UNIVERSITAT DE BARCELONA
Facultat de Matemàtiques i Informàtica

GRAU DE MATEMÀTIQUES

Treball final de grau

Equacions diferencials ordinàries i diferenciació automàtica

Autor: Clara Gubau Gubert

Director: Dr. Àngel Jorba
Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, 13 de juny de 2022

Abstract

Automatic differentiation is an alternative method to compute the derivatives of a function in a given point. This method requires that the function can be written as a sequence of elementary operations and basic functions like exponential or trigonometry ones. Once we have our function as a combination of those elements, we can compute it and find its derivatives.

Moreover, there are the Poincaré sections. This is a really common tool used to study dynamical systems, but the computation of its derivatives used to be a frequent computational problem. In order to solve this, we can use automatic differentiation. More precisely, we will study how to modify a numerical integrator to compute automatically the derivatives of the flow of a differential equation regarding some initial conditions. The numerical integrator that we will use is Runge-Kutta-Fehlberg of order 4 and 5.

Resum

La diferenciació automàtica és un mètode alternatiu per calcular el valor de les derivades d'una funció en un punt. Aquest mètode requereix que la funció es pugui escriure com una seqüència d'operacions elementals juntament amb algunes funcions bàsiques, com les trigonomètriques o l'exponencial. Un cop tenim la funció expressada com a combinació d'elements que podem programar en un ordinador, podrem trobar les derivades d'aquesta funció.

D'altra banda, tenim les seccions de Poincaré, una eina molt comuna per l'estudi dels sistemes dinàmics. El càlcul efectiu de les derivades d'una aplicació de Poincaré sol ser un problema computacional molt freqüent. És per això que s'utilitza la diferenciació automàtica per calcular-ne les derivades. Concretament, en aquest treball veurem com modificar un integrador numèric d'equacions diferencials ordinàries per fer que calculi les derivades del flux de l'equació diferencial respecte les condicions inicials. L'integrador numèric que usarem és el Runge-Kutta-Fehlberg d'ordres 4 i 5.

.

2020 Mathematics Subject Classification. 65L06, 70K42

Agraïments

Primer de tot, m'agradaria donar les gràcies al doctor Àngel Jorba, per tota la seva ajuda i dedicació, juntament amb l'innegable suport que m'ha donat fins a últim moment. Ha estat un veritable plaer.

També m'agradaria donar les gràcies a la meva família i amics, tant als de sempre com als que he pogut conèixer a base d'hores d'estudi junts. Sou els qui heu estat allà durant tot el meu recorregut com a futura matemàtica, heu tingut un gran paper en tot aquest procés. Finalment, gràcies Adrià, has estat el qui m'ha motivat i fet costat sempre que era necessari. El mèrit d'haver arribat al final per entregar aquest treball també és teu.

Índex

Introducció

Estructura del treball

El treball s'estructura en quatre grans blocs, que ens permetran anar generant eines per acabar aplicant la diferenciació automàtica en un sistema d'equacions diferencials ordinàries.

Els quatre blocs, tal com es pot veure en l'índex, són

- els mètodes d'integració, en concret de Runge-Kutta,
- l'estudi de les sèries formals de potències,
- la diferenciació automàtica,
- l'estudi de les derivades d'una aplicació de Poincaré.

Podem considerar que els conceptes s'enllacen en el moment de voler utilitzar l'aplicació de Poincaré per estudiar de manera més senzilla el comportament d'un sistema dinàmic determinat per un problema del Cauchy concret.

L'aplicació de Poincaré caracteritza la intersecció d'una òrbita periòdica d'un sistema dinàmic amb un subespai de dimensió inferior transversal al flux del sistema. Donat que l'aplicació de Poincaré està formada pels punts que intersequen aquesta secció, no té una expressió explícita en forma de funció amb la qual puguem treballar. Per tant, necessitem un mètode d'integració que ens permeti obtenir el flux del sistema al punt on es troba la secció de Poincaré.

En el nostre cas, per fer el procés d'integració, hem escollit el mètode de Runge-Kutta-Fehlberg d'ordres 4 i 5. Aquest mètode té una implementació òptima i relativament senzilla d'implementar, a més de no tenir un cost computacional considerablement alt.

Donat que volem estudiar l'equació del pèndol pertorbat, per aconseguir l'aplicació de Poincaré d'aquest sistema seran necessàries les derivades de la funció que descriu el moviment. Per evitar calcular la fórmula explícita d'aquestes derivades i poder-ho fer tot mitjançant la programació, utilitzarem la diferenciació automàtica. Aquest és un mètode que, fent servir la sobrecàrrega d'operadors i l'aritmètica de sèries, obté les derivades de la funció desitjada en un punt concret.

Podríem considerar que aquest mètode equival a canviar un valor real pel polinomi de Taylor d'una funció, i treballar amb aquest últim. D'aquesta manera, si tenim el mètode d'integració programat per una funció concreta, podem obtenir de manera més o menys senzilla les seves derivades.

Els diferents temes estan explicats en el capítol dedicat a l'estudi d'aquests. Finalment, a l'apèndix hi constà adjuntat el codi desenvolupat.

Objectius i motivació

Podem dividir els objectius d'aquest treball en dos tipus principals: els teòrics i els pràctics.

Els teòrics estan motivats pel fet de no haver cursat l'assignatura d'Equacions Algebraiques a la Universitat de Barcelona i la voluntat de profunditzar més en aquest àmbit de les matemàtiques, així com per continuar desenvolupant els coneixements sobre Mètodes Numèrics.

D'altra banda, els resultats pràctics consisteixen a poder aplicar tot el coneixement teòric adquirit. Dins aquest ampli propòsit, n'hi ha alguns de menors com aprendre completament el llenguatge de programació `C++`, el qual coneixia únicament de manera parcial, o poder estudiar de manera més precisa i senzilla els punts fixos de certes equacions diferencials ordinàries utilitzant la programació.

Capítol 1

Mètodes d'integració

En aquest apartat, ens centrarem a trobar un mètode viable computacionalment per trobar una solució al problema del valor inicial o problema de Cauchy. El problema de Cauchy consisteix a trobar la solució d'un sistema d'equacions diferencials del tipus

$$\begin{cases} x'(t) = f(t, x(t)) \\ x(a) = x_0 \end{cases} \tag{1.0.1}$$

on x és la funció que volem trobar, $x'(t)$ la seva derivada i $f : [a, b] \times \mathbb{R}^m \to \mathbb{R}^m$ el camp.

Per resoldre aquest problema, utilitzarem els mètodes d'integració, uns algorismes que permeten aproximar numèricament les solucions de les equacions descrites anteriorment. El mètode que estudiarem en profunditat és el de Runge-Kutta, juntament amb algun derivat d'aquest. Es tracta d'un mètode d'un sol pas, és a dir, que obté el valor del nou punt de l'òrbita utilitzant únicament el punt anterior.

La notació utilitzada serà la següent: dividirem l'interval $[a, b]$ en $N + 1$ parts. Cada punt d'aquesta divisió el notarem com (t_n, x_n) on t_n és el $t_n \in [a, b]$ determinat segons N, $x(t_n)$ és el valor exacte de l'òrbita i x_n és la seva aproximació.

Anomenem *pas* a la distància en la variable t entre un punt i l'anterior. Ho notarem com $h_n = t_{n+1} - t_n$.

Una part important de l'estudi dels mètodes d'integració de Runge-Kutta serà trobar el valor òptim del pas h. A l'hora d'escollir h, hi ha dos errors principals que podem cometre, i caldrà evitar. El primer és reduir massa el pas, ja que malgrat que les aproximacions seran més precises, el nombre de càlculs serà molt major. Això pot provocar un augment dels errors d'arrodoniment i un augment del cost computacional i, per tant, que es dificulti la simulació. D'altra banda, si el pas h és considerablement gran, perdrem precisió i l'error comès incrementarà, malgrat que la simulació esdevindrà menys costosa. Així doncs, haurem de trobar l'equilibri entre aquests dos casos.

1.1 Fórmules generals de Runge-Kutta

Els mètodes de Runge-Kutta són un conjunt de mètodes iteratius desenvolupats pels matemàtics C. Runge i M. W. Kutta als voltants de l'any 1900. En comparació a altres mètodes, com per exemple el d'Euler, aquest assegura una major precisió, ja que més endavant veurem que permet avaluar el camp en punts estratègics. A més, la seva implementació i el cost computacional permet una fàcil simulació.

Hi ha dos termes principals que cal definir i especificar a l'hora d'estudiar els mètodes de Runge-Kutta. El primer és el nombre d'avaluacions de la funció que fem en cada iteració, el qual identificarem amb una m, i el segon és l'ordre, que té a veure amb l'error que es comet entre un valor i el següent. L'anomenarem p.

Tal com enuncia Butcher [1], si un mètode de Runge-Kutta amb m avaluacions del camp té ordre p, es pot demostrar que el nombre d'avaluacions haurà de complir $m \geq p$, i si $p \geq 5$, llavors $m \geq p+1$. Aquest enunciat l'haurem de tenir en compte més endavant, quan calculem les fórmules del mètode de Runge-Kutta-Fehlberg d'ordres 4 i 5.

Les fórmules generals del mètode de Runge-Kutta de m passos es poden escriure com,

$$x_{n+1} = x_n + h \sum_{i=1}^{m} b_i \kappa_i, \tag{1.1.1}$$

on les κ_i les es defineixen a partir de la funció f i compleixen

$$\kappa_i = f(t_n + c_i h, x_n + h \sum_{j=1}^{m} a_{i,j} \kappa_j), \quad i = 1, \ldots, m. \tag{1.1.2}$$

Tal com demostra A. Iserles [10], p. 39, i més tard referenciat en D. Griffiths, D. Higham [9], podem assumir que

$$c_i = \sum_{j=1}^{m} a_{i,j}, \quad i = 1, \ldots, m.$$

El conjunt de variables lliures presents en aquest mètode solen expressar-se de manera mnemotècnica en forma de les taules de Butcher.

$$\begin{array}{c|cccc} c_1 & a_{1,1} & a_{1,2} & \ldots & a_{1,m} \\ c_2 & a_{2,1} & a_{2,2} & \ldots & a_{2,m} \\ \vdots & \vdots & \vdots & & \vdots \\ c_m & a_{m,1} & a_{m,2} & \ldots & a_{m,m} \\ \hline & b_1 & b_2 & \ldots & b_m \end{array}$$

Podem diferenciar dos tipus de mètodes de Runge-Kutta, els *implícits* i els *explícits*. La diferència està en el sistema d'equacions que es genera en calcular els valors de κ_i utilitzant la fórmula (1.1.2).

En cas que puguem determinar els resultats directament sense necessitat de resoldre equacions no-linears, és a dir, els valors c_i i $a_{i,j}$ compleixen,

$$\begin{aligned} c_1 &= 0, \\ c_i &= \sum_{j=1}^{i-1} a_{i,j}, \quad i = 2, \ldots, m, \end{aligned} \tag{1.1.3}$$

parlarem d'un mètode de Runge-Kutta *explícit*. Aquest és el més comú i el que estudiarem amb més detall. En concret, els valors de $a_{i,j} = 0$ per tot $j \geq i$.

D'altra banda, si necessitem un altre algoritme o algun tipus de càlcul per resoldre aquestes m equacions, es tractarà d'un mètode de Runge-Kutta *implícit*.

Durant l'estudi d'aquests mètodes, si no s'especifica el contrari, es tractarà d'un mètode explícit. La taula de Butcher corresponent serà,

$$\begin{array}{c|cccc} 0 & 0 & & & \\ c_2 & a_{2,1} & & & \\ c_3 & a_{3,1} & a_{3,2} & & \\ \vdots & \vdots & & \ddots & \\ c_m & a_{m,1} & a_{m,2} & \ldots & a_{m,4} \\ \hline & b_1 & b_2 & \ldots & b_m \end{array}$$

També podem simplificar la fórmula (1.1.2) que permet calcular les κ_i

$$\begin{aligned} \kappa_1 &= f(t_n, x_n), \\ \kappa_i &= f(t_n + c_i h, x_n + h \sum_{j=1}^{i-1} a_{i,j}\kappa_j), \quad i = 2, \ldots, m. \end{aligned} \tag{1.1.4}$$

Finalment, una part important serà l'estudi de l'ordre de cada mètode. Aquest vindrà determinat per l'error local de truncament que cometem en aproximar $x(t_{n+1})$.
Definirem l'error de truncament ε_{n+1} d'un mètode de Runge-Kutta en $t = t_{n+1}$ com la diferència, en valor absolut, entre el valor exacte $x(t_{n+1})$ i l'aproximació que hem trobat x_{n+1},

$$\varepsilon_{n+1} = |x(t_{n+1}) - x_{n+1}|$$

suposant que, en el pas previ, es complia $x_n = x(t_n)$. Si tenim $\varepsilon_{n+1} = \mathcal{O}(h^{p+1})$, direm que el mètode té ordre p.

1.2 Casos particulars Runge-Kutta

Per entendre més profundament com funcionen els mètodes de Runge-Kutta i com es calcula el seu ordre, estudiarem alguns dels mètodes amb menor nombre d'avaluacions de la funció.

1.2.1 Runge-Kutta amb una avaluació del camp

Es tracta del mètode de Runge-Kutta més senzill. Per calcular l'error d'aquest mètode, hem de trobar les expressions de les funcions x_{n+1} i $x(t_{n+1})$. Comencem per calcular la de x_{n+1}. Seguint les fórmules generals descrites anteriorment, i amb $m = 1$,

$$\begin{aligned} x_{n+1} &= x_n + hb_1\kappa_1, \\ \kappa_1 &= f(t_n, x_n). \end{aligned}$$

Aleshores,

$$x_{n+1} = x_n + hb_1 f(t_n, x_n) \tag{1.2.1}$$

Un cop trobada l'expressió general de x_{n+1}, buscarem la de $x(t_{n+1})$. Cal tenir en compte que $t_{n+1} = t_n + h$. Aplicant la fórmula de Taylor d'ordre 2, enunciada a l'Apèndix A.1.1, a la funció $x(t_n + h)$, obtenim

$$\begin{aligned} x(t_n + h) &= x(t_n) + hx'(t_n) + \frac{1}{2!}h^2x''(t_n) + \mathcal{O}(h^3) \\ &= x(t_n) + hf(t_n, x_n) + \frac{1}{2}h^2(f_t(t_n, x_n) + x'(t_n)f_x(t_n, x_n)) + \mathcal{O}(h^3) \\ &= x(t_n) + hf(t_n, x_n) + \frac{1}{2}h^2(f_t(t_n, x_n) + f(t_n, x_n)f_x(t_n, x_n)) + \mathcal{O}(h^3), \end{aligned} \tag{1.2.2}$$

on hem tingut en compte que $x'(t) = f(t, x)$ i la regla de la cadena. Com $x = x(t)$, llavors $f'(t, x) = f_t(t, x) + f_x(t, x) \cdot f(t, x)$.

Finalment, calculem l'error local de truncament

$$\begin{aligned} \varepsilon_{n+1} &= x(t_{n+1}) - x_{n+1} \\ &= h(1 - b_1)f(t_n, x_n) + \frac{1}{2}h^2(f_t(t_n, x_n) + f(t_n, x_n)f_x(t_n, x_n)) + \mathcal{O}(h^3). \end{aligned} \tag{1.2.3}$$

És clar que com menor sigui l'error, més precís serà el mètode. D'aquesta manera, se sol utilitzar el Runge-Kutta amb $b_1 = 1$ per tal que el primer terme de l'expressió (1.2.3) sigui igual a zero, llavors $\varepsilon_{n+1} = \mathcal{O}(h^2)$ i, per tant, el mètode tingui ordre 1.
Cal remarcar que com el segon terme de l'equació (1.2.3) mai serà zero, el mètode de Runge-Kutta amb una sola avaluació del camp ($m = 1$) serà, com a molt, d'ordre 1.

Per $b_1 = 1$, les equacions que descriuen l'òrbita són les mateixes que les del mètode d'*Euler*.

1.2.2 Runge-Kutta amb dues avaluacions del camp

Seguint amb el tipus de raonament que hem utilitzat per al mètode de Runge-Kutta 1, trobarem les expressions de x_{n+1} i de $x(t_{n+1})$.

El càlcul de $x(t_{n+1})$ es anàleg al que hem fet prèviament en (1.2.2). Tanmateix, en aquest cas agafarem un terme més de la sèrie de Taylor per poder calcular correctament l'ordre d'aquest mètode. Per simplificar la notació, utilitzarem $f_n := f(t_n, x_n)$ i considerarem les derivades avaluades en el punt (t_n, x_n). El desenvolupament dels càlculs explícits d'aquestes derivades es poden trobar a l'Apèndix A.2. L'expressió de $x(t_{n+1})$ resulta de

$$\begin{aligned} x(t_{n+1}) &= x(t_n) + hx'(t_n) + \frac{1}{2}h^2x''(t_n) + \frac{1}{3!}h^3x^{(3)}(t_n) + \mathcal{O}(h^4) \\ &= x(t_n) + hf_n + \frac{1}{2}h^2(f_t + f_nf_x) + \frac{1}{6}h^3(f_{tt} + 2f_{tx}f_n + f_{xx}f_n^2 + f_x(f_t + f_nf_x)) + \mathcal{O}(h^4). \end{aligned} \tag{1.2.4}$$

Per calcular x_{n+1} amb $m = 2$, utilitzarem les fórmules generals de Runge-Kutta

$$x_{n+1} = x_n + h \cdot (b_1\kappa_1 + b_2\kappa_2),$$

amb

$$\begin{aligned} \kappa_1 &= f(t_n, x_n), \\ \kappa_2 &= f(t_n + c_2h, x_n + h \cdot a_{2,1}\kappa_1) \\ &= f(t_n + ah, x_n + ahf(t_n, x_n)). \end{aligned}$$

Hem utilitzat la igualtat (1.1.3) deduint així que $c_2 = a_{2,1}$ i, per simplificar la notació, notarem $a := c_2 = a_{2,1}$.

$$x_{n+1} = x_n + h \cdot [b_1f_n + b_2f(t_n + ah, x_n + ahf_n))]. \tag{1.2.5}$$

Cal simplificar la funció $f(t_n + ah, x_n + ahf_n)$. La sèrie de Taylor d'ordre 3 per una funció genèrica $f(t+ah, x+bh)$, on totes les funcions les considerarem avaluades en el punt (t, x), correspon a

$$f(t + ah, x + bh) = f + h(af_t + bf_x) + \frac{1}{2}h^2(a^2f_{tt} + 2abf_{tx} + b^2f_{xx}) + \mathcal{O}(h^3).$$

Si l'apliquem en el càlcul de κ_2 amb $b = a \cdot f_n$, obtenim

$$\begin{aligned} \kappa_2 &= f_n + h(af_t + af_nf_x) + \frac{1}{2}h^2(a^2f_{tt} + 2a^2f_nf_{tx} + (af_n)^2f_{xx}) + \mathcal{O}(h^3) \\ &= f_n + ah(f_t + f_nf_x) + \frac{1}{2}(ah)^2(f_{tt} + 2f_nf_{tx} + f_n^2f_{xx}) + \mathcal{O}(h^3). \end{aligned}$$

Tornant a (1.2.5),

$$\begin{aligned} x_{n+1} &= x_n + h \cdot \left[b_1f_n + b_2 \cdot \left(f_n + ah(f_t + f_nf_x) + \frac{1}{2}(ah)^2(f_{tt} + 2f_nf_{tx} + f_n^2f_{xx})\right)\right] + \mathcal{O}(h^3) \\ &= x_n + hb_1f_n + hb_2f_n + ab_2h^2(f_t + f_nf_x) + \frac{1}{2}a^2b_2h^3(f_{tt} + 2f_nf_{tx} + f_n^2f_{xx}) + \mathcal{O}(h^4) \\ &= x_n + hf_n(b_1 + b_2) + ab_2h^2(f_t + f_nf_x) + \frac{1}{2}a^2b_2h^3(f_{tt} + 2f_nf_{tx} + f_n^2f_{xx}) + \mathcal{O}(h^4). \end{aligned}$$

Finalment, assumim que $x(t_n) = x_n$ i calculem l'error de truncament per saber-ne l'ordre,

$$\begin{aligned} \varepsilon_{n+1} &= x(t_{n+1}) - x_{n+1} \\ &= hf_n(1 - b_1 - b_2) + h^2\left((\frac{1}{2} - ab_2)(f_t + f_nf_x)\right) + \\ &\quad + h^3\left((\frac{1}{6} - \frac{1}{2}a^2b_2)(f_{tt} + 2f_nf_{tx} + f_n^2f_{xx}) + \frac{1}{6}(f_x(f_t + f_nf_x))\right) + \mathcal{O}(h^4). \end{aligned} \tag{1.2.6}$$

Així doncs, si es compleix $b_1 + b_2 = 1$, el mètode tindrà ordre 1 per qualsevol valor d'a, ja que $\varepsilon_{n+1} = \mathcal{O}(h^2)$. En cas que també tinguem $\frac{1}{2} = ab_2$, el mètode passarà a tenir ordre 2. Com hem dit abans, ens interessa que l'ordre sigui el més gran possible, ja que això implica que l'error és menor. És clar que el terme d'ordre h^3 mai serà nul, així que el màxim que podem obtenir és ordre 2. D'aquesta manera, sempre que utilitzem el mètode de Runge-Kutta amb $m = 2$, també forçarem que $b_1 + b_2 = 1$ i $ab_2 = \frac{1}{2}$.

Fixat $b_2 = \theta$, la família de mètodes de Runge-Kutta d'ordre 2, amb $\theta \neq 0$, té una taula de Butcher del tipus

$$\begin{array}{c|cc} 0 & 0 & \\ a & a & 0 \\ \hline & 1-\theta & \theta \end{array}$$

Els exemples més utilitzats són el mètode d'Euler millorat, amb $\theta = \frac{1}{2}$, i el mètode d'Euler modificat, on $\theta = 1$.

1.2.3 Runge-Kutta amb quatre avaluacions del camp

Es considera el mètode estàndard de Runge-Kutta i és el més utilitzat. Un dels motius és perquè és l'últim dels mètodes que permet tenir el mateix nombre d'avaluacions del camp i ordre, tal com hem enunciat prèviament. Aquest fet permet utilitzar un mètode amb el màxim ordre possible i el mínim nombre d'avaluacions. La taula de Butcher que presenta els números òptims (Lambert [13], p. 178), és

$$\begin{array}{c|cccc} 0 & 0 & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

1.3 Runge-Kutta-Fehlberg

Fins ara el valor de la variable de pas h s'escollia de manera que l'interval d'integració quedés dividit en subintervals de la mateixa longitud, és a dir, treballàvem amb un pas h fix. Això, com bé hem comentat, pot provocar alguns errors com la pèrdua de precisió si escollim un pas massa gran, o un augment en l'error provocat per fer moltes operacions aritmètiques si escollim un pas petit. Com que volem evitar aquests problemes, hem de buscar una alternativa que ens permeti modificar h i utilitzar una fórmula que calculi a cada iteració el valor més adequat.

Un mètode que ens permet fer això és el de Runge-Kutta-Fehlberg. Aquest consisteix en una modificació del Runge-Kutta que hem explicat anteriorment, afegint que permet calcular el pas h de manera òptima en cada punt de l'interval. Donat que estem treballant amb mètodes d'un sol pas, la modificació del valor de h no afectarà el càlcul de la resta de valors.

Aquest mètode d'integració fa servir dos mètodes de Runge-Kutta, amb nombres d'avaluacions del camp consecutius. En el nostre cas, considerarem els mètodes RK4 i RK5, però el procediment és el mateix per dos graus qualssevol diferents. El cost de l'algoritme no augmenta pel fet que en calcular els valors de RK5, obtenim tots els necessaris per al mètode RK4.

La idea general és la següent: de la mateixa manera que en el Runge-Kutta simple, donats un $x(t_n)$ i un pas h_n concret, volem aproximar el valor de $x(t_{n+1})$. Calculem $\bar{x}_{n+1}$ utilitzant RK4 i $\hat{x}_{n+1}$ utilitzant RK5. Si aquests dos valor són prou pròxims, és a dir, si

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

essent δ una tolerància fixada, llavors acceptem $\hat{x}_{n+1}$ com una bona aproximació de $x(t_{n+1})$. Agafem el valor resultant del mètode RK5, ja que es pot considerar és una millor aproximació pel fet de tenir menor ordre i, per tant, ser més precís que l'obtingut amb RK4. En cas que la diferència entre els dos valors sigui major que δ, tornem a calcular tant h_n, amb una fórmula que veurem a continuació, com els valors $\hat{x}_{n+1}$ i $\bar{x}_{n+1}$ amb el nou h_n.

La fórmula que permet fer aquest procés automàticament, és la següent,

$$|h_n| = 0.9 \cdot |h| \sqrt[5]{\frac{tol}{||\bar{x}_{n+1} - \hat{x}_{n+1}||}} \tag{1.3.1}$$

Observem que el valor h_n depèn únicament de la tolerància, de la variable h prèvia i de l'error entre els dos Runge-Kutta. El valor 0.9 s'utilitza com a control de pas, per evitar que h_n augmenti considerablement. Per tant, donat que h_n no té un cost computacional elevat, no suposarà un problema en comparació a utilitzar un pas fix.

Vegem d'on surt aquesta fórmula seguint l'esquema de la demostració que trobem en D. Griffiths, D. Higham [9], p.147.

Considerarem l'error de truncament del mètode de Runge-Kutta-Fehlberg $\varepsilon_{n+1} = \hat{x}_{n+1} - \bar{x}_{n+1}$. Com que estem treballant amb RK4 i RK5, l'error serà de l'ordre de h^5 i es pot expressar com

$$\varepsilon_{n+1} = H(t_n)h_n^5 + \mathcal{O}(h_n)^6$$

per alguna funció $H(t_n)$.

Calculant els valors x_{n+1} i t_{n+1} amb una h fixada, sovint la que hem utilitzat en el pas anterior, podrem estimar l'error que es cometrà. Anomenarem $\hat{\varepsilon}_{n+1}$ a aquesta aproximació de l'error ε_{n+1}. Ens podem trobar en dues situacions diferents:

- $\hat{\varepsilon}_{n+1} < tol$, llavors podríem haver agafat un valor de h_n major.
- $\hat{\varepsilon}_{n+1} > tol$ implica que el valor de h_n és massa gran i l'aproximació que hem fet no és vàlida.

El que hem d'aconseguir és que la nostra variable nova h_n forci que $tol = \varepsilon_{n+1}$. D'aquesta manera, ens assegurem que l'error comès no presenti cap dels problemes anteriors.

Com que $\hat{\varepsilon}_{n+1}$ és una aproximació de ε_{n+1}, segons la fórmula descrita anteriorment, podem assumir que $\hat{\varepsilon}_{n+1} \approx H(t_n)h^5$. Aïllant $H(t_n)$ d'aquesta equació, obtenim

$$H(t_n) \approx \frac{\hat{\varepsilon}_{n+1}}{h^5}$$

Així doncs, ajuntant les fórmules, obtenim $tol = \varepsilon_{n+1} \approx H(t_n)h_n \approx \frac{\hat{\varepsilon}_{n+1}}{h^5}h_n^5$. Finalment, aïllant h_n,

$$h_n = h\sqrt[5]{\frac{tol}{\hat{\varepsilon}_{n+1}}} = \sqrt[5]{\frac{tol}{||\bar{x}_{n+1} - \hat{x}_{n+1}||}} \tag{1.3.2}$$

Un cop vist com es calcula el pas del següent cas, només queda veure les fórmules exactes del mètode de Runge-Kutta-Fehlberg. Tal com va calcular Fehlberg (1969) donada una h_n, els valors òptims del mètode són

$$\begin{aligned}
\kappa_1 &= h_n f\left(t_n, x_n\right) \\
\kappa_2 &= h_n f\left(t_n + \frac{1}{4}h_n, x_n + \frac{1}{4}\kappa_1\right) \\
\kappa_3 &= h_n f\left(t_n + \frac{3}{8}h_n, x_n + \frac{3}{32}\kappa_1 + \frac{9}{32}\kappa_2\right) \\
\kappa_4 &= h_n f\left(t_n + \frac{12}{13}h_n, x_n + \frac{1932}{2197}\kappa_1 - \frac{7200}{2197}\kappa_2 + \frac{7296}{2197}\kappa_3\right) \\
\kappa_5 &= h_n f\left(t_n + h_n, x_n + \frac{439}{216}\kappa_1 - 8\kappa_2 + \frac{3680}{513}\kappa_3 - \frac{845}{4104}\kappa_4\right) \\
\kappa_6 &= h_n f\left(t_n + \frac{1}{2}h_n, x_n - \frac{8}{27}\kappa_1 + 2\kappa_2 - \frac{3544}{2565}\kappa_3 + \frac{1859}{4104}\kappa_4 - \frac{11}{40}\kappa_5\right),
\end{aligned} \tag{1.3.3}$$

Aquestes κ ens permeten calcular les aproximacions de $x(t_{n+1})$.

$$\begin{aligned}
\bar{x}_{n+1} &= x_n + \frac{25}{216}\kappa_1 + \frac{1408}{2565}\kappa_3 + \frac{2197}{4104}\kappa_4 - \frac{1}{5}\kappa_5, \\
\hat{x}_{n+1} &= x_n + \frac{16}{135}\kappa_1 + \frac{6656}{12825}\kappa_3 + \frac{28561}{56430}\kappa_4 - \frac{9}{50}\kappa_5 + \frac{2}{55}\kappa_6
\end{aligned}$$

Així doncs, la taula de Butchler resultant és la següent

0	0					
$\frac{1}{4}$	$\frac{1}{4}$					
$\frac{3}{8}$	$\frac{3}{32}$	$\frac{9}{32}$				
$\frac{12}{13}$	$\frac{1932}{2197}$	$\frac{-7200}{2197}$	$\frac{7296}{2197}$			
1	$\frac{439}{216}$	-8	$\frac{3680}{513}$	$\frac{-845}{4104}$		
$\frac{1}{2}$	$\frac{-8}{27}$	2	$\frac{-3544}{2565}$	$\frac{1859}{4104}$	$\frac{-11}{40}$	
	$\frac{25}{216}$	0	$\frac{1408}{2565}$	$\frac{2197}{4104}$	$\frac{-1}{5}$	
	$\frac{16}{135}$	0	$\frac{6656}{12825}$	$\frac{28561}{56430}$	$\frac{-9}{50}$	$\frac{2}{55}$

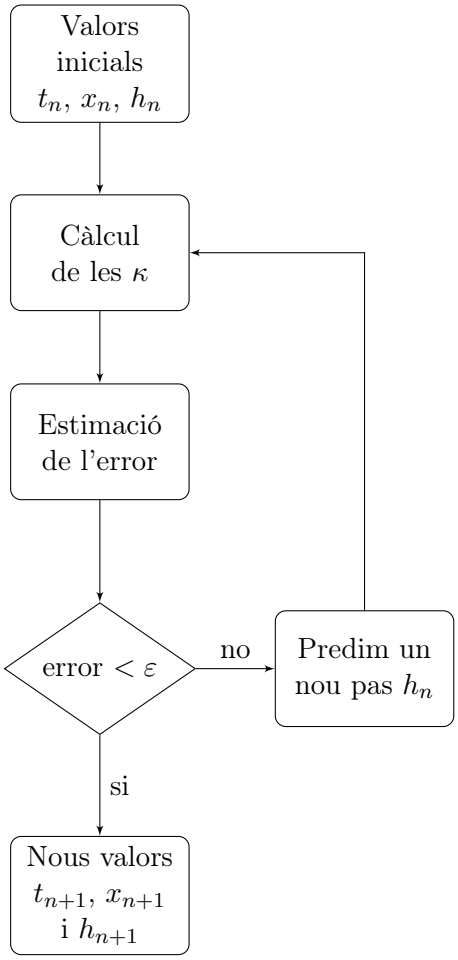

Figura 1.1: Diagrama de flux per RKF45

Com hem dit abans, l'estimació de l'error entre aquestes dues aproximacions es pot calcular utilitzant la següent fórmula i tenint en compte els valors obtinguts en (1.3.3).

$$||\hat{x}_{n+1} - \bar{x}_{n+1}|| = \left|\left|\frac{1}{360}\kappa_1 - \frac{128}{4275}\kappa_3 - \frac{2197}{75240}\kappa_4 + \frac{1}{50}\kappa_5 + \frac{2}{55}\kappa_6\right|\right|. \tag{1.3.4}$$

Un cop definides les equacions, podem crear el codi que ens permeti simular aquest procediment.

1.3.1 Implementació

Per entendre millor quin serà el procediment a seguir a l'hora de programar el mètode de Runge-Kutta-Fehlberg, vegem la figura 1.1. Aquesta representa el flux de processos que es produiran en la funció que s'encarregarà de programar el mètode.

La funció rebrà una sèrie de dades inicials, entre elles t_n, x_n i h_n, juntament amb

- una variable que conté el valor on s'acaba l'interval (`atf`),
- un punter que retorna l'error comès després de cada iteració (`aer`),

– un punter a una funció que conté l'equació del camp (`*ode`),

– la tolerància (`tol`) i

– una variable que permetrà decidir si volem utilitzar el control de pas o calcular h_n de la manera descrita anteriorment (`sc`).

Un cop la funció rebi totes aquestes dades, calcularem les diferents κ_i, $i = 1, \ldots, 6$ utilitzant una funció auxiliar, i estimarem l'error comès utilitzant la fórmula (1.3.4). Si aquest error és major que la tolerància definida, tornarem a calcular h_n i els valors de κ fins que l'error sigui menor.

Així doncs, un cop donat com a vàlid l'error i les noves κ_i, ja podem obtenir els nous valors t_{n+1}, x_{n+1} i h_{n+1}.

Finalment, farem una última verificació que consistirà a comprovar si $t_n + h_n > atf$, és a dir, comprovarem que si tornem a iterar un últim cop tot el procés, t_n no sigui major al temps final al qual volem arribar. En cas que això sigui cert, farem un últim càlcul de l'error i de les κ, i adaptarem els valors t_n i h_n per tal que coincideixin amb el final.

Un cop assolit aquest punt, el procés haurà acabat.

Per comprovar que el programa funciona, vegem el resultat d'aplicar aquest mètode a les equacions d'un oscil·lador harmònic.

$$\begin{cases} x'(t) = y(t) \\ y'(t) = -x(t) \end{cases}$$

amb les condicions inicials $(x_0, y_0) = (1, 0)$. Aquest sistema té per solució $x(t) = \cos t$, $(t) = -\sin t$, és a dir, es tracta d'un cercle de radi 1, que compleix $x^2 + y^2 = 1$. En el nostre cas, integrarem les equacions per $0 \leq t \leq 2\pi$. El resultat correspon a la figura 1.2.

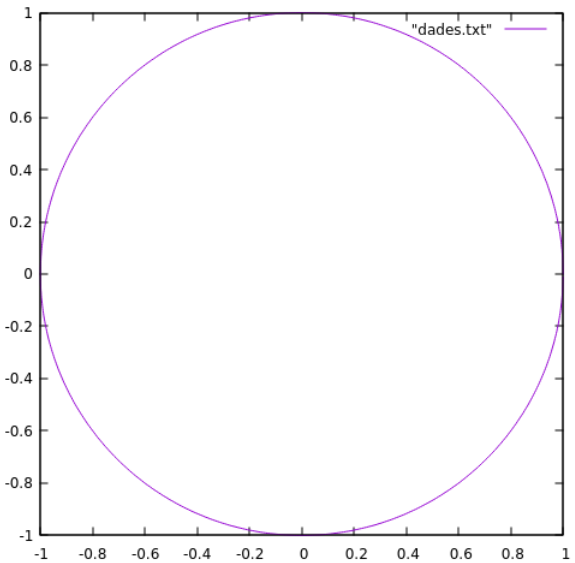

Figura 1.2: Integració RKF45

Capítol 2

Sèries formals de potències

En aquest capítol ens centrarem a definir com treballar amb aritmètica de sèries. Per tal de fer-ho, haurem de definir i especificar les fórmules de les operacions aritmètiques bàsiques que utilitzarem més endavant a l'hora d'aplicar el transport de derivades i la diferenciació automàtica.

Definició 2.0.1. *Una sèrie formal és una suma infinita que es considera independent de qualsevol noció de convergència. No obstant això, conserva les propietats bàsiques de les operacions aritmètiques de sèries.*

Les sèries formals són una abstracció de les sèries de potències analítiques, amb les quals estem acostumats a treballar. Aquestes estan formades per elements purament algebraics, sense fer ús de cap propietat que impliqui la representació de tal sèrie. D'aquesta manera, podem utilitzar tota classe de sèries, sense saber si existeixen o sense necessitat de definir les seves propietats analítiques.

Definició 2.0.2. *Una sèrie formal de potències és un tipus concret de sèrie formal, on els termes de la suma són $a_n s^n$, amb s^n la n-èssima potència de la variable s i a_n el coeficient corresponent. Anomenarem seqüència de coeficients als elements $a_n,\ n = 0, \ldots, \infty$.*

Definició 2.0.3. *Direm que dues sèries formals són iguals si la seva seqüència de coeficients coincideix exactament.*

Cal puntualitzar que la variable s s'utilitza per diferenciar els termes de la sèrie, és a dir, no es tracta com una incògnita de la mateixa manera que en els polinomis. És per això que moltes vegades es consideren una generalització dels polinomis per un grau infinit.

La principal referència del tractament de sèries de potències formals recau en L.Euler. Entre altres coses, va ser qui va trobar l'expressió com a sèries de potències de la funció $\arctan z$ i de e. Un llibre més recent que també treballa aquest àmbit amb profunditat és el de D. E. Knuth [12].

2.1 Sèries de potències formals d'una variable

A continuació s'estudiaran les sèries de potències formals d'una sola variable i les seves operacions aritmètiques. La notació durant aquest apartat serà

$$A = \sum_{n\geq 0} a_n s^n, \quad B = \sum_{n\geq 0} b_n s^n, \quad C = \sum_{n\geq 0} c_n s^n \tag{2.1.1}$$

Operacions aritmètiques bàsiques

Les operacions aritmètiques bàsiques, com la suma i resta, de dues sèries es defineixen de manera trivial

$$A \pm B = \sum_{n\geq 0} (a_n \pm b_n) s^n$$

El producte de dues sèries infinites $A \cdot B$ es pot calcular utilitzant el producte de Cauchy,

$$\left(\sum_{i\geq 0} a_i s^i\right) \cdot \left(\sum_{j\geq 0} b_j s^j\right) = \sum_{k\geq 0} c_k s^k, \text{ amb } c_k = \sum_{l=0}^{k} a_l b_{k-l}$$

el qual expressat en una sola equació equival a

$$\left(\sum_{i\geq 0} a_i s^i\right) \cdot \left(\sum_{j\geq 0} b_j s^j\right) = \sum_{k\geq 0} \left(\sum_{l=0}^{k} a_l b_{k-l}\right) s^k$$

La demostració d'aquest concepte és senzilla si utilitzem un pas n fix, tractem les sèries formals de potències com si treballéssim amb polinomis de grau infinit, fent ús de la propietat distributiva.

Finalment, el producte per escalar també es defineix de manera trivial.

$$\alpha \cdot A = \sum_{n\geq 0} \alpha \cdot a_n s^n.$$

Quocient

Donada una sèrie formal de potències B complint $b_0 \neq 0$, l'expressió de la sèrie de potències resultant $C = A/B$ per $n \geq 0$ resulta

$$c_n = \frac{1}{b_0}\left(a_n - \sum_{j=1}^{n} b_j c_{n-j}\right) \tag{2.1.2}$$

La deducció d'aquesta fórmula s'obté a partir de fer $A = BC$

$$\sum_{n\geq 0} a_i = \sum_{n\geq 0} \left(\sum_{j=0}^{n} b_j c_{k-j}\right)$$

i fixant la variable n obtenim

$$a_n = \sum_{j=0}^{n} b_j c_{n-j} = b_0 c_n + \sum_{j=1}^{n} b_j c_{n-j}$$

Finalment, aïllant c_n obtenim l'equació (2.1.2).

Derivades

Donat que les sèries de potències no deixen de ser un tipus de funcions, també podem considerar-ne les derivades. Cal tenir en compte que estem treballant amb sèries de potències formals, com hem definit a l'inici d'aquest apartat. D'aquesta manera, les derivades també les haurem de considerar formalment.

Definim la derivada formal d'una sèrie de potències de la mateixa manera que definim el terme analític, però en aquest cas sense relacionar-la amb la diferenciabilitat de la funció. Per derivar una sèrie de potències formal, ens limitarem a imitar el comportament del procés de derivar sèries de potències no formals. Notada com DA o A', la derivada serà del tipus

$$DA = A' = \sum_{n\geq 1} a_n n s^{n-1}$$

DA compleix les mateixes propietats que les derivades habituals, és a dir, és una funció linear,

$$(aA + bB)' = a \cdot A' + b \cdot B'$$

compleix la regla del producte,

$$(AB)' = A \cdot B' + A' \cdot B$$

i la regla de la cadena

$$(A \circ B)' = (A' \circ B) \cdot B'.$$

La demostració d'aquestes tres propietats és anàloga. Veiem la del producte. Siguin A i B les sèries definides a l'inici d'aquest apartat, i C el resultat de fer el producte $C = AB$, hem de demostrar $C' = AB' + A'B$.

$$\begin{aligned} AB' &= \sum_{n\geq 0} a_n s^n \sum_{n\geq 1} n b_n s^{n-1} \\ &= \sum_{n\geq 0} a_n s^n \sum_{n\geq 0} (n+1) b_{n+1} s^n \\ &= \sum_{n\geq 0} \sum_{k=0}^{n} (n+1-k) a_k b_{n+1-k} s^n \end{aligned}$$

Anàlogament,

$$\begin{aligned} A'B &= \sum_{n\geq 0} n a_n s^{n-1} \sum_{n\geq 0} b_n s^n \\ &= \sum_{n\geq 0} (n+1) a_{n+1} s^n \sum_{n\geq 0} b_n s^n \\ &= \sum_{n\geq 0} \sum_{k=0}^{n} (k+1) a_{k+1} b_{n-k} s^n. \end{aligned}$$

Si ens centrem únicament en els coeficients de s^n,

$$\begin{aligned}
AB' + A'B &= \sum_{k=0}^{n}(n-k+1)a_k b_{n-k+1} + \sum_{k=0}^{n}(k+1)a_{k+1}b_{n-k} \\
&= \sum_{k=0}^{n}(n-k+1)a_k b_{n-k+1} + \sum_{k=0}^{n-1}(k+1)a_{k+1}b_{n-k} + (n+1)a_{n+1}b_0 \\
&= (n+1)a_0 b_{n+1} + \sum_{k=1}^{n}(n-k+1)a_k b_{n-k+1} + \sum_{k=1}^{n} k a_k b_{n-k+1} + (n+1)a_{n+1}b_0 \\
&= (n+1)a_0 b_{n+1} + \sum_{k=1}^{n}(n+1)a_k b_{n-k+1} + (n+1)a_{n+1}b_0 \\
&= (n+1)\sum_{k=0}^{n+1} a_k b_{n+1-k} \\
&= (n+1)c_{n+1} = C'
\end{aligned}$$

L'últim pas correspon a la definició de C', tenint en compte que $C = AB$. Donat que les dues expressions coincideixen, queda demostrat que $C' = AB' + A'B$. A partir d'ara, podrem utilitzar la noció de derivada d'una sèrie de potències formal amb normalitat.

Potència d'una sèrie

Per trobar l'expressió de A^α, considerem $\alpha \in \mathbb{R}$ i $\alpha \neq 0, 1$, definim $B = A^\alpha$. Els casos per $\alpha = 0, 1$ són trivials. Derivant l'equació $B = A^\alpha$, obtenim $B' = \alpha A^{\alpha-1}A'$ per la regla de la cadena. Multiplicant per A tota la igualtat, i tenint en compte que $B = A^\alpha$, obtenim

$$B'A = \alpha BA'$$

Aquest resultat expressat com a sèries formals de potències, correspon a

$$\left(\sum_{n\geq 1} n b_n s^{n-1}\right)\left(\sum_{n\geq 0} a_n s^n\right) = \alpha\left(\sum_{n\geq 0} b_n s^n\right)\left(\sum_{n\geq 1} n a_n s^{n-1}\right)$$

Aplicant el producte de sèries definit anteriorment,

$$\sum_{n\geq 0}\sum_{i=0}^{n} k a_i b_{n-i} = \alpha \sum_{n\geq 0}\sum_{i=0}^{n} k b_i a_{n-i}$$

Si ens centrem en estudiar el terme de grau $n-1$,

$$\sum_{i=0}^{n} i b_i a_{n-i} = \alpha \sum_{i=0}^{n}(n-i)a_{n-i}b_i \tag{2.1.3}$$

i aïllem b_n, obtenim l'expressió de B per calcular els termes de $B = A^\alpha$

$$b_n = \frac{1}{n a_0}\sum_{i=0}^{n-1}[\alpha n - (\alpha+1)i]a_{n-i}b_i \tag{2.1.4}$$

Amb aquesta expressió, podem calcular els elements de b_n per $n \geq 1$, ja que el cas inicial $n = 0$ és $b_0 = a_0^\alpha$.

En concret, podem aplicar aquesta fórmula als casos $\alpha = -1$ i $\alpha = \frac{1}{2}$, és a dir, per computar la inversa i l'arrel d'una sèrie formal de potències.

Funcions trigonomètriques

Si definim $B = h(A)$, per alguna funció h, també es complirà $B' = h'(A)A'$, per les propietats de les derivades vistes anteriorment.

Ens centrarem a trobar les expressions de les sèries $B = h(A) = \sin(A)$ i $C = \cos(A)$. Si computem les derivades de B' i C', obtenim

$$\begin{aligned} B = \sin(A) &\to B' = \cos(A)A' \\ C = \cos(A) &\to C' = -\sin(A)A'. \end{aligned}$$

Podem simplificar aquestes equacions de la següent manera

$$\begin{aligned} B' &= CA' \\ C' &= -BA'. \end{aligned}$$

Ajuntant les fórmules de la derivada d'una sèrie de potències formal, el producte de dues sèries de potències i fixant un pas n, obtenim les equacions següents, a partir de les quals podrem deduir els coeficients de B i C.

$$\begin{aligned} nb_n &= \left(\sum_{k=0}^{n} c_k\right)\left(\sum_{k=1}^{n} ka_k\right) = \sum_{k=1}^{n} ka_k c_{n-k} \\ nc_n &= -\left(\sum_{k=0}^{n} b_k\right)\left(\sum_{k=1}^{n} ka_k\right) = -\sum_{k=1}^{n} ka_k b_{n-k} \end{aligned}$$

Aïllant les variables b_n i c_n per obtenir-ne les expressions,

$$\begin{aligned} b_n &= \frac{1}{n}\sum_{k=1}^{n} ka_k c_{n-k} \\ c_n &= \frac{-1}{n}\sum_{k=1}^{n} ka_k b_{n-k} \end{aligned} \tag{2.1.5}$$

Aquestes fórmules ens permeten computar el sin i cos d'una sèrie de potències formal. L'únic inconvenient és que cal utilitzar-les alhora, ja que depenen una de l'altra.

2.2 Sèries de potències formals de diverses variables

Les sèries de potències formals de m variables s'acostumen a anotar de la següent manera,

$$A = \sum_{n\geq 0}\sum_{|k|=n} a_k s^k$$

on $k = (k_1, \ldots, k_m) \in \mathbb{N}^m$, $|k| = k_1 + \cdots + k_m$ i $s^k = s_1^{k_1} \ldots s_m^{k_m}$.

El concepte d'aquesta notació és generar una subsèrie de potències formals on els graus dels coeficients sumen n, per a tot $n \geq 0$.

Els algoritmes de les operacions bàsiques de sèries de potències formals de diverses variables no divergeixen excessivament dels que hem estudiat en l'apartat anterior per sèries d'una variable. No obstant això, hi ha algun procediment que cal puntualitzar per tal de treballar-hi còmodament en els següents capítols.

Funcions aritmètiques bàsiques

En el cas de la suma i la resta, podem utilitzar la definició bàsica vista anteriorment.

$$A \pm B = \sum_{n \geq 0} \sum_{|k|=n} (a_k \pm b_k) s^k$$

El producte per un escalar és anàleg

$$\alpha \cdot A = \sum_{n \geq 0} \sum_{|k|=n} \alpha \cdot a_k s^k$$

El producte de dues sèries el definirem aplicant un canvi de variable de s_j per $s_j z$. La variable z s'utilitza per poder separar els dos sumatoris i alhora mantenir les propietats de les sèries de potències formals. La forma de la sèrie resultant correspon a,

$$A = \sum_{n \geq 0} A_n z^n, \text{ amb } A_n = \sum_{|k|=n} a_k s^k. \tag{2.2.1}$$

Aquest canvi ens permet treballar únicament amb el sumatori exterior, ja que la variable z l'acabarem igualant a 1 per obtenir la forma inicial de la sèrie. El fet de poder treballar amb el sumatori extern ens facilita les fórmules, ja que llavors les fórmules són anàlogues al cas d'una variable.

Llavors, el producte de dues sèries de potències formals de diverses variables resulta

$$A_n \cdot B_n = \sum_{k=0}^{n} A_k B_{n-k}$$

per un n fixat i A_n una sèrie del tipus $A_n = \sum_{|k|=n} a_k s^k$.

Quocient

Per tal de calcular el quocient $C = A/B$, on les tres sèries són formals de diverses variables, aplicarem també el canvi (2.2.1).

Tenim $C = A/B$, donades les sèries A i B.

$$\sum_{n \geq 0} C_n z^n = \sum_{n \geq 0} A_n z^n \Big/ \sum_{n \geq 0} B_n z^n$$

Podem aplicar el resultat vist en el cas d'una variable (2.1.2) i obtenir

$$C_n = \frac{1}{B_0}\left(A_n - \sum_{j=1}^{n} B_j C_{n-j}\right) \tag{2.2.2}$$

La principal diferència resulta que en aquest cas les variables A_n, B_n i C_n corresponen a sèries de potències del tipus $A_n = \sum_{|k|=n} a_k s^k$.

Derivades

Anteriorment, hem demostrat pel cas $m = 1$ que les propietats de les derivades es compleixen en el cas de les derivades formals de sèries de potències. Si seguim amb el mateix canvi de variable que pel quocient, és fàcil trivial veure que pel cas $m > 1$ també es compleixen.

Pel cas $m = 2$, les derivades formals parcials d'una sèrie de potències formal es poden calcular com

$$\frac{\partial}{\partial s_1}\sum_{i+j=k} a_{ij} s_1^i s_2^j = \sum_{i+j=k} i \cdot a_{ij} s_1^{i-1} s_2^j$$
$$\frac{\partial}{\partial s_2}\sum_{i+j=k} a_{ij} s_1^i s_2^j = \sum_{i+j=k} j \cdot a_{ij} s_1^{i-1} s_2^j$$

Potències

En aquest cas, utilitzarem la mateixa notació que pel producte, en (2.2.1). Així doncs, per trobar l'expressió de la sèrie formal de potències de B amb $B = A^\alpha$, donat que es tracta de la mateixa expressió que (2.1.4), obtenim

$$B_n = \frac{1}{nA_0}\sum_{i=0}^{n-1}[\alpha n - (\alpha + 1)i]A_{n-i}B_i \tag{2.2.3}$$

amb A_n i B_n sèries del tipus $A_n = \sum_{|k|=n} a_k s^k$.

Funcions trigonomètriques

Seguint un procediment similar que en el cas unidimensional, hem d'estudiar quins són els coeficients de les expressions $B = \sin(A)$ i $C = \cos(A)$, essent A una sèrie formal de potències amb n variables. Considerant el canvi (2.2.1) i les equacions que hem vist en (2.1.5), B i C venen determinades per les fórmules

$$\begin{aligned} B_n &= \frac{1}{n}\sum_{k=1}^{n} kA_k C_{n-k} \\ C_n &= \frac{-1}{n}\sum_{k=1}^{n} kA_k B_{n-k} \end{aligned} \tag{2.2.4}$$

on, de la mateixa manera que en els casos anteriors, A_n, B_n i C_n són sèries formals de potències.

2.3 Implementació

En aquesta secció estudiarem el tractament de les dades per tal d'optimitzar computacionalment les operacions que realitzarem.

En treballar amb aritmètica de sèries per estudiar el problema del pèndol pertorbat, cal tenir en compte la manera amb la qual emmagatzemem les dades per evitar que el cost computacional augmenti i dificulti la simulació.

El problema del pèndol pertorbat utilitza sèries formals de potències de dues variables, així que aquestes seran del tipus,

$$A = \sum_{n\geq 0} \sum_{|k|=n} a_k s^k = \sum_{n\geq 0} \sum_{k_1+k_2=n} a_{k_1,k_2} s_1^{k_1} s_2^{k_2}.$$

Representarem aquestes sèries de manera que cada fila de la matriu correspongui al polinomi de grau n, on n també coincideix amb el nombre de la fila. És a dir, els coeficients del polinomi de la n-èssima fila compliran $k_1 + k_2 = n$. La matriu resultant es pot representar de la següent manera,

$$\begin{array}{cccc} a_{0,0} & & & \\ a_{1,0}s_1 & a_{0,1}s_2 & & \\ a_{2,0}s_1^2 & a_{1,1}s_1s_2 & a_{0,2}s_2^2 & \\ \vdots & & \ddots & \\ a_{n,0}s_1^n & a_{n-1,1}s_1^{n-1}s_2 & \dots & a_{0,n}s_2^n \end{array}$$

Aquesta distribució ens permetrà tenir una millor organització de les dades i facilitar la visualització d'aquestes. No obstant això, caldrà procurar que el cost de les operacions aritmètiques no augmenti significativament, ja que en aquest cas, la simulació esdevindria molt costosa.

El fet de treballar amb punters ens facilita la manera amb la qual podem relacionar els elements d'aquesta matriu. Si guardem els elements de manera que estiguin relacionats consecutivament, podrem tractar la sèrie de potències com si fos una matriu i també com si tinguéssim un vector.

```
1    double **a;
2    a = (double**)malloc((n + 1) * sizeof(double*));
3    a[0] = (double*)malloc((n + 1) * (n + 2) / 2 * sizeof(double));
4    for(int i = 1; i <= n; i++)
5        a[i] = a[i-1] + i;
```

Si ho analitzem amb profunditat, el que estem fent és reservar $n+1$ espais de memòria (corresponents a les $n+1$ files) i assignar $\frac{(n+1)(n+2)}{2}$ espais al primer element de la matriu.

El valor $\frac{(n+1)(n+2)}{2}$ correspon al total d'elements que té la matriu triangular. Relacionant els primers elements de cada fila amb el seu element anterior $a[i-1]+i$, podrem accedir als elements consecutivament sumant i a l'espai de memòria anterior al desitjat. És a dir, les dues maneres per les quals podem accedir a tots els quocients de la matriu són,

```
    for(int i = 0; i <= (n+1)(n+2)/2; i++)
        *(*a+i);

    for(int i = 0; i <= n; i++)
        for(int j = 0; j <= i; j++)
            a[i][j];
```

2.3.1 Operacions aritmètiques

Les operacions aritmètiques bàsiques es computen tal com hem explicat en l'apartat anterior, utilitzant aritmètica de sèries. A continuació veurem com es programen la majoria d'operacions aritmètiques adaptant-ho a sèries de potències. En el capítol següent podrem veure com s'utilitza la sobrecàrrega d'operadors per crear una classe que anomenarem `Jet` que inclogui tota l'aritmètica de sèries corresponent.

Els programes que computen aquestes operacions (suma, resta i producte per un escalar) no presenten cap dificultat afegida més que programar la suma de dos vectors.

Donat que per la suma i resta de sèries s'operen els coeficients individualment, obtindrem una sèrie del tipus

$$\begin{array}{llll} a_{0,0} \pm b_{0,0} & & & \\ a_{1,0} \pm b_{1,0} & a_{0,1} \pm b_{0,1} & & \\ a_{2,0} \pm b_{2,0} & a_{1,1} \pm b_{1,1} & a_{0,2} \pm b_{0,1} & \\ \vdots & & \ddots & \\ a_{n,0} \pm b_{n,0} & a_{n-1,1} \pm b_{n-1,1} & \dots & a_{0,n} \pm b_{0,n} \end{array}$$

El producte per escalar és anàleg.

En tractar l'estructura com si fossin vectors, l'algoritme per calcular-ho resulta

```
    double **a;
    double **b;
    double **resultat;
    // guardem la memoria de les matrius com hem definit anteriorment
    for(int i = 0; i < (n+1)*(n+2)/2; i++)
        *(*resultat+i) = *(*a+i) + *(*b+i);
```

Producte

Per tal de reduir el cost computacional del producte de dues matrius triangulars, també considerarem que el nostre objectiu és computar el producte de dues sèries de potències expressades com a vector. L'algorisme serà,

```
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= n; j++) {
            if (i + j <= n) {
                *(*c + i + j) += *(*a + i) * *(*b + j);
            }
        }
    }
```

Quocient

Donada la fórmula definida en (2.2.2), podem calcular el resultat de $C = A/B$ utilitant l'algorisme,

```
    c.mat[0][0] = a.mat[0][0] / b.mat[0][0];
    for (int i = 1; i <= a.all_elem; i++) {
        double sum = 0;
        for (int j = 0; j <= i - 1; j++) {
            sum += *(*c.mat + j) * *(*b.mat + i - j);
        }
        *(*c.mat + i) = (1.0 / *(*b.mat)) * (*(*a.mat + i) - sum);
    }
```

Funcions trigonomètriques

El procediment per programar el sin d'una sèrie formal de potències es fa seguint el patró descrit en les fórmules (2.2.4). Cal puntualitzar que com és necessària la sèrie del cos per computar la del sin, hem de fer ús d'una variable `Jet` auxiliar.

```
    a[0] = cos(a[0][0]);
    aux[0][0] = sin(a[0][0];
    double a_sum;
    double b_sum;
    for (int n = 1; n <= deg; n++) {
        a_sum = 0;
        b_sum = 0;
        for (int j = 1; j <= n; j++) {
            a_sum += j * *(*aux.mat + n - j) * *(*a.mat + j);
            b_sum += j * a[n - j] * *(*a.mat + j);
        }
        a[n] = -(1.0 / n) * a_sum;
        *(*aux.mat + n) = (1.0 / n) * b_sum;
    }
```

El cas del cos és anàleg.

Capítol 3

Diferenciació automàtica

Per obtenir les derivades del flux d'una equació diferencial, s'utilitza la diferenciació automàtica, que consisteix en un conjunt de tècniques que permeten obtenir les derivades d'una funció avaluades en un punt en concret.

La diferenciació automàtica, a part de poder-la aplicar a la integració numèrica d'equacions diferencials, també s'acostuma a utilitzar en processos com l'optimització, la identificació de paràmetres o la resolució d'equacions no lineals.

Per tal d'aconseguir les derivades d'una funció f, es genera una sèrie de potències

$$A = \sum_{k \geq 0} a_k s^k$$

on els coeficients a_k corresponen a les derivades normalitzades de f.

Definició 3.0.1. *Sigui f una funció C^∞ definida al voltant del zero. Definim a_k com la k-èssima derivada normalitzada de f centrada en el punt 0, és a dir,*

$$a_n = \frac{1}{k!} f^{(k)}(0)$$

Un cop tenim la sèrie de potències formal amb aquests coeficients, el que farem serà aplicar l'aritmètica de sèries descrita en el capítol anterior al mètode d'integració desitjat. Això ens proporcionarà les derivades desitjades.

Donat que treballarem amb funcions multidimensionals, cal definir la derivada parcial normalitzada de f.

$$a_k = \frac{\partial^k f(0)}{k_1! \cdots k_n!}$$

És clar que treballant amb aquest tipus de sèries el que estem fent és treballar amb el polinomi de Taylor, en comptes d'utilitzar un valor concret. Aquest procés s'anomena transport de derivades, ja que estem transportant els càlculs que acostumem a fer utilitzant únicament dades en punt flotant a sèries de derivades. Computar aquest procediment és fàcil en llenguatges com ara `C++`, que permeten la sobrecàrrega d'operadors, que estudiarem més endavant.

Per poder treballar i programar aquestes sèries de potències formals, utilitzarem sèries truncades, les quals assumirem que tenen fins a ordre n. El fet d'utilitzar aquest tipus d'aritmètica en comptes de l'habitual no augmenta l'error del mètode d'integració, sinó que aquest es manté exactament igual. Ho veurem també més endavant.

3.1 Transport de derivades

Com hem comentat, la diferenciació automàtica és un procediment que s'aplica a un algoritme d'integració d'equacions diferencials. Per tant, donades unes condicions inicials i una equació diferencial ordinària $\dot{x} = f(t, x)$, obtindrem les derivades de f avaluades en el punt desitjat de l'òrbita.

El transport de derivades fa referència al fet de modificar l'aritmètica de punt flotant per l'aritmètica de sèries formals truncades a grau n. El que hem de tenir en compte per tal de fer aquest procediment és

- Utilitzar la sobrecàrrega d'operadors. Per tal que el programa pugui fer el canvi d'aritmètica, hem de substituir totes les operacions que siguin necessàries per al mètode d'integració.
- Tenir un mètode d'integració i una funció que permetin el transport de derivades. Donat que volem acabar aplicant el canvi a aritmètica de sèries, les funcions que utilitzem s'han de poder expressar com una seqüència d'operacions i funcions elementals.

Aquest procediment és molt útil, ja que obtenir les derivades de la funció d'una equació diferencial resulta molt costós.

El següent exemple ens permetrà visualitzar més clarament com obtindrem les derivades d'una funció donada a partir d'utilitzar sèries formals de potències.

Exemple 3.1.1. Tenim la funció $f(x_1, x_2) = \frac{x_1+x_2}{x_1 \cdot x_2}$ la qual volem avaluar en el punt $(2, 3)$ i obtenir-ne les derivades formals. Si fem el canvi de variable $x_1 = 2 + s_1$ i $x_2 = 3 + s_1$,

$$\begin{aligned} x_1 x_2 &= (2 + s_1)(3 + s_2) = 6 + 3s_1 + 2s_2 + s_1 s_2 \\ x_1 + x_2 &= 5 + s_1 + s_2 \end{aligned}$$

és a dir,

$$f(2 + s_1, 3 + s_2) = \frac{5 + s_1 + s_2}{6 + 3s_1 + 2s_2 + s_1 s_2}.$$

Comencem separant-ne els termes agrupant els coeficients de manera que $|k| = k_1 + \ldots + k_m = n$.

$$\begin{aligned} A_0 &= 5 \\ A_1 &= s_1 + s_2 \\ B_0 &= 6 \\ B_1 &= 3s_1 + 2s_2 \\ B_2 &= s_1 s_2 \end{aligned}$$

Podem utilitzar la fórmula (2.2.2) per trobar la solució.

$$\begin{aligned} f(s_1, s_2) &= C_0 + C_1 + C_2 \\ &= \frac{A_0}{B_0} + \frac{1}{B_0}(A_1 - B_1 C_0) + \frac{1}{B_0}(A_2 - B_1 C_1 - B_2 C_0) \\ &= \frac{5}{6} + \left[\frac{-1}{4}s_1 - \frac{1}{9}s_2\right] + \left[\frac{1}{8}s_1^2 + \frac{1}{27}s_2^2\right] \end{aligned}$$

És fàcil comprovar que, com hem dit anteriorment, els termes d'aquesta funció corresponen als termes de la sèrie de Taylor de segon ordre de $f(x_1, x_2)$ avaluada en el punt $(2, 3)$. Seguint la notació definida,

$$\begin{aligned} a_0 &= \frac{5}{6} \\ a_1 &= \frac{-1}{4}s_1 - \frac{1}{9}s_2 \\ a_2 &= \frac{1}{8}s_1^2 + \frac{1}{27}s_2^2 \end{aligned}$$

El fet que utilitzar el transport de derivades d'ordre n sigui equivalent a integrar les equacions variacionals fins al mateix ordre n no és trivial. Vegem-ho.

Teorema 3.1.2. *L'aplicació del transport de derivades d'ordre n en un integrador Runge-Kutta, amb un pas h concret, produeix exactament els mateixos resultats que la integració d'equacions variacionals fins a ordre n amb el mateix pas h. És a dir, els dos algoritmes són idèntics.*

El mètode d'integració que estudiarem és el mètode de Runge-Kutta, el qual n'hem parlat en capítols anteriors. La demostració es basa a demostrar que les equacions d'ambdós casos per $n = 1$ son exactament iguals. Per fer-ho, definirem una notació general

$$\dot{y} = F(t, y), \quad y(t_0) = y_0 \tag{3.1.1}$$

En el cas del transport de derivades, el que farem serà aplicar l'aritmètica de sèries al mètode d'integració de Runge-Kutta amb σ avaluacions del camp al problema del valor inicial

$$\dot{x} = f(t, x), \quad x(t_0) = x_0 \tag{3.1.2}$$

per una f infinitament diferenciable i $x \in \mathbb{R}^n$. En el segon cas, aplicarem Runge-Kutta d'ordre 1 a les equacions variacionals

$$\begin{aligned} \dot{x} &= f(t, x), & x\,(t_0) &= x_0 \\ \dot{v} &= D_x f(t, x)v, & v\,(t_0) &= v_0 \end{aligned} \tag{3.1.3}$$

amb $v(t)$ un vector de direcció arbitrària i dimensió n que compleix $v_0 \neq 0$.
Així doncs, fixat un pas h, les equacions de Runge-Kutta aplicades a (3.1.1) són

$$\begin{aligned} \kappa_i &= F(t_0 + c_i h, y_0 + h(a_{i,1}\kappa_1 + \ldots + a_{i,\sigma}\kappa_\sigma) \\ y_1 &= y_0 + h(b_1\kappa_1 + \ldots + b_\sigma\kappa_\sigma). \end{aligned} \tag{3.1.4}$$

Utilitzarem la notació $\bar{\kappa_i}$ per fer referència a la coordenada x dels valors de κ_i del problema (3.1.3) i $\hat{\kappa}_i$ a les coordenades de v.

Comencem veient quin és el resultat d'aplicar transport de derivades d'ordre 1 a les equacions (3.1.4) per resoldre el problema del valor inicial (3.1.2). Aplicar el transport de derivades implica fer el canvi de variables $y_0 = x_0 + v_0 \cdot s$. Com que, per definició $y_1 = y_0 + h\sum_{i=1}^{\sigma} b_i\kappa_i$, tenim

$$x_1 + v_1 \cdot s = x_0 + v_0 \cdot s + h\sum_{i=1}^{\sigma} b_i\kappa_i.$$

Si apliquem el transport de derivades també a les variables κ_i, obtenim

$$\begin{aligned}\bar{\kappa}_i + \hat{\kappa}_i s &= f(t_0 + c_i h, y_0 + h\sum_{i=1}^{\sigma} b_i\kappa_i) \\ &= f(t_0 + c_i h, x_0 + v_0 s + h\sum_{i=1}^{\sigma} b_i(\bar{\kappa}_i + \hat{\kappa}_i s)).\end{aligned}$$

Per simplificar la notació, definim $t = t_0 + c_i h$, $x = x_0 + h\sum_{i=1}^{\sigma} b_i\bar{\kappa}_i$ i $v = v_0 + h\sum_{i=1}^{\sigma} b_i\hat{\kappa}_i$ de manera que

$$\begin{aligned}\bar{\kappa}_i + \hat{\kappa}_i s &= f(t, x + vs) \\ &= f(t, x) + D_x f(t, x + vs) \cdot v \cdot s,\end{aligned}$$

on la segona igualtat resulta d'aplicar la sèrie de Taylor a la funció f. Finalment, donat que les equacions resultants són

$$\begin{aligned}\bar{\kappa}_i &= f(t, x) \\ \hat{\kappa}_i &= D_x f(t, x + vs) \cdot v,\end{aligned}$$

i coincideixen amb les equacions variacionals de primer ordre de (3.1.3), podem deduir que aquests dos processos són idèntics.

El fet que siguin equivalents, fa que l'error també sigui exactament el mateix. Així doncs, el transport de derivades no augmenta l'error que cometem en utilitzar el mètode d'integració de Runge-Kutta.

3.2 Sobrecàrrega d'operadors

Per fer el canvi d'aritmètica serà necessari la sobrecàrrega d'operadors. Aquest procés consisteix a definir una classe o estructura de dades i redefinir totes les funcions que poden ser necessàries per aplicar els mètodes d'integració. Els conceptes que s'han de tenir en compte a l'hora de fer sobrecàrrega d'operadors són els següents.

- Els operadors han de permetre la sobrecàrrega. En `C++` no tots els operadors ho permeten, com per exemple les funcions `sizeof` o `"."`.
- Fer un ús correcte de la memòria dinàmica. Moltes vegades s'utilitza la memòria dinàmica dins les noves classes o estructures de dades i si no desassignem correctament aquesta memòria, podríem arribar a emplenar la memòria no accessible de l'ordinador, donat que un cop acabat el programa, ja no es pot tornar a accedir a aquesta.

- Fer ús de les bones pràctiques com utilitzar els constructors i destructors, o bé definir totes les funcions, encara que no s'utilitzin, per tal d'evitar errors.

Abans de veure la implementació completa de l'aritmètica de sèries aplicada al mètode de Runge-Kutta vegem un exemple senzill de sobrecàrrega d'operadors.

Exemple 3.2.1. Donada una funció $f(x) = (x_1 - \alpha)^2 + (x_2 - \beta)^2$, definim una classe `adouble` que conté el valor real d'una funció i la seva derivada.

```
1    class adouble
2    {
3        double val;
4        double dot;
5    }
```

i definim un programa que computa el producte entre dos nombres reals

```
1    double x1, x2, y, a, b;
2    cin >> x1 >> x2 >> a >> b;
3    y = 0.0;
4    y = y + (x1-a)*(x1-a) + (x2-b)*(x2-b);
5    cout << y << endl;
```

D'aquesta manera, l'únic canvi a l'hora de fer la sobrecàrrega d'operadors consistirà a definir la suma i el producte entre dos valors de tipus `adouble` i modificar la manera com s'inicialitzen. Per exemple, la definició de la funció que sobrecarregarà l'operació producte, correspon a,

```
1    adouble operator*(adouble a, adouble b)
2    {
3        adouble c;
4        c.val = a.val * b.val;
5        c.dot = b.val * a.dot + a.val * b.dot;
6    }
```

El codi final que utilitza aquesta classe és

```
1    double a, b, dx1, dx2;
2    adouble x1, x2, y;
3    cin >> x1 >> x2 >> dx1 >> dx2;
4    cin >> a >> b;
5    x1.dot = dx1;
6    x2.dot = dx2;
7    y = y + (x1-a)*(x1-a) + (x2-b)*(x2-b);
8    cout << y << y.dot;
```

Com podem comprovar hi ha molt pocs canvis respecte al codi original.

3.3 Implementació

Com hem explicat al final del capítol de sèries formals de potències, l'estructura creada serà una matriu triangular inferior, amb els coeficients guardats de manera que també puguem utilitzar la classe com a vector. L'hem anomenat *Jet*, i els seus elements públics són

```
1     int n;
2     int all_elem;
3     double** mat;
4     Jet(int dim);
5     Jet();
6     Jet(double** in_mat, int dim);
7     ~Jet();
```

juntament amb totes les operacions aritmètiques que caldrà definir. Hi podem observar els tres atributs `n`, `all_elem` i `mat`. Aquests elements seran accessibles des de tots els fitxers del nostre programa. Llavors, el que hem de fer és programar les funcions descrites en l'apartat anterior.

Cal tenir en compte que per tal que el canvi d'aritmètica no impliqui res més que un canvi del nom de la variable, hem d'implementar totes les funcions trigonomètriques que utilitzem, tenint en compte que, per exemple, podem necessitar operar una variable de tipus `Jet` amb una de tipus `double`. És per això que la llista d'operacions que ha calgut computar són

```
1     Jet& operator=(const Jet jet);
2     Jet& operator=(const Jet* jet);
3     Jet operator+(const Jet jet);
4     Jet operator-(const Jet jet);
5     Jet operator*(const Jet jet);
6     Jet operator/(const Jet jet);
7     Jet operator+(double value);
8     Jet operator-(double value);
9     Jet operator*(double value);
10    Jet operator/(double value);
11    Jet operator+=(const Jet jet);
12    Jet operator-=(const Jet jet);
13    Jet operator*=(const Jet jet);
14    Jet operator/=(const Jet jet);
15    Jet operator+=(double value);
16    Jet operator-=(double value);
17    Jet operator*=(double value);
18    Jet operator/=(double value);
19    Jet operator-();
20    Jet sin();
21    Jet cos();
```

Donat que el producte d'un `Jet` amb un `double` estarà definit només en el cas `Jet * double`, s'ha considerat necessari afegir quatre funcions addicionals, que permeten al programa realitzar també l'operació `double * Jet`.

```
1     inline Jet operator+(double value, Jet jet);
2     inline Jet operator-(double value, Jet jet);
3     inline Jet operator*(double value, Jet jet);
4     inline Jet operator/(double value, Jet jet);
```

Tot el codi implementat es pot trobar a l'apèndix.

Filament, l'únic que hem de fer és aplicar tota aquesta aritmètica al programa de Runge-Kutta implementat i descrit prèviament. Per tal de fer-ho, només cal canviar el tipus de les variables x i k i la manera com es declaren i s'accedeix a elles.

Capítol 4

Derivades d'una aplicació de Poincaré

En aquest capítol ens centrarem a estudiar els objectes invariants d'una equació diferencial ordinària. Aquests són els que ens aporten informació sobre el comportament d'un sistema dinàmic a llarg termini.

L'aplicació de Poincaré és la caracterització de la intersecció d'una òrbita periòdica de l'estat de fases d'un sistema dinàmic continu amb un subespai d'una dimensió inferior, el qual és transversal al flux del sistema. El subespai mencionat s'anomena la secció de Poincaré.

Aquesta secció preserva la majoria d'òrbites periòdiques i quasi-periòdiques del sistema original, i l'aplicació de Poincaré redueix un flux continu a un mapeig a temps discret. Són aquests dos fets els que ens permeten analitzar el sistema dinàmic corresponent de manera més senzilla.

Donat que la secció de Poincaré està formada pels punts de l'òrbita que intersequen un subespai de dimensió inferior, no hi ha un mètode general per construir l'aplicació. És per això que no acostuma a tenir una expressió explícita, sinó que es limita a estudiar com es comporten les òrbites periòdiques. La manera de treballar amb l'aplicació de Poincaré sol ser utilitzar mètodes d'integració com el Runge-Kutta per obtenir els valors de l'equació diferencial ordinària en el punt desitjat.

4.1 Aplicació de Poincaré temporal

Definició 4.1.1. *Sigui $f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ una aplicació qualsevol. Si $T > 0$ és el mínim valor que compleix*

$$f(t, x) = f(t + T, x), \text{ per tot } (t, x) \in \mathbb{R} \times \mathbb{R}^n,$$

direm que la funció f és T-periòdica.

Donat el problema del valor inicial,

$$\begin{aligned} \dot{x} &= f(t,x), \\ x(t_0) &= a, \end{aligned} \tag{4.1.1}$$

assumirem que $\dot{x} = f(t,x)$ és T-periòdica. Per definir l'aplicació de Poincaré temporal, notarem $\phi(t;t_0,x^{(0)})$ a la solució a temps t del problema de Cauchy amb condicions inicials $(t_0,x^{(0)})$.

Definició 4.1.2. *Sigui $\phi(t;t_0,x^{(0)})$ el flux d'una equació diferencial T-periòdica, l'aplicació de Poincaré associada es defineix com*

$$P(x^{(0)}) = \phi(T;0,x^{(0)}) = x^{(1)},$$

és a dir, correspon a l'avaluació del flux a temps $t+T$ respecte al punt anterior.

Definició 4.1.3. *Donada l'aplicació de Poincaré P tal que compleix $P(x) = x$ per algun $x \in \mathbb{R}^n$, definirem com a òrbita periòdica el flux que té com a condició inicial x, és a dir, el flux $\phi(t;0,x)$ el qual la seva condició inicial és un punt fix de l'aplicació de Poincaré.*

Per trobar quines són les òrbites periòdiques del sistema amb un període T hem de trobar els punts fixos de l'aplicació de Poincaré. Per tal de trobar-los, definim

$$F(x) = P(x) - x$$

on $x \in \mathbb{R}^n$ i P és l'aplicació de Poincaré que retorna el valor del camp passa un període de temps T. Llavors, trobar els punts fixos de P serà equivalent a trobar els zeros de la funció F.

Donat que no tenim l'expressió explícita de l'aplicació de Poincaré, el que haurem de fer serà utilitzar el mètode de Runge-Kutta-Fehlberg descrit anteriorment per poder avaluar F en el punt $t = T$ i, d'aquesta manera, obtenir-ne les arrels.

Podem utilitzar el mètode iteratiu per trobar els zeros d'una funció que més ens convingui, essent Newton el més habitual. Com a recordatori, remarquem que les equacions d'aquest mètode són

$$\begin{aligned} D_xF(x)\cdot h &= -F(x) \\ x_{n+1} &= x_n + h \end{aligned}$$

Hem de tenir en compte que $F(x) = P(x) - x$ i llavors $D_xF(x) = D_xP(x) - Id$.

Per obtenir la diferencial de l'aplicació de Poincaré podem aplicar aritmètica de sèries al mètode de Runge-Kutta-Fehlberg i que aquest no només obtingui els punts desitjats en un t concret, sinó que també les derivades de P.

D'aquesta manera, si computem

$$P(x_0 + s_1, y_0 + s_2),$$

obtindrem la diferencial d'ordre 1 de P.
Aquest procediment seria anàleg, tal com hem demostrat el capítol anterior, a calcular els coeficients de la matriu variacional i obtenir el valor de h a partir d'aquesta matriu.

Pel que fa al programa, si prèviament havíem definit les funcions necessàries per realitzar tot aquest procediment (Runge-Kutta-Fehlberg, Newton i algunes funcions derivades d'aquestes), i l'aritmètica de sèries està ben definida, el canvi al transport de derivades no hauria de suposar més canvi que modificar el tipus de les variables.

4.2 Exemples

La descripció pel moviment d'un pèndol ens serveix per poder treballar amb un sistema d'equacions diferencials no lineals de manera relativament senzilla, ja que es tracta d'una situació que ens podem imaginar.

4.2.1 Pèndol simple

L'equació que descriu el moviment d'un pèndol simple correspon a

$$\ddot{x} + \sin x = 0.$$

Si modifiquem l'equació per obtenir dues equacions diferencials de primer ordre, obtenim

$$\begin{cases} \dot{x} = y \\ \dot{y} = -\sin x \end{cases}$$

Resolent $(y, -\sin x) = (0, 0)$, obtindrem els punts d'equilibri d'aquest sistema. Aquests, es troben en $(k\pi, 0)$ amb $k \in \mathbb{Z}$. Definim la funció $f(x, y) = (y, -\sin x)$.

Per determinar l'estabilitat dels punts d'equilibri, cal estudiar el comportament de la matriu diferencial

$$Df(x, y) = \begin{pmatrix} 0 & 1 \\ -\cos x & 0 \end{pmatrix}.$$

Sense perdre generalitat, estudiarem principalment els punts $(0, 0)$ i $(\pi, 0)$.

Si ens centrem primer en el punt $(0, 0)$, la matriu resultant $Df(0, 0)$ és

$$Df(x, y) = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

En calcular els valors propis de la matriu, per saber-ne l'estabilitat, obtenim

$$\lambda^2 - 1 = 0 \Rightarrow \lambda = \pm\sqrt{-1}.$$

Per tant, es tracta d'un punt d'equilibri el·líptic, ja que els valors propis són imaginaris.

Si executem el programa que utilitza el transport de derivades en els mètodes de Newton i Runge-Kutta per obtenir els punts fixos de l'aplicació de Poincaré, obtenim que a prop del punt $(-1, 0)$ es troba el punt fix

$$\begin{pmatrix} -3.944304526105059e-30 & 2.563797941968288e-30 \end{pmatrix}$$

és a dir, el punt $(0, 0)$. Donat que hem utilitzat el transport de derivades, també podem saber el valor de les derivades de P en aquest punt.

$$\begin{pmatrix} -0.2759485585971886 & -0.9611724054685751 \\ 0.9611724054685751 & -0.2759485585971886 \end{pmatrix}$$

Pel que fa al cas $(\pi, 0)$, obtenim

$$Df(x, y) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

on els valors propis són reals i podem deduir que el punt d'equilibri és hiperbòlic.

$$\lambda^2 + 1 = 0 \Rightarrow \lambda = \pm 1$$

Repetint el procediment realitzat pel punt $(0, 0)$, executant el programa en un punt proper al $(\pi, 0)$, el punt fix més proper resulta

$$\begin{pmatrix} 3.141592653589793 & 1.576382322784346e-15 \end{pmatrix}$$

i la seva matriu diferencial és

$$\begin{pmatrix} 42.58892331295026 & 42.57718155252144 \\ 42.57718155252134 & 42.58892331295039 \end{pmatrix}$$

Si estudiem el retrat de fase d'aquest sistema d'equacions diferencials 4.1, podem observar com clarament els punts d'equilibri descrits com $(k\pi, 0)$ amb $k = 2n$, són hiperbòlics i els representats com $((k\pi, 0)$ amb $k = 2n + 1$, són punts de sella.

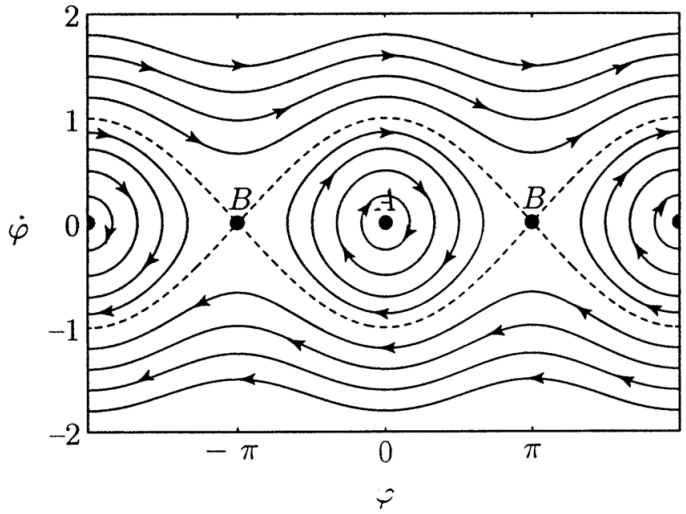

Figura 4.1: Retrat de fase d'un pèndol simple. Figura extreta de [16].

4.2.2 Pèndol pertorbat

L'equació d'un pèndol pertorbat descriu el moviment d'un pèndol sotmès a un camp magnètic de magnitud ε.

$$\ddot{x} + \sin x = \varepsilon \sin(\omega t)$$

Considerarem $\omega = \sqrt{2}$. Per poder treballar més còmodament, simplificarem l'equació de segon ordre obtenint el sistema d'equacions diferencials següent.

$$\begin{cases} \dot{x} = y \\ \dot{y} = -\sin x + \varepsilon \sin(\omega t). \end{cases} \tag{4.2.1}$$

Podem observar que es tracta d'un sistema $\frac{2\pi}{\omega}$-periòdic.

Donat que aquest sistema és una extensió del cas del pèndol simple, utilitzarem alguns dels resultats vistos en l'apartat anterior per desenvolupar aquest cas. Definim $\varepsilon = 0.01$.

Volem trobar les òrbites periòdiques de període $T = \frac{2\pi}{\omega}$, és a dir, saber quins són els punts que, passat el temps $t = T$, tornen al valor inicial.

Si utilitzem els programes desenvolupats per trobar els punts fixos de P i, alhora, els valors de la matriu $D_x P(x, y)$ en el punt inicial $(0, 0)$ utilitzant el transport de derivades respecte al mètode de Runge-Kutta-Fehlberg, obtenim que a prop del zero podem observar el punt fix

$$\left(3.559960291178132e-16-0.01414196925261475\right)$$

amb la matriu diferencial corresponent

$$\begin{pmatrix} -0.266308871451823 & -0.9639118415560569 \\ 0.9638636490721569 & -0.2663088714518204 \end{pmatrix}$$

Fent el mateix procediment per un valor proper al punt $(\pi, 0)$, el punt fix resultant és,

$$\left(3.14159265358979-0.004714047045742272\right)$$

amb la matriu diferencial corresponent

$$\begin{pmatrix} 42.51546632442331 & 42.50374362669686 \\ 42.50366491596443 & 42.51546632442357 \end{pmatrix}$$

4.2.3 Test

A continuació descriurem la manera de comprovar que el transport de derivades s'ha realitzat correctament.

Anomenem $p(x_1, x_2)$ a l'aplicació de Poincaré que retorna $(\hat{x_1}, \hat{x_2})$, on totes les incògnites són reals. Definim

$$P(x_1^{(0)} + s_1, x_2^{(0)} + s_2) = P(x^{(0)} + s) \tag{4.2.2}$$

amb $x^{(0)} = x_1^{(0)} + x_2^{(0)}$ i $s = s_1 + s_2$. Donat que això resultarà una sèrie de potències formal, ho expressem com,

$$P(x_1^{(0)} + s_1, x_2^{(0)} + s_2) = \sum_{|k|=0}^{N} p_k s^k + O_{N+1}(s).$$

El nostre objectiu principal és comprovar que el sumatori resultant és correcte.

Per tal de fer-ho, fixem un $\hat{s}$ prou petit. Avaluant $P(x^{(0)} + \hat{s})$ amb la funció inicial $P(x_1, x_2)$ obtenim el sumatori $\sum_{|k|=0}^{N} p_k \hat{s}^k$. Cal notar que en aquest cas els valors de les expressions són valors reals.

Per tant,

$$||P(x^{(0)} + \hat{x}) - \sum_{|k|=0}^{N} p_k \hat{s}^k|| = c_{N+1} \hat{s}^{N+1} + \dots$$

El que obtenim a partir de $\hat{s}^{N+2}$ es pot considerar negligible si escollim la variable $\hat{s}$ de manera que aquests valors siguin prou petits.

Per simplificar els càlculs, utilitzarem la notació

$$\alpha(\hat{s}) = ||P(x^{(0)} + \hat{x}) - \sum_{|k|=0}^{N} p_k \hat{s}^k|| = c_{N+1}\hat{s}^{N+1}$$

Ara, si calculem $\alpha(\hat{s}/2)$, obtindrem

$$\alpha(\hat{s}/2) = c_{N+1}\frac{\hat{s}^{N+1}}{2^{N+1}} + \dots$$

on els valors a partir de s^{N+2}, són també negligibles.

Finalment, es complirà

$$\frac{\alpha(\hat{s})}{\alpha(\hat{s}/2)} \approx 2^{N+1},$$

ja que el valor c^{N+1} coincideix en ambdós casos i, per tant, se simplifica.

Donat que N és fix i conegut, si el valor resultant d'aquest quocient no coincideix amb 2^{N+1}, obtindrem informació de quin és el N del sumatori resultant que ha fallat.

Per exemple, si el quocient resulta ser 2^3, és clar que a partir del 3r terme dels sumatoris, $\alpha(\hat{s})$ i $\alpha(\hat{s}/2)$ el càlcul és erroni.

Conclusions

Finalment, si considerem el balanç general respecte els resultats obtinguts, el valorarem com a positiu.

D'entrada, els objectius teòrics es poden assumir com a complerts, ja que aquests consistien a adquirir un major coneixement en àmbits com els mètodes numèrics o les equacions algebraiques.

Durant el treball, hem realitzat l'estudi dels mètodes de Runge-Kutta i implementació del mètode Runge-Kutta-Fehlberg d'ordres 4 i 5. A part, hem pogut fer el canvi d'aritmètica de punt flotant a aritmètica de sèries de forma satisfactòria, tal com mostra el codi i els resultats. El control de pas automàtic també ha sigut implementat i funciona correctament.

A part d'implementar el mètode de Runge-Kutta-Fehlberg, hem pogut utilitzar-lo per analitzar els punts fixos de l'equació diferencial que descriu el moviment d'un pèndol. Aquests punts s'han obtingut, tal com s'especifica en l'últim capítol del treball, utilitzant el mètode de Newton, el qual també ha funcionat correctament amb l'aritmètica de sèries.

Bibliografia

[1] J. C. Butcher. *Numerical methods for ordinary differential equations*. English. 3rd edition. Hoboken, NJ: John Wiley & Sons, 2016. ISBN: 978-1-119-12150-3; 978-1-119-12153-4. DOI: 10.1002/9781119121534.

[2] The MathJax Consortium. *The Poincaré map*. 2009-2017. URL: https://people.math.wisc.edu/~angenent/519.2016s/notes/poincare-map.html.

[3] R. P. Curiel. *Seno y coseno de una matriz*. Avaliable at: https://miscelaneamatematica.org/download/tbl_articulos.pdf2.88bf285c9363f498.353130322e706466.pdf. 2010.

[4] Amadeu Delshams i Teresa M. Seara. "An asymptotic expression for the splitting of separatrices of the rapidly forced pendulum". English. A: *Commun. Math. Phys.* 150.3 (1992), pàg. 433 - 463. ISSN: 0010-3616. DOI: 10.1007/BF02096956.

[5] Alexander M. Gofen. "The ordinary differential equations and automatic differentiation unified". English. A: *Complex Var. Elliptic Equ.* 54.9 (2009), pàg. 825 - 854. ISSN: 1747-6933. DOI: 10.1080/17476930902998852.

[6] Alexander M. Gofen. "The ordinary differential equations and automatic differentiation unified". English. A: *Complex Var. Elliptic Equ.* 54.9 (2009), pàg. 825 - 854. ISSN: 1747-6933. DOI: 10.1080/17476930902998852.

[7] Andreas Griewank i Andrea Walther. *Evaluating derivatives. Principles and techniques of algorithmic differentiation*. English. 2nd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 2008. ISBN: 978-0-898716-59-7; 978-0-89871-776-1. DOI: 10.1137/1.9780898717761.

[8] Andreas Griewank i Andrea Walther. *Evaluating derivatives. Principles and techniques of algorithmic differentiation*. English. 2nd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 2008. ISBN: 978-0-898716-59-7; 978-0-89871-776-1. DOI: 10.1137/1.9780898717761.

[9] David F. Griffiths i Desmond J. Higham. *Numerical methods for ordinary differential equations. Initial value problems*. English. Springer Undergrad. Math. Ser. London: Springer, 2010. ISBN: 978-0-85729-147-9; 978-0-85729-148-6. DOI: 10.1007/978-0-85729-148-6.

[10] Arieh Iserles. *A first course in the numerical analysis of differential equations*. English. 2nd ed. Camb. Texts Appl. Math. Cambridge: Cambridge University Press, 2009. ISBN: 978-0-521-73490-5. DOI: 10.1017/CBO9780511995569.

[11] Àngel Jorba i Maorong Zou. "A software package for the numerical integration of ODEs by means of high-order Taylor methods". English. A: *Exp. Math.* 14.1 (2005), pàg. 99 - 117. ISSN: 1058-6458. DOI: 10.1080/10586458.2005.10128904.

[12] Donald E. Knuth. *The art of computer programming. Vol. 2: Seminumerical algorithms*. English. 3rd ed. Bonn: Addison-Wesley, 1998. ISBN: 0-201-89684-2.

[13] J. D. Lambert. *Numerical methods for ordinary differential systems: the initial value problem*. English. Chichester etc.: John Wiley & Sons, 1991. ISBN: 0-471-92990-5.

[14] Gerald Teschl. *Ordinary differential equations and dynamical systems*. English. Vol. 140. Grad. Stud. Math. Providence, RI: American Mathematical Society (AMS), 2012. ISBN: 978-0-8218-8328-0.

[15] Herbert S. Wilf. *Generatingfunctionology*. English. 3rd ed. Wellesley, MA: A K Peters, 2006. ISBN: 1-56881-279-5.

[16] O. Winter i C. Murray. "Resonance and chaos: I. First-order interior resonances". A: *Astronomy and Astrophysics* 319 (febr. de 1997), pàg. 290 - 304.

Apèndix A

Resultats

En aquest capítol trobarem la formalització del desenvolupament de Taylor, resultat necessari per entendre certes parts del treball.

També hi podem trobar el procediment per calcular les derivades de la funció $x(t)$, que no s'han afegit durant el treball per evitar passos intermitjos prou clars.

A.1 Resultats preliminars

A.1.1 Desenvolupament de Taylor

Teorema A.1.1. *Sigui $k \geq 1$ un enter, i sigui $f : \mathbb{R} \to \mathbb{R}$ una funció diferenciable k vegades en el punt $a \in \mathbb{R}$. Existeix una funció $h_k : \mathbb{R} \to \mathbb{R}$ tal que*

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \ldots + \frac{f^{(k)(a)}}{k!}(x-a)^k + h_k(x)(x-a)^k$$

i també

$$lim_{x \to a} h_k(x) = 0$$

Per obtenir la fórmula del desenvolupament de Taylor d'una funció estàndard f en un punt $x+h$, partim de la fórmula original que hem anunciat anteriorment, i canviem x per $x+h$ i x_0 per x i, d'aquesta manera,

$$\begin{aligned} f(x+h) &= f(x) + f'(x)(x+h-x) + \frac{1}{2}f''(x)(x+h-x)^2 \frac{1}{3!}f'''(x)(x+h-x)^3 + \mathcal{O}(h^4) \\ &= f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{3!}f^{(3)}(x) + \mathcal{O}(h^4) \end{aligned} \tag{A.1.1}$$

A.2 Derivades de $x(t)$ pel mètode RK2

Els càlculs de les derivades de x en principi són trivials utilitzant la regla de la cadena. No obstant això, els especificarem a continuació per facilitar-ne la demostració. Sabem que $x'(t) = f(t, x(t))$ per l'enunciat del problema de Cauchy. La resta de derivades es deduiran a partir d'aquesta. Com que $x = x(t)$ és una funció que depèn de t, totes les derivades es faran respecte a aquesta variable. Per tant, tenim

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

Per tal de simplificar la notació, considerem les funcions avaluades en el punt (t, x) malgrat que no s'especifiqui.

$$\begin{aligned} x'''(t) = f''(t, x(t)) &= f'_t + (f_x f)' \\ &= f_{tt} + f_{tx}f + f_{tx}f + f_{xx}f^2 + f_x f' \\ &= f_{tt} + 2f_{tx}f + f_{xx}f^2 + f_x(f_t + f_x f) \end{aligned}$$

Apèndix B

Codi

Aquest apèndix recull tots els programes utilitzats per computar l'aritmètica de sèries, el mètode de Newton que permet trobar els punts fixos de l'aplicació de Poincaré i el mètode de Runge-Kutta-Fehlberg.

B.1 Classe abstracta Jet

```
1  extern int DEGREE;
2
3  class Jet {
4
5  public:
6      int n;
7      int all_elem;
8      double** mat;
9      Jet(int dim);
10     Jet();
11     Jet(double** in_mat, int dim);
12     ~Jet();
13
14     Jet& operator=(const Jet jet);
15     Jet& operator=(const Jet* jet);
16     Jet operator+(const Jet jet);
17     Jet operator-(const Jet jet);
18     Jet operator*(const Jet jet);
19     Jet operator/(const Jet jet);
20     Jet operator+(double value);
21     Jet operator-(double value);
22     Jet operator*(double value);
23     Jet operator/(double value);
24     Jet operator+=(const Jet jet);
25     Jet operator-=(const Jet jet);
26     Jet operator*=(const Jet jet);
27     Jet operator/=(const Jet jet);
28     Jet operator+=(double value);
29     Jet operator-=(double value);
30     Jet operator*=(double value);
31     Jet operator/=(double value);
32     Jet operator-();
33     Jet sin();
34     Jet cos();
35     double aval(double s);
36 };
```

B.2 Implementació de la classe Jet

```
#include "jet.h"
#include <algorithm>
#include <cmath>
#include <functional>
#include <iostream>
#include <math.h>
#include <stdlib.h>
#define DEGREE 6
using namespace std;

inline Jet::Jet(int dim)
{
    this->mat = new double*[dim + 1];
    if (!this->mat) {
        cout << "memory allocation failed\n";
        exit(1);
    }
    this->n = dim;
    this->all_elem = (this->n + 1) * (this->n + 2) / 2;
    this->mat[0] = new double[all_elem];
    if (!this->mat[0]) {
        cout << "memory allocation failed\n";
        exit(1);
    }
    for (int i = 1; i <= this->n; i++) {
        this->mat[i] = this->mat[i - 1] + i;
        if (!this->mat[i]) {
            cout << "memory allocation failed\n";
            exit(1);
        }
    }

    for (int i = 0; i <= dim; i++) {
        for (int j = 0; j <= i; j++) {
            this->mat[i][j] = 0.0;
        }
    }
}
inline Jet::Jet()
{
    this->mat = new double*[DEGREE + 1];
    this->n = DEGREE;
    this->all_elem = (this->n + 1) * (this->n + 2) / 2;
    this->mat[0] = new double[all_elem];

    for (int i = 0; i <= DEGREE; i++) {
        this->mat[i] = new double[i + 1];
    }

    for (int i = 0; i <= DEGREE; i++) {
        for (int j = 0; j <= i; j++) {
            this->mat[i][j] = 0.0;
        }
    }
}
inline Jet::~Jet()
{
    // if (this->mat[0])
    //     delete[] this->mat[0];
    // if (this->mat)
    //     delete[] this->mat;
}
inline Jet& Jet::operator=(const Jet jet)
{
    this->mat = new double*[jet.n + 1];
    if (!this->mat) {
```

```
67          cout << "memory allocation failed\n";
68          exit(1);
69      }
70      this->n = jet.n;
71      this->all_elem = (this->n + 1) * (this->n + 2) / 2;
72      this->mat[0] = new double[all_elem];
73      if (!this->mat[0]) {
74          cout << "memory allocation failed\n";
75          exit(1);
76      }
77      for (int i = 1; i <= this->n; i++) {
78          this->mat[i] = this->mat[i - 1] + i;
79      }
80
81      for (int i = 0; i <= jet.n; i++) {
82          for (int j = 0; j <= i; j++) {
83              this->mat[i][j] = jet.mat[i][j];
84          }
85      }
86      return *this;
87  }
88  inline Jet& Jet::operator=(const Jet* jet)
89  {
90
91      this->mat = new double*[jet->n + 1];
92      if (!this->mat) {
93          cout << "memory allocation failed\n";
94          exit(1);
95      }
96      this->n = jet->n;
97      this->all_elem = (this->n + 1) * (this->n + 2) / 2;
98      this->mat[0] = new double[all_elem];
99      if (!this->mat[0]) {
100         cout << "memory allocation failed\n";
101         exit(1);
102     }
103     for (int i = 1; i <= this->n; i++) {
104         this->mat[i] = this->mat[i - 1] + i;
105     }
106
107     for (int i = 0; i <= jet->n; i++) {
108         for (int j = 0; j <= i; j++) {
109             this->mat[i][j] = jet->mat[i][j];
110         }
111     }
112     return *this;
113 }
114 inline Jet Jet::operator+(const Jet jet)
115 {
116     if (jet.n >= this->n) {
117         Jet aux(jet.n);
118         for (int i = 0; i <= this->all_elem; i++) {
119             *(*aux.mat + i) = *(*this->mat + i) + *(*jet.mat + i);
120         }
121         for (int i = this->all_elem + 1; i <= jet.all_elem; i++) {
122             *(*aux.mat + i) = *(*jet.mat + i);
123         }
124         return aux;
125     } else {
126         Jet aux(this->n);
127         for (int i = 0; i <= jet.all_elem; i++)
128             *(*aux.mat + i) = *(*this->mat + i) + *(*jet.mat + i);
129         for (int i = jet.all_elem + 1; i <= this->all_elem; i++)
130             *(*aux.mat + i) = *(*this->mat + i);
131         return aux;
132     }
133 }
134 inline Jet Jet::operator-(const Jet jet)
135 {
136     if (jet.n >= this->n) {
```

```
137         Jet aux(jet.n);
138         for (int i = 0; i <= this->all_elem; i++)
139             *(*aux.mat + i) = *(*this->mat + i) - *(*jet.mat + i);
140 
141         for (int i = this->all_elem + 1; i <= jet.all_elem; i++)
142             *(*aux.mat + i) -= *(*jet.mat + i);
143         return aux;
144     } else {
145         Jet aux(this->n);
146         for (int i = 0; i <= jet.all_elem; i++)
147             *(*aux.mat + i) = *(*this->mat + i) - *(*jet.mat + i);
148         for (int i = jet.all_elem + 1; i <= this->all_elem; i++)
149             *(*aux.mat + i) = *(*this->mat + i);
150         return aux;
151     }
152 }
153 inline Jet Jet::operator*(const Jet jet)
154 {
155     if (jet.n >= this->n) {
156         Jet aux(jet.n);
157 
158         for (int i = 0; i <= this->all_elem; i++) {
159             for (int j = 0; j <= jet.all_elem; j++) {
160                 if (i + j <= jet.all_elem) {
161                     *(*aux.mat + i + j) += *(*this->mat + i) * *(*jet.mat + j);
162                 }
163             }
164         }
165         return aux;
166     } else {
167         Jet aux(this->n);
168 
169         for (int i = 0; i <= this->all_elem; i++) {
170             for (int j = 0; j <= jet.all_elem; j++) {
171                 if (i + j <= this->all_elem) {
172                     *(*aux.mat + i + j) += *(*this->mat + i) * *(*jet.mat + j);
173                 }
174             }
175         }
176         return aux;
177     }
178 }
179 inline Jet Jet::operator/(const Jet jet)
180 {
181     Jet aux(jet.n);
182     double* result = new double[jet.n + 1];
183 
184     aux.mat[0][0] = this->mat[0][0] / jet.mat[0][0];
185     for (int i = 1; i <= this->all_elem; i++) {
186         double sum = 0;
187         for (int j = 0; j <= i - 1; j++) {
188             sum += *(*aux.mat + j) * *(*jet.mat + i - j);
189         }
190         *(*aux.mat + i) = (1.0 / *(*jet.mat)) * (*(*this->mat + i) - sum);
191     }
192     return aux;
193 }
194 inline Jet Jet::operator+(double value)
195 {
196     Jet aux(this->n);
197     for (int i = 0; i <= this->all_elem; i++)
198         *(*aux.mat + i) = *(*this->mat + i);
199     aux.mat[0][0] += value;
200 
201     return aux;
202 }
203 inline Jet Jet::operator-(double value)
204 {
205     Jet aux(this->n);
206     for (int i = 0; i <= this->all_elem; i++)
```

```
207         *(*aux.mat + i) = *(*this->mat + i);
208     aux.mat[0][0] -= value;
209     return aux;
210 }
211 inline Jet Jet::operator*(double value)
212 {
213     Jet aux(this->n);
214 
215     for (int i = 0; i <= this->n; i++)
216         for (int j = 0; j <= i; j++)
217             aux.mat[i][j] = this->mat[i][j] * value;
218 
219     return aux;
220 }
221 inline Jet Jet::operator/(double value)
222 {
223     Jet aux(this->n);
224 
225     for (int i = 0; i <= this->n; i++)
226         for (int j = 0; j <= i; j++)
227             aux.mat[i][j] = this->mat[i][j] / value;
228 
229     return aux;
230 }
231 inline Jet Jet::operator+=(Jet jet)
232 {
233     *this = jet + *this;
234     return *this;
235 }
236 inline Jet Jet::operator-=(Jet jet)
237 {
238     *this = *this - jet;
239     return *this;
240 }
241 inline Jet Jet::operator*=(const Jet jet)
242 {
243     *this = *this * jet;
244     return *this;
245 }
246 inline Jet Jet::operator/=(const Jet jet)
247 {
248     *this = *this / jet;
249     return *this;
250 }
251 inline Jet Jet::operator+=(double value)
252 {
253     *this = *this + value;
254     return *this;
255 }
256 inline Jet Jet::operator-=(double value)
257 {
258     *this = *this - value;
259     return *this;
260 }
261 inline Jet Jet::operator*=(double value)
262 {
263     *this = *this * value;
264     return *this;
265 }
266 inline Jet Jet::operator/=(double value)
267 {
268     *this = *this / value;
269     return *this;
270 }
271 inline Jet operator+(double value, Jet jet)
272 {
273     Jet aux(jet.n);
274     for (int i = 0; i <= jet.all_elem; i++) {
275         *(*aux.mat) = *(*jet.mat + i);
276     }
```

```
277     aux.mat[0][0] += value;
278     return aux;
279 }
280 inline Jet operator-(double value, Jet jet)
281 {
282     Jet aux(jet.n);
283     for (int i = 0; i <= jet.all_elem; i++) {
284         *(*aux.mat) = *(*jet.mat + i);
285     }
286     aux.mat[0][0] -= value;
287     return aux;
288 }
289 inline Jet operator*(double value, Jet jet)
290 {
291     Jet aux(jet.n);
292     for (int i = 0; i <= jet.all_elem; i++) {
293         *(*aux.mat) = *(*jet.mat + i) * value;
294     }
295     return aux;
296 }
297 inline Jet operator/(double value, Jet jet)
298 {
299     Jet aux(jet.n);
300     for (int i = 0; i <= jet.all_elem; i++) {
301         *(*aux.mat) = *(*jet.mat + i) / value;
302     }
303     return aux;
304 }
305 inline Jet Jet::operator-()
306 {
307     Jet aux(this->n);
308     for (int i = 0; i <= n; i++)
309         for (int j = 0; j <= i; j++)
310             aux.mat[i][j] = -this->mat[i][j];
311     return aux;
312 }
313 inline Jet Jet::sin()
314 {
315     Jet aux(this->n);
316     int deg = this->all_elem;
317 
318     double* a = new double[deg + 1];
319     a[0] = std::cos(this->mat[0][0]);
320     aux.mat[0][0] = (std::sin(this->mat[0][0]));
321     double a_sum;
322     double b_sum;
323     for (int n = 1; n <= deg; n++) {
324         a_sum = 0;
325         b_sum = 0;
326         for (int j = 1; j <= n; j++) {
327             a_sum += j * *(*aux.mat + n - j) * *(*this->mat + j);
328             b_sum += j * a[n - j] * *(*this->mat + j);
329         }
330         a[n] = -(1.0 / n) * a_sum;
331         *(*aux.mat + n) = (1.0 / n) * b_sum;
332     }
333     delete[] a;
334     return aux;
335 }
336 inline Jet Jet::cos()
337 {
338     Jet aux(this->n);
339     int deg = this->all_elem;
340     double* b = new double[deg + 1];
341     aux.mat[0][0] = (std::cos(this->mat[0][0]));
342     b[0] = std::sin(this->mat[0][0]);
343     double a_sum;
344     double b_sum;
345     for (int i = 1; i <= deg; i++) {
346         a_sum = 0;
```

```
347         b_sum = 0;
348         for (int j = 1; j <= i; j++) {
349             a_sum += j * b[i - j] * *(*this->mat + j);
350             b_sum += j * *(*aux.mat + i - j) * *(*this->mat + j);
351         }
352         *(*aux.mat + i) = -(1.0 / i) * a_sum;
353         b[i] = (1.0 / i) * b_sum;
354     }
355     delete[] b;
356     return aux;
357 }
358 inline double Jet::aval(double s)
359 {
360     double result = 0.0;
361     for (int i = 0; i <= this->all_elem; i++) {
362         result += *(*this->mat + i) * pow(s, i);
363     }
364     return result;
365 }
```

B.3 Implementació dels mètodes de Newton i Runge-Kutta-Fehlberg

```
1  #include "jet.cpp"
2  #include <cmath>
3  #include <fstream>
4  #include <iomanip>
5  #include <iostream>
6
7  #define PI 3.14159265358979323846
8  double eps = 0.001;
9  double omega = sqrt(2);
10
11 using namespace std;
12
13 double E1 = (1. / 360.);
14 double E2 = (-128. / 4275.);
15 double E3 = (-2197. / 75240.);
16 double E4 = (1. / 50.);
17 double E5 = (2. / 55.);
18
19 double R1 = (16. / 135.);
20 double R2 = (6656. / 12825.);
21 double R3 = (28561. / 56430.);
22 double R4 = (-9. / 50.);
23 double R5 = (2. / 55.);
24
25 double C1 = (12. / 13.);
26 double C2 = (1932. / 2197.);
27 double C3 = (-7200. / 2197.);
28 double C4 = (7296. / 2197.);
29 double C5 = (439. / 216.);
30 double C6 = (3680. / 513.);
31 double C7 = (-845. / 4104.);
32 double C8 = (-8. / 27.);
33 double C9 = (-3544. / 2565.);
34 double C10 = (1859. / 4104.);
35
36 void camp(double t, Jet* x, int n, Jet* result)
37 {
38     double aux = eps * sin(omega * t);
39     result[0] = x[1];
40     result[1] = (-x[0].sin()) + aux;
41 }
```

```
42
43 void calcular_ks(double t, Jet* x, int n, double h, Jet** k, void (*camp)(double,↩
       Jet*, int, Jet*))
44 {
45     int i;
46     Jet aux[2];
47
48     aux[0] = new Jet(2);
49     aux[1] = new Jet(2);
50
51     // calcul de k_1
52     (*camp)(t, x, n, k[0]);
53     for (i = 0; i < n; ++i) {
54         k[0][i] *= h;
55     }
56
57     // calcul de k_2
58     for (i = 0; i < n; i++)
59         aux[i] = x[i] + k[0][i] * 0.25;
60
61     (*camp)(t + 0.25e0 * h, aux, n, k[1]);
62     for (i = 0; i < n; i++)
63         k[1][i] *= h;
64
65     // calcul de k_3
66     for (i = 0; i < n; i++)
67         aux[i] = x[i] + k[0][i] * (3. / 32) + k[1][i] * (9. / 32);
68     (*camp)(t + 0.375e0 * h, aux, n, k[2]);
69     for (i = 0; i < n; i++)
70         k[2][i] *= h;
71     for (i = 0; i < n; i++)
72         aux[i] = x[i] + k[0][i] * C2 + k[1][i] * C3 + k[2][i] * C4;
73
74     // calcul de k_4
75     (*camp)(t + C1 * h, aux, n, k[3]);
76     for (i = 0; i < n; i++)
77         k[3][i] *= h;
78     for (i = 0; i < n; i++)
79         aux[i] = x[i] + k[0][i] * C5 - k[1][i] * 8. + k[2][i] * C6 + k[3][i] * C7↩
             ;
80
81     // calcul de k_5
82     (*camp)(t + h, aux, n, k[4]);
83     for (i = 0; i < n; i++)
84         k[4][i] *= h;
85     for (i = 0; i < n; i++)
86         aux[i] = x[i] + k[0][i] * C8 + k[1][i] * 2 + k[2][i] * C9 + k[3][i] * C10↩
              - k[4][i] * (11. / 40);
87
88     (*camp)(t + 0.5 * h, aux, n, k[5]);
89     for (i = 0; i < n; ++i)
90         k[5][i] *= h;
91
92     return;
93 }
94
95 double calcular_error(int n, Jet** k)
96 {
97     double aux;
98     double error;
99     error = 0.0;
100    for (int i = 0; i < n; i++) {
101        aux = E1 * k[0][i].mat[0][0] + E2 * k[2][i].mat[0][0] + E3 * k[3][i].mat↩
              [0][0] + E4 * k[4][i].mat[0][0] + E5 * k[5][i].mat[0][0];
102        aux += E1 * k[0][i].mat[1][0] + E2 * k[2][i].mat[1][0] + E3 * k[3][i].mat↩
              [1][0] + E4 * k[4][i].mat[1][0] + E5 * k[5][i].mat[1][0];
103        aux += E1 * k[0][i].mat[1][1] + E2 * k[2][i].mat[1][1] + E3 * k[3][i].mat↩
              [1][1] + E4 * k[4][i].mat[1][1] + E5 * k[5][i].mat[1][1];
104        error += aux * aux;
105    }
```

```
106     return sqrt(error);
107 }
108
109 void step_rk(Jet* x, int n, Jet** k)
110 {
111     for (int i = 0; i < n; i++) {
112         x[i].mat[0][0] = x[i].mat[0][0] + k[0][i].mat[0][0] * R1 + k[2][i].mat↩
                [0][0] * R2 + k[3][i].mat[0][0] * R3 + k[4][i].mat[0][0] * R4 + k[5][↩
                i].mat[0][0] * R5;
113         x[i].mat[1][0] = x[i].mat[1][0] + k[0][i].mat[1][0] * R1 + k[2][i].mat↩
                [1][0] * R2 + k[3][i].mat[1][0] * R3 + k[4][i].mat[1][0] * R4 + k[5][↩
                i].mat[1][0] * R5;
114         x[i].mat[1][1] = x[i].mat[1][1] + k[0][i].mat[1][1] * R1 + k[2][i].mat↩
                [1][1] * R2 + k[3][i].mat[1][1] * R3 + k[4][i].mat[1][1] * R4 + k[5][↩
                i].mat[1][1] * R5;
115     }
116 }
117
118 int rkf45(double* at, Jet* x, int n, double* ah, int sc, double tol, double* atf,↩
       double* aer, void (*ode)(double, Jet*, int, Jet*))
119 {
120     Jet** k;
121     double error, t, h;
122     int flag = 0;
123
124     // óDefinici de k
125     k = new Jet*[6];
126     if (!k) {
127         cout << "memory problem" << endl;
128         exit(1);
129     }
130     k[0] = new Jet[6 * n];
131     if (!k[0]) {
132         cout << "memory problem" << endl;
133         exit(1);
134     }
135     k[0] = *k;
136     for (int i = 1; i < 6; i++) {
137         k[i] = k[i - 1] + n;
138     }
139
140     // guardem localment les variables t i h
141     t = *at;
142     h = *ah;
143
144     // comprovem si el pas actual àser lú'ltim
145     if (t + h > *atf) {
146         h = *atf - t;
147         flag = 1;
148     }
149
150     // calculem les kappa
151     calcular_ks(t, x, n, h, k, ode);
152
153     // sc = 0 indica que no hi ha control de pas
154     if (sc == 0) {
155         if (aer != NULL)
156             *aer = calcular_error(n, k);
157
158         step_rk(x, n, k);
159
160         *at = t + h;
161
162         delete[] k[0];
163         delete[] k;
164         return flag;
165     }
166
167     // en el cas que es tracti de lú'ltim pas
168     if (flag == 1) {
```

```
169         // calculem l'error i lú'ltim pas
170         *aer = calcular_error(n, k);
171         step_rk(x, n, k);
172 
173         // actualtizem les variables
174         *at = t + h;
175         *ah = h;
176 
177         delete[] k[0];
178         delete[] k;
179         return flag;
180     }
181 
182     // calculem l'error
183     error = calcular_error(n, k);
184 
185     // comprovem que aquest no sigui massa gran
186     while (error > tol) {
187         h = 0.9 * h * pow(tol / error, 0.2);
188 
189         calcular_ks(t, x, n, h, k, ode);
190         error = calcular_error(n, k);
191     }
192 
193     // fem el nou pass de Runge-Kutta
194     step_rk(x, n, k);
195 
196     h = 0.9 * h * pow(tol / error, 0.2);
197 
198     // actualitzem les variables
199     *at = t + h;
200     *ah = h;
201     *aer = error;
202 
203     delete[] k[0];
204     delete[] k;
205     return flag;
206 }
207 void cramer(Jet* x, double initial[2], double h[2])
208 {
209     double a1, b1, a2, b2, f1, f2;
210 
211     // calculem la diferencial de époincar DP - Id
212     a1 = x[0].mat[1][0] - 1.0;
213     b1 = x[0].mat[1][1];
214     a2 = x[1].mat[1][0];
215     b2 = x[1].mat[1][1] - 1.0;
216     // calculem els valors de -F(x) -> F(x) = punt final - inicial
217     f1 = initial[0] - x[0].mat[0][0];
218     f2 = initial[1] - x[1].mat[0][0];
219 
220     // calculem les h per cramer
221     h[0] = (f1 * b2 - f2 * b1) / (a1 * b2 - a2 * b1);
222     h[1] = (f2 * a1 - f1 * a2) / (a1 * b2 - a2 * b1);
223 
224     // calculem el nou punt com x1 = x0 + h
225     x[0].mat[0][0] = initial[0] + h[0];
226     x[1].mat[0][0] = initial[1] + h[1];
227 }
228 double norma(double x[2])
229 {
230     return sqrt(x[0] * x[0] + x[1] * x[1]);
231 }
232 void buscar_punt_fix(double* at, Jet* x, int n, double* ah, int sc, double tol, ↩
        double* atf, double* aer, void (*ode)(double, Jet*, int, Jet*))
233 {
234     double h_newton[2], initial[2];
235 
236     // guardem les variables locals i inicialitzem
237     h_newton[0] = 1.0;
```

```
238     h_newton[1] = 1.0;
239
240     // fem el procediment fins que la h sigui prou petita
241     while (norma(h_newton) > tol) {
242         // inicialitzem les variables
243         x[0].mat[1][0] = 1.0;
244         x[0].mat[1][1] = 0.0;
245         x[1].mat[1][0] = 0.0;
246         x[1].mat[1][1] = 1.0;
247         *at = 0;
248         *ah = 1.e-2;
249
250         initial[0] = x[0].mat[0][0];
251         initial[1] = x[1].mat[0][0];
252
253         // mentre rkf45 sigui 0, no haurem arribat al punt at = atf
254         while (rkf45(at, x, n, ah, sc, tol, atf, aer, ode) == 0)
255             ;
256
257         // trobem el valor de h per cramer
258         cramer(x, initial, h_newton);
259
260         // imprimim els resultats
261         cout << "h_newton = " << h_newton[0] << ", " << h_newton[1] << endl;
262         cout << "hnewton norma: " << norma(h_newton) << endl;
263     }
264
265     // comprovem que el resultat del determinant sigui 1
266     cout << endl
267          << "determinant = "
268          << x[0].mat[1][0] * x[1].mat[1][1] - x[0].mat[1][1] * x[1].mat[1][0]
269          << endl;
270 }
271
272 int main()
273 {
274     cout.precision(16);
275     int n = 2, sc = 1;
276     double t = 0, h = 1.e-2, tol = 1.e-10, tf = 2 * PI / omega, aer = 1;
277
278     Jet f[2];
279
280     f[0] = new Jet(2);
281     f[1] = new Jet(2);
282
283     f[0].mat[0][0] = 2 * PI / omega;
284     f[1].mat[0][0] = 0.0;
285     f[0].mat[1][0] = 1.;
286     f[0].mat[1][1] = 0.;
287     f[1].mat[1][0] = 0.;
288     f[1].mat[1][1] = 1.;
289
290     buscar_punt_fix(&t, f, n, &h, sc, tol, &tf, &aer, camp);
291     cout << "\npunt resultant = " << f[0].mat[0][0] << "\t" << f[1].mat[0][0] << ↩
            endl;
292     cout << "\nmatriu =\n"
293          << f[0].mat[1][0] << "\t\t" << f[0].mat[1][1] << "\n"
294          << f[1].mat[1][0] << "\t\t" << f[1].mat[1][1] << endl
295          << endl;
296     return 0;
297 }
```