1. Résolution d'équations algébriques ou transcendantes

1.1 Dichotomie

```
def dicho(f,a,b,epsilon):
    while b-a > epsilon:
        c = (a + b)/2
        if f(a)*f(c) < 0:
            b = c
        else:
            a = c
    return (a+b)/2
```

f désigne une fonction continue et monotone changeant de signe sur un intervalle [*a*, *b*].

Exemple : **def** f(x): ou f = **lambda** x: x**2-2
return x**2-2

dicho(f,1,2,10**-12) renvoie une approximation de $\sqrt{2}$ à 10^{-12} près.
dicho(numpy.sin,3,4,10**-12) renvoie une approximation de π.

Remarques

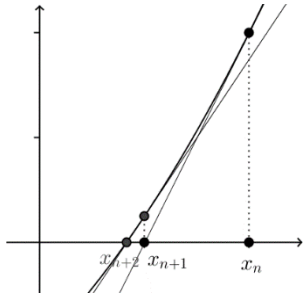
1) Le choix de ϵ doit être adapté à chaque situation.
En effet, supposons que l'on fixe ϵ à 10^{-12}, et que *x* prenne des valeurs de l'ordre 10^5 ou plus, Python qui calcule en décimal avec 16 chiffres significatifs ne sera pas assez précis.

A l'opposé, si *x* prend des valeurs de l'ordre de 10^{-10}, nous n'obtiendrons que 2 chiffres significatifs.

2) Soit *n* le nombre d'exécution de la boucle nécessaires pour atteindre la précision ϵ souhaitée. Alors : $\frac{b-a}{2^n} \leq \epsilon \Leftrightarrow n \geq \frac{\ln(b-a)-\ln\epsilon}{\ln 2}$
On peut donc prévoir le nombre d'exécutions théoriques (en pratique, il vaut mieux prendre *n* un peu plus grand que le nombre théorique à cause des erreurs d'arrondis).

On pourra éviter les boucles infinies en ajoutant un test sur le nombre d'exécutions de la boucle.

1.2 Méthode de Newton



Soit une fonction *f* de classe C^2 sur un intervalle ouvert *I* et α ∈ I tel que *f*(α) = 0 et *f'*(α) ≠ 0.

Soit la suite (x_n) définie par $x_0 \in$ I et pour tout $n \in \mathbb{N}$, $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.

D'après la formule de Taylor-Lagrange, il existe $c \in$ I tel que

$$f(\alpha) = f(x_n) + f'(x_n)(\alpha - x_n) + \frac{1}{2}f''(c)(\alpha - x_n)^2,$$

ce qui revient à : $\alpha - x_{n+1} = -\frac{f''(c)}{2f'(x_n)}(\alpha - x_n)^2$

On en déduit deux conditions suffisantes de convergence.

CS1 : S'il existe deux réels *M* et *m* tels que sur *I*, $|f''| < M$ et $|f'| > m$, alors pour tout *n* fixé, $|\alpha - x_{n+1}| \leq \frac{M}{2m}(\alpha - x_n)^2$.
Ainsi, si $|\alpha - x_0| < \frac{2m}{M}$, alors la suite $(x_n)_{n\in\mathbb{N}}$ est bien définie et converge vers 0.
Cette condition sera toujours vérifiée si x_0 est suffisamment proche de α.

CS2 : Si *f* est convexe sur *I* = [α, *b*] et si $f'(\alpha) > 0$, alors si $x_n \in I$, on a :

$$f'(x_n) > 0 \text{ et } \alpha \leq x_{n+1} < x_n$$

La suite (x_n) est donc bien définie, décroissante et minorée donc convergente vers $l \in I$ vérifiant $l = l - \frac{f(l)}{f'(l)}$, donc *l* = α.
Si $f'(\alpha) < 0$, on travaille sur *I* = [*a*, α].

Enoncés similaires avec *f* concave.

```
def newton(f,df,x,n):
    for i in range(n):
        x = x - f(x)/df(x)
    return x
```

```
def newton2(f,df,x,epsilon):
    y = x - f(x)/df(x)
    while abs(y-x) > epsilon:
        x,y = y,y - f(y)/df(y)
    return y
```

Exemples :

- newton(numpy.sin,numpy.cos,3,3) donne 15 décimales de π.
- newton(lambda x: x*x-1-1j,lambda x: 2*x,2,10) nous montre que la méthode de Newton fonctionne également avec des nombres complexes ! L'ensemble des valeurs initiales x_0 telles que la suite (x_n) converge vers la racine α est appelé bassin d'attraction et est souvent une fractale.
- la méthode de Newton appliquée à $x \mapsto x^2 - a$ n'est autre que le célèbre algorithme de Babylone, aussi connu comme méthode de Héron.

Avec estimation de la dérivée de *f*

```
def newtonh(f,x,h,n):
    for i in range(n):
        df = (f(x+h) - f(x-h))/(2*h)
        x = x - f(x)/df
    return x
```

Question : Pourquoi utilise-t-on $\frac{f(x+h)-f(x-h)}{2h}$ plutôt que $\frac{f(x+h)-f(x)}{h}$?

Exemple : newtonh(numpy.sin,3,0.001,3) donne 15 décimales de π.

1.3 Vitesse de convergence

Soit α une racine d'une fonction *f*. On suppose que la suite des valeurs (x_n) obtenues par dichotomie (en prenant le centre de chaque intervalle) ou suivant la méthode de Newton converge vers α.

Dichotomie : $|\alpha - x_n| \leq |b_n - a_n|$

La suite des longueurs des intervalles vérifie $|b_{n+1} - a_{n+1}| \leq \frac{|b_n - a_n|}{2}$.

On parle de convergence linéaire.

Méthode de Newton : $|\alpha - x_{n+1}| \leq \frac{M}{2m}|\alpha - x_n|^2$

La convergence est quadratique.

La méthode de Newton converge beaucoup plus vite… lorsqu'elle converge.

1.4 Utilisation des bibliothèques numpy et scipy

Les méthodes de dichotomie et de Newton (déjà programmées, testées et optimisées) se trouvent dans la bibliothèque scipy.optimize.

Dichotomie scipy.optimize.bisect(*f*, *a*, *b*)

```
In [2]: import numpy as np
In [2]: import scipy.optimize as sco
In [3]: sco.bisect(np.sin,3,4)
Out[3]: 3.141592653589214
```

On peut préciser d'autres paramètres : nombre maximum d'itérations, epsilon (xtol), …

```
In [4]: sco.bisect(numpy.sin,3,4,xtol=1e-15)
Out[4]: 3.141592653589793
```

Méthode de Newton scipy.optimize.newton(*f*, x_0, *f'*)

```
In [5]: sco.newton(np.sin,3,np.cos)
Out[5]: 3.1415926535897931
```

Sans la donnée de f', cette fonction utilise la méthode de la sécante :

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n)$$

Avec la donnée de f' et de f'', la fonction utilise la méthode de Halley (convergence cubique). On peut aussi modifier la valeur de epsilon (tol).

```
In [6]: sco.newton(lambda x: x**2-2,2,lambda x: 2*x,fprime2=lambda x: 2)
Out[6]: 1.4142135623730951
```

2. Résolution d'équations différentielles

2.1 Tracé de courbes

Dans ce paragraphe, nous utiliserons les bibliothèques matplotlib.pyplot et numpy. Les puristes préfèreront importer les deux afin de bien discerner la source des différentes fonctions utilisées avec :

```
import matplotlib.pyplot as plt
import numpy as np
```

Vous pouvez aussi utiliser pylab qui combine les fonctions de pyplot et numpy : import pylab as pl

plot([x_1, x_2,…, x_n],[y_1, y_2,…, y_n]) crée automatiquement la fenêtre graphique avec les axes et relie les points de coordonnées (x_i, y_i).

Python fixe par défaut la fenêtre graphique de sorte à ce qu'elle contienne tous les points, mais on peut la modifier avec plt.xlim([x_{min}, x_{max}]) et plt.ylim([y_{min}, y_{max}]).

Pour construire la courbe d'une fonction *f*, on crée deux listes ou tableaux (array) [x_1, …, x_n] et [y_1, …, y_n] tels que $y_i = f(x_i)$.

Le type array est défini dans la bibliothèque numpy. Les arrays ressemblent aux listes mais sont adaptés aux calculs.

```
In [1]: [1,2,3]*2                 →   Out[1]: [1, 2, 3, 1, 2, 3]
In [2]: np.array([1,2,3])*2       →   Out[2]: array([2, 4, 6])
```

Pour créer une subdivision régulière [x_1, …, x_n] d'un segment [a, b], on peut utiliser linspace(a, b, n) ou arange(a, b+h, h), fonctions de la bibliothèque numpy, qui renvoient un array. Python offre plusieurs possibilités pour construire [y_1, …, y_n], les boucles, les listes en compréhension, les fonctions vectorielles ou la fonction map(*f*, X).

```
In [3]: list(map(lambda x: x*x,range(11)))
Out[3]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Les fonctions vectorielles sont des fonctions s'appliquant aux arrays comme la fonction lambda x: x*x (car l'opération * est redéfinie pour les arrays).

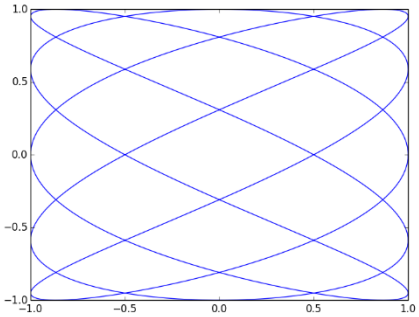
Les fonctions importées de numpy sont vectorielles. On obtient donc la courbe de la fonction cosinus sur [0, 6π] avec

```
In [4]: X=np.linspace(0,6*np.pi,100)
In [5]: plt.plot(X,np.cos(X))
```

La fonction math.cos n'est pas vectorielle et ne fonctionnerait pas ici.

La fonction np.vectorize(*f*) envoie une version vectorielle de la fonction *f*.

plot permet de tracer simplement des courbes paramétrées, comme la figure de Lissajous ci-contre.




```
In [6]: T=np.linspace(0,2*np.pi,500)
In [7]: X=np.sin(5*T)
In [8]: Y=np.cos(3*T)
In [9]: plt.plot(X,Y)
```

Sous certains environnements, l'instruction plot affiche directement le graphique, sous d'autres il faut entrer plt.show().

2.2 Méthode d'Euler

La méthode d'Euler fournit dans de nombreux cas une solution approchée à un problème de Cauchy de la forme $\begin{cases} y' = f(x,y) \\ y(a) = y_0 \end{cases}$ où y est une fonction dérivable de la variable x et f une fonction définie sur une partie de $\mathbb{R}^2$.

Le cours de première année nous garantit l'existence et l'unicité d'une solution dans le cas linéaire ($y' = A(x)y + B(x)$, avec A et B continues), mais la méthode d'Euler s'applique surtout dans de nombreux cas où on ne sait pas déterminer de solution analytique. Nous pouvons citer le théorème de Cauchy-Lipschitz (qui dépasse le cadre de notre programme mais un peu de culture mathématique n'a jamais fait de mal).

<u>Théorème</u>

Si f est une fonction continue de Ω (ouvert de $\mathbb{R}^2$) dans $\mathbb{R}$ et lipschitzienne par rapport à la deuxième variable, et si $(a, y_0) \in \Omega$, alors le problème de Cauchy $\begin{cases} y' = f(x,y) \\ y(a) = y_0 \end{cases}$ admet une solution maximale unique.

<u>Conséquences</u>

- Si f est de classe C^p, alors les solutions sont de classe C^{p+1}.
- Les solutions maximales forment une partition de Ω.

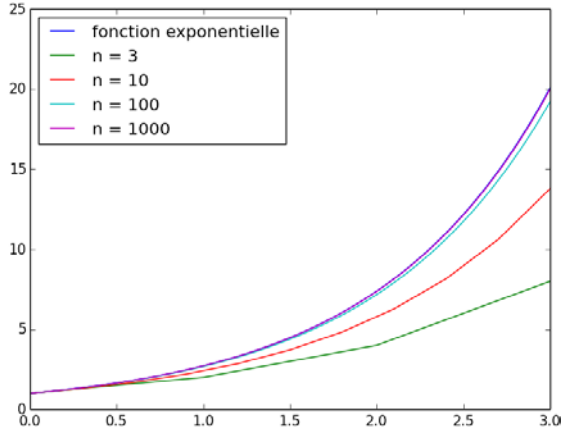
On considère un problème de Cauchy admettant une solution y définie sur un intervalle $[a, b]$ vérifiant $y(a) = y_0$. La méthode d'Euler consiste à construire une solution approchée de l'équation différentielle point par point. On démarre avec le premier point (a, y_0), la condition initiale.
On détermine une subdivision $(x_k)_{0\leq k\leq n}$ de $[a, b]$, et on construit de proche en proche les images y_k de chaque nombre x_k de cette subdivision.
Pour cela, connaissant y_k, on détermine y_{k+1} par $y_{k+1} = y_k + y'_k(x_{k+1} - x_k)$, soit $y_{k+1} = y_k + y'_k \times h$, si h est le pas de la subdivision, avec $y'_k = f(x_k, y_k)$, ce qui revient à approcher la courbe par sa tangente au point d'abscisse x_k. L'équation différentielle permet de déterminer y'_k.
La représentation graphique de la fonction cherchée s'obtient alors en reliant les points de coordonnées $(x_k, y_k)_{0\leq k\leq n}$.

```
def euler(f,a,b,y,n):
    X=np.linspace(a,b,n+1)
    h=(b-a)/n
    Y=[y]
    for k in range(n):
        y += h*f(X[k],y)
        Y.append(y)
    plt.plot(X,Y)
```

f est une fonction de deux variables, y est la valeur initiale $y(a)$ et n le nombre de segments de l'approximation affine.
On pourra choisir le pas h à la place de n et remplacer plt.plot(X,Y) par return X,Y selon les besoins.

Exemple : $\begin{cases} y' = y \\ y(0) = 1 \end{cases}$

Le graphique ci-contre donne les solutions obtenues sur [0,3] par la méthode d'Euler pour différentes valeurs de n.



2.3 Majoration de l'erreur (compléments)

On appelle erreur de consistance au pas k l'erreur commise par une itération de la méthode en supposant les valeurs précédemment calculées exactes. Ainsi, l'erreur de consistance au pas k de la méthode d'Euler est :
$\delta_{k,h} = y(x_{k+1}) - [y(x_k) + hf(x_k, y(x_k))] = y(x_{k+1}) - y(x_k) - hy'(x_k)$
$\delta_{k,h} = h^2y''(x_k) + o(h^2) = O(h^2)$
(d'après Taylor Young en supposant y de classe C^2)
On dit que la méthode d'Euler est d'ordre 1, ce qui n'est pas très bon. D'autres méthodes sont plus efficaces, dont la méthode de Runge-Kutta d'ordre 4 (erreur de consistance en $O(h^5)$).

En pratique, on peut estimer l'erreur en calculant $\max_{0\leq k\leq n} |y(k) - y_k|$ par exemple (il ne s'agit plus de l'erreur de consistance).

Exemple : dans l'exemple précédent (fonction exponentielle), on obtient :
- n = 10 000 → temps de calcul : 0,01 s, erreur : 0,009
- n = 100 000 → temps de calcul : 0,07 s, erreur : 0,000 9
- n = 1 000 000 → temps de calcul : 0,73 s, erreur : 0,000 09

2.4 Utilisation des bibliothèques numpy et scipy

La fonction odeint, pour ordinary differential equation integrate, à importer de la bibliothèque scipy.integrate, résout numériquement les problèmes de Cauchy du type $\begin{cases} y' = f(y, x) \\ y(a) = y_0 \end{cases}$.

- f est à définir sous la forme $f(y, x)$: la variable d'intégration est le deuxième argument
- t est un tableau des valeurs de la variable x (une subdivision de $[a, b]$ en principe)

odeint(f, y_0, t) renvoie alors le tableau des images des nombres présents dans le tableau t.

```
In [1]: from scipy.integrate import odeint
In [2]: odeint(lambda y,x: y,1,[0,0.5,1])
In [3]: odeint(lambda y,x: y,1,pl.linspace(0,3,100))
```

http://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html

3. Calcul d'intégrales

3.1 Lien avec la méthode d'Euler

Soit f une fonction continue sur un segment $[a, b]$ et F une primitive de f sur $[a, b]$. On considère le problème de Cauchy : $\begin{cases} y' = f(x) \\ y(a) = F(a) \end{cases}$

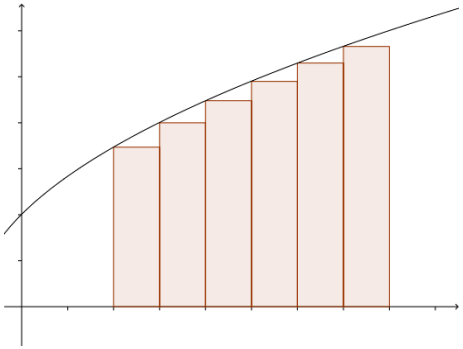
La solution y de ce problème est la fonction F. En admettant que la méthode d'Euler converge (ce qui se démontre), le dernier terme calculé y_n est une approximation de $F(b)$. Or, $y_n = F(a) + \sum_{k=1}^{n} h \times f(x_k)$.

Ainsi, $\sum_{k=1}^{n} h \times f(x_k) = y_n - F(a)$ donne une approximation de $F(b) - F(a)$.

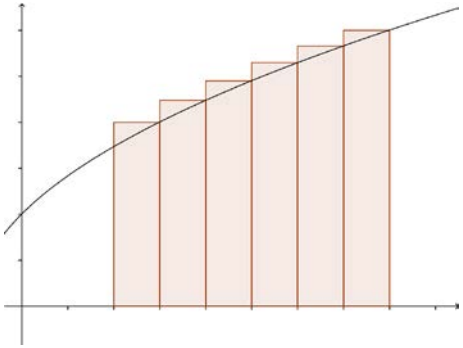
On retrouve que $\frac{b-a}{n}\sum_{k=1}^{n} f\left(a + \frac{k(b-a)}{n}\right)$ converge vers $\int_a^b f(t)dt$.

3.2 Méthode de calculs de valeurs approchées d'intégrales

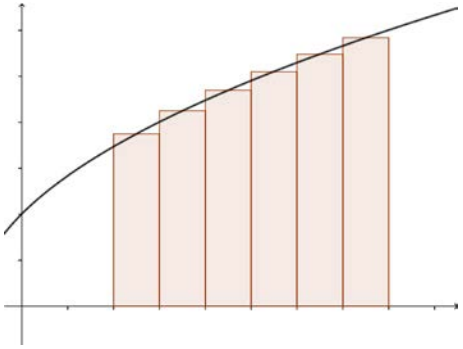
Les méthodes consistent à approcher f par une fonction en escalier (méthodes des rectangles) ou affine par morceaux (méthode des trapèzes).



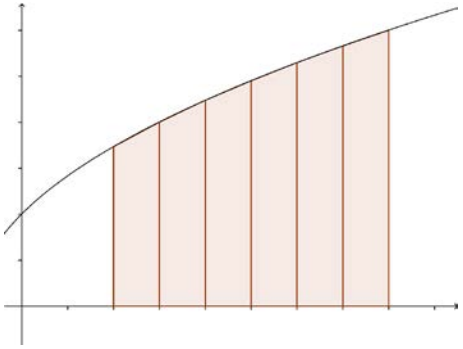
Méthode des rectangles à gauche



Méthode des rectangles à droite



Méthode du point médian



Méthode des trapèzes

On cherche à approcher l'intégrale A d'une fonction f continue sur un intervalle $[a, b]$. Pour $n \in \mathbb{N}^*$ (nombre de rectangles), on pose $h = \frac{b-a}{n}$.

Méthode des rectangles à gauche : $R_n^g = h\sum_{k=0}^{n-1} f(a + kh)$

Méthode des rectangles à droite : $R_n^d = h\sum_{k=1}^{n} f(a + kh)$

Méthode du point médian : $R_n^m = h\sum_{k=0}^{n-1} f\left(a + \frac{h}{2} + kh\right)$

Méthode des trapèzes : $T_n = h\sum_{k=0}^{n-1} \frac{f(a+kh)+f(a+h+kh)}{2} = \frac{R_n^g + R_n^d}{2}$

$$T_n = h\left(\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f(a + kh)\right)$$

```
def rectangleg(f,a,b,n):
    somme = 0
    h = (b-a)/n
    x = a
    for k in range(n):
        somme += f(x)
        x += h
    somme *= h
    return somme
```

Exercice : Ecrire les fonctions rectanglesd(f,a,b,n), rectanglem(f,a,b,n) et trapeze(f,a,b,n) calculant une approximation de $\int_a^b f(t)\mathrm{d}t$ suivant les méthodes des rectangles à droite, du point médian, des trapèzes.

3.3 Majoration de l'erreur (compléments)

Si f est C^1 sur $[a, b]$ et $m_1 = \sup_{[a,b]} |f'|$, alors :

$|A - R_n^g| \leq \frac{m(b-a)^2}{2n}$ et $|A - R_n^d| \leq \frac{m_1(b-a)^2}{2n}$ (méthodes d'ordre 1)

Si f est C^2 sur $[a, b]$ et $m_2 = \sup_{[a,b]} |f''|$, alors :

$|A - R_n^m| \leq \frac{m_2(b-a)^3}{24n^2}$ et $|A - T_n| \leq \frac{m_2(b-a)^3}{12n^2}$ (méthodes d'ordre 2)

3.4 Accélération de convergence de Romberg-Richardson

Principe : On admet que $T_n = A + \frac{c}{n^2} + o\left(\frac{1}{n^2}\right)$.

Alors, $T_{2n} = A + \frac{C}{4n^2} + o\left(\frac{1}{n^2}\right)$ et donc $S_n = \frac{4T_{2n} - T_n}{3} = A + o\left(\frac{1}{n^2}\right)$.
La suite S_n converge donc plus vite que la suite T_n.
L'algorithme obtenu est la méthode de Simpson. Elle est d'ordre 4.

3.5 Utilisation des bibliothèques numpy et scipy

Les fonctions permettant le calcul d'intégrales sont dans la bibliothèque scipy.integrate.

- quad(*f*, *a*, *b*) renvoie un tuple contenant la valeur approchée de $\int_a^b f(t)\mathrm{d}t$ et une majoration de l'erreur.

```
In [1]: from scipy.integrate import quad
In [2]: import numpy as np

In [3]: quad(np.cos,0,np.pi)
Out[3]: (4.9225526349740854e-17, 2.2102239425853306e-14)

In [4]: f=lambda t: quad(lambda x: np.exp(-t*x*x),0,np.inf)
```

- dblquad et tplquad permettent de calculer les intégrales doubles ou triples.

```
In [5]: from scipy.integrate import dblquad
In [6]: f=lambda x,y: np.exp(-x*x-y*y)
In [7]: dblquad(f,-np.inf,np.inf,lambda x: -np.inf,lambda x: np.inf)
Out[7]: (3.141592653589777, 1.474341858858635e-08)
In [8]: dblquad(lambda x,y: x+y,0,1,lambda x: 0,lambda x: 1-x)
Out[8]: (0.3333333333333333, 3.700743415417188e-15)
```

Les bornes de la première variable sont des nombres, les bornes de la deuxième variable des fonctions (de la première).

- trapz et simps effectuent des calculs approchés d'intégrales à partir d'un tableau de valeurs (liste ou array créé avec linspace, obtenu de manière expérimentale, par la fonction odeint…).

Syntaxe : trapz(*y*,*x*,d*x*) où *y* est le tableau des valeurs prises par *f*, *x* le tableau des valeurs prises par *x*, d*x* l'écart entre les valeurs (largeur des rectangles).On précise ou bien *x* ou bien d*x*. Par défaut, d*x* = 1.

Même syntaxe pour simps.

```
In [50]: import scipy.integrate as scii

In [51]: x=np.linspace(-1,1,101)

In [52]: scii.trapz(np.exp(-x*x),x)
Out[52]: 1.4935992143787031

In [53]: scii.simps(np.exp(-x*x),x)
Out[53]: 1.4936482682406362
```

http://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html

4. Complément : calcul formel avec Python

Le calcul formel n'est pas mentionné dans le programme d'informatique, mais peut être utile en mathématiques ou en sciences physique. On utilise le module sympy.

```
In [1]: import sympy as sp
In [2]: x=sp.Symbol('x')

In [3]: sp.expand((x+sp.Symbol('y'))**3)
Out[3]: x**3 + 3*x**2*y + 3*x*y**2 + y**3

In [4]: sp.limit((2*x+sp.exp(-x))/(3*x+1),x,sp.oo)        # sp.oo = +∞
Out[4]: 2/3

In [5]: sp.factor(sp.diff(x*sp.exp(x**2),x))
Out[5]: (2*x**2 + 1)*exp(x**2)

In [6]: sp.integrate(sp.log(x),x)
Out[6]: x*log(x) – x

In [7]: sp.integrate(sp.exp(-x**2),(x,0,sp.oo))
Out[7]: sqrt(pi)/2

In [8]: sp.solve(x**3+1,x)
Out[8]: [-1, 1/2 - sqrt(3)*I/2, 1/2 + sqrt(3)*I/2]
```

http://docs.sympy.org/latest/tutorial/index.html