Licence Semestre 2 **Portail MPI** **2019-2020**

Méthodologie

GROUPES A1, A3, A4 et A5

Support de cours

Responsable : Emmanuelle Rio emmanuelle.rio@u-psud.fr

Equipe enseignante :

MPI	**Groupe A1**	Astrid Decoene	astrid.decoene@u-psud.fr
	Groupe A3	Mikael Frosini	mikael.frosini@cea.fr
	Groupe A4	Mikael Frosini	mikael.frosini@cea.fr
	Groupe A5	Vincent Daruvar	v.daruvar@gmail.com
	Groupe A7	Marcello Civelli	marcello.civelli@u-psud.fr
	Groupe A8	Claude Pasquier	claude.pasquier@u-psud.fr
	Groupe B1	Julien Basset	julien.basset@u-psud.fr
	Groupe B2	Marcello Civelli	marcello.civelli@u-psud.fr
	Groupe B3	Laura Munteanu	laura.munteanu@cea.fr
	Groupe B4	Charis Quay	charis.quay@u-psud.fr
	Groupe B5	Claude Pasquier	claude.pasquier@u-psud.fr
PCST	**Groupe A1**	Emmanuelle Rio	emmanuelle.rio@u-psud.fr
	Groupe A2	Samrit Mainali	samrit.mainali@u-psud.fr
	Groupe A3	Carole Vouille	carole.vouille@u-psud.fr
	Groupe A4	Vincent Jeudy	vincent.jeudy@u-psud.fr
	Groupe A5	Frédéric Bouquet	frédéric.bouquet@u-psud.fr
	Groupe A6	Marie-Joëlle Ramage	marie.ramage@u-psud.fr
	Groupe A7	Emmanuelle Rio	emmanuelle.rio@u-psud.fr
	Groupe A8	Charis Quay	charis.quay@u-psud.fr

Introduction

Principes

La méthode scientifique est un mélange de conventions utilisées par les scientifiques, d'outils utilisés par les scientifiques et d'organisation propres à chacun. Le but de l'UE est

- De donner les conventions (ce qu'on s'attend à trouver dans un CR de TP, dans la rédaction d'un exercice…)
- De travailler les outils scientifiques (tracé de courbes et traitement des données, logique mathématique, dimensions, notions de programmation)
- De travailler sur votre propre organisation. Là, pas de recette miracle. Chacun est différent. L'idée est d'optimiser la manière de travailler de chacun, de voir ce qui marche…

Déroulement de l'UE

10 séances faites pour travailler sur des compétences liées à la méthodologie scientifique et au métier d'étudiant :

- Oral
- Ordres de grandeur
- Logique mathématique
- Résolution de problème
- Programmation et traitement de données

Evaluation

Oraux (coefficient 0,2) :

Chaque étudiant va passer une fois à l'oral avec un diporama et sera noté.

Travaux de groupes (coefficient 0,3)

Il y aura des travaux à rendre par groupe concernant les ordres de grandeur, la résolution de problème et la logique.

Compte rendu de TP (coefficient 0,5)

Il y a une séance sur machine et un CR de TP virtuel à rendre. Ces devoirs sont à rendre par mail à l'enseignant, deux jours après la séance. La communication par mail fait expressément partie de la note.

- Séance sur machine (fichier) coefficient **0.2** (dont 2 points/20 pour la rédaction du mail professionnel accompagnant le fichier).
- Compte rendu de TP coefficient **0.3**.

Une démarche scientifique

L’objectif de cette séance est de présenter un exemple de démarche scientifique et des outils à notre disposition pour critiquer et mettre en œuvre une telle démarche. L’exemple que nous avons choisi est celui de la question du changement climatique.

Partons de la citation suivante, extraite de BBC News (06/07/2004, un média instutionnel puisqu’il s’agit du service public de radio-télévision britannique) : "Au cours des cent dernières années, il y a eu une augmentation constante du nombre de taches solaires, en même temps que la Terre se réchauffait. Les données suggèrent que l'activité solaire influence le climat de la planète et provoque un réchauffement »

Discuter par petits groupes sur les courbes ci-après : que peut-on conclure de chaque courbe ?

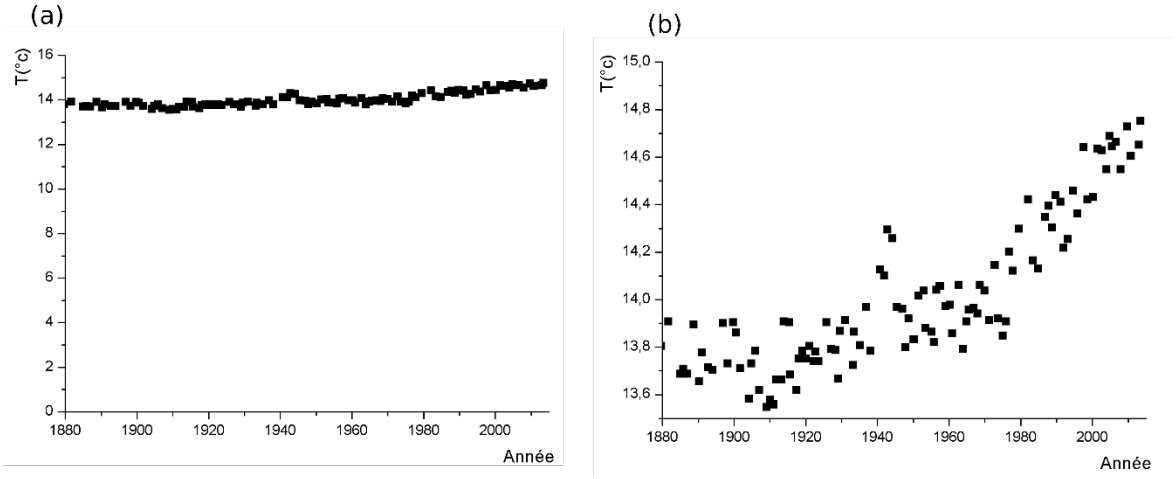


Fig 1 : Température moyenne annuelle (°c) sur la période 1880-2015 (a) avec apparition du zéro sur l’axe des ordonnées et (b) avec un zoom sur les données.

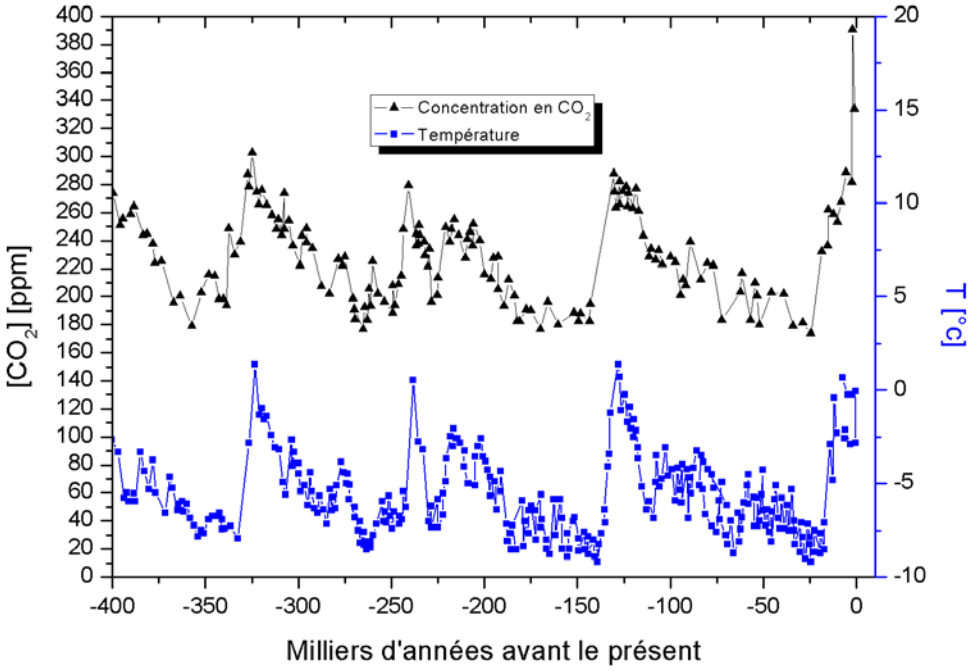


Fig 2 : Courbe de température moyenne (le 0 est placé à la température actuelle) et de concentration en CO_2 atmosphérique sur les 400 000 dernières années, d'après Petit et al., « Climate and atmosphéric history of past 420 000 years from the Vostok ice core , Antarctica », Nature (1999), 399, p. 429-436

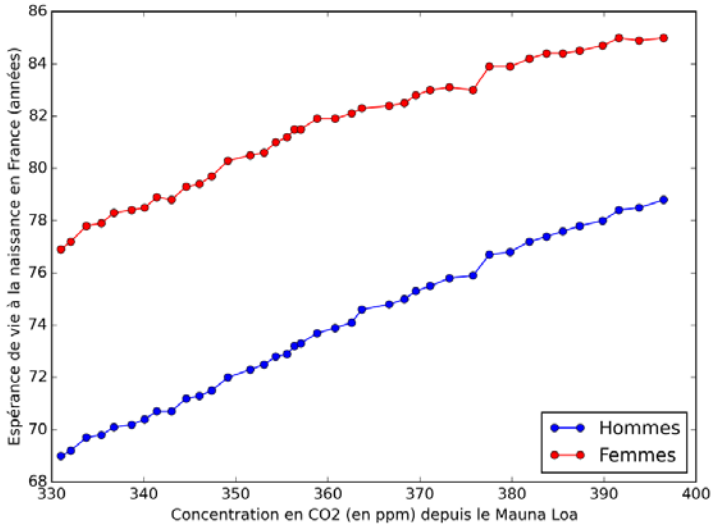


Fig 3 : espérance de vie à la naissance en France, pour les femmes et pour les hommes, en fonction de la concentration en CO_2 dans l'atmosphère

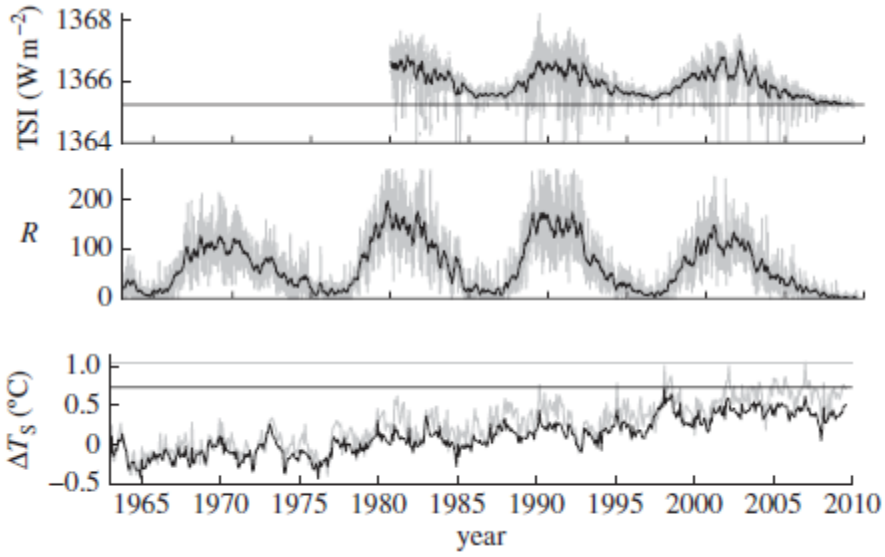


Fig 4 : TSI = Total Solar Irradiance : puissance reçue de la part du Soleil par unité de surface terrestre ; R = nombre de taches solaires observées ; ΔT_S = Global Mean Air Surface Temperature = écart de la température moyenne par rapport à la moyenne sur 1961-1990, prise comme référence ; source : Lockwood, « Solar change and climate : an update in the light of the current exceptional solar minimum », Proc. R. Soc. A (2010), 466, 303-329

Logique mathématique

Quelques éléments pour raisonner avec logique

La logique est la discipline – dont les bases ont été jetées par Aristote – permettant de comprendre ce qu'est un raisonnement, et en particulier de savoir si un raisonnement est juste ou faux. Elle est donc un prérequis essentiel à de nombreuses disciplines, telles que les mathématiques, l'informatique, la philosophie ou encore la rhétorique.

Ce document constitue une petite introduction informelle à la logique, en particulier sous l'angle mathématique. Attention, il ne s'agit en aucun cas d'un exposé rigoureux des fondements de la logique formelle.

1. Proposition

Définition 1 (Proposition). Une proposition est un énoncé déclaratif dont on peut dire s'il est vrai ou s'il est faux, indépendamment de tout contexte de lieu, de temps, ou de personne qui le prononce. Un énoncé qui est à la fois vrai et faux n'est pas une proposition.

Remarque : en mathématiques une proposition est dite vraie si elle est démontrable.

2. Les connecteurs logiques

A partir d'une ou plusieurs propositions, on peut créer de nouvelles propositions à l'aide de connecteurs logiques.

Définition 2. (Négation, « non »). La négation d'une proposition est une proposition qui est vraie si celle-ci est fausse et vice-versa. Elle peut se traduire par « non », « il est faux que . . .», « ne . . . pas ».

Exemple : (a) Le facteur n'est pas passé ce matin.
(b) Il est faux que nous soyons des privilégiés.

Définition 3. (Conjonction, « et »). La conjonction de deux propositions est une proposition qui est vraie si les deux propositions sont simultanément vraies. Elle est fausse dès que l'une au moins des deux propositions est fausse. Elle peut se traduire par « et », « mais », « bien que ».

Exemples : (a) Céline est un grand écrivain mais c'est un personnage controversé.
(b) Son point de vue est intéressant bien que je ne le partage pas.
(c) $n \in \mathbb{N} : (n \in \mathbb{Z})$ et $(n \geq 0)$.

Définition 4. (Disjonction, « ou »). La disjonction de deux propositions est une proposition qui est vraie dès que l'une au moins des deux propositions est vraie. Elle est fausse si les deux propositions sont simultanément fausses. Elle peut se traduire par « ou », «à moins que », « quoique ». Attention, il s'agit du « ou inclusif », c'est-à-dire que p et q peuvent être toutes deux vraies dans le cas où p ou q est vraie.

Exemples : (a) Ce médicament peut provoquer des troubles de l'équilibre ou de la vue.
(b) Simon viendra à moins qu'il soit malade.
(c) $n \in \mathbb{Z} : (n \geq 0)\text{ou}(n \leq 0)$.

Définition 5. (Implication, « si ... alors »). Si p et q sont deux propositions, alors l'implication « si p alors q » est une proposition qui est vraie si p est faux, ou bien si p et q sont simultanément vrais. Cette implication est fausse uniquement si p est vraie et q est fausse.

L'implication est à la base du raisonnement mathématique. En partant d'une proposition P, une démonstration aboutit à un résultat Q : si cette démonstration est faite sans erreur, alors P implique Q est vraie et on notera P $\implies$ Q (ce qui signifie que si P est vraie alors Q est vraie). Dans ce cas, on dit que P est une condition suffisante pour Q, et Q est une condition nécessaire pour P.

Exemples : (a) Fais de bonnes études, tu auras un bon travail.
(b) Si Jupin a de bons avocats alors il n'ira pas en prison.
(c) $(n \in \mathbb{N}) \Rightarrow (n \geq 0)$.

L'implication est transitive : c'est-à-dire que si p implique q et q implique r, alors p implique r :
(p $\implies$ q) et (q $\implies$ r) $\implies$ (p $\Rightarrow$ r)

Exemples :
(a) Les deux implications « Si Jean avait été là, le fusil n'aurait pas été chargé » et « Si le fusil n'avait pas été chargé, Martin ne serait pas mort » donnent « Si Jean avait été là, Martin ne serait pas mort ».
(b)$(X = a \Rightarrow Y = b)et(Y = b \Rightarrow Z = c) \Rightarrow (X = a \Rightarrow Z = c)$.

Définition 6. (Equivalence, « si et seulement si »). Si p et q sont des propositions, l'équivalence de p et q, notée p $\Leftrightarrow$ q, est vraie uniquement si p implique q et q implique p. L'équivalence peut se traduire par « p si et seulement si q », ce qui signifie « p si q » et « p seulement si q ».

En langage mathématique, si P $\Leftrightarrow$ Q est vraie on dit que P et Q sont équivalentes.

Exemples : (a) Nous irons à la plage sauf s'il pleut.
(b) $(n \in \mathbb{N}) \Leftrightarrow (n \in \mathbb{Z})$ et $(n \geq 0)$.

4. Equivalences

Associativité :
(p ou q) ou r $\Leftrightarrow$ p ou (q ou r)
(p et q) et r $\Leftrightarrow$ p et (q et r)

Commutativité :
p ou q $\Leftrightarrow$ q ou p
p et q $\Leftrightarrow$ q et p

Distributivité :
(p ou q) et r $\Leftrightarrow$ (p et r) ou (q et r)
(p et q) ou r $\Leftrightarrow$ (p ou r) et (q ou r)

Négation d'une négation : non (non p) $\Leftrightarrow$ p

Négation d'une conjonction : non (p et q) $\Leftrightarrow$ (non p) ou (non q)

Exemples :
(a) La négation de « la vie est belle mais elle est courte » est « la vie n'est pas belle à moins qu'elle

ne soit pas courte ».
(b) Le négation de $(x \geq 0)$ et $(y \leq 0)$ est $(x < 0)$ ou $(y > 0)$.

Négation d'une disjonction : non (p ou q) $\Leftrightarrow$ (non p) et (non q)

Exemples :
(a) La négation de « Un homme doit être beau ou intelligent » est « Un homme ne doit être ni beau ni intelligent. ».
(b) La négation de $(x \geq 0)$ ou $(y \leq 0)$ est $(x < 0)$ et $(y > 0)$.

Equivalence entre une implication et sa contraposée : (p $\Rightarrow$ q) $\Leftrightarrow$ (non q $\Rightarrow$ non p)

Exemples :
(a) La proposition « S'il n'y a pas d'essence dans le réservoir de la voiture, cette dernière ne démarre pas » implique sa contraposée. « La voiture démarre donc il y a de l'essence dans le réservoir de la voiture ». De même, la proposition « Si la voiture démarre alors il y a de l'essence dans le réservoir de la voiture » implique sa contraposée « Il n'y a pas d'essence dans le réservoir de la voiture, donc cette dernière ne démarre pas. »
(b) $(X \geq a \Rightarrow Y \geq b) \Leftrightarrow (Y < b \Rightarrow X < a)$

5. Quelques types de démonstrations

Modus Ponens
Il s'agit du type de démonstration la plus courante, correspondant à la déduction naturelle :

$$(\text{ p et (p } \Rightarrow \text{ q))} \Rightarrow \text{q.}$$

Un exemple classique : Tout homme est mortel. Socrate est un homme. Donc Socrate est mortel.

Contraposition ou *Modus tollens*
Il s'agit d'utiliser l'équivalence : (p $\Rightarrow$ q) $\Leftrightarrow$ (non q $\Rightarrow$ non p). Il est parfois plus facile de démontrer la contraposée d'un théorème plutôt que le sens direct.

Raisonnement par l'absurde
Ce type de démonstration consiste, pour démontrer une proposition p, à montrer que non p est faux. Elle repose sur l'équivalence suivante : (non p $\Rightarrow$ Faux) $\Leftrightarrow$ p.

Raisonner avec logique

1 Logique mathématique

Exercice 1. Compléter les pointillés par le connecteur logique qui s'impose : car, donc.
1. $x \in \mathbb{R} \quad x^2 = 4 \ldots\ldots x = 2$;
2. $z \in \mathbb{C} \quad z = \bar{z} \ldots\ldots z \in \mathbb{R}$;
3. $x \in \mathbb{R} \quad x = \pi \ldots\ldots e^{2ix} = 1$;

Compléter maintenant les mêmes pointillés par $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$.

Exercice 2. Ecrire à l'aide de quantificateurs les propositions suivantes :
1. Le carré de tout réel est positif. 2. Certains réels sont strictement supérieurs à leur carré.
3. Aucun entier n'est supérieur à tous les autres. 4. Tous les réels ne sont pas des quotients d'entier.
5. Il existe un entier multiple de tous les autres. 6. Entre deux réels distincts, il existe un rationnel.
7. Etant donné trois réels, il y en a au moins deux de même signe.

Exercice 3. Ecrire la négation des propositions suivantes :
3. Pour tout $\epsilon > 0$, il existe $q \in Q^{*+}$ tel que $0 < q < \epsilon$;
4. Pour tout $x \in \mathbb{R}$, on a $x^2 \geq 0$.

Exercice 4. Soit P, Q et R des propositions. Dans chacun des cas suivants, les propositions citées sont-elles la négation l'une de l'autre ?
1. $(P \textit{ et } Q)$; $(\textit{non } P \textit{ et non } Q)$; 2. $(P \textit{ ou } Q)$; $(\textit{non } P \textit{ et non } Q)$; 3. $(P \Rightarrow Q)$; $(\textit{non } P \Rightarrow \textit{non } Q)$.

Exercice 5. Soient f, g deux fonctions de $\mathbb{R}$ dans $\mathbb{R}$. Traduire en termes de quantificateurs les expressions suivantes :
1. f est majorée ;
2. f est bornée ;
3. f est paire ;
4. f est impaire ;
5. f ne s'annule jamais ;
6. f est périodique ;
7. f est croissante ;
8. f est strictement décroissante ;
9. f n'est pas la fonction nulle ;
10. f n'a jamais les mêmes valeurs en deux points distincts ;
11. f atteint toutes les valeurs de $\mathbb{N}$.

Exercice 6. Le but de cet exercice est de définir par contraposition la propriété P suivante pour $n \geq 2, n \in \mathbb{N}$:
P: Si l'entier $(n^2 - 1)$ n'est pas divisible par 8, alors l'entier n est pair.

1. Définir la contraposée d'une implication $A \Rightarrow B$, A et B représentant des assertions.
2. Ecrire la contraposée de la proposition P.
3. Démontrer q'un entier impair n s'écrit sous la forme $n = 4k + r$, avec $k \in N$ et $r \in \{1, 3\}$.
4. Prouver alors la contraposée de P.
5. A-t-on démontré la propriété de l'énoncé ?

Exercice 7. Soit n un entier naturel. On se donne $n + 1$ réels $x_0 \leq x_1 \leq \cdots \leq x_n$ de $[0, 1]$. On veut montrer par l'absurde la propriété P suivante :
P : Il y a deux de ces réels qui sont distants de moins de $\frac{1}{n}$.

1. Ecrire à l'aide de quantificateurs et des valeurs $x_i - x_{i-1}$ une formule logique équivalente à la propriété P.
2. Ecrire la négation de cette formule logique.
3. Rédiger une démonstration par l'absurde de la propriété.

Exercice 8. Considérons les deux affirmations suivantes :
- $\mathcal{P}_1$ « Les basketteurs de ce tournoi mesurent tous au moins deux mètres.»
- $\mathcal{P}_2$ « Un au moins de ces basketteurs fait plus de $2,40$ m.»

1. En notant B l'ensemble des basketteurs du tournoi et T(x) la taille du basketteur x, réécrire ces deux propositions sous forme mathématique en utilisant les quantificateurs $\exists$ et $\forall$.
2. Donner leurs négations.
3. Ecrire la proposition $\mathcal{P}_1$ à l'aide d'une implication. En donner une forme équivalente à l'aide d'une disjonction et vérifier qu'elle est bien équivalente à celle donnée en 1).

2 Formaliser : le langage propositionnel

Exercice 9. On considère un langage avec une constante `moi` (la personne qui parle, "je") et une constante `légumes` qui représente l'aliment correspondant, et deux relations binaires `mange` et `aime` : `mange(p,a)` représente la propriété "p mange a", et `aime(p,a)` représente la propriété "p aime a".

1. Donner des formules logiques qui correspondent aux expressions suivantes :
(a) J'aime tout ce que je mange.
(b) Il y a des choses que je n'aime pas mais que je mange quand-même.
(c) Ceux qui n'aiment pas les légumes ne mangent rien.
(d) Si tout le monde accepte de manger quelque chose qu'il n'aime pas alors je mange des légumes.

2. Exprimer sous forme de phrase en langage naturel les propriétés correspondant aux formules suivantes :
(a) $\forall x, \exists y$, `aime(x,y)` et `mange(x,y)`
(b) $\exists y, \forall x$, `aime(x,y)` et `mange(x,y)`

Exercice 10. On considère un langage avec une constante `moi` (la personne qui parle, "je"), une relation unaire et trois relations binaires :

- `triste(x)` : x est triste ;
- `pense(x,y)` : la personne x pense à y (y pouvant être une personne, un objet ou une pensée) ;

— `dit(x,y)` : la personne x dit y ;
— `x=y` : x et y sont égaux.

1. Donner des formules logiques qui correspondent aux expressions suivantes :
(a) Je ne dis jamais ce que je pense.
(b) Lorsque toutes mes pensées sont tristes, je me tais.
(c) Je ne dis jamais rien de triste, sauf si je le pense.
(d) Je ne peux pas penser à deux choses à la fois.

2. Exprimer sous forme de phrase en langage naturel les propriétés correspondant aux formules suivantes :
(a) $\forall x$, `pense(x,moi)` $\Rightarrow$ `pense(moi,x)`
(b) $\forall x$, `pense(moi,x)` et `triste(x)` $\Rightarrow$ non(`dit(moi,x)`)
(c) $\forall x, \exists y$, `pense(x,y)` et non(`dit(x,y)`)
(d) $\exists y, \forall x$, `pense(x,y)` et non(`dit(x,y)`)

3 Le langage propositionnel pour y voir plus clair

Exercice 11. Ecrire la négation des propositions suivantes, en s'aidant si nécessaire d'une formalisation en langage propositionnel :
1. Toutes les voitures sont rouges ;
2. Il existe un mouton écossais dont au moins un côté est noir ;
3. Dans toutes les écuries, tous les chevaux sont blancs ;
4. Tout étudiant se réveille au moins un jour par semaine avant 8h ;
5. Dans toutes les prisons tous les détenus détestent tous les gardiens ;
6. Tous les habitants de la rue du Havre qui ont les yeux bleus gagneront au loto et prendront leur retraite avant 50 ans.

Exercice 12. Ecrire la négation des propositions suivantes, en s'aidant si nécessaire d'une formalisation en langage propositionnel (P : Pierre aime Marie ; Q : Marie aime Pierre) :
1. Pierre et Marie s'aiment l'un l'autre.
2. Pierre et Marie ne s'aiment ni l'un ni l'autre.
3. Pierre aime Marie, mais Marie ne lui rend pas.
4. Il est faux que Marie aime Pierre et n'en soit pas aimée.
5. Pierre est aimé de Marie, mais il est faux que Pierre et Marie s'aiment mutuellement.
6. Marie n'est pas aimée de Pierre ou elle ne l'aime pas.
7. Il est faux que Pierre soit aimé de Marie et Marie de Pierre.
8. Il est faux que Marie aime Pierre ou qu'elle en soit aimée.

Exercice 13. Les raisonnements suivants sont-ils valides ? Si nécessaire, traduire les énoncés en formules propositionnelles.

1. Si les poules ont des dents, alors les poules sont des mammifères. Or les poules ne sont pas des mammifères. Donc les poules n'ont pas de dents.
2. Pour que Pierre réussisse le cours de logique, il est nécessaire et suffisant que premièrement, il assiste au cours, que, deuxièmement il cesse de bavarder avec sa voisine, et, finalement qu'il écoute le professeur. Mais s'il écoute le professeur, c'est qu'il assiste au cours et qu'il cesse de bavarder avec sa voisine. Donc, il est nécessaire et suffisant que Pierre écoute le professeur pour qu'il réussisse le cours de logique.

3. Si Pierre vient à la fête, alors Marie est triste. Si Marie est triste, Jean ne vient pas à la fête. Mais si Jean ne vient pas à la fête, Pierre ne vient pas non plus. Donc Pierre ne vient pas à la fête.

Exercice 14. Un homme qui semble divaguer déclare à toute la clientèle d'un café :

— (i) Le jour où je ne bois pas et où je dors, je ne suis pas content.
— (ii) Le jour où je bois, je ne suis pas content et je dors.
— (iii) Le jour où je ne mange pas, ou bien je ne suis pas content, ou bien je dors ou les deux.
— (iv) Le jour où je mange, ou bien je suis content, ou bien je bois ou les deux.
— (v) Aujourd'hui, je suis content.

Questions :

1. Introduire des variables propositionnelles pour représenter les principales notions et donner les formules correspondantes à chacune des affirmations précédentes.
2. On considère que toutes les affirmations précédentes sont vraies.
   (a) Par raisonnement élémentaire à partir de ces affirmations, montrer qu'il n'a pas bu.
   (b) Répondre en les justifiant par un raisonnement ou une table de vérité aux questions suivantes : a-t-il mangé ? a-t-il dormi ?

Exercice 15. *Attention aux implicites de l'esprit !*
Ce raisonnement vous semble-t-il logiquement valide ?

L'accusé n'a pu se rendre coupable du crime [C] que s'il était à New-York à 18 heures le 1er janvier [N]. Mais il a été établi qu'il était à ce moment-là à Washington [W]. Donc il n'est pas coupable du crime.

Mettez-le en formule. Vous semble-t-il logiquement valide maintenant ? Si non, que faut-il rajouter pour qu'il le soit ?

4 Trouver l'erreur

Exercice 16. Pourquoi les raisonnements suivants ne sont-ils pas logiquement valides ?

— *Ou vous êtes pour la guerre en Irak, ou vous êtes pour les terroristes. George W. Bush.*

— *On m'a dit « Si tu ne manges pas ta soupe, tu finiras en prison », or je mange ma soupe, donc je n'irai pas en prison.*

— *Nous ouvrons aujourd'hui le procès d'un ignoble meurtrier.*

— Une célèbre publicité de la Française des Jeux *«100% des gagnants auront tenté leur chance.»* était basée sur le raisonnement suivant : Tous ceux qui ont gagné ont joué. Donc si tu joues, tu gagnes.

— *Le Figaro du 28/01/10* présente en une le titre *« 9 Français sur 10 pour une réduction des dépenses publiques »*

L'article fait référence à un sondage de l'IFOP et indique que «pour faire face à la situation actuelle (crise économique, déficits publics élevés) 92% des personnes interrogées privilégient de réduire les dépenses de l'État et celles des collectivités locales (villes, départements, régions)». En consultant les détails de ce sondage sur le site de l'IFP, on constate que la question a été formulée de la façon suivante :
« Pour faire face à la situation actuelle (crise économique, déficits publics élevés) quelle solution faut-il selon-vous privilégier ?»
1/ «Réduire les dépenses de l'État et celles des collectivités locales (villes, départements, régions) »,
2/ « Augmenter les prélèvements obligatoires (impôts locaux, impôts sur le revenu) ».

5 Prêts pour résoudre des énigmes !

Exercice 17. Un voyageur perdu dans le désert arrive à une bifurcation à partir de laquelle sa piste se sépare en deux. Chaque piste peut soit mener à une oasis, soit se perdre dans un désert profond. Chaque piste est gardée par un sphinx. Les données du problème sont les suivantes :
A. Le sphinx de droite dit : "'Une au moins des deux pistes conduit à un oasis".
B. Le sphinx de gauche dit : "La piste de droite se perd dans le désert".
C. Soit les deux sphinx disent la vérité, soit ils mentent tous les deux.

Le voyageur aimerait bien savoir s'il y a un oasis au bout de l'un des deux chemins et si oui, quelle direction prendre.

Questions :

1. Introduire deux variables propositionnelles pour modéliser le problème et traduire les trois données en formules propositionnelles.
2. Résoudre l'énigme en justifiant la réponse.

Exercice 18. Une femme qui cherche un mari présente à ses prétendants 3 coffres numérotés d 1 à 3. Un seul de ces coffres contient son portrait qu'il faut découvrir. Chaque coffre comporte une inscription :
A. Le portrait est dans ce coffre.
B. Le portrait n'est pas dans ce coffre.
C. Le portrait n'est pas dans le coffre 1.

Sachant qu'une seule de ces inscriptions est vraie, en déduire dans quel coffre est caché le portrait.

Exercice 19. Un habitant X se tient à l'embranchement de deux routes. Une des deux routes va vers Paris (route gauche ou route droite). X est soit menteur (systématiquement), soit sincère (systématiquement), et ne répond que par oui ou par non. Quelle question peut-on poser à X pour déterminer la route conduisant à la capitale ?
Indication : Si P est "X dit la vérité" et Q est "la route de gauche va à Paris", constuire une formule φ sur $\{P, Q\}$ telle que la réponse de X à la question "φ est-elle vraie ?" est oui ssi Q est vraie.

Exercice 20. Les trois personnes X, Y, Z sont assises à une table. X dit la vérité, Y ment, Z est lunatique. Les personnes ne répondent que par oui ou par non. Poser deux questions à ces personnes de façon à en déduire qui est X, qui est Y et qui est Z. La première question peut être posée aux trois personnes.

PYTHON

Introduction à la programmation en langage Python

1. Présentation

Le Langage Python

Python est un langage de programmation (au même titre que le C, C++, fortran, java ...), développé en 1989. Ses principales caractéristiques sont les suivantes :
— «**open-source**» : son utilisation est gratuite et les fichiers sources sont disponibles et modifiables ;
— **simple et très lisible** ;
— doté d'une **bibliothèque de base très fournie** ;
— importante quantité de **bibliothèques disponibles** : pour le calcul scientifique, les statistiques, les bases de données, la visualisation ... ;
— **grande portabilité** : indépendant vis à vis du système d'exploitation (linux, windows, MacOS) ;
— **orienté objet** ;
— **typage dynamique** : le typage (association à une variable de son type et allocation zone mémoire en conséquence) est fait automatiquement lors de l'exécution du programme, ce qui permet une grande flexibilité et rapidité de programmation, mais qui se paye par une surconsommation de mémoire et une perte de performance ;
— présente un **support pour l'intégration d'autres langages**.

Comment faire fonctionner le code source ?

Il existe deux techniques principales pour traduire un code source en langage machine :
— la compilation : une application tierce, appelée compilateur, transforme les lignes de code en un fichier exécutable en langage machine. A chaque fois que l'on apporte une modification au programme, il faut recompiler avant de voir le résultat.
— l'interprétation : un interpréteur s'occupe de traduire ligne par ligne le programme en langage machine. Ce type de langage offre une plus grande commodité pour le développement, mais les exécutions sont souvent plus lentes.
Dans le cas de Python, on peut admettre pour commencer qu'il s'agit d'un langage interprété, qui fait appel des modules compilés. Pour les opérations algorithmiques coûteuses, le langage Python peut s'interfacer à des bibliothèques écrites en langage de bas niveau comme le langage C.

Les différentes versions

Il existe deux versions de Python : 2.7 et 3.3. La version 3.3 n'est pas une simple amélioration de la version 2.2. Attention, toutes les librairies Python n'ont pas effectué la migration de 2.7 à 3.3.

L'interpréteur

Dans un terminal, taper `python` (interpréteur classique) ou `ipython` (interpréteur plus évolué) pour accéder à un interpréteur Python. Vous pouvez maintenant taper dans ce terminal des instructions Python qui seront exécutées.

Utilisations de Python et librairies

— web : *Django, Zope, Plone,...*
— bases de données : *MySQL, Oracle,...*
— réseaux : *TwistedMatrix, PyRO, VTK,...*
— représentation graphique : *matplotlib, VTK,...*
— calcul scientifique : *numpy, scipy, ...*

Le mode programmation

Il s'agit d'écrire dans un fichier une succession d'instructions qui ne seront éffectuées que lorsque vous lancerez l'exécution du programme. Cela permet tout d'abord de sauvegarder les commandes qui pourront être utilisées ultérieurement, et d'autre part d'organiser un programme, sous-forme de fichier principal, modules, fonctions ....
Le fichier à exécuter devra avoir l'extension `.py`, et devra contenir en première ligne le chemin pour accéder au compilateur Python, ainsi que l'encodage :

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

Il pourra être exécuté en lançant dans un terminal la commande `python nomdufichier.py`.

L'environnement de programmation Spyder (sous Anaconda)

L'environnement Spyder permet de réaliser des programmes informatiques écrits avec le langage Python. Il est disponible avec la distribution Anaconda, qui présente de nombreux avantages, notamment celui d'être simple à installer. Son téléchargement se fait à l'adresse suivante : https ://store.continuum.io/cshop/anaconda/
Une fois la distribution Anaconda téléchargée et installée, on peut commencer à lancer Spyder en tapant `Spyder` dans un terminal.

Au lancement de Spyder, apparaît une fenêtre partagée en deux zones. La zone en-bas à droite, appelée console (shell en anglais), est celle où l'on peut travailler de façon interactive avec l'interprteur Python. La zone à gauche est un éditeur de texte, spécialement conçu pour écrire des programmes dans le langage Python.

2. Types et opérations de base

Le langage Python est orienté objet, c'est-à-dire qu'il permet de créer des objets, en définissant des attributs et des fonctions qui leur sont propres. Cependant, certains objets sont pré-définis dans le langage. Nous allons voir à présent les plus importants.

(a) **les nombres et les booléens**

— **entiers** (32 bits)
type : `int`
— **réels** (64 bits)
type : `float`
exemples de valeurs : `4.  5.1  1.23e-6`
— **complexes**
type : `complex`
exemples de valeurs : `3+4j  3+4J`
— **booléens** type : `bool`
exemples de valeurs : `True  False`

(b) **opérations de base**

— **affectation**

```
>>> i = 3    # i vaut 3
>>> a, k=True,  3.14159
>>> k=r=2.15
>>> x=complex(3,4)
```

— **affichage**

```
>>> i
3
>>> print(i)
3
```

— **Opérateurs addition, soustraction, multiplication, division**

`+, -,  *, \, \%,`

— **Opérateurs puissance, valeur absolue**

`**, pow, abs, \dots`

— **Opérateurs de comparaison**

`==, is, !=, is not, >, >=, <, <=`

— **Opérateurs logiques**

`or, and, not`

— **Conversion**

```
>>> int(3.1415)
3
>>> float(3)
3.
```

(c) **les chaînes de caractères**

Une chaîne de caractères (string en anglais) est un objet de la classe (ou de type) `str`. Une chaîne de caractères peut être définie de plusieurs façons :

```
 >>> "je suis une chaine"
'je suis une chaine'
>>> 'je suis une chaine'
'je suis une chaine'
```

```
>>> 'pour prendre l\'apostrophe'
'pour prendre l' apostrophe'

>>> "pour prendre l'apostrophe"
"pour prendre l'apostrophe"

>>> " " "ecrire
sur
plusieurs
lignes" " "
'ecrire\nsur\nplusieurs\nlignes'
```

— **concaténation**
On peut mettre plusieurs chaînes de caractères bout à bout avec l'opérateur binaire de concaténation, noté +.

```
>>> s = 'i vaut'
>>> i = 1
>>> print( s+i )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> print( s + " %d %s "%(i, "m."))
i vaut 1 m.
>>> print( s + ' ' + str(i))
i vaut 1
>>> print('*-' * 5)
*-*-*-*-*-
```

— **accès aux caractères**
Les caractères qui composent une chaîne sont numérotés à partir de zéro. On peut y accéder individuellement en faisant suivre le nom de la chaîne d'un entier encadré par une paire de crochets :

```
>>> "bonjour"[3]; "bonjour"[-1]
'j'
'r'
>>> "bonjour"[2:]; "bonjour"[:3]; "bonjour"[3:5]
'njour'
'bon'
'jo'
>>> "bonjour"[-1::-1];
'ruojnob'
```

— **méthodes propres**
- **len(s)** : renvoie la taille d'une chaîne,
- **s.find** : recherche une sous-chaîne dans la chaîne,
- **s.rstrip** : enlève les espaces de fin,
- **s.replace** : remplace une chaîne par une autre,
- **...**

(d) **les listes**
Une liste consiste en une succession ordonnée d'objets, qui ne doivent pas nécessairement être du même type. Les termes d'une liste sont numérotés à partir de 0 (comme pour les chaînes de caractères).

— **initialisation**

```
>>> []; list();
[]
[]
>>> [1,2,3,4,5]; ['point','triangle','quad'];
[1, 2, 3, 4, 5]
['point', 'triangle', 'quad']
>>> [1,4,'mesh',4,'triangle',['point',6]];
[1, 4, 'mesh', 4, 'triangle', ['point', 6]]
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(2,10,2)
[2, 4, 6, 8]
```

— **modification**
Contrairement aux chaînes de caractères, on peut modifier les éléments d'une liste :

```
>>> l=[1,2,3,4,5]
>>> l[2:]=[2,2,2]
>>> l
[1, 2, 2, 2, 2]
```

— **concaténation**

```
>>> [0]*7
[0, 0, 0, 0, 0, 0, 0]
>>> L1, L2 = [1,2,3], [4,5]
>>> L1
[1, 2, 3]
>>> L2
[4, 5]
>>> L1+L2
[1, 2, 3, 4, 5]
```

— **méthodes propres**
- **len(L)** : renvoie la taille de la liste L,
- **L.sort** : trie la liste L,
- **L.append** : ajoute un élément à la fin de la liste L,
- **L.reverse** : inverse la liste L,
- **L.index** : recherche un élément dans la liste L,
- **L.remove** : retire un élément de la liste L,
- **L.pop** : retire le dernier élément de la liste L,
- **...**

(e) **copie d'un objet**

```
>>> L = ['Dans','python','tout','est','objet']
>>> T = L
>>> T[4] = 'bon'
>>> T
['Dans', 'python', 'tout', 'est', 'bon']
>>> L
['Dans', 'python', 'tout', 'est', 'bon']
>>> L=T[:]
>>> L[4]='objet'
>>> T;L
['Dans', 'python', 'tout', 'est', 'bon']
```

```
['Dans', 'python', 'tout', 'est', 'objet']
```

(f) **quelques remarques importantes**

— en Python, tout est objet,
— une chaîne de caractères est immuable, tandis qu'une liste est muable,
— **type** permet de connaître le type d'un objet,
— **id** permet de connaître l'adresse d'un objet,
— **eval** permet d'évaluer une chaîne de caractères.

3. **Commentaires**

```
# ceci est un commentaire
```

4. **Noms de variables**

Python fait la distinction entre minuscules et majuscules.
Conseil : donner des noms significatifs aux variables et aux fonctions.

5. **Les structures de contrôle**

(a) **L'indentation**

Les fonctions Python n'ont pas de `begin` ou `end` explicites, ni d'accolades qui pourraient marquer là où commence et où se termine le code de la fonction. Le seul délimiteur est les deux points (« : ») et l'indentation du code lui-même. Les blocs de code (fonctions, instructions if, boucles for ou while etc) sont définis par leur indentation. L'indentation démarre le bloc et la désindentation le termine. Il n'y a pas d'accolades, de crochets ou de mots clés spécifiques. Cela signifie que les espaces blancs sont significatifs et qu'ils doivent être cohérents. Voici un exemple :

```
a = -150
if a < 0:
    print('a est negatif')

ligne d'en-tête
    première instruction du bloc
    . . .
    dernière instruction du bloc
```

Fonctionnement par blocs :

```
Bloc 1
 ...
Ligne d'en-tête :
      Bloc 2
       ...
      Ligne d'en-tête :
            Bloc 3
             ...
             Ligne d'en-tête :
      Bloc 2 (suite)
       ...
Bloc 1 (suite)
 ...
```

(b) **Le test de conditions**

Le test de condition se fait sous la forme générale suivante :

```
if < test1 > :
      < blocs d'instructions 1>
elif < test2 > :
      < blocs d'instructions 2 >
else :
      < blocs d'instructions 3 >
```

Un exemple :

```
a = 10.
if a > 0:
      print('a est strictement positif')
      if a >= 10:
            print (' a est un nombre ')
      else:
            print (' a est un chiffre ')
            a += 1
elif a is not 0:
      print(' a est strictement negatif ')
else:
      print(' a est nul ')
```

Un autre exemple :

```
L = [1, 3, 6, 8]
if 9 in L:
      print '9 est dans la liste L'
else:
      L.append(9)
```

(c) **La boucle conditionnelle**

La forme générale d'une boucle conditionnelle est

```
while < test1 > :
      < blocs d'instructions 1 >
      if < test2 > : break
      if < test3 > : continue
else :
      < blocs d'instructions 2 >
```

où l'on a utilisé les méthodes suivantes :

break : sort de la boucle sans passer par else,
continue : remonte au début de la boucle,
pass : ne fait rien.
La structure finale **else** est optionnelle. Elle signifie que l'instruction est lancée si et seulement si la boucle se termine normalement.

Quelques exemples :

— Boucle infinie :

```
while 1:
      pass
```

— y est-il premier ?

```
x = y/2
while x > 1 :
      if y%x ==0
            print (str(y)+' est facteur de '+str(x))
            break
      x = x - 1
else :
     print( str(y)+ ' est premier')
```

(d) **La boucle inconditionnelle**

La forme générale d'une boucle inconditionnelle est

```
for < cible > in < objet > :
      < blocs d'instructions 1 >
      if < test1 > : break
      if < test2 > : continue
else :
      < blocs d'instructions 2 >
```

Quelques exemples :

```
sum = 0
for i in [1, 2, 3, 4] :
      sum += 1

prod = 1
for p in range(1, 10) :
      prod *= p

s = 'bonjour'
for c in s :
      print c,

L = [ x + 10 for x in range(10) ]
```

6. **Les fonctions**

La structure générale de définition d'une fonction est la suivante

```
def < nom fonction > (arg1, arg2,... argN):
          ...
          bloc d'instructions
          ...
          return < valeur( s ) >
```

Voici quelques exemples :

```
def table7():
          n=1
          while n < 11:
          print n*7
          n+=1
```

Un fonction qui n'a pas de `return` renvoie par défaut `None`.

```
def table(base):
          n=1
          while n < 11:
          print n*base
          n+=1


def table(base, debut=0, fin=11):
          print ('Fragment de la table de multiplication par '+str(base)+' : ')
          n=debut
          l=[]
          while n < fin:
          print n*base
          l.append(n*base)
          n+=1
          return l
```

On peut également déclarer une fonction sans connaître ses paramètres :

```
>>> def f (*args, **kwargs):
          ... print(args)
          ... print(kwargs)

>>> f(1,3,'b',j=1)
(1, 3, 'b')
{'j': 1}
```

Il existe une autre façon de déclarer une fonction de plusieurs paramètres :

```
lambda argument 1, ... , argument N : expression utilisant les arguments
```

Exemple :

```
>>> f = lambda x, i :  x**i
>>> f(2,4)
16
```

7. Les modules

Un module est un fichier comprenant un ensemble de définitions et d'instructions compréhensibles par Python. Il permet d'étendre les fonctionnalités du langage. Voici tout d'abord un exemple de module permettant l'utilisation des nombres de Fibonacci.

Fichier fibo.py

```
# Module nombres de Fibonacci
def print_fib(n) :
            " " "
            ecrit la serie de Fibonacci jusqu'a n
            " " "
            a, b = 0, 1
            while b < n:
                  print(b),
```

```
                a, b = b, a + b
        print

def print_fib(n) :
        " " "
        retourne la serie de Fibonacci jusqu'a n
        " " "
        result, a, b = [], 0, 1
        while b < n:
                result.append(b),
                a, b = b, a + b
        return result
```

Utilisation du module fibo.py

```
>>> import fibo
>>> fibo.print_fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.list_fib(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

L'importation de modules

Il existe plusieurs façons d'importer un module :

— **import** fibo
— **import** fibo **as** f
— **from** fibo **import** print_ fib, list_ fib
— **from** fibo **import** * (importe tous les noms sauf les variables et les fonctions privées)

Le module math

Ce module fournit un ensemble de fonctions mathématiques pour les réels :

```
pi
sqrt
cos, sin, tan, acos, ...
...
```

Tapez `import math` pour importer le module, puis `help(math)` pour avoir toutes les informations sur le module : chemin, fonctions, constantes.

8. **Le module NumPy**

NumPy est un outil performant pour la manipulation de tableaux à plusieurs dimensions. Il ajoute en effet le type `array`, qui est similaire à une liste, mais dont tous les éléments sont du même type : des entiers, des flottants ou des booléens.

Le module NumPy possède des fonctions basiques en algèbre linéaire, ainsi que pour les transformées de Fourier.

(a) **Création d'un tableau dont on connaît la taille**

```
>>> import numpy as np
>>> a = np.zeros(4)
>>> a
```

```
array([ 0., 0., 0., 0.])
>>> nx, ny = 2, 2
>>> a=np.zeros((nx,ny))
>>> a
array( [ [ 0.,  0. ] ,
         [ 0.,  0.] ] )
```

Un tableau peut être multidimensionnel, comme ici, de dimension 3 :

```
>>> a=np.zeros((nx,ny,3))
>>> a
array( [ [ [ 0.,  0.,  0. ],
           [ 0.,  0.,  0. ] ],

         [ [ 0.,  0.,  0. ],
           [ 0.,  0.,  0. ] ] ] )
```

Mais nous nous limiterons dans ce cours aux tableaux uni et bi-dimensionnels.

Il existe également les fonctions `np.ones`, `np.eye`, `np.identity`, `np.empty`, ...
Par exemple `np.identity(3)` est la matrice identité d'ordre 3, `np.empty` est un tableau vide, `np.ones(5)` est le vecteur `[1 1 1 1 1]` et `np.eye(3,2)` est la matrice à 3 lignes et 2 colonnes contenant des 1 sur la diagonale et des zéro partout ailleurs.

Par défaut les éléments d'un tableau sont des `float` (un réel en double précision) ; mais on peut donner un deuxième argument qui précise le type (`int, complex`, `bool`, ...). Exemple :

```
>>> np.eye(2, dtype=int)
```

(b) **Création d'un tableau avec une séquence de nombre**

La fonction `linspace(premier, dernier, n)` renvoie un tableau unidimensionnel commençant par `premier`, se terminant par `dernier` avec `n` éléments régulièrement espacés. Une variante est la fonction `arange` :

```
>>> a = np.linspace(-4, 4, 9)
>>> a
array([-4., -3., -2., -1., 0., 1., 2., 3., 4.])
>>> a = np.arange(-4, 4, 1)
>>> a
array([-4, -3, -2, -1, 0, 1, 2, 3])
```

On peut convertir une ou plusieurs listes en un tableau via la commande `array` :

```
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
>>> b = np.array(range(10))
>>> b
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> L1, L2 = [1, 2, 3], [4, 5, 6]
>>> a = np.array([L1, L2])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
```

(c) **Création d'un tableau à partir d'une fonction**

```
>>> def f(x, y):
... return x**2 + np.sin(y)
...
>>> a = np.fromfunction(f, (2, 3))
>>> a
array([[ 0. , 0.84147098, 0.90929743],
[ 1. , 1.84147098, 1.90929743]])
```

La fonction `f` a ici été appliquée aux valeurs $x = [0, 1]$ et $y = [0, 1, 2]$, le résultat est

$$[\ [f(0,0), f(0,1), f(0,2)], [f(1,0), f(1,1), f(0,2)]\ ].$$

(d) **Manipulations d'un tableau**

i. Caractéristiques d'un tableau
   - **a.shape** : retourne les dimensions du tableau
   - **a.dtype** : retourne le type des éléments du tableau
   - **a.size** : retourne le nombre total d'éléments du tableau
   - **a.ndim** : retourne la dimension du tableau (1 pour un vecteur, 2 pour une matrice)

ii. Indexation d'un tableau

Comme pour les listes et les chaines de caractères, l'indexation d'un vecteur (ou tableau de dimension 1) commence à 0. Pour les matrices (ou tableaux de dimension 2), le premier index se réfère à la ligne, le deuxième à la colonne. Quelques exemples :

```
>>> L1, L2 = [1, -2, 3], [-4, 5, 6]
>>> a = np.array([L1, L2])
>>> a[1, 2]  #  extrait l'élément en 2ème ligne et 3ème colonne
6

>>> a[:, 1]  # extrait toute la deuxième colonne
array([-2, 5])

>>> a[1,0:2:2] # extrait les éléments d'indice [dbut=0, pas=2, fin=2]
               # de la première ligne, le dernier élément n'est pas inclus
array([-4])

>>> a[:, -1:0:-1]  # extrait les éléments d'indice [dbut=-1, pas=-1, fin=0]
                   # de la première colonne, le dernier élément n'est pas inclus
array([[3, -2],
[6, 5]])

>>> a[a < 0]    # extrait les éléments négatifs
array([-2, -4])
```

iii. Copie d'un tableau

Un tableau est un objet. Si l'on affecte un tableau A à un autre tableau B, A et B font référence au même objet. En modifiant l'un on modifie donc automatiquement l'autre. Pour éviter cela on peut faire une copie du tableau A dans le tableau B moyennant la fonction `copy()`. Ainsi, A et B seront deux tableaux identiques, mais ils ne feront pas référence au même objet.

```
>>> a = np.linspace(1, 5, 5)
>>> b = a
>>> c = a.copy()
>>> b[1] = 9
>>> a; b; c;
array([ 1., 9., 3., 4., 5.])
array([ 1., 9., 3., 4., 5.])
array([ 1., 2., 3., 4., 5.])
```

Une façon de faire une copie de tableau sans utiliser la méthode `copy()` est la suivante :

```
>>> d = np.zeros(b.shape, a.dtype)
>>> d[:] = b
>>> d
array([ 1., 2., 3., 4., 5.])
```

iv. Redimensionnement d'un tableau

Voici comment modifier les dimensions d'un tableau :

```
>>> a = np.linspace(1, 10, 10)
>>> a
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
>>> a.shape = (2, 5)
>>> a
array([[ 1., 2., 3., 4., 5.],
[ 6., 7., 8., 9., 10.]])
>>> a.shape = (a.size,)
>>> a
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
>>> a.reshape(2, 5)
array([[ 1., 2., 3., 4., 5.],
[ 6., 7., 8., 9., 10.]])
>>> a
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
```

Attention : la fonction `reshape` ne modifie pas l'objet, elle crée une nouvelle vue.
La fonction `flatten` renvoie une vue d'un tableau bi-dimensionnel en tableau uni-dimensionnel.

```
>>> a = np.linspace(1, 10, 10)
>>> a.shape = (2, 5)
>>> a
array([[ 1., 2., 3., 4., 5.],
[ 6., 7., 8., 9., 10.]])
>>> a.flatten
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
```

v. Boucles sur les tableaux

```
>>> a = np.zeros((2, 3))
>>> for i in range(a.shape[0]):
...     for j in range(a.shape[1]):
...         a[i, j] = (i + 1)*(j + 1)
>>> print a
[[ 1. 2. 3.]
[ 2. 4. 6.]]
```

```
>>> for e in a:
... print e
[ 1. 2. 3.]
[ 2. 4. 6.]
```

(e) **Calculs avec des tableaux**

Pour bien des calculs il est possible d'éviter d'utiliser des boucles (qui sont très consommatrices de temps de calcul). On peut faire directement les calculs sur des tableaux. Ceux-ci sont faits via des fonctions C, un langage de programmation bas niveau, et sont donc plus rapides. C'est ce qu'on appelle «vectoriser» un programme.

Voici un exemple, dans lequel on veut calculer $b = 3a - 1$:

```
>>> a = np.linspace(0, 1, 1E+06)
>>> %timeit b = 3*a -1
100 loops, best of 3: 10.7 ms per loop
>>> b = np.zeros(1E+06)
>>> %timeit for i in xrange(a.size): b[i] = 3*a[i] - 1
10 loops, best of 3: 1.31 s per loop
```

Voici un autre exemple de calcul fait directement sur un tableau :

```
>>> def f1(x):
...   return np.exp(-x*x)*np.log(1+x*np.sin(x))
...
>>> x = np.linspace(0, 1, 1e6)
>>> a = f1(x)
```

(f) **Produit matriciel**

Le calcul matriciel se fait avec la fonction `dot()`.
Attention, si le deuxième argument est un vecteur ligne, il sera transformé si besoin en vecteur colonne, par transposition. Ceci est dû au fait qu'il est plus simple de définir un vecteur ligne, par exemple `v=np.array([0,1,2])`, qu'un vecteur colonne, ici `v=np.array([[0],[1],[2]])`.
Exemple :

```
>>> A=np.array([[1,1,1],[0,1,1],[0,0,1]]
>>> v=np.array([0,1,2])
>>> np.dot(v,A)
array([0, 1, 3])
>>> np.dot(A,v)
array([3, 3, 2])
```

On peut aussi transposer préalablement un vecteur ligne :

```
>>> v=np.array([[0,1,2]])
>>> v=np.transpose(v)
>>> v
```

Attention, l'opération `A**2` correspond à une élévation au carré terme à terme. Pour lever une matrice au carré il faut taper `np.dot(A,A)`.

Pour lever une matrice carrée à une puissance n il faut faire une boucle :

```
>>> B=A.copy()
>>> for i in range(1,n):
...   B = np.dot(A,B)
```

A la sortie de cette boucle B contiendra A^n.

(g) **Produit scalaire**

Il faut utiliser la fonction `vdot()`.

(h) **Le sous-module linalg**

Ce module propose des méthodes numériques pour inverser une matrice, calculer son déterminant, résoudre un système linéaire ...

9. **Importer et exporter des données avec Python**

(a) Méthode manuelle

Pour ouvrir et fermer un fichier nommé `nomfichier.txt` on crée une variable `fichier` de type `file` à travers laquelle on pourra accéder au fichier.

```
fichier = open ('nomfichier.txt','r')
```

Le paramètre «r» indique qu'on accède au fichier en mode lecture. Pour accéder au fichier en mode écriture il faut utiliser le paramètre «w». A la fin des opérations sur le fichier, on le ferme en appelant la fonction `close` :

```
fichier.close()
```

Voici trois façons différentes de parcourir les lignes d'un fichier ouvert en mode écriture :

```
for ligne in fichier :
```

La variable ligne est une chaîne de caractère qui prendra les valeurs successives des lignes du fichier de lecture (sous la forme d'une chaîne de caractères).

```
ligne = fichier.readline()
```

A chaque fois que la fonction `readline` de fichier est appelée, la ligne suivante du fichier est lue et constitue la valeur de retour de la fonction.

```
lignes = fichier.readlines ()
```

Cette fonction lit toutes les lignes d'un coup et la valeur de retour est une liste de chaînes de caractères. Attention, si le fichier est gros cette méthode est à éviter car elle bloque instantannément une grande place mémoire. Il est dans ce cas préférable d'utiliser une méthode de lecture caractère par caractère (grâce à la fonction `read()`) ou ligne par ligne.

Enfin, la fonction `fichier.write(chaine)` permet d'écrire la chaîne de caractères en argument dans un fichier ouvert en mode écriture.

(b) Méthode automatisée

La fonction `numpy.loadtxt` lit les lignes du fichier en argument et renvoie une matrice dont les lignes correspondent aux lignes du fichier et les colonnes aux différentes valeurs délimitées par un espace.

Parmi les options les plus utiles, `skiprows` est le nombre de lignes à ne pas lire dans l'en–tête (par défaut 0) et `delimiter` remplace le délimiteur (espace) par la chaîne que l'on souhaite.

Exemple : le fichier `data.dat` contient les lignes :

```
chat, chien
3,4
2,5
1,8
```

On veut supprimer la première ligne qui contient le nom des colonnes mais pas de valeurs numériques, et on déclare que le délimiteur des valeurs est la virgule :

```
t = np.loadtxt ("data.dat", skiprows=1, delimiter=',')
t
```

renvoie

```
array([ [ 3., 4.],
    [ 2., 5.],
    [ 1., 8.]])
```

10. Représentation discrète d'une fonction avec Matplotlib

Pour tracer la courbe représentative d'une fonction f avec Matplotlib, il faut tout d'abord la "discrétiser" ; c'est-à-dire définir une liste de points $(x_i, f(x_i))$ qui va permettre d'approcher le graphe de la fonction par une ligne brisée. Bien sûr, plus on augmente le nombre de points, plus la ligne brisée est "proche" du graphe (en un certain sens).

Plus précisément, pour représenter le graphe d'une fonction réelle f définie sur un intervalle $[a, b]$, on commence par construire un vecteur X, discrétisant l'intervalle $[a, b]$, en considérant un ensemble de points équidistants dans $[a, b]$. Pour ce faire, on se donne un naturel N *grand* et on considère `X=numpy.linspace(a,b,N)` (ou, ce qui revient au même, on pose $h = \frac{b-a}{N-1}$ et on considère
`X=numpy.arange(a,b+h,h)`).

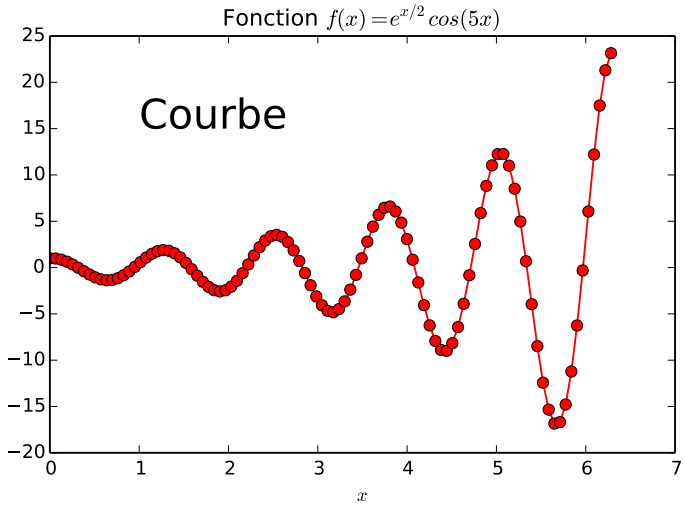
On construit ensuite le vecteur Y dont les composantes sont les images des X_i par f et on trace la ligne brise reliant tous les points (X_i, Y_i).

(a) La fonction `plot` du module matplotlib.pyplot

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

x = np.linspace(0., 2*np.pi, 100)
plt.plot(x, np.exp(x/2)*np.cos(5*x), '-ro')

plt.title('Fonction $f(x)=e^{x/2} cos(5x)$')
plt.xlabel('$x$')
plt.text(1, 15, 'Courbe', fontsize=22)
plt.savefig('1D_exemple.pdf')
plt.show()
```

Cette fonction permet de tracer des lignes brisées. Si X et Y sont deux vecteurs-lignes (ou vecteurs-colonnes) de même taille n, alors `plt.plot(X,Y)` permet de tracer la ligne brisée qui relie les points de coordonnées (X(i),Y(i)) pour $i = 1, \ldots, n$.

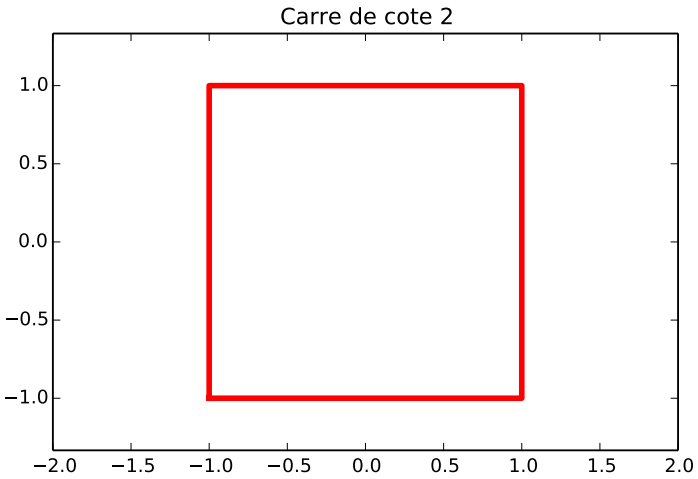
Pour connaître toutes les options, le mieux est de se référer à la documentation de Matplotlib.



Exemple : tracé d'un carré

Coordonnées des sommets : $X = (-1; 1; 1; -1; -1)$ et $Y = (-1; -1; 1; 1; -1)$.

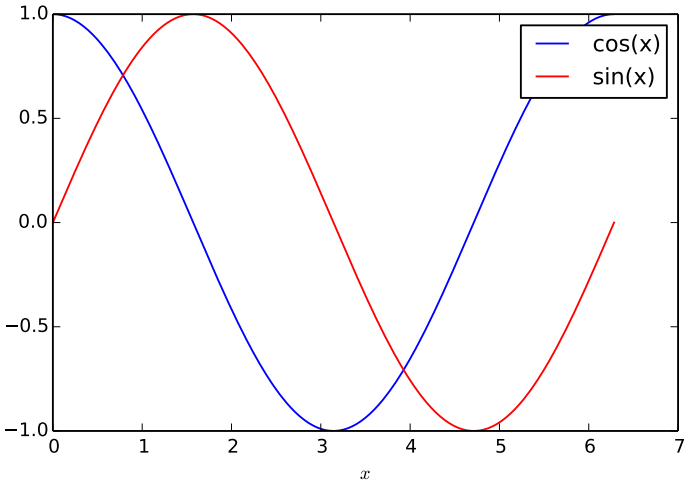
```
X=np.array([-1, 1, 1,-1,-1])
Y=np.array([-1, -1, 1,1,-1])
plt.plot(X,Y,'r',lw=3)
plt.axis('equal')
plt.axis([-2,2,-2,2])

plt.title('Carre de cote 2')
plt.savefig('carre_exemple.pdf')
plt.show()
```



(b) **Tracé de plusieurs lignes brisées**

On peut superposer plusieurs courbes dans le même graphique.

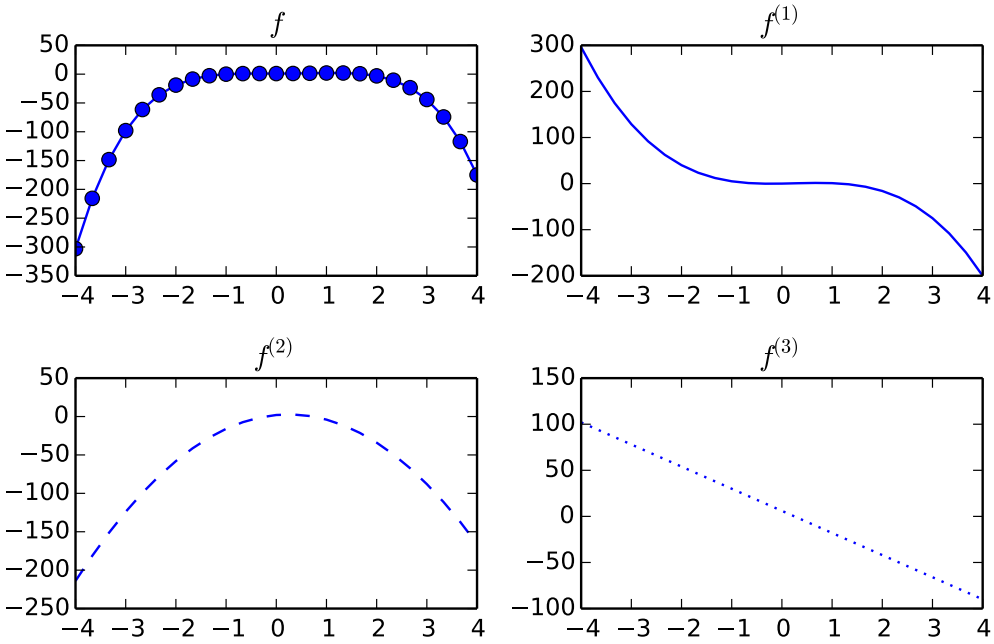
```
import matplotlib.pyplot as plt
import numpy as np
X = np.linspace(0, 2*np.pi, 256)
Ycos = np.cos(X)
Ysin = np.sin(X)
plt.plot(X,Ycos,'b')
plt.plot(X,Ysin,'r')
pp.show()
```



(c) Création de plusieurs graphiques dans une même fenêtre

La commande `subplot` permet de découper la fenêtre graphique en plusieurs sous-fenêtres. Voici un exemple d'utilisation de cette commande.

```
x = np.linspace(-4., 4., 25)
plt.subplot(2, 2, 1)
plt.plot(x, -x**4 + x**3 + x**2 + 1, 'o-')
plt.title("$f$")
plt.subplot(2, 2, 2)
plt.plot(x, -4*x**3 + 3*x**2 + 2*x, '-')
plt.title("$f^{(1)}$")
plt.subplot(2, 2, 3)
plt.plot(x, -12*x**2 + 6*x + 2, '--')
plt.title("$f^{(2)}$")
plt.subplot(2, 2, 4)
plt.plot(x, -24*x + 6, ':')
plt.title("$f^{(3)}$")
plt.tight_layout()
plt.savefig("1D_subplot2.pdf")
plt.show()
```



Matplotlib dispose d'autres commandes de tracé comme `loglog, polar`...

Le module NumPy

1. Présentation

NumPy est un outil performant pour la manipulation de tableaux à plusieurs dimensions. Il ajoute en effet le type `array`, qui est similaire à une liste, mais dont tous les éléments sont du même type : des entiers, des flottants ou des booléens.

Le module NumPy possède des fonctions basiques en algèbre linéaire, ainsi que pour les transformées de Fourier.

2. Création d'un tableau dont on connaît la taille

```
>>> import numpy as np
>>> a = np.zeros(4)
>>> a
array([ 0., 0., 0., 0.])
>>> nx, ny = 2, 2
>>> a=np.zeros((nx,ny))
>>> a
array( [ [ 0.,  0. ] ,
         [ 0.,  0.] ] )
```

Un tableau peut être multidimensionnel, comme ici, de dimension 3 :

```
>>> a=np.zeros((nx,ny,3))
>>> a
array( [ [ [ 0.,  0.,  0. ],
           [ 0.,  0.,  0. ] ],

         [ [ 0.,  0.,  0. ],
           [ 0.,  0.,  0. ] ] ] )
```

Mais nous nous limiterons dans ce cours aux tableaux uni,bi-dimensionnels.

Il existe également les fonctions `np.ones`, `np.eye`, `np.identity`, `np.empty`, ...
Par exemple `np.identity(3)` est la matrice identité d'ordre 3, `np.empty` est un tableau vide, `np.ones(5)` est le vecteur `[1 1 1 1 1]` et `np.eye(3,2)` est la matrice à 3 lignes et 2 colonnes contenant des 1 sur la diagonale et des zéro partout ailleurs.

Par défaut les éléments d'un tableaux sont des `float` (un réel en double précision) ; mais on peut donner un deuxième argument qui précise le type (`int, complex`, `bool`, ...). Exemple :

```
>>> np.eye(2, dtype=int)
```

3. Création d'un tableau avec une séquence de nombre

La fonction `linspace(premier, dernier, n)` renvoie un tableau unidimensionnel commençant

par `premier`, se terminant par `dernier` avec `n` éléments régulièrement espacés. Une variante est la fonction `arange` :

```
>>> a = np.linspace(-4, 4, 9)
>>> a
array([-4., -3., -2., -1., 0., 1., 2., 3., 4.])
>>> a = np.arange(-4, 4, 1)
>>> a
array([-4, -3, -2, -1, 0, 1, 2, 3])
```

On peut convertir une ou plusieurs listes en un tableau via la commande `array` :

```
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
>>> b = np.array(range(10))
>>> b
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> L1, L2 = [1, 2, 3], [4, 5, 6]
>>> a = np.array([L1, L2])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
```

4. **Création d'un tableau à partir d'une fonction**

```
>>> def f(x, y):
... return x**2 + np.sin(y)
...
>>> a = np.fromfunction(f, (2, 3))
>>> a
array([[ 0. , 0.84147098, 0.90929743],
[ 1. , 1.84147098, 1.90929743]])
```

La fonction `f` a ici été appliquée aux valeurs $x = [0, 1]$ et $y = [0, 1, 2]$, le résultat est

$$[\ [f(0,0), f(0,1), f(0,2)], [f(1,0), f(1,1), f(0,2)]\ ].$$

5. **Manipulations d'un tableau**

   (a) Caractéristiques d'un tableau
   - **a.shape** : retourne les dimensions du tableau
   - **a.dtype** : retourne le type des éléments du tableau
   - **a.size** : retourne le nombre total d'éléments du tableau
   - **a.ndim** : retourne la dimension du tableau (1 pour un vecteur, 2 pour une matrice)

   (b) Indexation d'un tableau

   Comme pour les listes et les chaines de caractères, l'indexation d'un vecteur (ou tableau de dimension 1) commence à 0. Pour les matrices (ou tableaux de dimension 2), le premier index se réfère à la ligne, le deuxième à la colonne. Quelques exemples :

```
>>> L1, L2 = [1, -2, 3], [-4, 5, 6]
>>> a = np.array([L1, L2])
```

```
>>> a[1, 2]  #  extrait l'élément en 2ème ligne et 3ème colonne
6

>>> a[:, 1]  # extrait toute la deuxième colonne
array([-2, 5])

>>> a[1,0:2:2] # extrait les éléments d'indice [début=0, pas=2, fin=2]
               # de la première ligne, le dernier élément n'est pas inclus
array([-4])

>>> a[:, -1:0:-1]  # extrait les éléments d'indice [début=-1, pas=-1, fin=0]
                   # de la première colonne, le dernier élément n'est pas inclus
array([[3, -2],
[6, 5]])

>>> a[a < 0]    # extrait les éléments négatifs
array([-2, -4])
```

(c) Copie d'un tableau

Un tableau est un objet. Si l'on affecte un tableau A à un autre tableau B, A et B font référence au même objet. En modifiant l'un on modifie donc automatiquement l'autre. Pour éviter cela on peut faire une copie du tableau A dans le tableau B moyennant la fonction `copy()`. Ainsi, A et B seront deux tableaux identiques, mais ils ne feront pas référence au même objet.

```
>>> a = np.linspace(1, 5, 5)
>>> b = a
>>> c = a.copy()
>>> b[1] = 9
>>> a; b; c;
array([ 1., 9., 3., 4., 5.])
array([ 1., 9., 3., 4., 5.])
array([ 1., 2., 3., 4., 5.])
```

Une façon de faire une copie de tableau sans utiliser la méthode `copy()` est la suivante :

```
>>> d = np.zeros(b.shape, a.dtype)
>>> d[:] = b
>>> d
array([ 1., 2., 3., 4., 5.])
```

(d) Redimensionnement d'un tableau

Voici comment modifier les dimensions d'un tableau :

```
>>> a = np.linspace(1, 10, 10)
>>> a
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
>>> a.shape = (2, 5)
>>> a
array([[ 1., 2., 3., 4., 5.],
[ 6., 7., 8., 9., 10.]])
>>> a.shape = (a.size,)
>>> a
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
```

```
>>> a.reshape(2, 5)
array([[ 1., 2., 3., 4., 5.],
[ 6., 7., 8., 9., 10.]])
>>> a
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
```

Attention : la fonction `reshape` ne modifie pas l'objet, elle crée une nouvelle vue.
La fonction `flatten` renvoie une vue d'un tableau bi-dimensionnel en tableau uni-dimensionnel.

```
>>> a = np.linspace(1, 10, 10)
>>> a.shape = (2, 5)
>>> a
array([[ 1., 2., 3., 4., 5.],
[ 6., 7., 8., 9., 10.]])
>>> a.flatten
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
```

(e) Boucles sur les tableaux

```
>>> a = np.zeros((2, 3))
>>> for i in range(a.shape[0]):
...     for j in range(a.shape[1]):
...         a[i, j] = (i + 1)*(j + 1)
>>> print a
[[ 1. 2. 3.]
[ 2. 4. 6.]]

>>> for e in a:
... print e
[ 1. 2. 3.]
[ 2. 4. 6.]
```

6. **Calculs avec des tableaux**

Pour bien des calculs il est possible d'éviter d'utiliser des boucles (qui sont très consommatrices de temps de calcul). On peut faire directement les calculs sur des tableaux. Ceux-ci sont faits via des fonctions C, un langage de programmation bas niveau, et sont donc plus rapides. C'est ce qu'on appelle «vectoriser» un programme.

Voici un exemple, dans lequel on veut calculer $b = 3a - 1$:

```
>>> a = np.linspace(0, 1, 1E+06)
>>> %timeit b = 3*a -1
100 loops, best of 3: 10.7 ms per loop
>>> b = np.zeros(1E+06)
>>> %timeit for i in xrange(a.size): b[i] = 3*a[i] - 1
10 loops, best of 3: 1.31 s per loop
```

Voici un autre exemple de calcul fait directement sur un tableau :

```
>>> def f1(x):
...   return np.exp(-x*x)*np.log(1+x*np.sin(x))
...
>>> x = np.linspace(0, 1, 1e6)
>>> a = f1(x)
```

7. **Produit matriciel**

Le calcul matriciel se fait avec la fonction `dot()`.
Attention, si le deuxième argument est un vecteur ligne, il sera transformé si besoin en vecteur colonne, par transposition. Ceci est du au fait qu'il est plus simple de définir un vecteur ligne, par exemple `v=np.array([0,1,2])`, qu'un vecteur colonne, ici `v=np.array([[0],[1],[2]])`.
Exemple :

```
>>> A=np.array([[1,1,1],[0,1,1],[0,0,1]]
>>> v=np.array([0,1,2])
>>> np.dot(v,A)
array([0, 1, 3])
>>> np.dot(A,v)
array([3, 3, 2])
```

On peut aussi transposer préalablement un vecteur ligne :

```
>>> v=np.array([[0,1,2]])
>>> v=np.transpose(v)
>>> v
```

Attention, l'opération `A**2` correspond à une élévation au carré terme à terme. Pour élever une matrice au carré il faut taper `np.dot(A,A)`.

Pour élever une matrice carrée à une puissance n il faut faire une boucle :

```
>>> B=A
>>> for i in range(1,n):
...   B = np.dot(A,B)
```

A la sortie de cette boucle B contiendra A^n.

8. **Produit scalaire**

Il faut utiliser la fonction `vdot()`.

9. `all`, `any` **et** `where`

```
>>> a = np.array([[1, 2, 4], [5, 6, 9]])
>>> np.any(a > 2)
True
>>> np.all(a > 2)
False
>>> np.where(a <= 1)
(array([0]), array([0]))
>>> np.where(a > 1)
(array([0, 0, 1, 1, 1]), array([1, 2, 0, 1, 2]))
```

10. **Autres opérations sur un tableau**

- `a.argmax` : renvoie un tableau d'indices des valeurs maximales selon un axe
- `a.max` : renvoie le maximum
- `a.astype` : renvoie un tableau de a converti sous un certain type
- `a.conj` : renvoie le conjugué de a
- `a.sum` : renvoie la somme des éléments de a
- `a.prod` : renvoie le produit des éléments de a
- `a.transpose` : renvoie le transposé de a

11. **Le sous-module linalg**

(a) Inversion d'une matrice carrée : la fonction `inv()`

Exemple :

```
>>> A=np.array([[1,1,1],[0,1,1],[0,0,1]]
>>> np.linalg.inv(A)
array([[ 1., -1., 0.],
[ 0., 1., -1.],
[ 0., 0., 1.]])
```

(b) Déterminant d'une matrice carrée : la fonction `det()`

(c) Résolution d'un système linéaire $Ax = b$: la fonction `solve(A,b)`

Exemple : on veut résoudre le système linéaire :

$$\begin{cases} x - y + z &= 1 \\ -x + y + z &= 1 \\ 2x - y - z &= 0 \end{cases}$$

```
>>> A=np.array([[1,-1,1],[-1,1,1],[2,-1,-1]])
>>> b= np.array([1,1,0])
>>> np.linalg.solve(A,b) # avec solve()
array([ 1., 1., 1.])
>>> np.dot(np.linalg.inv(A),b) # en inversant A
array([ 1., 1., 1.])
```

Représentation graphique avec le module Matplotlib

Représentation discrète d'une fonction

Pour tracer la courbe représentative d'une fonction f avec Matplotlib, il faut tout d'abord la "discrétiser" ; c'est-à-dire définir une liste de points $(x_i, f(x_i))$ qui va permettre d'approcher le graphe de la fonction par une ligne brisée. Bien sûr, plus on augmente le nombre de points, plus la ligne brisée est "proche" du graphe (en un certain sens).

Plus précisément, pour représenter le graphe d'une fonction réelle f définie sur un intervalle $[a, b]$, on commence par construire un vecteur X, discrétisant l'intervalle $[a, b]$, en considérant un ensemble de points équidistants dans $[a, b]$. Pour ce faire, on se donne un naturel N *grand* et on considère `X=numpy.linspace(a,b,N)` (ou, ce qui revient au même, on pose $h = \frac{b-a}{N-1}$ et on considère `X=numpy.arange(a,b+h,h)`).

On construit ensuite le vecteur Y dont les composantes sont les images des X_i par f et on trace la ligne brisée reliant tous les points (X_i, Y_i).

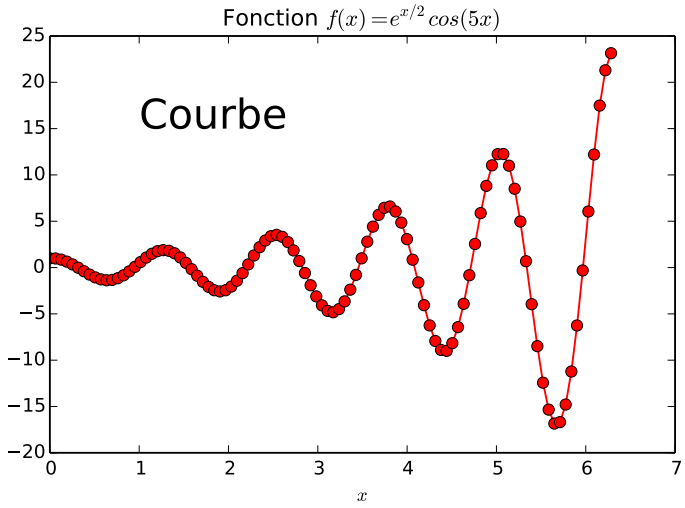
La fonction `plot` du module matplotlib.pyplot

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

x = np.linspace(0., 2*np.pi, 100)
plt.plot(x, np.exp(x/2)*np.cos(5*x), '-ro')

plt.title('Fonction $f(x)=e^{x/2} cos(5x)$')
plt.xlabel('$x$')
plt.text(1, 15, 'Courbe', fontsize=22)
plt.savefig('1D_exemple.pdf')
plt.show()
```

Cette fonction permet de tracer des lignes brisées. Si X et Y sont deux vecteurs-lignes (ou vecteurs-colonnes) de même taille n, alors `plt.plot(X,Y)` permet de tracer la ligne brisée qui relie les points de coordonnées (X(i),Y(i)) pour $i = 1, \ldots, n$.

Pour connaître toutes les options, le mieux est de se référer à la documentation de Matplotlib. Voyons ici quelques unes d'entre elles :

- **bornes** : spécifier un rectangle de représentation, ce qui permet un zoom, d'éviter les grandes valeurs des fonctions par exemple, se fait via la commande `plt.axis([xmin,xmax,ymin,ymax])`
- **couleur du trait** : pour changer la couleur du tracé une lettre g vert (green), r rouge (red), k noir, b bleu, c cyan, m magenta, y jaune (yellow), w blanc (white). `plt.plot(np.sin(x),'r')` tracera notre courbe sinus en rouge. Pour avoir différents niveaux de gris on peut utiliser `color='(un flottant entre 0 et 1)'`. Enfin pour avoir encore plus de couleurs, on peut utiliser le système RGB via `color=( R, G, B)` avec trois paramètres compris entre 0 et 1 (RGBA est possible aussi).



– **symboles** : mettre des symboles aux points tracés se fait via l'option `marker`. Les possibilités sont nombreuses parmi
`'+' | '*' | ',' | '.' | '1' | '2' | '<' | '>' | 'D' | 'H' |`

`'^' | '-' | 'd' | 'h' | 'o' | 'p' | 's' | 'v' | 'x' | '|' |`

`TICKUP | TICKDOWN | TICKLEFT | TICKRIGHT | 'None' | ' ' | " .`

– **style du trait** : pointillés, absences de trait, etc se décident avec `linestyle`. Au choix `'-'` ligne continue, `'--'` tirets, `'-.'` points-tirets, `':'` pointillés, sachant que `'None'`, `''`, `' '` donnent "rien-du-tout". Plutôt que `linestyle`, on peut utiliser la raccourci `ls`.

– **épaisseur du trait** : `linewidth`=flottant (comme `linewidth=2`) donne un trait, pointillé (tout ce qui est défini par style du trait) d'épaiseur "flottant". Il est possible d'utiliser le raccourci `lw`.

– **taille des symboles** : `markersize`=flottant comme pour l'épaisseur du trait. D'autres paramètres sont modifiables `markeredgecolor` la couleur du trait du pourtour, `markerfacecolor` la couleur de l'intérieur, `markeredgsize`=flottant l'épaisseur du trait du pourtour.

– **étiquettes** sur l'axe des abcisses/ordonnées : Matplotlib décide tout seul des graduations sur les axes. Tout ceci se modifie via `plt.xticks`(tf), `plt.yticks`(tl) où tf est un vecteur de flottants ordonnés de façon croissante.

– ajouter un titre : `plt.title`("Mon titre")

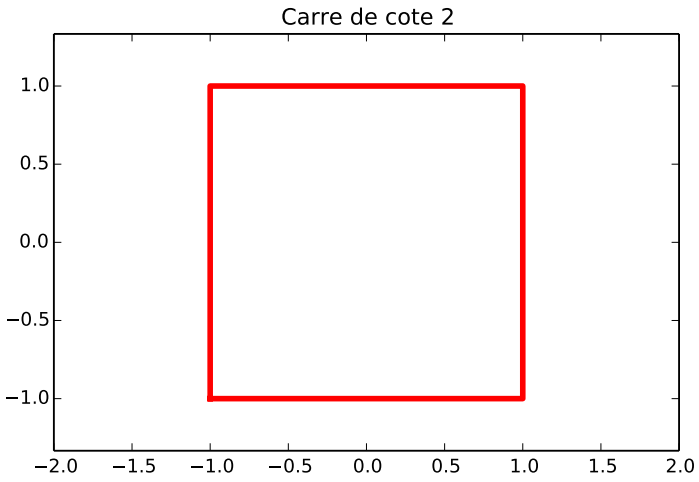
– légendes : `plt.legend`().

Exemple : tracé d'un carré

Coordonnées des sommets : $X = (-1; 1; 1; -1; -1)$ et $Y = (-1; -1; 1; 1; -1)$.

```
X=np.array([-1, 1, 1,-1,-1])
Y=np.array([-1, -1, 1,1,-1])
plt.plot(X,Y,'r',lw=3)
plt.axis('equal')
plt.axis([-2,2,-2,2])
```

```
plt.title('Carre de cote 2')
plt.savefig('carre_exemple.pdf')
plt.show()
```



Tracé de plusieurs lignes brisées

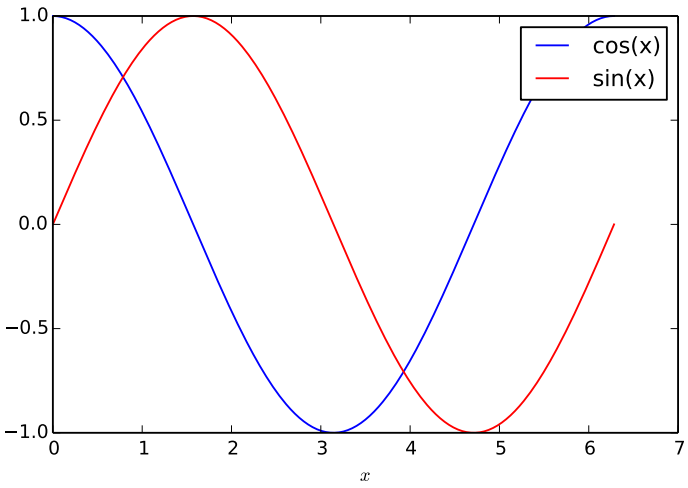
On peut superposer plusieurs courbes dans le même graphique.

```
import matplotlib.pyplot as plt
import numpy as np
X = np.linspace(0, 2*np.pi, 256)
Ycos = np.cos(X)
Ysin = np.sin(X)
plt.plot(X,Ycos,'b')
plt.plot(X,Ysin,'r')
pp.show()
```

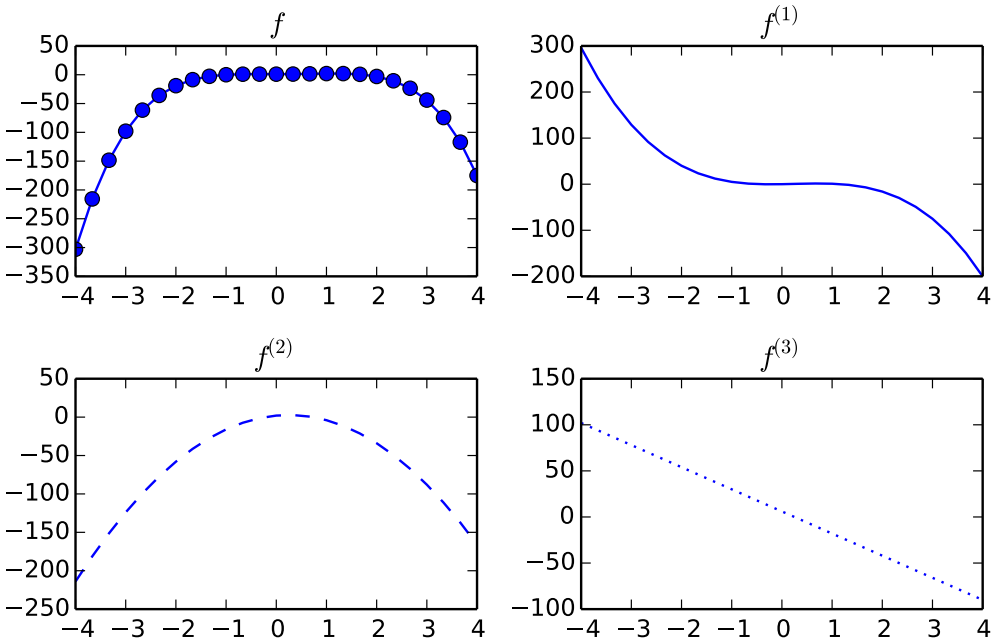
Création de plusieurs graphiques dans une même fenêtre

La commande `subplot` permet de découper la fenêtre graphique en plusieurs sous-fenêtres. Voici un exemple d'utilisation de cette commande.

```
x = np.linspace(-4., 4., 25)
plt.subplot(2, 2, 1)
plt.plot(x, -x**4 + x**3 + x**2 + 1, 'o-')
plt.title("$f$")
plt.subplot(2, 2, 2)
plt.plot(x, -4*x**3 + 3*x**2 + 2*x, '-')
plt.title("$f^{(1)}$")
plt.subplot(2, 2, 3)
plt.plot(x, -12*x**2 + 6*x + 2, '--')
plt.title("$f^{(2)}$")
```




```
plt.subplot(2, 2, 4)
plt.plot(x, -24*x + 6, ':')
plt.title("$f^{(3)}$")
plt.tight_layout()
plt.savefig("1D_subplot2.pdf")
plt.show()
```



Matplotlib dispose d'autres commandes de tracé comme `loglog, polar`...

Université Paris-Sud **Méthodologie**
L1 MPI **S4 - Année 2015-2016**

TP d'introduction à Python

Pour préparer votre séance de TP, taper les commandes suivantes dans un terminal

```
mkdir TP-Python
cd TP-Python
spyder
```

La commande `mkdir` sert à créer un répertoire ; la commande `cd` permet de se placer dans un répertoire.

Voici quelques commandes pour commencer à taper dans l'interpréteur (dans la `console`). Cherchez à comprendre le résultat renvoyé.

a) Affectation d'une variable.

```
>>> x=10
>>> y=5
```

b) Affectation d'un tableau.

```
>>> import numpy as np
>>> a=np.eye(5,5)
>>> print a
>>> x=np.arange(0,10,1)
>>> x
>>> y=np.arange(1,8,2)
>>> print(y)
>>> z=np.array([0, 1, 2, 5])
>>> print(z)
>>> v=np.array([[1,2],[2,3],[3,4]])
>>> print(v)
>>> type(v)
```

c) Opérations sur les vecteurs.

```
>>> import numpy as np
>>> x=np.arange(0,11,1)
>>> y=2*x
>>> np.size(x)
>>> np.shape(x)
>>> x[0]
>>> pow(x,2)
>>> x[1:]
>>> x[2:5]
>>> x[2:2:6]
>>> x=x[2:-1]
>>> y=y[1:-2]
>>> x+y
>>> np.vdot(x,y)
>>> np.dot(x,y)
```

d) Messages d'erreur : tapez l'une après l'autre les commandes suivantes, qui contiennent des erreurs, et lisez les messages d'erreur.

```
>>> toto
>>> import numpy as np
>>> x=np.arange(0,11,1)
>>> x(16)
>>> y=2*x
>>> x+y[2:]
>> size(x)
```

e) Boucle `for` : que font les instructions suivantes :

```
>>> for i in range(1,11):
...     print(i)
...
>>> a=0
>>> for i in range(1,11):
...     a=a+i
...     print(a)
...
>>> a=1
>>> for i in range(1,11,2):
...     a=a*i
...     print(a)
...

>>> for i in range(1,12,3):
...     print i
...
>>> u=[1,4,7,10]
>>> for i in u:
...     print(i)
...
```

f) Fonctions mathématiques : tapez les commandes suivantes et observez les résultats.

```
>>> import numpy as np
>>> import math
>>> pow(math.sin(2.5),2)+pow(math.cos(2.5),2)
>>> pow(np.sin(2.5),2)+pow(np.cos(2.5),2)
>>> math.exp(math.log(3))
>>> math.sqrt(-1)
>>> np.sqrt(-1)
```

A partir de maintenant vous allez écrire vos programmes dans des fichiers exécutables par Python. Pour cela, utilisez l'éditeur de Spyder et créez un fichier `exoNumero.py` pour chaque exercice.

Exercice 1 Que font les commandes (on aura préalablement importé `numpy` sous le nom `np`)

$$\texttt{np.ones}((3,1)), \quad \texttt{np.ones}((1,3)), \quad \texttt{np.ones}(3), \quad \texttt{np.eye}(3), \quad \texttt{np.eye}(3,2)?$$

Exercice 2 Définir de la façon la plus simple possible les vecteurs

$$v1 = (5, 9, 1, 3)$$
$$v2 = (2, 2, 2, \ldots, 2) \text{ de taille } 20$$
$$v3 = (1, 2, 3, 4, \ldots, 15, 16)$$
$$v4 = (28, 26, 24, 22, 20, 18, 16, 14, 12)$$

On utilisera pour cela les fonctions `array`, `ones`, `arange` et `linespace` du module `numpy`.

Exercice 3 Définir une matrice et un vecteur numpy de la manière suivante

```
import numpy as np
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
b = np.array([-3, -2, -1])
```

Question 1.
Quels sont les types des données de A et de b ?
Question 2.
Faire le produit matrice-vecteur et afficher le résultat.

Exercice 4 Définissez les matrices

$$A = \begin{bmatrix} 7 & 0 \\ -1 & 5 \\ -1 & 2 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 4 \\ -4 & 0 \end{bmatrix}, \quad C = \begin{bmatrix} 7 \\ -3 \end{bmatrix}, \quad D = \begin{bmatrix} 8 & 2 \end{bmatrix}.$$

Calculez avec Python les produits AB, AC, CA, DC, DBC, A^TA et AA^T en faisant appel à la fonction `dot` de `numpy`. La transposée d'une matrice se calcule avec la fonction `transpose`. Essayez de comprendre les éventuels messages d'erreur.

Exercice 5 On définit les vecteurs u_1, u_2, u_3 de $\mathbb{R}^5$ par

$$u_1 = \begin{bmatrix} 1 \\ -3 \\ 3 \\ 5 \\ 4 \end{bmatrix}, \quad u_2 = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 4 \\ 3 \end{bmatrix}, \quad u_3 = \begin{bmatrix} 2 \\ -5 \\ -1 \\ -6 \\ 1 \end{bmatrix}.$$

Construire la matrice A dont les colonnes sont formées des vecteurs $u_1, \ldots, u_3$.

Exercice 6 Définir la fonction qui à x fait correspondre $sin(x)$. L'appliquer au calcul du sinus de π puis au sinus du vecteur $(0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}, 2\pi)$.

Exercice 7 Un triangle quelconque peut être caractérisé par ses trois sommets $(x1; y1)$, $(x2; y2)$ et $(x3; y3)$. L'aire de ce triangle peut être calculé à l'aide de la formule suivante

$$aire \ = \ \frac{1}{2} \mid (x1 - x3)(y2 - y1) - (x1 - x2)(y3 - y1) \mid$$

Ecrire une fonction qui renvoie l'aire d'un triangle dont on donne les coordonnées des trois sommets en argument (par exemple `[ [0, 0], [0, 1], [1, 0] ]`).

Exercice 8 [Tracé de fonctions].

Importer le module `matplotlib.pyplot` et utiliser la fonction `plot` de ce module pour tracer les graphes des fonctions suivantes.

On rappelle que pour représenter le graphe d'une fonction f sur un intervalle $[a, b]$, il faut tout d'abord la «discrétiser» ; c'est-à-dire définir une liste de points $(x_i, f(x_i))$ qui va permettre d'approcher le graphe de la fonction par une ligne brisée. Bien sur, plus on augmente le nombre de points, plus la ligne brisée est «proche» du graphe (en un certain sens). Plus concrètement, on commence par construire un vecteur X qui discrétise l'intervalle $[a, b]$, en considérant un ensemble de points équidistants dans $[a, b]$. Pour ce faire on se donne un entier N *grand* et on définit $X = (a, a + h, a + 2h, \ldots, b - h, b)$ où $h = \frac{b-a}{N-1}$.

i) Représenter graphiquement la fonction f définie par : $f(x) = \cos^2(x),\ x \in [0, 5]$.
Pour le tracé, utiliser 6 points, puis 26 et enfin 301 points équirépartis dans le segment $[0, 5]$. Préciser dans chaque cas le pas de la subdivision utilisée.

ii) Représenter sur la même figure les fonctions :
$f(x) = \cos^2(x),\ g(x) = \cos(2x),\ h(x) = \cos(x^2),\quad x \in [0, 5]$.
Penser à utiliser des symboles différents pour les différentes représentations graphiques et à mettre une légende.

Exercice 9 .
Soient $f_1,\ f_2,\ f_3$ et f_4 les fonctions définies par $f_1(x) = \sin(x), f_2(x) = \sin(2x), f_3(x) = \sin(3x),\ f_4(x) = \sin(4x),\quad x \in \mathbb{R}$.

i) Représenter sur la même figure les fonctions f_1, f_2, f_3 et f_4 sur l'intervalle $[0, 4\pi]$.

ii) On pose $g(x) = \sum_{k=1}^{4} \sin(kx),\quad x \in \mathbb{R}$. Dans la figure utilisée à la question 1, tracer la courbe représentative de g pour $x \in [0, 4\pi]$.

TP Numérique

UNIVERSITÉ PARIS SUD

Travaux Pratiques de Python

Cours de Méthodologie

—Licence MPI L1 S2 - Info 121—

Traitement de donnée

Dans cette séance de TP, nous allons appliquer les méthodes de traitement de donnée à l'étude du temps de calcul de trois algorithmes d'arithmétique sur les grands nombres.

1 Arithmétique de la machine et des grands entiers

Dans beaucoup de langage de programmation, le type des entiers (par exemple `int` en `Java`, `C` ou `C++`) ne permet de représenter qu'un intervalle fini des entiers relatifs (typiquement $[2^{63}, 2^{63} - 1]$, sur beaucoup de machines modernes). Ceci est dû au fait que le processeur de la machine code les entiers en binaire avec un nombre fixe de chiffres (typiquement 64). On appellera les entiers de cet intervalle **petits entiers**. Chaque opération sur les petits entiers est programmée par une unique instruction du langage de la machine et est directement effectuée par un circuit électronique dont le temps de calcul ne dépend pas de la taille de l'entier. Par exemple, l'addition de deux nombres se fait, sur les machines de la famille `i86`, par l'instruction `ADD` dont le temps de calcul est (pour faire simple) un cycle d'horloge, c'est-à-dire 0.5 ns sur une machine à 2 Ghz. Pour cette raison, on appelle aussi les petits entiers : «**entiers machines**».

En Python, quand on sort de l'intervalle des petits entiers, on bascule automatiquement sur une bibliothèque de **grands entiers** ou **entiers multiprécision** nommée GMP. Pour noter le passage aux entiers GMP, Python ajoute le suffixe `L` à la fin de l'entier. Voici, par exemple, quelques puissances de 2 :

```
>>> 2**3
8
>>> 2**10
1024
>>> 2**62
4611686018427387904
>>> 2**63
9223372036854775808L
>>> 2**64
18446744073709551616L
>>> 2**100
1267650600228229401496703205376L
```

Quand on passe aux grands entiers, les opérations usuelles (addition, multiplication...) sont effectuées par un sous-programme dont le temps de calcul augmente avec la taille de l'entier. Dans ce travail, on veut mesurer et étudier **la variation de ce temps de calcul en fonction de la taille de l'entier**.

Après cette première partie introductive, la deuxième partie explicite la manipulation des grands entiers en Python à travers des fonctions (i) de représentation des grands entiers, (ii) de somme de grands entiers, (iii) de produits de grands entiers selon deux algorithmes différents, la multiplication

dite «naive» et la multiplication rapide de Karatsuba. Le troisième partie propose trois modèles pour les temps de calcul des trois opérations citées, la quatrième partie indique comment mesurer un temps de calcul en Python. Et pour finir la cinquième partie énumère les différentes étapes du travail à effectuer en séance de TP.

Note 1 *Pour la séance de TP, vous n'avez pas besoin de comprendre le fonctionnement interne des fonctions présentées dans la partie 2. Il vous suffit ce comprendre ce qu'elles calculent et comment les utiliser. Pour ceci, vous pouvez vous inspirer des exemples fournis dans les fonctions.*

2 Manipulation des grands entiers en Python

La bibliothèque GMP utilise des codes extrêmement optimisés qui sont difficiles à lire (c'est un mélange de `C` et de langage machine) et dont les temps de calcul sont très courts donc difficiles à mesurer. De plus, depuis Python, on n'a pas le choix de l'algorithme utilisé pour effectuer les opérations, il est choisi automatiquement. Pour les besoin de ce TP, on va utiliser un code écrit en Python qui a pour objectif d'être compréhensible et ne vise absolument pas à avoir de bonnes performances.

2.1 Représentation des entiers

Note 2 *On ne représentera que des entiers* positifs ou nuls. *Un entier n sera représenté par la liste de ses chiffres en base* 10 dans l'ordre du poids faible au poids fort*, c'est-à-dire l'ordre inverse.*

L'avantage est que le chiffre correspondant à la puissance i de 10 est stocké dans la case i de la liste. Ainsi, le nombre qui s'écrit 42348 en base 10, se décompose comme ceci

$$42348 = 8 * 10^0 + 4 * 10^1 + 3 * 10^2 + 2 * 10^3 + 4 * 10^4$$

et sera donc représenté par la liste `[8, 4, 3, 2, 4]`

Note 3 *Le nombre* 0 *est représenté par la liste vide `[]`.*

Voici les fonctions de conversion entier `int` vers liste qui sont appelées `nb` dans le code. Les parties de code entre `"""` sont des commentaires qui contiennent des exemples.

```
Base = 10

def intnb(nbint):
    """
    Convertis un nombre en liste

    >>> intnb(0)
    []
    >>> intnb(1)
    [1]
    >>> intnb(3245)
    [5, 4, 2, 3]
    """
    res = []
    while nbint:
```

```
        res.append(nbint % Base)
        nbint = nbint // Base
    return res

def nbint(nb):
    """
    Convertis une liste en nombre

    >>> nbint([])
    0
    >>> nbint([1])
    1
    >>> nbint([5, 4, 2, 3])
    3245

    On verifie que les deux conversions sont bien inverse:

    >>> for i in range(200):
    ....:     assert(nbint(intnb(i)) == i)
    """
    res = 0
    for i in nb[::-1]:
        res = res*10 + i
    return res
```

2.2 Somme

On n'a pas écrit directement la fonction qui ajoute deux entiers. On a écrit à la place la fonction plus générale `ajoute_decal` qui ajoute à la liste `n` le nombre `m` décalé de `d` (on en aura besoin pour la multiplication). Voici, une illustration de `ajoute_decal(123,13, 2)` :

```
  123
+13..
-----
 1423
```

```
def ajoute_decal(n, m, d):
    """
    Ajoute a n le nombre m decale de d
    On modifie n

    >>> n = [3,2,1]; ajoute_decal(n, [3,1], 2); n
    [3, 2, 4, 1]
    >>> n = []; ajoute_decal(n, [1], 1); n
    [0, 1]
    >>> n = [9, 9]; ajoute_decal(n, [1], 1); n
    [9, 0, 1]
    >>> n = [9, 9]; ajoute_decal(n, [1, 1], 1); n
    [9, 0, 2]
    """
```

```
    if m == []: return
    # On complete n avec des zero si besoin
    n.extend([0]*(len(m) + d - len(n)))
    retenue = 0
    for i in range(len(m)):
        n[d] += m[i]+retenue
        if n[d] >= Base:
            n[d] -= Base
            retenue = 1
        else:
            retenue = 0
        d += 1
    while d < len(n) and retenue:
        n[d] += retenue
        if n[d] >= Base:
            n[d] -= Base
            retenue = 1
        else:
            retenue = 0
        d += 1
    if retenue:
        n.append(retenue)
```

Maintenant pour faire la somme de `n` et `m` , il suffit de copier `n` et de lui ajouter `m` sans décaler :

```
def somme(n, m):
    """
    Retourne la somme des nombres n et m

    >>> somme([], [])
    []
    >>> somme([1], [])
    [1]
    >>> somme([], [1])
    [1]
    >>> somme([4,4,5,2],[2,8])
    [6, 2, 6, 2]

    >>> for i in range(1000):
    ....:     for j in range(1000):
    ....:         assert(nbint(somme(intnb(i), intnb(j))) == i + j)
    """
    res = n[:]
    ajoute_decal(res, m, 0)
    return res
```

On écrit de manière similaire la différence (voir le code fourni par l'enseignant, fichier mult.py).

2.3 Multiplication naive

Pour faire la multiplication de deux entiers, on procède comme à la petite école, en multipliant le premier nombre par les chiffres du deuxième nombre et en décalant, puis on ajoute le tout :

```
          36
         x 25
         ----
36*5 :    180
36*2 :   +72
         ====
          900
```

On commence par écrire une fonction qui multiplie un entier par un chiffre :

```
def produit_chiffre(n, c):
    """
    Multiplie le nombre n par un chiffre

    >>> produit_chiffre([4,4,5,2],9)
    [6, 9, 8, 2, 2]
    """
    if c == 0: return []
    if c == 1: return n
    retenue = 0
    res = [0]*len(n)
    for i in range(len(n)):
        resc = n[i]*c + retenue
        res[i] = resc % Base
        retenue = resc // Base
    if retenue:
        res.append(retenue)
    return res
```

Ensuite, il suffit d'ajouter en décalant :

```
def produit_naif(n, m):
    """
    >>> produit_naif([9,9],[9,9])
    [1, 0, 8, 9]
    >>> for i in range(1000):
    ....:     for j in range(1000):
    ....:         assert(nbint(produit_naif(intnb(i), intnb(j))) == i * j)
    >>> a = 129806570567313324234234
    >>> nbint(produit_naif(intnb(a), intnb(a)))
    16849745762446893790131992088457114322497566756
    """
    res = []
    for i in range(len(n)):
        ajoute_decal(res, produit_chiffre(m, n[i]), i)
    return res
```

Note 4 *Ainsi, en utilisant la méthode naive, pour faire le produit d'un nombre à n chiffres par un nombre à m chiffres, on a besoin de faire $n * m$ produit de chiffres.*

2.4 Multiplication rapide de Karatsuba

L'algorithme inventé en 1960 par le russe Karatsuba (Анатолий Алексеевич Карацуба) permet pour les grands nombres d'aller plus vite (voir `https://fr.wikipedia.org/wiki/Algorithme_de_Karatsuba`). L'idée est la suivante : le calcul de

$$(a \times 10^k + b)(c \times 10^k + d) \tag{1}$$

qui, sous forme développée s'écrit

$$ac \times 10^{2k} + (ad + bc) \times 10^k + bd \tag{2}$$

semble nécessiter le calcul des quatre produits ac, ad, bc et bd. Pour de grands nombres, en regroupant les calculs sous la forme suivante :

$$(a \times 10^k + b)(c \times 10^k + d) = ac \times 10^{2k} + (ac + bd - (a-b)(c-d)) \times 10^k + bd \tag{3}$$

la méthode peut être appliquée de manière récursive pour les calculs de ac, bd et $(a-b)(c-d)$ en scindant à nouveau a, b, c et d en deux et ainsi de suite. C'est un algorithme de type diviser pour régner.

Exécutons l'algorithme pour calculer le produit 1237×2587.

- Pour calculer, 1237×2587, on écrit $1237 \times 2587 = a_0 10^4 + (a_0 + a_2 - a_1)10^2 + a_2$ où $a_0 = 12 \times 25$, $a_1 = (12-37) \times (25-87) = 25 \times 62$ et $a_2 = 37 \times 87$.
  - Pour calculer 12×25, on écrit $12 \times 25 = a'_0 10^2 + (a'_0 + a'_2 - a'_1)10 + a'_2$ où $a_0 = 1 \times 2$, $a_1 = (1-2) \times (2-5) = -1 \times -3$ et $a_2 = 2 \times 5$.
    - Les calculs $1 \times 2 = 2$, $2 \times 5 = 10$ et $-1 \times -3 = 3$ sont tous des multiplications de chiffres (on ne les redécompose pas). Ils sont faits par les opérations sur les petits entiers.
    - On obtient $12 \times 25 = 2 \times 100 + (2 + 10 - 3) \times 10 + 10 = 300$
- De la même façon, on obtient $a_1 = 25 \times 62 = 1550$.
- De la même façon, on obtient $a_2 = 37 \times 87 = 3219$.
- d'où

$$1237 \times 2587 = 300 \times 100^2 + (300 + 3219 - 1550) \times 100 + 3219 = 3000000 + 196900 + 3219 = 3200119\,.$$

Note 5 *Le calcul complet ne demande que* 9 *produits de deux chiffres au lieu de* $4 \times 4 = 16$ *par la méthode usuelle.*

Bien entendu, cette méthode, fastidieuse à la main, révèle toute sa puissance pour une machine devant effectuer le produit de grands nombres.

En Python, on peut écrire le code :

```
def Karatsuba(n, m, seuil = 10):
    """
    >>> Karatsuba([9,9],[9,9])
    [1, 0, 8, 9]
    >>> for i in range(100):
    ....:     for j in range(100):
    ....:         assert(nbint(Karatsuba(intnb(i), intnb(j))) == i * j)
    >>> a = 129806570567313324234234
    >>> nbint(Karatsuba(intnb(a), intnb(a)))
    16849745762446893790131992088457114322497566756
    """
```

```
# On suppose que n est le plus long.
if len(n) < len(m):
    n, m = m, n
if m == []: return []
if len(m) == 1: return produit_chiffre(n, m[0])
if len(m) <= seuil: return produit_naif(n, m)
cut = len(n) // 2
n1 = n[:cut]; n2 = n[cut:]
m1 = m[:cut]; m2 = m[cut:]
n1m1 = Karatsuba(n1, m1)
n2m2 = Karatsuba(n2, m2)
mid = Karatsuba(somme(n1, n2), somme(m1, m2))
retranche(mid, n1m1)
retranche(mid, n2m2)
res = n1m1
ajoute_decal(res, mid, cut)
ajoute_decal(res, n2m2, 2*cut)
return res
```

3 Notion de complexité

On s'intéresse au temps de calcul des trois fonctions `somme`, `produit_naif` et `Karatsuba`. L'objectif est d'estimer le temps de calcul nécessaire pour faire des opérations sur des nombres à un million de chiffres !

Pour faire la somme de deux nombres à n chiffres, on a n additions de chiffres à faire, il est donc raisonnable de penser que le temps de calcul de la somme va être de l'ordre de grandeur de n fois le temps de calcul de la somme de deux chiffres et va donc suivre une loi de la forme :

$$\text{temps}_{\texttt{somme}}(n) \approx C_{\texttt{somme}} \times n + T_0 \tag{4}$$

où C est une constante (qui dépend de la vitesse de la machine) à trouver. Le temps T_0 correspond au temps des initialisations et est tout petit. On va le négliger et donc ne garder que :

$$\text{temps}_{\texttt{somme}}(n) \approx C_{\texttt{somme}} \times n \tag{5}$$

Pour ce qui est de la multiplication naive, on a vu que pour multiplier deux nombres à n chiffres, il faut faire n^2 multiplications de chiffres. On va chercher une loi de la forme :

$$\text{temps}_{\texttt{naif}}(n) \approx C_{\texttt{naif}} \times n^2 \tag{6}$$

Le cas de la multiplication de Karatsuba est plus compliqué. On va chercher une loi de la forme

$$\text{temps}_{\texttt{Kara}}(n) \approx C_{\texttt{Kara}} \times n^{\alpha} \tag{7}$$

Le but de ce TP est de
- vérifier que les lois théoriques ci-dessus correspondent bien à ce que l'on observe sur une machine.
- de déterminer les valeurs de $C_{\texttt{somme}}$, $C_{\texttt{naif}}$, $C_{\texttt{Kara}}$ et α.
- d'extrapoler pour estimer le temps pris par l'addition et la multiplication de deux nombres à un million de chiffres.

4 Mesure d'un temps de calcul en Python

Pour mesurer le temps d'un calcul en Python, on va utiliser la fonction `time.time()`. Pour ceci, il faut charger le module `time` :

```
>>> import time
```

Ensuite, un appel à `time.time()` donne le nombre de secondes écoulées depuis la date référence `epoch` c'est à dire le 1er janvier 1970 :

```
>>> time.time()
1486144342.05344
```

Pour trouver le temps de calcul pris par la multiplication naive du nombre formé de 100 chiffre 9 par lui même il, il faut donc faire :

```
>>> nb = [9]*100
>>> avant=time.time(); res = produit_naif(nb, nb); apres=time.time()
>>> apres-avant
0.040435075759887695
```

Ici le calcul a pris environ 0.04 s

5 Travail demandé

1. Écrire une fonction `mesure(fun, nb_chiffres)` qui étant donnée une opération `fun` sur les grands nombres retourne le temps de calcul de `fun` sur deux entiers ayant `nb_chiffres` chiffres.
2. Créer une liste `tailles` contenant les nombres de chiffre des nombres que l'on va multiplier. On part de nombres à 10 chiffres jusqu'à des nombres à 2560 chiffres et en multipliant par deux à chaque fois.
3. Mesurer les temps de calcul des trois fonctions `somme`, `produit_naif` et `Karatsuba` pour les nombres de tailles précédentes.
4. Tracer les courbes correspondantes.
5. La seule courbe qu'un être humain est capable de reconnaître à l'œil est une droite. Après avoir commenté les trois tracées, on restreindra à l'étude des deux opérations de multiplication (naive et Karatsuba). Pour chacun des deux modèles correspondants, déterminer une fonction $y(x)$ qui sera une droite si le modèle est vérifié, i.e. dire ce qu'il faut prendre pour y et pour x pour que les données s'alignent sur une droite si le modèle est valide.
6. Pour chacune des deux fonctions, créer le vecteur y et le vecteur x correspondant à partir des données expérimentales.
7. Faire les ajustements de droite (utiliser le module `scipy.optimize` de Python) sur les mesures de temps de calcul obtenues précédemment et en déduire les valeurs de $C_{\texttt{naif}}$, $C_{\texttt{Kara}}$ et α.
8. Faire les tracés de comparaison modèle/données, commenter et extrapoler pour estimer le temps de calcul pour des nombres à 1 million de chiffres.

Rédaction d'un Compte-rendu de TP

Mettez votre nom sur le Compte-Rendu de TP.

Votre compte-rendu doit comporter les 6 points suivants :

1. Introduction/Objectif du TP:

Présentez le contexte général (En 3-4 lignes) et expliquez en quelques mots (une ou deux phrases) quel est l'objectif de ce TP.

2. Présentation de l'étude :

Expliquez en quelques lignes les opérations pour lesquelles vous souhaitez mesurer le temps de calcul et les fonctions python qui les réalisent et que vous souhaitez comparer (ne pas réécrire le texte de la feuille du TP !). Expliquez comment vous allez mesurer et caractériser ces temps de calcul.

3. Mesures ou observations :

Donnez les résultats bruts sous forme des courbes dans un ou plusieurs graphiques. Décrivez qualitativement les courbes obtenues.

4. Exploitation des mesures:

Exploitez vos résultats. Déduisez les lois que suivent les temps de calcul (en fonction du nombre de chiffres des entiers manipulés). Extrapolez pour estimer le temps pris par les différentes opérations pour deux nombres à un million de chiffres.

5. Conclusion :

Résumez vos résultats et dites quelles conclusions vous pouvez en tirer.

Attention :

- **Les figures** doivent être numérotées, légendées et appelées dans le texte.
- **Les graphiques** doivent être clairs et comporter une légende s'il y a plusieurs courbes. Il est nécessaire d'écrire ce qui est porté en abscisse et en ordonnée avec les unités.

En résumé, quelques conseils :

- **Rendez un compte-rendu court et précis** : faites des phrases, mais elles ne doivent pas être trop longues.
- **Hiérarchisez votre compte-rendu** : toutes les informations ne doivent pas être mises sur le même plan. Dégager les parties importantes.
- **Ne recopiez pas ce qui est déjà écrit quelque part.**

Barème Code

1. Fonction mesure : 3 points

2. Création du vecteur taille : 2points

3. Application de la fonction mesure aux fonctions somme, produit_naif et Karatsuba : 2 points

4. Tracé des courbes correspondantes : 3 points

5. Tracé des droites : 2 points

6. Régression linéaire : 2 points

7. Estimation des valeurs Cnaif, CKara et alpha : 2 points

8. Tracer les comparaisons modèles/données : 2 points

9. Extrapolation : 2 points

Barème Compte Rendu de TP

Présentation générale (4 pts)

- Mettre son nom et un titre, clarté des explications, plan (1 pts).
- Orthographe (1 pt).
- Figures numérotées et appelées dans le texte (1 pt).
- Graphiques comportant des axes et une légende (1 pt).

Introduction (2 pts)

Problématique (que va-t-on mesurer et pourquoi ?).

Présentation de l'étude (4 pts)

- Description (sans recopier le code) des trois fonctions testées pour multiplier deux grands nombres. (2 pts)
- Description de la méthode utilisée pour comparer leur temps de calcul (2 pts).

Résultats bruts (2 pts)

- Insertion des résultats bruts dans un graphique (1 pts).
- Description qualitative de la courbe (1 pt).

Traitement des résultats (6 pts)

- Description du modèle utilisé/de la fonction avec laquelle on va comparer les données (2 pt).
- Comparaison qualitative aux données (2 pt).
- Comparaison quantitative aux données, mesure de C_{naif}, C_{kara} et $\langle$ (1.5 pts).
- Commentaire (0.5 pt).

Conclusion (2 pts)

- Résumé (1 pt).
- Commentaire (1 pt).

Estimations

Ordres de grandeur

Questions de Fermi

Le but de ce chapitre est de vous présenter des façons de faire de la physique utilisant le raisonnement, les dimensions, des estimations et finalement très peu d'équations. Ces méthodes peuvent donner l'impression d'être floues, non rigoureuses et imprécises, mais ce n'est pas le cas. Simplement leur objectif n'est pas de donner un résultat chiffré précis, mais de donner un ordre de grandeur du résultat, ce qui est parfois suffisant. Ces façons de faire sont souvent utilisées, soit quand on aborde un problème pour la première fois et qu'on veut avoir une idée rapide de sa réponse, soit après avoir résolu un problème en ayant obtenu un résultat après de longs calculs, afin d'en vérifier rapidement la validité. Attention, contrairement à ce que l'on pourrait croire, cela ne veut pas dire que c'est facile !

Cette séance devrait donc vous aider à acquérir des outils qui sont utiles dans tous les domaines scientifiques afin :

- D'aborder un problème à partir de rien, y compris quand le raisonnement n'est pas dirigé par des questions très détaillées ;
- De réfléchir sur les ordres de grandeurs ce qui permet de ne pas faire des erreurs de plusieurs facteurs 10 sur un résultat, par exemple.

Méthodes

Dans chaque exercice nous vous demandons avant toutes choses (et en particulier avant de répondre à vos questions) :

- De faire un **schéma**
- De lister les **grandeurs pertinentes** et de leur **donner un nom**
- De donner **leur dimension** et de **les estimer** (il ne s'agit pas de savoir exactement combien elles valent mais d'avoir une idée de leur ordre de grandeur).

Estimations

1- Combien y a-t-il de grains de riz dans un paquet d'un kilogramme ?
2- Combien y a-t-il de litres d'eau dans la mer méditerranée ?
3- Si on dénoyaute des mirabelles pendant les matches d'une coupe du monde de foot, combien cela fait-il confiture ?
4- Quel est le nombre de conducteurs de TGV en France ?

Rédaction de votre réponse

- **Introduisez succinctement le problème**

Quelle est la question que l'on vous pose ?

- **Faire un schéma**

Le schéma doit être suffisamment grand et clair. Il doit modéliser le problème. Il ne s'agit pas d'une œuvre d'art mais d'un dessin qui permet de représenter votre modélisation. Il doit donc comporter les grandeurs pertinentes (avec leur nom et sans leur valeur).

- **Lister les grandeurs pertinentes, les nommer, les estimer**

Il s'agit ici de faire une liste des grandeurs pertinentes contenant leur nom (c'est-à-dire une lettre), leur signification, leur estimation quand c'est nécessaire et leur unité.

- **Présenter votre démarche**

Il s’agit ici d’expliquer succinctement comment vous allez résoudre le problème. Il s’agit de donner les grandes lignes de la résolution. Des équations peuvent illustrer cette partie.

- **Obtenir une expression littérale**

- **Effectuer l’application numérique**

- **Présenter le calcul de propagation des incertitudes**

- **Conclure sur votre résultat.**

Il faut tout d’abord formuler la réponse à la question puis commenter le résultat, éventuellement en le comparant à une deuxième source (internet…).

Annexe 2 : barème

Présentation

Nom des étudiants, titre, présentation générale (1 pt)

Appropriation du problème

Schéma (3 pt)

1 pt pour la propreté, taille, clarté

1 pt pour la modélisation

1 pt pour la présence des paramètres pertinents sur le schéma

Liste des paramètres (1 pt)

Nommer les paramètres (1 pt)

Estimer les paramètres (1 pt)

Résolution du problème

Formule littérale (3 pt)

Application numérique (1 pt) avec les unités (1 pt)

Calcul de la propagation des incertitudes (2 pt)

Rédaction de la solution

Introduire le problème (2 pt)

Expliquer la démarche (2 pt)

Commenter la réponse et conclure (2 pt)

Annexe 2 : Méthodes de calcul pour une grandeur fonction de plusieurs variables

(tirées du poly de Phys102)

a) Cas général

De façon générale, connaissant les mesures x_{exp} et y_{exp} des grandeurs x et y, et connaissant les incertitudes δx et δy de ces mesures, on cherche à connaître l'incertitude sur la grandeur dérivée $G(x,y)$. La méthode par encadrement est toujours applicable, mais fastidieuse.

Soit G une fonction de 2 grandeurs x et y. L'incertitude absolue δG se calcule à partir des incertitudes absolues de mesure δx et δy, et des dérivées partielles de la fonction G aux points de mesure x_{exp} et y_{exp} :

$$\delta G = \left|\frac{\partial G}{\partial x}(x_{exp}, y_{exp})\right| \delta x + \left|\frac{\partial G}{\partial y}(x_{exp}, y_{exp})\right| \delta y$$

La dérivée partielle de G par rapport à x s'obtient en considérant que y, ou plus généralement, toutes les variables autres que x, sont constantes.

b) Cas d'une loi de puissance : dérivée logarithmique

Si la grandeur G s'exprime sous forme d'un produit de puissances de x et y, $G = k\, x^m\, y^n$, il est plus facile de calculer l'incertitude relative $\delta G/|G|$ que l'incertitude absolue δG.

L'incertitude relative $\delta G/|G|$ s'écrit simplement comme la somme des incertitudes relatives des grandeurs mesurées, pondérée par les puissances :

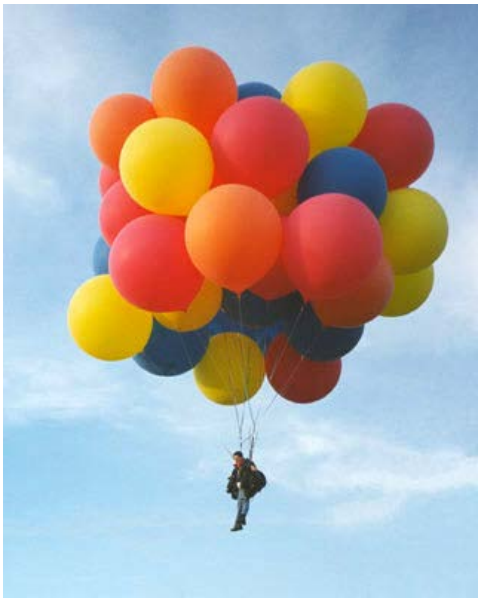
$$\frac{\delta G}{|G|} = m\frac{\delta x}{|x|} + n\frac{\delta y}{|y|}$$

Elle est indépendante de la constante k.

Résolution de problèmes

Problème : et si on volait ?

Rappelons la loi d'Archimède : "Tout corps plongé dans un fluide reçoit de celui-ci une force dirigée vers le haut et égale au poids de fluide déplacé"



1 : Un pratiquant du "balloon clustering"

2 : Une image du film d'animation "UP"

Les pratiquants du "balloon clustering" volent grâce à une grappe de ballons gonflés à l'Hélium. L'objectif de l'exercice est de répondre aux deux questions suivantes :

1. Combien de ballons faut-il pour soulever un être humain ?
2. Soulever une maison à l'aide de ballons de baudruche comme dans le film "Là-haut" est-il inconcevable, réaliste ou entre les deux ?

Comme pour toutes les « résolutions de problèmes » nous vous invitons vivement à commencer par :

- Représenter la situation physique par un schéma qui modélise la situation physique et fait apparaître les grandeurs pertinentes.
- Identifier les grandeurs physiques qui vous paraissent importantes pour répondre à la question.
- Donner leurs dimensions, leur donner un nom et estimer leurs valeurs.
- Enoncer les lois physiques qui vous paraissent pertinentes (nom, énoncé, équation).

Barème : Résolution de problèmes

Présentation

Nom des étudiants, titre, présentation générale (1 pt)

Appropriation du problème

Schéma (2.5 pt)

- 0.5 pt pour la propreté, taille, clarté
- 1 pt pour la modélisation
- 1 pt pour la présence des paramètres pertinents sur le schéma

Liste des paramètres (0.5 pt)

Nommer les paramètres (0.5 pt)

Estimer les paramètres (0.5 pt)

Résolution du problème

Formuler les lois de la physique qui seront nécessaires à votre calcul. (1 pt)

Formule littérale (2 pt)

Application numérique (1 pt) avec les unités (0.5 pt)

Calcul de la propagation d'erreur (1.5 pts)

Rédaction de la solution

Introduire le problème (2 pt)

Expliquer la démarche (2 pt)

Présenter la propagation d'incertitudes (1 pt)

Commenter la réponse et conclure (2 pt)

Structure et clarté des explications (2 pt)