Package DYNODE pour Python

1. Introduction

DYNODE est un Package d'interface avec des solveurs d'équations différentielles ordinaires. Il comporte des équations prédéfinies de systèmes dynamiques physiques. Il permet également de définir les équations différentielles par une fonction Python.

Les intersections des trajectoires avec des surfaces de l'espace des phases peuvent être obtenues pendant l'intégration, ce qui permet d'obtenir les sections de Poincaré.

DYNODE comporte 3 modules :

- `cvode` : interface pour le solveur CVODE de SUNDIALS.
- `verlet` : solveur pour systèmes hamiltoniens (méthode de Stormer-verlet).
- `plot` : programme d'affichage 3D des trajectoires (OpenGL), permettant aussi d'effectuer des animations d'objets.

2. Compilation

La compilation nécessite les bibliothèques suivantes :

- Python 2.6 ou 2.7
- Numpy (1.3 ou ultérieure)
- Sundials 2.4
- freeglut
- GLEW
- win32 pthread

La bibliothèque pthread est toujours présente sous Linux (threads POSIX)

3. Installation

4. Interface pour le solveur CVODE

4.a. Description

CVODE est un solveur d'équations différentielles ordinaires faisant partie de la bibliothèque SUNDIALS. Il utilise une méthode à pas multiples à ordre variable explicite (Adams) ou implicite (BDF).

L'interface est fournie par la classe `CVOde` définie dans le module `dynode.main`.

```
CVOde.CVOde(equation,method,iteration)
```

Constructeur : ouverture d'un solveur, sélection d'une équation différentielle et de la méthode numérique.

- `equation (integer)` : identifiant du système d'équations différentielles ou 0 pour un système définie par une fonction Python.

- ▷ `method (integer)` : méthode d'intégration (OdeAdams ou OdeBDF).
- ▷ `iteration (integer)` : méthode de résolution pour la méthode BDF (OdeFunctional ou OdeNewton).

Remarque : il est possibles de créer plusieurs instances de CVOde pour effectuer des calculs en parallèle (maximum 100) sur les processeurs multicœurs.

```
CVOde.close()
```

Fermeture du solveur (libération de l'espace mémoire utilisé)

```
CVOde.set_cst(cst)
```

Affecter des valeurs aux constantes du système d'équations différentielles.

- ▷ `cst (float list)` : liste des constantes.

```
CVOde.init(t0,y0,reltol,abstol)
```

Choix de la condition initiale et des tolérances.

- ▷ `t0 (float)` : temps initial.
- ▷ `y0 (float list)` : liste des valeurs initiales des variables.
- ▷ `reltol (float)` : tolérance relative.
- ▷ `abstol (float list)` : liste des tolérances absolues (une par variable ou liste d'une seule valeur).

```
data=CVOde.solve(dt,T)
```

Intégration et récupération des valeurs sur une durée T (à partir du dernier point calculé), avec un intervalle de temps dt. La fonction retourne à la fin du calcul.

- ▷ `dt (float)` : intervalle de temps entre deux instants mémorisés.
- ▷ `T (float)` : durée totale.
- ▷ `data (ndarray)` : tableau des valeurs mémorisées (tableau numpy ndarray), dont les lignes sont dans l'ordre : les instants t, les N variables (N+1 lignes).

```
CVOde.start_solve(dt,T)
```

Démarrage de l'intégration. La fonction s'exécute de manière asynchrone, i.e. retourne après lancement du calcul sur un thread secondaire.

- ▷ `dt (float)` : intervalle de temps entre deux instants mémorisés.
- ▷ `T (float)` : durée totale.

Remarque : cette fonction permet de lancer des calculs pour plusieurs instances de CVOde en parallèle. Ceci permet de maximiser l'utilisation des processeurs multicœurs.

```
CVOde.stop_solve()
```

Arrêt du calcul lancé par `start_solve`.

```
CVOde.wait()
```

Attendre la fin du calcul lancé par `start_solve` (indispendable avant d'en lancer un nouveau).

```
data=CVOde.get_data(first)
```

Obtention des données du calcul en cours, ou du dernier calcul.

- ▷ `first (integer)` : indice du premier point à récupérer (0 pour obtenir toutes les données).
- ▷ `data (ndarray)` : tableau des valeurs mémorisées (tableau numpy ndarray), dont les lignes sont dans l'ordre : les instants t, les N variables (N+1 lignes).

```
CVOde.root_find(set,zerocross)
```

Activer ou désactiver la recherche des racines.

- ▷ `set (integer)` : 1 pour activer, 0 pour désactiver
- ▷ `zerocross (integer)` : 1 pour obtenir les racines par valeur croissante, -1 par valeur décroissante, 0 pour obtenir les deux.

```
data=CVOde.get_roots(index,first)
```

Obtenir les points correspondant aux racines.

- ▷ `index (integer)` : indice de l'équation.
- ▷ `first (integer)` : indice du premier point à récupérer (0 pour obtenir toutes les données).
- ▷ `data (ndarray)` : tableau des racines (tableau numpy ndarray), dont les lignes sont dans l'ordre : les instants t, les valeurs des N variables, et le sens de franchissement (-1 ou +1), soit au total N+2 lignes.

```
CVOde.set_eqn(nvar,eqn)
```

Définir le système différentiel par une fonction Python, lorsque l'argument `equation` de `CVOde(equation,method,iteration)` est égal à 0.

- ▷ `nvar (integer)` : nombre de variables.

▷ `eqn (function)` : système différentiel défini par une fonction de la forme suivante (exemple du pendule) :

```
def equation(t,y):
    ydot=[0,0]
    ydot[0] = y[1]
    ydot[1] = -39.4784176044*math.sin(y[0])
    return ydot
```

```
CVOde.set_jac(jac)
```

Définir le jacobian du système différentiel par une fonction Python, à utiliser conjointement avec la fonction `set_eqn()`.

▷ `jac (function)` : jacobian défini par une fonction de la forme suivante :

```
def jacobian(t,y):
    J=[[0,0],[0,0]]
    J[0][0] = 0.0
    J[0][1] = 1.0
    J[1][0] = -39.4784176044*math.cos(y[0])
    J[1][1] = 0.0
    return J
```

```
CVOde.set_rootfn(neq,rootfn)
```

Définir les équations algébriques à résoudre par une fonction Python, à utiliser conjointement avec la fonction `set_eqn()`.

▷ `neq (integer)` : nombre d'équations algébriques.

▷ `rootfn (function)` : fonction définissant les équations, de la forme suivante :

```
def rootfn(t,y):
    g=[y[0],y[1]]
    return g
```

```
CVOde.set_maxsteps(maxsteps)
```

Modifier le nombre de pas maximal par séquence de calcul (i.e. pour chaque intervalle de temps `dt` défini dans `solve` ou dans `start_solve`. La valeur par défaut (500) peut être insuffisante pour les systèmes raides.

▷ `maxsteps (integer)` : nombre de pas maximal. Lorsque le nombre maximal est atteint avant la fin du calcul, une erreur est générée.

4.b. Exemple

L'exemple suivant montre l'utilisation de l'interface dans le cas d'un système prédéfini (modèle de Lorenz).

On commence par importer les différents modules puis on ouvre un solveur en définissant le système à résoudre et la méthode numérique :

```
import dynode.main as dyn
import numpy
from pylab import *
s=dyn.CVOde(dyn.OdeLorenz,dyn.OdeAdams,dyn.OdeFunctional)
```

On affecte les constantes. Les trois premières interviennent dans le système différentiel alors que la 4ième est la côte z du plan de coupe (section de Poincaré).

```
s.set_cst([10.0,28.0,8.0/3,27.0])
```

On active la recherche des racines (coupe de Poincaré z=27) :

```
s.root_find(1,dyn.OdeZeroCrossUp)
```

On fixe les conditions initiales et les tolérances :

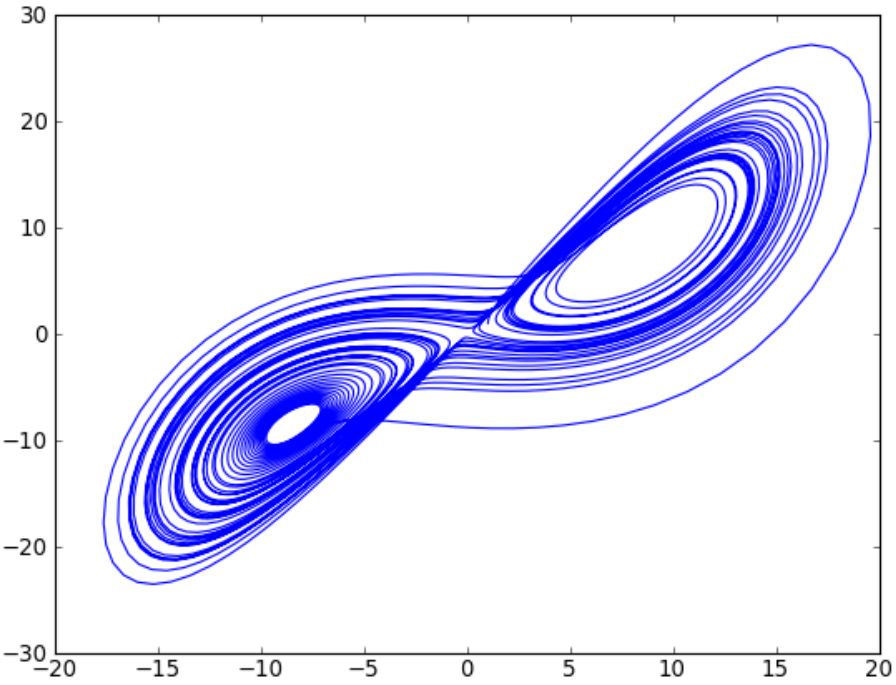
```
s.init(0.0,[1.0,1.0,1.0],1e-6,[1e-7])
```

Enfin une intégration jusqu'à l'instant t=100 :

```
data1=s.solve(0.01,50.0)
```

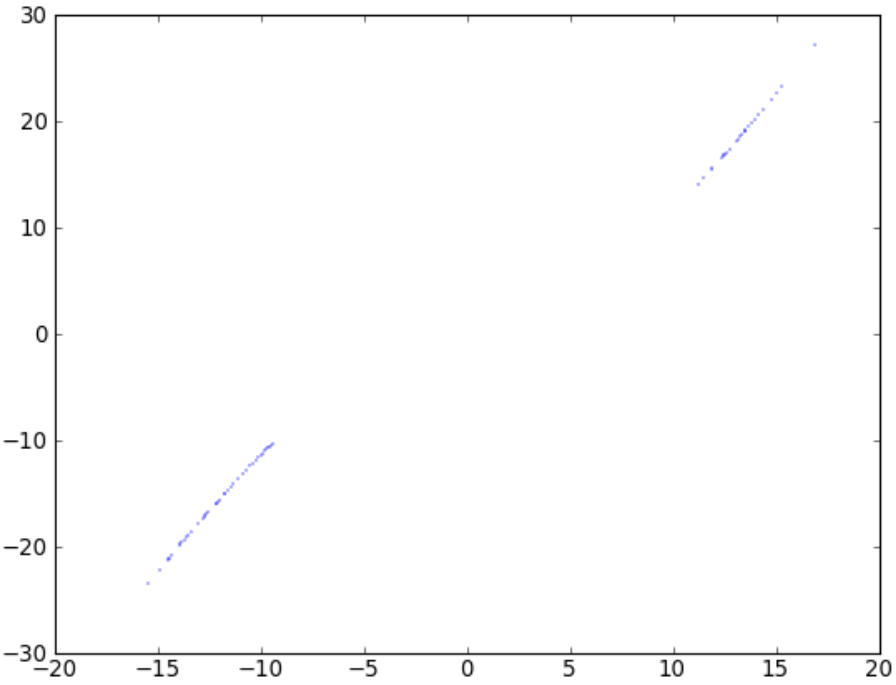
On trace la trajectoire avec pylab (matplotlib) :

```
t=data1[0]
x=data1[1]
y=data1[2]
z=data1[3]
figure(1)
plot(x,y)
```



On récupère les points vérifiant la condition z=27 :

```
data2=s.get_roots(0,0)
t=data2[0]
x=data2[1]
y=data2[2]
z=data2[3]
figure(2)
plot(x,y,marker=".",linestyle="",markersize=1)
```




```
s.close()
```

5. Module graphique 3D

5.a. Description

Le module graphique `plot` est accessible par différentes classes définies dans `dynode.main`.

La classe `Plot` (une seule instance) contrôle l'ouverture du module et le lancement de la boucle de rendu.

```
Plot.Plot()
```

Constructeur : ouverture du module. Une seule instance peut être créée.

```
Plot.show(refresh=0)
```

Lancement de la boucle de rendu des fenêtres graphiques. Cette fonction est asynchrone, i.e. retourne dès que la boucle est lancée.

▷ `refresh (float)` : temps de mise à jour de l'affichage en millisecondes (pour les animations et tracés temps-réel). Valeur 0 pour les tracés statiques.

```
Plot.wait()
```

Attente de la fin de la boucle de rendu, qui se fait lorsque l'utilisateur ferme l'une des fenêtres ou appuie sur la touche q.

```
Plot.close()
```

Fermeture du module.

La classe `Figure` permet de définir une figure, c'est-à-dire une fenêtre graphique. Les figures apparaissent lorsque la fonction `Plot.show` est appelée. Le nombre de figures est illimitée.

```
Figure.Figure()
```

Constructeur : nouvelle figure.

```
Figure.viewbox(xmin,xmax,ymin,ymax,zmin,zmax)
```

Définir la zone de l'espace rendue dans la fenêtre. Par défaut, la projection est orthoscopique.

- ▷ `xmin (float)` :
- ▷ `xmax (float)` :
- ▷ `ymin (float)` :
- ▷ `ymax (float)` :
- ▷ `zmin (float)` :
- ▷ `zmax (float)` :

Les fonctions suivantes de type `set_xxx` doivent être appelées avant le lancement de la boucle de rendu.

```
Figure.set_camera(z)
```

Acivation de la projection perspective avec positionnement de la caméra.

- ▷ `z (float)` : distance de la caméra par rapport au point origine. Plus la caméra est proche, plus la perspective est prononcée.

```
Figure.set_lighting()
```

Activation de l'éclairage de la scène.

```
Figure.set_ambient_light(rgb)
```

Définir la couleur de la lumière ambiante.

- ▷ `rgb (float list)` : couleur sous forme d'une liste de trois nombres [r,g,b] compris entre 0.0 et 1.0.

```
Figure.set_diffuse_light(rgb)
```

Définir la couleur de la lumière diffusée.

▷ `rgb (float list)` : couleur sous forme d'une liste de trois nombres [r,g,b] compris entre 0.0 et 1.0.

```
Figure.set_antialias()
```

Activation de l'anticrénelage (antialiasing).

```
Figure.set_legends(x,y,z)
```

Attribuer des légendes aux axes X,Y,Z. Par défauts les légendes sont x,y, et z.

▷ `x (string)` : légende de l'axe X.

▷ `y (string)` : légende de l'axe Y.

▷ `z (string)` : légende de l'axe Z.

Les 3 fonctions suivantes permettent d'obtenir une copie de la figure sous forme d'une image pixmap générée lorsque l'utilisateur appuie sur la touche s. Une seule image peut être générée par figure. Si l'utilisateur n'utilise pas la touche s, l'image est générée à partir du dernier état de la figure (lorsque l'utilisateur appuie sur la touche q). Ces fonctions doivent être appelées après la fermeture de la boucle de rendu, i.e. après l'appel de `Plot.wait`.

```
size=Figure.get_pixels_size()
```

Obtenir la taille de l'image.

▷ `size (float list)` : taille de l'image sous la forme [width,height].

```
data=Figure.get_pixels()
```

Obtenir l'image sous forme d'une liste de valeurs RGB

▷ `data (integer list)` : liste des valeurs RGB des pixels (une valeur par pixel)

```
img=Figure.get_image()
```

Fournit une instance de la classe `Image` du module PIL (si celui-ci est installé).

▷ `img (Image)` : instance de la classe Image.

L'image peut être ensuite enregistrée sous forme PNG par :

```
img.save("figure.png","PNG")
```

Les classes suivantes permettent de créer des objets graphiques dans une figure. Les instances doivent être créées avant l'exécution de `Plot.show()`.

La classe `Grid` représente une grille de repérage, perpendiculaire à un des axes X,Y, ou Z.

```
Grid.Grid(figure,xmin,xmax,nx,ymin,ymax,ny,z,orient=PlotOrientZ,solid=0,ticks=0,r
```

Construction d'une grille attachée à une figure.

- ▷ `figure (Figure)` : instance de la classe Figure.
- ▷ `xmin, xmax (float)` : bornes de l'axe x de la grille.
- ▷ `nx (integer)` : nombre de traits sur l'axe x.
- ▷ `ymin, ymax (float)` : bornes de l'axe y de la grille.
- ▷ `ny (integer)` : nombre de traits sur l'axe y.
- ▷ `z (float)` : position de la grille sur l'axe perpendiculaire à la grille (définie ci-après).
- ▷ `orient (integer)` : orientation de la grille, définie par une des constantes suivantes : `PlotOrientX, PlotOrientMX` (grille perpendiculaire à l'axe X, orientée suivants X+ ou X-), `PlotOrientY, PlotOrientMY` et `PlotOrientZ, PlotOrientMZ`.
- ▷ `solid (integer)` : 0 pour une grille constituées de traits, 1 pour obtenir un damier.
- ▷ `ticks (integer)` : 1 pour tracer les graduations sur les axes, 0 sinon.
- ▷ `rgb (float list` : couleur sous forme d'une liste de trois nombres [r,g,b] compris entre 0.0 et 1.0.

La classe `Plot3d` permet de tracer des points dans l'espace, éventuellement reliés par des traits pour former une courbe.

```
Plot3d.Plot3d(figure,x,y,z,rgb=[1,0,0],style=PlotStylePoints,width=1.0)
```

Constructeur.

- ▷ `figure (Figure)` : instance de la classe Figure.
- ▷ `x,y,z (ndarray)` : listes des coordonnées des points, sous forme de numpy.ndarray de type double (numpy.float64).
- ▷ `rgb (float list` : couleur sous forme d'une liste de trois nombres [r,g,b] compris entre 0.0 et 1.0.
- ▷ `style (integer)` : une des constantes `PlotStyleLines` et `PlotStylePoints`.
- ▷ `width (float)` : épaisseur du trait ou largeur des points.

```
Plot3d.clear()
```

Effacer les points (ou la courbe). L'espace mémoire correspondant est libéré.

La classe `Plot2d` génère des points dans le plan $z = 0$.

```
Plot2d.Plot2d(figure,x,y,rgb=[1,0,0],style=PlotStylePoints,width=1.0)
```

Constructeur.

- ▷ `figure (Figure)` : instance de la classe Figure.
- ▷ `x,y (ndarray)` : listes des coordonnées des points.
- ▷ `rgb (float list` : couleur sous forme d'une liste de trois nombres [r,g,b] compris entre 0.0 et 1.0.
- ▷ `style (integer)` : une des constantes `PlotStyleLines` et `PlotStylePoints`.
- ▷ `width (float)` : épaisseur du trait ou largeur des points.

```
Plot2d.clear()
```

Effacer les points (ou la courbe). L'espace mémoire correspondant est libéré. Cette fonction peut être appelée pendant l'exécution de la boucle de rendu.

```
Surface.Surface(figure,xrg,yrg,zarray,solid,rgb)
```

Surface de la forme z=f(x,y). Les z sont fournis sous forme d'un tableau. La surface générée est statique. Le chargement des grandes surfaces (plus de 100000 points) comporte un délai de quelques secondes.

- ▷ `figure (Figure)` : instance de la classe Figure.
- ▷ `xrg (float list)` : domaine de x sous la forme [xmin,xmax].
- ▷ `yrg (float list)` : domaine de y sous la forme [ymin,ymax].
- ▷ `zarray (ndarray)` : valeurs de z sous la forme d'un tableau numpy.ndarray de type double (numpy.float64).
- ▷ `solid (integer)` : 0 pour le tracé des arêtes seulement, 1 pour des faces opaques.
- ▷ `rgb (float list)` : couleur sous forme d'une liste de trois nombres [r,g,b] compris entre 0.0 et 1.0.

Les classes suivantes génèrent des objets graphiques qui peuvent être déplacés (pour animation) pendant la boucle de rendu.

```
Cube.Cube(figure,x,y,z,phi,theta,psi,width,solid=0,rgb=[1,0,0])
```

Constructeur d'un cube. Les attributs (width, solid et rgb) ne peuvent être modifiés après lancement de la boucle de rendu.

- ▷ `figure (Figure)` : instance de la classe Figure.
- ▷ `x,y,z (float)` : coordonnées du centre du cube.
- ▷ `phi,theta,psi (float)` : angles d'Euler d'orientation du cube.
- ▷ `solid (integer)` : 0 pour le tracé des arêtes seulement, 1 pour des faces opaques.

▷ `rgb (float list)` : couleur sous forme d'une liste de trois nombres [r,g,b] compris entre 0.0 et 1.0.

```
Cube.position(x,y,z,phi,theta,psi)
```

Positionne et oriente le cube. Cette fonction doit être appelée pendant l'exécution de la boucle de rendu, et permet d'animer le cube.

▷ `x,y,z (float)` : coordonnées du centre du cube.

▷ `phi,theta,psi (float)` : angles d'Euler d'orientation du cube.

```
Sphere.Sphere(x,y,z,r,solid=0,slices=20,stacks=20,rgb=[1,0,0])
```

Constructeur.

▷ `figure (Figure)` : instance de la classe Figure.

▷ `x,y,z (float)` : coordonnées du centre de la sphère.

▷ `r (float)` : rayon de la sphère.

▷ `slices,stacks (integer)` : nombre de cercles de longitude et de latitude.

▷ `solid (integer)` : 0 pour le tracé des arêtes seulement, 1 pour des faces opaques.

▷ `rgb (float list)` : couleur sous forme d'une liste de trois nombres [r,g,b] compris entre 0.0 et 1.0.

```
Sphere.position(x,y,z)
```

Positionne la sphère. Cette fonction doit être appelée pendant l'exécution de la boucle de rendu, et permet d'animer la sphère.

▷ `x,y,z (float)` : coordonnées du centre du cube.

```
Cylinder.Cylinder(figure,x,y,z,phi,theta,psi,h,r0,rh,solid=0,slices=20,rgb=[1,0,0
```

Constructeur d'un cylindre (ou d'un cône).

▷ `figure (Figure)` : instance de la classe Figure.

▷ `x,y,z (float)` : coordonnées du centre du disque de base.

▷ `phi,theta,psi (float)` : angles d'Euler d'orientation de l'axe du cylindre.

▷ `h (float)` : hauteur du cylindre.

▷ `r0 (float)` : rayon du disque de base.

▷ `rh (float)` : rayon du disque de hauteur h. Si ce rayon est différent de r0, la forme générée est un cône.

▷ `slices (integer)` : nombre de segments.

▷ `solid (integer)` : 0 pour le tracé des arêtes seulement, 1 pour des faces opaques.

▷ `rgb (float list)` : couleur sous forme d'une liste de trois nombres [r,g,b] compris entre 0.0 et 1.0.

```
Cylinder.position(x,y,z,phi,theta,psi)
```

Positionne et oriente le cylindre. Cette fonction doit être appelée pendant l'exécution de la boucle de rendu, et permet d'animer le cylindre.

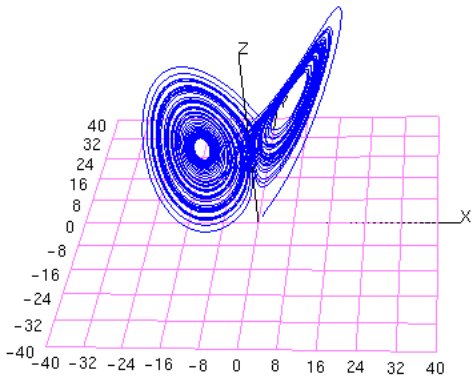
▷ `x,y,z (float)` : coordonnées du centre du cylindre.

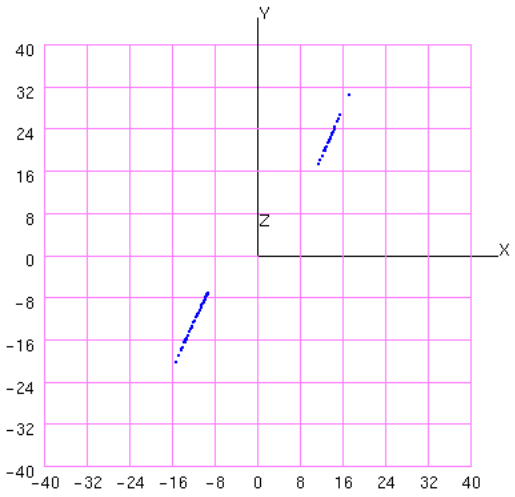
▷ `phi,theta,psi (float)` : angles d'Euler d'orientation du cylindre.

5.b. Exemple

On reprend les données calculées plus haut (système de Lorenz) pour tracer la trajectoire dans l'espace X,Y,Z et la section de Poincaré $z = 27$.

```
plot=dyn.Plot()
fig1=dyn.Figure()
fig1.set_camera(300)
fig2=dyn.Figure()
a=50.0
b=40.0
fig1.viewbox(-a,a,-a,a,-a,a)
fig2.viewbox(-a,a,-a,a,-a,a)
g1=dyn.Grid(fig1,-b,b,10,-b,b,10,0,ticks=1,rgb=[1.0,0.5,1.0])
p1=dyn.Plot3d(fig1,data1[1],data1[2],data1[3],style=dyn.PlotStyleLines,rgb=[0,0,1.0],
g2=dyn.Grid(fig2,-b,b,10,-b,b,10,0,ticks=1,rgb=[1.0,0.5,1.0])
p1=dyn.Plot3d(fig2,data2[1],data2[2],data2[3],style=dyn.PlotStylePoints,rgb=[0,0,1.0]
plot.show()
plot.wait()
path="../../../../figures/numerique/dynode/pydynode/"
img=fig1.get_image()
img.save(path+"plot3.png","PNG")
img=fig2.get_image()
img.save(path+"plot4.png","PNG")
plot.close()
```





6. Solveur de systèmes hamiltoniens

6.a. Description

Le module `verlet` est un solveur symplectique pour systèmes hamiltoniens utilisant la méthode de Stormer-verlet. Il traite les systèmes à hamiltoniens séparables, éventuellement dépendant du temps. Un champ magnétique (ou une force de Coriolis) peut être ajouté (système hamiltonien à champ magnétique).

Les sytèmes physiques concernés sont ceux dont les forces internes ne dépendent que des coordonnées, avec un champ extérieur dépendant éventuellement du temps.

L'interface avec le solveur est fournie par la classe `Verlet`, définie dans le module `dynode.main`.

```
Verlet.Verlet(equation)
```

Constructeur : ouverture d'un solveur et sélection d'une équation différentielle.

▷ `equation (integer)` : identifiant du système d'équations différentielles ou 0 pour un système définie par une fonction Python.

Remarque : il est possibles de créer plusieurs instances de la classe `Verlet` pour effectuer des calculs en parallèle (maximum 100) sur les processeurs multicœurs.

```
Verlet.set_cst(cst)
```

Affecter des valeurs aux constantes du système d'équations différentielles.

▷ `cst (float list)` : liste des constantes.

```
Verlet.init(t0,q0,p0,h)
```

Choix de la condition initiale et du pas de temps.

- ▷ `t0 (float)` : temps initial.
- ▷ `q0 (float list)` : liste des valeurs initiales des coordonnées.
- ▷ `p0 (float list)` : liste des valeurs initiales des moments.
- ▷ `h (float)` : pas de temps.

```
data=Verlet.solve(dt,T)
```

Intégration et récupération des valeurs sur une durée T (à partir du dernier point calculé), avec un intervalle de temps dt. La fonction retourne à la fin du calcul.

- ▷ `dt (float)` : intervalle de temps entre deux instants mémorisés.
- ▷ `T (float)` : durée totale.
- ▷ `data (ndarray)` : tableau des valeurs mémorisées (tableau numpy ndarray), dont les lignes sont dans l'ordre : le temps t, les coordonnées q, les moments p et l'énergie du système.

```
Verlet.start_solve(dt,T)
```

Démarrage de l'intégration. La fonction s'exécute de manière asynchrone, i.e. retourne après lancement du calcul sur un thread secondaire.

- ▷ `dt (float)` : intervalle de temps entre deux instants mémorisés.
- ▷ `T (float)` : durée totale.

Remarque : cette fonction permet de lancer des calculs pour plusieurs instances de Verlet en parallèle. Ceci permet de maximiser l'utilisation des processeurs multicœurs.

```
Verlet.stop_solve()
```

Arrêt du calcul lancé par `start_solve`.

```
Verlet.wait()
```

Attendre la fin du calcul lancé par `start_solve` (indispendable avant d'en lancer un nouveau).

```
data=Verlet.get_data(first=0)
```

Obtention des données du calcul en cours, ou du dernier calcul.

- ▷ `first (integer)` : indice du premier point à récupérer (0 pour obtenir toutes les données).
- ▷ `data (ndarray)` : tableau des valeurs mémorisées (tableau numpy ndarray), dont les lignes sont dans l'ordre : le temps t, les coordonnées q, les moments p et l'énergie du système.

```
Verlet.root_find(set,zerocross,rstol=1e-3)
```

Activer ou désactiver la recherche des racines.

▷ `set (integer)` : 1 pour activer, 0 pour désactiver

▷ `zerocross (integer)` : 1 pour obtenir les racines par valeur croissante, -1 par valeur décroissante, 0 pour obtenir les deux.

▷ `rstol (float)` : tolérance pour la recherche des racines (tolérance relative au pas de temps).

```
data=Verlet.get_roots(index,first)
```

Obtenir les points correspondant aux racines.

▷ `index (integer)` : indice de l'équation.

▷ `first (integer)` : indice du premier point à récupérer (0 pour obtenir toutes les données).

▷ `data (ndarray)` : tableau des racines (tableau numpy ndarray), dont les lignes sont dans l'ordre : le temps t, les coordonnées q, les moments p, et le sens de franchissement du zéro (-1 ou 1).

```
data=Verlet.set_eqn(nvar,dKfn,dVfn,efn)
```

Définir le système différentiel par des fonctions Python, lorsque l'argument `equation` de `Verlet(equation)` est égal à 0.

▷ `nvar (integer)` : nombre de variables, c.a.d. nombre de coordonnées q.

▷ `dKfn (function)` : fonction définissant la dérivée de l'énergie cinétique par rapport aux moments p.

▷ `dVfn (function)` : fonction définissant la dérivée de l'énergie potentielle par rapport aux coordonnées q.

▷ `efn (function)` : fonction définissant l'énergie totale du système.

Voici un exemple de système défini par des fonctions : une particule (3 coordonnées cartésiennes) est soumise à une force centrale élastique.

```
def dKinetic(t,p):
    dq = p
    return dq

def dPotential(t,q):
    dp = [0,0,0]
    dp[0] = q[0]
    dp[1] = q[1]
    dp[2] = q[2]
    return dp
```

```
def energy(t,q,p):
    return 0.5*(p[0]*p[0]+p[1]*p[1]+p[2]*p[2])+0.5*(q[0]*q[0]+q[1]*q[1]+q[2]*q[2]);
```

```
Verlet.set_bfield(typ,bfn)
```

Ajout d'un champ magnétique (ou d'une force de Coriolis), au système défini par `set_eqn`.

- ▷ `typ (integer)` : Champ uniforme (BFiedUniform) ou non uniforme (BFieldNonUniform).
- ▷ `bfn (function)` : Fonction définissant le champ. Celui-ci est défini par le vecteur vitesse angulaire Ω (vecteur cyclotronique ou vecteur vitesse angulaire du référentiel tournant).

Lorsque le champ est uniforme, il doit être colinéaire à l'axe z. Voici par exemple une fonction définissant un champ magnétique uniforme :

```
def bfield(t,x):
    return [0,0,-1]
```

```
Verlet.set_rootfn(neq,rootfn)
```

Définir les équations algébriques à résoudre par une fonction Python, à utiliser conjointement avec la fonction `set_eqn()`.

- ▷ `neq (integer)` : nombre d'équations algébriques.
- ▷ `rootfn (function)` : fonction définissant les équations, de la forme suivante :

```
def rootfn(t,q,p):
    return [q[0],q[2]]
```

6.b. Exemple

L'exemple reprend les fonctions définies ci-dessus pour définir le système (oscillateur dans un champ magnétique uniforme).

```
import dynode.main as dyn
solver=dyn.Verlet(0)
solver.set_eqn(3,dKinetic,dPotential,energy)
solver.set_bfield(dyn.BFieldUniform,bfield)
solver.set_rootfn(2,rootfn)
solver.init(0.0,[1.0,0.0,0.0],[0.0,1.0,1],0.01)
data=solver.solve(0.1,100)
t=data[0]
x=data[1]
```

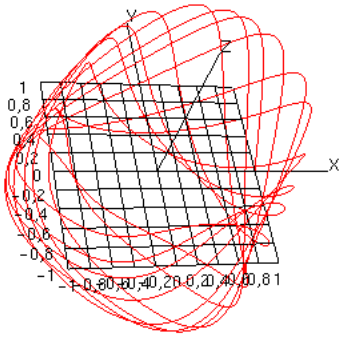
```
y=data[2]
z=data[3]
e=data[7]
```

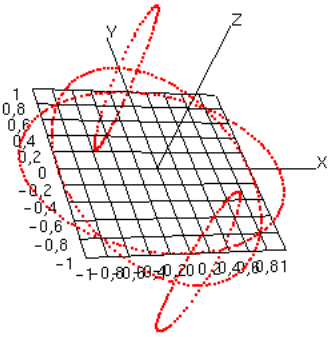
Recherche des racines sur une durée plus longue :

```
solver.root_find(1,0,1e-3)
data=solver.solve(1.0,1000)
roots1=solver.get_roots(0)
roots2=solver.get_roots(1)
solver.close()
```

Tracé de la trajectoire et des racines :

```
plot=dyn.Plot()
fig1=dyn.Figure()
fig1.set_camera(10)
fig1.viewbox(-2.0,2.0,-2.0,2.0,-2.0,2.0)
g1=dyn.Grid(fig1,-1.0,1.0,10,-1.0,1.0,10,0,ticks=1)
p1=dyn.Plot3d(fig1,x,y,z,style=dyn.PlotStyleLines)
fig2=dyn.Figure()
fig2.set_camera(10)
fig2.viewbox(-2.0,2.0,-2.0,2.0,-2.0,2.0)
g2=dyn.Grid(fig2,-1.0,1.0,10,-1.0,1.0,10,0,ticks=1)
p2=dyn.Plot3d(fig2,roots1[1],roots1[2],roots1[3],style=dyn.PlotStylePoints,width=2.0)
p3=dyn.Plot3d(fig2,roots2[1],roots2[2],roots2[3],style=dyn.PlotStylePoints,width=2.0)
plot.show()
plot.wait()
path="../../../../figures/numerique/dynode/pydynode/"
img=fig1.get_image()
img.save(path+"plot5.png","PNG")
img=fig2.get_image()
img.save(path+"plot6.png","PNG")
plot.close()
```


Y
Z
X
1
0,8
0,6
0,4
0,2
0
-0,2
-0,4
-0,6
-0,8
-1


Y
Z
X
1
0,8
0,6
0,4
0,2
0
-0,2
-0,4
-0,6
-0,8
-1