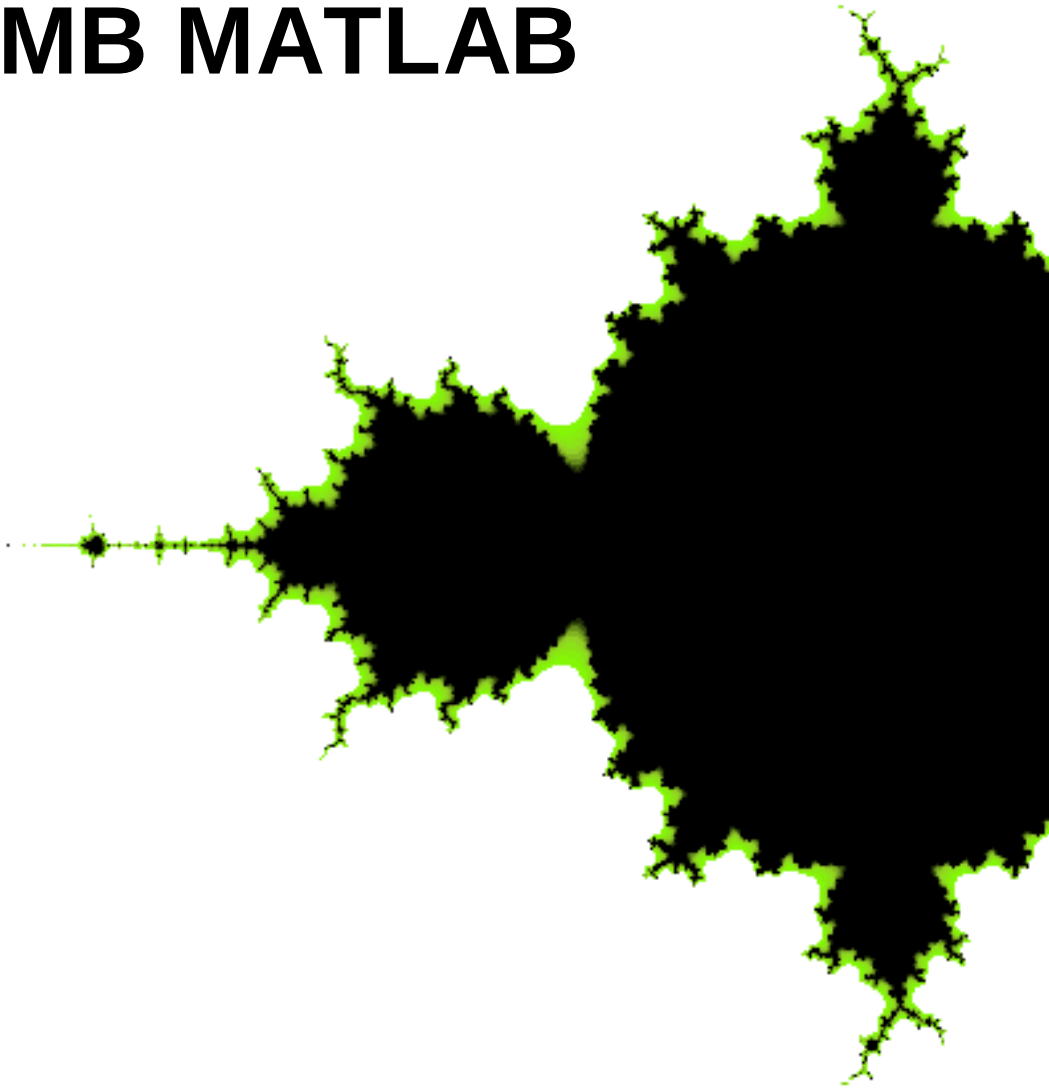
MÈTODES NUMÈRICS AMB MATLAB



Departament de Matemàtiques

Curs 2008/09

ÍNDEX

INTRODUCCIÓ

El treball de recerca és una fita important que necessitem superar per poder progressar en el nostre procés d'aprenentatge. Així doncs, quan se'ns presenta la opció de realitzar un treball com aquest no ens ho hem de prendre com a obligació, sinó com al més semblant a una ajuda per obrir camí i espavilar-nos. En aquest cas, el treball de recerca que ara mateix teniu a les mans, s'ha intentat marcar uns punts per assolir i uns aspectes a tractar. Podem trobar-hi dues columnes principals, la matemàtica i la informàtica. Definit el camp, només cal esbrinar com relacionar-ho.

Des de temps ancestrals, la matemàtica ha estat l'eina de moltes ciències, avui en dia continua sent-ho de les mateixes i algunes noves que han sorgit recentment. Un exemple clar el trobem en les pàgines següents a aquesta introducció, on s'han unit teoremes i mètodes matemàtics amb eines de programació. Però no us penseu que hem trobat Amèrica! Ja des de principis de l'existència de l'ésser humà s'ha mantingut en la recerca de màquines i processos que ens ajudessin en el processat de números. En aquest cas, fem us de les ultimes aplicacions al nostre abast, utilitzem un entorn de programació, el Matlab, que és una potent eina, eficaç i tant creativa com ens ho permeti el nostre cap. Per saber una mica què és això del Matlab, dir que és un *software* de programació que utilitza un llenguatge propi anomenat *M*. En el seu principi va ser creat pensant en utilitzar subrutines de codis *Fortran* per aplicar-ho a anàlisi lineal i numèric. Així doncs, a través d'entendre diversos mètodes numèrics intentarem escriure programes d'ordinador per a resoldre problemes matemàtics.

Ara bé, què són els mètodes numèrics? Són eines que ens permeten, a través d'un càlcul concret, trobar aproximacions a la solució del problema. L'avantatge que suposen aplicats en el mon de la enginyeria, és que ens permet realitzar infinites (entenent infinites com a sentit figurat) iteracions o repeticions d'una operació, de tal manera que ens acostem tant com ens interessi a la solució desitjada, i així podem ser capaços de resoldre models biològics, físics, químics, etc. L'avantatge que suposen els mètodes numèrics davant les solucions analítiques és que aquests ens permeten analitzar satisfactòriament certs problemes que d'altre forma, buscant una solució de forma analítica ens seria impossible de realitzar, o bé per la inexistència de la solució analítica o per la immensa complexitat que suposa el mètode analític.

OBJECTIUS

Els objectius d'aquest treball són, com en tots els altres treballs de recerca, l'aprenentatge en la cerca i síntesi d'informació. Aprendre a estructurar i redactar un document d'aquestes dimensions també es una cosa nova on toca parar-hi atenció. A part de la cerca d'informació però, cada treball també té els seus objectius propis, en aquest cas intentarem:

- Trobar els mètodes numèrics capaços de donar solucions a problemes que s'ajusten a models físics o biològics i donar aplicacions.
- Familiaritzar-se amb l'entorn de treball Matlab aconseguint un aprenentatge introductori en el món de la programació.
- Realitzar programes utilitzant els mètodes, intentant crear aplicacions internes i externes a Matlab.
- Assolir coneixements matemàtics per ser capaç de resoldre numèricament equacions polinòmiques i diferencials.

Un dels altres objectius no tant evidents i més personal que integra aquest treball és recuperar l'interès perdut en el món de les matemàtiques i poder entendre millor alguns conceptes que d'altra banda poden semblar avorrits i feixucs.

El que s'intentarà a continuació és l'explicació de tres mètodes numèrics per resoldre equacions del tipus $f(x)=0$, altres tres mètodes per resoldre equacions diferencials $dy/dx=f(x,y)$ i fer la explicació de com convertir desenvolupaments matemàtics en aplicacions informàtiques. Posteriorment s'estudiaran diverses situacions on es plantegen problemes amb la seva aplicació informàtica corresponent.

INTRODUCCIÓ ALS MÈTODES PER TROBAR ZEROS DE FUNCIONS

En el primer apartat d'aquest treball s'explicaran com poder resoldre equacions de forma numèrica, no analítica. Tots sabem, per exemple, que existeix una formula que ens permet resoldre una equació amb incògnita de segon grau. Aquest tipus de deducció de la solució és de forma analítica. Ara bé, si volem resoldre de forma no analítica aquesta equació hem de fer referència a uns mètodes que ens permetran fer una aproximació a la solució tant precisa que no considerarem que no sigui la solució "real".

El mètode és una eina, que de forma informàtica pot ser molt eficaç. Ara bé, tot mètode necessita una teoria bàsica. En aquest cas per trobar el zero d'una funció farem referència al teorema de Bolzano. Aquest matemàtic, procedent de la zona que actualment és la república Txeca va publicar l'any 1817 un teorema que ens és molt útil i ens ajudarà a entendre els mètodes a explicar. El teorema de Bolzano diu:

$$Si\ f(x) \text{és}\ continua\ a\ l'intèrval\ [a,b]\ i\ f(a)\ \text{té}\ diferent\ signe\ de\ f(b)$$
$$llavors\ \exists\ c \in (a,b)\ tal\ que\ f(c) = 0$$

Tenim una funció *f(x)* en un determinat interval tancat *[a,b]* que és contínua creixent o decreixent calculem *f(a)* i *f(b)*. Si es satisfà la condició de que *f(a)* i *f(b)* siguin de signe contrari, un positiu i l'altre negatiu o viceversa, podem assegurar que existeix un punt en l' interval *(a,b)* anomenat *c* tal que *f(c)* serà zero.

A partir d'aquest teorema farem l'estudi de dos mètodes per trobar solucions a funcions polinòmiques, que podem descriure-les per f*(x)=0*. Cal remarcar que els mètodes a estudiar només són capaços de trobar una sola arrel. Si tinguéssim el cas de voler resoldre una equació de tercer grau per exemple, s'haurien d'escriure tres intervals de cerca diferents i operar en ells per separat. Ara bé, veurem que a través de les eines informàtiques, mètodes que de forma manual podrien semblar no gaire convenients degut al seu volum d'operacions, utilitzant l'ordinador, podem fer moltes iteracions de forma automàtica i així trobar zeros de funcions amb eficàcia i rapidesa.

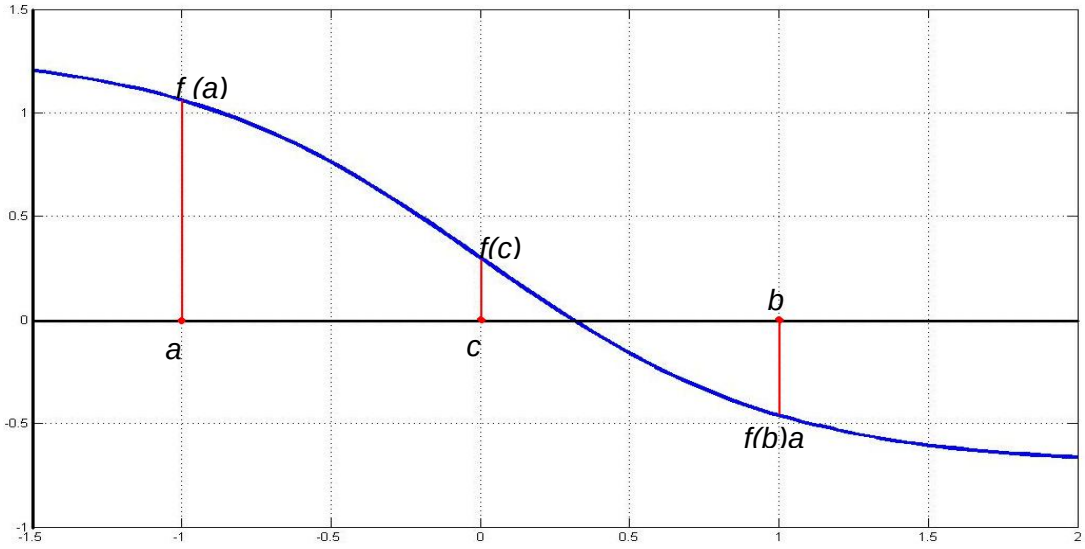
MÈTODE DE LA BISECCIÓ

Mètode matemàtic:

El primer mètode estudiat és el que s'anomena mètode de dividir l'interval (*halving interval*) o també conegut amb el nom de mètode de la bisecció. Aquest mètode ens serveix per trobar el zero d'una funció. Per tant, el mètode de bisecció que s'estudiarà serveix per resoldre equacions d'una forma més o menys senzilla.

El seu funcionament és força fàcil. Donada una funció determinada *f(x)* s'agafen dos punts a l'atzar, això si, a prop d'on hi ha el canvi de signe de la funció. Necessitem que per un dels punts, *a*, la funció *f(a)* sigui positiva i per l'altre punt, *b*, la funció *f(b)* sigui negativa o bé al contrari. Això és necessari ja que així ens assegurem que per un punt *c* que es troba dins [*a,b*] la funció tingui un zero, o sigui, *f(c)=0*. Un cop designat aquest interval [*a*,*b*] amb la solució en un punt desconegut d'entre els dos extrems prosseguim a fer-ne la meitat. La formula que designa com trobar aquest nou punt, anomenat *c*, és:

$$c = \frac{a+b}{2}$$



Com es pot comprovar, el que hem fet és trobar un punt *c* que és just al mig de l'interval designat. El següent pas és saber si la solució és *a*<*x*<*c* o bé entre *c*<*x*<*b*. Per fer-ho avaluarem si la funció *f(c)* té el mateix o diferent signe que la funció els altres punts. Per tant:

$si\ signe\ f(a) = signe\ f(c)$ → la solución no es troba a l'intèrval [*a*,*c*]

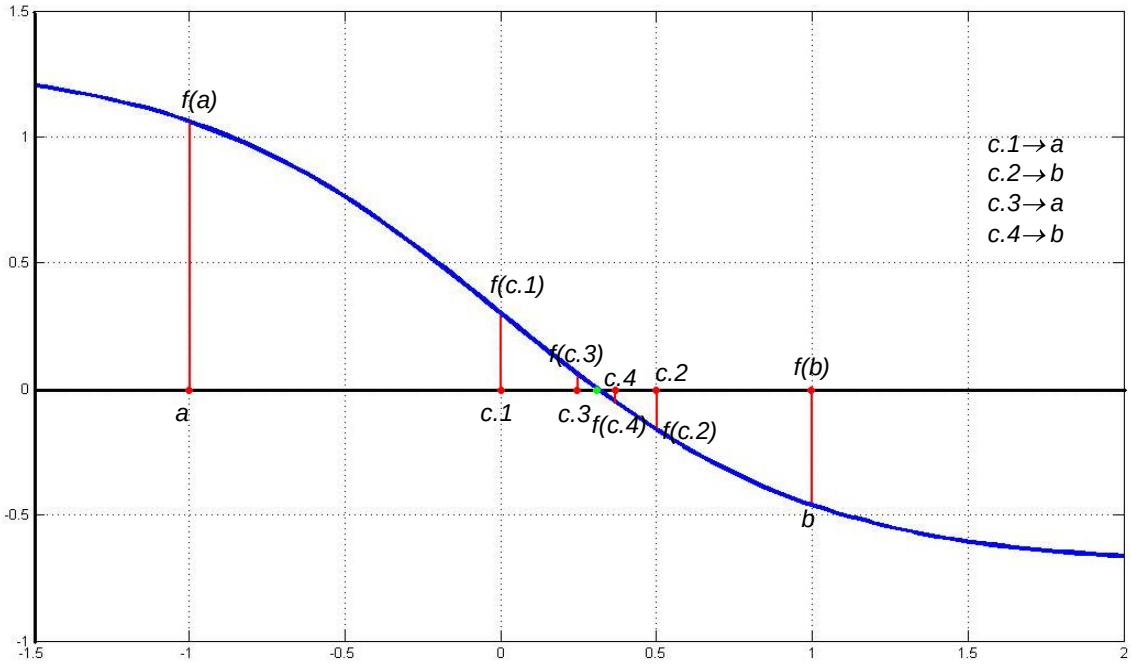
$si\ signe\ f(b) = signe\ f(c)$ → la solución no es troba a l'intèrval [*c*,*b*]

Ara que ja sabem a quin interval es troba la solució hem de reanomenar la *c* per *a* o la *b*, i així poder refer el procés i repetir-lo tantes vegades com calgui fins a acostar-nos tant a la solució com vulguem. Per reanomenar els valors seguirem el següent esquema:

$$si\ signe\ f(a) = signe\ f(c) \ \Rightarrow \quad a = c \quad f(a) = f(c)$$

$$si\ signe\ f(b) = signe\ f(c) \ \Rightarrow \quad b = c \quad f(b) = f(c)$$

Ara doncs ens trobem igual que al començament però amb un interval la meitat de gran. Repetirem el procés fins a trobar la solució. Cada un d'aquests cicles s'anomena iteració.



Cal dir que aquest i molts dels mètodes no donen mai el valor concret de la solució, sinó que fem aproximacions amb un màxim d'error que nosaltres mateixos podem definir. En aquest cas, per cada iteració que fem, reduïm l'error a raó d'una meitat. Per tant, si volem ser molt precisos ens caldrà fer moltes iteracions o bé utilitzar algun altre mètode més elaborat que amb menys iteracions faci menys error. Aquest serà el cas del següent mètode.

Algoritme informàtic:

Ara que ja hem vist el funcionament del mètode, podeu comprovar que aquest només és útil si s'utilitza de manera repetitiva. Si intentéssim fer el mètode a mà, ens en sortiríem, però necessitaríem molt de temps i precisament els mètodes són per no perdre'n! Per tant, aquí és on entra l'àmbit computacional. En aquest cas treballarem amb una eina de programació especialitzada en l'àmbit del processat de mètode numèrics; el Matlab. Més endavant es dedicarà un capítol a explicar-ne l'entorn.

El funcionament del programa amb el mètode de la bisecció és molt simple. Vegem-ne primer l'algoritme i després una breu explicació.

```
Clc; Clear all

while abs(fc)>tol              % Loop i control convergència
    c=(a+b)/2; fc=funcio_1(c)  % Càlcul punt intermedi
    plot(c,fc,'or')            % Dibuix iteració
    if sign(fa)==sign(fc)      % Si signe(fa)==signe(fc)
        a=c; fa=fc;            % llavors c substitueix a
    else                       % En cas contrari
        b=c; fb=fc;            % c substitueix b
    end
    niter=niter+1 ;            % Compta iteracions
end                            % Fi del loop
```

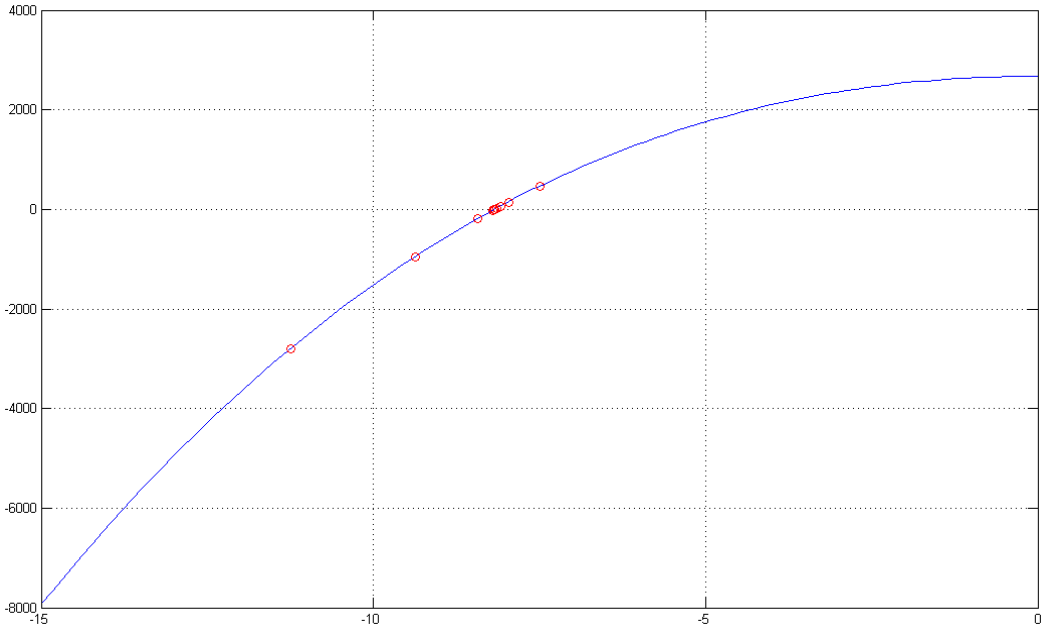
El seu funcionament és de la següent manera

- Prèviament hem de definir els valors i dades sobre les que treballarem, però la part del programa que treballa sobre el mètode de la bisecció és la especificada aquí a dalt.
- Mentres *f(c)* sigui més gran que el grau de tolerància que hem definit prèviament torna a calcular un punt intermedi.
- Dibuixa cada una de les iteracions que el programa fa (cada pas)
- Reanomena la *c* per una *a* o una *b* seguint el criteri establert
- Cada cop que el programa fa això, conta un pas més a les iteracions i torna a començar

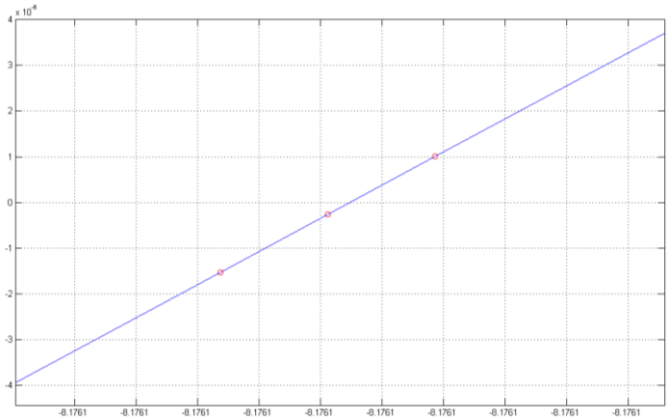
En aquest dibuix creat amb Matlab podem observar cada una de les iteracions fetes (vermell) fins a trobar una solució molt propera a l'arrel de l'equació. En aquest cas és només un exemple, que s'ajusta a l'equació escrita a continuació. Durant l'explicació dels mètodes per buscar zeros de funcions, utilitzarem aquesta funció com a exemple.

$$f(x) = x^3 - 30x^2 + 2552$$

Si ens acostem prou en el gràfic podem observar que la solució que dona no és ben bé exacte. Això és degut a que tots els mètodes, com ja hem dit abans, no donen una solució exacte, sinó que en fan una aproximació molt exacte, tant com nosaltres determinem.



En aquest cas acotem la exactitud del mètode dintre l'ordre *while* amb el tros de codi referent a tol, la tolerància. Al programar determinem la precisió del mètode dient l'error màxim que pot cometre. En tots els casos s'ha fet servir una tolerància amb valor de 10^{-6}.

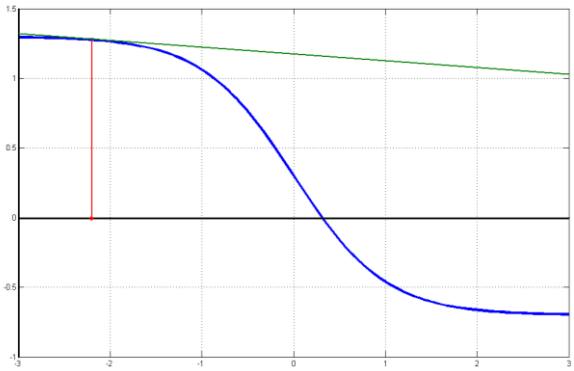


Observant bé el gràfic veurem que trobem una solució molt propera a zero però que se'n desvia una mica. Hem de tenir en compte l'escala a la que dibuixem, 10^{-6}, valor de la tolerància.

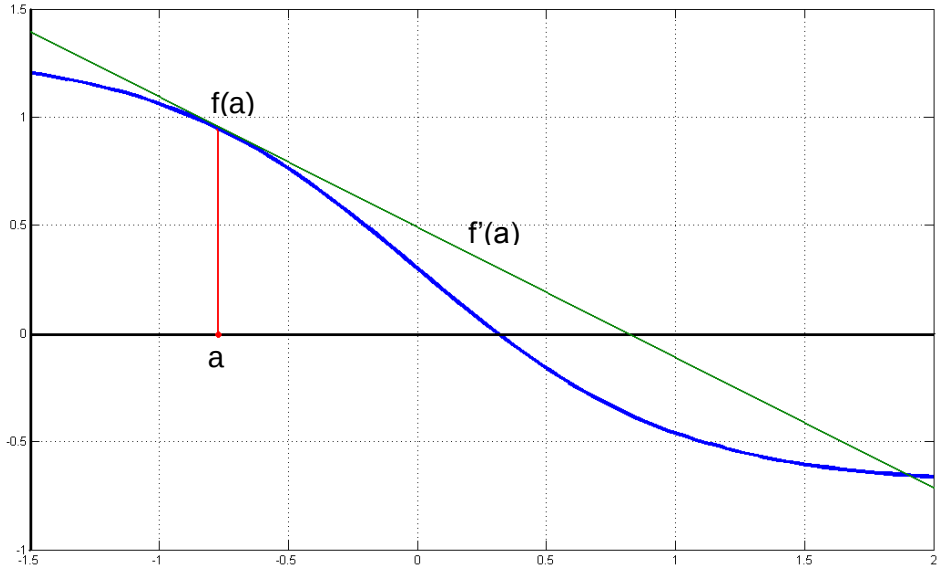
NEWTON-RAPHSON O MÈTODE DE NEWTON

Mètode matemàtic:

Un dels mètodes més eficaços i elegants que podem utilitzar a l'hora de buscar solucions o zeros de funcions és el que s'anomena mètode de la tangent. També és més conegut sota el nom de mètode de Newton-Raphson. És pràcticament el mètode més utilitzat i conegut entre els matemàtics per resoldre equacions.



Una de les normes més importants que cal tenir en compte abans de qualsevol explicació és la necessitat del mètode de començar a buscar una arrel des de un punt proper a la solució. Una condició evident és que hi ha d'haver canvi de signe de la funció (teorema de Bolzano), d'aquesta manera sabrem que hi ha una solució. Si la funció es derivable una o fins i tot dues vegades també serà un altre indici de que hi ha una solució. Ara bé, en aquest cas també necessita complir d'altres condicions no tant evidents. És necessari començar a buscar dins d'un interval que no contingui cap màxim ni cap mínim de la funció, per tant hem d'utilitzar el mètode dintre un interval on la funció sigui monòtona creixent o decreixent. Aquest fet és necessari perquè sinó la tangent no convergiria i no trobaríem mai la solució. Un exemple gràfic de una tangent que no convergeix, a dalt, i una altre en el que podrem utilitzar el mètode, a baix. Per tant, a partir d'això deduïm que és molt important triar un bon punt d'inici a l'utilitzar aquest mètode.



El funcionament del mètode implica utilitzar la derivada de la funció. Començant amb un punt *a* que contingui tots els requisits ja explicats prosseguirem a trobar un nou punt més proper a la solució. Aquest punt n'hi direm *b*, i és trobat a partir de:

$$b = a - \frac{f(a)}{f'(a)}$$

El nou punt és la resta del punt inicial menys la divisió del punt que designa *f(a)* menys la seva derivada primera. Aquesta expressió la podem deduir-lo a traves de l'equació de la recta que passa per *f(a)* i té com a pendent *f'(a)*. Per tant, ens quedarà una equació de tipus recta punt – pendent. La seva forma genèrica és:

$$y - y_o = m\,(x - x_o)$$

I escrivint-la amb les variables que ens interessen.

$$y - f(a) = f'(a)\,(x - a)$$

Llavors necessitem trobar la solució del punt on aquesta talla a l'eix d'abscisses, altrament dit, eix de les *x*. El valor de x que fa y=0, és doncs la solució de:

$$0 - f(a) = f'(a)\,(x - a)$$

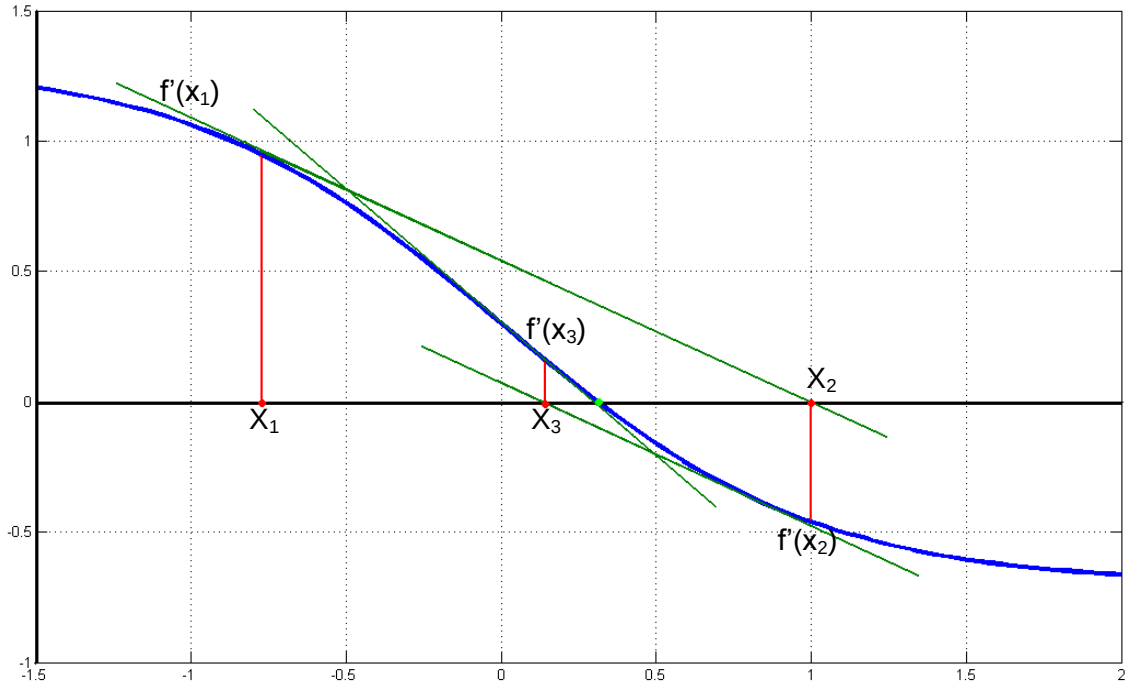
I ara només cal arreglar una mica la formula i aïllar x, que és el nou punt que estem buscant, que de forma genèrica n'hi diem *b*.

$$\frac{-f(a)}{f'(a)} = x - a \;\rightarrow x = a - \frac{f(a)}{f'(a)}$$

Una forma més general de descriure literalment aquesta equació seria que la nova *x*, o també escrita com a *x* del pas *n+1* es igual a la *x* del pas n menys la seva funció dividida per la seva derivada. Aquesta és la formula que hem deduït, però escrita de forma més pensada en ser utilitzada en el mètode. Això seria transcrit així:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Una vegada el procés d'iteracions comença, el mètode convergeix d'una forma molt més ràpida que en el mètode vist anteriorment.



Com a gran avantatge que ens proposa el mètode de Newton-Raphson és l'error *E* que comet, ja que es veu reduït a raó de E^2 per cada iteració que realitzem. Per tant, podem arribar a ser molt precisos utilitzant poques "tirades" i això implica reduir el rendiment que se li demana a la màquina a l'hora d'aplicar el mètode computacionalment.

Algoritme informàtic:

El tros de codi que correspon a aquest mètode és una mica més complicat d'escriure que els anteriors. Començarem doncs, fent-ne una comparació. En l'anterior mètode s'ha utilitzat sempre una mateixa funció d'exemple f(x). Al buscar el zero de la funció amb el mètode de la bisecció, hem trigat 33 iteracions. Sorprenentment, amb les mateixes condicions utilitzades en el programa anterior, el mètode de Newton-Rhapson només necessita 5 iteracions per resoldre l'equació. Això ens dona una petita idea de la millora que suposa el mètode estudiat.

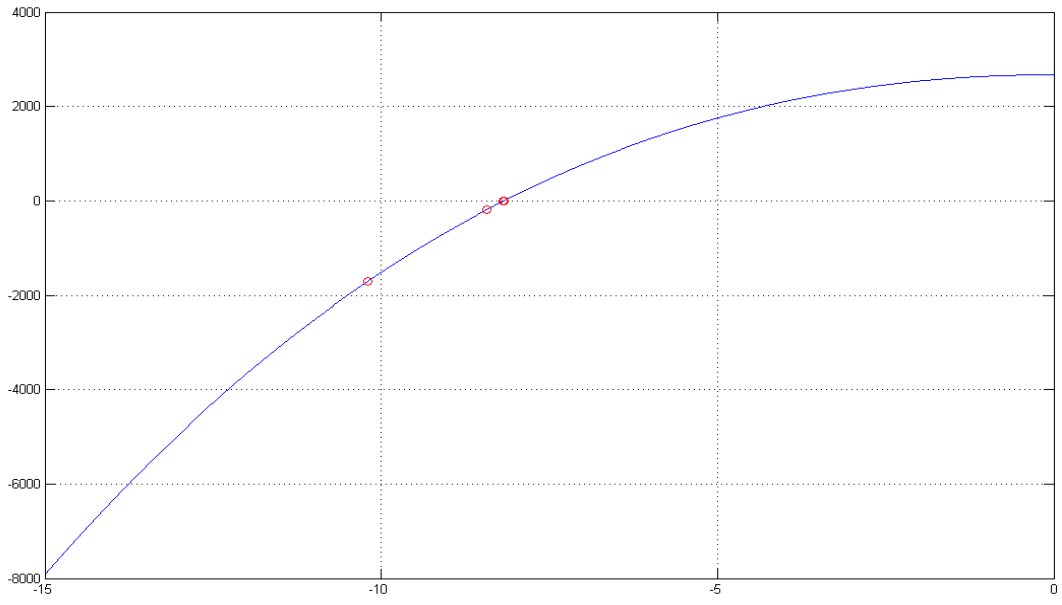
Acabada la comparació dels mètodes per buscar zeros de funcions, prosseguirem a explicar el funcionament de Newton-Rhapson amb el programa Matlab. Primer vegem-ne el codi:

```
Clc; Clear all

niter=0                                ; % Comptador d'iteracions
fx2=2*abs(tol)                         ; % Assegura (fx2)>tol perquè entri en while
while abs(fx2)>tol & niter<100           % Itera abs(fx2)>tol i - de 100 iteracions
    [fx1,dfx1]=funcio_i_derivada_1(x1) ; % Calcula fx1 i dfx1 amb el valor de x1
    x2=x1-fx1/dfx1                     ; % Troba un nou valor de x2
    [fx2,dfx2]=funcio_i_derivada_1(x2) ; % Amb el nou valor de x2 calcula fx2 i dfx2
    x1=x2                              ; % Copia el valor de x2 sobre x1
    niter=niter+1                      ; % Comptador d'iteracions
    disp(['niter= ',num2str(niter),'  x2=',num2str(x2,16),' fx2=',num2str(fx2,16)]);
                          % Escriu cada iteració  amb 16 decimals
    plot(x2,fx2,'or');    % Dibuixa cada iteració (vermell) sobre la funció
end                       % Deixa de funcionar
```

En aquest cas , per la seguretat del programa és necessari definir un nombre màxim d'iteracions que pot fer. Això és necessari degut a la utilització de l'ordre *while*. Més endavant, en un altre apartat veurem les diferencies entre diferents ordres de programació. El problema que pot presentar una ordre *while* és la possibilitat de quedar-se iterant infinitament el programa. Definint un nombre màxim d'iteracions ens assegurem que no entri en un *loop* infinit, i això és la primera ordre, que no és estrictament del mètode, però és necessària per fer-ne un control. Una altre ordre que tampoc és estrictament del mètode de Newton-Rhapson és la segona. Aquí és necessari assegurar-se que la primera iteració que fem entri dintre el *loop*, i això podem fer-ho doblant el valor de *fx2*.

Tot seguit la explicació de l'algoritme:

- El tros de codi que correspon explícitament al mètode és l'algoritme que comença amb l'ordre *while*, seguida de les condicions que ha de complir. Quan es superi alguna d'aquestes condicions, deixarà de funcionar o és donarà per acabat. Per funcionar comprovem que *f(x)* on *x=x2*, per tant (*fx2*), sigui més gran que la tolerància. Quan sigui més petit, voldrà dir que som prou propers a la solució com per poder dir que l'hem trobada.
- Tot seguit fem calcular *f(x)* i *f'(x)* pel valor de *x1*. Això servirà per poder calcular el pas següent *(x2)* utilitzant la formula del mètode. Aquesta ordre no calcula màgicament el valor de la funció i la derivada, sinó que llegeix aquesta ordre d'un altre programa exterior on hi ha escrita la funció que volem treballar. Això ens serveix per poder utilitzar el mètode amb altres equacions
- Calculem *f(x)* i *f'(x)* amb el valor de *x2* que hem trobat. Aquest valor és el que s'avaluarà com a condició per continuar iterant o no.
- Reanomenem el valor de *x2* per *x1* i així és possible continuar iterant.
- Seguidament augmentem un nombre el comptador d'iteracions i diem al programa que escrigui els valors de cada una.
- Finalment ordenem dibuixar cada iteració.



Podem observar fàcilment que el nombre de "tirades" necessàries és redueix força. Això implica que la màquina necessita menys temps de computació per la CPU del sistema perquè el nombre de càlculs és menor.

INTRODUCCIÓ ALS MÈTODES PER SOLUCIONAR EQUACIONS DIFERENCIALS

Després de veure el funcionament de diversos mètodes que serveixen per trobar zeros de funcions ara treballarem amb altres mètodes que ens serviran per resoldre equacions diferencials ordinàries. Aquestes solen ser del tipus

$$\frac{dx}{dt} = f(t, x(t))$$

Una equació diferencial és aquella que apareix com a una funció amb una de les seves derivades. En aquest cas poden ser de primer, segon, tercer, etc. ordre depenent del grau de la derivada que hi surti. Si hi apareix una derivada segona, serà una equació diferencial de segon ordre. La solució per a una equació diferencial és una funció específica que s'ajusta a aquesta mateixa. Aquesta funció podem determinar-la d'una forma analítica en alguns casos, però en molts d'altres no hi ha solució analítica i s'ha de buscar amb mètodes numèrics, que aplicats a tècniques informàtiques són molt eficients. Per resoldre una equació diferencial podem utilitzar diversos mètodes. Seguidament estudiarem tres dels més utilitzats: Mètode d'Euler, Mètode de Runge-Kutta de segon ordre i Mètode de Runge-Kutta de quart ordre.

Veurem el seu funcionament informàtic, que en tots els casos serà semblant, i aprofitarem per comparar els mètodes entre ells mateixos. Per entendre com resoldrem numèricament la integració, hem d'imaginar-nos que fem la funció a trossos tant petits que podem considerar que és un conjunt de petites rectes. Així també veurem que no només intervé quin mètode utilitzem, referint-nos a la seva eficàcia, sinó que també dependrà d'altres factors com el nombres de passos i l'avanç de temps que utilitzem.

MÈTODE DE EULER-CAUCHY

Mètode matemàtic:

Com a mètode inicial que s'estudiarà per resoldre equacions diferencials ordinàries presentem el mètode Euler-Cauchy, encara que per fer-ho més senzill s'anomena simplement mètode d'Euler. Aquest mètode veurem que malgrat tenir un senzill sistema d'iteracions pot arribar a ser prou bo per fer una primera aproximació en la integració d'equacions diferencials.

En general cal tenir en compte que per a poder trobar la solució numèrica que busquem d'una equació de ordre *n*, necessitem convertir-la en *n* equacions de primer ordre. Llavors, una equació diferencial de segon ordre és necessari que sigui convertida en dues equacions de primer ordre. Per fer-ho més entenedor vegem-ho en un exemple. En aquest cas treballarem sobre el model físic d'un dels problemes que solucionarem posteriorment, la caiguda lliure. El problema es modelitza per aquesta equació, que surt de la llei física ∑F= m*a.

$$\frac{d^2x}{d^2t} = g - \frac{F_{fricció}\,(v)}{m}$$

S'ha escrit $F_{fricció}(v)$ per recordar que la força de fricció és funció de la velocitat. Per convertir aquesta equació diferencial de segon ordre en un sistema de dues equacions, comencem per introduir una nova variable que sigui igual a la derivada primera de x, que en aquest cas coincideix amb una variable que té sentit físic, la velocitat. Per tant, la velocitat és la derivada primera de la posició respecte el temps, i la derivada segona que hi havia en l'equació original es converteix en la derivada de la velocitat respecte el temps, o sigui:

$$\frac{dx}{dt} = v \qquad\qquad \frac{dv}{dt} = g - \frac{F_{fricció}\,(v)}{m}$$

El mètode d'integració per Euler es pot deduir de forma simplificada substituint les derivades per un quocient d'increments que s'augmenta de forma finita. Fem diversos increments per tal de poder calcular un nou valor de la funció. Com més petit sigui el nombre que augmentem en cada iteració, més acurat serà el nostre càlcul però també necessitarà més "tirades". Finalment queda que:

$$\frac{dx}{dt} = \lim_{\Delta t \to 0} \frac{\Delta x}{\Delta t} \approx \frac{\Delta x}{\Delta t} = v \qquad\qquad \frac{dv}{dt} = \lim_{\Delta t \to 0} \frac{\Delta v}{\Delta t} \approx \frac{F_{fricció}\,(v)}{m}$$

Ara que ja hem vist la manipulació matemàtica necessària per utilitzar Euler, podem escriure de forma computacional com queda el sistema a resoldre. Ho farem utilitzant de forma general una expressió que ens diu com passar del pas *n*, que és el valor que actualment sabem, al pas *n+1*, que és el nou valor que es vol trobar. Recordem que continuem treballant sobre l'exemple.

$$x_{n+1} = x_n + \Delta t * v$$

$$v_{n+1} = v_n + \Delta t * \left(g - \frac{F_{fricció}\,(v)}{m} \right)$$

Un cop entesa la explicació del mètode, conclourem escrivint-ne la seva equació general per poder-ho utilitzar en qualsevol cas. L'equació diferencial que es vol resoldre és:

$$\frac{dy}{dt} = f(t, y)$$

I el mètode d'avanç de la solució es descriu així :

$$y_{n+1} = y_1 + f(t, y) * dt$$

Trobarem el nou valor de la variable sobre la qual treballem, y*(n+1)*, utilitzant el valor ja conegut, y*(n)* , més un increment *h* o *dt* que definirem al principi multiplicat per la seva derivada primera. Aquest increment pot ser que tingui un sentit físic en el problema i que sigui el pas de temps (*dt*) o que simplement és un increment que fem en la funció (*h*). També s'ha de definir un valor inicial per cada funció que integrem. Per exemple, en un equació diferencial de segon ordre doncs, haurem de definir un valor inicial per la derivada primera i un per la segona. En l'exemple utilitzat doncs, s'hauria de definir un valor inicial per la posició i un per la velocitat.

Finalment, la formula que descriu el mètode d'Euler és:

$$y = f(x, y) * h \quad \rightarrow \quad y_{n+1} = y_n + hf(x_n, y_n)$$

Algoritme informàtic:

El funcionament d'aquest mètode podem anunciar que és força diferent dels vistos anteriorment. El primer punt que cal remarcar és que treballarem amb Matlab. És un punt important degut a la diferència de funcionament que pot haver-hi entre diversos llenguatges de programació. A diferència dels altres programes vistos anteriorment, aquest necessita uns passos previs per poder funcionar. Com ja s'ha explicat en el mètode matemàtic és necessari definir uns valors inicials per poder començar a desenvolupar els càlculs. En aquest tros de codi podem veure que les dades inicials és defineixen com a dades del pas numero *1*. A partir d'aquí podrem trobar un nou valor *(n+1)* que en la següent iteració serà anomenat *n*.

```
% Dades inicials
y(1)=y0 ;
v(1)=v0 ;
t(1)=0  ;
```

$$a(n+1) = a(n) + a'(n) * dt$$

La gran diferència que representa un mètode com és el d'Euler, en comparació als mètodes estudiats de trobar zeros de funcions, és la necessitat de guardar uns valors que necessitarem per la següent iteració. De fet, aquests valors són part de la funció que resol l'equació diferencial. Hem vist que el funcionament del mètode es complau a trobar un nou punt que és un "passet" més endavant a partir de la dada que ja tenim. A més, s'utilitza la derivada d'aquesta mateixa multiplicat per un pas de temps, *dt*, o increment com es sol anomenar. Veurem que la precisió que pot desenvolupar aquest mètode depèn molt de la mida d'aquest increment de temps que utilitzem. En una primera aproximació per saber l'error comès, podem deduir que com més gran sigui el pas de temps, més error crearà el nostre programa.

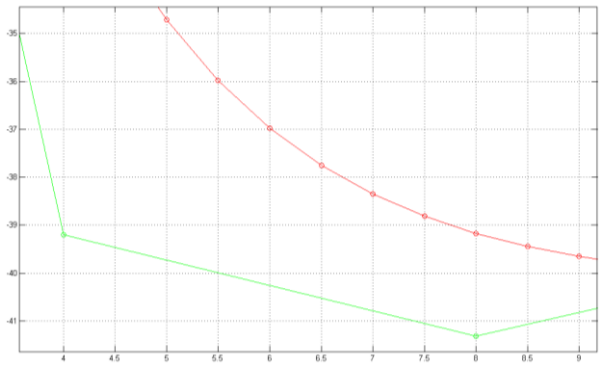
Com ja s'ha dit, necessitem guardar una sèrie de valors per treballar amb aquest mètode utilitzant la màquina. En la majoria de mètodes per resoldre equacions diferencials, necessitem reservar un espai de memòria que guardi les dades ja obtingudes. Com que el mètode treballa sobre els resultats ja coneguts, accions del pas *n*, per calcular els del nou pas, *n+1*, hem de reservar un tros de la memòria en el funcionament del programa perquè pugui escriure i recuperar aquestes dades. Això ho aconseguirem creant uns vectors amb un espai definit de dades, que anirà des del numero *1* i fins al desitjat. En un primer moment, el programa omplirà tots aquests vectors de zeros, per tant, estaran buits. A mesura que cada iteració trobi el nou valor, l'anirà copiant a una casella del vector corresponent i així podrà utilitzar aquest numero per la següent iteració.

```
% Vectors
np=5000          ;
t=zeros(np,1)    ;
y=zeros(size(t)) ;
v=zeros(size(t)) ;
```

Ara que ja hem vist els passos previs a la ordre que calcula iterativament els valors de la funció, podem acabar de veure el tros de codi que farà el funcionament en si del programa referent al mètode d'Euler.

```
% Càlcul d'iteracions

for n=1:np-1                % Itera des de n=1 fins al màxim -1
y(n+1)=y(n)+v(n)*dt  ;      % Càlcul de la nova y
a=funcio(y(n),v(n),t(n)); % Càlcul de la a com a f(y,v,t).
v(n+1)=v(n)+a*dt;           % Càlcul de la nova v
t(n+1)=t(n)+dt ;            % Augmenta el temps
    if y(n+1)<=0            % Si la nova y és = o  bé < de zero
        break               % Para d'iterar
    end                     % Fi de l'ordre condicional
end                         % Fi de les iteracions
```

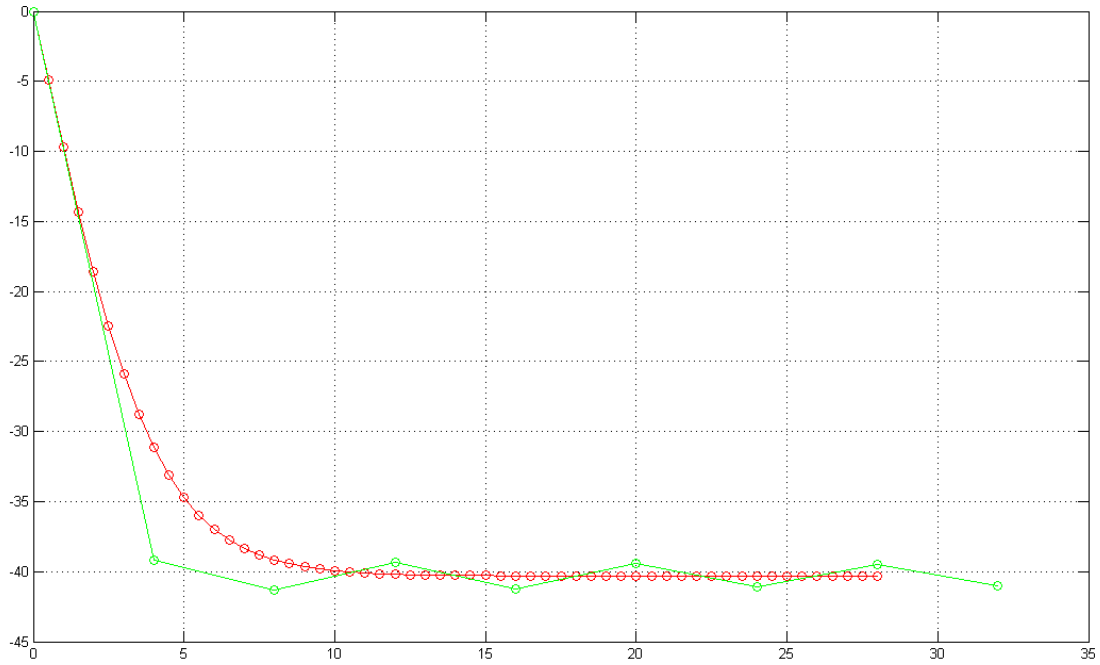
Com es pot veure, el programa va fent iteracions fins que troba l'ordre condicional que mana que pari de treballar. Un altre fet a remarcar és la primera línia del codi. Com que prèviament ja havíem definit un valor inicial, *n=1*, després de l'ordre *for* indiquem que iteri des d'aquest primer valor fins al màxim de nombre de passos menys un, *np-1*, que hem definit. Això és necessari perquè sinó tindríem els valors de nombre de passos (*np)* més el primer que hem definit, i cal recordar la importància de definir els vectors correctament, sinó el programa pot esdevenir inestable, lent i acabar concloent en un avort del funcionament de la màquina.



Les línies de codi que s'han presentat per fer l'explicació computacional del mètode formen part del mateix exemple que s'ha utilitzat per explicar el mètode matemàtic, un problema de caiguda lliure. Ara bé, com que tenen una aplicació pràctica, que serà vista en l'apartat de problemes, podem utilitzar ara el programa escrit per fer una comparativa dintre el mateix mètode d'Euler. Fent referència a l'importància de l'error que generen els mètodes numèrics en aquest cas es digne de remarcar que el mètode d'Euler té un error proporcional a dt^2 o h^2, valor que nosaltres podem definir. Si utilitzem un pas de temps petit el mètode serà molt més precís que si n'utilitzem un de més gran.

Per fer-ne un exemple molt gràfic suposem que volem dibuixar un cercle. En un cas disposem només de cinc barres de fusta per fer-ho, i en un altre en disposem de vint. La figura que s'assemblarà més a un cercle serà la que utilitzi els trossets més petits de fusta per fer la construcció perquè podrà utilitzar-ne més. Els mètodes numèrics creen aproximacions amb trossos de línia recte semblant a l'exemple que hem posat. Per veure-ho millor aquí un gràfic fent la comparativa del mateix mètode integrant la mateixa funció, un utilitzant un *dt* de 0,5 segons (vermell) i l'altre de 3 segons (verd).

En l'exemple gràfic de la pàgina anterior s'aprecia la diferència que crea en les corbes utilitzar o no un pas de temps petit. En el gràfic complet es veu com el mètode soluciona molt millor l'equació diferencial utilitzant un pas de temps més petit. Amb el més gros, comença a esdevenir oscil·lant, cosa que el fa inestable i no gens fiable. Ara si, cal remarcar també, que utilitzant un pas de temps menor, estem demanant més esforç computacional a la màquina, ja que en un cas acaba amb menys iteracions i que en l'altre en requereix més. Per tant, cal buscar un equilibri entre l'ús del mètode i l'esforç computacional que necessiti.



També dir que amb les potències dels ordinadors actuals no suposa cap problema operar amb mètodes com aquests, ja que els processadors actuals són capaços de treballar amb quantitats grans de números de forma ràpida.

RUNGE-KUTTA DE SEGON ORDRE O MÈTODE DE RUNGE-KUTTA SIMPLIFICAT

Mètode matemàtic:

Podem considerar que el mètode d'Euler que hem vist és un bon mètode sempre i quan utilitzem un pas de temps prou petit perquè funcioni correctament. Ara bé, podem millorar el mètode prèviament estudiat i incrementar-ne el rendiment fent que consumeixi menys esforç per la màquina i que amb menys iteracions aconseguir un resultat més precís. Això serà gràcies al mètode Runge-Kutta. El mètode de Runge-Kutta (a partir d'ara referit com RK) rep aquest nom en honor als seus creadors, una parella de matemàtics alemanys, C. Runge i W.M. Kutta. La gran fiabilitat i estabilitat que dona aquest mètode ha fet que sigui un dels més usats per solucionar equacions diferencials ordinàries.

El funcionament més senzill és atribuït al mètode de segon ordre, posteriorment s'explicarà el de quart ordre, que és encara millor.

El mètode de RK de segon ordre té un funcionament similar al d'Euler. Com hem vist el mètode d'Euler troba el nou valor de la funció incrementant el valor anterior amb el producte del pas de temps per la derivada. Ara bé, en el mètode d'Euler el valor de la derivada que s'utilitza pel càlcul s'ha calculat només al començament de l'interval i pot ser que aquesta derivada no sigui massa propera al valor mitjà de la derivada de tot l'interval. El funcionament del RK de segon ordre crea una millor aproximació de la funció fent servir la mitjana de diversos valors d'aquesta, com és el temps, dins de l'interval del pas de temps. En el cas del RK de segon ordre s'avaluen dues funcions, en el de quart ordre, quatre. Per tant, es poden escriure RK de *n* ordre avaluant *n* funcions, encara que els més comuns són els de segon i quart ordre.

Recordant el mètode d'Euler, aquest el podríem escriure així:

$$k1 = f(x, y(n)) * h$$
$$y(n+1) = y(n) + k1$$

Podríem interpretar el mètode d'Euler com a un RK de primer ordre i això seria del tot cert. Ara bé, no s'ha anomenat mai així i ha estat sempre conegut com a mètode d'Euler.

El mètode de RK de segon ordre busca un segon valor, *k2*, i en fa un promig amb *k1* per poder avançar la solució, d'aquesta manera:

$$k1 = f\big(x, y(n)\big) * h$$

$$k2 = f\left(x + \frac{h}{2}, y(n) + \frac{k1}{2}\right) * h$$

$$y(n+1) = \frac{k1 + k2}{2}$$

El valor de *k1* és idèntic a el mètode d'Euler. El valor de *k2* és l'increment de la funció que es calcularia si s'avalués la derivada avançant només mig pas de temps, o sigui a x+*h/2* i a *y(n)+k1/2*. Finalment per trobar el nou valor de la funció, *n+1*, es fa la mitjana dels dos increments que s'han trobat. Per fer-ho sumem *k1* més *k2* i ho dividim per la meitat, ja que sinó trobaríem un valor el doble de gran.

El valor de *h* és l'increment utilitzat, també anomenat *dt* en anteriors casos si fa referència al pas de temps. El mètode de RK és un mètode molt més precís que l'anterior estudiat. Aquí podem fer aproximacions de l'ordre de h^n en qualsevol mètode RK de n ordre. Llavors, en el mètode de RK de segon ordre crearem un error de h^2, en un de tercer ordre h^3, i així successivament.

Algoritme informàtic I:

A l'hora de programar RK podem aprofitar les línies de codi utilitzades en Euler. D'aquesta manera ja tindrem les dades inicials definides i els vectors ja definits. Sinó ho estiguessin, s'haurien de definir novament.

Per fer la part on hi haurà el *loop* que repeteixi cada vegada el càlcul, podem plantejar-ho de dues formes. Una consta d'escriure els passos *k1* i *k2* i després avançar la solució. Això ens permetria utilitzar el programa a la perfecció. Ara bé, si el que ens interessa és poder tenir un programa amb un mètode de RK escrit i fer-lo servir per diversos problemes amb diferents equacions per integrar i no volem escriure de nou el programa en cada cas, podem definir-lo en funció d'una variable externa. Vegem primer el cas més simple, d'escriure els passos *k1* i *k2* dintre el mateix programa.

```
% Dades inicials

t(1)=0  ; % temps
x(1)=x0 ; % conills
y(1)=y0 ; % guineus
```

Amb els vectors definits i les dades inicials procedirem a crear un *loop* finit. Ho farem amb l'ordre *for*, on indicarem que calculi des de *n=1* fins a *np-1*, per impedir sobrepassar els vectors definits. Cal recordar que el valor de nombre de passos o *np* el definim prèviament. Primerament farem el càlcul de *k1*, definint la *x(n)* com a *x1*. Això servirà pel funcionament intern de *k1*, on treballarem amb *x1*, que en el fons serà el valor *x(n)*. L'acció és requerida ja que per fer el càlcul de *k2* utilitzaríem també *x(n)* i utilitzar un mateix valor amb el mateix nom per calcular dos aspectes diferents (*k1* i *k2*) no es gens saludable pel programa. Per tant farem el mateix amb cada una de les derivades, suposant que n'hi hagin més de una. Recordem que una equació de *n* ordre té *n* derivades. En el nostre cas d'exemple, veurem que el problema agafat com a exemple (model Predator-prey de Lotka-Volterra) té dues derivades.

```
% Vectors

dt=0.01 ;
np=5000 ;
t=zeros(np,1) ;
x=zeros(np,1) ;
y=zeros(np,1) ;
```

Una vegada definit el valor per calcular *k1*, ho calcularem utilitzant la formula. Tot seguit s'escriurà l'ordre de *k2*, on definirem igualment *x2* com a *x(n)* més *k1/2*. Això és necessari perquè com ja hem vist a la formula *k2* es calcula amb la meitat de *k1*, així podem avaluar un cop més la funció i després fer-ne la mitjana. A continuació el càlcul de *k2* seguint la formula, però cal tenir en compte que ja hem atorgat el valor de *k1/2* a *k2*.

En el següent tros de codi es pot apreciar com el càlcul de *k1* i *k2* són idèntics. També es podria fer d'una altre forma indicant que *k1/2* va sumat en el càlcul de *k2*.

```
for n=1:np-1

    %K1
  x1=x(n)             ; % assigna x(n) a x1 per fer el càlcul de k1x
  y1=y(n)             ; % assigna x(n) a x2 per fer el càlcul de k1y
  k1x=dt*(a*x1-b*x1*y1)    ; % càlcul de k1 sobre variable x
  k1y=dt*(-c*y1+p*x1*y1    ; % càlcul de k1 sobre variable y

    %K2
   x2=x(n)+k1x/2       ; % assigna x(n) a x2 per fer el càlcul de k2x
  y2=y(n)+k1y/2        ; % assigna x(n) a y2 per fer el càlcul de k2y
  k2x=dt*(a*x2-b*x2*y2)    ; % càlcul de k2 sobre variable x
  k2y=dt*(-c*y2+p*x2*y2)   ; % càlcul de k2 sobre variable y
```

Com es pot comprovar en aquestes línies de codi només consta els càlculs de *k1* i *k2*, ara bé, és necessari calcular la nova solució, *n+1*, per poder començar de nou. Així doncs, a continuació de les ordres vistes hi va el següent tros de codi que és l'encarregat de fer el càlcul complert i fer la mitja dels valors que hem calculat.

```
  %Avanç solución

   x(n+1)=x(n)+(k1x+k2x)/2  ; % avança el càlcul de la variable x
   y(n+1)=y(n)+(k1y+k2y)/2  ; % avança el càlcul de la variable y
   t(n+1)=t(n)+dt           ; % avança pas de temps
end
```

Com a ordre final és necessari tancar el *for* amb l'ordre *end*.

Algoritme informàtic II:

Com hem comentat al principi de l'explicació de l'algoritme informàtic I, el mètode de RK pot ser programat d'una altre forma, cridant una funció externa del programa. En aquesta funció externa és on hi ha les equacions que es volen integrar. D'aquesta manera, aconseguim que escrivint una vegada el programa de RK, puguem utilitzar-lo per qualsevol altre equació diferencial que volem integrar, i en el fons, per altres problemes.

En aquest cas també és necessari determinar uns valors inicials i deixar uns vectors plens de zeros per poder-hi posar les dades que vulguem guardar. L'única diferència que comporta és l'apartat del *loop*, on calcularem els valors necessaris cridant la funció externa. Això significa que prèviament hem hagut de crear un altre arxiu amb atribut de funció i amb les variables necessàries. Continuant amb l'exemple del Predator-prey, vegem-ne la funció externa que serà necessària pel ple funcionament del programa.

```
function [kx,ky]=f_calcula_rk(x,y,dt,a,b,c,p)

      kx=dt*(a*x-b*x*y)   ;
      ky=dt*(-c*y+p*x*y)  ;

return
```

En la primera ordre diem que les següents línies són en atribut de funció amb l'algoritme *function*, n'indiquem el nom de les equacions que definirem posteriorment amb els símbols [ ]. Això equivaldrà a un nom que serà utilitzat en el programa i entre parèntesis posem les variables que intervenen en les equacions a integrar, que a la vegada són necessàries per completar la funció. Més avall escrivim la formula de les equacions i tanquem la funció amb l'ordre *return*.

Si volguéssim canviar de problema i haguéssim de treballar en un altre problema, el tros de codi que necessitaria ser canviat és aquest d'aquí dalt. Només necessitaríem crear un nou document dient-hi un altre nom i ordenant que utilitzi totes les derivades necessàries.

Una vegada escrita la funció que cridarem des de dintre el programa procedim a escriure el *loop* de càlcul en el programa, dient que faci els càlculs des d'aquí. Això s'indica de la següent forma

```
for n=1:np-1

    [k1x,k1y]=f_calcula_rk(x(n)      ,y(n)      ,dt,a,b,c,p) ;
    [k2x,k2y]=f_calcula_rk(x(n)+k1x/2,y(n)+k1y/2,dt,a,b,c,p) ;

%Avanç solució
    x(n+1)=x(n)+(k1x+k2x)/2 ; % avança solució x
    y(n+1)=y(n)+(k1y+k2y)/2 ; % avança solució y
    t(n+1)=t(n)+dt          ; % avança pas de temps

end
```

Dintre l'ordre de repetició direm que les variables (*k1x*,*k1y*) equivalen i han de ser llegides des de la funció que hem escrit abans, indicant que en cada cas utilitzi la *x(n)* i *y(n)*. Les altres variables són sempre fixes i per això no han estat definides com a vectors. Cal tenir en compte que a l'hora de fer el càlcul de *k2* hi ha d'afegir la meitat del valor de *k1* per després poder-ne fer la mitjana del valor. Després només serà necessari indicar que avanci la solució per poder repetir el procés.

El programa ofereix la mateixa precisió escrit de les dues maneres, ara bé, la segona forma d'escriure el programa ens permet poder resoldre altres problemes o equacions diferents sense haver d'escriure de nou tot el programa, només fa falta que canviem la funció externa i potser algunes variables que són de valor fix dins el programa principal.

RUNGE-KUTTA DE 4T ORDRE O MÈTODE CLÀSSIC DE RUNGE-KUTTA

Mètode matemàtic:

El mètode de quart ordre de Runge-Kutta es pot fer a partir del funcionament del de segon ordre. Simplement és una millora que ens permet incrementar el rendiment del programa necessitant menys volum de CPU i creant menys error, cosa que ens permet trobar la solució amb un nombre mes petit d'iteracions i el fa un programa molt més fiable.

Per entendre el seu funcionament partirem del RK de segon ordre explicat prèviament. Així com en l'altre cas només s'avaluava la funció en dos casos, dominats *k1* i *k2* i després se'n feia la mitjana, en aquest cas farem el mateix però avaluant la funció en quatre casos. Podem definir mètodes de Runge-Kutta de qualsevol ordre, avaluant la funció *f(x)* tantes vegades com *x* ordre volem que tingui el nostre mètode. Ara bé, es considera que el mètode de quart ordre genera una relació fiabilitat – requeriment molt acurat. Amb poc esforç matemàtic podem aconseguir resultats molt més precisos que els mètodes ja estudiats.

Per explicar-ne el funcionament matemàtic utilitzarem la nomenclatura que utilitzàvem en el mètode anterior. S'ha de fer un càlcul de *k1*, *k2*, *k3*, *k4* i després calcular-ne el valor mitjà. Refrescant la memòria, trobem que :

$$k1 = f\big(x, y(n)\big) * h$$

$$k2 = f\left(x + \frac{h}{2}, y(n) + \frac{k1}{2}\right) * h$$

Ara doncs, els càlculs de *k3* i *k4* resulten ser tant senzills com aquests:

$$k3 = f\left(x + \frac{h}{2}, y(n) + \frac{k2}{2}\right) * h$$

$$k4 = f(x + h, y(n) + k3) * h$$

Podem observar que per fer el següent càlcul sempre utilitzem el valor anterior ja calculat per avaluar-lo de nou. D'aquesta manera en comptes de fer una "tirada" cada vegada, en fem quatre i d'aquestes en fem el promig, que es fa seguint:

$$y(n+1) = y(n) + \frac{1}{6}(k1 + 2k2 + 2k3 + k4)$$

Referint-nos a l'error que cometem, podem arribar a fer aproximacions de l'ordre de h^4, llavors es comprensible que el mètode de Runge-Kutta de quart ordre sigui molt més utilitzat, ja que ens permet ser més precisos.

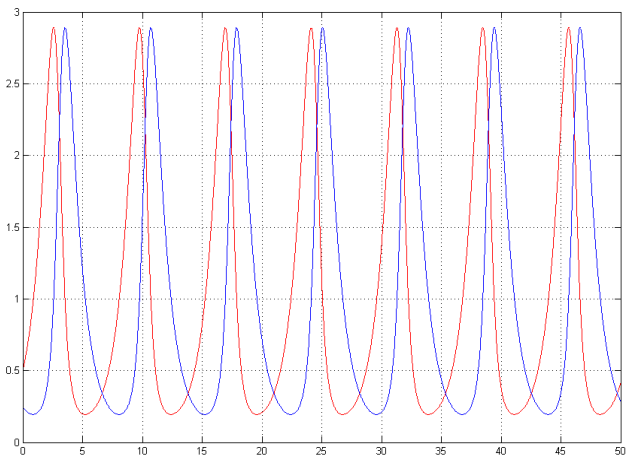
Per entendre una mica millor els avantatges que pot suposar un RK de quart ordre, davant un de segon, o fins i tot davant d'un mètode d'Euler, que en el fons podem considerar-lo com un RK de primer ordre, intentarem fer una petita comparativa entre aquests mètodes. Per posar-ne exemples treballarem amb les equacions del problema de Predador – presa (Predator-prey), seguint les equacions del model biològic de Lotka-Volterra. En l'apartat de problemes la equació serà explicada amb més detall, ara per ara, dir que segueix un model de dues equacions diferencials:

$$\frac{dx}{dt} = ax - bxy \qquad\qquad \frac{dy}{dt} = -cy - pxy$$

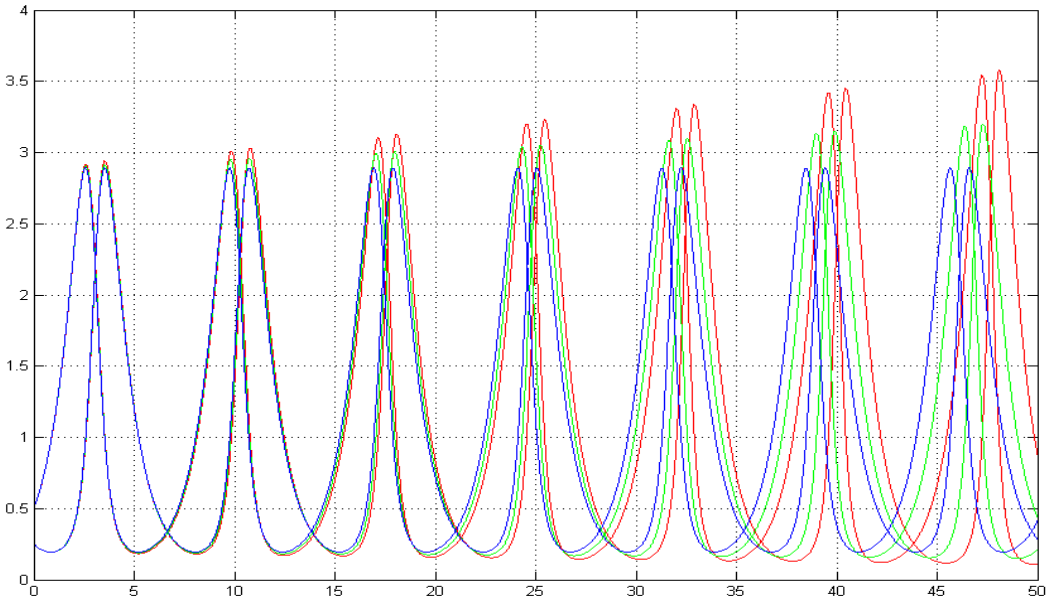
On les constants a, b, c, i p són positives.

Aquest model explica el creixement de una població respecte l'altre. En aquest cas, una població *x* que disposa d'aliment i una població *y* que disposa de *x* per alimentar-se. Pot ser perfectament la representació d'una població de conills i llops. Per resoldre el model esmentat amb un mètode d'Euler, s'han trigat 0,171 segons, fent 5000 passos i utilitzant increments de temps de l'ordre de 0,01 segons. Per resoldre el mateix problema amb ídem nombre de passos i *dt* utilitzant RK de segon ordre, l'ordinador triga 0,140 segons, i per fer-ho amb el mètode de quart ordre 0,139. Les diferències en el temps computacional no són però, pas gaire importants en aquest cas. La millora que ofereix un mètode de major ordre en el programa és la seva estabilitat i fiabilitat. Així doncs, utilitzant un mètode de baix ordre veiem que es crea un error que cada cop ens distancia més de la solució. Això significa que al fer moltes iteracions, el resultat que ens dona el programa és fals ja que no és gens precís.

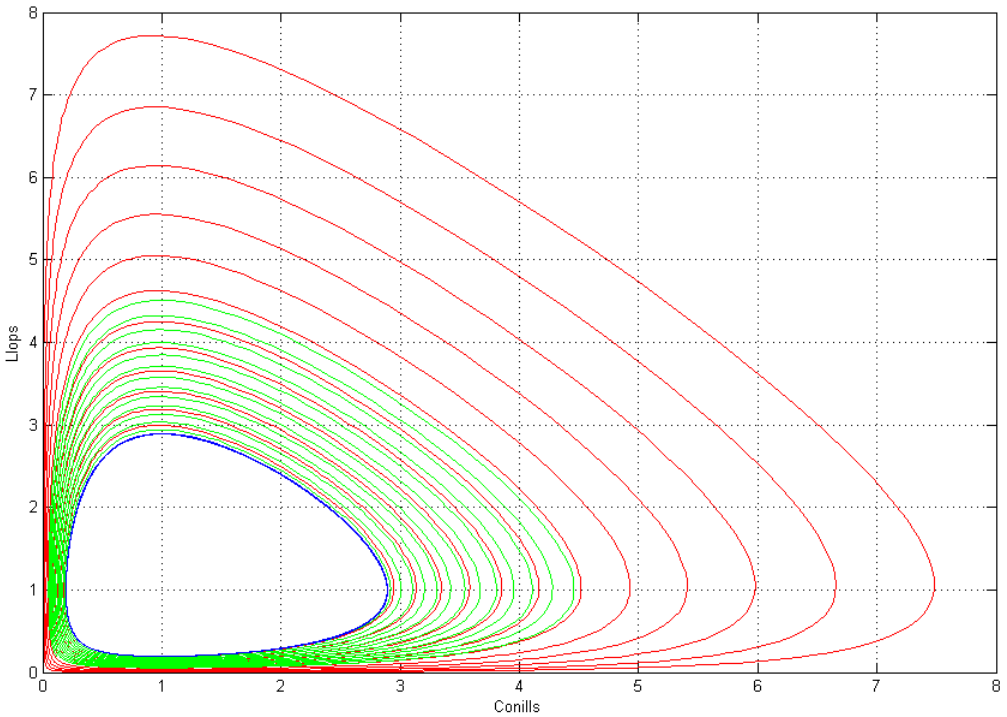
Per veure-ho més gràficament observem el dibuix. En aquest cas es una mostra de un comportament normal en el model biològic. Veiem que una població creix i decreix periòdicament al llarg del temps.



Ara bé, per veure que entenem per inestabilitat i poca fiabilitat, vegem un exemple gràfic del mateix model, amb les mateixes condicions utilitzant els mètodes d'Euler, Runge-Kutta de segon ordre i de quart ordre.



Cada parella de línies de color representen el creixement de parelles biològiques (predador – presa). En color vermell simbolitzat el mètode d'Euler, en color verd un RK de segon ordre i en color blau, RK quart ordre. Veiem clarament que no tots els mètodes resolen igual de bé les equacions, ja que hauria de sortir un sistema cíclic i no que vagi en augment. Podem dibuixar aquest fet representant una població sobre la altre i no fer-ho per separat i en funció del temps.



En el gràfic de la pàgina anterior són representats amb la mateixa llegenda de colors segons el mètode els diferents mètodes estudiats. Observem que de color blau trobem un sistema que és cíclic i estable. Aquest és el mètode de grau més alt, RK de quart ordre. Si baixem de grau en el mètode, RK de segon ordre representat en color verd, el sistema ja no és tancat i per tant no és fiable. Amb el color vermell, Euler, la diferència ja és abismal i esdevé molt inestable.

Com a conclusió no hem de dir que Euler i RK de segon ordre siguin mals mètodes. Podríem aconseguir la mateixa estabilitat que en el quart ordre, ara bé, seria necessari donar un pas de temps molt més petit i augmentar el nombre de passos. Per tant, així, podem justificar el fet d'afirmar que el mètode de Runge-Kutta de quart ordre sigui el mètode més utilitzat entre enginyers i matemàtics, ja que la relació que hi ha entre el preu computacional – resultat que ofereix el mètode és molt bona i per això és un molt bon mètode.

Només recordar que els gràfics i significats d'aquests seran explicats amb més detall a l'apartat del problema corresponent. Aquí només ens hem servit dels dibuixos per poder donar una idea gràfica de l'eficàcia i error que comet cada mètode.

Algoritme informàtic :

Com ja hem vist en el cas anterior, podem escriure els programes de diverses maneres. En aquest cas s'explicarà directament com s'escriu el programa deixant-ho referent a una funció externa. Com en l'altre cas es necessari crear un *script* on deixem escrita la funció a avaluar amb totes les variables i constants necessàries. Com que estem treballant sobre el mateix exemple, en aquest cas la funció és:

```
function [kx,ky]=f_calcula_rk(x,y,dt,a,b,c,p)

      kx=dt*(a*x-b*x*y)   ;
      ky=dt*(-c*y+p*x*y)  ;

return
```

Com abans es necessari donar la ordre de indicar que es una funció externa, on hi entren uns valors i s'utilitzen per fer el càlcul. Després aquests valors són reclamats des del programa principal. Això ho aconseguim donant l'ordre *function* i concatenant els arguments que treballarem. Així doncs, com hem vist en la part matemàtica, el funcionament es idèntic al RK de segon ordre. Definim valors inicials, vectors plens de zeros i prosseguim a donar l'ordre d'iteració i avancem el pas de temps o increment per tornar a iterar.

```
for n=1:np-1

    [k1x,k1y]=f_calcula_rk(x(n)       ,y(n)       ,dt,a,b,c,p) ;
    [k2x,k2y]=f_calcula_rk(x(n)+k1x/2,y(n)+k1y/2,dt,a,b,c,p) ;
    [k3x,k3y]=f_calcula_rk(x(n)+k2x/2,y(n)+k2y/2,dt,a,b,c,p) ;
    [k4x,k4y]=f_calcula_rk(x(n)+k3x   ,y(n)+k3y   ,dt,a,b,c,p) ;

%Avanç solució
    x(n+1)=x(n)+(k1x+2*k2x+2*k3x+k4x)/6 ; % avanç pas solució x
    y(n+1)=y(n)+(k1y+2*k2y+2*k3y+k4y)/6 ; % avanç pas solució y
    t(n+1)=t(n)+dt                      ; % avança pas de temps

end
```

El funcionament és exactament igual que abans, però veiem que utilitzem des de *k1* fins a *k4* i després al fer-ne la mitjana també canvia l'algoritme. La resta de funcionament del programa és idèntic al de segon ordre.

Per veure l'algoritme complet de cada programa podem fer referència a l'annex, ja que en l'explicació dels algoritmes referents als mètodes només es mostra la part que correspon explícitament al mètode.

EL PER QUÈ DELS PROBLEMES

Per trobar una utilitat pràctica a aquests mètodes i teoremes que hem vist farem l'estudi de diversos problemes. Per tant doncs, portarem a la pràctica el que fins ara hem vist com a cos teòric.

Quan diem que farem uns problemes volem dir que n'estudiarem el model físic i el desenvoluparem matemàticament per arribar a trobar l'equació a resoldre. Així doncs, podrem aplicar els mètodes que hem estudiat i trobar una utilitat a trobar zeros de funcions o resoldre equacions diferencials.

Els problemes són situacions plantejades on ja es coneix el model a resoldre. En el cas que no coneguéssim aquest model podríem trobar-lo a través de tècniques experimentals, cosa que seria més complicat i necessita més recursos. Ara bé, si podem trobar els models de cada problema, podem centrar-nos més a solucionar-lo, i també a la part de programació que hi fa referència. Com veurem, solucionar models informàticament és una molt bona eina per fer estudis sobre sistemes, ja que canviant les variables que intervenen en cada cas ens permet fer una predicció de resultats amb un simple clic de ratolí, no es necessari fer-ho part experimentalment. Aquesta filosofia, de preveure amb la modelització informàtica és un camp molt utilitzat en ciències, d'aquesta forma podríem estalviar temps i recursos.

PROBLEMA: CAIGUDA LLIURE

Un Super-agent del servei secret del GFS té problemes. Ha de saltar de l'avió on va per culpa de l'avaria d'un motor i vol saber alguns aspectes que per ell, ara per ara, són pràctics. És l'encarregat de transportar unes càpsules de plutoni, cosa que no es de gaire confiança ja que és altament tòxic i radioactiu. L'aspecte que més el preocupa de saltar és la velocitat màxima que pugui assolir, si haurà de suportar una estrebada molt forta a l'obrir el paracaigudes i sobretot el temps de vol que haurà d'aguantar el plutoni simplement arrapat al seu cos sense una protecció especial. Per sort, el nostre Super-agent disposa d'un bon ordinador i uns quants llibres de física i matemàtiques al seu abast, i es disposa a barrinar una mica, ja que la feina dels Super-agents no només és enredar xicotes maques i conduir cotxes cars. Per tant ens disposem a salvar el món de una catàstrofe i resoldre un problema de caiguda lliure.

Resolució:

Així doncs, procedim a l'explicació del problema que s'ha plantejat; la caiguda lliure. Abans però, marcar uns objectius que intentarem assolir amb la resolució del problema.

- Com varia l'altura en funció del temps transcorregut?
- Quan de temps trigarem a arribar al terra?
- Quina velocitat assolirem? I a l'arribar al terra?
- Quantes "g" haurem de suportar?

Un cop establerts els nostres objectius procedirem a treballar el model físic que resol el nostre problema. En la caiguda lliure hem de remetre'ns al camp de la dinàmica, fent referència a la segona llei de Newton:

$$\sum F = m * a \quad \rightarrow \quad -mg + F_f = m * a$$

Sabem que el sumatori de forces equival al producte de la massa per l'acceleració. Així doncs en el nostre cas tenim una força que és produïda pel camp gravitatori sobre el cos que cau, *-mg*, i una força en direcció contrària que és la força de fregament, F_f, que es genera per la diferència de velocitat entre el paracaigudista que cau i l'aire. Aquesta força de fregament es pot calcular a través d'un coeficient de fricció, C_d, que es defineix com:

$$C_d = \frac{\frac{F_f}{A}}{\frac{1}{2}\rho V^2}$$

Justament, per el cas del paracaigudista, es pot prendre com el valor de C_d=1,4 (White, 1994). Com veiem en el càlcul de la força de fricció hi intervenen variables com la velocitat de caiguda del paracaigudista, *V*, l'àrea frontal del paracaigudes, *A*, i la densitat, ρ, del fluid, en aquest cas aire. Així podem expressar la F_f com:

$$F_f = Cd * A\frac{\rho * V^2}{2}$$

Substituint aquesta expressió en la segona llei de Newton i aïllant el valor de l'acceleració obtenim:

$$a = -g + \frac{C_d * \rho}{2m} * A * V^2$$

El model matemàtic es completa escrivint les equacions del moviment:

$$\frac{dy}{dt} = V \qquad \frac{dV}{dt} = a = -g + \frac{C_d * \rho}{2m} * A * V^2$$

Aquest és un sistema de dues equacions diferencials de primer ordre on la variable independent és el temps i les dues variables dependents són la posició, *y*, i la velocitat, *V*. Aquest sistema el resoldrem utilitzant el mètode de Runge-Kutta de quart ordre. Ara bé, cal dir que abans d'arribar a utilitzar aquest mètode també podem passar per Euler i Runge-Kutta de segon ordre. Per tant, les equacions per tal de fer els càlculs dels increments en funció el temps són:

$$\frac{dy}{dt} \lim_{\Delta \to 0} \approx \frac{\Delta y}{\Delta t} \quad \rightarrow \quad \Delta y = \Delta t * V$$

$$\frac{dV}{dt} \lim_{\Delta \to 0} \approx \frac{\Delta V}{\Delta t} \quad \rightarrow \quad \Delta V = \Delta t * (-g + \frac{C_d * \rho}{2m} * A * V^2)$$

Solució computacional:

Tornant a referir-nos al nostre heroi ara que ja sap quines equacions ha d'integrar i amb quin mètode ha de resoldre-les, començarem a escriure el programa que ens ajudarà a respondre les inquietants preguntes. Recordem que tots els mètodes per integrar equacions diferencials han de compondre's d'uns valors inicials, vectors lliures de números i el "cervell" bàsic per la integració de les equacions que utilitzi un mètode adequat.

Com que no podem cometre gens d'error i volem ser molt precisos utilitzarem el mètode de Runge-Kutta de quart ordre. No cal despreciar però els altres mètodes, ja que utilitzant un pas de temps petit i fent moltes iteracions podem arribar a ser igual de precisos.

Per a la realització del programa procedim en primer pas a la definició de dades i variables que anirem utilitzant. Un dels avantatges de la programació és la capacitat de definició de dades i relacionar-les amb valors. D'aquesta forma quan ens remetem a aquests no cal entrar els números de la dada sinó que podem utilitzar la definició que hi hem donat. En aquest requadre en tenim un exemple. Per calcular la densitat no cal entrar l'operació amb els números sinó que podem fer-ho amb les paraules que nosaltres mateixos hem definit com a valors. D'aquesta forma, la programació es torna més intuïtiva i no tant feixuga.

```
Massa=200              ;
y0=2000                ;
radi=3                 ;
volum=4/3*pi*radi^3    ;
d=massa/volum          ;
```

```
t(1)=0          ;
y(1)=y0         ;
v(1)=v0         ;
a(1)=g          ;
ar(1)=area1     ;
```

Una vegada donades les dades inicials com l'altura, la massa, l'altura d'obertura, l'àrea del paracaigudes i de la persona, la gravetat, el coeficient de fregament, etc. S'han de definir uns vectors de la mida del nombre de passos totals i posteriorment omplir el primer pas de cada vector amb les dades inicials. Per referir-nos-hi només cal indicar-ho com es veu a l'exemple. La posició del primer pas, *v(1)* té el valor de la posició inicial que havíem definit, *v0*. Llavors *y(1)=y0* i consecutivament en cada variable.

Com a següent pas ja podem introduir el codi que fa referència al mètode que utilitzem. En aquest cas de la caiguda lliure hem de determinar una ordre per variar l'àrea abans i després de l'obertura del paracaigudes, ja que el que possibilitarà arribar a terra i sobreviure a l'experiència serà tenir una àrea suficient com per aconseguir un valor de fricció prou gran com perquè la velocitat en el moment final sigui suportable pel cos humà. Per aconseguir aquesta obertura de paracaigudes hem determinat una alçada d'obertura que va en funció de l'alçada inicial. En aquest cas s'ha agafat un criteri de 2/6 de l'espai recorregut total i fent el procés d'obertura en un desplaçament de 180 metres, ja que si obrim de cop el paracaigudes, la força de fregament augmenta sobtadament i produeix una força de fricció tant gran que destrossaria el paracaigudista.

Els valors agafats han estat triats a partir de prova i ajust amb el programa ja fet, comprovant que l'acceleració resultant era un valor suportable pel cos humà.

Això podríem transcriure-ho a Matlab així.

```
for n=1:np-1
    if y(n)>yobre1
        area=area1 ;
    elseif y(n)<=yobre1 & y(n)>yobre2
        area=((y(n)-yobre1)/(yobre2-yobre1))*(area2-area1)+area1;
    else
        area=area2 ;
    end
    ar(n+1)=area;
```

En aquest tros de codi ordenem el canvi d'àrees, per tant, en realitat el que fem es procedir a l'obertura del paracaigudes. Primer dient amb un condicional, *if*, si la posició de *y* és per sobre la determinada, l'àrea és la de la persona. Després, amb un segon condicional, *elseif*, si l'alçada és igual o inferior al valor establert d'obertura, però és major que el segon valor de *y* d'obertura total, que és una diferència de 180 metres, procedeix a l'obertura dient que l'àrea va en funció de l'alçada. Finalment, utilitzant un tercer condicional, *else*, quan l'altura és inferior al segon valor de *y*, llavors utilitza tota l'àrea del paracaigudes. Aquest fet a la realitat potser no es dona ben bé així, però n'és una bona aproximació i serveix perquè ens surtin valors creïbles que pot suportar un cos humà.

Un cop arranjada l'obertura del paracaigudes podem escriure el mètode que ens resol el model plantejat.

```
%Runge-Kutta

[k1y,k1v]=f_calcula_rk(y(n)      ,v(n)      ,g,dt,cd,area,massa,d_aire) ;
[k2y,k2v]=f_calcula_rk(y(n)+k1y/2,v(n)+k1v/2,g,dt,cd,area,massa,d_aire) ;
[k3y,k3v]=f_calcula_rk(y(n)+k2y/2,v(n)+k2v/2,g,dt,cd,area,massa,d_aire) ;
[k4y,k4v]=f_calcula_rk(y(n)+k3y  ,v(n)+k3v  ,g,dt,cd,area,massa,d_aire) ;

%Avanç solucio

y(n+1)=y(n)+(k1y+2*k2y+2*k3y+k4y)/6 ;
v(n+1)=v(n)+(k1v+2*k2v+2*k3v+k4v)/6 ;
a(n+1)=(v(n+1)-v(n))/dt             ;
t(n+1)=t(n)+dt                      ;
 if y(n+1)<=0
    break
 end
end
```

Tota l'ordre, des del control d'obertura fins l'ultima ordre de parar d'iterar en arribar al terra és dins una ordre *for...end* que farà iteracions fins a aconseguir un valor de *y* igual o més petit que zero. Això voldrà dir que ja hem arribat a terra.
Com a pas final hem de tallar els vectors restants i escriure el codi per realitzar els dibuixos. En aquest cas com que és un model on el moviment és molt important i és bo veure-ho, s'ha dibuixat el gràfic de forma animada i que ens permeti resoldre els dubtes que es plantejava el nostre Super-agent.

Aplicacions:

Una vegada quedi escrit el programa, la GFS quedarà com a una agència molt eficaç, amb uns empleats eficients. En aquest cas el seu agent ha hagut de tirar-se des de una alçada inicial de 2000 metres i la seva massa, més el paracaigudes i les càpsules de plutoni feien un còmput total de 200 kg. El seu radi és de 1 metre i el del paracaigudes de 4. Ara el que ha de fer el personatge és anar al programa que ha creat, i introduir com a *y0=2000* i com a *massa=200*. Les dades que obté són:

- Temps trigat = 149,1 s
- Velocitat màxima = 52,5 m/s
- Velocitat terminal = 7,36 m/s
- Acceleració màxima/g = 3,5 G

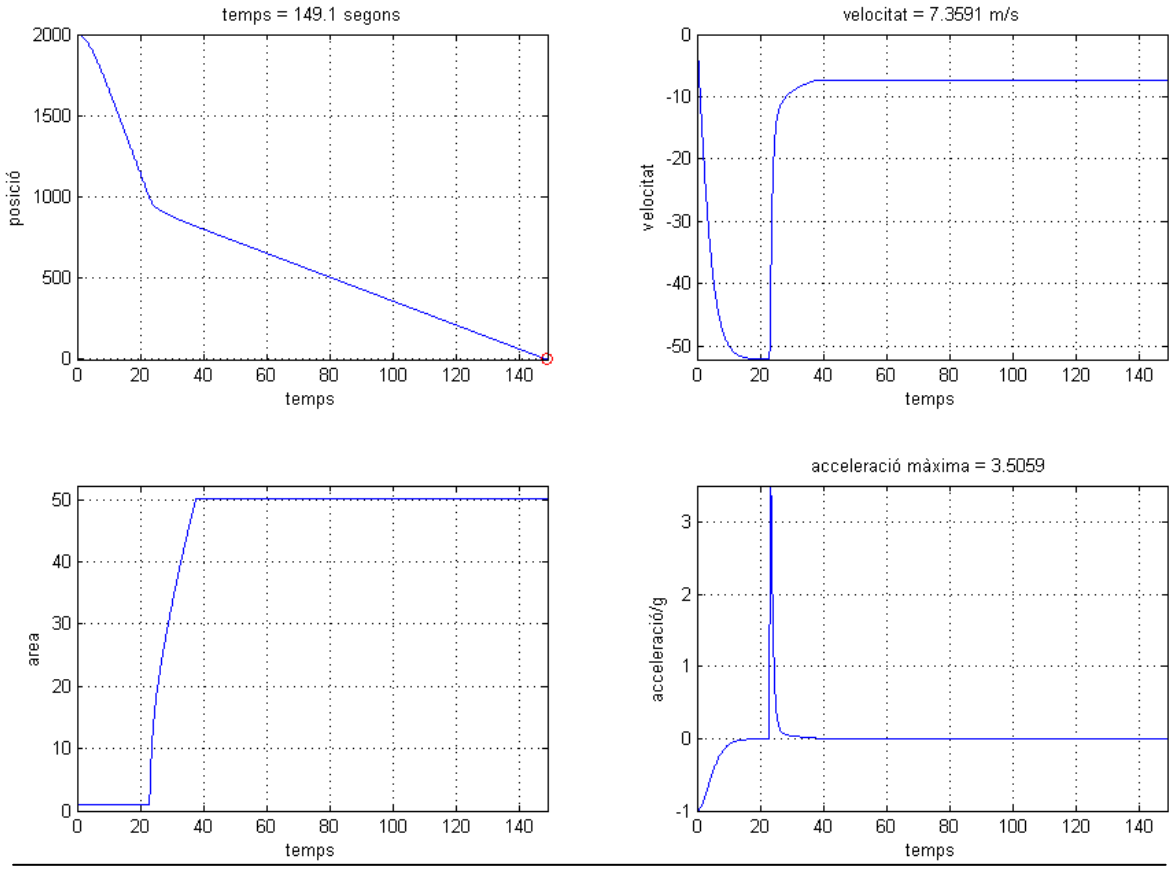
L'acceleració s'ha adimensionalitzat dividint-la pel valor de la gravetat, g=9,8 m/s^2, per tant el valor d'acceleració adimensional màxima de 3,5 G correspon a un valor absolut de 3,5*9,8 =34,3 m/s^2. Per justificar aquest fet anem al camp de la biologia. Mentre que no hi ha un límit de velocitat que pugui suportar el cos, en l'acceleració si que existeix. Els òrgans tous interns, com els ulls, en valors massa elevats d'acceleració pateixen lesions i poden trencar-se o rebentar. Així podem fer-nos una idea si ens es factible saltar amb paracaigudes o buscar un pla alternatiu. Segons estudis realitzats a la NASA (Wikipedia, G-force) una persona normal perd el coneixement al superar les 5 G, malgrat que amb entrenament, com els pilots d'avions supersònics o els Super-agents, podem suportar valors de fins a 9 G sense conseqüències.

Com a aplicació final veurem el dibuix de la simulació. En aquest cas crearem una graella de 2 * 2, de forma que farem quatre gràfics. En un hi representarem la posició vers el temps, en una altre la velocitat en funció del temps, la tercera contindrà l'acceleració/g i l'ultima hi representarem l'àrea de cada moment. Per fer aquestes representacions utilitzarem les eines de dibuix del Matlab, on podem donar les ordres

escrivint les línies de codi corresponent de forma adequada. L'ordre per excel·lència de dibuixar amb Matlab és *plot*. En aquest cas però, per construirem la graella dels gràfics utilitzarem l'ordre *subplot(x,y,posició)*.

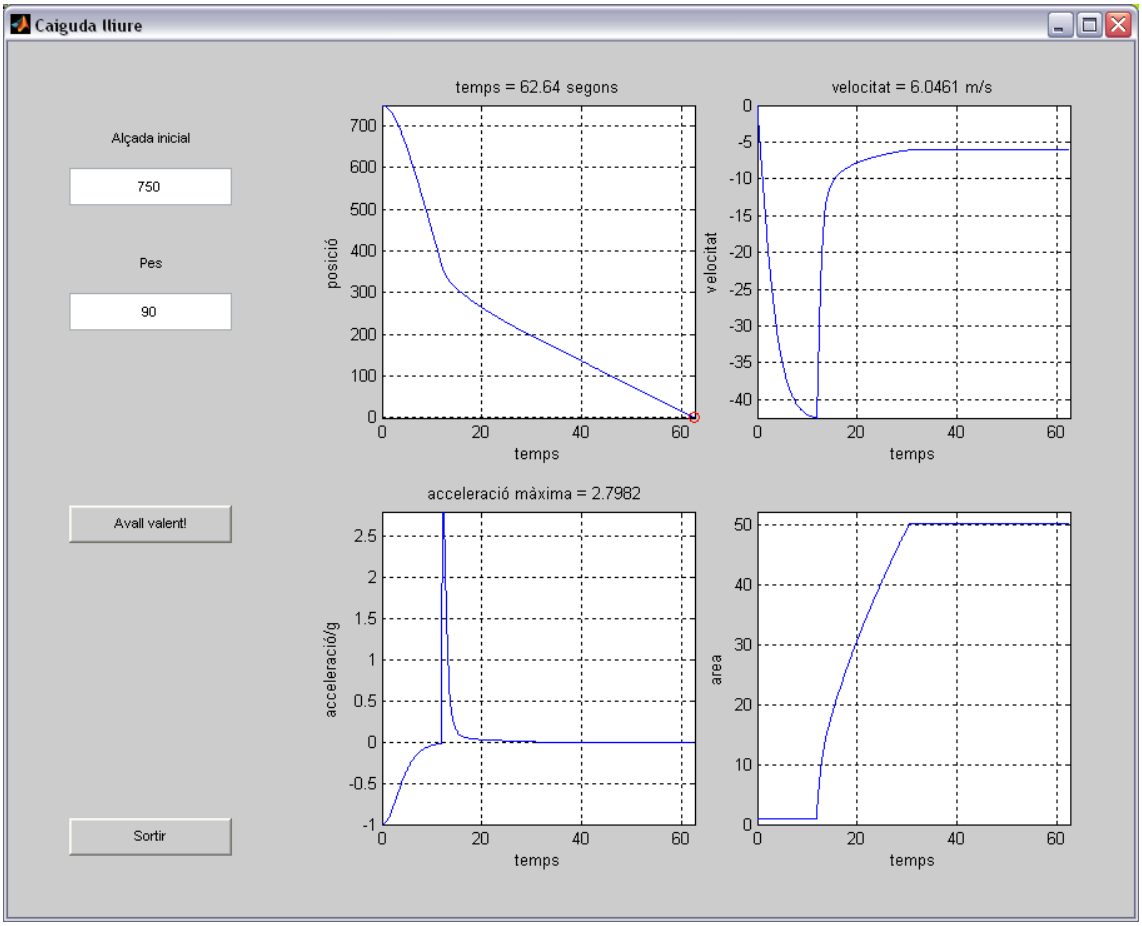
```
for nplot=1:10:ntot
    subplot(2,2,1);
        plot(t(1:nplot),y(1:nplot),'-b',t(nplot),y(nplot),'or') ;
        xlabel('temps') ; ylabel('posició');
        title(['temps = ',num2str(t(ntot)),' segons'])
        axis([t(1),t(ntot),min(y),max(y)]) ; grid ;
    subplot(2,2,2);
        plot(t(1:nplot),v(1:nplot)) ; xlabel('temps') ; ylabel('velocitat');
        title(['velocitat = ',num2str(abs(v(nplot))),' m/s'])
        axis([t(1),t(ntot),min(v),max(v)]) ; grid ;
    subplot(2,2,4);
        plot(t(1:nplot),a(1:nplot)/abs(g)) ; xlabel('temps') ;
        ylabel('acceleració/g');
        title(['acceleració màxima = ',num2str(max(abs(a/abs(g)))),' '])
        axis([t(1),t(ntot),min(a/abs(g)),max(a/abs(g))]) ; grid ;
    subplot(2,2,3);
        plot(t(1:nplot),ar(1:nplot)) ; xlabel('temps') ; ylabel('area');
        axis([t(1),t(ntot),0,max(ar+2)]) ; grid ;
        drawnow
end
```

Donant aquesta ordre manem que faci una graella de tants requadres per *x*, tants per y i al final indiquem quin volem omplir. Després manem el dibuix de cada una de les variables que volem, donant primer les dades de l'eix *x* , després *y* i al final ordres de color o mida. Prèviament per fer això les dades han de ser guardades, però en aquest cas com que ja ho tenim fet no suposarà cap problema. Finalment el resultat és:



Si a més a més volem que sembli una animació i fer un gràfic que tingui moviment, farem un gràfic diferent. Manarem fer un dibuix cada deu iteracions, per abaratir el cost en memòria, llavors a cada dibuix l'hi posarem a sobre l'anterior, així el resultat final serà un gràfic animat.

Per complementar aquest problema a més a més s'ha creat un programa que es pugui executar independentment de Matlab. Així, s'ha compilat el codi que estava escrit en llenguatge *M* i s'ha convertit a llenguatge *C*, present a tots els ordinadors. El resultat final és un programa amb caràtula (o també dit GUI) pròpia que ens deixa triar l'alçada inicial i la massa.



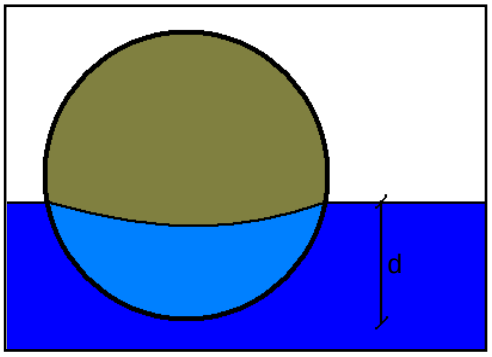
Per veure el codi que fa la caràtula i el funcionament de tot el programa podem fer referència als annexos i fitxers adjunts al treball.

PROBLEMA: LA PILOTA FLOTANT

El nostre Super-agent un cop més, necessita l'ajut de les matemàtiques i la informàtica! Aquest cop la GFS vol infiltrar-lo en uns laboratoris secrets vora la mar Secreta i necessita que el nostre personatge hi arribi sense ser vist, així doncs la possibilitat de tirar-se amb paracaigudes és descartada. Per fer-ho, amb el programa ja fet, calculen si seria possible tirar-lo des d'un avió dins una esfera, plena d'aire per dintre. Al veure que és possible decideixen enviar a l'agent i tot el seu material secret, que no us diré què és perquè sinó no seria secret, dintre l'esfera inflable. Així podria ser tirat sobre el mar i sobreviure, sense necessitat del paracaigudes. El problema és que l'esfera no pot estar coberta més de una determinada alçada, sinó el sistema de seguretat la faria explotar a l'instant. Aquesta alçada és la meitat del diàmetre total. Tirar el Super-agent i comprovar fins on arribaria l'aigua, seria divertit, però és més pràctic fer una mica de recerca i simular-ho computacionalment.

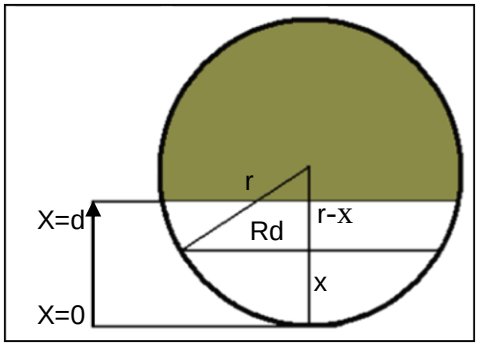
Resolució:

En aquest problema el que volem esbrinar és l'alçada que sobresortirà una esfera que està flotant en un fluid. Així doncs, primer de tot hem de descobrir quina serà la equació que voldrem resoldre.



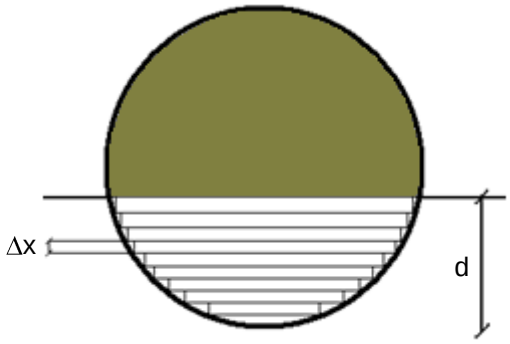
El problema es regeix pel principi d'Arquimedes. Aquest diu que tot cos submergit en un fluid experimenta una força de flotació, l'empenta E, igual al pes del volum del líquid desplaçat. En el moment en que l'esfera arriba un punt d'equilibri, per tant no s'enfonsa més, llavors es quan el pes del cos equival a l'empenta. Llavors el sumatori de forces equival a zero.

$$\sum F = 0 \qquad E = m_{pilota} * g$$



Un cop vist el principi d'Arquimedes podem prosseguir a entendre el model. El tros o alçada del volum submergit, d, podem calcular-lo com la suma del volum de varis cilindres petits, de radi *Rd* i alçada Δx. Fent us de Pitàgores,

$$R_d{}^2 + (r-x)^2 = r^2 \;\rightarrow\; R_d{}^2 = r^2 - (r-x)^2$$



i el volum de cada cilindre elemental és:

$$\pi(r^2 - (r-x)^2) * \Delta x$$

Si fem la suma d'infinits volums elementals, fent que *Δx* tendeixi a zero trobem que:

$$\int_{x=0}^{x=d} \pi\,(r^2 - (r-x)^2) * dx = V_d$$

Així podem esbrinar el volum que està sota l'aigua, *Vd*. Ara bé, vegem com progressa la resolució de la nostre integral...

$$V_d = \pi \int_0^d (r^2 - (r^2 - 2rx + x)^2)dx \quad = \quad \pi \int_0^d (2rx - x^2)dx = \pi \left[2r\frac{x^2}{2} - \frac{x^3}{3} \right]_0^d \quad = \cdots$$

$$= \pi \left(rd^2 - \frac{d^3}{3} \right) - (0 - 0) = \pi \left(rd^2 - \frac{d^3}{3} \right) = \pi \left(\frac{3rd^2 - d^3}{3} \right) = \pi d^2 \frac{3r - d}{3}$$

Un cop hem deduït quin serà el volum submergit en funció de *d*, podem saber la força de flotació *E*. També, segons l'equació del volum d'una esfera, podem saber el pes de la pilota, que es l'espai ocupat per la densitat i per la gravetat. Si portem el sistema a un estat d'equilibri llavors es compleix que:

$$E = \pi d^2 \frac{3r - d}{3} * \rho_{liquid} * g \qquad P_{pilota} = \frac{4}{3}\pi\, r^3\, \rho_{pilota} * g$$

$$\pi d^2 \frac{3r - d}{3} * \rho_{liquid} * g = \frac{4}{3}\pi\, r^3\, \rho_{pilota} * g$$

Finalment, igualant l'equació a zero i deixant-ho en funció de la densitat relativa de la pilota, $\rho = \rho_{pilota/}\rho_{líquid}$, trobem l'equació que ens interessa resoldre. En aquest cas, només hem de treballar amb una densitat, que és la ρ relativa de l'objecte flotant. Si treballem en unitats del SI i el líquid és aigua, podem trobar la densitat relativa dividint el producte de la massa pel volum de l'esfera per 1000 kg/m^3. Per tant aconseguim simplificar l'equació.

$$\pi \frac{(d^3 - 3d^2r + 4r^2\rho)}{3} = 0$$

Així doncs, hem de resoldre una equació on volem trobar el valor de *d*, que és l'espai submergit i coneixem *r* que és el radi de la pilota i la seva densitat ρ. Com podem observar, tenim una equació de tercer grau, això ens dona pistes que segurament hi pot haver tres solucions, encara que en el nostre cas només una seria la correcte.

Solució computacional:

Com bé haureu vist, en aquest cas no hem de resoldre cap equació diferencial, només estem buscant el zero d'una funció polinòmica. Per tant, podem utilitzar qualsevol dels mètodes vistos; mètode de la bisecció, mètode d'interpolació lineal o bé mètode de Newton. En aquest cas, farem el problema deixant l'equació d'aquest com a una funció externa, cosa que ens farà molt fàcil passar d'un mètode a l'altre si volem.

El primer pas que hem de fer, com en tots els casos, és definir valors inicials. Donar uns primers valors de *a* i de *b*, juntament amb valors necessaris pel problema .El radi i la densitat de l'objecte són els que podrem variar depenent del cas en que ens trobem. Una vegada donats els valors inicials podem prosseguir a escriure el tros de codi encarregat de fer les iteracions amb el mètode que vulguem utilitzar. Recordem que en aquest tipus de problemes no cal definir uns vectors, ja que no treballem sobre nombres passats. Ara bé, si ens interessa poder fer un dibuix i marcar-hi les diferents iteracions que hem fet, és necessari fer-ho. També és necessari donar un valor de tolerància, que serà la aproximació màxima que farà el nostre programa. Una vegada els nous valors aconseguits siguin més petits que la tolerància, deixarem d'iterar. Llavors la tolerància condiciona com d'acurat és el nostre programa, s'ha de trobar un equilibri entre una tolerància prou bona per obtenir resultats fiables i no fer que el programa iteri fins a l'eternitat. Així doncs, en un primer tros de codi podem veure la definició de les dades i la creació d'espais per guardar els valors.

```
radi=0.8 ;
m=200 ;
v=(4/3)*pi*(r^3) ;
d=(m/v)/1000 ;
a=0 ;      fa=funcio_2(a,r,d) ;
b=2 ;      fb=funcio_2(b,r,d) ;
x=linspace(a,b,100) ;
y=funcio_2(x,r,d) ;
tol=1e-6 ;
```

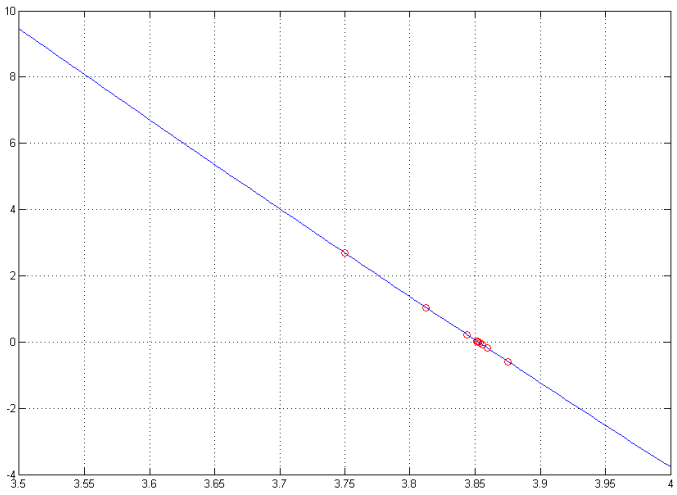
Un cop això, podem prosseguir a escriure estrictament el tros del mètode que resol el problema. En aquest cas utilitzarem un condicional *While...end*. L'ús d'aquestes ordres en ves d'un *for...end* és que assegurem que el programa no entri en un *loop* infinit d'iteracions i quedi l'ordinador i el programa penjat. La seva explicació és deguda a que una ordre *while* sempre va seguida d'un estament de condició. Si aquesta condició no es compleix el programa deixa de funcionar. Així doncs, definim una ordre d'iterar

quant els valors de *a*, *b* i *c* siguin majors que la tolerància definida i així ens assegurem un bon funcionament del programa. Després d'escriure tota la part que fa referència al mètode podem fer, com en apartats anteriors, que dibuixi la funció i que hi pinti les iteracions que s'han fet. No és estrictament necessari, però donar aplicació gràfica als problemes ens ajuda a entendre el procés de resolució que hi apliquem.

Aplicacions:

Sortosament el servei de la GFS té un bon equip de matemàtics i entesos en física. Així han pogut portar a terme el desenvolupament d'un programa que els hi permet saber l'alçada que sobresurt una pilota només indicant-ne el radi i la seva densitat. Una altre variable que s'ha de tenir en compte és que l'equació final tal i com s'ha deixat escrita només funciona en els líquids de densitat com l'aigua, sinó caldria arreglar la formula deixant-ho en funció de les densitats de l'objecte flotant i del fluid sobre el qual s'aguanta.

El nostre Super-agent es tirarà dintre una pilota de 0.8 metres de radi, amb una densitat relativa de 0,01. Aquesta densitat l'hem pogut calcular sabent el volum de l'esfera dividit per la massa de l'agent més la pilota i tot el material secret, que tampoc us puc dir quin percentatge del pes total és, ja que potser esbrinaríeu què és. Per aconseguir que la pilota floti hem de fer que la densitat sigui menor de 1, ja que aquesta és la densitat de l'aigua i si la superés, l'objecte s'enfonsaria. Així, amb una massa total de 200 Kg hem de tirar el Super-agent dintre una pilota de 0,8 metres de radi. El problema que suposa aquest llançament és saber si quedarà l'espai necessari per obrir l'esfera sense problemes. En aquest cas hem d'assegurar-nos que l'aigua no passi de la meitat de l'alçada total de l'esfera, sinó no es podria obrir.



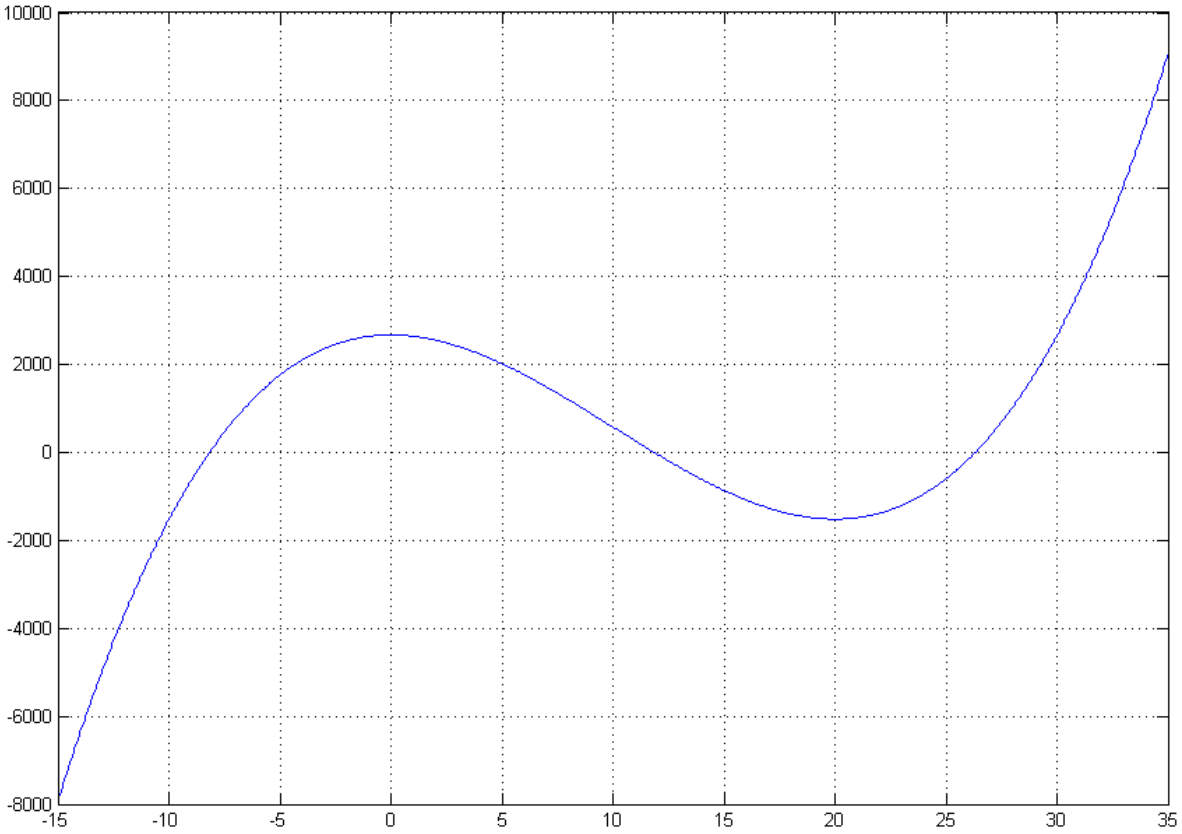
Utilitzant el nostre programa, trobem que amb les condicions establertes, l'aigua arribarà fins a una alçada de 0.3 metres, així que el principi de la nostre operació serà un èxit.

Com podem veure gràficament, el programa realitza iteracions fins que al final acaba trobant una aproximació de l'ordre de la nostre tolerància. Així doncs, els mètodes numèrics no donen valors "reals" però s'hi aproximen tant que els hi podem considerar.

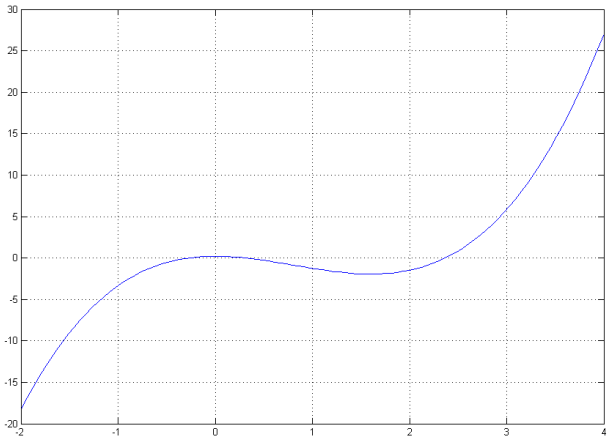
Extensió:

El servei d'intel·ligència de la GFS ha tingut algun problema a l'hora de trobar la solució correcte. El problema que se'ls hi plantejava era que el programa no sempre funcionava si es determinava un interval inicial massa gros. La explicació d'això és molt fàcil. Estem operant sobre una equació de tercer grau, això vol dir que hi haurà més d'una solució i al funcionar el programa aquest pot convergir a qualsevol de les tres possibles solucions. Així doncs, intentarem trobar les tres solucions del problema, ara bé, no totes tres tindran un sentit físic.

Si variem les condicions del problema, per veure un bon exemple, i definim dades de r=10 metres i una densitat relativa de ρ=0.638 (Mathews i Fink, 2000) començant amb un valor de a=-15 i un valor de b=35, observem que la gràfica descriu una trajectòria així.



En el cas del nostre problema la diferència no era tant extrema, per això s'han utilitzat aquests valors per fer l'explicació i les representacions gràfiques. Si haguéssim utilitzat els valors del problema tindríem el mateix fenomen però no de forma tant diferenciada. Vegem-ho amb la representació gràfica, que per



apreciar les tres arrels hem de treballar en un interval prou petit (-2,4) com perquè sigui visible a simple vista. Així doncs, per poder posar un bon exemple s'han utilitzat els altres valors.

Podem observar clarament que hi ha tres interseccions amb l'eix *OX*, per tant, això vol dir que tenim tres solucions de l'equació. Si volem trobar-les totes tres, hem de definir tres intervals de treball, complint les condicions necessàries per poder iterar en cada una. Això ho aconseguirem fent la concatenació de diversos valors. L'únic que volem dir amb això és que per molt que escrivim els números de *a* i *b*, en aquest cas *x11* i *x22* seguits, són parelles per separat. També podríem definir els tres intervals de forma independent, però després es molt més còmode treballar així, perquè donant la mateixa ordre i indicant si és la primera, la segona o la tercera posició n'hi ha prou. Per tant, després això ho hem d'indicar en el cervell del programa que fa les iteracions. Només caldrà indicar amb un condicional quina parella és al principi de cada ordre. Per fer això utilitzarem el condicional per excel·lència, l'ordre *if...end*, on s'escriu l'estament just després del condicional i es tanca al haver escrit tota l'ordre desitjada.

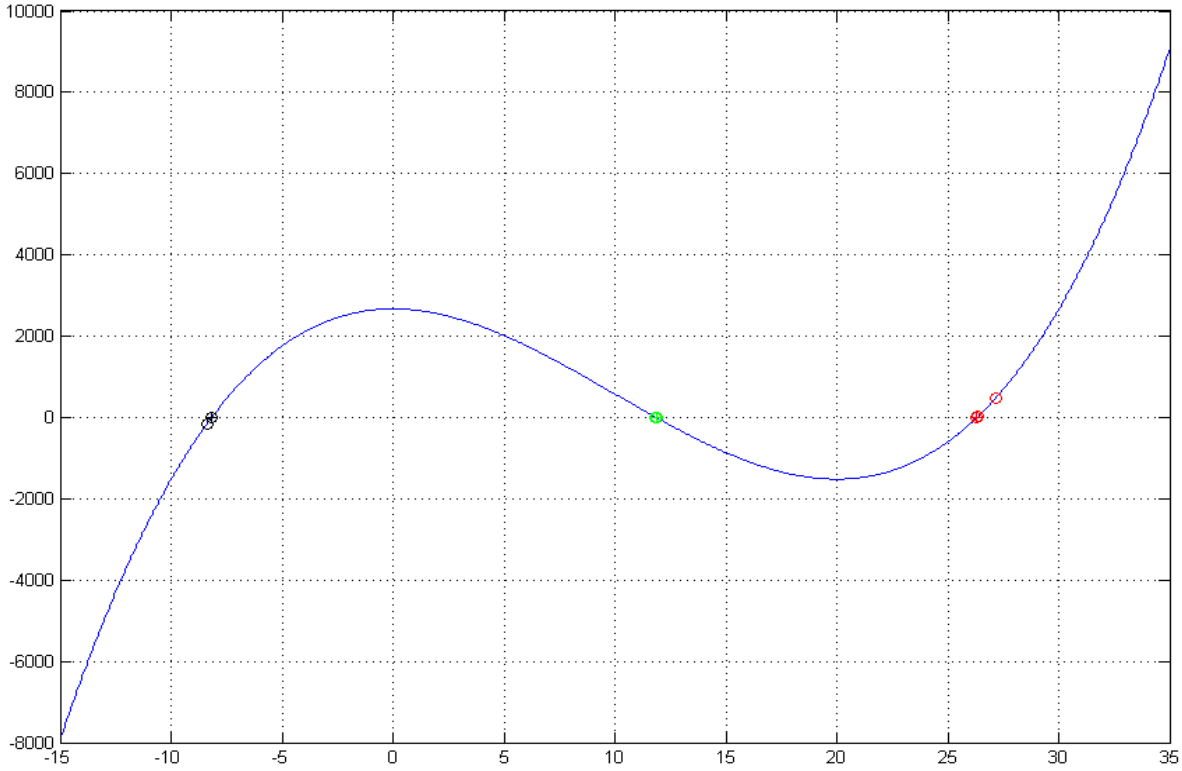
```
x11=[-15,10,25] ;
x22=[-5 ,15,35] ;
```

```
if n==1 ; plot(x2,fx2,'ok') ; end
if n==2 ; plot(x2,fx2,'og') ; end
if n==3 ; plot(x2,fx2,'or') ; end
```

Després podrem representar tots els valors, però també ho haurem de fer per separat. Indicant l'ordre de dibuix farem la mateixa jugada indicant davant de l'ordre si és el primer, segon o tercer cas.

```
if n==1 ; plot(x2,fx2,'+k') ; end
if n==2 ; plot(x2,fx2,'+g') ; end
if n==3 ; plot(x2,fx2,'+r') ; end
```

Per fer-ho més visual, podem utilitzar diversos colors en cada paquet d'iteracions per diferenciar-ho. El resultat final del gràfic queda:



Com hem dit abans, no totes les solucions tenen un sentit físic. De les tres aconseguides, en trobem una de negativa. Podem descartar-la ràpidament, ja que no serà possible tenir un valor negatiu de l'alçada d'una esfera. Com a segon valor, tenim un numero pròxim a 11,8. Recordant que teníem una esfera de 20 metres de diàmetre, aquesta solució podria ser considerada real. Finalment, obtenim un tercer valor que excedeix el diàmetre total de l'esfera, per tant, també queda descartat com a valor real.

CONCLUSIONS

Després d'haver passat un bon grapat d'hores entre llibres i pantalles d'ordinador, valoro el treball positivament, ja que ha cobert les seves expectatives i n'ha donat de noves. La recerca de mètodes per poder resoldre els models físics i biològics ha estat satisfactòria. Com s'ha vist hem trobat més d'un mètode per poder resoldre un mateix tipus de problema, i cal dir que en molts dels llibres consultats, aquests mètodes només eren presentats com els més simples, potser per això també els més utilitzats. No cal entendre com a simple el més dolent, ja que hem vist que amb petites bases podem obtenir bons resultats. En el món dels mètodes numèrics tenim eines molt potents per resoldre qualsevol problema que se'ns presenti, i si això hi sumem la programació, fem una bona combinació perquè cap incògnita és resisteixi.

La part positiva de la modelització informàtica és que a part de trobar solucions concretes, també permet jugar amb els valors i fer estimacions de resultats amb noves condicions sense haver de realitzar cap prova física, biològica, química, etc. Tot això podem aconseguir-ho amb els llenguatges de programació existents avui en dia, en aquest cas hem utilitzat el Matlab, un descendent del que és el llenguatge informàtic de les matemàtiques, el *Fortran*. Podem donar com a altre punt satisfactòriament assolit, la coneixença del Matlab i el seu entorn de treball. S'han adquirit les nocions principals de programació i part de les ordres bàsiques de la programació. A diferència del principi d'aquest treball, ara puc resoldre numèricament equacions, trobar-ne la solució i integrar equacions diferencials, entenent el seu desenvolupament matemàtic i sabent-ne fer un programa que hi escaigui.

Com a conclusió final afegiré que després de la realització d'un executable d'un programa, creant la caràtula i botons d'interacció recomano que no es faci si no és d'extrema necessitat. Matlab és un llenguatge que va molt bé per programar i treballar amb els números, però alhora de realitzar caràtules i la part visual dels programes és més còmode treballar amb altres llenguatges de programació. Si es vol fer, és una feina que pren molt de temps, encara que els resultats siguin satisfactoris. Al cap i a la fi, el funcionament intern del programa és el necessari per fer l'estudi de qualsevol model i podem treballar qualsevol problema sense la necessitat de fer-ho amb boniques caràtules.

Per altra banda, Matlab és una molt bona eina pel processat i edició de gràfics.

PROJECCIÓ

Com a continuació del treball podem donar-hi diferents sortides: l'estudi de nous mètodes, treballar més amb qualsevol dels models fets, la recerca de mètodes per resoldre altres tipus d'equacions, l'ampliació dels programes o la millora dels algorismes serien possibles continuïtats del treball.

En un primer cas, podríem anar a la recerca de més mètodes que convergissin més ràpidament o que integressin amb més precisió utilitzant menys memòria i temps de CPU. Si només fent l'estudi de tres mètodes de cada ja hem trobat diferències substancials, podríem embarcar-nos en una enorme comparativa per arribar a conclusions importants. També podem buscar mètodes per resoldre altres tipus d'equacions, no només del tipus $f(x)= 0$ i diferencials de primer ordre.

Si ens centrem més en els programes fets, durant el treball van sorgir noves idees que no s'han portat a terme per la manca de coneixement. En la realització dels problemes vam veure que podíem treure més "suc" d'aquestos aprofundint en ells. Si ens ve a la memòria el problema de la caiguda lliure, podríem arribar a fer un potent simulador de caiguda on podríem esbrinar qualsevol cosa, des de saber si tirant dos cossos amb temps diferents podrien arribar-se a trobar a l'aire fins a calcular el desplaçament horitzontal que fem si ens tirem des d'una altura concreta. Així, en cada problema podem veure infinites possibilitats per estudiar, i amb el programa fet podríem taulejar valors i trobar mides òptimes o alçades adients.

Referint-nos a la part dels algorismes, amb temps i pràctica poden millorar-se el funcionament d'alguns programes, escrivint-los d'altres formes. Dedicar-se únicament a la programació d'eines per resoldre equacions també seria una bona forma de donar continuïtat al treball. Crear programes independents del Matlab que fessin la feina de trobar el zero d'una funció o que poguessin integrar una equació seria una altre possible via.

AGRAÏMENTS

Sincerament dono les gràcies al meu tutor del treball que m'ha ajudat a encarrilar el que penso que és una bona feina. Sense la seva serenitat podria haver sortit de tot menys un treball amb una estructura clara i uns bons conceptes. També vull agrair al Dr. Anton Vernet, professor de l'Escola Tècnica Superior d'Enginyeria Química de la Universitat Rovira i Virgili per haver-me ajudat en diferents temes de Matlab referents a processos informàtics i d'estètica. Donar les gràcies per l'orientació que m'ha donat el meu pare, la meva germana pel seu suport i a la meva mare, que encara que no sàpiga res del tema, també m'ha cuidat molt. Finalment donar les gràcies a les muntanyes del Tirol.

BIBLIOGRAFIA

- BURGHES D., BORRIE M. (1981). *Modelling with differential equations*. Ed. Ellis horwood limited, London
- CHENEY Ward, KINCAID David (1985). *Numerical mathematics and computing.* Ed. Brooks/Cole Publishing Company, Monterey.
- CLEVE, Moler (2004). *Numerical Computing with Matlab.* Ed. Siam, Philadelphia.
- CULLEN M. (1985). *Linear models in biology – linear systems analysis with biological applications*. Ed. Ellis Horwood Series, Los Angeles.
- FOGLER H. Scott (2006). *Elements of chemical reaction engineering*. Ed. Prentice Hall, Massachusetts.
- GERALD Curtis, WHEATLEY Patrick (1970). *Applied Numerical Analysis*. Ed Addison-Wesley publishing company, California.
- MATHEWS Jhon, FINK Kurtis (2000). *Métodos numéricos com Matlab*. Ed. Prentice Hall, Madrid.
- PÄRT-ENANDER Eva, SJÖBERG Anders (1999). *The Matlab handbook*. Ed. Prentice Hall, Great Britain.
- PRESS William, FLANNERY Brian, TOUKOLSKY Saul, VETTERLING William (1986). *Numerical Recipies - the art of Scientific Computing*. Ed. Cambridge University Press, Cambridge.
- VILENKIN N. (1984). *Método de aproximaciones sucesivas – lecciones populares de matemáticas*. Ed. Mir, Moskvá.
- WHITE Frank (1994). *Fluids Mechanics*. Ed. Mc Grow-Hill, United States of America.

PÀGINES WEB

http://en.wikipedia.org/wiki/G-force#NASA_g-tolerance_data

http://www.scholarpedia.org/article/Predator-prey_model

www.math.duke.edu/education/webfeatsII/Word2HTML/Predator-prey.doc

ANNEX I

PROBLEMA: PREDADOR - PRESA

PROBLEMA: PREDADOR – PRESA

La GFS no sempre salva el món dels dolents i de cataclismes mundials, també es preocupa pel medi ambient. Per tant, ara agafarem les armes d'un Eco-agent, persona que cuida, estudia i preserva tot el que fa referència a la natura. Així doncs, ha arribat una carta molt preocupant a l'Ecoservei de la GFS: en una illa vora les costes de Catalunya on s'hi cultivaven plantes medicinals, tenen un greu problema. Un pescador va deixar-hi per error una família de conills i ara aquests amenacen l'ecosistema de l'illa. S'ha pensat d'introduir-hi una nova espècie que controli aquesta plaga, les guineus. Ara bé, la pregunta és, la interacció entre dues espècies pot ser auto regulativa? Existeix un ecosistema que combini amb harmonia les preses i els depredadors? La resposta la tenen els eco-informàtics de la GFS. Tenen el deure moral de salvar l'illa de la completa ruïna i crear un model informàtic de població que ens permeti saber, abans de que succeeixi, què passarà amb els nous habitants de l'illa. Acabaran colonitzant l'illa o podran viure-hi amb harmonia?

Resolució:

Per poder fer la representació de dues espècies estudiant-ne la dinàmica de població, hem de fer referència al model biològic de Lotka-Volterra. Aquest model biològic va ser ideat per un matemàtic, Vito Volterra, i un matemàtic biòleg, Alfred Lotka. Curiosament, el model biològic actual és la combinació dels models creats per aquests dos científics, que van treballar separadament sense saber-ho. L'un va crear el model per donar explicació a la variació de població de bancs de peixos en el mar Adriàtic, i el segon va idear un model semblant per preveure resultats de la dinàmica de població entre plantes herbívores i animals dependents d'aquestes. En tots dos casos (avui en dia simplement es parla del model de Lotka-Volterra, no per separat) les hipòtesis en que es basa el model són:

- L'espècie depredadora és totalment dependent en aliment de les preses.
- L'espècie presa té aliment il·limitat i cap altre espècie apart de la depredadora l'amenaça.
- L'espècie depredadora no té cap amenaça, el seu creixement només depèn de l'interacció amb les preses.

Com podem veure, ens trobem en un cas una mica idealitzat, però és necessari per poder determinar les interaccions entre les preses, que anomenarem *x*, i els predadors, que anomenarem *y*.

Si no hi hagués predadors, com que les preses tenen aliment il·limitat, el seu creixement només dependria del nombre de preses que hi ha i que poden reproduir-se, per tant anomenant una constant a que és la velocitat de creixement:

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

Ara bé, com que tenim una població de predadors, *y*, hem de definir el creixement de la població, *x*, com l'equació anterior més un creixement negatiu causat per l'acció dels predadors sobre les preses. Les dues hipòtesis que ens falten per completar el model són:

- La velocitat a la qual els predadors troben les preses és proporcional a la mida de les dues poblacions, o sigui, el producte *x*y*.
- Només una determinada proporció, *b*, dels encontres entre els predadors i les preses provoquen la mort de la presa.

Per tant la velocitat de creixement de les preses podem definir-la com:

$$\frac{dx}{dt} = ax - bxy$$

Ara considerem la població dels predadors. Suposant que no hi hagués aliments els predadors acabarien morint a una velocitat proporcional a la població *y* que hi ha, de manera que ho modelitzarem:

$$\frac{dy}{dt} = -cy$$

On *-c* es la constant d'extinció de la població. Per sort, els predadors tenen aliment; les preses. Per tant poden créixer en la mesura que capturen les preses, que ha de ser un terme també proporcional a la mida de les dues poblacions, de la forma p*x*y. El model final de l'evolució de predadors és:

$$\frac{dy}{dt} = -cy + pxy$$

Aquesta equació, junt amb l'anterior són les que ens permeten modelitzar la dinàmica de població de les dues espècies, tenint-ne en compte les seves interaccions. Finalment doncs, el sistema que hem d'integrar és:

$$\frac{dx}{dt} = \text{ax} - \text{bxy} \qquad \frac{dy}{dt} = -cy + pxy$$

On *a, b, c, i p* són constants positives.

Solució Computacional:

La resolució d'aquest problema escau en la integració de dues equacions diferencials, per tant podem utilitzar qualsevol dels tres mètodes explicats: Euler, Runge-Kutta de segon ordre o de quart. En aquest cas però, el nostre Super-agent, a part de voler aprendre a escriure cadascun dels mètodes també vol saber com fer-ho per esbrinar quin anirà millor. Així doncs, consulta un treball de recerca que va fer fa temps sobre mètodes numèrics i hi troba la solució.

Com ja sabem, per escriure un programa d'aquest tipus hem de complir algunes condicions. En aquest cas hem de determinar les constants *a*, *b*, *c*, i *p.* Experimentalment podríem trobar valors aproximats d'aquests paràmetres però fent recerca en diversos llibres, també podem trobar-les (H. Scott, 2006). Així doncs, agafarem els valors d'una població de conills i guineus amb els valors de:

a = 0,2	Constant de creixement dels conills
b = 0,3	Constant de creixement de les guineus
c = 0,01	Constant d'extinció dels conills
p = 0,009	Constant d'extinció de les guineus

A part d'aquests valors també hem de dir la població inicial que tenim de cada espècie. Per tenir sentit biològic posarem que tenim deu exemplars de cada espècie, encara que podem variar-ho segons vulguem.

```
a=0.2 ;
c=0.4 ;
b=0.01 ;
p=0.009 ;
x0=10 ;
y0=10 ;
```

```
dt=0.1 ;
np=3000 ;
t=zeros(np,1);
x=zeros(np,1);
y=zeros(np,1);

t(1)=0 ;
x(1)=x0 ;
y(1)=y0
```

Acte seguit, és necessari definir els vectors per guardar els valors que anem obtenint, definir un pas de temps i un nombre total de passos que farà el programa. Jugant amb aquesta correlació podem arribar a veure prous resultats per preveure el comportament de la població sempre, i no caldrà haver de fer moltes iteracions, que l'únic que fan és robar molt de temps de treball de la CPU. També caldrà indicar que en la primera iteració de totes utilitzarem els valors inicials.

Com ja s'ha vist en l'apartat dels mètodes referents a RK, podem escriure el *kernel* d'aquest programa de dues formes diferents. Si a part de resoldre el problema també volem poder fer una comparació serà altament recomanable escriure el programa en relació a una funció externa, on simplement donarem l'ordre de les equacions que s'han d'integrar.

```
function [kx,ky]=f_calcula_rk(x,y,dt,a,b,c,p)

kx=dt*(a*x-b*x*y)   ;
ky=dt*(-c*y+p*x*y)  ;

return
```

Ara doncs és increïblement senzill calcular amb Runge-Kutta de quart ordre tots els passos. Només hem d'indicar el pas en que ens trobem fent referència a la funció externa, i escriure en cada pas la relació entre els valors que ens marca el mètode. També hem d'indicar que *x* i *y*, que són els conills i les guineus, són variables en cada iteració. Treballarem doncs amb *x(n)* i *y(n)*, indicant que cada element té un valor diferent en cada iteració, ja que després de fer el càlcul farem l'avanç de la solució indicant-ho al final i el següent cop que el programa torni a fer un procés de càlcul ja agafarà el nou valor, perquè denominarem *x(n)+1* el valor de *x(n)* més l' increment de *kx* (per la *y* justament el mateix). Tot això, traduït a algoritme de Matlab doncs és:

```
for n=1:np-1

    [k1x,k1y]=f_calcula_rk(x(n)      ,y(n)      ,dt,a,b,c,p) ;
    [k2x,k2y]=f_calcula_rk(x(n)+k1x/2,y(n)+k1y/2,dt,a,b,c,p) ;
    [k3x,k3y]=f_calcula_rk(x(n)+k2x/2,y(n)+k2y/2,dt,a,b,c,p) ;
    [k4x,k4y]=f_calcula_rk(x(n)+k3x  ,y(n)+k3y  ,dt,a,b,c,p) ;

    %Avanç solució
    x(n+1)=x(n)+(k1x+2*k2x+2*k3x+k4x)/6 ;
    y(n+1)=y(n)+(k1y+2*k2y+2*k3y+k4y)/6 ;
    t(n+1)=t(n)+dt                      ;

end
```

Veiem que cridem la funció externa fent referència al nom que hem donat i introduïm les variables amb les que volem treballar. Tota l'ordre va tancada amb un *for*...*end* indicant des de quin valor inicial fins a quin valor inicial ha d'iterar.

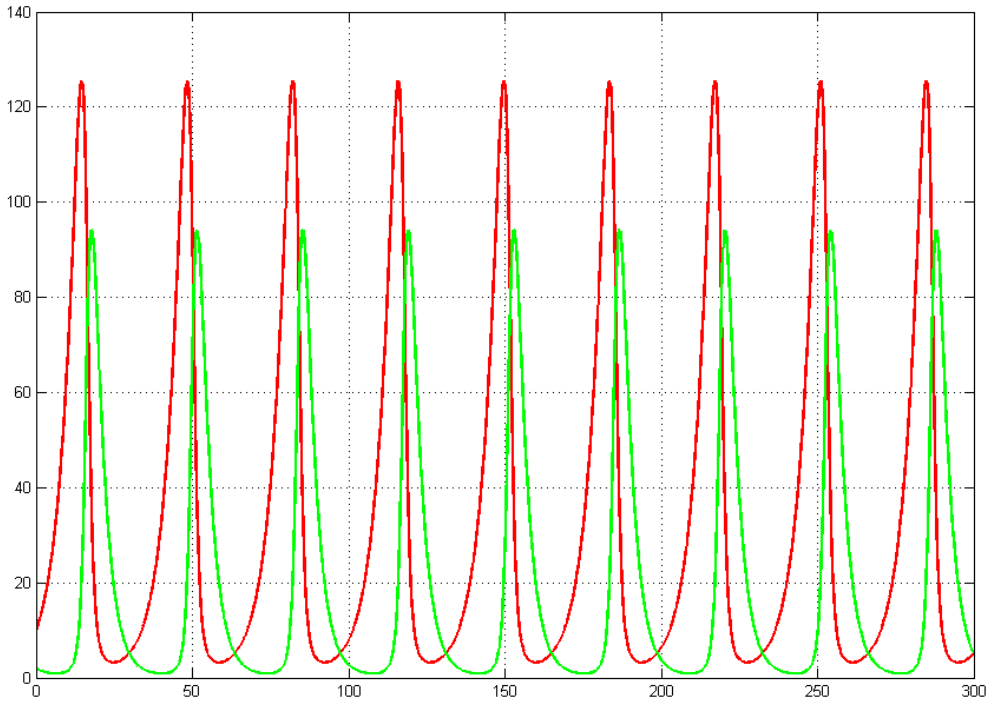
Aplicacions:

Ara que hem pogut escriure el programa, solucionar el mètode matemàtic i escriure un bon mètode, podrem donar resposta als habitants. No han de patir per res. El servei de la GFS ha desenvolupat un programa, que basant-se en el model de Lotka-Volterra de dinàmiques de població, assegura que si introduïm una família de guineus, aquesta s'alimentarà dels conills i crearem un sistema auto regulable. Quan les guineus

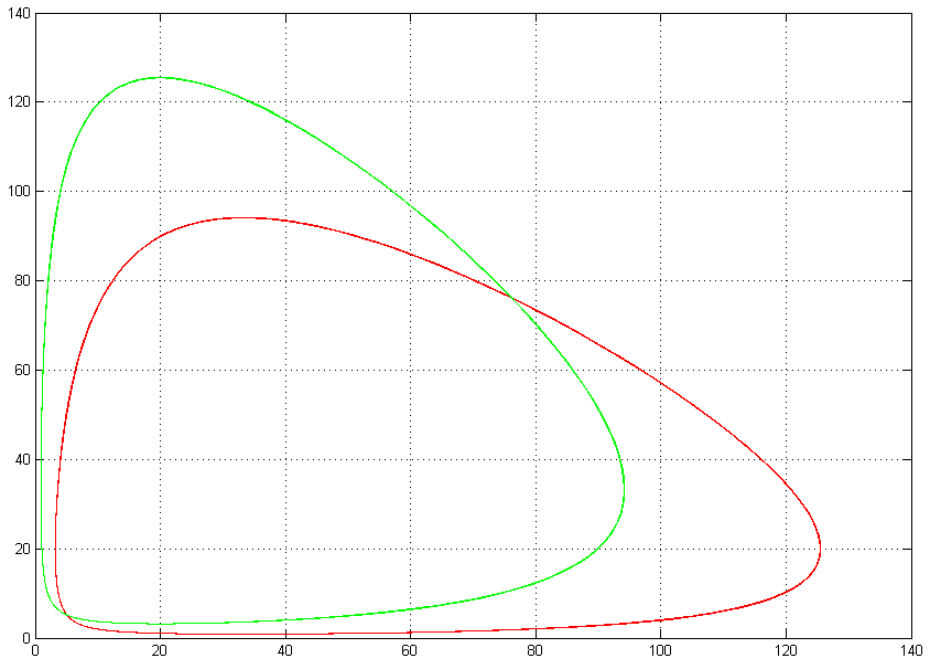
comencin a afartar-se de conills i es reprodueixin el nombre de conills disminuirà. Conseqüentment, després d'això, quan ja tinguem un elevat nombre de guineus però pocs conills, a les guineus els hi costarà trobar aliment i la seva població es farà més petita. Llavors serà l'oportunitat dels conills de tornar-se a reproduir. Així tindrem un cicle tancat on la població d'una espècie creixerà on l'altre decreixi. La població de les dues espècies serà estable i ja no s'ha de patir pels conreus de les plantes medicinals tant preuades.

Així doncs, als habitants de l'illa els hi ha arribat una carta amb un gràfic i unes paraules tranquil·litzadores.

En el gràfic podem veure com els conills, representats de color vermell i les guineus, representades amb color verd, creen un sistema estable, que sempre torna al punt d'origen. Això ens assegurarà una preservació de l'ecosistema de l'illa ja que les dues famílies d'animals serien autosuficients, els conills alimentant-se de poques plantes medicinals i les guineus dels conills. Per veure del tot si és o no un ecosistema sostenible podem dibuixar la evolució de les dues espècies una en funció de l'altre, no cadascuna en funció del temps. Si el gràfic final queda un sistema tancat podem donar-nos per satisfets, sinó es que hi haurà algun moment on la població es descontrolarà.



En aquest cas hem dibuixat en vermell la població de conills sobre la de guineus i en verd la de guineus sobre la de conills. En els dos casos veiem que es crea un sistema tancat i això satisfà als habitants de l'illa, ja que significa que és autoregulable.



Extensió:

Com havíem dit al començament del problema, la finalitat d'aquest programa no només era resoldre el model i poder obtenir una resposta pels habitants de l'illa. La GFS ha pogut donar uns resultats tant acurats degut a un previ estudi entre mètodes utilitzats. Durant el procés d'aprenentatge dels mètodes s'ha vist que no tots ells són igual de precisos, tot depèn del pas de temps i el nombre de passos que hi determinem. Ara bé, si volem fer un programa per poder veure les inestabilitats de cada programa, no ens hem de trencar massa el cap! Tal com hem vist podem programar deixant les equacions a una funció externa que cridarem quan vulguem. Per poder muntar una comparativa dels mètodes hem de seguir els passos fets ara però dividir el programa amb tres nuclis diferents.

Com a primer pas, la definició de dades inicials, vectors i tot el que no sigui el cervell d'iterar serà comú sempre, per tant només cal que ho definim una vegada al començar el programa. Ara bé, si volem compara un mètode d'Euler, Runge-Kutta de segon i de quart ordre, hem d'escriure diferents trossos de codi per cada un. En un primer pas escriurem cada *kernel* d'iteracions dins l'ordre *for...end*. Una de les característiques que té Matlab en l'executat dels programes és que sempre utilitza cada variable amb l'últim valor que hi donem. Això vol dir que si en un principi denominem un valor a una

y, i més endavant tornem a donar un valor a aquesta y, el programa treballarà amb l'últim que haguem definit. Això és important ja que si no volem haver d'escriure tres dades inicials, tres vectors diferents, etc. hem d'enganyar una miqueta el programa. La solució més senzilla és guardar cada variable amb un nom diferent del que hem utilitzat al final de cada processat d'operacions.

Per veure-ho més clar, exemplifiquem-ho sobre el codi programat. Si volem treballar amb les variables, que anomenem t, y, x en tots tres casos, només caldrà que després de les línies del mètode afegim l'ordre de reanomenar la variable amb uns altres caràcters i guardar aquests en memòria. Podem fer-ho dient que cada una de les variables té igualtat a un altre nom i després, amb l'ordre *save* i els valors que volem guardar és suficient. És important escriure aquest canvi de nom i guardat després de cada mètode, sinó quan executéssim el programa aquest arrossegaria els valors i no serviria de res.

```
tk1=t; xk1=x; yk1=y;
save rk1 tk1 xk1 yk1

tk2=t; xk2=x; yk2=y;
save rk2 tk2 xk2 yk2

tk4=t; xk4=x; yk4=y;
save rk4 tk4 xk4 yk4
```

Finalment, per veure bé si tots els mètodes obtenen els mateixos resultats podem comparar les dades, o representar-les gràficament, que ens donaran una idea més directa de si algun d'ells esdevé inestable o no. Ja que teníem les dades de cada mètode guardades en memòria només hem d'ordenar amb un *plot* que dibuixi cada variable.

```
plot(tk1,xk1,'-r',tk1,yk1,'-r',tk2,xk2,'-g',tk2,yk2,'-g',tk4,xk4,...
'-b',tk4,yk4,'-b'); grid;
```

Bàsicament, totes les ordres que comportin el processat de dades, en Matlab són introduïdes en l'ordre de: eix de les x, eix de les y, configuració de la representació. Si en tenim diferents casos simplement l'escrivim un després de l'altre. La ordre que fa referència al color, amplada de la línia, si és puntejada o no sempre ha d'anar entre cometes, ja que així estem dient que no és cap valor, que només és una ordre del programa intern. Matlab té diverses configuracions estendards que poden ser cridades amb poques lletres.

b	blau	.	punt petit	-	línia solida
g	verd	o	punt gros	:	puntejat
r	vermell	x	creu	-.	Punt - ratlla

Així doncs, per exemple si volem ordenar que dibuixi una línia de color verd puntejada hem de donar l'ordre '-.g'. també podem canviar molts paràmetres de dibuix dient nosaltres manualment la configuració de cada element, entrant en l'àmbit de configuració de dibuix amb l'ordre *set (gfc 'element que volem canviar')*, ara bé, el processat d'imatge pot ser una feina llarga i feixuga de fer, ja que s'han d'indicar totes les ordres.

Una ordre que també s'utilitza molt per fer la representació gràfica és la d'afegir una graella el gràfic. Simplement escrivint *grid* després del plot, farà que ens aparegui una graella al fons del gràfic fet.

ANNEX II

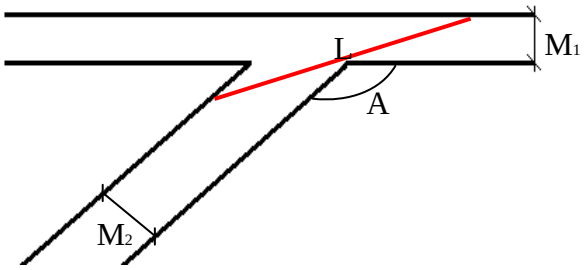
PROBLEMA: L'ESCALA MÉS LLARGA

PROBLEMA: L'ESCALA MÉS LLARGA

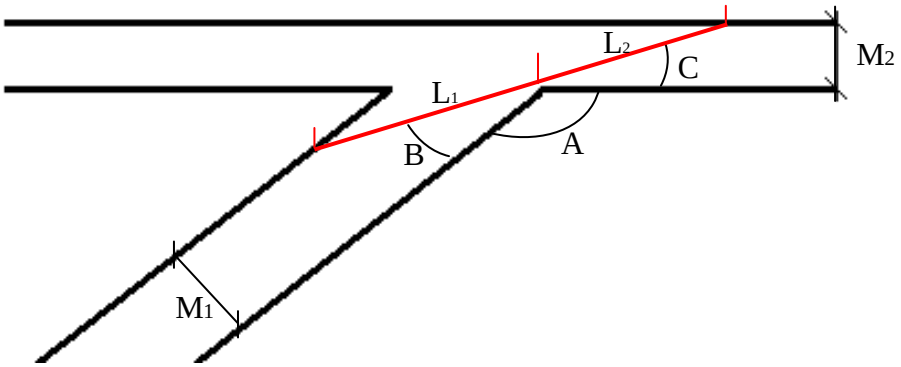
"*On és la Joana Smith?*" Fou la pregunta que formulà el nostre Super-agent secret. La Joana era la companya de treball del nostre heroi, i aquest cop estava en perill! Uns dolents malvats l'han segrestat i està en una cova, tancada sense cap possible escapatòria. Només el nostre heroi la pot salvar. El gran problema és que per arribar a la cambra on està retinguda s'ha de travessar un tallat de vint metres d'horitzontal i un centenar de vertical. La idea del Super-agent és portar una escala per poder-la utilitzar com a pont improvisat, un pla molt bonic, però amb un inconvenient: Ha d'agafar una bifurcació des del camí principal i no sap si podrà passar-hi una escala prou llarga com per satisfer les seves idees. Només pot fer dos viatges a l'interior de la cova (sé que és injust però les regles de les històries de Super-agents són així) i decideix gastar-ne un per entrar-hi amb una cinta mètrica i un mesurador d'angles. Es proposa esbrinar la llargada màxima que pot tenir una escala per poder superar aquell tomb.

Resolució:

Aquest cop la GFS farà referència a un model que pot explicar-se gràcies a la geometria. Hem d'imaginar-nos un passadís horitzontal que és tallat perpendicularment per un altre formant un angle *A* entre els dos.



També coneixem l'amplada dels dos passadissos, M_1 i M_2. El problema que volem resoldre és saber la llargada màxima que podrà tenir una escala coneixent només aquests tres paràmetres. Hem d'imaginar-nos l'escala que volem passar, *L*, com si fossin dos trossos separats. Així doncs parlarem de L_1 i L_2 i la suma dels dos trossos són la llargada total de l'escala. Gràficament el problema serà així:



Per trigonometria podem esbrinar cada tros d'escala quina longitud tindrà. En el fons el que estem fent es calcular la tangent, ja que dividim el catet contigu amb l'oposat.

$$L1 = \frac{M1}{\sin B} \qquad\qquad L2 = \frac{M2}{\sin C}$$

Però també podem fer-ho tot el funció del mateix angle. Sabem que l'escala es recte, per tant *A*+*B*+*C* = 180º o si treballem amb radiants *A*+*B*+*C* equivaldrà a π. Si ho deixem tot en funció de *C*, i contant que sabem el valor que te la *A*, llavors

$$B = \pi - A - C$$

Per tant doncs, podem dir que la longitud total de l'escala és:

$$L = L1 + L2 = \frac{M1}{\sin(\pi - A - C)} + \frac{M2}{\sin C}$$

Tant en el cas que *C* o B= $\pi - A - C$ tendeixin a zero, la longitud de l'escala seria infinita en L_2 i L_1 respectivament. El que ens interessa trobar és per quin valor de *C* la longitud *L* és mínima, ja que l'escala més gran que podrem fer girar entre els dos passadissos tindrà com a màxim aquesta longitud. Així doncs, el que cal és trobar l'extrem (el mínim) de *L* com una funció de *C*. Per tant, derivant *L* respecte a *C* i igualant a zero obtenim

$$\frac{dL}{dC} = \frac{-M1 * \cos(\pi - A - C)(-1)}{\sin 2(\pi - A - C)} + \frac{-M2 * \cos C}{\sin 2 C} = 0$$

Per tant el valor de *C* que minimitza la longitud de l'escala serà el zero d'aquesta funció, que anomenarem C_0

$$f(C) = \frac{M1 * \cos(\pi - A - C)}{\sin^2(\pi - A - C)} - \frac{M2 * \cos C}{\sin^2 C} = 0$$

La longitud de l'escala l'obtindrem substituint el valor de C_0 que trobem en l'equació de la seva longitud.

$$L = \frac{M1}{\sin(\pi - A - C_0)} + \frac{M2}{\sin C_0}$$

Solució Computacional:

Per resoldre aquest problema utilitzarem el mètode de la bisecció. El seu funcionament és com tots els mètodes per trobar zeros de funcions que hem vist amb la excepció de que al final hem d'utilitzar el valor de la solució per trobar la longitud màxima que tindrà l'escala.

```
clc; clear all;

tol=1e-7;                                          % Tolerància
m1=10 ;                                            % Amplada 1
m2=7 ;                                             % Amplada 2
a=120 ;                                            % Angle entre els passadissos(graus)

c1=10 ;          fc1=funcio_escala(c1,m1,m2,a) ; % Punt inicial de l'interval de cerca
c2=180-a-10 ;    fc2=funcio_escala(c2,m1,m2,a) ; % Punt final de l'interval de cerca
x=linspace(c1,c2,100) ;                            % Dibuix i definició de c com a vector
y=funcio_escala(x,m1,m2,a) ;                       % Dona valors a y per dibuixar
plot(x,y,'-b') ; grid ; hold on ;                  % Dibuixa la funció
c3=(c1+c2)/2; fc3=funcio_escala(c3,m1,m2,a) ;      % Càlcul punt intermedi
niter=0 ;                                          % Comptador d'iteracions

while abs(fc3)>tol & niter<100                     % Loop i control convergència
    c3=(c1+c2)/2; fc3=funcio_escala(c3,m1,m2,a) ;% Càlcul punt intermedi
    plot(c3,fc3,'or')                              % Dibuix iteració a la funció
    if sign(fc1)==sign(fc3)                        % Si sign(fa)==sign(fc) c = a
        c1=c3; fc1=fc3;
    else                                           % En cas contrari c substitueix b
        c2=c3; fc2=fc3;
    end
    niter=niter+1 ;                                % Compta iteracions
end                                                % Fi del loop
```

Així doncs definim l'amplada dels passadissos, el valor de l'angle A que formen aquests dos i ja podem escriure el cos del programa com tots els altres de buscar zeros de funcions. Definim un interval de cerca per fer les iteracions i amb l'ordre *while...end* fem el *kernel* per iterar. Per fer un òptim control del programa i evitar que aquest quedi iterant infinitament en cas d'entrar en un *loop* sense sortida, una de les ordres que podem posar després del *while* és un nombre màxim d'iteracions que farà el programa. Per fer això només cal pensar d'escriure un comptador d'iteracions que va des de 0 i cada cop que en fem una, la sumem al valor anterior. Si el programa fes més del nombre màxim d'iteracions (en aquest cas 100) deixaria de funcionar.

Dins el *loop* escriurem les ordres que fan que es compleixi el mètode de la bisecció amb condicionals *if*, *else*, i tancant amb un end.

```
lmax=(w2/sind(180-a-c3))+(w1/sind(c3)) ;       % Amb la c busquem la L màx de l'escala
title(['longitud maxima =',num2str(lmax)]) ;   % Digues aquet valor al gràfic (títol)
```

Després d'això trobarem un valor de C que utilitzarem per torbar la longitud màxima de l'escala. Podem fer que ens digui aquest valor exacte com a títol, amb el gràfic de totes les iteracions que ha fet el programa.

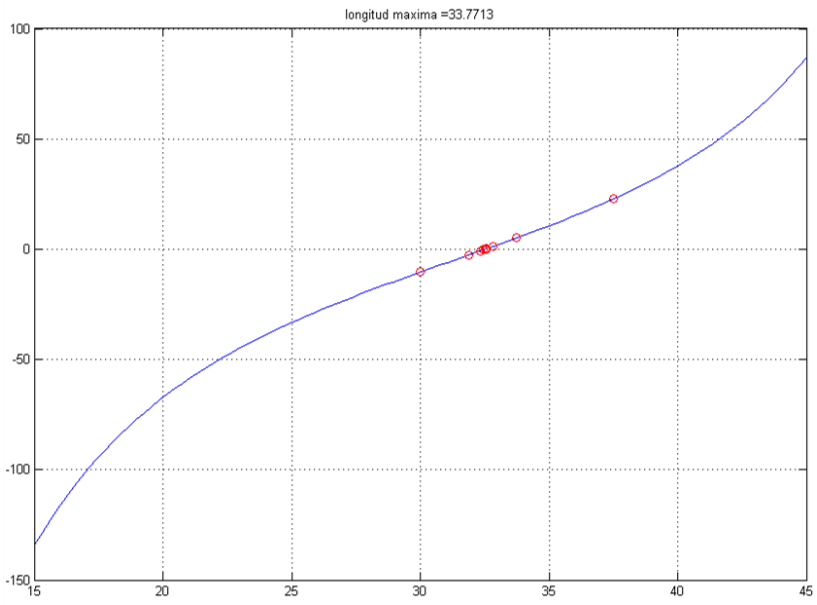
Cal recordar que el programa està escrit utilitzant una funció externa. En aquest cas remarcar les diferències que s'han de tenir em compte si treballem amb graus o radiants.

```
function [dldc]=funcio_escala(c,m1,m2,a)

dldc=(m2*cosd(180-a-c)./sind(180-a-c).^2)-(m1*cosd(c)./sind(c).^2);

return

% Ull, cal usar ./ i .^2 en lloc de / i ^2 per si cridem la funcio amb c
% que sigui un vector i no un sol escalar
```

Com que el programa informàtic l'hem escrit en graus per indicar les funcions de sinus i cosinus el llenguatge de Matlab fa servir: *sind* i *cosd*. Si treballéssim amb radiants podríem utilitzar sinus i cosinus escrivint-ho com a *sin* i *cos*. Un altre fet que remarcarem és l'ús del punt avanç d'operar. Com que els valors de *C* són variables, cada iteració n'utilitzarà un de diferent. Per fer entendre al programa que volem que utilitzi el valor de c de l'anterior iteració i no sempre el mateix, utilitzem el . davant el símbol d'operació, així doncs surt el ./ i .^2.

Aplicacions:

Després de fer el raonament geomètric i escriure el programa que l'hi permetrà saber si pot salvar o no la seva estimada companya d'equip, el Super-agent introdueix les dades que ha aconseguit amb les mesures preses. Amb un angle de 120 graus, un passadís de 10 metres, i un altre de 7 pot passar una escala que com a màxim faci 33,77 metres de llarg. Així doncs, no ha de patir per res, podrà salvar el tallat de vint metres i salvar la seva companya. Si no us ho creieu, mireu el gràfic que ens proposa el seu programa.



Finalment, com totes les històries de Super-agents, ha acabat bé.

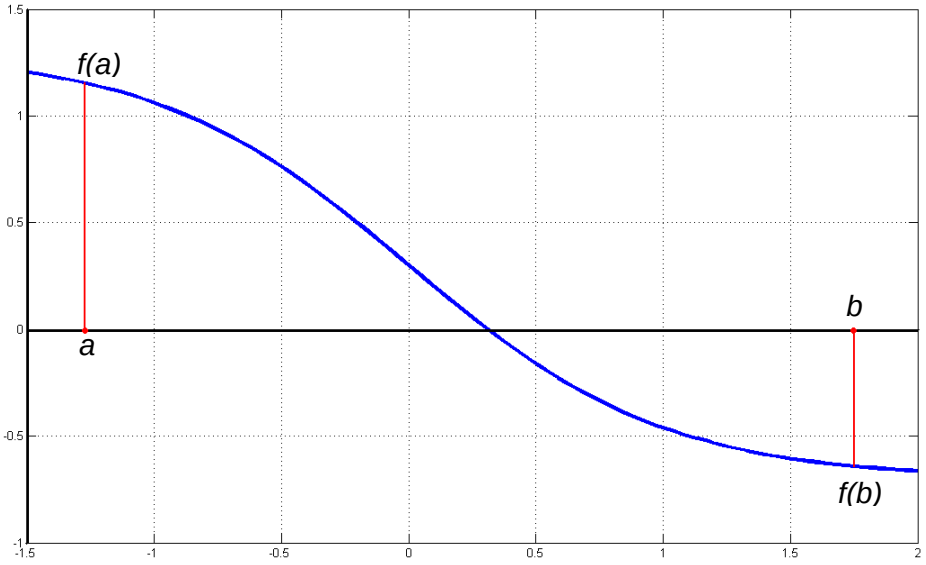
ANNEX III

UN ALTRE MÈTODE PER TROBAR ZEROS DE FUNCIONS

INTERPOLACIÓ LINEAL

Mètode matemàtic:

Un altre mètode estudiat que serveix per trobar zeros de funcions és anomenat interpolació lineal, però també és conegut pel nom de mètode de falsa posició o Regula falsi. Aquest té exactament la mateixa utilitat que els seus companys explicats als principi, trobar la solució d'una equació o l'arrel, ara bé, conté diferències. El mètode és més eficient i més precís que el mètode de la bisecció però no tant com el de Newton.



Per entendre el funcionament del mètode de Regula falsi tornarem a partir d'un interval denominat per un punt *a* i un punt *b*. És necessari que la funció que desitgem resoldre, la qual anomenarem *f(x)*, passi per zero en algun punt de l'interval [*a,b*]. Si no és així, el mètode no convergeix, no funciona, perquè això vol dir que no hi ha cap solució a l'interval buscat.

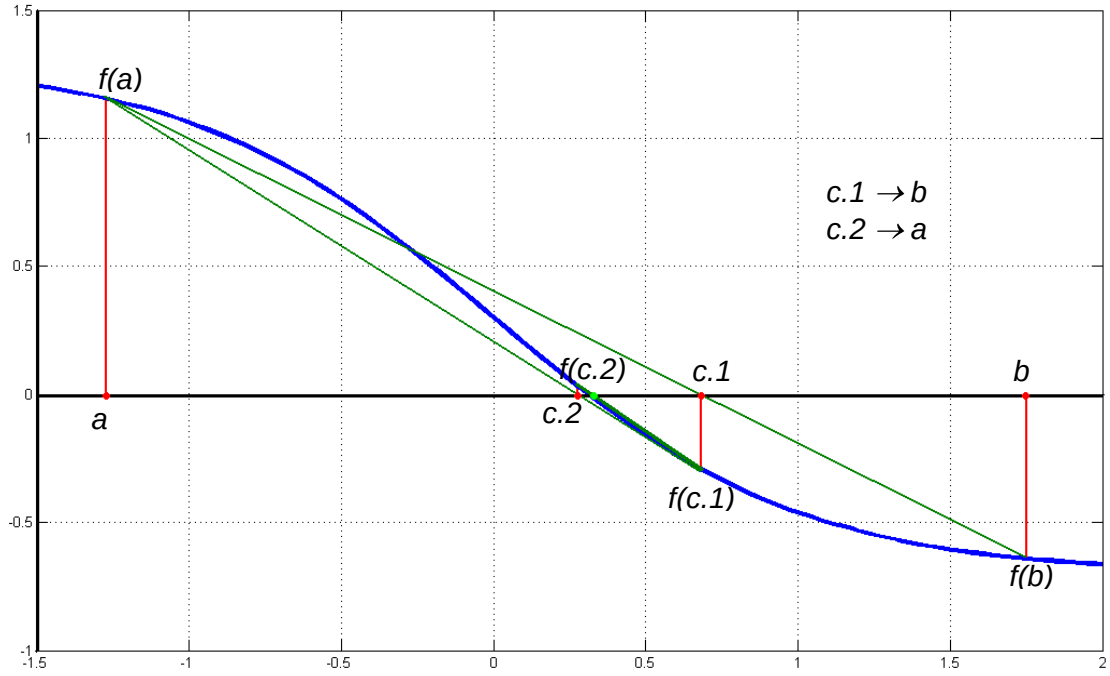
Amb aquest fet aconseguim trobar un *f(a)* que serà positiu o negatiu i a la vegada una *f(b)* que serà sempre el contrari de *f(a).* El següent pas a fer és construir una línia que uneixi *f(a)* i *f(b)* per trobar un nou valor anomenat *c*, que és on la recta talla l'eix d'ordenades. Aquest nou punt pot ser trobat gràficament com s'ha explicat o bé seguint la formula:

$$c = b - \frac{f(b)(b-a)}{f(b)-f(a)}$$

Pot donar-se el cas que en la primera iteració de qualsevol mètode ja és trobi la solució desitjada, però treballarem amb la suposició de que no tindrem tanta bona sort i que caldrà fer més iteracions. Per tant, com en el mètode estudiat anteriorment necessitarem unes normes per poder reanomenar *c* i així poder repetir el procés i fer una nova iteració.

$$si\ signe\ f(a) = signe\ f(c) \Rightarrow \quad a = c \quad f(a) = f(c)$$

$$si\ signe\ f(b) = signe\ f(c) \Rightarrow \quad b = c \quad f(b) = f(c)$$



Com podem veure en aquest exemple gràfic d'una funció *f(x),* el mètode quasi que ja convergeix en tant sols tres iteracions. En d'altres casos serien necessàries més "tirades" per poder trobar la solució, però comparant-ho amb l'exemple del mètode de la bisecció, veiem que va més ràpid.

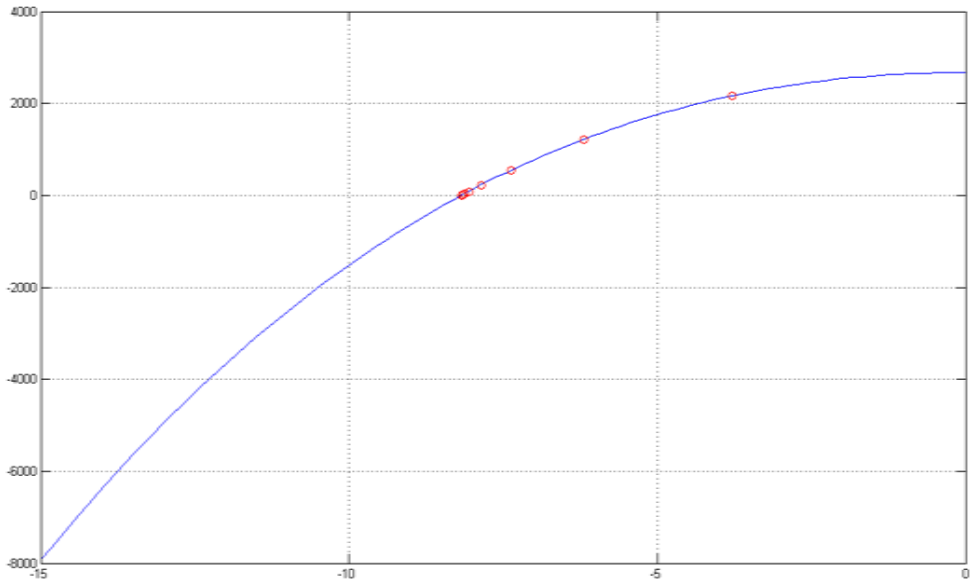
Després de veure'n l'explicació hem pogut comprovar que aquest mètode és molt semblant al de la bisecció, però té una forma diferent de treballar que el fa més acurat. Dient acurat s'ha d'entendre que podrem trobar la solució més aviat, per tant doncs, fent menys iteracions trobarem la solució. Aquest fet és important, ja que computacionalment augmenta el rendiment del programa ja que requereix menys esforç de la CPU o temps de càlcul.

Algoritme informàtic:

A l'hora de crear un programa amb el mètode de regula falsi, ens trobem amb un plantejament semblant al de la bisecció amb l'única variació de la forma com trobar el nou punt o com n'hi hem dit en aquest cas, la *c*. Llavors l'algoritme és idèntic excepte en la instrucció de com buscar el nou punt.

```
Clc; Clear all

while abs(fc)>tol          % Loop i control convergència
    c=b-(fb*(b-a))/(fb-fa); fc=funcio_1(c); %Càlcul punt intermedi
    plot(c,fc,'or')        % Dibuix iteració
    if sign(fa)==sign(fc)  % Si signe(fa)==signe(fc)
        a=c; fa=fc;        % llavors c substitueix a
    else                   % En cas contrari
        b=c; fb=fc;        % c substitueix b
    end
    niter=niter+1 ;        % Compta iteracions
end                        % Fi del loop
```

Fent una petita comparació amb l'anterior mètode cal remarcar que utilitzant bisecció són necessàries 33 iteracions per resoldre *f(x)=0*, utilitzant la funció que fem servir d'exemple. Al resoldre la mateixa arrel utilitzant el mètode estudiat de Regula falsi, només se'n necessiten 24 per trobar la mateixa solució. En ambdós casos estem demanant que iteri fins a aproximar-se a la solució amb una tolerància de 10^{-6}. Gràficament podem apreciar una mica el nombre inferior d'iteracions fetes.



ANNEX IV

ELS PROGRAMES

```
%PILOTA FLOTANT – FUNCIÓ EXTERNA 1

function [f] = funcio_1(x)

f=((x.^3-30*x.^2+2552)*pi)/3;

return

% En aquest cas la densitat del cos és sempre constant i no està en
%funció de la massa i %el volum total. Cal tenir en compte que l'ordre
%de càlcul de la densitat s'ha descriure %al programa.
```

```
%PILOTA FLOTANT – FUNCIÓ EXTERNA 1

function [f] = funcio_2(x,r,d)

f=((x.^3-3*x.^2*r+4*r^3*d)*pi)/3;

return

% Així calcularem la densitat en relació a la massa i el volum.
```

```
%PILOTA FLOTANT -MÈTODE BISECCIÓ

clc; clear all;

tol=1e-6 ;                          % Definim l'error màxim que volem fer
r=0.8;                              % Radi
m=200            ;
v=(4/3)*pi*(r^3)        ;
d=(m/v)/1000 ;                      % Així calculem la densitat segons massa i volum
% d=0.683                           % Densitat independent massa i volum
a=-1 ;   fa=funcio_2(a,r,d) ;       % Punt inicial de busqueda
b=3;     fb=funcio_2(b,r,d) ;       % Punt final de busqueda
x=linspace(a,b,100) ;               % Valors de x per dibuixar la funció
y=funcio_2(x,r,d) ;                 % Valors de y per dibuixar la funció
plot(x,y,'-b') ; grid ; hold on ;% Si ho dibuixo aquí la funció no passa per tots els punts
c=(a+b)/2; fc=funcio_2(c,r,d) ;     % Càlcul punt intermedi
niter=0 ;                           % Comptador d'iteracions
tic                                 % Comptador del temps

while abs(fc)>tol                           % Loop i control convergència
    c=(a+b)/2; fc=funcio_2(c,r,d) ;         % Càlcul punt intermedi
    x=[x,c] ;                               % Posa c a la llista de x per dibuixar
    plot(c,fc,'or')    ;  hold on ;         % Dibuix iteració
    if sign(fa)==sign(fc)                   % Si sign(fa)==sign(fc) llavors c=a
        a=c; fa=fc ;
    else                                    % En cas contrari c substitueix b
        b=c; fb=fc ;
    end
    niter=niter+1 ;                         % Compta iteracions
end                                         % Fi del loop

x=sort(x) ;                       % Ordena la llista de x per dibuixar
y=funcio_2(x,r,d) ;               % Calcula valors de y per dibuixar
plot(x,y,'-b') ; grid ;           % Dibuixa una funció que passa per tots els punts c calculats
toc                               % Fi del comptador del temps
disp(c)
```

```
%PILOTA FLOTANT – MÈTODE INTERPOLCIÓ LINEAL

clc; clear all;

tol=1e-6 ;                          % Definim el valor de la tolerància
a=-15    ; fa=funcio_1(a) ;         % donem un valor a i fem que calculi f(a)
b=0      ; fb=funcio_1(b) ;         % donem un valor a b i també calculem f(b)
x=linspace(a,b,100);                % Valors de x per dibuixar la funció
y=funcio_1(x);                      % Valors de y per dibuixar la funció
plot(x,y,'-b') ; grid ; hold on ;   % Dibuixem la funció sense iteracions, per veure que talli de veritat
%c=(a+b)/2; fc=funcio_1(c);         %
niter=0 ;                           % Comptador d'iteracions a 0
while abs(fc)>tol                   % Control on si el valor absolut de f(c) més gran que la tolerància
    c=b-(fb*(b-a))/(fb-fa); fc=funcio_1(c); % Calcular el nou valor de c0
    plot(c,fc,'or')                 % Dibuixa'l al gràfic
    if sign(fa)==sign(fc)           % Si sign(fa)==sign(fc) llavors c=a
        a=c; fa=fc;                 %
    else                            % En cas contrari b=a
        b=c; fb=fc;
    end                             % Acaba el loop d'iteracions
%  a=b ; fa=fb ; % Fet així en lloc del if anterior...
%  b=c ; fb=fc ; % això es el mètode de la secant
    niter=niter+1 ;                 % Suma 1 al comptador d'iteracions
end


% A l'utilitzar les línies de codi de color verd fem servir el mètode de la secant, que és un mètode molt semblant
%al d'interpolació. La %diferencia que té es que en cada cas s'agafa el valor més pròxim a la solució, així s'itera
%de forma molt més ràpida.
```

```
%PILOTA FLOTANT – NEWTON RHAPSON

clc; clear all

tol=1e-6 ;    % Tolerància
x1=5 ;        % Punt a
x2=15 ;       % Punt b
x=linspace(x1,x2,1000);              % Valors de x per dibuixar
[y,dy]=funcio_i_derivada_1(x);       % Càlcul de les y
plot(x,y,'-b') ; grid ; hold on ;    % Dibuixa la funció
tic                                  % Comença a comptar el temps
x1=(x1+x2)/2 ;                       % Fent això ens assegurem que comenci l'iteracio des del mig de l'interval
[fx1,dfx1]=funcio_i_derivada_1(x1) ; % Càlcul de la primera fx1 i dfx1
for niter=1:100                      % Fes com a màxim 100 iteracions
    x2=x1-fx1/dfx1; [fx2,dfx2]=funcio_i_derivada_1(x2) ; % Càlcul nou punt
    plot(x2,fx2,'og')                % Dibuixa'l
    x1=x2 ; fx1=fx2 ; dfx1=dfx2 ;    % Agafa el nou valor per tornar a iterar
    disp(['niter=',num2str(niter),' x1=',num2str(x1,16),' fx1=',num2str(fx1)]) ; % Ensenya els números per pantalla
    if abs(fx1)<tol                  % Si el valor és més petit que la tolerància
        break                        % Para d'iterar
    end
end
   plot(x2,fx2,'+g')                 % Dibuixa una + a l'ultima iteració
toc                                 % Fi del comptador del temps
```

```
%PILOTA FLOTANT – LES TRES SOLUCIONS DE L'EQUACIÓ


clc; clear all;

tol=1e-6 ;
x11=[-15,10,25] ; % La equació que descriu el fet té tres solucions, només una és la de la pilota...
x22=[-5 ,15,35] ; % per tant hem de definir tres intervals de cerca
x=linspace(min(x11),max(x22),1000);          % Crea un vector amb nombre min i max depenent de x
[y,dy]=funcio_i_derivada_1(x);               % Per cada valor de x, troba en relació la funció el valor de y
plot(x,y,'-b') ; grid ; hold on ;            % A dibuixar!
tic

for n=1:3 % Per assegurar-nos que el mètode entra dins el loop comencem a iterar des del mig de l'interval
    x1=(x11(n)+x22(n))/2 ; [fx1,dfx1]=funcio_i_derivada_1(x1) ; % Càlcul de la primera x1, fx1, dfx1
    for niter=1:100                          % Després itera (100 com a max)
        x2=x1-fx1/dfx1; [fx2,dfx2]=funcio_i_derivada_1(x2) ; % Nou punt
      if n==1 ; plot(x2,fx2,'ok') ; end      % dibuix solució 1
      if n==2 ; plot(x2,fx2,'og') ; end      % dibuix solució 2
      if n==3 ; plot(x2,fx2,'or') ; end      % dibuix solució 3
        x1=x2 ; fx1=fx2 ; dfx1=dfx2 ;        % copia el valor nou al lloc del vell
        disp(['niter=',num2str(niter),' x1=',num2str(x1,16),'fx1=',num2str(fx1)]) ;% Números a la pantalla
        if abs(fx1)<tol                      % si el valor absolut de fx1 (solució) es més petit que la
              break                          % ...tolerància, acaba d'iterar
        end
    end
   if n==1 ; plot(x2,fx2,'+k') ; end % Dibuixa una + a l'ultima solució
   if n==2 ; plot(x2,fx2,'+g') ; end
   if n==3 ; plot(x2,fx2,'+r') ; end
    disp([' ']) ;
end
toc
```

```
% CAIGUDA LLIURE - EULER

clc; clear all;

% Dades

y0=1000 ;              % Alçada inicial
v0=0 ;                 % Velocitat inicial
radi=0.2 ;             % Radi de l'objecte
area=pi*radi^2 ;       % Àrea
massa=10 ;             % Massa
volum=4/3*pi*radi^3 ;  % Volum
d=massa/volum ;        % Densitat
g=9.8 ;                % Constant gravitatòria
d_aire=1.2 ;           % Densitat aire
cd=0.8 ;               % Coeficient fregament

% Vectors

dt=0.5 ;               % Pas de temps
np=5000 ;              % Nombre de passos
t=zeros(np,1) ;        % Vector temps
y=zeros(size(t)) ;     % vector posició
v=zeros(size(t)) ;     % Vector velocitat

% Dades inicials

y(1)=y0 ; % Atribuïm al primer pas, el valor inicial en...
v(1)=v0 ; % la posició, velocitat i temps.
t(1)=0 ;

% Iteracions a sac, tio!

for n=1:np-1                    % Itera des de 1 fins a np-1
    y(n+1)=y(n)+v(n)*dt ;       % Troba la nova posició, la nova...
    v(n+1)=v(n)+(-g+(cd*area/massa)*0.5*d_aire*v(n)^2)*dt ; %velocitat
    t(n+1)=t(n)+dt ;                                        % i temps
    if y(n+1)<=0        % Si la nova posició és més petita o igual a 0
        break           %atura't
    end
end

ntot=n+1 ;             % Nombre total de passos fets
y=y(1:ntot) ;          % Talla els trossos de vectors que sobrin...
v=v(1:ntot) ;          % de posició, velocitat i temps
t=t(1:ntot) ;

% Dibuix

subplot(2,1,1) ; plot(t,y) ; grid ;
subplot(2,1,2) ; plot(t,v) ; grid ;

yeu=y; veu=v ; teu=t;
save eu teu yeu veu
```

```
% CAIGUDA LLIURE - COMPARATIVA EULER AMB DIFERENT PAS DE TEMPS
clc; clear all;

% Dades inicials
y0=1000 ;
v0=0 ;
radi=0.2 ;
area=pi*radi^2 ;
massa=10 ;
volum=4/3*pi*radi^3 ;
d=massa/volum ;
g=9.8 ;
d_aire=1.2 ;
cd=0.8 ;
% Vectors   ->    Definim dos vectors, un per cada pas de temps.
np=5000 ;
dt1=0.5 ;               % Pas de temps de 0.5 segons
t1=zeros(np,1) ;
y1=zeros(size(t1)) ;
v1=zeros(size(t1)) ;
dt2=4 ;                 % Pas de temps de 2 segons
t2=zeros(np,1) ;
y2=zeros(size(t2)) ;
v2=zeros(size(t2)) ;
% Dades inicials
y1(1)=y0 ; y2(1)=y0 ; % Donem el valor de la primera iteració...
v1(1)=v0 ; v2(1)=v0 ; % per cada cas. Així ho hem de definir...
t1(1)=0 ;  t2(1)=0 ;  % dues vegades

% Iteracions a sac, tio!
for n=1:np-1          % Primer paquet d'iteracions amb dt 0.5
    y1(n+1)=y1(n)+v1(n)*dt1 ;
    v1(n+1)=v1(n)+(-g+(cd*area/massa)*0.5*d_aire*v1(n)^2)*dt1 ;
    t1(n+1)=t1(n)+dt1 ;
    if y1(n+1)<=0
        break
    end
end

ntot1=n+1 ;           % Nombre total de passos fets
y1=y1(1:ntot1) ;      % Tallem el tros de vector que ens sobri
v1=v1(1:ntot1) ;
t1=t1(1:ntot1) ;

for n=1:np-1          % Segon paquet d'iteracions amb dt 2
    y2(n+1)=y2(n)+v2(n)*dt2 ;
    v2(n+1)=v2(n)+(-g+(cd*area/massa)*0.5*d_aire*v2(n)^2)*dt2 ;
    t2(n+1)=t2(n)+dt2 ;
    if y2(n+1)<=0
        break
    end
end

ntot2=n+1 ;           % Nombre total de passos fets
y2=y2(1:ntot2) ;      % Tallem el tros de vector que ens sobri
v2=v2(1:ntot2) ;
t2=t2(1:ntot2) ;
subplot(2,1,1) ; plot(t1,y1,'-or',t2,y2,'-og') ; grid ; % Posició
subplot(2,1,2) ; plot(t1,v1,'-or',t2,v2,'-og') ; grid ; % Velocitat
```

```
% CAIGUDA LLIURE - RUNGE-KUTTA 2N ORDRE
% Equacions iterades:
% dy/dt=v ;
% dv/dt=-g+(cd*area/2*massa)*d_aire*v^2
% y(n+1)=y(n)+(ky1+ky2)/2
% v(n+1)=v(n)+(kv1+kv2)/2

clc; clear all ;

% Dades
y0=1000 ;
v0=0 ;
radi=0.2 ;
area=pi*radi^2 ;
massa=10 ;
volum=4/3*pi*radi^3 ;
d=massa/volum ;
g=-9.8 ;
d_aire=1.2 ;
cd=0.8 ;

% Vectors
dt=1.5                                  ;
np=1000                                 ;
t=zeros(np,1)                           ;
y=zeros(size(t))                        ;
v=zeros(size(t))                        ;

% Definicio de dades inicials
t(1)=0                                  ;
y(1)=y0                                 ;
v(1)=v0                                 ;


% Cosa que itera
for n=1:np-1
[k1y,k1v]=f_calcula_rk(y(n)      ,v(n)      ,g,dt,cd,area,massa,d_aire);
[k2y,k2v]=f_calcula_rk(y(n)+k1y/2,v(n)+k1v/2,g,dt,cd,area,massa,d_aire) ;
    %Avanç solucio
    y(n+1)=y(n)+(k1y+k2y)/2 ; % Per fer el càlcul del nou pas...
    v(n+1)=v(n)+(k1v+k2v)/2 ; % fem servir k1 i k2 dividint-ho entre..
    t(n+1)=t(n)+dt          ; % 2, ja que així fem el valor mitjà
    if y(n+1)<=0
        break
    end
end
ntot=n+1 ;
t=t(1:ntot) ;
y=y(1:ntot) ;
v=v(1:ntot) ;

subplot(2,1,1); plot(t,y) ; grid ;
subplot(2,1,2); plot(t,v) ; grid ;
```

```
% CAIGUDA LLIURE - COMPARATIVA RUNGE-KUTTA 2N/4T ORDRE

% Equacions iterades:
% dy/dt=v ;
% dv/dt=-g+(cd*area/2*massa)*d_aire*v^2
% y(n+1)=y(n)+(ky1+ky2)/2
% v(n+1)=v(n)+(kv1+kv2)/2

clc; clear all ;

% Dades
y0=1000 ;
v0=0 ;
radi=0.2 ;
area=pi*radi^2 ;
massa=10 ;
volum=4/3*pi*radi^3 ;
d=massa/volum ;
g=-9.8 ;
d_aire=1.2 ;
cd=0.8 ;

% Vectors
dt=1.5                                  ;
np=1000                                 ;
t=zeros(np,1)                           ;
y=zeros(size(t))                        ;
v=zeros(size(t))                        ;

% Definicio de dades inicials
t(1)=0                                  ;
y(1)=y0                                 ;
v(1)=v0                                 ;

%=========================================================================
% Cosa que itera UTILITZANT 2N ORDRE
for n=1:np-1
  [k1y,k1v]=f_calcula_rk(y(n)      ,v(n)      ,g,dt,cd,area,massa,d_aire) ;
  [k2y,k2v]=f_calcula_rk(y(n)+k1y/2,v(n)+k1v/2,g,dt,cd,area,massa,d_aire) ;
    %Avanç solucio
      y(n+1)=y(n)+(k1y+k2y)/2 ;
      v(n+1)=v(n)+(k1v+k2v)/2 ;
      t(n+1)=t(n)+dt          ;
      if y(n+1)<=0
          break
      end
end
ntot=n+1 ;
t=t(1:ntot) ;
y=y(1:ntot) ;
v=v(1:ntot) ;


subplot(2,1,1); plot(t,y) ; grid ;
subplot(2,1,2); plot(t,v) ; grid ;
```

```
%=========================================================================
% Cosa que itera UTILITZANT 4N ORDRE

for n=1:np-1

  [k1y,k1v]=f_calcula_rk(y(n)      ,v(n)      ,g,dt,cd,area,massa,d_aire) ;
  [k2y,k2v]=f_calcula_rk(y(n)+k1y/2,v(n)+k1v/2,g,dt,cd,area,massa,d_aire) ;
  [k3y,k3v]=f_calcula_rk(y(n)+k2y/2,v(n)+k2v/2,g,dt,cd,area,massa,d_aire) ;
  [k4y,k4v]=f_calcula_rk(y(n)      ,v(n)+k3v  ,g,dt,cd,area,massa,d_aire) ;

     y(n+1)=y(n)+(k1y+2*k2y+2*k3y+k4y)/6 ;
     v(n+1)=v(n)+(k1v+2*k2v+2*k3v+k4v)/6 ;
     t(n+1)=t(n)+dt          ;
     if y(n+1)<=0
         break
     end
end
ntot=n+1 ;
t=t(1:ntot) ;
y=y(1:ntot) ;
v=v(1:ntot) ;

subplot(2,1,1); plot(t,y) ; grid ;
subplot(2,1,2); plot(t,v) ; grid ;



% En aquest cas el que estem fent és dibuixar el gràfic que ens genera...
% el mètode de RK 2n ordre i a damunt hi dibuixem el que ens genera un...
% RK de quart ordre. Per tant és un funcionament independent de dos...
% nuclis de programa que embastem sobre un mateix gràfic.
```

```
% CAIGUDE LLIURE – RUNGE KUTTA 4T ORDRE PREPARAT PER LA GUI

function f_rk4_caigualliure % transformat a funció per poder compilar i crear-ne un executable extern a Matlab

global handles


load handel; yhandel=y ; clear y; % Carrega la musica

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% button='Yes' ;
% while strcmp(button,'Yes') ;
%
% Cosa per triar y0 i massa integrant la GUI dintre el programa. No queda tant bonic, ja que no crea una pantalla
% nova on poder treballar-hi. Així doncs ho deixarem escrit però desactivat (ha sigut el pas previ a l'altre GUI)
%
% prompt={'Entra l''alçada','Entra la massa'};
% name='Caiguda lliure';
% numlines=[1,40] ;
% defaultanswer={'500','100'};
% options='on' ;
% resposta=inputdlg(prompt,name,numlines,defaultanswer,options);
% y0 = str2num(resposta{1}) ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


y0 =str2num(get(handles.AlInput,'String'))
massa =str2num(get(handles.PesInput,'String'))
```

```
% Dades

% y0=1000 ;  % La definició de alçada inicial la voldrem triar a l'iniciar l'executat del programa, no ara.
v0=0 ;
yobre1=y0*3/6;
yobre2=yobre1-180;
radi1=1 ;
radi2=4 ;
area1=1 ;
area2=pi*radi2^2 ;
% massa=10 ;  % El mateix passarà amb la massa. El programa agafarà els valors que nosaltres diguem al moment..
% volum=4/3*pi*radi^3 ;  % de fer corra el programa, no són introduïts per aquí.
% d=massa/volum ; % La densitat va en funció de la massa, per tant encara no la podem calcular
g=-9.8 ;
d_aire=1.2 ;
cd=0.8 ;

% Vectors

dt=0.08                                 ;
np=50000                                ;
t=zeros(np,1)                           ;
y=zeros(size(t))                        ;
v=zeros(size(t))                        ;
a=zeros(size(t))                        ;
ar=zeros(size(t))                       ;

% Definicio de dades inicials

t(1)=0                                  ;
y(1)=y0                                 ;
v(1)=v0                                 ;
a(1)=g                                  ;
ar(1)=area1                             ;
```

```
%Control de obertura del paracaigudes
for n=1:np-1
    if y(n)>yobre1   % Cal tenir en compte la obertura del paracaigudes. Jugarem com si dividíssim la caiguda en ...
        area=area1 ; % dos trossos, cada un d'ell té una àrea diferent. Així donem la ordre de control dintre el ...
    elseif y(n)<=yobre1 & y(n)>yobre2 % loop total de control de caiguda.
        area=((y(n)-yobre1)/(yobre2-yobre1))*(area2-area1)+area1;
    else
        area=area2 ;
    end
    ar(n+1)=area;

%Runge-Kutta

    [k1y,k1v]=f_calcula_rk(y(n)      ,v(n)      ,g,dt,cd,area,massa,d_aire) ;
    [k2y,k2v]=f_calcula_rk(y(n)+k1y/2,v(n)+k1v/2,g,dt,cd,area,massa,d_aire) ;
    [k3y,k3v]=f_calcula_rk(y(n)+k2y/2,v(n)+k2v/2,g,dt,cd,area,massa,d_aire) ;
    [k4y,k4v]=f_calcula_rk(y(n)+k3y  ,v(n)+k3v  ,g,dt,cd,area,massa,d_aire) ;

%Avanç solució

    y(n+1)=y(n)+(k1y+2*k2y+2*k3y+k4y)/6 ;
    v(n+1)=v(n)+(k1v+2*k2v+2*k3v+k4v)/6 ;
    a(n+1)=(v(n+1)-v(n))/dt             ;
    t(n+1)=t(n)+dt                      ;
    if y(n+1)<=0
        break
    end
end

% Tallem els vectors que sobren
ntot=n+1      ;
t=t(1:ntot)   ; % Indiquem que cada vector té des de la primera iteració fins a la ultima, que n'hi hem dit ntot.
y=y(1:ntot)   ;
v=v(1:ntot)   ;
a=a(1:ntot)   ;
ar=ar(1:ntot) ;
```

```
% Dibuixa la caiguda des del principi al final amb paquets de deu iteracions
player = audioplayer(yhandel, Fs);                % Reprodueix la musica
play(player,[1 (get(player, 'SampleRate')*38)]); % Temps de reproducció

for nplot=1:10:ntot  % Farem el gràfic a trossos cada 10 iteracions
%      subplot(2,2,1);
axes(handles.uiaxes1);
        plot(t(1:nplot),y(1:nplot),'-b',t(nplot),y(nplot),'or') ;
        xlabel('temps') ; ylabel('posició');     % Nom dels eixos
        title(['temps = ',num2str(t(ntot)),' segons'])
        axis([t(1),t(ntot),min(y),max(y)]) ; grid ;
        drawnow

%      subplot(2,2,2);
axes(handles.uiaxes2);
        plot(t(1:nplot),v(1:nplot)) ; xlabel('temps') ; ylabel('velocitat');
        title(['velocitat = ',num2str(abs(v(nplot))),' m/s'])
        axis([t(1),t(ntot),min(v),max(v)]) ; grid ;
        drawnow

%      subplot(2,2,4);
axes(handles.uiaxes3);
        plot(t(1:nplot),a(1:nplot)/abs(g)) ; xlabel('temps') ; ylabel('acceleració/g');
        title(['acceleració màxima = ',num2str(max(abs(a/abs(g)))),' '])
        axis([t(1),t(ntot),min(a/abs(g)),max(a/abs(g))]) ; grid ;
        drawnow

%      subplot(2,2,3);
axes(handles.uiaxes4);
        plot(t(1:nplot),ar(1:nplot)) ; xlabel('temps') ; ylabel('area');
        axis([t(1),t(ntot),0,max(ar+2)]) ; grid ;
        drawnow
end
% Aquest es el funcionament del programa. A partir d'ara ve la programació per crear la caràtula, botons, gràfics, etc.
% això és el que se'n diu GUI (Grafical User Interface). El programa per tant l'haurem d'executar des d'aquí després
% d'haver carregat en memòria tot l'anterior.
```

```
function mainProg

clc
clear

global handles

MainFig=figure('MenuBar','none',...
               'Name','Caiguda lliure',...
               'NumberTitle','off',...
               'ToolBar','none',...
               'Position', [30 50 900 700],...
               'Tag','MainFig',...
               'Resize','on');



% ---------------------------------------- input dades

AlTxt=uicontrol(MainFig,'Style','text',...
                    'String','Alçada inicial',...
                    'Tag','AlTxt',...
                    'BackgroundColor',get(MainFig,'Color'),...
                    'Position',[50 600 130 30]);

AlInput=uicontrol(MainFig,'Style','edit',...
                    'BackgroundColor','w',...
                    'String','750',...
                    'Tag','AlInput',...
                    'Position',[50 570 130 30]);

PesTxt=uicontrol(MainFig,'Style','text',...
                    'String','Pes',...
                    'Tag','PesTxt',...
                    'BackgroundColor',get(MainFig,'Color'),...
                    'Position',[50 500 130 30]);

PesInput=uicontrol(MainFig,'Style','edit',...
                    'BackgroundColor','w',...
                    'String','90',...
                    'Tag','PesInput',...
                    'Position',[50 470 130 30]);

% ---------------------------------------- botons

RunButton=uicontrol(MainFig,'Style','pushbutton',...
                    'String','Avall valent!',...
                    'Tag','RunButton',...
                    'BackgroundColor',get(MainFig,'Color'),...
                    'Position',[50 300 130 30],...
                    'Callback',{@RunButton_callback});

ExitButton=uicontrol(MainFig,'Style','pushbutton',...
                    'String','Sortir',...
                    'Tag','ExitButton',...
                    'BackgroundColor',get(MainFig,'Color'),...
                    'Position',[50 50 130 30],...
                    'Callback',{@ExitButton_callback});
```

```
% ---------------------------------------- eixos gràfics

uiaxes1=axes('Parent',MainFig,...
             'Tag','uiaxes1',...
             'Box','on',...
             'Ytick',[ ], ...
             'Xtick',[ ], ...
             'units','pixels',...
             'Position',[300 400 250 250])

uiaxes2=axes('Parent',MainFig,...
             'Tag','uiaxes2',...
             'Box','on',...
             'Ytick',[ ], ...
             'Xtick',[ ], ...
             'units','pixels',...
             'Position',[600 400 250 250])

uiaxes3=axes('Parent',MainFig,...
             'Tag','uiaxes3',...
             'Box','on',...
             'Ytick',[ ], ...
             'Xtick',[ ], ...
             'units','pixels',...
             'Position',[300 75 250 250])

uiaxes4=axes('Parent',MainFig,...
             'Tag','uiaxes4',...
             'Box','on',...
             'Ytick',[ ], ...
             'Xtick',[ ], ...
             'units','pixels',...
             'Position',[600 75 250 250])

 handles=guihandles(MainFig);


%  al=str2num(get(handles.AlInput,'String'))
 end
% ------------------------------------------------------------------
function RunButton_callback(hObject,eventdata)
global handles

f_rk4_caigualliure

% al=str2num(get(handles.AlInput,'String'))
end
% ------------------------------------------------------------------
function ExitButton_callback(hObject,eventdata)
global handles

close all
return

end
```

```
% DEPREDADOR PRESA – MÈTODE D'EULER

clc; clear all ;

% Dades

a=2                 ; % Factor creixement conills
c=1                 ; % Factor creixement guineus
b=1                 ; % Constant relacio conills-guineus
p=1                 ; % Constant relacio guineus conills
x0=0.5              ; % Poblacio conills
y0=0.25             ; % Poblacio guineus

% Vectors

dt=0.01             ; % Pas de temps de cada càlcul
np=5000             ; % Nombre passos
t=zeros(np,1)       ; % Temps
x=zeros(np,1)       ; % Conills
y=zeros(np,1)       ; % Guineus
niter=zeros(np,1)   ; % Iteracions

% Definicio de dades inicials

t(1)=0                                  ; % Temps en el primer pas
x(1)=x0                                 ; % Conills primer pas
y(1)=y0                                 ; % Guineus primer pas

% Cosa que itera

for n=1:np-1
    x(n+1)=x(n)+(a*x(n)-b*x(n)*y(n))*dt  ; % Calcula població conills
    y(n+1)=y(n)+(-c*y(n)+p*x(n)*y(n))*dt ; % Calcula poblacio guineus
    t(n+1)=t(n)+dt                       ; % Avança pas de temps
    niter=niter+1;
end

plot(t,x,t,y) ; grid ;                     % Dibuixa
```

```
% DEPREDADOR PRESA - RUNGE KUTTA QUART ORDRE (LLARG)

clc; clear all ;

% Dades
a=2                 ; % Factor creixement conills
c=1                 ; % Factor creixement guineus
b=1                 ; % Constant relació conills - guineus
p=1                 ; % Constant relació guineus conills
x0=0.5              ; % Població conills
y0=0.25             ; % Població guineus

% Vectors
dt=0.01             ; % Pas de temps de cada càlcul
np=5000             ; % Nombre passos
t=zeros(np,1)       ; % Temps
x=zeros(np,1)       ; % Conills
y=zeros(np,1)       ; % Guineus

% Definicio de dades inicials
t(1)=0                      ; % Temps en el primer pas
x(1)=x0                     ; % Conills primer pas
y(1)=y0                     ; % Guineus primer pas

% Cosa que itera
for n=1:np-1
    %K1
    x1=x(n) ;
    y1=y(n) ;
    k1x=dt*(a*x1-b*x1*y1)   ; % calcula nova població de conills
    k1y=dt*(-c*y1+p*x1*y1)  ; % calcula nova població de guineu
    %K2
    x2=x(n)+k1x/2 ;
    y2=y(n)+k1y/2 ;
    k2x=dt*(a*x2-b*x2*y2)   ; % calcula nova població de conills
    k2y=dt*(-c*y2+p*x2*y2)  ; % calcula nova població de guineu
    %K3
    x3=x(n)+k2x/2 ;
    y3=y(n)+k2y/2 ;
    k3x=dt*(a*x3-b*x3*y3)   ; % calcula nova població de conills
    k3y=dt*(-c*y3+p*x3*y3)  ; % calcula nova població de guineu
    %K4
    x4=x(n)+k3x ;
    y4=y(n)+k3y ;
    k4x=dt*(a*x3-b*x3*y3)   ; % calcula nova població de conills
    k4y=dt*(-c*y3+p*x3*y3)  ; % calcula nova població de guineu
    %Avanç solució
    x(n+1)=x(n)+(k1x+2*k2x+2*k3x+k4x)/6 ;
    y(n+1)=y(n)+(k1y+2*k2y+2*k3y+k4y)/6 ;

    t(n+1)=t(n)+dt          ; % avança pas de temps
end

plot(t,x,t,y); grid ;
```

```
% DEPREDADOR PRESA - RUNGE KUTTA QUART ORDRE (FUNCIÓ)


clc; clear all ;

% Dades

a=0.2               ; % factor creixement conills
c=0.3               ; % factor creixement guineus
b=0.01              ; % constant relacio conills-guineus
p=0.009             ; % constant relacio guineus conills
x0=10               ; % poblacio conills
y0=2                ; % poblacio guineus

% Vectors

dt=0.1              ; % Pas de temps de cada càlcul
np=3000             ; % Nombre passos
t=zeros(np,1)       ; % Temps
x=zeros(np,1)       ;
y=zeros(np,1)       ;

% Definicio de dades inicials

t(1)=0                                      ;  % Temps en el primer pas
x(1)=x0                                     ;  % conills primer pas
y(1)=y0                                     ;  % guineus primer pas

% Cosa que itera
for n=1:np-1
    [k1x,k1y]=f_calcula_rk(x(n)      ,y(n)      ,dt,a,b,c,p) ;
    [k2x,k2y]=f_calcula_rk(x(n)+k1x/2,y(n)+k1y/2,dt,a,b,c,p) ;
    [k3x,k3y]=f_calcula_rk(x(n)+k2x/2,y(n)+k2y/2,dt,a,b,c,p) ;
    [k4x,k4y]=f_calcula_rk(x(n)+k3x  ,y(n)+k3y  ,dt,a,b,c,p) ;

    %Avanç solució
    x(n+1)=x(n)+(k1x+2*k2x+2*k3x+k4x)/6 ;
    y(n+1)=y(n)+(k1y+2*k2y+2*k3y+k4y)/6 ;
    t(n+1)=t(n)+dt                      ;

plot(t,x,'r','LineWidth',2,t,y,'b','LineWidth',2); grid ;
plot(t,x,'r','LineWidth',2); hold on;plot(t,y,'g','LineWidth',2);grid;

% En aquesta ordre final de dibuixar configurem nosaltres mateixos...
% l'amplada de la linia per fer el dibuix.

%================================FUNCIÓ=================================

function [kx,ky]=f_calcula_rk(x,y,dt,a,b,c,p)

kx=dt*(a*x-b*x*y)  ;
ky=dt*(-c*y+p*x*y) ;

return
```

```
% DEPREDADOR PRESA - COMPARATIVA TRES MÈTODES

clc; clear all ;

% Dades

a=1                  ; % Factor creixement conills
c=1                  ; % Factor creixement guineus
b=2                  ; % Constant relació conills-guineus
p=1                  ; % Constant relació guineus conills
x0=0.5               ; % Població conills
y0=0.25              ; % Població guineus

% Vectors

dt=0.02             ; % Pas de temps de cada càlcul
np=5000             ; % Nombre passos
t=zeros(np,1)       ; % Temps
x=zeros(np,1)       ;
y=zeros(np,1)       ;

% Definicio de dades inicials

t(1)=0              ; % Temps en el primer pas
x(1)=x0             ; % conills primer pas
y(1)=y0             ; % guineus primer pas

% RK1=================================================================
for n=1:np-1
    [k1x,k1y]=f_calcula_rk(x(n)      ,y(n)      ,dt,a,b,c,p) ;
    %Avança solució
    x(n+1)=x(n)+(k1x);
    y(n+1)=y(n)+(k1y);
    t(n+1)=t(n)+dt   ;
end

tk1=t; xk1=x; yk1=y;
save rk1 tk1 xk1 yk1

% RK2=================================================================
for n=1:np-1
    [k1x,k1y]=f_calcula_rk(x(n)      ,y(n)      ,dt,a,b,c,p) ;
    [k2x,k2y]=f_calcula_rk(x(n)+k1x/2,y(n)+k1y/2,dt,a,b,c,p) ;
    %Avança solució
    x(n+1)=x(n)+(k1x+k2x)/2 ;
    y(n+1)=y(n)+(k1y+k2y)/2 ;
    t(n+1)=t(n)+dt          ;
end

tk2=t; xk2=x; yk2=y;
save rk2 tk2 xk2 yk2

% RK4=================================================================
for n=1:np-1
    [k1x,k1y]=f_calcula_rk(x(n)      ,y(n)      ,dt,a,b,c,p) ;
    [k2x,k2y]=f_calcula_rk(x(n)+k1x/2,y(n)+k1y/2,dt,a,b,c,p) ;
    [k3x,k3y]=f_calcula_rk(x(n)+k2x/2,y(n)+k2y/2,dt,a,b,c,p) ;
```

```
    [k4x,k4y]=f_calcula_rk(x(n)+k3x  ,y(n)+k3y  ,dt,a,b,c,p) ;
    %Avança solució
    x(n+1)=x(n)+(k1x+2*k2x+2*k3x+k4x)/6 ;
    y(n+1)=y(n)+(k1y+2*k2y+2*k3y+k4y)/6 ;
    t(n+1)=t(n)+dt                      ;
end

tk4=t; xk4=x; yk4=y;
save rk4 tk4 xk4 yk4

plot(tk1,xk1,'-r',tk1,yk1,'-r',tk2,xk2,'-g',tk2,yk2,'-g',tk4,xk4,...
'-b',tk4,yk4,'-b'); grid;

% En l'ultima ordre de dibuixar introduïm totes les dades al mateix gràfic %
i així obtenim gràfics que comparats ens serveixen per distingir quin
% mètode va millor.
```

ANNEX V

EL FRACTAL DE MANDELBROT

```
% Donar les gràcies a Andreas Klimke, de l'universitat de Stuttgart que
% és el creador d'aquest programa. Personalment només he cambiat els
% codis de color del fractal per utilizar-lo a la portada.

function mandelbrot(resolution, escape, iter, dx, dy)

if str2double(version('-release')) < 13
    % If the release is less than 13 (V6.5.x), perform vectorized
    % version of the algorithm

    x = linspace(dx(1),dx(2),resolution);
    y = linspace(dy(1),dy(2),resolution*(dy(2)-dy(1))/(dx(2)-dx(1)));
    [X,Y] = meshgrid(x,y);
    Z = X + i.*Y;
    r = zeros(size(Z));
    c = Z;
    for k = 1:iter
        disp(['Step #: ' num2str(k)]);
        Z = Z.^2 + c;
        r = r + (abs(Z) <= escape);
    end
    disp('Done!');
else
    % Release number is 13 or more, i.e. a JIT compiler is
    % present. Perform the non-vectorized version of the algorithm
    % instead.
    dx1 = dx(1);
    dx2 = dx(2);
    dy1 = dy(1);
    dy2 = dy(2);

    yresolution = round(resolution*(dy2-dy1)/(dx2-dx1));
    r = zeros(yresolution,resolution);

    stepadd = (dx2-dx1)/resolution;
    escape = escape^2;
    zrn = 0;

    ci = dy1;
    for m = 1:yresolution
        ci = ci + stepadd;
        cr = dx1;
        for n = 1:resolution
            cr = cr + stepadd;
            zr = cr;
            zi = ci;
            rmax = iter;
            for k = 1:iter
                zrn = zr * zr - zi * zi + cr;
                zi  = 2 * zi * zr + ci;
                zr = zrn;
                if (zr*zr+zi*zi) > escape
                    rmax = k;
                    break
                end
            end
            r(m,n)=rmax;
        end
    end
end
```

```
imagesc(r);
axis equal; axis tight;

%================================GUI==================================

if nargin == 0  % LAUNCH GUI

    fig = openfig(mfilename,'reuse');

    % Use system color scheme for figure:
    set(fig,'Color',get(0,'defaultUicontrolBackgroundColor'));

    % Generate a structure of handles to pass to callbacks, and store
it.
    handles = guihandles(fig);
    guidata(fig, handles);
    set(handles.cmap,'Value',10);
    set(fig,'Units','pixels');
    resetbutton_Callback([],[],handles);
    guiviewsize = get(fig,'Position');
    screensize = get(0,'ScreenSize');
    screensize = screensize(3:4);
    viewsize = [400 300];
    viewoffset = [50 200];
    set(fig,'Position', [screensize(1)-viewsize(1)- ...
                                        2*viewoffset(1)-guiviewsize(3),
screensize(2)- ...
                                        viewoffset(2)-guiviewsize(4),
...
                                        guiviewsize(3:4)]);
    set(gcf,'Position',[screensize(1)-viewoffset(1)-viewsize(1), ...
            screensize(2)-viewoffset(2)-viewsize(2), viewsize])
    if nargout > 0
        varargout{1} = fig;
    end

elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION OR CALLBACK

    try
        [varargout{1:nargout}] = feval(varargin{:}); % FEVAL switchyard
    catch
        disp(lasterr);
    end

end

% --------------------------------------------------------------------
function varargout = edit1_Callback(h, eventdata, handles, varargin)
return;

% --------------------------------------------------------------------
function varargout = popupmenu1_Callback(h, eventdata, handles,
varargin)
% Stub for Callback of the uicontrol handles.popupmenu1.
s = get(handles.cmap,'String');
colormap(s{get(handles.cmap,'Value')});

% --------------------------------------------------------------------
```

```
function varargout = resetbutton_Callback(h, eventdata, handles, ...
varargin)

s = get(handles.cmap,'String');
colormap(s{get(handles.cmap,'Value')});
resolution = str2double(get(handles.resolution,'String'));
iter = str2double(get(handles.iter,'String'));
escape = str2double(get(handles.escape,'String'));
dx1 = str2double(get(handles.dx1,'String'));
dx2 = str2double(get(handles.dx2,'String'));
dy1 = str2double(get(handles.dy1,'String'));
dy2 = str2double(get(handles.dy2,'String'));
set(handles.dx1,'UserData',[dx1, dx2, dy1, dy2, resolution]);
mandelbrot(resolution, escape, iter, [dx1 dx2], [dy1 dy2]);
xlabel(['x-interval: [' num2str(dx1) ', ' num2str(dx2) ']']);
ylabel(['y-interval: [' num2str(dy1) ', ' num2str(dy2) ']']);
set(gca,'XTick',[]);
set(gca,'YTick',[]);

% ------------------------------------------------------------------------
function varargout = redrawbutton1_Callback(h, eventdata, handles, ...

varargin)

s = get(handles.cmap,'String');
colormap(s{get(handles.cmap,'Value')});
resolution = str2double(get(handles.resolution,'String'));
iter = str2double(get(handles.iter,'String'));
escape = str2double(get(handles.escape,'String'));
userd = get(handles.dx1,'UserData');
if isempty(userd)
    dx1 = str2double(get(handles.dx1,'String'));
    dx2 = str2double(get(handles.dx2,'String'));
    dy1 = str2double(get(handles.dy1,'String'));
    dy2 = str2double(get(handles.dy2,'String'));
    oldres = resolution;
else
    dx1 = userd(1);
    dx2 = userd(2);
    dy1 = userd(3);
    dy2 = userd(4);
    oldres = userd(5);
end
nx = get(gca,'XLim') - 0.5;
ny = get(gca,'YLim') - 0.5;
nx = (nx/oldres)*(dx2-dx1)+dx1;
oldresy = oldres*(dy2-dy1)/(dx2-dx1);
ny = (ny/oldresy)*(dy2-dy1)+dy1;
userd = [nx ny resolution];
set(handles.dx1,'UserData',userd);
mandelbrot(resolution, escape, iter, nx, ny);
xlabel(['x-interval: [' num2str(nx(1),10) ', ' num2str(nx(2),10) ']']);
ylabel(['y-interval: [' num2str(ny(1),10) ', ' num2str(ny(2),10) ']']);
set(gca,'XTick',[]);
set(gca,'YTick',[]);

% ------------------------------------------------------------------------
function varargout = helpbutton_Callback(h, eventdata, handles,
varargin)
helpwin('mandelbrot_gui.m');
```