Introducció a la simulació

Àngel Jorba
Josep Masdemnt Soler

Aquesta obra fou guardonada en el tercer concurs
“Ajut a l’elaboració de material docent” convocat per la UPC l’any 1993.

Primera edició: setembre de 1995

Aquesta publicació s’acull a la política de normalització lingüística
i ha comptat amb la col·laboració del Departament de Cultura i
de la Direcció General d’Universitats, de la Generalitat de Catalunya.

En col·laboració amb el Servei de Llengües i Terminologia de la UPC.

Disseny de la coberta: Antoni Gutiérrez

© Autor o els autors, 1995
© Edicions UPC, 1995
Edicions de la Universitat Politècnica de Catalunya, SL
Jordi Girona Salgado 31, 08034 Barcelona

Producció: Servei de Publicacions de la UPC i
CPDA (Centre de Publicacions d’Abast)
Av. Diagonal 647, ETSEIB, 08028 Barcelona

Dipòsit legal: B-27.022-95
ISBN: 84-7653-572-4

Són rigorosament prohibides, sense l'autorització escrita dels titulars del copyright, sota les sancions establertes a la llei, la reproducció total o parcial d'aquesta obra per qualsevol procediment, inclosos la reprografia i el tractament informàtic, i la distribució d'exemplars mitjançant lloguer o préstec públics.

Pròleg

Aquest llibre eś pensat per cobrir un buit que actualment existeix en el mercat respecte dels mètodes de simulació. Les tècniques de simulació tenen una història relativament recent, però alhora es basen en unes eines matemàtiques que provenen de molt més enrere.

Si ens proposem simular seriosament un cert procés o l'evolució d'una certa situació, necessitem tres ingredients bàsics fonamentals. El primer és la base teòrica que ens permetrà formular el problema que volem simular, en termes de models que s'ajustin a un estudi de rigorositat matemàtica, i que permet entendre el problema a grans trets des d'un punt de vista abstracte. Per a models reals, aquest ingredient potser ens deixarà veure aspectes qualitatius del nostre model però difícilment, llevat que el model sigui molt senzill, ens proporcionarà els resultats quantitatius que també esperem. Per això en cal l'ajut de les eines del càlcul numèric. Fent ús dels seus algorismes, fonamentats també en la base matemàtica, aconseguirem explorar el nostre model d'una manera més local i quantitativa. Finalment, el coneixement d'un llenguatge de programació ens permetrà portar a la pràctica les simulacions. I en tot cas, a partir d'aquí, realimentar el primer punt, sigui per descobrir nous aspectes qualitatius que seguidament quantificarem, sigui per corregir el model, en cas que els resultats obtinguts diferissin dels reals observats, per damunt d'una certa precisió demanada.

Això fa que els llibres actuals d'aquest tema tinguin un contingut matemàtic difícilment accessible a la persona que el que vol és senzillament simular i obtenir els possibles resultats d'un cert experiment o d'una situació a partir d'uns certs supòsits. A més, els continguts normalment es presenten en llibres específics que tracten cadascun dels tres punts anteriors d'una manera separada i, per tant, les eines buscades per damunt de tot es troben dins uns amplis mars matemàtics en el quals la persona no especialitzada necessita molt de temps per arribar al port correcte i aplicar el descobriment amb èxit.

Aquest llibre vol introduir el lector en el món de la simulació d'una manera autocontinguda i molt pràctica. L'hem pensat de manera que un estudiant dels primers cursos d'una llicenciatura o enginyeria pugui entendre bàsicament el seu contingut i aplicar-lo en un cas pràctic que li sigui proposat. Però és adreçat sobretot a estudiants de segon cicle o persones ja llicenciades que es trobin amb problemes de simulació que, malgrat haver fet uns cursos inicials de matemàtiques durant el primer cicle de la seva carrera, les poden tenir massa oblidades per entendre un llibre específic, però no tant per poder recordar i treure'n el màxim d'aprofitament a partir d'una lectura ràpida i selectiva.

Hem dividit el contingut en dues parts diferenciades. La primera d'elles, que abasta els capítols 1 i 2, fa referència a les tècniques de simulació discreta, entenent per aquestes les que fan referència als processos discrets que tenen un gran component aleatori. Exemples típics

podrien ser la utilització de la simulació per determinar el nombre de caixers actius que ha de tenir un supermercat al llarg del dia, problemes semblants en peatges d'autopistes, en finestretes de bancs o d'altres establiments, determinació del nombre apropiat de places de pàrquing en una certa estació o en un aeroport, nombre d'autobusos o metros que calen en una certa línia. En general podríem pensar en problemes de tipus social que porten associada una gran càrrega d'aleatorietat però malgrat això es modelen bé amb tècniques matemàtiques adequades.

Hem separat la part més teòrica en el capítol 1, en què es fa èmfasi en la generació de nombres pseudoaleatòris, en la importància de tenir bons generadors i en la manera de provar-los correctament, de la part més pràctica del capítol 2 on es troba la implementació i la metodologia de la gestió d'informació i de dades. En aquest darrer capítol es presenten també alguns exemples resolts.

La segona part del llibre abasta els capítols 3 i 4 i fa referència a les tècniques de simulació contínua, enteses com aquestes les que provenen de models governats per equacions diferencials. Tractem només el cas on el model és donat per equacions diferencials ordinàries i, essencialment, el problema de valor inicial que consisteix a saber quin estat tindrà el model en un temps donat sabent el seu estat en un temps determinat anterior. Són els models deterministes, dels quals estudis de dinàmica de poblacions, d'explotacions de recursos comercials, de propagació d'informació, o models derivats de problemes mecànics en són els exemples més ampliament coneguts.

Dels dos capítols que componen aquesta part, el primer introdueix el que és una equació diferencial ordinària, incidint en els aspectes qualitatius bàsics, aborda el problema del càlcul d'òrbites i arriba a donar un bon mètode d'integració numèrica (un Runge-Kutta 4-5 Fehlberg). El segon presenta una introducció molt senzilla al que és la teoria dels sistemes dinàmics, definint essencialment la fenomenologia qualitativa amb la qual ens podem trobar quan tenim models definits per equacions diferencials ordinàries, i acaba amb uns exemples típics, tant analítics com numèrics, d'aquest tipus de modelització.

Com que els exemples resolts s'acompanyen de programes en C, i molts capítols, a fi de facilitar la implementació dels algorismes numèrics, contenen funcions escrites en aquest llenguatge, el llibre acaba amb dos apèndixs relacionats amb la programció. El primer d'ells presenta el llenguatge C i és pensat no com un tractat extens, sinó per tal que una persona amb coneixements mínims de programació d'algun llenguatge pugui conèixer o recordar fàcilment les funcions d'entrada i sortida, incloses les que es fan per un arxiu, el govern de flux de programa, el manejament d'informació emmagetzemada en estructures, vectors o matrius. I en general, tot el que pugui necessitar per fer un programa de simulació, inclosos els problemes amb els quals es pot trobar derivats del tipus de variables emprades. El segon apèndix és dedicat a la sistematització de la part gràfica i de visualització, de la qual se'n fa un ús extens en els exemples numèrics del capítol 4.

Índex

Referències

[1] Bonet C. i d'altres: *Càlcul Numèric.* Aula teòrica 23, Edicions UPC, Barcelona (1994).

[2] Bratley P., Bennett L. F., Schrage L. E.: *A Guide to Simulation.* Springer-Verlag, Nova York (1987).

[3] Braun M.: *Differential Equations and Their Applications.* Springer Verlag, Nova York (1983).

[4] Braun M., Coleman C.S., Drew D.A. ed.: *Differential Equation Models. Modules in Applied Mathematics*, Vol 1, Springer Verlag, Nova York (1983).

[5] Collins W. J: *Intermediate Pascal Programming: a Case Study Approach.* McGraw-Hill, Nova York, pàg. 157 (1986).

[6] Devaney R.L.: *An Introduction to Chaotic Dynamical Systems.* The Benjamin/Cummings Publishing Company, Inc. California (1986).

[7] *Estudio de Apoyo al Segundo Plan de Carreteras: La Autopista A-7.* Advanced Logistic Group. Generalitat Valenciana, Conselleria d'Obres Públiques, Urbanisme i Transports (1993).

[8] Haberman R.: *Mathematical Models: Mechanical Vibrations, Population Dynamics and Trafic Flow.* Prentice Hall, Englewood Cliffs, Nova Jersey (1977).

[9] Hale J., Koçac H.: *Dynamics and Bifurcations.* Springer Verlag, Nova York (1991).

[10] Kernighan B.W, Ritchie D. M.: *The C Programming Language* (segona edició). Prentice Hall, Englewood Cliffs, Nova Jersey (1988).†

[11] Knuth D. E.: *Seminumerical Algorithms. The Art of Computer Programming*, vol. 2. Addison-Wesley, Reading (1981).

[12] Lewis P. A., Goodman A. S., Miller J. M.: "A Pseudo-Random Number Generator for the System/360". *IBM Syst. J.* 8 (2), pp. 136–146 (1969).

[13] Marsaglia G.: "Random Numbers fall mainly in the Planes". *Proc. Nat. Acad. Sci.* 61, pp. 25–28 (1961).

†Hi ha una versió castellana.

[14] Park S. K., Miller K.W.: "Random Number Generators: Good Ones are Hard to find". *Communications of the ACM*, vol. 31, pp. 1192–1201 (1988).

[15] Peña D.: *Estadística. Modelos y Métodos. Fundamentos*, vol. 1. Alianza Universidad Textos, Madrid (1989).

[16] Perko L.: *Differential Equations and Dynamical Systems.* Springer Verlag. Nova York (1991).

[17] Pidd M.: *Computer Simulation in Management Science* (tercera edició). John Wiley & Sons, Nova York (1992).

[18] Piskunov N.: *Cálculo Diferencial e Integral.* Montaner y Simón, Barcelona (1978).

[19] Press W. H., Teukolsky S. A., Vetterling W. T., Flannery B. P.: *Numerical Recipes in C* (segona edició). Cambridge Univ. Press, Cambridge (1992).

[20] Marcus-Roberts H., Thompson M. Ed.: *Life Science Models. Modules in Applied Mathematics*, vol. 4, Springer Verlag, Nova York, (1983).

[21] *SAS User's Guide: Basics, Version 5 Edition.* SAS Institute Inc., Cary, N. C., pp. 278-280 (1985).

[22] *TurboC v2.0.* Borland International Inc., California (1987).

[23] *System/360 Scientific Subroutine Package, Version III, Programmer's Manual.* IBM, White Plains, Nova York, pàg. 77 (1968).

[24] Zill D.G.: *Ecuaciones Diferenciales con Aplicaciones* (segona edició). Grupo editorial Iberoamericana, México DF (1988).

Índex alfabètic

Capítol 1 Generació de nombres aleatoris

1.1 Introducció: models i món real

La modelització de problemes del món real exigeix, a vegades, la descripció d'algunes variables que no tenim manera de conèixer. Per posar un exemple, suposem que volem simular un peatge d'autopista. Estem interessats a comparar el rendiment per diverses distribucions de cabines (automàtiques, manuals i "teletac"), amb l'objectiu d'obtenir una configuració eficient per a la propera operació retorn. Evidentment, no sabem (ni tenim manera de saber) de manera precisa com serà l'arribada de cotxes al peatge. Noteu que sense aquesta informació no podem, en principi, fer la simulació de manera fiable.

El que s'acostuma a fer en casos com aquest és suposar que la variable desconeguda és aleatòria, tot i que habitualment no sigui així. Tornant a l'exemple anterior, noteu que els cotxes no arriben al peatge a l'atzar: els seus ocupants han decidit l'hora de retorn en funció de conveniències personals i/o familiars, informació sobre l'estat del trànsit donada per la radio o la televisió, etc. Noteu, però, que l'arribada d'un gran nombre de vehicles al peatge pot fer que aquesta *sembli* aleatòria.

Per tant, en un primer pas, intentarem conèixer les principals propietats estadístiques de l'arribada de cotxes (pautes, tendències, etc). Per això cal fer una sèrie de suposicions, com per exemple que la propera operació retorn serà similar a les últimes. Llavors, s'estudia quina llei aleatòria (o quina combinació de lleis aleatòries) és mes semblant a la nostra arribada de cotxes. Aquest estudi es compon d'una primera part de recollida de dades, i d'una segona part on s'analitzen. En aquesta segona part, és important "capturar" les pautes i tendències més rellevants que presenten les dades, per seleccionar lleis aleatòries que les presentin. Aquesta part la considerem més pròpia de l'estadística que no pas de la simulació i, per tant, no l'hem inclòs en aquest llibre.

Suposem, per tant, que les lleis aleatòries que usarem per simular l'arribada (i les característiques) dels cotxes ja estan definides. El primer pas per fer la simulació d'aquest problema amb un ordinador consisteix a tenir una manera d'obtenir nombres aleatoris segons les lleis que hem determinat abans. És important generar els nombres aleatoris de manera adequada: si la nostra generació no presenta les mateixes pautes i biaixos que les lleis que hem preestablert, el resultat de la simulació pot ser del tot incorrecte. La generació de nombres aleatoris és, doncs, un dels punts més importants de tota simulació discreta.

Per altra banda, és impossible generar successions aleatòries en un ordinador: aquest és

una màquina determinista que no pot fer altra cosa que executar línia rere línia d'un conjunt d'instruccions (programa) preestablertes. Els seus resultats són, llavors, totalment deterministes.

En aquest capítol discutirem la generació de successions aleatòries en un ordinador. Veurem com es pot solventar l'objecció que hem fet en l'últim paràgraf, i veurem alguns mètodes concrets per fer-ho.

1.2 Successions aleatòries i pseudoaleatòries

El propòsit d'aquesta secció és explicar els conceptes de successió aleatòria i pseudoaleatòria. Per fer-ho, usarem un exemple. Suposem que efectuem tirades amb un dau i obtenim una seqüència de nombres compresos entre 1 i 6. Si el dau no té cap defecte, és clar que es verifiquen les propietats següents:

P1 Si el nombre de tirades és prou gran, el nombre de vegades en què apareix cada dígit és (aproximadament) 1/6 del total de tirades efectuades.

P2 Cada terme de la seqüència és independent dels altres.

Aquí observem les principals característiques d'una successió aleatòria. La propietat P1 fa referència a la llei que segueix la successió (en aquest cas, que tots els nombres tenen la mateixa probabilitat de sortir), i la propietat P2 indica que no hi ha cap relació entre els termes de la sèrie (és a dir, que el resultat d'una tirada no influeix en el resultat de les tirades posteriors).

Val la pena discutir una mica aquesta última propietat. Suposem que, per algun procediment (que no ve ara al cas), hem produït una succesió de nombres enters compresos entre 0 i 9 (ambdós inclosos), que satisfà la propietat P1. Suposem que aquesta sèrie de nombres verifica que després d'un nombre senar sempre en ve un de parell i viceversa (noteu que aquesta condició és perfectament compatible amb P1). És clar que aquesta successió no verifica P2, i que no és apta per a un gran nombre d'aplicacions (simular un sorteig, per exemple). Aquesta successió no és totalment impredictible, ja que després de sortir un 2, per exemple, sabem que no poden sortir els nombres 0, 2, 4, 6 i 8. Aquest és el motiu pel qual demanem que les successions verifiquin P2.[1]

Com hem comentat abans, la nostra intenció és generar successions aleatòries amb l'ordinador. Donat que això (per les causes que ja hem comentat) és impossible, ens conformarem amb una mica menys. Per exemple, si volem simular un dau, en tindrem prou de generar successions que verifiquin P1 i que, *aparentment*, verifiquin P2. Com veurem més endavant, ens serà impossible aconseguir que es verifiqui P2, però el que sí podrem aconseguir és que la relació entre termes consecutius de la sèrie sigui tan complexa, que sembli que no hi ha cap relació. Evidentment, caldrà quantificar aquesta correlació usant tècniques estadístiques, cosa que també comentarem més endavant.

Anomenarem, doncs, successions pseudoaleatòries aquelles successions que, tot i presentar relacions entre els diversos termes de la sèrie, són admisibles per a una aplicació donada. Aquesta definició depèn del nostre problema concret: una successió que pot ser apta per a una aplicació (per exemple, en un videojoc es requereix que el generador sigui ràpid, però no cal que sigui gaire bo) pot no ser-ho per a una altra (en alguns problemes de simulació cal que el

[1] Hom pot definir successions aleatòries que no verifiquin P2, però no les considerarem aquí, ja que no ens caldran per a cap aplicació.

generador sigui bo, però la seva rapidesa no és tan important perquè el càlcul no es fa en temps real).

Finalment, volem remarcar que no té sentit demanar-se si un nombre aïllat (que no forma part de cap successió) és aleatori o no. El concepte d'aleatorietat requereix que el nombre formi part d'una successió, i en aquest cas el que podem demanar-nos és si la sèrie és aleatòria. Per dir-ho amb un exemple: per saber si un dau "funciona bé" (és a dir, si compleix les propietats P1 i P2), no en tenim prou amb una tirada. Ens cal efectuar moltes tirades per observar la sèrie resultant, i decidir si aquesta sèrie és o no és aleatòria.

1.3 Generació de lleis uniformes

Per començar, considerem el problema de generar una llei uniforme a l'interval $[0,1]$. Per denotar aquest tipus de llei escriurem, d'ara en endavant, $U([0,1])$.

1.3.1 Condicions d'aleatorietat

Suposem que tenim un generador que ens produeix nombres reals amb una llei $U([0,1])$. Diem $\{x_n\}_n = \{x_1, x_2, \ldots\}$ a una successió obtinguda amb aquest generador. Llavors és clar que aquesta sèrie ha de verificar

C1 Condició 1: Els números x_1, x_2, x_3, etc. estan uniformement distribuïts a l'interval $[0,1]$.

C2 Condició 2: Les parelles (x_1,x_2), (x_2,x_3), (x_3,x_4), etc. estan uniformement distribuïdes al quadrat unitat $[0,1]^2 = [0,1]\times[0,1]$.

C3 Condició 3: Les ternes (x_1,x_2,x_3), (x_2,x_3,x_4), etc. estan uniformement distribuïdes al cub unitat $[0,1]^3$.

$\vdots$

CN Condició N: Les tuples $(x_1,x_2,\ldots,x_N)$, $(x_2,\ldots,x_{N+1})$, etc. estan uniformement distribuïdes a $[0,1]^N$.

$\vdots$

Comentem una mica el significat d'aquestes condicions. La condició 1 vol dir que els termes de la successió "recobreixen" de manera uniforme tot l'interval $[0,1]$. Noteu que aquesta condició és, essencialment, la condició P1 de la pàgina 14.

La condició 2 demana que les parelles (x_1,x_2), (x_2,x_3), etc. vistes com a punts de $\mathbb{R}^2$, "recobreixin" de manera uniforme tot el quadrat unitat $[0,1]\times[0,1]$. Observeu que el fet que es compleixi C1 no implica que es compleixi C2. Veiem-ho amb un exemple. Siguin $\{a_n\}_n$ i $\{b_n\}_n$ successions $U([0,1/2])$ i $U([1/2,1])$ respectivament. Definim una nova successió $\{c_n\}_n$ de la manera següent: $c_1 = a_1$, $c_2 = b_1$, $c_3 = a_2$, $c_4 = b_2$, $c_5 = a_3$, etc. És a dir,

$$c_n = \begin{cases} a_{(n+1)/2} & \text{si } n \text{ és senar} \\ b_{n/2} & \text{si } n \text{ és parell} \end{cases}$$

És facil veure que aquesta successió verifica C1. Veiem que no verifica C2: les parelles (c_1, c_2), (c_3, c_4), etc. són, en aquest cas, (a_1, b_1), (b_1, a_2), (a_2, b_2), etc. Aquests punts recobreixen només el conjunt $[0, 1/2] \times [1/2, 1] \cup [1/2, 1] \times [0, 1/2]$, en lloc del $[0, 1] \times [0, 1]$ que caldria per complir C2. Per altra banda, la correlació entre termes consecutius de la sèrie és òbvia: després d'un nombre més gran que 1/2 en ve sempre un de més petit que 1/2, i viceversa. Per tant, podem dir que C2 "detecta" (algunes) correlacions entre termes consecutius de la sèrie.

La discussió del paràgraf anterior s'estén fàcilment a les altres condicions C3, C4, etc. En el cas de C3, per exemple, el que estem demanant és l'absència de correlacions entre tres termes consecutius de la successió. Igual que abans, que una successió verifiqui les condicions C1, C2, ..., CN no implica que verifiqui la $N + 1$.

Habitualment s'acostuma a definir successió pseudoaleatòria aquella que verifica les condicions C1, C2, ..., CN per a un N prou gran per a l'aplicació en què estem treballant.

Per al lector interessat a aprofundir en el tema de les successions aleatòries i les seves propietats recomanem la consulta de la referència [11].

1.3.2 Generadors de congruència lineal

Començarem per la generació de nombres enters pseudoaleatoris. Per iniciar la discussió, considerem el mètode basat en la fórmula següent:

$$X_i = (a X_{i-1} + c) \bmod m, \tag{1.1}$$

on mod m denota l'operació "fer mòdul m" (dividir per m i quedar-se amb la resta), i m, a i c són enters positius tals que $m > \max\{a, c\}$. El mètode funciona de la manera següent: triem un enter X_0 tal que $0 \leq X_0 < m$. Apliquem la fórmula (1.1) per obtenir un valor X_1. Apliquem de nou aquesta fórmula per obtenir un nou X_2 a partir del X_1 anterior, i així successivament. Amb això, hem generat una successió de nombres enters compresos entre 0 i $m - 1$, ambdós inclosos. Veiem-ne un exemple: considerem el generador definit per la fórmula

$$X_i = (7X_{i-1} + 1) \bmod 10. \tag{1.2}$$

Si prenem $X_0 = 8$, per exemple, obtenim $X_1 = 7$, $X_2 = 0$, $X_3 = 1$, etc. Noteu que per a cada X_0 obtenim una successió diferent.

DEFINICIÓ: Anomenarem *generador de congruència lineal* la fórmula (1.1). Anomenarem *llavor* el valor X_0 utilitzat per inicialitzar la successió definida per un d'aquests generadors.

Considerem ara la possibilitat d'utilitzar fórmules com la (1.1) per produir nombres enters aleatoris.[2] Per exemple, podria servir la successió generada usant (1.2) com a successió aleatòria de nombres enters entre 0 i 9? Si calculem uns quants termes més de la sèrie, obtenim:

$$8, 7, 0, 1, 8, 7, 0, 1, 8, 7, 0, 1, 8, \ldots$$

Evidentment, aquesta seqüència no és una bona successió de nombres aleatoris. La primera observació a fer és que aquesta sèrie és periòdica, i la segona que no produeix tots els valors possibles entre 0 i 9. Passem ara a discutir de quina manera es poden resoldre aquests problemes.

[2] Com que totes les successions que apareixeran seran pseudoaleatòries, en el que segueix abusarem del llenguatge i les anomenarem aleatòries. Aquest abús és habitual en llibres, manuals, etc.

Període d'un generador

Considerem ara un generador qualsevol, com el donat per (1.1). Veiem que, a causa del fet que els valors possibles del generador són finits (entre 0 i $m-1$), en algun moment s'haurá de repetir algun terme de la successió, i que quan es repeteix un terme de la successió tota ella es repeteix. Això vol dir que les successions obtingudes a partir de fórmules com la (1.1) sempre són periòdiques.

DEFINICIÓ: Anomenarem *període* de la successió el nombre de termes que hi ha entre dos termes iguals de la sèrie.

En l'exemple (1.2) el període era 4. El període d'un generador depèn, en general, de la llavor escollida.

El període màxim que pot tenir un generador com (1.1) és m. Per altra banda, si un generador com (1.1) té període m, llavors la successió corresponent pren tots els valors possibles entre 0 i $m-1$, ambdós inclosos.

Aquestes observacions ens portaran a buscar generadors amb m gran i amb període màxim.

Ratxes

Suposem ara que estem llançant una moneda, i que apuntem el resultat obtingut en forma de tira de caràcters (per exemple, `c` quan surt "cara" i `x` quan surt "creu"). Amb això obtenim una seqüència com `cxccxxxcx`...

DEFINICIÓ: Una *ratxa* és la repetició d'un valor en una successió aleatòria.

L'existència de ratxes és una característica d'aquest tipus de successió aleatòria. La raó és molt senzilla: després d'una `c`, la probabilitat d'obtenir una altra `c` és 1/2, i això implica que, aproximadament, la meitat de les `c` obtingudes han d'anar seguides d'una altra `c`. Igualment, com que la probabilitat d'obtenir una tercera `c` després d'haver-ne obtingut ja dues és 1/2 podem afirmar que, aproximadament, la quarta part de les `c` de la sèrie van seguides de dues `c` més, i així successivament.

El nombre de ratxes no pot ser arbitrari, sinó que el seu nombre el governen les lleis de la successió que estem generant. De fet, un possible test per estudiar el caràcter aleatori d'una successsió és comptar les ratxes que presenta i veure si el nombre s'ajusta al valor teòric esperat. Si això no passa, vol dir que hi ha alguna correlació entre termes consecutius de la sèrie que fa que aquesta no es pugui considerada com aleatòria.

Finalment, remarquem que les successions que produeixen els generadors de congruència lineal o bé no presenten cap ratxa, o bé en tenen una de longitud infinita: si per a algun n passa que $X_{n-1} = X_n$, llavors forçosament ha de passar que $X_{n+1} = X_n$, $X_{n+2} = X_{n+1}$, etc. Evidentment, aquest és un inconvenient més que tenen els generadors de congruència lineal, i més endavant veurem com podem resoldre'l.

Generadors de congruència multiplicativa

Un cas particular dels generadors de congruència lineal es dóna quan la c de la fórmula (1.1) és zero.

DEFINICIÓ: Anomenarem *generador de congruència multiplicativa* la fórmula

$$X_i = (aX_{i-1}) \bmod m. \tag{1.3}$$

Noteu que en aquests generadors no es pot usar la llavor $X_0 = 0$, perquè produeix una successió estacionària $X_n = 0$. Per altra banda, si comencem per un X_0 diferent de zero, ens interessa que mai aparegui aquest valor (ja que a partir d'aquest moment la sèrie s'estacionarà a 0). Una condició suficient per a això és que m (vegeu (1.3)) sigui un nombre primer.[3]

També volem fer notar que el període màxim d'aquest tipus de generadors és $m-1$, donat que el zero està exclòs del conjunt de valors admissibles per a X_n.

1.3.3 Generació de lleis $U([0,1])$

Suposem que, per algun mitjà, hem obtingut una successió X_n de nombres enters aleatoris compresos entre 0 i $m-1$, amb distribució uniforme.[4] Llavors, la successió $\{\alpha_n\}_n$ definida per

$$\alpha_n = \frac{X_n}{m}, \quad n = 0, 1, 2, \ldots, \tag{1.4}$$

és una successió dins l'interval $[0,1]$. Mirem ara si aquesta successió segueix una llei $U([0,1])$. Noteu que els termes α_n no poden prendre qualsevol valor entre 0 i 1. Només poden tenir valors dins del conjunt

$$F_m = \left\{0, \frac{1}{m}, \frac{2}{m}, \ldots, \frac{m-1}{m}\right\}.$$

Per tant, estrictament parlant, no podem dir que les $\{\alpha_n\}_n$ segueixen una llei $U([0,1])$.

Suposem ara que estem treballant (amb un ordinador) amb variables reals de, per exemple, 7 decimals. Si el valor de m és 10^7 (o més gran) llavors el conjunt F_m és, dins aquest ordinador, el mateix que el conjunt $[0,1]$. Com que en les nostres aplicacions els nombres aleatoris que obtindrem estaran representats per nombres decimals amb una quantitat finita de decimals (habitualment 6 o 7), considerarem la successió $\{\alpha_n\}_n$ com una $U([0,1])$.

Noteu que el valor "1" no és mai generat per aquesta successió. En general això no és cap problema, perquè

1. El valor "1" té probabilitat 0 en una $U([0,1])$.
2. Podem generar nombres tan propers a 1 com vulguem (dins dels, per exemple, 7 decimals de l'ordinador).

Si per alguna raó volem que aparegui el valor "1", es pot reemplaçar la definició anterior de α_n per

$$\alpha_n = \frac{X_n}{m-1}.$$

Algunes vegades ens pot interessar que α_n no prengui el valor 0 (més endavant veurem casos en què cal treure el logaritme d'una $U([0,1])$ i, naturalment, voldrem que aquesta llei no prengui el valor "0" per evitar el corresponent error en el programa). Això es pot aconseguir fàcilment definint

$$\alpha_n = \frac{X_n + 1}{m}.$$

Ara, α_n no pot ser "0", però si que pot ser "1".

[3] El producte aX_{i-1} no pot ser mai m, ja que tant a com X_{i-1} són menors que m, i m és primer. Aquest producte tampoc pot ser cap múltiple de m per la mateixa raó.

[4] Això implica que X_n pot prendre qualsevol valor entre 0 i $m-1$.

En les seccions següents considerarem el cas en què la successió X_n està generada per un mètode de congruència lineal o multiplicativa. A causa de les observacions anteriors, cal triar el valor m (de la fórmula (1.1) o de la (1.3)) prou gran per treballar amb variables de 7 decimals.[5]

Per acabar, volem dir (sense demostració) que *es poden aconseguir generadors de congruència multiplicativa, de la mateixa qualitat que els generadors de congruència lineal*, si es trien a i m de manera adequada. Com els generadors de congruència multiplicativa funcionen una mica més de pressa que els de congruència lineal (ens estalviem una suma), ens centrarem únicament en aquest tipus de generador.

1.3.4 Inconvenients de les $U([0,1])$ obtingudes usant generadors de congruència multiplicativa

El propòsit d'aquesta secció és discutir com verifiquen els generadors de congruència multiplicativa les condicions C1, ...,CN.

Donada la dificultat de verificar aquestes condicions de manera teòrica, habitualment es fa amb l'ajut de l'ordinador (més endavant en veurem els detalls, a la secció 1.4).

La condició C1 és fàcil de verificar. Abans ja hem comentat que cal tenir m prou gran (en relació al nombre de decimals amb què estem treballant) i període màxim. De fet, es poden escollir valors per a m i a (a la fórmula (1.3)) de manera que la condició C1 es compleixi de manera satisfactòria. A la secció 1.3.5 en veurem alguns exemples.

Les condicions C2, C3, etc. són més complicades de verificar, i la manera concreta de fer-ho la discutirem a la secció 1.4. Aquí només direm que no es verifica de manera completament satisfactòria. La raó és en el següent resultat:

Propietat: Sigui $\{\alpha_n\}_n$ una successió obtinguda per mitjà de (1.4), on X_n s'obté usant un generador de congruència multiplicativa. Definim la successió de punts p_n de $[0,1]^d$ de la manera següent:

$$p_n = (\alpha_n, \alpha_{n+1}, \ldots, \alpha_{n+d-1}), \quad n = 0, 1, 2, 3, \ldots$$

Llavors, els punts p_n estan situats sobre plans. El nombre d'aquests plans és, com a molt, $(d\,!\,m)^{1/d}$, on m és el de la fórmula (1.3).

Demostració: Vegeu [13].

A la taula 1.1 s'ha tabulat l'expressió $(d\,!\,m)^{1/d}$ per a diversos valors de d i m. Noteu la dràstica reducció del nombre de plans en créixer d. Això és una limitació important d'aquest tipus de successions.

S'ha usat el valor $m = 2^{31}$ perquè els generadors que usarem més endavant tenen valors de m similars a aquest. El valor $m = 2^{16}$ l'hem inclòs perquè hi ha molts generadors comercials que usen aquest valor.

Volem recalcar que els valors mostrats en la taula 1.1 són una cota superior del nombre de plans. *Si a i/o m no són adequadament escollits, aquest nombre pot ser molt més petit.* Insistim, doncs, en la necessitat de triar a i m de manera acurada. En la secció 1.3.5 veurem algunes possibilitats per escollir aquests nombres.

Per acabar, diem que aquest defecte dels generadors de congruència multiplicativa[6] és intrínsec al mètode. Més endavant (en la secció 1.3.8) veurem un procediment que permet resoldre aquest problema.

[5] En el cas de programar en C, ens referim a variables tipus `float`. Veure l'apèndix A per a més detalls.

[6] Els generadors de congruència lineal presenten les mateixes dificultats.

$m = 2^{31}$		$m = 2^{16}$	
d	$[(d\,!\,m)^{1/d}]$	d	$[(d\,!\,m)^{1/d}]$
1	2147483648	1	65536
2	65536	2	362
3	2344	3	73
4	476	4	35
5	191	5	23
6	107	6	19
7	72	7	16
8	55	8	15

Taula 1.1: *Nombre màxim d'hiperplans.* $[\cdot]$ *denota part entera.*

Hi ha un altre defecte d'aquests generadors que val la pena comentar: si agafem els ultims dígits de cada terme de la successió $\{X_n\}_n$, veiem que aquests són molt menys aleatoris que els primers. És un fet que cal tenir present quan volem generar nombres enters dins d'un rang curt. Veiem-ne un exemple: suposem que volem produir una sèrie de nombres entre 0 i 9 (ambdós inclosos), i disposem d'un generador de congruència lineal (o multiplicativa) amb un m gran (per exemple, de l'ordre de 10^6). Llavors, per produir nombres entre 0 i 9 tenim, en principi, dues possibilitats: Podem fer

$$d_n = X_n \bmod 10,$$

que correspon a agafar l'últim dígit de X_n, o bé

$$d_n = [10\alpha_n] = \left[10\left(\frac{X_n}{m}\right)\right],$$

on, com sempre, $[\cdot]$ indica la part entera. Aquest segon cas fabrica el dígit d_n a partir dels primers dígits de α_n o de X_n. Ara és clar que és molt millor la segona opció que la primera.

1.3.5 Exemples concrets

Veiem ara uns exemples concrets de generadors de congruència multiplicativa. Un exemple que podem considerar clàssic és el següent:

$$a = 7^5 = 16807, \quad m = 2^{31} - 1 = 2147483647. \tag{1.5}$$

Aquest generador té període màxim ($2^{31} - 1$ és un nombre primer). És proposa per primera vegada a [12], i s'estudia en diversos llocs (vegeu [14]). Alguns autors (per exemple, [14] i [19]) l'anomenen l'*estàndard mínim*, en el sentit que és el generador més senzill amb la qualitat suficient per ser usat en molts exemples.

Hi ha altres possibilitats. Per exemple podem citar $a = 48271$ i $a = 69621$, ambdues amb $m = 2^{31} - 1$.

1.3.6 Implementació en C

Anem ara a veure de quina manera podem implementar un d'aquests generadors en un llenguatge d'alt nivell com C.

Per començar, considerem el generador (1.5):

$$X_i = 16807 X_{i-1} \bmod 2147483647. \tag{1.6}$$

Recordem (vegeu secció A.2.2) que, en C, les variables enteres més grans que podem tenir son les `long int`. Aquestes variables poden emmagatzemar nombres enters compresos entre -2^{31} i $2^{31}-1$. Com que el període de (1.6) és el màxim, X_{i-1} pot prendre el valor $2147483646 = 2^{31}-2$, i per tant podem emmagatzemar tots els termes de la sèrie en variables de tipus `long int`. Noteu, però, que el producte $16807 X_{i-1}$ pot ser més gran que $2^{31}-1$, i això desborda la capacitat d'un `long int`. Per poder fer aquest producte ens cal, en principi, disposar de variables enteres més grans. Per resoldre aquest problema, donarem el resultat següent.

Proposició: Sigui

$$q = \left[\frac{m}{a}\right], \quad r = m \bmod a,$$

és a dir, $m = aq + r$. Sigui $x \in \{1, 2, \ldots, m-1\}$. Llavors, si $r < q$,

1. Els valors $a(x \bmod q)$ i $r[x/q]$ estan compresos entre 0 i $m-1$ (ambdós inclosos).
2. Definim $u = a(x \bmod q) - r[x/q]$. Llavors,

$$ax \bmod m = \begin{cases} u & \text{si } u \geq 0, \\ u + m & \text{si } u < 0. \end{cases}$$

Demostració: Vegeu [2].

Aquesta proposició permet implementar el càlcul del producte aX_{i-1}, sense tenir el problema del desbordament de nombres enters: el càlcul de $a(x \bmod q)$ i $r[x/q]$ pot fer-se sense cap problema (estan compresos entre 0 i $m-1$), i llavors el valor u ha d'estar comprès entre $-(m-1)$ i $m-1$. Per tant, tenint en compte que $m = 2^{31}-1$ i que els `long int` poden guardar nombres compresos entre -2^{31} i $2^{31}-1$, podem assegurar que tot el procés pot fer-se sense cap problema.

Per aplicar aquest mètode a (1.6), agafem $q = 127773$ i $r = 2836$. Una possible implementació (que anomenarem `alea0`) és la següent:

```
#include <stdio.h>
#include <stdlib.h>

#define A       16807L
#define M  2147483647L
#define Q      127773L
#define R        2836L

float alea0(long int *ap_llavor)
{
   if (*ap_llavor <= 0) {puts("alea0: llavor <= 0"); exit(1);}
   *ap_llavor=A*(*ap_llavor%Q)-R*(*ap_llavor/Q);
   if (*ap_llavor<0) *ap_llavor += M;
   return(*ap_llavor/((float)M));
```

```
}

#undef R
#undef Q
#undef M
#undef A
```

Veiem com funciona aquesta rutina. Per cridar-la per primer cop cal donar un valor inicial a la llavor (correspon al X_0 de la fórmula (1.6)) que, com ja hem vist abans, no pot ser ni zero ni negatiu. Llavors cal cridar a `alea0` passant la llavor per adreça, i la mateixa rutina `alea0` ens actualitzarà la llavor i ens tornarà el nombre aleatori.

Veiem-ne un exemple: el programa següent calcula 10 nombres aleatoris i els escriu a la pantalla.

```
#include <stdio.h>

void main(void)
{
   float alea0(long int *ap_llavor);
   float x;
   long int la;
   int j;
   la=1995;
   for (j=0; j<10; j++)
   {
      x=alea0(&la);
      printf("%f\n",x);
   }
}
```

El valor `1995` usat per inicialitzar la llavor és un número qualsevol, donat que tots ells produeixen resultats de qualitat similar. Observeu també que, la llavor, només cal inicialitzar-la un cop al principi del programa.

La resta del programa hauria de quedar clar: es va cridant a `alea0` (passant-li la llavor per adreça), i ell ja s'encarrega d'actualitzar la llavor i de retornar el corresponent nombre aleatori. Per tant, l'usuari no ha d'alterar el valor de la llavor durant tota l'execució del programa (tret de la inicialització, naturalment).

Noteu també que si, per alguna raó, a mig programa es vol tornar a generar un altre cop la mateixa sèrie de números, només cal reinicialitzar la llavor al valor que tenia al principi.

Per implementar els altres generadors mencionats en la secció anterior només cal canviar les variables dels `#define` que hi ha al principi del programa, per als valors següents:

$a = 48271,$	$m = 2147483647,$	$q = 44488,$	$r = 3399,$
$a = 69621,$	$m = 2147483647,$	$q = 30845,$	$r = 23902.$

Cal dir que els generadors obtinguts amb aquestes constants tenen un comportament similar a `alea0`. Es recomana no usar valors de a i m diferents dels donats aquí.

1.3.7 Alguns exemples de generadors dolents

Lamentablement, no és difícil trobar generadors dolents en paquets comercials i fins i tot en alguns llibres de text. La raó sembla que es deu al fet que molt poca gent es preocupa d'aquests temes, creient (equivocadament) que és fàcil produir nombres aleatoris.

Un exemple d'això és el generador anomeant `RANDU`:

$$X_i = (65539 X_{i-1}) \bmod 2^{31}.$$

Aquest generador va aparèixer per primer cop a [23], i s'ha usat en diversos paquets de *software* i llibres de text. Els seus principals defectes són que no té període màxim, i que no satisfà la condició C3 (secció 1.3.1). Una variant d'aquest generador és la definida per

$$X_i = (16807 X_{i-1}) \bmod 2^{31},$$

que es pot trobar en diversos llocs, com ara [17]. Té els mateixos defectes que el `RANDU`. El paquet comercial SAS en la seva cinquena versió (vegeu [21]) incorpora aquest generador amb una barreja addicional (vegeu secció 1.3.8) per millorar el seu *output*.

Un altre exemple curiós de generador dolent és el següent:

$$X_i = (9806 X_{i-1} + 1) \bmod 131071.$$

Aquest generador es pot trobar a [5]. Es pot verificar que si agafem la llavor $X_0 = 37911$ obtenim $X_1 = 37911$, i per tant la successió així generada és constant. Cal dir, però, que, si evitem aquesta llavor, el generador sembla funcionar correctament.

Per al lector interessat a col·leccionar generadors dolents recomanem la consulta de [14], on trobarà una llarga llista de generadors amb problemes, com també dels llocs on aquests generadors estan usats o recomanats.

1.3.8 Mètode de la barreja

Fins ara hem vist mètodes basats en congruències lineals, i hem discutit les seves qualitats i defectes. A continuació veurem una tècnica que permet millorar qualsevol generador. Aquesta tècnica l'aplicarem al cas concret de `alea0`.

El mètode consisteix a "barrejar" la successió aleatòria, alterant l'ordre en què apareixen els diferents nombres. Donem, en primer lloc, l'algorisme i després el discutirem amb detall.

Usarem la notació següent: X_n serà la successió de nombres enters que volem barrejar. Suposarem que el màxim que pot assolir X_n és $m-1$ (si X_n s'ha obtingut amb un mètode de congruència lineal o multiplicativa, llavors m és el valor amb el qual fem mòdul). Anomenarem T un vector de k components, tal que la primera component és T_0 i l'última T_{k-1}. El valor concret de k el deixem per a més endavant. Anomenarem P una variable (entera) auxiliar. Suposarem que T_0, T_1, ..., T_{k-1} i P han estat inicialitzats amb els valors X_0, X_1, ..., X_{k-1} i X_k respectivament. Aquesta inicialització es farà només un cop al principi del programa. Ara, l'algorisme és

Pas 1. Sigui $j = [k(P/m)]$.

Pas 2. Sigui $P = T_j$.

Pas 3. Omplim T_j amb el següent terme de la successió X_n.

Pas 4. El resultat és P.

És a dir, un cop feta l'etapa d'inicialització, cada "execució" dels 4 passos de l'algorisme produirà un nou terme de la sèrie barrejada.

Comentem ara aquest algorisme. En el pas 1 seleccionem una de les components (que anomenen j) del vector T, usant els primers dígits del valor P. En el pas 2 actualitzem el valor de P amb T_j. Aquest nou valor de P serà la sortida del generador (pas 4). Per acabar, s'actualitza T_j amb el següent terme de la successió X_n.

El mètode de la barreja es pot implementar usant qualsevol generador. Com que aquest mètode permet millorar el generador amb molt poc esforç, és molt recomanable utilitzar-lo.

Finalment volem comentar que hi ha altres variants del mètode de la barreja que no comentarem aquí. Es recomana al lector interessat la consulta de [11].

Implementació del mètode de la barreja

A continuació veurem una modificació de la funció `alea0` que incorpora barreja. La funció, que hem anomenat `alea1`, és la següent:

```
#include <stdio.h>
#include <stdlib.h>

#define A       16807L
#define M  2147483647L
#define Q      127773L
#define R        2836L
#define K          32

float alea1(long int *ap_llavor)
{
   static long int t[K],p;
   static int ini=0;
   int i,j;
   if (ini == 0)
      {
         if (*ap_llavor > 0) {puts("alea1: llavor > 0"); exit(1);}
         ini=1;
      }
   if (*ap_llavor <= 0)
      {
         if (*ap_llavor == 0) {puts("alea1: llavor == 0"); exit(1);}
         *ap_llavor=-(*ap_llavor);
         for (i=0; i<K; i++)
         {
            *ap_llavor=A*(*ap_llavor%Q)-R*(*ap_llavor/Q);
            if (*ap_llavor<0) *ap_llavor += M;
            t[i]=*ap_llavor;
         }
```

```
        *ap_llavor=A*(*ap_llavor%Q)-R*(*ap_llavor/Q);
        if (*ap_llavor<0) *ap_llavor += M;
        p=*ap_llavor;
      }
   j=(K*(p/((float)M)));
   p=t[j];
   *ap_llavor=A*(*ap_llavor%Q)-R*(*ap_llavor/Q);
   if (*ap_llavor<0) *ap_llavor += M;
   t[j]=*ap_llavor;
   return(p/((float)M));
}

#undef R
#undef Q
#undef M
#undef A
#undef K
```

L'ús d'aquesta funció és molt similar al de la funció `alea0`, amb la diferència que *la llavor ha de ser negativa*. Això és així perquè el primer cop que es crida `alea0` cal fer les corresponents inicialitzacions. Per tant, quan la llavor sigui negativa, la funció canviarà el signe de la llavor i s'inicialitzarà de manera adequada. En les crides següents, com que la llavor serà positiva, ja no es farà aquesta etapa d'inicialització. La successió aleatòria que estem barrejant és la de la funció `alea0`.

El funcionament de la funció és senzill: en primer lloc, verifiquem si el primer cop que es crida la rutina es fa amb una llavor negativa (recordeu que al principi cal inicialitzar la taula T i el valor P). Si hom està segur de cridar correctament aquesta funció (és a dir, d'usar sempre llavors negatives), llavors pot eliminar aquestes línies (són la declaració de `ini` i també tot el `if (ini == 0)`...). Nosaltres, però, recomanem de mantenir aquestes línies.[7]

El següent `if` és per inicialitzar la funció, en cas que la llavor sigui negativa. La resta de línies són la traducció directa a C de l'algorisme, i no haurien de presentar cap problema per al lector.

Avantatges i inconvenients dels mètodes de barreja

Genèricament, els mètodes de barreja no empitjoren la qualitat de la successió que estem generant, i per tant són sempre recomanables. La seva principal virtut és reduir la correlació existent entre termes consecutius de la successió. En aquest sentit, podem dir que si apliquem el mètode de la barreja a un generador de congruència lineal o multiplicativa, la sèrie resultant verifica molt millor les condicions C2, C3, etc. És a dir, en barrejar "trenquem" l'estructura de plans que presenta la successió de punts donats per termes consecutius de la successió original.

Els inconvenients són molt petits: el generador amb barreja és una mica més lent i usa una mica més de memòria. Per tant, la conclusió és que sempre és recomanable aplicar un métode de barreja a un generador de congruència lineal.

[7] És habitual en programes de simulació el provar diversos generadors de nombres aleatoris (és una manera d'assegurar que els resultats no depenen del generador emprat). Com que hi ha generadors que usen llavors positives (com `alea0`) i generadors que n'usen de negatives (`alea1`), és molt fàcil equivocar-se de llavor.

1.3.9 Altres mètodes

Els mètodes que hem vist aquí són els més bàsics dins de la generació de nombres aleatoris. Cal dir que la majoria de mètodes sofisticats es basen en els generadors de congruéncia lineal.

Una tècnica comuna és combinar diversos generadors de congruència lineal per fabricar les diverses parts del nombre. Per exemple, podem usar un generador per obtenir els primers decimals del nombre aleatori (entre 0 i 1) i un altre per obtenir-ne els últims. La idea és compensar el fet (comentat anteriorment) que els últims dígits són menys aleatoris que els primers.

També hi ha altres versions del mètode de la barreja. Nosaltres aquí hem utilitzat la mateixa successió per obtenir els nombres que per barrejar. És possible utilitzar una successió per obtenir els nombres i una altra per barrejar-los. En aquest cas, si s'escullen les dues successions de manera adequada, el període del generador resultant és molt més gran: en molts casos, és el mínim comú múltiple dels períodes dels dos generadors. Cal dir, però, que si les dues successions no estan triades de manera adequada, la sèrie resultant de la barreja pot tenir un comportament molt menys aleatori que la sèrie que estem barrejant (vegeu [11] per a més detalls). Cal dir també que aquests problemes no apareixen quan barrejem una successió usant la pròpia successió per fer-ho (que és el que hem fet a la rutina `alea1`).

Naturalment, és possible combinar les dues possibilitats descrites i utilitzar tres successions: dues per fabricar els nombres i una per barrejar-los.

En tot cas, recomanen al lector interessat la consulta de [11], on trobarà, de manera detallada, tots aquests mètodes (i d'altres) juntament amb una discussió de les seves propietats més importants. A [19] es poden trobar les implementacions en C d'alguns d'aquests algorismes.

1.4 Tests d'aleatorietat

En aquesta secció ens proposem fer una petita discussió dels tests més bàsics que hi ha per detectar si la successió que generem pot ser considerada, o no, la realització d'una successió de variables aleatòries independents amb llei $U([0,1])$.

La manera de construir un test és buscar alguna propietat que una llei $U([0,1])$ hagi de complir, i mirar si la nostra sèrie la compleix o no. Arribats en aquest punt, cal fer notar el següent: qualsevol propietat d'una successió aleatòria es defineix a partir de tota la successió (és a dir, dels seus infinits termes). Com que nosaltres només disposarem d'un nombre finit de termes a la successió, mai podrem estar completament segurs de si la propietat es verifica o no. Per tant, el resultat del test sempre s'haurà d'entendre com una probabilitat que la succesió sigui aleatória o que no ho sigui.

Podem il·lustrar aquest fet amb un exemple. Suposem que, utilitzant un generador (que no sabem si és aleatòri o no), simulem 100 tirades d'una moneda de manera que la cara i la creu tinguin la mateixa probabilitat.[8] Imaginem que hem obtingut 100 cares. Podem dir que el generador és dolent? En principi, sembla que sí. Analitzem-ho amb més detall. Si estiguèssim utilitzant nombres aleatoris de debò, la probabilitat d'obtenir 100 cares consecutives és $(1/2)^{100} \approx 7.89 \times 10^{-31}$. Aquesta probabilitat és diferent de zero, i per tant és possible obtenir la tira de 100 cares, tot i que la seva probabilitat és molt baixa. Un pot estar temptat de dir que el generador no és bo perquè ha produït una tirada de probabilitat excessivament petita. Noteu que aquest argument no és vàlid, ja que *totes les tires de 100 cares i creus tenen probabilitat* $(1/2)^{100}$ *d'aparèixer*. Per tant, quin criteri pot decidir si la tira de 100 cares és o no aleatòria?

[8] Per exemple, si el generador ens dóna un nombre més gran que 1/2 agafem cara, altrament agafem creu.

Per contestar aquesta pregunta, enfocarem el problema d'una altra manera: busquem alguna propietat que tota successió aleatoria (uniforme) de cares i creus hagi de complir, i mirem si la nostra successió la compleix o no. Una primera propietat molt senzilla és que, quan la longitud de la successió tendeix a infinit, el nombre de cares i de creus ha de tendir a ser el mateix. Per tant, un criteri general pot ser comptar la freqüència de les cares i, en funció del seu valor, decidir si considerem la successió aleatòria o no.

Abans de continuar, volem remarcar que els tests estadístics ens serviran per identificar (amb un cert marge d'error) sèries no aleatòries, però mai podrem usar-los per afirmar que una determinada successió és aleatòria. La raó és òbvia: del fet que es verifiqui una certa propietat no se'n pot deduir la aleatorietat de la successió. Usant l'exemple anterior, que el nombre de cares i creus sigui aproximadament el mateix no implica en absolut que la sèrie sigui aleatòria. Per exemple, un generador que produeixi cares i creus alternativament verifica aquesta propietat, però no és aleatori ja que és totalment predictible.

En les seccions següents veurem de manera esquemàtica els tests estadístics més bàsics. El lector interessat en més informació pot consultar [15] o [11].

1.4.1 Test χ^2

Aquest és potser el test més conegut. S'aplica a successions discretes (cada terme només pot agafar valors dins d'un conjunt finit de valors admissibles), com per exemple les tirades d'una moneda.

Es comença suposant que la successió segueix una determinada llei o, el que és el mateix, suposant la probabilitat de cada un dels possibles valors (aquesta és la hipòtesi que volem contrastar). A continuació es compten les vegades que apareix cada un dels possibles valors en la successió i es comparen amb els valors esperats. Si els valors són massa diferents (d'aquí un moment quantificarem això) acceptarem que la successió no s'ajusta a la llei.

De moment, donem l'algorisme: Sigui s el nombre de possibles valors diferents que pot pendre cada terme de la successió, i anomenem aquests valors $1, 2, \ldots, s$. Sigui n_j el nombre de vegades que apareix l'element j en la successió, $1 \leq j \leq s$. Suposem que la probabilitat per a cada un d'aquests valors és p_j (aquesta és la hipòtesi que estem contrastant). Sigui n el nombre total de termes de la successió. Definim

$$V = \sum_{j=1}^{s} \frac{(n_j - np_j)^2}{np_j} = \frac{1}{n} \sum_{j=1}^{s} \left(\frac{n_j^2}{p_j} \right) - n.$$

La igualtat és immediata usant $\sum_j n_j = n$ i $\sum_j p_j = 1$. Noteu que si el valor de n_j obtingut coincideix amb el valor esperat np_j, llavors $V = 0$, i que com més s'aparti n_j d'aquest valor, més gran serà el valor de V.

Ara triem un nivell de confiança per fer el test (típicament el 95%) i busquem en una taula de la χ^2 (per exemple, a [15] o [11] en podeu trobar) si, amb el nivell de confiança escollit, no es rebutja la hipòtesi.

Aplicació a una $U([0,1])$

Si es vol aplicar a una successió de nombres reals a l'interval $[0,1]$ amb la intenció de verificar la seva uniformitat, llavors és habitual trencar l'interval en intervals disjunts $[0,1] = I_1 \cup I_2 \cup \ldots \cup I_s$. Llavors, es compta quantes vegades cau un terme de la successió dins de cada interval i s'aplica

el test χ^2 a aquests valors. Naturalment, la probabilitat p_j que un nombre caigui dins de l'interval I_j és precisament la longitud de I_j.

Finalment, notem que aquest test comprova la uniformitat dels nombres dins l'interval $[0,1]$ (la condició que anomenàvem C1) i, per tant, ens servirà per detectar generadors (dolents) que no omplin de manera adequada aquest interval.

1.4.2 Test de Kolmogorov-Smirnov

Aquest test s'aplica a successions que segueixen una llei contínua. El propòsit és verificar si la distribució empírica que s'obté a partir de la successió s'ajusta a la distribució esperada. Quan considerem el cas de lleis $U([0,1])$, aquest mètode és similar al χ^2 en el sentit que els dos comproven la "repartició uniforme" de la sèrie dins l'interval $[0,1]$. La diferència rau en el fet que aquí no ens cal trencar l'interval a trossos,[9] ja que el que anem a verificar és la funció de distribució.

Veiem com es fa el test en el cas d'una llei uniforme $[0,1]$: sigui $\alpha_1, \alpha_2, \ldots, \alpha_n$ una tira de nombres obtinguda amb el nostre generador. Posem aquests nombres en ordre creixent i tornem a anomenar α_j els termes de la sèrie ordenada, és a dir, $\alpha_1 \leq \alpha_2 \leq \cdots \leq \alpha_n$. Calculem els valors K_n^+ i K_n^- segons

$$K_n^+ = \sqrt{n} \max_{1 \leq j \leq n} \left(\frac{j}{n} - \alpha_j \right), \quad K_n^- = \sqrt{n} \max_{1 \leq j \leq n} \left(\alpha_j - \frac{j-1}{n} \right).$$

Si la distribució dels α_j és uniforme, els valors K_n^+ i K_n^- haurien de ser petits. El procediment és similar al d'abans: triem un nivell de confiança (per exemple, el 95%) i busquem en la taula de distribució (del test Kolmogorov-Smirnov) si no rebutjem la hipòtesi. Podeu trobar més detalls a [15] o [11].

1.4.3 Test espectral

Ja hem comentat a la secció 1.3.4 que, si agrupem termes consecutius d'una successió generada a partir d'un mètode de congruència (lineal o multiplicatica) per formar punts (del quadrat unitat, cub unitat, etc.), aquests punts no es distribueixen de manera uniforme sinó que s'agrupen en plans. El propòsit del test espectral és detectar aquest fenòmen.

Hem vist (a la secció 1.3.4) que el nombre màxim d'aquests plans es $(n\,!\,m)^{1/n}$ (n és la dimensió de l'espai i m el mòdul del generador) i que, si les constants del generador no estaven ben escollides, aquest nombre podia ser molt menor. Per tant, donats dos generadors de congruència, un criteri per comparar-los és calcular el nombre de plans concrets que tenim per a cada dimensió n. Un càlcul equivalent és buscar la distància entre aquests plans. Aquest valor dependrà, naturalment, de la dimensió en què estem treballant. Anomenem δ_2 la distància quan la dimensió és 2, és a dir, estem agrupant els punts de la sèrie en parelles i estem mirant quan properes estan les corresponents rectes del quadrat unitat. Direm δ_3 a la distància entre plans quan fem el test a dimensió 3 i, en general, direm δ_n a la distància entre hiperplans quan fem el test a dimensió n. És habitual anomenar ν_n a $1/\delta_n$.

Aquest test s'acostuma a aplicar per a valors de n no gaire grans, típicament $n = 2, 3, \ldots, 6$. El càlcul de ν_n per a aquests casos dóna informació de com el generador verifica les condicions C2, ..., C6 vistes abans.

[9] El problema de trencar l'interval és en el fet que és molt difícil saber quants trossos cal fer per detectar que una successió no és uniforme.

No donarem els detalls d'aquest test aquí, perquè la seva complexitat no el fa adequat (en la nostra opinió) per a un llibre introductori com aquest. El lector interessat pot consultar [11], on trobarà tant els detalls més teòrics com la seva implementació. El nostre propòsit en incloure aquest breu comentari sobre el test és recalcar que la construcció d'un bon generador de nombres aleatòris no és una tasca senzilla. La verificació del seu bon funcionament es basa en la aplicació d'una gran quantitat de tests, alguns d'ells força complexos (com aquest).

1.5 Generació de lleis no uniformes

En aquesta secció veurem alguns mètodes per obtenir altres lleis, suposant que disposem d'una $U([0,1])$. Es pot trobar més informació sobre aquests temes a [2] o [11]. A [19] hi podeu trobar, a més, la corresponent implementació en C.

En la resta de la secció suposarem, sense dir-ho de manera explícita, que $\{\alpha_n\}_n$ és una successió aleatòria obtinguda a partir d'una llei $U([0,1])$.

1.5.1 Llei exponencial

És molt fàcil d'obtenir, ja que la successió $\{e_n\}_n$ definida per

$$e_n = -\frac{\ln \alpha_n}{\lambda},$$

és una successió aleatòria que segueix una llei exponencial de paràmetre λ (la demostració es pot trobar a molts llocs, com per exemple [2], [11] o [19]). A [11] podeu trobar altres algorismes per a obtenir aquesta distribució.

Implementació

Creiem que no és necessari donar els detalls de la implementació en C d'aquest algorisme, donada la seva senzillesa. Volem només notar que, si el generador de $U([0,1])$ que usem produeix el valor 0, obtindrem un error en la funció logaritme (no es pot treure el logaritme de zero). Per tant, si el nostre generador pot produir el valor 0, caldrà afegir un `if` per controlar aquest fet. En cas que haguem generat el valor 0, l'ignorarem i farem una altra crida a la funció de nombres aleatòria per obtenir el següent valor de la seqüència.

1.5.2 Llei normal

Hi ha diversos mètodes per derivar lleis normals a partir d'uniformes $[0,1]$. Nosaltres hem triat el següent:[10] Definim la successió de $\mathbb{R}^2$ $\{(x_n, y_n)\}_n$ com

$$x_n = \cos(2\pi\alpha_{2n})\sqrt{-2\ln\alpha_{2n-1}}, \tag{1.7}$$

$$y_n = \sin(2\pi\alpha_{2n})\sqrt{-2\ln\alpha_{2n-1}}. \tag{1.8}$$

És a dir, (x_1, y_1) s'obté a partir de α_1 i α_2, (x_2, y_2) a partir de α_3 i α_4, etc. Llavors es pot demostrar (vegeu [2], [11] o [19]) que la successió

$$x_1, y_1, x_2, y_2, x_3, y_3, \ldots,$$

segueix una llei normal de mitjana zero i variància 1.

[10] Aquest mètode és conegut com mètode de Box-Muller.

Implementació

Abans d'entrar en els detalls de la implementació, volem remarcar un fet important: suposem que la successió $\{\alpha_n\}_n$ ha estat generada usant una congruència lineal:

$$X_n = (aX_{n-1} + c) \bmod m, \quad \alpha_n = \frac{X_n}{m}. \tag{1.9}$$

Llavors, és fàcil veure que (1.7) i (1.8) es poden reescriure com

$$\begin{aligned} x_n &= \cos\left(2\pi\left(a\alpha_{2n-1} + \frac{c}{m}\right)\right)\sqrt{-2\ln\alpha_{2n-1}}, \\ y_n &= \sin\left(2\pi\left(a\alpha_{2n-1} + \frac{c}{m}\right)\right)\sqrt{-2\ln\alpha_{2n-1}}. \end{aligned}$$

Noteu que els valors x_n, y_n estan fortament correlacionats. De fet, si els valors α_{2n-1} es mouen sobre tot $[0,1]$, els punts (x_n, y_n) es mouen sobre una espiral de $\mathbb{R}^2$ (recordeu que l'equació d'una espiral, en forma paramètrica, es pot posar com $x = t\cos(t)$, $y = t\sin(t)$, $t \in \mathbb{R}$). Per solventar aquest problema, cal usar un generador $U([0,1])$ que *no sigui de congruència lineal*. Això invalida el generador `alea0` que hem vist al principi. En canvi, no hi ha problema d'usar `alea1`, ja que la barreja fa que no tinguem la relació (1.9) i per tant tampoc tindrem el problema que acabem de comentar.

Una possible implementació és la següent:

```
#include <math.h>

float normal(long int *ap_llavor)
{
   float alea1(long int *ap_llavor);
   static int ctl=0;
   static float y,dospi=6.283185307;
   float x,a,b;
   if (ctl == 1) {ctl=0; return(y);}
   a=dospi*alea1(ap_llavor);
   b=sqrt(-2*log(alea1(ap_llavor)));
   x=b*cos(a);
   y=b*sin(a);
   ctl=1;
   return(x);
}
```

Aquesta versió genera una normal de mitjana 0 i variància 1. Per aconseguir mitjana i variància arbitràries podem usar el fet ben conegut (vegeu, per exemple, [15]) que, si $\{\beta_n\}_n$ és una successió que segueix una llei normal de mitjana 0 i variància 1, llavors la successió $\{\nu_n\}_n$ definida per

$$\nu_n = \sigma\beta_n + \mu,$$

segueix una llei normal de mitjana μ i variància σ^2.

Capítol 2 Tècniques de simulació discreta

En aquest capítol veurem com es fan simulacions de processos discrets que depenen d'algun paràmetre aleatori. Per donar la idea de com farem les simulacions, veurem primer un cas concret.

2.1 Un exemple senzill

Suposem que tenim una màquina amb quatre "peces" que cal canviar periòdicament, a causa del seu desgast. Aquesta màquina està sempre funcionant, i només es para per canviar alguna peça, quan s'espatlla.

Considereu ara una altra alternativa, motivada pel fet següent: com que per canviar una de les peces cal "obrir" la màquina, el temps necessari per canviar les quatre peces de cop és molt menor que quatre vegades el temps de canviar-ne una. Ens podem qüestionar, doncs, si és rentable l'estratègia següent: "cada cop que una peça s'espatlla, canviar totes quatre peces". La resposta depèn, evidentment, de quan costa cada peça, de quan perdem (o deixem de guanyar) si la màquina està parada, dels temps de substituir-ne una en relació al temps per substituir-les les quatre, etc.

Per tal de detallar la metodologia, donem valors a aquests paràmetres. Suposem que el temps de canviar una peça es pot modelar amb una llei normal de mitjana 15 minuts i variància 1 minut, i que el temps necessari per canviar-ne quatre es pot modelar per una normal de mitjana 25 minuts i variància 4 minuts. El temps de vida d'una peça el modelem per una llei exponencial de mitjana 500 minuts. Suposem que cada peça costa 50 (la unitat monetària és irrellevant), i que cada minut que la màquina resta aturada costa 10.

Volem avisar que aquest problema, a causa de la seva senzillesa, es pot resoldre analíticament (amb "paper i bolígraf").

Anem a veure com es pot organitzar una simulació d'aquest procés. En primer lloc, necessitem funcions que generin les quantitats aleatòries que ens calen. Anomenem `expo` la funció que genera exponencials de mitjana donada, i `normal` la que genera lleis normals amb mitjana i variància prefixades.

Noteu que, de fet, necessitem dos programes: un per simular la primera estratègia (canviar una peça cada cop) i avaluar-ne el cost, i un altre per a la segona estratègia.

2.1.1 Esdeveniments

Abans de detallar els organigrames, ens cal introduir alguns conceptes i definicions. Dividirem la simulació en una successió d'esdeveniments. Els esdeveniments són canvis en l'estat del sistema que estem simulant (en aquest cas, els esdeveniments són les avaries de les peces). Mentre no succeeix cap esdeveniment, el programa de simulació no ha de fer res. Quan hi ha un esdeveniment, el programa l'ha de "resoldre" (en aquest cas, ha de simular el reemplaçament de la peça espatllada). Per tant, orientarem el nostre programa d'acord amb aquests esdeveniments: per cada esdeveniment, farem les accions necessàries (essencialment, canviar la peça) i *calcularem quan passarà el següent esdeveniment*. Això ho podem fer perquè podem calcular el temps que duren les peces (exponencial de mitjana 500), i ens permetrà passar directament d'un esdeveniment al següent. Amb aquesta tècnica, aconseguirem que el programa sigui un bucle on, a cada passada, es va tractant cada esdeveniment. Noteu que, de fet, tenim quatre esdeveniments: les avaries de cada una de les peces, i que ens cal tenir aquests esdeveniments ordenats pel temps (hem de tractar-los en ordre cronològic).[1]

Un altre detall a remarcar és que els esdeveniments es componen de dues dades: el *que passa* (avaria, etc.) i *quan passa*. Aquest últim és l'hora en què l'esdeveniment succeeix.

Noteu que tota simulació té dos esdeveniments més: l'hora d'inici de la simulació i l'hora d'acabar-la. En l'exemple que estem tractant, l'hora d'inici correspon a engegar la màquina, i l'hora d'acabar-la pot ser o bé l'hora a la qual es para la màquina o bé l'hora a la qual nosaltres decidim que ja hem simulat prou. En aquest cas concret suposem que la màquina no para mai. Agafarem un temps de simulació de, per exemple, 100 hores.

2.1.2 Gestió dels esdeveniments

Per simplificar el programa, una bona opció és escriure unes funcions que s'encarreguin d'emmagatzemar de manera ordenada els esdeveniments futurs de la simulació. Aquest conjunt de funcions s'anomena AGENDA.

La agenda consta, essencialment, de dues funcions: una que introdueix esdeveniments (que anomenarem `posa_agenda`, i una altra que els treu de manera ordenada segons el temps (que anomenarem `treu_agenda`). Quan es posa un esdeveniment a l'agenda, aquest s'afegeix a la llista d'esdeveniments que ja hi ha a l'agenda. Quan es treu un esdeveniment, aquest s'esborra de l'agenda.

Més endavant (en la secció 2.1.4) veurem una implementació en C d'una agenda per a aquest cas concret.

2.1.3 Organigrama

Considerem el cas del primer programa (i.e., només canviem les peces espatllades). Per uniformitzar el programa, comptarem el temps en minuts (és a dir, la durada de la simulació serà 100 hores $\equiv$ 6000 minuts),

L'organigrama de la simulació podria ser el següent:

Pas 1. Posem a l'agenda l'esdeveniment `INICI` a temps 0, que correspon a engegar la màquina amb les quatre peces noves.

[1] En aquest exemple concret es poden fer alguns trucs per estalviar-se l'ordre dels quatre esdeveniments. Nosaltres no considerarem aquests tipus d'optimitzacions perquè no són aplicables en general, i el nostre propòsit és descriure una metodologia aplicable al major nombre possible de casos.

Pas 2. Posem a l'agenda l'esdeveniment `FINAL` a temps 6000, que correspon a acabar la simulació.

Pas 3. Treiem de l'agenda el proper esdeveniment.

Pas 4. Si l'esdeveniment és `INICI`, calculem el temps de vida de les quatre peces, i guardem els quatre esdeveniments (de tipus `AVARIA`) a l'agenda, cadascun amb el seu temps.

Pas 5. Si l'esdeveniment és `AVARIA` a temps `t0`, calculem el temps `tc` necessari per canviar la peça segons una normal de variància 1 minut i mitjana 15 minuts, i calculem el temps de vida (`t1`) de la nova peça segons una exponencial de mitjana 500. Posem a l'agenda l'esdeveniment `AVARIA` a temps `t0+tc+t1`.

Pas 6. Si l'esdeveniment és `FINAL`, la simulació ha acabat.

Pas 7. Anar al Pas 3.

Comentem-lo una mica. El Pas 1 és per inicialitzar el bucle principal del programa. El Pas 2 és per fixar el temps al qual pararem la simulació. El Pas 4 només s'executa un cop, al principi del programa. Arriben així al pas interessant, que és el 5. Aquest pas és el nucli de tota la simulació. Si de l'agenda ha sortit un esdeveniment `AVARIA`, mirem a quin temps passa aquesta avaria (això també ho diu l'agenda). Anomenem aquest temps `t0`. A continuació obtenim el temps necessari `tc` per canviar la peça i també la seva durada `t1`. Per tant, la peça que s'acaba d'instal·lar fallarà a temps `t0+tc+t1`, i per aquest motiu introduïm a l'agenda una avaria a aquesta hora.

Creiem que ara el funcionament de l'algorisme queda clar. Volem fer notar que a l'agenda només hi guardem, per cada peça, el següent temps en què fallarà. No hi guardem tots els temps de fallada. Això fa que sols estem guardant cinc esdeveniments: el `FINAL` i els quatre `AVARIA`.

2.1.4 Implementació en C

Veiem ara com seria una primera versió en C de l'algorisme que acabem de descriure.

L'agenda

En primer lloc, ens cal decidir com guardarem els esdeveniments. Donat que cada esdeveniment consta de dues dades (l'hora i el tipus), usarem una estructura. Per comoditat, podem definir un nou tipus de variable, que podem anomenar `esdev`, que contingui una variable de tipus `float` per al temps, i una variable de tipus `int` per al tipus d'esdeveniment.

Llavors, una implementació per a l'agenda pot ser la següent:

```
#include <stdio.h>
#include <stdlib.h>

#define N 5

typedef struct
{
   float quan;
```

```
   int que;
} esdev;

esdev agenda[N];
int ara=-1;

void posa_agenda(esdev e)
{
   int i;
   ++ara;
   if (ara == N) {puts("error. agenda plena."); exit(1);}
   for (i=ara; i>0; i--)
   {
      if (e.quan <= (agenda[i-1]).quan) break;
      agenda[i]=agenda[i-1];
   }
   agenda[i]=e;
}
int treu_agenda(esdev *e)
{
   if (ara == -1) return (0); /* agenda buida */
   *e=agenda[ara];
   --ara;
   return (1);
}
```

Observeu que hem creat un vector d'esdeveniments (anomenat `agenda`) de 5 components (ja hem comentat que, en aquest cas, cinc és el màxim nombre d'esdeveniments que tindrem). Aquest vector és global: l'hem declarat fora de qualsevol funció, de manera que és visible des de tot el fitxer. També hem creat una variable entera anomenada `ara`, i l'hem fet global. Indicarà l'última component plena del vector `agenda`, i ens servirà per controlar els esdeveniments que hi tenim.

Veiem què fan les funcions. La funció `posa_agenda` comprova, en primer lloc, si el vector `agenda` està ple. En aquest cas, la simulació no pot continuar (ens cal guardar un esdeveniment que no podem guardar), i el programa es para. Si això passés en aquest exemple, voldria dir que tenim un error de programació en alguna banda del programa, perquè ja hem dit que 5 esdeveniments són suficients. Si el vector `agenda` no és ple, llavors es posa aquest esdeveniment dins l'agenda, però de manera ordenada: volem que l'esdeveniment més distant en el temps (el que trigarà més temps a passar) estigui a la component 0 del vector, i que el més proper en el temps (el següent que passarà) estigui a la component més alta. Per això fem una cerca pel vector i insertem l'esdeveniment `e` allà on toqui. Els detalls d'aquesta ordenació no són difícils, i els deixem per al lector.

La funció `treu_agenda` retorna el proper esdeveniment de la simulació. Com que la rutina `posa_agenda` ja ha col·locat els esdeveniments de manera ordenada, aquesta funció és molt senzilla: només ha d'agafar la component `ara` de l'agenda i retornar-la. En cas que l'agenda fos buida, la funció retorna un zero (altrament, retorna un 1). Noteu que, a diferència de la

rutina `posa_agenda`, no hem de parar la simulació: el programa pot continuar executant-se i el programa principal ja decidirà si el fet que l'agenda estigui buida és un error o no (més endavant veurem exemples on no és cap error que l'agenda es quedi buida a mitja simulació).

El programa principal

Una versió del programa principal és la següent:

```
#include <math.h>
#include <stdio.h>

#define   INICI  1
#define  AVARIA  2
#define   FINAL  3

void main(void)
{
   typedef struct
   {
      float quan;
      int que;
   } esdev;
   float normal(long int *ap_llavor);
   float expo(float m, long int *ap_llavor);
   void posa_agenda(esdev e);
   int treu_agenda(esdev *e);
   long int llavor;
   float t,cost;
   int fi,j;
   esdev e;
   llavor=-1;
   fi=0;
   e.que=INICI;
   e.quan=0.0;
   posa_agenda(e);
   e.que=FINAL;
   e.quan=6000.0;
   posa_agenda(e);
   cost=0.0;
   while (fi == 0)
   {
      treu_agenda(&e);
      switch (e.que)
      {
         case INICI:
            e.que=AVARIA;
            for (j=0; j<4; j++)
```

```
            {
               e.quan=expo(500,&llavor);
               posa_agenda(e);
            }
            break;
         case AVARIA:
            t=15+normal(&llavor);
            cost += 50+10*t;
            e.quan += t+expo(500,&llavor);
            posa_agenda(e);
            break;
         case FINAL:
            fi=1;
            break;
         default:
            puts("error: esdeveniment desconegut.");
      }
   }
   printf("cost total: %f\n",cost);
}
float expo(float m, long int *ap_llavor)
{
   float alea1(long int *ap_llavor);
   return(-m*log(alea1(ap_llavor)));
}
```

Comentem una mica el seu funcionament. En primer lloc, omplim l'agenda amb els dos esdeveniments que coneixem (l'hora d'inici i acabament de la simulació) i inicialitzem la variable `cost` a zero (aquí hi guardarem el cost de l'estratègia que estem simulant).

A continuació ve el bucle principal del programa: es va cridant l'agenda i, per cada esdeveniment que passa anem fent les accions oportunes. Si l'esdeveniment és `INICI`, calculem el temps d'avaria de les 4 peces i els posem a l'agenda. Si és `AVARIA`, calculem el temps usat per canviar la peça, que ens dóna el cost de l'avaria. A continuació calculem el temps de vida de la nova peça amb el qual podem saber l'hora a què s'espatllarà (hora de l'avaria actual + temps de canviar la peça + temps de vida de la nova peça). Per tant, posem aquest esdeveniment a l'agenda. Si l'esdeveniment és `FINAL`, parem la simulació (tot i que encara queden esdeveniments a l'agenda).

Per simplificar una mica el programa, la generació del temps de vida de les peces s'ha posat a part al final, en forma de funció anomenada `expo`.

2.1.5 Comentaris

El propòsit d'aquesta secció ha estat donar una visió general del tipus de tècniques que usarem per simular. Volem subratllar com el fet d'introduir l'agenda ha simplificat la simulació, de manera que el programa principal es redueix a un bucle senzill. Per altra banda, l'agenda tampoc és gens complexa. El gran avantatge és en el fet que, per a moltes simulacions, es pot usar la mateixa agenda (o amb canvis mínims). Es pot dir que l'agenda és el que tenen en comú

totes (bé, gairebé totes) les simulacions. Per tant, és una bona política separar l'agenda de la resta i programar-la de manera general. Això ens simplificarà molt l'escriure els programes corresponents.

En les següents seccions donarem algunes (petites) modificacions a l'agenda que hem vist aquí per fer-la el més general possible. També veurem la manera de fer un grup de rutines (amb la mateixa filosofia de l'agenda) per gestionar cues. Totes aquestes rutines ens seran molt útils en les següents seccions, en les quals construirem simuladors per a alguns problemes concrets.

2.2 Gestió d'agendes

Ja hem explicat el funcionament i propòsit de l'agenda. Ara donarem només unes petites modificacions per fer-la una mica més general.

Un inconvenient que té l'agenda que hem vist és el següent: suposem que volem fer moltes simulacions (amb llavors diferents) del problema de la secció anterior, amb la idea de veure la mitjana i la variància dels resultats. Una manera simple seria posar un bucle més en el programa principal i fer variar la llavor, i anar escrivint els diferents resultats. L'inconvenient és que, després d'haver fet una simulació, ens poden quedar esdeveniments a l'agenda que impedeixen tornar-la a usar (per exemple, en l'exemple de la màquina amb quatre peces ens queden a l'agenda quatre esdeveniments de tipus `AVARIA`). Per tant, afegirem a l'agenda una rutina que s'encarregui de buidar-la.

Una segona versió de l'agenda podria ser la següent:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct
{
   float quan;
   int que;
} esdev;

static esdev *agenda;
static int n_esd,ara;

void ini_agenda(int n)
{
   n_esd=n;
   agenda=(esdev*)malloc(n_esd*sizeof(esdev));
   if (agenda == NULL) {puts("agenda: falta memoria."); exit(1);}
   ara=-1;
}
void posa_agenda(esdev e)
{
   int i;
   ++ara;
   if (ara == n_esd) {puts("error. agenda plena."); exit(1);}
   for (i=ara; i>0; i--)
```

```
   {
      if (e.quan <= (agenda[i-1]).quan) break;
      agenda[i]=agenda[i-1];
   }
   agenda[i]=e;
}
int treu_agenda(esdev *e)
{
   if (ara == -1) return (0); /* agenda buida */
   *e=agenda[ara];
   --ara;
   return (1);
}
void buida_agenda(void)
{
   ara=-1;
}
void llibera_agenda(void)
{
   free(agenda);
}
```

Com podeu veure, els canvis són mínims: hem afegit la rutina `ini_agenda` per poder crear una agenda de qualsevol dimensió des del programa principal. La rutina `buida_agenda` permet posar a zero l'agenda de cara a fer una altra simulació. Finalment, la rutina `llibera_agenda` serveix per alliberar la memòria reservada per `ini_agenda`. Aquesta rutina ens caldrà si volem, dins d'un mateix programa, fer dues simulacions que requereixin agendes de diferent longitud, sense haver de dimensionar sempre a la longitud de l'agenda més llarga.

Més endavant veurem exemples que usen aquesta agenda.

2.3 Gestió de cues

Suposem ara que volem fer una simulació que ens requereix gestionar una cua (per exemple, els caixers d'un banc o supermercat). Ens seria molt útil disposar d'un conjunt de rutines que s'encarreguessin de fer-ho: voldríem rutines que posessin gent a la cua, que ens traiessin el següent de la cua, que ens diguessin quan llarga és la cua, etc. A més, seria molt pràctic que aquestes rutines admetessin la possibilitat de gestionar més d'una cua, ja que hi ha molts casos en què això es requereix. Finalment, voldríem que fos el més independent possible del cas concret que estem simulant, per poder-lo usar en molts problemes diferents.

En aquesta secció veurem maneres d'implementar gestors de cues per satisfer les demandes del paràgraf anterior. Per simplificar la discussió, veurem primer un cas senzill, en què només tenim una cua. Més endavant veurem casos més complexos, on hi ha diverses cues amb règims de funcionament diferent.

2.3.1 Una única cua

Suposem que volem simular l'evolució de les cues que es formen davant d'un caixer automàtic (de moment, suposarem que només n'hi ha un). Estem interessats a obtenir informació com: longitud màxima de la cua, temps màxim i temps mitjà d'espera dels clients, etc. Suposarem que el temps que passa entre l'arribada de dos clients segueix una llei exponencial de 2 minuts de mitjana, i que el temps que passa cada client davant del caixer és de la forma $1 + t$, on t segueix una exponencial d'1 minut de mitjana.

Primera versió

Per a aquesta simulació usarem l'agenda que hem vist a la secció 2.2. A més, ens anirà molt bé disposar d'un joc de funcions que s'encarreguin de gestionar la cua (de la mateixa manera que ho hem fet amb l'agenda). Una possible implementació d'aquest gestor de cues (per al cas d'una única cua) podria ser la que donem tot seguit:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct
{
   float tar;
} el_cua;

static el_cua *cua;
static int max_cua,ini_cua,fin_cua,lon_cua;

void crea_cua(int l)
{
   max_cua=l;
   cua=(el_cua*)malloc(max_cua*sizeof(el_cua));
   if (cua == NULL) {puts("ini_cua: error 1."); exit(1);}
   ini_cua=0;
   fin_cua=0;
   lon_cua=0;
}
int posa_cua(el_cua c)
{
   if (lon_cua == max_cua) return(0);
   ++lon_cua;
   cua[fin_cua]=c;
   ++fin_cua;
   if (fin_cua == max_cua) fin_cua=0;
   return(1);
}
int treu_cua(el_cua *c)
{
   if (lon_cua == 0) return(0);
```

```
   --lon_cua;
   *c=cua[ini_cua];
   ++ini_cua;
   if (ini_cua == max_cua) ini_cua=0;
   return(1);
}
int long_cua(void)
{
   return(lon_cua);
}
void elim_cua(void)
{
   free(cua);
}
```

Comentem el seu funcionament. En primer lloc, definim una estructura anomenada `el_cua` que representa la persona que està fent cua. En aquesta estructura guardem l'hora a la qual la persona ha arribat a la cua (`tar`). Amb això, quan la persona deixi la cua per anar al caixer, podrem saber quant temps ha estat fent cua. El fet d'usar una estructura (en lloc d'una simple variable `float`) permet afegir informació addicional sobre la persona que fa cua. Per exemple, si la cua correspon al caixer d'un supermercat, podrem afegir un nou membre amb el nombre de productes que porta el client, amb el propòsit de calcular quant de temps estarà pagant (el temps de pagament depèn del nombre de productes). Noteu que si afegim més membres a l'estructura `el_cua` no cal modificar el gestor de cues, ja que aquests membres no apareixen enlloc del programa.

Cua circular

A continuació tenim la declaració de l'apuntador `cua`. L'usarem per crear un vector d'elements de tipus `el_cua`, que contindrà la cua que simulem. De moment, usarem una cua de tipus circular:[2] una cua circular és un vector (on guardarem la cua) amb dos índexs que indiquen la primera i l'última component de la cua. Inicialment el vector es buit i els dos índexs (els podem dir `ini_cua` i `fin_cua`, com en el programa) valen zero. Quan arriben els primers clients a la cua, el vector es va omplint i l'índex que indica el seu final es va incrementant de manera adient (és a dir, `ini_cua` continua valent zero, pero `fin_cua` ja no). Quan algú ha de sortir de la cua (naturalment, el primer que surt és el primer que hi ha arribat), l'índex `ini_cua` ens diu quin és el primer que ha de sortir. Per tant, traiem aquest client i incrementem el valor de `ini_cua`. Amb aquest procediment, la cua es "desplaça" al llarg del vector on la guardem. Evidentment, el més natural és que en algun moment l'índex `fin_cua` arribi a l'última component del vector sense que la cua estigui plena (segurament, alguns clients hauran sortit de la cua i `ini_cua` tindrà un cert valor positiu). En aquest moment, si cal afegir un altre client a la cua, "donarem la volta" al vector (com si la primera i l'última component estiguessin connectades) i posarem aquest nou client a la component zero del vector, fent que `fin_cua` valgui també zero. Per tant, la condició de cua plena no serà que `fin_cua` arribi al final del vector, sinó que "atrapi" a `ini_cua`. Naturalment, quan `ini_cua` arribi al final del vector `cua`, també li "donarem la volta" i el farem tornar a començar per zero.

[2] A la secció 2.3.4 veurem un altre sistema per gestionar la cua.

Noteu que l'avantatge d'aquest esquema és la seva rapidesa. Tant posar un client a la cua com treure'l necessita molt poques operacions, que a més són fàcils de programar. Els seu principal inconvenient és que cal saber, al principi de la simulació, quant llarg ha de ser el vector `cua`. De moment, per mantenir la simplicitat dels programes, ens acontentarem a donar "a ull" la longitud màxima d'aquesta cua. En la secció 2.3.4 veurem una altra tècnica de gestió de cues que no presenta aquest inconvenient.

Finalment, hem afegit un altre índex (`lon_cua`) que conté el nombre de persones que hi ha a la cua. Estrictament parlant aquest índex és innecessari, ja que el seu valor pot ser deduït a partir dels valors de `ini_cua` i `fin_cua`. La raó d'incloure'l ha estat que, en la nostra opinió, el programa queda una mica més clar.

Les funcions

La rutina `crea_cua` dimensiona el vector `cua` a la longitud demanada. Al mateix temps inicialitza els diversos índexs del programa: a més dels que acabem de comentar, tenim l'índex `max_cua`, que conté la longitud del vector cua (correspon al nombre màxim de persones que podem tenir fent cua). Naturalment, cal cridar aquesta rutina abans d'usar qualsevol de les funcions que gestionen la cua.

La funció `posa_cua` serveix per afegir un nou client a la cua. El seu funcionament és molt senzill (tenint en compte el que s'ha dit de cues circulars): primer verifiquem que la cua no estigui plena, després posem el client a la component indicada per `fin_cua`, i incrementem aquest índex. Si, en fer això, `fin_cua` és més gran que `max_cua`, vol dir que hem de "donar la volta" al vector i per tant posem `fin_cua` a zero. El valor retornat per la funció és 1 si tot ha anat bé, i 0 si la cua era plena i no hem pogut afegir el client. Noteu que, en aquest últim cas, ens caldrà parar la simulació, ja que hem arribat a una situació "irresoluble": no podem guardar el client (no tenim espai a la cua) i tampoc podem ignorar-lo (la simulació perdria sentit). Generalment, cal tornar a començar amb un vector `cua` més llarg.

La funció `treu_cua` ens retorna el primer client de la cua. Cal passar-li una estructura de tipus `el_cua` per adreça i ens retornarà el primer client que surt de la cua dins aquesta estructura. Creiem que el seu funcionament hauria de quedar clar amb les explicacions anteriors i, per tant, no el comentem. Només direm que la funció retorna un 0 si la cua es buida i un 1 en cas contrari. Noteu que, a diferència de la funció `posa_cua`, si la cua és buida no tenim cap problema.

Per acabar, tenim les funcions `long_cua` i `elim_cua`. La funció `long_cua` retorna la llargada de la cua, i la funció `elim_cua` allibera la memòria reservada per `crea_cua` al crear la cua.

El programa principal

Veiem ara com seria un programa principal per fer la simulació que estem comentant.

```
#include <math.h>
#include <stdio.h>

#define     OBRIR  1
#define  ARRIBADA  2
#define   SORTIDA  3
#define    TANCAR  4
```

```
void main(void)
{
   typedef struct
   {
      float quan;
      int que;
   } esdev;
   typedef struct
   {
      float tar;
   } el_cua;
   void ini_agenda(int n);
   void posa_agenda(esdev e);
   int treu_agenda(esdev *e);
   void buida_agenda(void);
   void crea_cua(int l);
   int posa_cua(el_cua c);
   int treu_cua(el_cua *c);
   int long_cua(void);
   void elim_cua(void);
   float expo(float m, long int *ap_llavor);
   esdev e;
   el_cua c;
   float t,tmax;
   long int llavor;
   int bn,caixa,j,nca;
   ini_agenda(3);
   crea_cua(100);
   llavor=-1995;
   e.que=OBRIR;
   e.quan=0.0;
   posa_agenda(e);
   e.que=TANCAR;
   e.quan=720.0;
   posa_agenda(e);
   bn=0;
   caixa=0;
   nca=0;
   tmax=0.0;
   while (treu_agenda(&e) != 0)
   {
      switch (e.que)
      {
         case OBRIR:
            bn=1;
            caixa=0;
```

```
            e.que=ARRIBADA;
            e.quan=expo(2,&llavor);
            posa_agenda(e);
            break;
         case ARRIBADA:
            if (bn == 1)
            {
               t=e.quan;
               if (caixa == 0)
               {
                  caixa=1;
                  e.quan += 1+expo(1,&llavor);
                  e.que=SORTIDA;
                  posa_agenda(e);
               }
               else
               {
                  c.tar=t;
                  j=posa_cua(c);
                  if (j == 0) {puts("error: cua massa petita"); exit(1);}
               }
               e.quan=t+expo(2,&llavor);
               e.que=ARRIBADA;
               posa_agenda(e);
            }
            break;
         case SORTIDA:
            ++nca;
            j=treu_cua(&c);
            if (j != 0)
{
               t=e.quan-c.tar;
               if (t > tmax) tmax=t;
               e.quan += 1+expo(1,&llavor);
               e.que=SORTIDA;
               posa_agenda(e);
}
            else
            {
               caixa=0;
            }
            break;
         case TANCAR:
            bn=0;
            break;
         default:
            puts("error: esdeveniment desconegut.");
```

```
      }
   }
   printf("nombre de clients atesos: %d\n",nca);
   printf("temps maxim de cua: %f\n",tmax);
}
float expo(float m, long int *ap_llavor)
{
   float alea1(long int *ap_llavor);
   return(-m*log(alea1(ap_llavor)));
}
```

Aquest programa utilitza l'agenda de la secció 2.2, el gestor de cues (secció 2.3.1) i la funció `alea1`. El seu funcionament és molt similar al del programa de la secció 2.1.4: en primer lloc definim els tipus d'esdeveniments que tindrem en la simulació: `OBRIR` (posar en marxa el caixer, correspon a l'inici de la simulació), `ARRIBADA` (arriba un client), `SORTIDA` (un client se'n va, després d'haver estat atès) i `TANCAR` (parem el caixer). Naturalment, ens cal definir els tipus `esdev` i `el_cua`.

El programa comença inicialitzant l'agenda (noteu que només hi guardarem 3 esdeveniments alhora), la cua (el valor 100 l'hem posat "a ull"; si és massa petit, el programa ja avisarà) i la llavor per als nombres aleatoris. Després posem els esdeveniments `OBRIR` i `TANCAR` a l'agenda, que indicaran el principi i la fi de la simulació.[3] La variable `bn` és una bandera que usarem per saber si el caixer és obert (valor 1) o no (valor 0). La raó d'incloure aquesta bandera és que, per tancar el caixer, el que farem serà tancar les portes d'accés (per no acceptar més clients) i deixar que acabin les persones que fan cua. Per aquest motiu, la simulació no acabarà amb l'esdeveniment `TANCAR`, sinó que continuarà fins que no quedi ningú per atendre. La bandera `bn` ens indicarà, doncs, si hem d'acceptar arribades o no (tota arribada d'un client quan `bn` sigui 0 no ha de ser considerada). Una altra variable auxiliar és `caixa`, que indica si el caixer és ocupat (val 1) o no (val 0). L'usarem per saber si, quan arriba un client, l'hem de posar al caixer (ho farem si `caixa` val 0) o a la cua (si `caixa` val 1). A més, el programa usa altres variables auxiliars, com `nca` i `tmax`, que usarem per acumular el nombre de clients atesos i el temps màxim que una persona ha estat fent cua.

A continuació ve el bucle principal de la simulació: es van traient esdeveniments de l'agenda i es prenen les corresponents accions. Si l'esdeveniment és `OBRIR`, les accions que cal fer són: posar `bn` a 1, `caixa` a 0, generar l'arribada del primer client i posar-lo a l'agenda.

Si l'esdevenimemt és `ARRIBADA`, mirem si les portes són obertes (és a dir, si `bn` és igual a 1), perquè només llavors tindrem en compte l'arribada. Si aquest és el cas, si el caixer és buit posem la persona al caixer (fem `caixa=1`), calculem el temps que hi serà, i posem a l'agenda un esdeveniment `SORTIDA` a l'hora que marxarà (hora d'arribada + temps al caixer). Si quan arriba el client el caixer està ocupat, el posem a la cua. Noteu que cal guardar l'hora a què ha arribat, perquè quan surti de la cua voldrem saber quan temps hi ha estat. Finalment, i tant si l'hem posat al caixer com a la cua, generem l'arribada del següent client. Noteu que això només es pot fer en aquest punt del programa: com que el que sabem és la llei que segueix el temps entre arribades de clients, quan tenim l'arribada d'un client podem generar l'arribada del següent.

[3] De moment, hem posat un temps de funcionament de 12 hores. Evidentment, no hi ha cap problema a posar un altre valor.

El tractament de l'esdeveniment `SORTIDA` és com segueix: en primer lloc, incrementem en 1 el nombre de clients atesos i mirem si hi ha un client a la cua. Si hi és, mirem a quina hora aquest client ha començat la cua i deduïm el temps que hi ha estat. Això ens serveix per actualitzar la variable que conté el temps màxim que algú ha estat en cua. A continuació, calculem el temps que aquest client estarà ocupant el caixer i posem el corresponent esdeveniment `SORTIDA` a l'agenda. Naturalment, si quan el client se'n va no hi ha ningú fent cua, només cal indicar que el caixer queda buit (`caixa=0`).

Per acabar, si l'esdeveniment és `TANCAR`, posem la bandera `bn` a zero. L'últim `default` és de control: no pot aparèixer cap esdeveniment que no sigui un d'aquests quatre. Si això succeeix, vol dir que hi ha un error de programació.

Deixem com a exercici per al lector modificar aquest programa per mesurar altres paràmetres, com el temps mitja de cua, el percentatge de gent que no ha fet cua, la longitud màxima de la cua, l'evolució de la longitud de la cua respecte del temps, etc. Noteu que per fer això no cal alterar l'algorisme de simulació, només cal afegir (als llocs adequats) variables que "comptin" el que volem observar.

Segona versió

Suposem ara que volem modificar el programa per estudiar una variant d'aquest problema. La variant és la següent: tenim l'oportunitat de comprar caixers d'un nou tipus més ràpid, cosa que implica que els temps de cua seran menors. Volem quantificar aquesta reducció del temps de cua, per decidir si val la pena fer la inversió o no. Per ser precisos, suposem que el temps de servei amb els nous caixers és 45 segons més una exponencial de mitjana 1 minut. Noteu que no costa gens modificar el programa anterior per simular aquest cas: només cal canviar el lloc on es genera el temps d'estada davant el caixer.

Moltes vegades estem interessats a fer comparacions sota les mateixes condicions: en el cas que estem tractant aquí, volem saber la reducció de cua deguda als nous caixers. Si per cada un dels dos casos usem arribades diferents de clients, podria ser que la reducció de cua observada fos deguda a un cert tipus d'arribada de clients que, per casualitat, s'ha produït en un dels casos. Cal dir també que, per obtenir resultats fiables, cal fer moltes simulacions i observar mitjana, variància, etc. dels resultats. Si tenim la mateixa arribada de clients en els dos casos, la convergència de les simulacions és més ràpida que si aquestes arribades són diferents. Quan l'arribada de clients és igual en els dos casos es poden aplicar tècniques de reducció de variància (vegeu, per exemple, [2] o [17]) que permeten obtenir resultats més acurats. Nosaltres no tractarem aquestes tècniques aquí, ja que són més apropiades per a un curs d'estadística. Com ja hem dit, el nostre únic propòsit és donar les eines necessàries per poder fer les simulacions.

Observeu que si modifiquem el programa de la manera que hem dit abans (simplement canviant la generació del temps de servei al caixer) *estem modificant l'arribada de clients*, tot i que usem la mateixa llavor per a les dues simulacions. La raó és que per tenir la mateixa arribada de clients (portant cada client la mateixa "feina" per fer al caixer) cal que es mantingui l'ordre de les crides a les rutines de nombres aleatoris. Si canviem els temps d'estada al caixer, pot passar que un client que abandonava el caixer després d'una certa arribada ara ho faci abans, per tant els nombres aleatoris (de `alea1`) intercanvien els papers. El nombre aleatori que abans s'usava per calcular una nova arribada, ara es fa servir per calcular el temps d'estada al caixer de la següent persona de la cua.

Anem ara a donar una nova versió del programa, modificada per permetre fer les dues simulacions amb la mateixa arribada de clients.

```
#include <math.h>
#include <stdio.h>

#define     OBRIR  1
#define  ARRIBADA  2
#define   SORTIDA  3
#define    TANCAR  4

void main(void)
{
   typedef struct
   {
      float quan;
      int que;
   } esdev;
   typedef struct
   {
      float tar;
      float tse;
   } el_cua;
   void ini_agenda(int n);
   void posa_agenda(esdev e);
   int treu_agenda(esdev *e);
   void buida_agenda(void);
   void crea_cua(int l);
   int posa_cua(el_cua c);
   int treu_cua(el_cua *c);
   int long_cua(void);
   void elim_cua(void);
   float expo(float m, long int *ap_llavor);
   esdev e;
   el_cua c;
   float t,tmax,tserv;
   long int llavor;
   int bn,caixa,j,nca;
   ini_agenda(3);
   crea_cua(100);
   llavor=-1995;
   e.que=OBRIR;
   e.quan=0.0;
   posa_agenda(e);
   e.que=TANCAR;
   e.quan=720.0;
   posa_agenda(e);
   bn=0;
   caixa=0;
   nca=0;
```

```
tmax=0.0;
while (treu_agenda(&e) != 0)
{
   switch (e.que)
   {
      case OBRIR:
         bn=1;
         caixa=0;
         e.que=ARRIBADA;
         e.quan=expo(2,&llavor);
         posa_agenda(e);
         tserv=1+expo(1,&llavor);
         break;
      case ARRIBADA:
         if (bn == 1)
         {
            t=e.quan;
            if (caixa == 0)
            {
               caixa=1;
               e.quan += tserv;
               e.que=SORTIDA;
               posa_agenda(e);
            }
            else
            {
               c.tar=t;
               c.tse=tserv;
               j=posa_cua(c);
               if (j == 0) {puts("error: cua massa petita"); exit(1);}
            }
            e.quan=t+expo(2,&llavor);
            e.que=ARRIBADA;
            posa_agenda(e);
            tserv=1+expo(1,&llavor);
         }
         break;
      case SORTIDA:
         ++nca;
         j=treu_cua(&c);
         if (j != 0)
 {
            t=e.quan-c.tar;
            if (t > tmax) tmax=t;
            e.quan += c.tse;
            e.que=SORTIDA;
            posa_agenda(e);
```

```
    }
            else
            {
               caixa=0;
            }
            break;
         case TANCAR:
            bn=0;
            break;
         default:
            puts("error: esdeveniment desconegut.");
      }
   }
   printf("nombre de clients atesos: %d\n",nca);
   printf("temps maxim de cua: %f\n",tmax);
}
float expo(float m, long int *ap_llavor)
{
   float alea1(long int *ap_llavor);
   return(-m*log(alea1(ap_llavor)));
}
```

Abans de comentar aquesta nova implementació, creiem que val la pena explicar la idea de l'algorisme. El problema de la versió anterior venia de generar el temps d'estada al caixer en el moment que el client l'ocupava. Si generem el temps d'estada al caixer en el moment que la persona arriba al sistema, aquest problema desapareix: per cada `ARRIBADA`, calculem el temps que passarà al caixer i calculem l'hora d'arribada del client següent. D'aquesta manera, els nombres aleatòris s'usen amb el mateix ordre: el primer s'usa per calcular l'arribada del primer client, el segon per calcular el seu temps de servei al caixer, el tercer per a l'arribada del segon client, el quart pel seu temps de servei, etc. Si alterem la llei que regeix el temps de servei al caixer, les hores d'arribada no varien.

Aquest mètode ens obliga a "apuntar", per a cada client que és en cua, el seu temps de servei prèviament calculat. Una manera fàcil de fer-ho és afegir un nou camp (diem-li `tse`, per `t`emps de `s`ervei) a l'estructura `el_cua`. El fet d'afegir aquest camp pràcticament no altera el gestor de cues: només cal modificar la declaració de `el_cua` a la capçalera del programa perquè inclogui el nou camp `tse`. També necessitem una variable auxiliar (li hem dit `tserv`) per apuntar el temps de servei de la propera `ARRIBADA`: el fet de generar el temps de servei juntament amb l'hora d'arribada provoca que haguem de guardar aquest temps de servei fins que es produeix de manera efectiva aquesta arribada.[4] En aquest moment posem el client al caixer o a la cua (amb el temps de servei `tserv`) i generem l'arribada del següent i el seu temps de servei (que guardem a `tserv`).

Les modificacions del programa són molt poques. Essencialment es tracta d'alterar la gestió de l'esdeveniment `ARRIBADA` de manera que s'encarregui també de la generació del temps d'arribada. Finalment recordem que, per fer funcionar el programa, cal usar el gestor de cues amb la modificació d'incloure el camp `tse` a l'estructura `el_cua`.

[4] Una altra possibilitat és afegir un nou camp a l'estructura `esdev` per guardar-hi aquest valor `tserv`. Naturalment, aquest nou camp només l'usariem en cas que l'esdeveniment fos una `ARRIBADA`.

2.3.2 Més d'una cua: cues del mateix tipus

Ara volem estendre la simulació anterior al cas en què tenim més d'un caixer, amb una cua davant de cada caixer. Estem interessats a saber com depèn la longitud i el temps mitjà d'espera (a la cua) del nombre de caixers que tenim installats.

L'agenda

En aquesta simulació, el nombre d'esdeveniments possibles no està determinat d'entrada: a més dels `OBRIR`, `TANCAR` i `ARRIBADA`, tenim un esdeveniment del tipus `SORTIDA` per cada un dels caixers. Aquests han de ser diferents, perquè per saber de quina cua hem de treure el client que posarem al caixer cal saber en quin caixer s'ha produït la `SORTIDA`.

Si el nombre de caixers és molt reduït, una solució és distinguir les diferents sortides: per exemple, `SORTIDA1` indicaria una sortida al primer caixer, `SORTIDA2` al segon, etc. Evidentment, aquesta solució és impracticable si el nombre de caixers és gran. Una solució és que "apuntem", per cada esdeveniment `SORTIDA`, el caixer en què s'ha produït. És molt similar a allò que hem fet a la secció anterior amb la cua, però en aquest cas caldrà fer-ho a l'agenda.

El que farem, doncs, serà afegir el camp `on` (de tipus `int`) a l'estructura `esdev`. Noteu que aquest canvi no repercuteix en el funcionament de l'agenda que hem vist a l'apartat 2.2. Aquest nou camp només l'usarem en el cas que l'esdeveniment sigui `SORTIDA`, per apuntar-hi el número de caixer en què aquesta s'ha produït.

Per simplificar la declaració de les variables, hem creat el fitxer `agenda.h`, que conté les declaracions del tipus `esdev` i les funcions de l'agenda:

```
typedef struct
{
   float quan;
   int que;
   int on;
} esdev;

void ini_agenda(int n);
void posa_agenda(esdev e);
int treu_agenda(esdev *e);
void buida_agenda(void);
void llibera_agenda(void);
```

El nou gestor de cues

Aquí és on hi ha els canvis més importants respecte del que ja hem fet. Ara volem un gestor capaç de manejar un nombre arbitrari de cues, amb un funcionament similar al gestor d'una sola cua.

Igual que en el cas de l'agenda, hem introduït el fitxer `cues.h`, amb les declaracions dels tipus de variables i funcions que calen per usar el gestor de cues:

```
typedef struct
{
   float tar;
```

```
   float tse;
} el_cua;

void ini_cues(int n, int l);
int posa_cua(el_cua c, int k);
int treu_cua(el_cua *c, int k);
int long_cua(int k);
int cua_mes_curta(void);
void elim_cues(void);
```

A continuació donem el codi en C d'aquest gestor i després en discutirem el funcionament.

```
#include <stdio.h>
#include <stdlib.h>

#include "cues.h"

static el_cua **cues;
static int *ini_cua,*fin_cua,*lon_cua;
static int num_cues,max_cua;

void ini_cues(int n, int l)
{
   int i;
   num_cues=n;
   max_cua=l;
   cues=(el_cua**)malloc(num_cues*sizeof(el_cua*));
   if (cues == NULL) {puts("ini_cues: error 1."); exit(1);}
   for (i=0; i<num_cues; i++)
   {
      cues[i]=(el_cua*)malloc(max_cua*sizeof(el_cua));
      if (cues[i] == NULL) {puts("ini_cues: error 2."); exit(1);}
   }
   ini_cua=(int*)calloc(num_cues,sizeof(int));
   if (ini_cua == NULL) {puts("ini_cues: error 3."); exit(1);}
   fin_cua=(int*)calloc(num_cues,sizeof(int));
   if (fin_cua == NULL) {puts("ini_cues: error 4."); exit(1);}
   lon_cua=(int*)calloc(num_cues,sizeof(int));
   if (lon_cua == NULL) {puts("ini_cues: error 5."); exit(1);}
}
int posa_cua(el_cua c, int k)
{
   if ((k >= num_cues) || (k < 0)){puts("posa_cua: cua inexistent"); exit(1);}
   if (lon_cua[k] == max_cua) return(0);
   ++lon_cua[k];
   cues[k][fin_cua[k]]=c;
   ++fin_cua[k];
```

```
   if (fin_cua[k] == max_cua) fin_cua[k]=0;
   return(1);
}
int treu_cua(el_cua *c, int k)
{
   if ((k >= num_cues) || (k < 0)){puts("treu_cua: cua inexistent"); exit(1);}
   if (lon_cua[k] == 0) return(0);
   --lon_cua[k];
   *c=cues[k][ini_cua[k]];
   ++ini_cua[k];
   if (ini_cua[k] == max_cua) ini_cua[k]=0;
   return(1);
}
int long_cua(int k)
{
   return(lon_cua[k]);
}
int cua_mes_curta(void)
{
   int j,k;
   j=max_cua;
   for (k=0; k<num_cues; k++) if (j > lon_cua[k]) j=k;
   return(j);
}
void elim_cues(void)
{
   int i;
   free(lon_cua);
   free(fin_cua);
   free(ini_cua);
   for (i=0; i<num_cues; i++) free(cues[i]);
   free(cues);
}
```

Aquestes funcions són una extensió natural de les del gestor de cues anterior. Ara, en lloc d'un vector on guardar la cua, hi tenim un vector de vectors (és a dir, una matriu), anomenat `cues`. Cada un d'aquests vectors (o sigui, files de la matriu) és una cua (a la secció A.12 s'explica com es crea una matriu d'aquest tipus). De manera anàloga, els enters que ens donaven el primer i l'últim element de la cua i també la seva llargària són reemplaçats per vectors, on cada component fa aquesta funció per a la corresponent cua. Els seus noms són: `cues` és la matriu de cues, `ini_cua` és el vector on es guarda el principi de cada cua, `fin_cua` on es guarda el final i `lon_cua` indica la seva llargària. Finalment, el nombre de cues es guarda a `num_cues` i la longitud a la qual dimensionem els vectors cua (és a dir, la seva capacitat màxima) es posa a `max_cua`.

La rutina `ini_cues` s'encarrega de dimensionar tots els vectors necessaris per usar la resta de funcions, com també d'inicialitzar les variables globals del fitxer. La funció `posa_cua` s'encarrega de posar un `el_cua` a la cua que se li indica. La funció `treu_cua` treu un cli-

ent de la cua que especifiquem. També tenim dues funcions (`long_cua` i `cua_mes_curta`) que s'encarreguen de retornar la longitud d'una cua prefixada i de determinar quina és la cua més curta. Finalment, per completesa, hem inclòs una rutina que allibera la memòria reservada per a `ini_cues`.

El programa principal

Veiem ara el programa principal que s'encarrega de fer tota la simulació. El seu llistat en C és:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "agenda.h"
#include "cues.h"

#define     OBRIR  1
#define  ARRIBADA  2
#define   SORTIDA  3
#define    TANCAR  4

void main(void)
{
   int primer_caixer_buit(int *c, int n);
   float expo(float m, long int *ap_llavor);
   esdev e;
   el_cua c;
   float t,tmax,tserv;
   long int llavor;
   int bn,*caixa,j,nca,ntc,k;
   llavor=-1995;
   puts("nombre total de caixers?");
   scanf("%d",&ntc);
   ini_agenda(2+ntc);
   ini_cues(ntc,100);
   caixa=(int*)malloc(ntc*sizeof(int));
   if (caixa == NULL) {puts("falta memoria."); exit(1);}
   e.que=OBRIR;
   e.quan=0.0;
   posa_agenda(e);
   e.que=TANCAR;
   e.quan=720.0;
   posa_agenda(e);
   bn=0;
   nca=0;
   tmax=0.0;
   while (treu_agenda(&e) != 0)
```

```
{
   switch (e.que)
   {
      case OBRIR:
         bn=1;
         for (j=0; j<ntc; j++) caixa[j]=0;
         e.que=ARRIBADA;
         e.quan=expo(2,&llavor);
         posa_agenda(e);
         tserv=1+expo(1,&llavor);
         break;
      case ARRIBADA:
         if (bn == 1)
         {
            t=e.quan;
            k=primer_caixer_buit(caixa,ntc);
            if (k >= 0)
            {
               caixa[k]=1;
               e.quan += tserv;
               e.que=SORTIDA;
               e.on=k;
               posa_agenda(e);
            }
            else
            {
               c.tar=t;
               c.tse=tserv;
               k=cua_mes_curta();
               j=posa_cua(c,k);
               if (j == 0) {puts("error: cua massa petita"); exit(1);}
            }
            e.quan=t+expo(2,&llavor);
            e.que=ARRIBADA;
            posa_agenda(e);
            tserv=1+expo(1,&llavor);
         }
         break;
      case SORTIDA:
         ++nca;
         j=treu_cua(&c,e.on);
         if (j != 0)
 {
            t=e.quan-c.tar;
            if (t > tmax) tmax=t;
            e.quan += c.tse;
            posa_agenda(e);
```

```
    }
            else
            {
               caixa[e.on]=0;
            }
            break;
         case TANCAR:
            bn=0;
            break;
         default:
            puts("error: esdeveniment desconegut.");
      }
   }
   printf("nombre de clients atesos: %d\n",nca);
   printf("temps maxim de cua: %f\n",tmax);
}
int primer_caixer_buit(int *c, int n)
{
   int i=0;
   while (i < n)
   {
      if (c[i] == 0) return(i);
      ++i;
   }
   return(-1);
}
float expo(float m, long int *ap_llavor)
{
   float alea1(long int *ap_llavor);
   return(-m*log(alea1(ap_llavor)));
}
```

El seu funcionament és anàleg al del programa que ja hem vist per al cas d'una única cua, i per tant només comentarem les diferències principals.

En la declaració de les variables, la principal diferència és en la variable `caixa`, que ara ha passat a ser un vector. La seva funció serà la mateixa d'abans: indicar si un determinat caixer és ocupat (valor `1`) o lliure (valor `0`). Com que el nombre de caixers es llegeix per teclat, hem usat la funció `malloc` per reservar la corresponent memòria.

En tractar l'esdeveniment `ARRIBADA`, necessitem saber si hi ha un caixer lliure i quin és. Per aquest motiu hem creat una petita funció auxiliar (anomenada `primer_caixer_buit`) que explora el vector `caixa` per trobar la primera component igual a `0`. Si la troba, ens retorna quina és aquesta component. Si no hi ha cap component igual a `0` ens retorna `-1` (hem posat aquesta funció al final del programa). La resta és com segueix: si hi ha un caixer buit, posem el client al caixer i introduïm a l'agenda l'esdeveniment `SORTIDA`, que conté l'hora de sortida del caixer *i el número del caixer*. Si tots els caixers són plens, enviem el client a la cua més curta. El càlcul de l'arribada següent és igual que en el cas d'una cua.

Si l'esdeveniment és una `SORTIDA`, el camp `on` ens indica de quin caixer ha estat. Per tant, treiem un client de la cua corresponent i el passem al caixer. Si no hi ha ningú fent cua, apuntem (al vector `caixa`) que aquest caixer queda buit.

Deixem com a exercici modificar aquest programa per mesurar altres paràmetres, com la longitud màxima de les cues o el percentatge de gent que no necessita fer cua. Un altre paràmetre interessant de mesurar és el percentatge de temps d'ocupació de cada un dels caixers.

2.3.3 Més d'una cua: cues de diferents tipus

Considerem ara el cas d'un supermercat que disposa d'un cert nombre de caixers "ràpids" (caixers reservats als clients que porten un nombre reduït de productes, per exemple menys de 10) i de caixers "lents" (caixers oberts a tothom). Volem fer simulacions per comparar el rendiment de diverses configuracions de caixers ràpids i lents. El programa que acabem de veure en l'exemple anterior no serveix per a aquest cas, ja que aquí cal un tractament diferent per als dos tipus de cues.

A partir d'ara anomenarem `ncr` el nombre de caixers ràpids, i `ntc` al nombre total de caixers. Identificarem els caixers ràpids amb els números 0, 1, ..., `ncr`-1. Les modificacions que farem al programa anterior seran, essencialment, tenir en compte que els caixers ràpids són els `ncr` primers i que la resta són els lents.

Gestió de les cues

El gestor de cues serà bàsicament el mateix que hem usat abans, amb una modificació a la funció `cua_mes_curta`, que permet buscar la cua lenta més curta i la cua ràpida més curta, de manera separada. La funció modificada és la següent:

```
int cua_mes_curta(int a, int b)
{
   int j,k;
   if (a < 0) {puts("cua_mes_curta: error 1"); exit(1);}
   if (b > num_cues) {puts("cua_mes_curta: error 2"); exit(1);}
   j=max_cua;
   for (k=a; k<b; k++) if (j > lon_cua[k]) j=k;
   return(j);
}
```

La modificació consisteix a poder especificar entre quines cues es fa la cerca de la més curta. El gestor de cues no necessita cap altra modificació tret d'aquesta.

El programa principal

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "agenda.h"
#include "cues_2t.h"
```

```
#define     OBRIR  1
#define  ARRIBADA  2
#define   SORTIDA  3
#define    TANCAR  4

void main(void)
{
   int primer_caixer_buit(int *c, int a, int b);
   float expo(float m, long int *ap_llavor);
   int num_prod(long int *ap_llavor);
   esdev e;
   el_cua c;
   float t,tmcr,tmcl;
   long int llavor;
   int bn,*caixa,j,nca,ntc,ncr,ncl,np,pcr,k;
   llavor=-1995;
   puts("nombre total de caixers?");
   scanf("%d",&ntc);
   puts("nombre de caixers rapids?");
   scanf("%d",&ncr);
   ncl=ntc-ncr;
   puts("nombre de productes maxim per poder usar un caixer rapid?");
   scanf("%d",&pcr);
   ini_agenda(2+ntc);
   ini_cues(ntc,100);
   caixa=(int*)malloc(ntc*sizeof(int));
   if (caixa == NULL) {puts("falta memoria."); exit(1);}
   e.que=OBRIR;
   e.quan=0.0;
   posa_agenda(e);
   e.que=TANCAR;
   e.quan=720.0;
   posa_agenda(e);
   bn=0;
   nca=0;
   tmcr=0.0;
   tmcl=0.0;
   while (treu_agenda(&e) != 0)
   {
      switch (e.que)
      {
         case OBRIR:
            bn=1;
            for (j=0; j<ntc; j++) caixa[j]=0;
            e.que=ARRIBADA;
            e.quan=expo(2,&llavor);
            np=num_prod(&llavor);
```

```
      e.on=(np <= pcr) ? 1 : 0;
      e.ts=0.1*np+0.5+expo(0.5,&llavor);
      posa_agenda(e);
      break;
   case ARRIBADA:
      if (bn == 1)
      {
         t=e.quan;
         if (e.on == 1)
               {k=primer_caixer_buit(caixa,0,ncr);}
            else
               {k=primer_caixer_buit(caixa,ncr,ntc);}
         if (k >= 0)
         {
            caixa[k]=1;
            e.quan += e.ts;
            e.que=SORTIDA;
            e.on=k;
            posa_agenda(e);
         }
         else
         {
            c.tar=t;
            c.tse=e.ts;
            if (e.on == 1)
                  {k=cua_mes_curta(0,ncr);}
               else
                  {k=cua_mes_curta(ncr,ntc);}
            j=posa_cua(c,k);
            if (j == 0) {puts("error: cua massa petita"); exit(1);}
         }
         e.quan=t+expo(2,&llavor);
         e.que=ARRIBADA;
         np=num_prod(&llavor);
         e.on=(np <= pcr) ? 1 : 0;
         e.ts=0.1*np+0.5+expo(0.5,&llavor);
         posa_agenda(e);
      }
      break;
   case SORTIDA:
      ++nca;
      j=treu_cua(&c,e.on);
      if (j != 0)
      {
         t=e.quan-c.tar;
         if (e.on < ncr)
               {if (t > tmcr) tmcr=t;}
```

```
                  else
                     {if (t > tmcl) tmcl=t;}
               e.quan += c.tse;
               posa_agenda(e);
            }
            else
            {
               caixa[e.on]=0;
            }
            break;
         case TANCAR:
            bn=0;
            break;
         default:
            puts("error: esdeveniment desconegut.");
      }
   }
   printf("nombre de clients atesos: %d\n",nca);
   printf("temps maxim en cua rapida: %f\n",tmcr);
   printf("temps maxim en cua lenta : %f\n",tmcl);
}
int primer_caixer_buit(int *c, int a, int b)
{
   int i=a;
   while (i < b)
   {
      if (c[i] == 0) return(i);
      ++i;
   }
   return(-1);
}
float expo(float m, long int *ap_llavor)
{
   float alea1(long int *ap_llavor);
   return(-m*log(alea1(ap_llavor)));
}
int num_prod(long int *ap_llavor)
{
   float expo(float m, long int *ap_llavor);
   int n;
   n=1+(int)expo(20,ap_llavor);
   return(n);
}
```

Aquest programa és una (petita) modificació del de la secció 2.3.2. La primera diferència la tenim en la generació de l'esdeveniment `ARRIBADA`: juntament amb el temps d'arribada, generem el nombre de productes i el temps de servei. A partir del nombre de productes ja podem saber

si el client anirà a un caixer ràpid o a un de lent, i apuntem aquest fet al camp `on` (`1` vol dir cua ràpida, `0` cua lenta). Això ens servirà per decidir, quan tractem aquest esdeveniment, a quina cua posem el client.

La diferència fonamental és en el tractament de l'esdeveniment `ARRIBADA`. En primer lloc mirem si hi ha un caixer lliure. La cerca d'aquest caixer ha de tenir en compte si el client ha d'anar a un caixer ràpid o lent, i per això hem modificat la funció `primer_caixer_buit`, per poder especificar el rang de cerca. Si trobem un caixer lliure, la resta és igual que en l'exemple anterior. Si no hi ha cap caixer lliure, llavors hem de buscar la cua més curta d'entre les cues ràpides o lentes, segons correspongui. Per aquest motiu hem modificat la funció `cua_mes_curta`, per poder especificar el rang de cues per a la cerca.

Per simplificar el càlcul del nombre de productes que porta cada client, hem definit la funció `num_prod`. El seu funcionament no hauria de presentar cap problema.[5]

No hi ha més modificacions importants en la simulació. Les altres modificacions es deuen al fet que ara no mesurarem el temps màxim d'estada a les cues en general, sinó que obtindrem el temps màxim per a cues ràpides i lentes. Això ens obliga a introduir uns petits canvis en el tractament de l'esdeveniment `SORTIDA`, que és on es prenen aquestes mesures.

Nauralment, es poden prendre moltes altres mesures, com per exemple l'evolució de la longitud de les cues amb el temps, etc. Deixem com a exercici per al lector la modificació d'aquests programes per obtenir aquesta informació addicional.

Altres extensions del programa

Podriem continuar introduint canvis en el programa amb la intenció de simular sistemes concrets, pero no ho farem. El nostre propòsit ha estat introduir una metodologia, el més senzilla possible, que permeti estudiar molts problemes concrets de simulació discreta. Creiem que el lector no hauria de tenir gaires dificultats a modificar els programes aquí descrits per adaptar-los a una gran quantitat de casos.

També volem comentar l'elecció de les lleis que segueixen els processos simulats aquí (arribada de persones, nombre de productes, etc.). Les lleis usades en aquests exemples són molt simples, donat que la nostra intenció és mostrar els algorismes de simulació. Si es volen efectuar simulacions una mica realistes d'un cert procés, cal canviar aquestes lleis per d'altres que siguin molt més adequades al problema concret. Per exemple, en un problema de cues en un supermercat, el temps que passa entre arribades de clients hauria de dependre de l'hora del dia. Per altra banda, volem remarcar que els programes de simulació descrits aquí no estan afectats per l'elecció de les lleis, perquè aquestes són funcions del programa que poden ser reemplaçades sense cap dificultat. La qüestió és disposar d'una implementació en C d'aquestes lleis. Si la llei és d'un tipus molt general (Gamma, Poisson, etc.) es poden trobar implementacions en molts llocs (per exemple, vegeu la referència [19]). Si la llei és molt complexa, però, no tindrem més remei que implementar-la nosaltres mateixos.

2.3.4 Llistes enllaçades

El propòsit d'aquesta secció és explicar una manera de gestionar cues sense haver-ne d'especificar prèviament una longitud màxima. La tècnica concreta que explicarem es basa en manejar la

[5] La unitat que sumem a la part entera de l'exponencial és per, essencialment, evitar que poguem generar l'arribada d'un client sense productes. Si es vol, també es pot fer servir una llei de Poisson per generar el nombre de productes, sumant-li una unitat per la mateixa raó d'abans.

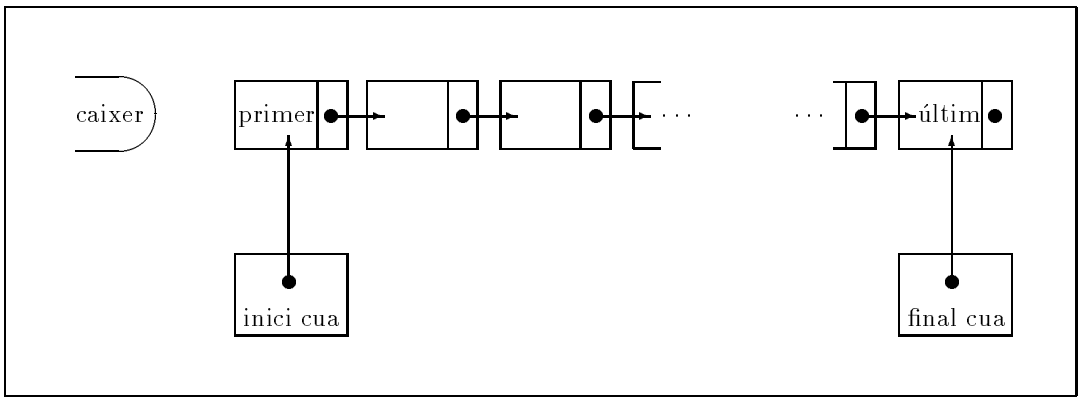

Figura 2.1: *Una cua com a llista enllaçada*

cua com una llista enllaçada.

Abans haviem tractat cada element de la cua com una estructura que contenia informació relativa a la persona que estava fent cua (l'hora d'arribada, el temps de servei i altres valors que es vulguin afegir). La cua era llavors un vector d'estructures d'aquest tipus.

Si no volem utilitzar un vector per emmagatzemar la cua, el que necessitem és un sistema alternatiu per "connectar" els elements de la cua, que no requereixi l'ús d'un vector. Necessitem saber qui va darrera de qui, qui és el primer i qui és l'últim. Amb aquest propòsit afegirem un nou camp a l'estructura `el_cua`, que ens servirà per saber qui va després de qui:

```
typedef struct ec
{
   float tar;
   float tse;
   struct ec *seg;
} el_cua;
```

El nou camp `seg` és un apuntador cap a una estructura de tipus `el_cua`, i ens servirà per apuntar el següent de la cua. Al principi de la declaració (en la primera línia) hem anomenat `ec` aquesta estructura. Això s'ha fet per poder declarar `seg` com `struct ec *` (en el moment que ens cal fer aquesta declaració el `typedef` no està acabat i per tant no podem usar el tipus `el_cua`). També declararem unes variables (globals per al fitxer on hi haurà el gestor de cues) auxiliars:

```
static el_cua *ini_cua,*fin_cua;
```

que ens apuntaran el primer i l'últim de la cua (si la cua és buida, contindran el valor `NULL`). En els gestors de cues que havíem vist abans, ja teniem dues variables enteres (amb el mateix nom que aquestes) que jugaven aquest paper. En la Figura 2.1 s'il·lustra aquesta forma de guardar la cua.

Afegir un element a la cua

Per afegir un element a la cua cal fer les operacions següents:

- Crear l'espai de memòria per guardar aquest element, fent un `malloc` d'una variable de tipus `el_cua`.
- Copiar el nou element de la cua dins aquest espai de memòria.
- Fer que el fins ara últim element de la cua apunti a aquest nou espai de memòria.
- Fer que `fin_cua` apunti també a aquest nou espai de memòria.

Treure un element de la cua

El procés per treure un element de la cua és també molt simple:

- Llegim les dades de la persona que surt de la cua.
- Fem que el primer de la cua sigui el següent del que ara se'n va.
- Destruïm (`free`) la memòria ocupada pel que se'n va.

Una implementació en C

Finalment donem una implementació en C de l'algorisme que acabem d'explicar. El donem només per al cas d'una cua, però el lector no hauria de tenir cap problema per escriure les versions per als cassos amb més d'una cua.

```
#include <stdlib.h>

typedef struct ec
{
   float tar;
   float tse;
   struct ec *seg;
} el_cua;

static el_cua *ini_cua,*fin_cua;
static int lon_cua;

void prep_cua(void)
{
   ini_cua=NULL;
   fin_cua=NULL;
   lon_cua=0;
}
int posa_cua(el_cua c)
{
   el_cua *nou;
   nou=(el_cua*)malloc(sizeof(el_cua));
```

```
   if (nou == NULL) return(0);
   *nou=c;
   nou->seg=NULL;
   if (lon_cua == 0) ini_cua=nou; else fin_cua->seg=nou;
   fin_cua=nou;
   ++lon_cua;
   return(1);
}
int treu_cua(el_cua *c)
{
   if (lon_cua == 0) return(0);
   --lon_cua;
   if (lon_cua == 0) fin_cua=NULL;
   *c=*ini_cua;
   free(ini_cua);
   ini_cua=c->seg;
   return(1);
}
int long_cua(void)
{
   return(lon_cua);
}
```

2.4 Altres tècniques de simulació

Hi ha altres metodologies de simulació discreta apart de les explicades aquí, que es poden classificar segons criteris diversos. Si ens fixem en la manera de fer avançar el temps, tenim dues possibilitats: la simulació sincrònica i l'asincrònica.

En la sincrònica es fa avançar el temps a petits intervals, i per a cada instant es mira (via generació de nombres aleatoris) si ha passat algun esdeveniment o no. Si no ha passat, es torna a avançar el temps. Si ha passat, es tracta aquest esdeveniment de la manera corresponent i s'avança el temps. El pas amb què es fa avançar el temps ha de ser l'adequat: si és massa petit, el programa estarà molta estona incrementant el temps, veient que no passa res, i tornant a incrementar. Si és massa gran, convertirem en simultanis esdeveniments que no ho eren.

La simulació asincrònica és la que hem fet servir en aquest text. Es basa a avançar el temps de manera que saltem de cada esdeveniment al següent. Per això, després de cada un d'ells hem de calcular quan passarà el pròxim.

Podeu trobar una discussió més extensa d'aquest tipus (i d'altres) de simulació a [2] o [17].

2.4.1 Paquets comercials de simulació

Actualment, hom pot trovar molts paquets de simulació en el mercat, que es poden classificar de diverses maneres. Nosaltres només en comentarem alguns i, per al lector interessat, recomanem consultar [17] o, en el seu defecte, [2].

Llenguatges de programació

Hi ha alguns llenguatges de programació específics per a simulació. La seva principal virtut és disposar d'una sintaxi adequada, a més d'incloure moltes de les funcions usades en una simulació. Possiblement, els més coneguts són SIMULA, SIMSCRIPT i ECSL.

Sistemes de diagrama de flux

A diferència dels anteriors, aquests sistemes no exigeixen (gaires) coneixements de programació per part de l'usuari. El que requereixen és una descripció o diagrama de l'activitat del sistema a simular, en un format adequat. Entre els més coneguts podem citar HOCUS i GPSS.

Generadors de programes

Produeixen el codi font d'un programa a partir d'una descripció del sistema a simular. Potser el més conegut d'aquests és CAPS. El seu *output* és un programa en ECSL que es pot editar, modificar (si es creu convenient), compilar i executar amb un compilador d'aquest llenguatge.

Biblioteques

Hi ha algunes biblioteques de rutines orientades a la simulació. Per utilitzar-les, cal escriure un programa principal amb la simulació que es vol fer. La biblioteca proporciona una col·lecció de funcions per facilitar la feina de programar, com gestors de cues, agendes, nombres aleatoris, etc. Entre elles, potser la més coneguda és la GASP, escrita en Fortran.

"Faci-ho vostè mateix"

Aquest seria l'apartat en què podriem classificar aquest text. Consisteix a escriure un conjunt de rutines adaptades al nostre cas. Un cop es tenen aquestes funcions, la programació de la simulació no és gaire complicada (de fet, estem en el cas del subapartat anterior). Els avantatges d'aquest procediment són la pràctica absència de restriccions, la portabilitat dels corresponents simuladors i la seva velocitat d'execució, entre d'altres. El principal inconvenient és que, tot i no ser gaire difícils de programar, requereixen un cert nivell d'expertesa per fer-ho.

Capítol 3 Equacions diferencials ordinàries. Integració numèrica

3.1 Conceptes bàsics de les equacions diferencials ordinàries

3.1.1 Introducció

Aquest capítol és una introducció molt senzilla a les equacions diferencials ordinàries. No des d'un punt de vista eminentment matemàtic, sinó presentant aquestes equacions com a eines que, usades convenientment i sense molta càrrega teòrica, poden donar-nos una visió del problema que volem modelitzar, en primer lloc *qualitativa*. I, posteriorment, *quantitativa*, si és que el nostre model és prou acurat.

Podríem dir que una equació diferencial ordinària és una igualtat que ens lliga una funció (real de variable real) amb les seves derivades. Dit així, és clar que aquestes equacions poden ser expressions molt complicades i difícils de tractar. Ara bé, en el nostre cas ens restringirem a aquelles en què la funció únicament apareix amb derivada primera, i aquesta, la podem aïllar. En el cas de la modelització, això no representa en principi una restricció gaire forta ja que, com veurem més tard, moltes equacions que a primer cop d'ull no semblen d'aquesta manera s'hi poden posar. I en segon lloc, moltes vegades, en el moment de modelitzar, el que tenim és la variació de les variables a modelitzar en funció dels valors d'aquestes mateixes variables, i això ja ens dóna directament el tipus d'equació, o sistema d'equacions, que tractarem.

Comencem veient tot el que hem dit amb l'exemple més senzill. Suposem que x és una magnitud que depèn del temps t. Usualment ho notarem com $x(t)$, encara que quan posem, x, ja suposarem que hi depèn implícitament i moltes vegades escriurem x en lloc de $x(t)$. La variable x és una variable *dependent*, en aquest cas del temps t, que anomenem variable *independent*. Per a cada valor de t tenim un valor concret de x (es diu que x és una funció real de variable real). Per exemple, x podria representar el nombre d'insectes d'un cert hàbitat, la densitat de població d'una certa comunitat, la concentració d'una certa substància en un medi, el preu de venda d'un cert producte o, en general, qualsevol magnitud que evoluciona amb el temps i de la qual nosaltres volem saber l'estat en un temps futur o passat.

És clar que si el que volem es veure com varia la magnitud x, el que ens cal és fixar una *llei d'evolució*. Aquesta llei l'obtindrem en base a l'experiència i a observacions del fenòmen a modelitzar. És en la formulació matemàtica d'aquesta llei que ens pot aparèixer una equació

diferencial ordinària. Per exemple, suposem que la nostra magnitud, x, compleix que en cada instant de temps, la seva velocitat de creixement és proporcional al seu valor en aquell instant. Tenint en compte que la variació de x, és a dir, la velocitat de creixement, és la seva derivada, que podem trobar notada com $x'(t)$, $\dot{x}(t)$ o dx/dt, resulta que la llei d'evolució ens dóna l'equacio diferencial,

$$\frac{dx}{dt} = ax,$$

on a és un nombre real que representa el coeficient de proporcionalitat.

En aquest cas, podem resoldre aquesta equació fàcilment ja que és de les anomenades de *variables separades*. Malgrat que no és el propòsit d'aquest llibre buscar les solucions de les equacions diferencials de manera explícita, ho farem en certs casos, a fi i efecte de mostrar algunes propietats característiques.

Separant variables, la qual cosa essencialment vol dir deixar les x a un costat de la igualtat i les t a l'altra, ens queda

$$\frac{dx}{x} = adt.$$

Ara primitivitzemt els dos costats de l'equació per obtenir, $\log x = at + C$ on C és una constant arbitrària. Finalment, prenent exponencials als dos costats de l'equació, resulta (notem explícitament la dependència de x en t),

$$x(t) = Ke^{at}, \tag{3.1}$$

on K és una constant arbitrària. Això es diu *solució general* de l'equació diferencial.

La primera de les coses que mostrem és que (3.1) representen totes les solucions possibles i d'aquí el nom de solució general. Per això suposem que $v(t)$ sigui una solució qualsevol. És a dir, suposem que $v(t)$ compleix $v'(t) = av(t))$. Si calculem la derivada de $v(t)e^{-at}$, tenim

$$\frac{d}{dt}(v(t)e^{-at}) = v'(t)e^{-at} - av(t)e^{-at}.$$

Usant ara que $v(t)$ és solució resulta que l'expressió anterior val zero. Resulta doncs que la derivada de $v(t)e^{-at}$ és zero i per tant aquesta funció ha de ser constant:

$$v(t)e^{-at} = K,$$

el que ens porta a $v(t) = Ke^{at}$, que és el que volíem veure.

Vegem ara que la constant K que ens apareix a la solució general ve determinada en fixar les *condicions inicials*. Quan fixem una condició inicial, i per tant determinem un valor de K, obtenim una *solució particular*, també anomenada més endevant *órbita*. En el nostre cas, si posem per exemple el temps a zero, $t = 0$, obtenim

$$x(0) = Ke^0 = K.$$

Per tant K representa el valor de $x(0)$ que usualment notarem per x_0.

Un altre valor important del nostre primer model és la constant a. A aquests valors que apareixeran en els nostres models els direm *paràmetres*.

Òbviament, quan ens posem a estudiar un model concret, els paràmetres prenen uns certs valors numèrics fixats. Però moltes vegades és interessant veure què passa a les solucions del

nostre model quan canviem els paràmetres. Per exemple, en el nostre cas, la solució general del model és $x(t) = Ke^{at}$, per tant les *gràfiques* són essencialment exponencials. Ara bé, el seu comportament presenta un substancial canvi *qualitatiu* segons si $a < 0$, $a = 0$, o de si $a > 0$, tal com podem veure a la figura 3.1.

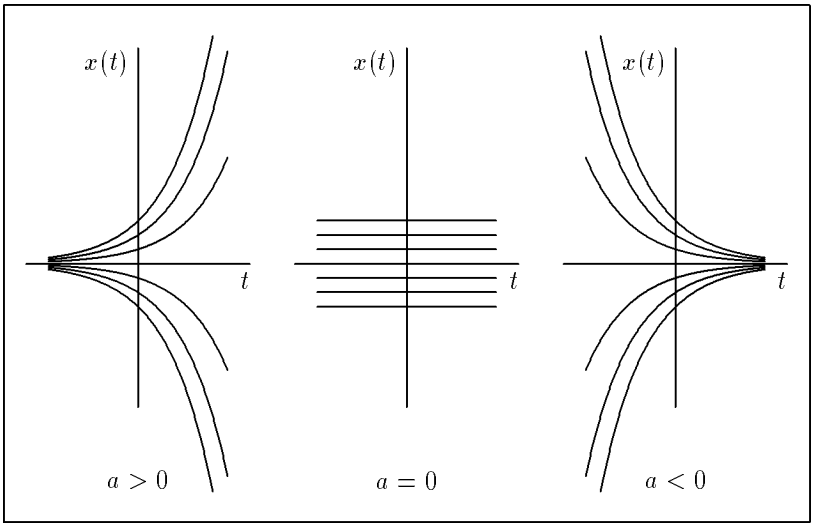

Figura 3.1: *Gràfiques de la funció* $x(t) = Ke^{at}$ *depenent del valor de* a *i per a diferents valors de* K*.*

Notem que, si bé per a diferents valors de a positius les gràfiques són diferents (amb més o menys creixement) i per tant variarien *quantitativament*, si no posem escales al dibuix no podem distingir entre les gràfiques per a un cert valor de a o un altre. En canvi en passar d'un valor de a negatiu a un de positiu, o a l'inrevés, s'observa un *canvi qualitatiu* important que fa les gràfiques totalment diferents, ja que els límits quan la t tendeix a infinit o a menys infinit canvien.

Quan es té un cert valor del paràmetre, o paràmetres, pel qual el model mostra diferents comportaments qualitatius si es prenen valors del paràmetre en un entorn d'aquest valor concret, direm que aquell valor del paràmetre és un *valor de bifurcació*. Per al model que estem analitzant tenim que $a = 0$ és un valor de bifurcació, ja que a banda i banda d'ell s'observen comportaments qualitatius diferents.

Els valors de bifurcació són molt importants en l'estudi de models. Normalment els paràmetres dels nostres models estaran determinats sobre la base de mesures experimentals susceptibles d'errors i, per tant, poques vegades o mai disposarem del valor exacte. És important, llavors, veure què passa amb l'evolució del nostre model, no només per als valors concrets dels paràmetres calculats, sinó també dins del seu marge d'error, a fi i efecte que el comportament qualitatiu donat pel model sigui "robust" enfront de la seva evolució real. Si els paràmetres calculats són propers a un valor de bifurcació, segurament el model no serà gaire fiable, ja

que la realitat pot tenir un paràmetre lleugerament distint que doni comportaments totalment diferents. Recíprocament, donat un model per exemple ecològic, podem suposar que l'acció humana el pertorba, i com a resultat canvien els valors dels paràmetres que el governen. Una bona qüestió és conèixer fins on es pot arribar a pertorbar, de manera que conservi les seves propietats qualitatives actuals, i de manera que l'ecosistema no passi per un valor de bifurcació, el qual pot representar l'evolució envers una catràstrofe ecològica.

Si malgrat tot succeeix una bifurcació, l'atra qüestió que ens interessa és saber si tornaríem a l'estat o evolució inicial en cas de suprimir la pertorbació que ha provocat el canvi qualitatiu.

3.1.2 Sistemes d'equacions diferencials lineals

Continuem la nostra marxa pel món dels models regits per equacions diferencials ordinàries, i veiem què és un *sistema d'equacions* d'aquest tipus.

Quan modelitzem, moltes vegades ens trobarem que ens interessa l'evolució de més d'una variable dependent. Llavors ens caldrà escriure diverses equacions, i el seu conjunt l'anomenem sistema.

Per exemple en lloc d'una magnitud que varia proporcionalment a la seva presència, com és el cas del primer model que hem vist, en podem tenir dues, x_1 i x_2, que variïn de la mateixa manera, cadascuna amb la seva constant de proporcionalitat.

$$\begin{cases} \dot{x}_1 &= a_1 x_1, \\ \dot{x}_2 &= a_2 x_2. \end{cases}$$

Aquest cas és clarament un dels més senzills que ens podem trobar. Veiem que la primera equació només conté la funció x_1, i la segona equació només conté la funció x_2. Direm que en aquest cas el sistema està *desacoblat*. Això vol dir que podem resoldre cada equació per separat, de la mateixa manera que ho fèiem per al model inicial, i obtenir

$$x_1(t) = K_1 e^{a_1 t}, \quad x_2(t) = K_2 e^{a_2 t},$$

on aquesta vegada tenim K_1 i K_2, dues constants arbitràries que vénen determinades en fixar les condicions inicials.

Anem però a mirar les solucions del nostre sistema des d'un punt de vista més geomètric que ens facilitarà la seva visió qualitativa. Podem considerar que la parella de solucions $(x_1(t), x_2(t))$ forma una corba en el pla, *parametritzada* per la seva variable independent, que en aquest cas és el temps, t. Llavors, partint d'un punt del pla per exemple per a $t = 0$[1], per a cada valor de t tenim una parella de valors (x_1, x_2), que si els anem representant en el pla resulten una corba que passa pel punt inicial. A cada una de les corbes possibles que ens poden sortir en anar canviant les condicions inicials els direm *òrbites* i al dibuix qualitatiu de totes elles, *retrat de fase*. Comentarem molt més aquest fet.

Usarem en primer lloc la notació vectorial, $x(t)$, per referir-nos al vector, $(x_1(t), x_2(t))$, posat en columna. Aleshores, el sistema de dues equacions que tenim el podem notar com,

$$x' = Ax,$$

on A és una matriu 2×2, en aquest cas diagonal ($A = diag(a_1, a_2)$).

[1] La qual cosa determina K_1 i K_2.

Al costat dret d'aquesta equació vectorial, és a dir al terme Ax, li direm *camp vectorial* o senzillament *camp*, ja que el podem interpretar com una aplicació dins el pla, en què a cada punt x li fem correspondre el vector Ax. Així, per exemple, si en el nostre cas $a_1 = 2$ i $a_2 = -1/2$ tenim que al punt $(1,1)$ li fem correspondre el vector $(2,-1/2)$, tal com representem a la figura (3.2).

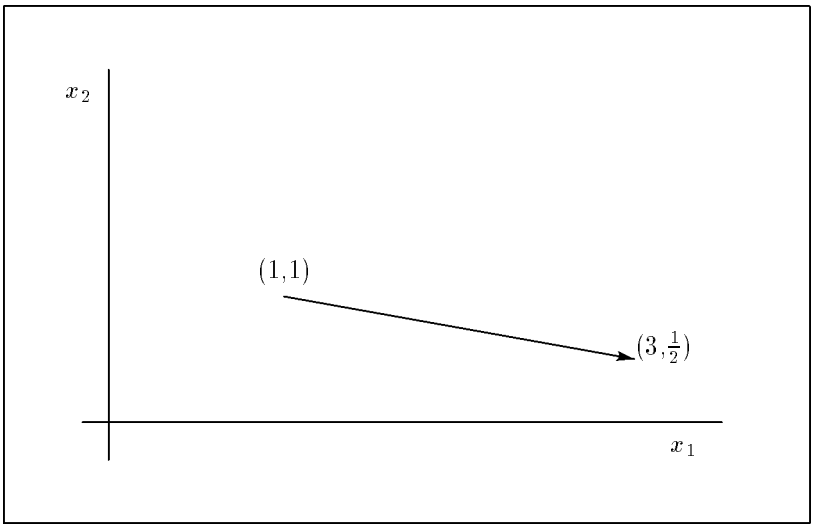

Figura 3.2: *Vector* $(2,-\frac{1}{2})$ *amb base el punt* $(1,1)$. *El seu extrem resulta el punt* $(3,\frac{1}{2})$.

I és clar que això es pot fer per a cada un dels punts del pla, i obtenir el dibuix equemàtic de la figura 3.3.

Veiem doncs que el nom de camp vectorial queda totalment justificat pel fet que cada punt del pla té el seu vector associat. Ara bé, l'equació $x' = Ax$ ens diu que a cada punt de la corba, $x(t)$, el vector tangent, $x'(t)$ (o vector velocitat), ha de ser precisament el vector Ax que hem dibuixat basat en aquell punt. D'aquí deduïm la interpretació geomètrica del que significa trobar una òrbita solució del sistema d'equacions diferencials inicial: situats inicialment en un punt del pla, l'òrbita solució del sistema, amb la condició inicial donada, és la corba $x(t) = (x_1(t), x_2(t))$ en la qual sortint del lloc inicial triat, en cada un dels punts de la trajectòria, el vector velocitat $x'(t)$ coincideix amb el vector del camp.

I el dibuix del conjunt de totes les trajectòries, que de fet ja s'endevina en dibuixar el camp, s'anomena retrat de fase (vegeu la figura 3.4).

Les òrbites que formen el retrat de fase no es poden creuar transversalment, ja que, de ser així, en el punt de creuament hi hauria dos vectors tangents diferents (un per a cada òrbita) i el camp només en dóna un. D'altra banda només direm que la impossibilitat d'existència d'òrbites tangents, la qual cosa assegura la unicitat de la trajectòria, ve donada pel conegut *teorema d'existència i unicitat de solucions* de les equacions diferencials que és cert sota hipòtesis de regularitat del camp. El lector interessat en aquest tema pot consultar qualsevol llibre d'introducció a les equacions diferencials.

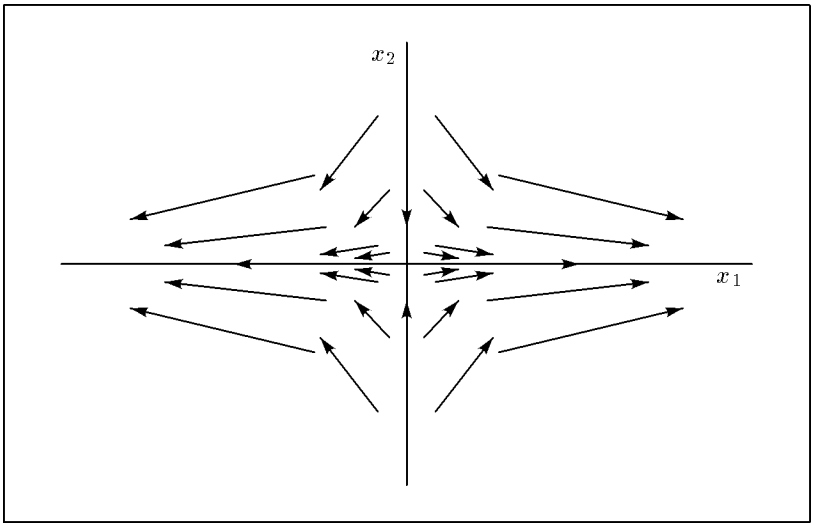

Figura 3.3: *Camp vectorial al pla.*

Un punt especial que hem d'esmentar dins el nostre retrat de fase és el $(0,0)$, ja que damunt d'ell el vector del camp val $(0,0)$. Els punts on el camp s'anul·la els direm *punts d'equilibri*. L'òrbita del punt d'equilibri sempre roman damunt del punt d'equilibri per a tot temps, t. Podem pensar que en un punt d'equilibri el vector velocitat és nul i per tant no hi ha moviment. Més endavant veurem que els punts d'equilibri poden ser *estables* o *inestables*.

Notem també que el model pot ser considerat com un *sistema dinàmic*, en el sentit que, si en un cert instant de temps considerem un subconjunt U del pla, cada un dels punts d'aquest subconjunt segueix la seva trajectòria en variar el temps. Al cap d'un cert temps resulta un altre subconjunt V que podem anomenar *el transportat de U pel flux* del sistema dinàmic o model en qüestió. És a dir que l'evolució del conjunt d'órbites (el flux del sistema) ens dóna el que anomenem sistema dinàmic. Si el conjunt U es transporta un cert temps t pel flux, de manera que resulta un altre conjunt V, s'acostuma a indicar com $V = \Phi_t(U)$.

Tractem ara el model

$$\begin{cases} \dot{x}_1 &= 8x_1 - 10x_2, \\ \dot{x}_2 &= 5x_1 - 7x_2, \end{cases}$$

que és molt semblant al de l'exemple anterior. El camp també és del tipus Ax, però ara A no és una matriu diagonal sinó que és

$$A = \begin{pmatrix} 8 & -10 \\ 5 & -7 \end{pmatrix}.$$

En aquest cas el sistema està *acoblat*, ja que hi ha equacions que lliguen més d'una variable dependent.

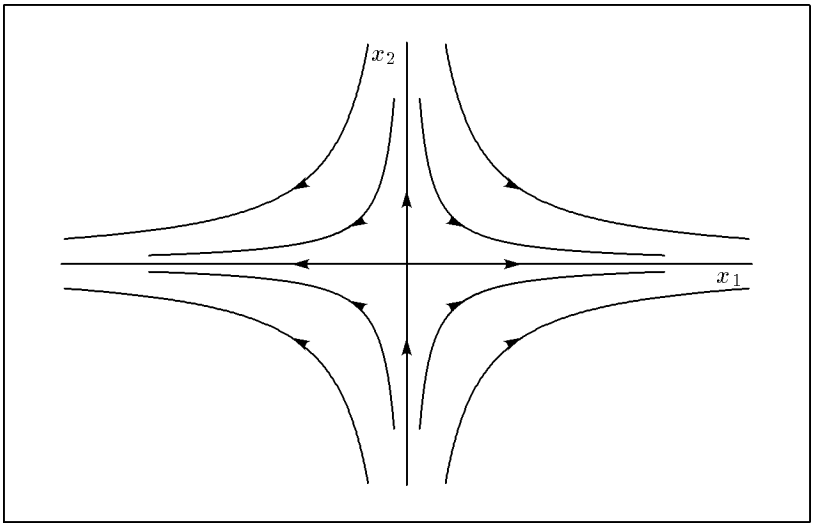

Figura 3.4: *Retrat de fase corresponent a la figura 3.3.*

Usarem una tècnica que s'aplica a camps del tipus Ax i que, en molts casos, desacoblarà el sistema dinàmic. Al mateix temps veurem com es pot usar el que s'anomena *canvi de variables* per veure el model en unes altres coordenades, que de vegades ens pot proporcionar un punt de vista molt més adient a les nostres necessitats. El lector familiaritzat amb l'àlgebra lineal hi reconeixerà ràpidament la tècnica de diagonalització de matrius i canvi de base, per bé que aplicada d'una manera "menys elegant".

Comencem calculant els *valors propis* de la matriu A. Són les solucions, λ, de l'equació

$$\det(A - \lambda I) = 0,$$

on I representa la matriu identitat. Calculem doncs i igualem a zero el determinant,

$$\begin{vmatrix} 8-\lambda & -10 \\ 5 & -7-\lambda \end{vmatrix} = (8-\lambda)(-7-\lambda) + 50 = 0,$$

que ens dóna l'equació

$$\lambda^2 - \lambda - 6 = 0,$$

amb solucions (és a dir, els valors propis de A) $\lambda_1 = -2$, $\lambda_2 = 3$.

Cada valor propi té associada una direcció donada per un *vector propi*. Aquest es calcula pel valor propi λ, agafant una solució qualsevol, v, del sistema indeterminat

$$(A - \lambda I)v = 0.$$

Així per exemple per al valor propi $\lambda = \lambda_1 = -2$, el vector propi associat, (v_1, v_2), ha de ser solució del sistema d'equacions lineals

$$\begin{aligned} 10v_1 - 10v_2 &= 0, \\ 5v_1 - 5v_2 &= 0. \end{aligned}$$

Agafem per exemple el vector $(1,1)$ com a vector propi associat al valor propi -2. Procedint de manera anàloga agafem el vector $(2,1)$ com a vector propi associat al valor propi 3.

Un cop tenim els valors i vectors propis, fem un *canvi de variable*. Passarem de la variable vectorial, $x = (x_1, x_2)$, a una de nova, que notarem $u = (u_1, u_2)$, per mitjà d'un lligam que les relaciona.

Un bon canvi de variable per a sistemes en què el camp és del tipus Ax és

$$x = Bu,$$

on les coordenades "velles" es posen igual a una matriu, B, multiplicada per les "noves", i on aquesta matriu, B, està formada pels vectors propis posats en columnes:

$$B = \begin{pmatrix} 1 & 2 \\ 1 & 1 \end{pmatrix}.$$

És a dir, tenim el canvi

$$\begin{aligned} x_1 &= u_1 + 2u_2, \\ x_2 &= u_1 + u_2. \end{aligned}$$

Aïllar de la relació anterior les u en termes de les x, és el que s'anomena "invertir el canvi" i resulta,

$$\begin{aligned} u_1 &= -x_1 + 2x_2, \\ u_2 &= x_1 - x_2. \end{aligned}$$

Vegem ara com queda el model inicial escrit en les noves variables u_1 i u_2. Com que, igual que $x = x(t)$, la u també depèn del temps, $u = u(t)$, comencem derivant la primera equació del canvi invers i substituïm-hi els valors de $\dot{x}_1$ i $\dot{x}_2$ per les expressions corresponents donades pel camp:

$$\dot{u}_1 = -\dot{x}_1 + 2\dot{x}_2 = -(8x_1 - 10x_2) + 2(5x_1 - 7x_2),$$

cosa que resulta, $\dot{u}_1 = 2x_1 - 4x_2$. Posant ara x_1 i x_2 en termes de u_1 i u_2 mitjançant el canvi de variables queda

$$\dot{u}_1 = 2x_1 - 4x_2 = 2(u_1 + 2u_2) - 4(u_1 + u_2) = -2u_1.$$

De manera anàloga procedim amb $\dot{u}_2$ per a obtenir fàcilment $\dot{u}_2 = 3u_2$. És a dir, en les noves coordenades, u_1 i u_2, el sistema dinàmic inicial es veu com

$$\begin{cases} \dot{u}_1 &= -2u_1, \\ \dot{u}_2 &= 3u_2, \end{cases}$$

i per tant està desacoblat. Observem el fet, no casual, que en aquestes coordenades el sistema dinàmic té associada una matriu diagonal que està formada pels valors propis.

De la mateixa manera que en el segon exemple, podem fer el retrat de fase. Primer mirem la direcció del camp damunt dels eixos i després fora d'ells tenint en compte que, per estar

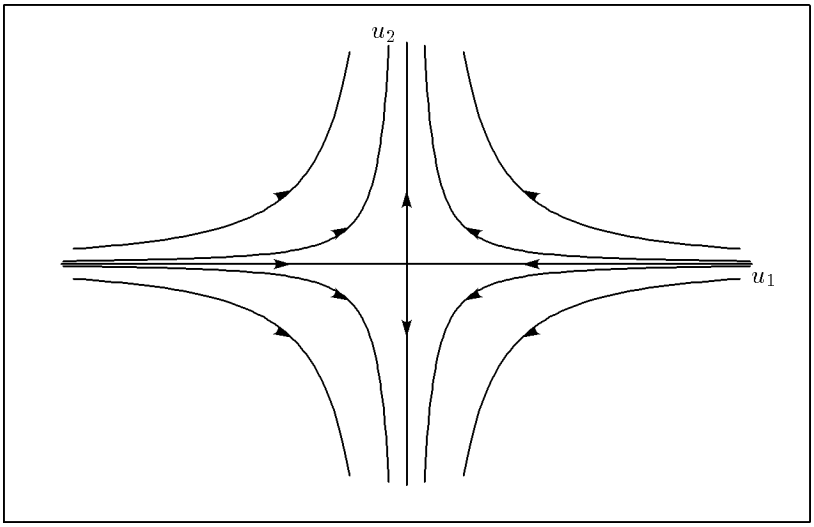

Figura 3.5: *Retrat de fase del sistema desacoblat corresponent a les coordenades* (u_1, u_2).

desacoblat, el moviment és composició del moviment horitzontal donat per $\dot{u}_1$ més el vertical donat per $\dot{u}_2$. Podem veure el resultat a la figura 3.5.

Seguidament veurem el retrat de fase d'aquest sistema dinàmic però en les coordenades inicials x_1, x_2. Per fer això hem de mirar el retrat de fase anterior via el canvi de variables.

Comencem mirant els eixos. Si agafem l'eix u_1, que té per equació $u_2 = 0$, veiem que amb el canvi de variable invers resulta

$$0 = x_1 - x_2,$$

és a dir, l'eix u_1 es transforma en la recta $x_2 = x_1$. De la mateixa manera, l'eix u_2 (d'equació $u_1 = 0$) es transforma en la recta $x_2 = x_1/2$.

Notem també el fet que els vectors directors d'aquestes rectes coincideixen amb els vectors propis calculats abans.

Representant el camp damunt d'aquestes rectes de la mateixa manera que ho està en les seves corresponents en el pla u_1-u_2 (els eixos). O bé tenint en compte la següent regla, d'altra banda fàcil de deduïr: *Per a un model del tipus* $\dot{x} = Ax$ *(tipus lineal), el camp avaluat en qualsevol punt d'una recta que passi per l'origen i tingui com a vector director un vector propi, està contingut en la mateixa recta i s'allunya o apropa al punt d'equilibri depenent de si el valor propi associat és positiu o negatiu respectivament.*

El fet que en els punts d'aquestes rectes el camp hi estigui contingut implica que aquestes rectes són òrbites, o el que és el mateix, es diu que són *rectes invariants*.

Un cop dibuixat el flux damunt de les rectes invariants donades pels vectors propis, una simple inspecció del dibuix ens permet dibuixar-lo en els altres llocs. Obtenim finalment el retrat de fase donat en la figura 3.6.

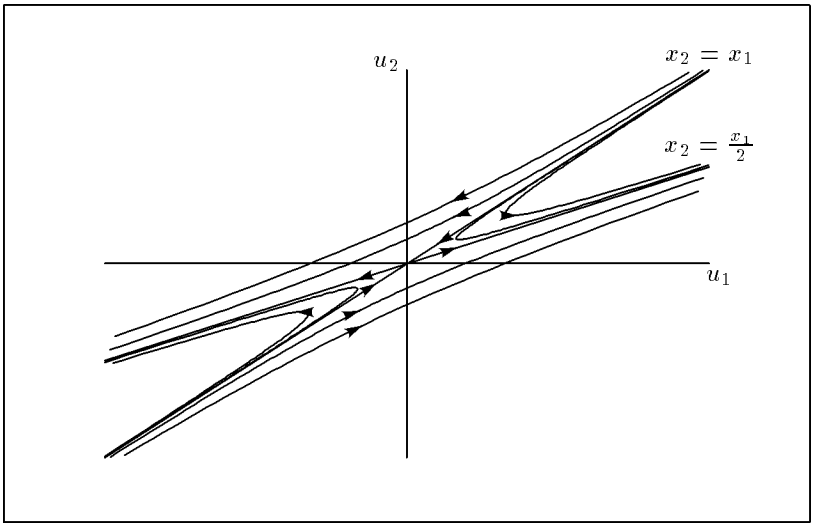

Figura 3.6: *Retrat de fase del sistema acoblat corresponent a les coordenades* (x_1, x_2). *Els vectors directors de les rectes* $x_2 = x_1$ *i* $x_2 = \frac{1}{2}x_1$ *es corresponen amb els vectors propis de la matriu del sistema.*

Finalment buscarem la parametrització de les òrbites en funció del temps, de la mateixa manera que ho hem fet per al primer sistema desacoblat estudiat com a exemple preliminar.

Com sabem, la solució del nostre sistema desacoblat és

$$\begin{aligned} u_1(t) &= K_1 e^{-2t}, \\ u_2(t) &= K_2 e^{3t}, \end{aligned}$$

on K_1 i K_2 coincideixen amb el valors de $u_1(0)$ i $u_2(0)$ respectivament. Siguin ara C_1 i C_2 els valors respectius de $x_1(0)$ i $x_2(0)$, via la relació del canvi de variable invers avaluat per a $t = 0$ tenim que

$$\begin{aligned} K_1 &= -C_1 + 2C_2, \\ K_2 &= C_1 - C_2, \end{aligned}$$

el que ens dóna

$$\begin{aligned} u_1(t) &= (-C_1 + 2C_2)e^{-2t}, \\ u_2(t) &= (C_1 - C_2)e^{3t}. \end{aligned}$$

Introduint això al canvi de variable resulta

$$\begin{aligned} x_1(t) &= (-C_1 + 2C_2)e^{-2t} + 2(C_1 - C_2)e^{3t}, \\ x_2(t) &= (-C_1 + 2C_2)e^{-2t} + (C_1 - C_2)e^{3t}, \end{aligned}$$

que és la parametrització de les òrbites, $(x_1(t), x_2(t)$, del retrat de fase amb $(x_1(0), x_2(0)) = (C_1, C_2)$.

Un bon exercici és usar un PC per dibuixar[2] aquestes òrbites per a diferents valors de C_1 i C_2, i veure, d'aquesta manera, com es va obtenint el retrat de fase de la figura 3.6.

L'objectiu d'aquest llibre, però, no és donar mètodes de com obtenir de manera explícita les trajectòries parametritzades en funció del temps, ja que moltes vegades és impossible, o bé en resulten expressions molt complicades i difícils de tractar. El que farem és proporcionar el que s'anomenen *mètodes d'integració*, que consisteixen en algorismes que, donat un punt inicial, segueixen l'òrbita que passa per aquest punt de manera que es pot anar tabulant. Si es vol veure la trajectòria que es segueix només cal anar dibuixant la taula de valors obtinguts.

Observem algunes coses d'aquests retrats de fase anomenats *selles*. El punt d'equilibri, $(0, 0)$, constitueix per si sol una òrbita, que s'obté agafant $C_1 = C_2 = 0$ en unes coordenades, o bé $K_1 = K_2 = 0$ en les altres.

A aquest punt d'equilibri hi van a parar asimptòticament, quan t tendeix a infinit, dues òrbites. Les dues semirectes que són damunt la recta que té per vector director el vector propi associat al valor propi negatiu. De la mateixa manera en surten dues òrbites (tendeixen al punt quan t tendeix a menys infinit) que són les dues semirectes de damunt la recta associada al vector propi de valor propi positiu.

El fet d'haver-hi direccions d'escapament fa que el punt d'equilibri sigui *inestable*, això vol dir que si un estat ve representat pel punt d'equilibri, qualsevol pertorbació que ens desplaci d'aquest estat ens farà entrar en una òrbita que a la llarga s'allunyarà d'aquest punt d'equilibri.

D'altra banda, les òrbites constituïdes per les quatre semirectes reben el nom de *separatrius* ja que separen comportaments totalment diferents a la llarga. Així l'òrbita que comença al punt $C_1 = C_2 = 1$, que és damunt d'una separatriu, va a parar asimptòticament al punt d'equilibri. Però condicions inicials arbitràriament properes a la $C_1 = C_2 = 1$, agafades a banda i banda de la separatriu, tenen a la llarga comportaments totalment diferents. Així per exemple, si el punt de partida s'agafa lleugerament per sobre la separatriu, el valor de x_1 damunt d'aquesta òrbita tendirà cap a menys infinit. Mentre que si el punt de partida s'agafa lleugerament per sota de la separatriu, a la llarga el valor de x_1 tendirà a més infinit.

Són fets com aquests, el de l'estabilitat, les separatrius i possibles bifurcacions, d'entre d'altres, que són importants a l'hora d'estudiar un model, ja que l'evolució del sistema pot ser totalment diferent canviant poca cosa de l'estat inicial.

Tot "l'utillatge" necessari per estudiar els models des d'aquests punts de vista el proporciona la *Teoria qualitativa de les equacions diferencials*. Aquesta segona part del llibre vol donar mètodes, que de fet són quantitatius, a fi i efecte de poder simular fàcilment l'evolució d'un model partint d'estats inicials. Es poden obtenir així, resultats i conclusions empíriques sense necessitat d'usar eines teòriques més pròpies del món matemàtic.

3.1.3 Models més complexos

Hem tractat models pels quals el camp en el punt x és del tipus Ax, és a dir, un camp lineal i en dimensió 2. És clar, però, que la funció vectorial que ens dóna el camp pot ser força més

[2] A l'apèndix B s'indica com es poden obtenir dibuixos treballant amb llenguatge C.

general. Aleshores els nostres models seran

$$\dot{x} = f(x),$$

on f pot tenir n components, $f(x) = (f_1(x), f_2(x), \ldots, f_n(x))$ i el vector x també, $x(t) = (x_1(t), x_2(t), \ldots, x_n(t))$.

D'altra banda pot resultar que la funció f depengui explícitament de la variable independent (en aquest cas del temps, t) i tinguem un model del tipus

$$\dot{x} = f(t, x)$$

. Direm llavors que el camp és *no autònom* en contraposició als *camps autònoms* que són els que no depenen explícitament del temps.

Així tenim que el model

$$\begin{cases} \dot{x}_1 &= x_1 + x_2 \sin x_3, \\ \dot{x}_2 &= x_2 + x_3 \cos x_1, \\ \dot{x}_3 &= x_3 + x_1 \sin x_2, \end{cases}$$

és un model no lineal autònom en dimensió 3, mentre que

$$\begin{cases} \dot{x}_1 &= x_1 + t \sin x_2, \\ \dot{x}_2 &= x_2 + \cos x_1, \end{cases}$$

és un model no lineal i no autònom en dimensió 2.

No podem fer el retrat de fase de sistemes no autònoms tret que hi afegim la component temporal, ja que el camp canvia en cada instant de temps. Ara bé, podem "autonomitzar" el sistema no autònom afegint-hi l'equació, $\dot{t} = 1$, i fer el retrat de fase en una dimensió més, considerant que el temps t és una altra variable lligada amb la relació anterior.

Finalment notem que, per al cas de dimensió 1, en el sistema $\dot{x} = ax$, que és el primer exemple que hem estudiat, els retrats de fase que s'obtenen estan representats en la figura 3.7. Cal comparar els dibuixos amb la figura 3.1, que representa les gràfiques de les solucions, i notar que els retrats de fase s'obtenen, de fet, en projectar les gràfiques de les solucions en l'eix x, a mesura que el temps avança. El retrat de fase dóna per tant una visió qualitativa, i ens diu cap on evoluciona la trajectòria. Però no ens dóna informació de la velocitat damunt de la trajectòria. Així per exemple, en els retrats de fase en dimensió 2 que hem vist, les separatrius de la sella tendeixen al punt d'equilibri; però enlloc no apareix que hi tendeixen quan el temps tendeix a infinit.

3.2 Mètodes d'integració numèrica

3.2.1 La idea essencial dels mètodes d'integració

En aquesta secció donarem alguns algorismes que ens permetran, un cop posicionats en un punt qualsevol del retrat de fase, seguir la trajectòria i tenir-la parametritzada pel temps.

Suposem que, després de fer unes certes hipòtesis, hem aconseguit escriure el nostre model en termes d'un sistema d'equacions diferencials ordinàries. Com que l'escriptura de models a partir d'hipòtesis inicials és un tema que no veurem fins en un capítol posterior, imaginem per

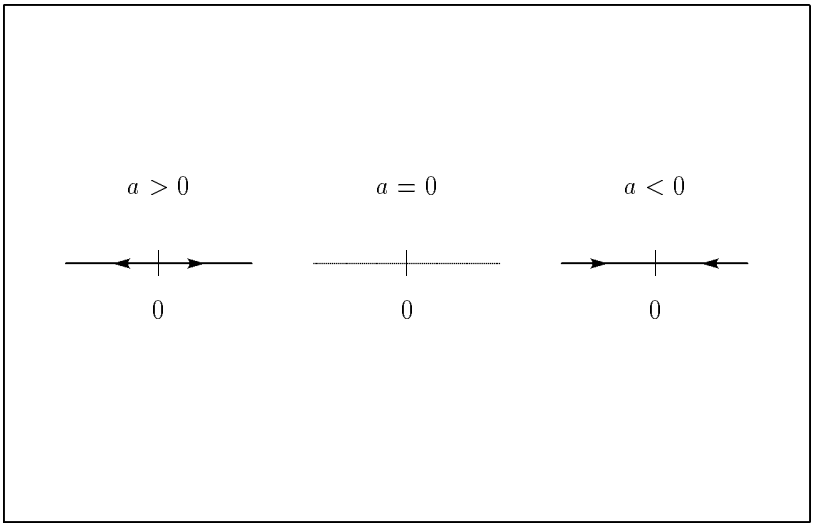

Figura 3.7: *Retrats de fase del sistema $\dot{x} = ax$ en dimensió 1 segons els diferents valors de a.*

exemple que hem obtingut el sistema lineal d'equacions diferencials ordinàries acoblades tractat en la secció anterior:

$$\begin{cases} \dot{x}_1 &= 8x_1 - 10x_2, \\ \dot{x}_2 &= 5x_1 - 7x_2. \end{cases}$$

Aquest sistema via un canvi de variable, ha estat possible desacoblar-lo i hem aconseguit escriure les trajectòries, parametritzades pel temps en termes de combinacions lineals d'exponencials. Hem obtingut una *representació analítica* de les solucions. Això ha estat possible perquè el sistema és lineal i per tant molt senzill.

Ara bé, la majoria de vegades això no és possible i potser tot i essent possible la representació analítica de les solucions no és operativa, ja que ens surten funcions no elementals molt complicades.

En aquesta secció tractem els mètodes que, un cop triat el punt de sortida, segueixen la trajectòria i donen una taula de valors per a certs valors del temps. De manera que si unim els punts obtinguts, per exemple amb rectes (o interpolant amb funcions convenients, per als lectors que coneguin algorismes d'interpolació), obtindrem una bona representació de la trajectoria seguida; d'aquesta manera podrem dibuixar el retrat de fase allà on ens convingui i obtenir a més una visió quantitativa, ja que a més sabrem per a quin valor del temps ens trobem en cada punt.

Tot això està molt bé, però només és la idea essencial; la realitat és una mica diferent. Els algorismes que segueixen la trajectòria són només aproximats, i per tant pot passar que comencem seguint una trajectòria, però que la taula de valors obtinguda, s'aparti a poc a poc de l'òrbita que hem començat a seguir. Per exemple, suposem que estem seguint una separatriu d'una sella en direcció al punt d'equilibri. Si tot fos exacte, a mesura que avança el temps hem d'estar cada vegada més a prop del punt d'equilibri. Ara bé, com que l'algorisme és aproximat

(i a més hi ha errors de càlcul produïts pel fet de treballar amb un nombre finit de decimals), pot passar que en un cert moment saltem fora de la separatriu i aleshores a la llarga sabem que es poden produir comportaments molt diferents, depenent de si hem anat a parar a un costat o a l'altre. Per motius com aquests cal anar amb certa cura a l'hora d'utilitzar aquests algorismes; i un bon test que es pot realitzar és veure si, sortint de punts bastant propers, al cap d'un cert temps continuem trobant-nos en punts propers. És el que s'anomena *estabilitat de l'òrbita*.

Els algorismes que presentarem en aquesta secció se'ls anomena *mètodes d'un pas*, ja que s'obtè el valor d'un nou punt de l'òrbita únicament segons el (darrer) punt en el qual estavem en l'instant anterior. Per als lectors interessats en el tema direm que hi ha *mètodes multipàs* que avancen per la trajectòria usant més d'un punt dels que s'han obtingut amb anterioritat.

La construcció dels diferents mètodes d'integració que donarem i les seves propietats teòriques quant a precisió, la farem tot seguit. Calen però lleugeres nocions de càlcul diferencial en una i diverses variables. Si el lector no té "frescos" o no usa amb soltura algun d'aquests conceptes, com poden ser les sèries de Taylor, podrà seguir perfectament tota la resta del llibre, i obtenir resultats dels seus models, usant únicament els algorismes que aquí donarem assenyalats, i que anirem presentant seguint una idea conductora.

La formulació matemàtica del que volem fer ve donada per la integració numèrica de l'anomenat *problema de Cauchy*:

$$\begin{cases} x'(t) &= f(t, x(t)), \\ x(a) &= x_0, \end{cases}$$

on x representa un vector de m components. Així $x(t)$ representa m funcions, $x_1(t), \ldots, x_m(t)$, i $x'(t)$ el mateix que $\dot{x}(t)$, és a dir, la seva derivada. A la funció $f : [a, b] \times \mathbb{R}^m \to \mathbb{R}^m$ se li diu, com ja sabem, el *camp* i el suposem prou regular. A la condició $x(a) = x_0$, se li diu *condició inicial* i indica que, a l'instant $t = a$, el vector x ha de valer x_0, que és un valor fixat per nosaltres. Per aquest fet el problema de Cauchy també se l'anomena *problema de valor inicial*.

Imaginem que volem calcular (de fet ja sabem que serà aproximar) $x(t)$ per a t a l'interval $[a, b]$. La idea més simple consisteix a dividir $[a, b]$ en N trocets de longitud $h = \frac{b-a}{N}$ i calcular $x(t_n)$, on $t_n = a + nh$ des de $n = 0$ fins a $n = N$.

Suposem el cas $m = 1$; fent això obtenim la funció, $x(t)$, tabulada en $N + 1$ punts (vegeu la figura 3.8).

Durant aquest procés notarem amb $x(t_n)$ el valor real de $x(t)$ per a $t = t_n$ i direm x_n al valor aproximat que donarà el nostre algorisme per a $x(t_n)$.

La condició inicial del problema de Cauchy ens diu que, en l'instant inicial, el valor real $x(a) = x(t_0)$ coincideix amb x_0.

Resumint el que fem a continuació en aquesta secció, direm que primer de tot desenvolupem el mètode d'integració més senzill anomenat *mètode d'Euler*. Aquest mètode és molt fàcil d'implementació en un ordinador, però no és gaire precís.

De fet, els mètodes d'integració donen un valor, x_k, més proper a $x(t_k)$ en reduir el pas, h; és a dir en arribar al mateix t_k però havent fet més passos, ja que x_{n+1} s'obté a partir de x_n i l'error que s'introdueix depèn essencialment del pas h, com més gran més error.

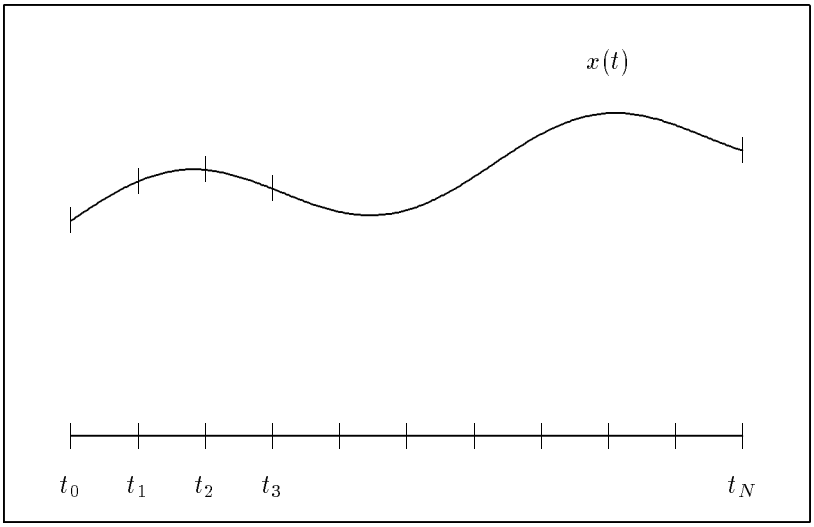

Figura 3.8: *Punts on tabularem la funció $x(t)$ que busquem com a solució d'un problema de Cauchy.*

El mètode d'Euler necessita normalment reduir molt el pas per obtenir bons resultats teòrics. Ara bé, quan es redueix molt el pas, a part que es comença a alentir la integració, s'acumulen errors numèrics a causa de la gran quantitat d'operacions que cal fer. Per aquest fet, al mètode d'Euler no se li pot demanar molta precisió i busquem altres mètodes que, fent servir un pas més gran donin, resultats numèrics millors.

Fent una petita anàlisi més general s'arriba als *mètodes de Taylor*, que poden tenir una "precisió" (més endavant es defineix això en termes del que es diu *ordre*) molt més elevada, però tenen l'inconvenient que la seva implementació numèrica és poc senzilla.

Aquest problema es resol amb els *mètodes de Runge-Kutta*, anomenats "RK", els quals avaluen el camp en "punts estratègics", a fi i efecte d'aproximar amb la mateixa precisió les funcions del mètode de Taylor, que són difícils d'obtenir explicitament i que féien difícil la seva implementació numèrica. Els mètodes de Runge-Kutta són de fàcil implementació numèrica i tenen bona precisió.

Acabem la secció veient com, a partir de dos mètodes Runge-Kutta, podem fer un *control de pas* que permet fer la integració variant el pas, h, en el temps, de manera que l'error es manté per sota d'una cota fixada a priori. Són els *mètodes Runge-Kutta-Fehlberg*, que permeten a l'usuari estalviar-se la molèstia d'haver de triar el pas d'integració. Es fixa una cota per a l'error i el mètode mateix ajusta el pas convenient i integra per sota del marge d'error donat.

3.2.2 El mètode d'Euler

Donat l'interval de temps $[a, b]$ definit en el problema de Cauchy i subdividit en N trocets, vegem com calculem x_1 a partir del valor inicial $x(a) = x_0$.

Notem que de la primera equació del problema de Cauchy obtenim, substituint la t per $a = t_0$,

$$x'(t_0) = f(t_0, x(t_0)) = f(t_0, x_0).$$

Aproximem llavors $x(t)$ a $[t_0, t_1]$ pel desenvolupament de Taylor de primer ordre al voltant de t_0, és a dir:

$$x(t) \simeq x(t_0) + x'(t_0)(t - t_0) = x_0 + f(t_0, x_0)(t - t_0),$$

per a $t \in [t_0, t_1]$. Així, per a $t = t_1$, obtenim

$$x(t_1) \simeq x_0 + f(t_0, x_0)(t_1 - t_0) = x_0 + f(t_0, x_0)h \equiv x_1.$$

Notem que x_1 s'obté de fet avançant un pas h en la direcció de la tangent a l'òrbita buscada en el punt x_0 i que aproxima el valor real $x(t_1)$ (vegeu la figura (3.9)).

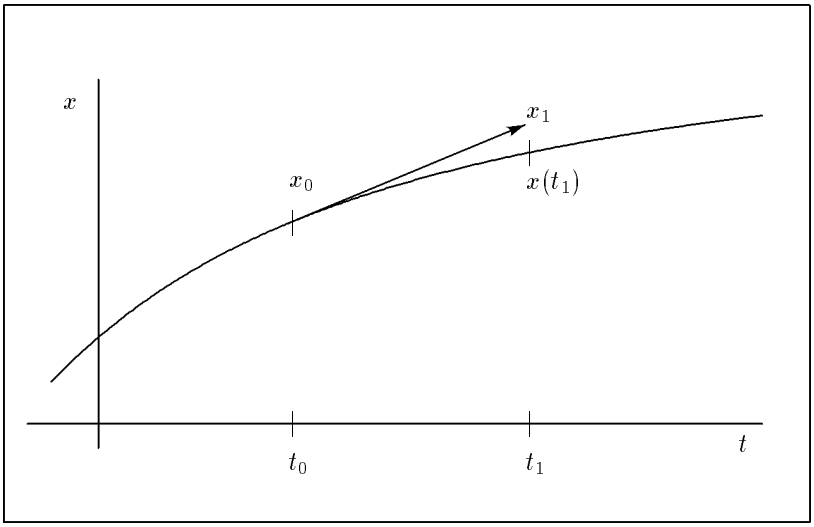

Figura 3.9: *Un pas del mètode d'Euler partint d'un punt inicial x_0. El mètode ens dóna l'aproximació x_1 del valor real $x(t_1)$.*

A continuació suposem que x_1 és una bona aproximació de $x(t_1)$ i considerem a $[t_1, t_2]$ el problema de Cauchy:

$$\begin{cases} x_1'(t) &= f(t, x_1(t)), \\ x_1(t_1) &= x_1, \end{cases}$$

per a $t \in [t_1, t_2]$.

Repetim aleshores el procés anterior per a $x_1(t)$,

$$x(t) \simeq x_1(t) \simeq x_1 + x_1'(t_1)(t - t_1) = x_1 + f(t_1, x_1)(t - t_1),$$

per a $t \in [t_1, t_2]$. I per tant fent $t = t_2$ obtenim

$$x(t_2) \simeq x_1(t_2) \simeq x_1 + f(t_1, x_1)(t_2 - t_1) = x_1 + f(t_1, x_1)h \equiv x_2.$$

Seguint de forma recurrent, tenim que l'algorisme d'Euler ve donat per

$$\begin{array}{lcl} x_0 & = & x(a), \\ x_{n+1} & = & x_n + hf(t_n, x_n), \quad n = 0 \ldots N-1. \end{array}$$

Exemple: Donat el problema de Cauchy

$$\left\{ \begin{array}{lcl} x'(t) & = & x(t), \\ x(0) & = & 1, \end{array} \right.$$

volem calcular $x(1)$ aplicant el mètode d'Euler.

Recordem que aquest problema és el primer que hem tractat en la segona part d'aquest llibre. Estem buscant la solució de $\dot{x} = x$ que per al temps $t = 0$ passa per $x = 1$.

En aquest cas la funció, f, que dóna el camp és, $f(t, x) = x$, i altres valors relacionats amb la notació del problema de Cauchy són: $x_0 = 1$, $a = 0$ i $b = 1$.

Per a aquest problema haviem trobat la solució analítica, que resulta ser $x(t) = e^t$, per tant el valor que busquem sabem que ha de sortir $x(1) = e = 2.71828\ldots$, i al mateix temps es fàcil comparar els valors intermedis.

Un cop triat N, l'algorisme d'Euler ens dóna:

$$\begin{array}{lcl} x_0 & = & x(0) = 1, \\ x_{n+1} & = & x_n + hx_n = (1+h)x_n, \quad n = 0 \ldots N-1, \end{array}$$

d'on, si fem $N = 2$, és a dir, $h = 1/2$, resulta la taula

t_n	x_n	$x(t_n)$
0	1.00000	1.00000
1/2	1.50000	1.64872
1	2.25000	2.71828

mentre que, si fem $N = 8$, cosa que ens dóna un pas de $h = 1/8$, tenim

t_n	x_n	$x(t_n)$
0	1.00000	1.00000
1/8	1.12500	1.13314
1/4	1.26562	1.28402
3/8	1.42382	1.45499
1/2	1.60180	1.64872
5/8	1.80203	1.86824
3/4	2.02728	2.11700
7/8	2.28069	2.39887
1	2.56578	2.71828

Si bé el resultat amb $h = 1/8$ millora bastant respecte de l'obtingut usant $h = 1/2$, el valor de les dècimes encara està malament. A fi i efecte de millorar l'aproximació usant passos no gaire petits, indicarem breument com es produeixen els errors en els mètodes numèrics d'integració.

3.2.3 Errors en els mètodes numèrics d'integració

En els mètodes numèrics d'integració tenim dues fonts d'errors:

- *Error de discretització o de truncament.* Generat pel fet que usem un algorisme aproximat, normalment derivat del mètode de Taylor, que s'ajusta més a la realitat com més petit és un cert pas h.

- *Error d'arrodoniment.* Generat pel fet que les operacions aritmètiques es fan amb un nombre finit de decimals i, per tant, quan el pas es fa petit, a part de l'acumulació d'operacions, pot resultar que nombres petits se sumin a quantitats molt més grans perdent-se dígits significatius. (Per exemple si treballem amb nombres que tenen mantissa de 6 decimals, la suma de la quantitat 10^{-7} a 0.111111 no l'afecta ja que el nou decimal no es pot guardar enlloc).

Aquests dos fets fan que l'expressió de l'error obtingut en funció del pas, h, tingui un comportament semblant al de la figura 3.10. Hi ha un pas òptim h_o pel qual l'error és mínim. Si es prenen $h > h_o$, els errors de discretització dominen sobre els d'arrodoniment i si $h < h_o$ passa a l'inrevés.

Fixat l'algorisme, no podrem mai reduir el valor de l'error per sota del seu $E(h_o)$. Per tant l'objectiu serà buscar algorismes tals que el valor de $E(h_o)$ sigui el menor possible i si és possible pel rang de h al voltant de h_0 el més gran possible.

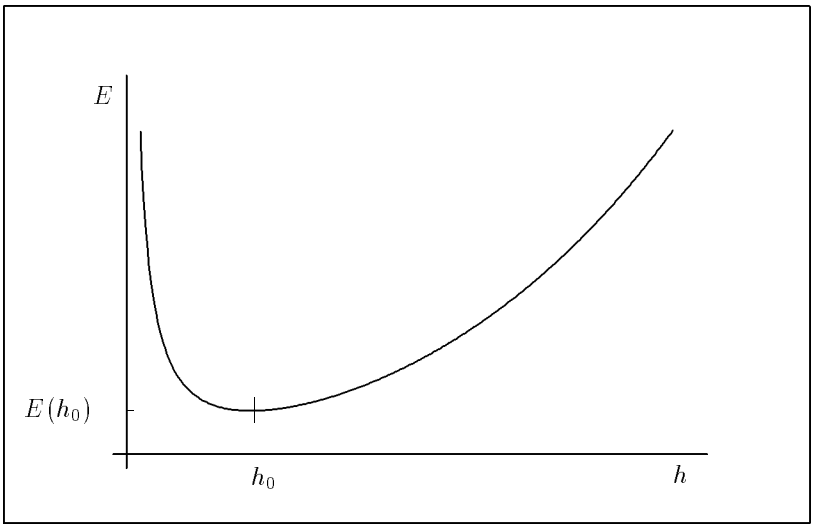

Figura 3.10: *Comportament de l'error $E(h)$ segons els diferents passos h. h_0 representa el valor òptim de l'algorisme.*

En el fons els problemes venen d'haver de prendre el pas, h, petit per aconseguir una bona aproximació teòrica. Si trobem algorismes que donin un error de discretització petit per a valors de h relativament grans, els errors d'arrodoniment seran gairebé negligibles, ja que el nombre

d'operacions decreixerà substancialment. A més, el nostre algorisme "correrà més" ja que per obtenir la mateixa precisió podrà usar passos molt més grans. Per aquest fet estudiarem l'error de truncament suposant que els errors d'arrodoniment són negligibles.

Tots els mètodes d'integració d'un pas que veurem per integrar el problema de Cauchy seguiran un algorisme del tipus

$$\begin{aligned} x_0 &= x(a), \\ x_{n+1} &= x_n + h\Phi(t_n, x_n, h), \end{aligned}$$

on la Φ suposarem que és una funció prou regular.

Amb tot això definirem *l'error de truncament* en el punt, x_n, al valor,

$$\varepsilon_n = |x(t_n) - x_n|,$$

i direm que un mètode d'integració té *ordre* p, $p \in N$, notat com $O(h^p)$, si hi ha valors $h_0 > 0$ i $k > 0$ tals que

$$\varepsilon_n \leq kh^p,$$

per a $h \in [0, h_0]$ i per a $n = 0 \ldots N$.

Si $p \geq 1$, llavors $\lim_{h\to 0} \varepsilon_n = 0$. Per tant, llevat d'errors d'arrodoniment, com més petita sigui la h millor serà l'aproximació. Es diu que el *mètode és convergent*. Veiem també que com més gran sigui el valor de p, més ràpida és la convergència. Així per exemple, amb caire illustratiu, si suposem que per a un cert mètode d'integració i interval de temps, $k = 1$, l'error de truncament, ε_n, queda fitat per h^p. Si prenem $h = 10^{-2}$ un mètode d'ordre 1 ens donarà una fita d'error 0.01, un d'ordre 2, 0.0001, i així anàlogament i successiva amb diferents passos i ordres. Per tant, heurísticament i de manera informal, passar d'un mètode d'ordre 1 a un d'ordre 2 conservant el mateix pas, ens pot suposar aconseguir el doble de decimals bons de precisió.

A fi de veure de quin ordre és un cert mètode d'integració, tenim el resultat següent. Suposem que les derivades de $x(t)$ són acotades a $[a, b]$. Si

$$\frac{x(t+h) - x(t)}{h} - \Phi(t, x(t), h) = O(h^p),$$

per un cert $p \in N$ per a tot $t \in [a, b]$, aleshores el mètode d'integració té ordre p.

Exemple: El mètode d'Euler té ordre 1.

En el mètode d'Euler, $\Phi(t, x(t), h) = f(t, x(t))$. Aleshores, si desenvolupem per Taylor, hi ha $\eta(t) \in [a, b]$ tal que:

$$\begin{aligned} \left| \frac{x(t+h) - x(t)}{h} - \Phi(t, x(t), h) \right| &= \left| \frac{x(t+h) - x(t)}{h} - f(t, x(t)) \right| = \\ &= \left| x'(t) + \frac{x''(\eta(t))}{2!} h - f(t, x(t)) \right|, \end{aligned}$$

i com que $x(t)$ és solució de $x'(t) = f(t, x(t))$ queda

$$\left| \frac{x''(\eta(t))}{2!} h \right| \leq kh.$$

3.2.4 Els mètodes de Taylor

El propòsit d'aquests mètodes és obtenir algorismes d'ordre més elevat que el mètode d'Euler. Seguint la idea analítica de la construcció del mètode d'Euler, els mètodes de Taylor es basen a aproximar la funció desconeguda, $x(t)$, a $[t_n, t_{n+1}]$, $n = 0 \ldots N-1$, pel desenvolupament de Taylor, al voltant de t_n i d'ordre k, de la solució del problema de Cauchy,

$$\begin{cases} x'(t) &= f(t, x(t)), \\ x(t_n) &= x_n, \end{cases}$$

per a $t \in [t_n, t_{n+1}]$. S'obté així un mètode d'integració d'ordre k.

Notem també que el mètode d'Euler no és altra cosa que el mètode de Taylor d'ordre 1.

Busquem el mètode de Taylor d'ordre 2 per al problema de Cauchy general:

$$\begin{cases} x'(t) &= f(t, x(t)), \\ x(t_0) &= x_0. \end{cases}$$

En tota aquesta secció suposarem que $x(t)$ és una funció en lloc d'un vector de funcions (és a dir, suposarem que la m que indica la dimensió del problema val 1). Notarem llavors $f_t = \frac{\partial f}{\partial t}$ i $f_x = \frac{\partial f}{\partial x}$; ara bé, interpretant les derivades parcials com a matrius diferencials convenients, tot queda generalitzat al cas m qualsevol.

A partir de les dades, $x(t_0) = x_0$, tal com en el cas del mètode d'Euler, ja sabem que tenim $y'(t_0) = f(t_0, x_0)$. Anem però a derivar l'equació diferencial del problema de Cauchy per a conèixer $x''(t_0)$.

Derivem la igualtat, $x'(t) = f(t, x(t))$, respecte de t, usant la *regla de la cadena* en diverses variables:

$$x''(t) = f_t(t, x(t)) + f_x(t, x(t))x'(t) = f_t(t, x(t)) + f_x(t, x(t))f(t, x(t)).$$

Per tant si ho avaluem per a $t = t_0$ ens queda

$$x''(t_0) = f_t(t_0, x(t_0)) + f_x(t_0, x(t_0))f(t_0, x(t_0)) = f_t(t_0, x_0) + f_x(t_0, x_0)f(t_0, x_0).$$

Si, de manera anàloga al desenvolupament del mètode d'Euler, aproximem $x(t)$, $t \in [t_0, t_1]$ pel seu desenvolupament de Taylor de segon ordre al voltant de t_0, tenim

$$\begin{aligned} x(t) &\simeq x(t_0) + x'(t_0)(t - t_0) + y''(t_0)\frac{(t-t_0)^2}{2!} = \\ &= x_0 + f(t_0, x_0)(t - t_0) + (f_t(t_0, x_0) + f_x(t_0, x_0)f(t_0, x_0)), \frac{(t-t_0)^2}{2!}, \end{aligned}$$

per tant, fent $t = t_1$ i tenint en compte que $t_1 - t_0 = h$, resulta

$$\begin{aligned} x(t) &\simeq x_0 + f(t_0, x_0)(t_1 - t_0) + (f_t(t_0, x_0) + f_x(t_0, x_0)f(t_0, x_0))\frac{(t_1-t_0)^2}{2!} = \\ &= x_0 + hf(t_0, x_0) + (f_t(t_0, x_0) + \frac{h^2}{2!}f_x(t_0, x_0)f(t_0, x_0)) \equiv x_1. \end{aligned}$$

Igual que ho fem pel mètode d'Euler, aquest procés el seguim de forma recurrent, plantejant el problema de Cauchy prenent com a condició inicial el darrer punt trobat. Això dóna lloc a l'algorisme de Taylor d'ordre 2:

$$\begin{aligned} x_0 &= x(a), \\ x_{n+1} &= x_n + hf(t_n, x_n) + (f_t(t_n, x_n) + \tfrac{h^2}{2!} f_x(t_n, x_n) f(t_n, x_n)), \quad n = 0 \ldots N-1. \end{aligned}$$

A títol informatiu direm que, així com el mètode d'Euler consisteix a aproximar l'òrbita solució usant el vector tangent, el mètode de Taylor d'ordre 2 aproxima l'òrbita solució per mitjà de paràboles.

Exemple: Tornem al problema de Cauchy

$$\begin{cases} x'(t) &= x(t), \\ x(0) &= 1, \end{cases}$$

per calcular $x(1)$ usant el mètode de Taylor d'ordre 2.

Tal com passa a l'exemple d'Euler, $x_0 = 1$, $a = 0$, $b = 1$ i $f(t, x) = x$. Ara, però, ens cal calcular a més les derivades parcials de f que resulten ser $f_t(t, x) \equiv 0$ i $f_x(t, x) \equiv 1$.

Substituint en l'algorisme general obtenim

$$\begin{aligned} x_0 &= 1, \\ x_{n+1} &= x_n + hx_n + \tfrac{h^2}{2!} x_n = (1 + h + \tfrac{h^2}{2!}) x_n, \quad n = 0 \ldots N-1, \end{aligned}$$

d'on, si fem $N = 2$, és a dir $h = 1/2$, resulta la taula,

t_n	x_n	$x(t_n)$
0	1.00000	1.00000
1/2	1.62500	1.64872
1	2.64062	2.71828

mentre que, si fem $N = 8$, el que ens dóna un pas de $h = 1/8$, obtenim

t_n	x_n	$x(t_n)$
0	1.00000	1.00000
1/8	1.13281	1.13314
1/4	1.28326	1.28402
3/8	1.45369	1.45499
1/2	1.64676	1.64872
5/8	1.86547	1.86824
3/4	2.11323	2.11700
7/8	2.39390	2.39887
1	2.71184	2.71828

És evident la millora obtinguda respecte dels resultats del mètode d'Euler en què hem usat els mateixos passos.

Finalitzarem aquesta secció veient que l'algorisme de Taylor de segon ordre que hem obtingut té efectivament ordre 2. Per això usarem el resultat de la secció anterior, tenint en compte que l'algorisme que estudiem és un mètode d'un pas amb

$$\Phi(t,x,h) = f(t,x) + \frac{h}{2}(f_t(t,x) + f_x(t,x)f(t,x)).$$

Per tant, desenvolupant per Taylor $x(t+h)$ al voltant de t, resulta que hi ha $\eta(t) \in [a,b]$ tal que

$$\left|\frac{x(t+h)-x(t)}{h} - \Phi(t,x(t),h)\right| =$$

$$= \left|\frac{x(t+h)-x(t)}{h} - f(t,x(t)) - \frac{h}{2}(f_t(t,x(t)) + f_x(t,x(t))f(t,x(t))\right| =$$

$$= \left|x'(t) + x''(t)\frac{h}{2} + x'''(\eta(t))\frac{h^2}{6} - f(t,x(t)) - \frac{h}{2}(f_t(t,x(t)) + f_x(t,x(t))f(t,x(t))\right|,$$

i com que $x(t)$ és solució de $x'(t) = f(t,x(t))$ i a partir d'aquí havíem calculat, $x''(t) = f_t(t,x(t)) + f_x(t,x(t))f(t,x(t))$, l'expressió anterior resulta

$$\left|x'''(\eta(t))\frac{h^2}{6}\right| \leq kh^2,$$

suposant, com sempre, que la derivada tercera està acotada.

Els mètodes de Taylor poden tenir un ordre tan gran com es vulgui, per això només ens cal desenvolupar $x(t)$ fins l'ordre desitjat, emprant la regla de la cadena. Ara bé, són difícils d'implementar en un ordinador, ja que hem de calcular i escriure en forma de funcions, les derivades (o diferencials del camp) que depenen del model que es vol integrar. Per poc complex que sigui el model, les derivades d'ordre superior poden ser molt pesades i, per tant, si ens animem a calcular-les, pot ser fàcil equivocar-nos tant en el càlcul com en la implementació. A més, si s'ha de modificar lleugerament el model, s'haurà de repetir tot el procés de nou. Per evitar tot això presentem els mètodes Runge-Kutta.

3.2.5 Els mètodes Runge-Kutta

Com hem dit en la secció anterior, aquests mètodes resolen la problemàtica que tenen els mètodes de Taylor a l'hora de calcular i implementar les diferencials del camp. La idea essencial consisteix a aproximar, amb el mateix ordre que el mètode d'integració, les diferencials dels mètodes de Taylor per combinacions d'avaluacions del camp en allò que podríem dir "punts estratègics". Per tant, la implemementació numèrica dels mètodes Runge-Kutta és molt senzilla. L'única cosa que s'haurà de canviar quan es modifiqui el model serà la funció que avalua el camp; mentre que la funció que conté l'algorisme general no s'haurà de tocar mai.

Per a introduir-los, direm que els mètodes Runge-Kutta vénen definits per algorismes de la forma

$$\begin{aligned} x_0 &= x(a), \\ x_{n+1} &= x_n + h\sum_{i=1}^{k} c_i \kappa_i^n, \end{aligned}$$

on $k \in N$ està prefixada a l'hora de buscar el mètode. Les c_i, $i = 1 \ldots k$ són constants a determinar i les $\kappa_i^n = \kappa_i^n(t_n, x_n, h)$, $i = 1 \ldots k$ són funcions definides recurrentment per

$$\begin{aligned} \kappa_1^n &= f(t_n, x_n), \\ \kappa_i^n &= f(t_n + a_i h, x_n + h \sum_{j=1}^{i-1} b_{ij} \kappa_j^n), \quad i = 2 \ldots k, \end{aligned}$$

amb a_i i b_{ij} constants a determinar, de manera que el mètode d'integració tingui ordre màxim.

Notem que k és el nombre d'avaluacions del camp que caldrà fer a cada pas ja que, a causa de l'estructura recurrent, el càlcul de κ_i^n és l'avaluació del camp en un cert punt calculat segons avaluacions anteriors.

Com veiem, els mètodes de Runge-Kutta són mètodes d'un pas amb

$$\Phi(t, x, h) = \sum_{i=1}^{k} c_i \kappa_i^n(t, x, h),$$

i per a $k = 1$ s'obté el mètode d'Euler, el qual també podem anomenar RK1 ja que és un Runge-Kutta d'ordre 1.

3.2.6 Runge-Kutta d'ordre 2

Busquem ara el Runge-Kutta d'ordre màxim que podem obtenir mitjançant dues avaluacions del camp. Es a dir, busquem el millor Runge-Kutta amb $k = 2$, i per tant tenim

$$\Phi(t, x, h) = c_1 \kappa_1 + c_2 \kappa_2 = c_1 f(t, x) + c_2 f(t + a_2 h, x + h b_{21} f(t, x)),$$

on hem per claredat, suprimit els índexs "n".

Desenvolupem per Taylor de diverses variables la funció Φ al voltant del punt (t, x). Per a les derivades parcials seguim la notació que hem introduït en la secció dels mètodes de Taylor.

$$\begin{aligned} \Phi(t, x, h) &= c_1 f(t, x) + c_2 (f(t, x) + a_2 h f_t(t, x) + h b_{21} f_x(t, x) f(t, x) + \frac{(a_2 h)^2}{2} f_{tt}(t, x) + \\ &\quad + a_2 b_{21} h^2 f_{tx}(t, x) f(t, x) + \frac{(h b_{21})^2}{2} f_{xx}(t, x) f^2(t, x)) + O(h^3). \end{aligned}$$

Desenvolupem ara per Taylor $x(t + h)$ al voltant de t i calculem $\frac{x(t+h)+x(t)}{h}$,

$$\frac{x(t + h) + x(t)}{h} = x'(t) + \frac{h}{2!} x''(t) + \frac{h^2}{3!} x'''(t) + O(h^3).$$

Com que ja sabem que $x'(t) = f(t, x(t))$, anàlogament que pels mètodes de Taylor, calculem $x''(t)$ i $x'''(t)$ en termes de la $f(t, x)$ i de les seves diferencials. El resultat l'inserim directament en l'expressió anterior i resulta,

$$\begin{aligned} \frac{x(t + h) + x(t)}{h} &= f(t, x) + \frac{h}{2} (f_t(t, x) + f_x(t, x) f(t, x)) + \\ &\quad + \frac{h^2}{3!} (f_{tt}(t, x) + 2 f_{tx}(t, x) f(t, x) + f_t(t, x) f_x(t, x) + \\ &\quad + f_{xx}(t, x) f^2(t, x)) + O(h^3). \end{aligned}$$

Per tant, si calculem

$$\frac{x(t+h)-x(t)}{h} - \Phi(t, x(t), h),$$

que com sabem ens serveix per conèixer l'ordre de l'integrador, obtenim

$$\begin{aligned}
&(1-c_1-c_2)f(t,x) + \\
&+\left((\frac{1}{2}-a_2c_2)f_t(t,x) + (\frac{1}{2}-c_2b_{21})f(t,x)f_x(t,x)\right)h + \\
&+\left((\frac{1}{6}-\frac{a_2^2c_2}{2})f_{tt}(t,x) + (\frac{1}{6}-\frac{b_{21}^2c_1}{2})f_{xx}(t,x)f^2(t,x) + \right. \\
&\left.+\frac{1}{6}(f_t(t,x)f_x(t,x) + f_x^2(t,x)f(t,x))\right)h^2 + O(h^3).
\end{aligned}$$

En aquesta expressió ens interessa triar les constants que tenim lliures, de manera que el desenvolupament comenci amb la potència de h més alta possible.

El terme $(f_tf_x + f_y^2 f)\frac{h^2}{6}$ no serà en general idènticament nul; per tant, amb $k=2$, com a màxim podrem assolir ordre 2. Vegem que efectivamnet podem aconseguir ordre 2. Per això ens cal que els termes d'ordre 0 i d'ordre 1 en h siguin idènticament nuls, es a dir ens cal que

$$\left\{\begin{array}{rcl} 1-c_1-c_2 & = & 0, \\ \frac{1}{2}-a_2c_2 & = & 0, \\ \frac{1}{2}-c_2b_{21} & = & 0. \end{array}\right.$$

Aquest sistema no lineal té infinitat de solucions; totes elles donen lloc a un Runge-Kutta d'ordre 2 que en principi és tan bo com qualsevol altre. Els resultats donats per a un o altre no seran idèntics però tenen el mateix ordre quant a l'error; és a dir que grollerament podríem dir que "ens donen el mateix nombre de decimals bons".

Potser el RK2 més conegut és el que s'obté prenent $a_2 = b_{21} = 1$ i $c_1 = c_2 = \frac{1}{2}$, que a més té l'avantatge de poder guardar exactes aquests valors dins la memòria de l'ordinador. Posant aquests valors en la definició dels mètodes Runge-Kutta obtenim l'algorisme RK2:

$$\begin{array}{rcl} x_0 & = & x(a), \\ x_{n+1} & = & x_n + h(\frac{1}{2}f(t_n,x_n) + \frac{1}{2}f_t(t_n+h, x_n+hf(t_n,x_n))), \quad n=0\ldots N-1. \end{array}$$

3.2.7 Runge-Kutta d'ordres superiors

De la mateixa manera es poden obtenir mètodes de Runge-Kutta d'ordre superior. Si per exemple posem $k=4$, obtenim mètodes RK4, el més conegut del quals és segurament:

$$\begin{array}{rcl} x_0 & = & x(a), \\ x_{n+1} & = & x_n + \frac{h}{6}(\kappa_1^n + 2\kappa_2^n + 2\kappa_3^n + \kappa_4^n), \quad n=0\ldots N-1, \end{array}$$

on

$$\begin{array}{rclrcl} \kappa_1^n & = & f(t_n, x_n), & \kappa_3^n & = & f(t_n+\frac{h}{2}, x_n+\frac{h}{2}\kappa_2^n), \\ \kappa_2^n & = & f(t_n+\frac{h}{2}, x_n+\frac{h}{2}\kappa_1^n), & \kappa_4^n & = & f(t_n+h, x_n+h\kappa_3^n). \end{array}$$

Una cosa que cal remarcar és que un mètode de Runge-Kutta d'ordre p necessita almenys p avaluacions del camp. De fet, els Runge-Kutta d'ordre 4 són els mètodes d'ordre més gran

pels quals el nombre d'avaluacions del camp coincideix amb l'ordre. Per construïr un RK5, el valor de k ja ha de valer 6; cal notar doncs que el valor de k no ha de coincidir forçosament amb l'ordre del mètode.

Per acabar aquesta secció, tornem al nostre exemple habitual:
Exemple: Donat el problema de Cauchy

$$\begin{cases} x'(t) &= x(t), \\ x(0) &= 1, \end{cases}$$

calculem $x(1)$ usant el mètodes Runge-Kutta d'ordres 2 i 4.

El primer apartat es deixa com a exercici. El lector que expliciti l'algorisme RK2 per a aquest problema s'adonarà que dóna el mateix que el que obtenim amb el mètode de Taylor d'ordre 2. Aquest fet però ha estat casual. En general els algorismes de Taylor d'ordre 2 i el RK2 que hem explicitat no han de coincidir necessàriament.

Tenint en compte que $f(t,x) = x$, si explicitem l'algorisme RK4 per aquest problema –la qual cosa, com veurem en apartats posteriors, no es fa mai quan s'implementen numèricament–, ens queda

$$\begin{aligned} x_0 &= 1, \\ x_{n+1} &= (1 + h + \tfrac{h^2}{2} + \tfrac{h^3}{6} + \tfrac{h^4}{24})x_n, \qquad n = 0 \ldots N-1, \end{aligned}$$

d'on, si fem $N = 2$, és a dir $h = 1/2$, resulta la taula

t_n	x_n	$x(t_n)$
0	1.00000	1.00000
1/2	1.64843	1.64872
1	2.71734	2.71828

mentre que, si fem $N = 8$, és a dir un pas de $h = 1/8$, tenim

t_n	x_n	$x(t_n)$
0	1.00000	1.00000
1/8	1.133148	1.133148
1/4	1.284025	1.284025
3/8	1.454990	1.454991
1/2	1.648720	1.648721
5/8	1.768244	1.868246
3/4	2.116997	2.117000
7/8	2.398872	2.398875
1	2.718277	2.718282

És una comprovació fàcil veure que l'error decreix segons ordre 4. És a dir, $E \approx Kh^4$ on K és en aquest cas un valor proper a 0.2.

3.2.8 Integració automàtica. Control de pas

Ja hem vist que podem obtenir mètodes d'integració d'implementació numèrica senzilla i d'ordre tan gran com vulguem. Hi ha però un problema pràctic que trobem quan hem de seguir una trajectòria d'un model a partir d'una condició inicial. És l'elecció del pas h en el temps. Un cop fixat aquest valor, els mètodes que tenim produeixen una taula de valors equiespaiada en el temps: $x(t_0)$, $x(t_0+h)$, $x(t_0+2h)\ldots$

Mantenir el pas h constant al llarg de tota la integració pot no ser convenient, ja que en certes regions pot passar que el camp sigui "suau" i, per tant, es cometi un error molt petit usant un pas relativament gran; mentre que, al cap d'una estona, la trajectòria pot entrar en una regió on el camp varïi molt, i si volem tenir poc error en la integració dins d'aquesta regió ens cal prendre un pas molt petit. Si usem un pas constant durant tota la integració, i volem mantenir-nos per sota d'un nivell d'error, ens veurem obligats a usar el pas petit durant tota la trajectòria, cosa que pot alentir notablement el procés de càlcul.

Canviar el pas durant la integració en principi no porta cap problema, ja que els mètodes que usem són d'un sol pas i, per tant, sempre podem aproximar un $x(t_{n+1})$, a partir d'un $x(t_n)$ anterior, havent fet un pas qualsevol, $h_n = t_{n+1} - t_n$. El problema és determinar quin ha de ser el pas de manera que x_{n+1} aproximi bé a $x(t_{n+1})$.

D'ara endavant suposarem que el pas, h, no és constant i que la successió t_0, t_1, $t_2\ldots$ no està equiespaiada, sinó que la diferència, $t_{n+1} - t_n = h_n$, pot anar variant durant el procés d'integració.

Com que l'elecció del pas, h_n, en cada moment és, si cap, encara més difícil que triar un pas h constant, el que farem serà fixar un "nivell d'error". Aleshores, el *mètode de control de pas* que veurem ens calcularà el pas h_n més gran possible, a fi i efecte de passar al nou punt per sota del nivell d'error donat.

Llavors el maneig de la funció d'integració serà molt còmode. Només donant la fita d'error d'integració, ella mateixa s'anirà regulant el pas a fi de mantenir l'error per sota del demanat; correrà més quan pugui i anirà més a poc a poc en els llocs on el camp varïi més.

D'entre els mètodes d'integració amb control de pas que hi ha, hem triat els *Runge-Kutta-Fehlberg*. La idea essencial d'aquests mètodes l'expliquem a continuació.

Suposem que estem en el punt $x(t_n)$ i donat un pas, h_n, volem aproximar el punt $x(t_{n+1})$. El que fem és calcular una aproximació, $\bar{x}_{n+1}$, de $x(t_{n+1})$ usant un mètode Runge-Kutta d'un cert ordre, i una altra aproximació $\hat{x}_{n+1}$ amb un mètode Runge-Kutta d'un ordre superior. Si les aproximacions coincideixen "prou bé" agafarem $\hat{x}$ com a aproximació de $x(t_{n+1})$, en cas contrari reduirem el pas h_n i repetirem el procés fins que les aproximacions donades pels dos mètodes Runge-Kutta coincideixin "prou bé".

Un cop fet això "predirem" el nou pas h_{n+1} que ens permetrà passar de l'aproximació de $x(t_{n+1})$ a la de $x(t_{n+2})$, de la mateixa manera que hem passat de la de $x(t_n)$ a la de $x(t_{n+1})$.

Que les aproximacions coincideixin "prou bé" ho fixa l'usuari i es mesura per $|\hat{x}_{n+1} - \bar{x}_{n+1}|$. La idea és que si els dos valors, donats per mètodes de diferent ordre, coincideixen en els j dígits significatius primers, aquests han d'estar bé (i segurament algun més en el cas de $\hat{x}_{n+1}$, ja que ens ve donat pel mètode d'ordre més gran).

El procés de reducció de h_n i la predicció de h_{n+1} per a la següent integració, el comentarem (i deduirem) una mica més avall. Ara direm només que es basa en l'ordre dels mètodes.

Una de les principals gràcies que tenen els mètodes Runge-Kutta-Fehlberg és que les avaluacions del camp s'aprofiten per calcular els mètodes Runge-Kutta de diferents ordres. És a dir, els sistemes d'equacions no lineals que donen lloc als diferents mètodes Runge-Kutta d'un cert ordre, es resolen de tal manera que les avaluacions del camp necessàries pel Runge-Kutta d'ordre superior es poden aprofitar per a construir també el Runge-Kutta d'ordre inferior.

Com que les avaluacions del camp normalment són la part més costosa quant a temps d'ordinador de tot l'algorisme, la feina que fem és essencialment la mateixa que si integréssim utilitzant només el Runge-Kutta d'ordre superior i no la que es faria amb dos mètodes d'ordre diferent.

El mètode que hem triat per presentar és el Runge-Kutta-Fehlberg d'ordres 4 i 5. Usualment el notarem com a RK45F. Aquest mètode aprofita 6 avaluacions del camp per fer un RK5, un RK4 i estimar el pas òptim. Per als nostres propòsits té una precisió més que suficient i al mateix temps dóna prestacions molt bones pel que fa a rapidesa d'integració.

Com ja hem dit, la construcció d'aquests algorismes és de fet la mateixa que hem vist pels mètodes Runge-Kutta. Per aquest motiu només en donarem el resultat.

Donat un pas, h_n, presentem com s'obté el x_{n+1} a partir de x_n, per mitjà d'un RK4 i un RK5 adients al mètode Runge-Kutta-Fehlberg. En primer lloc calculem les κ:

$$\begin{aligned}
\kappa_1 &= h_n f(t_n, x_n),\\
\kappa_2 &= h_n f(t_n + \frac{1}{4}h_n, x_n + \frac{1}{4}\kappa_1),\\
\kappa_3 &= h_n f(t_n + \frac{3}{8}h_n, x_n + \frac{3}{32}\kappa_1 + \frac{9}{32}\kappa_2),\\
\kappa_4 &= h_n f(t_n + \frac{12}{13}h_n, x_n + \frac{1932}{2197}\kappa_1 - \frac{7200}{2197}\kappa_2 + \frac{7296}{2197}\kappa_3),\\
\kappa_5 &= h_n f(t_n + h_n, x_n + \frac{439}{216}\kappa_1 - 8\kappa_2 + \frac{3680}{513}\kappa_3 - \frac{845}{4104}\kappa_4),\\
\kappa_6 &= h_n f(t_n + \frac{1}{2}h_n, x_n - \frac{8}{27}\kappa_1 + 2\kappa_2 - \frac{3544}{2565}\kappa_3 + \frac{1859}{4104}\kappa_4) - \frac{11}{40}\kappa_5),
\end{aligned} \tag{3.2}$$

a partir d'aquests valors, o vectors si la dimensió del sistema m és $m \geq 2$, tenim l'estimació, x_{n+1}, donada per un RK4:

$$\bar{x}_{n+1} = x_n + \frac{25}{216}\kappa_1 + \frac{1408}{2565}\kappa_3 + \frac{2197}{4104}\kappa_4 - \frac{1}{5}\kappa_5,$$

i la donada per un RK5:

$$\hat{x}_{n+1} = x_n + \frac{16}{135}\kappa_1 + \frac{6656}{12825}\kappa_3 + \frac{28651}{56430}\kappa_4 - \frac{9}{50}\kappa_5 + \frac{11}{40}\kappa_6. \tag{3.3}$$

L'estimació de l'error, tal com hem comentat, ve donada per

$$||\hat{x}_{n+1} - \bar{x}_{n+1}||,$$

però com que de fet tenim els valors de les κ i el RK4 normalment no cal usar-lo, restant les expressions per a les dues aproximacions, RK4 i RK5, es té

$$||\frac{1}{360}\kappa_1 - \frac{128}{4275}\kappa_3 - \frac{2197}{75240}\kappa_4 + \frac{1}{50}\kappa_5 + \frac{2}{55}\kappa_6||_2, \tag{3.4}$$

on el subíndex 2 vol dir la norma 2 del vector interior, és a dir, l'arrel quadrada de la suma dels quadrats de les seves components.

Seguidament vegem com obtenim la fórmula del pas òptim. Per a això observem que tenim dues estimacions del valor $x(t_{n+1})$, donades pel RK4 i pel RK5 i que es poden posar respectivament com

$$\bar{x}_{n+1} = x_n + h\Phi_4(t_n, x_n, h), \tag{3.5}$$
$$\hat{x}_{n+1} = x_n + h\Phi_5(t_n, x_n, h), \tag{3.6}$$

on la h és la h_n que fem servir en aquell moment.

Si ara notem

$$\Delta = \Delta(t, x, h) = \frac{x(t+h) - x(t)}{h},$$

sabem que el RK4 té ordre 4 i, per tant, amb el resultat de la pàgina 83,

$$-\Delta(t_n, x_n, h) + \Phi_4(t_n, x_n, h) = N_4(t_n)h^4 + O(h^5).$$

De la mateixa manera, el RK5 té ordre 5 i per tant

$$-\Delta(t_n, x_n, h) + \Phi_5(t_n, x_n, h) = N_5(t_n)h^5 + O(h^6).$$

Si ara restem les expressions (3.5) i (3.6), i tenim en compte aquestes dues darreres equacions ens queda

$$\bar{x}_{n+1} - \hat{x}_{n+1} = h(\Phi_4 - \Phi_5) = N_4(t_n)h^5 + O(h^6).$$

Per tant l'estimació de l'error té el comportament asimptòtic de $N_4(t_n)h^5$, és a dir, d'ordre 5. Per a aquells que no coneguin aquesta terminologia podríem dir que, "negligint termes d'ordre 6", quan la h és prou petita es té

$$\bar{x}_{n+1} - \hat{x}_{n+1} = h(\Phi_4 - \Phi_5) = N_4(t_n)h^5,$$

i per tant

$$||N_4(t_n)|| = \frac{||\bar{x}_{n+1} - \hat{x}_{n+1}||}{h^5}. \tag{3.7}$$

Suposem ara que ε és una tolerància d'error que nosaltres fixem d'entrada. Si suposem que el pas ha tingut èxit tindrem

$$||\bar{x}_{n+1} - \hat{x}_{n+1}|| \leq \varepsilon.$$

El nostre propòsit és predir el nou pas, h_N, de manera que al següent pas d'integració l'estimació d'error es continuï mantenint per sota de la tolerància permesa. És a dir volem que passi

$$\bar{x}_{n+2} - \hat{x}_{n+2} = N_4(t_{n+1})h_N^5 \leq \varepsilon. \tag{3.8}$$

Ara bé, si desenvolupem per Taylor $N_4(t_{n+1})$ al voltant de t_n tenim

$$N_4(t_{n+1}) = N_4(t_n + h_N) = N_4(t_n) + O(h_N),$$

la qual cosa posada a (3.8) ens dóna

$$N_4(t_n)h_N^5 + O(h^6) \leq \varepsilon.$$

Llavors, negligint els termes d'ordre $O(h_N^6)$ i aplicant una altra vegada (3.7) queda

$$h_N^5 \leq \frac{\varepsilon}{||N_4(t_n)||} = \frac{h^5\varepsilon}{||\bar{x}_{n+1} - \hat{x}_{n+1}||}.$$

Agafarem el màxim h_N que doni la precisió demanada, i d'aquesta manera obtindrem la *fórmula de predicció del nou pas*:

$$|h_N| = |h| \sqrt[5]{\frac{\varepsilon}{||\bar{x}_{n+1} - \hat{x}_{n+1}||}}, \tag{3.9}$$

on recordem que $||\bar{x}_{n+1} - \hat{x}_{n+1}||$ és l'error estimat a partir d'un RK4 i un RK5, que es pot calcular a partir de (3.4). ε és el màxim error permès d'aquesta estimació durant la integració i el fixa l'usuari.

Hem posat valors absoluts a h i a h_N, ja que la integració de la mateixa manera que l'hem considerat temps endavant ($h > 0$) podria anar temps enrera ($h < 0$).

Com a nota final direm, basats en l'experiència, que hi ha "retocs" d'aquesta fórmula; però no els comentarem.

3.2.9 Diagrama de flux per al RK45F i llista de l'integrador

Donem ara la funció d'integració que implementa el mètode Runge-Kutta-Fehlberg d'ordres 4 i 5 vist en la secció anterior.

A la funció d'integració que presentem, hi entrem, a més de la tolerància d'integració ε, que anomenem `tol`, dos valors positius `hmax` i `hmin` que donen, en valor absolut, els rangs entre els quals pot variar la predicció del nou pas. D'aquesta manera, si volem integrar amb pas constant només ens caldrà fer `hmin` = `hmax` a l'entrada, i si el que volem és dibuixar gràfiques unint amb rectes els punts de l'òrbita que es van obtenint, posant un `hmax` no gaire gran evitarem que en alguns casos la gràfica no es vegi "massa poligonal". Amb aquests tres valors donats, el diagrama de flux de la rutina és el de la figura 3.11.

Notes sobre del seu funcionament:

- L'integrador té com a entrades t_n, x_n i h_n i torna de sortida t_{n+1}, x_{n+1} i el pas, h_{n+1}, que recomana per a la crida següent. Per tant es poden anar fent crides successives i, sense tocar res, va seguint la trajectòria amb la fita d'error, ε, fixada.
- Calcular les κ vol dir usar les fórmules (3.2).
- Estimar l'error vol dir calcular el valor donat per l'equació (3.4).
- Predir un nou pas, h_N, vol dir actualitzar la variable h, que conté el pas, utilitzant l'equació (3.9) i mantenint el signe de h, i després fer $h_N = \frac{h}{|h|}$`hmin` si resulta que $|h| <$ `hmin`, o $h_N = \frac{h}{|h|}$`hmax` si resulta que $|h| >$ `hmax`.

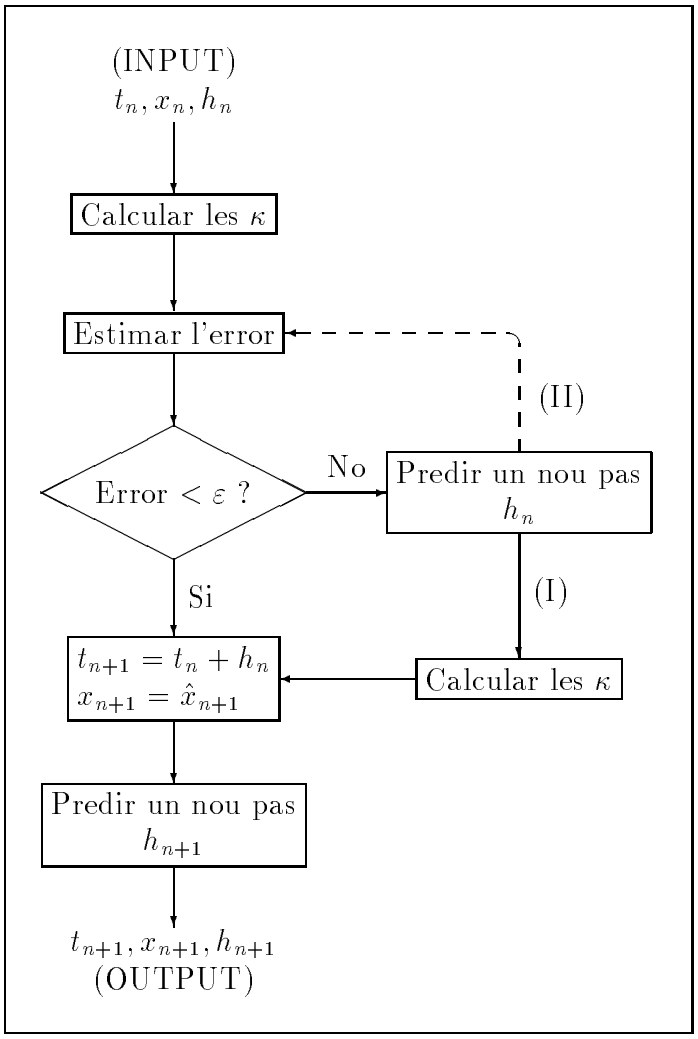

Figura 3.11: *Diagrama de flux. Funció* `rk45f`

- Actualitzar x_{n+1} amb $\hat{x}_{n+1}$ vol dir usar l'integrador RK5 donat per la igualtat (3.3).

- El lector podria pensar que el camí (I) hauria de ser en realitat (II) i involucrar un procés iteratiu. Ara bé el fet d'usar l'integrador RK5 per avançar, mentre que l'error ha estat estimat amb un RK4 i un RK5, fa que una sola correcció del pas, en cas que l'estimació d'error no tingués èxit, sigui suficient. L'únic problema pot trobar-se en la primera crida, en la qual hem de donar un pas inicial que creguem convenient. Si aquest pas fos massa gran, donat que les estimacions de correcció del nou pas són asimptòtiques, potser caldria fer més d'una correcció. És per això que el pas inicial de la primera crida no ha de ser massa gran.

- Si li pasem un pas, h, fora del rang, `hmin-hmax`, si l'estimació d'error té èxit, avança amb el pas donat, encara que a la sortida donarà el nou pas dins el rang indicat. Això va bé perquè a vegades podrem tenir un algorisme que ens busqui el pas a fi i efecte de calcular per exemple el tall de l'òrbita amb una certa recta. Aquest algorisme pot ser iteratiu, i resultar que el pas calculat es vagi reduint a mesura que afinem més i més el punt de tall. Si el primer que fes l'integrador fos mirar si el pas donat està dins el rang permès, cada vegada hauríem de canviar el rang[3].

Donem seguidament la llista de l'integrador RK45F en llenguatge C. Aquest integrador cal posar-lo en un arxiu i, un cop compilat, ens servirà per integrar qualsevol model donat per equacions diferencials, de la forma

$$\dot{x}(t) = f(t, x(t)), \qquad x \in \mathbb{R}^m,$$

sense haver de tocar res.

Com a darrers comentaris d'aquesta funció fixem-nos que el camp passa via un apuntador a funció, i que les κ es calculen a la funció `calcular_ks` i queden enmagetzemades en un sol vector, `k`. S'hi accedeix via els apuntadors `k1` ...`k6`. La variable d'entrada `tol` ha de contenir el límit d'error ε entre les estimacions donades pel RK4 i pel RK5.

Llista de la funció rk45f

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define E1 (1.e0/360.e0)
#define E2 (-128.e0/4275.e0)
#define E3 (-2197.e0/75240.e0)
#define E4 (1.e0/50.e0)
#define E5 (2.e0/55.e0)
#define R1 (16.e0/135.e0)
#define R2 (6656.e0/12825.e0)
#define R3 (28561.e0/56430.e0)
#define R4 (-9.e0/50.e0)
```

[3] D'això hi ha exemples a la secció 4.8.

```
#define R5 (2.e0/55.e0)
#define C1 (12.e0/13.e0)
#define C2 (1932.e0/2197.e0)
#define C3 (-7200.e0/2197.e0)
#define C4 (7296.e0/2197.e0)
#define C5 (439.e0/216.e0)
#define C6 (3680.e0/513.e0)
#define C7 (-845.e0/4104.e0)
#define C8 (-8.e0/27.e0)
#define C9 (-3544.e0/2565.e0)
#define C10 (1859.e0/4104.e0)

 void rk45f (double *t, double x[], int n, double *h, double hmin,
            double hmax, double tol, void (*camp)(double, double*,
                                                  int, double*) )
 {
  double *k,*k1,*k2,*k3,*k4,*k5,*k6,error,aux;
  int i;
  void calcular_ks(double x,double y[],int n,double h,double *k,
                   void (*camp)(double, double*, int, double*));
  k=(double *)malloc(6*n*sizeof(double));
  if (k==NULL) {puts("rk45f. No hi ha memoria per les k's\n");exit(1);}
  k1=k;k2=k1+n;k3=k2+n;k4=k3+n;k4=k3+n;k5=k4+n;k6=k5+n;
  calcular_ks(*t,x,n,*h,k,camp);
  error=0.e0;
  for (i=0;i<n;i++)
   {
    aux=E1*k1[i]+E2*k3[i]+E3*k4[i]+E4*k5[i]+E5*k6[i];
    error+=aux*aux;
   }
  error=sqrt(error);
  if (error>tol)
   {
    *h=*h*(pow(tol/error,0.2e0));
    if (fabs(*h)<hmin) *h=*h/fabs(*h)*hmin;
    if (fabs(*h)>hmax) *h=*h/fabs(*h)*hmax;
    calcular_ks(*t,x,n,*h,k,camp);
   }
  *t=*t+*h;
  for (i=0;i<n;i++)
    x[i]=x[i]+R1*k1[i]+R2*k3[i]+R3*k4[i]+R4*k5[i]+R5*k6[i];
  *h=*h*(pow(tol/error,0.2e0));
  if (fabs(*h)<hmin) *h=*h/fabs(*h)*hmin;
  if (fabs(*h)>hmax) *h=*h/fabs(*h)*hmax;
  free(k);
 }
```

```
void calcular_ks (double t, double x[], int n, double h, double *k,
                  void (*camp)(double, double*, int, double*) )
{
 double *aux_x,*k1,*k2,*k3,*k4,*k5,*k6;
 int i;
 k1=k;k2=k1+n;k3=k2+n;k4=k3+n;k4=k3+n;k5=k4+n;k6=k5+n;
 aux_x=(double *)malloc(n*sizeof(double));
 if (aux_x==NULL) {puts("calcular_ks. No hi ha memoria \n");exit(1);}
 (*camp) (t,x,n,k1);
 for (i=0;i<n;i++) k1[i]*=h;
 for (i=0;i<n;i++) aux_x[i]=x[i]+0.25e0*k1[i];
 (*camp) (t+0.25e0*h,aux_x,n,k2);
 for (i=0;i<n;i++) k2[i]*=h;
 for (i=0;i<n;i++) aux_x[i]=x[i]+0.09375e0*k1[i]+0.28125e0*k2[i];
 (*camp) (t+0.375e0*h,aux_x,n,k3);
 for (i=0;i<n;i++) k3[i]*=h;
 for (i=0;i<n;i++) aux_x[i]=x[i]+C2*k1[i]+C3*k2[i]+C4*k3[i];
 (*camp) (t+C1*h,aux_x,n,k4);
 for (i=0;i<n;i++) k4[i]*=h;
 for (i=0;i<n;i++) aux_x[i]=x[i]+C5*k1[i]-8.e0*k2[i]+C6*k3[i]+C7*k4[i];
 (*camp) (t+h,aux_x,n,k5);
 for (i=0;i<n;i++) k5[i]*=h;
 for (i=0;i<n;i++)
    aux_x[i]=x[i]+C8*k1[i]+2.e0*k2[i]+C9*k3[i]+C10*k4[i]-0.275e0*k5[i];
 (*camp) (t+0.5e0*h,aux_x,n,k6);
 for (i=0;i<n;i++) k6[i]*=h;
 free(aux_x);
}
```

Com a exemple d'aplicació, i per poder comprovar que no s'ha comès cap error d'escriptura, es poden fer dos tests senzills.

El primer consisteix a integrar el model donat per

$$\begin{cases} \dot{x}_1 &= -x_2, \\ \dot{x}_2 &= x_1. \end{cases}$$

És fàcil veure que les òrbites d'aquest sistema són circumferències centrades a l'origen i de radi R arbitrari,

$$(x_1(t), x_2(t)) = (R\cos t, R\sin t).$$

De fet també, si multipliquem la primera equació per x_1, la segona per x_2 i les sumem, obtenim

$$x_1\dot{x}_1 + x_2\dot{x}_2 = 0,$$

i per tant, per integració immediata, s'obté $x_1^2 + x_2^2 = C$. La qual cosa vol dir que la funció $F(x_1, x_2) = x_1^2 + x_2^2$, que no és res més que el radi al quadrat, pren un valor constant damunt de cada òrbita solució i aquest valor varia en canviar d'òrbita.

Una funció[4] no constant que pren valors constants damunt de les òrbites del model s'anomena *integral primera*. Si disposem d'una integral primera, la podem usar per veure que la integració es realitza de manera correcta.

Donat un punt inicial qualsevol s'avalua la integral primera i aquest valor s'ha de conservar (dins el marge d'error permès) al llarg de tota la trajectòria.

Tenint en compte tot això, el que fem és escriure en un arxiu, a part d'aquell on hi tenim l'integrador, el seguent programa principal i la funció que ens dóna el model a integrar. Com que les òrbites que han de sortir en aquest cas han de ser circumferències, a la funció que conté el camp li hem dit `cercle`. Aquesta funció, l'única cosa que ha de fer és avaluar el costat dret de les equacions i tornar el seu valor en un vector de dimensió igual al nombre d'equacions. Cal recordar que en llenguatge C els índexs d'un vector de dimensió n van de 0 a $n-1$.

```
#include <stdio.h>
#include <math.h>

main()
{
 void rk45f(double *x, double y[], int n, double *h, double hmin,
   double hmax, double tol, void (*camp)(double, double*, int, double*));
 void cercle(double t,double x[], int n, double y[]);
 int n;
 double t,x[2],h,r,hmin,hmax,tol;
 /* incialitzem el temps inicial i la dimensio del sistema */
 t=0.e0;
 n=2;
 /* posem el punt de sortida i els valors minim i maxim del pas */
 x[0] = 1.e0;
 x[1] = 0.e0;
 hmin=1.e-3;
 hmax=1.e0;
 puts ("tolerancia ?");
 scanf("%le",&tol);
 /* una bona eleccio del pas inicial es la seguent */
 h=1.e-2;
 /* valor inicial del radi al quadrat*/
 r=x[0]*x[0]+x[1]*x[1];
 puts("Valors inicials:");
 printf("t=%11.4le x0=%11.4le x1=%11.4le r=%24.16le h=%11.5le \n",
        t,x[0],x[1],r,h);
 while (t<10.e0)
 {
  rk45f(&t,x,n,&h,hmin,hmax,tol,cercle);
  r=x[0]*x[0]+x[1]*x[1];
  printf("t=%11.4le x0=%11.4le x1=%11.4le r=%24.16le h=%11.5le \n",
        t,x[0],x[1],r,h);
```

[4] De classe C^1.

```
 }
}


void cercle(double t,double x[], int n, double y[])
{
 y[0] = -x[1];
 y[1] = x[0];
}
```

Podem compilar l'arxiu que conté aquest programa principal i la funció `cercle`. Després muntar-lo amb l'arxiu compilat que conté l'integrador, i obtenir així un programa executable.

Fent córrer aquest programa per a diferents valors de la tolerància `tol`, per exemple des de 10^{-2} fins a 10^{-10}, s'ha d'observar que el valor de `r`, que de fet és el quadrat del radi, és conserva més proper al valor que té en el moment inicial de la integració, quan més petit es dóna el valor de `tol`[5].

A causa de la simetria del camp, el pas h s'estabilitza al cap de poques crides, el valor d'estabilització[6] depèn òbviament del valor que s'hagi donat a `tol`.

És important fixar-nos en com passen els paràmetres i la funció que conté el camp, a la funció `rk45f`. Si es tenen dubtes sobre aquest fet, cal consultar l'apèndix de sobre llenguatge C d'aquest llibre. Finalment observem també que la funció que conté el camp (en el nostre cas `cercle`) sempre ha de tenir els paràmetres indicats encara que no es facin servir. En el nostre cas no es fa servir ni `t`, ja que el sistema és autònom, ni la dimensió del sistema `n`. Malgrat que en la compilació apareixeran *warnings*, tal com l'hem escrit és la manera correcta de fer-ho.

Com que l'exemple anterior és autònom es pot fer un segon test per comprovar que la funció `rk45f` funciona bé també per a equacions que depenguin del temps. Un prova senzilla pot ser integrar l'equació

$$\dot{x} = tx,$$

que anomenem `expo` i la solució de la qual és

$$x(t) = Ke^{\frac{t^2}{2}},$$

on K és una constant arbitrària que queda fixada al triar les condicions inicials.

Si per exemple $x(0) = 1$, llavors hem triat l'òrbita amb $K = 1$ i per tant $x(2) = e^2 = 7.399056\ldots$

El següent programet serveix per calcular aquest valor. La tècnica és integrar fins a passar el valor $t = 2$ i després tornar enrera amb el pas necessari a fi d'ajustar exactament el temps igual a 2. Amb pocs canvis es pot buscar el resultat per a qualsevol altre temps.

És convenient que el lector usi aquest i d'altres exemples amb resultats coneguts per veure quins marges d'actuació té amb els valors de la tolerància, `tol`, i els passos mínim i màxim.

```
#include <stdio.h>
#include <math.h>
```

[5] Sempre que ens mantinguem dins el rang de precisió permès per les variables `double`.

[6] Si el valor de `h` sempre coincideix amb `hmin` és que hi ha algun error en l'escriptura de la funció.

```
main()
{
 void rk45f(double *x, double y[], int n, double *h, double hmin,
   double hmax, double tol, void (*camp)(double, double*, int, double*));
 void expo(double t,double x[], int n, double y[]);
 int n;
 double t,x[1],h,ve,hmin,hmax,tol,tf,err;
 /* incialitzem el temps inicial, final i la dimensio del sistema */
 t=0.e0;
 tf=2.e0;
 n=1;
 /* posem el punt de sortida i els valors minim i maxim del pas */
 x[0] = 1.e0;
 hmin=1.e-3;
 hmax=1.e0;
 puts ("tolerancia ?");
 scanf("%le",&tol);
 h=1.e-2;
 while (t<tf)
 {
  rk45f(&t,x,n,&h,hmin,hmax,tol,expo);
 }
  h=tf-t;
  rk45f(&t,x,n,&h,hmin,hmax,tol,expo);
  ve=exp(tf);
  err=fabs(ve-x[0]);
  printf(" tf=%24.16le \n vexacte=%24.16le \n",t,ve);
  printf(" vaprox=%24.16le \n err=%24.16le \n",x[0],err);
}


void expo(double t,double x[], int n, double y[])
{
 y[0] = t*x[0];
}
```

Capítol 4 Alguns models donats per equacions diferencials ordinàries

4.1 El model malthusià

Suposarem una població de gent, animals, bactèries o qualsevol altra cosa que es reprodueixi "per ella mateixa". En aquest sentit, el mot població també es pot referir a una certa malaltia infecciosa (la malaltia es reprodueix per ella mateixa dins la comunitat), o bé a la gent assabentada d'una determinada notícia, d'una certa informació, o a la qual ha arribat la propaganda d'un cert producte. El nostre propòsit és modelar la seva evolució al llarg del temps i, fonamentalment, poder predir el que passarà en un temps futur.

Per començar amb l'exemple més senzill, en principi suposarem que la reproducció (o escampament de la notícia) es du a terme d'una manera contínua, prenent-hi part tots els seus membres per igual, sense distinció del que podria ser l'edat o el sexe. Considerarem en aquest cas un ritme, r, de creixement mitjà per càpita, que més endavant especificarem.

Notarem amb $N(t)$ a la població (o gent afectada) a l'instant t. Si $N(t)$ compta individus o coses concretes hauria de ser un valor enter. Ara bé, en el nostre cas com que usarem models continus, suposarem que la funció $N(t)$ és "suau" al llarg del temps. De fet només estem buscant una aproximació del que passa realment i per tant si $N(t)$ a l'instant que ens interessa ens dóna un nombre decimal, el que farem serà arrodonir-lo a l'enter més proper. De vegades, però, $N(t)$ pot ser una quantitat escalada, per exemple el seu valor representa milers d'unitats; o bé una quantitat normalitzada, com per exemple una densitat o el tant per cent de població afectada pel que estem estudiant. En aquests casos és clar que, inclús en la realitat, no serà un valor enter.

El model més senzill de creixement el va donar l'economista britànic Malthus al voltant de l'any 1800. Direm que la població creix segons una *llei malthusiana* si la seva velocitat de creixement és proporcional a la quantitat:

$$\frac{dN}{dt}(t) = rN(t).$$

Notem en primer lloc que això no es res més que l'equació

$$\dot{x} = ax,$$

vista a la seccio 3.1.1, però escrita en altres variables. Ja sabem aleshores que la seva solució general és

$$N(t) = N(t_0)e^{r(t-t_0)},$$

i per tant el ritme de creixement és exponencial. Es té que si $r < 0$ (creixement negatiu) la població a la llarga s'estingeix, mentre que si $r > 0$ la població tendeix a infinit. Aquest darrer fet va portar Malthus a pronosticar, pessimístament, que les necessitats de la població humana depassarien àmpliament la producció d'aliments que només creixien, segons ell, linealment en el temps.

Abans de millorar aquest model introduint-hi alguns canvis, vegem més precisament el significat del valor r, que al començament d'aquest capítol hem anomenat *ritme mitjà de creixement per càpita*, és a dir, el tant per u per unitat de temps del creixement.

En un model malthusià és relativament fàcil de mesurar si fem la següent consideració. Busquem el temps que ha de passar per què la població es dobli seguint la llei de Malthus. Diem T_2 a aquest interval de temps, i suposem que la població passa de I individus a $2I$ durant aquest temps. Per la llei de Malthus tenim,

$$2I = Ie^{rT_2},$$

per tant la quantitat I és simplifica i, d'entrada, T_2 és independent de si la població és molt gran o no. Seguint la llei de Malthus, sempre tarda $T_2 = \frac{\ln 2}{r}$ unitats de temps en doblar-se.

Recíprocament, r es pot calcular com

$$r = \frac{\ln 2}{T_2},$$

on recordem que $\ln 2 \simeq 0.693147$.

Així per exemple si una població creix instantàniament al ritme del 3% a l'any trigarà $T_2 \simeq \frac{0.693147}{0.03} \simeq 23$ anys a doblar-se. I si una notícia o un fet és conegut cada dia pel doble de persones que el dia anterior el ritme r d'escampament és aproximadament del 3% al dia.

És clar que no cal esperar que es doblin el nombre d'individus per poder mesurar r. Si a l'instant t_0 es mesuren N_0 individus i a l'instant t_1 se'n mesuren N_1, d'acord amb la llei de Malthus es té $N_1 = N_0 e^{r(t_1-t_0)}$, i per tant,

$$r = \frac{\ln N_1 - \ln N_0}{t_1 - t_0}.$$

Com que els valors obtinguts per r poden variar segons els instants t_0 i t_1 en què es prenen les mesures, podem fer dues coses. Si varien lleugerament, podem pendre "una mitjana" d'entre els diferents valors que hem obtingut, mentre que si el ritme de variació és gran, podem considerar que la r és una funció del temps $r(t)$, o del nombre d'individus $r(N)$, i la podem determinar o predir mitjançant mètodes d'interpolació i extrapolació. El cas particular, i potser suficient en la majoria dels casos, on r és una funció lineal del nombre d'indiviuds l'estudiem tot seguit.

4.2 El model logístic

El model de Malthus s'adapta bé a l'estudi de fenòmens que creixen sense competència. Per exemple, quan es deixa una petita mostra de llevat en un cultiu, o apareix una malaltia infecciosa en una població gran, o bé quan surt una notícia coneguda en principi per poques persones. En tots aquests processos al principi es detecta un creixement exponencial i per tant es segueix la llei malthusiana. Però a la llarga la mateixa saturació porta a la ralentització del creixement i s'acosta a un estat estacionari, de manera que contradiu la llei de Malthus que considera el creixement exponencial i il·limitat.

Posem per exemple una plaga d'insectes que ataqui un cert cultiu sense defenses. Al principi els pocs insectes que apareixen no troben cap resistència del medi i creixen exponencialment, ja que podem considerar que els seus recursos són il·limitats, si més no en comparació amb la quantitat d'insectes. Però, a poc a poc, la massificació crea una competència interna i a mesura que el temps avança s'arriba a un equilibri entre l'aliment i el nombre d'insectes. En aquest punt, el creixement de la plaga (malgrat que el camp està saturat) és nul.

El mateix podem dir de la propagació d'una notícia. Al principi s'estén ràpidament, però arriba un dia (o al cap de poques hores) que el nombre de persones no assabentades, i que se n'assabenten cada dia que passa, és cada vegada menor. En aquest cas el nivell de saturació és el nombre de persones de la societat a les quals afecta la notícia, i en cap cas es pot considerar infinit.

Per aquests motius usualment no és significatiu considerar que la r és constant al llarg del temps, i s'ha provat molt adequat suposar que la r és funció del nombre d'individus (i per tant del temps) lligada segons un comportament lineal

$$r(N) = a - bN,$$

on a i b són constants positives.

Si es considera la r d'aquest tipus, aleshores el mateix raonament instantani que ens dóna el creixement malthusià ens porta ara a

$$\frac{dN}{dt} = N(a - bN),$$

que és l'anomenada *equació logística* i que porta al creixement logístic introduït per Verlhust a finals de la dècada de 1830.

Fixem-nos que el fet essencial d'aquesta equació és que si $a - bN > 0$, la qua cosa passarà quan N sigui relativament petita, es té creixement; mentre que si la N es prou gran, de manera que $a - bN < 0$, resulta que el nombre d'individus no pot ser suportat per l'habitat que els manté i per tant la població decreix, ja que la derivada de la població és negativa.

L'equació logística es pot escriure també de la manera

$$\frac{dN}{dt} = rN(1 - \frac{N}{K}),$$

on $r = a$ i $K = a/b$.

Aquesta representació potser té una interpretació més directa a partir de la llei de Malthus, ja que, si N és petit, $1 - \frac{N}{K} \simeq 1$, i per tant la llei és essencialment la malthusiana, d'on un altre

cop el ritme de creixement r es pot estimar a partir d'observacions fetes en el moment que no hi ha competència interna. Mentre que el signe de $\frac{dN}{dt}$ canvia segons si $N < K$, que implica creixement de la població, o si $N > K$, que implica decreixement de la població.

Com que $N = K$ implica creixement zero, l'òrbita $N(t) = K$ és estacionària i per tant K és el *nivell de saturació* de la població, al qual es tendeix asimptòticament quan el temps tendeix a infinit. En aquest cas direm que K és *asimptòticament estable*, ja que petites desviacions de la posició d'equilibri s'esmorteeixen i, a la llarga, es torna sempre al valor K (vegeu els retrats de fase de la figura 4.2).

És fàcil veure que la solució general de l'equació logística és

$$N(t) = \frac{K}{1 + Ce^{-rt}},$$

on C és la constant arbitrària que es determina a partir d'una condició inicial. Així per exemple, si sabem que $N(t_0) = N_0$, llavors

$$C = e^{rt_0}(\frac{K}{N_0} - 1).$$

El comportament de les gràfiques de les solucions $N(t)$ per a valors de població inferiors al nivell de saturació, estan representades a la figura 4.1. Notem que el punts d'inflexió de creixement (on la funció passa de convexa a còncava) es donen sempre per al valor de $N = \frac{K}{2}$.

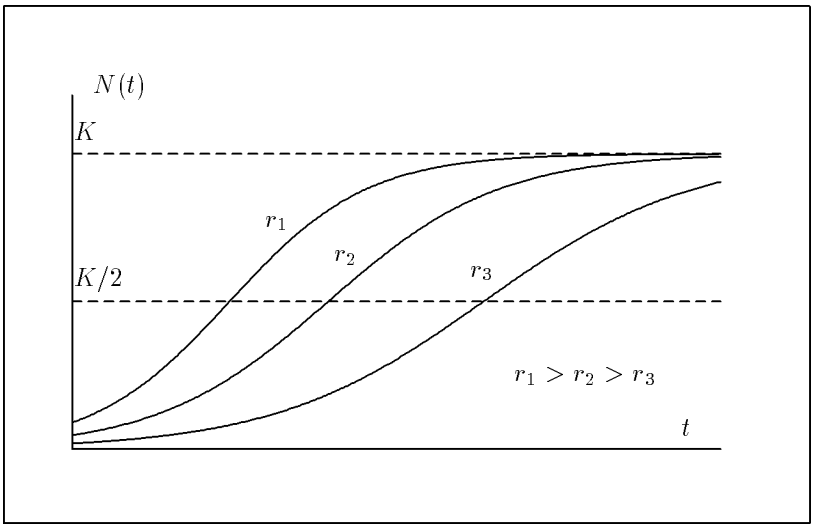

Figura 4.1: *Corbes logístiques. Totes elles tenen la saturació en el valor K i la inflexió en $K/2$.*

És clar que podem afegir d'altres modificacions sobre el model logístic per tal d'ajustar-lo més al nostre cas real. Per exemple, suposem el cas en què la població té creixement logístic,

però té també una certa immigració o emigració proporcional al nombre d'individus. En aquest cas el model és

$$\dot{N} = rN(1 - \frac{N}{K}) \pm EN,$$

amb $E > 0$ i on el signe l'agafem positiu o negatiu segons si tenim immigració o emigració respectivament. Per exemple, si la unitat de temps és un dia i N representa la quantitat de peixos d'un llac o d'un viver, suposant que la pesca diària és d'1 de cada 100, llavors $E = 0.01$ amb el signe negatiu.

Una altra adaptació senzilla és pensar que el nivell de saturació K variï al llarg del temps, és a dir que tinguem $K(t)$; llavors

$$\dot{N}(t) = rN(t)(1 - \frac{N(t)}{K(t)}),$$

ja que podem trobar per exemple que la capacitat màxima depengui d'algun factor com la temperatura, o la llum del sol, que al seu temps pot dependre de les estacions de l'any....

Fixem-nos que tots els casos que hem vist, llevat del que $K = K(t)$, són models del tipus

$$\dot{N} = f(N),$$

és a dir el creixement o el decreixement només depèn del nombre d'individus.

Per a aquests casos fer el retrat de fase és molt fàcil, ja que en els punts on $f(N) > 0$ es té creixement, els punts on $f(N) = 0$ són punts d'equilibri i on $f(N) < 0$ es té decreixement. La figura 4.2 representa qualitativament el cas de creixement logístic i un altre cas pel qual, si la població no arriba a un cert nivell mínim N_1, es produeix l'extinció.

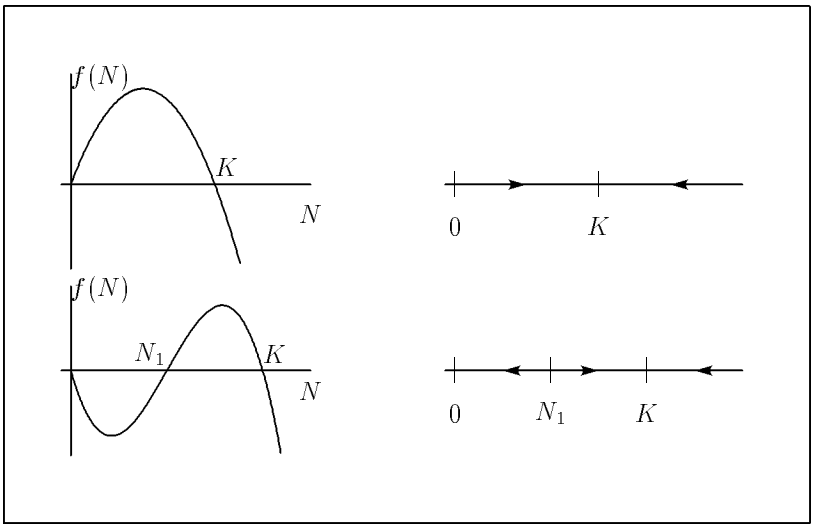

Figura 4.2: *Retrats de fase per a models unidimensionals del tipus* $\dot{N} = f(N)$.

4.3 Calibratge d'un model logístic

En aquesta secció donarem un exemple real de com es pot calibrar un model logístic usant el mètode de mínims quadrats.

El model que anem a calibrar estudia els peatges d'autopistes. Aquí només donarem les dades necessàries i calibrarem un exemple. El lector interessat en aquest tema pot consultar [7].

Les autopistes són eixos de comunicació que normalment discorren paral·lelament a d'altres carreteres. Els vehicles que entren a l'autopista de peatge paguen a canvi d'un servei com pot ser la comoditat o el guany de temps. El peatge podem considerar que és el factor determinant a fi que un cotxe utilitzi o no l'autopista. Si el peatge és car entraran pocs cotxes mentre que si és barat n'entraran molts. El tant per cent de cotxes que usa l'autopista podem modelar-lo per una corba logística respecte del peatge.

Recordem que la solució del model logístic és donada per l'equació

$$N(t) = \frac{K}{1 + Ce^{-rt}}. \tag{4.1}$$

En el cas que ens ocupa, la N representarà el tant per u de cotxes que usen l'autopista i la t el preu del peatge. Canviarem els seus noms per P i τ, respectivament.

Notem que com que P representa un tant per u, el valor K, que és el nivell de saturació de la corba logística, val 1, i per tant en el nostre cas (4.1) s'escriu com

$$P(\tau) = \frac{1}{1 + e^{a+b\tau}}, \tag{4.2}$$

on hem canviat r per $-b$ i la constant C per e^a, essent a i b els paràmetres de la corba logística que hem de trobar.

A fi d'estudiar el comportament dels vehicles a l'hora d'entrar o no a l'autopista, s'han seleccionat a tot l'estat espanyol 16 trams d'autopista en què la carretera que discorre paral·lela representi una alternativa acceptable.

Els trams i les seves alternatives estan representats a la taula 4.1.

Mitjançant estacions d'aforament adequades es prenen mesures a les carreteres alternatives a fi de poder estimar el nombre de cotxes que podent passar per l'autopista opten per la via alternativa. La taula 4.2 mostra les mesures preses quant a intensitat mitjana diària de trànsit pels trams d'autopista i pel seus traçats alternatius de la taula 4.1 durant l'any 1990. També, la mateixa taula, dóna la tarifa mitjana de peatge de cada tram mesurada en pessetes per quilòmetre en el mateix any. Els resultats donats corresponen únicament al cas de vehicles lleugers.

Les dades de les columnes 4 i 5 de la taula 4.2 són les que hem d'usar a fi d'ajustar els valors de a i b de (4.2). Així, per exemple, usant el primer tram d'autopista caldria imposar que

$$0.565 = \frac{1}{1 + e^{a+10.7b}}.$$

En fer això per a les 16 dades de què disposem ens resulta un sistema de 16 equacions amb només 2 incògnites i, per tant, cal buscar la solució en el sentit de *mínims quadrats*.

Els mètodes d'aproximació per mínims quadrats es poden trobar comentats en els capítols dels llibres de càlcul numèric dedicats a l'aproximació de funcions. Per exemple, a [1] hi ha una bona introducció a aquest tema.

	Autopista	Concessionari	Tram	Carretera
1	A-1	EUROVÍAS	Pancorbo-Miranda	N-I
2	A-2	ACESA	Bujaraloz-Fraga	N-II
3	A-2	ACESA	Lleida-Les Borges B.	N-II
4	A-4	AUMAR	Los Palacios-Las Cabezas	N-IV
5	A-7	ACESA	Figueres S.-L'Escala	N-II
6	A-7	ACESA	Cassà-Lloret	N-II
7	A-7	ACESA	A. del Ebro-El Vendrell	N-340
8	A-7	AUMAR	Cambrils-L'Hospitalet	N-340
9	A-7	AUMAR	Torreblanca-Oropesa	N-340
10	A-7	AUMAR	Oliva-Ondarra	N-332
11	A-8	EUROPISTAS	Basauri-Galdakao	N-634
12	A-9	AUDASA	Pontevedra-S. Moaña	N-550
13	A-19	ACESA	Montgat-Alella	N-II
14	A-66	AUCALSA	Campomanes-Villablino	N-630
15	A-68	AVASA	Agoncillo-Calahorra	N-232
16	A-69	AVASA	Gallur-Alagón	N-232

Taula 4.1: *Trams d'autopistes usats en el calibratge i les seves alternatives.*

	IMD Au.	IMD Ca.	% Au.	Preu km
1	7080	5460	56,5	10,7
2	8330	3130	72,7	8,0
3	12070	8700	58,1	8,0
4	7690	5920	56,5	9,8
5	12390	7390	62,6	7,9
6	18270	10580	63,3	8,2
7	23740	9440	71,6	7,4
8	12080	7730	61,0	11,3
9	9110	7840	53,8	11,3
10	9190	10820	45,9	11,2
11	19580	24990	43,9	13,7
12	11250	11440	49,6	10,9
13	41410	41700	49,8	12,3
14	3370	3070	52,3	12,7
15	3240	5020	39,2	13,2
16	5700	6140	48,1	13,2

Taula 4.2: *Intensitats mitjanes diàries de trànsit de vehicles lleugers per autopista i carretera donades en vehicles per dia, i preus mitjans de peatge en pessetes per quilòmetre segons el tram.*

En el cas que ens ocupa només direm que el mètode de mínims quadrats s'aplica usualment en els casos d'ajustos com el que tenim, en què no hi ha una "solució exacta", i serveix per trobar la "millor solució" en un cert sentit aproximatiu.

En l'exemple que tractem veurem com relacionar el que tenim amb la coneguda *regressió lineal*, que no és res més que un cas particular del mètode dels mínims quadrats. Per això usem l'equació (4.2) i aïllem el terme $a + b\tau$

$$a + b\tau = \ln(\frac{1}{P(\tau)} - 1).$$

Ara usant els valors de la taula 4.2 per a les parelles $(\tau_k, P(\tau)_k)$, $k = 1 \ldots 16$ ens resulta un sistema lineal sobredeterminat, on a i b no són res més que els coeficients de la *recta de regressió* que ajusta les dades de què disposem.

Si notem per $f_k = \ln(\frac{1}{P(\tau_k)} - 1)$ aleshores els valors de a i b que donen la recta de regressió s'obtenen resolent el sistema lineal 2×2

$$\begin{pmatrix} n & \sum_{k=1}^{n} \tau_k \\ \sum_{k=1}^{n} \tau_k & \sum_{k=1}^{n} \tau_k^2 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sum_{k=1}^{n} f_k \\ \sum_{k=1}^{n} \tau_k f_k \end{pmatrix},$$

on, en el nostre cas, $n = 16$.

Fent els càlculs corresponents, per al nostre cas obtenim el sistema

$$\begin{pmatrix} 16.00 & 169.80 \\ 169.80 & 1871.32 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} -3.6081424 \\ -27.261649 \end{pmatrix},$$

la solució del qual resulta ser $a = -1.91416$ i $b = 0.15912$ i, per tant, la corba logística ajustada a la pràctica es pot prendre com

$$P(\tau) = \frac{1}{1 + e^{-1.914 + 0.159\tau}}.$$

Hem representat aquesta corba juntament amb els valors que han servit per trobar-la a la figura 4.3. Notem que, si mirem el resultat obtingut, quan $\tau = 0$, és a dir, si l'autopista és de franc, encara continua passant una petita proporció de cotxes per la carretera, la qual cosa és natural.

4.4 Models continus versus discrets deterministes

Dins de la modelització contínua suposem que el procés de creixement es fa d'una manera contínua en el temps. Ara bé hi ha organismes, per exemple, en els quals el procés reproductiu es realitza periòdicament (per exemple cada any o cada mes) durant un interval molt curt de temps. Per a aquests casos, ens pot passar que el model logístic, tal com l'hem formulat, no sigui adequat sinó que s'hagi de formular en termes del que s'anomena *equacions en diferències*.

Com que tractar els models donats per equacions en diferències no és el propòsit d'aquest llibre, en aquesta secció únicament presentem una pinzellada del que són. A fi i efecte de mostrar que *poden tenir un comportament qualitatiu molt diferent al del seu "associat" continu*, i per tant en alguns casos ens podem trobar que el nostre model continu no s'ajusti a la realitat,

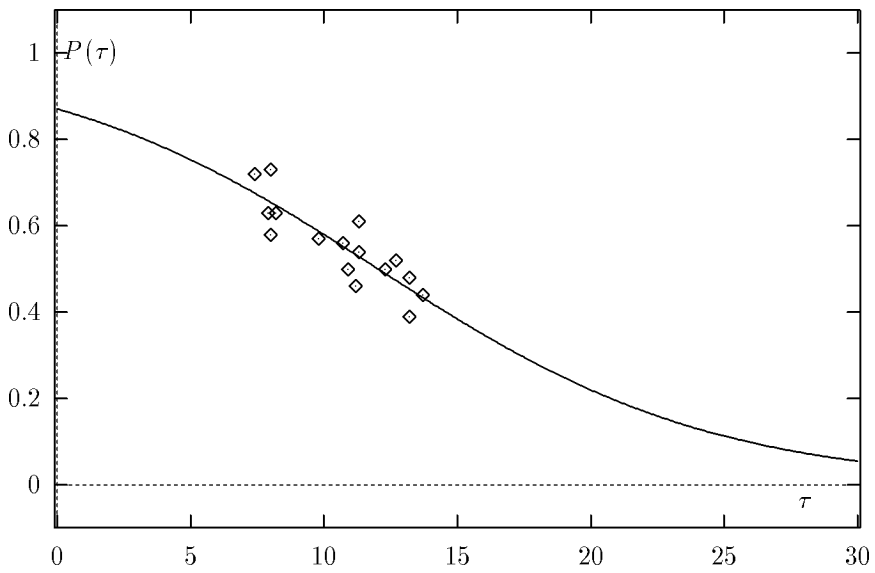

Figura 4.3: *La corba logística i els valors que han servit per trobar-la. Aquests darrers es troben situats en el centre dels seus quadrats respectius.*

ja que, de fet, l'hauríem d'haver formulat en termes d'un model discret determinista usant equacions en diferències.

Si, tal com apuntàvem en el capítol inicial d'aquesta part, les equacions diferencials són equacions on la incognita és una funció, les equacions en diferències són equacions on les incognites són successions.

Suposem que la periodicitat de canvi de la nostra població és de Δt unitats de temps que estem usant. Per tant la població passa discretament pels valors

$$\begin{array}{rcl} N_0 & = & N(0), \\ N_1 & = & N(\Delta t), \\ N_2 & = & N(2\Delta t), \\ \ldots & , & \\ N_n & = & N(n\Delta t), \\ & \ldots & \end{array}$$

On $N_j = N(j\Delta t)$ representarà la població durant el j-è interval de temps que dura Δt.

En un model discret (equació en diferències), el que volem és relacionar un $N(i\Delta t)$ amb altres $N(j\Delta t)$. Usualment és el $N((n+1)\Delta t)$ amb el $N(n\Delta t)$. Per exemple, a partir de l'equació logística, podem obtenir un model discret aproximant la derivada $\dot{N}(t)$ amb quocient incremental

$$\dot{N}(t) \approx \frac{N(t+\Delta t) - N(t)}{\Delta t}.$$

Aleshores, usant aquesta aproximació i avaluant la funció $N(t)$ del model logístic als instants de temps $t = n\Delta t$, obtenim

$$N((n+1)\Delta t) = N(n\Delta t) + rN(n\Delta t)(1 - \frac{N(n\Delta t)}{K})\Delta t,$$

que, usant la simplificació de notació $N_n = N(n\Delta t)$, resulta

$$N_{n+1} = N_n + rN_n(1 - \frac{N_n}{K})\Delta t. \tag{4.3}$$

És a dir que el nombre d'individus de la generació $n+1$ és el nombre d'individus de la generació n més el terme $rN_n(1 - \frac{N_n)}{K})\Delta t$.

Ara veurem que la "versió contínua" i la "versió discreta" del model logístic poden donar resultats molt distints ja des del punt de vista qualitatiu.

Observem en primer lloc que la successió $N_n \equiv K$ és solució del model discret, ja que $N_n = K$, per a $n = 0, 1, 2 \ldots$, i per tant K és un punt d'equilibri. Linealitzem l'equació al voltant de K. Farem per a això el canvi de variable

$$u_n = N_n - K,$$

és a dir, la successió u_n mesura el que s'aparta la succesió N_n del punt d'equilibri K a mesura que n tendeix a infinit.

Així, si donada una desviació inicial petita $u_0 = N_0 - K$ tenim que $\lim_{n\to\infty} u_n = 0$, direm que K és *asimptòticament estable* tal com passava en el model logístic continu, ja que petits canvis respecte de la posició d'equilibri a la llarga s'esmorteeixen i es torna a l'equilibri.

Si substituïm l'expressió $N_n = u_n + K$ a l'equació en diferències per N_n de l'equació (4.3), obtenim la següent equació en diferències per a u_n:

$$u_{n+1} = u_n - \frac{r\Delta t}{K} u_n(u_n + K),$$

que, si la linialitzem (el que equival a pensar intuitivament que si u_n és "petit" podem despreciar els termes en u_n^2), ens porta a l'equació en diferències lineal

$$u_{n+1} = u_n(1 - r\Delta t),$$

de la qual es pot demostrar que essencialment té el mateix comportament qualitatiu que la no linealitzada (4.3).

Aquesta equació es pot resoldre fàcilment ja que

$$\begin{aligned} u_1 &= u_0(1 - r\Delta t) \\ u_2 &= u_1(1 - r\Delta t) = u_0(1 - r\Delta t)^2 \\ u_3 &= u_2(1 - r\Delta t) = u_0(1 - r\Delta t)^3 \end{aligned}$$

i per tant, en general,

$$u_n = u_0(1 - r\Delta t)^n.$$

Amb la qual cosa u_n tendeix a zero si i només si $|1 - r\Delta t| < 1$, és a dir, si $0 < r\Delta t < 2$.

Si $r\Delta t > 2$, la població no torna a l'equilibri, la qual cosa contrasta amb el model continu on K era asimptòticament estable. Tenim per tant que *el model discret és sensible al creixement per càpita r però també a l'interval de temps Δt entre les reproduccions*; segons el valor del seu producte es té una evolució o una altra. D'aquest fet, aparentment simple, en poden derivar comportaments molt complicats.

En aquest sentit caldria esmentar també que, si $r\Delta t = 2$, aleshores l'aproximació lineal no serveix per veure l'estabilitat o la inestabilitat i els termes d'ordre superior no es poden ser negligir.

Finalment notem que, si Δt és petit, els resultats qualitatius obtinguts pel model discret i pel continu són molt semblants. Això és d'esperar ja que, si Δt es petit, el quocient incremental aproxima bé la derivada $\dot{N}$. De fet, si ens hi fixem, l'equació en diferències obtinguda a partir de l'equació diferencial usant aquest fet, no és altra cosa que el que ens dóna l'algorisme d'integració del mètode d'Euler amb pas $h = \Delta t$.

Si el valor de Δt és gran, i no té sentit aproximar les derivades del que seria un model continu per quocients incrementals, això ve a dir que el model continu no és adequat al fet que estem estudiant i, per tant, cal usar un model discret. En aquest cas potser, en lloc d'aproximar derivades per quocients incrementals i obtenir així un model discret, és millor raonar en "termes de generacions". És a dir, usar fets de l'estil: "el nombre d'individus de la generació $n+1$ serà els que hi ha a la generació n més, menys, ...", si bé l'ajut donat en l'aproximació de derivades sobre la base de quocients incrementals d'ordre 1 (com el que hem usat) o d'ordre superior[1] pot ser molt valuós.

4.5 Models d'interacció senzills

Considerarem en aquest apartat dues poblacions, enteses com sempre de manera general, que interaccionen de diferents maneres entre elles. El propòsit és veure com es pot modelar l'evolució de les dues poblacions al llarg del temps, usant un model donat per equacions diferencials ordinàries. Els exemples que tractem són senzills i només donem els trets generals del que podríem dir "l'esquelet del model" d'alguns "casos esterotips". En aplicacions concretes, igual que en la secció anterior per al model logístic, segurament caldrà afegir algun terme o modificar alguns coeficients i fer-los funció del temps o potser funció de les mateixes poblacions.

També pot donar-se el cas que el que estem estudiant no es comporti com un cas tipus dels que presentem sinó que tingui trets comuns a uns quants d'ells. És clar que també de manera anàloga, es poden considerar interaccions entre més de dues poblacions.

En tots aquests exemples notarem amb $N_1(t)$ el nombre d'individus, la densitat o qualsevol magnitud que mesurem, referida a la població P1 a l'instant t, ho farem i anàlogament amb $N_2(t)$ per a la població P2.

Presentem a continuació alguns exemples senzills d'interacció.

[1] Per exemple, una aproximació de segon ordre de $f'(x)$ és $\frac{f(x+h)-f(x-h)}{2h}$ si h és petit.

4.5.1 Exemple 1. Depredador-Presa

El nom d'aquest model (i d'algun altre que veurem) és degut al fet que inicialment es van utilitzar per estudiar situacions ecològiques. Podríem dir que la població P1 corresponia a l'hervívor (*presa*) i la P2 al carnívor (*depredador*). Nosaltres, però, encara que continuarem usant paraules relacionades amb la biologia, les interpretarem de manera general. Així, el fet essencial és que P2 *necessita de P1 per existir i augmentar*, mentre que P1 *subsisteix i augmenta per si sol* usant recursos que té de manera suficient en el seu entorn.

En aquest model fem les hipòtesis següents:

- La població P1 s'alimenta de reserves il·limitades d'aliment i rep el nom de *presa*.
- La població P2 s'alimenta fonamentalment de P1 i rep el nom de *depredador*.
- Si hi ha prou existències de P1, P2 incrementa en nombre. En cas contrari, disminueix.
- Si P1 es troba aïllat, sense que P2 hi pugui accedir, la població P1 creix logísticament.
- Si deixem P2 aïllat, sense accés a P1, la població P2 decreix segons la llei malthusiana.
- Com és natural, la presència mútua (la interacció) ha de fer augmentar la població P2 i fer disminuir la de P1.
- Per modelar la interacció, experimentalment s'ha provat que és eficient suposar que els encontres entre les dues espècies succeeixen a causa d'una mena de "recerca i fugida", que és proporcional a la quantitat de maneres en que es poden trobar P1 i P2. Es a dir, proporcional a $N_1 N_2$.

Tenint en compte tot això obtenim el model següent:

$$\begin{cases} \dot{N}_1 &= rN_1(1-\frac{N_1}{K}) - \alpha N_1 N_2, \\ \dot{N}_2 &= -cN_2 + \beta N_1 N_2, \end{cases}$$

on r, K, α, c i β són constants positives.

És fàcil reconèixer tot el que hem anat dient en els termes d'aquest model: el creixement logístic a ritme r i saturació K per a P1; el decreixement malthusià a ritme c per a P2; finalment els termes d'interacció, positiu per a P2 i negatiu per a P1, cadascun d'ells amb el seu grau d'aprofitament α i β. El coeficient β usualment es diu *eficiència de captura pel depredador*, ja que ve a ser un tant per u del profit que treu de cada captura per unitat de temps.

Aquest model suposa l'hàbitat tancat, sense factors externs. A la pràctica poden donar-se fets més complicats, com per exemple que les preses s'amaguin bé dels depredadors, que els depredadors a més mengin d'altres coses, o que la presa no disposi de quantitats il·limitades d'aliment. Tot això es pot anar afegint al nostre model. Però moltes vegades el que és realment important és saber trobar un model, com més senzill millor, que s'apropi al màxim a allò que realment succeeix.

4.5.2 Exemple 2. Competició

En aquest model considerem els fets següents:

- Les dues poblacions, P1 i P2, lluiten per la mateixa font d'alimentació.
- Si una espècie està aïllada de l'altra, té creixement logístic.
- La interacció, que modelem igual que per al cas depredador-presa, interfereix a les dues espècies i per tant es negativa per a totes dues.

Tenint en compte tot això obtenim el model següent:

$$\begin{cases} \dot{N}_1 &= rN_1(1-\frac{N_1}{K}) - \alpha N_1 N_2, \\ \dot{N}_2 &= sN_2(1-\frac{N_2}{L}) - \beta N_1 N_2, \end{cases}$$

on r, K, α, s, L i β són constants positives.

Les constants α i β representen ara relacions competitives d'avantatge per càpita d'una espècie sobre l'altra. Si $\beta > \alpha$ direm que P1 és un competidor més eficient que P2, i a l'inrevés si $\alpha > \beta$.

4.5.3 Exemple 3. Combats

En aquest model considerem els fets següents:

- Suposem que P1 i P2 són dues forces militars oposades en situació de guerrilla.
- Suposem que les forces estan aïllades, sense reforços.
- De manera semblant a l'exemple 1, les pèrdues a cada bàndol es donen en proporció al nombre d'encontres.
- A més, a cada bàndol hi ha pèrdues internes degudes per exemple a accidents o desercions, que suposarem en proporció a la quantitat de tropa.

Tenint en compte tot això obtenim el model següent:

$$\begin{cases} \dot{N}_1 &= -aN_1 - \alpha N_1 N_2, \\ \dot{N}_2 &= -bN_2 - \beta N_1 N_2, \end{cases}$$

on a, b, α, i β són constants positives.

Si hi haguessin reforços de tropes constats per unitat de temps, caldria afegir aquests termes a les equacions i sumar una quantitat constant a l'equació corresponent.

Si el que hi ha és un reclutament esporàdic en un instant de temps donat, per modelar-lo convenientment faríem la integració de la trajectòria partint de la situació inicial fins a arribar a aquest instant de temps. Llavors modificaríem P1 o P2 afegint-hi el reforç i continuaríem la integració amb el mateix model que teníem anteriorment. És a dir, s'observaria un salt en la trajectòria, ja que de fet hem saltat d'una primera trajectòria, donada per les condicions

inicials, a una altra que té com a condicions inicials el resultat de la integració de la primera més els reforços que han tingut les tropes.

Si bé la interació entre guerrilles es modela de manera semblant al cas depredador-presa, ja que el procés de "recerca i fugida" és semblant, en cas d'un combat entre exèrcits convencionals, en el qual podem suposar que les dues forces es veuen o si més no "escombren" tota una localització, les pèrdues degudes a l'enemic són proporcionals a la força que té i, per tant, el model més senzill és

$$\begin{cases} \dot{N}_1 &= -aN_1 - \alpha N_2, \\ \dot{N}_2 &= -bN_2 - \beta N_1. \end{cases}$$

4.5.4 Anàlisi de l'evolució

Abans de tractar alguns dels models anteriors amb més detall, farem uns breus comentaris qualitatius sobre el seu comportament.

Tots els exemples que hem presentat són un cas especial de l'anomenat *model quadràtic*:

$$\begin{cases} \dot{N}_1 &= N_1(a_1N_1 + b_1N_2 + d_1), \\ \dot{N}_2 &= N_2(a_2N_1 + b_2N_2 + d_2), \end{cases}$$

del qual, malgrat la seva senzillesa, és molt difícil dir el que passa en general. Ara bé, hi ha un fet i una conseqüència important per al dibuix del seu retrat de fase:

> Els eixos $N_1 = 0$ i $N_2 = 0$ són invariants, ja que les condicions $N_1 = 0$ i $N_2 = 0$ impliquen $\dot{N}_1 = 0$ i $\dot{N}_2 = 0$ respectivament.
>
> A causa de la unicitat de solucions, com que les òrbites no es poden creuar, si una solució comença al quadrant positiu ($N_1 \geq 0$ i $N_2 \geq 0$) es mantindrà sempre al mateix quadrant positiu. Es diu que aquest quadrant és invariant per al flux.

Com a exemple, farem una petita anàlisi del model de competició utilitzant isoclines. Proposem el lector que faci anàlisis semblants amb altres models.

Recordem que el model de competició és

$$\begin{cases} \dot{N}_1 &= rN_1(1 - \frac{N_1}{K}) - \alpha N_1N_2, \\ \dot{N}_2 &= sN_2(1 - \frac{N_2}{L}) - \beta N_1N_2. \end{cases}$$

Per al que podríem anomenar el "cas genèric" tenim d'entrada tres punts d'equilibri, que, recordem, són els llocs on el camp s'anul·la (és a dir, $\dot{N}_1 = \dot{N}_2 = 0$). Són

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \begin{pmatrix} K \\ 0 \end{pmatrix}, \quad \begin{pmatrix} 0 \\ L \end{pmatrix}.$$

A més podem tenir el

$$\begin{pmatrix} \bar{N}_1 \\ \bar{N}_2 \end{pmatrix},$$

on $\bar{N}_1$ i $\bar{N}_2$ són solucions del sistema lineal

$$\begin{aligned} \bar{N}_1 &= \frac{s}{\beta} - \frac{s}{\beta L}\bar{N}_2, \\ \bar{N}_2 &= \frac{r}{\alpha} - \frac{r}{\alpha K}\bar{N}_1, \end{aligned}$$

en el cas que els dos valors, $\bar{N}_1$ i $\bar{N}_2$, resultin positius; ja que normalment un valor de població negatiu no té sentit.

Els llocs on el camp és vertical, és a dir, on no hi ha component horitzontal i per tant s'obtenen fent $\dot{N}_1 = 0$, i els llocs on el camp és horitzontal, és a dir, on no hi ha component vertical i per tant s'obtenen fent $\dot{N}_2 = 0$, són les corbes que anomenem *isoclines*[2] i que en aquest cas són rectes, ja que si fem $\dot{N}_1 = 0$ obtenim

$$N_1(r - \frac{r}{K}N_1 - \alpha N_2) = 0,$$

d'on queda $N_1 = 0$, o bé $N_2 = \frac{-r}{\alpha K}N_1 + \frac{r}{\alpha}$.

Mentre que, si fem $\dot{N}_2 = 0$, obtenim

$$N_2(s - \frac{s}{L}N_2 - \beta N_1) = 0,$$

que dóna $N_2 = 0$ o bé $N_2 = -\frac{\beta L}{s}N_1 + L$.

Per tant, a part dels eixos, resulten dues rectes més, i allà on es creuen tenim el punt d'equilibri donat per $\bar{N}_1$ i $\bar{N}_2$ (ja que s'anul·len les components tant vertical com horitzontal del camp); la seva posició relativa de les quals pot prendre una de les quatre formes indicades a la figura 4.4

En fer els retrats de fase cal tenir en compte que, quan una òrbita talla una d'aquestes rectes isoclines, ho fa paral·lelament a l'eix N_1 si talla a $\dot{N}_2 = 0$, o paral·lelament a l'eix N_2 si talla a $\dot{N}_1 = 0$.

Si, per exemple, considerem el cas superior dret de la figura 4.4 veiem que:

- Si $N_2 = 0$ (és a dir, ens situem a l'eix N_1), tenim creixement logístic asimptòtic a K.

- Si $N_1 = 0$ (és a dir, ens situem a l'eix N_2), tenim creixement logístic asimptòtic a L.

- Quan una isoclina talla un dels eixos, i el punt de tall no és un punt d'equilibri, per continuïtat, el sentit del camp en els punts de la isoclina propers a l'eix ha de ser el mateix que el que té a l'eix.

- El sentit del camp en una isoclina sempre és horitzontal o vertical. De fet, però, quan ens acostem al punt d'equilibri, el seu mòdul es va fent petit, s'anul·la en el punt d'equilibri i, per altra banda haurà canviat de sentit ja que haurà passat per un zero simple de $\dot{N}_1$ o de $\dot{N}_2$.

[2] De fet les isoclines es defineixen com les corbes on el camp té un pendent constant i poden servir per fer un bon esbós del retrat de fase. En el nostre cas, però, només considerarem els llocs geomètrics on el camp és vertical o horitzontal.

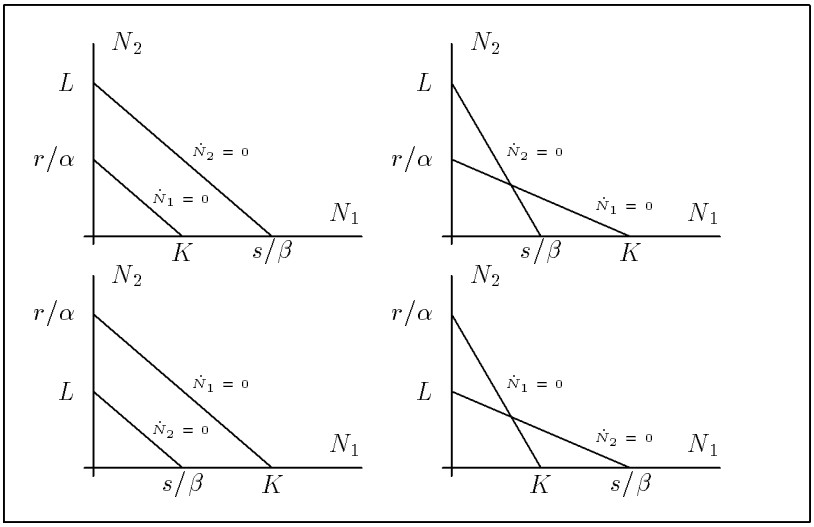

Figura 4.4: *Possibilitats de les rectes isoclines en els models quadràtics.*

- Si els dos darrers punts no han quedat clars, podem indicar que a causa de les isoclines, el quadrant positiu queda dividit en quatre parts, cadascuna de les quals té associat un signe per a les components $\dot{N}_1$ i $\dot{N}_2$. Aquest fet ens determina també la direcció i el sentit de les òrbites.

Les consideracions anteriors permeten dibuixar el retrat de fase aproximatiu de la figura 4.5.

4.6 Estudi analític d'alguns models

En aquesta secció tractarem de l'estudi d'alguns models que, malgrat la seva simplicitat, s'ajusten bé al seu propòsit. Alguns d'ells queden compresos en la secció anterior, però ara els adaptarem a fets concrets i en deduïrem algunes consequències analítiques. Es a dir, trobarem algunes de les seves propietats sense fer exploracions de tipus numèric amb els integradors vistos al capítol 3.

L'esperit d'aquesta secció és que el lector vegi essencialment com es poden modelar diferentes interaccions i que, un cop obtingut el model, de vegades és possible, i a més molt profitós, un estudi analític que ens doni propietats més generals que les obtingudes usant un integrador numèric.

4.6.1 Desplaçament del punt d'equilibri en un model depredador-presa

El fenòmen que ara presentarem té el seu origen a la Primera Guerra Mundial. Durant la dècada de 1920, un biòleg italià anomenat Umberto d'Ancona estudiava els diferents tipus de peix del mar Mediterrani i les seves interaccions. Es va adonar que, durant el temps de guerra, els percentatges de peix dolent (en general peixos depredadors del Mediterrani que no es compren per al consum) respecte de peix bo (el peix que es ven al mercat) havia augmentat de manera

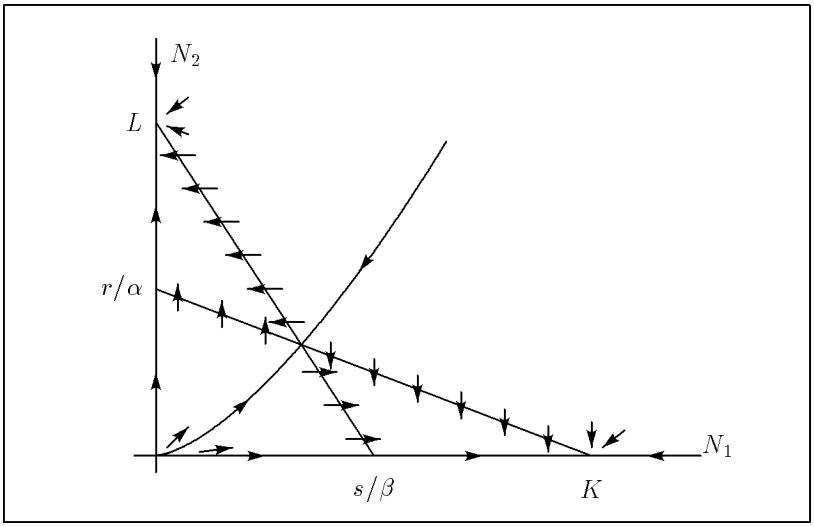

Figura 4.5: *Esbós d'un retrat de fase del model de competició.*

considerable. En particular el tant per cent de peix dolent al port italià de Fiume durant els anys 1914-1923 és el següent,

1914	1915	1916	1917	1918
11.9%	21.4%	22.1%	21.2%	36.4%
1919	1920	1921	1922	1923
27.3%	16.0%	15.9%	14.8%	10.7%

Per d'Ancona era clar que la raó era deguda a la gran reducció de pesca durant els anys de guerra, però no hi trobava l'explicació lògica.

Si bé a causa de la disminució de flota pesquera els depredadors tenien molt més menjar al seu abast, també hi havia d'haver al mateix temps molts més peixos bons. En una primera visió intuïtiva no s'explica per què la pesca beneficia percentualment més als peixos bons que als dolents. Es trobava davant d'un problema interessant ja no solament des del punt de vista biològic sinó també industrial.

Un cop esgotades, sense èxit, totes les possibles explicacions biològiques, va portar el problema al matemàtic italià Vito Volterra, el qual el va formular en termes del que hem anomenat un model depredador-presa. Volterra va fer les hipòtesis següents:

- La població total de peix a l'instant t, la suposem dividida en dos grups. Els peixos bons per al consum, $N_1(t)$, i els que no ho són, $N_2(t)$.

- S'escau que els peixos dolents per al consum són generalment els depredadors que s'alimenten de peix bo, i els bons són les preses, que no s'alimenten de peix.

- Els peixos bons no tenen competència entre ells, ja que la seva font d'aliment, el mar, la considerem prou abundant, i la seva distribució no és gaire densa. Per tant podem pensar que creixen segons la llei malthusiana. (O si més no tenen creixement logístic, però donada la seva poca densitat sempre es troben en "la part malthusiana" d'aquest creixement.)

- El nombre de contactes entre les dues espècies és proporcional al producte d'individus de cada espècie, tal com havíem indicat en el model depredador-presa general. A més, el contacte és positiu per al depredador i negatiu per a la presa.

- El creixement del depredador, de fet, únicament és fruit dels encontres amb les preses.

- La mort del les preses, de fet, únicament és fruit d'encontres amb els depredadors. Mentre que els depredadors moren de manera natural i proporcional al seu nombre.

Amb totes aquestes hipòtesis, Volterra va obtenir el sistema d'equacions següent, que ens modela la interacció entre les dues classes de peixos quan no hi ha pesca,

$$\begin{cases} \frac{dN_1}{dt} &= aN_1 - bN_1N_2, \\ \frac{dN_2}{dt} &= -cN_2 + dN_1N_2, \end{cases}$$

on a, b, c i d són constants positives.

El que farem a continuació és estudiar algunes de les propietats d'aquest model i veure com hi influeix la pesca.

En primer lloc observem que, en el retrat de fase d'aquest model, els eixos $N_1 = 0$ i $N_2 = 0$ són invariants i, per tant, tota òrbita que comenci en el primer quadrant hi roman per sempre més. De fet, si $N_2 = 0$, ens queda per la primera equació $\dot{N_1} = aN_1$, que com sabem té per solució $N_1(t) = N_1(0)e^{at}$; si $N_1 = 0$, de la segona equació tenim $N_2(t) = N_2(0)e^{-ct}$, cosa que ens dóna les trajectòries dels eixos parametritzades explícitament pel temps.

El nostre model té dos punts d'equilibri que s'obtenen resolent, com sempre, el sistema d'equacions, $\dot{N_1} = 0$ i $\dot{N_2} = 0$. Els que obtenim són $(0, 0)$ i $(c/d, a/b)$.

Si dividim les dues equacions del model, resulta l'equació diferencial

$$\frac{dN_2}{dN_1} = \frac{-cN_2 + dN_1N_2}{aN_1 - bN_1N_2},$$

la qual, traient N_2 factor comú al numerador del costat dret i N_1 al denominador del mateix costat, resulta que és de variables separades:

$$\frac{a - bN_2}{N_2}dN_2 = \frac{-c + dN_1}{N_1}dN_1,$$

és a dir,

$$(\frac{a}{N_2} - b)dN_2 = (\frac{-c}{N_1} + d)dN_1,$$

i per tant, integrant els dos costats respecte de les seves respectives variables, ens queda

$$a \ln N_2 - bN_2 + c \ln N_1 - dN_1 = C,$$

on C és una constant arbitrària.

Finalment prenent exponencials resulta

$$\frac{N_2^a}{e^{bN_2}}\frac{N_1^c}{e^{dN_1}} = K,$$

on $K = \exp C$.

Fixem-nos que si considerem la funció de dues variables

$$F(N_1, N_2) = \frac{N_2^a}{e^{bN_2}}\frac{N_1^c}{e^{dN_1}},$$

la relació que tenim, $F(N_1, N_2) = K$, és el tall de la gràfica de la funció F amb el nivell K. Per això es diu que en variar K s'obtenen les *corbes de nivell* de la funció F.

Usant tècniques de càlcul infinitesimal es pot veure que, donat un valor de $K > 0$, si hi ha parelles (N_1, N_2) que satisfan la relació anterior, formen una corba tancada (vegeu [3] o [4]). És a dir, podem pensar que la gràfica de $F(N_1, N_2)$ és com "la superfície d'una muntanya". Si la tallem per un pla proper a la base surten corbes tancades. Aquestes corbes es van fent més petites a mesura que tallem per un nivell més proper al cim. Finalment, quan tallem per un pla que toca només el cim, s'obté un sol punt. Mentre que per a talls a alçades superiors a les del cim no s'obté res. En el cas concret que ens ocupa es té que si

$$K = \bar{K} = \frac{(a/b)^a}{e^a}\frac{(c/d)^c}{e^c},$$

aleshores tenim un sol punt. Si $K > \bar{K}$ no s'obté res, i si $K < \bar{K}$ s'obté una corba tancada.

Resulta doncs que la funció $F(N_1, N_2)$ es comporta de la mateixa manera que el radi quan integrem el sistema d'equacions donat per `cercle` de la pàgina 98. $F(N_1, N_2)$ és una integral primera pel nostre sistema depredador-presa. Les òrbites solució han d'estar contingudes en una corba de nivell, o bé ser el punt d'equilibri.

Una corba de nivell, en aquest cas, només pot contenir una òrbita ja que, si n'hi hagués més, forçosament hauria de contenir també algun punt d'equilibri. La qual cosa no és possible pel fet que el punt $(c/d, a/b)$ és l'únic punt d'equilibri del sistema tret del $(0, 0)$.

Si les corbes de nivell de F del nostre exemple només poden contenir una òrbita solució del sistema, resulta que les corbes de nivell són les òrbites del retrat de fase del sistema. Només cal inspeccionar el sentit del camp en un punt o mirar el camp en els eixos per saber la manera en què es recorren les trajectòries. Obtenim així el retrat de fase representat a la figura 4.6.

A les òrbites que formen una corba tancada se'ls diu *òrbites periòdiques*. Una òrbita periòdica té la propietat que, donada una condició inicial damunt d'ella en un temps t_0 qualsevol, $(N_1(t_0), N_2(t_0))$, existeix un cert valor de temps, T, tal que passat aquest valor es torna a ser en el mateix punt que per al temps t_0. És a dir,

$$(N_1(t_0 + T), N_2(t_0 + T)) = (N_1(t_0), N_2(t_0)).$$

El mínim valor de temps T que compleix aquesta relació s'anomena el *període de l'òrbita*.

Tornant a l'estudi del model, notem que els valors donats per D'Ancona corresponen a una determinada mitjana de depredadors i de preses mesurats al llarg d'un any. Cal trobar les

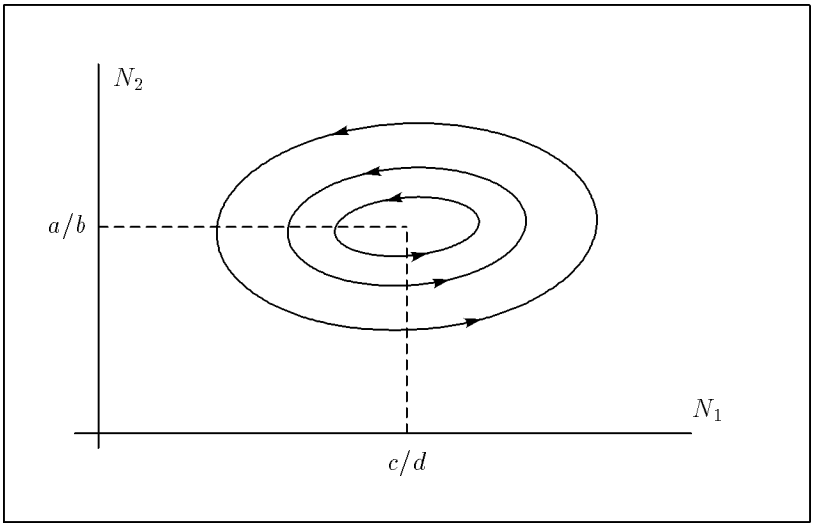

Figura 4.6: *Representació qualitativa de les corbes de nivell que coincideixen amb les òrbites del model desenvolupat per Volterra.*

mitjanes de cada espècie en el model proposat per Volterra. Per això ens cal definir el que són els *valors mitjans damunt d'una òrbita periòdica.*

Suposem que $(x(t), y(t))$ és una solució periòdica de període T, d'un model

$$\begin{cases} \dot{x} &= f(x,y), \\ \dot{y} &= g(x,y). \end{cases}$$

Es defineix el *valor mitjà* de x, que notarem amb $\bar{x}$ com a

$$\bar{x} = \frac{1}{T}\int_0^T x(t)dt.$$

La definició de mitjana té un significat intuïtiu ben clar ja que

$$\int_0^T x(t)dt$$

és l'àrea que la gràfica de la funció $x(t)$ deixa entre ella i l'eix d'abscisses en exactament un període[3]. El valor $\bar{x}$ es calcula de manera que $\bar{x}T$ resulti el valor d'aquesta àrea. Per tant, el valor mitjà $\bar{x}$ és l'alçada que té un rectangle de base $(0,T)$ i de la mateixa àrea que la deixada per la funció (vegeu la figura 4.7).

[3] És un exercici senzill de càlcul infinitesimal veure que la mitjana $\bar{x}$ no depèn de l'interval d'integració, sempre que sigui un període complet. És a dir, $\int_0^T x(t)dt = \int_a^{T+a} x(t)dt$ per a qualsevol nombre real a.

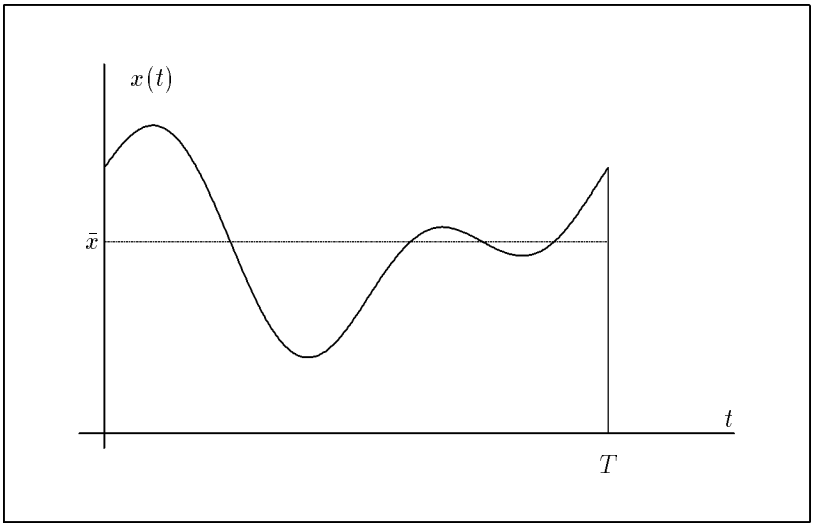

Figura 4.7: *El valor $\bar{x}$ correspon la mitjana de la funció $x(t)$ de 0 a T. L'àrea del rectangle $(T,0)\times(0,\bar{x})$ és la mateixa que hi ha entre l'eix d'abscisses i la funció $x(t)$ per $t\in(0,T)$.*

És clar que una definició anàloga es té per a la mitjana de y, $\bar{y}$, o de qualsevol altra variable que pugui tenir associada l'òrbita periòdica.

Vegem que en el nostre cas particular tenim que per a qualsevol òrbita periòdica

$$\bar{N_1}=\frac{c}{d}\quad\text{i que}\quad\bar{N_2}=\frac{a}{b}.$$

És a dir, que els valors mitjans damunt d'una òrbita periòdica qualsevol del nostre model, coincideixen amb les coordenades del punt d'equilibri que hi ha al seu interior.

Per això comencem agafant la primera equació del nostre model,

$$\frac{dN_1}{dt}=aN_1-bN_1N_2.$$

Dividim els dos costats per $N_1(t)$ i el resultat l'integrem des de 0 fins a T:

$$\int_0^T\frac{\dot{N_1}(t)}{N_1(t)}=\int_0^T(a-bN_2(t))dt.$$

Ara, per al primer membre d'aquesta equació tenim

$$\int_0^T\frac{\dot{N_1}(t)}{N_1(t)}=\ln N_1(T)-\ln N_1(0),$$

però, com que les òrbites són periòdiques de període T, resulta que $N_1(T)=N_1(0)$ i per tant s'obté que aquest primer membre és 0.

Tenint en compte això, del segon membre obtenim

$$\int_0^T bN_2(t)dt = \int_0^T adt = aT,$$

d'on, traient la constant b fora de la integral i dividint la igualtat resultant per bT, s'arriba a $\bar{N}_2 = a/b$, aplicant la definició de mitjana de N_2.

Passa el mateix amb la segona de les equacions del model. Dividint-la per $N_2(t)$ i procedint com en els paràgrafs anteriors, s'obté $\bar{N}_1 = c/d$.

Ara a introduirem la pesca en el nostre model. Suposarem que els vaixells de pesca surten a pescar amb xarxa i per tant agafen peixos bons i dolents a un ritme proporcional a la seva presència. Tindrem, doncs, que la pesca fa decréixer la població de peixos bons a un ritme $\epsilon N_1(t)$ i la de dolents a un ritme $\epsilon N_2(t)$, on la constant ϵ reflecteix la intensitat de pesca que s'està duent a terme. És a dir, ϵ està essencialment relacionat amb el nombre de vaixells que surten a pescar i amb el monbre de xarxes que cada dia es llancen a l'aigua.

S'obté d'aquesta manera el següent model que té en compte la pesca:

$$\begin{cases} \frac{dN_1}{dt} &= aN_1 - bN_1N_2 - \epsilon N_1, \\ \frac{N_2}{dt} &= -cN_2 + dN_1N_2 - \epsilon N_2, \end{cases}$$

i que podem escriure com

$$\begin{cases} \frac{dN_1}{dt} &= (a-\epsilon)N_1 - bN_1N_2, \\ \frac{N_2}{dt} &= -(c+\epsilon)N_2 + dN_1N_2. \end{cases}$$

Per tant, si ϵ és petit, és a dir, si hi ha un ritme moderat de pesca, es té que $a - \epsilon > 0$ i, qualitativament, aquest model és el mateix que el que tenim inicialment sense pesca. Les constants a i c, positives, del model inicial, han estat substituïdes respectivament per $a - \epsilon$ i $c + \epsilon$ que continuen essent positives si ϵ és prou petit.

Si la pesca no és moderada, ϵ és gran, i per tant es pot donar el cas que, $a - \epsilon < 0$, la qual cosa portaria al fet que la població, N_1, decreixeria sempre en el temps fins a desaparèixer (ja que la seva derivada seria sempre negativa) i com a consequència desapareixeria també el depredador N_2.

Suposant doncs que $a - \epsilon > 0$, podem aplicar tot el que hem obtingut pel model que no considerava la pesca. En particular les mitjanes de N_1 i de N_2, quan hi ha un ritme de pesca ϵ, coincidiran amb el punt d'equilibri que ara resulta ser

$$\bar{N}_1 = \frac{c+\epsilon}{d}, \qquad \bar{N}_2 = \frac{a-\epsilon}{b}.$$

Això explica el que va observar D'Ancona ja que, en mitjana (que és el que observava D'Ancona per a cada any), un ritme de pesca, ϵ, prou moderat per no canviar qualitativament l'evolució del sistema és més beneficiós per als peixos bons que per als dolents.

La mitjana de peixos dolents sense haver-hi pesca és de a/b, mentre que amb un ritme ϵ de pesca aquesta mitjana passa a ser de $(a-\epsilon)/b$. Anàlogament, la mitjana de peixos bons passa de c/d sense pesca a $(c+\epsilon)/d$ quan hi ha un ritme moderat de pesca.

El resultat és que un ritme moderat de pesca és més beneficiós per als peixos bons que per als dolents. Aquest fet biològic de desplaçament del punt d'equilibri és conegut com el *principi de Volterra* i ens explica matemàticament el perquè de les observacions de D'Ancona.

El principi de Volterra té aplicacions molt clares a l'hora de saber si podem intervenir en un ecosistema o no. També hi ha casos per als quals no s'obtenen uns resultats tan satisfactoris des del punt de vista pràctic, com passa en el cas dels peixos del Mediterrani que acavem d'analitzar. Una prova d'això són els resultats que es poden obtenir en el tractament de plagues d'insectes; i com a confirmació es té que durant l'any 1868 es va introduir als Estats Units, de manera accidental, un insecte (*Icerya purchasi*) provinent d'Austràlia, que amenaçava, de destruïr la indústria cítrica del país. A Austràlia aquest insecte té un altre insecte escarabat associat (*Novius cardinalis*), que és el seu depredador natural i que manté l'equilibri.

Com que als Estats Units el depredador natural no existia, l'insecte introduït creixia a ritme malthusià. Per la qual cosa es va haver d'introduir l'escarabat depredador i aquest va reduir els insectes dolents a un nivell acceptable.

Més tard, quan es va descobrir el DDT per a tractar plagues d'insectes, el resultat va ser que, d'acord amb el principi de Volterra, va créixer la població d'insectes dolents i va decréixer la de bons. Ja que en aquest cas els insectes dolents són les preses i els bons són els depredadors.

La consequència és, doncs, que si l'insecticida no pot acabar amb la plaga d'insectes (arribar a $a - \epsilon < 0$), pot passar que els resultats no siguin els esperats de la seva aplicació.

4.6.2 Models d'interacció per combat

Els models que presentem a continuació tenen el seu origen a la Primera Guerra Mundial i varen ser introduïts per F.W. Lanchester. Estan, en principi, pensats per a competicions que poden ser des d'una batalla a una guerra. Però, com és natural, també poden servir per a modelitzar d'altres competicions o jocs que es desenvolupin d'una manera semblant a la que exposarem.

En els subapartats següents notarem amb N_1 i N_2 les forces dels dos bàndols que entren en combat. En principi suposarem que aquestes quantitats estan directament relacionades amb el nombre d'homes que formen la tropa de cada bàndol. Ara bé, també es podria pensar que representen el potencial ofensiu del seu bàndol, és a dir, que a més de la tropa, d'alguna manera s'hi mesura també la quantitat i el tipus d'armament del qual disposen. Notem que, si el potencial ofensiu quant a armament és molt semblant, llavors és plausible el considerar que $N_1(t)$ i $N_2(t)$ són el nombre de combatents dels respectius bàndols a l'instant t.

Combat convencional

En aquest primer subapartat suposarem els fets següents:

- Tenim dos exèrcits convencionals N_1 i N_2.
- Els exèrcits combaten de manera aïllada, sense reforços per part del seus respectius bàndols.
- Les seves pèrdues operacionals degudes a desercions o bé a accidents interns són negligibles. Per tant les úniques pèrdues que té cada bàndol són les degudes al combat.

- Cada exèrcit produeix baixes al seu oponent en proporció directa a la capacitat ofensiva que té. Això es justifica per la manera d'actuar dels exercits convencionals. Podem pensar que usen "de manera uniforme" la seva força a fi de fer una batuda a la regió on es creu que hi ha l'enemic.

Mitjançant aquests fets podem escriure el model d'interacció entre dos exèrcits convencionals:

$$\begin{cases} \frac{dN_1}{dt} &= -bN_2, \\ \frac{dN_2}{dt} &= -cN_1. \end{cases}$$

Els coeficients constants i positius, b i c, se'ls anomena *coeficients d'efectivitat* del seu respectiu exèrcit. Així en el nostre cas b és el coeficient d'efectivitat de l'exèrcit N_2, mentre que c ho és de N_1 i vénen a representar el grau de preparació i habilitat de cada una de les tropes.

Els coeficients d'efectivitat són difícils d'estimar sobre tot a priori. Normalment les anàlisis de les batalles a posteriori, un cop han finalitzat, en donen els valors d'una manera molt més precisa encara que del tot inútil per a la batalla que ja s'ha acabat. Si no es disposa de cap altra experiència semblant es pot suposar que

$$b = r_{N_2} p_{N_2}, \qquad c = r_{N_1} p_{N_1},$$

on la r representa el "ritme de foc" mesurat en trets per combatent i per dia i la p és la probabilitat que un tret mati un oponent.

Continuem fent l'anàlisi del nostre model. Per això dividim les dues equacions del model i obtenim

$$\frac{dN_2}{dN_1} = \frac{cN_1}{bN_2},$$

que és una equació de variables separades que integrada ens dóna

$$bN_2^2(t) - cN_1^2(t) = K, \tag{4.4}$$

on K és una constant que podem determinar en el moment inicial de la batalla, ja que la relació $N_2^2(t) + N_1^2(t) = K$ s'ha de complir per a tot t i en particular per a $t = 0$:

$$K = bN_2^2(0) - cN_1^2(0).$$

Això ens permet escriure l'equació (4.4) com

$$b(N_2^2(t) - N_2^2(0)) = c(N_1^2(t) - N_1^2(0)),$$

la qual cosa és coneix com la *llei dels quadrats de Lanchester o llei hiperbòlica de combat.*

Les òrbites del retrat de fase que representen l'evolució de totes les possibles batalles, les obtenim en dibuixar les gràfiques de l'equació (4.4) si varia K. Per això veiem que si $K = 0$ s'obté un parell de rectes, mentre que si $K \neq 0$ el que s'obté són hipèrboles.

Donat el significat del nostre model només ens interessen els resultats dins del primer quadrant, és a dir, els que tenen $N_1 \geq 0$ i $N_2 \geq 0$. El retrat de fase d'aquest model queda representat en la figura 4.8. El sentit de les òrbites es determina fàcilment, ja que com que no

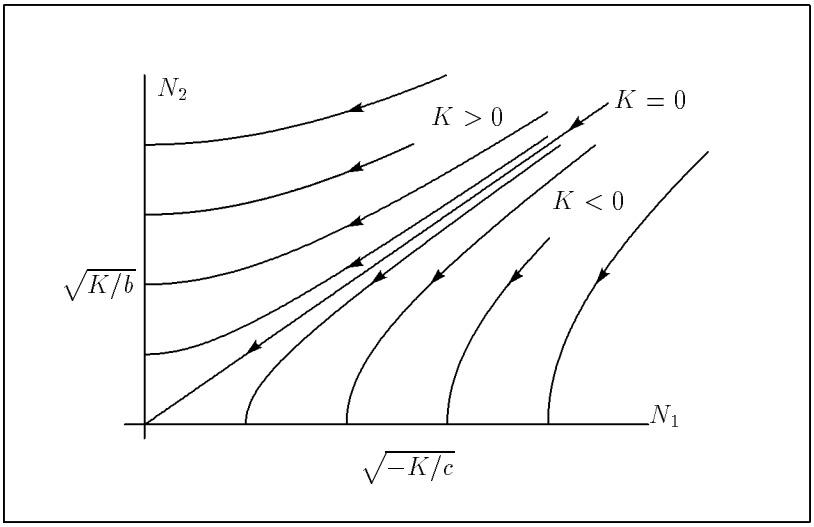

Figura 4.8: *Retrat de fase del model de combat convencional. Les òrbites són branques d'hipèrboles.*

hi ha reforços per part dels bàndols respectius, les funcions $N_1(t)$ i $N_2(t)$ han de ser monòtones decreixents respecte del temps t.

Si avaluant la K a l'instant inicial resulta que $K = 0$, aleshores aplicant l'equació (4.4) es té que per a tot t

$$bN_2^2(t) - cN_1^2(t) = 0,$$

i per tant si definim com "guanyar la batalla" el fet d'aniquilar l'exèrcit enemic, tenim que si un exèrcit desapareix també ho fa l'altre, es a dir, si $K = 0$ la batalla acabarà en un empat i l'òrbita que es segueix és la representada per la línia recta que separa els dos feixos d'hipèrboles a la figura 4.8.

En cas que $K > 0$, mitjançant l'equació (4.4) es té que per a tot temps, t,

$$bN_2^2(t) - cN_1^2(t) > 0.$$

Així doncs, N_2 guanyarà la batalla, ja que sempre $bN_2^2(t) > cN_1^2(t)$, i quan s'arribi al temps final de la batalla, t_f, tindrem $N_1(t_f) = 0$ i, per tant, $bN_2^2(t_f) = K$. El nombre de supervivents del bàndol N_2 serà $\sqrt{K/b}$.

Les òrbites de les possibles batalles amb $K > 0$, que dónen com a guanyador el bàndol N_2, queden representades a la figura (4.8) per les hipèrboles que es troben per damunt de la recta d'empat, obtinguda per a $K = 0$.

Anàlogament, si $K < 0$ aleshores per a tot t es té $bN_2^2(t) < cN_1^2(t)$ i, per tant, N_1 serà el bàndol guanyador amb $\sqrt{-K/c}$ supervivents.

Com ja hem vist, el valor de K queda determinat en el moment inicial de la batalla i ve donat per $K = bN_2^2(0) - cN_1^2(0)$. Per tant, si ens posem del costat del bàndol N_2, el que ens

interessarà serà que K sigui positiu, és a dir,

$$bN_2^2(0) > cN_1^2(0),$$

la qual cosa la podem escriure com

$$\left(\frac{N_2(0)}{N_1(0)}\right)^2 > \frac{c}{b}.$$

Aquesta inequació involucra les relacions $N_2(0)/N_1(0)$ i c/b però la primera hi apareix elevada al quadrat. Això fa que si b és el doble de c, l'exèrcit 2 dobla en força l'exèrcit 1. Però, si N_2 és el doble de N_1, llavors l'exèrcit 2 quatriplica en forca l'exèrcit 1. És a dir, com que els coeficients b i c representen de fet el grau de preparació de cada exèrcit, la llei dels quadrats de Lanchester diu que, si un exèrcit té el doble d'homes que el seu contrincant, a aquest li caldrà una preparació quatre vegades superior per compensar la batalla.

Finalment, veiem com es pot calcular l'evolució de la batalla en funció del temps. Agafem la primera de les equacions del model i la derivem respecte del temps

$$\frac{d^2N_1}{dt} = -b\frac{dN_2}{dt}.$$

Si ara usem la segona equació del model per substituir la derivada de N_2 que apareix obtenim

$$\frac{d^2N_1}{dt} - bcN_1 = 0,$$

que és una equació diferencial lineal i homogènia de segon ordre a coeficients constants. Tenint en compte que, per a $t = 0$, N_1 ha de valer $N_1(0)$ i que

$$\frac{dN_1}{dt}(0) = -bN_2(0),$$

es comprova fàcilment que la solució és

$$N_1(t) = N_1(0)\cosh\sqrt{bc}t - \sqrt{\frac{b}{c}}N_2(0)\sinh\sqrt{bc}t.$$

De manera anàloga es té

$$N_2(t) = N_2(0)\cosh\sqrt{bc}t - \frac{N_1(0)}{\sqrt{b/c}}\sinh\sqrt{bc}t.$$

Combat entre guerrilles

De manera semblant al cas d'exèrcits convencionals tractat a l'exemple anterior, suposarem ara que tenim dues guerrilles que combaten aïllades, és a dir, no tenen reforços per part dels seus respectius bàndols. Suposarem a més que no tenen pèrdues operacionals. Per tant, les pèrdues de cada bàndol són també únicament degudes al combat.

Ara bé, la manera de combatre de les guerrilles és molt diferent de la dels exèrcits. Com que les gerrilles són reduïdes i actuen camufladades respecte del seu oponent, podem modelar

aquest exemple de manera semblant als encontres de recerca i fugida que tenim en el model depredador presa.

Així, si suposem que les guerrilles N_1 i N_2 es troben dins una regió R, donat que romanen amagades una de l'altra, les baixes en cada bàndol es produiran en el moment dels encontres. Encontres que són proporcionals tant al nombre de combatents de N_1 com al de N_2.

Tenint en compte això, el model de combat entre dues guerrilles ve donat per

$$\begin{cases} \frac{dN_1}{dt} &= -gN_1N_2, \\ \frac{dN_2}{dt} &= -hN_1N_2, \end{cases}$$

on les constants positives g i h continuen essent els coeficients d'efectivitat en el combat per a N_2 i N_1 respectivament. Ara bé, segons Lanchester, les estimacions varien respecte del que teníem pel al cas entre dos exèrcits convencionals. Concretament es proposa que

$$g = r_{N_2}\frac{A_{rN_2}}{A_{N_1}}, \quad \text{i} \quad h = r_{N_1}\frac{A_{rN_1}}{A_{N_2}},$$

on la r és el mateix que per al combat entre exèrcits convencionals, però la probabilitat, p, de matar a un oponent s'ha substituït per un quocient entre l'àrea d'efectivitat d'un tret, A_r, que de fet és l'àrea d'un guerriller combatent en cobert, i l'àrea A ocupada per la guerrilla dins la regió R.

Procedint com en el cas anterior, si dividim les equacions del nostre model s'obté

$$\frac{dy}{dx} = \frac{h}{g},$$

que és també una equació diferencial de variables separades, i integrada dóna

$$gN_2(t) - hN_1(t) = L,$$

on L és una constant arbitrària que es pot determinar, tal com passava en el cas de combat entre exèrcits convencionals, en el moment inicial de la batalla,

$$L = gN_2(0) - hN_1(0).$$

Això ens porta a la relació

$$g(N_2(t) - N_2(0)) = h(N_1(t) - N_1(0)),$$

coneguda com la *llei lineal de combat*.

De manera semblant al cas d'exèrcits convencionals, el valor de L, determinat al principi de la batalla, decideix el bàndol guanyador. Si $L = 0$, s'arriba a un empat. Si $L > 0$, guanya la guerrilla N_2 amb L/g supervivents; mentre que si $L < 0$, guanya la guerrilla N_1 amb $-L/h$ supervivents.

El dibuix de les trajectòries (rectes) de $gN_2(t) - hN_1(t) = L$, en el quadrant $N_1 > 0$, $N_2 > 0$ representa el retrat de fase de totes les possibles batalles i el donem a la figura 4.9.

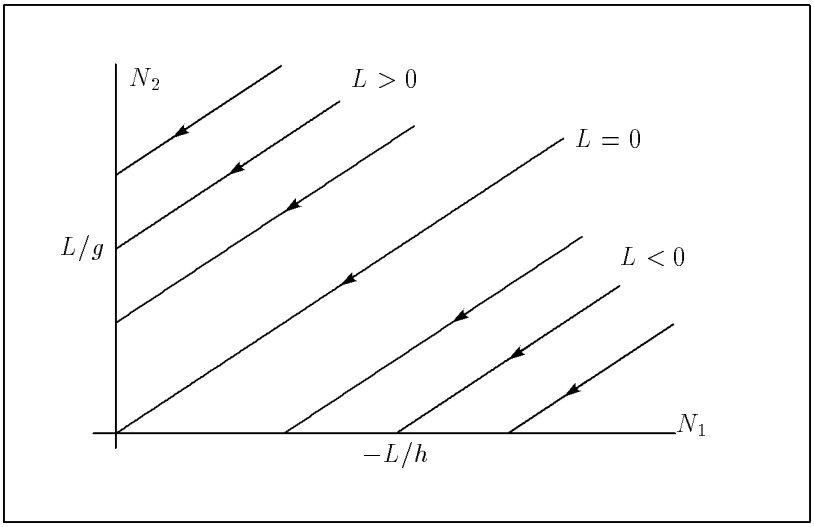

Figura 4.9: *Retrat de fase del model de combat entre guerrilles. Les òrbites són rectes.*

En aquest cas, si ens posem en el costat N_2, ens caldrà que $L > 0$ per guanyar la batalla, és a dir

$$\frac{N_2(0)}{N_1(0)} > \frac{h}{g},$$

i per tant en aquest cas la relació entre el nombre d'homes i la d'efectivitats és lineal. Per tant, si la guerrilla N_1 té inicialment el doble d'homes que N_2, el combat es pot compensar si g és el doble de h, és a dir si els combatents de N_2 tenen una preparació dues vegades superior a la dels de N_1. Recordem que en el cas de combat entre exèrcits convencionals, la preparació hauria de ser quatre vegades millor.

Combat entre guerrilla i exèrcit

Com en el casos anteriors suposarem ara una guerrilla N_1 i un exèrcit convencional N_2 que combaten aïllats i amb les úniques pèrdues causades pel bàndol contrari.

Si tenim en compte la manera d'actuar d'un i altre bàndol, comentada en els apartats anteriors, resulta que la guerrilla, com que no té dificultats per trobar l'exèrcit, produeix baixes proporcionals a la seva força d'atac. Mentre que l'exèrcit, com que no pot assegurar que dispara contra un guerriller que es troba camuflat, produeix baixes proporcionals tant al nombre de la tropa que té l'exèrcit, com al nombre de guerrillers que hi ha a la regió de combat.

Així obtenim el model

$$\begin{cases} \frac{dN_1}{dt} &= -gN_1N_2, \\ \frac{dN_2}{dt} &= -cN_1, \end{cases}$$

on les constants positives c i g són els coeficients d'efectivitat de la guerrilla N_1 i de l'exèrcit N_2 respectivament.

Dividint les dues equacions del model obtenim

$$\frac{dN_2}{dN_1} = \frac{c}{gN_2},$$

que torna a ser una equació de variables separades, la qual integrada dóna

$$gN_2^2(t) - 2cN_1(t) = M,$$

on M és una constant que es pot determinar a l'instant inicial de la batalla,

$$M = gN_2^2(0) - 2cN_1(0).$$

De la qual cosa obtenim

$$g(N_2^2(t) - N_2^2(0)) = 2c(N_1(t) - N_1(0)),$$

coneguda com la *llei parabòlica de combat.*

De manera semblant al casos anteriors, el valor de M, determinat al principi de la batalla, decideix el bàndol guanyador. Si $M = 0$ s'arriba a un empat. Si $M > 0$ guanya l'exèrcit N_2 amb $\sqrt{M/g}$ supervivents. Mentre que, si $M < 0$, guanya la guerrilla N_1 amb $-M/2c$ supervivents.

El dibuix de les trajectòries (paràboles) de $gN_2^2(t) - 2cN_1(t) = M$, en el quadrant $N_1 > 0$, $N_2 > 0$, representa el retrat de fases de totes les possibles batalles; el tenim representat a la figura 4.10.

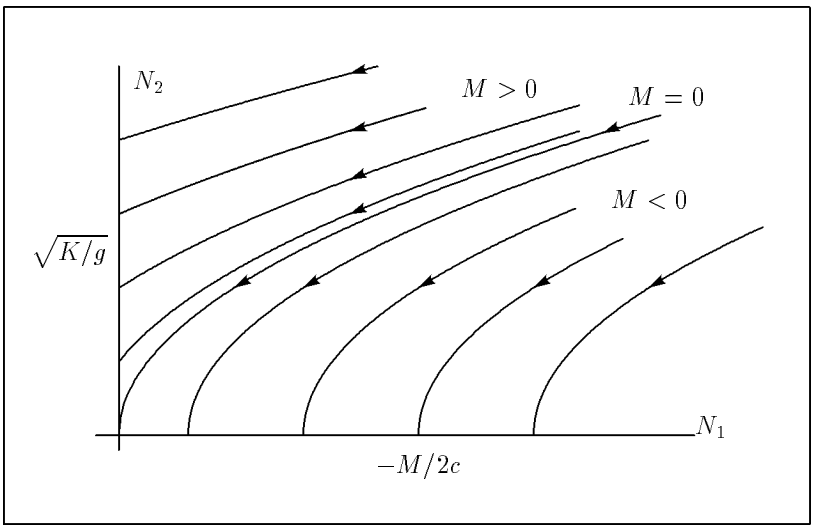

Figura 4.10: *Retrat de fase del model de combat entre guerrilla i exèrcit. Les òrbites són paràboles.*

Aplicarem la llei parabólica per veure quines proporcions asseguren una victòria de l'exèrcit N_2. Això, tal com hem vist, passa si $M > 0$, és a dir, si

$$\left(\frac{N_2(0)}{N_1(0)}\right)^2 > \frac{2c}{g}\frac{1}{N_1(0)} = 2\frac{r_{N_1}}{r_{N_2}}\frac{A_{N_1}p_{N_1}}{A_{rN_2}}\frac{1}{N_1(0)},$$

on hem usat les estimacions dels coeficients d'efectivitat

$$c = r_{N_1}p_{N_1} \quad \text{i} \quad g = r_{N_2}\frac{A_{rN_2}}{A_{N_1}}$$

per a cada bàndol.

Suposem que els "ritmes de foc" de cada bàndol són semblants $r_{N_1} \simeq r_{N_2}$, que la probabilitat que un tret d'un guerriller mati un oponent és $p_{N_1} = 0.1$ i que la part exposada del cos d'un guerriller en combat és de $A_{rN_2} = 0.25\text{m}^2$. Aleshores la relació anterior queda com

$$\left(\frac{N_2(0)}{N_1(0)}\right)^2 > \frac{0.8A_{N_1}}{N_1(0)}.$$

Si suposem que cada guerriller cobreix una àrea d'uns 125m^2, tindrem que l'àrea ocupada inicialment per la guerrilla és $A_{N_1} = 125N_1(0)$. Amb la qual cosa ens queda la relació

$$\frac{N_2(0)}{N_1(0)} > 10$$

i per tant l'exèrcit convencional té avantatge si el seu nombre de tropes és com a mínim deu vegades el nombre de guerrillers.

Per acabar aquesta secció direm que, repassant els conflictes que s'han anat desenvolupant al llarg de la història, la victòria normalment correspon a l'exèrcit convencional en els casos en què el nombre d'homes ha estat com a mínim vuit vegades el de la guerrilla (vegeu [4]).

Un dels casos més destacats el tenim en la Guerra del Vietnam. Durant la primavera de 1968, l'exèrcit convencional estava format aproximadament per uns 510.000 americans, 1.100.000 sud-vietnamites (meitat tropa regular i meitat defensa local) i per 70.000 altres aliats. Mentre que la guerrilla estava formada aproximadament per uns 50.000 nord-vietnamites i uns 230.000 soldats del vietcong. Si fem la relació entre els totals veurem que l'exèrcit convencional era aproximadament sis vegades la guerrilla, la qual cosa decantava el conflicte clarament del costat d'aquesta.

En aquest moment, ja cap al final del conflicte, el general Westmoreland demanava 206.000 homes més al president Johnson. Si aquest reclutament s'hagués dut a terme, segons el nostre model, hauria estat del tot infructuós per a l'exèrcit, ja que la relació N_2/N_1 passava a ser aproximadament de 6,7. A més, per tornar la relació a 6, la guerrilla només hauria hagut de reclutar 34.000 homes més.

4.6.3 El principi de competició exclusiva

El fet que ara estudiarem ja va ser observat per Darwin cap a l'any 1859. En aquell temps, Darwin es va adonar que la competència entre dues espècies semblants, de les que avui podríem dir que ocupen un mateix nínxol ecològic quant a hàbitat i costums, era molt més intensa que

entre dues espècies adversàries diferents. Fins al punt que en molts casos es produïa la total desaparició d'una d'elles.

Per explicar aquest fenòmen considerarem que cada una de les espècies que tenim segueix una llei de creixement logístic

$$\frac{dN}{dt} = aN(1 - \frac{N}{K}),$$

on recordem que a és el ritme de creixement i K el nivell de saturació.

Suposem que tenim dues espècies N_1 i N_2 amb ritmes de creixement r i s, i nivells respectius de saturació K i L El model de competició vist a la secció 4.5.2,

$$\begin{cases} \dot{N_1} &= rN_1(1 - \frac{N_1}{K}) - \alpha N_1 N_2, \\ \dot{N_2} &= sN_2(1 - \frac{N_2}{L}) - \beta N_1 N_2, \end{cases}$$

en el nostre cas el podem escriure com

$$\begin{cases} \dot{N_1} &= rN_1(\frac{K - N_1 - m_2}{K}), \\ \dot{N_2} &= sN_2(\frac{L - N_2 - m_1}{L}), \end{cases}$$

on m_1 és "la nosa" que fa la primera espècie a la segona, i de manera semblant, m_2 la nosa que fa la segona a la primera.

En el cas de model de competició habitual, $m_1 = \bar{\beta}N_1$ i $m_2 = \bar{\alpha}N_2$, per a unes determinades, $\bar{\alpha}$ i $\bar{\beta}$, que mesuren no tan sols el bon competidor que és el contrari, sinó també l'ús que cada espècie fa de l'entorn. Així per exemple podria passar que $\bar{\alpha}$ fos gran, no perquè N_2 sigui un bon competidor, sinó perque N_2 produeix unes certes substàncies tòxiques que perjudiquen N_1 o bé li devasta l'entorn d'una manera improductiva.

En el cas que estudiem estem suposant que les dues espècies són gairebé iguals i es comporten de la mateixa manera. Per tant $\bar{\alpha} = \bar{\beta} = 1$ i obtenim el model,

$$\begin{cases} \dot{N_1} &= rN_1(\frac{K - N_1 - N_2}{K}), \\ \dot{N_2} &= sN_2(\frac{L - N_2 - N_1}{L}). \end{cases}$$

El *principi de competició exclusiva* és dedueix d'aquest sistema d'equacions diferencials i diu que, si $K > L$, aleshores tota òrbita $N_1(t)$, $N_2(t)$ del model s'acosta al punt d'equilibri $(K, 0)$, quan el temps t tendeix a infinit. És a dir, si les espècies 1 i 2 són gairebé idèntiques, però l'hàbitat pot contenir més espècies del tipus 1 que del tipus 2, a la llarga l'espècie 2 s'extingeix.

Per justificar aquest fet farem un esbós del retrat de fase d'aquest model. Per això comencem buscant els punts d'equilibri, que com sabem els obtenim resolent

$$\begin{aligned} rN_1(\frac{K - N_1 - N_2}{K}) &= 0, \\ sN_2(\frac{L - N_2 - N_1}{L}) &= 0. \end{aligned}$$

De la primera equació obtenim $N_1 = 0$ o $K - N_1 - N_2 = 0$. Posant $N_1 = 0$ a la segona equació s'obtenen els punts

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad \text{i} \quad \begin{pmatrix} 0 \\ L \end{pmatrix}.$$

Mentre que, usant $K - N_1 - N_2 = 0$ en la segona equació, només s'obté com a nou punt el

$$\begin{pmatrix} K \\ 0 \end{pmatrix}.$$

Notem també que les *isoclines* en què el camp és vertical o horitzontal són les rectes

$$N_1 = 0, \quad \text{i} \quad K - N_1 - N_2 = 0,$$

on s'anul·la la component $\dot{N}_1$ i per tant el camp és vertical ja que només hi ha component en $\dot{N}_2$, i les rectes

$$N_2 = 0, \quad \text{i} \quad L - N_1 - N_2 = 0,$$

on s'anul·la la component $\dot{N}_2$ i per tant damunt d'elles el camp és horitzontal.

Una altra cosa que cal veure és que els eixos són invariants (la direcció del camp està continguda en el mateix eix), i que les òrbites damunt dels eixos van a parar asimptòticament al punt d'equilibri que hi ha. Això es veu directament mirant el signe del camp en els diferents punts dels eixos o tenint present que en cada eix es segueix la llei de creixement logístic per a cada espècie.

Finalment, les rectes paral·leles $K - N_1 - N_2 = 0$ i $L - N_1 - N_2 = 0$, de les quals suposarem que $K > L$, divideixen el primer quadrant en tres regions: la regió I per sota les dues rectes, la regió II enmig de les dues rectes i la regió III per damunt d'elles (vegeu la figura 4.11). Notem que en passar d'una regió a l'altra es creua una d'aquestes rectes i, per tant, el sentit del camp canvia en la direcció que s'anul·la al creuar la recta. D'aquesta manera dins la regió I es té $\dot{N}_1 > 0$ i $\dot{N}_2 > 0$. A la regió II, $\dot{N}_1 > 0$ i $\dot{N}_2 < 0$. Mentre que a la regió III, $\dot{N}_2 < 0$ i $\dot{N}_2 < 0$. Això fa que la regió II sigui una *regió invariant pel flux*, és a dir, tota òrbita que en un cert moment es troba a la regió II hi romandrà per sempre més.

Si agafem una òrbita que comenci a la regió I, es pot justificar rigorosament el fet intuïtiu que, donat que dins d'aquesta regió $\dot{N}_1 > 0$, l'òrbita anirà a parar a la regió II[4].

De la mateixa manera, si una òrbita comença dins de la regió III, com que el camp hi té components $\dot{N}_1 < 0$ i $\dot{N}_2 < 0$, és a dir, "avall i a l'esquerra", l'òrbita a la llarga entrarà dins la regió II o bé anirà a parar al punt d'equilibri $(K, 0)$.

Per acabar de veure que totes les òrbites arriben a la llarga asimptòticament al punt $(K, 0)$, notem que dins la regió II el camp té components $\dot{N}_1 > 0$ i $\dot{N}_2 < 0$, és a dir, "avall i a la dreta". Per tant, com que tota òrbita que és dins la regió II no en pot sortir, i sempre ha d'anar en la direcció indicada, ha d'acabar tendint al punt $(K, 0)$. Tots aquests fets indicats aquí de manera intuïtiva, es poden justificar rigorosament mitjançant eines de la *teoria qualitativa de les equacions diferencials ordinàries*.

[4]No pot anar a parar al punt $(0, L)$ ja que, com que $\dot{N}_1 > 0$ a la regió I i a la regió II, el punt $(0, L)$ és un punt de sella.

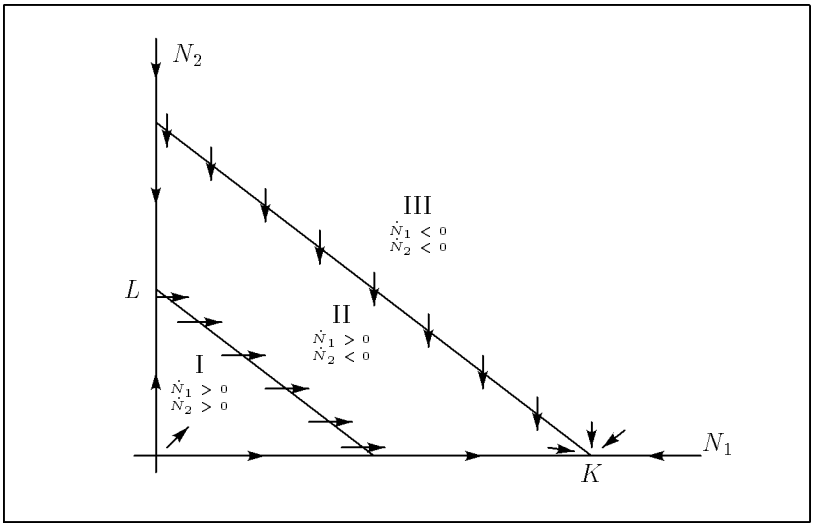

Figura 4.11: *Regions que obtenim en el retrat de fase associat a un model de competició exclusiva, d'acord amb els signes de les dues components del camp.*

4.7 Model per al filtratge de cigarretes

Com a darrer exemple presentem un petit estudi de com s'acumulen les substàncies nocives dins una cigarreta a mesura que va essent fumada. Hem deixat aquest model per al final ja que s'aparta una mica del que és el contingut general d'aquest llibre i, a més, requereix algunes eines matemàtiques pròpies d'un curs de càlcul infinitesimal d'una o diverses variables.

L'exemple, però, és interessant, ja que es veu el procés de modelització usant balanços per tal de deduir la llei que governa el fenomen.

Suposem que tenim una cigarreta amb filtre de longitud total L. Si representem per $x = 0$ l'extrem per on s'encén, tenim que la sortida del filtre té la posició $x = L$. Suposem a més que $x = l$ és el començament del filtre i, per tant, la part de tabac inicialment té longitud l, mentre que el filtre sempre té longitud $L - l$.

La manera de fumar és molt distinta de persona a persona. D'una banda, hi ha individus que només cremen el tabac i, d'altra banda, hi ha fumadors que no paren de pipar. Aquest fet és, doncs, molt difícil de modelitzar, i a fi de simplificar el model suposarem que el fumador xucla uniformement el fum a una velocitat constant, que representarem per V, i del tabac que fuma, només la proporció p, $0 < p < 1$, és la que entra dins la cigarreta. L'altra part, de proporció $1 - p$, s'allibera a l'atmosfera sense entrar dins la cigarreta. Notem que variant els valors de V i de p es pot aconseguir modelar molts tipus de fumadors o, si més no, el que podríem anomenar "l'estil mitjà" d'un cert fumador.

Continuant amb les hipòtesis simplificadores, com que el nostre fumador fuma a "velocitat constant", el foc també es desplaça a velocitat constant per la cigarreta. Anomenem v aquesta velocitat. Si suposem que la cigarreta s'encén quan $t = 0$, passat un temps t, el foc, en cas de

no haver arribat al filtre, es troba a $x = vt$. Notem que $v << V$, la qual cosa ens permetrà d'estudiar el problema dividint-lo en dues parts que veurem tot seguit.

Representarem per S qualsevol de les substàncies que té el tabac i de la qual volem estudiar la deposició.

Estudi fixant el foc i cremant tabac no contaminat

Com que $v << V$, en primer lloc estudiem el problema suposant que el foc no es mou. Suposem que la cigarreta estigui cremant en una certa posició (fixa) x_0. Notem per $\rho(x)$ a la densitat lineal de S dins el fum de la cigarreta. $\rho(x)$ ens mesura com tenim de "carregat" el fum de dins la cigarreta a la posició x.

Notem que si el foc no es mou, aquesta densitat és independent del temps, ja que el fum que es crea en el lloc de combustió tindrà sempre la mateixa càrrega de S i aquesta s'anirà dipositant de la mateixa manera al llarg de la cigarreta. Això si suposem que la capacitat d'absorció de S per part del tabac no varia amb el pas del temps, la qual cosa és plausible ja que per als fumadors, les cigarretes tampoc no duren gaire.

Aleshores, com que el fum és l'únic factor que transporta a S, la funció

$$r(x) = V\rho(x)$$

representa la massa de S que passa pel lloc x per unitat de temps i, per tant, és un flux.

Agafem ara un tros de cigarreta de longitud Δx, que s'estengui de x a $x + \Delta x$, i a la qual li arribi el fum que es genera a la posició x_0. Per la llei de conservació de la massa, la quantitat de S que surt per l'extrem $x + \Delta x$ és la quantitat que entra per l'extrem x menys la quantitat de S que es diposita de x a $x + \Delta x$.

Si anomenem a el coeficient d'absorció que té el tabac per a la substància S, coeficient que, com hem dit, el suposarem constant al llarg del temps, la quantitat de S absorbida a l'interval $(x, \Delta x)$ per unitat de temps és,

$$a \int_x^{x+\Delta x} \rho(s)ds. \tag{4.5}$$

Anàlogament, si l'interval $(x, x + \Delta x)$ es troba dins el filtre, la quantitat de S absorbida per unitat de temps té la mateixa forma i només cal canviar el coeficient a d'absorció del tabac pel coeficient d'absorció del filtre que anomenarem α.

Finalment, si apliquen la llei de conservació de la massa a l'interval $(x, \Delta x)$ per unitat de temps tenim

$$r(x + \Delta x) = r(x) - a \int_x^{x+\Delta x} \rho(s)ds$$

expressió que, si la dividim per $V\Delta x$, resulta

$$-\frac{\rho(x + \Delta x) - \rho(x)}{\Delta x} = \frac{a}{\Delta x} \int_x^{x+\Delta x} \rho(s)ds$$

que en fer el límit quan $\Delta x \to 0$ obtenim una equació diferencial ja coneguda per nosaltres,

$$\frac{d\rho}{dx}(x) = -\frac{a}{V}\rho(x), \qquad x_0 \leq x \leq l,$$

i per la qual ja d'entrada podem dir que la densitat de substància S continguda en el fum decau de manera exponencial a causa de l'absorció.

És clar que per al fum que passa pel filtre tenim una llei semblant, en què només canvia el coeficient d'absorció:

$$\frac{d\rho}{dx}(x) = -\frac{\alpha}{V}\rho(x), \qquad l \leq x \leq L.$$

Separant variables per resoldre cada una de les equacions en el seu tros obtenim

$$\rho(x) = \begin{cases} Ae^{-\frac{a}{V}x}, & x_0 \leq x \leq l, \\ Be^{-\frac{\alpha}{V}x}, & l \leq x \leq L, \end{cases}$$

on les constants A i B les determinem a fi que es compleixin les condicions de contorn.

Suposem que en el lloc $x = x_0$ on crema el tabac s'alliberen G unitats de massa de S per unitat de temps. Com que p és la proporció que entra dins la cigarreta, tenim que el flux en el punt $x = x_0$ val pG, i ha de coincidir amb $r(x_0) = V\rho(x_0)$, és a dir, $pG = AVe^{-(a/V)x_0}$, d'on tenim

$$A = \frac{pG}{V}e^{\frac{a}{V}x_0}.$$

D'altra banda, busquem B imposant la continuïtat de $\rho(x)$ en el punt $x = l$, és a dir,

$$Ae^{-\frac{a}{V}l} = Be^{-\frac{\alpha}{V}l}$$

d'on, aïllant B i usant el valor de A, obtenim

$$B = \frac{pG}{V}e^{\frac{a}{V}(x_0-l)+\frac{l}{V}\alpha}$$

i, per tant, la funció $\rho(x)$ és

$$\rho(x) = \begin{cases} \frac{pG}{V}e^{-\frac{a}{V}(x-x_0)}, & x_0 \leq x \leq l, \\ \frac{pG}{V}e^{-\frac{a}{V}(l-x_0)-\frac{\alpha}{V}(x-l)}, & l \leq x \leq L, \end{cases} \tag{4.6}$$

on notem que $x - x_0$ és la distància del punt x al punt encès de la cigarreta. Aquesta distància entra directament a l'exponencial pel tros on només hi ha tabac, mentre que es parteix en dues parts $x - x_0 = (l - x_0) + (x - l)$ si la posició x és dins el filtre.

Vegem el flux de substància S a l'extrem final del filtre. Sabem que el flux en aquest punt és $r(L) = V\rho(L)$ i, per tant, val

$$r(L) = pGe^{-\frac{a}{V}(l-x_0)-\frac{\alpha}{V}(L-l)}.$$

Com ja hem vist en la solució de l'equació, això es pot interpretar de la manera següent. El flux inicial de S que entra dins la cigarreta en el lloc de combustió val pG. Aquest flux queda filtrat pel tabac en recórrer l'espai $l - x_0$, i a l'entrada del filtre el flux val

$$pGe^{-\frac{a}{V}(l-x_0)}.$$

Aquest darrer flux es depura de nou dins el filtre de la cigarreta, la qual cosa veiem que és expressada matemàticament en multiplicar-ho pel factor de filtratge del filtre

$$e^{-\frac{\alpha}{V}(L-l)}.$$

Tenim, doncs, que el flux inicial, pG, al punt x_0 queda filtrat pel tabac i pel filtre de la cigarreta segons els factors respectius

$$e^{-\frac{a}{V}(l-x_0)} \quad \text{i} \quad e^{-\frac{\alpha}{V}(L-l)}.$$

Si volem que el filtre de la cigarreta sigui eficaç caldrà que $\alpha >> a$. En cas contrari és millor fumar la cigarreta sense filtre. Suposant sempre, és clar, que la cigarreta no s'apura massa i al final la burilla té la longitud $L-l$.

Estudi per a $v > 0$ i tabac contaminat

Vegem ara el que passa quan el foc de la cigarreta va avançant. La idea és essencialment la mateixa que en el cas anterior però ara cal tenir en compte el fet següent.

Atès que el tabac també actua de filtre, resulta que aquest es va contaminant de la substància S. Quan arriba el foc, la quantitat de S que s'allibera no és la mateixa que la que s'ha alliberat en llocs anteriors de la cigarreta. A mesura que avança el foc el fum es va fent més dens i, per tant, fixada una posició x, a la qual encara no hagi arribat el foc, la densitat de S dins el fum en aquesta posició és funció del temps. És a dir que ara ρ depèn de x i de t, $\rho(x,t)$.

Com que ara també ens cal saber la quantitat de substància nociva que té el tabac, notarem per $T(x,t)$ la densitat de S en el tabac en el lloc x i a l'instant t. És a dir que en un cert temps fixat, t, la quantitat de S en un interval no cremat, (i_1, i_2), és

$$\int_{i_1}^{i_2} T(x,t)dx.$$

Com que per a $t = 0$ el foc és a $x = 0$ i aquest avança a velocitat constant v, el foc en un instant t es troba a $x = vt$.

Agafem un interval de temps $(t, t+\Delta t)$. En aquest temps s'haurà cremat l'interval de cigarreta $(vt, v(t+\Delta t))$. Com que en cremar tota la quantitat de S de l'interval s'allibera, el que s'allibera és

$$\int_{vt}^{v(t+\Delta t)} T(x, \frac{x}{v})dx.$$

I, per tant, el flux de S degut al fet de cremar, que és la quantitat alliberada per unitat de temps i que a l'apartat anterior és la constant G, ara depèn del temps i val

$$G(t) = \lim_{\Delta t \to 0} \frac{1}{\Delta t} \int_{vt}^{v(t+\Delta t)} T(x, \frac{x}{v})dx = \lim_{\Delta t \to 0} vT(t+\Delta t, \frac{v(t+\Delta t)}{v}) = vT(vt,t). \qquad (4.7)$$

A fi de trobar l'expressió per a la funció $T(x,t)$, notem que la quantitat de S que es va dipositant damunt del tabac és precisament la que va perdent el fum quan és filtrat pel tabac. Si a l'instant t considerem un interval de tabac no cremat $(x, x+\Delta x)$, tenint en compte que el ritme de deposició de S damunt del tabac és donat per (4.5), que el foc es troba a $x_0 = vt$

i usant a més (4.6), amb el supòsit que $v << V$ i que ara les quantitats també depenen del temps, resulta que la quantitat de S dipositada damunt del tabac per unitat de temps val

$$a\int_x^{x+\Delta x}\rho(s,t)ds = \frac{apG(t)}{V}\int_x^{x+\Delta x} e^{-\frac{a}{V}(s-vt)}ds.$$

Valor que, d'altra banda, ha de ser, segons la definició que hem donat de $T(x,t)$,

$$\frac{d}{dt}\int_x^{x+\Delta x} T(s,t)ds.$$

D'aquesta manera, igualant les dues expressions i substituint la $G(t)$ pel seu valor trobat a (4.7), tenim

$$\frac{d}{dt}\int_x^{x+\Delta x} T(s,t)ds = \frac{apv}{V}T(vt,t)e^{\frac{av}{V}t}\int_x^{x+\Delta x} e^{-\frac{a}{V}s}ds.$$

Ara, integrant respecte de t des de $t=0$ fins a un cert temps t, dividint per Δx els dos costats i fent tendir $\Delta x \to 0$, resulta

$$T(x,t) = T(x,0) + \frac{apv}{V}e^{-\frac{a}{V}x}\int_0^t T(v\tau,\tau)e^{\frac{av}{V}\tau}d\tau. \tag{4.8}$$

Notem que hem obtingut una relació per a la funció T en la qual apareixen integrals d'aquesta funció. És el que s'anomena una *equació integral*.

El valor $T(x,0)$ és la distribució inicial de S en el tabac. Com que podem suposar la cigarreta homogènia, és un valor constant que representarem per T_0.

L'equació integral que tenim la podem resoldre pels valors $x = vt$, és a dir, podem trobar la quantitat de S en el tabac en el moment de ser cremat, $T(vt,t)$. Per això agafem (4.8) posant $x = vt$ i multiplicant-la per $e^{avt/V}$:

$$T(vt,t)e^{\frac{av}{V}t} = T_0e^{\frac{av}{V}t} + \frac{apv}{V}\int_0^t T(v\tau,\tau)e^{\frac{av}{V}\tau}d\tau,$$

la qual cosa ens dóna una equació integral que només depèn de t. Ara, notant per $f(t) = T(vt,t)e^{\frac{av}{V}t}$, resulta

$$f(t) = T_0e^{\frac{av}{V}t} + \frac{apv}{V}\int_0^t f(\tau)d\tau,$$

i derivant respecte de t

$$f'(t) = \frac{apv}{V}f(t) + \frac{av}{V}T_0e^{\frac{av}{V}t}, \tag{4.9}$$

és a dir, una *equació diferencial lineal de primer ordre* per a la qual usant l'equació integral anterior veiem que ha de complir $f(0) = T_0$.

El mètode de resolució d'aquest tipus d'equacions, com també el de variables separades, es pot trobar a qualsevol llibre bàsic d'equacions diferencials ordinàries com, per exemple, [24] o bé [3]. Aquí només notem que la solució de la part homogènia és

$$f(t) = Ke^{\frac{apv}{V}t},$$

on K és una constant arbitrària, i que usant el mètode de variació de constants usual per aquests tipus d'equacions, la solució de (4.9) amb la condició $f(0) = T_0$ és

$$f(t) = \frac{T_0}{1-p}(e^{\frac{av}{V}t} - pe^{\frac{apv}{V}t}).$$

D'aquesta manera obtenim

$$T(vt,t) = \frac{T_0}{1-p}(1 - pe^{\frac{av(p-1)}{V}t}). \tag{4.10}$$

Inserint aquest resultat dins la integral del costat dret de (4.8) i fent les operacions, podem trobar l'expressió de $T(x,t)$ per a qualsevol valor de x i t:

$$T(x,t) = T_0\left(1 + \frac{p}{1-p}e^{-\frac{av}{V}x}(e^{\frac{av}{V}t} - e^{\frac{avp}{V}t})\right).$$

En el cas que ens ocupa, el flux de S a la sortida del filtre també és funció del temps i val $r(L,t) = V\rho(L,t)$. Tornant a usar l'expressió de ρ de (4.6) però depenent del temps i l'expressió de $G(t)$ obtinguda a (4.7) resulta

$$r(L,t) = pG(t)e^{-\frac{a}{V}(l-vt)-\frac{\alpha}{V}(L-l)} = pvT(vt,t)e^{-\frac{a}{V}(l-vt)-\frac{\alpha}{V}(L-l)}.$$

Com que $T(vt,t)$ el tenim calculat a (4.10), ens queda que el flux de S a l'extrem del filtre val

$$r(L,t) = \frac{pvT_0}{1-p}(1 - pe^{\frac{av(p-1)}{V}t})e^{-\frac{a}{V}(l-vt)-\frac{\alpha}{V}(L-l)},$$

i la massa total de S que surt pel filtre s'obté integrant el flux durant el temps que dura la cigarreta, que en cas de ser "apurada" és l/v. Representarem aquest valor per M_l:

$$M_l = \int_0^{\frac{l}{v}} r(L,t)dt.$$

Fent aquesta integral, després d'uns pocs càlculs resulta

$$M_l = \frac{pVT_0}{a(1-p)}e^{-\frac{\alpha}{V}(L-l)}(1 - e^{-\frac{al}{V}(1-p)}). \tag{4.11}$$

d'on es veu clarament que la part de filtratge deguda al filtre de la cigarreta surt en el factor $e^{-(\alpha/V)(L-l)}$ i el que resta és la massa de S que arriba just al davant de l'entrada del filtre.

De tot això podem treure algunes conclusions de com es comporta el filtre de la cigarreta.

En primer lloc, suposem que usem una cigarreta de la mateixa longitud L però sense filtre. Suposem també que fumem la cigarreta fins a $x = l$. És a dir que podem considerar que la burilla del tabac que resta fa de filtre. Aleshores, per calcular la quantitat de S que arriba al final de la cigarreta, i que representarem per M_l^s, podem aplicar el mateix resultat que a (4.11) però canviant la α per a. Llavors, el quocient M_l/M_l^s és

$$\frac{M_l}{M_l^s} = e^{-\frac{L-l}{V}(\alpha-a)}$$

i, per tant, la relació de filtratge és exponencial decreixent tant per la mida del filtre, $L-l$, com per la diferència d'absorció respecte del tabac, $\alpha - a$, sempre que s'hagi usat un filtre prou bo i $\alpha > a$; en cas contrari és millor no usar filtre.

Finalment, vegem la relació que hi ha entre fumar la cigarreta totalment o deixar-la al cap d'una fracció. Per això representarem per $M_{l/n}$ la quantitat de S que ha sortit al final del filtre quan s'ha fumat un tros l/n de cigarreta. És clar que aquesta quantitat, de manera semblant a M_l, es calcula integrant $r(L,t)$ des del temps 0 fins a $l/(nv)$. Després d'uns pocs càlculs s'obté

$$M_{\frac{l}{n}} = \frac{pVT_0}{a(1-p)} e^{-\frac{\alpha}{V}(L-l)} e^{-\frac{a}{V}\frac{(n-1)l}{n}} \left(1 - e^{-\frac{a}{V}\frac{l}{n}(1-p)}\right),$$

on igual que a (4.11) veiem el factor de filtratge del filtre, però a més un factor de filtratge degut al tabac que no s'ha fumat representat per $e^{-\frac{a}{V}\frac{(n-1)l}{n}}$.

La relació entre $M_{l/n}$ i M_l és, doncs,

$$\frac{M_{\frac{l}{n}}}{M_l} = e^{-\frac{a}{V}\frac{(n-1)l}{n}} \left(\frac{1 - e^{-\frac{a}{V}\frac{l}{n}(1-p)}}{1 - e^{-\frac{a}{V}l(1-p)}} \right).$$

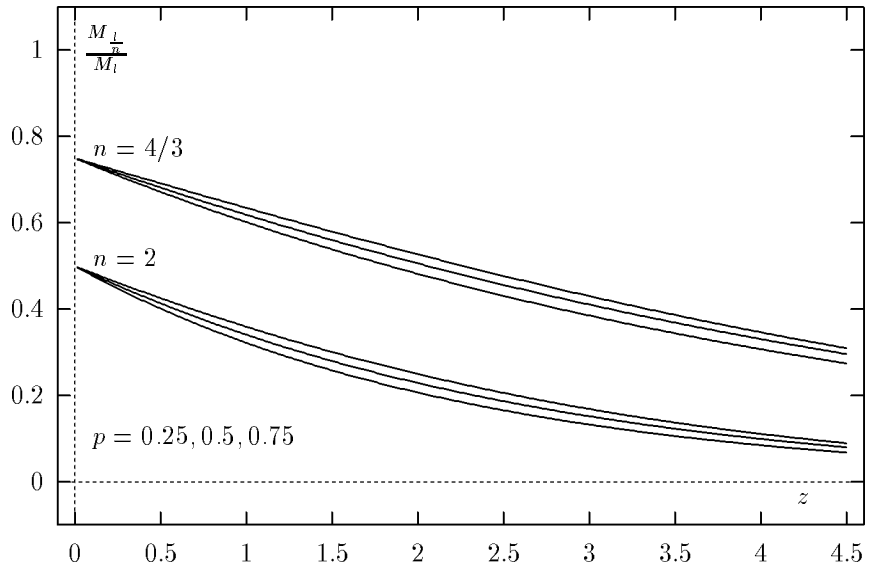

Figura 4.12: *Proporció de la quantitat de substància S que arriba a l'extrem final del filtre havent fumat un tros de cigarreta $\frac{l}{n}$ respecte d'haver-la fumat tota de la mateixa manera. El valor de la proporció es representa respecte del valor $z = \frac{al}{V}$. Les gràfiques són agrupades en tres corbes que corresponen als diferents valors de p presos. La corba inferior sempre correspon al valor de p més baix, i a l'inrevés.*

Si representem per $z = \frac{al}{V}$ podem representar la relació anterior en funció d'aquest valor z i

per a diferents valors de p i de n. Això és, la representació de

$$\frac{M_{\frac{l}{n}}}{M_l} = e^{-\frac{(n-1)z}{n}} \left(\frac{1 - e^{-\frac{z}{n}(1-p)}}{1 - e^{-z(1-p)}} \right).$$

A la figura 4.12 mostrem els resultats per a valors de $n = 2$, la qual cosa significa fumar la meitat de la cigarreta, i per al valor de $n = \frac{4}{3}$, que significa fumar les tres quartes parts de la cigarreta. Els càlculs s'han fet per als valors de $p = 0.25$, 0.50 i 0.75.

Tal com veiem a la gràfica, els resultats són poc sensibles als valors de p, però això és degut al fet que la gràfica correspon al càlcul de la proporció entre la part fumada respecte de la que s'obté fumant tota la cigarreta però amb la mateixa p. És clar que si p és petit els seus avantatges es veuen amb una simple inspecció de (4.11) que mostra que la relació entre p i M_l és de proporcionalitat directa.

Finalment, notem també que si z és petit, per exemple perquè V és molt gran, la qual cosa equival a xuclar el fum molt fort, o bé si a és petit, que vol dir que el tabac no absorbeix la substància S, llavors el resultat tendeix a ser com haver fumat per complet la part de la cigarreta considerada. Mentre que si el valor de z és gran, la part de cigarreta fumada és en realitat, a efectes de salut, molt més petita que la longitud cremada.

4.8 Estudi numèric d'alguns models

Vegem com podem usar les eines numèriques per estudiar alguns models donats per equacions diferencials ordinàries.

Els models que presentem no obeeixen en principi a cap cas real, però si que són il·lustratius de les possibilitats que ofereixen els mètodes numèrics per al seu estudi.

4.8.1 Un model depredador-presa

En aquest apartat estudiarem el model que comentem tot seguit,

$$\begin{cases} \dot{N_1} &= rN_1(1 - \frac{N_1}{K}) - \beta\frac{N_1N_2}{\alpha+N_1}, \\ \dot{N_2} &= sN_2(1 - \frac{N_2}{\nu N_1}), \end{cases}$$

on per fer l'estudi numèric prendrem $r = 2$, $K = 11$, $\alpha = 5$, $\beta = 2$, $s = 0.1$ i $\nu = 4$.

Podem pensar que N_1 i N_2 representen unes certes densitats de població d'unes espècies. De fet el model obeeix a un esquema depredador-presa on N_1 és la presa i N_2 el depredador, però amb unes certes variacions importants. Observem els fets següents:

- Sense depredador ($N_2 = 0$), la presa creix logísticament, $rN_1(1 - \frac{N_1}{K})$, al ritme r i amb nivell de saturació K.

- El depredador creix logísticament, $sN_2(1 - \frac{N_2}{\nu N_1})$, però el seu nivell de saturació és νN_1 i, per tant, depèn del nombre de preses.

- El depredador afecta la presa segons el terme $\beta\frac{N_1N_2}{\alpha+N_1}$. Quan N_1 és petit, aquest terme ve a ser proporcional a N_1N_2; mentre que, si N_1 és gran, val aproximadament βN_2. És a dir, quan hi han poques preses, el depredador l'ha de trobar segons el "procediment de recerca i fugida" semblant al cas de les guerrilles. Quan hi ha moltes

preses el depredador actua proporcionalment al seu nombre, d'una manera semblant al cas d'exèrcits convencionals.

Abans de parlar de com es poden realitzar estudis numèrics d'aquest model, farem fins on es pugui, una petita anàlisi del nostre model. Això sempre és important ja que ens pot servir de guia per a les coses que volguem explorar.

Si calculem els punts d'equilibri, és a dir, resolem el sistema

$$\begin{aligned} rN_1(1-\tfrac{N_1}{K}) - \beta\tfrac{N_1N_2}{\alpha+N_1} &= 0, \\ sN_2(1-\tfrac{N_2}{\nu N_1}) &= 0, \end{aligned}$$

obtenim les solucions

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \begin{pmatrix} K \\ 0 \end{pmatrix}, \quad \begin{pmatrix} \hat{N}_1 \\ \hat{N}_2 \end{pmatrix},$$

on $\hat{N}_1$ i $\hat{N}_2$ és el punt d'intersecció de la paràbola

$$N_2 = \frac{r}{K\beta}(K-N_1)(\alpha+N_1),$$

que s'obté en igualar a zero la primera equació. I de la recta

$$N_2 = \nu N_1,$$

que s'obté en igualar a zero la segona equació.

Quant a les isoclines tenim els fets següents:

- L'eix $N_2 = 0$ és invariant, ja que el camp només té sentit horitzontal.
- L'eix $N_1 = 0$ no té sentit físic, ja que N_1 divideix en la segona equació del model. De fet, però, es podria interpretar com que, si $N_1 = 0$, llavors $\dot{N}_2 = -\infty$, i per tant "N_2 desapareix a l'instant". El que ens diu que, si una òrbita s'acosta a l'eix $N_1 = 0$, el que fa és baixar ràpidament.
- Damunt de la paràbola anterior, $N_2 = \frac{r}{K\beta}(K-N_1)(\alpha+N_1)$, tenim que $\dot{N}_1 = 0$ i per tant les òrbites la tallen verticalment. Mirant el signe de $\dot{N}_2$ damunt de la paràbola, tenim que és negatiu (i per tant les òrbites van cap avall) si $N_1 < \hat{N}_1$, i positiu (i per tant les òrbites van cap amunt) si $N_1 > \hat{N}_1$.
- Damunt de la recta anterior, $N_2 = \nu N_1$, tenim que $\dot{N}_2 = 0$ i per tant les òrbites la tallen horitzontalment. Mirant el signe de $\dot{N}_1$ damunt d'aquesta recta, tenim que és positiu (i per tant les òrbites van cap a la dreta) si $N_1 < \hat{N}_1$, i negatiu (i per tant les òrbites van cap a l'esquerra) si $N_1 > \hat{N}_1$.

Partint d'aquests fets, a la figura (4.13) donem un esbós de les isoclines amb les direccions del camp.

A partir de la figura (4.13) ja veiem que les òrbites semblem girar al voltant del punt d'equilibri $(\hat{N}_1, \hat{N}_2)$. El que no queda clar és el que fan més precisament. Farem per a això un petit programa que, donat un estat inicial $(N_1(0), N_2(0))$, ens dibuixi la trajectòria durant un cert temps.

En primer lloc fem la funció que avalua el camp, tal com especifiquem a la secció dedicada a l'ús de la rutina `rk45f` amb l'exemple `cercle`.

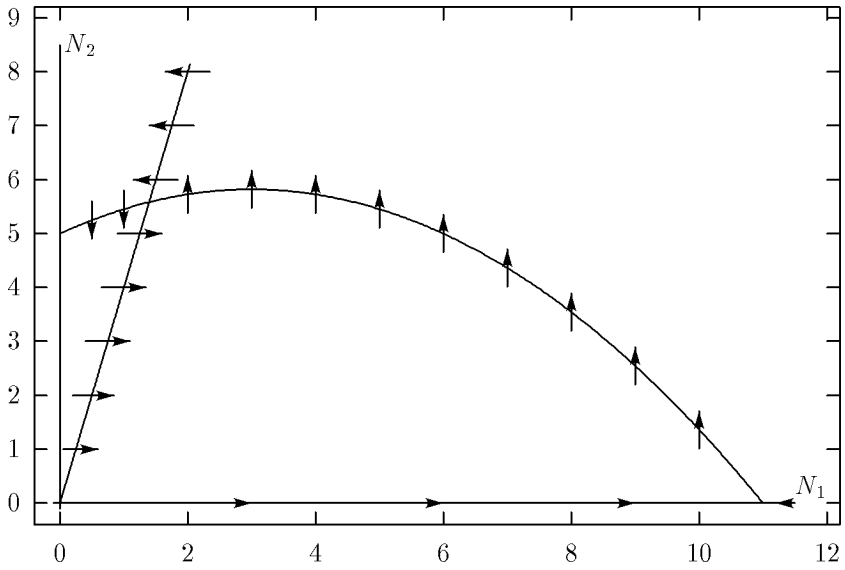

Figura 4.13: *Isoclines del model depredador-presa que estudiem.*

```
void prepre(double t, double x[], int n, double y[])
 {
  y[0]=2.e0*x[0]*(1.e0-x[0]/11.e0)-2.e0*x[0]*x[1]/(5.e0+x[0]);
  y[1]=0.1e0*x[1]*(1.e0-x[1]/(4.e0*x[0]));
 }
```

Fem servir el vector `x` per a contenir N_1 i N_2. Concretament N_1=`x[0]` i N_2=`x[1]`.

Programa principal de dibuix de trajectòria:

```
#include<stdio.h>
#include"grafbas.h"

main()
 {
  int n,col;
  double t,tmax,x[2],h,hmin,hmax,tol;
  void rk45f(double *t,double x[],int n,double *h,double hmin,double hmax,
             double tol,void(*camp)(double,double*,int,double*));
  void prepre(double t,double x[],int n,double y[]);
  puts("dona els valors inicials de N1 i N2");
  scanf("%le %le",&x[0],&x[1]);
  n=2;
  tmax=200.e0;
  hmin=1.e-4;
  hmax=1.e0;
```

```
  tol=1.e-10;
  t=0.e0;
  h=1.e-2;
  inigraf();
  finestra(0.e0,10.e0,0.e0,10.e0);
  col=prencolor();
  pospun(x[0],x[1],col);
  while(t<tmax)
   {
    rk45f(&t,x,n,&h,hmin,hmax,tol,prepre);
    lina(x[0],x[1],col);
    /* printf("t, N1 i N2: %le %le %le \n",t,x[0],x[1]); */
   }
  getch();
  tancagraf();
  printf("Valors finals de t, N1 i N2: %le %le %le\n",t,x[0],x[1]);
}
```

Les funcions `inigraf`, `finestra`, `prencolor`, `pospun`, `lina` i `tancagraf` fan la feina gràfica i es troben declarades a l'include `grafbas.h`. El seu ús està totalment detallat a l'apèndix B.

Notem que aquest programa dibuixa la trajectòria partint de la condició inicial que se li dona pel teclat fins a arribar al temps `tmax`.

És interessant observar que partint de qualsevol punt s'arriba sempre al mateix lloc. L'òrbita s'apropa a un *cicle límit* que no és altra cosa que una òrbita periòdica. Per exemple, l'òrbita que s'obté partint del punt (2,2) es troba representada a la figura 4.14.

Calculem ara aquesta òrbita periòdica. En primer lloc notem que calcular una òrbita periòdica vol dir conèixer un punt que estigui damunt d'ella i el seu període; ja que, integrant el model a partir d'aquest punt i durant un temps igual al període, podem obtenir qualsevol punt de l'òrbita.

Per calcular-la considerarem el fet que l'òrbita que busquem és atractora. Al mateix temps en calcularem el període. Per a això només ens cal aprofitar el fet que, partint de qualsevol punt i integrant prou temps, ens hi acostem tant com volem. Si l'òrbita en lloc d'atractora fos repulsora, podríem calcular-la igualment ja que només ens caldria integrar temps enrere. En cas que l'òrbita periòdica no sigui ni atractora ni repulsora, es pot calcular amb algorismes numèrics adequats, sempre que es conegui una aproximació inicial prou propera. Però aquest darrer tema queda fora de l'abast d'aquest curs introductori.

Tornant al càlcul de la nostra òrbita periòdica, veiem que, si volem tan sols una aproximació del seu període, el que podem fer és agafar un punt inicial qualsevol –per exemple el (2,2) d'abans– i integrar el model durant un temps prou gran fins gairebé trobar-nos damunt de l'òrbita periòdica, tal com fa el programa presentat anteriorment. Després, partint del darrer punt donat per la integració, que suposem que és el punt (n_1, n_2) el qual el podem considerar de l'òrbita periòdica, i posant el temps de nou a zero, tornem a integrar el nostre model començant per $N_1(0) = n_1$ i $N_2(0) = n_2$. Fem així una llista de t, $N_1(t)$ i $N_2(t)$. Quan veiem que per a un cert valor de $t = T$ resulta que $N_1(T) \simeq n_1$ i al mateix temps $N_2(T) \simeq n_2$, tenim que T representa aproximadament el període de l'òrbita.

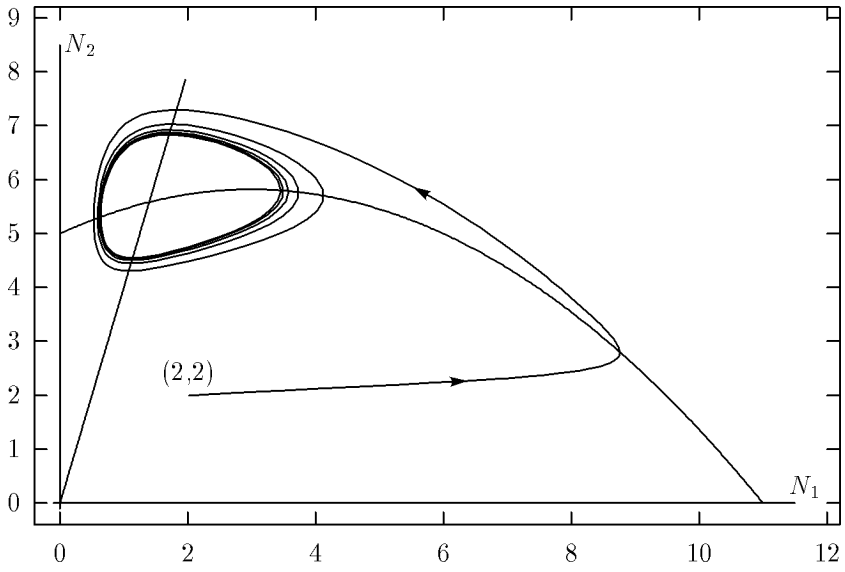

Figura 4.14: *Evolució de l'òrbita que surt del punt* $(2,2)$ *en el model depredador presa que estudiem.*

Però vegem com podem calcular l'òrbita periòdica i el seu període amb la precisió que volguem, sempre dins els límits de precisió numèrica de les variables `double`. Per aquest fet, el primer que farem és fixar una *superficie de secció* que en aquest cas no serà altra cosa que una recta vertical donada per l'equació $N_1 = A$. De tal manera que aconseguirem parar la integració just damunt d'aquesta recta en el moment que l'òrbita la talli en passar de $N_1 > A$ a $N_1 < A$. És a dir, partint d'un punt al temps t_i, $(N_1(t_i), N_2(t_i))$, que es trobi damunt d'aquesta superfície de secció, i per tant amb $N_1(t_i) = A$, podem aconseguir trobar un nou punt al temps t_f, $(N_1(t_f), N_2(t_f))$ tal que també compleixi que $N_1(t_f) = A$.

L'aplicació que passa, mitjançant el flux del camp, d'un punt damunt de la superfície de secció a un altre punt damunt de la secció, la tenim representada a la figura 4.15 i es coneix com *l'aplicació de Poincaré*.

L'aplicació de Poincaré, la implementem mitjançant la funció `poinca`, la qual, donat un valor inicial de t, `ti`, i uns valors de N_1 i N_2 en aquest temps (i que guardem a `xi[0]` i a `xi[1]` respectivament) ens torna el valor de temps t_f, `tf`, i els valors de N_1 i N_2, en aquest temps, tf (també guardats a `xf[0]` i a `xf[1]` respectivament), tal que $N_1(t_f)$=`xf[0]`$= A$.

Per aconseguir trobar el punt de tall de l'òrbita que seguim amb la recta $N_1 = A$, aplicarem el *mètode de Newton* per calcular zeros de funcions. Aquest mètode es pot trobar comentat en qualsevol llibre bàsic de càlcul numèric (vegeu per exemple [1]). Aquí només en donarem la fòrmula a fi i efecte de poder fer una aplicació immediata.

Suposem que volem trobar la solució, t_s, d'una certa equació $F(t) = 0$, on $F(t)$ és una funció derivable. Si t_0 és un valor proper a la solució, el mètode de Newton construeix una successió $t_0, t_1, t_2, \ldots$ donada per

$$t_{n+1} = t_n - \frac{F(t_n)}{F'(t_n)},$$

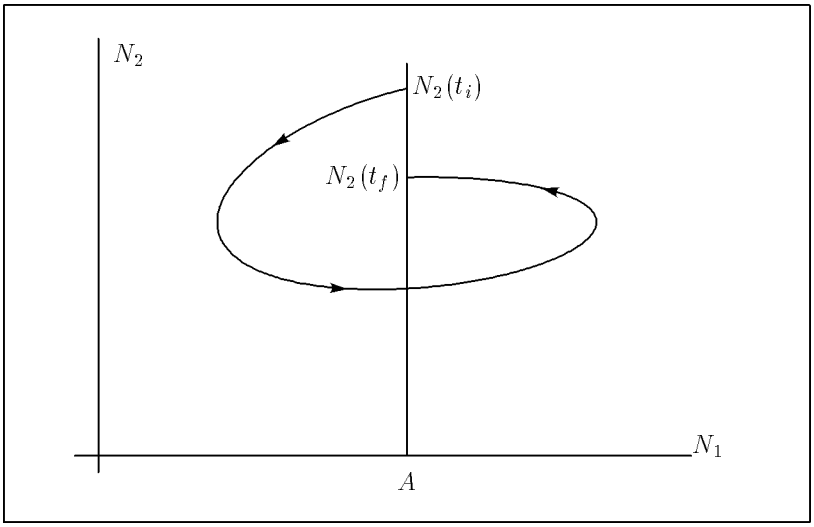

Figura 4.15: *Aplicació de Poincaré associada a la superfície de secció* $N_1 = A$.

de manera que sota hipòtesis genèriques convergeix ràpidament al valor t_s.

En el nostre cas ens interessa trobar el t_f tal que $N_1(t_f) = A$, per tant volem trobar un zero de l'equació $F(t) = N_1(t) - A = 0$. Aplicant el mètode de Newton tenim que la successió, $(t_n)_{n\geq 0}$, que convergeix cap a t_f, ve donada per

$$t_{n+1} - t_n = -\frac{N_1(t_n) - A}{\dot{N}_1(t_n)},$$

començant per un t_0 que "estigui prou a prop".

Però per trobar aquest valor inicial, t_0, només ens cal integrar fins a detectar el tall amb la superfície de secció (cosa que sabrem en el moment que N_1 passi de ser $N_1 > A$ a $N_1 < A$). En aquell instant tenim el t_0 i a més el valor $N_1(t_0)$ guardat a `xf[0]` (tal com veurem en el codi de la funció). El valor que ens falta, $\dot{N}_1(t_0)$, l'obtenim senzillament avaluant el camp en aquell instant.

A partir d'aquí podem calcular t_1 per mitjà de l'algorisme de Newton. Però com que $t_1 - t_0$ és el pas que va de t_0 a t_1, el millor és fer la integració, és a dir, cridar `rk45f`, amb el pas

$$h = -\frac{N_1(t_0) - A}{\dot{N}_1(t_0)},$$

i repetir el procés fins que $|N_1(t_n) - A|$ sigui prou petit, la qual cosa vol dir que tenim el t_f i el punt de tall.

El criteri de parada damunt de la secció el fixem mirant si $|N_1(t_n) - A| <$`toln`, on `toln` és una variable que passarem com a paràmetre de la funció i que en el nostre cas pot valer de

l'ordre de 10^{-16} sense que apareguin problemes de convergència[5]. Els altres paràmetres que es passen a la funció són els necessaris per a la crida de la funció `rk45f` que té dins seu.

Llista de la funció poinca

```
#include<stdio.h>
#include<math.h>

void poinca(double ti, double *tf, double xi[], double xf[], double tol,
            double toln, double hmin, double hmax, int n, double A,
            void(*camp)(double, double*, int, double*))
 {
  double h,va,y[2];
  void rk45f(double *t,double x[],int n,double *h,double hmin,double hmax,
             double tol,void(*camp)(double,double*,int,double*));
  *tf=ti;
  xf[0]=xi[0];
  xf[1]=xi[1];
  h=0.01;
  /* La seguent crida nomes es per separar el punt de la seccio */
  rk45f(tf,xf,n,&h,hmin,hmax,tol,camp);
  do
    {
     va=xf[0];
     rk45f(tf,xf,n,&h,hmin,hmax,tol,camp);
    } while (va<A || xf[0]>A);
  /* Quan s'arriba aqui s'ha detectat el tall amb la seccio.
     Usem el metode de Newton per a refinar-lo. */
  while (fabs(xf[0]-A) > toln)
   {
    (*camp)(*tf,xf,n,y);
    h=(A-xf[0])/y[0];
    /* printf("Newton. tf,h,error: %le %le %le\n",*tf,h,A-xf[0]); */
    rk45f(tf,xf,n,&h,hmin,hmax,tol,camp);
   }
 }
```

Fem només unes precisions sobre la funció anterior

- En el `while` del mètode de Newton s'ha afegit un `printf` comentat, que pot servir per veure el nombre d'iteracions i la rapidesa de la convergència d'aquest mètode si es descomenta.

- Durant el métode de Newton hem de cridar la funció `rk45f` amb el pas, `h`, (positiu o negatiu) que s'ha calculat. En principi sembla que s'hauria de fer `hmin=hmax=h`. Però

[5] Cal remarcar, però, que no és necessari calcular el punt damunt de la secció amb 16 decimals si la integració es fa amb una tolerància `tol` molt més gran.

recordem que a causa del funcionament de la funció `rk45f`, si es crida aquesta funció amb un pas que en valor absolut és menor que `hmin`, i l'estimació d'error, resulta més petita que `tol`, avança amb el pas donat (malgrat que el que retorna si que està en valor absolut entre `hmin` i `hmax`). Com que el pas predit per Newton serà menor que el de la darrera integració, la qual s'havia fet per sota del marge d'error `tol`, ens estalviem els canvis de `hmin` i de `hmax`.

Finalment només ens falta un programa principal que utilitzi la funció `poinca` per trobar l'òrbita periòdica i el seu període. Notem que crides successives a la funció `poinca` ens aniran donant valors (N_1, N_2) que sempre tindran $N_1 = A$, ja que es trobaran damunt la superfície de secció, mentre que els valors de N_2 s'aniran acostant a un valor pel qual (A, N_2) es troba damunt l'òrbita periòdica. Fixarem així un valor `erro` de manera que, quan tinguem dos valors consecutius de N_2 donats per la funció `poinca` i que difereixin en menys de `erro`, entendrem que tenim l'òrbita calculada amb la precisió volguda. Usem la variable `error` per contenir el valor actual de la diferència entre dues crides consecutives de la funció `poinca`.

Donem a continuació el programa principal que ens pot servir per a aquest propòsit

Programa principal per al càlcul de l'òrbita periòdica

```
#include<stdio.h>
#include<math.h>

main()
 {
  int n,iter;
  double ti,tf,xi[2],xf[2],tol,toln,hmin,hmax,vtall,erro,error,erra;
  void poinca(double ti,double *tf,double xi[],double xf[],double tol,
            double toln,double hmin,double hmax,int n,double A,
            void(*camp)(double, double*, int, double*));
  void prepre(double t,double x[],int n,double y[]);
  puts("Dona els valors inicials de N1 i N2");
  scanf("%le %le",&xi[0],&xi[1]);
  n=2;
  hmin=1.e-4;
  hmax=1.e0;
  tol=1.e-10;
  toln=1.e-12;
  vtall=2.e0;
  erro=1.e-10;
  ti=0.e0;
  poinca(ti,&tf,xi,xf,tol,toln,hmin,hmax,n,vtall,prepre);
  iter=0;
  erra=1.e0; /* Inicialitzem un valor arbitrari != 0 */
  do
   {
    ti=0.e0;
```

```
    xi[0]=xf[0];
    xi[1]=xf[1];
    poinca(ti,&tf,xi,xf,tol,toln,hmin,hmax,n,vtall,prepre);
    error=xf[1]-xi[1];
    iter++;
    printf("Iteracio: %d Error orbita periodica: %le \n",iter,error);
    printf("Factor de millora: %le\n",error/erra);
    erra=error;
   } while (fabs(error) > erro);
  puts("\n      ORBITA PERIODICA TROBADA \n");
  printf("N1, N2 d'orbita periodica: %24.10le %24.10le\n",xf[0],xf[1]);
  printf("Periode: %24.12le\n",tf);
 }
```

Notem que en aquest programa apareix una variable, `erra`, la qual s'inicialitza a un valor distint de zero i serveix per recordar la diferència de valors entre la crida penúltima i última de la funció `poinca`. És a dir, conté el valor de la variable `error` obtingut per a la crida anterior de `poinca`. Tot seguit n'indiquem la utilitat.

Si usem aquest programa per calcular l'òrbita periòdica donant com a valor inicial de (N_1, N_2) el $(2, 2)$, resulta que al cap de 41 iteracions el programa es para indicant que ha trobat l'òrbita periòdica.

Acceleració de la convergència

En el càlcul que ens ocupa, esperar les 41 iteracions per obtenir l'òrbita periòdica és del tot plausible. Ara bé, en altres casos pot passar que el temps necessari per fer una iteració sigui realment gran i que el càlcul de l'òrbita usant el mètode anterior fins a la precisió volguda sigui inviable.

El nostre programa únicament el que fa és trobar una successió de punts, $(y_k)_{k\geq 0}$, damunt de la superfície de secció `vtall`$= A$. Aquesta successió tendeix a un punt fix, y_p, que representa l'òrbita periòdica donada per la condició inicial (A, y_p):

$$y_0, y_1, y_2, \ldots, y_n, y_{n+1}, y_{n+2}, \ldots \to y_p\,.$$

Els punts y_k es van obtenint per aplicació reiterada de la funció `poinca` i, de fet, damunt de la secció $N_1 = A$ sempre s'agafa com a nou punt inicial el darrer que ens dóna aquesta funció. És a dir, integrem sempre la mateixa òrbita i no ens parem fins que hem arribat prou a prop de la periòdica.

El programa ens dóna, però, a més, una informació que és bastant valuosa. Notem que si la funció `poinca` ens acaba de donar el valor y_{n+2} aleshores les variables `erro` i `erra` prenen els valors

$$\texttt{erro} = y_{n+2} - y_{n+1} \quad \text{i} \quad \texttt{erra} = y_{n+1} - y_n\,.$$

I el que s'escriu amb el nom de "`factor de millora`" no és res més que

$$\frac{\texttt{erro}}{\texttt{erra}} = \frac{y_{n+2} - y_{n+1}}{y_{n+1} - y_n}\,.$$

Aquesta quantitat a mesura que avança el procés, tal com es veu en fer córrer el programa, s'acosta a un valor constant, r, proper a 0.5885. És a dir, que

$$y_{n+2} - y_{n+1} \simeq r(y_{n+1} - y_n), \tag{4.12}$$

i, per tant, ens acostem al criteri de parada multiplicant cada vegada pel factor r. Una convergència d'aquest tipus s'anomena *convergència lineal* i ve a dir que si $\varepsilon_n = y_p - y_n$ és l'error que tenim al n-è iterat llavors $\varepsilon_{n+1} \simeq \varepsilon_n$.

Els processos que tenen convergència lineal se'ls pot accelerar sense gaire esforç usant *l'algorisme d'Aitken*. Aquest mètode i les seves propietats de convergència es poden trobar en molts capítols de llibres de càlcul numèric dedicats al càlcul de zeros de funcions, per exemple a [1]. Aquí només donarem una idea intuïtiva de l'obtenció de l'expressió que ens porta a l'algorisme d'Aitken.

Suposem que en lloc de (4.12) tinguéssim $y_{n+2} - y_{n+1} = r(y_{n+1} - y_n)$ per a tot n d'un cert lloc en endavant. Aleshores el valor y_p es podria obtenir com

$$y_p = y_n + \sum_{k=n}^{\infty} (y_{k+1} - y_k),$$

però segons la nostra hipòtesi les distàncies $d_k = y_{k+1} - y_k$ estan en progressió geomètrica de raó r amb $|r| < 1$. Per tant, usant la fórmula de la suma infinita per a una progressió geomètrica de raó $|r| < 1$ tenim

$$y_p = y_n + \frac{y_{n+1} - y_n}{1 - \frac{y_{n+2} - y_{n+1}}{y_{n+1} - y_n}},$$

que ens dóna

$$y_p = y_n - \frac{(y_{n+1} - y_n)^2}{y_{n+2} - 2y_{n+1} + y_n}. \tag{4.13}$$

És clar que aplicada per a un cas concret la relació de distàncies no és exactament la donada per la igualtat sinó que tenim la donada per (4.12) i, per tant, l'aplicació de (4.13) no dóna exactament y_p, sinó un valor y'_n que en principi s'ha d'acostar més a y_p que el que tindríem calculant y_{n+3} pel procés habitual:

$$y'_n = y_n - \frac{(y_{n+1} - y_n)^2}{y_{n+2} - 2y_{n+1} + y_n}. \tag{4.14}$$

Per tant, a fi d'usar el mètode d'Aitken calcularem tres valors y_n, y_{n+1} i y_{n+2} consecutius de la manera habitual. A partir d'aquest extrapolarem el valor y'_n i l'usarem per calcular dos nous valors de la manera habitual. A partir d'aquests tres darrers valors (el y'_n i els dos calculats de la manera usual) tornarem a extrapolar per Aitken i repetirem el procés. Notem, doncs, que podem extrapolar per Aitken quan tenim tres valors que compleixen (4.12). Per això, a partir d'un valor trobat per Aitken abans de poder tornar a aplicar Aitken cal calcular dos valors més pel procediment que usa el programa anterior.

Si es vol implementar l'algorisme d'Aitken al programa anterior només cal fer dues coses. La primera d'elles és afegir les declaracions d'algunes variables que fan falta; per això posem

```
double jm3,xai[3],del1,del2;
```

al lloc de declaracions.

Finalment inserim el tros següent de codi:

```
jm3=iter%3;
if (jm3 == 0 && iter !=0)
 {
  del1=(xai[1]-xai[0]);
  del2=(xai[2]-2.e0*xai[1]+xai[0]);
  xi[1]=xai[0]-(del1*del1)/del2;
  xi[0]=vtall;
  printf("Extrapolo per Aitken, N2=%24.16le \n",xi[1]);
 }
xai[jm3]=xi[1];
```

justament abans de la crida de la funció `poinca` que es troba dins el "`do`". És a dir entre les línies `xi[1]=xf[1];` i `poinca(ti,&tf,xi,xf,tol,toln,hmin,hmax,n,vtall,prepre);`.

Volem fer notar que durant el procés de compilació pot aparèixer algun *warning* degut al fet que el vector `xai` sembla que es fa servir abans de ser inicialitzat. Això en realitat no és així ja que la línia

```
if (jm3 == 0 && iter !=0),
```

té cura que això no passi. Hem preferit que apareguin els *warnings* ja que d'aquesta manera la modificació del programa és molt clara.

Usant aquest algorisme modificat, l'òrbita periòdica es calcula ara en només 13 iteracions en lloc de les 41 anteriors. Donant com a valors

$$y_p \simeq 6.7875942107, \quad \text{i} \quad \text{període} = 16.0730686084.$$

A més el lector pot comprovar que demanant precisions més altes a l'òrbita periòdica és quan Aitken realment es posa de manifest[6]. Així, si es posa `erro`$= 10^{-14}$, es passa de 58 iteracions sense Aitken a només 15 usant Aitken. La diferència en els salts de 41 a 58 respecte de les 13 a 15 és degut al fet que l'extrapolació d'Aitken funciona quan més a prop estem del fet $y_{n+2} - y_{n+1} = r(y_{n+1} - y_n)$. En les primeres iteracions això és lluny de ser cert i, per tant, Aitken extrapola valors més dolents. Mentre que en estar a prop de l'òrbita periòdica això és gairebé cert i les extrapolacions donades per Aitken són realment molt bones. És convenient que el lector usi aquests programes per veure les millores de la convergència usant Aitken segons els valors inicials i les toleràncies d'error permeses.

Un cop trobada l'òrbita periòdica es poden calcular altres valors com poden ser els nivells màxims i mínims de cada espècie o bé les seves mitjanes definides com

$$\bar{N}_1 = \frac{1}{T}\int_0^T N_1(t)dt, \quad \bar{N}_2 = \frac{1}{T}\int_0^T N_2(t)dt,$$

on T representa el període de l'òrbita.

Per al propòsit del càlcul de la integral que defineix les mitjanes, es pot usar el mètode dels trapezis, que es troba en qualsevol llibre bàsic de càlcul numèric (per exemple, [1]), o es pot

[6] En aquest cas caldrà també canviar la tolerància `tol` d'integració de l'òrbita periòdica.

modificar el camp a fi i efecte d'integrar al mateix temps les dues integrals que defineixen les mitjanes.

Per tancar aquest exemple donem el programa principal que es pot fer servir per calcular els valors màxim i mínim de cada espècie, com també les seves mitjanes pel mètode dels trapezis. Per fer-ho senzill, la integració es realitza a un pas constant, `h`, determinat pel nombre de subintervals, `ncr`, en què se subdivideix l'òrbita periòdica, però també es pot fer amb pas variable canviant convenientment el programa.

Càlcul dels valors extrems i mitjanes

```
#include<stdio.h>
#include<math.h>

main()
 {
  int i,n,ncr;
  double t,x[2],h,hmin,hmax,tol,vmin1,vmin2,vmax1,vmax2,pro1,pro2;
  double perio;
  void rk45f(double *t,double x[],int n,double *h,double hmin,double hmax,
             double tol,void(*camp)(double,double*,int,double*));
  void prepre(double t,double x[],int n,double y[]);
  n=2;
  /* Aqui posem les condicions inicials de l'orbita periodica i el
     seu periode */
  x[0]=2.e0;
  x[1]=6.7875942107;
  perio=16.0730686084;
  /* Poseu a la linia seguent el nombre de subintervals que es volen
     usar per a la integracio numerica */
  ncr=200;
  h=perio/ncr;
  hmin=fabs(h);
  hmax=fabs(h);
  /* Integrant amb pas constant la linia seguent no te sentit */
  tol=1.e-8;
  t=0.e0;
  vmin1=x[0];
  vmin2=x[1];
  vmax1=x[0];
  vmax2=x[1];
  pro1=x[0];
  pro2=x[1];
  for (i=1;i<=ncr;i++)
   {
    rk45f(&t,x,n,&h,hmin,hmax,tol,prepre);
    if (x[0] < vmin1) vmin1=x[0];
```

```
    if (x[0] > vmax1) vmax1=x[0];
    if (x[1] < vmin2) vmin2=x[1];
    if (x[1] > vmax2) vmax2=x[1];
    if (i==ncr)
     {
      pro1+=x[0];
      pro2+=x[1];
     }
    else
     {
      pro1+=2.e0*x[0];
      pro2+=2.e0*x[1];
     }
    /* printf("t,N1 i N2: %le %le %le \n",t,x[0],x[1]); */
   }
  pro1=(0.5e0*h)*pro1/perio;
  pro2=(0.5e0*h)*pro2/perio;
  printf("\n\n Resultats integracio amb pas ctant h=%le\n\n",h);
  printf(" Valors maxim i minim de N1: %le %le \n",vmax1,vmin1);
  printf(" Valor mitja de N1: %le \n",pro1);
  printf(" Valors maxim i minim de N2: %le %le \n",vmax2,vmin2);
  printf(" Valor mitja de N2: %le \n",pro2);
}
```

Aquest programa dóna els resultats següents,

	màxim	mínim	mitjana
N_1	3.39956	0.65302	1.90675
N_2	6.81415	4.57188	5.62189

4.8.2 Comportament caòtic. Sistema de Hénon-Heiles

En aquest exemple donarem una noció molt bàsica del que s'entén per *caos* en els sistemes dinàmics. Ho il·lustrarem amb l'anomenat sistema de Hénon-Heiles fent diversos dibuixos *d'aplicacions de Poincaré* de les quals ja n'hem parlat una mica a l'exemple anterior.

El model del sistema de Hénon-Heiles ve donat pel següent sistema de quatre equacions diferencials ordinàries,

$$\begin{cases} \dot{x}_0 &= x_2, \\ \dot{x}_1 &= x_3, \\ \dot{x}_2 &= -x_0 - 2x_0x_1, \\ \dot{x}_3 &= -x_1 - x_0^2 + x_1^2, \end{cases}$$

i per tant les seves trajectòries són corbes $(x_0(t), x_1(t), x_2(t), x_3(t))$ dins l'espai 4-dimensional, que podrem seguir amb el nostre integrador `rk45f`, un cop haguem triat els quatre valors, $x_0(0)$, $x_1(0)$, $x_2(0)$ i $x_3(0)$, que constitueixen una condició inicial.

El primer que notarem és que la funció de quatre variables

$$H(x_0, x_1, x_2, x_3) = \frac{1}{2}(x_0^2 + x_1^2 + x_2^2 + x_3^2) + x_0^2x_1 - \frac{1}{3}x_1^3,$$

és una *integral primera* del sistema d'equacions. Això vol dir que, damunt de cada trajectòria, la funció H pren un cert valor constant, valor que canvia de trajectòria en trajectòria. A aquesta funció H, per raons mecàniques sobre l'origen del sistema d'Hénon-Heiles, li direm "Energia". Per tant, afirmem que l'energia d'una trajectòria es conserva al llarg d'ella[7]. Això es pot veure matemàticament comprovant que

$$\frac{d}{dt}H\left(x_0(t), x_1(t), x_2(t), x_3(t)\right) = 0,$$

és a dir, veient que la derivada respecte del temps de la funció H al llarg d'una trajectòria val zero i per tant H hi ha de ser constant.

En el nostre cas és fàcil de comprovar, ja que si la derivem respecte del temps, usant la regla de la cadena, tenim

$$\frac{d}{dt}H\left(x_0(t), x_1(t), x_2(t), x_3(t)\right) = \sum_{i=0}^{3} \frac{\partial H}{\partial x_i}\dot{x}_i =$$

$$= (x_0 + 2x_0x_1)\dot{x}_0 + (x_1 + x_0^2 - x_1^2)\dot{x}_1 + x_2\dot{x}_2 + x_3\dot{x}_3,$$

que s'anul·la si substituïm $\dot{x}_0$, $\dot{x}_1$, $\dot{x}_2$, i $\dot{x}_3$ per les seves expressions donades en el model inicial.

Com que cada òrbita té una energia que es conserva al llarg d'ella, podem classificar les òrbites per la seva energia. Intuïtivament, d'aquesta manera si en principi hem de fixar els quatre valors $x_0(0)$, $x_1(0)$, $x_2(0)$ i $x_3(0)$ per tenir una condició inicial; si el que volem és que les òrbites que integrem tinguin totes elles una certa energia, E, només podrem fixar tres dels quatre valors de la condició inicial, ja que el que ens queda l'haurem de triar de manera que

$$H\left(x_0(0), x_1(0), x_2(0), x_3(0)\right) = E.$$

Procedint com acavem d'apuntar, podrem anar representant les òrbites del model Hénon-Heiles segons l'energia que tinguin. El problema és que, només fixant l'energia, el conjunt d'òrbites que ens resta encara és molt gran, ja que hem de triar tres valors arbitraris i això es pot fer de moltes maneres. Tenim el que es diu *tres graus de llibertat*.

A fi de reduir els graus de llibertat, fixarem una *superfície de secció*, tal com haviem fet en l'exemple de la secció anterior. Aleshores només ens mirarem les òrbites en el moment de tallar aquesta superfície de secció.

La superfície de secció la pendrem amb la condició[8] $x_0 = 0$, i representarem *l'aplicació de Poincaré* damunt d'ella. Així, doncs, agafarem una condició inicial $(x_0(0), x_1(0), x_2(0), x_3(0))$ damunt de la secció (és a dir amb $x_0(0) = 0$) i amb una energia, E, donada (és a dir amb $H(0, x_1(0), x_2(0), x_3(0)) = E$). Integrarem llavors la trajectòria fins a tornar a tallar la secció. És el que s'anomena, tal com hem indicat en l'exemple anterior, trobar la imatge del punt de la secció per l'aplicació de Poincaré.

A una condició incial damunt de la superfície de secció, li anem aplicant reiteradament l'aplicació de Poincaré. Així es veu el que fa l'òrbita mitjançant "el rastre" de punts que va deixant damunt de la superfície de secció en atravessar-la.

[7] D'una manera semblant a la que conservava el radi en l'exemple "`cercle`" usat com a test de la funció `rk45f`.
[8] Més endavant veurem que falta una condició addicional.

Comentem a continuació el procés que seguim per fer tot això.

Dels quatre valors que necessitem per a una condició inicial tenim que $x_0(0) = 0$ i per tant ens cal buscar els altres tres. Fixem un valor E de l'energia, llavors la relació

$$H(0, x_1(0), x_2(0), x_3(0)) = E$$

ve donada per

$$2E = x_1^2(0) + x_2^2(0) + x_3^2(0) - \frac{2}{3}x_1^3(0),$$

d'on podem aïllar $x_2(0)$ segons

$$x_2(0) = \pm\sqrt{2E - x_1^2(0) - x_3^2(0) + \frac{2}{3}x_1^3(0)}.$$

Suposem que hem fixat un valor de l'energia E, i donem valors arbitraris a $x_1(0)$ i a $x_3(0)$. Aleshores podem calcular $x_2(0)$ per la fórmula anterior i, com que $x_0(0) = 0$, ja tenim completa la condició inicial.

Notem però que per al càlcul de $x_2(0)$ tenim dues possibilitats: el signe positiu o el negatiu. A més, segons el model de Hénon-Heiles $x_2(0)$, es correspon amb $\dot{x}_0(0)$. Resulta que la superfície de secció no queda només determinada per a $x_0 = 0$ sinó que també hem de triar que el tall sempre es produeixi en el mateix sentit[9]. Podem triar arbitràriament un signe o l'altre, però sempre el mateix. En el nostre cas triem la superfície de secció com a $x_0 = 0$ i $\dot{x}_0 > 0$ és a dir amb $x_2 > 0$.

Sabem trobar doncs una condició inicial $(x_0(0), x_1(0), x_2(0), x_3(0))$. Ara escriurem la funció `aplpoi` que fa l'aplicació de Poincaré. Donada aquesta condició inicial en el vector `x` i a l'instant `t` (que inicialment el prendrem zero), torna, dins les mateixes variables `t` i `x`, l'instant i el punt de tall amb la secció. Es a dir, integra la condició inicial donada fins que `x[0]=0` i `x[2]>0`.

Fem el càlcul del punt de tall, de manera anàloga que a l'exemple de la secció anterior, o sigui, aplicant el mètode de Newton per trobar un zero de la funció $F(t) = x_0(t) = 0$, i per tant en les iteracions de refinament prenem el pas

$$h = -x_0(t)/\dot{x}_0(t) = -x_0(t)/x_2(t),$$

fins que $|x_0(t)| <$`toln`, on `toln` és un paràmetre que es passa a la rutina.

Els altres paràmetres que té la funció `aplpoi` són els que li calen a la funció d'integració `rk45f` que es crida dins seu.

Aplicació de Poincaré per a Hénon-Heiles

```
#include<stdio.h>
#include<math.h>

void aplpoi(double *t, double x[], double tol, double toln, double hmin,
          double hmax, int n, void(*camp)(double, double*, int, double*))
```

[9] Això ja ens ho havíem trobat a l'exemple anterior quan miràvem que el tall es produís de $N_1 > A$ a $N_1 < A$.

```
{
 double h,va;
 void rk45f(double *t,double x[],int n,double *h,double hmin,double hmax,
            double tol,void(*camp)(double,double*,int,double*));
 h=0.01;
 /* La seguent crida nomes es per separar el punt de la seccio */
 rk45f(t,x,n,&h,hmin,hmax,tol,camp);
 do
   {
    va=x[0];
    rk45f(t,x,n,&h,hmin,hmax,tol,camp);
   } while (va>0.e0 || x[0]<0.e0); /* Aqui es busca x0=0 i x2>0$
 /* Quan s'arriba aqui s'ha detectat el tall amb la seccio.
    Usem el metode de Newton per a refinar-lo. */
 while (fabs(x[0]) > toln)
  {
   h=-x[0]/x[2];
   /* printf("Newton. tf,h,error: %le %le %le\n",*t,h,x[0]); */
   rk45f(t,x,n,&h,hmin,hmax,tol,camp);
  }
}
```

Aquesta funció és molt semblant a la funció `poinca` de l'exemple anterior, i de fet se'n pot fer una d'única que valgui per als dos casos.

Com sempre, necessitem també la funció que ens avaluï el camp en un punt i un instant donat. Li diren `henhe`.

```
void henhe(double t, double x[], int n, double y[])
 {
  y[0]=x[2];
  y[1]=x[3];
  y[2]=-x[0]-2.e0*x[0]*x[1];
  y[3]=-x[1]-x[0]*x[0]+x[1]*x[1];
 }
```

Finalment, ja només ens falta escriure el programa principal. Abans, però, anem a precisem una mica més el que farem.

Suposem que fixem d'entrada el nivell d'energia E, que notarem per `ener` en el programa. En el pla (x_1, x_3), que és el punt en què hem d'agafar els valors que tenim arbitraris, hi farem una finestra rectangular de vèrtexs (`xmim`,`ymin`), (`xmax`,`ymax`). Dins d'aquesta finestra triem una parella de valors $(x_1(0), x_3(0))$ mitjançant un reticulat de `npx` punts en el costat de x_1 i de `npy` punts en el de x_3, tal com es veu a l'exemple de la figura 4.16.

Per a cada punt $(x_1(0), x_3(0))$ del reticle, calculem el valor de x_2 partint del fet que

$$x_2(0) = \sqrt{2E - x_1^2(0) - x_3^2(0) + \frac{2}{3}x_1^3(0)}.$$

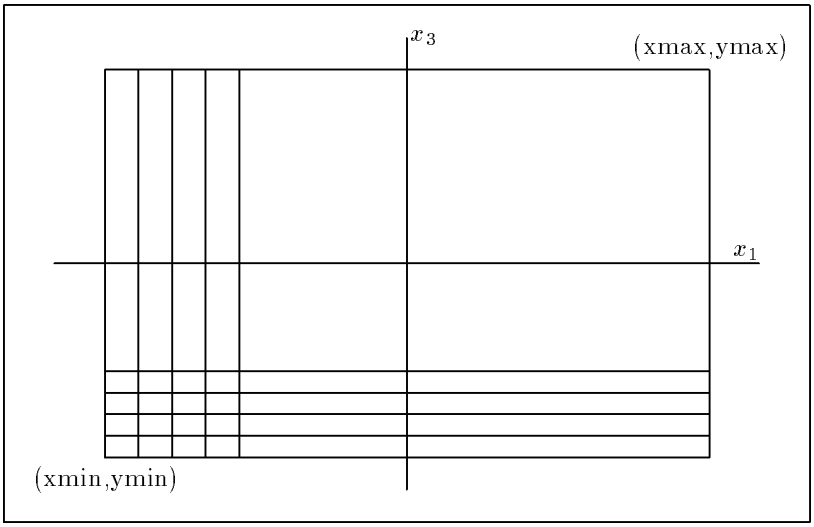

Figura 4.16: *Reticle emprat en la representació de les aplicacions de Poincaré del model de Hénon-Heiles. Els valors de x_1 i de x_3 corresponents a les condicions inicials els prenem de les interseccions que formen el reticle.*

Però si per a algun valor de $(x_1(0), x_3(0))$ l'expressió de dins l'arrel és negativa, tindrem que $x_2^2(0) < 0$, la qual cosa és impossible i ens diu que no hi ha cap òrbita, amb l'energia E donada, que en un cert moment tingui $x_1 = 0$ i x_1 i x_3 com els valors triats. Quan trobem $x_2^2 < 0$, descartarem doncs la parella $(x_1(0), x_3(0))$ i passarem a un nou punt del reticle.

Pel procediment anterior obtenim condicions inicials $(0, x_1(0), x_2(0), x_3(0))$ amb l'energia, E, donada d'entrada i damunt de la superfície de secció. Aleshores amb la funció `aplpoi` busquem el següent punt de la trajectòria damunt de la secció. A partir d'aquest darrer punt buscarem el següent, i així successivament iterem l'aplicació de Poincaré un nombre prou gran de vegades (`iterp` en el programa). Els punts obtinguts per l'aplicació de Poincaré, els anem dibuixant en el pla (x_1, x_3).

El programa principal que fa tot això és el següent:

Programa principal per al sistema Hénon-Heiles

```
#include<stdio.h>
#include<math.h>
#include"grafbas.h"

main()
 {
  int n,ix,iy,npx,npy,it,iterp,col;
  double t,x[4],tol,toln,hmin,hmax,xmin,xmax,ymin,ymax,pasx,pasy,ener;
  double aux;
```

```
 void aplpoi(double *t,double x[],double tol,double toln,double hmin,
         double hmax,int n,void(*camp)(double, double*, int, double*));
 void henhe(double t,double x[],int n,double y[]);
 /* Constants necessaries per a la funcio aplpoi */
 n=4;
 tol=1.e-10;
 hmin=1.e-4;
 hmax=1.e0;
 toln=1.e-14;
 /* Valor de l'energia, finestra de treball, reticle  i nombre
    d'iterats de l'aplicacio de Poincare */
 ener=1.e0/20.e0;
 xmin=-0.8e0;
 xmax=0.8e0;
 ymin=-0.8e0;
 ymax=0.8e0;
 npx=50;
 npy=50;
 iterp=200;
 /* comencem els calculs i el dibuix */
 pasx=(xmax-xmin)/(npx-1.e0);
 pasy=(ymax-ymin)/(npy-1.e0);
 inigraf();
 finestra(xmin,xmax,ymin,ymax);
 col=prencolor();
 for (ix=0; ix<npx; ix++)
   for (iy=0; iy<npy; iy++)
    {
     x[1]=xmin+ix*pasx;
     x[3]=ymin+iy*pasy;
     aux=2.e0*ener-x[1]*x[1]-x[3]*x[3]+(2.e0/3.e0)*x[1]*x[1]*x[1];
     if (aux < 0.e0) continue;
     x[2]=sqrt(aux);
     x[0]=0.e0;
     t=0.e0;
     pospun(x[1],x[3],col);
     for (it=0; it<iterp; it++)
      {
       aplpoi(&t,x,tol,toln,hmin,hmax,n,henhe);
       pospun(x[1],x[3],col);
      }
    }
 getch();
 tancagraf();
}
```

El lector pot fer servir aquest programa per fer diferents dibuixos, amb diferents valors de

l'energia i diferents reticles en el pla (x_1, x_3). A les figures 4.17–4.21 representem alguns dels resultats.

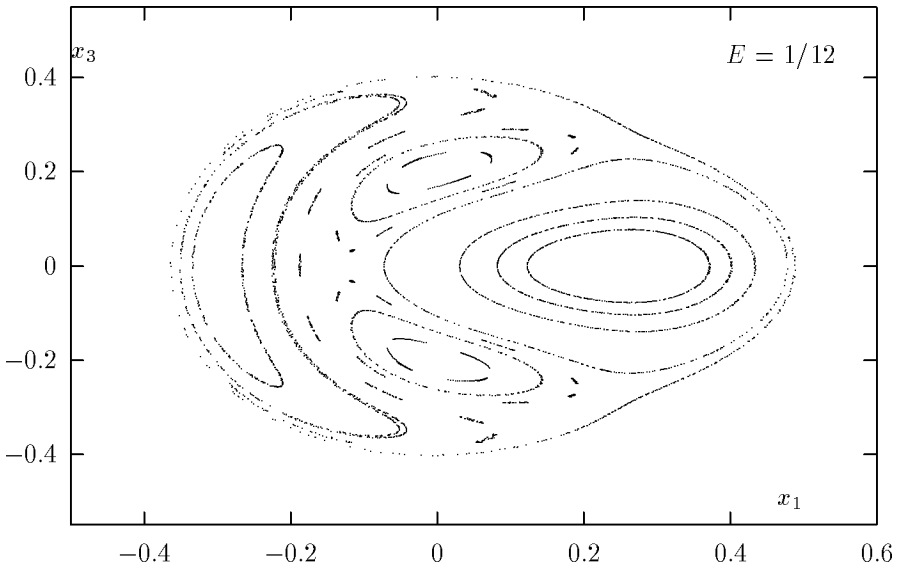

Figura 4.17: *Representació de l'aplicació de Poincaré del model de Hénon-Heiles per al valor de l'energia* $H = 1/12$.

D'aquestes figures, és interessant observar que per a valors baixos de l'energia, en tallar la superfície de secció les òrbites produeixen, o millor dit emplenen, (ja que no es fa de manera consecutiva), unes corbes que al voltant del punt central semblen a el·lipses. Totes les corbes que apareixen reben el nom de *corbes invariants*, la qual cosa vol dir que, si fixem una òrbita, quan aquesta talli la secció sempre ho farà en algun punt de la corba invariant que "té assignada". De vegades una corba invariant la componen diverses corbes tancades més petites anomenades *illes*. A més, enmig de les corbes queden uns punts aïllats que són periòdics per l'aplicació de Poincaré. És a dir, si es parteix d'un d'aquests punts, al cap d'una o més iteracions de l'aplicació de Poincaré s'hi torna exactament. El que vol dir que són punts pels quals passa una òrbita periòdica. Amb tot això es té l'efecte que quan apareixen aquestes corbes hi ha un "moviment ordenat".

Quan s'augmenta l'energia, les corbes anteriors van desapareixent i es veuen àmplies regions, les quals semblen totes elles recorregudes per una sola òrbita en les seves iteracions. És impredictible el fet de saber on aniran a parar els propers iterats. Ja no es disposen ordenadament damunt d'una corba, com passa en els dibuixos d'energies més baixes, sinó que emplenen els anomenats *mars caòtics* i el moviment esdevé molt més "desordenat". Es té l'anomenat *caos*, el qual fa impossible predir on es trobarà exactament el punt de tall d'unes poques iteracions en endavant.

Realment cal dir però que, per a les energies petites, els mars caòtics també existeixen. El que passa és que són tan petits que no s'observen en el dibuix. És en augmentar l'energia que es posen realment de manifest.

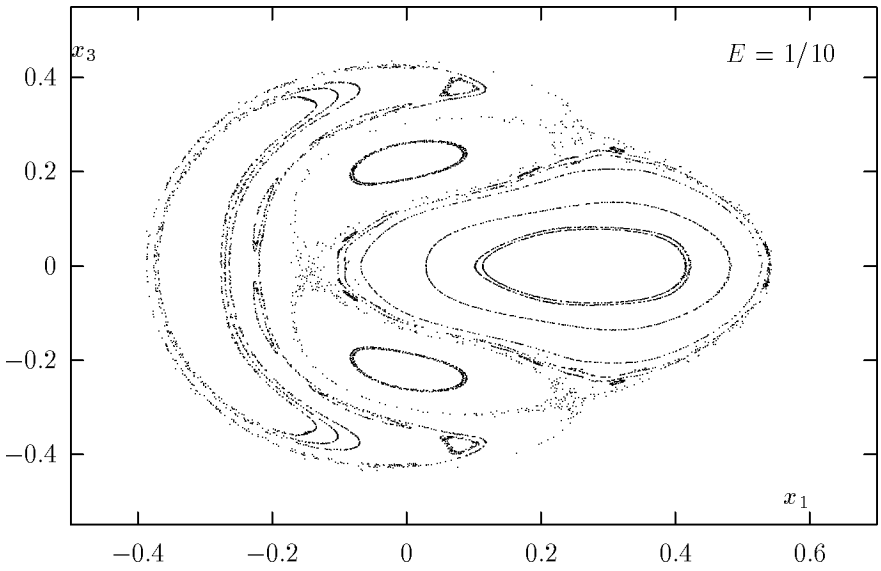

Figura 4.18: *Representació de l'aplicació de Poincaré del model de Hénon-Heiles per al valor de l'energia* $H = 1/10$.

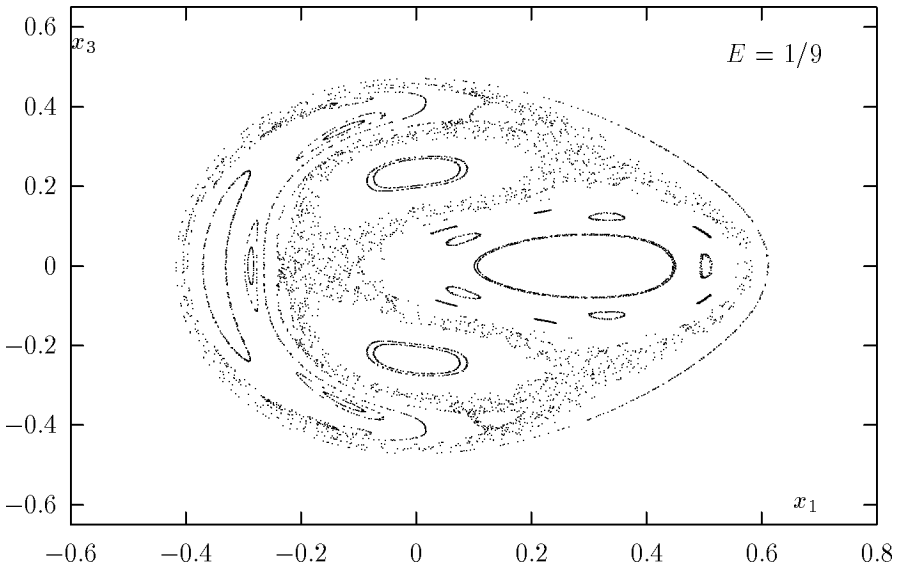

Figura 4.19: *Representació de l'aplicació de Poincaré del model de Hénon-Heiles per al valor de l'energia* $H = 1/9$.

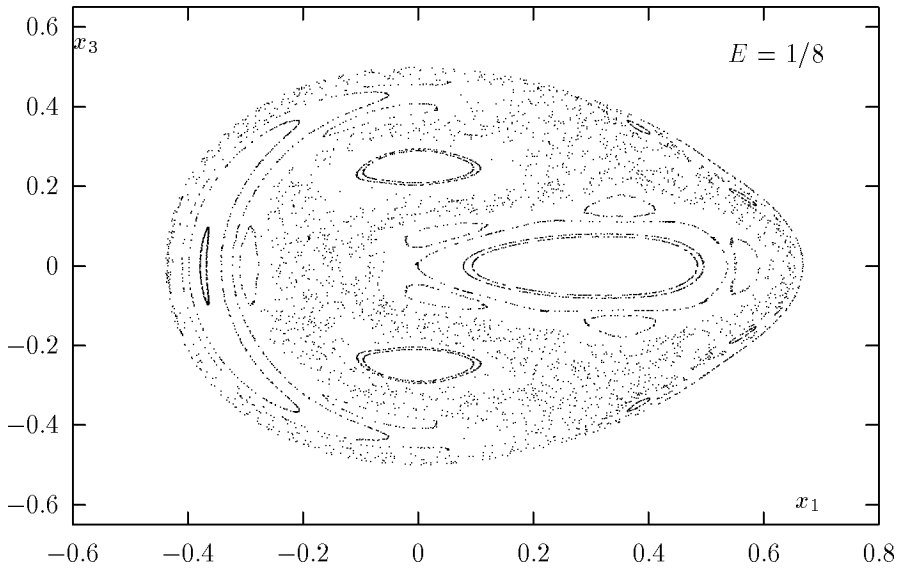

Figura 4.20: *Representació de l'aplicació de Poincaré del model de Hénon-Heiles per al valor de l'energia* $H = 1/8$.

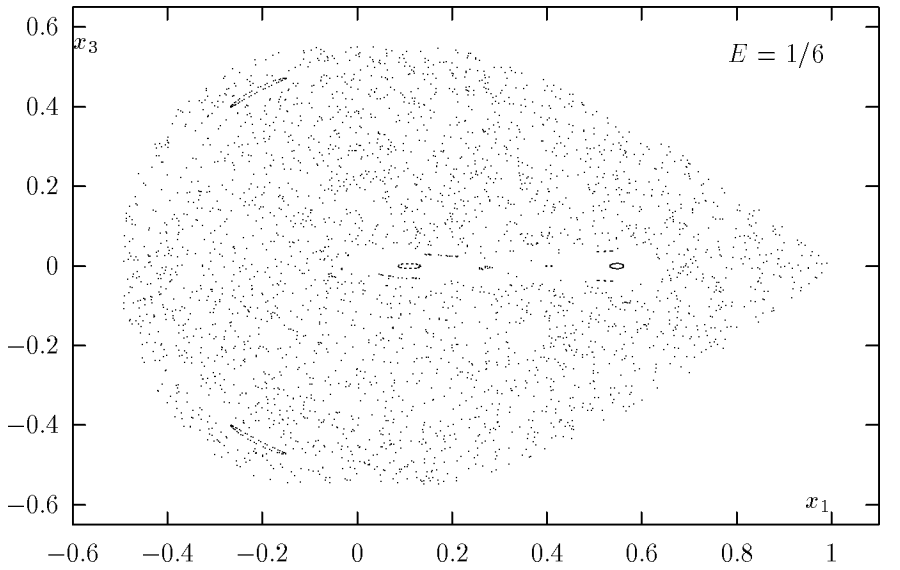

Figura 4.21: *Representació de l'aplicació de Poincaré del model de Hénon-Heiles per al valor de l'energia* $H = 1/6$.

Apèndix A Resum de C

A.1 Introducció

En aquest apèndix trobem una descripció resumida del llenguatge de programació C. Está pensat per a alumnes que ja han estudiat prèviament aquest llenguatge, i que necessiten o bé repassar-lo, o bé buscar algunes qüestions puntuals que no recorden. S'ha posat una mica més d'èmfasi en les parts que solen produir més confusió (com per exemple els apuntadors).

No es pretén resumir totes les possibilitats del llenguatge, sinó només les parts necessàries per poder implementar els algorismes vistos al llarg del llibre. En tot cas, al final es donen algunes referències per al lector que estigui interessat a ampliar coneixements.

Finalment, cal dir que la versió de C que aquí es resumeix és la definida pel corresponent estàndard ANSI.

A.2 Identificadors, variables i operadors

En aquesta secció donem una descripció dels tipus de variables que poseeix el C, juntament amb les operacions bàsiques que s'hi poden fer. Hem de recordar que, en C, cal declarar totes les variables usades.

A.2.1 Identificadors

Un identificador és una tira de lletres i dígits que serveix per donar nom a les variables i funcions del programa. Cal que el primer caràcter sigui una lletra, i el caràcter `_` (línia de subratllar) es pren com una lletra. Es diferencien les lletres majúscules de les minúscules (és a dir, `A` i `a` són dues lletres diferents). Un identificador pot estar format per qualsevol nombre de caràcters, però només s'usen els 31 primers per distingir les variables. Així, dues variables diferents amb els 31 primers caràcters del seu nom iguals, són la mateixa variable.[1]

A.2.2 Tipus de variables

En C tenim diversos tipus de variables:

[1] Aquesta norma no prohibeix que alguns compiladors utilitzin més de 31 caràcters per diferènciar variables.

- **Variables caràcter.** Es declaren amb `char`, ocupen un byte de memòria i contenen un únic caràcter. Les constants que són d'aquest tipus les posarem entre cometes. Per exemple, si declarem `char c;` podem assignar a `c` la constant `z` fent `c='z';`.

- **Variables enteres.** Es declaren amb la paraula `int`, i per regla general ocupen tants bits com la longitud de paraula del processador (en un PC la paraula acostuma a ser de 16 bits). Aquestes característiques es poden alterar amb l'ús dels qualificadors `long, short` i `unsigned`. `long` fa que es reservin 32 bits per a les variables, `short` fa que siguin 16 i `unsigned` fa que les variables no siguin enteres sinó naturals. En aquest últim cas les variables poden contenir valors més grans que quan tenen signe. Així, mentre una variable declarada com `short int` pren valors entre -32768 i 32767, una altra declarada com `unsigned short int`, els pren entre 0 i 65535. Una variable `long int` pot prendre qualsevol valor enter entre -2^{31} i $2^{31}-1$. Les constants enteres de tipus `short` s'escriuen de la manera usual, mentre que a les de tipus `long` se'ls posa una `L` al final: `2` és una constant `short`, però `123456L` és de tipus `long`.

- **Variables reals.** N'hi ha de dos tipus, que es declaren amb les paraules `float` i `double`. Les variables tipus `float` ocupen 4 bytes de memòria mentre que les de tipus `double` n'ocupen 8. Com hom pot imaginar, les variables `double` tenen més dígits significatius i un rang més gran per als exponents que les `float`. Les constants reals s'escriuen en la notació científica habitual: una part decimal amb un punt, una `e` o `E` i un exponent. Per exemple `3.14e-2`.

- **Apuntadors.** Són variables que contenen l'adreça (d'això en direm "apuntar") d'una altra variable, i les declararem del mateix tipus que la variable apuntada, però posant un `*` davant del seu nom. Per exemple, la declaració `int *a;` declara `a` com un apuntador cap a una variable de tipus enter. Més endavant parlarem amb més detall d'aquestes variables.

És possible donar un valor a una variable al mateix temps que és declarada. Per exemple,

```
float p=1.14;
```

crea la variable `p` de tipus `float` amb el valor inicial `1.14`.

A.2.3 Operadors

En C disposem dels operadors aritmètics convencionals `+, -, *, /` i també de l'operador mòdul `%` : si `a` i `b` són variables `int`, `a % b` retorna la resta de la divisió de `a` entre `b`. La precedència d'aquests operadors és la usual, però podem usar parèntesis per alterar-la tal com veurem en el subapartat següent.

Els operadors relacionals són `<` (més petit), `>` (més gran), `<=` (més petit o igual), `>=` (més gran o igual), `==` (igual) i `!=` (no igual).

Els operadors lògics són `&&` (and), `||` (or) i `!` (not).

També tenim operadors que treballen a nivell de bits. Són `&` (and), `|` (or), `^` (or exclusiu), `~` (not (complement a 1)), `>>` (shift cap a la dreta) i `<<` (shift cap a l'esquerra).

Els operadors per treballar amb adreces i apuntadors són `*` i `&`. Més endavant en veurem el funcionament.

Disposem també d'un operador (`sizeof`) que retorna la quantitat de bytes que ocupa un tipus de variable dins la memòria de l'ordinador. Així, `sizeof(float)` és el nombre de bytes necessaris per emmagatzemar una variable de tipus `float`.

Precedències

És important conèixer la precedència d'aquests operadors. La precedència indica quina operació es fa primer quan en tenim més d'una en una instrucció. Així, que la multiplicació tingui més precedència que la suma vol dir que en una instrucció com `a*b+c`, primer es fa el producte entre `a` i `b`, i al resultat se li suma `c`. Per alterar la precedència s'usen parèntesis: si, en el cas d'abans, volem que primer es faci la suma i després el producte, cal posar `a*(b+c)`. La precedència dels operadors aritmètics de C és l'habitual en matemàtiques, però la precedència dels altres operadors ja no queda tant clara. Es per això que hem inclòs la taula A.1.

Conversions de tipus

Quan en una operació intervenen variables de diferent tipus es fan unes conversions prèvies abans d'operar. En general, si un operand aritmètic (com el `+` o el `%`) involucra dos arguments de diferent tipus, el tipus més simple es converteix al més complex, i el resultat és d'aquest mateix tipus. Així, si dividim dos enters, la divisió es farà com una divisió entera i el resultat serà un enter. Si dividim un enter i un `float`, l'enter es passarà a `float` abans de dividir, la divisió es farà com `float`, i el resultat també serà `float`.

Existeix també un operador per forçar la conversió de tipus anomenat *cast*. S'usa posant el nom del tipus al qual volem convertir una expressió, entre parèntesis just abans d'aquesta. Veiem-ne un exemple: suposem que tenim la variable `a` declarada com `double` i `n` i `m` com `int`. Volem assignar a `x` el quocient de `n` i `m`. Si posem `a = n/m;` resultarà que la divisió de `n` entre `m` serà entera (el seu resultat serà enter) i, per tant, en molts casos anirà malament (per exemple, si `n` és 1 i `m` és 3, el resultat de la divisió serà 0). Una solució per a aquest problema és usar l'operador *cast*. En aquest exemple podem fer `a = (double)n/m;`. Com que el *cast* té més precedència que la divisió, es convertiria el valor de `n` a `double` i després es dividiria aquest valor per `m`, essent el resultat final l'esperat.

Hi ha un altre sistema per forçar les conversions de tipus, que és válid també en altres llenguatges. Veiem-lo amb el mateix exemple d'abans. Fem `a = (1.e0*n)/m;`. Aquí, la primera operació que es fa és la que va entre parèntesis i que converteix `n` a `double`. Després el resultat es divideix per `m`, convertint primer `m` a `double`, i es produeix el resultat d'aquest mateix tipus.

Operadors adicionals

Disposem, a més, d'operadors per incrementar i decrementar variables, que són `++` i `--` respectivament. Aquests operadors poden anar com a sufix o bé com a prefix de la variable sobre la qual actuen, essent el seu significat diferent. Veiem-ne un exemple: suposem que `n` i `m` són variables enteres. Si assignem `n=3;`, llavors `m=++n;` fa que *primer* s'incrementi el valor de `n` i que *després* s'assigni aquest valor a `m`. Llavors ens queda que `n` val 4 i `m` també val 4. Si el que posem és `m=n++;`, llavors *primer* assignem el valor de `n` a `m` i *després* incrementem `n`. Obtenim que `n` val 4 i `m` val 3. Evidentment, si no usem el resultat de l'increment, és indiferent posar-lo al davant que al darrere (així, en l'exemple anterior, és equivalent escriure `n=3; ++n; m=n;` o `n=3; n++; m=n;`).

Operadors	Associativitat
`()  []  ->  .`	esquerra a dreta
`!  ~  ++  --  +  -  *  &  (`*cast*`)  sizeof`	dreta a esquerra
`*  /  %`	esquerra a dreta
`+  -`	esquerra a dreta
`<<  >>`	esquerra a dreta
`<  <=  >  >=`	esquerra a dreta
`==  !=`	esquerra a dreta
`&`	esquerra a dreta
`^`	esquerra a dreta
`|`	esquerra a dreta
`&&`	esquerra a dreta
`||`	esquerra a dreta
`? :`	dreta a esquerra
`=  +=  -=  *=  /=  %=  &=  ^=  |=  <<=  >>=`	esquerra a dreta
`,`	esquerra a dreta

Taula A.1: *Precedència i associativitat dels operadors de C.*

Hi ha també uns operadors d'assignació. Entre ells destaquem `+=`, `-=`, `*=`, `/=` i `%=`. Si a i b són expressions, aleshores a `op=` b; és el mateix que `(`a`) = (`a`) op (`b`);`. Per exemple `n*=m-3;` és equivalent a `n=n*(m-3);`.

C disposa d'un operador que en algunes ocasions ens servirà per substituir estructures `if-else`. És l'operador `?`. Es fa servir de la manera següent: si a, b i c són expressions, llavors el valor de l'expressió a `?` b `:` c; s'obté fent el següent. Primer s'avalua a i si el resultat és cert el valor resultant és el que surt d'avaluar b. Altrament és el de c. Per exemple, després d'executar `i=6; j=4; min=(i<j) ?  i :  j;` (on `i`, `j` i `min` són variables enteres) es té que el valor de `min` és 4 (l'ús del parèntesi no és necessari, com es pot veure a la taula de precedències).

Hi ha alguns operadors més que no hem comentat aquí, com són l'operador coma (`,`) i els usats per treballar amb les estructures i les unions (`.` i `->`).

A.3 Control de flux

Ara veurem les sentències més importants de control de flux.

if-else

Serveix per prendre decisions. La forma general d'una instrucció `if` és

```
if (expr) {
        instruccions-1;
    } else {
        instruccions-2;
    }
```

Aquesta instrucció funciona avaluant primer *expr*, i si el resultat és cert, s'executa el bloc corresponent a *instruccions-1*. Altrament s'executa *instruccions-2*. Si *instruccions-1* o *instruccions-2* es redueix a una sola instrucció, no cal posar els corresponents { }. Si *instruccions-2* no hi és, podem ometre la part de l'`else`. Si tenim una seqüència de `if` aniuats, cada `else` s'associarà amb l'`if` sense `else` més proper. Aquest tipus de construcció s'usa, per exemple, quan volem fer una decisió múltiple.

while

S'usa per construir bucles. La seva forma general és

```
while (condició) {
       instruccions;
}
```

Es van executant les *instruccions* mentre la *condició* sigui certa. Si al principi *condició* és falsa, no s'executa cap vegada el bucle. Igual que abans, si *instruccions* es redueix a una sola instrucció, no cal posar els { }.

do-while

És molt similar a l'anterior. La seva forma és

```
do {
   instruccions;
   } while (condició)
```

Funciona igual que el `while` que acabem de veure, però fa la verificació de la condició al final del bucle. Per tant, en aquest cas es pot garantir que el bucle d'instruccions s'executarà almenys una vegada.

for

És una de les sentències que més usarem per fer bucles. La seva forma general és

```
for (inici; condició; increment) {
    instruccions;
}
```

Per explicar el seu funcionament només direm que és equivalent a

```
inici;
while (condició) {
       instruccions;
       increment;
}
```

break

Serveix per abandonar l'execució d'un bucle en qualsevol punt d'aquest.

continue

Igual que l'anterior, s'usa dins d'un bucle i serveix per forçar la següent iteració d'aquest.

switch

Transfereix l'execució a un altre punt del programa, en funció del valor que pren una determinada expressió. La seva forma és

```
switch(expr) {
   case const-1:
      instruccions-1;
      break;
   case const-2:
      instruccions-2;
      break;
      ⋮
   case const-n:
      instruccions-n;
      break;
   default:
      instruccions;
      break;
   }
```

El seu funcionament és com segueix. En primer lloc, s'avalua la expressió *expr* (el valor produït per aquesta expressió ha de ser un enter o un caràcter[2]). Un cop es té el valor d'aquesta expressió, es compara amb les expressions *const-1*, ..., *const-n*, que apareixen després de la instrucció `case`. Si és igual a alguna d'elles (per exemple a *const-2*), el control del programa es transfereix al corresponent grup d'instruccions (en aquest exemple, a *instruccions-2*). Altrament el control és transferit al grup `default`. Finalment, la sentència `break` fa que la execució continuï a partir de la } que tanca el `switch`. Si no es posa aquesta sentència, l'execució continua pel grup d'instruccions corresponent al següent `case` (en l'exemple anterior, s'executaria *instruccions-3*), i si després d'això no es troba un `break`, *instruccions-4* i així succesivament, fins que o bé s'acaba el switch, o bé trobem un `break` que ens fa sortir).

A.4 Funcions

La forma general d'una funció és la següent:

```
tipus nom-funció (paràmetres)
{
      cos de la funció;
}
```

El `tipus` és el tipus de la variable que la funció retorna, que pot ser qualsevol. Si la funció no retorna res, el tipus és `void`. Si no en posem cap, se suposa que retorna un `int`. *paràmetres*

[2]De fet, hi ha alguna possibilitat més. Consulteu un manual de C (per exemple [10]) per als detalls.

és la llista separada per comes dels paràmetres que rep la funció. Cal posar davant de cada paràmetre d'aquesta llista el tipus al qual pertany, com si es tractés d'una declaració. Aixi per exemple, si volem una funció anomenada `fun`, que retorni un valor `double` i tingui dos paràmetres, un d'enter que volem anomenar `a` i un de `float` que volem anomenar `r`, hem de posar

```
double fun(int a, float r)
{
        ⋮
}
```

Si la funció no rep paràmetres, la llista serà buida. Cal remarcar que les instruccions `int a` i `float r` de la capçalera ja declaren les variables `a` i `r` que s'usaran dins de la funció, per tant no cal tornar-les a declarar. També es important notar que el nom d'aquestes variables en el programa principal (o funció) que crida a `fun` pot ser un altre. Únicament ha de coincidir el tipus. Finalment, entre { } es troba el cos de la funció. Aquí es posen totes les instruccions, començant per les declaracions, com si es tractés d'un altre programa.

Hi ha una instrucció molt important pròpia de les funcions, que és

return

Fa que la funció deixi d'executar-se per passar el control del programa a la rutina o programa que l'ha cridat, i al mateix temps fa que retorni el valor de l'expressió que és just després del `return` (per exemple, `return(a);` o `return(i+j);`). Si la funció és de tipus `void`, no es posa cap valor després del `return` (només es posa `return;`). Totes les variables que usa la funció són privades per defecte.

El pas de paràmetres es fa per valor, en el cas de les variables no indexades, i per adreça quan es tracta de vectors o matrius. Això vol dir que els valors dels paràmetres no indexats de la rutina "principal" es copien dins de les variables de la funció i, quan s'acaba l'execució d'aquesta, els valors es perden. En el cas de les variables indexades (vectors i matrius) la funció hi treballa directament ja que no rep cap còpia, sino la direcció on están enmagatzemades. Com que en alguns casos voldrem que la funció retorni diversos valors que no correspondran a variables amb índex, vegem un sistema per fer-ho. Ens caldrà saber una mica més sobre apuntadors. Abans, però, parlarem d'un altre tipus de variables.

A.5 Vectors i matrius

La manera més general de declarar un vector és fer `tipus` *nom*`[`*num*`];`. D'aquesta manera hem declarat un vector anomenat *nom* de *num* components, on cada component és una variable de tipus `tipus`. La primera d'aquestes components és *nom*`[0]` i l'última *nom*`[`*num*`-1]`. Estan posades de manera consecutiva dins la memòria de l'ordinador. Això vol dir que, si declarem `short int v[10]` i l'adreça de `v[0]` és, per exemple, el lloc 1000 dins la memòria de l'ordinador, l'adreça de `v[1]` és el lloc 1002 (=adreça de `v[0]` + nombre de bytes que ocupa `v[0]` = 1000 + 2), la de `v[2]` és 1004, i així succesivament.

La declaració de matrius es fa de manera molt similar. Per exemple, per declarar `a` com una matriu real quadrada de 10 files i 10 columnes fem `double a[10][10]`, on els índexs aniran de 0 a 9. En C, les matrius es guarden per files dins la memòria de l'ordinador. Així, en l'exemple

anterior tenim que la matriu `a` s'enmagatzema com 10 vectors de 10 components cadascun d'ells. Posats un darrere l'altre dins la memòria, cada vector s'identifica amb la corresponent fila de la matriu.

A.6 Apuntadors

Un apuntador és una variable que conté una adreça de memòria. Quan aquesta adreça sigui la d'una altra variable, direm que la primera variable apunta a la segona. Com ja hem dit, la forma general de declarar un apuntador és fer `tipus *`*var*, amb la qual cosa *var* queda declarada com un apuntador a variables del tipus `tipus`.

A.6.1 Operacions amb apuntadors

Hi ha diversos operadors que tenen relació amb apuntadors. L'operador `&` torna l'adreça d'una variable. Per exemple, si `ap` és un apuntador a variables enteres i `n` n'és una, `ap = &n;` fa que `ap` contingui l'adreça de `n` i per tant direm que `ap` apunta a `n`. Un altre operador és `*`, que s'aplica a apuntadors per referir la variable apuntada. Continuant amb l'exemple anterior, la instrucció `m=(*ap) + 1;` fa que `m` contingui el valor de la variable apuntada per `ap`, és a dir, `n`, incrementada en una unitat. Noteu que el parèntesi no és necessari (consulteu la taula de precedències dels operadors de C).

En C disposem també d'una aritmètica d'apuntadors. Per veure com funciona, considerem els apuntadors `ap` i `aq` que apunten cap a variables enteres del tipus `short` (recordem que aquestes variables ocupen 2 bytes) i sigui `v[10]` un vector del mateix tipus. Si fem `ap = &v[0];` `ap` apunta al primer element de `v`. Si ara fem `aq=ap+1;` tenim que `aq` apunta al següent element (`v[1]`) del vector `v`, i no al segon byte del primer element. Si fem `aq=ap+5` tindrem que `aq` conté l'adreça de `v[5]` i així succesivament. Amb el mateix esperit podem usar els operadors `-`, `++`, `--`, `+=` i `-=`.

De fet, la relació entre apuntadors i vectors és més estreta del que hem dit aquí. Seguint amb l'exemple anterior, una altra manera de referenciar el segon element de `v` és fer `*(v+1)`. Això és degut al fet que el nom d'un vector s'identifica amb un apuntador constant (no el podem modificar) al primer element del vector. Així, l'assignació `ap = &v[0];` es pot escriure com `ap = v;`. D'altra banda, un cop feta aquesta assignació també és possible fer `ap[5]` per referir-se a `v[5]`.

A.6.2 Apuntadors i funcions

Una de les utilitats dels apuntadors és permetre que les funcions puguin modificar variables dels programes que les criden. La manera de fer-ho és passar l'adreça d'aquestes variables a les funcions i, dins d'aquestes, declarar els corresponents apuntadors. Veiem-ho amb un exemple:

```
   ⋮
main()
{
    int a,b,c;
    int f(int x,int *y);
      ⋮
```

```
    a=1;
    b=2;
    c=f(a,&b);
      ⋮
```

on la funció `f` és

```
int f(int x,int *y)
{
    int z;
    ++x;
    z=*y+x;
    *y*=2;
    return z;
}
```

Després de la línia `c=f(a,&b);` del programa principal, les variables `a`, `b` i `c` d'aquest valen 1, 4 i 4 respectivament. Es deixa com a exercici el funcionament de `f`.

Arribats aquí volem recalcar el "perill" dels apuntadors: fan el codi més complex alhora que (quan s'usen malament) produeixen errors difícils de detectar. És per això que es recomana fortament d'utilitzar-los només quan sigui estrictament necessari.

A.6.3 Apuntadors a funcions

Tot i que les funcions no són variables, és possible tenir apuntadors que les apuntin. La manera de declarar un apuntador a una funció és fer

`tipus (*nom)(`*paràmetres*`);`

Això declara `nom` com un apuntador cap a una funció que retorna `tipus`, on *paràmetres* és una llista (opcional) del tipus de paràmetres que rep la rutina. La direcció de la funció s'obté usant el nom d'aquesta sense parèntesis ni paràmetres. La manera de cridar-la és fer

`(*nom)(`*paràmetres*`)`

Els parèntesis són necessaris tant en la declaració com en la crida. Veiem-n'he un exemple

```
double trap(double a,double b,int n,double (*fun)(double))
{
       double h,s;
       int i;
       h = (b-a)/n;
       s = ((*fun)(a)+(*fun)(b))/2.e0;
       for (i=1;i<n;i++)
           s += (*fun)(a+i*h);
       return (s*h);
}
```

Aquesta funció aproxima una integral pel mètode dels trapezis. Els límits d'integració són `a` i `b`, `n` és el nombre de punts i `fun` és un apuntador a la funció a integrar. Una manera de cridar-la pot ser

```
main()
{
    double trap(double a,double b,int n,double (*fun)(double));
    double dist(double),a;
    int n;
    ⋮
    a=trap(0.e0,1.e0,n,dist);
    ⋮
}
```

i així calcularem l'aproximació de la integral entre 0 i 1 de la funció `dist` aplicant trapezis amb `n` punts.

A.6.4 Apuntadors a matrius

Comentem ara com podem passar matrius a funcions. Recordem primer, però, com es fa en el cas de vectors. Si, per exemple, volem escriure una funció que calculi el producte escalar de dos vectors de variables `double` podem fer

```
double escal(double u[],double v[],int n)
{
       double s;
       int i;
       s = u[0]*v[0];
       for (i=1; i<n; i++)
           s += u[i]*v[i];
       return (s);
}
```

on `u` i `v` són els vectors i `n` la seva dimensió.

Recordeu que hi ha una altra manera de declarar els vectors `u` i `v`. És fer-ho com apuntadors a variables `double`:

```
double escal(double *u,double *v,int n)
```

Ara farem una cosa similar amb matrius. Com exemple, escriurem una funció que calculi el producte de dues matrius quadrades d'elements `double` i posi el resultat en una tercera matriu del mateix tipus. El primer "inconvenient" que trobem és que el compilador necessita saber el nombre de columnes de les matrius (és a dir, no són vàlides declaracions com `double a[n][n]` dins de la funció) mentre que, d'altra banda, volem que la nostra funció sigui el més general possible i que, per tant, no tingui fixada la dimensió de la matriu. Tenim diverses maneres de resoldre això. Una d'elles consisteix a declarar un vector d'apuntadors, on fem apuntar cada apuntador a una de les files de la matriu. Veiem-ho.

```
main()
{
     void promaq(double **a,double **b,double **c,int n);
     double a[3][3],b[3][3],c[3][3],*aa[3],*ab[3],*ac[3];
     int i,j;
```

```
        for(i=0; i<3; i++)
        {
           aa[i] = &(a[i][0]);
           ab[i] = &(b[i][0]);
           ac[i] = &(c[i][0]);
        }
        ⋮
        promaq(aa,ab,ac,3);
        ⋮
}
void promaq(double **a,double **b,double **c,int n)
{
        int i,j,k;
        for (i=0; i<n; i++)
           for (j=0; j<n; j++)
           {
               c[i][j] = a[i][0]*b[0][j];
               for (k=1; k<n; k++)
                   c[i][j] += a[i][k]*b[k][j];
           }
        return ;
}
```

Comentem en primer lloc la funció `promaq`. La declaració `double **a` fa que `a` sigui un apuntador a un altre apuntador. De fet, en aquest cas, `a` és un apuntador a un vector d'apuntadors, cada un dels quals apunta a un vector del tipus `double` (que són les files de la matriu). La relació que hi ha entre vectors i apuntadors (que hem vist abans) fa que això sigui equivalent a declarar `double *a[]`. En tot cas, quan dins del cos de la funció escrivim `a[i]`, ens estem referint a l'apuntador de la i-èsima fila, i quan posem `a[i][j]` ens referim a l'element j-èsim d'aquesta mateixa fila.

Veiem ara alguns detalls del programa principal. Juntament amb les matrius hem declarat uns vectors d'apuntadors, i immediatament hem fet que apuntin a les files de les matrius. Aquests vectors són els que passem a la funció `promaq`.

Aquest mètode té l'inconvenient que, per a cada matriu, cal declarar i inicialitzar un vector d'apuntadors a les seves files. D'altra banda, aquest sistema és un dels més ràpids per accedir als elements de la matriu. Més endavant comentarem aquest fet en parlar d'altres mètodes per fer això mateix (de fet, el mètode per manejar matrius que recomanem és una petita modificació d'aquest, el veurem a la secció A.11).

A.7 Estructures

Les estructures són variables formades per agrupament d'altres variables, sota un mateix nom. Veiem-ho amb un exemple:

```
        struct punt {
           float x;
```

```
   float y;
};
```

Aquí definim una estructura anomenada `punt`, composta per dues variables `float` que anomenarem `x` i `y`. Volem recalcar que la definició anterior no declara cap variable, únicament defineix el tipus `punt`. Un cop fet això, podem declarar variables d'aquest nou tipus de la manera següent:

```
struct punt a,b,c;
```

Aquesta declaració diu que les variables `a`, `b` i `c` són estructures del tipus `punt`. Tant la definició del tipus `punt` com la declaració de les variables `a`, `b` i `c` es posen al principi del programa (o de la funció), igual que es fa amb les altres declaracions. Es poden fer les dues coses alhora posant

```
struct punt {
   float x;
   float y;
} a,b,c;
```

L'accés als membres de l'estructura es fa amb l'operador ".":

```
a.x=0.0;
a.y=1.0;
```

Això inicialitza la variable `a` amb el punt $(0, 1)$. Si volem assignar el contingut d'una variable de tipus `punt` a una altra del mateix tipus (per exemple, `b`) podem fer

```
b=a;
```

Finalment, volem recalcar que una estructura no és res més que una sèrie de variables (que poden ser de diferents tipus) agrupades sota un nom comú. El seu propòsit és ajudar a gestionar dades complicades, reduint el nombre de variables amb les quals treballem.

A.7.1 Typedef

La comanda `typedef` proporciona un sistema per donar noms nous a tipus de dades ja existents. Per exemple, la declaració

```
typedef int enter;
```

defineix `enter` com un sinònim de `int`. És a dir, després de fer això podrem posar

```
enter i,j;
```

per declarar `i` i `j` com a variables enteres. De fet, l'ús més habitual de `typedef` és redefinir els noms dels diversos tipus d'estructures. Per veure-ho, repetim l'exemple del tipus `punt`. La definició del tipus es pot posar com

```
typedef struct {
   float x;
   float y;
} punt;
```

i, a partir d'aquest moment, és possible declarar estructures del tipus `punt` fent només

```
punt a,b,c;
```

Aquest serà el sistema de declarar estructures que usarem d'ara endavant.

A.7.2 Vectors d'estructures

La manera de declarar un vector de estructures no es diferencia de com es declaren la resta de vectors. Si volem un vector de 10 components d'estructures `punt` definides abans, cal fer:

```
punt v[10];
```

on suposem que prèviament s'ha fet el corresponent `typedef` per definir el tipus `punt`. Llavors si, per exemple, volem posar tots els `punt`s a zero, podem fer

```
for(i=0; i<10; i++)
{
   v[i].x=0;
   v[i].y=0;
}
```

on es suposa que `i` ha estat declarada prèviament com una variable entera.

A.7.3 Apuntadors a estructures

Passades com arguments a les funcions, les estructures es comporten igual que la resta de les variables. En principi, es passen per valor (la qual cosa implica que les modificacions fetes damunt d'elles es perden en sortir de la funció). Si es volen conservar les modificacions que s'han fet cal passar-les per adreça. Veiem-ho amb un exemple. Suposem que `a` és una estructura de tipus `punt` (seguim amb l'exemple d'abans). Per passar `a` per adreça a una funció anomenada `h` i de tipus `void` fem

```
⋮
h(&a);
⋮
```

Veiem com funciona `h`. Per simplificar, suposem que aquesta funció canvia de signe el membre `x` de l'estructura que li passem com a argument. Llavors, el seu codi podria ser el següent:

```
void h(punt *p)
{
   (*p).x=-(*p).x;
   return;
}
```

Habitualment, però, no s'usa aquesta notació, sinó que es posa

```
void h(punt *p)
{
   p->x=-p->x;
   return;
}
```

on la notació `p->x` és una abreviació de `(*p).x`.

A.8 El preprocessador de C

El C disposa d'un preprocessador que permet usar instruccions que, tot i no formar part del propi llenguatge, ens faciliten la feina. Aquí només veurem un petit nombre d'aquestes instruccions, que es caracteritzen per començar amb el caràcter '`#`'.

#define

Permet definir un identificador com una expressió, que s'anirà substituint en tot el programa. Com a conveni, utilitzarem lletres majúscules per a aquests identificadors i així podrem diferenciar-los de les variables del programa. Per exemple, si fem `#define NMAX 20`, el preprocessador substituirà qualsevol aparició de '`NMAX`' per '`20`' abans que es compili el programa (òbviament aquesta substitució no es farà dins de les tires de caràcters). A aquest tipus de definició l'anomenarem macrodefinició o macro. Les macros poden acceptar paràmetres. Si per exemple, fem

```
#define MAX(A,B) ((A) > (B) ? (A) : (B))
```

al començament del fitxer on tenim el programa, qualsevol expressió del tipus `m = MAX(x+1,y);` serà substituïda per

```
m = ((x+1) > (y) ?  (x+1) :  (y));.
```

Aquesta tècnica permet donar una altra manera de passar matrius a funcions. Recordem que, en C, les matrius es guarden per files. Això vol dir que si declarem `double a[10][10]` tindrem que les files estan posades una darrere l'altra dins la memòria de l'ordinador, com si es tractés d'un vector de 100 components. Veiem ara com podem usar aquesta informació per construir una funció que transposi una matriu.

```
#define a(A,B) a[(A)*n+(B)]
void transp(double *a,int n)
{
     double x;
     int i,j;
     for (i=0; i<n-1; i++)
         for (j=i+1; j<n; j++)
         {
             x = a(i,j);
             a(i,j) =a(j,i);
             a(j,i) = x;
         }
     return ;
}
```

Una manera de cridar aquesta funció pot ser `transp(c,3);`, on `c` s'ha declarat com `double c[3][3]`. Veiem-ne ara el funcionament. El paràmetre `a` és un apuntador al primer element de la matriu, i `n` és la seva dimensió. Com que sabem que `a[0]` correspon a l'element `[0][0]`, `a[1]` a `[0][1]`, ..., `a[n]` a `[1][0]` i així succesivament, tenim que la correspondència entre els elements de la matriu original i del vector `a` es pot representar per `[i][j]` $\mapsto$ `i*n+j`. D'aquesta manera podem definir una macro que ens permet fer aquesta relació més còmodament:

```
#define a(A,B) a[(A)*n+(B)].
```

La presència de parèntesis no és necessària en aquest exemple concret, però ho pot ser en d'altres. Si temin una línia com `a(i+1,j)=0.e0;` i no posem parèntesis en la macro, el preprocessador la convertirà en `a[i+1*n+j]`, amb la qual cosa el programa no anirà bé i tindrem un error molt difícil de localitzar. Per aquest motiu és recomanable tenir l'hàbit de posar els corresponents parèntesis a les macros.

#undef

Permet "esborrar" macros, i ens servirà perquè aquestes només estiguin definides en el tros de programa que les necessita.

#include

Ha d'anar seguida del nom d'un fitxer entre cometes (`" "`) o entre angles (`< >`). Fa que s'inclogui aquest fitxer dins del programa a compilar en el lloc de la línia `#include`. Si el fitxer el posem entre cometes, per exemple `#include "meu.h"`, el preprocessador el buscarà al directori en el qual estem treballant. Si el posem entre angles, com `#include <stdio.h>`, el buscarà dins d'un directori específic del compilador. Els fitxers d'aquest directori contenen les declaracions de les funcions que són subministrades amb el compilador.

Si, per exemple, tenim un programa que utilitza la funció sinus, cal que aquesta estigui declarada com a `double`, perquè si no posem cap declaració el compilador entendrà que aquesta funció retorna valors enters. Una manera ràpida de declarar totes les funcions matemàtiques és posar la sentència `#include <math.h>` al principi de l'arxiu en què tenim el programa.

A.9 Entrada i sortida

El llenguatge C no disposa d'instruccions per fer operacions d'entrada i sortida. En el seu lloc tenim una sèrie de funcions que vénen amb el compilador i que s'encarreguen de fer aquesta tasca. Aquí en veurem només algunes. Primerament cal dir que quasi totes les que veurem tenen un paràmetre comú: una tira de caràcters que diu a la funció amb quin format ha de llegir o escriure les dades. No detallarem totes les posibilitats que ofereix aquest format, sinó solament les que usarem. Cal dir també que la seva declaració és dins del fitxer `stdio.h` i que, per tant, només cal posar `#include <stdio.h>` al principi del programa i d'aquesta manera es tenen totes declarades.

puts

Serveix per escriure tires de caràcters a la pantalla. La manera d'usar aquesta funció és posar `puts("`*caràcters*`")`.

printf

Serveix per escriure a la pantalla. La seva sintaxi és `printf("`*caràcters*`",` *paràmetres*`)`. La tira `"`*caràcters*`"` indica com s'han d'escriure les dades. Pot contenir caràcters "normals", que s'escriuen tal qual a la pantalla (aquí incloem també caràcters com `\n`, que provoquen un salt de línia), i formats per escriure els *paràmetres*, els quals comencen per `%`. Després del `%` podem

posar una sèrie de dígits per indicar quants espais reservem per a la dada, i hem d'acabar aquesta especificació amb un caràcter de conversió. Els caràcters de conversió que utilitzarem són:

- `d`: decimal (int).
- `u`: decimal sense signe (unsigned int).
- `c`: caràcter.
- `s`: tira (vector) de caràcters.
- `e`: `double` (notació científica). Per defecte, escriu 6 decimals.
- `f`: `double` (notació decimal, part entera . part decimal). Per defecte, escriu 6 decimals.

Hi ha dos modificadors que serveixen per indicar que un nombre enter és `short` o `long`. Són `h` i `l` respectivament. Així, per escriure una variable `long int` usarem com a caràcter de conversió `ld`.

Observeu que no hem donat cap caràcter de conversió per a les variables `float`. La raó és que les variables `float` s'envien a `printf` convertides a `double` i, per tant, un cop dins aquesta funció, són tractades com a tals. En aquest cas, és responsabilitat del programador no usar un format que impliqui l'escriptura de més de 6 dígits significatius.

Veiem-ne uns exemples: suposem que `a` és una variable `float` i que `x` és `double`. Llavors, `printf("a = %e\n",a);` escriu `a` amb 6 decimals i notació científica, mentre que `printf("x = %24.16e\n",x);` escriu `x` amb 16 decimals i notació científica. El `24` és el nombre total d'espais que s'ocuparan per escriure el nombre, dels quals 16 seran per als decimals. La diferència $24 - 16 = 8$ són els espais emprats per escriure la `e` de l'exponent, el punt decimal, els signes (tant de l'exponent com del nombre), i per posar espais en blanc davant del nombre, per separar-lo d'altres números que puguem estar escrivint.

scanf

Serveix per llegir de la consola. La seva sintaxi és `scanf("`*caràcters*`",` *paràmetres*`)`. La diferència que hi ha entre els paràmetres d'aquesta rutina i els de l'anterior és que aquí tots han de ser apuntadors (recordeu els comentaris fets anteriorment sobre la manera de passar els paràmetres a les funcions en C). Aquest fet ens obligarà a diferenciar els `float` dels `double`, amb el caràcter `l`. Si, per exemple, volem llegir una variable `double` i posar-la en la variable `x` del mateix tipus, cal fer `scanf("%le",&x);`.

fopen

Serveix per obrir arxius. La seva sintaxi és `fopen("`*nom*`", "`*modus*`")`, on *nom* és el nom del fitxer a obrir i *modus* és un dels següents:

- `r`: Obre un fitxer ja existent per a lectura.
- `w`: Crea un fitxer i l'obre per a l'escriptura. Si ja hi ha un fitxer amb aquest nom, l'esborra i el crea de nou.

(Hi ha més opcions que no comentem; podeu consultar [10] per als detalls). La funció retorna un apuntador del tipus `FILE`. Si es produeix algun error, es retorna l'apuntador `NULL`. Un exemple de la seva utilització el tenim tot seguit.

```
#include <stdio.h>
main()
{
     FILE *fl;
     .
     .
     .
     fl = fopen("prova.dad","r");
     if (fl == NULL)
         {
               puts("error a l'obrir prova.dad");
               exit(1);
         }
     .
     .
     .
}
```

Aquí s'obre el fitxer `prova.dad` per llegir-lo i es comprova que no hi ha hagut cap error (com ara que el fitxer no existís). La funció `exit` serveix per interrompre l'execució del programa i retornar al sistema operatiu.

fclose

Serveix per tancar un fitxer obert, i buidar el *buffer* associat dins del fitxer si aquest estava obert per a escriptura. Retorna un 0 si no hi ha cap error, i un valor diferent de 0 en cas contrari. La seva sintaxi és `fclose(`*ptr*`)`, on *ptr* és l'apuntador de tipus `FILE` que apunta a l'arxiu desitjat i que ha estat inicialitzat prèviament per la funció `fopen`.

fprintf

La seva sintaxi és `fprintf(`*ptr*`, "`*caràcters*`",` *paràmetres*`)`, on *ptr* és un apuntador `FILE` inicialitzat per la funció `fopen`, i els altres paràmetres fan el mateix paper que a la funció `printf`, amb la diferència que ara l'escriptura es fa dins del fitxer associat a l'apuntador `ptr`. Per exemple

```
#include <stdio.h>
main()
{
     FILE *res;
     double x[100],y[100];
     int i,n=100;
     .
     .
     .
     res = fopen("prova.res","w");
     if (res == NULL)
         {
               puts("error a l'obrir prova.res");
```

```
                exit(1);
            }
        for (i=0; i<n; i++)
            fprintf(res,"%d %24.16le %24.16le\n", i,x[i],y[i]);
        fclose(res);
        ⋮
}
```

Aquí obrim el fitxer `prova.res` per a escriptura, hi posem una taula de valors i finalment tanquem el fitxer.

fscanf

Serveix per llegir valors d'un arxiu i el seu funcionament és anàleg al de `fprintf`. Veiem-ne un exemple

```
#include <stdio.h>
main()
{
     FILE *dad;
     double x[20][20],a;
     int i,j,n=20;
     for (i=0; i<n; i++)
         for (j=0; j<n; j++)
             x[i][j] = 0.e0;
     dad = fopen("prova.dad","r");
     if (dad == NULL)
         {
             puts("error al obrir prova.dad");
             exit(1);
         }
     while (fscanf(dad,"%d %d %le",&i,&j,&a) != EOF)
         x[i][j] = a;
     ⋮
}
```

Només direm que la rutina `fscanf` retorna un `EOF` (End Of File) quan s'intenta llegir més enllà de la fi del fitxer. De fet, `EOF` és una constant entera definida per un `#define` dins el fitxer `stdio.h`.

A.9.1 Tires de caràcters

Val la pena fer uns petits comentaris sobre les tires de caràcters. En C, una tira de caràcters és un vector de tipus `char`, que té una component més que el nombre de caràcters que hi volem guardar. En aquesta component extra hi ha un codi de "fi de tira de caràcters" (el podem anomenar codi zero) que naturalment va al final. És important recordar aquest fet quan es dimensionen els vectors corresponents, per evitar tenir errors que després costen molt de

trobar. Així, per llegir la paraula HOLA del teclat cal dimensionar un vector `char` de com a mínim 5 components.

```
char s[5];
scanf("%s",s);
⋮
```

La rutina `scanf` afegeix automaticament el codi zero al final de la tira. Si omplim nosaltres mateixos el vector en un programa o en una rutina ho haurem de fer així:

```
char s[5];
s[0]='h';
s[1]='o';
s[2]='l';
s[3]='a';
s[4]='\0';
⋮
```

Aquest codi 0 és utilitzat per les rutines de sortida[3] (`puts`, `scanf`, etc) per saber fins on s'ha d'imprimir.

Un error molt comú relacionat amb les tires de caràcters és el següent: suposem que tenim un programa que llegeix i/o escriu en un fitxer, però volem donar el nom d'aquest fitxer a través del teclat. En un PC amb DOS, la longitud màxima que pot tenir el nom d'un fitxer és de 12 caràcters (8 del nom + 3 de la extensió + el punt), per tant es suficient amb dimensionar els vectors `char` amb 13 components. Fins aquí tot correcte. El problema ve quan, un cop el programa està fet i funciona, alguna persona dóna per exemple el nom de fitxer següent:

```
a:\practica\dades\prob3.dad
```

Com que la rutina `scanf` no comprova si la tira que entrem cap o no dins del vector que tenim (de fet, no pot fer-ho, ja que no coneix la longitud del vector!), resulta que part dels caràcters s'escriuen "desbordant" el vector, amb la qual cosa estem sobreescribint posicions de memòria, i podem així alterar els valors d'altres variables del programa, destruir part del propi codi del programa, etc. (excepcionalment pot ser que no passi res pel fet que aquestes posicions de memória estiguin lliures). Noteu que l'error corresponent es produirà més endavant en l'execució del programa i, aparentment, no tindrà res a veure amb el nom del fitxer. Fins i tot, és possible que aquest programa funcioni bé en un ordinador i malament en un altre: depenent de com es guardin les variables dins la memòria, el "desbordament" del vector de caràcters pot ser que alteri posicions de memòria essencials (el programa no funcionarà), o bé que aquestes posicions sobreescrites estiguin lliures (el programa funcionarà correctament). Com es pot imaginar, aquests errors poden costar molt de trobar.

Una manera simple de protegir-se contra aquests errors consisteix a declarar aquests vectors molt llargs (per exemple, de 80 caràcters) i, quan es demana el nom del fitxer, escriure un avís del tipus "el nom del fitxer no pot tenir més de xx caràcters".

No volem acabar sense esmentar un altre tipus d'error que es pot cometre quan es treballa amb tires de caràcters. Suposem que un programa ha d'accedir al fitxer `config.cfg` del directori `c:\soft`. La manera d'obrir aquest fitxer des del programa és fer

[3] De fet, totes les rutines del sistema que manegen caràcters usen aquest codi per saber on acaben les tires.

```
fl=fopen("c:\\soft\\config.cfg","r");
```

Posar `"c:\soft\config.cfg"` com a nom del fitxer que s'ha d'obrir és incorrecte, perquè el "\" dins d'una tira de caràcters té un significat especial. Recordeu, per exemple, que "`\n`" indica salt de línia. Quan es vol indicar "\" dins d'una tira de caràcters, cal posar "\\".

A.10 Funcions matemàtiques

Com hem dit abans, les funcions matemàtiques estan declarades al fitxer `math.h`, el qual hem de posar al principi de l'arxiu que les utilizi amb un `#include`. Les funcions que més farem servir són les funcions trigonomètriques (`sin`, `cos` i `tan`), on l'argument s'ha de donar en radians, la funció exponencial (`exp`), el logaritme neperià (`log`), l'arrel quadrada (`sqrt`) i el valor absolut (`fabs`). Totes aquestes funcions retornen valors de tipus `double`.

A.11 Assignació dinàmica de memòria

En C és posible reservar espai per a noves variables en temps d'execució del programa. Veurem només 3 funcions declarades al fitxer `stdlib.h`.

malloc

La cridarem posant `malloc(`*num*`)`, on *num* és el nombre de bytes que volem reservar. La funció retorna un apuntador de tipus `void` cap a la memòria demanada. Si no troba prou memòria, retorna l'apuntador `NULL`. Veiem-ne un exemple

```
#include <stdio.h>
#include <stdlib.h>
main()
{
     double *x;
     int i,n;
     FILE *d;
     d = fopen("prova.dad","r");
     if (d == NULL)
         {
              puts("error al obrir prova.dad");
              exit(1);
         }
     fscanf(d,"%d",&n);
     x = (double*)malloc(n*sizeof(double));
     if (x == NULL)
         {
              puts("no hi ha espai per llegir la taula");
              exit(1);
         }
     for(i=0; i<n; i++)
         fscanf(d,"%le",&(x[i]));
```

```
        ⋮
}
```

En primer lloc, declarem uns apuntadors a variables `double` i obrim un fitxer per a llegir-lo. El primer valor del fitxer és un enter que indica quants valors hi ha (suposem que el fitxer conté una llista de valors `double`). Llavors fem una crida a la rutina `malloc` per reservar espai per a aquests valors, on `sizeof` és un operador que retorna el nombre de bytes que ocupa una variable. En aquest cas, `n*sizeof(double)` és el nombre de bytes que necessitem per guardar `n` valors `double`. Seguidament convertim l'apuntador que ens torna `malloc` a tipus `double*` (apuntador a `double`) utilitzant un *cast*, i l'assignem a `x`, comprovem que aquest apuntador no és `NULL` i llegim la taula.

calloc

El seu ús és molt similar al de `malloc`. La seva forma és `calloc(`*num*`, `*mida*`)`, on *num* és el nombre de zones (consecutives dins la memòria de l'ordinador) de mida *mida* que volem reservar. La diferència entre `malloc` i `calloc` (tret que la primera té només un argument i la segona dos) és en què `calloc` posa a zero tota la memòria que reserva, mentre que `malloc` no ho fa.

free

S'usa posant `free(`*ptr*`)` on *ptr* és un apuntador a una zona de memòria reservada per una crida a `malloc` o `calloc`. Allibera la memòria reservada de manera que pot utilitzar-se de nou per a altres assignacions. No retorna cap valor.

Un dels usos típics d'aquestes funcions és reservar espai de treball (típicament un vector o matriu) dins de les funcions en les quals es necessiten. En aquest cas, és molt important recordar de fer el corresponent `free` abans de sortir de la funció. Si no el fem, aquesta memòria continuarà estant reservada durant la resta del programa. Si, a més, cridem a aquesta funció moltes vegades, llavors és possible que ens quedem sense memòria, ja que a cada crida anem reservant un nou espai de treball, però mai n'alliberem cap.

A.12 Més sobre matrius

A la secció A.6.4 hem vist un mètode per passar matrius a una funció. Recordem que, en aquell exemple, les dimensions de les matrius dins la funció no estaven prefixades, però sí que ho estaven dins el programa principal. És a dir, si es vol executar el programa per a matrius d'unes altres dimensions, cal tocar les corresponents dimensions dins del programa principal, compilar i linkar de nou. Ara veurem un sistema per fer que les matrius es "dimensionin" a `n`, on `n` és un valor que es llegirà (per exemple per mitja del teclat) quan s'executi el programa. Veiem-ho amb un exemple:

```
#include <stdlib.h>
main()
{
    double **a;
```

```
    int i,n;
    ⋮
    ⋮ /* aqui es calcula o es llegeix el valor de n */
    ⋮
    a=(double**)malloc(n*sizeof(double*));
    if (a==NULL) {puts("falta memoria"); exit(1);}
    for(i=0; i<n; i++)
    {
       a[i]=(double*)calloc(n,sizeof(double));
       if (a[i]==NULL) {puts("falta memoria"); exit(1);}
    }
    ⋮
}
```

El funcionament d'aquest tros de programa és el següent: en primer lloc es declara `a` com un apuntador d'apuntadors. A continuació s'obté la dimensió `n` de la matriu que es vol utilitzar (per exemple, llegint-la per mitja del teclat o d'un fitxer). Després, la línia del `malloc` reserva espai per a `n` apuntadors de tipus `double`. Cada un d'aquests apuntadors ens assenyalarà una fila de la matriu (recordem que un apuntador i un vector són pràcticament el mateix). Per acabar, reservem espai per a cada una de les files: a l'apuntador de la fila `i`, `a[i]`, li fem un `calloc` de `n` components. Un cop acabat aquest bucle, ja tenim la matriu a punt per treballar: si volem accedir a l'element `(i,j)`, només cal fer `a[i][j]`. Repassem com funciona això: `a` és l'apuntador del vector que conté els apuntadors a les files de la matriu, `a[i]` és l'element `i`-èsim d'aquest vector i, per tant, l'apuntador de la fila `i`. Llavors, `a[i][j]` és l'element `j` d'aquesta fila, és a dir, l'element `(i,j)` de la matriu.

Quan es volen passar aquestes matrius a funcions, senzillament s'envien com un parametre més. Per exemple, per enviar la matriu `a` de l'exemple anterior a una funció anomenada `fm`, podem fer `fm(a,n)`, on declararem la matriu que arriba (diguem-li de nou `a`) com `double **a`. Per usar-la ho fem com en el programa principal, és a dir, accedim a l'element `(i,j)` posant `a[i][j]`.

Es recomana, com a exercici, comparar aquest mètode amb els altres vistos abans, especialment amb el de la secció A.6.4.

Finalment, volem recalcar la manera d'alliberar la memòria reservada:

```
for(i=0; i<n; i++)
{
   free(a[i]);
}
free(a);
```

A.12.1 Rang dels subíndexs

És possible alterar el rang on varien els subíndexs d'un vector o matriu. Per exemple, si volem un vector `v` de `n` components que comenci per 1 i acabi per `n` (en lloc de 0 i `n-1` com és habitual) podem fer

```
int *v,n;
⋮            /* n pren un cert valor */
v=(int*)malloc(n*sizeof(int));
if (v == NULL) {puts("falta memoria"); exit(1);}
--v;
⋮
```

Després de fer `--v`, la primera component del vector ha passat de ser la `[0]` a ser la `[1]`, la `[1]` ha passat a ser la `[2]`, etc. Això fa que ara la primera component del vector sigui `v[1]` i l'última `v[n]`. Si per alguna raó es volgués que la primera component fos la `n` i l'última la `2*n-1`, només caldria substituir la instrucció `--v` per `v-=n`.

Aquest mateix truc es por aplicar a les matrius. Per exemple, per crear una matriu amb subíndexs entre 1 i `n` (ambdós inclosos), podem modificar el programa anterior de la manera següent:

```
a=(double**)malloc(n*sizeof(double*));
if (a==NULL) {puts("falta memoria"); exit(1);}
--a;
for(i=1; i<=n; i++)
{
   a[i]=(double*)calloc(n,sizeof(double));
   if (a[i]==NULL) {puts("falta memoria"); exit(1);}
   --a[i];
}
```

Similarment, es poden aconseguir altres rangs per als subíndexs.

A.13 Temps de vida i visibilitat de les variables

Anomenarem bloc el conjunt d'instruccions que es troben entre { } en qualsevol programa i/o funció.[4] Aprofitem per recordar que les declaracions s'han de posar al principi de cada bloc en què les volem usar. Quan nosaltres declarem una variable (com per exemple `int k`) al principi d'un bloc, estem creant un espai de memòria on es guardarà aquesta variable mentre duri l'execució del seu bloc (això és el que s'anomena temps de vida de les variables). Per tant, les variables del `main` existiran durant tota la execució del programa, però les variables de les funcions només ho faran quan s'estigui executant la corresponent funció. La implicació més important d'aquest fet és que, si una funció es crida dues vegades, els valors de les seves variables internes a l'entrada de la segona crida no ténen res a veure amb els valors de les variables internes just en sortir de la primera crida. Una altra observació a fer és que les declaracions només són vàlides dins del bloc on són fetes. A això fa referència el concepte de visibilitat: la regió de visibilitat (que és la regió on es coneix el valor de la variable i, per tant, on es pot usar) és el seu corresponent bloc.

[4] Per simplificar la lectura (i la comprensió) podeu pensar que els únics blocs que tenim en compte són els que defineixen el cos del programa principal o el cos de les funcions. De fet, però, tot el que es diu en el text és igualment vàlid per a qualsevol altre tipus de bloc (el cos d'un `for`, per exemple). Per a més detalls consulteu un manual de C, com per exemple [10].

El que acabem de dir és el que assumeix C per defecte. Hi ha maneres d'alterar aquestes assumpcions. Veiem-ne unes quantes.

Temps de vida

Posant davant de la declaració la paraula `static` (per exemple, fent `static int k`), estendrem el temps de vida de la corresponent variable al temps d'execució del programa. Amb aquesta tècnica podem aconseguir, per exemple, que una funció "recordi" el valor de les seves variables internes entre dues crides successives.

Visibilitat

Aquest concepte és una mica més complex i no parlarem de totes les possibilitats que presenta. De fet, ens centrarem en dos punts.

Si traiem la declaració de la variable fora de qualsevol bloc (és a dir, que no hi hagi cap bloc que la contingui), aquesta variable esdevé estàtica i la seva visibilitat s'estén a tot el fitxer (compte: hem posat fitxer, no programa) on es troba. Per exemple

```
int k;
main()
{
      ⋮
```

És possible estendre la visibilitat d'aquesta variable als altres fitxers dels quals es compon el programa. No donem, però, els detalls. Si esteu interessats, busqueu en un manual (per exemple [10]) la paraula clau `extern` i en trobareu més.

Una altra possibilitat és fer el mateix que abans, però afegint la paraula `static` davant la declaració, d'aquesta manera:

```
static int k;
main()
{
      ⋮
```

En principi, això fa el mateix que el cas anterior (el cas en què no hi havia l'`static`), però amb una petita diferència: ara és del tot impossible conèixer aquesta variable des de fora del fitxer on hem posat aquesta declaració (tot i que, com abans, continua coneguda dins del seu fitxer). Això serveix per evitar que, per accident, una variable d'un altre fitxer que hagi de ser diferent d'aquesta es converteixi en la mateixa variable. Aquest fet improbable en programes petits, pot provocar un error molt difícil de localitzar en programes grans. La conclusió és doncs que, si una variable està declarada fora de qualsevol bloc i no ha de ser coneguda en cap altre fitxer, li posarem un `static`.

Noteu així que la paraula `static` té dos significats diferents, segons sigui dins d'un bloc o no.

Apèndix B Funcions bàsiques de dibuix per a llenguatge C

Malauradament les funcions de dibuix que tenen incorporades els diferents compiladors de C no són estàndards, sinó que reben noms i s'usen de manera difererent segons les diferents marques que hi ha en el mercat. Per això, un programa que en una certa màquina funciona en una altra potser no, o bé en canviar el compilador del PC ens trobem que programes que abans funcionaven bé quant a dibuix ara no ho fan.

Per aquest motiu és bo portar un cert ordre en els programes que dibuixen. I com que no hi ha homogeneïtat entre les diferents funcions gràfiques dels compiladors, cal que "un mateix" la faci a fi que canviar de màquina o de compilador no ens suposi reescriure tot el programa de nou.

La solució més senzilla és crear un mòdul propi (*driver*) amb les funcions de dibuix més usuals, les quals "batejarem" amb el nom que ens sembli més adient. Sempre que haguem d'usar una funció de dibuix, cridarem una funció usant el nom que tingui en el nostre mòdul, i mai més farem una crida directa a la del compilador que no és estàndard. Aleshores en canviar de màquina o de compilador només caldrà retocar les funcions del nostre mòdul i tots els programes que tenim continuaran essent vàlids senzillament recompilant-los de nou.

D'aquesta manera podem tenir a la llarga diferents mòduls per a diferentes màquines o compiladors, però els nostres programes sempre usen les mateixes crides amb els mateixos paràmetres. Usar un cert programa amb una certa màquina o un cert compilador només implica muntar el programa amb el mòdul corresponent a la màquina i el compilador que s'usa.

Les funcions més bàsiques de dibuix que ha de tenir el nostre mòdul de dibuix poden ser:

- Una funció que inicialitzi el mode gràfic. Tasca habitual abans d'usar funcions gràfiques.
- Una funció que ens defineixi la finestra de treball. Ja que de vegades als punts de la pantalla (anonenats píxels) s'hi accedeix de maneres relativament complicades.
- Una funció que situï el "llapis de dibuix" en un cert punt de la finestra, però que no dibuixi res.

- Una funció que dibuixi un punt en la nostra finestra.

- Una funció que ens serveixi per dibuixar línies rectes.

- Una funció que tanqui el mode gràfic i torni a l'alfanumèric, que és el modus habitual de treballar en pantalla quan no es dibuixa.

- D'altres funcions opcionals potser relacionades amb els colors del dibuix, etc.

Cal distingir bé quines són les funcions que han d'estar dins el mòdul i quines no. En principi, el mòdul no ha de ser gaire gran ja que cada nova màquina o compilador ens suposa retocar-lo. És per això que aconsellem que contingui *el que sigui imprescindible i independent*. Qualsevol altra funció que es pugui escriure en termes de les que ja tenim en el mòdul, probablement ja és convenient posar-la en un altre fitxer que contingui d'altres funcions de dibuix i que ja no caldrà retocar en cas de canvi. Per exemple una funció que dibuixi un quadrat o un cercle es pot posar en termes de les que situen el llapis i dibuixen punts o línies rectes. Per tant si la construïm la posarem fora del mòdul.

A partir del nostre mòdul bàsic de dibuix es poden escriure altres funcions, a partir d'aquestes darreres se'n poden fer d'altres de nivell superior i així successivament podem tenir una bona "biblioteca" de funcions gràfiques, fins al punt que, si volem, podem gestionar un sistema de finestres. En cas de canvi només ens caldrà tocar el mòdul més intern.

B.1 Exemple d'un mòdul de dibuix bàsic

Pel mòdul bàsic de rutines de dibuix que presentem, hem escollit les següents funcions, a les quals tots els nostres programes es refereixen amb aquests noms:

- `inigraf`. Inicialitza els gràfics i és obligatòria la seva crida abans d'usar qualsevol altre funció de dibuix.

- `finestra`. Defineix la pantalla com la finestra de treball amb els límits de les coordenades reals que volem usar per dibuixar. També és obligatori haver definit la finestra de dibuix abans de començar a dibuixar res.

- `capa`. Situa el "llapis de dibuix" en un cert punt de la finestra sense dibuixar res.

- `lina`. Dibuixa una línia recta des del punt de la finestra on és el llapis fins el punt que se li indica.

- `pospun`. Dibuixa un punt (encén un píxel) a la posició que se li diu.

- `tancagraf`. Tanca el mode gràfic, esborra el dibuix i torna al mode alfanumèric habitual. Si es vol tornar a dibuixar de nou caldrà tornar a fer una crida de la funció `inigraf`.

- `colfons`. Canvia el color del fons de la pantalla.

- `prencolor`. Retorna el valor enter corresponent al color del llapis de dibuix que usem.

Si un determinat compilador no tingués la possibilitat de fer alguna de les coses considerades[1], com podria ser el canvi de colors, el nostre mòdul contindria la funció corresponent però la deixariem en blanc, o bé fent la tasca més aproximada a allò que es requereix.

El mòdul que presentem a continuació està dissenyat per al compilador `TurboC v2.0` de la casa `Borland` per a PC (vegeu [22]), que és el que habitualment fem servir en la docència a l'Escola d'Enginyers Industrials de la Universitat Politècnica de Catalunya.

Aquest mòdul, continuant amb l'esperit indicat al principi d'aquest apèndix, elimina l'us de l'estructura del tipus `viewporttype`, que potser no es troba en d'altres compiladors. La funció `inigraf` condensa feines d'altres funcions del TurboC com són `detectgraph`, `initgraph`, `getviewsettings` i `clearviewport`, per la qual cosa a l'usuari no li cal ni tant sols saber què fan. Les funcions de moure el llapis i dibuixar substitueixen essencialment les funcions `moveto`, `putpixel` i `lineto` del `TurboC`. Mentre que les funcions que tracten els colors i la de tancar el mode gràfic del `TurboC`: `setcolor`, `setbkcolor`, `getcolor` i `closegraph`, senzillament es pot dir que les hem anomenat en termes del "nostre estàndard", posant els paràmetres que ens semblen més convenients en general.

El mòdul comença amb l'include de les funcions gràfiques del `TurboC` i la declaració de la funció `punpix`, que és una funció d'ús intern específic d'aquest mòdul i no s'ha d'emprar mai fora, ja que no és pròpia de dibuix i un altre mòdul pot tenir-la o no. Serveix per a convertir les coordenades (x, y) d'un punt de la finestra a les coordenades del corresponent píxel de pantalla, i es troba relacionada al final de l'arxiu. Seguidament es declara la variable interna (estructura) `marge`, que guarda la finestra que li indiquem via la funció `finestra`, i l'estructura `pantalla`, que és d'un tipus intern al `TurboC` i que guarda els marges de la pantalla en termes de píxels. Ambdues estructures, `marge` i `pantalla`, són transparents a l'usuari i es gestionen interiorment dins el mòdul. Mai no s'han d'emprar fora d'ell i per això ja els hem posat, com a la rutina `punpix`, el qualificador `static`. Finalment donem les diferents funcions comentades[2].

Mòdul de funcions de dibuix pel TurboC v2.0

```
#include<graphics.h>
static void punpix(double x, double y, int *i, int *j);

typedef struct {
   double xmin;
   double xmax;
   double ymin;
   double ymax ;} limitsf;

static limitsf marge;
static struct viewporttype pantalla;

void inigraf()
 /* Inicialitza el mode grafic i es obligatoria la crida
    abans d'usar qualsevol altre funcio de dibuix.
```

[1] Cosa difícil ja que totes són molt habituals.

[2] Com és habitual no posem accents en els comentaris de les funcions.

```
    A la funcio interior initgraph que te, se li ha d'indicar
    el directori on estan guardats els arxius *.BGI del TURBOC.
    En el nostre cas es a c:\tc\grafics. */
{
 int a,b;
 detectgraph(&a,&b);
 initgraph(&a,&b,"c:\\tc\\grafics");
 getviewsettings(&pantalla);
 clearviewport();
}

void finestra(double XMIN,double XMAX,double YMIN,double YMAX)
/* Defineix la finestra amb les coordenades reals que volem usar
   per dibuixar. Aixi l'angle inferior esquerre de la pantalla
   passa a ser el punt de coordenades (XMIN,YMIN), i l'angle
   superior dret passa a ser el punt (XMAX,YMAX). */
{
  marge.xmin=XMIN;
  marge.xmax=XMAX;
  marge.ymin=YMIN;
  marge.ymax=YMAX;
}

void capa(double x, double y)
/* Situa el cursor en la posicio (x,y) de la finestra. */
{
 int i,j;
 punpix(x,y,&i,&j);
 moveto(i,j);
}

void lina(double x, double y, int col)
/* Dibuixa una linia des de la posicio actual del cursor fins
   a la posicio (x,y) de la finestra usant el color col.
   El cursor queda finalment situat a la posicio (x,y). */
{
 int i,j;
 punpix(x,y,&i,&j);
 setcolor(col);
 lineto(i,j);
}

void pospun(double x,double y, int col)
/* Encen el pixel (dibuixa el punt) corresponent a la posicio
   (x,y) de la finestra usant el color col.
   El cursor queda finalment situat a la posicio (x,y). */
{
```

```
 int i,j;
 punpix(x,y,&i,&j);
 putpixel(i,j,col);
 moveto(i,j); /* Cal usar-la ja que putpixel no mou el llapis */
}

void tancagraf()
/* Tanca el mode grafic i torna a l'alfanumeric.
   El dibuix queda esborrat de la pantalla. */
{
 closegraph();
}

void colfons(int col)
/* Canvia el color del fons de la pantalla. */
{
 setbkcolor(col);
}

int prencolor()
 /* Retorna el valor enter corresponent al color amb
    el qual estem treballant. */
{
 int col;
 col=getcolor();
 return col;
}

static void punpix(double x, double y, int *i, int *j)
 /* Converteix les coordenades (x,y) d'un punt de la finestra en
    les coordenades del corresponent pixel de la pantalla. */
{
  *i=(x-(marge.xmin))*(pantalla.right)/((marge.xmax)-(marge.xmin));
  *j=((marge.ymax-y))*(pantalla.bottom)/((marge.ymax)-(marge.ymin));
}
```

Aquest mòdul ha de funcionar amb el compilador indicat anteriorment en un PC, sempre que l'opció "Graphics library" dins "Options-Linker" estigui "on" i que el paràmetre de la rutina `initgraph` dins de `inigraf` que conté "`c:\\tc\\grafics`[3]" indiqui el directori on són els "`*.BGI`" de la nostra instal·lació de `TurboC`.

A més del mòdul anterior, que el guardarem per exemple a l'arxiu `grbastc.c`, podem fer un arxiu "d'include" que contingui les declaracions i així no haver d'escriure-les en els programes que utilitzin aquestes funcions bàsiques de dibuix del nostre mòdul. A aquest nou arxiu li podem dir `grafbas.h` (notem que és independent de la màquina i del compilador) i la llista de declaracions que presenta en el cas del mòdul que hem dissenyat és la següent:

[3] Noteu que el doble \ és necessari dins una tira de caràcters.

Arxiu d'include grafbas.h

```
void inigraf();
void finestra(double XMIN,double XMAX,double YMIN,double YMAX);
void capa(double x, double y);
void lina(double x, double y, int col);
void pospun(double x,double y, int col);
void tancagraf();
void colfons(int col);
int prencolor();
```

Finalment presentem un esbós de l'estructura d'un programa principal que fa ús de les funcions gràfiques del nostre mòdul. Com es pot veure, la llista d'aquest programa és totalment independent del compilador que usem. Els comentaris els posem seguint el conveni del llenguatje C.

```
        /* Includes necessaris del programa */
#include<.....>
        /* Include de les funcions del nostre modul grafic */
#include"grafbas.h"

main()
 {
        /* Declaracions de funcions fora dels includes i que
           s'usin en el programa.
           Declaracions de variables que usa el programa i que
           no apareixen en la part grafica. Aqui incloem les
           variables double com a, b, x i y que se suposa es van
           calculant dins del programa i que usarem
           genericament per denotar un punt */

        /* Declarem seguidament les variables necessaries per
           al dibuix */
  double xmin,xmax,ymin,ymax;
  int col;

        /* Fixem el rang de la nostra finestra de dibuix per
           exemple de -1 a 2 en les abcises i de -3 a 4 en les
           ordenades */
  xmin=-1.e0;
  xmax=2.e0;
  ymin=-3.e0;
  ymax=4.e0;

        /* Inicialitzem els grafics i la finestra. Agafem el
           color d'escriure habitual de l'ordinador */
  inigraf();
```

```
 finestra(xmin,xmax,ymin,ymax);
 col=prencolor();

       /* Seguidament el programa podria fer calculs */
 .........
 .........

        /* Donem l'exemple d'us d'algunes funcions grafiques.
           Recordem que dibuixarem amb el color col fins que
           es canvii amb prencolor */
        /* Posem el cursor en el punt (0,1) */
 capa(0.e0,1.e0);

       /* Tirem linia recta des del punt on es el cursor fins
          al punt de coordenades (x,y) que suposem calculat */
 lina(x,y,col);

       /* El programa podria fer mes calculs */
 .........
 .........

      /* Encenem el pixel de la posicio (a,b) amb color col */
 pospun(a,b,col);

      /* Mes calculs i dibuixos */
 .........
 .........

     /* Finalment un cop vist el dibuix tanquem el grafic.
        Un getch() evita que desaparegui ``abans de ser vist'' */
 getch();
 tancagraf();

    /* El programa pot continuar fent calculs, presentant
       resultats o inicialitzant un grafic nou */
 .........
 .........
}
```