Pràctiques Integrades
1er de Matemàtiques

Tipus de llistes. Programació.

Curs 2003–04

Índex

Part II
Tipus de llistes. Programació.

6 Llistes, conjunts i seqüències

El programa Maple permet treballar amb diferents *conjunts* o estructures de dades. Aquests *conjunts* es poden definir de diverses maneres i a més en podem canviar l'estructura d'un tipus a un altre segons ens interessi. Les estructures de dades que analitzarem en aquesta pràctica són les llistes, conjunts, seqüències i taules.

6.1 Llistes

Una llista és un conjunt ordenat d'expressions on es permeten repeticions. L'entrada al Maple es fa entre claudàtors (`[ ]`) i separant els elements que conté amb comes.

Exemple 6.1

Definim la variable a com la llista següent $[1, 2, 3, 2, 1, 2]$:

```
> a:=[1,2,3,2,1,2];
```

Els elements d'una llista es poden cridar o recuperar un a un mitjançant la posició que ocupen escrivint aquesta entre claudàtors. Una particularitat d'aquest mètode és que quan utilitzem nombres negatius retorna l'element corresponent començant per la cua.

Exemple 6.2

Si volem recuperar o extreure el tercer element de la llista `a` escriurem:

```
> a[3];
```

Mentre que si volem l'últim podem fer-ho amb la comanda següent:

```
> a[-1];
```

De la manera semblant també podem extreure una subllista de la llista original. Si volem una subllista amb els elements entre les posicions i i j cal introduir l'argument `i..j` dins els claudàtors.

Exemple 6.3

En aquest exemple recuperarem la subllista que conté els elements des de la posició 2 fins a la 4.

```
> a[2..4];
```

Una manera de construir llistes a partir d'altres és mitjançant la unió, és a dir, també podem construir una llista com la unió de vàries. Per a això necessitem abans "*treure*" els claudàtors de les llistes inicials i afegir-los després a la unió o concatenació obtinguda. La comanda que ens permet treure els claudàtors és la comanda `op( )`. Un cop hem tret els claudàtors podem concatenar dues llistes. Per exemple, observeu el funcionament de la comanda `op` executant la següent línia de Maple.

```
> op(a);
```

Exercici 6.1

Definiu la funció `concatenar`, depenent de dues variables, que retorni la unió de dues llistes donades. Comproveu que la definició que heu fet és correcta fent la prova amb les llistes $a = [1, 2, 3, 2, 1]$ i $b = [3, 2, 4]$. El resultat ha de ser la llista $c = [1, 2, 3, 2, 1, 3, 2, 4]$.

6.2 Conjunts

En un conjunt de dades no hi ha repeticions i les dades no tenen un ordre prefixat (és a dir, Maple pot variar l'ordre, en funció del context). L'entrada a Maple es fa entre claus ({ }) i separant els elements que conté per comes. La majoria d'opcions que hem vist per les llistes són també vàlides per als conjunts, però tenint en compte les possibles diferències.

Exemple 6.4

Definiu b com el conjunt $\{1, 2, 5, 1, 3\}$ i observeu la sortida que us torna.

```
> b:={1,2,5,1,3};
```

Els elements d'un conjunt també es poden cridar mitjançant els claudàtors, tenint en compte que, a priori, no coneixem l'ordre en que Maple guarda els elements ja que són estructures no ordenades de dades.

Exemple 6.5

Per a cridar el tercer element del conjunt b fem:

```
> b[3];
```

Observeu que el tercer element no coincideix amb el que hem introduït inicialment en la posició 3 quan hem definit b.

Exercici 6.2

Definiu a i b com els conjunts $\{x, y, z\}$ i $\{t, u\}$. Definiu la unió d'aquests dos conjunts $c = a \cup b$. Construïu una funció que, a partir de dos conjunts, doni com a resultat la seva unió. Proveu-la amb els conjunts $A = \{1, 4, 9, 16\}$ i $B = \{2, 4, 6, 8, 10\}$.

6.3 Seqüències

Una seqüència d'elements és un conjunt ordenat però que s'interpreta com n arguments (i no una sola llista), on n és el nombre d'elements de la seqüència. L'entrada al Maple es realitza amb les dades separades entre comes i sense cap delimitador.

Exemple 6.6

Si volem definir com a la seqüència $1, 4, 9, 16$ posem

```
> a:=1,4,9,16;
```

Una altra manera d'aconseguir seqüències és mitjançant la comanda `seq` que ja va ser introduïda en el primer bloc i vàreu utilitzar en alguns exemples.

Exemple 6.7

La seqüència c amb els primers 10 quadrats enters es construeix amb la comanda `seq` de la manera següent.

```
> c:=seq(i^2,i=1..10);
```

Igual que fèiem en el cas de les llistes o conjunts podem extreure o recuperar un element d'una seqüència, unir seqüències, ...

Exercici 6.3

Definiu la funció de dues variables $f(x, y) = x^2 - y^2$. Considereu $a = (5, 7)$. Com s'ha d'introduir a per aconseguir que en escriure $f(a)$ Maple doni com a resultat el valor -24?

6.4 Taules

Totes les estructures per emmagatzemar dades que hem vist fins ara admeten només una indexació mitjançant nombres enters. Maple té una estructura de dades més sofisticada que admet com a conjunt de indexació pràcticament qualsevol cosa. És l'estructura anomenada `table`. Com veurem més endavant fins i tot admet cadenes de caràcters com a índexs. La millor manera de veure com funciona es mitjançant uns exemples. Ja sabeu que podeu aprendre més de les seves propietats utilitzant l'ajuda de Maple.

Exemple 6.8

En aquest exemple veurem com es defineix aquesta estructura i el seu comportament respecte assignació de valors.

```
> T:=table([1=1,2=3,3=5,x=u,y=v,z=w,t=sin(2*t)]);
> T[1];
> T[2];
> T[x];
> T[5];
> t:=5;
> T[t];
> T[5];
> subs(u=3,T[x]);
```

I podem afegir-hi nous elements,

```
> T[5]:=m;
> T[5];
```

Com heu pogut observar el conjunt de índexs és qualsevol cosa i per tant, no és immediat saber el que s'ha de posar entre claudàtors per a recuperar cada un dels elements d'una estructura `table`. De la feina de recuperar el conjunt d'índexs per als que s'ha introduït algun valor en una taula se n'encarrega la comanda `indices`.

Exemple 6.9

```
> indices(T);
```

Exemple 6.10

En aquest exemple utilitzarem el format `table` d'estructura de dades per crear un petit diccionari.

```
> Traductor:=table();
> Traductor[dilluns]:=Monday;
> Traductor[dimarts]:=Tuesday;
> Traductor[dimecres]:=Wednesday;
> Traductor[dijous]:=Thursday;
> Traductor[divendres]:=Friday;
> Traductor[dissabte]:=Saturday;
> Traductor[diumenge]:=Sunday;
> Traductor[dimecres];
> Traductor[divendres];
```

Fixeu-vos que la taula `Traductor` s'ha definit inicialment sense cap assignació de valors i que cada un dels valors s'ha afegit en les comandes posteriors.

6.5 Canvis de format entre llistes, seqüències i conjunts. La comanda `convert`

Maple permet transformar una expressió que té guardada com una llista, seqüència o conjunt als altres formats.

Un d'aquests canvis ja l'hem utilitzant. A l'apartat corresponent a les llistes ja hem vist com podem canviar una llista de dades en una seqüència, mitjançant la comanda `op( )`.

Exemple 6.11

Convertirem la llista $a = [1, 2, 3, 2, 1]$ en una seqüència b.

```
> a:=[1,2,3,2,1];
> b:=op(a);
```

Un cop tenim una seqüència podem transformar-la un altre cop en una llista simplement afegint els delimitadors que caracteritzen les llistes: els claudàtors.

Exemple 6.12

Si volem transformar la seqüència b en una llista només cal afegir els claudàtors:

```
> [b];
```

El procediment anàleg funciona amb els conjunts i les seqüències. Recordeu que els delimitadors que caracteritzen els conjunts són les claus. és molt important que tingueu en compte que quan passem una seqüència o una llista a format conjunt perdem les repeticions i l'ordre que tenien originalment.

Exercici 6.4

Convertiu la llista $a = [3, 2, 1, 4, 2, 2, 2]$ en un conjunt b i feu una nova llista c amb els elements de b.

Una altra manera de fer conversions entre llistes, seqüències i conjunts és amb la comanda `convert( )`. Aquesta comanda és molt versàtil i s'utilitza també per a canvis en altres contexts dins de Maple (taules a matrius, ...).

Per a passar d'un format a l'altre tan sols cal conèixer la terminologia de Maple corresponent a cada un d'ells: `set` per a conjunts i `list` per a llistes. Per seguretat escriurem aquestes paraules entre comes ja que si haguessin estat definides com a variables serien substituïdes pel seu valor i així només són una paraula.

Exemple 6.13

Considerem la llista a formada pels elements $a = [3, 2, 1, 4, 3]$. Podem passar-la a format conjunt mitjançant:

```
> a:=[3,2,1,4,3];
> b:=convert(a,'set');
```

I podem tornar al format llista mitjançant:

```
> convert(b,'list');
```

6.6 Comptar elements de llistes i conjunts

La funció que permet comptar el nombre d'elements d'una llista o conjunt és la funció `nops( )`.

Exemple 6.14

Volem saber quants elements diferents té la llista $a = [3, 4, 2, 1, 3, 5, 3, 4, 3, 4, 5, 3, 2, 2, 4]$. La funció `nops( )` aplicada directament a la llista

```
> a:=[3,4,2,1,3,5,3,4,3,4,5,3,2,2,4];
> nops(a);
```

retorna el nombre d'elements de la llista (amb repeticions incloses). Una manera de saber quants elements diferents té és convertir primer la llista a conjunt b i seguidament comptar el nombre d'elements del conjunt:

```
> b:=convert(a,'set'); nops(b);
```

Exercici 6.5

Quants elements té la llista `a:=[[1,2,3],[x,y],{2, 3,4}, x, 1, 2]`? Raoneu la resposta.

6.7 Exemple: grups de permutacions

Recordeu que una permutació d'un conjunt de n elements $\{1, 2, \ldots, n\}$ és una aplicació bijectiva:

$$\begin{array}{rccc} \sigma: & \{1,2,\ldots,n\} & \longrightarrow & \{1,2,\ldots,n\} \\ & i & \mapsto & \sigma(i) \end{array}$$

El conjunt de totes aquestes permutacions es designa per S_n.

Les permutacions s'acostumen a escriure mitjançant la següent notació:

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & n \\ \sigma(1) & \sigma(2) & \sigma(3) & \cdots & \sigma(n) \end{pmatrix}$$

o bé mitjançant un producte de cicles disjunts.

Per a treballar amb permutacions utilitzant Maple necessitem carregar el paquet **group** que conté comandes específiques per a la seva manipulació.

```
> with(group);
```

6.7.1 Entrar elements

Inicialment, la manera més immediata d'introduir una permutació en Maple és mitjançant les imatges de cada posició posades en una llista de la forma `[`$\sigma(1), \ldots, \sigma(n)$`]`. Ara bé, per la majoria de funcions és necessari escriure-la en la seva descomposició en cicles disjunts i això es fa introduint una *llista de llistes* de la forma `[`$c_1, \ldots, c_k$`]`, on cada c_i és una llista que representa un cicle. En particular l'element neutre (la permutació identitat) es representa en aquest context per una llista buida (`[ ]`).

Exemple 6.15

Per a entrar la permutació $\sigma = (1,2,3)(4,5)$ (producte dels cicles $(1,2,3)$ i $(4,5)$) del grup simètric S_5 definim

```
> sigma:=[[1,2,3],[4,5]];
```

En determinats casos la manera com tindrem definida una permutació serà com una llista ordenada amb les imatges de cada posició, o sigui, `[`$\sigma(1), \sigma(2), \ldots, \sigma(n)$`]`. Tot i això, si volem operar amb ella haurem de transformar-la a un producte de cicles disjunts mitjançant la comanda `convert`. En el següent exemple veurem com passar d'un format a un altre.

Exemple 6.16

Si considerem la permutació $\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 1 & 3 & 5 \end{pmatrix}$ i la volem introduir amb aquesta notació escriurem,

```
> sigma:=[2,4,1,3,5];
```

Si la volem descriure com a producte de cicles disjunts utilitzarem la comanda `convert` de la manera següent,

```
> convert(sigma,'disjcyc');
```

Tenint en compte que sempre podem recuperar la notació inicial utilitzant també la comanda `convert`.

```
> convert(%,'permlist',5);
```

on l'argument 5 és per a concretar que és un element de S_5.

Exercici 6.6

Donada la permutació `tau` expressada com a producte dels cicles disjunts `[[1,3,5],[2,7,6]]`, podeu preveure quina diferència hi ha entre els resultats que s'obtenen quan executeu les comandes `convert(tau, 'permlist', 7)` i `convert(tau, 'permlist', 10)`? I si intenteu fer `convert(tau, 'permlist', 5)`?

6.7.2 Operacions amb permutacions

Les permutacions es poden multiplicar (composar) i invertir. En aquest apartat veurem com realitzar aquestes operacions usant comandes del paquet **group** que hem carregat.

Per a multiplicar elements utilitzem la funció `mulperms( )`.

Exemple 6.17

Per a multiplicar les permutacions $\sigma_1 = (1,2,3)(4,5)$ amb $\sigma_2 = (3,4)$ escriurem:

```
> sigma1:=[[1,2,3],[4,5]];
> sigma2:=[[3,4]];
> mulperms(sigma1,sigma2);
```

Què ha calculat: $\sigma_1 \circ \sigma_2$ o bé $\sigma_2 \circ \sigma_1$?

Per a calcular la inversa d'una permutació s'utilitza la funció `invperm( )`.

Exemple 6.18

Calculem la inversa de la permutació $(1,2,3)(5,6,4)$:

```
> sigma:=[[1,2,3],[6,5,4]];
> beta:=invperm(sigma);
```

Podem comprovar que `beta` és la inversa de `sigma`.

```
> mulperms(sigma,beta);
> mulperms(beta,sigma);
```

Exercici 6.7

Considereu les permutacions següents de S_8: $\sigma_1 = (1,2,3)(6,7,8)$, $\sigma_2 = (5,4,3,1)$ i $\sigma_3 = (2,3,4,5,6,7)$. Calculeu les composicions $\sigma_1 \circ \sigma_2 \circ \sigma_3$, $\sigma_3 \circ \sigma_2 \circ \sigma_1$, $\sigma_1^{-1} \circ \sigma_2^{-1} \circ \sigma_3^{-1}$, $\sigma_3^{-1} \circ \sigma_2^{-1} \circ \sigma_1^{-1}$, $(\sigma_1 \circ \sigma_2 \circ \sigma_3)^{-1}$ i $(\sigma_3 \circ \sigma_2 \circ \sigma_1)^{-1}$.

Quines relacions veieu entre els resultats que heu obtingut? Recordeu que el producte $\sigma_2 \circ \sigma_1$ correspon a la comanda de Maple `multperms(sigma1,sigma2)`.

6.7.3 Signe d'una permutació

Maple té implementada la funció signe d'una permutació amb el nom de `parity( )`. L'argument és una permutació que ha d'estar definida com a producte de cicles disjunts i aleshores Maple ens retorna els valors 1 o -1 depenent del signe de la permutació.

Exemple 6.19

Per calcular el signe de les permutacions $(2,1,3,5)(4,9,10)$ i $(3,4,1)(5,6,7)$ amb Maple ho faríem de la manera següent:

```
> parity([[2,1,3,5],[4,9,10]]);
> parity([[3,4,1],[5,6,7]]);
```

Exercici 6.8

Considereu la permutació de S_{10} donada per $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 6 & 9 & 10 & 4 & 7 & 8 & 2 & 1 & 5 \end{pmatrix}$. Determineu el seu signe.

6.7.4 Ordre d'una permutació

Recordeu que l'ordre d'una permutació σ és el més petit nombre natural n tal que $\sigma^n = \sigma \circ \overset{(n)}{\ldots} \circ \sigma$ és la permutació identitat. Per tant, si es vol calcular aquest ordre es pot anar calculant les potències successives de σ fins a obtenir per primer cop la identitat. Naturalment, anar fent `mulperms(mulperms(...(`σ,σ`)),`σ`)` no és gaire pràctic. Aquest procés es pot simplificar si es defineix prèviament una funció `ms( )` que multipliqui qualsevol permutació per σ, d'aquesta manera només s'ha de fer `ms(ms(ms...(`σ`)))`. Maple preveu poder fer aquest tipus d'operació d'una manera ràpida utilitzant l'operador d'iteració `@@` que permet escriure `(ms@@n)(`σ`)`, on `n` és el nombre de cops que apliquem `ms`, per a abreujar l'operació anterior.

Exercici 6.9

Donada la permutació $\sigma = (2, 3, 5, 4, 6, 1, 8, 7)$ de S_8, definiu la funció `ms` que aplicada sobre una permutació qualsevol τ doni com a resultat la permutació producte de σ per τ.

Utilitzant la funció `ms` i l'operador `@@` determineu l'ordre (i totes les potències diferents) de σ.

7 Programació: lògica, iteracions i procediments

No sempre Maple tindrà les comandes que necessitem per a realitzar algun treball o càlcul concret, o bé ens agradaria iterar l'execució d'un paquet de comandes vàries vegades. En aquests casos necessitarem definir les nostres pròpies comandes (o funcions). Per això hem d'aprendre com crear procediments (comandes pròpies o blocs de programa) on podrem incloure funcions lògiques, iteracions i altres procediments.

Si ja heu programat en algun llenguatge informàtic fàcilment podreu reconèixer els operadors i funcions lògiques que apareixeran al llarg de la pràctica.

7.1 Lògica: `if-then-elif-else-end if`

La comanda bàsica que ens permetrà implementar raonaments lògics és la comanda `if` amb totes les seves variants. Farà que Maple executi diferents seqüències de comandes en funció del compliment d'unes condicions (que Maple avaluarà com certes o falses). El cas més simple correspon a `if...then...end if` que executarà una seqüència de comandes si es compleix una condició.

```
> restart;
> a:=3;
> if a=4 then b:=a+1 end if;
> if a=3 then b:=a end if;
```

Ara bé, aquest operador lògic `if` admet moltes variants que ens permetran combinar diferents condicions. Per exemple l'operador `else` que precedeix el codi que caldrà executar en el cas que no es compleixi la condició. També admet l'operador `elif` que ens permetrà considerar més d'una condició. En l'exemple següent podem veure el seu funcionament.

Exemple 7.1

Suposem que volem definir una funció `f(x)` que valgui -1 si $x < 0$, 0 si $x = 0$ i 1 si $x > 0$. Podríem fer-ho de la següent manera utilitzant `if`.

```
> f:=x-> if x<0 then -1 elif x=0 then 0 else 1 end if;
```

Comproveu que funciona executant:

```
> f(-.5);f(0);f(.5);
```

De fet, la funció que acabem de definir a l'exemple anterior ja existeix en Maple amb el nom de `signum()`.

```
> signum(-.5);signum(0);signum(.5);
```

El que hem fet és definir la funció `f` on a la seva definició hauríem de traduir `if` per "*si*", `elif` per "*altrament si*", `else` per "*altrament*" i `end if` per "*hem acabat*". Llavors podem pensar la comanda `if` com una comanda amb tres elements:

- El primer element que hem de tenir en compte és la sintaxi de la funció. Hem d'entrar els arguments `if ... then, elif ... then, else,` i `end if` sempre en aquest ordre. Els operadors `if..then..end if` són necessaris, la resta són opcionals.
- El segon és la introducció de condicions lògiques després dels corresponents `if`. Això són expressions com $x > 0$, $y \leq 3$, ... i que en Maple s'han d'introduir mitjançant els operadors relacionals següents:

`<`	més petit que,
`<=`	més petit o igual que,
`>`	més gran que,
`>=`	més gran o igual que,
`=`	igual,
`<>`	diferent.

Quan Maple troba alguna d'aquestes expressions l'avalua com una "*expressió Booleana*", o sigui, com una expressió que pot ser "*certa*" o "*falsa*". Les avaluacions booleanes d'expressions també es poden realitzar directament mitjançant la funció `evalb( )`. Analitzeu el resultat obtingut a l'executar la següent línia de comanda:

```
> evalb(7>5);evalb(3=4);evalb(3<>4);
```

Podem fer construccions més complicades, combinant els operadors anteriors amb els operadors lògics `and`, `or`, `xor` i `not`. Podeu consultar l'ajuda per a obtenir més detalls sobre els operadors lògics escrivint `?boolean;`.

- El tercer element que formarà part d'una comanda lògica són les comandes que s'han d'executar en cada un dels casos que es poden produir. En l'exemple tan sols havíem d'introduir -1, 0 i 1, però es poden introduir expressions més complexes, Per exemple, volem definir una funció `g` que prengui valors segons la següent definició:

```
> g:=x-> if x<0 then 1/(1+x^2) else cos(x) end if;
```

Podem comprovar que funciona amb alguns exemples,

```
> g(-1.5);g(2.5);
```

Tot i això, què passa en el següent exemple?

```
> g(Pi);
```

El missatge d'error diu que no ha pogut avaluar la condició booleana que hem introduït. Això és a causa del fet que `Pi` no és exactament un número. De fet, obtenim exactament el mateix error quan avaluem `g` a una variable indeterminada x:

```
> g(x);
```

Una manera de solucionar el problema és utilitzant la comanda `evalf()` i avaluar `g` en una aproximació numèrica de `Pi`.

```
> g(evalf(Pi));
```

Finalment intentem ara representar gràficament la funció `g` que hem definit:

```
> plot(g(x),x=-10..10);
```

Tornem a obtenir errors en les expressions booleanes. El problema és que quan substitueix x a la comanda `g` aquesta encara no és un número i per això retorna l'error. En aquest cas el que podem fer és introduir la "*notació d'operadors*", o sigui, no escriure la variable `x`. Recordeu que no era necessària a l'hora d'usar la comanda `plot` per representar gràficament una funció (no una expressió).

```
> plot(g,-10..10);
```

Però si volem utilitzar la funció `g` per a definir altres funcions, per exemple, si volem calcular la seva derivada, també tindrem problemes. Vegeu-ho en el següent exemple.

```
> diff(g(x),x);
```

La funció `if` és molt útil en segons quins contexts, però si el que volem és definir una funció a trossos, no és la més apropiada com acabem de veure. Per a fer això és més versàtil la funció `piecewise( )`. Podem redefinir la funció `g` utilitzant aquesta nova comanda:

```
> g:=x->piecewise(x<0, 1/(1+x^2),x>=0, cos(x));
```

Fixeu-vos en els paràmetres que necessita, és a dir, hem d'introduir entre els parèntesis les corresponents seqüències de "condició, funció" separades per comes. Si escrivim una última funció sense cap condició prèvia, aquesta és la que executarà en cas que no es compleixin cap de les condicions anteriors en la definició.

Ara podem treballar amb `g` com amb qualsevol altre funció, per exemple,

```
> g(Pi); g(a);
> plot(g(x),x=-5..5);
```

i calcular la seva derivada:

```
> diff(g(x),x);
```

Per tant per a definir funcions a trossos utilitzarem la comanda `piecewise`, mentre que reservarem `if-then-elif-else-end if` per a utilitzar-la en altres ocasions.

Exercici 7.1

Utilitzeu la comanda `piecewise` per a definir amb Maple la funció que avalua la següent funció i feu la gràfica a l'interval $(-4, 4)$.

$$f(t) = \begin{cases} \dfrac{(2+t)^3}{6} & \text{si } t \in [-2,-1) \\ \dfrac{-3t^3-6t^2+4}{6} & \text{si } t \in [-1,0) \\ \dfrac{3t^3-6t^2+4}{6} & \text{si } t \in [0,1) \\ \dfrac{-(t-2)^3}{6} & \text{si } t \in [1,2) \\ 0 & \text{si } t \notin [-2,2) \end{cases}$$

Segons com feu la definició haureu d'introduir rangs amb límit inferior i límit superior. Per a això hem d'introduir dues desigualtats, i això ho hem de fer amb l'ajuda de l'operador lògic `and`. (Encara que si s'ordenen adequadament les diferents condicions es pot escriure una comanda sense `and`).

7.2 Iteracions: `for`, `do`, `end do` i `while`

Una iteració consisteix en executar una secció de codi vàries vegades. Moltes comandes a Maple contenen iteracions: les comandes `sum`, `fsolve`, ... La comanda `sum` és semblant a la comanda `seq` que ja coneixeu amb la diferència que retorna la suma total de la seqüència de valors obtinguts. Com a exemple, calculem la suma dels enters entre 1 i 100 elevats al quadrat:

```
> sum(i^2,i=1..100);
```

En alguna part del procés Maple ha aplicat una iteració per a fer el càlcul. Tot i això algun cop necessitarem definir les nostres pròpies iteracions. Fet "a mà" hauríem fet:

Iteració suma:

Decidim primer el nom de la variable, i la inicialitzem a zero.

```
> resp:=0;
```

Seguidament cal introduir la iteració. Per a això utilitzem les comandes `for... from... by... to... while... do... end do`. El bloc de comandes a iterar s'ha d'introduir sempre entre `do` i `end do`.

```
> for i from 1 to 100 do
  resp:=resp + i^2;
  end do;
```

Per a evitar la sortida de tots els passos, podem canviar el punt i coma (`;`) per uns dos punts (`:`) i demanar el valor de la variable `resp` al final del grup de comandes.

```
> resp:=0; for i from 1 to 100 do
> resp:=resp + i^2; end do:
> resp;
```

Exercici 7.2

Calculeu la suma de tots els nombre senars entre 1 i 99 utilitzant la comanda `for` (consulteu l'ajuda per veure la opció `by` dins de la comanda `for`).

Més iteracions:

A vegades no sabrem el nombre exacte d'iteracions que volem fer sinó que la informació que tindrem serà que hem d'iterar un bloc de comandes fins que alguna condició fixada es compleixi. Per exemple, què passa si pitgeu indefinidament la tecla "*cosinus*" de la vostra calculadora, començant pel valor 5? A l'exemple següent ho farem 20 vegades:

```
> x:=5.; for i from 1 to 20 do x:=cos(x);
> end do;
```

Podeu observar que sembla que la successió es va apropant a un valor determinat. De fet, podeu comprovar que aquest valor ha de complir l'equació $\cos(x) = x$. Tot i això, veureu que amb 20 cops no n'hi ha prou per a obtenir una successió que no variï (fixant 8 dígits correctes), per tant, el que volem fer és iterar el procediment fins que la diferència entre dos passos sigui inferior a un número fixat, per exemple, 10^{-8}.

Necessitem dues variables, una serà `x` i la segona serà `xold`, que en tot moment serà el valor de la iteració anterior (haurem de donar-li un valor inicial diferent del que donem a `x`).

```
> x:= 5.; xold:= 0.;
```

Llavors hem d'iterar mentre es compleixi que $|\texttt{x} - \texttt{xold}| > 10^{-8}$. La comanda la podríem entrar de la manera següent.

```
> while abs(x-xold)>1e-8 do
  tempx:=cos(x):
  xold:=x:
  x:=tempx:
  end do;
> xold; x;
```

En qualsevol iteració l'única part obligatòria és el `do..end do`. Les parts `for` i `while` són opcionals i serveixen per controlar el número d'iteracions que cal fer per obtenir el resultat desitjat.

Les diferents maneres de realitzar iteracions de blocs de comandes es poden combinar, és a dir, podem tenir un `while` dins d'un `for`.

Exercici 7.3

Llegiu el següent bloc de comandes i abans d'executar-lo escriviu el que vosaltres creieu que serà la sortida. La comanda `isprime( )` és una funció Booleana que retorna el valor `true` o `false` segons si el número és primer o no.

```
> for p from 1000 by -1 while not isprime(p) do
  end do;
  p;
```

Hi ha una comanda que permet aturar una iteració en qualsevol punt del bloc de comandes que iterem: `break`. Observeu el següent exemple,

```
> for p from 1000 by -1 do
  if isprime(p) then break end if;
  end do;
  p;
```

7.3 Procediments

Al definir una funció pot passar que necessitem incloure-hi vàries iteracions, condicions lògiques, altres funcions, ... En general potser necessitarem dues o més instruccions de Maple encadenades per a fer els càlculs. També és bastant comú que ens aquests casos utilitzem variables que només tenen sentit dins el càlcul de la funció que estem definint (mentre aquesta s'executi), per buidar-se en acabar l'execució, i no a la resta del full de treball. Això és el que anomenarem **variables locals**, mentre que les que són vàlides a tot el full de treball les anomenarem **variables globals**.

Per a definir funcions que inclouen variables locals i globals hem d'utilitzar la funció `proc`. El procediment o bloc de programa està definit per `proc ... end proc`. Al costat de `proc` hem de definir els paràmetres que el nostre procediment utilitzarà. Després de fixar els paràmetres, cal declarar les variables que utilitzarem, això es realitza amb la comanda `local` si són locals (es destruiran en acabar l'execució del programa) i/o amb la comanda `global` si són globals.

La millor manera d'entendre la sintaxi de definició d'un procediment és analitzant un exemple concret.

Exemple 7.2

Mireu l'estructura de la definició següent d'un procediment:

```
> f:=proc(x,y)
  local c,b,z;
  global d;
  c:=3.;b:=17.;
  z:=c*b*x*y*d;
  end proc;
```

Podem veure com s'executa aquesta definició provant:

```
> f(5,5);
> d:=2.; f(5,5);
> d:='d'; f(5,5);
> c; b; z;
```

Exemple 7.3

La funció següent avalua el cosinus d'un angle en graus:

```
> cosdeg:=proc(theta)
  local resultat,phi;
  phi:=theta*Pi/180;
  resultat:=cos(phi);
  end proc;
```

Proveu calculant el cosinus de 45 graus.

Si voleu veure tots els passos que segueix un procediment des dels paràmetres que li passem fins al resultat final, ho podeu fer amb la funció `trace`. Aquesta funció permet "*activar*" per pantalla els passos que hi ha dins de la funció `proc`.

Exemple 7.4

Per a activar la funció `cosdeg` hem de fer:

```
> trace(cosdeg);
```

llavors l'execució fa:

```
> cosdeg(45);
```

Si volem desactivar aquesta la visualització podem fer:

```
> untrace(cosdeg);
```

7.4 Exercicis

Els exercicis següents s'han pensat per a que practiqueu els diferents conceptes de "*programació*" que hem introduït, però també pretenen il·lustrar alguns conceptes matemàtics. No us oblideu d'interpretar els resultats que aneu obtenint.

Exercici 7.4

Considereu la funció:

$$\texttt{aprcos}(x,n) = \sum_{i=0}^{n} (-1)^i \frac{x^{2i}}{(2i)!}$$

on x és un paràmetre real i n és un enter positiu. Dibuixeu en un sol gràfic les funcions `cos(x)`, `aprcos(x,1)`, `aprcos(x,2)` i `aprcos(x,4)` per a `x` entre els valors -2π i 2π.

Exercici 7.5

Definiu una funció que depengui de tres paràmetres `aprox(funcio,x,iteracions)` i que retorni una llista amb els valors de la forma `[x + (1/i),f(x + (1/i) )]` per a `i=1..iteracions`.

Avalueu `aprox` a la funció $\dfrac{\sin(x)}{x}$, per a $x = 0$ amb 50 iteracions.

Modifiqueu la funció `aprox` per a que a més retorni un dibuix dels punts calculats.

Exercici 7.6

Definiu una funció `ordre` tal que donada una permutació σ en calculi el seu ordre determinant el primer nombre natural n tal que σ^n és la permutació identitat.

Calculeu l'ordre de les permutacions $\sigma_1 = (1,3,5)(2,4)$ i $\sigma_2 = (1,2,3,4,5)(6,7,8)$ utilitzant la funció `ordre` que acabeu de definir.

Exercici 7.7

Feu un procediment que donat n faci la llista de totes les permutacions de n elements, cada una en format de llista (és a dir, la permutació σ es representa per la llista `[`σ`(1), ...`σ`(n)]`).

8 Programació recursiva. La successió de Fibonacci

8.1 Programació recursiva

Un procediment es diu recursiu si es crida a ell mateix d'una manera directa o indirecta. En aquesta mena de programació heu d'anar molt en compte ja que podríeu crear un programa que es cridés a ell mateix indefinidament per això heu de tenir clares dues coses: l'algorisme recursiu i les condicions que fan que la recursió s'acabi.

L'exemple més clàssic és el de crear un procediment que calculi el factorial d'un número natural, $n!$. Ja sabeu que Maple té una comanda que el calcula directament però tot i això crearem la nostra comanda pròpia `Factorial( )`.

El factorial d'un número enter positiu es defineix com

1. $0! = 1$.
2. $n! = n \cdot (n-1)!$ si $n > 0$.

Observeu que aquesta és una fórmula recursiva amb una condició inicial.

Exercici 8.1

Construïu un procediment `Factorial` que calculi el factorial segons la fórmula anterior. A més, feu que aparegui un error per pantalla quan $n < 0$. Per escriure missatges per pantalla podeu fer servir les comandes `printf` o `print`, amb l'ajuda de Maple observeu les seves diferències.

Amb la comanda `trace` observeu què passa quan calculeu el factorial de -1, 0 i 4. En l'últim cas veureu com el procediment es crida diferents vegades.

8.2 La successió de Fibonacci

La successió dels nombres de Fibonacci f_n és la que s'obté a partir de les condicions $f_1 = f_2 = 1$ i $f_n = f_{n-1} + f_{n-2}$ per a n més gran o igual a 3.

Exercici 8.2

Definiu una funció `fibon(n)` que doni com a resultat el `n`-èssim nombre de Fibonacci. Comproveu que funciona correctament calculant `fibon(-1)`, `fibon(0)`, `fibon(1)`, `fibon(2)`, `fibon(3)` i `fibon(4)`.

Fibonacci va descobrir aquesta successió de números l'any 1202 quan se li va proposar el següent problema que consistia en analitzar la velocitat amb que es reproduïen els conills en condicions ideals. Més concretament, el problema era el següent. Suposem que en un camp tancat s'hi introdueix una parella de conills (mascle i femella) acabats de néixer. Els conills s'aparellen per primer cop al cap d'un mes d'haver

nascut i neix una nova parella al final del següent mes. A partir de llavors les parelles s'aparellen i crien cada mes. Suposem que els nostres conills no es moren i que a cada cria obtenim una nova parella (un mascle i una femella). La pregunta que se li va fer a Fibonacci és: quantes parelles tindrem al final del primer any? Anem a comptar.

Al final del primer mes només tinc una parella que s'han aparellat.

Al final del segon mes la femella produeix una nova parella així doncs en tenim dues.

Al final del tercer mes, tenim les dues parelles anteriors més una nova parella que ha produït la femella original (la nova femella encara és al seu primer més i no ha criat), així en portem tres.

Al final del quart mes tenim les tres parelles anteriors més dues noves parelles que han nascut de les femelles del segon més (les femelles nascudes al mes anterior encara no han criat).

I així, successivament. Podeu observar la relació d'aquest problema amb la successió de números que hem definit a l'exercici 8.2?

Exercici 8.3

Quina és la resposta al problema de Fibonacci?

Exercici 8.4

Feu un gràfic dels nombres `fibon(n)` per a `n=3..20`. és a dir, representeu els punts de la forma `[n,fibon(n)]`.

Exercici 8.5

Feu un gràfic de la funció `ln(fibon(n))` pels mateixos valors de l'exercici anterior. Què observeu? A partir d'aquesta gràfica, què podeu deduir del tipus de creixement dels valors de l'exercici anterior?

Exercici 8.6

Que podem deduir del gràfic del quocient `fibon(n+1)/fibon(n)` per a `n=1..50`? Aquests nombres representen la proporció entre dos números consecutius a la successió de Fibonacci. Feu una llista amb aquests valors. Què observeu?

A l'exercici anterior haureu observat que la proporció entre dos números consecutius a la successió de Fibonacci estabilitza cap a un número. Aquest número s'anomena el número auri o la raó aurea. Si teniu curiositat podeu buscar informació sobre aquest número i veureu que apareix a molts fenòmens naturals.

Exercici 8.7

Analitzant les gràfiques vistes fins ara, que podríem deduir dels valors de la successió `fibon(n)`?

Exercici 8.8

La successió de Fibonacci ha estat definida de forma recursiva però també té una fórmula pel terme general que ens calcula el terme n-éssim directament. Utilitzeu la funció `rsolve` (mireu l'ajuda de Maple per veure com funciona) per a trobar el terme general de la successió de Fibonacci.

Observeu que passa quan es substitueixen diferents valors de n a l'expressió que retorna la comanda `rsolve` i en calculeu una aproximació numèrica.

Calculeu una aproximació numèrica de $\frac{1+\sqrt{5}}{2}$ i compareu-la amb el valor obtingut a l'exercici 8.6.

Exercici 8.9

Què tenen a veure aquests valors amb el quocient de les longituds dels costats del DNI o d'una tarja de crèdit?

9 Límits de successions.

En aquesta secció veurem com definir, estudiar i calcular límits de successions de números reals amb Maple.

9.1 Experimentació amb els valors d'una successió.

Una successió no és res més que una funció que permet obtenir un valor associat a cada nombre natural. Maple permet definir funcions (i per tant successions), avaluar-les i fins i tot representar-ne gràficament els valors.

Exercici 9.1

Definiu una funció (`n->expr`) o un procediment `proc` que permeti calcular (i ens retorni) els valor de la successió a_n per a qualsevol número natural n donat. Feu una llista amb els 20 primers termes i un gràfic d'aquests valors per a les successions següents:

- $a_n = \dfrac{2\,n-5}{n^2+1}$.
- $a_n = \dfrac{1}{n\,(n+1)}$.
- $a_n = n^2 \ln(n)$.
- La successió que compleix $a_1 = 0$ i $a_{n+1} = -{a_n}^2 + 4\,a_n - 2$.
- La successió que compleix $a_1 = 1$ i $a_{n+1} = -{a_n}^2 + 4\,a_n - 2$.
- La successió que compleix $a_1 = 2.8$ i $a_{n+1} = -{a_n}^2 + 4\,a_n - 2$.
- La successió que compleix $a_1 = 1$ i $a_{n+1} = \sqrt{a_n + 6}$.
- La successió que compleix $a_1 = 3$ i $a_{n+1} = \dfrac{{a_n}^2 - 1}{2\,a_n - 3}$.

Exercici 9.2

Calculeu per a diferents valors dels paràmetres a i b els valors dels 20 primers termes de les successions donades per:

- $x_n = \sqrt[n]{a^n + b^n}$.
- $x_n = \dfrac{a^n - b^n}{a^n + b^n}$.

Després d'aquests càlculs, quin valor creieu que tindrà el límit d'aquestes successions en funció de a i b?

Com ja haureu vist, una de les formes típiques de definir una successió sense donar explícitament quin és el valor de cada un dels termes en funció del seu índex és la de donar un primer terme (a_1) i després expressar com cada un dels altres termes a_{n+1} és funció de l'anterior $a_{n+1} = \mathrm{f}(a_n)$. És ben conegut que en molts casos el límit d'una successió d'aquest tipus és un valor l que satisfà la relació $l = \mathrm{f}(l)$. Amb Maple es pot il·lustrar aquest fet amb una certa facilitat com veureu en els següents exercicis.

Exercici 9.3

Feu una funció o procediment `proc` amb nom `recurr(n,ini,f)` que tingui com arguments un índex `n`, un valor inicial `ini`, i el nom d'una funció `f` (que haurà d'haver estat definida prèviament) que doni com a resultat el valor del terme a_n de la successió definida per la recurrència $a_{n+1} = \mathrm{f}(a_n)$ amb valor inicial $a_1 =$ `ini`.

Exercici 9.4

Utilitzeu la funció `recurr` que acabeu de definir per a investigar el límit de la successió definida per les condicions $1 < a_1$ i $a_{n+1} = 2 - \dfrac{1}{a_n}$ en funció del valor inicial a_1.

Exercici 9.5

Feu un gràfic per a la successió d'abans amb un dibuix del gràfic de $\mathrm{f}(x) = 2 - \dfrac{1}{x}$, el de $y = x$ i els punts de la forma $[a_1, a_2]$, $[a_2, a_2]$, $[a_2, a_3]$, $[a_3, a_3]$,..., $[a_i, a_{i+1}]$, $[a_{i+1}, a_{i+1}]$... units per línies (de fet, podeu fer una funció o un procediment que realitzi totes aquestes operacions en general).

9.2 Càlcul de límits amb la comanda `limit`.

La comanda de Maple que calcula el límit d'una expressió és `limit`. Si teniu una successió definida per $a_n = \mathrm{f}(n)$ podeu saber el seu límit posant `limit(f(n),n=infinity);` . Per exemple,

Exemple 9.1

```
> limit(n/(n+1),n=infinity);
> limit(n!/n^n,n=infinity);
> limit(2^(n^2)/n!,n=infinity);
> limit((1+1/n)^n,n=infinity);
> limit((-1)^n* sqrt(n)* sin(n^n)/(n+1),n=infinity);
> limit(cos(2*n*Pi),n=infinity);
```

Noteu que en aquest últim exemple Maple ha donat com a resultat el rang de valors `-1..1` i no ha reconegut que els cosinus d'angles que són múltiples enters de $2\,\pi$ sempre valen 1 i que, per tant, la successió $\{\cos(2\,n\,\pi)\}$ és constant amb valor 1. Això és conseqüència del fet que, si no li indiquem res, la variable `n` pren per defecte valors reals i, per tant, no podem assegurar que la successió sigui constant i igual a 1.

9.2.1 Declaració de tipus.

Maple permet declarar que una variable ha de ser d'un cert tipus. Per a declarar que la variable `n` només pren valors enters positius es pot utilitzar la comanda `assume` de la següent manera.

```
> assume(n,posint);
> limit(cos(2*n*Pi),n=infinity);
```

Si consulteu l'ajuda de la comanda `assume` i de la comanda `type` podreu veure diferents formes de restringir els valors d'una variable i quins són els diferents tipus possibles.

9.3 Exercicis

Exercici 9.6

Utilitzeu `limit` per a conèixer els límits de les successions:

- $x_n = \dfrac{1+2^\alpha+3^\alpha+\cdots+n^\alpha}{n^{(\alpha+1)}}$ per a diferents valors (positius) del paràmetre α.
- $x_n = \dfrac{1+\dfrac{1}{2}+\dfrac{1}{3}+\cdots+\dfrac{1}{n}}{\ln(n)}$.
- $x_n = \dfrac{1}{n+1}+\dfrac{1}{n+2}+\cdots+\dfrac{1}{2\,n}$.
- $x_n = \left(\dfrac{\ln(n+1)}{\ln(n)}\right)^{(n\ln(n))}$.
- $x_n = (\sqrt{n+1}-\sqrt{n}+1)^{\sqrt{n}}$.