INTEGRATION DES SYSTEMES DIFFERENTIELS PAR RUNGE-KUTTA

par Charles Hubert

Le 03/01/2011

Introduction

Un système différentiel d'ordre n peut toujours s'écrire

$$\frac{dx}{dt} = Eqd(x) \qquad x \in R^n$$

où "t" est la variable indépendante et "x" une fonction inconnue de t qui prend ses valeurs dans l'espace vectoriel réel de dimension n. La fonction "Eqd" décrit les équations différentielles. Quand t est le temps on dit que x est le vecteur d'état et que ses composantes sont les variables d'état. La variable indépendante t peut très bien figurer dans les équations, c'est-à-dire dans la fonction Eqd ; il faut pour cela que t soit une composante du vecteur x et on indiquera dans la fonction Eqd que sa dérivée vaut un. Pour intégrer numériquement un système différentiel, la méthode de Runge-Kutta d'ordre 4, très utilisée, part d'un état initial

$$x(t_0)$$

et, le pas d'intégration "Δt" étant donné, elle calcule itérativement

$$x(t_0 + \Delta t) \quad x(t_0 + 2\Delta t) \quad x(t_0 + 3\Delta t) \quad ... \quad x(t_0 + k\Delta t)$$

en sollicitant la fonction Eqd(x) quatre fois à chaque pas Δt.

Elle est précise quand Δt est suffisamment petit : l'erreur cumulée entre deux valeurs de t fixées est approximativement proportionnelle à la puissance 4 de Δt. Mais si Δt est trop grand la méthode diverge, donnant des résultats faux.

Cet article présente des fonctions qui la mettent en oeuvre quel que soit le système différentiel. Ces fonctions ont quelque longueur, mais on les entre une fois pour toutes dans une bibliothèque. On les copie pour chaque application, et la longueur des fonctions qu'il faut écrire est déterminée par celle du système différentiel à intégrer. Le vecteur d'état est géré suivant la méthode décrite dans l'article

TABLEAUX DANS UN VECTEUR UNIQUE

L'algorithme d'intégration

On intègre le système différentiel par la fonction "RunKut". Son fonctionnement n'est pas influencé par □IO.

Pour expliquer son utilisation prenons un exemple dont la solution analytique est connue, ce qui permettra de vérifier la précision. Soient les deux fonctions

$$x = \frac{a}{1 + bt^2} \qquad y = \frac{-2b}{1 + bt^2}$$

dépendant de 2 constantes arbitraires "a" et "b". Par dérivation et élimination de ces constantes on trouve qu'elles sont la solution générale du système différentiel

$$x' = txy^2 \qquad y' = ty$$

Faisons de t une variable d'état vérifiant l'équation évidente

$$t' = 1$$

On décrit le système différentiel par la fonction suivante qui calcule les dérivées "Detat" du vecteur d'état à partir du vecteur d'état "etat" lui-même

```
    ∇  Detat←Eqd0 etat
[1]    Detat←etat
[2]    Dt 1      ⍝ t' = 1
[3]    Dx t×x×y  ⍝ x' = t x y
[4]    Dy t×y*2  ⍝ y' = t y y
    ∇
```

Le résultat de Eqd0 a nécessairement la même dimension que le vecteur d'état, et puisque toutes les cases du tableau Detat seront affectées, la solution la plus simple est la ligne Eqd0[1]. Les autres lignes de Eqd0 calculent t', x', y' à partir de t, x, y conformément aux équations différentielles.

On prépare l'intégration au moyen de la fonction d'initialisation

```
    ∇  Detat←Ini0 X;t0;etat
[1]    ⎕IO←0
[2]    (t0 a b)←c''3↑X
[3]    1 VarEtat 't x y'
[4]    Dt t0             ⍝ t initial
[5]    Dx   a ÷1+b×t0*2  ⍝ x initial
[6]    Dy ¯2×b÷1+b×t0*2  ⍝ y initial
    ∇
```

Ini0[2] récupère dans l'argument "X" le t initial et affecte les constantes a, b qui serviront à la vérification ultérieure. Ini0[3] déclare les 3 variables d'état scalaires. Les fonctions créées par VarEtat ne permettent d'affecter des valeurs qu'à Detat, c'est cette variable qui est choisie comme vecteur d'état initial à l'intérieur de la fonction Ini0 et ses valeurs sont affectées par Ini0[4], Ini0[5], Ini0[6] ; c'est la méthode d'initialisation la plus simple.

Pour faire certains calculs sur le vecteur d'état il est nécessaire que Eqd0 puisse s'appliquer à une liste de vecteurs d'état. On a choisi ici que cette liste soit une matrice dont chaque ligne est une valeur du vecteur d'état ; l'argument gauche de VarEtat égal à ''1'' indique ce choix et "RunKut" le saura parce que son argument droit est une matrice.

Pour intégrer le système Eqd0 de t = t0 à t0+2 avec 100 pas d'intégration pour t0=0, a=1, b=2 écrivons l'une des instructions

```
      etat← 2 100 'Eqd0' RunKut Ini0 0 1 2
      etat←(2 100)'Eqd0' RunKut Ini0 0 1 2
      etat← 'Eqd0' 2 100 RunKut Ini0 0 1 2
```

car RunKut trie les nombres et les caractères de l'argument gauche. Si le nom des équations est omis ou s'il est vide, c'est "EqaDif" qui est pris par défaut. RunKut indique les valeurs par défaut de l'argument gauche si on l'appelle sans cet argument. On trouve

```
      ⍴etat
101 3
```

Vérifions t, x, y d'après la solution connue :

```
      ⌈/|t-0.02×⍳101
1.3323E¯15
      ⌈/''| x y - (a,¯2×b)÷c1+b×t*2
6.1162E¯9 2.4465E¯8
```

On peut encore subdiviser le pas 0.02 en "n" parties, le vrai pas étant alors 0.02÷n, sans retenir les états intercalés :

```
⍝                   n
⍝                  ┌┐
      ⍴etat← 2 100 2 'Eqd0' RunKut Ini0 0 1 2
101 3
      ⌈/|t-0.02×⍳101
1.5543E¯15
```

cela améliore la précision :

```
      ⌈/''| x y - (a,¯2×b)÷c1+b×t*2
3.8093E¯10 1.5237E¯9
```

et en comparant avec le pas non subdivisé :

```
      6.1162E¯9 2.4465E¯8 ÷ 3.8093E¯10 1.5237E¯9
16.056 16.056
```

on voit que l'erreur a été approximativement divisée par 2 puissance 4, comme annoncé dans l'introduction.

On peut intégrer plusieurs cas simultanément, chacune des données de l'argument de Ini0 est enfermée dans une case par Ini0[2] :

```
      ρetat← 2 100 'Eqd0' RunKut Ini0 0 1 (1+⍳5)
101 3
```

ce qui exécute RunKut une fois sur des tableaux gigognes. Comparons avec ce que donne l'instruction suivante qui exécute RunKut cinq fois sur des tableaux simples en consommant plus de temps de calcul :

```
      ⌈/|etat-⊃[0]⊃(⊂ 2 100 'Eqd0')RunKut''Ini0''(⊂ 0 1),''1+⍳5
 0 0 0 0 0  0 0 0 0 0  0 0 0 0 0
```

Si on choisit de faire de la liste des états un vecteur gigogne au lieu d'une matrice, on appellera RunKut avec un vecteur comme argument droit, au lieu d'une matrice ; chaque colonne de celle-ci est enfermée dans la case correspondante du vecteur. On indiquera ce choix à VarEtat en lui donnant ''0'' comme argument gauche. Si on veut intégrer plusieurs cas en même temps, il faut alors que les diverses valeurs de tout paramètre soient plus profonds d'un niveau, le niveau 3 au moins ; car le niveau 2 est réservé aux diverses valeurs du vecteur d'état. La fonction Eqd0 ne change pas :

```
    ∇  Detat←Eqd0 etat
[1]    Detat←etat
[2]    Dt 1      ⍝ t' = 1
[3]    Dx t×x×y  ⍝ x' = t x y
[4]    Dy t×y*2  ⍝ y' = t y y
    ∇
```

tandis que la fonction Ini0 s'écrit alors

```
    ∇  Detat←Ini0 X;t0;etat
[1]    ⎕IO←0
[2]    (t0 a b)←⊂''⊂''3↑X
[3]    0 VarEtat 't x y'
[4]    Dt t0             ⍝ t initial
[5]    Dx   a ÷1+b×t0*2  ⍝ x initial
[6]    Dy ¯2×b÷1+b×t0*2  ⍝ y initial
    ∇
```

seules les lignes [2] et [3] changent.

On écrit encore

```
      ρetat← 2 100 'Eqd0' RunKut Ini0 0 1 2
3
```

et on peut vérifier :

```
      ⌈/''|t-⊂0.02×⍳101
1.3323E¯15
      ⌈/''''| x y - (a,¯2×b)÷⊂1+b×t*2
6.1162E¯9 2.4465E¯8
```

La liste des variables d'état est dans la variable "varEtat". La liste des objets créés par VarEtat est dans la variable "objEtat".

Intégration abandonnée

Dans certains problèmes on veut arrêter l'intégration si une condition survient, par exemple parce que quelque chose est jugé anormal. Pour cela on affectera la valeur ''0'' à une variable globale dans Eqd0 si l'intégration peut être poursuivie et ''1'' s'il faut l'arrêter. Reprenons l'exemple précédent sur tableaux simples :

```
    ∇  Detat←Eqd0 etat
[1]    Detat←etat
[2]    arret←∨/100<|¢y
[3]    Dt 1       ⍝ t' = 1
[4]    Dx t×x×y   ⍝ x' = t x y
[5]    Dy t×y*2   ⍝ y' = t y y
    ∇
```

la variable "arret" indique qu'il faut arrêter si l'une des valeurs absolues de y dépasse 100.
Reprenons l'exemple déjà traité

```
      ρetat← 2 100 'Eqd0' 'arret' RunKut Ini0 0 1 2
101 3
      ⌈/"| x y -(a,¯2×b)÷c1+b×t*2
6.1162E¯9 2.4465E¯8
```

et rien n'est changé.

Mais si $b = -2$ on sait que la solution est infinie pour $t = \sqrt{1/2} = 0.70711...$; essayons :

```
      ρetat← 2 100 'Eqd0' 'arret' RunKut Ini0 0 1 ¯2
36 3
      33 0 ↓etat
 0.66  7.7619  31.047
 0.68 13.268   53.072
 0.7  45.396  181.59
```

l'intégration a été abandonnée et la précision est catastrophique :

```
      ⌈/"| x y -(a,¯2×b)÷c1+b×t*2
4.6037 18.415
```

En subdivisant le pas par 10 :

```
      ρetat← 2 100 10 'Eqd0' 'arret' RunKut Ini0 0 1 ¯2
36 3
      33 0 ↓etat
 0.66   7.764  31.056
 0.68  13.298  53.191
 0.694 27.227 108.91
```

la précision s'est améliorée :

```
      ⌈/"| x y -(a,¯2×b)÷c1+b×t*2
0.00018087 0.00072346
```

Fonction "équations" avec plusieurs arguments

Considérons maintenant l'équation

$x'' - Sx' + Px = 0$ avec $S = a + b \quad P = ab$

qu'il faut écrire avec une fonction inconnue supplémentaire :

$x' = y \quad y' = Sy - Px$

afin de n'avoir que des dérivées premières. Ici aussi la solution générale est connue. Limitons-nous au cas où a et b sont réels différents :

$x = A \exp at + B \exp bt$
$y = aA \exp at + bB \exp bt$

"A" et "B" étant deux constantes arbitraires.

Le système différentiel dépend des deux paramètres S et P, les coefficients de l'équation initiale. Si on veut que ceux-ci soient en argument gauche des équations on écrit

```
    ∇  Detat←SP Eqd1 etat;P;S
[1]    (S P)←SP
[2]    Detat←etat
[3]    arret←∨/100<|¢y
[4]    Dt 1          ⍝  t' = 1
[5]    Dx y          ⍝  x' = y
[6]    Dy (S×y)-P×x  ⍝  y' = S y - P x
    ∇
```

Avec la fonction d'initialisation

```
    ∇  Detat←Ini1 X;w;etat
[1]   ⍝ X ← t0 a b A B
[2]    ⎕IO←0
[3]   ⍝ S ← a+b    P ← ab
[4]    ab←X[1 2] ⋄ AB←X[3 4] ⋄ S←+/ab ⋄ P←×/ab
[5]    1 VarEtat 't x y'
[6]    Dt X[0]        ⍝ t initial
[7]    w←⊃(⍟X[0]×ab)+.××\AB,[0.5]ab
[8]    Dx w[0]        ⍝ x initial
[9]    Dy w[1]        ⍝ y initial
    ∇
```

on peut intégrer :

```
      ρetat← 5 100 'S P Eqd1' 'arret' RunKut Ini1 0 ¯0.5 ¯1 2 ¯2
101 3
```

et vérifier :

```
      ⌈/¨| x y -c[0]⊃(⍟t°.×ab)+.××\AB,[0.5]ab
3.7937E¯8 3.8944E¯8
```

Ou encore

```
      ρetat← 5 100 5 'S P Eqd1' 'arret' RunKut Ini1 0 1 2 2 ¯2
35 3
      32 0 ↓etat
 1.6  ¯39.159  ¯88.224
 1.65 ¯43.811  ¯98.037
 1.66 ¯44.802 ¯100.12
      ⌈/¨| x y -c[0]⊃(⍟t°.×ab)+.××\AB,[0.5]ab
2.394E¯7 4.8024E¯7
```

Autre méthode, S et P étant en argument droit des équations :

```
    ∇  Detat←Eqd1 X;S;P;etat
[1]    (S P etat)←X
[2]    Detat←etat
[3]    arret←∨/100<|∈y
[4]    Dt 1          ⍝  t' = 1
[5]    Dx y          ⍝  x' = y
[6]    Dy (S×y)-P×x  ⍝  y' = S y - P x
    ∇
```

et on écrit

```
      ρetat← 5 100 'Eqd1 S P' 'arret' RunKut Ini1 0 ¯0.5 ¯1 2 ¯2
101 3
      ⌈/¨| x y -c[0]⊃(⍟t°.×ab)+.××\AB,[0.5]ab
3.7937E¯8 3.8944E¯8
```

Technique de programmation de RunKut

La fonction RunKut crée une fonction locale `δ_Int.` C'est cette fonction locale qui exécute chaque pas d'intégration.

Pour cela RunKut sélectionne parmi ses propres lignes celles qui commencent par `⍝i`, y copie 4 fois la syntaxe de la fonction "équations" et le cas échéant 1 fois la syntaxe de la condition "arrêt". Par exemple, si Runkut est appelée par

```
⍝                équations
⍝              ┌─────────┐
      ... 5 100 'a b Equat c' RunKut ...
```

la copie textuelle de la fonction "équations" lui fait solliciter les équations ainsi

```
⍝          équations
⍝        ┌─────────┐
      ... a b Equat c (argumentLocal)
```

On peut lister cette fonction locale en insérant `⎕CR` au début de la ligne RunKut[15].

Cette méthode évite d'enfermer dans une boucle la primitive exécute `⍎` qui appellerait les équations, ce qui coûterait plus cher en temps de calcul.

Application à l'oscillateur de Van der Pol

Cet oscillateur obéit au système différentiel

$$x' = y \qquad y' = (k_0 - k_1 x^2)\, y - x$$

qu'on peut programmer ainsi :

```
    ∇  Detat←EqdVdp etat
[1]    Detat←etat
[2]    Dt 1                      ⍝  t' = 1
[3]    Dx y                      ⍝  x' = y
[4]                              ⍝                  2
[5]    Dy (y×k[0]-k[1]×x*2)-x    ⍝  y' = y(k - k x ) - x
[6]                              ⍝           0    1
    ∇
```

On choisit

$$k_0 = 0.5 \qquad k_1 = 0.05$$

et on abandonne cet oscillateur avec y = 0, et une liste de valeurs pour x. La fonction d'initialisation peut être

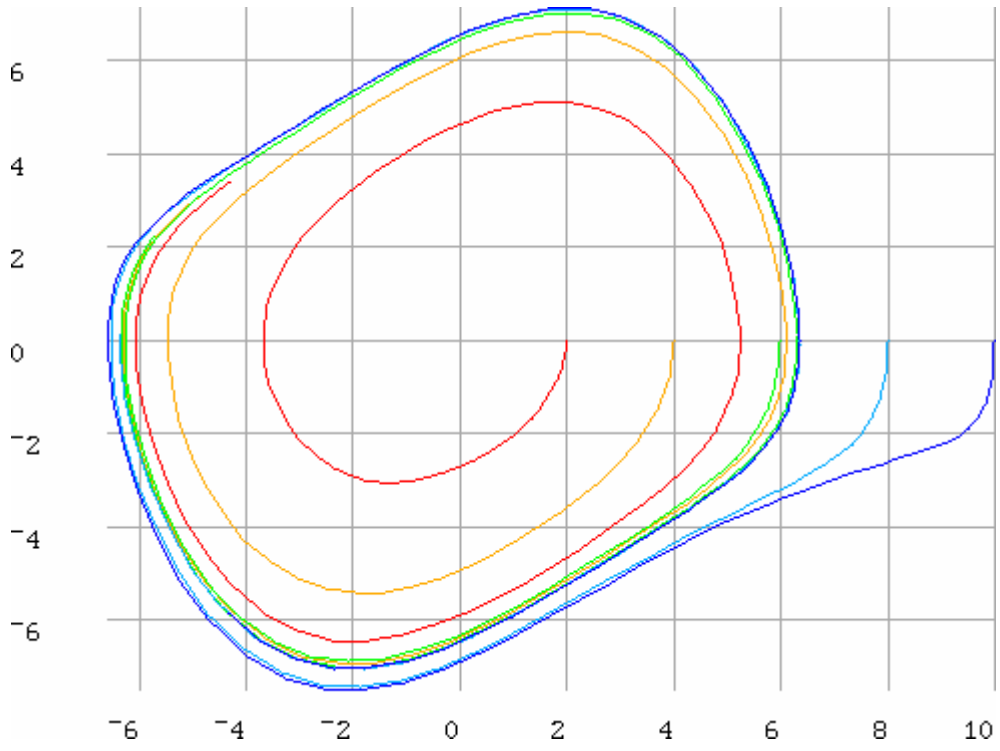
```
    ∇  Detat←IniVdp x0;etat
[1]    ⎕io←0
[2]    1 VarEtat 't x y'
[3]    k← 0.5 0.05       ⍝  k     k
[4]                      ⍝   0     1
[5]    Dx ⊂x0            ⍝  x initial
    ∇
```

Les variables non initialisées (t et y) démarrent à zéro.

L'intégration de t = 0 à 10 avec 2, 4, 6, 8, 10 pour valeurs initiales de x peut se faire par l'instruction

```
etat← 10 100 'EqdVdp' RunKut IniVdp 2 4 6 8 10
```

La figure (appelée plan de phase) montre les solutions avec x en abscisses et y en ordonnées :



Application au problème des N corps en mécanique céleste

Considérons N corps en interaction gravitationnelle, en négligeant l'interaction de leurs rotations avec leurs mouvements orbitaux. Les méthodes analytiques ne savent pas intégrer ce système d'équations pour N > 2. Posons

$\vec{r}_j$ = position du corps j $\vec{v}_j$ = vitesse du corps j

m_j = masse du corps j t = temps

G = constante de gravitation = 6.6732×10^{-11} $m^3\,s^{-2}\,kg^{-1}$

Le mouvement de ces corps obéit au système différentiel

$$\vec{r}_j' = \vec{v}_j \qquad \vec{v}_j' = -\sum_{k\neq j} Gm_k \frac{\vec{r}_j - \vec{r}_k}{|\vec{r}_j - \vec{r}_k|^3}$$

On peut s'arranger pour que l'expression sous le signe de sommation s'annule sans incident pour k = j grâce à une butée inférieure ε positive très petite et on somme pour toutes les valeurs de k :

$$\vec{v}_j' = -\sum_k \frac{Gm_k}{\max\left[\left[(\vec{r}_j - \vec{r}_k)^2\right]^{3/2}, \varepsilon\right]} (\vec{r}_j - \vec{r}_k)$$

Suivant la méthode exposée ici le système différentiel peut se programmer (en unités SI) de la façon suivante

```
    ∇  Detat←m EqdNcor etat;d
[1]    Detat←etat
[2]    Dt 1
[3]    Dr v
[4]    d←⊃[0 3]d°.-d←c[0 2]r
[5]    Dv +/[2](¯6.6732E¯11×m)×[2]d÷[0 1 2]1E¯300⌈(+/d×d)*1.5
    ∇
```

On a pris $1E^{-300}$ pour la butée ε.

EqdNcor[4] calcule les différences de positions "d" et EqdNcor[5] calcule les accélérations. On initialise avec

```
    ∇  Detat←IniNcor x;n;t0;etat
[1]   ⍝ x ← t0 (⊃ () ... (m   x   vx  y   vy  z   vz) ... () )
[2]   ⍝                   └──────────────────────────┘
[3]   ⍝                              j-ième corps
[4]    ⎕IO←0
[5]    (t0 x)←x
[6]    x←((x/¯1↓⍴x),7)⍴((¯1↓⍴x),7)↑x
[7]    n←⍴m←x[;0]
[8]    1 VarEtat 't r' n 3 'v' n 3
[9]    Dt t0
[10]   Dr x[; 1 3 5]
[11]   Dv x[; 2 4 6]
    ∇
```

On peut intégrer ce système différentiel pour un nombre de corps N quelconque, par exemple les 10 principaux corps du système solaire. Pour prendre un autre exemple, dans le plan orbital de Jupiter, on peut construire un triangle équilatéral dont le Soleil et Jupiter sont deux sommets ; le troisième sommet est un point stable de Lagrange. Il y a évidemment deux points de Lagrange, l'un en avance d'1/6 de tour sur Jupiter, l'autre en retard. On appelle planètes troyennes les corps capturés au voisinage de ces deux points. Prenons comme exemple le Soleil, Jupiter et une planète troyenne.

La fonction suivante construit un tableau "x" à 3 lignes : Soleil, Jupiter, planète troyenne. La

masse attribuée à cette dernière est une valeur arbitraire. La position initiale du Soleil est l'origine et sa vitesse initiale est nulle. La position et la vitesse initiales de la planète troyenne sont calculées de manière qu'elle coïncide en permanence avec un point de Lagrange :

```
    ∇  x←Troy;⎕IO;a;c;T;cs
[1]    ⎕IO←0
[2]    a←778.3E9 ⋄ c←37.4E9 ⋄ T←3.7434E8
[3]    x← 3 7 ρ0 ⋄ x[;0]← 2E30 1.899E27 1E23
[4]    x[1;1]←a-c
[5]    x[1;4]←○a×(((a+c)÷a-c)*0.5)÷T×0.5
[6]    x[2; 1 2]←⊃x[1; 2 2 ρ 1 3 2 4]-.×cs← 0.25 0.75 *0.5
[7]    x[2; 3 4]←⊃x[1; 2 2 ρ 1 3 2 4]+.×⌽cs
    ∇
```

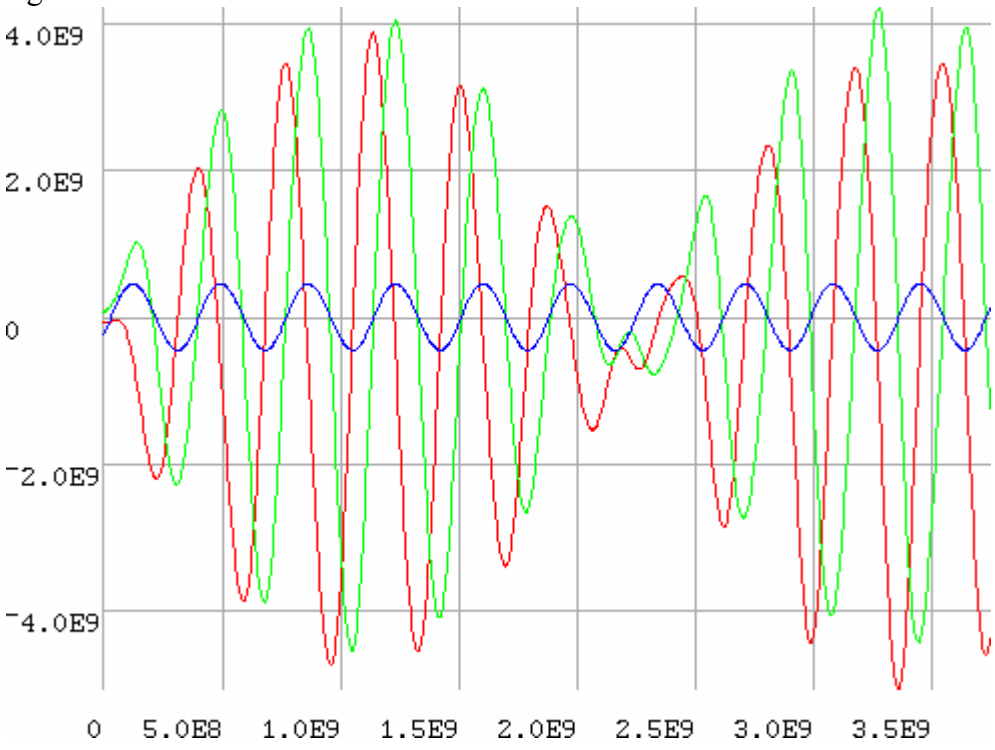
On peut alors initialiser la planète troyenne à une position et une vitesse décalées par rapport à la coïncidence avec un point de Lagrange calculé précédemment et intégrer avec un pas égal à 1/200 de la période orbitale :

```
X←Troy ⋄ X[2;1+⍳6]←X[2;1+⍳6]+(6ρX[1; 1 4]×0.0001)× ¯1 0 1 2 ¯3 5
etat← 3.7434E9 1000 2 'm EqdNcor' RunKut IniNcor 0 X
```

On peut calculer les fluctuations "F" autour du point de Lagrange instantané par les instructions

```
cs← 2 2 ρ 1 1 ¯1 × 0.25 0.75 0.75 *0.5
F←(-/[1]r[; 2 0 ;])-(⊃(-/[1]r[; 1 0 ; 0 1])+.×cs),0
```

et tracer les 3 composantes de F en fonction du temps t. On observe alors des oscillations non divergentes.



Dans la solution obtenue le Soleil n'est pas tout à fait immobile, ce qui est conforme à la théorie. On peut aussi vérifier la constance de la quantité de mouvement totale $\vec{P}$, du moment cinétique total $\vec{L}$ et de l'énergie totale H (hamiltonien) :

$$\vec{P} = \sum_j m_j \vec{v}_j \qquad \vec{L} = \sum_j m_j \vec{r}_j \wedge \vec{v}_j$$

$$H = \sum_j \frac{1}{2} m_j \vec{v}_j^{\,2} - \sum_{j\,k<j} \sum \frac{Gm_j m_k}{|\vec{r}_j - \vec{r}_k|}^{-1}$$

Si on calcule la dernière somme pour $k \neq j$ au lieu de $k < j$ la somme est doublée ; si on s'arrange

pour que le terme pour lequel k = j s'annule sans incident on somme pour tous les k en écrivant

$$H = \sum_j \frac{1}{2} m_j \vec{v}_j^{\,2} - \sum_j \frac{G}{2} m_j \sum_k m_k \left[(\vec{r}_j - \vec{r}_k)^2 \right]^{1/2} \left[\max\left[\varepsilon, (\vec{r}_j - \vec{r}_k)^2 \right] \right]^{-1}$$

La fonction suivante calcule ces grandeurs et cherche leurs plus grandes variations comparées aux modules de la quantité de mouvement, du moment cinétique et à l'énergie cinétique maximale

```
    ∇  z←CsvNcor;⎕io;d;H;L;P;T;mv;cs;m;F;X
[1]    ⎕io←0
[2]    SimNcor
[3]    mv←m×[1]v
[4]    P←+/[1]mv ⋄ P←(+/(((⌈≠P)-L≠P)⋆2)÷+/P[0;]⋆2)⋆0.5
[5]    L←+/[1]-/r[;; 3 2 ⍴ 1 2 2 0 0 1]×mv[;; 3 2 ⍴ 2 1 0]
[6]    L←(+/(((⌈≠L)-L≠L)⋆2)÷+/L[0;]⋆2)⋆0.5
[7]    d←+/d×d←⊃[0 3]d°.-d←⊂[0 2]r
[8]    T←0.5×+/+/mv×v
[9]    H←T-⊃(⊃((d⋆0.5)÷1E¯205⌈d)+.×m)+.×m×0.5×6.6732E¯11
[10]   H←((⌈/H)-L/H)÷⌈/T
[11]   z← P L H
    ∇
```

Vérifions la constance de ces grandeurs :

```
      CsvNcor
6.7475E¯15 2.9987E¯8 5.6818E¯8
```

Précision contrôlée

Comme on l'a dit dans l'introduction, la méthode de Runge-Kutta donne des résultats faux si le pas d'intégration est trop grand.

Considérons par exemple l'oscillateur de relaxation qui obéit aux équations

$u = x + y$ $\qquad s = \min(1, \max(-1, c\,u))$

$x' = a(s - x)$ $\qquad y' = -\,b\,u$

où "x" et "y" sont les deux variables d'état. On peut décrire ces équations par la fonction

```
    ∇  Detat←EqdRlax etat;u
[1]    Detat←etat
[2]    arr←1E6≤|etat
[3]    Dt 1
[4]    u←x+y
[5]    Dx a×(1L¯1⌈c×u)-x
[6]    Dy u×-b
    ∇
```

où on a prévu une variable d'arrêt "arr". On prépare l'intégration par la fonction qui choisit a, b, c

```
    ∇  IniRlax;etat;Detat
[1]    1 VarEtat 't x y'
[2]    a←100 ⋄ b←1 ⋄ c←5
    ∇
```

Essayons

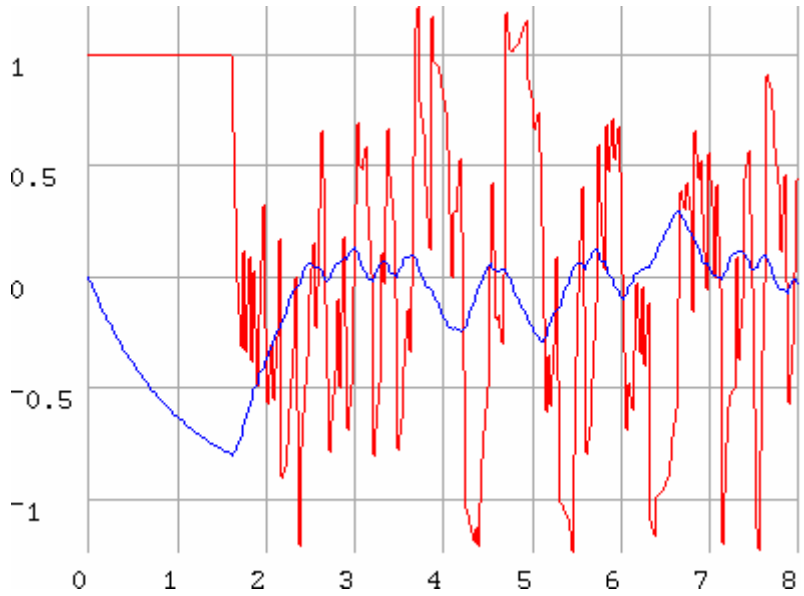
```
    ρetat← 8 128 'EqdRlax' 'arr' RunKut 1 3 ⍴ 0 1 0
32 3
    etat[31;]
1.9375 ¯1.1406E7 ¯1.1521E5
```

L'intégration diverge et c'est pire avec un pas d'intégration plus grand.

Essayons avec un pas de calcul 2 fois plus petit.

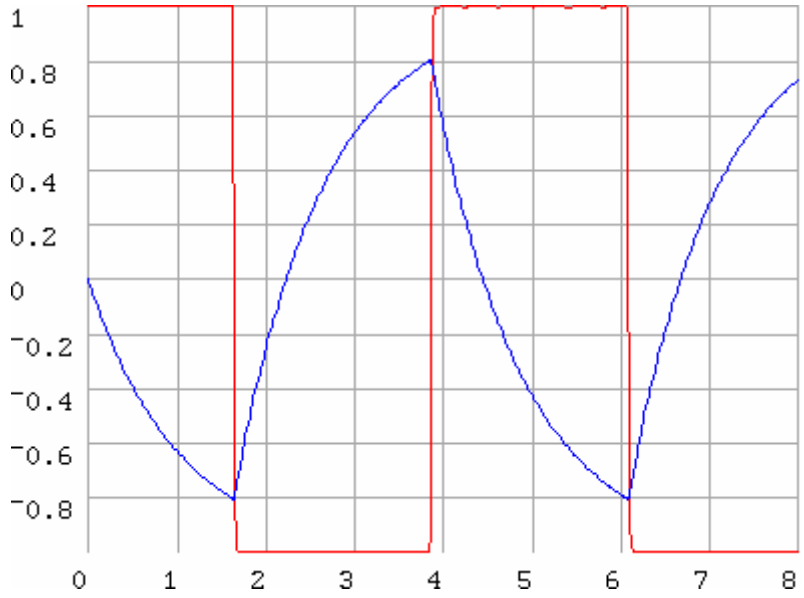
```
    ρetat← 8 256 'EqdRlax' 'arr' RunKut 1 3 ⍴ 0 1 0
257 3
```

L'intégration ne s'est pas arrêtée ; mais c'est le chaos :



Une fonction "RunKutCtl" permet de contrôler la précision de l'intégration, évitant ainsi ces difficultés :

```
⍝         nécessaires
⍝           |     |
    ρetat← (8 256) 'EqdRlax' 'arr' RunKutCtl 1 3 ρ 0 1 0
399 3
```



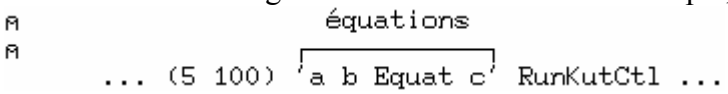
résultat correct.

Si on complète l'argument gauche de cet exemple on a

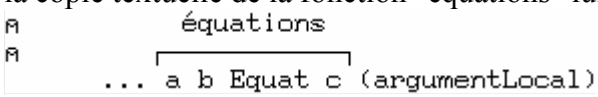
```
⍝              tndt      prec  équations arrêt
⍝           ┌────────┐ ┌─────┐ ┌────────┐ ┌─────┐
    etat← (8 256 1024) 1E¯3   'EqdRlax' 'arr' RunKutCtl 1 3 ρ 0 1 0
⍝                  └──────── précision
⍝                  └──────── nombre max de subdivisions
⍝                  └──────── nombre min d'intervalles
```

Si on l'appelle sans argument gauche RunKutCtl indique sa syntaxe et les valeurs par défaut de cet argument. Il peut y avoir deux sous-arguments gauches numériques : les étapes de l'intégration (tndt) et la précision (prec). La parenthèse évite les ambiguïtés. La structure de prec doit être compatible avec celle d'un état du système. Le choix d'appeler RunKutCtl avec une matrice ou un vecteur pour argument droit obéit aux mêmes règles que pour RunKut. La programmation de

RunKutCtl est analogue à celle de RunKut. Par exemple, si on appelle RunKutCtl par



la copie textuelle de la fonction "équations" lui fait solliciter les équations ainsi



Pour estimer l'erreur RunKutCtl rassemble les trois états de deux pas consécutifs égaux

x(t) x(t+h) x(t+2h)

les équations "Eqd" en déduisent les dérivées correspondantes

x'(t+nh) = Eqd(x(t+nh)) n∈{0, 1, 2}

RunKutCtl y applique la formule de Simpson et calcule la différence avec la variation de l'état sur ces deux pas

$$x(t+2h) - x(t) - \frac{h}{3}\left[x'(t) + 4\,x'(t+h) + x'(t+2h)\right]$$

Si la valeur absolue de cette erreur estimée est supérieure à la précision demandée, RunKutCtl rejette ces deux pas et recommence à x(t) avec un pas h/2. Si la valeur absolue de cette erreur estimée est inférieure à 1/16 de la précision demandée, RunKutCtl poursuit à x(t+2h) avec un pas 2h. Sinon RunKutCtl poursuit à x(t+2h) avec le même pas h.

Mieux vaut utiliser RunKut quand on est sûr de la précision, car elle est plus rapide que RunKutCtl.

Applications diverses

On peut utiliser ces fonctions pour intégrer tous les systèmes différentiels en mathématique, mécanique, physique, électronique, automatique, thermique, chimie,... On dispose du code source, il n'y a donc pas de boîte noire.

Les fonctions

On peut trouver les fonctions

RunKut, RunKutCtl, VarEtat, IndVec, ValVec

dans le fichier APL*PLUS

EQUAT.SF

La composante 1 est une table des matières, les autres composantes sont les ⎕vr des fonctions de ce fichier.

Les codes des fonctions IndVec et ValVec, appelées par VarEtat, sont listés dans l'article

TABLEAUX DANS UN VECTEUR UNIQUE

```
    ∇  δ_r VarEtat δ_x;δ_o
[1]    δ_o← 'Detat' 'etat' 'varEtat' 'objEtat'
[2]    δ_x←IndVec '1 ',,⍕δ_x
[3]    varEtat←(⎕io+1)⊃δ_x
[4]    δ_o←δ_o,⊃,¨/2↑δ_x
[5]    Detat←etat←(∈(δ_rρ1),¯1↑δ_x)ρ0
[6]    δ_o←δ_o,('. ',(δ_rρ';'),(⎕io+1)⊃δ_o)   ValVec 2↑δ_x
[7]    δ_o←δ_o,('D ',(δ_rρ';'),(⎕io⊃δ_o),'←') ValVec 2↑δ_x
[8]    objEtat←δ_o
    ∇
```

```
    ∇  δ_z←δ_m RunKut δ_x;δ_Int;δ_g;δ_h;δ_i;δ_k;δ_u
[1]    δ_i←⊂'EqaDif'
[2]   ⍝∇ x(... t0 ... tn) ← (tn-t0){ n m} {'Eq'} {'arr'} RunKut x(
      ... t0)
[3]   ⍝∇ Eq : équations  x'(t) ← Eq(x(t))  ; % par défaut
[4]   ⍝∇ n   ≡ nb de pas d'intégration     ; 1 par défaut
[5]   ⍝∇ m   ≡ nb de subdivisions cachées  ; 1 par défaut
[6]   ⍝∇ arr ≡ arrêter l'intégration
[7]   ⍝∇ tableaux gigognes si 2>ρρx(... t0)
[8]    δ_z←(δ_z[;⎕io]='⍝')≠δ_z←⎕cr 'RunKut' ◇ →(⎕nc 'δ_m')ρδ_A
[9]    δ_z←'⍝', 0 2 ↓(δ_z[;⎕io+1]='∇')≠δ_z ◇ (('%'=∈δ_z)/∈δ_z)←δ_i
[10]   δ_z←⊃∈⍠⊂[⎕io+1]δ_z ◇ δ_z←(⍵∨\∨≠' '≠⍵δ_z)/δ_z ◇ →0
[11]  δ_A:δ_k←0≠↑⍠0ρ⍠δ_m←∈⍠,δ_m ◇ δ_h←∈(~δ_k)/δ_m ◇ (δ_u δ_k)←2ρ(∈⍠
      δ_k/δ_m),⊂''
[12]   δ_u←↑(δ_u^.=' ')↓ δ_u δ_i ◇ δ_z← 0 2 ↓(δ_z[;⎕io+1]='i')≠δ_z
[13]   →(δ_k^.=' ')↓δ_B ◇ δ_z←(δ_z^.≠'#')≠δ_z
[14]  δ_B:(('#'≡⍠,δ_z)/,δ_z)←⊂δ_k ◇ (('⍵'≡⍠,δ_z)/,δ_z)←⊂δ_u
[15]   δ_z←⎕fx⊃∈⍠⊂[⎕io+1]δ_z
[16]   δ_h←3↑δ_h,(ρδ_h)↓ 1 1 1 ◇ →(δ_g←2>ρρδ_z←δ_x)ρδ_C ◇ δ_x←⊂[¯1↓⍳ρρ
      δ_x]δ_x
[17]  δ_C:δ_x←,δ_x++/(ρ⍠δ_x)ρ⍠0 ◇ →(1↑ρδ_z←⊃[⎕io],⍠δ_x)↓0
[18]   δ_x←(¯1,¯1↑ρδ_z)↑δ_z ◇ →δ_g↓δ_D ◇ δ_x←⊂⍠δ_x
[19]  δ_D:δ_i←↑ρδ_z ◇ δ_m←0⌈⍳1↓δ_h ◇ δ_x←((δ_g↓1),¯1↑1,ρδ_x)ρδ_x
[20]   →(¯1↑ρδ_z←(((↑ρδ_z)+↑δ_m),¯1↑ρδ_x)ρδ_z)↓δ_F ◇ δ_h←δ_h[⎕io]÷×/
      δ_m,2
[21]   δ_k←0,(1↓δ_m)/(δ_i↓⍳↑ρδ_z)-⎕io
[22]   →δ_g↓δ_E ◇ δ_z←⊂⍠δ_z
[23]  δ_E:δ_Int
[24]  δ_F:→δ_g↓0 ◇ δ_z←⊂[⎕io]⊃δ_z
[25]  ⍝i δ_Int
[26]  ⍝i δ_i←0 ◇ →δ_W
[27]  ⍝iδ_U:δ_u←δ_h×⍵ δ_x
[28]  ⍝i →(∨/0<∈⍝)↓δ_V ◇ δ_z←((1+↑(δ_i-1)↓δ_k),¯1↑ρδ_z)ρδ_z ◇ →0
[29]  ⍝iδ_V:δ_u←δ_u+2×δ_m←δ_h×⍵(δ_x+δ_u)
[30]  ⍝i δ_u←δ_u+δ_m←2×δ_h×⍵(δ_x+δ_m)
[31]  ⍝i δ_u←δ_u+δ_h×⍵(δ_x+δ_m)
[32]  ⍝i δ_x←δ_x+δ_u÷3
[33]  ⍝i δ_z[⎕io+δ_k[⎕io+δ_i];]←δ_x
[34]  ⍝iδ_W:→((ρδ_k)>δ_i←δ_i+1)ρδ_U
    ∇
```

```
    ∇  δ_z←δ_a RunKutCtl δ_x;δ_Int;δ_g;δ_kM;δ_nh;δ_h2;δ_er;δ_h;δ_k;
      δ_ke;δ_d;δ_b;δ_e;δ_i;δ_u;δ_y
[1]    δ_y← 'EqaDif' '2⋆10' '1E¯3'
[2]   ⍝∇ x(...t0...tn) ← ((tn-t0){ n m}) {prec} {'Eq'} {'arr'}
      RunKutCtl x(...t0)
[3]   ⍝∇ Eq : équations  x'(t) ← Eq(x(t))        ; % par défaut
[4]   ⍝∇ n    ≡ nombre min de pas d'intégration  ; 1 par défaut
[5]   ⍝∇ m    ≡ nombre max de subdivisions       ; % par défaut
[6]   ⍝∇ prec ≡ précision                       ; % par défaut
[7]   ⍝∇ arr  ≡ arrêter l'intégration
[8]   ⍝∇ tableaux gigognes si 2>ρρx(...t0)
[9]    δ_d←(δ_d[;⎕io]='⍝')≠δ_d←⎕cr 'RunKutCtl' ◇ →(⎕nc 'δ_a')ρδ_A
[10]   δ_d←'⍝', 0 2 ↓(δ_d[;⎕io+1]='∇')≠δ_d ◇ (('%'=∈δ_d)/∈δ_d)←δ_y
[11]   δ_d←⊃∈⍠⊂[⎕io+1]δ_d ◇ δ_z←(⍵∨\∨≠' '≠⍵δ_d)/δ_d ◇ →0
[12]  δ_A:δ_b←' '=↑⍠0ρ⍠δ_a←,δ_a ◇ (δ_u δ_k)←,⍠2ρ(δ_b/δ_a),⊂''
[13]   δ_u←↑(δ_u^.=' ')↓ δ_u (⎕io⊃δ_y)
[14]   δ_d← 0 2 ↓(δ_d[;⎕io+1]='i')≠δ_d
[15]   →(δ_k^.=' ')↓δ_B ◇ δ_d←(δ_d^.≠'#')≠δ_d
[16]  δ_B:(('#'≡⍠,δ_d)/,δ_d)←⊂δ_k ◇ (('⍵'≡⍠,δ_d)/,δ_d)←⊂δ_u
[17]   δ_d←⎕fx⊃∈⍠⊂[⎕io+1]δ_d
```

```
[18]   δ_u←(~δ_b)/δ_a
[19]   δ_e←∈1↑δ_u ⋄ δ_er←,↑(1↓δ_u),⊂0 ⋄ δ_er←δ_er,(ρδ_er)↓±(⎕io+2)⊃δ_y
[20]   (δ_h2 δ_nh δ_kM)←δ_e,(ρδ_e)↓ 0 1 ,±(⎕io+1)⊃δ_y ⋄ δ_nh←0.5⌈
      δ_nh÷2
[21]   δ_h←δ_h2÷δ_nh ⋄ δ_kM←÷δ_kM
[22]   δ_er←(⊂0⌈δ_er)÷ 2 32
[23]   →(δ_g←2>ρρδ_z←δ_x)ρδ_C ⋄ δ_x←⊂[¯1↓⍳ρρδ_x]δ_x
[24]  δ_C:δ_x←,δ_x++/(ρ¨δ_x)ρ¨0 ⋄ →(1↑ρδ_z←⊃[⎕io],¨δ_x)↓0
[25]   δ_x←(¯1,¯1↑ρδ_z)↑δ_z ⋄ →δ_g↓δ_D ⋄ δ_x←⊂¨δ_x ⋄ δ_z←⊂¨δ_z
[26]  δ_D:δ_x←((δ_g↓1),¯1↑1,ρδ_x)ρδ_x ⋄ δ_i←¯2+↑ρδ_z ⋄ δ_ke←1
[27]   δ_Int δ_d←δ_e←δ_y←0 ⋄ →δ_F
[28]  δ_E:→(δ_kM>δ_ke←δ_k×0.5)/δ_I ⋄ δ_x←↑δ_y ⋄ δ_u←↑δ_d
[29]  δ_F:δ_d←⊂δ_u ⋄ δ_y←⊂δ_x ⋄ δ_h2←δ_h×δ_k←δ_ke-δ_ke|δ_nh
[30]   δ_Int δ_h2×0.5
[31]   δ_Int δ_h2×0.5
[32]   →(∨/¨∈¨δ_er<⊂⍳,(⊃-/ 1 0 1 /δ_y)+δ_h2×(⊃+/ 1 4 1 ×δ_d)÷6)/
      δ_E δ_G
[33]   δ_ke←1⌊2×δ_k
[34]  δ_G:→((¯1+1↑ρδ_z)>δ_i←δ_i+2)ρδ_H ⋄ δ_z←δ_z;(1000,¯1↑ρδ_x)ρδ_z
[35]  δ_H:δ_z[δ_i+⍳2;]←⊃,¨1↓δ_y
[36]   →(δ_e<0<δ_nh←δ_nh-δ_k)ρδ_F
[37]  δ_I:δ_z←((1⌈δ_i+2-δ_e),¯1↑ρδ_z)ρδ_z
[38]   →δ_g↓0 ⋄ δ_z←⊂[⎕io]⊃δ_z
[39]  ⍝i δ_Int δ_h
[40]  ⍝i →(0≠δ_h)↓δ_U
[41]  ⍝i δ_u←δ_h×δ_u
[42]  ⍝i δ_u←δ_u+2×δ_a←δ_h×⍵(δ_x+δ_u×0.5)
[43]  ⍝i δ_u←δ_u+2×δ_b←δ_h×⍵(δ_x+δ_a×0.5)
[44]  ⍝i δ_u←δ_u+δ_h×⍵(δ_x+δ_b)
[45]  ⍝i δ_x←δ_x+δ_u÷6
[46]  ⍝iδ_U:δ_u←⍵ δ_x
[47]  ⍝i δ_e←∨/0<∈#
[48]  ⍝i δ_d←δ_d,⊂δ_u
[49]  ⍝i δ_y←δ_y,⊂δ_x
    ∇
```