In [ ]:
```
%reset -f
# Reset le kernel: toutes cellules déjà exécutées ne le sont plus
%autosave 300
# Sauvegarde le document toutes les 300 secondes
# !! Ne permet pas de commentaire sur la même ligne !!
# D'autres méthodes propres au Jupyter Notebook existent.
```

```
Autosaving every 300 seconds
```

Introduction

J'ai initialement créé ce document pour moi, en tant que fiche rassemblant tous mes cours de Python. Je me suis ensuite rendu compte que cela pourrait intéresser bien du monde. Bref, si vous avez des suggestions, faites-le moi savoir (je ne sais pas encore comment faire des suggestions via GitHub, mais on peut découvrir ensemble).

> Guide pour bien démarrer la programmation:
>
> Pour programmer confortablement, il faut un **IDE** (environnement de développement intégré). Il vous permet d'avoir des couleurs, signaler les erreurs,... Bref, rendre la vie plus facile. Je vous recommande Vs code, un des principaux sur le marché et, bien que complet, facile à prendre en main. Il fonctionne avec des **extensions** (icône de boîtes sur le panneau de gauche). Je vous conseille Python, Jupyter Notebook (vous en lisez un), markdown all in one (le langage de cette cellule), et vscode-pdf (utile pour tout avoir dans le même environnement). Car on peut **ouvrir un fichier** entier, et organiser les fenêtres comme on veut, comme sur le bureau windows ! Tout est dans l'icône documents du panneau de gauche. Certans fichiers comme les notebook (.ipynb) ont une architecture. Je vous laisse tester *outline*, toujours dans l'onglet Explorer.
>
> Voilà, je vous laisse explorer le reste sur Internet, les ressources ne manquent pas !

Sommaire

Histoire

> Vous etes vraiment pas obligé de passer par là

C'est un langage:

- **interpreté**, n'est pas compilé mais directement lu à l'exécution
- **impératif**
- **orienté objet**: tout est fondamentallement un objet avec des propriétés et des méthodes propres à cet objet. Voir section correspondante (à venir)

Son créateur est:

- Guido Van Rossum (1956)
- Dictateur bienveillant à vie
- Parcours: Navigateur Grail ; Google ; Dropbox ; Microsoft.

Les versions ont une compatibilité ascendante depuis la 3.0.0 seulement, celle-ci sera maintenue.

Voir les différentes voies d'utilisation.

Ressources:

Rubrique fourre-tout pour tout ce dont j'ai du mal à me rappeler.

Math cheatsheet

- $\overset{x_0}{=}$: `\overset{x_0}{=}`
- $\exists$: `\exists`
- $(...)$: `\left(...\right)`
- $\Leftrightarrow$: `\Leftrightarrow`
- $\xrightarrow[\infin]{overtext}$: `\xrightarrow[\infin]{overtext}` from mathtools, keep brackets if only undertext wanted

For internal (within the same notebook) links use this code: `[section title](#section-title)` For the text in the parentheses, replace spaces and special characters with a hyphen.

Alternatively, you can add an ID for a section right above the section title. Use this code: `<a id="section_ID"></a>`

Make sure that the section_ID is unique within the notebook. Use this code: `[section title](#section_ID)`

External links

Use this code: `[external link text](http://url_here)`

Install venv

> je formaterais plus tard

```
source /home/alpha/Documents/Git/Ressources/.venv/python/bin/activate
alpha@alpha-tester:~/Documents/Git/Ressources$ source /home/alpha/Documents/
Git/Ressources/.venv/python/bin/activate
(python) alpha@alpha-tester:~/Documents/Git/Ressources$ python3
Python 3.12.3 (main, Aug 14 2025, 17:47:21) [GCC 13.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
Ctrl click to launch VS Code Native REPL
>>> sys.prefix
'/home/alpha/Documents/Git/Ressources/.venv/python'
>>> sys.base_prefix
'/usr'
>>> exit()
(python) alpha@alpha-tester:~/Documents/Git/Ressources$ pip freeze >
requirements.txt
(python) alpha@alpha-tester:~/Documents/Git/Ressources$ pip install -r
requirements.txt
```

```
(python) alpha@alpha-tester:~/Documents/Git/Ressources$ pip install sympy
Collecting sympy
```

Bases python

- Mots interdits/utiles

and	as	assert	break	class	**continue**	def	del	elif	else	with
exept	false	finally	for	from	global	if	import	in	is	while
lambda	none	nonlocal	not	or	**pass**	raise	return	true	try	yield

- Types de variables : `type(a)` : `list` , `str` , `int` , `float` , `tuple` (liste non modifiable), `dict` , `set`

- `print` permet d'afficher du texte dans la console:

```
print("fzg",
      sep=' ',  # Séparateur
      end='\n', # Fin d'impression
      file=sys.stdout,  # Fichier d'impression, défaut console
      flush=False) # Si vrai, met à jour la console en temps réèl.
```

> Seul le premier argument est nécessaire, file peut être utilisé pour une écriture rapide, sans ouverture, et flush est utile pour les UI avec un temps à notifier. Sinon, le code attendra son bon vouloir ou la fin du code pour afficher touts les prints (optimisation).

- Pour les variables, on peut les unpack (dépaqueter), en créer une poubelle `_` *(ne sera jamais attribuée, utile pour n'extraire que partie d'un élément)* et indiquer plusieurs de leur types:

```
L:list[int|bool]=(1,2,True)
print(L)
print(*L)


# In console:
(1,2,True)
1 2 True
```

- `int` : 32 bits (4 octets)
- `bool` : True/False ; 0/1. `not` inverse. Attention aux priorités !
- `float` : $\in \mathbb{R}$. Notation décimale `0.1` ou exponentielle `1e-1`
- `char` : caractères codés en UTF-8 (4 octets max). Concaténable `+` et répétable `*` .

`type` permet de le connaître, on peut aussi le changer dynamiquement (`int()` par ex).

Attention, le langage étant haut niveau, la mémoire peut faire ce qu'elle veut !

- `from dis import dis` pour visualiser à bas niveau les tâches effectuées par le processeur.

- Indentations obligatoires et normalisées !

- creuser `# -*- coding: utf-8 -*-` en debut de fichier.

- Opérations de base

    - `//` : Quotient division
    - `%` : Reste
    - `==` : Egal
    - `!=` : Différend
    - `< > <= >=` : Comparaisons
    - `in` : appartient
    - `assert` , `input()`
- Pour un code propre:
    - Utiliser le module `typing` : `Callable[[type_entrée], type_sortie]` permet de préciser le type d'une fonction.
    - `# type: ignore[nom-erreur]` permet d'ignorer certaines erreurs. `[nom-erreur]` est facultatif.

Fonctions

> Permet de définir une routine, un bout de code qu'on aura plus à écrire. Les variables utilisées sont locales, `return` les sort. On peut aussi utiliser `global` pour les généraliser mais dangereux.

- On utilise généralement le mot-clé `def` :

```
def nom_de_fonction(arg1,arg2:int|float,arg3="ToTo")->tuple[int]:
  """
  Description accessible en utilisant help ou l'éditeur.
  """
  # insert code here
  return (1,2,3)
```

On peut donc mettre plusieurs arguments, et spécifier leur type, tout comme on peut spécifier le type de sortie. Cela sera ignoré par l'éditeur, seul l'IDE l'utilise.

On peut donner des valeurs par défault aux derners arguments, sans spécifier le type qui est alors implicite.

> Il existe plein d'arguments spéciaux, a creuser.

`return` quitte la boucle, et renvoie les arguments suivant sa ligne. Voir les boucles pour des mots-clés du même type.

- Pour définir de simples fonctions mathématiques rapidement, on utilise `lambda` :

```
f=lambda x: x**2 # définit f(x)=x**2.
rect:function=lambda x,T:1 if -T/2<x<T/2 else 0 # définit rect_T(x).
```

On ne peut pas annoter la fonction, mait peut contourner avec `Callable` (a creuser).

On peut par contre mettre des conditions !

Listes

- `l[début:fin:pas]` : appele des éléments de la liste. Un pas négatif est possible ! (parcours inverse)
- `[i**2 for i in range(debut,fin,pas) if i<n]` : liste par compréhension. On peut y mettre des conditions, mais pas `elif` :

```
[[3 if i==j else -1 if abs(i-j)==1 else 0 for i in range(10)]
   for j in range(10)]
```

- Méthodes:
  - `sum(l)` : somme les éléments de l ssi ce sont tous des nombres
  - `len(l)` : longueur de l
  - `del l[i]` : supprime l'élément de position i
  - `sorted(l)` : renvoie une liste triée
  - `max(l)` `min(l)`
  - `l.append(x)` , `l.remove(x)` , `l.count(x)` , `l.insert(i,x)` , `l.clear(x)`
  - `l.extend(L)` ajoute les éléments de L en fin de l, équivaut à l+L
  - `l.pop(i)` renvoie l'élément d'indice i dans l puis le supprime
  - `l.index(x)` renvoie l'indice de la première occurrence de x dans l
  - `l.sort()` **modifie** la liste a en la triant.
  - `l.reverse()` **modifie** la liste a en inversant les éléments
- Parcourir listes dans une boucle `i for i in ...` :
  - `l` parcourt les éléments de l
  - `range(len(l))` parcourt ses indices
  - `enumerate(l)` donne la paire (indice,éléments)
  - `zip` parcourt n listes d'un coup, s'arrête au dernier rang de la plus petite liste

`i,(ai,bi) in enumerate(zip(a,b))` **Parcours ultime !**

> Voir module numpy pour matrices

Gestion de texte ; Lecture/écriture

Pour du micromanagement de caractères, `ord` *donne le code Unicode du caractère en entrée et* `chr` *est l'opération inverse.*

Pour les caractères uniques, on utilise `' '` ; et les chaînes `" "` . Habitude à prendre pour les cas peu permissifs.

f-string : `f"texte1 {(variable à insérer) 1:.(nb décimales voulues)f} suite du texte"` D'autres modes que `.f` existent.

- Caractères spéciaux
  - `\n` retour a la ligne
  - `\t` tab
  - `\a` bib système
  - `\\` antislash
  - `\'` ou `\"` : écrire `'` ou `"` dans un texte

`f=open("test.txt","r")` : `r` = read ; `w` = write ; `r+` = both ; `a` = append

- Méthodes principales :
  - `read(n)` si n omis, retourne un str contenant tous les caractères du fichier. Sinon retourne une chaine contenant n caractères
  - `f.readline()` retourne la ligne en cours de lecture (permet de le parcourir)
  - `f.readlines()` retourne toutes les lignes du fichier dans une liste
  - `l.rstrip("\n\r")` enlève le superflu

    - `l.replace(",",".")` convertit le format des valeurs
    - `l.split(\t)` renvoie un liste avec tab comme séparateur
- Autres méthodes:

    - `f.index("x")` indice de la première occurrence de x
    - `f.count("x")` nombre d'occurrence de "x"
    - `f.lower()` `f.upper()` **nouvelle** chaîne où tous les caractères de f sont en minuscule/majuscule
    - `f.capitalize()` **nouvelle** chaîne où la **première lettre du premier mot** de f est en majuscule
    - `f.title()` **nouvelle** chaîne où la **première lettre de chaque mot** de s est en majuscule

Boucles et récursivité

- `if` , `elif` , `else` : *si 1, sinon 2, sinon*. Permettent de placer un ensemble d'actions à conditions.
- `for k in` : *pour*. Itère un élément. Avec `range(n)` , accomplit n fois la boucle.
- `while` : *tant que*. Tant que les conditions spécifiées sont True, continue. Attention aux boucles infinies !

`and` , `or` , `if` , ... peuvent ajouter der spécifications aux boucles selon les besoins. Ne pas oublier `:` à la fin et l'indentation qui suit !

- Méthodes avancées :

    - `break` sort de la plus petite boucle **et saute les `else` de cette indentation !**
    - `continue` passe directement à la prochaine itération *(utile pour éviter des cas impossibles)*

Voir le tableau en début de document pour plus d'inspiration !

Certains tris sont récursifs.

Tris classiques

> Voici plusieurs méthodes de tri et de manipulation de liste.

In [ ]:
```
# Je ne les ai pas testés depuis longtemps, signalez-moi toute erreur svp.

def plus_proche_voisin(L : list[int]) -> tuple :
    """
    renvoie les deux éléments voisins les plus proches l'un de l'autre
    (dont la valeur absolue de la différence est la plus petite, s'ils sont côte à cô
    """
    a,b=L[0],L[1]
    for i in range(1,len(L)):
        diff=abs(a-b)
        difi=abs(L[i-1]-L[i])
        if (difi)<(diff):
            a,b=L[i-1],L[i]
        if (difi)==(diff):
                continue
    return a,b
```

```
def dicho(l:list,x:float ):    # cherche si x appartient à l
    """
    Cherche x dans l (liste triée), renvoie sa position s\'il existe, False sinon.
    """
    debut=0
    fin=len(l)-1
    for i in range (len(l)):
        m=l[int(debut-fin)]
        if m==x:
            return int(debut-fin)
        elif m<x:
            fin=fin/2
        elif m>x:
            debut=fin/2
    return False

def tri_bulle(nombres : list) -> list:
    """
    Le tri bulle. Change la liste en la triant. Youtube peut aider à le comprendre.
    """
    n = len(nombres)
    for i in range(n-1, 0, -1):
        for j in range(0,i):
            if nombres[j]>nombres[j+1]:
                nombres[j],nombres[j+1]=nombres[j+1],nombres[j]
    return nombres

def tri_insertion(T : list[float]):
    """
    Le tri par insertion. Change la liste en la triant. Youtube peut aider à le compre
    Précision supposée: O(n²)
    """
    for i in range(1,len(T)):
        j=i
        x=T[j]
        while j>0 and T[j-1]>x:
            T[j]=T[j-1]
            j=j-1
            T[j]=x
    return T

# Tri quicksort. Deux fonctions car pourquoi pas. Je modifierai peut-être un jour.
def echange(T:list,i:int,j:int):
    T[i],T[j]=T[j],T[i]
    return T

def partition (T:list[float],g:int,d:int):
    assert g<d ,"mauvaises bornes !" # configure une erreur. Stoppe l'exécution !
    x=T[g]
    m=g
    for i in range (g+1,d):
        if T[i]<x:
            m=m+1
            echange(T,i,m)
        if m!=g:
             echange(T, g, m)
    return m
```

Bibliothèques:

`import` permet d'importer des bibliothèques de fonctions facilitant grandement la vie.

In [ ]:
```
import numpy as np                    # Meilleure méthode, l'alias.
```

```
import matplotlib.pyplot as plt     # Ces deux bibliothèques sont les principales: tal
from random import randint          # Importe un nombre limité de fonctions en les nor
from turtle import * # type: ignore # Importe tous les éléments de la bibliothèque. Ri
from copy import deepcopy           # Ces trois sont plus spécifiques, créer des valeu

#from tutor import tutor            # Permet de suivre un code étape par étape
#from rcviz import*                 # Pour suivre une récursivité
```

Numpy

> Permet l'utilisation de tableaux/matrices, en amélioration des listes. On ne peut pas modifier sa taille, mais est bien plus intuitif: permet d'appliquer une opération à chaque élément de la liste.

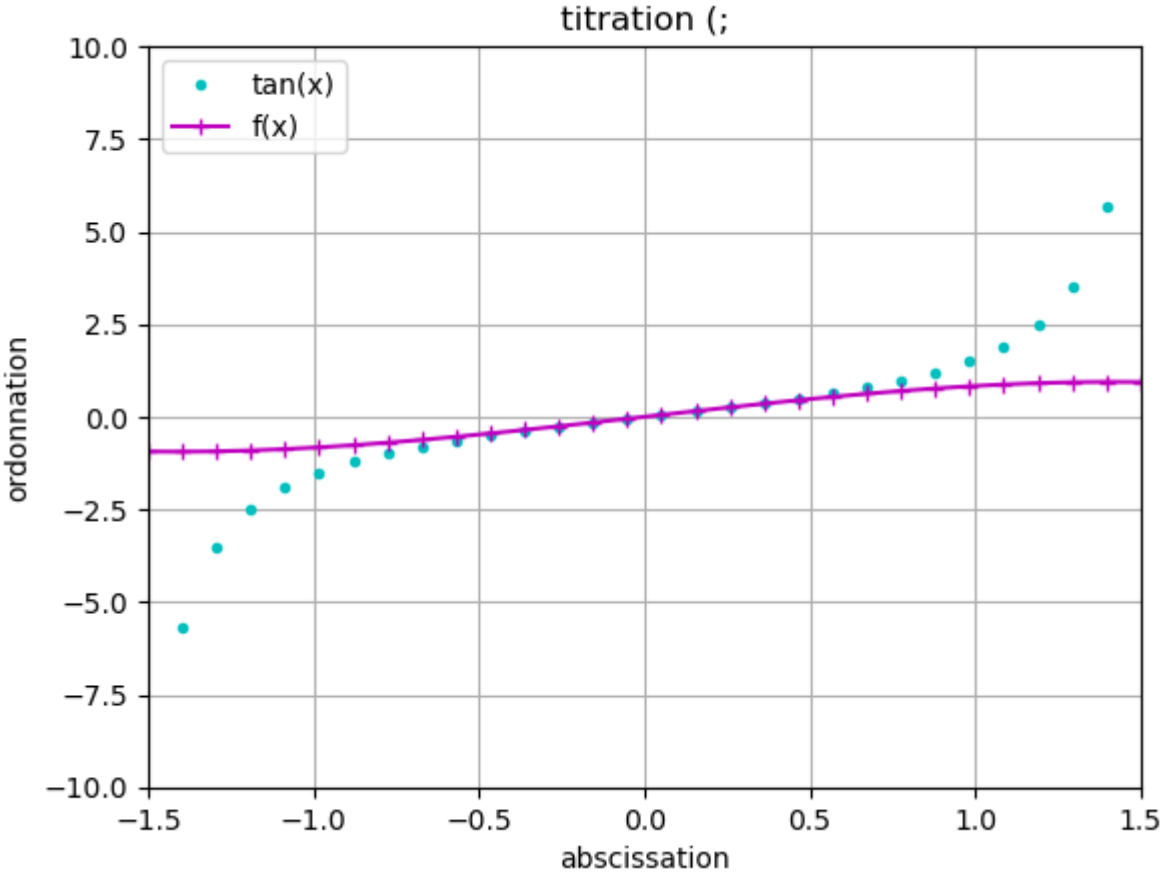
- La plupart des fonctions de maths y sont et sont transparentes.
- `array([...])` : transforme une liste normale en tableau
- `linspace(a,b,n)` : Liste de n éléments $\in [a, b]$
- ...

Matplotlib

> Possède des sous-bibliothèques: matplotlib.pyplot, .colors ...

> Permet de tracer des graphes de tous types
>
> ## Tracé de fonctions : Plot

In [ ]:
```
# Exemple avec tan(x) et une fonction quelconque
x = np.linspace(-1.5, 1.5, 30)
yt = np.tan(x)
yx = x-(1/6)*x**3

import matplotlib.pyplot as plt
plt.figure() # initie le tracé

plt.plot(x, yt, 'c.',label='tan(x)')
plt.plot(x, yx, 'm+-',label='f(x)')

plt.legend()
plt.grid()
#plt.errorbar(x, y, kwargs)
plt.axis([-1.5, 1.5, -10, 10]) # type: ignore
plt.title('titration (;')
plt.xlabel('abscissation')
plt.ylabel('ordonnation')
plt.show() # affiche le/les graphes initiés
plt.close() # vide la mémoire, et évite de se retrouver avec le même graphe plus loin
```

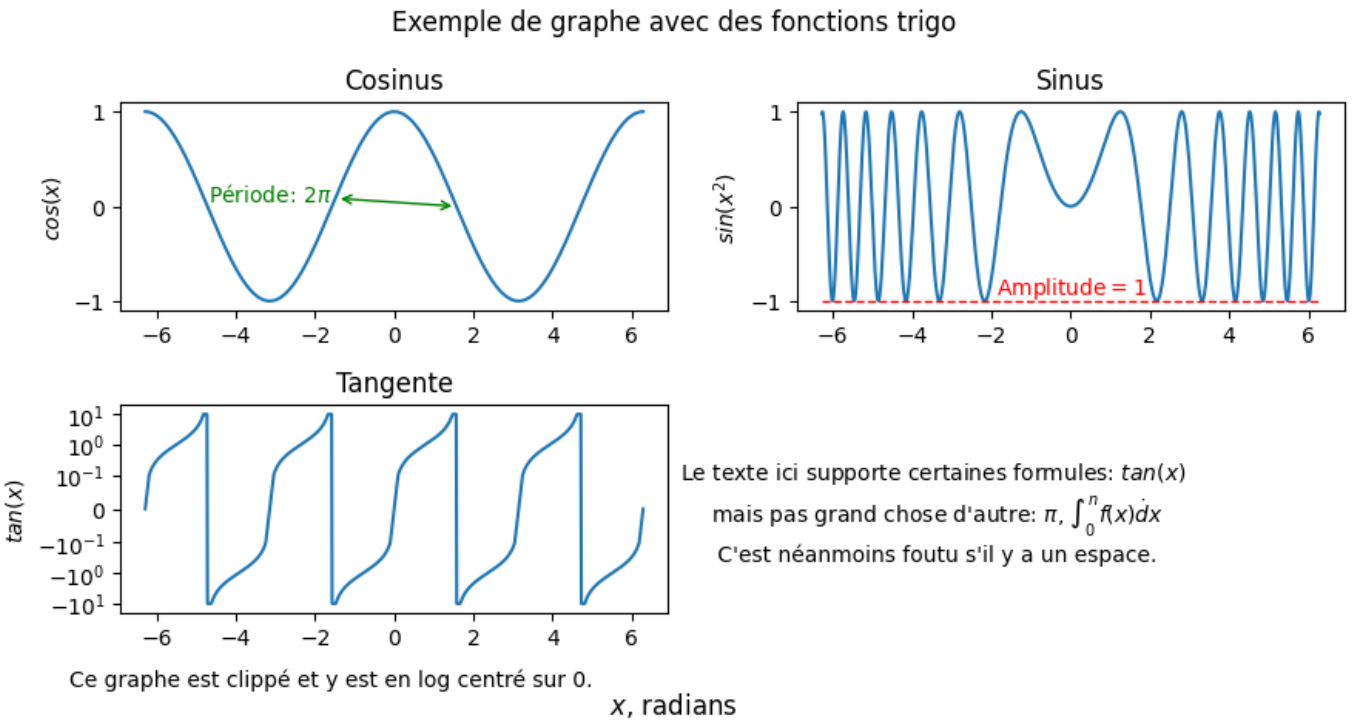


Dans un stade avancé, on peut mettre plusieurs graphes dans la même image avec `subplots`, commenter avec `annotate`, ...

Attention, la plupart des commandes doivent alors avoir `set_` comme préfixe.

In [58]:
```
import numpy as np
import matplotlib.pyplot as plt

cos=lambda x:np.cos(x)
sin=lambda x:np.sin(x**2)
tan=lambda x:np.tan(x)

x=np.linspace(-2*np.pi,2*np.pi,1000)

fig,ax=plt.subplots(2,2,figsize=(9,4.75)) # crée 2*2 graphes dans fig, réglables dans
fig.suptitle("Exemple de graphe avec des fonctions trigo")
fig.supxlabel("$x$, radians")

ax[0,0].plot(x,cos(x))
ax[0,0].annotate("Période: $2\pi$", #type: ignore
                 xy=(np.pi/2,0), xytext=(-np.pi,0),
                 arrowprops=dict(arrowstyle="<->",
                                 color="green",
                                 linewidth=1),
                 ha="center", # horizontalalignment
                 va="bottom", # verticalalignment
                 color="green") # Attributs de Text, global dans plt.
ax[0,0].set_title("Cosinus") # title ne fonctionne pas !
ax[0,0].set_ylabel("$cos(x)$")

ax[0,1].plot(x,sin(x))
ax[0,1].plot(x,np.array([-1 for i in x]),'--r',linewidth=1) # Enveloppe inférieure
ax[0,1].text(0,-1,
             "Amplitude$= 1$",
             ha="center", # Attributs de Text...
             color="red",
             va="bottom")
```

```
ax[0,1].set_title("Sinus")
ax[0,1].set_ylabel("$sin(x^2)$")

ax[1,0].plot(x,np.clip(tan(x),-10,10))
#ax[1,0].annotate()
ax[1,0].set_title("Tangente")
ax[1,0].set_ylabel("$tan(x)$")
ax[1,0].set_yscale('symlog', linthresh=0.1)
#ax[1,0].set_ylim(-10, 10)

ax[1, 1].axis('off')
plt.tight_layout() # quasi obligatoire avec subplots, permet d'évitre toute superposi
plt.text(0.25,0.25,
         "Le texte ici supporte certaines formules: $tan(x)$ \n mais pas grand chose d
         ha="center")
plt.text(-0.85,-0.35, # Réglé par tatonement, après tight layout pour le superposer
         "Ce graphe est clippé et y est en log centré sur 0.", # sans gêner l'agenceme
         ha="center")
plt.show()
plt.close()
```



Tracé de matrices : imshow

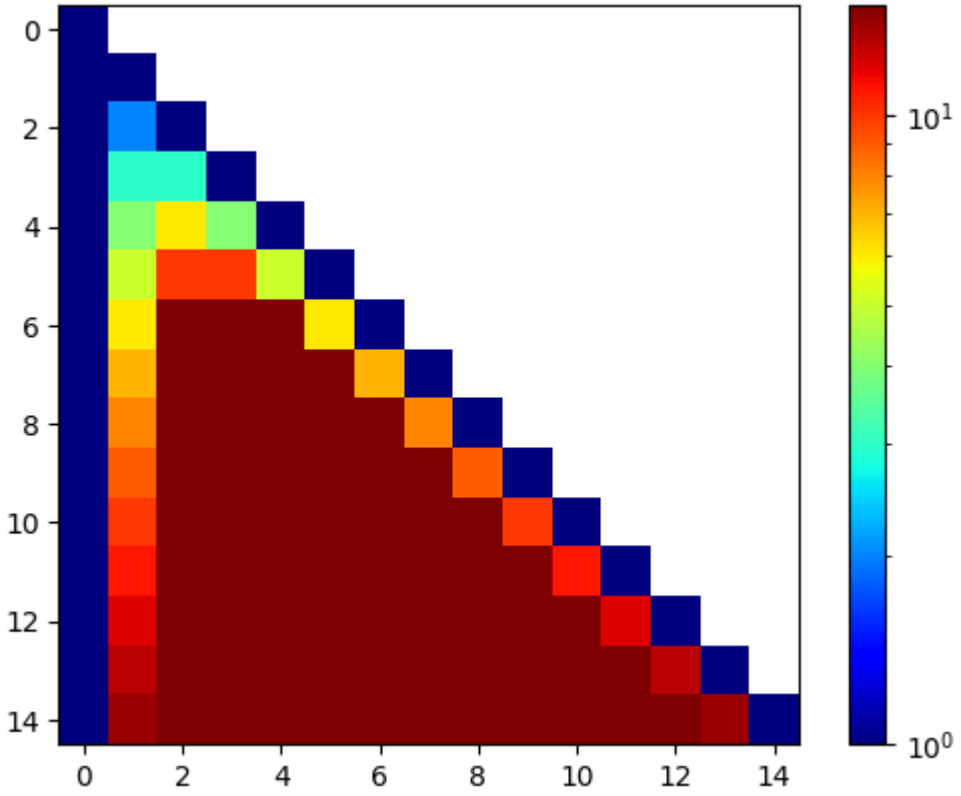
Cette méthode m'est très récente, comme oour tout info manquante, lisez la documentation.

In [4]:
```
import numpy as np
def pascal(n:int)->np.ndarray[tuple[int,int],np.dtype]:
    """Renvoie le tableau de Pascal"""
    l=np.zeros((n,n),dtype=int)
    l[:,0]=1
    for i in range(n-1):
        for j in range(n-1):
            l[i+1,j+1]=l[i,j]+l[i,j+1]
    return l

p=pascal(15) # Modifier pour obtenir une résolution différente
print("Pascal par matrice au rang 8:\n",pascal(8))

import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm # pour de meilleures couleurs

f=plt.imshow(p,
```

```
            cmap="jet", # hot
            norm=LogNorm(vmin=1,vmax=len(p)))
plt.colorbar(f)
# De meilleurs couleurs existent sûrement
```

```
Pascal par matrice au rang 8:
 [[ 1  0  0  0  0  0  0  0]
 [ 1  1  0  0  0  0  0  0]
 [ 1  2  1  0  0  0  0  0]
 [ 1  3  3  1  0  0  0  0]
 [ 1  4  6  4  1  0  0  0]
 [ 1  5 10 10  5  1  0  0]
 [ 1  6 15 20 15  6  1  0]
 [ 1  7 21 35 35 21  7  1]]
Matplotlib is building the font cache; this may take a moment.
```

Out[4]: `<matplotlib.colorbar.Colorbar at 0x771935b9d4c0>`



Sympy

> Bibliothèque permettant de faire du **calcul formel**: elle permet de résoudre des problèmes mathématiques comme on le ferait sur une feuille.

Bases:

In [5]:
```
import sympy as sy

t,c,a,x,y=sy.symbols('t c a x y',real=True) # Définit les variables, il peut y en avoi
f=sy.cos(2*x**3)*x**2                       # Pas une fonction mais une formule avec x co
F=sy.integrate(f,x)                         # Plus qu'à faire du calcul formel !! On appe
fp=sy.diff(f,x)                             # f'(x).
fp.diff(x,2)                                # f''(x). 2 est le degré. On peut généralemen
fp_np= sy.lambdify(x,fp,"numpy")            # Permet de passer de sympy à numpy, mais dif
                                                # mais ne s'affiche pas correctement
display(x,f,F,fp,fp_np)                     # affiche les formules au format markdown, à
print(sy.latex(F))                          # donne le code (LaTex ou formules Markdown)
print(f"\n{f=}\n{F=}\n{fp=}\n{fp_np=}") # les affiche au format python

f=2*x+x-3*x+4-5
```

```
display(f.simplify())                    # Permet de simplifier une formule. Ne fonctio
display(sy.Rational(1/2))                # Le force à rester rationnel, les floats sont
```

x

$x^2 \cos\left(2x^3\right)$

$\frac{\sin\left(2x^3\right)}{6}$

$-6x^4 \sin\left(2x^3\right) + 2x \cos\left(2x^3\right)$

```
<function _lambdifygenerated(x)>
\frac{\sin{\left(2 x^{3} \right)}}{6}

f=x**2*cos(2*x**3)
F=sin(2*x**3)/6
fp=-6*x**4*sin(2*x**3) + 2*x*cos(2*x**3)
fp_np=<function _lambdifygenerated at 0x771935734540>
```

-1

$\frac{1}{2}$

Résolution de systèmes d'équations

On utilise `Eq(x,y)` pour entrer l'équation $x = y$; qu'on résoud avec `solve([...],(var))` (Attention à sa syntaxe !).

Exemple: Cherchons l'intersection entre une parabole et une droite:

In [ ]:
```
parabol=sy.Eq(x**2,4*y)
droite=sy.Eq(x,2*y-1)
sol=sy.solve([parabol,droite],(x,y))
display(sol)
```

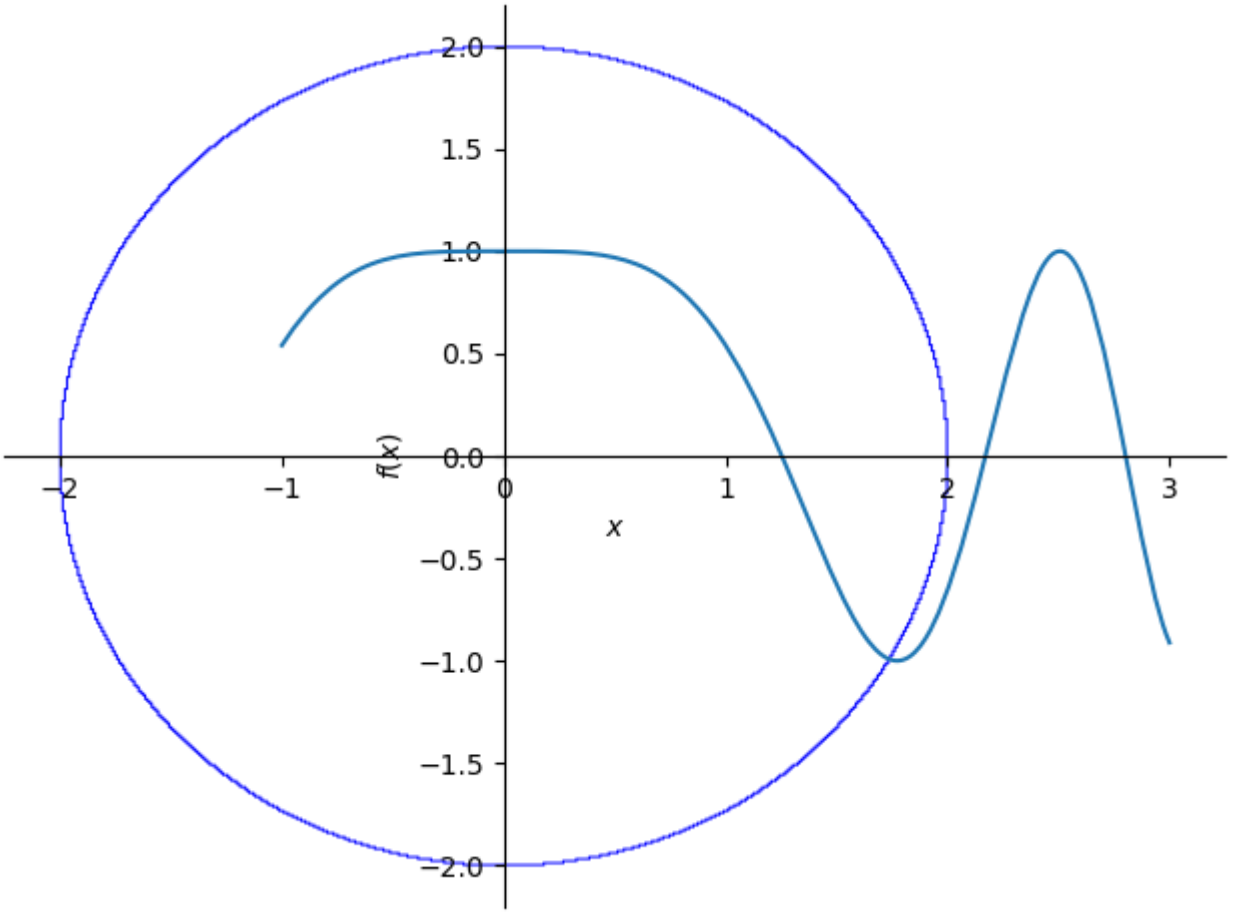
```
(1 - sqrt(3), 1 - sqrt(3)/2)
(1 + sqrt(3), sqrt(3)/2 + 1)
```

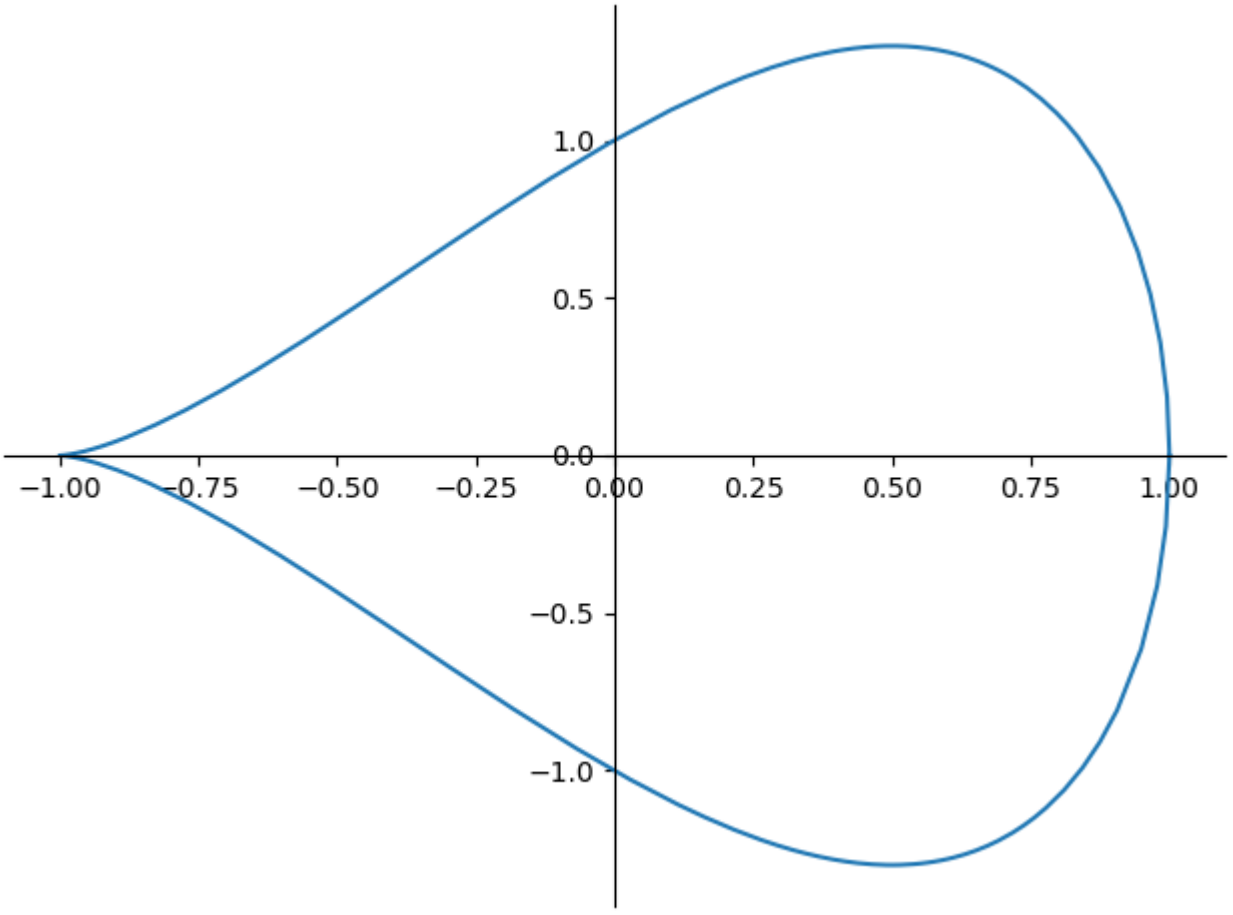
Représentation graphique

- `plot` montrera une **fonction** ;
- `plot_implicit` montrera une fonction implicite: le résultat d'une **équation** ;
- `plot_parametric` montrera le résultat d'un système d'équations paramétriques.

On pourra aussi fusionner des graphes.

In [ ]:
```
p1=sy.plot(sy.cos(x**2),
           (x,-1,3),
             show=False);

p2=sy.plot_implicit(sy.Eq(x**2 + y**2, 4),
                    show=False);

p1.extend(p2)
p1.show()

sy.plot_parametric((sy.cos(t),
                    sy.sin(t)+sy.sin(2*t)/2,
                     (t, 0, 2*sy.pi)));
```





Evaluation numérique

- `evalf()` permet d'**évaluer une expression symbolique**. On peut ajouter le nombre de chiffres significatifs en argument.
- `subs(x,...)` **substitue** une variable symbolique par une valeur numérique

In [12]:
```
f=1/x
```

```
display(f.subs(x,3),
        f.subs(x,3).evalf(5),
        f.subs(x,3).evalf())
```

$$\frac{1}{3}$$

$$0.33333$$

$$0.333333333333333$$

Algèbre linéaire

Matrices

On partira du principe que tout a déjà été vu en cours de maths (ce qui est faux pour moi).

- `sy.Matrix([[1,1],[1,1]])` donne $\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$, pour simplement définir une matrice
- `sy.diag(1,2,3)` donne $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$, matrice diagonale.
- `sy.ones(rang)` donne une matrice de 1.
- `sy.eye(rang)` donne l'identité, I_{rang}.
- `B[:,2]` On peut toujours récupérer une partie de la matrice, ici toutes les lignes de la $3^{\text{è}}$ colonne.
- `A.norm()` renvoie la norme du vecteur
- Les **calculs usuels** s'appliquent avec des subtilités: `+`, `*`, `**`, `.T`. Addition, multiplication de matrices, puissance par un scalaire, et **transposée** de matrices

On peut alors réaliser du calcul matriciel en utilsant ses propriétés:

- `sy.det(A)` donne le **déterminant** ;
- `A.inv()` donne son **inverse**. Cette opération est prioritaire sur la multiplication !
- `A.eigenvals()` donne les **valeurs propres** et leur **multiplicité** (respectivement) dans un dictionnaire ;
- `A.eigenvects()` donne les **valeurs propres**, leur **multiplicité**, et les **vecteurs propres** dans une **liste de tuples** (respectivement). Les vecteurs propres sont aussi dans une liste quand ils sont nombreux.
- `A.nullspace()` donne le **noyau** ;
- `A.rank()` donne le rang ;
- `A.columnspace()` donne son image sous forme de liste de matrices.
- `A.diagonalize` renvoie un tuple (P, D) tel que $A = PDP^{-1}$. D est donc la diagonale.

> Attention, pour les **tests logiques** entre A et B, on utilise `A.equals(B)`, car une égalité matricielle n'en est pas une scalaire !

Résolution de systèmes

> Résolvons $Ax = b$:

On pourrait trivialement inverser A, mais cela provoque des calculs inutiles. On préfèrera le pivot de Gauss, la factorisation PLU :

Exemple:

In [10]:
```
A=sy.Matrix([[1, 2, 0],[3, 2, 2],[2, 0, 0]])

L,U,perm=A.LUdecomposition()             # Application de la méthode
P = sy.eye(A.rows).permuteFwd(perm)      # La sortie de P demande une modification.

display(A,P,L,U)
print("On peut vérifier le résultat:")
display(L*U)
```

$$\begin{bmatrix} 1 & 2 & 0 \\ 3 & 2 & 2 \\ 2 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 0 \\ 0 & -4 & 2 \\ 0 & 0 & -2 \end{bmatrix}$$

```
On peut vérifier le résultat:
```

$$\begin{bmatrix} 1 & 2 & 0 \\ 3 & 2 & 2 \\ 2 & 0 & 0 \end{bmatrix}$$

Citation du cours *(M35 L2)*:

> La factorisation $P^T LU$ de A est donnée par $PA = LU$, où P est une matrice de permutation associée aux pivots utilisés, L est une matrice triangulaire inférieure avec des uns sur la diagonale, et U est une matrice triangulaire supérieure, celle obtenue à la fin du processus de transformation par méthode de pivot de Gauss. Si on dispose de la factorisation, on peut alors résoudre le système linéaire en trois étapes :
>
> 1. On résout le système linéaire $Ly = Pb$ pour y.
> 2. On résout le système linéaire $Ux = y$ pour x.
> 3. On a trouve la solution du système linéaire $Ax = b$.

Il y a aussi d'autres méthodes(`x=A.LUsolve(b)` , peut-être plus pratique), à creuser.

Je préfère personellement `A.inv(method="LU")` , bien plus simple.

Polynôme caractéristique

`A.charpoly(lambda)` est le **polynôme caractéristique** de A. On est censé savoir que ses racines (`sy.roots(A)`) sont valeurs propres de A, vérifiable avec `eigenvals` . C'est un polynôme qu'il faudra convertir en une expression avec `.as_expr()` .

```
A.charpoly(lmbda).as_expr()
```

Résolution numérique

Résolution de $f(x) = 0$

Résumé général

- Mathématiquement, on caractérise l'existence de la solution, tente de réduire le champ des possibles à l'aide de propriétés, pour finalement arriver à une **possible solution exacte.**
- Numériquement, on ne peut qu'approximer: On fixe une précision $p > 0$ ($10^{-3} < p < 10^{-12}$) et considère $f(x) = 0$ résolu si:
  - $|f(x) < p|$
  - $|x - \hat{x}| < p$
  - $|x - \hat{x}| + f(\hat{x}) < p$

  Avec x l'ancienne approximation, $\hat{x}$ la nouvelle.

Ordre de convergence

L'ordre de convergence d'une suite (x_n) est dite:

- Linéaire si

$$\lim_{n \to +\infin} \frac{|x_{n+1} - \hat{x}|}{|x_n - \hat{x}|} = \alpha \in]0, 1[$$

- Superlinéaire si

$$\lim_{n \to +\infin} \frac{|x_{n+1} - \hat{x}|}{|x_n - \hat{x}|} = 0$$

- D'ordre $p > 1$ si

$$\lim_{n \to +\infin} \frac{|x_{n+1} - \hat{x}|}{|x_n - \hat{x}|^p} \in \mathbb{R}_+^*$$

La convergence (Point fixe à compléter)

Méthode composite

Généralement, on va mettre en place une méthode peu précise. Il nous suffit alors de *diviser pour mieux régner*: On applique cette méthode petit bout par petit bout en divisant l'intervalle de calcul.

C'est la méthode composite

Méthodes numériques

Bonus : Méthode de héron

Un des premiers algorithmes pour la recherche de racine, proche de celui de Newton.

$$x_{k+1} = \frac{1}{2}\left(\frac{A}{x_k} + x_k\right), k \in \mathbb{N}^*$$

En effet, on montre que (x_k) est décroisante et minorée par $\sqrt{A}$:

$$x_{k+1}^2 - A = \frac{A + x_k^2}{4x_k^2} - A = \frac{(A + x_k^2)^2}{4x_k} \geq 0 \iff x_{k+1}^2 \geq A. \text{ Cqfd.}$$

De plus,

$$x_{k+1} - x_k = \frac{A - x_k^2}{2x_k} \leq 0 \Leftrightarrow x_k \leq \sqrt{A}, \forall k.$$

Par réccurence, on montre que cela est vrai ssi $x_0 > \sqrt{A}$.

On voit donc immédiatement que $(x_k) \xrightarrow[\infin]{} \sqrt{A}$ si $x_0 > \sqrt{A}$.

In [5]:
```
def heron(A,xk,maxITER): # Méthode de Héron
    print ("Valeur initiale : ", xk)
    for k in range(maxITER):
        xk = 0.5*(A/xk+xk)
        print ("Iteration ",k+1, " : ", xk)
    return xk
```

Méthode par dichotomie:

Soit f de classe C^1 et changeant de signe entre $[a_0, b_0]$. Selon le TVI, $f(x) = 0$ admet au moins 1 solution.

> Méthode: On divise l'intervalle en deux, cherche de quel côté on change de signe et reccomence de ce côté.

On définit donc (a_n), (b_n) et (m_n) par réccurence:

$$\begin{cases} m_n = \frac{a_n + b_n}{2} \\ a_{n+1} = b_n \text{ si } f(a_n)f(b_n) < 0, m_n \text{ sinon} \\ b_{n+1} = m_n \text{ si } f(a_n)f(b_n) < 0, b_n \text{ sinon} \end{cases}$$

On peut montrer que $\forall n \in \mathbb{N}$, (a_n) est croissante, (b_n) décroissante et

$$|b_{n+1} - a_{n+1}| = \frac{1}{2^{n+1}}|b_0 - a_0| \to 0 \text{ en } + \infin$$

Ces suites sont adjacentes !

Donc, comme f est continue, $f^2(c) = \lim f(a_n)f(b_n) \leq 0$. Ainsi, $f'(c) = 0$, la limite est bel et bient la solution du problème.

> La dichotomie est **lente mais imparable**: en 10 itérations, on obtient une précision de $2^{10} \approx 10^3$

In [ ]:
```
def dicho(a,b,f,itemax):
```

```
    m=(a+b)/2
    for i in range(itemax):
        if f(m)==0:
            return m
        else:
            if f(a)*f(m)<0:
                a,b=b,m
            else:
                a=m
    return m
```

Méthode de Lagrange:

> **Amélioration de la dichotomie** qui change l'expression de m_n:

m_n devient le point d'intersection entre les abscisses et $[(a_n, f(a_n)); (b_n, f(b_n))]$.

Donc (m_n) vérifie $f(a_n)\frac{f(b_n)-f(a_n)}{b_n-a_n}(m_n - a_n)$

$$m_n = \frac{f(b_n)a_n - f(a_n)b_n}{f(b_n) - f(a_n)}$$

Mais On ne peut plus prendre $|b - a| \geq p$ comme critère car il ne tend plus vers 0.

On le remplace par $|m_{n+1} - m_n| \geq p$, qui a besoin de m_n à initier et stocker.

> Plus rapide que la dichotomie mais demande une variable en plus

Méthode de la sécante :

$$\forall n >= 0, x_{n+1} = \frac{f(x_n) - f(x_{n-1})x_n}{f(x_n) - f(x_{n-1})}$$

(classes bootstrap)

> Remarque:
>
> Pour que x_n soit bien défini, $\forall n >= 0, f(x_{n-1}) \neq f(x_n)$

Théorème :

Soit f de classe C^2 sur $]a, b[$ avec $f(\hat{x}) = 0$;

On suppose $f'(\hat{x}) \neq 0$. Si les points initiaux x_{-1} et x_0 sont choisis suffisamment proches de $\hat{x}$, la suite (x_n) converge vers $\hat{x}$ et est d'ordre $\phi = \frac{1+\sqrt{5}}{2}$

In [ ]:
```
f=lambda x: x # fonction à traiter

def secante(xo,xi,ep):
    x,y=xo,xi
    z=(f(x)*y - f(y)*x)/(f(x)-f(y))
    while abs(f(z))>ep:
        x=y
        y=z
        z=(f(x)*y - f(y)*x)/(f(x)-f(y))
    """
    d=1
```

```
    while d >ep:
        x=y
        y=z
        z=(f(x)*y - f(y)*x)/(f(x)-f(y))
        d=abs(z-y)
    """
```

Méthode de Newton simple:

> Plutôt que de résoudre $f(x) = 0$, on remplace f par son Dl_1 au point initial x_0

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + o(|x - x_0|)$$

$$\approx f(x_0) + f'(x_0)(x - x_0)$$

On considère donc

$$f(x_0) + f'(x_0)(x - x_0) = 0$$

dont la solution est

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Finalement, la suite est donc:

$$\forall n \geq 0, x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

> x_{n+1} est le point d'intersection entre la tangente de f en $(x_n, f(x_n))$ et l'axe des abscisses

> Remarque: Il faut que $\forall n \in \mathbb{N}, f'(x_n) \neq 0$,

Théorème : Soit f de classe C^2 sur $]a, b[$ avec $f(\hat{x}) = 0$;

Si $f'(\hat{x}) \neq 0$, et x_0 suffisamment proche de $\hat{x}$, la suite par newton converge vers $\hat{x}$ quadratiquemt (ordre 2)

> Très rapide ! A utiliser si on peut trouver f' facilement

In [ ]:
```
f=lambda x: ...
fp=lambda x: ...

def newt(xo,ep):
    x=xo
    while abs(f(x))>ep: # type: ignore
        x=x-f(x)/fp(x) # type: ignore
    """
    d=1
    while d>ep:
        y=x-f(x)/fp(x)
        d=abs(y-x)
        x=y
    """
```

Méthode de Newton-Rhapson vectorielle

> Méthode de résolution vectorielle pour fonctions à **deux variables** ou pouvant s'y ramener.

Soit $f : \begin{cases} \mathbb{R}^2 \to \mathbb{R}^2 \\ x, y \to fx, fy \end{cases}$ .Cherchons $\hat{x}, \hat{y}$ tels que $f(\hat{x}, \hat{y}) = 0$.

En s'inspirant de la méthode scalaire $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$, on peut tenter un développement limité à l'ordre 1, en introduisant h, le déplacement infinitésimal. f et h se déclinent suivant x ou y.

$$\begin{cases} f_x(x_{n-1} + h_x, y_{n-1} + h_y) = f_x(x_{n-1}, y_{n-1}) + h_x \frac{\partial f_x(x_{n-1}, y_{n-1})}{\partial x} + h_y \frac{\partial f_x(x_{n-1}, y_{n-1})}{\partial y} \\ f_y(x_{n-1} + h_x, y_{n-1} + h_y) = f_y(x_{n-1}, y_{n-1}) + h_x \frac{\partial f_y(x_{n-1}, y_{n-1})}{\partial x} + h_y \frac{\partial f_y(x_{n-1}, y_{n-1})}{\partial y} \end{cases}$$

On passe en matriciel pour simplifier. Intuitivement, $\begin{cases} h_x = x_n - x_{n-1} \\ h_y = y_n - y_{n-1} \end{cases}$

$$\begin{pmatrix} f_x(x_n, y_n) \\ f_y(x_n, y_n) \end{pmatrix} = \begin{pmatrix} f_x(x_{n-1}, y_{n-1}) \\ f_y(x_{n-1}, y_{n-1}) \end{pmatrix} + \begin{pmatrix} \frac{\partial f_x}{\partial x} & \frac{\partial f_x}{\partial y} \\ \frac{\partial f_y}{\partial x} & \frac{\partial f_y}{\partial y} \end{pmatrix} \begin{pmatrix} h_x \\ h_y \end{pmatrix}$$

Or, $\begin{pmatrix} \frac{\partial f_x}{\partial x} & \frac{\partial f_x}{\partial y} \\ \frac{\partial f_y}{\partial x} & \frac{\partial f_y}{\partial y} \end{pmatrix}$ est la **matrice jacobienne**, notée ici J:

$$\Leftrightarrow \begin{pmatrix} 0 \\ 0 \end{pmatrix} = J^{-1} \begin{pmatrix} f_x(x_n, y_n) \\ f_y(x_n, y_n) \end{pmatrix} + \begin{pmatrix} x_n \\ y_n \end{pmatrix} - \begin{pmatrix} x_{n-1} \\ y_{n-1} \end{pmatrix}$$

$$\Leftrightarrow \begin{pmatrix} x_n \\ y_n \end{pmatrix} = \begin{pmatrix} x_{n-1} \\ y_{n-1} \end{pmatrix} - J^{-1} \begin{pmatrix} f_x(x_{n-1}, y_{n-1}) \\ f_y(x_{n-1}, y_{n-1}) \end{pmatrix}$$

> On peut donc facilement généraliser cette méthode pour des fonctions à $n \in \mathbb{N}$ variables.

In [1]:
```python
import numpy as np

def newton_vectorielle(f,j,x:float,y:float,h=1e-2,n=500)->tuple:
    """
    Renvoie les coordonées en 0 de f et le nombre d'itérations,
    avec j sa jacobienne, x et y les points de départ,
    h la précision, et n le nombre d'itérations max.
    """
    X=np.array([x,y])
    i=0
    while i<n and np.linalg.norm(f(*X)) > h:
        b= np.array([*f(*X)])
        X-=np.linalg.solve(j(*X),b)
        i+=1
    return X,i
```

Interpolation de fonction

Méthode de Lagrange

Soit $N+1$ points de coordonées $(x_l, f(x_l))_{0 \geq l \geq N}$ avec $x_i \neq x_j$ si $i \neq j$.

L'unique polynôme interpolant ces points peut s'écrire :

$$p(x) = \sum_{i=0}^{N} f(x_i) \prod_{\substack{m=0 \\ m \neq i}}^{N} \frac{x - x_m}{x_i - x_m}$$

> **Erreur :**
>
> Soit f la fonction et p l'interpolation de f en $x_i \in [a, b]$. Si f est $C^{n+1}([a, b])$, $\exists \xi \in [\min(x, x_0), max(x, x_n)]$ tq:
>
> $$f(x) - p(x) = \prod_{i=0}^{n} \frac{(x - x_i)}{(n+1)!} f^{(n+1)}(\xi)$$
>
> Remarque: Plus de points d'interpolation ne donne pas nécessairement un meilleur résultat.

Différences finies, ou approximation de dérivée

Rappel:

Dérivée première simple

Par développement limité, on peut retrouver $f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}$.

En effet, $f(x_0) \overset{x_0}{=} f(x_0 + h) = f(x_0) + f'(x_0)h + \frac{h^2}{2} f''(x_0) + O(h^3)$

Donc $f(x_0 + h) - f(x_0) = f'(x_0)h + O(h^2)$, qui nous donne $f'(x)$ en divisant par h:

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} + O(h).$$

Dérivée première centrée

On fait simplement une Dl_2, et applique une méthode similaire:

$$f(x_0 - h) = f(x_0) - f'(x_0)h + \frac{h^2}{2} f''(x_0) + O(h^3)$$

> *Astuce:* $f(0 + h) - f(0 - h)$:
>
> $$f(x_0 + h) - f(x_0 - h) = 2f'(x_0)h + 0f''(x_0) + O(h^3)$$

Ainsi,

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} + O(h^2).$$

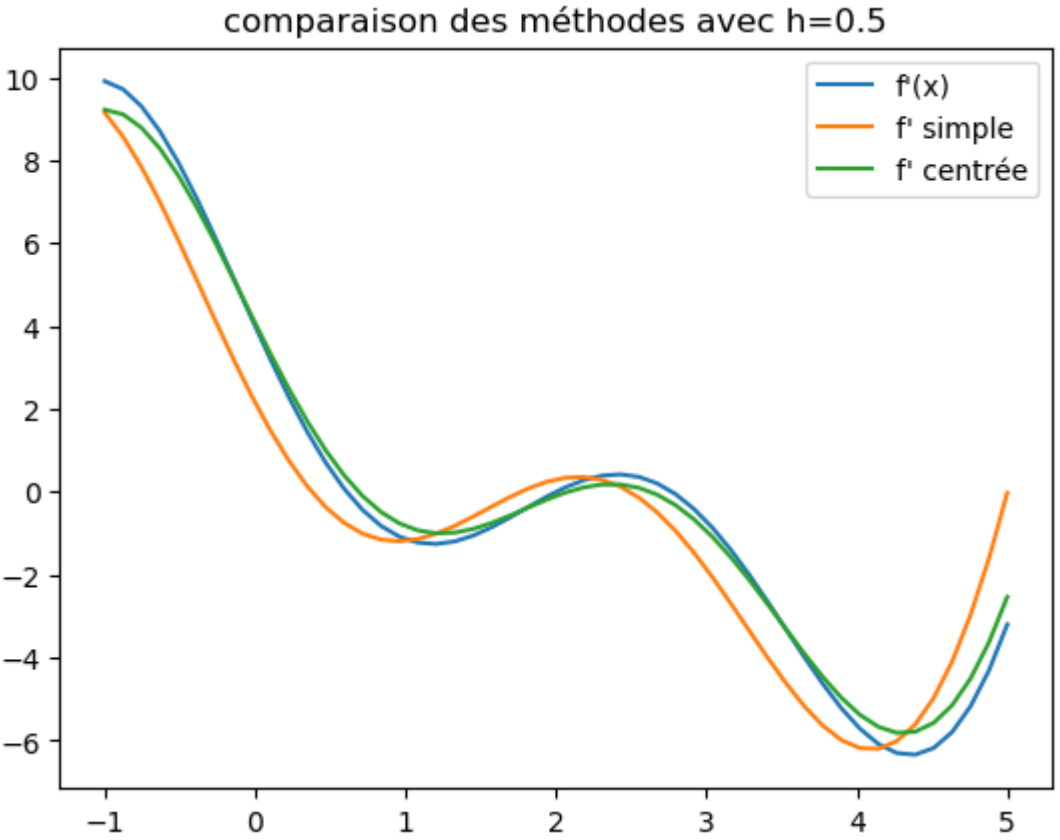
Dérivée seconde

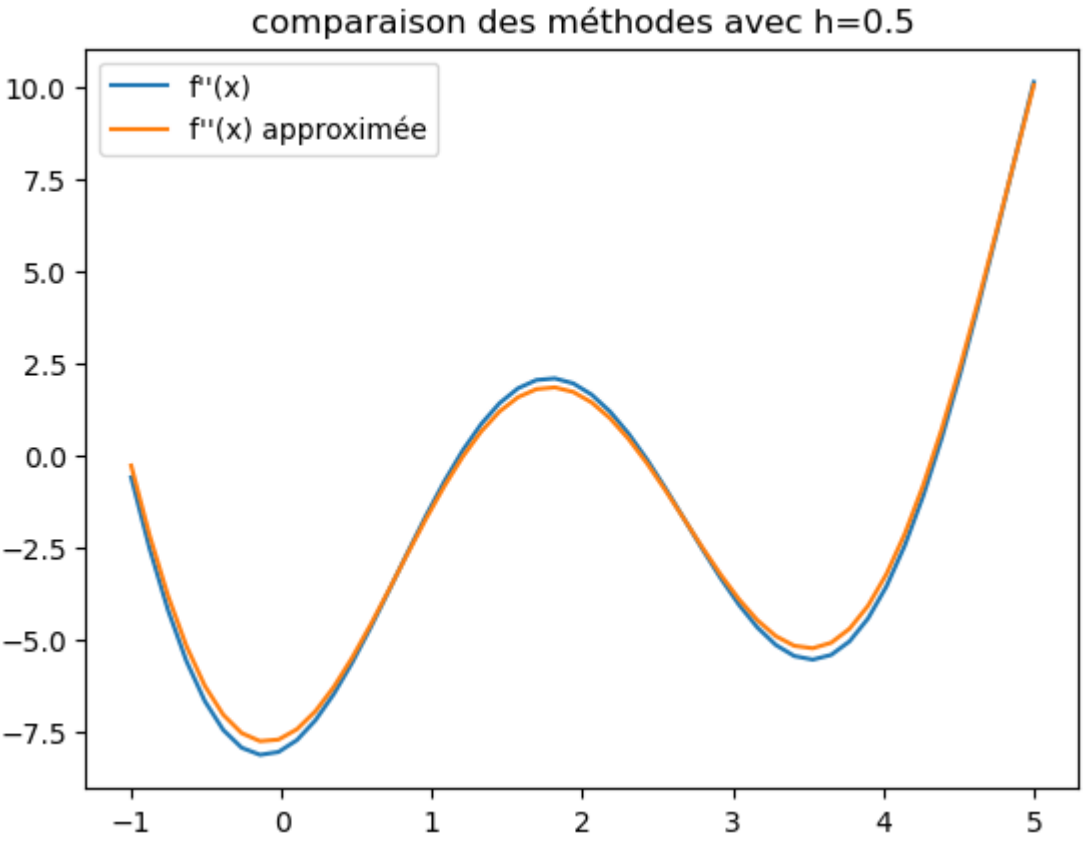
> Formule de f'' centrée non prouvée, sûrement $f'' = (f')'$, obtenue par Dl_2

$$f''(x_0) = \frac{f(x_0+h) - 2f(x_0) + f(x_0-h)}{h^2} + O(h^2).$$

In [ ]:

```
## Comparaison des approximations des dérivées:

def approx_fp(x:float,f,h=1.e-3) ->float:
    """approxime f'(x) à la précision h."""
    return (f(x+h)-f(x))/h

def approx_fp_centrée(x:float,f,h=1.e-3)->float:
    """approxime f'(x)à la précision h**2"""
    return (f(x+h)-f(x-h))/(2*h)

def approx_fpp(x:float,f,h=1.e-3)->float:
    """approxime f''(x) à la précision h**2"""
    return (f(x+h)-2*f(x)+f(x-h))/(h*h)

import numpy as np
import matplotlib.pyplot as plt

## Calcul d'erreur (modifier h, différences visibles à h=0.5):
h=1/2

f=lambda x:np.sin(x)*(x-2)**2
x=np.linspace(-1,5)
y,fp,yp,ypc,ypp=f(x),(x-2)**2*np.cos(x)+(2*x-4)*np.sin(x),approx_fp(x,f,h),approx_fp_

## comparaison des dérivées premières:
plt.plot(x,fp,label='f\'(x)')
plt.plot(x,yp,label='f\' simple')
plt.plot(x,ypc,label='f\' centrée')

plt.legend()
plt.title(f"comparaison des méthodes avec {h=}")
plt.show()
plt.close()
## comparaison des dérivées secondes:
fpp=-(x - 2)**2*np.sin(x) + 2*(2*x - 4)*np.cos(x) + 2*np.sin(x)
plt.plot(x,fpp,label='f\'\'(x)')
plt.plot(x,ypp,label='f\'\'(x) approximée')

plt.legend()
plt.title(f"comparaison des méthodes avec {h=}")
plt.show()
plt.close()
```





Quadrature, ou intégration numérique

Une quadrature utilise comme principe

$$F = \int_a^b f(x)\,dx = \lim_{n \to \infin} \sum_{k=0}^{n} a_k f(x_k)$$

avec a_k des **coefficients à déterminer** et x_k des points répartis sur $[a, b]$

Rectangles à droite

Sur $[a,b]$, on peut approximer la courbe par un rectangle de longueur $b-a$ et hauteur $f(a)$. Pour plus de précision, on discrétise à n intervalles:

$$F \approx \frac{b-a}{n}\sum_{k=0}^{n-1} f(a+k\frac{b-a}{n}) = h\sum_{k=0}^{n-1} f(a+kh), h = \frac{b-a}{n}$$

Rectangles à gauche

Même principe, la hauteur devient simplement $f(b)$, il suffit de **changer les bornes** de la somme:

$$F \approx \frac{b-a}{n}\sum_{k=1}^{n} f(a+k\frac{b-a}{n}) = h\sum_{k=1}^{n} f(a+kh)\ , h = \frac{b-a}{n}$$

Précisions et théorèmes

On démontre que le calcul approché de F converge vers l'intégrale de f quand $n \to +\infin$.

Plus précisément, si f est de classe C^1 sur $[a,b]$,

$$\epsilon = \left| \int_a^b f(x)\,dx - F_n \right| \leq \frac{(b-a)^2}{2n} max|f'(x)|$$

Trapèzes

Preuve par la méthode générale:

Soit f linéaire sur $[a,b]$. Cherchons α et β tel que $\int_a^b f(x)\,dx = \alpha f(a) + \beta f(b)$:

On évalue stratégiquement,

- Si $f(x) = 1$:

$$\int_a^b 1\,dx = b-a = \alpha f(a) + \beta f(a) = \alpha + \beta$$

- Si $f(x) = x-a$:

$$\int_a^b f(x)\,dx = \frac{(b-a)^2}{2} = \alpha \times 0 + \beta(b-a)$$

Ainsi,

$$\begin{cases} \beta = \frac{b-a}{2} \\ \alpha = (b-a) - \beta = \beta \end{cases}$$

$$\int_a^b f(x)\,dx = \frac{b-a}{2}(f(a)+f(b))$$

Si f est affine

Généralisation:

On pose $h = \frac{b-a}{n}$, $\begin{cases} a_k = a + kh \\ b_k = a + h + kh \end{cases}$. Donc, si $k \in [0, n-1]$ et n est le nombre d'intégrations:

$$F \approx \sum_{k=0}^{n} \frac{a + kh - a - kh + h}{2}(f(a + kh) + f(a + h + kh))$$

$$F \approx \sum_{k=0}^{n-1} \frac{h}{2}(f(a + kh) + f(a + h(1 + k)))$$

Précisions et théorèmes

Si f est de classe C^2 sur $[a, b]$,

$$\epsilon = \left| \int_a^b f(x)\,dx - F_n \right| \leq \frac{(b-a)^3}{12n^2} max|f''(x)|$$

Simpson

Preuve: Utilise 3 points pour interpôler la fonction

Soit f un polynôme du second degré sur $[a, b]$. Cherchons α, β et γ tel que $\int_a^b f(x)\,dx = \alpha f(a) + \beta f(\frac{a+b}{2}) + \gamma f(b)$:

On évalue stratégiquement,

- Si $f(x) = 1$:

$$\int_a^b 1\,dx = b - a = \alpha + \beta + \gamma$$

- Si $f(x) = x - a$:

$$\int_a^b x - a\,dx = \frac{(b-a)^2}{2} = \beta\frac{b-a}{2} + \gamma(b-a)$$

Ainsi,

$$\begin{cases} \frac{b-a}{2} = \frac{\beta}{2} + \gamma \\ \frac{b-a}{3} = \frac{\beta}{4} + \gamma \end{cases}$$

$$\int_a^b f(x)\,dx = \frac{b-a}{6}\left(f(a) + 4f(\frac{a+b}{2}) + f(b)\right)$$

Si f est un trinôme

Généralisation composite:

On pose $h=\frac{b-a}{n}$, $\begin{cases} a_k = a+kh \\ b_k = a+h+kh \end{cases}$. Donc, si $k \in [0, n-1]$ et n est le nombre d'intégrations:

$$F \approx \sum_{k=0}^{n-1} \frac{a+kh-a-kh+h}{6}(f(a+kh)+4f(\frac{a+kh+a+kh+h}{2})+f(a+h+kh))$$

Ce qui donne:

$$F \approx \sum_{k=0}^{n-1} \frac{h}{6} f(a+kh) + 4f\left(a+kh+\frac{h}{2}\right) + f(a+h(1+k))$$

Précision

Si f est de classe C^4 sur $[a,b]$,

$$\epsilon = \left| \int_a^b f(x)\,dx - F_n \right| \leq \frac{(b-a)^5}{2880n^4} max|f^{(4)}(x)|$$

In [ ]:
```
def int_simpson_composite(f,a:float,b:float,n:int):
    """
    Retourne l'intégrale de f de a à b par n méthodes de simpson
    """
    if a>b:
        a,b=b,a
    F=0.0
    h=(b-a)/n
    for k in range(0,n-1,1):
        F+= (h/6)*(f(a+k*h)+4*f(a+k*h+h/2)+f(a+h*(1+k))) # Pour les autres méthodes, 
    return F

# Exemple et calcul d'erreur
# Sympy permet de déterminer le résultat attendu, il n'approxime rien.

import sympy as sp # cette bibliothèque est explicitée plus bas
from numpy import cos , exp

x=sp.symbols('x')

f1: sp.Expr = sp.cos(x)*sp.exp(x)  # type: ignore
# fonction arbitraire choisie pour le test
f=lambda x: cos(x)*exp(x) # ma fonction ne semble pas compatible sympy

F=sp.integrate(f1,(x,1,2))
F1=int_simpson_composite(f,1,2,10000)

display("On intègre",f1,"Et attend",F)
epsilon = abs(sp.N(F)-F1) # Calcul d'erreur
print(f"Approximation par Simpson composite:\n{F1= :.4f};\nRésultat théorique: \nF= {
```

```
'On intègre'
```

$e^x \cos(x)$

```
'Et attend'
```

$$\frac{e^2\cos(2)}{2} - \frac{e\sin(1)}{2} - \frac{e\cos(1)}{2} + \frac{e^2\sin(2)}{2}$$

```
Approximation par Simpson composite:
F1= -0.0558;
Résultat théorique:
F= -0.0561;
Erreur:
epsilon= 0.00031
```

précision- erreur a ajouter f(x)=0: ϕ -> précision (phi'=0 -> quadratiq)... voir photos 08/04/25

Equations différentielles par Euler

`scipy.integrate.odeint(...)` permet de calculer numériquement.

Ordre 1

> On linéarise avec le taux d'accroissement et se ramène au problème de $f(x) = 0$.

On va étudier $\begin{cases} y' = \phi(y(x), x) \\ y_0 = y(x_0) \end{cases}$ sur $I = [a, b] \in \mathbb{R}$, problème de Cauchy au premier ordre.

En posant $h = \dfrac{b-a}{n} = x_{k+1} - x_k$, on a :

$$y'(x_k) = \frac{y(x_{k+1}) - y(x_k)}{x_{k+1} - x_k} \Rightarrow y_{k+1} = y_k + h\phi(y_k, x_k)$$

In [5]:
```python
import numpy as np
from typing import Callable

def edl1(f:Callable,ya:float,a:float,b:float,n:int)-> tuple[np.ndarray, np.ndarray]:
    """
    Résoud l'EDL1 y'(x)=f(y,x) (ou y'=my+p) avec y(a)=yo entre a et b. Renvoie la lis
    """
    import numpy as np
    y,x=np.zeros(n), np.linspace(a,b,n)
    y[0]=ya
    for i in range(n-1):
        y[i+1]=(y[i]+((b-a)/(n-1))*f(y[i],x[i]))
    return x, y
```

Ordre 2

On va étudier $\begin{cases} y'' = \phi(y'(x), y(x), x) \\ yp_0 = y'(x_0) \\ y_0 = y(x_0) \end{cases}$ sur $I = [a, b] \in \mathbb{R}$, problème de Cauchy au second ordre.

Méthode odeint

> `scipy.integrate.odeint` étant récursive, on peut l'utiliser simplement: `(phi, Yo, t)`

Méthode vectorielle

> Avec $y' = z$ la fonction auxiliaire, on se ramène à deux équations du premier ordre.

On pose $\begin{cases} y'(x) = z(x) \\ z' = \phi(z, y, x) \end{cases}$, puis les vecteurs $\begin{cases} Y(x) = (y(x); z(x)) \\ \phi((y; z), x) = (z; \phi(z, y, x)) \end{cases}$

Ainsi, on se ramène au problème $\begin{cases} Y'(x) = \phi(Y(x), x) \\ Y(0) = (y_0; z_0) \end{cases}$, problème de Cauchy vectoriel du premier ordre.

Méhode explicite

> On approxime la dérivée seconde

$$y''(x) = \frac{y(x+h) + y(x-h) - 2y(x)}{h^2} \Rightarrow y_{k+1} = \dots$$

Il serait intéressant de lier cet exemple à la formule générale:

$$y_{k+1} = \frac{y_k(2 + dt\frac{k}{m}) - y_{k-1} - dt^2\frac{g}{l}sin(y_k)}{1 + dt\frac{k}{m}}$$

Implémentation pour un pendule amorti, voir `TP1_equa-diff_S232` pour plus de détails.

In [ ]:
```python
import numpy as np

N=100000
gl,km=98.1,0.2

# Méthode odeint
def pendule1(N:int)->tuple[np.ndarray, np.ndarray]:
    """
    Retourne l'axe des x et y de la solution de l'équation selon la méthode odeint
    """
    from scipy import integrate
    T=np.linspace(0,30,N)

    F1=lambda y1,t: np.array([y1[1],-gl*np.sin(y1[0])-km*y1[1]])
    yo=np.array([np.pi/2,0])
    y1=integrate.odeint(F1,yo,T) # Fonction récursive !

    return T,y1[:,0]

# Méthode système
def pendule2(N:int)->tuple[np.ndarray, np.ndarray, np.ndarray]: # On ajoute z pour le
    """
    Retourne l'axe des x et y et z=y' de la solution de l'équation selon la méthode d
    """
    T=np.linspace(0,30,N)
    lt,dt=len(T),((T[-1]-T[0])/len(T))

    F2=lambda y,z:-km*z-gl*np.sin(y)
    y2,z=np.zeros(lt),np.zeros(lt)
    y2[0],z[0]=np.pi/2,0
    for i in range(lt-1):
        y2[i+1]=y2[i]+dt*z[i]
        z[i+1]=z[i]+dt*F2(y2[i],z[i])
    return T, y2, z
```

```python
# Méthode explicite
def pendule3(N:int)->tuple[np.ndarray, np.ndarray]:
    """
    Retourne l'axe des x et y de la solution de l'équation selon la méthode d'Euler ex
    """
    T=np.linspace(0,30,N)
    lt,dt=len(T),((T[-1]-T[0])/len(T))

    y3=np.zeros(lt)
    y3[0],y3[1]=np.pi/2,np.pi/2
    for i in range(1,N-1):
        y3[i+1]=(y3[i]*(2+dt*km)-y3[i-1]-dt**2*gl*np.sin(y3[i]))/(1+km*dt/1)
    return T,y3
```