo argomento, test il cui risultato è frutto di un processo deterministico, dunque sicuramente esatto, per "piccoli" interi.

Di seguito un paio di esempi:

```
gap>3*5;
15
gap>2^120;
1329227995784915872903807060280344576
gap> Factorial(30);
265252859812191058636308480000000
gap> Cos(2.0);
-0.416147
gap> Exp(1.1);
3.00417
gap> Phi(2^13 - 1);
8190
gap> 2^13-1; # Questo prova che 8191 è primo
8191
```

1.3 Errori e Break Loop

Specialmente se ci si trova a scrivere codici particolarmente lunghi e complessi, o se si richiamano metodi non molto familiari, si incorre inevitabilmente in qualche errore.

Se ci troviamo di fronte ad un errore sintattico, o a qualche espressione matematicamente insensata, GAP ci fornirà un'indicazione di errore, entrando in un break loop.

Il banner di gap si modifica, e viene stampato un messaggio di errore che ci informa sulla natura dello sbaglio commesso.

All'interno del break loop GAP ci dà usualmente la possibilità di rimediare al problema, invitandoci ad inserire dei parametri corretti tramite il comando *return*. Si faccia tuttavia particolare attenzione a quale input GAP richiede di sostituire: specialmente all'interno di funzioni, può succedere che il processo sia già entrato in qualche subroutine al momento dell'identificazione dell'errore, e che quindi vi sia richiesto di inserire argomenti validi per funzioni di cui non conoscete nemmeno lo scopo. In tal caso, o se il messaggio di errore non vi è chiaro, inserite *quit;* per abbandonare il loop ed evitare ulteriori danni.

Si noti ancora che più cicli break possono essere incastrati l'uno nell'altro, fenomeno che può presentarsi nel caso di un errore commesso all'interno di un ciclo più esterno.

Un paio di esempi per focalizzare meglio la situazione:

```
gap> 3/0;
Error, Rational operations: <divisor> must not be zero not in any
function at line 2 of *stdin*
you can replace <divisor> via 'return <divisor>;'
brk> return 0.5;
6.
gap> Factorial(12/5);
Error, Range: <last> must be an integer less than 2^60 (not a
rational) in return pr( [ 1 .. n ], 1, n ); called from <function
"Factorial">( <arguments> ) called from read-eval loop at line 3
of *stdin* you can replace <last> via 'return <last>;'
brk> quit;
gap>
```

1.4 Variabili e assegnamento

Visto come ottenere dei risultati basilari, e come gestire piccoli problemi che possono sorgere nell'utilizzo di GAP, iniziamo a sfruttare compiutamente le proprietà di un linguaggio di programmazione.

Innanzitutto è utile poter mantenere il risultato di calcoli fatti in precedenza. Per quanto tramite comandi quali **last, last2, last3** sia possibile recuperare i precedenti tre risultati ottenuti nel corso dell'attuale sessione, è conveniente disporre di un mezzo per conservare una quantità maggiore di informazioni per un tempo indefinito.

Un'espressione della forma:

$$var := obj;$$

è detta assegnamento. Con tale comando facciamo in modo che il valore di *obj* sia legato all'identificatore *var* scelto dall'utente.

Obj può essere un numero, così come una stringa, un vettore, una matrice, un gruppo o persino una funzione, come vedremo successivamente.

Ogni componente che in GAP può essere assegnata ad una variabile viene detta *oggetto*. Esempi di input che non corrispondono a oggetti possono essere costruzioni sintattiche del tipo $2 + 3$, per quanto il valore di tale espressione sia invece ovviamente assegnabile (ed è proprio quello l'oggetto che sarà assegnato), così come la stringa "$2 + 3$".

Due oggetti possono essere confrontati tramite l'operatore $=$. Il confronto produrrà un valore booleano.

Da notarsi che il risultato di tale confronto non sancirà la totale identità di due elementi: verificherà il contenuto dei due oggetti, non il fatto che occupino la stessa posizione di memoria (come accade per esempio con comandi analoghi per gli oggetti di Java).

Ogni oggetto appartiene ad una certa *classe*, che raggruppa elementi dotati di particolari proprietà.

Sono presenti alcune particolari funzioni, dette filtri, che consentono di verificare se un oggetto appartiene o meno ad una certa classe. In generale il nome di tali funzioni è **IsClassName**.

Per quanto riguarda restrizioni sui nomi possibili per gli identificatori, è sufficiente con un po' di accortezza evitare caratteri speciali, stringhe che iniziano con una cifra o parole chiave del sistema, una lista delle quali è ottenibile con il comando *GAPInfo.keyords;*.

Un'ultima cosa da notare è che, dopo un assegnamento, GAP stamperà nuovamente il valore dell'oggetto appena assegnato. La cosa può essere utile, se ad esempio vogliamo controllare che il valore assegnato sia quello corretto, ma specialmente se l'output dovesse essere particolarmente esteso, potrebbe essere fonte di disordine. In generale per sopprimere l'output, è necessario aggiungere un ulteriore ; al termine del comando.

Ecco alcuni esempi:

```
gap> t := 5;;
gap> g := 5;;
gap> t+g;
10
gap> last;
10
gap> t = g;
true
gap> t = [1,2,3]; # Questo non è un assegnamento!
false
gap> t := [1,2,3];
[1, 2, 3]
gap> h := [1,2,4];
[1, 2, 4]
gap> t = h;
false
```

Ora che siamo in grado di effettuare assegnamenti, procediamo ad esaminare una breve presentazione della sintassi da utilizzare per costruire diversi oggetti.

Da qui in avanti ometteremo il ; che come si è potuto osservare deve essere sempre presente al termine di un qualunque comando.

1.5 Costruzione e gestione di oggetti elementari, o di interesse

Liste Una lista è una sequenza di elementi tutti appartenenti alla stessa classe. Una lista può avere una lunghezza variabile, ed essere modificata dopo la creazione. Non è necessario dichiarare anticipatamente la lunghezza della lista che si sta andando a creare, addirittura, è possibile costruire liste con "buchi".

La sintassi da seguire è questa:

$$[\ el_1,\ \ldots\ , el_n\ ]$$

Si noti che le componenti sono numerate da 1 a n, non da 0!

Una volta che abbiamo creato una lista e l'abbiamo assegnata ad una variabile, diciamo L, è possibile accedere alle sue componenti con il semplice comando $L[n]$, dove n è l'indice della componente che vogliamo raggiungere, o aggiungerne di nuove in questo modo:

$$L[n] := el$$

In quest'ultimo caso n può essere un qualunque intero positivo, non necessariamente minore rispetto alla lunghezza della lista. (Ovviamente se n corrisponde alla posizione di un elemento già presente, si andrà a sostituirlo)

Funzioni per liste Prese L, M liste di oggetti di una stessa classe, troviamo definite le seguenti funzioni:

- **Length**(L) che ne restituisce la lunghezza
- **Append**(L,M) che attacca la lista M alla fine di L
- **Add**(L,el) che aggiunge l'elemento el coerente con la lista alla fine di L
- **Position**(L, el) restituisce la posizione di el in L o fail se el non è presente
- el **in** L che verifica la presenza dell'elemento el in L
- **Sort**(L) ordina la lista, posto che sia definito per le sue componenti un ordinamento
- **Filtered**(L, func) restituisce una lista di elementi di L tali per cui la funzione func a valori booleani assume il valore true
- **ShallowCopy**(L) produce una copia di L

Range Un Range, è una lista senza buchi composta di interi ordinati che distano l'uno dall'altro un certo passo p indicato, o 1 per default. La sintassi è la seguente:

$$[inizio,\ inizio + passo\ ..\ fine\ ]$$

L'intero fine deve essere ovviamente raggiungibile con l'incremento indicato a partire dall'intero inizio.

Caratteri e stringhe Un qualunque simbolo, stampabile o meno, può essere identificato come carattere con l'apposizione di apici (') a destra e a sinistra dell'espressione che lo identifica. Si noti che anche lo spazio vuoto è un carattere, così come $\backslash n$, che indica l'operazione di andare a capo o $\backslash t$ la tabulazione.

Tra i caratteri vige una forma di ordinamento lessicografico. Secondo tale ordine, ogni carattere speciale è minore di ogni carattere rappresentante una cifra (le cifre sono ordinate secondo il loro corrispettivo valore), che a sua volta è minore di ogni carattere rappresentante una lettera (ordine alfabetico).

È possibile creare espressioni più complesse, composte a partire da più caratteri. Tali oggetti sono detti stringhe. La sintassi per una stringa è la seguente:

$$S := "frase";$$

Le stringhe si comportano in maniera simile alle liste: anche per esse è possibile chiamare le funzioni Length e Position, o accedere alle varie componenti. Ed in effetti, GAP vede le stringhe proprio come liste di caratteri.

Vediamo ora un paio di esempi, in cui osserviamo anche il funzionamento di un filtro:

```
gap> r := [3, 1, 0, 3, -2];;
gap> t := [1, 5, 7, 8];;
gap> Append(r,t);
gap> r;
[ 3, 1, 0, 3, -2, 1, 5, 7, 8 ]
gap> 0 in r;
true
gap> Position(r,0);
3
gap> s := ['d','c','a'];
"dca" Una lista di caratteri viene rappresentata come stringa
gap> Sort(s);
gap> s;
"acd"
gap> IsString(s); Ecco un paio di filtri, i nomi sono significativi
true
gap> IsList(s);
true
gap> x := [1 .. 10];
[ 1 .. 10 ]
```

Vettori e Matrici La creazione di una matrice, e come caso particolare di un vettore, avviene con la seguente sintassi

$$[\ [n_{11},\ \ldots\ ,\ n_{1m}],\ \ldots\ ,\ [n_{h1},\ \ldots\ ,\ n_{hm}]\ ]$$

Si noti che in particolare per ottenere un vettore riga, si dovrà inserire:

$$[\ [v_1\ ,\ \ldots\ ,\ v_n]\ ]$$

Se procediamo in maniera più intuitiva, creando il vettore tramite

$$[\ v_1\ ,\ \ldots\ ,\ v_n\ ]$$

quello ottenuto si comporterà sia come un vettore riga, sia come un vettore colonna, a seconda di quanto richiesto dalla situazione, andando probabilmente a mascherare qualche errore che potremmo avere commesso.

Possiamo operare senza problemi somme e prodotti di matrici e vettori, avendo cura però di controllare la compatibilità degli elementi con cui an-

diamo a lavorare,GAP non darà alcun segnale di errore, e procederà ad effettuare un qualche tipo di operazione con gli input assegnati, restituendo comunque un qualche risultato!

Aggiungiamo solo un metodo utile per la visualizzazione di matrici su campi finiti, **Display**, che, applicato ad un tale oggetto, lo stamperà in una forma più leggibile.

Sono presenti innumerevoli metodi per operare con le matrici, oltre ad operazioni elementari quali trasposizione, rango, determinante, eliminazione di Gauss. Non andremo oltre ciò che è stato detto, in quanto tali metodi non saranno utilizzati nel prosieguo.

Funzioni Come già è stato anticipato, anche una funzione è un particolare oggetto. Esistono due modi per costruire nuove funzioni: il primo metodo è adatto a definizioni brevi, effettuabili in una riga, e presenta la seguente sintassi:

$$func := var \text{ -> } f(var)$$

Ovviamente func sarà il nome della funzione, che poi potrà essere chiamata con la solita sintassi *func(arg)*.

Per funzioni più complesse, dovremo adottare una diversa metodologia:

$$func := function(args) \ ... \ end;$$

In questo caso dobbiamo scrivere il corpo della funzione nello spazio compreso fra function(args), dove args ovviamente rappresenta gli argomenti della funzione, e la dichiarazione end;.

È conveniente, nella prima riga della funzione, utilizzare il comando local a cui devono seguire i nomi delle variabili che saranno utilizzate nel metodo, separate da virgole, che renderà le variabili interne alla definizione della funzione. Ciò significa che, nel caso avessimo utilizzato precedentemente una variabile con lo stesso nome nella sessione, tale variabile manterrà esternamente alla funzione il suo valore (in effetti, è come se utilizzassimo identificatori diversi per le due).

Ancora, se la funzione deve restituire un qualche valore, diciamo salvato nella variabile *output*, dobbiamo inserire prima della dichiarazione conclusiva, il comando return output;

```
gap> f := x -> 2*x;
function( x ) ... end
gap> f(2);
4
gap> f := x -> x[1] + x[2];
function( x ) ... end
gap> f([1,2]);
3
gap> Pmer := x -> IsPrime(2^x -1);
function( x ) ... end
gap> Pmer(2);
true
```

Per il momento non consideriamo esempi che richiedano l'utilizzo della seconda formulazione, in quanto ci mancano ancora i mezzi per produrre risultati di interesse. Riprenderemo tale caso nei successivi capitoli.

1.6 Confronti e operazioni booleane

Già nella sezione (1.1.4) , parlando degli oggetti, abbiamo introdotto l'operatore =. Come è naturale aspettarsi, sono presenti anche gli altri possibili tipi di confronto <, <=, >, >=.

È possibile ottenere operazioni booleane più complesse andando a sfruttare i connettivi logici & e |, che in GAP si indicano semplicemente come *and* e *or*, e la negazione, data da *not*.

1.7 Iterazioni e condizionali

Nei capitoli precedenti abbiamo visto come costruire e gestire alcuni tipi di oggetto, tra cui le funzioni. A questo punto, supponendo un qualche tipo di esperienza di programmazione nel lettore, esaminiamo semplicemente la sintassi di cicli for, while e if, senza spiegarne il significato.

Per il ciclo for si procede in questo modo:

for ind **in** list **do** ... **od**;

ind rappresenta l'indice su cui andiamo ad effettuare il ciclo, *list* rappresenta per l'appunto una lista di elementi (Che non necessariamente sono numeri!).

Il ciclo while ha la seguente sintassi:

while esBool **do** ... **od**;

esBool è un espressione booleana che controlla l'uscita dal ciclo.

Il condizionale si ottiene in questo modo:

if esBool1 **then** ... [**elif** esBool2 **then** ...] (**else** ...) **fi**;

Le istruzioni impartite dopo il primo then vengono eseguite se esBool1 risulta vera, se invece è falsa, si passa agli eventuali elif (possono essere in qualunque numero) che vedono le loro istruzioni eseguite se esBool2 è vera; se anche queste sono tutte false o se non sono presenti elif, sono eseguiti i comandi all'interno di else. Si noti che la parte strettamente necessaria è solo quella esclusa dalle parentesi.

Esiste un'ulteriore espressione che useremo, la quale si comporta sostanzialmente come un ciclo while, ma eseguirà sempre almeno una volta il codice che contiene. La sintassi è la seguente:

repeat ... **until** esBool

A volte ci troveremo ad utilizzare il comando break il cui scopo è per l'appunto interrompere il ciclo più interno all'interno di cui è posto.

Per concludere, prima di un esempio, ricordiamo che per interrompere l'esecuzione di un programma (magari nel caso avessimo fatto un errore nel formulare l'espressione di controllo di un ciclo while) è sufficiente digitare **Ctrl** + **C**.

```
gap>CyclotomicCosetsU := function(p, n)
> local YetOccurred, list, m, g, i, j, s , t;
> if not Gcd(p,n) = 1 then
>     return fail;
> fi;
> YetOccurred := [[]];
> list := [ 0 .. n-1];
> m := 0;
> for i in list do
>     s := false;
>     for t in [1 .. m] do
>         if i in YetOccurred[t] then
>             s := true;
>             break;
>         fi;
>     od;
>     if not s then
>         g := [];
>         j := i;
>         repeat
>             Append(g, [j]);
>             j:= p*j mod n;
>         until j = i;
>         m := m + 1;
>         YetOccurred[m] := g;
>     fi;
> od;
> return YetOccurred;
>end;
function( p, n ) ... end
gap> CyclotomicCosetsU(2,15);
[ [ 0 ], [ 1, 2, 4, 8 ], [ 3, 6, 12, 9 ], [ 5, 10 ], [ 7, 14, 13, 11 ] ]
```

Abbiamo così potuto osservare l'implementazione di una funzione per il calcolo dei p-laterali ciclotomici modulo n. Nell'implementazione si è utilizzata la funzione **Gcd**, che calcola il MCD fra due elementi di un anello euclideo. Analogamente esiste anche la funzione **Lcm** per il mcm.

1.8 Lettura di file

Come si è notato, l'implementazione di una funzione può essere piuttosto articolata, ed è decisamente sconveniente effettuarla direttamente sul terminale.

Procederemo dunque a scrivere il codice in un editor di testo, andando

poi a leggerne il contenuto tramite il comando **Read**("pathToDirectory"). Se abbiamo dei dubbi sull'attuale posizione in cui ci troviamo, tramite il comando **DirectoryHome**(), GAP ci fornirà la stringa iniziale a cui far seguire il percorso fino al file.

Ad esempio pensando di voler recuperare la funzione per i laterali ciclotomici, salvata nel file cyclotomicCosets.g, procediamo nel seguente modo:

```
gap> DirectoryHome();
dir("/home/pc/")
gap> Read("/home/pc/Scrivania/algebra/gap_code/cyclotomicCoset
s.g");
gap> CyclotomicCosetsU(3, 16);
[ [ 0 ], [ 1, 3, 9, 11 ], [ 2, 6 ], [ 4, 12 ], 5, 15, 13, 7 ], [ 8 ], [ 10, 14 ] ]
```

Chapter 2

Gruppi e campi finiti in GAP

A questo punto siamo pronti per affrontare la gestione di strutture algebriche di base tramite GAP. Ancora una volta la trattazione sarà essenziale, andando a considerare solo elementi che possano essere di interesse per i nostri obiettivi.

In particolare ci interessiamo al momento di gruppi di permutazioni.

2.1 Gruppi di permutazioni

Permutazioni Le permutazioni vengono rappresentate in GAP sotto forma di prodotto di cicli disgiunti (per ricordare la giustificazione, si veda ad esempio [3]), pertanto ogni oggetto permutazione sarà creato con la seguente sintassi:

$$P := (a_{11},\ ...,\ a_{1m})...(a_{n1},\ ...,\ a_{nm})$$

Per applicare una permutazione ad un elemento si dovrà usare l'operatore ^ in questo modo:

n^P

Si può ottenere anche la controimmagine di n attraverso P tramite /:

$$n/P$$

Possiamo comporre due permutazioni con l'operatore *, ovviamente la composizione non è commutativa.

Di seguito una breve lista di funzioni per permutazioni:

- **SignPerm**(P) : Restituisce il segno di P
- **Inverse**(P) : Restituisce P^{-1}
- **AsPermutation**(f) : Restituisce una rappresentazione di f come permutazione, o fail, se f non è una funzione valida
- **PermutationMat**(P, m) : Restituisce una matrice di permutazione di ordine m che rappresenta P

Gruppo Simmetrico Gli unici gruppi di cui ci occuperemo, per avere un'idea generale, saranno quelli di permutazioni.

La costruzione del gruppo simmetrico avviene con l'utilizzo del metodo **SymmetricGroup**(n), dove evidentemente n sarà la cardinalità dell'insieme finito su cui si andrà a costruire il gruppo. Più in generale possiamo costruire un gruppo di permutazioni tramite **Group**(P_1, ..., P_n), dove l'argomento del metodo consiste di un elenco di generatori (ovviamente sono permutazioni) del gruppo che si intende creare. *Si noti che il metodo Group può costruire anche altri tipi di gruppi, accettando in input valori diversi.*

Introduciamo anche un metodo per conoscere gli elementi mossi dalle permutazioni in un certo gruppo: **MovedPoints**(G) che restituisce una lista contenente tali elementi.

Sappiamo inoltre da un teorema di Cayley, (si veda [3]), che ogni gruppo G è isomorfo ad un sottogruppo del gruppo simmetrico S_n di un certo ordine. Il metodo **IsomorphismPermGroup**(G), restituisce un isomorfismo adeguato.

Concludiamo questa breve sezione con una lista di alcuni metodi basilari validi per tutti i gruppi. A seguire osserveremo alcuni esempi che andranno a riprendere anche temi del paragrafo precedente.

- **Elements(G)** Restituisce una lista con gli elementi di G
- **Size(G)** Restituisce la cardinalità di G, equivalente a **Order**
- **DerivedSubgroup(G)** Restituisce il sottogruppo generato da tutti i commutatori di G
- **IsCylic, IsSolvableGroup, IsAbelian, IsPerfectGroup** Verificano le varie proprietà (Ne esistono molti altri)
- **AllSubgroups(G), NormalSubgroups(G)** Stampano una lista contenente tutti i sottogruppi e i soli normali rispettivamente
- **StructureDescription(G)** Stampa una stringa contenente informazioni sulla struttura di G, per la spiegazione del significato si rimanda a [2]

```
gap> P := (1,2,5,8);
(1,2,5,8)
gap> 2^P;
5
gap> 1/P;
8
gap> P*(2,3);
(1,3,2,5,8)
```

```
gap> s := (1,2,5,8)(3,4);
(1,2,5,8)(3,4)
gap> Inverse(s);
(1,8,5,2)(3,4)
gap> s*last;
()
gap> S4 := SymmetricGroup(4);
Sym( [ 1 .. 4 ] )
gap> Size(S4);Factorial(4);
24
24
gap> DerivedSubgroup(S4); # Che è proprio il gruppo A4
Alt( [ 1 .. 4 ] )
gap> K := DerivedSubgroup(last);
Group([ (1,4)(2,3), (1,2)(3,4) ])
gap> Elements(K); # Questo è il gruppo di Klein
[ (), (1,2)(3,4), (1,3)(2,4), (1,4)(2,3) ]
gap> IsAbelian(last);
true
gap> IsSolvableGroup(S4); # Ed effettivamente S4 è risolubile
true
gap> S5 := SymmetricGroup(5);
Sym( [ 1 .. 5 ] )
gap> DerivedSubgroup(S5);
Alt( [ 1 .. 5 ] )
gap> DerivedSubgroup(A5);
Group([ (1,4)(2,3), (2,5,3), (2,5)(3,4) ])
gap> last = A5;
true
gap> IsPerfect(A5);
true
gap> IsSolvableGroup(S5);
false
```

A quanto già detto precedentemente aggiungiamo anche l'esistenza di una libreria di gruppi comprendente tutti i gruppi di ordine pari o inferiore a 2000, esclusi quelli di ordine 1024, e numerosi altri gruppi con caratteristiche particolari, per le quali si veda [2]. Il metodo per ricavare un gruppo da questa libreria è **SmallGroup**(order, index). Index è la posizione del gruppo nella lista, il significato di order è palese.

2.2 Campi finiti e polinomi

Per lo studio della teoria dei codici ciclici, siamo interessati a campi finiti e agli anelli dei polinomi su tali campi. In questa sezione affrontiamo la gestione di tali elementi.

Cenni sui campi finiti Da un noto risultato di algebra, si veda [3, 391-396], sappiamo che esiste un campo finito di ogni cardinalità della forma p^m con p primo, corrispondente alla caratteristica del campo, ed m naturale, e che tali campi sono gli unici, a meno di isomorfismi. Pertanto ogni campo finito è completamente descritto, una volta che abbiamo un metodo per produrre campi con p^m elementi.

La costruzione adottata utilizza l'anello degli interi modulo p, chiamiamolo J_p, nel caso m = 1, e quozienti dell'anello dei polinomi su J_p per l'ideale generato da un polinomio primitivo di grado m, per $m \neq 1$.

La capacità di GAP di creare campi finiti di dimensione p, anche per p "grande" è effettivamente illimitata, anche se trovare primi di grandi dimensioni non è certo un problema banale, invece, per campi di ordine p^m, possono sorgere problemi dovuti al costo computazionale dell'individuazione di un polinomio primitivo.

Sostanzialmente lo schema adoperato da GAP per tali costruzioni adopera una particolare classe di polinomi primitivi minimali, detti polinomi di Conway, che se di grado elevato (per esempio superiore a 121 per un polinomio a coefficienti in J_2) può non essere calcolabile in tempi stretti.

Detto ciò, non ci occuperemo ulteriormente di dettagli tecnici, si segnala solamente la funzione **IsCheapConwayPolynomial**(p, m) che verifica se il polinomio di Conway di grado m su J_p sia calcolabile o meno in breve tempo.

Campi di Galois Un campo finito costruito come nel paragrafo precedente viene detto Campo di Galois. La creazione di un tale oggetto avviene in GAP tramite la funzione **GaloisField**(p, m), o in modo più compatto **GF**(p, m). Da notarsi che è possibile passare al metodo un argomento unico della forma q, avendo però cura di controllare che sia effettivamente un parametro valido (i.e.: $q = p^m$).

Per il problema di costo computazionale esposto nel paragrafo precedente, alcuni parametri in entrata produrranno un messaggio di avvertimento, che bloccherà l'esecuzione della funzione è richiederà all'utente di voler confermare la computazione di tale campo, nonostante la spesa ingente in tempo e memoria prevista.

Gli elementi del campo di Galois creato, vengono rappresentati come potenze di un elemento primitivo dello stesso, elemento identificato dal simbolo **Z**(p^m). Se stiamo lavorando con un campo nella forma J_p, tramite il metodo **Int**(elem), ci sarà possibile convertire la rappresentazione in funzione di elementi primitivi in numeri interi.

Aggiungiamo altri quattro metodi utilizzabili per i campi finiti, prima di passare all'esame dei polinomi:

- **Size**(K) : La dimensione del campo
- **Dimension**(K) : Restituisce il grado di K come estensione sul suo sottocampo fondamentale
- **Characteristic**(K) : Restituisce la caratteristica di K
- **PrimeField**(K) : Restituisce il sottocampo fondamentale

```
gap> q := Primes[17]^5;
714924299
gap> K := GF(q);
GF(59^5)
gap> Size(K);
714924299
gap> PrimeField(K);
GF(59)
gap> Characteristic(K);
59
gap> GF(2,122);
Error, Conway Polynomial 2^122 will need to computed and
might be slow return to continue called from FFECON-
WAY.SetUpConwayStuff( p, d ); called from FFECONWAY.ZNC(
p, d ) called from LargeGaloisField( p, d ) called from <function
"GaloisField">( <arguments> ) called from read-eval loop at line 49
of *stdin* you can 'quit;' to quit to outer loop, or you can 'return;'
to continue
brk> quit;
```

Anelli di polinomi su campi finiti Ogni polinomio in GAP viene creato su uno specifico anello, che dipende dalla dichiarazione dell'incognita in cui viene scritto. Per creare dunque un polinomio che prenda valori in un campo finito è necessario anzitutto dichiarare una variabile adeguata:

$$\text{x} := \textbf{Indeterminate}(\text{F, varName})$$

F indica il campo, o più in generale l'anello con unità su cui andiamo a definire la variabile; varName è un argomento opzionale contenente una stringa rappresentante il nome con cui sarà stampata la variabile, *non sarà possibile chiamare l'incognita appena definita tramite varName, a meno che tale stringa non coincida con x.* Per default le incognite sono chiamate x_nr , dove nr è il numero dell'incognita.

Una volta ottenuta l'incognita sul campo desiderato, creare il polinomio è intuitivo:

$$P := a_n * x\hat{}n + ... + a_0$$

È possibile creare anche anelli di polinomi, su un anello desiderato, tramite il metodo **PolynomialRing**(R, [list of varNames]).

Di seguito una lista di metodi per gestire i polinomi:

- **Value**(P, val) : valuta il polinomio P in val
- **RootsOfUPol**([Ring], P) : restituisce le radici di P in Ring in una lista
- **Derivative**(P) : calcola la derivata formale di P
- **Discriminant**(P) : calcola il discriminante di P
- **SplittingField**(P) : calcola il campo di spezzamento di P
- **Factors**([PolRing], P) : restituisce una fattorizzazione di P in elementi di PolRing
- **IsIrreducible**(P) : controlla l'irriducibilità di P
- **RandomPrimitivePolynomial**(F, n) : trova un polinomio irriducibile di grado n su F (F può essere un anello, o una potenza di primo q, nel qual caso F si intende GF(q))
- **ConwayPolynomial**(p, n) : trova il polinomio di Conway di grado n su GF(p)

```
gap> x := Indeterminate(GF(2), "x");
x
gap> P := x^5 + x^4 + x^2 + 1;
x^5+x^4+x^2+Z(2)^0
gap> Factors(P);
[ x+Z(2)^0, x^4+x+Z(2)^0 ]
gap> RootsOfUPol(P);
[ Z(2)^0 ]
gap> Value(P,Z(2)^0);
0*Z(2)
gap> SplittingField(P);
GF(2^4)
gap> RootsOfUPol(GF(2^4),P);
[ Z(2)^0, Z(2^4), Z(2^4)^2, Z(2^4)^4, Z(2^4)^8 ]
gap> R := PolynomialRing(GF(2^4), "x");
GF(2^4)[x]
gap> Factors(R,P);
[ x+Z(2)^0, x+Z(2^4), x+Z(2^4)^2, x+Z(2^4)^4, x+Z(2^4)^8 ]
```

```
gap> q := RandomPrimitivePolynomial(GF(2^4), 4);
x^4+Z(2^4)*x^3+Z(2^4)^9*x^2+x+Z(2^4)^8
gap> x2 := Indeterminate(GF(3),"y");
y
gap> C := ConwayPolynomial(3, 4);
y^4-y^3-Z(3)^0
gap> SplittingField(C);
GF(3^4)
gap> IsIrreducible(q);
true
```

Per la conclusione del capitolo, esponiamo l'implementazione di un metodo per individuare estensioni di campo contenenti n-esime radici dell'unità.

```
FieldWithNthRootOfUnity := function(F, n)
    local p,i, m;
    m := n;
    p := Characteristic(F);
    if m mod p = 0 then
        repeat m := m/p; until not m mod p = 0;
    fi;
    for i in PositiveIntegers do
        if Lcm(m, p^i - 1) = p^i -1 then
            Print("GF(",p,"^",i,")");
            break;
        fi;
    od;
end;
# Ora proviamo ad applicarlo
gap> Read("/home/pc/Scrivania/algebra/gap_code/nthRootOfUni
ty.g");
gap> FieldWithNthRootOfUnity(GF(2), 15);
GF(2^4)
gap> FieldWithNthRootOfUnity(GF(5), 22);
GF(5^5)
gap> x := Indeterminate(GF(5), "x");
x
gap> f := x^22 - 1;
x^22-Z(5)^0
gap> RootsOfUPol(GF(5^4),f);
[ Z(5)^2, Z(5)^0 ]
gap> RootsOfUPol(GF(5^5),f);
[ Z(5)^2, Z(5)^0, Z(5^5)^1704, Z(5^5)^1846, Z(5^5)^1988,
Z(5^5)^2130,
```

```
Z(5^5)^2272,      Z(5^5)^2414,      Z(5^5)^2556,      Z(5^5)^2698,
Z(5^5)^2840,
Z(5^5)^2982, Z(5^5)^142, Z(5^5)^284, Z(5^5)^426, Z(5^5)^568,
Z(5^5)^710,  Z(5^5)^852,  Z(5^5)^994,  Z(5^5)^1136,  Z(5^5)^1278,
Z(5^5)^1420
]
```

Chapter 3

Teoria dei codici correttori

Concluso un lungo preambolo, ci interessiamo finalmente del pacchetto GAP per la teoria dei codici. Al momento dell'avvio del programma, alcuni pacchetti fondamentali verranno caricati per default, il pacchetto per i codici, chiamato "guava", non è fra questi.

Prima di iniziare l'esame del pacchetto, introduciamo dunque il metodo per caricare pacchetti dalla libreria di GAP:

LoadPackage("StringName", [options])

Per quanto ci riguarda l'espressione che useremo sarà sempre: **LoadPackage("guava", false)**.

Il valore booleano false indica al programma di non visualizzare il banner del pacchetto al momento del caricamento. Ad ogni modo non è strettamente necessario e può essere omesso.

Un'ultima nota riguarda a questo punto gli utilizzatori di sistemi operativi diversi da Linux: alcuni metodi di "guava" sono implementati in C e non funzioneranno per Windows, cercheremo di evitare l'utilizzo di tali metodi nella trattazione. Per informazioni più dettagliate e approfondimenti è possibile consultare la guida a "guava", disponibile sul sito indicato in [2].

Sebbene in "guava" sia possibile lavorare con codici costruiti su campi finiti qualsiasi, la maggior parte dei metodi implementati funziona in modo più efficiente per codici binari, pertanto nel seguito ci riferiremo sempre a codici su GF(2).

Iniziamo ad occuparci di alcune operazioni elementari, prima di provare qualche applicazione.

3.1 Creazione e gestione di codici lineari

Codeword Gli elementi di un codice sono un particolare tipo di oggetto con proprietà simili a quelle di una lista, o di un vettore. A differenza, però,

degli oggetti incontrati in precedenza, ogni Codeword è *immutabile*, ossia il suo contenuto è accessibile, ma non può essere riscritto.

Le Codeword vengono create tramite l'omonimo metodo

Codeword(V, [F, n])

In cui V può essere una lista di coefficienti interi, o un polinomio, F rappresenta il campo finito su cui vogliamo creare la Codeword, n la dimensione dell'oggetto da creare.

La rappresentazione standard di una Codeword in GAP è la seguente:

$$[\ c_1\ c_2\ ...\ c_n\ ]$$

Supponendo di aver creato due Codewords e di averle salvate nelle variabili c1, c2 rispettivamente, sarà possibile effettuare alcune operazioni, in maniera simile a quanto accadeva per le liste: c1[i] restituisce il valore dell'i-esimo coefficiente di c1, c1 = c2 verifica l'uguaglianza tra le due parole. Possiamo anche sommare o sottrarre le Codewords semplicemente tramite le operazioni +, -. Quanto le rende invece differenti dalle normali liste è che c1[i] := z, non sarà accettato, per l'immutabilità dell'oggetto.

Se vogliamo modificare una Codeword dobbiamo necessariamente passare attraverso la creazione di una copia mutabile, tramite il metodo ShallowCopy, introdotto per le liste.

Aggiungiamo ora alcuni metodi per la gestione delle Codewords:

- **PolyCodeword**(c) : Restituisce la rappresentazione di c sotto forma di polinomio: se c_i erano le entrate di c, per i in [0, n], avremo l'output $\sum_0^n c_i x^i$, funziona anche se c è una lista di Codewords
- **VectorCodeword**(c) : Restituisce la rappresentazione vettoriale di c, come per il metodo precedente, c può essere una lista di Codewords.
- **Weight**(c) : Restituisce il peso di c, ossia il numero di coefficienti diversi da zero.
- **DistanceCodeword**(c1,c2) : Restituisce la distanza di Hamming tra c1 e c2
- **NullWord**(n[, F]) : Restituisce la parola nulla di lunghezza n sul campo F, per default F = GF(2).

Rimandiamo gli esempi al paragrafo successivo.

Costruire e gestire un codice La costruzione di un codice può avvenire in innumerevoli modi: ci concentreremo su alcuni semplici metodi per la creazione di codici lineari e ciclici.

Possiamo generare codici lineari partendo dalla matrice generatrice M, o dalla matrice di controllo H (parity check matrix), sfruttando i metodi **GeneratorMatCode**(M, F) o **CheckMatCode**(H, F), rispettivamente.

Possiamo anche ottenere alcuni codici specifici, ad esempio i codici di Hamming, tramite **HammingCode**(r, F), in cui il parametro r indica la ridondanza del codice, o il codice di Golay, binario e ternario, rispettivamente tramite **BinaryGolayCode**() e **TernaryGolayCode**(). Per una lista completa di tali metodi particolari si rimanda al manuale di "guava" in [2].

Nel caso mancassimo della motivazione necessaria a creare codici a mano, sono disponibili anche un metodo per generare codici lineari casuali, su F^n di dimensione k fissata tramite **RandomLinearCode**(n, k, F) ed uno per costruire il miglior codice lineare conosciuto, (al Maggio 2006), fissati i precedenti parametri, **BestKnownLinearCode**(n, k, F).

Ora che siamo in grado di costruire qualche codice, andiamo ad esaminare il modo in cui "guava" ce li presenta.

Immaginiamo di creare un nuovo codice C, tramite il comando HammingCode(3), l'output restituito sarà: *a linear [7,4,3]1 Hamming (3,2) code over GF(2).*

GAP dunque stampa alcune informazioni riguardanti il codice appena generato. Secondo la notazione standard i numeri fra parentesi quadre [n, k, d] rappresentano nell'ordine, lunghezza delle parole del codice, dimensione del codice stesso come sottospazio vettoriale, e distanza minima tra le parole del codice.

Il valore subito esterno alle parentesi indica il raggio di copertura (covering radius), ovvero quel valore r per cui, dato un qualunque vettore v appartenente allo spazio ambiente del codice C (in questo caso $GF(2)^7$), $d(v, C) \leq r$, dove d è la distanza di Hamming.

Si noti che qualora questi ultimi due valori non fossero noti, saranno stampati un limite inferiore e superiore per gli stessi.

A seguire troviamo il nome del codice, se notevole, e il campo che funge da sostegno per lo stesso.

Forniamo ora alcuni dei numerosi metodi per lavorare con i codici:

- **Random**(C) : estrae una parola da C secondo un algoritmo pseudocasuale. In generale Random può funzionare su un qualunque insieme di elementi fornito come argomento
- **Size**(C) : restituisce il numero di elementi di C
- **WordLength** : restituisce il parametro n (lunghezza delle parole)
- **Dimension**(C) : restituisce il parametro k
- **MinimumDistance**(C [,w]) : Calcola il parametro d, o semplicemente lo restituisce se già noto. Se viene fornito l'argomento w, ovvero

una codeword appartenente allo spazio ambiente di C, la distanza tra quest'ultima e C sarà computata.

- **MinimumWeight**(C) : [metodo esterno] calcola il peso minimo per codici binari o ternari
- **WeightDistribution**(C) : restituisce una lista contenente il numero delle parole di C dei vari pesi (da 0 a n)
- **AutomorphismGroup**(C) : [metodo esterno] Calcola il gruppo degli automorfismi di C
- **CoveringRadius**(C) : Calcola il parametro r
- **Elements**(C) : restituisce una lista contenente tutti gli elementi di C
- **GeneratorMat**(C) : restituisce la matrice generatrice M, è un *oggetto immutabile*
- **CheckMat**(C) : restituisce la parity check Matrix H, immutabile
- **PutStandardForm**(M) : Modifica (non restituisce) la matrice M portandola nella forma standard (I_k | A). M deve essere mutabile
- **Syndrome**(C, v) : Calcola la sindrome del vettore v
- **SyndromeTable**(C) : Calcola l'omonima tabella
- **PuncturedCode**(C, h) : restituisce il codice C*, ottenuto rimuovendo la colonna h dalla matrice generatrice di C
- **ExtendedCode**(C [, i]) : restituisce il codice C esteso i volte

Concludiamo con esempi di applicazione:

```
gap> H := HammingCode(3);
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> Display(GeneratorMat(H));
1 1 1 . . . .
1 . . 1 1 . .
. 1 . 1 . 1 .
1 1 . 1 . . 1
gap> IsMutable(GeneratorMat(H));
false
gap> M := ShallowCopy(GeneratorMat(H));;
gap> PutStandardForm(M);;
```

```
gap> Display(M);
1 . . . . 1 1
. 1 . . 1 . 1
. . 1 . 1 1 .
. . . 1 1 1 1
gap> Display(CheckMat(H));
. . . 1 1 1 1
. 1 1 . . 1 1
1 . 1 . 1 . 1
gap> c := Random(H);
[ 1 0 1 0 1 0 1 ]
gap> c in H;
true
gap> Elements(H);
[ [ 0 0 0 0 0 0 0 ], [ 0 0 0 1 1 1 1 ], [ 0 0 1 0 1 1 0 ], [ 0 0 1 1 0 0 1 ],
[ 0 1 0 0 1 0 1 ], [ 0 1 0 1 0 1 0 ], [ 0 1 1 0 0 1 1 ], [ 0 1 1 1 1 0 0 ],
[ 1 0 0 0 0 1 1 ], [ 1 0 0 1 1 0 0 ], [ 1 0 1 0 1 0 1 ], [ 1 0 1 1 0 1 0 ],
[ 1 1 0 0 1 1 0 ], [ 1 1 0 1 0 0 1 ], [ 1 1 1 0 0 0 0 ], [ 1 1 1 1 1 1 1 ] ]
gap> WeightDistribution(H);
[ 1,  0,  0,  7,  7,  0,  0,  1 ]
gap> AutomorphismGroup(H);
Group([ (1,2)(5,6), (2,4)(3,5), (2,3)(4,6,5,7), (4,5)(6,7), (4,6)(5,7) ])
gap> StructureDescription(last);
"PSL(3,2)"
gap> B := BinaryGolayCode();
a cyclic [23,12,7]3 binary Golay code over GF(2)
gap> Display(GeneratorMat(B));
1 . 1 . 1 1 1 . . . 1 1 . . . . . . . . . . .
. 1 . 1 . 1 1 1 . . . 1 1 . . . . . . . . . .
. . 1 . 1 . 1 1 1 . . . 1 1 . . . . . . . . .
. . . 1 . 1 . 1 1 1 . . . 1 1 . . . . . . . .
. . . . 1 . 1 . 1 1 1 . . . 1 1 . . . . . . .
. . . . . 1 . 1 . 1 1 1 . . . 1 1 . . . . . .
. . . . . 1 . 1 . 1 1 1 . . . 1 1 . . . . . .
. . . . . 1 . 1 . 1 1 1 . . . 1 1 . . . . .
. . . . . 1 . 1 . 1 1 1 . . . 1 1 . . . .
. . . . . 1 . 1 . 1 1 1 . . . 1 1 . . .
. . . . . 1 . 1 . 1 1 1 . . . 1 1 . .
. . . . . 1 . 1 . 1 1 1 . . . 1 1 .
. . . . . 1 . 1 . 1 1 1 . . . 1 1
gap> IsLinearCode(B);
true
gap> BE := ExtendedCode(B);
a linear [24,12,8]4 extended code
```

```
gap> Display(GeneratorMat(BE));
1 . 1 . 1 1 1 . . . 1 1 . . . . . . . . . . 1
. 1 . 1 . 1 1 1 . . . 1 1 . . . . . . . . . 1
. . 1 . 1 . 1 1 1 . . . 1 1 . . . . . . . . 1
. . . 1 . 1 . 1 1 1 . . . 1 1 . . . . . . . 1
. . . . 1 . 1 . 1 1 1 . . . 1 1 . . . . . . 1
. . . . . 1 . 1 . 1 1 1 . . . 1 1 . . . . . 1
. . . . . 1 . 1 . 1 1 1 . . . 1 1 . . . . . 1
. . . . . 1 . 1 . 1 1 1 . . . 1 1 . . . . . 1
. . . . . 1 . 1 . 1 1 1 . . . 1 1 . . . . . 1
. . . . . 1 . 1 . 1 1 1 . . . 1 1 . . . . . 1
. . . . . 1 . 1 . 1 1 1 . . . 1 1 . . . . . 1
. . . . . 1 . 1 . 1 1 1 . . . 1 1 . . . . . 1
gap> c := Random(B);
[ 1 0 0 1 1 0 0 0 1 1 1 1 0 0 1 1 0 1 1 0 0 1 0 ]
gap> PolyCodeword(c);
x_1^21+x_1^18+x_1^17+x_1^15+x_1^14+x_1^11+x_1^10+
x_1^9+x_1^8+x_1^4+x_1^3+Z(2)^0
gap> Weight(c);
12
gap> d := Random(B);
[ 1 1 0 0 1 1 0 0 1 1 1 0 0 0 1 1 0 0 1 0 1 0 1 ]
gap> DistanceCodeword(c, d);
8
```

Codifica e decodifica Introdotti i metodi per la costruzione di codici e parole, ci interessa ora il processo di codifica e decodifica di un messaggio.

La codifica può avvenire secondo almeno due diversi procedimenti, consideriamo un messaggio m, che può essere inserito come una stringa contenente le cifre nel messaggio, e un codice lineare C che abbiamo già creato in un primo momento.

La sintassi più semplice per la codifica prevede di utilizzare semplicemente l'operatore *, componendo un'espressione della forma **m*C**.

Il risultato di tale operazione è identico a quello che otteniamo moltiplicando a sinistra il vettore v, le cui entrate contengono le cifre componenti il messaggio, per la matrice generatrice del codice C, e applicando il metodo Codeword al prodotto.

Se il messaggio che viene fornito è sottodimensionato rispetto al parametro k, le entrate mancanti verranno per default considerate contenenti 0. Nel caso in cui il messaggio fosse sovradimensionato, le entrate in eccesso verranno semplicemente ignorate.

Come nel caso dei prodotti matrice vettore, non avremo nessun segnale di avvertimento se uno dei precedenti fatti incorresse; è necessario pertanto mantenere un'attenzione costante a quanto stiamo facendo.

Per quanto riguarda la decodifica, esiste un apposito metodo, chiamato appunto **Decode**(C,r), dove C è il codice lineare con il quale lavoriamo, r il messaggio ricevuto.

Dopo aver individuato la parola c appartenente a C più vicina ad r, il metodo restituirà una codeword m tale che m*C = c.

Se abbiamo una parola r la cui appartenenza a C è già stata verificata, possiamo ricavare il messaggio iniziale senza passare attraverso Decode, usando direttamente il metodo **InformationWord**(C,c).

Osserviamo alcuni esempi di codifica prima di passare all'esame di un esercizio abbastanza corposo.

```
gap> C:= BestKnownLinearCode(13,5,GF(2));
a linear [13, 5, 5]4..5 shortened code
gap> c := "10"*C;
[ 0 1 1 0 0 1 1 0 0 0 0 0 1 ]
gap> c := "10000"*C;
[ 0 1 1 0 0 1 1 0 0 0 0 0 1 ] # Le tre cifre mancanti equivalgono a 0
gap> c := "10110"*C;
[ 0 1 0 1 1 1 0 0 1 1 1 0 0 ]
gap> c := "1011011"*C;
[ 0 1 0 1 1 1 0 0 1 1 1 0 0 ] # Le cifre in eccesso sono ignorate
gap> M := GeneratorMat(C);;
gap> v := [1,0,1,1,0];
[ 1, 0, 1, 1, 0 ]
gap> Codeword(v*M);
[ 0 1 0 1 1 1 0 0 1 1 1 0 0 ] # Più laborioso, ma equivalente
gap> c := "10010"*C;
[ 0 1 0 0 1 1 1 1 0 0 1 0 0 ]
gap> InformationWord(C,c);
[ 1 0 0 1 0 ] # L'operazione opposta conviene effettuarla così
gap> d := ShallowCopy(c);; d[1] := d[1] + 1;; d[3] := d[3] + 1;; d :=
Codeword(d);
[ 1 1 1 0 1 1 1 1 0 0 1 0 0 ] # Abbiamo introdotto 2 errori
gap> d in Elements(C);
false
gap> Decode(C,d);
[ 1 0 0 1 0 ] # Il messaggio è stato recuperato (ricordiamo d = 5)
gap> d := ShallowCopy(c);; d[1] := d[1] + 1;; d[3] := d[3] + 1;; d[4]
+ 1;; d[5] + 1;; d := Codeword(d);
[ 1 1 1 0 1 1 1 1 0 0 1 0 0 ] # Questa volta gli errori sono 4
```

```
gap> Decode(C,d);
[ 1 0 0 1 0 ] # Il messaggio è ancora corretto, ma se consideriamo...
gap> d := ShallowCopy(c);; d[1] := d[1] + 1;; d[3] := d[3] + 1;; d[6]
:= d[6] + 1;; d := Codeword(d);
[ 1 1 1 0 1 0 1 1 0 0 1 0 0 ] # Gli errori sono 3, funzionerà?
gap> Decode(C,d);
[ 1 1 0 0 0 ] # Il messaggio recuperato non è quello corretto
```

3.2 Comunicazione secondo il modello BSC

Andiamo ora a creare una simulazione di comunicazione, secondo il modello BSC (Binary Symmetric Channel). Ricordiamo brevemente che tale semplice modello consente di simulare l'invio di un singolo bit alla volta e fissa una probabilità di incrocio p, dove l'incrocio rappresenta il cambiamento di valore del bit inviato.

Per la trattazione del modello si veda [1, pag. 39].

Tutti i metodi costruiti nel seguito verranno salvati nel file SimulazioneCodifica.g.

Il nostro primo compito sarà dunque realizzare un metodo in grado di replicare l'idea del BSC. La cosa è abbastanza semplice, abbiamo infatti a disposizione la funzione Random, per simulare la distribuzione di probabilità desiderata, e gli interi modulo 2 per rappresentare il bit.

L'implementazione proposta è la seguente, i rappresenta il valore del bit spedito, p la crossover probability espressa come razionale:

```
BSC := function(i,p)
    local r,s;
    s:=[0*Z(2),Z(2)];
    if not i in s then
        return fail; # blocca il metodo se i ∉ GF(2)
        break;
    fi;
    r := Random(1,1000)/1000;
    if r < 1-p then
        return i;
    else
        return s[(Int(i)+1) mod 2 + 1]; # scambio dei valori
    fi;
end;
```

Ora abbiamo a disposizione il desiderato BSC, tuttavia per la simulazione, probabilmente spedire lunghe liste di cifre non risulterà partico-

larmente significativo, addirittura può divenire difficile accorgersi di errori qualora si verifichino.

Implementiamo dunque un metodo per rappresentare messaggi costituiti da stringhe. Ci accontenteremo di una selezione di caratteri limitata a 32 elementi, prendendo solo le lettere minuscole non accentate e qualche segno di punteggiatura.

Un veloce conto ci mostra dunque che, lavorando su GF(2), avremo bisogno di Information words di lunghezza 5.

Creiamo dunque un metodo che trasformi il nostro messaggio in una lista di vettori, e uno che effettui l'operazione inversa. Inoltre, notando che spedire numerosi vettori un bit alla volta sarebbe un'impresa assai lunga, implementiamo un ulteriore metodo che ci consenta di inviare i messaggi in blocco.

```
alph := [ 'a', 'b', 'c', 'd','e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '.', ',', ';', ':', '?' ,' ',];;
Num := ["00000", "00001", "00010", "00011", "00100", "00101", "00110",
"00111", "01000", "01001", "01010", "01011", "01100", "01101", "01110",
"01111", "10000", "10001", "10010", "10011", "10100", "10101", "10110",
"10111", "11000", "11001", "11010", "11011", "11100", "11101", "11110",
"11111"];;
cdNm := Codeword(Num, 5, GF(2));;

# Nel seguito text è la stringa contenente il messaggio, alph l'alfabeto in
# uso, cd, l'insieme delle codewords corrispondenti, codice il codice scelto
#per trasmetterle, mex la lista di codewords ricevute

TextToCodewords := function(alfabeto, cd, codice, text)
    local m, G, i;
    m := [];
    G := GeneratorMat(codice);# Codifichiamo tramite matrice
    for i in [1 .. Length(text)] do
        m[i] := cd[Position(alfabeto,text[i])];
        m[i] := m[i]*G;
    od;
    return m;
end;

DecodeText := function(alfabeto, cd, codice, mex)
    local testo, letter, i;
    testo := "";
    for i in [ 1 .. Length(mex)] do
        letter := Decode(codice, mex[i]);
        testo[i] := alfabeto[Position(cd, letter)];
```

```
    od;
    return testo;
end;

# Le due seguenti implementano la spedizione di una lista di Vettori
# p è sempre la crossover probability, v un vettore, a una lista di vettori

Send := function(v,p)
    local i, s, output;
    s:= [];
    for i in [1 .. v!.WordLength] do
        s[i] := BSC(v[i],p);
    od;
    output := Codeword(s);
    return output;
end;

Trasmetti := function(a,p)
    local t, i, j;
    t := [];
    j := 1;
    for i in a do
        t[j] := Send(i, p);
        j := j + 1;
    od;
    return t;
end;
```

A questo punto siamo quasi soddisfatti, andiamo solo ad introdurre una funzione per simulare la codifica tramite Whole Space Code, ovvero un codice di dimensione n, con matrice generatrice I_n, ed un ultimo metodo, che simulerà la comunicazione risparmiandoci la chiamata dei metodi.

```
TextToWholeSpaceCodewords := function(alfabeto, cd, text)
    local m, G, i;
    m := [];
    for i in [1 .. Length(text)] do
        m[i] := cd[Position(alfabeto,text[i])];
    od;
    return m;
end;

DecodeWholeSpaceText := function(alfabeto, cd, mex)
    local testo, letter, i;
```

```
    testo := "";
    for i in [ 1 .. Length(mex)] do
        letter := mex[i];
        testo[i] := alfabeto[Position(cd, letter)];
    od;
    return testo;
end;

SimulaComunicazione := function( mex, p)
    local v, w, z, mes, C1, C2;
    C1 := BestKnownLinearCode(13, 5, GF(2));
    C2 := BestKnownLinearCode(20, 5, GF(2));
    v := TextToWholeSpaceCodewords(alph, cdNm, mex);
    w := TextToCodewords(alph, cdNm, C1, mex);
    z := TextToCodewords(alph, cdNm, C2, mex);
    v := Trasmetti(v, p);
    w := Trasmetti(w, p);
    z := Trasmetti(z, p);
    mes := DecodeWholeSpaceText(alph, cdNm, v);
    Print("Il messaggio recuperato senza un codice     correttore","\n",mes,"\n");
    mes := DecodeText(alph, cdNm, C1, w);
    Print("Il messaggio ottenuto con un codice lineare di lunghezza 13, d =
    5","\n",mes,"\n");
    mes := DecodeText(alph, cdNm, C2, z);
    Print("Il messaggio ottenuto con un codice lineare di lunghezza 20, d =
   9","\n",mes,"\n");
end;
```

L'ultimo metodo confronterà l'operato di due codici lineari particolari, con l'equivalente dello Whole Space Code (che ricordiamo non corregge alcun errore). Sarà ovviamente possibile a chi fosse interessato tentare simulazioni con codici differenti, applicando minimi cambiamenti all'ultima funzione.

Ecco un esempio di quello che possiamo ottenere:

```
gap> Read("/home/pc/Scrivania/algebra/gap_code/SimulazioneCo
difica.g");
gap> mex := "stiamo facendo un test";;
gap> p := 1/40;;
gap> SimulaComunicazione(mex, p);
Il messaggio recuperato senza un codice correttore
stia;g fagundo en xest
```

```
Il messaggio ottenuto con un codice lineare di lunghezza 13, d = 5
stiamo facendo un test
Il messaggio ottenuto con un codice lineare di lunghezza 20, d = 9
stiamo facendo un test
gap> p := 1/12;; # Esageriamo un po'le dimensioni di p
gap> SimulaComunicazione(mex, p);
Il messaggio recuperato senza un codice correttore
qtiaeo fecendm un te.r
Il messaggio ottenuto con un codice lineare di lunghezza 13, d = 5
stiamo facendo u; test
Il messaggio ottenuto con un codice lineare di lunghezza 20, d = 9
stiamo facendo un test
```

3.3 Codici ciclici

Ci focalizziamo, nella conclusione di questa dispensa, sui codici ciclici.

Diciamo che un codice C è ciclico se, preso un suo qualunque elemento c, e applicandogli uno shift circolare, otteniamo un elemento c' ancora in C. (Ricordiamo che uno shift è una permutazione per cui ogni componente viene spostata in quella alla sua destra, o alla sua sinistra, e l'ultima componente viene spostata nella prima, o la prima nell'ultima, secondo il verso scelto).

Un codice ciclico può essere rappresentato come ideale di $F_q[x]/(x^n - 1)$.

Costruzione di codici ciclici In base a quanto abbiamo precedentemente visto, uno specifico codice può essere in generale creato tramite la matrice generatrice o la parity check. Data la particolare struttura dei codici ciclici, a queste due possibilità ne aggiungiamo due, sostanzialmente equivalenti, date dai metodi **GeneratorPolCode**(g, n, F) e **CheckPolCode**(h, n, F). Per entrambi i metodi, con la già utilizzata notazione, F indica il campo che funge da sostegno, n la lunghezza delle parole. I due parametri g ed h invece conterranno rispettivamente il polinomio generatore e il polinomio di controllo, soddisfacenti la proprietà $gh = x^n - 1$. Inoltre per ogni parola c(x) del codice ciclico con polinomio generatore g, vale $c(x) = g(x)m(x)$ e $R_{x^n-1}c(x)h(x) = 0$.

Da notare che nel caso il polinomio g passato al metodo non fosse un divisore di $x^n - 1$, il codice sarà generato a partire dal polinomio risultante da Gcd($x^n - 1$, g). Lo stesso vale per il polinomio h.

Per procurarci polinomi adatti allo scopo, evitando di procedere per tentativi, è conveniente, una volta stabilita la lunghezza del codice desiderato ed il campo da utilizzare, sfruttare il metodo Factors per ottenere una lista dei fattori di $x^n - 1$ che andremo poi ad utilizzare per costruire il polinomio g.

Anche nel caso dei codici ciclici esistono metodi specifici per particolari famiglie di codici; citeremo soltanto i metodi **BCHCode**(n [, b], δ, F) e **RootsCode**(n, list).

Per quanto riguarda il primo, il parametro δ indica la distanza designata, ovvero un limite inferiore per la minima distanza del codice BCH che andiamo a costruire, b è l'esponente della radice primitiva dell'unità che andremo a prendere come primo elemento dell'insieme delle radici del codice, per default vale 1.

Il secondo costruisce un codice ciclico a partire dalla lista delle radici del suo polinomio generatore. Per ottenere una radice primitiva è utile il metodo **PrimitiveUnityRoot**(p, n) che individua una n-esima radice primitiva dell'unità in un'estensione di F_p. Per poter sfruttare il limite BCH è conveniente adoperare il metodo **CyclotomicCosets**(p, n) per controllare i p-laterali ciclotomici modulo n.

Per una dettagliata spiegazione riguardante i codici BCH e le limitazioni inferiori per la distanza minima rimandiamo a [1, 151-].

Aggiungiamo i tre seguenti metodi specifici per i codici ciclici:

- **CheckPol**(C) : restituisce il polinomio h
- **GeneratorPol**(C) : restituisce il polinomio g
- **RootsOfCode**(C) : restituisce la lista delle radici di g

```
gap> a := PrimitiveUnityRoot(2, 15);
Z(2^4)
gap> CyclotomicCosets(2,15);
[ [ 0 ], [ 1, 2, 4, 8 ], [ 3, 6, 12, 9 ], [ 5, 10 ], [ 7, 14, 13, 11 ] ]
gap> C1 := RootsCode(15, [a^0, a, a^3]); a cyclic [15,6,2..6]4..7 code
defined by roots over GF(2)
gap> MinimumDistance(C1);
6
gap> g := GeneratorPol(C1);
x_1^9+x_1^6+x_1^5+x_1^4+x_1+Z(2)^0
gap> Value(g, a^4);
0*Z(2)
gap> RootsOfCode(C1);
[ Z(2)^0, Z(2^4), Z(2^4)^2, Z(2^4)^3, Z(2^4)^4, Z(2^4)^6,
Z(2^4)^8, Z(2^4)^9, Z(2^4)^12 ]
gap> x := Indeterminate(GF(2));; p := x^15 - 1;;
gap> Factors(p);
```

```
[ x_1+Z(2)^0, x_1^2+x_1+Z(2)^0, x_1^4+x_1+Z(2)^0,
x_1^4+x_1^3+Z(2)^0, x_1^4+x_1^3+x_1^2+x_1+Z(2)^0
]
gap> g := last[1]*last[3]*last[4];
x_1^9+x_1^7+x_1^6+x_1^3+x_1^2+Z(2)^0
gap> C := GeneratorPolCode(g, 15, GF(2));
a cyclic [15,6,1..6]4..7 code defined by generator polynomial over
GF(2)
gap> h := p/g;
x_1^6+x_1^4+x_1^3+x_1^2+Z(2)^0
gap> C2 := CheckPolCode(h, 15, GF(2));
a cyclic [15,6,1..6]4..7 code defined by check polynomial over GF(2)
gap> C = C2;
true
```

Codifica e decodifica di codici ciclici Ci occupiamo a questo punto della codifica per i codici ciclici. Oltre ai metodi generali validi per i codici lineari, possiamo procedere tramite moltiplicazione per il polinomio generatore. In questo caso dobbiamo ovviamente fornire il messaggio sotto forma di polinomio, o alternativamente passare attraverso i metodi Codeword e PolyCodeword applicati in quest'ordine alla stringa contenente il messaggio.

Per recuperare l'information word da una parola del codice, in questo caso possiamo realizzare un metodo decisamente semplice, sfruttando l'eguaglianza $m = c/g$:

```
InformationWordCyclic := function(C,c)
    local g, mex;
    g := GeneratorPol(C);
    c := PolyCodeword(c);
    mex := c/g;
    mex := Codeword(mex);
    return mex;
end;
```

Per la decodifica, GAP fornisce un algoritmo specifico per i codici ciclici, che sfrutta le proprietà del gruppo degli automorfismi del codice stesso, **PermutationDecode**(C,c).

Nell'ultimo paragrafo presenteremo un algoritmo di decodifica alternativo, appartenente comunque alla famiglia degli algoritmi di decodifica per permutazione.

Meggit Decoding Per la teoria riguardante l'algoritmo di decodifica presentato, rimandiamo a [1, 158-].

In generale comunque l'idea è la seguente: ogni parola appartenente al codice verificherà l'uguaglianza $R_g(c) = 0$, pertanto se per un errore riceviamo y = c(x) + e(x), la nuova parola produrrà un resto differente da zero. Calcoleremo dunque un valore legato all'errore commesso per un certo numero di errori possibili, e sfrutteremo le proprietà del codice ciclico per individuare gli errori tramite traslazioni.

Il primo metodo necessario calcolerà la tabella contenente la sindrome desiderata.

```
x := Indeterminate(GF(2));

MeggitSyndromeT := function(C, t)
    local Table, NewTable, Result, Result2, j, n,k, i,s,g;
    Table := SyndromeTable(C);
    g := GeneratorPol(C);
    n := WordLength(C);
    k := C!.Dimension;
    NewTable := [];
    NewTable[1] := Table[1];
    j := 2;
    for i in [2 .. Length(Table)] do
        if Table[i][1][n] = Z(2)^0 then
            NewTable[j] := Table[i];
            j := j + 1;
        fi;
    od;
    j := 1;
    Result2 := [];
    for s in [1 .. Length(NewTable)] do
        if Weight(NewTable[s][1]) <= t then
            Result2[j] := NewTable[s];
            j := j+1;
        fi;
    od;
    Result := [];
    Result[1] := [Codeword(NullVector(n)),Codeword(NullVector(n-k))];
    for i in [2 .. Length(Result2)] do
        Result[i] := [ 0, 0];
    od;
    for i in [ 2 .. Length(Result2)] do
        Result[i][1] := Result2[i][1];
        Result[i][2] := (x^(n-k)*PolyCodeword(Result[i][1])) mod g;
        Result[i][2] := Codeword(Result[i][2], n-k);
    od;
```

```
    return Result;
end;
```

Di seguito il processo di decodifica vero e proprio: come precedentemente per la simulazione di codifica, anche per questo metodo salveremo il codice in un file di testo, che chiameremo Meggit.g in cui includeremo anche il metodo precedente e InformationWordCyclic.

```
MeggitDecoding := function(c, C)
    local Table, pol, k, n, i, j, corr, found, S, t;
    if c in Elements(C) then
        return c;
    fi;
    pol := PolyCodeword(c);
    t := QuoInt(MinimumDistance(C) -1, 2);
    Table := MeggitSyndromeT(C, t);
    k := C!.Dimension;
    n := WordLength(C);
    found := false;
    S := (x(n-k)*pol) mod C!.GeneratorPol;
    S := Codeword(S);
    for i in [1 .. Length(Table)] do
        if S = Table[i][2] then
            corr := Table[i][1];
            found := true;
            break;
        fi;
    od;
    if not found then
        j := 1;
        while not found do
            pol := pol*x;
            S := (x^(n-k)*pol) mod C!.GeneratorPol;
            S := Codeword(S, n-k);
            for i in [1 .. Length(Table)] do
                if S = Table[i][2] then
                    corr := Table[i][1];
                    found := true;
                    break;
                fi;
            od;
            j := j + 1;
        od;
        corr := -(x^(n - j + 1)*PolyCodeword(corr)) mod (x^n - 1);
```

fi;
return(c + corr);
end;

I metodi precedenti sono un po' più complicati di quelli visti in precedenza, ma si tratta semplicemente di un'implementazione passo passo dell'algoritmo di Meggit.

Si noti anche che tale implementazione è ben lungi dall'essere efficiente, e i tempi richiesti per lavorare con codici con n abbastanza elevato sono ingenti.

Per concludere la dispensa ne verifichiamo il funzionamento con un esempio:

```
gap> C := BCHCode(15, 5, GF(2));
a cyclic [15,7,5]3..5 BCH code, delta=5, b=1 over GF(2)
# La distanza minima è 5 quindi almeno 2 errori saranno corretti
gap> c := "1001"*C; # Codifica
x^11 + x^10 + x^9 + x^8 + x^6 + x^4 + x^3 + 1
gap> d := ShallowCopy(c);; d[1] := d[1] + 1;; d[3] := d[3] + 1;; d:=
Codeword(d);; # Inseriti due errori
gap> MeggitDecoding(d,C);
[ 1 0 0 1 1 0 1 0 1 1 1 1 0 0 0 ]
gap> InformationWordCyclic(C, last);
x^3 + 1
gap> Codeword(last);
[ 1 0 0 1 ] # E ci siamo i due errori sono stati corretti
```

Bibliography

[1] W.Cary Huffman e Vera Pless, *Fundamentals of error correcting codes* , Cambridge

[2] *GAP – Groups, Algorithms, and Programming*, Version 4.7.8, The GAP Group (2015), http://www.gap-system.org.

[3] I. N. Herstein, *Algebra*, Editori Riuniti