In [1]:

```
from IPython.display import display, Latex
from IPython.core.display import HTML
css_file = './custom.css'
HTML(open(css_file, "r").read())
```

Out[1]:

M62 TP 1 - notebook IPython : calcul numérique

Vous pouvez commencer à exécuter un serveur de notebook en écrivant dans un terminal :

`jupyter notebook &`

Cela imprimera des informations dans votre console et ouvrira un navigateur Web.
La page d'arrivée est le tableau de bord qui affiche les notebook actuellement disponibles (par défaut, le répertoire à partir duquel le serveur du bloc-notes a été démarré).

Vous pouvez

- soit créer un nouveau notebook à partir du tableau de bord avec le bouton (en haut à droite) `Nouveau`
- soit ouvrir un notebook existant en cliquant sur leur nom (par exemple, vous pouvez télécharger le notebook de présentation ici (http://nbviewer.jupyter.org/url/faccanoni.univ-tln.fr/user/enseignements/20182019/M62-CM1.ipynb).)

Pour ce TP, on importera tous les modules necessaires comme suit:

In [2]:

```
%reset -f
%matplotlib inline
%autosave 300

from math import *
from random import *
from matplotlib.pylab import *
```

```
Autosaving every 300 seconds
```

In [3]:

```
# Matplotlib can be configured through global settings.
# The pyplot interface to the runtime interface is through pyl
# The common aspects I override are:
rcParams['font.family'] = 'serif'
# rcParams['font.serif'] = 'Ubuntu'
# rcParams['font.monospace'] = 'Ubuntu Mono'
rcParams['font.size'] = 12
rcParams['axes.labelsize'] = 12
# rcParams['axes.labelweight'] = 'bold'
rcParams['xtick.labelsize'] = 10
rcParams['ytick.labelsize'] = 10
rcParams['legend.fontsize'] = 12
rcParams['figure.titlesize'] = 14
```

Table of Contents

1 Exercice : apprendre à écrire

en markdown et LaTeX

1. Écrire les énoncés des exercices (une cellule par exercice). Chaque exercice commence par un titre de niveau 2.
2. Après chaque énoncé ajouter une cellule qui ne contient que le mot **Correction** écrit en gras (ce n'est pas un titre).
3. Après chaque cellule **Correction** écrire la résolution analytique des exercices (lorsque cela est possible).
4. Décrire brièvement le code que vous allez écrire, ajouter le code python et l'éxecuter.

2 Exercice : suites

Soit les suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ définies par

$$\begin{cases} u_0 = v_0 = 1, \\ u_{n+1} = u_n + v_n, \\ v_{n+1} = 2u_n - v_n. \end{cases}$$

Calculer u_{100} et v_{100}.

Correction

Calcule explicite du 100-ème terme:

In [4]:

```
u,v = 1,1
for n in range(100):
    (u,v)=(u+v,2.*u-v)
print("u_100 =",u," v_100 =",v)
```

```
u_100 = 7.178979876918526e+23  v_100 = 7.17897987
6918526e+23
```

3 Exercice : suites et arret

Soit $(u_n)_{n\in\mathbb{N}}$ la suite définie par $u_n = (0.7)^{3n}$. Quel est le plus petit n tel que $u_n < 10^{-4}$?

Correction

$u_n = \left(\left(\frac{7}{10}\right)^3\right)^n = \left(\frac{343}{1000}\right)^n$. Il s'agit d'une suite géométrique de raison $0 < q < 1$: elle est donc décroissante. On a

$$u_n < 10^{-4} \iff \left(\frac{343}{1000}\right)^n < 10^{-4}$$
$$\iff \log_{10}\left(\frac{343}{1000}\right)^n < -4$$
$$\iff n > -\frac{4}{\log_{10}\left(\frac{343}{1000}\right)} = -\frac{4}{\log_{10}(343) - \log_{10}(10^3)} =$$

La valeur cherchée est donc $n = 9$.

Vérifions nos calculs:

In [5]:

```
n=0
u=1
while u>=1.e-4:
    n+=1
    u=(0.7)**(3*n)
    print(n,u)
```

```
1 0.3429999999999999
2 0.11764899999999996
3 0.04035360699999998
4 0.01384128720099999
5 0.004747561509942996
6 0.001628413597910447
7 0.0005585458640832833
8 0.00019158123138056612
9 6.571236236353417e-05
```

4 Exercice : `function`

Fabriquer une fonction qui calcule le volume v d'un cylindre de révolution de hauteur h et dont la base est un disque de rayon r.

Correction

In [6]:

```
#def volume(h,r):
#    return r**2 * h * pi
# equivalent a
volume = lambda h,r : (r**2 * h * pi)
h,r=1,1
print("Le volume d'un cylindre de hauter %2.2f cm et rayon %2.
```

```
Le volume d'un cylindre de hauter 1.00 cm et rayo
n 1.00 cm est 3.141593 cm^2
```

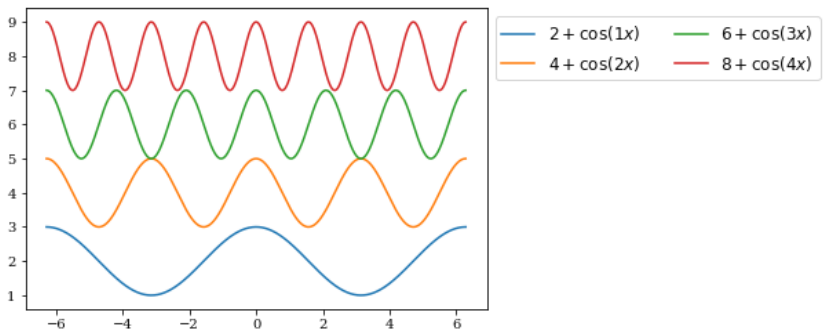
5 Exercice : afficher des graphes

Tracer dans le même repère le graphe des fonctions $f_n(x) = 2n + \cos(nx)$ pour $n = 1, 2, 3, 4$.

Correction

In [7]:

```
f = lambda x,n : 2*n+cos(n*x)
xx=linspace(-2*pi,2*pi,1001)
for n in range(1,5):
    yy=[f(x,n) for x in xx]
    plot(xx,yy, label=r'${}+\cos({}x)$'.format(2*n,n))
legend(bbox_to_anchor=(1, 1),ncol=2);
```



6 Exercice : calcul approché d'une dérivée

Soit $f(x) = \sin(x)$.

1. Écrire une lambda function `f(x)` qui évalue f en x.
2. Calculer f' et écrire une lambda function `df_exact(x)` qui évalue f' en x.

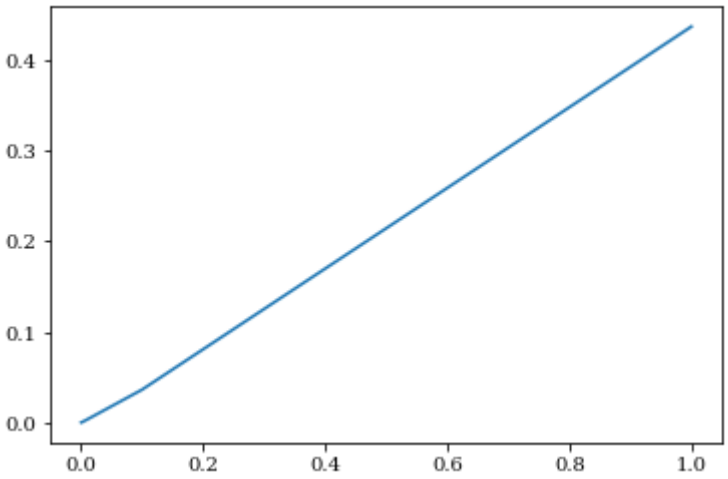
3. Écrire une lambda function `df_approx(x,h)` qui calcule $g(x,h) = \frac{f(x+h)-f(x)}{h}$.
4. Comparer `df_exact(pi/4)` et `df_approx(pi/4,h)` pour différentes valeurs de h.
5. Tracer un graphe avec en absisses h et on ordonnée la valeur absolue de l'erreur.

Correction

In [8]:

```
f         = lambda x  : sin(x)
df_exact  = lambda x  : cos(x)
df_approx = lambda x,h: (f(x+h)-f(x))/h

exact=df_exact(pi/4)
print("f'(pi/4)=",exact)

H=[10**(-i) for i in range(14)]
approx=[df_approx(pi/4,h) for h in H]
print(approx)

err=[abs(approxi-exact) for approxi in approx ]
plot(H,err);
```

```
f'(pi/4)= 0.7071067811865476
[0.2699544827129282, 0.6706029729039897, 0.703559
4916892096, 0.7067531099743674, 0.707071424669303
3, 0.7071032456451575, 0.7071064277441863, 0.7071
067453789937, 0.7071067842367995, 0.7071068175434
903, 0.70710770572191, 0.7071121466140085, 0.7071
010443837622, 0.7072120666862247]
```



7 Exercice : `function`

Comme π est la somme de la série

$$\pi = \sum_{n\in\mathbb{N}} 16^{-n}\left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6}\right)$$

on peut calculer une approximation de π en sommant les n premiers termes, pour n assez grand.

- Écrire une fonction pour calculer les sommes partielles de cette série.
- Pour quelles valeurs de n obtient-on une approximation de π aussi précise que celle fournie par la variable π?

Correction

Pour $n = 10$ on obtient une approximation de π qui coïncide (à la précision Python) avec la variable interne `math.pi` . Cet algorithme est en effet extrêmement efficace et permet le calcul rapide de centaines de chiffres significatifs de π.

In [9]:

```
def piapproche(N):
    approx=0
    for n in range(N):
        n8=8*n
        approx+=( 4/(n8+1) - 2/(n8+4) - 1/(n8+5) - 1/(n8+6) )*
    return approx

N=0
while abs(pi-piapproche(N))>0:
    N+=1

print("       N  = ",N)
print("pi approx = ",piapproche(N))
print(" math.pi  = ",pi)
```

```
       N  =  11
pi approx =  3.141592653589793
 math.pi  =  3.141592653589793
```

8 Exercice : liste de compréhension

Conversion des dégrées Celsius en dégrées Fahrenheit: une température de 0°C correspond à 32°F tandis que 100°C correspondent à 212°F. La formule permettant la conversion d'une valeur numérique x de la température en (°C) vers l'unité (°F) est affine:

$$y = \frac{9}{5}x + 32$$

Définir une liste de liste dont la première composante contient la valeur en Celsius (on considère des températures allant de 0°C à 100°C par paliers de 5°C) et la deuxième l'équivalente en Fahrenheit. Utiliser une liste de compréhension.

In [10]:

```
Cdegrees=range(0,101,5)
Fdegrees=[(9./5)*x+32 for x in Cdegrees]
for i in range(len(Cdegrees)):
    print('%5d %5.1f' % (Cdegrees[i], Fdegrees[i]))
```

```
    0  32.0
    5  41.0
   10  50.0
   15  59.0
   20  68.0
   25  77.0
   30  86.0
   35  95.0
   40 104.0
   45 113.0
   50 122.0
   55 131.0
   60 140.0
   65 149.0
   70 158.0
   75 167.0
   80 176.0
   85 185.0
   90 194.0
   95 203.0
  100 212.0
```

9 Exercice : liste de compréhension

Depuis l'ajustement du calendrier grégorien, l'année sera bissextile si l'année est divisible par 4 et non divisible par 100, ou est divisible par 400. Ainsi,

- 2019 n'est pas bissextile car non divisible
- 2008 était bissextile suivant la première règle (divisible par 4 et non divisible par 100)
- 1900 n'était pas bissextile car divisible par 4, mais aussi par 100 (première règle non respectée) et non divisible par 400 (seconde règle non respectée).
- 2000 était bissextile car divisible par 400.

Écrire la liste des années bissextiles entre l'année 1800 et l'année 2099.

In [11]:

```
print([b for b in range(1800,2100) if (b%4==0 and b%100!=0) or
```

```
[1804, 1808, 1812, 1816, 1820, 1824, 1828, 1832,
1836, 1840, 1844, 1848, 1852, 1856, 1860, 1864, 1
868, 1872, 1876, 1880, 1884, 1888, 1892, 1896, 19
04, 1908, 1912, 1916, 1920, 1924, 1928, 1932, 193
6, 1940, 1944, 1948, 1952, 1956, 1960, 1964, 196
8, 1972, 1976, 1980, 1984, 1988, 1992, 1996, 200
0, 2004, 2008, 2012, 2016, 2020, 2024, 2028, 203
2, 2036, 2040, 2044, 2048, 2052, 2056, 2060, 206
4, 2068, 2072, 2076, 2080, 2084, 2088, 2092, 209
6]
```

10 Exercice : visualiser une limite

On considère l'algorithme suivant pour calculer π: on génère n couples $\{(x_k, y_k)\}$ de nombres aléatoires dans l'intervalle $[0, 1]$, puis on calcule le nombre m de ceux qui se trouvent dans le premier quart du cercle unité:

$$\pi = \lim_{n \to +\infty} 4\frac{m}{n}.$$

Écrire un programme pour calculer cette suite et observer comment évolue l'erreur quand n augmente.

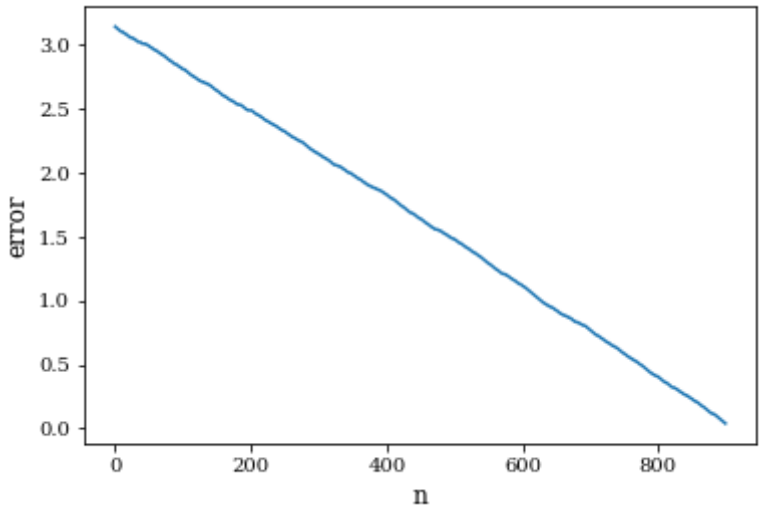
Correction

La commande `random()` du module `random` génère une suite de nombres pseudo-aléatoires. On exécute le programme pour différentes valeurs de n. Plus n est grand, meilleure est l'approximation de π. Par exemple, pour $n = 1000$ on obtient ≈ 3.1120, tandis qu'avec $n = 300000$ on a ≈ 3.1406 (naturellement, comme les nombres sont générés aléatoirement, les résultats obtenus pour une même valeur de n peuvent changer à chaque exécution).

In [12]:

```
n=900
tirage=range(n)
err=[]
m=0
for i in tirage:
    x,y = random(),random()
    if (x**2+y**2<=1):
        m+=1
    err.append(abs(pi-4*m/n))
print("pi ~=", 4*m/n)
plot(tirage,err)
xlabel('n')
ylabel('error')
show()
```

```
pi ~= 3.102222222222222
```



11 Exercice : visualisation d'une suite et test d'arret

On achète un ordinateur portable à 430 €. On estime qu'une fois sorti du magasin sa valeur u_n en euro après n mois est donnée par la formule

$$u_n = 40 + 300 \times (0.95)^n.$$

1. Que vaut l'ordinateur à la sortie du magasin?
2. Que vaut après un an de l'achat?
3. À long terme, à quel prix peut-on espérer revendre cet ordinateur?
4. Déterminer le mois à partir duquel l'ordinateur aura une valeur inférieure à 100 €.

Correction

- À la sortie du magasin $u_0 = 340$
- Après un an de l'achat on a $u_{12} = 40 + 300 \times (0.95)^{12} = 202.11$

- À long terme, on peut espérer revendre cet ordinateur à $\lim_{n\to+\infty} u_n = 40$.
- À partir du 32-ème mois l'ordinateur aura une valeur inférieure à 100 € car:

$$40 + 300 \times (0.95)^n < 100 \iff (0.95)^n < \frac{100-40}{300} = \frac{1}{5} = 5^-$$

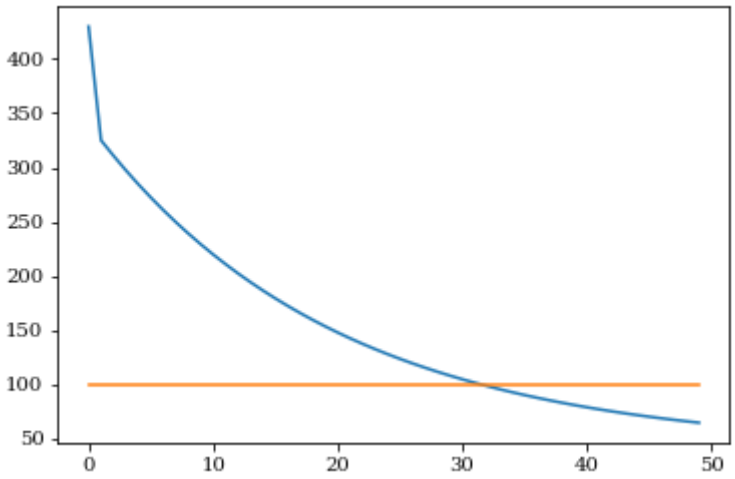
$$-\frac{\ln(5)}{\ln(0.95)} \simeq 31.377$$

Vérifions nos calculs:

In [13]:

```
nn=[0]
uu=[430]
for n in range(1,50):
    nn.append(n)
    uu.append(40+300*0.95**n)
plot(nn,uu, nn,[100 for n in nn])
indice=([u>100 for u in uu]).count(True)
print(indice)
print(uu[indice-1],uu[indice])
```

```
32
101.17204772373711 98.11344533755026
```



12 Exercice : cadeaux

Vous recevrez un cadeau de 1 centime aujourd'hui. Demain, vous recevrez le double (2 centimes). Le lendemain, vous recevrez à nouveau le double (4 centimes). Etc. Une fois que vous aurez reçu plus de 1 million d'euros, vos cadeaux cesseront.
Écrivez le code pour déterminer combien de jours vous recevrez des cadeaux, combien sera le dernier cadeau et combien vous recevrez au total.

Corection

In [14]:

```
total, cadeau, jour = 0, .01, 1
while (True):
    total += cadeau
    if total >= 1000000:
        break
    cadeau *= 2
    jour += 1

print('Le {} jour, mon cadeau est de {:,}€ ainsi la somme tota
    jour, cadeau, total))
```

```
Le 27 jour, mon cadeau est de 671,088.64€ ainsi l
a somme totale est 1,342,177.27€.
```

13 Exercice : coïncidences et anniversaires

Combien faut-il réunir de personne pour avoir une chance sur deux que deux d'entre elles aient le même anniversaire?

Au lieu de nous intéresser à la probabilité que cet événement se produise, on va plutôt s'intéresser à l'événement inverse: quelle est la probabilité pour que n personnes n'aient pas d'anniversaire en commun (on va oublier les années bissextiles et le fait que plus d'enfants naissent neuf mois après le premier de l'an que neuf mois après la Toussaint.)

- si $n = 1$ la probabilité est 1 (100%): puisqu'il n'y a qu'une personne dans la salle, il y a 1 chance sur 1 pour qu'elle n'ait pas son anniversaire en commun avec quelqu'un d'autre dans la salle (puisque, fatalement, elle est toute seule dans la salle);
- si $n = 2$ la probabilité est $\frac{364}{365}$ $(= 99{,}73\%)$: la deuxième personne qui entre dans la salle a 364 chances sur 365 pour qu'elle n'ait pas son anniversaire en commun avec la seule autre personne dans la salle;
- si $n = 3$ la probabilité est $\frac{364}{365} \times \frac{363}{365}$ $(= 99{,}18\%)$: la troisième personne qui entre dans la salle a 363 chances sur 365 pour qu'elle n'ait pas son anniversaire en commun avec les deux autres personnes dans la salle mais cela sachant que les deux premiers n'ont pas le même anniversaire non plus, puisque la probabilité pour que les deux premiers n'aient pas d'anniversaire en commun est de $364/365$, celle pour que les 3 n'aient pas d'anniversaire commun est donc $364/365 \times 363/365$ et ainsi de suite;
- si $n = k$ la probabilité est $\frac{364}{365} \times \frac{363}{365} 5 \times \frac{362}{365} \times \cdots \times \frac{365-k+1}{365}$. On obtient la formule de récurrence

$$\begin{cases} P_1 = 1, \\ P_{k+1} = \frac{365-k+1}{365} P_k. \end{cases}$$

1. Tracer un graphe qui affiche la probabilité que deux personnes ont la même date de naissance en fonction du nombre de personnes.
2. Calculer pour quel k on passe sous la barre des 50%.

Source: blog Choux Romanesco, Vache qui rit et Intégrales curvilignes (http://eljjdx.canalblog.com/archives/2007/01/14/3691670.html)

Correction

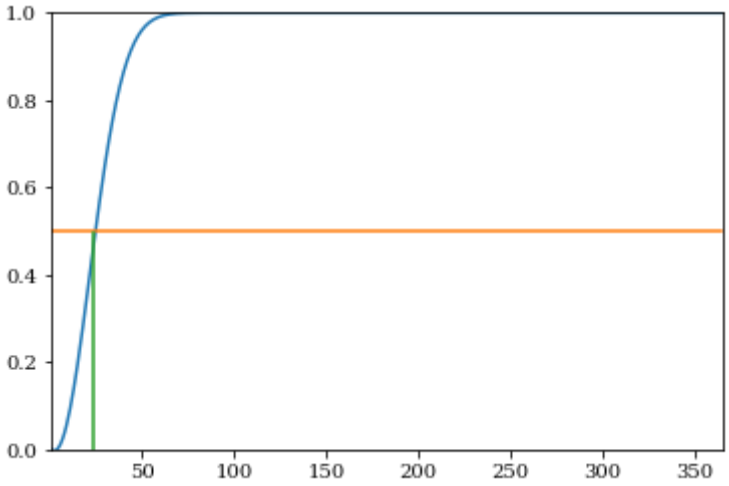
Montrons que dans un groupe de 23 personnes (`=indice`), il y a plus d'une chance sur deux (`seuil=0.5`) pour que deux personnes de ce groupe aient leur anniversaire le même jour (`tot=365`). Ou, dit autrement, il est plus surprenant de ne pas avoir deux personnes qui ont leur anniversaire le même jour que d'avoir deux personnes qui ont leur anniversaire le même jour (et avec 57, on dépasse les 99% de chances!)

On peut s'amuser à adapter les calculs à d'autres problèmes, par exemple on a `seuil=` 61% de chances que parmi `indice=` 5 personnes prises au hasard, deux ont le même signe astrologique (`tot=12`).

In [15]:

```
tot=365
seuil=0.5
nn=range(1,tot)
P=[1]
for k in range(tot-2):
    P.append( (tot-k+1)*P[k]/tot )

nP=[1-p for p in P]
indice=([np<seuil for np in nP]).count(True)
print(indice-1)
plot(nn,nP, [min(nn),max(nn)],[seuil,seuil],[indice,indice],[0
axis([1,tot,0,1]);
```

23



14 Notion de précision ...

- Calcul numérique: calcul en utilisant des nombres, en général en virgule flottante.
- Calcul numérique $\neq$ calcul symbolique.
- Nombre flottant: signe + mantisse («chiffres significatifs») + exposant.
- Valeur d'un flottant = $(-1)^{\text{signe}} + 1.\text{mantisse} * 2^{\text{exposant}}$
- Précision avec les flottants Python (double précision = 64 bits):
  - 1 bit de signe
  - 52 bits de mantisse ($\Rightarrow$ environ 15 à 16 décimales significatives)
  - 11 bits d'exposant ($\Rightarrow$ représentation des nombres de 10^{-308} à 10^{308})

14.1 Exemple: un calcul de π

On admet que

$$\arctan(1) = \frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

Pour calculer π avec des nombres flottants, il y a plusieurs problèmes de précision:

1. Le résultat attendu n'est pas un flottant représentable (il n'est même pas rationnel). Le meilleur calcul de π possible ne pourra donner qu'un arrondi à $\approx 10^{-15}$ près.
2. En utilisant des nombres flottants, chaque opération (/, +, ...) peut faire une erreur d'arrondi (erreur relative de 10^{-15}). Les erreurs peuvent se cumuler.
3. La formule mathématique est infinie. Le calcul informatique sera forcé de s'arrêter après un nombre fini d'itérations.

In [16]:

```
def gregory_leibniz(N):
    somme = 0
    denom = 1
    signe = 1
    for k in range(N):
        somme += signe / float(denom)
        denom += 2
        signe = -signe
    return 4*somme

print(gregory_leibniz(1))
print(gregory_leibniz(10))
print(gregory_leibniz(100))
print(gregory_leibniz(1000))
```

```
4.0
3.0418396189294032
3.1315929035585537
3.140592653839794
```

In [17]:

```
# Changer la précision de l'affichage (et seulement de l'affic
%precision 18
```

Out[17]:

```
'%.18f'
```

In [18]:

```
gregory_leibniz(10 ** 8)
```

Out[18]:

```
3.141592643589325995
```

In [19]:

```
# Valeur prédefinie par le module math
from math import *
pi
```

Out[19]:

```
3.141592653589793116
```

In [20]:

```
# Retour à la valeur par défaut
%precision
```

Out[20]:

```
'%r'
```

14.2 Les « mauvaises » propriétés des nombres flottants

> Il ne faut jamais se fier trop vite au résultat d'un calcul obtenu avec un ordinateur...

L'expérimentation numérique dans les sciences est un sujet passionnant et un outil fort utile, devenu indispensable pour certains scientifiques. Malgré la puissance vertigineuse de calcul de nos ordinateurs aujourd'hui, et encore plus de certains centres de calculs, on aurait tort d'oublier complètement la théorie et de trop se moquer de comment fonctionne la machine, au risque d'avoir quelques surprises...

Observons des calculs quelque peu surprenants:

In [21]:

```
0.1 + 0.1 + 0.1 - 0.3
```

Out[21]:

```
5.551115123125783e-17
```

In [22]:

```
1.1 + 2.2
```

Out[22]:

```
3.3000000000000003
```

Que s'est-il passé? Tout simplement, les calculs effectués ne sont pas exacts et sont entachés d'erreurs d'arrondis. En effet, tout nombre réel possède un développement décimal soit fini soit illimité. Parmi les nombres réels, on peut alors distinguer les rationnels (dont le développement décimal est soit fini soit illimité et périodique à partir d'un certain rang) des irrationnels (dont le développement décimal est illimité et non périodique). Il est aisé de concevoir qu'il n'est pas possible pour un ordinateur de représenter de manière exacte un développement décimal illimité, mais même la représentation des développements décimaux finis n'est pas toujours possible. En effet, un ordinateur stocke les nombres non pas en base 10 mais en base 2. Or, un nombre rationnel peut tout à fait posséder un développement décimal fini et un développement binaire illimité! C'est le cas des décimaux 1.1 et 2.2 qui, en base 2, s'écrivent $01.\overline{01}$ et $10.\overline{001}$ respectivement.

14.2.1 Erreurs d'arrondis

In [23]:

```
1.0 / 3 - 1.0 / 4 - 1.0 / 12
```

Out[23]:

```
-1.3877787807814457e-17
```

14.2.2 Non-commutativité

In [24]:

```
1 + 1e-16 - 1
```

Out[24]:

```
0.0
```

In [25]:

```
-1 + 1e-16 + 1
```

Out[25]:

```
1.1102230246251565e-16
```

14.2.3 Répresentation décimale inexacte

In [26]:

```
# 1.2 n'est pas représentable en machine. L'ordinateur utilise
1.2 - 1.0 - 0.2
```

Out[26]:

```
-5.551115123125783e-17
```

14.2.4 Conséquences

- On ne peut pas espérer de résultat exact
- La précision du calcul dépend de beaucoup d'éléments
- En général, pour éviter les pertes de précision, on essayera autant que faire se peut d'éviter:
  - de soustraire deux nombres très proches.
  - d'additionner ou de soustraire deux nombres d'ordres de grandeur très différents. Ainsi, pour calculer une somme de termes ayant des ordres de grandeur très différents (par exemple dans le calcul des sommes partielles d'une série), on appliquera le principe dit *de la photo de classe*: les petits devant, les grands derrière.
- tester `x == 0.0` avec un flottant est presque toujours une erreur
- *Si on s'y prend bien*, on perd $\approx 10^{-16}$ en précision relative à chaque calcul $\Rightarrow$ acceptable par rapport à la précision des données.
- *Si on s'y prend mal*, le résultat peut être complètement faux!

14.2.5 Le module `fractions`

In [27]:

```
from fractions import Fraction
print(0.1 + 0.1 + 0.1 - 0.3, "\t vs ", Fraction(1,10) + Fracti
print(1.1 + 2.2, "\t vs ", Fraction(11,10) + Fraction(22,10))
print(1.0 / 3 - 1.0 / 4 - 1.0 / 12, " vs ", Fraction(1,3) + Fr
print( 1 + 1e-16 - 1, " vs ", Fraction(1,1) + Fraction(1,10**1
print(-1 + 1e-16 + 1, " vs ", -Fraction(1,1) + Fraction(1,10**
print(1.2 - 1.0 - 0.2, " vs ", Fraction(6,5) - Fraction(1,1) -
```

```
5.551115123125783e-17    vs  0
3.3000000000000003       vs  33/10
-1.3877787807814457e-17  vs  1/2
0.0  vs  1/10000000000000000
1.1102230246251565e-16  vs  1/10000000000000000
-5.551115123125783e-17  vs  0
```

14.3 Exemple: sommes et produits

Illustrons ce problème d'arrondis en partant de l'identité suivante:

$$xy = \left(\frac{x+y}{2}\right)^2 - \left(\frac{x-y}{2}\right)^2.$$

Dans le programme suivant on compare les deux membres de cette égalité pour des nombres x et y de plus en plus grands:

In [28]:

```
prod = lambda x,y : x*y
diff = lambda x,y : ((x+y)/2)**2-((x-y)/2)**2

a = 6553.99
b = a+1
print( "-"*9,"a" ,"-"*9, "|" , "-"*9, "b", "-"*9, "|" , "ab-((
for i in range(6):
    produit = prod(a,b)
    difference = diff(a,b)
    print( "%1.15e | %1.15e |%1.15e" % (a,b,produit-difference
    a, b = produit, a+1
```

```
--------- a --------- | --------- b --------- | a
b-((a+b)/2)^2-((a-b)/2)^2
6.553990000000000e+03 | 6.554990000000000e+03 |0.
000000000000000e+00
4.296133891010000e+07 | 6.554990000000000e+03 |-
5.859375000000000e-02
2.816111469423164e+11 | 4.296133991010000e+07 |1.
667072000000000e+06
1.209839220626197e+19 | 2.816111469433164e+11 |-
4.798256640190465e+21
3.407042105375514e+30 | 1.209839220626197e+19 |1.
046648870315659e+44
4.121973165408151e+49 | 3.407042105375514e+30 |1.
404373613177356e+80
```

On constate que la divergence est spectaculaire!

14.4 Exercice : suites récurrentes

Calculer analytiquement et numériquement les premiers 100 termes des deux suites:

$$\begin{cases} u_0 = \frac{1}{4}, \\ u_{n+1} = 5u_n - 1, \end{cases} \qquad \begin{cases} v_0 = \frac{1}{5}, \\ v_{n+1} = 6v_n - 1. \end{cases}$$

Correction

Clairement $u_i = \frac{1}{4}$ et $v_i = \frac{1}{5}$ pour tout $i \in \mathbb{N}$. Cependant, lorsqu'on calcul les premiers 100 termes de ces deux suites avec Python (ou avec un autre langage de programmation) on a quelques surprises.

Si on écrit

In [29]:

```
u = 1/4
for n in range(1,10):
    u = 5*u-1
    print(u)
```

```
0.25
0.25
0.25
0.25
0.25
0.25
0.25
0.25
0.25
```

on trouve bien $u_i = 0.25$ pour tout $i = 0, \dots, 99$.

Mais si on écrit

In [30]:

```
v = 1/5
for n in range(1,5):
    v = 6*v-1
    print(v)
```

```
0.20000000000000018
0.20000000000000107
0.2000000000000064
0.20000000000003837
```

on obtient $v_i \simeq 0.2$ pour $i = 0, \dots, 5$, ensuite les erreurs d'arrondis commencent à se voir:

In [31]:

```
v = 1/5
for n in range(1,30):
    v = 6*v-1
    print(v)
```

```
0.20000000000000018
0.20000000000000107
0.2000000000000064
0.20000000000003837
0.20000000000023022
0.2000000000013813
0.20000000000828777
0.20000000004972662
0.20000000029835974
0.20000000179015842
0.20000001074095053
0.20000006444570317
0.20000038667421904
0.20000232004531426
0.20001392027188558
0.2000835216313135
0.20050112978788093
0.20300677872728556
0.21804067236371338
0.3082440341822803
0.8494642050936818
4.096785230562091
23.580711383372545
140.48426830023527
841.9056098014116
5050.43365880847
30301.60195285082
181808.6117171049
1090850.6703026295
```

Pour calculer la bonne valeur nous allons utiliser le module `fractions` qui évite les erreurs d'arrondis:

In [32]:

```
from fractions import Fraction
v = Fraction(1,5)
for n in range(1,30):
    v = 6*v-1
    print(v)
```

```
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
1/5
```

14.5 Exercice: suite de Muller

Considérons la suite

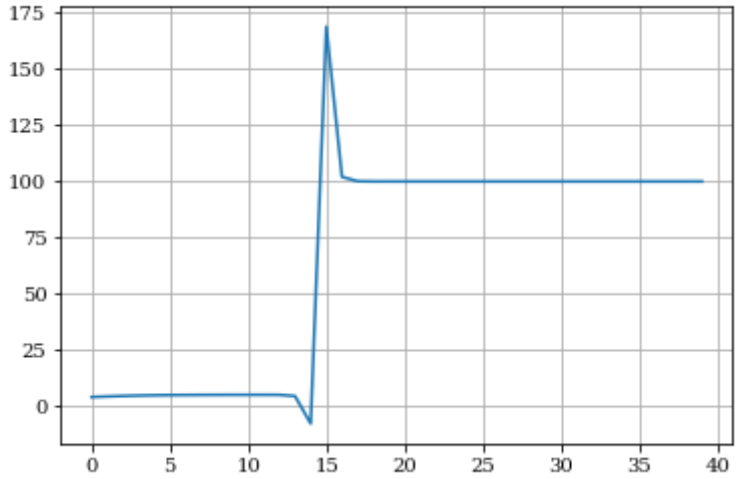
$$\begin{cases} x_0 = 4, \\ x_1 = 4.25, \\ x_{n+1} = 108 - \dfrac{815 - \frac{1500}{x_{n-1}}}{x_n}. \end{cases}$$

On peut montrer que $\lim_{n\to+\infty} x_n = 5$ (voir par exemple https://scipython.com/blog/mullers-recurrence/ (https://scipython.com/blog/mullers-recurrence/)) Qu'obtient-on numériquement?

Correction

In [33]:

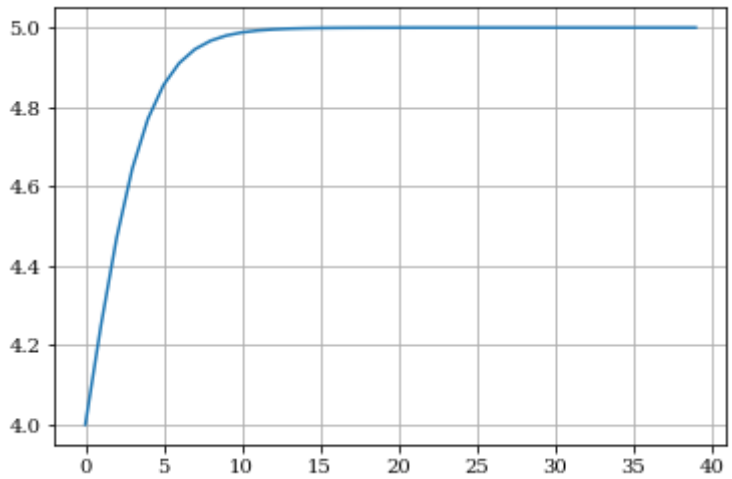
```
n=40
x=[4, 4.25]
for i in range(2,n):
    x.append(108-(815-(1500/x[-2]))/x[-1])
plot(range(n),x)
grid(True)
```



Pour calculer la bonne valeur nous allons utiliser le module `fractions` qui évite les erreurs d'arrondis:

In [34]:

```
from fractions import Fraction
n=40
x=[4, Fraction(17, 4)]
for i in range(2,n):
    x.append(108 - Fraction((815 - Fraction(1500, x[-2])), x[-
plot(range(n),x)
grid(True)
```



14.6 Exercice: evaluer la fonction de Rump

Évaluer au point $(x, y) = (77617, 33096)$ la fonction de deux variables suivante:

$$f(x, y) = \frac{1335}{4}y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + \frac{11}{2}y^8 + \frac{x}{2y}$$

Correction

In [35]:

```
def f(x,y):
    return 1335*y**6/4+x**2*(11*x**2*y**2-y**6-121*y**4-2) + 1
print(f(77617,33096))
```

```
1.1726039400531787
```

Si vous faites le calcul à la main (ou vous utilisez le module `fractions` ou encore le module `sympy` comme ci-dessous), vous trouvez une valeur exacte de environ -0.8273960599: non seulement l'ordinateur a calculé une valeur très éloignée du résultat mais en plus, le signe n'est même pas le bon.

In [36]:

```
from fractions import Fraction
def f(x,y):
    return Fraction(1335,4)*y**6+x**2*(11*x**2*y**2-y**6-121*y
sol=f(77617,33096)
print(sol,"=",float(sol))
```

```
-54767/66192 = -0.8273960599468214
```

In [37]:

```
import sympy
x,y,g= sympy.symbols('x,y,g')
g = 1335*y**6/4+x**2*(11*x**2*y**2-y**6-121*y**4-2) + 11*y**8/
sol=g.subs({x:77617, y:33096})
print(sol,"=",sol.evalf())
```

```
-54767/66192 = -0.827396059946821
```

14.7 Exercice: calcul d'intégrale par récurrence

On veut approcher numériquement l'intégrale $I_n = \int_0^1 x^n e^{\alpha x} \mathrm{d}x$ pour $n = 50$. On remarque que, en intégrant par partie I_n, on a

$$
\begin{aligned}
I_n &= \int_0^1 x^n e^{\alpha x} \mathrm{d}x \\
&= \left[x^n \frac{1}{\alpha} e^{\alpha x} \right]_0^1 - \frac{n}{\alpha} \int_0^1 x^{n-1} e^{\alpha x} \mathrm{d}x \\
&= \frac{1}{\alpha} e^{\alpha} - \frac{n}{\alpha} I_{n-1}
\end{aligned}
$$

On décide alors de calculer I_{50} par la suite récurrente suivante:

$$
\begin{cases}
I_0 = \frac{e^{\alpha}-1}{\alpha}, \\
I_{n+1} = \frac{1}{\alpha} e^{\alpha} - \frac{n+1}{\alpha} I_n, \text{ pour } n \in \mathbb{N}.
\end{cases}
$$

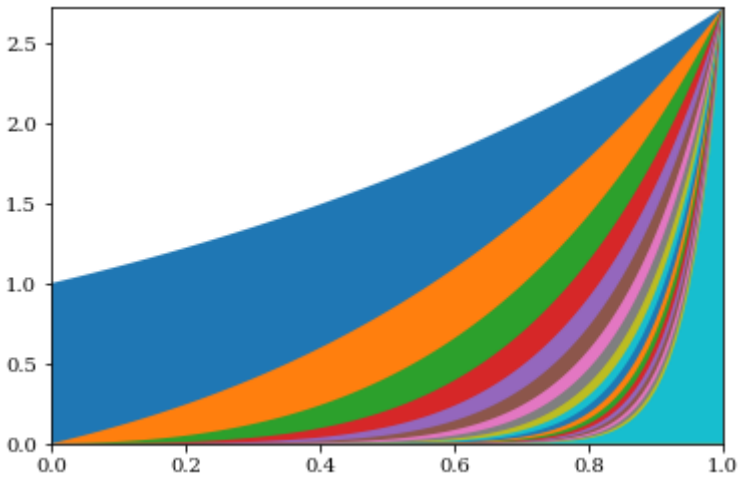
Écrire un programme pour calculer cette suite. Comparer le résultat numérique avec la limite exacte $I_n \to 0$ pour $n \to +\infty$.

Correction

On commence par afficher I_n pour différentes valeurs de n (il s'agit de l'aire coloriée): on voit que $I_{n+1} < I_n$.

In [38]:

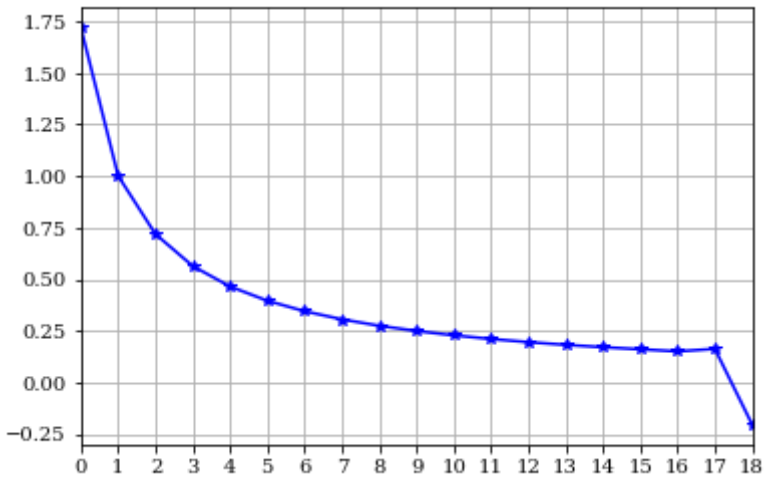
```
%reset -f
%matplotlib inline

from matplotlib.pylab import *
import matplotlib.animation as animation
from IPython.display import display, clear_output
import time

alpha=1
x=linspace(0,1,101)

fig = figure()
axis([0,1,0,max(1,exp(alpha))])
line, = plot([],[],lw=2)

animate = lambda n: fill_between(x,0,x**n*exp(alpha*x))

# CI
fill_between(x,0,exp(alpha*x))

# Marche en temps
for n in range(1,20):
    animate(n)
    clear_output(wait=True)
    display(fig)
    time.sleep(0.5)    # pause 0.5 seconds
clear_output(wait=True)
```



Si on calcule I_n avec la formule de récurrence avec $\alpha = 1$, on remarque que $0 < I_{n+1} < I_n$ pour $n < 17$, mais $I_{18} < 0$ et la suite est instable. On a le même comportement pour les autres valeurs de α.

In [39]:

```
%matplotlib inline
from matplotlib.pylab import *

alpha=1
I=[(exp(alpha)-1)/alpha]

N=18
for n in range(N):
    I.append(exp(alpha)/alpha -(n+1)*I[-1]/alpha)

plot(range(len(I)),I,'b-*');
xlim(0,N)
xticks(range(N+1))
grid(True)
```



La suite obtenue avec le programme ci-dessus ne tend pas vers zéro quand n tend vers l'infini. Pourquoi un tel comportement numérique?

Ce comportement est une conséquence directe de la propagation des erreurs d'arrondi: en passant de I_n à I_{n+1}, l'erreur numérique (accumulation des erreurs de représentation et des premiers calculs) est multipliée par n:

$$\varepsilon_{n+1} = I_{n+1}^{\text{exacte}} - I_{n+1}^{\text{approx}} = \left(\frac{1}{\alpha}e^{\alpha} - \frac{n+1}{\alpha}I_n^{\text{exacte}}\right) - \left(\frac{1}{\alpha}e^{\alpha} - \frac{n+1}{\alpha}I\right.$$
$$\left. - \frac{n+1}{\alpha}\varepsilon_n\right.$$

L'erreur numérique $|\varepsilon_n|$ sur l'évaluation de I_n croit donc comme $\frac{n!}{\alpha^n}|\varepsilon_0|$.

Ici on ne peut pas utiliser le module `fraction` car $e \notin \mathbb{Q}$.

14.8 Exercice: un contre-exemple du dernier theorème de Fermat ?

Le dernier théorème de Fermat (prouvé par Andriew Wiles (https://fr.wikipedia.org/wiki/Andrew_Wiles) en 1994) affirme que, pour tout entier $n > 2$, trois entiers positifs x, y et z ne peuvent pas satisfaire l'équation $x^n + y^n - z^n = 0$. Expliquez le résultat de cette commande qui donne un contre-exemple apparent au théorème:

In [40]:

```
print(844487.**5 + 1288439.**5 - 1318202.**5)
```

```
0.0
```

Correction

Le problème, bien sûr, vient de l'utilisation des nombres à virgule flottante double précision et que la différence entre la somme des deux premiers termes et celle du troisième est inférieure à la précision de cette représentation.

Si on utilise des entiers on a bien:

In [41]:

```
print(844487**5 + 1288439**5 - 1318202**5)
```

```
-235305158626
```

En effet, la précision finie de la représentation en virgule flottante utilisée tronque les décimales avant que cette différence soit apparente:

In [42]:

```
print("Exact =", 844487**5 + 1288439**5, "Arrondis =", 844487.
print("Exact =", 1318202**5, "Arrondis =", 1318202.**5)
```

```
Exact = 398024523518563901305561949740 6 Arrondis
= 3.980245235185639e+30
Exact = 398024523518563901329092465603 2 Arrondis
= 3.980245235185639e+30
```

Ceci est un exemple d'annulation catastrophique.