Considérons le problème de Cauchy

$$\begin{cases} y'(t) = f(t, y(t)) \quad t \in [t_0, t_0 + T[\\ y(t_0) = y_0. \end{cases} \tag{1}$$

La fonction f est à valeurs dans $\mathbb{R}$ ou $\mathbb{R}^n$ et on suppose que tout se passe bien du point de vue mathématique.

Il est bien sûr possible de programmer soit même les schémas numériques usuels (ce sera une des choses à faire en TP). L'autre solution consiste à utiliser le module `integrate` de Scipy qui fournit deux fonctions

- `odeint(func, y0, t, args=(), Dfun=None, col_deriv=0, full_output=0, ml=None, mu=None, rtol=None, atol=None, tcrit=None, h0=0.0, hmax=0.0, hmin=0.0, ixpr=0, mxstep=0, mxhnil=0, mxordn=12, mxords=5, printmessg=0)` qui fait appel à lsoda de la librairie FORTRAN odepack. D'après les explications la librairie s'adapte selon que le problème soit non raide (méthode multi-pas d'Adams) ou raide (méthode BDF, différence finie retrograde ou encore méthode de Gears).
- `ode` un « wrapper » (emballage, encapsulation, interface, bref quelque chose de perfectionnée). `ode` est une nouvelle classe et plusieurs librairies d'intégrations sont disponibles : `vode, zvode, dopri5, dop853`.

1. `odeint`

Nous nous limiterons à la syntaxe simple `odeint(func,y0,t)` où `func` est la fonction f de (1), `y0` est la donnée initiale de (1) et `t` un vecteur des valeurs de t dont le premier élément est t_0 : la routine nous retourne une approximation de la solution en les valeurs de `t`.

Remarque 1 (Attention)**.** Il faut bien respecter la syntaxe pour la fonction `func` qui est `func(y,t,...)` et pas `func(t,y,...)` comme l'écriture mathématique habituelle.

Le paramètre optionnel `Dfun` est liée au gradient (le jacobien) de f.

```
>>> def func(y,t):
>>>     return y
>>> y0=1.
>>> t0=array([0., 1.])
>>> return itg.odeint(func,y0,t0)
```

Pour tracer il suffit alors de taper

```
>>> t=linspace(0,1,100)
>>> plt.plot(t,itg.odeint(func,y0,t))
```

et on obtient la figure 1 (ou presque).

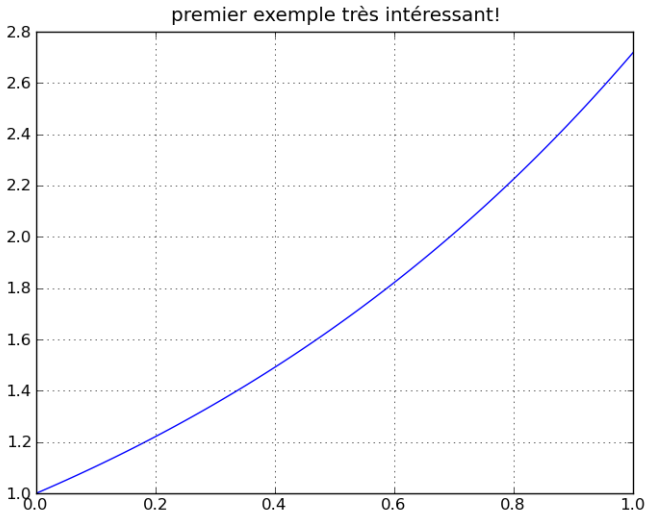

FIGURE 1. $y' = y$

1.1. **Problème raide.** Dans le cas d'une EDO raide on peut constater que la routine s'adapte en lui demandant d'afficher des messages. Reprenons l'exemple du cours, $y' = -150y + 30$, $y(0) = 1/5 + \epsilon$:

```
>>> def func(y,t):
>>> return -150.*y+30
>>> y0=1./5.+10**(-1)
>>> t0=array([0., 1., 2., 5.])
>>> return itg.odeint(func,y0,t0,printmessg=True,full_output=1)
```

```
Integration successful.
Out[79]:
(array([[ 0.3],
       [ 0.2],
       [ 0.2],
       [ 0.2]]),
 {'hu': array([  1.95218815,   1.95218815,  19.52188146]),
  'imxer': -1,
  'leniw': 21,
  'lenrw': 36,
  'message': 'Integration successful.',
  'mused': array([2, 2, 2], dtype=int32),
  'nfe': array([158, 158, 161], dtype=int32),
  'nje': array([3, 3, 4], dtype=int32),
  'nqu': array([1, 1, 1], dtype=int32),
  'nst': array([78, 78, 80], dtype=int32),
  'tcur': array([  2.52660357,   2.52660357,  24.00067317]),
  'tolsf': array([  4.94065646e-324,   4.94065646e-324,   4.94065646e-324]),
  'tsw': array([ 0.12541215,  0.12541215,  0.12541215])})
```

Le champ `mused` vaut 1 (non raide méthode d'Adams) ou 2 (raide méthode BDF). Pour comprendre les autres informations il faut taper `help itg.odeint` et connaître les détails des méthodes implémentées.

1.2. **Système.** La résolution d'un système différentiel du premier ordre se pratique de la même façon.

On considère le système de proie–prédateur de Lotka–Volterra, c'est-à-dire deux espèces animales en interaction, les prédateurs et les proies de densité de population respective $p(t)$ et $q(t)$ modélisées par le système différentielle suivant

$$\begin{cases} p'(t) = ap(t)q(t) - bp(t), \quad t \in \mathbb{R}^+ \\ q'(t) = -cp(t)q(t) + dq(t), \\ p(0) = p_0, \quad q(0) = q_0 \end{cases} \tag{2}$$

Les constantes a et b sont strictement positives et correspondent respectivement aux taux de croissance des prédateurs par capture proies et à leur taux de mortalité propre. Les constantes (strictement positives) c et d représentent le taux de disparition des proies par capture et leur taux de prolifération.

On suppose que $p_0, q_0 > 0$. On donne en annexe quelques éléments qui permettent d'obtenir les propriétés suivantes des solutions $(p(t), q(t))$:

– le système admet une solution unique définie sur $\mathbb{R}^+$.

– $\forall\, t > 0$ on a $p(t) > 0$, $q(t) > 0$ (ce qui est « normal »pour une densité de population).

– $\exists C > 0$ tel que $p(t) < C$ et $q(t) < C$, $\forall\, t \in \mathbb{R}^+$.

– le caractère périodique du système.

– les valeurs moyennes de $p(t)$ et $q(t)$ (i.e. $\lim_{t\to+\infty} \frac{\int_0^t p(s)ds}{t}$ et $\lim_{t\to+\infty} \frac{\int_0^t q(s)ds}{t}$) sont indépendantes des données initiales et valent respectivement d/c et b/a.

On prend $a = 1$, $b = 2$, $c = 1$, $d = 3$. Voici la simulation avec Scipy

```
>>> a=1.
>>> b=2.
>>> c=1.
>>> d=3.
>>> def func(y,t):
>>> w=zeros(2)
>>>     return array([a*y[0]*y[1]-b*y[0],
>>>     -c*y[0]*y[1]+d*y[1]])
>>> y0=array([1.,1.])
>>> t0=array([0, 1., 2., 5.])
>>> t=linspace(0,10,500)
>>> v=itg.odeint(func,y0,t)
>>> pl=plt.figure()
>>> plt.grid()
>>> plt.plot(t,v[:,0], label=u'Prédateur')
>>>plt.plot(t,v[:,1], label=u'Proie')
>>> plt.legend(loc='best')
>>> plt.title('Evolution␣des␣populations␣de␣renards␣et␣de␣lapins')
>>> w=itg.odeint(func,y0,t0)
```

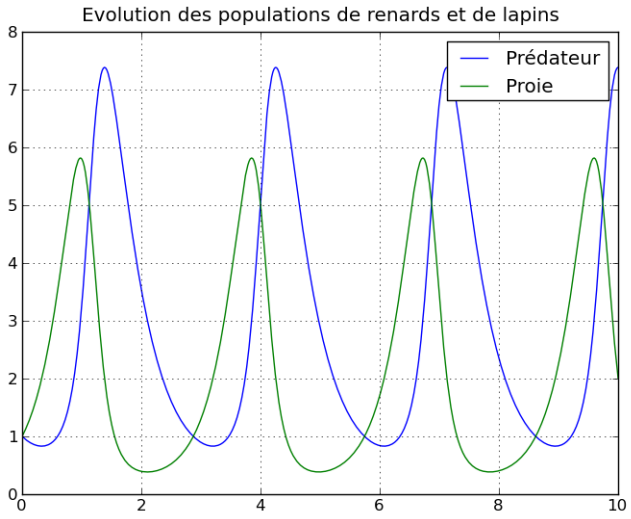

FIGURE 2. Système de proie/prédateur

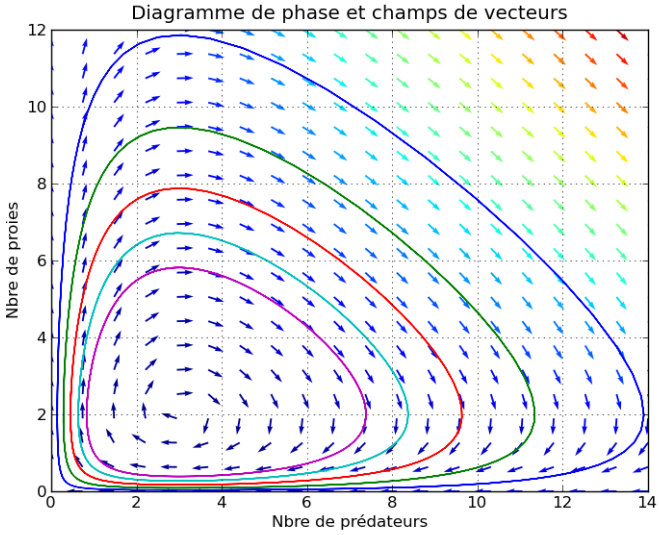

FIGURE 3. Plusieurs trajectoires et diagramme de phase

Le vecteur `w` contient les valeurs des approximations pour $t \in \{0,1,2,5\}$:

```
array([[ 1.        ,  1.        ],
       [ 3.25685482,  5.80175028],
       [ 3.54373184,  0.39522892],
       [ 2.88594899,  0.38509811]])
```

On comprend alors aisément que pour tracer le graphique de chaque population (voir figure 2 on utilise `v[:,0]` et `v[:,1]`. Il existe une autre solution comme par exemple (pris sur une page d'exemples de Scipy)

```
>>> v=itg.odeint(func,y0,t)
>>> proie, pred=v.T
```

Remarque 2. La syntaxe utilisée pour définir la fonction vectorielle associée au système différentielle est préférable à celle-ci

```
>>> def func(y,t):
>>>     w=zeros(2)
>>>     w[0]=a*y[0]*y[1]-b*y[0]
>>>     w[1]=-c*y[0]*y[1]+d*y[1]
>>>     return w
```

En effet cette fonction n'est pas vectorisée et ne permet pas pour un vecteur de données de $\mathbb{R}^2$ de renvoyer les valeurs de F pour les données. Ceci est utile car le système différentiel ne dépend pas du temps (on dit alors autonome) et que le tracé du champs de vecteur donne des informations sur les trajectoires.

Pour un tel système différentiel autonome il est intéressant de tracer le diagramme de phase, c'est-à-dire, la courbe paramétrée $(p(t),q(t))$ ainsi que les champs de vecteurs normalisés. Le champ de vecteurs consiste pour une grille de points $((x_1^{(i)},y_1^{(i)}))$ à calculer les vecteurs normalisés de même direction que $F((x_1^{(i)},y_1^{(i)}))$. La fonction de Matplotlib est `quiver(X,Y,U,V,C)` où `X,Y` sont les coordonnées (la grille), `U,V` les composantes des vecteurs et `C` un argument optionnel de couleur. Voici la suite et le résultat (figure 3).

```
>>> p2=plt.figure()
>>> plt.grid()
>>> for i in linspace(0.3,1.,5):
>>>     v=itg.odeint(func,i*y0,t)
>>>     plt.plot(v[:,0],v[:,1])
>>> x=linspace(0,14,20)
>>> y=linspace(0,12,20)
>>> plt.title(u"Diagramme␣de␣phase␣et␣champs␣de␣vecteurs")
>>> plt.xlabel(u'Nbre␣de␣prédateurs')
>>> plt.ylabel(u'Nbre␣de␣proies')
>>> X1, Y1= meshgrid(x,y)
>>> DX1, DY1 = func([X1,Y1],0)
>>> M= (hypot(DX1,DY1))
>>> M [ M== 0] =1.
>>> DX1 /= M
>>> DY1 /= M
>>> plt.quiver(X1,Y1,DX1,DY1,M)
```

Remarque 3. Dans le programme précédent on trace plusieurs courbes $(p(t),q(t))$ pour différentes valeurs initiales.

Pour le champ de vecteurs, la fonctions `hypot` est là pour normaliser. La commande `hypot(DX1,DY1)` renvoie le tableau des normes des vecteurs et avec la précaution d'éviter la valeur 0 (commandes `M [ M== 0] =1` la normalisation s'effectue via `DX1 /= M, DY1 /= M`. L'option `M` donnée à `quiver` est un tableau de même caractéristique que la grille, de valeurs positives, et permet d'avoir la variation de couleur.

2. ode

La commande `odeint` est dédiée exclusivement à la librairie Fortran ODEPACK. Afin de pouvoir utiliser différents solveurs d'EDO mais sans démultiplier les commandes style `odetruc`, une interface a été développée : une nouvelle classe `ode` à laquelle on peut ajouter des librairies dédiées à la résolution numérique des EDO. Actuellement il est possible d'utiliser

- `vode` : intégration de systèmes différentiels réels. Pour les problèmes raides c'est une méthode BDF (ordre ≤ 5) et pour les problèmes non raide c'est une méthode d'Adams (ordre ≤ 12).
- `zvode` : idem que `vode` mais à valeurs complexes.
- `dopri5` : méthode explicite de Runge-Kutta d'ordre (4)5 avec contrôle du pas due à Dormand et Prince
- `dop853` : idem mais d'ordre 8(5,3)

Le fonctionnement de `ode` est complètement différent de celui de `odeint`.

2.1. **L'intégrateur** `vode`**.** Voici un exemple avec le système de proie-prédateur :

```
>>> a=1.
>>> b=2.
>>> c=1.
>>> d=3.
>>> # définition de f attention f(t,y) !!
>>> def f(t,y,a,b,c,d):
>>>     return array([a*y[0]*y[1]-b*y[0],
>>>     -c*y[0]*y[1]+d*y[1]])
>>> # donnée initiale
>>> t0=0.
>>> y0=array([1., 1.])
>>> # définition de notre edo
>>> r=itg.ode(f)
>>> # choix du solveur
>>> r.set_integrator('vode',method='adams')
>>> # on balance les paramètres a,b,c,d
>>> r.set_f_params(a,b,c,d)
>>> # on balance la donnée initiale
>>> r.set_initial_value(y0,t0)
>>> # Peut-on enfin intégrer ?
>>> print r.t
>>> v=r.integrate(r.t+10.)
>>> print v, r.t
```

La routine `r.integrate` n'accepte pas comme un argument un tableau de valeurs de t. La variable `r.t` contient initialement la valeur Pour tracer un graphique équivalent à celui de la figure 2, il est courant de faire

```
>>> r.set_initial_value(y0,t0)
>>> t1 = 10.
>>> dt = .1
>>> t = [t0]
>>> sol = y0
>>>while r.successful() and r.t < t1 :
>>>     r.integrate(r.t+dt)
>>>     t.append(r.t)
>>>     sol=vstack((sol,r.y))
>>> p1=plt.figure()
>>> plt.plot(t,sol[:,0])
>>> plt.plot(t,sol[:,1])
>>> plt.show()
>>> p2=plt.figure()
>>> plt.plot(sol[:,0].T,sol[:,1])
```

Le mécanisme de `ode.integrate` fait que pour avoir une meilleure précision il faut prendre un pas de temps plus petit (`dt=0.05`) ou encore laisser faire la routine pour le choix du pas de temps avec l'option `step=True` dans la routine `integrator`. On obtient un tableau de 397 valeurs et les figures 6 et 7

```
>>> r.set_initial_value(y0,t0)
```

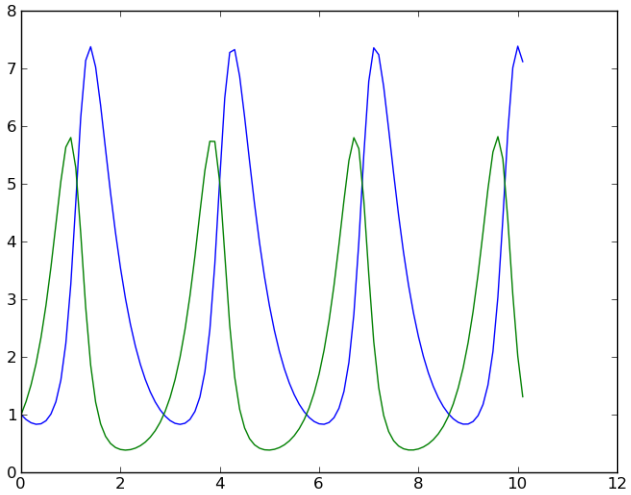

FIGURE 4. VODE

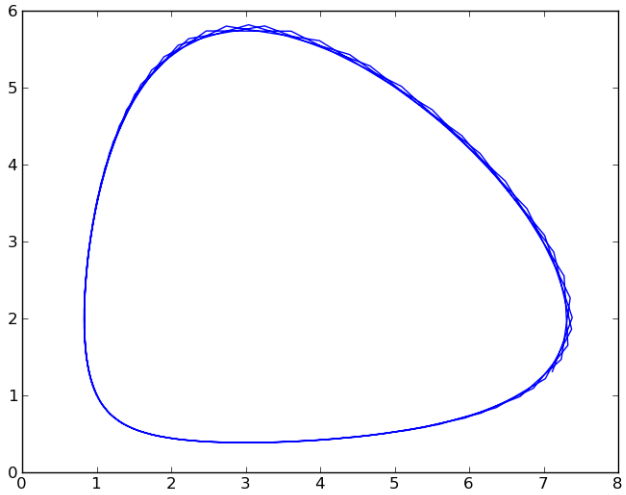

FIGURE 5. VODE

```
>>> t1 = 10.
>>> t = [t0]
>>> sol = y0
>>> while r.successful() and r.t < t1 :
>>>     # Laissons faire l'intégrateur
>>>     r.integrate(t1,step=True)
>>>     t.append(r.t)
>>>     sol=vstack((sol,r.y))
>>> print size(t)
>>> p3=plt.figure()
>>> plt.plot(t,sol[:,0])
>>> plt.plot(t,sol[:,1])
>>> plt.show()
>>> p4=plt.figure()
>>> plt.plot(sol[:,0].T,sol[:,1])
```

Remarque 4 (Choix `adams` vs `bdf`)**.** C'est très important, car sur un problème raide, il est possible que la méthode d'Adams échoue. Mais la routine vous préviendra

```
def exemple6():
    def f(t,y):
        return -150.*y+30
    y0=1./5.+10**(-1)
    t0=0.
    r=itg.ode(f)
    # mauvais choix du solveur
    r.set_integrator('vode',method='adams',order=1)
    r.set_initial_value(y0,t0)
    print r.integrate(2.)
    print r.integrate(10.)
    print r.integrate(25.)
```

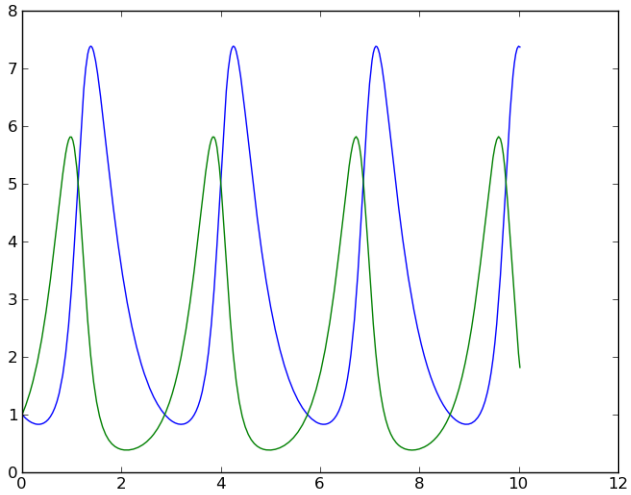

FIGURE 6. VODE choisit le pas de temps

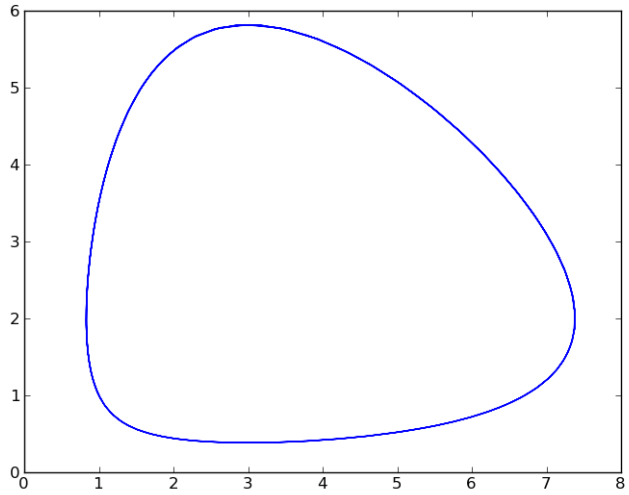

FIGURE 7. VODE choisit le pas de temps

2.2. **Les intégrateurs** `dopri5` **et** `dop853` **à toute vitesse.** Bizarrement le passage des paramètre via `set_f_params` ne semble pas fonctionner ?