Analyse numérique avec Python

PTSI Lycée Eiffel

5 mai 2015

Retour au Python pour ce dernier gros chapitre de l'année (un tout petit chapitre final sera sûrement consacré aux rudiments de Scilab), où nous allons étudier ensemble (et programmer en Python) quelques algorithmes classiques d'analyse numérique. Le but est de résoudre des problèmes mathématiques fréquemment rencontrés en modélisation (donc dans les autres sciences, par exemple des équations différentielles), en ne cherchant surtout pas à comprendre les mathématiques quisont cachées derrière, mais en tentant de trouver une méthode efficace pour déterminer une solution utilisable en pratique (mais approchée) au problème donné. On essaiera donc d'insister sur le côté numérique de la résolution, en tentant d'évaluer l'efficacité des algorithmes (complexité, précision des résultat obtenus), et les limites des modèles présentés (problèmes d'arrondis, de validité de certains tests, instabilité numérique).

1 Résolution approchée d'équations du type $f(x) = 0$.

Exemple : Quantité de problèmes en modélisation se ramènent à la résolution d'équations à une inconnue réelle. Pour en donner un tout à fait ordinaire mais fondamentale en physique, on lance un projectile soumis uniquement à la force de gravitation avec une vitesse initiale et une hauteur initiale données, et on souhaite savoir la distance qu'il parcourt avant de toucher le sol. Dans sa modélisation la plus simple, ce problème se ramène à la résolution d'une équation du second degré. Plus généralement, on se placera dans la situation où on dispose des données suivantes :

- une fonction f continue et un intervalle sur lequel elle s'annule au moins une fois.
- une valeur initiale x_0 (qui peut simplement être une borne de l'intervalle).
- une précision souhaitée pour la valeur approchée de la solution.

1.1 Résolution exacte.

Ces méthodes ne nous concernent pas dans ce cours, elles sont plutôt du ressort de votre professeur de mathématiques. Il faut tout de même être conscient que :

- on ne sait résoudre de façon exacte que très très peu d'équations (en gros tout ce qui se ramène à une équation du premier ou du second degré).
- même quand on sait le faire, on est confrontés à des problèmes numériques. Pour l'équation du second degré, on a besoin de calculer $\sqrt{\Delta}$, mais comment effectue-t-on un tel calcul numériquement ? Ce sera certainement de façon approchée, et il faut un algorithme pour effectuer le calcul (cf plus bas).

1.2 Dichotomie.

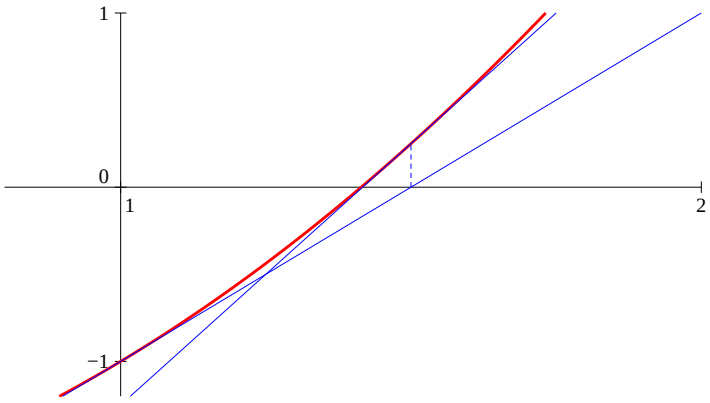
Cette méthode a déjà été vue dans le chapitre 2. Rappelons qu'elle consiste à construire deux suites (a_n) et (b_n) en partant de $a_0 = a$ et $b_0 = b$ (a et b étant les bornes de l'intervalle d'étude), et en divisant l'intervalle en deux à chaque étape. Citons simplement le résultat suivant :

Théorème 1. En notant α une solution de l'équation $f(x) = 0$, par la méthode de dichotomie, on aura toujours $|a_n - \alpha| \leqslant \dfrac{|b-a|}{2^n}$.

On peut donc maîtriser facilement la précision de l'appriximation lorsqu'on utilise la méthode de dichotomie. Le seul inconvénient est que la méthode est relativement peu efficace. Pour obtenir une précision de 10 chiffres après la virgule en partant d'un intervalle de largeur 1, il faut une bonne trentaine d'étapes. On gagne en gros trois chiffres significatifs toutes les 10 étapes, puisque $\frac{1}{2^{10}} \simeq \frac{1}{1000}$. Les seuls limites numériques que peut rencontrer cet algorithme sont dues aux nombreux tests de signe effectués (un à chaque étape), qui peuvent devenir imprécis quand la fonction f prend des valeurs très proches de 0.

1.3 Méthode de Newton.

Le principe de la méthode de Newton est le suivant : sous des hypothèses plus ou moins fortes sur la fonction f (en première approximation, on aura besoin que f soit dérivable sur l'intervalle d'étude), on part d'un point x_0, et on construit une suite récurrente convergeant vers la solution de l'équation en prenant pour x_{n+1} l'abscisse du point d'intersection de l'axe des abscisses et de la tangente à la courbe de f en son point d'abscisse x_n. La tangente étant « proche » de la courbe, il parait raisonnable d'imaginer que ce point sera relativement proche du point d'intersection de la courbe elle-même avec l'axe des abscisses. Un petit dessin pour illustrer tout ça :



Sur cette figure (qui correspond à la fonction $f(x) = x^2 - 2$ reprise en exemple ci-dessous), on est partis de $x_0 = 1$, et on trouve une bonne approximation de la racine après seulement deux étapes. Plus généralement, la convergence de la méthode de Newton est très rapide :

Théorème 2. En notant α la racine recherchée, par la méthode de Newton, on aura $\log|x_n - \alpha| \leqslant 2^n \log(K|x_0 - a|) - \log(K)$, où K est une constante dépendant de f définie par $K = \frac{\max_I |f''|}{2\min_I |f'|}$, I étant l'intervalle d'étude.

Autrement dit, l'écart entre x_n et α sera en gros de l'ordre de $\frac{1}{2^{2^n}}$, ce qui est gigantesque. À chaque étape, la précision est doublée ! Il ne faut que quelques étapes (à peine cinq en général) pour obtenir des valeurs approchées à 10^{-10} près à l'aide de la méthode de Newton. Quels peuvent alors être les inconvénients de la méthode de Newton ? Il y en a quelques-uns :

- Il faut connaitre la dérivée f' de la fonction f pour calculer les termes de la suite récurrente. Sinon, il faudra approher la valeur de $f'(x_n)$ (on peut calculer une dérivée approchée en calculant la pente de la droite reliant deux points de la courbe proches de celui d'abscisse x_n), ce qui augmente largement les imprécisions de calcul.
- La majoration de l'écart est beaucoup moins pratique, et constitue un critère d'arrêt de l'algorithme peu performant en pratique (il devient rapidement plus faible que les erreurs d'arrondi !). Il vaut mieux prendre un critère du genre $|x_{n+1} - x_n| < \varepsilon$ comme condition d'arrêt, mais l'erreur commise peut alors être difficile à estimer.

- Surtout, Newton va marcher très mal sur des fonctions qui ne sont pas suffisamment régulières ou sur des intervalles trop grands. Notamment, si la dérivée de f s'annule à un endroit, on est en gros danger (ou même si elle devient trop petite, on risque de sortir de l'intervalle). En pratique, Newton marche très bien sur une fonction convexe (ou concave) sur un intervalle donné.

Exemple pratique : calcul approché de $\sqrt{2}$. Pour obtenir une valeur approchée de $\sqrt{2}$, il suffit d'appliquer la méthode de Newton à la fonction $f : x \mapsto x^2-2$, en partant d'une valeur positive de x_0, par exemple $x_0 = 1$. L'équation de la tangente en x_n à la courbe de f étant $y = f'(x_n)(x-x_n)+f(x_n)$, on aura $x_{n+1} = x_n - \dfrac{f(x_n)}{f'(x_n)}$ (la condition revient à poser $y = 0$ dans l'équation précédente). Ici, $f'(x) = 2x$, et on obtient simplement $x_{n+1} = x_n - \dfrac{x_n^2-2}{2x_n} = \dfrac{x_n}{2} + \dfrac{1}{x_n}$. Un programme Python très simple appliquant la méthode de Newton dans ce cas est le suivant (on donne comme argument la valeur initiale et le nombre d'itérations souhaité) :

```
> def Newton(x,n) :
>         a=x
>         for i in range(n) :
>                 a=a/2+1/a
>         return a
```

Terminons avec un petit tableau récapitulatif des performances de nos deux algorithmes. À gauche, la dichotomie a été effectuée à partir de $a_0 = 1$ et $b_0 = 2$, à droite Newton a été effctué en partant de $x_0 = 1$.

n	dichotomie	Newton
2	1.25	1.4166666666666
3	1.375	1.41421568
4	1.375	1.41421356237
5	1.40625	
10	1.4140625	
20	1.4142131805419922	
30	1.4142135614529252	

Toutes les décimales affichées pour Newton quand $n = 4$ sont déjà exactes, elle ne bougent plus ensuite.

1.4 Méthode de la sécante

Il existe des méthodes plus ou moins proches de la méthode de Newton, évitant de devoir connaitre la dérivée pour calculer la valeur approchée de la racine. Parmi celles-ci, la méthode de la sécante consiste à partir de deux points, à tracer la droite reliant les deux points correspondants sur la courbe de f, et à remplacer le premier des deux points (plus généralement l'avant-dernier point calculé) par l'abscisse du point d'intersection de cette droite avec l'axe des abscisses. Cette méthode est moins bonne que Newton, mais pas tellement.

1.5 Déjà disponible en Python

Comme pour tous les algorithmes de ce chapitre, nous ne ferons que réimplémenter des fonctions déjà existantes en Python. Sans faire une description de ce qui existe déjà, je vous donnerai à chaque fois les modules Python contenant les fonctions utiles, et libre à vous d'aller en regarder les fonctionnalités précises de plus près, puisque tous les modules sont documentés en ligne. Bien sûr, cette documentation est en anglais, et pas toujours très claire, mais elle indique pour chaque fonction les arguments et options disponibles, et il faut que vous vous entrainiez à utiliser cette aide.

Concernant les méthodes de résolution approchée d'équations, tout se trouve dans le module **scipy.optimize**, qui est lui-même un sous-module du (gros) module d'analyse numérique scipy. Il contient entre autres les fonctions suivantes :

Fonctions utiles du module scipy.optimize.

- **brentq(f,a,b)** : détermine une racine de la fonction f dans l'intervalle $[a, b]$ par la méthode de Brent (pas étudiée dans ce cours !).
- **bisect(f,a,b)** : détermine une racine de f dans $[a, b]$ en effectuant une dichotomie.
- **newton(f,x0)** : détermine une racine par la méthode de Newton ou approché en partant de x0 (si on donne la dérivée en argument supplémentaire, c'est la méthode de Newton que nous avons vue qui sera utilisée ; on peut également donner la dérivée seconde pour qu'une méthode encore plus efficace soit mise en oeuvre ; en l'absence de dérivée, c'est une méthode du type sécante qui sera utilisée).
- **root(fun,x0)** : détermine une racine de la fonction fun, qui peut ici être une fonction de plusieurs variables (des options supplémentaires permettent de choisir une méthode particulière, mais celles-ci ne sont pas à notre programme).

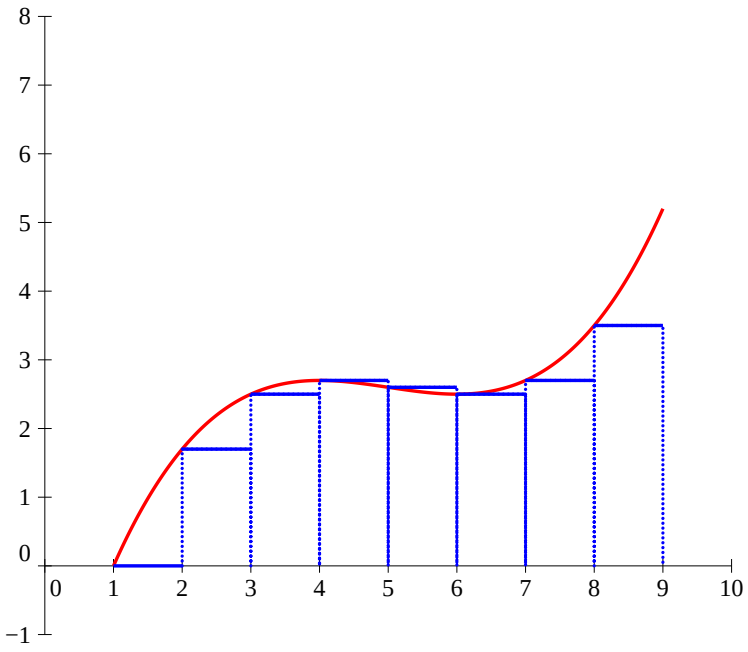
2 Intégration numérique.

Dans cette section, nous nous intéresserons aux algorithmes permettant le calcul numérique d'intégrales. Il s'agit bien sûr de faire à nouveau du calcul approché, et donc de ne pas utiliser de calcul de primitive, même si aujourd'hui beaucoup de logiciels de calcul formel (et de calculatrices) sont capables de faire de l'intégration exacte. Comme dans le cas des équations, il faut de toute façon avoir conscience qu'on ne sait calculer exactement que très peu d'intégrales, même quand elle font intervenir des fonctions usuelles. Ainsi, la fonction $x \mapsto e^{-x^2}$, d'une importance fondamentale en probabilités, n'admet pas de primitive exprimable à l'aide des fonctions usuelles (oui, oui, ce genre de résultat se démontre !). Le principe général commun aux trois méthodes que nous allons présenter est simple : découper l'intervalle d'intégration en petits morceaux, et approcher sur chacun de ces petits intervalles la courbe représentative de la fonction f par une courbe très simple pour laquelle le calcul d'aire est facile.

2.1 Méthode des rectangles.

Quoi de plus simple comme fonction qu'une fonction constante ? Et quoi de plus simple comme aire à calculer qu'une aire de rectangle ? La première méthode, la plus rudimentaire, que nous allons voir, consiste donc à approcher notre fonction sur chaque sous-intervalle par la fonction constate prenant la même valeur que f à gauche de l'intervalle :

Définition 1. Soit f une fonction continue sur un segment $[a, b]$. Pour calculer son intégrale approchée par la méthode des rectangles, on pose $h = \dfrac{b-a}{n}$ et on pose $x_i = a + ih$, pour tout entier $i \in \{0, \ldots, n\}$ (ainsi, $a_0 = a$ et $a_n = b$). On pose ensuite $S_n(f) = h \displaystyle\sum_{i=0}^{n-1} f(x_i)$.



Sur la figure, on a $a = 1$, $b = 9$, $n = 8$ (donc $h = 1$), l'aire sous la courbe est approchée par la somme des aires des rectangles bleus, qui ont pour largeur commune h (donc 1), et pour hauteur $f(x_i)$.

Théorème 3. L'aire approchée obtenue par la méthode des rectangles converge vers l'intégrale de la fonction f quand n tend vers $+\infty$. Plus précisément, si f est une fonction de classe $\mathcal{C}^1$ sur le segment $[a, b]$, on a la majoration de l'erreur suivante : $\left| S_n(f) - \displaystyle\int_a^b f(t)\ dt \right| \leqslant \frac{M(b-a)^2}{2n}$, où M est un majorant de $|f'|$ sur $[a, b]$.

Remarque 1. La méthode des rectangles nécessite de faire n évaluations de la fonction, ainsi que n sommes (qui peuvent être considérées comme négligeables). C'est donc un algorithme linéaire par rapport au nombre d'intervalles utilisés pour le découpage. Sa convergence est hélas très lente, ce qui en fait un algorithme peu utilisé en pratique. En négligeant le facteur constant $\frac{M(b-a)^2}{2}$, il faudra un ordre de grandeur de 10^{10} intervalles pour obtenir une valeur approchée correcte à dix décimales près. On peut légèrement modifier l'algorithme pour prendre comme valeur constate sur chaque intervalle, non plus $f(x_i)$ (valeur à gauche), mais $f\left(\frac{x_i + x_{i+1}}{2}\right)$, c'est-à-dire la valeur de f au point situé au milieu de l'intervalle. cette méthode, aussi connue sous le nom de méthode du point médian, donne en pratique des résultats légèrement meilleurs que la méthode des rectangles classique.

Programme Python pour la méthode des rectangles :

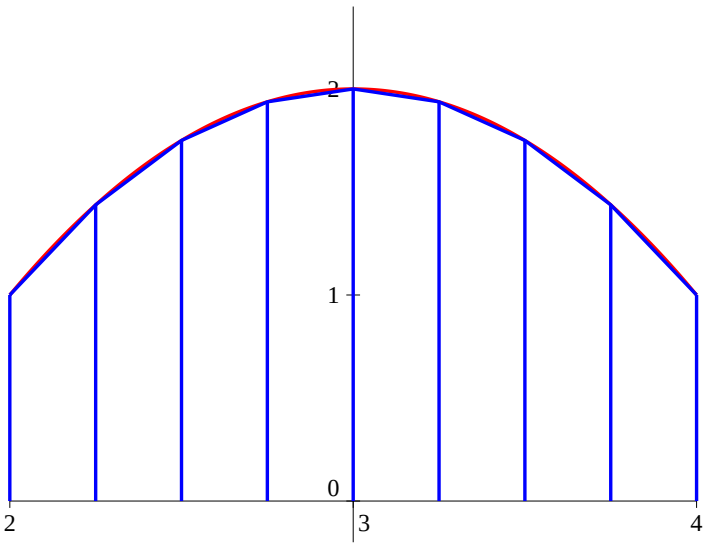
```
> def rectangles(f,a,b,n) :
>         h=(b-a)/float(n)
>         z=0
>         for i in range(n) :
>                 z=z+f(a+i*h)
>         return h*z
```

2.2 Méthode des trapèzes.

Le principe général est exactement le même que celui de la méthode des rectangles, mais on approche cette fois-ci la courbe sur le segment $[x_i, x_{i+1}]$ par le segment de droite reliant les deux

points de la courbe d'abscisses x_i et x_{i+1}, ce qui revient à calculer une somme d'aires de trapèzes pour approcher l'intégrale :

Définition 2. Avec les mêmes notations que précédemment, la méthode des trapèzes approche l'intégrale de f par la somme $T_n(f) = h\sum_{i=0}^{n-1} \frac{f(x_i)+f(x_{i+1})}{2}$.



Sur cette figure, $a = 2$, $b = 4$ et $n = 8$. Visuellement, l'impression est nettement meilleure que pour la méthode des rectangles.

Remarque 2. En fait, la méthode des trapèzes est de complexité très proche de la méthode des rectangles. On peut écrire légèrement différemment la somme : $T_n(f) = h\left(\frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f(x_i)\right)$ pour se ramener à $n+1$ évaluations de la fonction f. En fait, la seule différence avec la formule des rectangles est que le $f(a)$ initial est transformé en $\frac{f(a)+f(b)}{2}$.

Théorème 4. Si f est une fonction de classe $\mathcal{C}^2$ sur le segment $[a, b]$, on a la majoration de l'erreur suivante pour la méthode des trapèzes : $\left|T_n(f) - \int_a^b f(t)\,dt\right| \leqslant \frac{M_2(b-a)^3}{12n^2}$, où M_2 est un majorant de $|f''|$ sur $[a, b]$.

Remarque 3. Même si ces estimations ne sont que des majorants de l'erreur commise, il est déjà manifeste que cette méthode convergence beaucoup plus vite que la précédente. Pour obtenir une dizaine de décimales correctes, en négligeant le facteur constant, il faudra de l'ordre de 10^5 intervalles, ce qui est un ordre de grandeur raisonnable pour une machine.

Programme Python pour la méthode des trapèzes :

```
> def trapezes(f,a,b,n) :
>         h=(b-a)/float(n)
>         z=0.5*(f(a)+f(b))
>         for i in range(1,n) :
>                 z=z+f(a+i*h)
>         return h*z
```

2.3 Méthode de Simpson.

Après avoir approché notre fonction par une fonction constante (méthode des rectangles), puis par une fonction affine (méthode des trapèzes), l'étape logique suivante est de tenter une approximation par des courbes de degré 2, donc des paraboles. C'est le principe de la méthode de Simpson (qui partage avec ses cousines la première étape de découpage de l'intervalle en morceaux), mais une question se pose tout de même : pour définir une parabole, il faut en connaitre trois points, on ne peut donc pas se contenter de prendre les extrémités des intervalles comme pour les trapèzes. Pas grave, on prendra comme troisième point le milieu de chaque intervalle. Une fois ces trois points choisis, quelle formule obtient-on ? Calculer l'aire sous une parabole est un peu moins évident que sous une droite, et comme j'ai promis qu'on ne ferait pas de maths dans ce chapitre, on va simplement admettre la formule suivante :

Définition 3. Avec les mêmes notations que précédemment, la méthode de Simpson approche l'intégrale de f par la somme $U_n(f) = h\displaystyle\sum_{i=0}^{n-1} \frac{f(x_i) + 4f(\frac{x_i+x_{i+1}}{2}) + f(x_{i+1})}{6}$.

Remarque 4. Comme dans le cas des trapèzes, on peut remanier un peu la somme pour minimiser le nombre de calculs à effectuer : $U_n(f) = h\left(\dfrac{f(a)+f(b)}{6} + \dfrac{1}{3}\displaystyle\sum_{i=1}^{n-1} f(x_i) + \dfrac{2}{3}\sum_{i=0}^{n-1} f\left(\dfrac{x_i+x_{i+1}}{2}\right)\right)$. Il y a donc $2n+1$ évaluations de la fonction f à faire, soit environ deux fois plus que par les deux méthodes précédentes.

Théorème 5. Si f est une fonction de classe $\mathcal{C}^4$ sur le segment $[a,b]$, on a la majoration de l'erreur suivante pour la méthode de Simpson : $\left|U_n(f) - \displaystyle\int_a^b f(t)\ dt\right| \leqslant \dfrac{M_4(b-a)^5}{180n^4}$, où M_4 est un majorant de $|f^{(4)}|$ sur $[a,b]$.

Remarque 5. La convergence est vraiment nettement plus rapide que pour les rectangles ou même les trapèzes. En supposant que M_4 ne prenne pas des valeurs extrêmement élevées, on a toutes les chances d'obtenir nos 10 chiffres significatifs pour $n = 100$ ou à peu près. On peut également constater un phénomène intéressant : la méthode des rectangles donne une valeur exacte de l'intégrale pour des fonctions constantes ; la méthode des trapèzes fait de même pour les fonctions affines ; sans surprise, la méthode de Simpson est exacte pour les polynômes de degré 2, mais aussi pour ceux de degré 3. En pratique, cette méthode est largement assez efficace pour tous les calculs que nous pourrions avoir envie de faire. Les plus curieux se demanderont quand même si on peut continuer à créer des méthodes de plus en plus précises. La réponse est oui, bien entendu, mais elles nécessiteraient de plus en plus d'évaluations de la fonction f, et des formules de plus en plus compliquées à mettre en place. En pratique, les autres méthodes existantes (par exemple la méthode de Romberg) procèdent autrement, en accélérant la convergence obtenue par une autre méthode.

Programme Python pour la méthode de Simpson :

```
> def Simpson(f,a,b,n) :
>         h=(b-a)/float(n)
>         z=(f(a)+f(b))/6
>         for i in range(1,n) :
>                 z=z+f(a+i*h)/3
>         for i in range(n) :
>                 z=z+f(a+(2*i+1)*h/2)*2/3
>         return h*z
```

Exemple de comparaison entre les trois méthodes. Nous allons faire calculer à Python par chacune des trois méthodes l'intégrale $I = \displaystyle\int_0^1 \frac{1}{1+x}\ dx$, qui vaut, comme peut le calculer n'importe

quel élève de PTSI, ln(2). Commençons par donner une valeur approchée de la valeur exacte de l'intégrale (obtenue avec Python !) : $\ln(2) \simeq 0.69314718055994529$.

n	rectangles	trapèzes	Simpson
3	0.7833333333333333	0.7	0.6931697931697931
5	0.7456349206349208	0.6956349206349208	0.6931502306889303
10	0.718771403175428	0.693771403175428	0.693147374665116
50	0.6981721793101955	0.6931721793101955	0.6931471808723674
100	0.6956534304818242	0.6931534304818241	0.6931471805794752
1000	0.6933972430599373	0.6931472430599374	0.6931471805599484
1000000	0.6931474305600013	0.6931471805600012	0.69314718055998

2.4 Déjà disponible en Python

Pour l'intégration numérique, tout se trouve dans le module **scipy.integrate**. Vous allez me dire qu'on aurait pu le deviner au vu du nom du module, mais celui-ci contient en fait également des fonctions consacrées à ce que nous allons étudier dans la section suivante de ce cours, à savoir les résolutions d'équations différentielles. Cela n'a rien de surprenant dans la mesure où résoudre une équation différentielle revient souvent à faire un calcul de primitive (d'ailleurs les anglo-saxons parlent effectivement d'intégrer une équation différentielle plutôt que de la résoudre).

Fonctions utiles du module scipy.integrate.

- **quad(f,a,b)** : calcule une valeur approchée de $\int_a^b f$ (méthode non spécifiée, optimisée par Python).
- **dblquad(f,a,b)** et **tplquad(f,a,b)** : même chose pour des intégrales doubles ou des intégrales triples.
- **romberg(f,a,b)** : calcule une valeur approchée de $\int_a^b f$ à l'aide de la méthode de Romberg (encore plus efficace que ce que nous avons vu dans cette section).
- **cumtrapz(y)** et **simps(y)** : calculent une intégrale cumulée de y à l'aide de la méthode des trapèzes ou de la méthode de Simpson. Attention, la variable ici est un tableau de valeurs (les valeurs prises par f aux points de découpage de notre intervalle, donc la liste des $f(x_i)$ avec nos notations) et la fonction resort également un tableau, constituée des valeurs approchées prises par la primitive de f aux même points.

3 Résolution approchée d'équations différentielles.

Après les équations numériques, les équations différentielles. Comme dans la première partie de ce chapitre, le constat est amer : on ne sait pratiquement rien résoudre de façon exacte dans ce domaine (d'ailleurs, la résolution des équations différentielles et plus généralement des équations aux dérivées partielles sur les fonctions à plusieurs variables constitue l'un des domaines de recherche les plus vastes en mathématiques à l'heure actuelle). Pourtant, on ne peut pas faire de sciences sans tomber très rapidement sur des problèmes nécessitant l'étude de telles équations. Pour donner un exemple extrêmement classique, un pendule simple a un mouvement régi par l'équation différentielle $\ddot{\theta} = \sin(\theta)$, équation impossible à résoudre de façon exacte (non, n'insistez pas, même moi je ne sais pas le faire). La solution classique en physique consiste à trouver des solutions exactes d'une équation approchée, en considérant que $\sin(\theta) \simeq \theta$ pour des valeurs faibles de l'angle θ. Ici, on aura l'approche exactement inverse, puisqu'on va chercher des solutions approchées d'équations exactes.
On l'a déjà signalé dans le paragraphe précédent, les équations différentielles ont un lien évident avec l'intégration numérique. C'est donc sans surprise que l'on va utiliser dans cette section des méthodes

proches de celles de la précédente : découpage de l'intervalle de résolution en n morceaux, puis approximation de la solution sur chacun de ces intervalles. C'est le principe de base de la méthode d'Euler, qui est la principale méthode de résolution numérique d'équations différentielles, sur laquelle nous allons concentrer tous nos efforts.

3.1 Équations du premier ordre.

Considérons donc pour commencer une équation du type $y'(t) = a(t)y(t) + b(t)$ (on modifiera l'écriture un peu plus loin pour se rendre compte qu'on peut également utiliser notre méthode avec des équations non linéaires), qu'on souhaite résoudre sur un intervalle $[t0, tf]$, avec comme condition initiale $y(t0) = y0$ (toutes les résolutions que nous allons faire ici sont des résolutions d'équations avec conditions initiales, autrement dit de problèmes de Cauchy en termes techniques ; seuls les mathématiciens s'intéressent à l'ensemble de toutes les solutions). Pour fixer les notations, on notera $h = \dfrac{tf - t0}{n}$ le pas de résolution (n étant le nombre de morceaux dans le découpage de notre intervalle) et $t_i = t_0 + i * h$. La méthode d'Euler consiste à calculer successivement des valeurs approchées de tous les réels $y(t_i)$ en approchant la courbe sur l'intervalle $[t_i, t_{i+1}]$ par sa tangente en t_i (si on connait une valeur approchée de $y(t_i)$, on en connait aussi une de $y'(t_i)$ à l'aide de l'équation). Autrement dit, on écrira que $y'(t_i) \simeq \dfrac{y(t_{i+1}) - y(t_i)}{h}$, soit $y(t_{i+1}) \simeq y(t_i) + h * y'(t_i)$, où $y(t_i) = a(t_i)y(t_i) + b(t_i)$.

Remarque 6. Signalons immédiatement les problèmes que pose cette approche :

- La courbe approchée obtenue sera nécessairement celle d'une fonction affine par morceaux puisqu'on ne fait que calculer les valeurs de y en certains points.
- Surtout, les approximations effectuées sont de plus en plus mauvaises : une seule approximation pour $y(t_1)$, mais deux pour $y(t_2)$ (puisque $y'(t_1)$ est déjà une valeur approchée), etc. Il est vraisemblable que nos valeurs soient de moins en moins précises au fur et à mesure que le temps avance. Pour essayer de gommer cet effet, on peut bien sûr augmenter la valeur de n, mais la précision accrue va-t-elle suffire à compenser le nombre d'étapes supplémentaire ? Rien ne nous garantit à priori que notre méthode va « converger » (ce qu'il faudrait d'ailleurs définir proprement) quand n tend vers $+\infty$. En fait, c'est un problème compliqué, que nous allons donc soigneusement esquiver, nous contentant de constater qu'en pratique, ça ne marche pas si mal. Il faudra tout de même être prudent si la dérivée de la solution prend des valeurs très grandes ou varie trop vite.

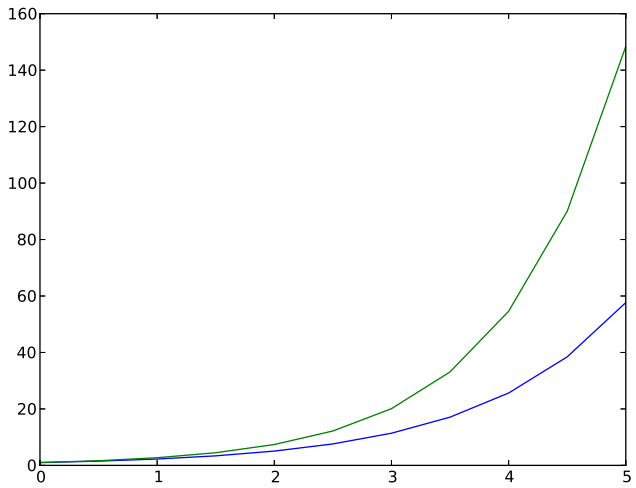
Pour la programmation, nous allons légèrement modifier la présentation de l'équation. Les paramètres que l'on va donner à notre fonction Python seront bien sûr les réels t0, tf et y0, ainsi que l'entier n, mais il faut aussi décrire l'équation. Le plus simple serait de donner les deux fonctions $a(t)$ et $b(t)$, mais on peut en fait généraliser largement le nombre d'équations possible en écrivant notre équation sous la forme $y'(t) = F(t, y(t))$, où $F(t, z)$ est une fonction à deux variables quelconque qui constituera le dernier argument (ou plutôt le premier) de notre fonction Python. Ainsi, l'équation $y'(t) = 3ty(t) + 2$ correspondra à $F(t, z) = 3tz + 2$, mais une équation non linéaire comme $y' = 3ty^2(t) - \dfrac{t}{y(t)}$ est tout à fait possible, avec $F(t, z) = 3tz^2 - \dfrac{t}{z}$. Notre méthode d'Euler consistera simplement, avec ces notations, à calculer $y(t_{i+1}) \simeq y(t_i) + h * F(t_i, y(t_i))$. Dans le programme, on va créer deux listes contenant les différentes abscisses et ordonnées des points de la courbe qu'on calcule. Tant qu'à faire, on tracera la courbe approchée à l'aide du module **matplotlib** que vous maîtrisez déjà sur le bout des doigts.

Programme Python pour la méthode d'Euler :

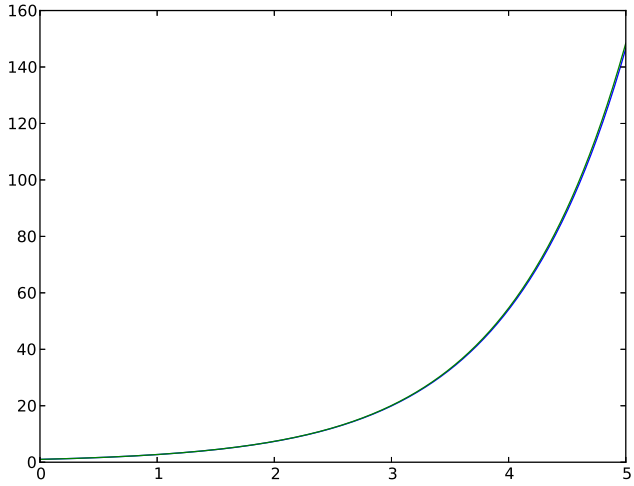
```
> import matplotlib.pyplot as plt
> def Euler(F,t0,tf,y0,n) :
>         t=t0
>         y=y0
```

```
>       h=(tf-t0)/float(n)
>       temps=[t0]
>       fonction=[y0]
>       for i in range(n) :
>           y=y+h*F(t,y)
>           t=t+h
>           temps.append(t)
>           fonction.append(y)
>       plt.plot(temps,fonction)
>       return fonction
```

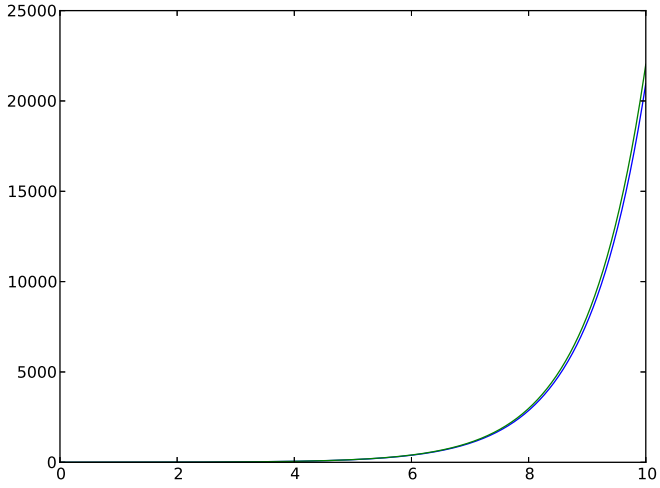
Exemple 1 : Approximation de la courbe de la fonction exponentielle par la méthode d'Euler. Tout le monde sait bien que la fonction exponentielle est solution de l'équation différentielle $y' = y$, avec condition initiale $y(0) = 1$. On peut donc tester notre programme en comparant la courbe approchée obtenue avec la vraie courbe de l'exponentielle. Dans un premier temps, prenons t0= 0, tf= 5 et $n = 10$ (avec bien sûr y0= 1). On obtient les courbes suivantes (en bleu la courbe approchée, en vert la courbe « réelle », qui est en fait celle d'une fonction affine par morceaux passant par les points de coordonnées (t_i, e^{t_i})) :



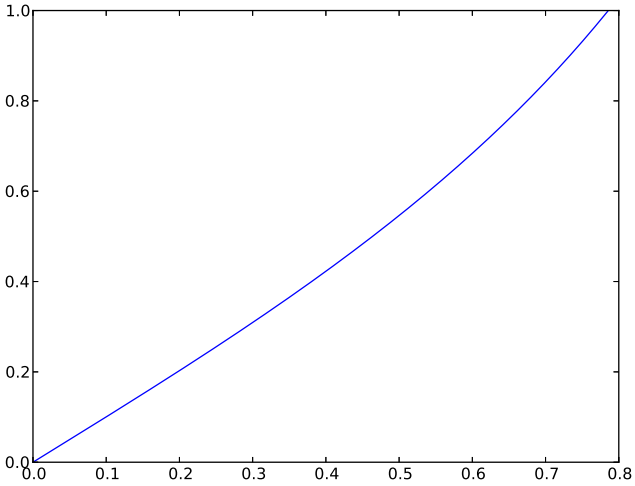
C'est pas terrible, mais ça n'a rien de surprenant avec une valeur de n aussi petite. Recommençons avec $n = 1000$ (même intervalle) :



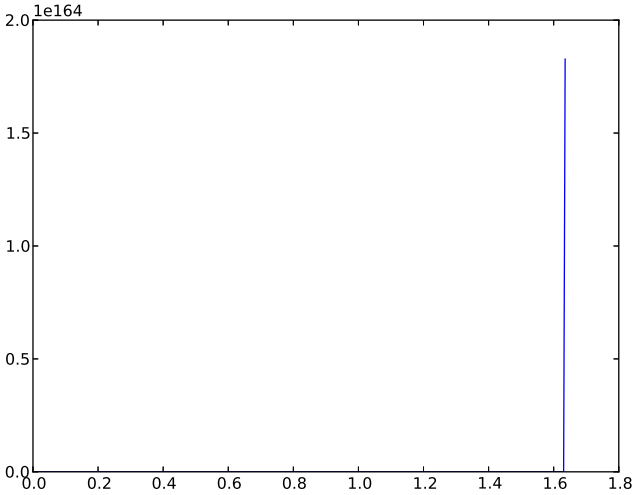
C'est nettement mieux ! Sans surprise, la méthode d'Euler ne fonctionnera correctement (mais nous n'avons hélas pas de mesure de l'erreur commise) que pour des valeurs de n relativement grandes. L'efficacité dépendra aussi de la largeur de l'intervalle. Ainsi, pour l'exponentielle, si on garde $n = 1000$ mais qu'on augmente tf pour le rendre égal à 10, l'approximation est moins bonne (on voit à l'oeil nu l'écart entre les deux courbes en fin d'intervalle) :



Exemple 2 : Une équation non linéaire. Pour tester notre Euler sur une équation non linéaire, prenons l'équation suivante $y'(t) = 1+y^2(t)$, avec comme condition initiale $y(0) = 0$. Vous connaissez très bien la solution mathématique exacte, il s'agit de la fonction tangente. Regardons ce que donne notre programme avec t0= 0, tf= $\dfrac{\pi}{4}$ et $n = 1000$:

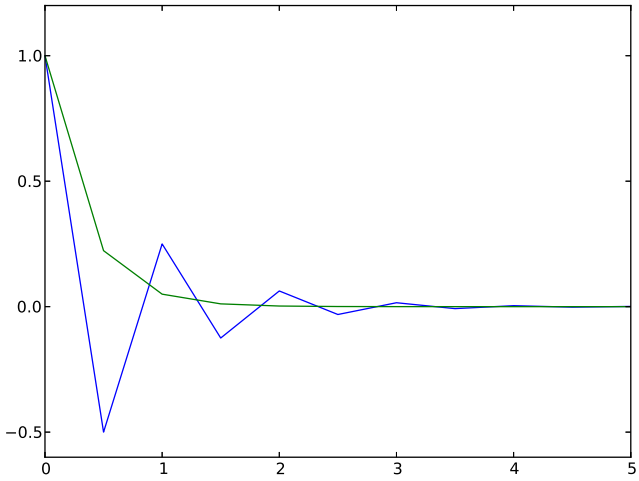


Eh bien c'est tout à fait correct, on obtient notamment une valeur en $\frac{\pi}{4}$ très proche de 1 (environ 0.999456). C'est normal, la dérivée de la fonction tangente a des variations très faibles sur cet intervalle. Mais que se passe-t-il si on tente le coup sur un intervalle où la fonction tangente n'est pas toujours définie, par exemple en prenant tf= $\frac{3\pi}{2}$?



La réponse est simple : n'importe quoi. En gros, les valeurs de y ont explosé pour atteindre des valeurs tellement délirantes qu'on ne voit plus rien, notamment parce que l'échelle n'est plus du tout adaptée. Ce n'est pas gênant, il faut simplement éviter d'appliquer la méthode d'Euler à n'importe quoi, et surtout sur n'importe quel intervalle.

Exemple 3 : Les limites de la méthode d'Euler. Un dernier exemple tout bête pour montrer ce qui se produire si on applique la méthode avec des valeurs de n vraiment trop petite, l'équation $y'(t) = -3y(t)$, avec pour condition initiale $y(0) = 1$. On connait bien la solution théorique, la fonction $t \mapsto e^{-3t}$, qu'on va superposer à la solution approchée :



Pour $n = 10$, on obtient ici une courbe approchée vraiment très mauvaise, avec des variations aléatoires (même si on finit par se rapprocher de la vraie courbe). Bien entendu, en prenant $n = 1000$ sur le même intervalle, on a quelque chose de tout à fait satisfaisant.

3.2 Systèmes d'équations du premier ordre.

Pour ce nouveau problème, nous allons quitter un peu votre domaine habituel de compétences pour nous intéresser à un exemple zoologique. Nous souhaitons étudier l'évolution simultanée de deux populations $x(t)$ et $y(t)$ (disons pour fixer les idées que x représente des lapins et y des renards, en tout cas il faut que l'une des deux espèces bouffe l'autre) soumises aux équations suivantes : $\begin{cases} x'(t) &= x(t)(3-2y(t)) \\ y'(t) &= y(t)(-4+x(t)) \end{cases}$. Ce genre de modélisation est connu sous le nom de système de Lotka-Volterra, les plus curieux iront se renseigner pour mieux comprendre le modèle (en gros, l'idée est que la population de lapins est naturellement croissante en l'absence de renards, mais diminue proportionnellement au nombre de prédateurs, et qu'au contraire la population de renards est naturellement décroissante en l'absence de lapins et augmente proportionnellement à la présence de proies).
Il n'est pas nécessaire de changer de méthode sous prétexte qu'on a désormais un système d'équations à résoudre, le principe de la méthode d'Euler reste tout à fait valable. La seule différence sera la nécessité de décrire le système d'équations par deux fonctions à trois variables $F(t,x,y)$ et $G(t,x,y)$ telles que $x'(t) = F(t,x(t),y(t))$ et $y'(t) = G(t,x(t),y(t))$. Dans notre exemple, on aura ainsi $F(t,x,y) = x*(3-2y)$ et $G(t,x,y) = y*(x-4)$. Les deux fonctions sont ici indépendantes de t, mais on peut très bien imaginer d'autres exemples où ce ne sera pas le cas. Notons par ailleurs que notre système n'est pas linéaire, ce qui ne pose aucun problème. Seule autre petite différence dans notre programme : il faudra choisir ce qu'on souhaite représenter graphiquement à l'issue de notre résolution approchée. Les choix les plus classiques consistent à représenter simultanément les deux fonction x et y en fonction du temps, où à représenter dans un même repère x et y (chacune sur un axe, en faisant disparaitre le temps), ce qui correspond à la notion de portrait de phase en physique (nous reviendrons dessus au paragraphe suivant).

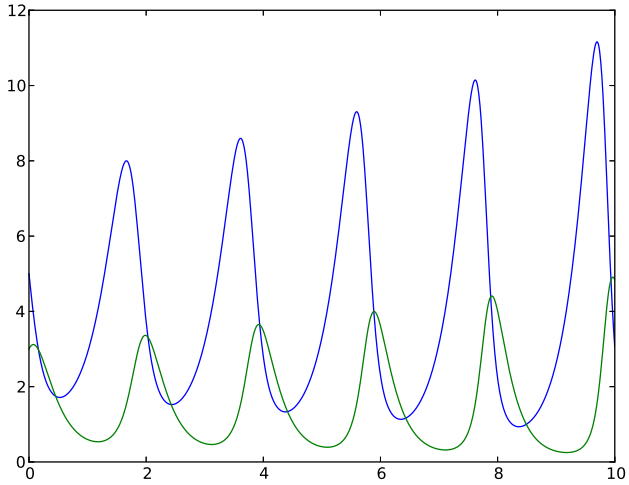
Programme Python pour la méthode d'Euler à deux fonctions inconnues :

```
> import matplotlib.pyplot as plt
> def Euler2(F,G,t0,tf,x0,y0,n) :
>         t=t0
>         x=x0
>         y=y0
>         h=(tf-t0)/float(n)
```

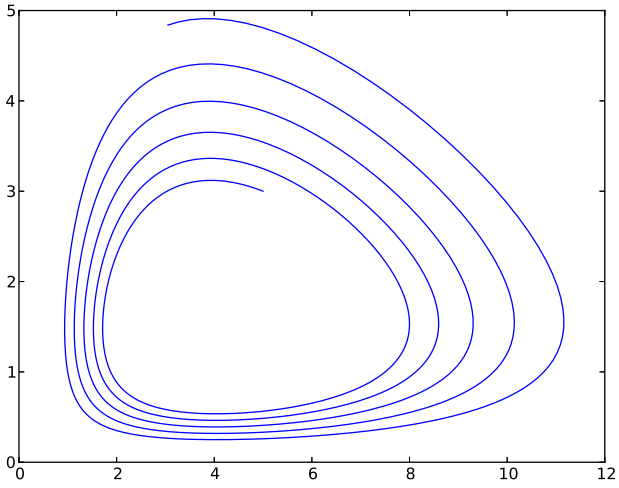
```
>       temps=[t0]
>       lapins=[x0]
>       renards=[y0]
>       for i in range(n) :
>           x,y=x+h*F(t,x,y),y+h*G(t,x,y)
>           t=t+h
>           temps.append(t)
>           lapins.append(x)
>           renards.append(y)
>       plt.plot(temps,lapins)
>       plt.plot(temps,renards)
>       return lapins,renards
```

Remarque 7. On peut aisément généraliser le principe à un système contenant plus de deux équations. Dans ce cas, il deviendra utile de représenter l'ensemble des fonctions inconnues par une variable de type liste contenant les valeurs de toutes les fonctions, et l'équation par un unique fonction vectorielle F.

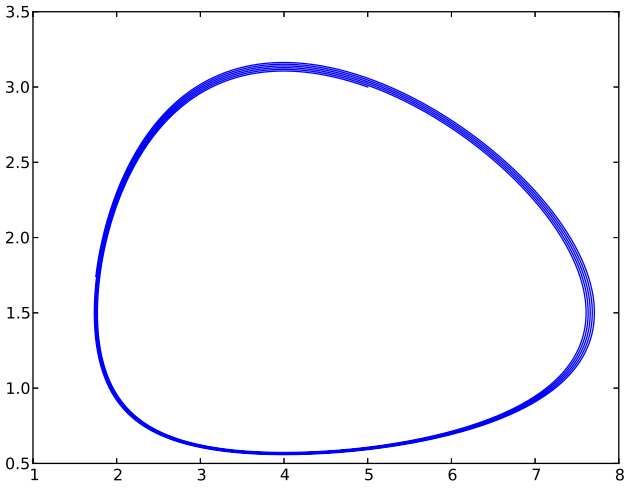
En faisant tourner notre programme avec les fonctions F et G définies précédemment, et des paramètres t0= 0, tf= 10, x0= 5, y0= 3 et n=1000, on obtient les courbes suivantes :



Les lapins sont en bleu et les renards en vert. On constate que les deux courbes sont pratiquement périodiques (en fait, si on effectue une résolution exacte, elles sont vraiment périodiques), avec un léger décalage temporel entre l'évolution des lapins et celle des renards (ce qui est logique : la population de lapins augmente quand le nombre de renards est bas, ce qui fait augment ensuite le nombre de renards, jusqu'à ce qu'il en y ait trop et que ça fasse baiosser le nombre de lapins, baisse qui influence à son tour l'évolution des renards etc). Si on souhaite visualiser le portait de phase correspondant, on remplace les deux plot à la fin du programme par un plt.plot(lapins,renards), et on obtient ceci :

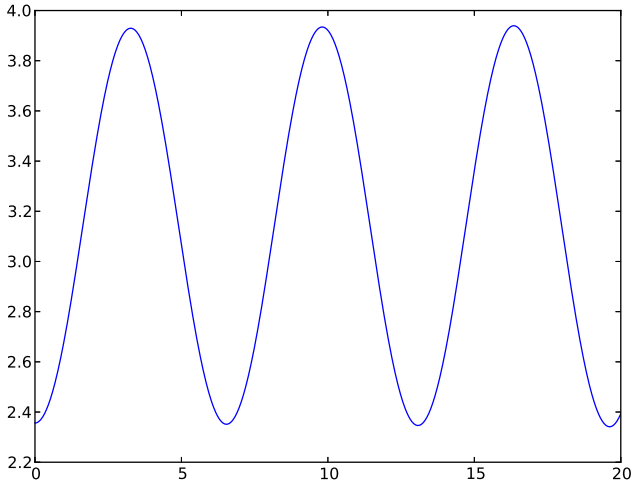


On observe là aussi la quasi-périodicité des courbes. La trajectoire obtenue « tourne » autour d'un point d'équilibre situé à $x = 4$ et $y = \dfrac{3}{2}$ (ce sont les valeurs pour lesquelles les deux dérivées s'annulent dans notre système), avec une distance au point d'équilibre qui dépend des conditions initiales choisies. Pour mieux observer la périodicité, augmentons la valeur de n à 20000 :

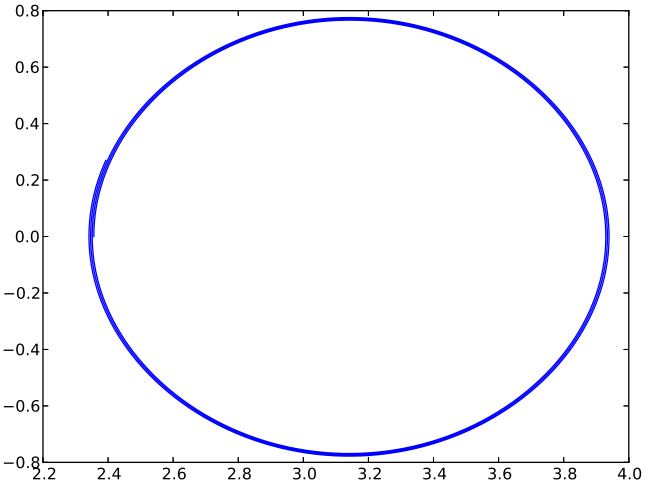


3.3 Équations du deuxième ordre.

Revenons à notre problème initial, celui du pendule simple. À l'aide d'une résolution de système comme nous venons de les étudier, on peut en fait résoudre par la méthode d'Euler n'importe quelle équation du deuxième ordre. Il suffit de penser à prendre comme fonctions inconnues les deux fonctions $y(t)$ et $y'(t)$ et de constater que l'équation du deuxième ordre peut s'écrire $y''(t) = F(t, y(t), y'(t))$. Pour reprendre les notations du paragraphe précédent, posons $x(t) = \theta(t)$ et $y(t) = \dot{\theta}(t)$, l'équation du pendule simple $\ddot{\theta}(t) = \sin(\theta(t))$ est alors équivalente au système de deux équations $\begin{cases} x'(t) &= & y(t) \\ y'(t) &= & \sin(x(t)) \end{cases}$. En faisant tourner notre programme **Euler2** avec les conditions initiales x0= $\dfrac{3\pi}{4}$ et y0= 0 (vitesse initiale nulle), et pour t0= 0, tf= 20 et $n = 10000$, on obtient la courbe suivante (ici, on ne trace que y en fonction de t, la courbe de y' n'ayant que peu d'intérêt) :



La courbe est joliment pseudo-périodique, avec tout de même une curieuse tendance à avoir une amplitude qui augmente au cours du temps. Si on trace le portrait de phase $(y(t), y'(t))$, on a quelque chose de ce genre :



Le résultat est très convaincant, la symétrie est évidemment normal, avec un centre en $(\pi, 0)$. La vitesse est maximale quand l'angle est nul, et nulle quand l'angle est extrême, ce qui est physiquement normal.

3.4 Déjà disponible en Python.

Comme nous l'avons déjà signalé à la section précédente, tout se trouve dans le module **scipy.integrate**. En fait, c'est très simple puisque l'unique fonction **odeint** sert à résoudre toutes les équations différentielles, et fonctionne sur le même mode que notre Euler : on lui donne une fonction f (éventuellement vectorielle) représentant l'équation (ou le système), une condition initiale y0 (qui sera un vecteur dans le cas d'un système), et un tableau de nombres correspondant aux valeurs t_i pour lesquelles on souhaite calculer une valeur approchée de $y(t_i)$, et la fonction ressort un tableau de valeurs qu'on peut évidemment représenter graphiquement si on le souhaite.