Épreuve orale de « *Mathématiques et algorithmique* » de la Banque PT – Rapport 2018

Les futurs candidats trouveront dans ce rapport des remarques et des conseils qui pourraient leur être utiles pour leur futur passage. Ce rapport n'est pas exhaustif et ne met l'accent que sur quelques points jugés importants par l'équipe d'interrogateurs de cet oral. Nous suggérons aux futurs candidats de consulter le **site de la Banque PT**, où ils pourront trouver le mémento *Python* fourni lors de l'oral, les exercices types d'informatique, ainsi que les rapports des années antérieures comportant des informations complémentaires en regard du présent rapport.

Nous attirons l'attention des futurs candidats ainsi que des enseignants de classes préparatoires sur le fait que **le mémento fourni à l'oral change à la rentrée 2018**. Il est joint en annexe du présent document.

Est fournie également en annexe une petite partie des exercices qui ont été posés lors des dernières sessions, à titre d'exemples.

1 – Objectifs

Le but d'une telle épreuve est d'abord de contrôler l'assimilation des connaissances des programmes de mathématiques et d'informatique (items 2, 3 et 5) de toute la filière (première et deuxième années), sans oublier celle des connaissances de base du programme des classes du lycée (seconde, première, terminale).

Cette épreuve permet aussi d'examiner :

- l'aptitude du candidat à lire attentivement un sujet et à répondre précisément à la question posée ;
- son aisance à exposer clairement ses idées avec un vocabulaire précis ;
- sa capacité d'initiative et son autonomie et, en même temps, son aptitude à écouter l'interrogateur, à prendre en compte ses indications, à lui demander des précisions si besoin ;
- son aptitude à mettre en œuvre ses connaissances et son savoir-faire pour résoudre un problème (par la réflexion et non par la mémorisation de solutions toutes faites) ;
- sa maîtrise des algorithmes et manipulations de base, des calculs sur des nombres entiers, décimaux ou complexes, et du langage de programmation pour mettre en œuvre une solution informatique ;
- sa faculté à critiquer, éventuellement, les résultats obtenus et à changer de méthode en cas de besoin.

2 – Modalités de cette épreuve

La durée de cet oral de « *Mathématiques et algorithmique* » est de 1 heure, préparation incluse.

Il comporte deux exercices de durées comparables :

- l'un porte sur le programme de mathématiques des deux années de la filière PTSI/PT (algèbre, analyse, géométrie et probabilités) et se déroule au tableau ;
- l'autre exercice porte sur les items 2, 3 et 5 du programme d'informatique et se déroule sur ordinateur. Pour ce deuxième exercice, les candidats disposent d'un ordinateur (Windows 7, clavier français Azerty) dans lequel sont installés *Python* 3.5 et ses principales bibliothèques (dont `numpy`, `scipy`, `matplotlib`, `random`, aides incluses) [1], un mémento plastifié en couleurs au format A3, joint au présent rapport, et de feuilles de brouillon, qu'il ne faut pas hésiter à utiliser. L'environnement de développement est *IDLE*, comme annoncé depuis 2014, muni de l'extension *IDLEX* qui permet

1. *Scilab* 5.5 est également installé mais, depuis 2015, aucun candidat n'a demandé à programmer en *Scilab*.

notamment d'afficher plus clairement les numéros de ligne, de faire exécuter une partie d'un programme seulement (F9 au lieu de F5), ou de rappeler dans la console une commande déjà saisie (flèches montante et descendante). Quelques candidats ont avoué avoir préparé l'oral avec *Spyder* ou *Pyzo*, ce qui est un peu surprenant. Nous ne pouvons que conseiller de se placer dans les conditions de passage de l'oral tout au long des deux années de préparation.

3 – Organisation

Cette dernière session s'est déroulée dans des conditions identiques aux sessions précédentes. Comme les autres années, elle a eu lieu dans les locaux de « *Arts et Métiers ParisTech* », 155 boulevard de l'Hôpital à Paris (13^{e}). En raison du plan Vigipirate, il n'a pas été possible cette année d'accueillir de futurs candidats ; deux enseignants de classes préparatoires ont pu assister à un oral, après en avoir fait la demande au Service Concours.

4 – Conseils généraux

Lors d'une épreuve orale, le candidat doit être extrêmement vigilant :

- lire attentivement le sujet et bien écouter une question permet de répondre à la question effectivement posée ;
- écouter les consignes de l'interrogateur est en général utile ;
- lorsqu'une indication est donnée pour aider le candidat, il faut savoir l'écouter et réagir à celle-ci.

Ces capacités d'attention, d'écoute et de réaction sont des éléments d'évaluation. De manière générale, la passivité, l'attentisme ou l'obstination dans une voie infructueuse sont déconseillés lors de l'oral.

Les exercices peuvent être de longueurs variables. L'objectif poursuivi est l'évaluation par l'interrogateur des capacités de chaque candidat grâce à l'exercice proposé, et non pas que le candidat termine nécessairement l'exercice.

L'oral, contrairement à une « *colle* », ne sert qu'à évaluer les capacités du candidat et non plus à participer à sa formation ; des indications seront en général données par l'interrogateur si le candidat reste bloqué trop longtemps, ou si celui-ci demande de l'aide par des questions dont il reconnaît implicitement ignorer la réponse (exemples : « *Est-ce que je peux utiliser tel théorème ?* », ou « *Pourquoi la figure ne s'affiche-t-elle pas ?* »).

Quelques détails utiles en mathématiques comme en informatique :

- la correspondance entre un point du plan et son affixe, ainsi que les interprétations géométriques du module, de l'argument, des parties réelles et imaginaires, du conjugué d'un nombre complexe sont supposées maîtrisées ;
- de même pour des manipulations géométriques de base comme le calcul des coordonnées du milieu d'un segment, de la longueur de ce segment (en repère orthonormé), des coordonnées des sommets d'un polygone usuel – en vue par exemple de faire tracer les côtés de ce polygone à l'écran –, de l'aire de polygones usuels (triangle, trapèze, carré, rectangle), ou de la hauteur d'un triangle équilatéral en fonction de la longueur de son côté ;
- il peut également être bon de connaître l'expression du coefficient binomial $\binom{n}{k}$ sous la forme $\dfrac{n \times (n-1) \times \cdots \times (n-k+1)}{k \times (k-1) \times \cdots \times 1}$, et pas seulement sa définition avec des factorielles.

5 – Conseils pour l'exercice de mathématiques

5.1 – Généralités

- L'oral n'est pas un écrit sur tableau ; les justifications et commentaires doivent être donnés au moment où l'on est interrogé ; le temps étant limité, il est inutile d'écrire de longues phrases, notamment pour justifier une linéarité ou une continuité triviales, et encore moins de recopier l'énoncé que l'interrogateur et le candidat connaissent tous les deux.
- Le candidat doit être précis dans ses propos, et, en particulier lorsqu'il énonce une définition, une propriété ou un théorème au programme de mathématiques, il doit énoncer l'ensemble des hypothèses sans en oublier ; le jury attend d'un candidat qu'il connaisse les résultats de cours.
- On attend également d'un candidat qu'il maîtrise les techniques de calcul en connaissant les concepts sous-jacents ; par exemple, maîtriser le procédé de calcul puis de recherche des racines du polynôme caractéristique ne dispense pas de connaître les définitions de valeur propre et de sous-espace propre ; lorsque plusieurs procédés de calcul sont possibles, par exemple pour la résolution d'un système linéaire ou la détermination du rang d'une matrice (méthode du pivot, substitution, combinaisons linéaires, etc.), le candidat peut utiliser celui qu'il préfère à condition d'être efficace.
- Les candidats doivent s'attendre à être interrogés sur la nature des objets qu'ils manipulent ; ils doivent pouvoir dire s'ils manipulent un nombre, une fonction, un vecteur ; par exemple, il n'est pas acceptable à ce niveau de confondre aire et primitive.
- Il est bon de connaître et de savoir justifier succinctement certaines inégalités usuelles, même si elles ne sont pas explicitement dans les programmes, comme par exemple : $\ln(x) \leqslant x-1$ sur $\mathbb{R}_+^*$, $1+x \leqslant \exp(x)$ sur $\mathbb{R}$, $|\sin(x)| \leqslant |x|$ sur $\mathbb{R}$, $x \leqslant \tan(x)$ sur $[0, \pi/2[$, etc.
- Le candidat doit toujours être très attentif : une mauvaise lecture de l'énoncé ou une erreur de calcul grossière du type « $x^a\, x^b = x^{a\,b}$ », de plus en plus fréquentes, auront en général des conséquences négatives sur l'évaluation du candidat.

5.2 – Polynômes à coefficients réels et complexes

- La résolution dans $\mathbb{C}$ d'équations polynomiales n'est pas toujours maîtrisée.
- Il peut être intéressant de savoir justifier qu'un polynôme à coefficients réels de degré impair admet nécessairement au moins une racine réelle, en particulier lorsque ce dégré est 3.
- On n'est pas obligé de passer par un calcul de discriminant pour résoudre $x^2-4=0$; cela permettra de disposer de plus de temps pour traiter les autres questions.
- Les racines n-ièmes de l'unité et leurs propriétés, notamment leur somme, doivent être connues.
- De même pour les identités et factorisations remarquables comme : $x^2+2\,x+1$, x^n-1, $x^{2\,n+1}+1$, etc.
- La confusion entre « *polynôme scindé* » et « *polynôme scindé à racines simples* » est hélas trop répandue, révélant un manque de précision du langage ; cette confusion est bien évidemment problématique lorsqu'on étudie le polynôme caractéristique d'une matrice carrée.

5.3 – Algèbre linéaire

- En algèbre comme ailleurs, on doit veiller à utiliser un vocabulaire précis ; il a trop souvent été constaté une confusion entre « *matrice symétrique* » et « *matrice de symétrie* », entre « *matrice symétrique* » et « *matrice orthogonale* », entre « *matrice diagonalisable* » et « *matrice inversible* », entre « *endomorphisme symétrique* » et « *isométrie* », entre le sous-espace vectoriel $\{\mathbf{0}\}$ et l'ensemble vide $\emptyset$; d'autre part, caractériser une matrice *orthogonale* ne se limite pas au simple calcul de son déterminant, contrairement au cas d'une matrice *inversible*.
- Les liens entre les notions de valeur propre, de rang, de noyau, gagneraient en général à être mieux

connus ; par exemple, les équivalences entre $\det(A) \neq 0$ et $\ker(A) = \{\mathbf{0}\}$, entre $\dim(\ker(A)) \geqslant 1$ et « 0 *est valeur propre de* A », entre « le vecteur non nul $\mathbf{u}$ est invariant par l'endomorphisme f » et « $\mathbf{u}$ *est vecteur propre de* f *pour la valeur propre* 1 ».

- Le calcul de l'image d'un vecteur par une projection, une symétrie, ou une autre application linéaire, ne nécessite pas toujours la détermination de la matrice de celle-ci dans une base donnée ; des considérations géométriques élémentaires permettent souvent d'arriver efficacement au résultat.
- Dans le même ordre d'idées, on peut déplorer que l'interprétation géométrique des symétries et projections, et de leurs sous-espaces propres, soit négligée au profit d'une simple propriété opérationnelle sur $f \circ f$.

5.4 – Analyse

- En plus de savoir déterminer le développement limité d'une fonction à l'ordre n au voisinage de x_0, il est nécessaire de comprendre son caractère local (au contraire des séries entières) et ses interprétations géométriques (tangente à la courbe et, si cette tangente est horizontale, le type de point singulier).
- Bien faire la différence entre les différents théorèmes de base : « *valeurs intermédiaires* », « *Théorème de Rolle* », « *accroissements finis* ».

5.5 – Intégration

- Lorsqu'on étudie l'intégrabilité d'une fonction sur un intervalle, penser à regarder en premier lieu si celle-ci est continue sur l'intervalle fermé ou, à défaut, sur l'intervalle ouvert, avant de détailler les problèmes éventuels aux bords.
- La recherche d'équivalents pour des fonctions (ou des suites) à valeurs positives, en vue d'étudier la convergence d'une intégrale ou d'une série, mériterait d'être davantage travaillée pendant l'année pour gagner du temps lors de l'oral.
- Dans l'étude de la convergence d'intégrales généralisées, il faut bien penser à énoncer les vérifications de base (continuité de la fonction, convergence de chaque intégrale dans le cas d'une décomposition en somme d'intégrales, etc.).
- La confusion entre primitive et intégrale a encore été trop souvent observée.

5.6 – Suites et séries

- Les suites récurrentes doivent être maîtrisées, ce qui est heureusement souvent le cas mais pas toujours, comme nous avons pu le déplorer lors de la session 2018.
- Les séries géométriques doivent être parfaitement maîtrisées, ce qui est heureusement très souvent le cas.
- Des lacunes ont pu être observées, ce qui peut révéler un travail préalable à améliorer, sur la recherche de limites ou d'équivalents lorsque n tend vers l'infini de suites de terme général comme : $\binom{n}{k} n^{-k}$, $(1 - a/n)^{bn}$, ou $2\left(\sqrt{n} - \sqrt{n-1}\right) - 1/\sqrt{n}$ (exemples).

5.7 – Géométrie

- De nombreux sujets de géométrie sont posés, y compris parmi les exercices d'informatique. C'est une particularité de la filière PT. Il est plus que conseillé de faire un dessin lisible ; cela permet de mieux comprendre le sujet, et est très apprécié par les examinateurs.
- Les sujets de géométrie utilisent fréquemment la trigonométrie ; il convient donc de pouvoir donner rapidement les formules utiles à l'exercice, et aussi d'être capable d'étudier des fonctions trigonométriques simples, qui paramètrent souvent les courbes.
- Il faut surtout que les candidats, au lieu de se précipiter sur les calculs, mettent en place une démarche de résolution et annoncent à l'examinateur la liste des tâches pour arriver à la solution du problème posé.
- Trop peu de candidats ont réussi à mener à bien l'étude d'une courbe paramétrée.
- Le lien entre la géométrie dans le plan et les nombres complexes (parties réelle et imaginaire, conjugaison, multiplication par un complexe de module 1, etc.) doit être maîtrisé, d'autant plus que cette correspondance est également très utilisée en informatique, notamment dans certains exercices posés.

5.8 – Fonctions de plusieurs variables et géométrie des courbes et surfaces

Liées aux notions de champs, de courbes et de surfaces, les fonctions de plusieurs variables, indispensables notamment en ingénierie mécanique, mériteraient davantage d'attention. En particulier, il est nécessaire de :

- savoir étudier leur continuité (ou plus généralement leur régularité $\mathcal{C}^1$) ;
- connaître la définition de ses dérivées partielles et savoir les calculer ;
- savoir utiliser la *règle de la chaîne* (dans le programme PT : « *Calcul des dérivées partielles d'ordres 1 et 2 de* $(u,v) \mapsto f(x(u,v), y(u,v))$ ») ;
- savoir déterminer les points critiques et leur nature ;
- savoir déterminer la tangente et la normale à une courbe ainsi que le plan tangent à une surface, à partir d'équations cartésienne ou paramétrique. Sur ce dernier point, une amélioration a pu être observée en 2018 ; une confirmation de cette amélioration est espérée dans les années qui viennent.

5.9 – Équations différentielles linéaires

- La résolution d'équations différentielles linéaires à coefficients constants avec second membre doit être maîtrisée, ce qui est heureusement très souvent le cas.
- Il en est de même pour les équations différentielles linéaires du premier ordre sans second membre et à coefficients non constants, pour lesquelles des lacunes ont été observées lors de la session 2018.

5.10 – Probabilités

- Encore plus qu'ailleurs, il faut lire attentivement l'énoncé et être précis dans son vocabulaire.
- On apprécie qu'un candidat justifie naturellement un résultat obtenu (probabilités totales, conditionnelles, etc) et donne des définitions correctes, notamment celle de l'indépendance de deux évènements, ou de deux variables aléatoires. Il est bien de prononcer le terme « *système complet d'évènements* » et encore mieux d'être en mesure de détailler de quoi il s'agit.
- On a cependant pu relever parfois des confusions entre : un évènement et sa probabilité ; évènements incompatibles et évènements indépendants ; probabilité de « A et B » et probabilité de « A sachant B » ; évènement et variable aléatoire, révélée par des notations fautives comme $\mathbb{P}(X)$ ou $\mathbb{E}(X > 0)$.

- Pour les variables aléatoires à valeurs dans $\mathbb{N}$, il peut être utile de connaître la formule de l'espérance du programme : $\mathbb{E}(X) = \sum_{n=1}^{\infty} \mathbb{P}(X \geqslant n) = \sum_{k=0}^{\infty} \mathbb{P}(X > k)$.

6 – Exercice d'algorithmique/simulation numérique

Les candidats ont en général été bien préparés pour cet oral.

L'ensemble des interrogateurs a pu constater une meilleure préparation moyenne des candidats qu'aux sessions précédentes, avec cependant trop souvent des lacunes sur l'aspect « *simulation numérique* » et notamment l'utilisation de la bibliothèque **numpy**. L'effort doit être poursuivi, notamment sur les points détaillés ci-dessous.

6.1 – Conseils généraux

- Lire attentivement l'énoncé ; il arrive très souvent que plusieurs phrases introductives présentent le contexte de l'exercice ; ne pas hésiter à solliciter l'interrogateur si on a le moindre doute, pour clarifier le problème et éviter tout contresens qui pourrait induire des réponses « *hors sujet* ».
- Si quelques lignes de code sont proposées à la compréhension, il est conseillé au candidat de taper ce code et de le comprendre en modifiant certains paramètres ; expliquer un code n'est pas le lire mot à mot mais décrire globalement ce qu'il fait et à quoi il sert.
- Ne pas hésiter à utiliser le brouillon mis à disposition avant de se jeter trop rapidement dans la programmation ou pour décrire l'ébauche d'un algorithme à l'interrogateur.
- Ne pas négliger les premières questions : elles contiennent le plus souvent des éléments de réponse pour la suite, voire des rappels.
- Ne pas hésiter à utiliser le mémento, surtout si le conseil en est donné par l'interrogateur.
- Ne pas hésiter à utiliser l'interpréteur pour effectuer des vérifications élémentaires.
- Il est indispensable de savoir utiliser les instructions `help` et `numpy.info` : il est normal de ne pas connaître toutes les fonctions apparaissant dans les exercices ; le nom de la fonction à utiliser est très souvent suggéré dans l'énoncé, notamment si cette fonction n'apparaît pas dans le mémento, et il faut donc savoir se renseigner à son sujet.
- Il faut savoir mettre en œuvre une démarche en cas d'erreur : faire des tests élémentaires dans la console, insérer des `print` pour contrôler pas à pas une exécution, lire attentivement et savoir utiliser les messages d'erreurs, etc. Il s'agit d'une compétence valorisée par le jury.
- Bien faire la distinction entre les entiers (type `int`) et les nombres à virgule flottante (type `float`), et maîtriser les conséquences induites. La connaissance des opérateurs `//` et `%` et la manipulation des complexes (notation `1j` et des écritures `z.real`, `z.imag` et `z.conjugate()`) n'étaient hélas pas maîtrisées par tous les candidats en 2018.
- La manipulation des chaînes de caractères fait aussi partie des capacités exigibles, et en particulier la connaissance des méthodes **split**, **strip**, **replace** qui peuvent être utiles pour la lecture de données structurées dans un fichier ASCII.
- La numérotation des éléments, le découpage et la concaténation des chaînes de caractères comme des listes doit être aussi maîtrisée, dont l'utilisation de l'indexation négative qui ne nécessite pas de connaître le nombre d'éléments (**nom[-1]** pour le dernier élément, **nom[-2]** pour l'avant-dernier, **nom[-3:]** pour les trois derniers, etc).
- Sauf si cela est spécifié dans l'énoncé (en vue de tester le candidat sur un algorithme de base), toutes les fonctions du langage et de ses bibliothèques peuvent être utilisées (`sum`, `max`, `min`, `sorted`, etc.).

- Il n'est pas nécessaire de définir systématiquement une fonction pour chaque tâche demandée, et, plus généralement, il n'y a aucun style de programmation imposé ; le candidat est évalué sur la maîtrise des outils mis à sa disposition et non sur le respect dogmatique de telle ou telle règle ou interdiction le plus souvent arbitraire.
- En revanche, **une fonction doit toujours être testée**, soit dans l'éditeur (F5 ou F9), soit dans la console, comme cela est spécifié dans l'en-tête de chaque énoncé.
- Préférer une boucle `for` à un `while` quand le nombre d'itérations est connu à l'avance. Préférer également une boucle non indexée « `for objet in iterable` » à une boucle indexée « `for i in range(len(iterable))` » lorsque la connaissance de l'indice i ne sert à rien.
- La distinction entre une liste (type `list`) et un vecteur (tableau `numpy.ndarray` à un seul indice) doit être parfaitement comprise ; les avantages et les inconvénients de ces deux types complémentaires doivent être connus, ainsi que les fonctions de conversion pour passer de l'un à l'autre (la fonction `numpy.array` et la méthode `tolist`).
- L'utilisation de variables globales n'est pas conseillée, et encore moins exigée.

6.2 – Gestion du temps

Quelques candidats perdent un temps considérable avec des pratiques peu adaptées pour une épreuve de 30 minutes :

- Il est bon de connaître et de savoir utiliser par exemple les fonctions intrinsèques `min`, `max`, `sum`, `sorted`, les méthodes `append`, `extend`, `sort`, `index` pour les listes, les méthodes `min`, `max`, `argmin`, `argmax`, `sum`, `mean`, `std`, `transpose`, `conjugate`, ... pour les tableaux `numpy.ndarray` (`T.real` et `T.imag` aussi pour un tableau de complexes), ainsi que les techniques de *slicing* (`U[debut:fin:pas]` pour une liste, une chaîne de caractères, ou un vecteur, `M[Ldeb:Lfin:dL,Cdeb:Cfin:dC]` pour une matrice, etc.).
- Il a encore été observé cette année un abus de la méthode `append` pour créer des listes simples. Un exemple caricatural observé plusieurs fois :

```
L = []
a = -1.2
L.append(a)
b = 3.4
L.append(b)
```

au lieu de

```
L = [-1.2,3.4]
```

- Même si les listes en compréhension ne sont pas exigibles, leur utilisation maîtrisée permet de gagner en efficacité et en lisibilité.
- Le tracé de la courbe représentative d'une fonction f sur un intervalle ne doit pas prendre plus que quelques minutes et quelques lignes. Par exemple :

```
X = np.linspace(x_min, x_max, nombre_de_points)
Y = f(X)  (ou Y = [ f(x) for x in X ] si la fonction f n'est pas vectorisée)
plt.plot(X, Y)
plt.show()
```

  `np` et `plt` désignant respectivement les modules `numpy` et `matplotlib.pyplot` ; la fonction f sera en général vectorisée si on utilise les fonctions mathématiques de `numpy`.
- Ne pas hésiter à réutiliser les fonctions créées dans les questions précédentes, ou même à créer de petites fonctions intermédiaires si cela peut être utile ; les exercices sont très souvent structurés dans cet esprit.

- L'écriture systématique de commentaires et d'en-têtes (`"docstring"`) pour les fonctions est déconseillée pour l'oral ; même si elle est légitimement préconisée en génie logiciel, elle fait perdre un temps précieux ; les explications peuvent être données oralement par le candidat.

6.3 – Algorithmique

- Les algorithmes du cours et leurs coûts de calcul doivent être connus (algorithmes de tri, méthodes par dichotomie, de Newton, d'Euler, des trapèzes, pivot de Gauss, algorithme d'orthonormalisation de Gram-Schmidt, algorithme d'Euclide, etc.). Leur connaissance est fréquemment évaluée. Lors de la session 2018, de trop nombreux candidats les ignoraient.
- D'autres algorithmes plus simples, comme par exemple le recensement d'éléments distincts et de leurs effectifs (ou de leurs fréquences) dans un itérable, doivent également être maîtrisés.
- La distinction claire entre *algorithme récursif* et *algorithme itératif* doit être acquise ; dans l'écriture d'une fonction récursive, un soin particulier doit être porté à la condition d'arrêt.

6.4 – À propos des fonctions

Trop de candidats n'ont pas parfaitement assimilé le concept de fonction.

La distinction entre ce que fait la fonction, ce que renvoie la fonction, et ce qu'affiche la fonction doit être parfaitement claire.

Voici quelques symptômes que l'on retrouve hélas trop souvent également parmi les étudiants de première année en école d'ingénieurs :

- Le nom de la fonction est réutilisé comme nom d'un autre d'objet dans la définition de la fonction, ce qui traduit une incompréhension.
- En dépit du bon sens, on effectue de nombreux appels de la même fonction avec les mêmes arguments, sans se rendre compte du caractère catastrophique d'une telle démarche. En voici un exemple caricatural, où `f` désigne une fonction d'arguments a, b, c, qui renvoie une liste de couples :

```
X = []
Y = []
for i in range(len(f(2.3,-1.2,0.4))) :
    X.append(f(2.3,-1.2,0.4)[i][0])
    Y.append(f(2.3,-1.2,0.4)[i][1])
```

au lieu de, par exemple,

```
P = f(2.3,-1.2,0.4)
T = numpy.array(P)
X = T[:,0].tolist()
Y = T[:,1].tolist()
```

  Si, en plus, la fonction `f` effectue à chaque fois un tirage pseudo-aléatoire de points, le résultat obtenu, en plus de prendre beaucoup trop de temps, devient faux.
- Beaucoup de candidats éprouvent des difficultés lorsque l'argument de la fonction est un objet de structure un peu complexe, comme par exemple une liste de couples, chaque couple contenant lui-même une chaîne de caractères suivie d'un couple de coordonnées ; dans ce cas, de nombreux candidats veulent créer d'abord l'objet, dans ou avant la définition de la fonction, au lieu d'extraire de l'objet les données à utiliser comme suit :

```
def f(L) :
    labels,X,Y = [],[],[]
    for element in L :
        label,coordonnees = element
        labels.append(label)
        x,y = coordonnees
        X.append(x)
        Y.append(y)
    ...
```

ou de façon plus compacte :

```
def f(L) :
    labels,X,Y = [],[],[]
    for (label, (x,y)) in L :
        labels.append(label)
        X.append(x)
        Y.append(y)
    ...
```

ou encore, en utilisant des listes en compréhension (non exigibles) :

```
def f(L) :
    labels = [ e[0] for e in L ]
    X = [ e[1][0] for e in L ]
    Y = [ e[1][1] for e in L ]
    ...
```

Notons que cette dernière difficulté est du même ordre que celle rencontrée lors de la lecture de données structurées dans un fichier ASCII.

6.5 – Simulation numérique

La maîtrise d'un outil informatique de simulation numérique et de calcul scientifique est indispensable pour réussir cette oral mais également dans la poursuite probable en école d'ingénieurs de la formation du candidat.

L'aptitude à faire tracer des courbes représentatives à l'aide du module **matplotlib.pyplot**, déjà évoquée ci-avant, doit être complétée par celle à faire tracer une courbe paramétrique (avec le même outil), et aussi à faire tracer un nuage de points, sans relier ceux-ci, à partir d'un tableau de valeurs.

Il est conseillé de bien connaître les propriétés des listes d'une part, et des tableaux **ndarray** du module **numpy** d'autre part, en particulier ce qui les différencie, de façon à pouvoir choisir le type le mieux adapté au problème à résoudre.

Trop souvent, un candidat utilise systématiquement des listes alors que des tableaux numériques seraient plus adaptés au cas à traiter.

Pour le calcul vectoriel et matriciel, il faut bien connaître les outils de base du module **numpy**, pour calculer numériquement le produit scalaire de deux vecteurs, le produit de deux matrices, le produit d'une matrice par un vecteur, le déterminant d'une matrice carrée (sous-module **numpy.linalg**), son inverse, ses valeurs propres, ses vecteurs propres, etc.

7 – Analyse des résultats

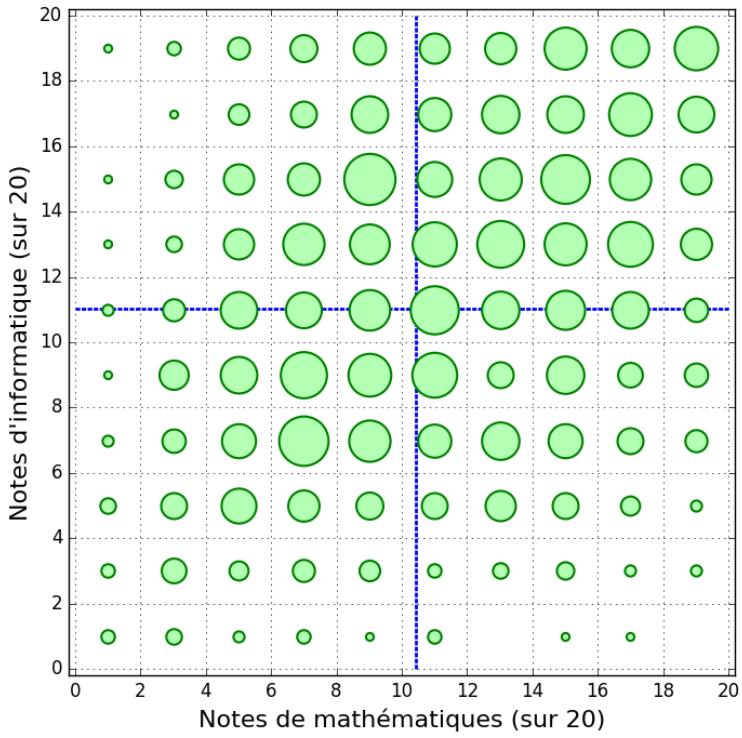
En 2018, 1480 candidats ont passé l'oral de « *Mathématiques et algorithmique* ». Chacun des 11 jours de l'oral, les 8 ou 9 jurys se sont efforcés de poser des exercices balayant l'ensemble du programme, tant en mathématiques qu'en algorithmique et simulation numérique.

Ainsi, 221 exercices différents d'analyse et de probabilités ont été proposés à 798 candidats contre 183 exercices différents de géométrie et d'algèbre proposés à 682 candidats.

162 exercices différents d'informatique à dominante algorithmique ont été posés à 768 candidats, contre 165 exercices à dominante « *simulation numérique* » pour 712 candidats.

Les statistiques sur les notes sont les suivantes [2] :

Oral 2018	**Note (sur 20)**	Math. (sur 10)	Algo. (sur 10)
Moyenne	**10,73**	5,22	5,51
Écart-type	**3,85**	2,37	2,36
Minimum	1	0	0
Maximum	20	10	10



Distribution des notes 2018

Éric Ducasse, Coordonnateur de l'épreuve orale de « *Mathématiques et algorithmique* » de la Banque PT, Le 12 juillet 2018.
eric.ducasse@ensam.eu

Annexe 1 – Nouveau mémento pour l'oral à compter d'août 2018

Ce nouveau mémento, destiné uniquement au passage de l'oral, est « *plus resserré* » que le précédent. Il sera fourni au candidat sous forme d'une seule feuille plastifiée au format A3 recto-verso.

Un mémento à vocation pédagogique, plus fourni, est disponible sur l'espace numérique de travail *Arts et Métiers* https://savoir.ensam.eu/moodle/course/view.php?id=1428 destiné aux étudiants des Arts et Métiers. Des supports de référence sont également disponibles sur ce site.

2. Rappelons que seule la note globale est communiquée au candidat.

Mémento Python 3 pour l'Oral de la Banque PT

Version PT–1.0
Forme inspirée initialement du mémento de Laurent Pointal, disponible ici : https://perso.limsi.fr/pointal/python:memento

Cette version sur la Banque PT : http://www.banquept.fr/spip.php?article237
Ceci est une version abrégée du mémento pédagogique utilisé à l'ENSAM disponible ici : https://savoir.ensam.eu/moodle/course/view.php?id=1428

Aide

`help(`*nom*`)` *aide sur l'objet* ***nom***
`help("nom_module.nom")` *aide sur l'objet* `nom` *du module* `nom_module`
`dir(`*nom*`)` *liste des noms des méthodes et attributs de* ***nom***
F1

Types de base

Entier, décimal, complexe, booléen, rien

`int`	`783`	`0` *zéro*	`-192`	`0b010` *binaire*	objets non mutables
`float`	`9.23`	`0.0`	`-1.7e-6`	(→ -1,7×10^{-6})	
`complex`	`1j`	`2+3j`	`1.3-3.5e2j`		
`bool`	`True`	`False`			
`NoneType`	`None`	(une seule valeur : « rien »)			

Objets itérables

- **Conteneurs numérotés** (*listes, tuples, chaînes de caractères*)

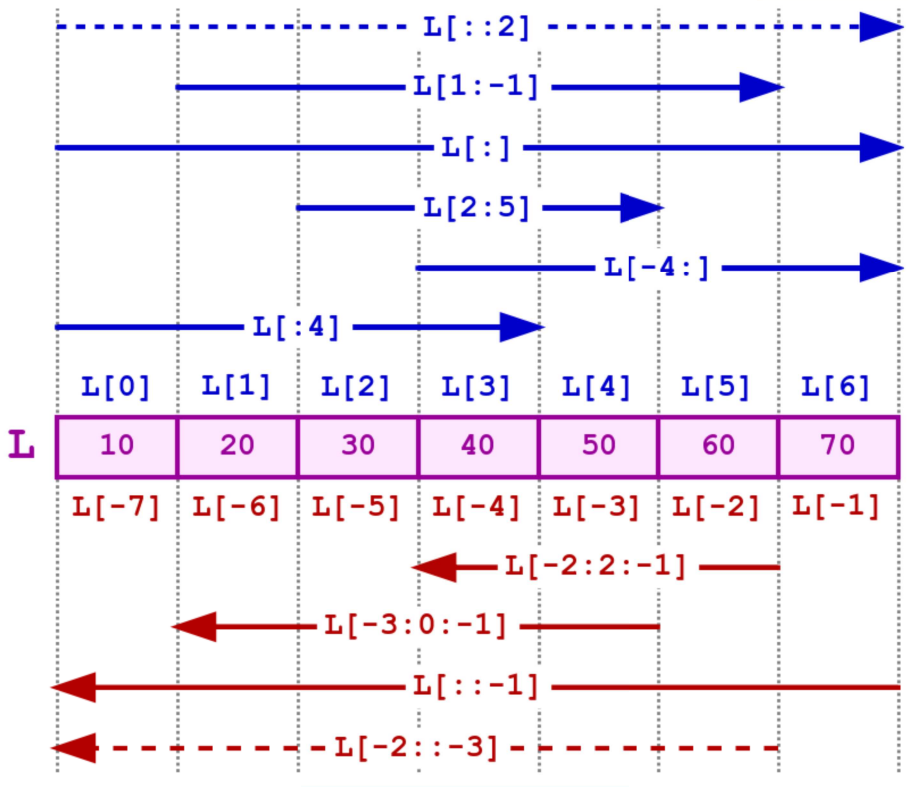
`list`	`[1,5,9]`	`["abc"]`	`[]`	`["x",-1j,["a",False]]`
`tuple`	`(1,5,9)`	`("abc",)`	`()`	`11,"y",[2-1j,True]`
`str`	`"abc"`	`"z"` *Singleton*	`""` *Objet vide*	
Nombre d'éléments `len(`*objet*`)` donne :	`3`	`1`	`0`	`3`

Conteneurs hétérogènes
Objets non mutables
expression juste avec des virgules → `tuple`

- **Itérateurs** (*objets destinés à être parcourus par* `in`)

`range(`*n*`)` : pour parcourir les *n* premiers entiers naturels, de 0 à *n*-1 inclus.
`range(`*n*`,`*m*`)` : pour parcourir les entiers naturels de *n* inclus à *m* exclus par pas de 1.
`range(`*n*`,`*m*`,`*p*`)` : pour parcourir les entiers naturels de *n* inclus à *m* exclus par pas de *p*.

Parcours de conteneurs numérotés



index à partir de 0

Accès à chaque **élément** par `L[`*index*`]`

`L[0]` → `10` ⇒ *le premier*
`L[1]` → `20` ⇒ *le deuxième*
`L[-1]` → `70` ⇒ *le dernier*
`L[-2]` → `60` ⇒ *l'avant-dernier*

Accès à une **partie** par `L[`*début inclus* `:` *fin exclue* `:` *pas*`]`

`L[2:5]`→ `[30,40,50]` ⇒ *indices 2,3 et 4*
`L[:4]` → `[10,20,30,40]` ⇒ *les 4 premiers*
`L[-4:]`→ `[40,50,60,70]` ⇒ *les 4 derniers*
`L[::2]`→ `[10,30,50,70]` ⇒ *de 2 en 2*
`L[:]` *tous : copie superficielle du conteneur*
`L[::-1]` *tous, de droite à gauche*
`L[-2::-3]`→ `[60,30]` ⇒ *de -3 en -3 en partant de l'avant-dernier*

Conteneurs : opérations génériques

`len(c)` `min(c)` `max(c)` `sum(c)`
nom `in c` → booléen, test de présence dans `c` *d'un élément identique (comparaison* `==`*) à* ***nom***
nom `not in c` → *booléen, test d'absence*
`c1 + c2` → *concaténation*
`c * 5` → *5 répétitions* (`c+c+c+c+c`)
`c.index(`***nom***`)` → *position du premier élément identique à* ***nom***
`c.index(`***nom***`,`***idx***`)` → *position du premier élément identique à* ***nom*** *à partir de la position* ***idx***
`c.count(`***nom***`)` → *nombre d'occurrences*

Opérations sur listes

modification « en place » de la liste `L` ***originale ces méthodes ne renvoient rien en général***

`L.append(`***nom***`)` *ajout d'un élément à la fin*
`L.extend(`***itérable***`)` *ajout d'un itérable converti en liste à la fin*
`L.insert(`***idx***`,`***nom***`)` *insertion d'un élément à la position* ***idx***
`L.remove(`***nom***`)` *suppression du premier élément identique (comparaison* `==`*) à* ***nom***
`L.pop()` *renvoie et supprime le dernier élément*
`L.pop(`***idx***`)` *renvoie et supprime l'élément à la position* ***idx***
`L.sort()` *ordonne la liste (ordre croissant)*
`L.sort(reverse=True)` *ordonne la liste par ordre décroissant*
`L.reverse()` *renversement de la liste*
`L.clear()` *vide la liste*

Chaînes de caractères

Caractères spéciaux : `"\n"` *retour à la ligne* `"\t"` *tabulation* `"\\"` *« backslash \ »* `"\""` *ou* `'"'` *guillemet "* `"'"` *ou* `'\''` *apostrophe '*

Méthodes sur les chaînes

Une chaîne n'est pas modifiable ; ces méthodes renvoient en général une nouvelle chaîne ou un autre objet

`"b-a-ba".replace("a","eu")` → `'b-eu-beu'` *remplacement de toutes les occurrences*
`" \tUne phrase.\n ".strip()` → `'Une phrase.'` *nettoyage du début et de la fin*
`"des  mots\tespacés".split()` → `['des','mots','espacés']`
`"1.2,4e-2,-8.2,2.3".split(",")` → `['1.2','4e-2','-8.2','2.3']`
`" ; ".join(["1.2","4e-2","-8.2","2.3"])` → `'1.2 ; 4e-2 ; -8.2 ; 2.3'`

Conversions

`int("15")` → `15`
`int(-15.56)` → `-15` (*troncature*)
`round(-15.56)` → `-16` (*arrondi*)
`float(-15)` → `-15.0`
`float("-2e-3")` → `-0.002`
`complex("2-3j")` → `(2-3j)`
`complex(2,-3)` → `(2-3j)`
`list(`*x*`)` *Conversion d'un itérable en liste*
`list(range(4,-1,-1))` → `[4,3,2,1,0]`
`sorted(`*x*`)` *Conversion d'un itérable en liste ordonnée (ordre croissant)*
`sorted(x,reverse=True)` *Conversion d'un itérable en liste ordonnée (ordre décroissant)*
`tuple(`*x*`)` *Conversion en tuple*
`str(`*x*`)` *Conversion en chaîne de caractères*

Mathématiques

- *Opérations*

`+ - * /`
`**` *puissance* `2**10` → `1024`
`//` *quotient de la division euclidienne*
`%` *reste de la division euclidienne*

- *Fonctions intrinsèques*

`abs(`*x*`)` *valeur absolue / module*
`round(`*x*`,`*n*`)` *arrondi du* `float` *x à n chiffres après la virgule*
z`.real` → *partie réelle de z*
z`.imag` → *partie imaginaire de z*
z`.conjugate()` → *conjugué de z*

Logique booléenne

- *Opérations booléennes*

`not A` « *non A* »
`A and B` « *A et B* »
`A or B` « *A ou B* »
`(not A) and (B or C)` *exemple*

`True` *ou* `False`

- *Opérateurs renvoyant un booléen*

nom1 `is` ***nom2*** *2 noms du même objet ?*
nom1 `==` ***nom2*** *valeurs identiques ?*
Autres comparateurs :
`<` `>` `<=` `>=` `!=` (≠)
nom_objet `in` ***nom_iterable***
l'itérable ***nom_iterable*** *contient-il un objet de valeur identique à celle de* ***nom_objet*** *?*

Évaluation d'une durée d'exécution, en secondes — `time`

```
from time import time
debut = time()
⋮ (instructions)
duree = time() - debut
```

Liste en compréhension

- *Inconditionnelle / conditionnelle*

```
L = [ f(e) for e in itérable ]
L = [ f(e) for e in itérable if b(e)]
```

Importation de modules

Module `mon_mod` ⇔ Fichier `mon_mod.py`

- *Importation d'objets par leurs noms*

```
from mon_mod import nom1,nom2
```

- *Importation avec renommage*

```
from mon_mod import nom1 as n1
```

- *Importation du module complet*

```
import mon_mod
    ⋮
... mon_mod.nom1 ...
```

- *Importation du module complet avec renommage*

```
import mon_mod as mm
    ⋮
... mm.nom1 ...
```

L'examinateur n'attend pas du candidat une connaissance encyclopédique du langage Python, mais une utilisation raisonnée des principes algorithmiques et une mise en pratique des connaissances de base.
L'utilisation de l'aide en ligne est encouragée, mais ne doit pas masquer une ignorance sur ces aptitudes.

Ce mémento est fourni à titre indicatif. Il ne faut le considérer :
✧ *ni comme exhaustif (en cas de problème sur un exercice particulier, si une fonction ou une commande indispensable était absente de la liste, l'interrogateur pourrait aider le candidat),*
✧ *ni comme exclusif (une fonction ou une commande absente de cette liste n'est pas interdite : si un candidat utilise à très bon escient d'autres fonctions MAIS sait aussi répondre aux questions sur les fonctions de base, il n'y a pas de problème),*
✧ *ni comme un minimum à connaître absolument (l'examinateur n'attend pas du candidat qu'il connaisse parfaitement toutes ces fonctions et ces commandes).*
Les fonctions et commandes présentées doivent simplement permettre de faire les exercices proposés aux candidats.

Mémento Python 3 pour l'Oral de la Banque PT

Simulation numérique et calcul scientifique

Fonctions mathématiques

```
import numpy as np
```

Les fonctions de numpy sont vectorisées

`np.pi, np.e` → *Constantes π et e*

`np.abs, np.sqrt, np.exp, np.log, np.log10, np.log2` → `abs`, *racine carrée, exponentielle, logarithmes népérien, décimal, en base 2*

`np.cos, np.sin, np.tan` → *Fonctions trigonométriques (angles en radians)*

`np.arccos, np.arcsin, np.arctan` → *Fonctions trigonométriques réciproques*

`np.arctan2(y,x)` → *Angle dans*]–π,π]

`np.cosh, np.sinh` *(trigonométrie hyperbolique)*

Tirages pseudo-aléatoires

Modules `random` *et* `numpy.random`

```
import random
```

`random.random()` → *Valeur flottante dans l'intervalle* [0,1[*(loi uniforme)*

`random.randint(a,b)` → *Valeur entière entre* ***a*** *inclus et* ***b*** *inclus (équiprobabilité)*

`random.choice(L)` → *Un élément de la liste* ***L*** *(équiprobabilité)*

`random.shuffle(L)` → `None`, *mélange la liste* ***L*** *« en place »*

```
import numpy.random as rd
```

`rd.rand(n0,…,nd-1)` → *Tableau de forme* $(n_0,\dots,n_{d-1})$, *de flottants dans l'intervalle* [0,1[*(loi uniforme)*

`rd.randint(a,b,shp)` → *Tableau de forme* ***shp***, *d'entiers entre* ***a*** *inclus et* ***b*** *exclu (équiprobabilité)*

`rd.choice(Omega,shp)` → *Tableau de forme* ***shp***, *d'éléments tirés avec remise dans* ***Omega*** *(équiprobabilité)*

Aide numpy/scipy

`np.info(nom_de_la_fonction)`

Conversion ndarray ↔ liste

`T = np.array(L)` → *Liste en tableau, type de données automatique*

`L = T.tolist()` → *Tableau en liste*

Tableaux numpy.ndarray : généralités

Un tableau `T` *de type* `numpy.ndarray` *(« n-dimensional array »)* *est un conteneur homogène dont les valeurs sont stockées en mémoire de façon séquentielle.*

`T.ndim` → *« dimension* ***d*** *» = nombre d'indices (1 pour un vecteur, 2 pour une matrice)*

`T.shape` → *« forme » = plages de variation des indices, regroupées en* `tuple` $(n_0,n_1,\dots,n_{d-1})$ *: le premier indice varie de* `0` *à* n_0-1, *le deuxième de* `0` *à* n_1-1, *etc.*

`T.size` → *nombre d'éléments, soit* $n = n_0 \times n_1 \times \cdots \times n_{d-1}$

`T.dtype` → *type des données contenues dans le tableau*

Création d'un tableau

shp *est la forme du tableau créé,* ***data_type*** *le type de données contenues dans le tableau (*`np.float` *si l'option* `dtype` *n'est pas utilisée)*

`T = np.zeros(shp,dtype=data_type)` → *tout à* `0`/`False`

`T = np.ones(shp,dtype=data_type)` → *tout à* `1`/`True`

Générateurs

`np.eye(n)` → *matrice identité d'ordre* ***n***

`np.diag(V)` → *matrice diagonale dont la diagonale est le vecteur* `V`

Générateurs

`np.linspace(a,b,n)` → ***n*** *valeurs régulièrement espacées de* ***a*** *à* ***b*** *(bornes incluses)*

`np.arange(xmin,xmax,dx)` → *de* x_{min} *inclus à* x_{max} ***exclu*** *par pas de* ***dx***

Vecteurs

- ***Un vecteur*** `V` ***est un tableau à un seul indice***
- *Comme pour les listes,* `V[i]` *est le (****i*** *+1)-ième coefficient, et l'on peut extraire des sous-vecteurs par :* `V[:2]`, `V[-3:]`, `V[::-1]`, etc.

Si `c` est un nombre, les opérations `c*V`, `V/c`, `V+c`, `V-c`, `V//c`, `V%c`, `V**c` *se font sur chaque coefficient*

Si `U` est un vecteur de même dimension que `V`, les opérations `U+V`, `U-V`, `U*V`, `U/V`, `U//V`, `U%V`, `U**V` *sont des* ***opérations terme à terme***

- ***Produit scalaire*** : `U.dot(V)` *ou* `np.dot(U,V)` *ou* `U@V`

Matrices

- ***Une matrice*** `M` ***est un tableau à deux indices***
- `M[i,j]` *est le coefficient de la (****i*** *+1)-ième ligne et (****j***+1)-ième colonne
- `M[i,:]` *est la (****i*** *+1)-ième ligne,* `M[:,j]` *la (****j***+1)-ième colonne, `M[i:i+h,j:j+l]` *une sous-matrice* $h \times l$
- ***Opérations terme à terme : voir « Vecteurs » ci-contre***
- ***Produit matriciel*** : `M.dot(V)` *ou* `np.dot(M,V)` *ou* `M@V`

`M.transpose(),M.trace()` → *transposée, trace*

Matrices carrées *uniquement (algèbre linéaire) :*

```
import numpy.linalg as la        ("Linear algebra")
```

`la.det(M), la.inv(M)` → *déterminant, inverse*

`vp = la.eigvals(M)` → `vp` *vecteur des valeurs propres*

`vp,P = la.eig(M)` → `P` *matrice de passage*

`la.matrix_rank(M), la.matrix_power(M,p)`

`X = la.solve(M,V)` → *Vecteur solution de* `M X = V`

Intégration numérique

```
import scipy.integrate as spi
```

`spi.odeint(F,Y0,Vt)` → *renvoie une solution numérique du problème de Cauchy* $\mathbf{Y}'(t) = \mathbf{F}(\mathbf{Y}(t),t)$, *où* $\mathbf{Y}(t)$ *est un vecteur d'ordre n, avec la condition initiale* $\mathbf{Y}(t_0) =$ `Y0`, *pour les valeurs de t dans le vecteur* `Vt` *commençant par* t_0, *sous forme d'une matrice* $n \times k$

`spi.quad(f,a,b)[0]` → *renvoie une évaluation numérique de l'intégrale :* $\int_a^b f(t)\,dt$

Statistiques

Sans l'option `axis`, *un tableau* `T` *est considéré comme une simple séquence de valeurs*

`T.max(), T.min(), T.sum()`

`T.argmax(), T.argmin()` → *indices séquentiels du maximum et du minimum*

`T.sum(axis=d)` → *sommes sur le (****d****-1)-ème indice*

`T.mean(),T.std()` → *moyenne, écart-type*

Graphiques

```
import matplotlib.pyplot as plt
```

`plt.figure(mon_titre, figsize=(W,H))` *crée ou sélectionne une figure dont la barre de titre contient* ***mon_titre*** *et dont la taille est* $W \times H$ *(en inches, uniquement lors de la création de la figure)*

`plt.plot(X,Y,dir_abrg)` *trace le nuage de points d'abscisses dans* `X` *et d'ordonnées dans* `Y` *;* ***dir_abrg*** *est une chaîne de caractères qui contient une couleur (*`"r"`*-ed,* `"g"`*-reen,* `"b"`*-lue,* `"c"`*-yan,* `"y"`*-ellow,* `"m"`*-agenta,* `"k"` *black), une marque (*`"o"` *rond,* `"s"` *carré,* `"*"` *étoile,…) et un type de ligne (*`""` *pas de ligne,* `"-"` *plain,* `"--"` *dashed,* `":"` *dotted, …) ; options courantes :* `label=…`, `linewidth=…`, `markersize=…`

`plt.axis("equal"), plt.grid()` *repère orthonormé, quadrillage*

`plt.xlim(a,b), plt.ylim(a,b)` *plages d'affichage ; si a > b, inversion de l'axe*

`plt.xlabel(axe_x,size=s,color=(r,g,b)), plt.ylabel(axe_y,…)` *étiquettes sur les axes, en réglant la taille* ***s*** *et la couleur de la police de caractères (****r****,* ***g*** *et* ***b*** *dans* [0,1])

`plt.legend(loc="best",fontsize=s)` *affichage des labels des* `"plot"` *en légende*

`plt.show()` *affichage des différentes figures et remise à zéro*

Lecture de fichier texte

Le « ***chemin*** » d'un fichier est une chaîne de caractères

- ***Lecture intégrale d'un seul bloc***

```
with open(chemin,"r") as f:
    texte = f.read()
```

ou

```
f = open(chemin,"r")
texte = f.read()
f.close()
```

(***ne pas oublier de fermer le fichier***)

- ***Lecture de la liste de toutes les lignes***

```
with open(chemin,"r") as f:
    lignes = f.readlines()
```

ou

```
f = open(chemin,"r")
lignes = f.readlines()
f.close()
```

(***Nettoyage éventuel des débuts et fins de lignes***)

```
lignes = [c.strip() for c in lignes])
```

- ***Lecture et traitement simultanés, ligne par ligne***

```
with open(chemin,"r") as f:
    for ligne in f :
        (traitement sur ligne)
```

Annexe 2 – Exemples d'exercices d'informatique

Ces exercices ayant été posés de nombreuses fois au cours des dernières sessions, ils sont communiqués aux futurs candidats à titre d'exemples. Attention : il ne sont pas forcément représentatifs de tous les exercices se trouvant dans la banque d'exercices.

D'autres exercices publiés sont joints aux rapports 2015 et 2016.

2018.1 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

Un échiquier est un plateau de 8 lignes et 8 colonnes. Ces lignes et ces colonnes seront, dans cet exercice, numérotées de 0 à 7. Une position de l'échiquier est un couple $[i, j]$ d'entiers compris entre 0 et 7 inclus, avec i le numéro de ligne et j le numéro de colonne.
Un cavalier placé sur l'échiquier se déplace en bougeant de 2 cases dans une direction (verticale ou horizontale) et de 1 case perpendiculairement. Si le cavalier est loin des bords de l'échiquier, il a 8 possibilités de déplacements, mais il en a moins s'il est près des bords.

1. Illustrer sur un brouillon les deux cas énoncés précédemment.
2. Écrire une fonction `valide` prenant en argument deux entiers relatifs i et j et vérifiant que le couple $[i, j]$ est bien une position de l'échiquier. `valide` doit renvoyer un booléen.
3. Écrire une fonction `coupsSuivants` prenant en argument une position $[i, j]$ et renvoyant la liste des positions que peut atteindre un cavalier placé en $[i, j]$ en un seul coup.
4. Écrire une fonction `cavalier` prenant en argument une position $[a, b]$ et renvoyant un tableau T carré d'ordre 8 telle que $T[i, j]$ est le nombre minimum de coups nécessaires à un cavalier placé en position $[a, b]$ pour arriver à la position $[i, j]$.

2018.2 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

---

1. On considère un nombre n. Que donne la commande `list(str(n))` ?
2. Comment récupérer le chiffre des unités d'un nombre entier donné ? Celui des dizaines ?
   Tester cette méthode sur le nombre 2015.
3. Écrire une fonction **`transforme`** d'argument un entier naturel n et qui renvoie le nombre obtenu par la méthode suivante :
   - les chiffres de rang impair (en partant des unités) sont inchangés.
   - les chiffres de rang pair (en partant des unités) sont doublés, si ce double est supérieur ou égal à 10 on le remplace par la somme de ses chiffres (par exemple, 3 est remplacé par 6 et 7 est remplacé par 5).

   Exemple : 43281 est transformé en 46271
4. Déterminer les nombres inférieurs à 10000 invariants par la fonction **`transforme`**. Expliquer le résultat obtenu.

2018.3 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

Dans cet exercice, un segment $[a, b]$ $(a \leqslant b)$ est représenté par une liste de taille 2 : `[a,b]`.

1. Deux segments sont disjoints si leur intersection est vide. Écrire une fonction **`disjoints`** de deux arguments **`i1`** et **`i2`** qui teste si les segments **`i1`** et **`i2`** sont disjoints. La fonction renvoie le booléen **`True`** s'ils sont disjoints, **`False`** sinon.
2. La fusion de deux segments a comme minimum le plus petit des minima des deux segments, et comme maximum le plus grand des maxima des deux segments. Écrire une fonction **`fusion`** de deux arguments **`i1`** et **`i2`** et qui renvoie le segment correspondant à la fusion de **`i1`** et **`i2`**.
3. Une « *liste bien formée* » est une liste de segments qui vérifie les propriétés suivantes :
   - les segments sont deux à deux disjoints ;
   - les segments de la liste sont classés par ordre croissant, en considérant qu'un segment **`i1`** est strictement plus petit qu'un segment **`i2`** si et seulement si le maximum de **`i1`** est strictement inférieur au minimum de **`i2`**.

   a) Les listes suivantes sont-elles bien formées ?
      - `L1=[[0,1],[2,5],[3,6]]`
      - `L2=[[2,5],[0,1],[3,6]]`
      - `L3=[[0,1],[2,3],[4,6]]`

   b) Écrire une fonction récursive **`verifie`** qui teste si une liste de segments est bien formée.
4. Écrire une fonction récursive **`appartient`** de deux arguments **`x`** et **`L`** qui teste si la valeur **`x`** est élément d'un des segments de la liste **`L`** bien formée.

2018.4 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

Dans cet exercice, un segment $[a, b]$ ($a \leqslant b$) est représenté par une liste de taille 2 : `[a,b]`.

1. Deux segments sont disjoints si leur intersection est vide. Écrire une fonction **`disjoints`** de deux arguments `i1` et `i2` qui teste si les segments `i1` et `i2` sont disjoints. La fonction renvoie le booléen `True` s'ils sont disjoints, `False` sinon.
2. Une « *liste bien formée* » de segments est une liste qui vérifie :
   - les segments sont deux à deux disjoints ;
   - la liste est ordonnée selon ses segments croissants : un segment `i1` est strictement plus petit qu'un segment `i2` si et seulement si le maximum de `i1` est strictement inférieur au minimum de `i2`.

   Écrire une fonction **`decouper`** de deux arguments, une valeur `x` et une liste de segments `L` bien formée, qui renvoie un triplet composé de :
   - une liste bien formée contenant les segments strictement inférieurs à la valeur `x` ;
   - une liste bien formée contenant, s'il existe, le segment contenant la valeur `x`, et sinon, une liste vide ;
   - une liste bien formée contenant les segments strictement supérieurs à la valeur `x`.
3. Écrire une fonction **`inserer`** de deux arguments, un segment `i` et une liste de segments `L` bien formée. Cette fonction renvoie une liste bien formée de segments contenant les segments de `L` qui sont disjoints de `i` et :
   - soit l'intervalle `i` si tous les segments contenus dans `L` lui sont disjoints ;
   - soit le segment union de `i` et des segments contenus dans `L` qui ne sont pas disjoints de `i`.

2018.5 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

Dans cet exercice, on manipule des suites (finies) d'entiers sous forme de listes d'entiers. Ainsi la suite $(0, 1, 2, 3)$ sera représentée par la liste `[0,1,2,3]`.

La liste est croissante (respectivement décroissante, monotone) si la suite est croissante (respectivement décroissante, monotone).

1. Écrire une fonction **`estCroissante`** qui teste si une liste d'entiers est croissante. Cette fonction devra être de complexité linéaire.
2. Écrire de même une fonction **`estDecroissante`** qui teste si une liste d'entiers est décroissante.
3. Écrire également une fonction **`estMonotone`** qui teste si une liste d'entiers est monotone.
4. Soit la liste $\mathtt{L} = [\mathtt{u}_0, \mathtt{u}_1, \cdots, \mathtt{u}_{n-1}]$ de longueur n. Une monotonie de `L` est un couple d'indices (i, j) tel que $0 \leqslant i < j < n$, que la sous-liste $[\mathtt{u}_i, \mathtt{u}_{i+1}, \cdots, \mathtt{u}_j]$ est monotone et qu'elle ne l'est plus si on l'étend, à droite ou à gauche, d'un élément supplémentaire (lorsque c'est possible). La monotonie est dite « *banale* » lorsque $j = i + 1$.

   a) Proposez une liste d'entiers de longueur 5 qui ne présente que des monotonies banales. Peut-on avoir deux termes consécutifs égaux dans une liste ne présentant que des monotonies banales ?

   b) Écrire une fonction **`cahots`**, de complexité linéaire, qui teste si une liste ne comporte que des monotonies banales.

   c) Après avoir créé une liste arbitraire `L` de valeurs toutes distinctes, on peut l'ordonner par `L.sort()`. Imaginer ensuite une méthode pour réordonner `L` de manière à ce qu'elle ne comporte que des monotonies banales.

2018.6 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

---

1. **Compréhension de code** : que fait le code *Python* suivant ? Quelle est l'une de ses particularités ?

```
def ordonner(L) :
    if len(L) <= 1 :
        return L
    e0 = L[0]
    n = 1
    Linf, Lsup = [], []
    for ei in L[1:]:
        if ei == e0 :
            n += 1
        elif ei < e0 :
            Linf.append(ei)
        else :
            Lsup.append(ei)
    return ordonner(Linf)+n*[e0]+ordonner(Lsup)
```

2. Écrire une fonction **less** de deux complexes **z1** et **z2** qui renvoie **True** si et seulement si $\mathcal{R}e(z_1) < \mathcal{R}e(z_2)$ ou « $\mathcal{R}e(z_1) = \mathcal{R}e(z_2)$ et $\mathcal{I}m(z_1) < \mathcal{I}m(z_2)$ ».
3. En s'inspirant du code définissant la fonction **ordonner**, écrire puis tester une fonction **ordonnerDansC** qui ordonne une liste de complexes selon la relation d'ordre définie précédemment.
4. Tester la fonction **ordonnerDansC** sur une liste de complexes dont les parties réelles et imaginaires sont des entiers tirés au hasard entre -10 et 10. On pourra utiliser l'instruction **randint** du module **random**.
5. En utilisant le module **cmath**, créer la liste des $(\,20+\cos(10\,a)\,)\exp(\mathrm{i}\,a)$, pour a variant de $-\pi$ à π avec un pas de $\pi/100$. Ordonner ces points par **ordonnerDansC**. Faire tracer dans le plan complexe le nuage des points ainsi que la ligne les reliant. On utilisera **plot**, **axis** et **show** du module **matplotlib.pyplot** et le repère sera rendu orthonormé par **axis('equal')**.

2018.7 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

Soit n un entier supérieur à 2. On considère l'ensemble S_n des bijections de l'ensemble $\{0, 1, 2, \cdots, n-1\}$ dans lui-même.
On code une bijection f par la liste des n premiers entiers donnée par $[f(0), f(1), \cdots, f(n-1)]$. Par exemple, la liste $[3, 0, 4, 1, 2, 5]$ code $\phi \in S_6$ la bijection $\phi : \{0, 1, 2, 3, 4, 5\} \to \{0, 1, 2, 3, 4, 5\}$ définie par

$$\phi(0) = 3, \quad \phi(1) = 0, \quad \phi(2) = 4, \quad \phi(3) = 1, \quad \phi(4) = 2, \quad \text{et } \phi(5) = 5.$$

1. Soit f une bijection codée par une liste L. Que vaut $f(i)$?
2. Écrire une fonction `rond` qui reçoit en arguments deux listes $L1$, $L2$ (codant deux bijections f et g de S_n) et renvoie la liste codant $f \circ g$.

   On définit l'application $\epsilon : S_n \to \mathbb{Q}$ en posant

$$\epsilon(f) = \prod_{0 \leqslant i < j \leqslant n-1} \frac{f(i) - f(j)}{i - j}$$

3. Coder ϵ.
4. Que fait l'instruction `sample` du module `random` ?
5. Vérifier pour quelques valeurs de n et sur 10 bijections f de S_n choisies aléatoirement que l'on a $\epsilon(f) = \pm 1$.
6. En s'inspirant de la méthode de la question précédente, conjecturer une relation entre $\epsilon(f \circ g), \epsilon(f)$ et $\epsilon(g)$.

2018.8 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

On souhaite créer la liste de tous les nombres premiers inférieurs au égaux à un certain nombre n, puis tester la primalité d'un entier naturel.

1. Créer une liste `L30` de 31 booléens `True` (indices de de 0 à 30), qui servira de liste-test.

   On va créer une liste L de booléens contenant des `False` aux indices qui ne sont pas premiers et des `True` aux indices qui le sont. Il faudra donc modifier tous les éléments de la liste L dont l'indice n'est pas premier.
2. Écrire une fonction **`modifier`** de deux arguments, une liste L de booléens et un entier naturel p, qui modifie la liste L de la façon suivante : Si p vaut zéro, met le premier élément de la liste L à `False` ; si p vaut un, met le deuxième élément de la liste L à `False` ; sinon, met à `False` tous les éléments de L dont l'indice est un multiple de p autre que p. Noter que cette fonction ne renvoie rien.
3. En déduire une fonction **`premiers`** d'argument n renvoyant la liste des nombres premiers inférieurs ou égaux à n. Rappelons qu'un nombre inférieur ou égal à n est premier s'il n'est divisible par aucun entier inférieur ou égal à $\sqrt{n}$.
4. Tester la primalité des nombres suivants en minimisant le temps de calcul :
   `696937, 4862592 , 9412213 , 39841247, 99770809, 5263996587, 2358869562457`.

2018.9 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

Soit la suite $(u_n)_{n\in\mathbb{N}}$ de nombres entiers dépendant de sa valeur initiale $u_0 \in \mathbb{N}^*$ et définie par :

$$\forall n \in \mathbb{N}\ ,\ u_{n+1} = \begin{cases} u_n \ / \ 4 \text{ si } u_n \text{ est un multiple de 4} \\ (3\,u_n + 2) \ / \ 2 \text{ sinon (division entière)} \end{cases}$$

1. À l'aide notamment des instructions **`plot`** et **`show`** de la bibliothèque **`matplotlib.pyplot`**, représenter graphiquement les 30 premiers termes de la suite pour $u_0 = 25$, puis pour $u_0 = 37$.
2. On conjecture que pour tout entier naturel u_0 non nul, il existe un plus petit entier n, appelé « *durée de vol* », tel que $u_n = 1$.
   Que se passe-t-il s'il existe $n \in \mathbb{N}$ tel que $u_n = 1$?
3. Écrire une fonction **`vol`** d'argument un entier d, renvoyant la durée de vol lorsque $u_0 = d$ ainsi que la valeur maximale atteinte par la suite, que l'on appellera « *altitude* ».
4. Représenter cette altitude en fonction du point de départ d, pour les valeurs de d inférieures à 1000.
5. Pour quelles valeurs de u_0 inférieures à 1000, atteint-on une altitude supérieure à 100 000 ? Donner alors les valeurs de u_0, de la durée de vol et de l'altitude.
6. Vérifier que ces valeurs correspondent aussi à des durées de vol maximales.

2018.10 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

On considère les triangles de nombres du type

```
1  1  0  0  1  0  1
0  1  0  1  1  1
1  1  1  0  0
0  0  1  0
0  1  1
1  0
1
```

Chaque ligne est constituée de 0 et de 1. La première est donnée et les suivantes sont construites de la manière suivante : on prend les deux entiers situés respectivement au-dessus et au-dessus à droite de son emplacement. On les additionne et on garde le reste de la division euclidienne de cette somme par 2.

On représente le triangle comme une liste de listes.

Un triangle de ce type qui compte autant de 0 que de 1 est dit *equilibré*.

1. Écrire une fonction **`suivante`** d'argument une liste de zéros et de uns correspondant à une ligne et renvoyant la liste représentant la ligne suivante.
2. Écrire une fonction **`triangle`** d'argument une liste donnant la première ligne et renvoyant la liste de listes représentant le triangle. La tester avec la première ligne de l'exemple.
3. Écrire une fonction **`afficher`** d'argument une liste de listes représentant un triangle, qui ne renvoie rien mais qui affiche dans la console le triangle correspondant, comme sur l'exemple ci-dessus.
4. Écrire une fonction **`equilibreQ`** d'argument un triangle et testant si ce triangle est équilibré.
5. Déterminer le nombre de triangles équilibrés ayant une première ligne de 4 nombres.
6. Écrire une fonction **`nbteq`** d'argument n renvoyant le nombre de triangles équilibrés ayant une première ligne de n nombres.

   Faire afficher le résultat pour n variant de 1 à 15.

2018.11 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

Pour un entier naturel n non nul fixé, on appelle *vecteur creux* de $\mathbb{R}^n$ un vecteur dont au moins la moitié des coefficients sont nuls. On code ces vecteurs à l'aide d'une liste de deux listes. La première est la liste des valeurs des coefficients non nuls, la seconde celle de leurs indices, rangés en ordre croissant.

Par exemple, le vecteur

$$v = \begin{pmatrix} 1 & 3 & 0 & 4 & 0 & 0 & 1 & 2 & 0 & 0 & 0 & 0 \end{pmatrix}$$

est codé par

$$[[1, 3, 4, 1, 2], [0, 1, 3, 6, 7]]$$

On a en effet $v_0 = 1$, $v_1 = 3$, $v_3 = 4$, $v_6 = 1$ et $v_7 = 2$. Les autres coefficients sont nuls.

1. Écrire une fonction **`creux`** d'argument une liste simple représentant un vecteur v qui renvoie un booléen indiquant si v est creux ou pas.
2. Écrire deux fonctions : **`coder`** d'argument un vecteur qui renvoie son codage « creux » ; **`decoder`** de deux arguments C et n, qui restitue le vecteur de dimension n à partir de son codage « creux » C.

   On construit maintenant des fonctions adaptées pour le codage « creux ».
3. Écrire une fonction **`smul`** de deux arguments C et a, où C est le codage « creux » d'un vecteur v et a un scalaire, qui renvoie le codage « creux » du vecteur $a\,v$.
4. Écrire une fonction **`pscal`** de deux arguments **`C1`** et **`C2`**, codages « creux » de deux vecteurs v_1 et v_2, qui renvoie le produit scalaire de ces deux vecteurs.
5. Écrire une fonction **`add`** de deux arguments **`C1`** et **`C2`**, codages « creux » de deux vecteurs v_1 et v_2, qui renvoie le codage « creux » du vecteur $v_1 + v_2$. Cette somme est-elle toujours un vecteur creux ?

2018.12 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

On dit qu'un nombre entier n est une puissance finale d'un nombre entier non nul a si l'écriture en base 10 de a^n se termine par n, par exemple : $2^{36} = 68719476736$ donc 36 est une puissance finale de 2.

1. Écrire une fonction booléenne `PF` de deux arguments a et n, qui teste si n est une puissance finale de a.
2. Écrire une fonction `LF` de deux arguments a et N, qui renvoie la liste des puissances finales de a inférieures ou égales à N. Tester `LF(2,1000)` et `LF(7,1000)`.
3. Écrire une fonction `LF2` de deux arguments A et n, qui renvoie la liste des entiers inférieurs ou égaux à A dont n est une puissance finale. Tester `LF2(100,36)` et `LF2(1000,13)`.
4. Écrire une fonction `FF` d'un argument entier non nul a, qui renvoie le premier nombre n puissance finale de a en limitant le temps de recherche à 10 secondes (on pourra utiliser la fonction `time` du module `time`). Tester `FF(8)` et `FF(10)`.

2018.13 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

Dans cet exercice, les nombres entiers sont représentés par leur écriture en base 10 et leur écriture en base 2, chacune de ces écritures étant une chaîne de caractères. Par exemple, le nombre qui s'écrit `'18'` en base 10 s'écrit `'10010'` en base 2.

1. Écrire une fonction **`somchi`** d'argument une chaîne de caractères représentant un nombre entier n en base 10 et renvoyant la somme de ses chiffres décimaux.
2. Trouver tous les nombres entiers inférieurs à 10 000 égaux à la puissance quatrième de la somme de leurs chiffres en base 10.
3. Écrire une fonction de conversion **`DixVersDeux`** ayant pour argument une chaîne de caractères représentant l'écriture d'un nombre entier en base 10 et renvoyant son écriture en base 2.
4. Écrire la fonction de conversion réciproque **`DeuxVersDix`**.
5. Trouver tous les nombres entiers inférieurs à 10 000 égaux à la puissance quatrième de la somme des chiffres de leur écriture en base 2.
6. Modifier la fonction **`DixVersDeux`** en une fonction **`DixVersBase`** de telle manière que la chaîne de caractères renvoyée comme résultat soit écrite en base b. La base b de numération sera passée comme deuxième argument.

   Trouver tous les nombres entiers inférieurs à 10 000 égaux à la puissance quatrième de la somme des chiffres de leur écriture en base b pour toutes les bases de 3 à 9.

2018.14 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

---

1. On considère le code *Python* suivant :

```
def chiffres(n):
    if n == 0 :
        return [0]
    L = []
    while n != 0:
        L.append(n%10)
        n = n//10
    return L[::-1]
```

   Que fait la fonction `chiffres` ?

2. Soit n un entier naturel non nul possédant p chiffres (en base 10). Dans cet exercice, on dit que n est un « *nombre narcissique* » si la somme des puissances p-ième de ses chiffres vaut n.
   Montrer que `93 084` est un nombre narcissique.

3. Écrire une fonction `narcisse` d'argument n qui renvoie un booléen indiquant si n est un nombre narcissique, ou pas.

4. Afficher tous les nombres narcissiques compris entre `0` et `10 000`.

5. Écrire une fonction `narcis_suiv` d'arguments n et N qui renvoie le premier entier naturel narcissique plus grand que n, en limitant la recherche aux nombres inférieurs ou égaux à N.

6. Déterminer l'ensemble des nombres premiers et narcissiques compris entre 1 et 10000.

2018.15 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

---

1. Écrire une fonction **`partage`** de deux arguments, une liste L d'entiers et un entier a, renvoyant le couple $(L_{\text{inf}}, L_{\text{sup}})$, où L_{inf} (resp. L_{sup}) désigne la liste des éléments de L inférieurs ou égaux (resp. strictement supérieurs) à a.
2. À l'aide de la fonction précédente, écrire une fonction réalisant le tri rapide (« quicksort ») d'une liste.
3. Modifier les fonctions précédentes de manière à renvoyer, outre la liste triée, le nombre total de comparaisons effectuées au cours du tri.
4. Tester votre programme sur une liste composée de 40 000 entiers aléatoirement choisis entre -999 et 999 (on pourra utiliser par exemple la fonction **`randint`** de la bibliothèque **`random`**).

2018.16 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

---

1. Écrire une fonction `C1` d'argument un entiel naturel n qui renvoie la liste ordonnée de tous les carrés parfaits (d'entiers naturels) inférieurs ou égaux à n. Par exemple, `C1(10)` donne [0,1,4,9]. Tester `C1` pour $n = 100$.
2. Écrire une fonction `C2` d'argument un entiel naturel n qui renvoie la liste ordonnée de tous les entiers naturels inférieurs ou égaux à n qui s'écrivent comme somme de deux carrés d'entiers naturels. Tester `C2` pour $n = 100$.
3. Écrire une fonction `decomp2` d'argument un entiel naturel m qui renvoie la liste de tous les couples (p, q) d'entiers naturels tels que $m = p^2 + q^2$ et $p \leqslant q$.
4. Écrire une fonction `C3` d'argument un entiel naturel n qui renvoie la liste ordonnée de tous les entiers naturels inférieurs ou égaux à n qui s'écrivent comme somme de trois carrés d'entiers naturels. Tester `C3` pour $n = 100$.
5. Vérifier que les entiers inférieurs à 2017 qui ne peuvent pas s'écrire comme la somme de trois carrés d'entiers sont exactement ceux qui peuvent s'écrire $4^k\,(8\,q + 7)$.
6. Vérifier que tous les entiers inférieurs à 2017 peuvent s'écrire comme la somme de quatre carrés d'entiers.

2018.17 – **Exercice à dominante simulation numérique**

Cet exercice est prévu pour le langage Python (et ses bibliothèques numpy, scipy, matplotlib). À chaque question, les instructions ou les fonctions écrites devront être testées.

---

1. Définir trois points $A(28,5)$, $B(13,27)$, $C(2,2)$, puis faire tracer le triangle ABC.
2. Soit la suite récurrente de points $(T_n)_{n\in\mathbb{N}}$ définie par : $T_0 = D$ (fixé) ; T_{n+1} est le milieu de T_n et d'un point tiré aléatoirement parmi A, B et C (équiprobabilité).
   Écrire une fonction **`suivant`** d'argument un point T et qui renvoie le point S, milieu de T et d'un point tiré aléatoirement parmi A, B et C.
   On pourra utiliser la fonction **`rand`**, du module **`numpy.random`** de *Python*, qui tire un nombre au hasard dans l'intervalle $[0,1]$.
3. Pour $D(1,27)$, construire une matrice (11×2) contenant les abscisses et les ordonnées de T_n, pour n variant de 0 à 10. Tracer le nuage de points correspondant.
4. Tracer les 10000 premiers points de la suite (T_n).
5. Définir la fonction **`PT`** de cinq arguments A, B, C, D et N qui renvoie les N premiers points de la suite $(T_n)_{n\in\mathbb{N}}$, sous forme d'une matrice $(N\times2)$.
   Utiliser cette fonction pour tracer les 10000 premiers points de la suite pour des points A, B, C et D tirés au hasard dans le carré $[0,30]\times[0,30]$.

2018.18 – **Exercice à dominante simulation numérique**

Cet exercice est prévu pour le langage Python (et ses bibliothèques numpy, scipy, matplotlib). À chaque question, les instructions ou les fonctions écrites devront être testées.

On se place dans le plan muni d'un repère orthonormé. Chaque point du plan est caractérisé par une liste de deux réels du type `[x,y]`.
On rappelle que, si C est un point et r un réel, l'image M' du point M par l'homothétie de centre C et de rapport r est l'unique point M' tel que :

$$\overrightarrow{CM'} = r\,\overrightarrow{CM}\ .$$

1. Écrire une fonction `H` de trois arguments, un point C, une valeur réelle r et un point M qui renvoie l'image de M par l'homothétie de centre C et de rapport r.

On se donne m homothéties du plan de centres C_i et de rapports $r_i \in]0,1[$.
On appelle *jeu du chaos* la suite de points $(M_k)_{k\geqslant 0}$ construite de la manière suivante :
— le point M_0 est donné ;
— pour tout $k \geqslant 0$, on pose $M_{k+1} = \mathtt{H}(C_i, r_i, M_k)$ où i est tiré au sort à chaque itération avec une loi équiprobable. Avec *Python*, on pourra utiliser la fonction **`randint`** de la bibliothèque **`random`**.

2. Écrire une fonction **`tirer`** de quatre arguments, la liste des centres des homothéties, la liste des rapports des homothéties, le point M_0 et le nombre n d'itérations qui renvoie la liste des $n+1$ premiers points de la suite.
3. Tracer les 5001 premiers points de la suite définie par les trois homothéties de rapport 0.5 et de centres trois sommets d'un triangle équilatéral et de premier point l'un de ces sommets.
4. Définir une fonction `T` de trois arguments m, r, et n qui trace les $n+1$ premiers points de la suite obtenue pour un jeu d'homothéties de même rapport r et dont les centres sont les sommets d'un polygone régulier à m côtés et de premier point l'un de ces sommets.

   Tester `T(5,0.37,10000)`.

2018.19 – **Exercice à dominante simulation numérique**

Cet exercice est prévu pour le langage Python (et ses bibliothèques numpy, scipy, matplotlib). À chaque question, les instructions ou les fonctions écrites devront être testées.

---

1. Définir une fonction f d'argument x qui renvoie $1 + \lfloor 1/x \rfloor$, où $\lfloor t \rfloor$ désigne la *partie entière de t* (`floor`), si x est strictement positif, et zéro sinon.

À toute liste $B = [b_1, b_2, \cdots, b_\ell]$ de ℓ valeurs non nulles ($\ell \geqslant 2$), on associe :

$$S(B) = \frac{1}{b_1}\ \left(1 + \frac{1}{b_2}\ \left(1 + \cdots\ \left(1 + \frac{1}{b_\ell}\right) \cdots\right)\right) \quad .$$

Par exemple, $S([1,3,2]) = \dfrac{1}{1}\ \left(1 + \dfrac{1}{3}\ \left(1 + \dfrac{1}{2}\right)\right) = \dfrac{3}{2}$.

On pose également $S([b_1]) = \dfrac{1}{b_1}$.

2. En remarquant que $S(B)$ s'écrit $\dfrac{1}{b_1}\ (1 + S(B'))$, définir la fonction S.

   Tester cette fonction sur la liste `[1,3,2]`.

3. À partir d'un nombre x strictement positif, on définit la suite (a_n) par :
   — $a_0 = f(x)$.
   — $\forall n \in \mathbb{N}^\star, a_n = f(a_0\, a_1 \cdots a_{n-1}\ (x - S([a_0, a_1, \cdots, a_{n-1}])))$.

   Écrire la fonction `LA` de deux arguments x et n, qui renvoie les $n+1$ premiers termes de la suite (a_n) en partant de x.

4. Tester `LA(1,5)` , puis `S(LA(1,5))`.
   Faire de même pour `LA(5/7,4)` , puis `S(LA(5/7,4))`.
   Tester `LA(5/7,5)`. Commenter.

5. Écrire une fonction `Srec` de deux arguments x et d qui renvoie la première liste $[a_0, a_1, \cdots, a_\ell]$ trouvée telle que $x - S([a_0, a_1, \cdots, a_\ell]) \leqslant 10^{-d}$. Tester cette fonction pour $x = 1$ et $d = 7$.

2018.20 – **Exercice à dominante simulation numérique**

Cet exercice est prévu pour le langage Python (et ses bibliothèques numpy, scipy, matplotlib). À chaque question, les instructions ou les fonctions écrites devront être testées.

Une année est bissextile si elle est divisible par 4 mais pas par 100, ou si elle est divisible par 400.

1. *Question préliminaire* : soit $n = 4321$. Comment obtient-on le reste de la division euclidienne de n par 25 ? Comment obtient-on le quotient ?
2. Écrire une fonction **`Bissextile`** d'argument a et renvoyant un booléen, déterminant si une année a est bissextile ou pas. La tester avec les dates 1900, 1995, 1996, 2000.
3. Écrire une fonction **`NumeroJour`** de trois arguments, le jour **`j`**, le mois **`m`**, et l'année **`a`**, renvoyant le numéro du jour dans l'année **`a`**, compris entre 1 et 365 ou 366. Tester votre procédure avec le 5 juillet 2015 (réponse : 186).
4. Écrire une fonction **`NombreJours`** de trois arguments, le jour **`j`**, le mois **`m`**, et l'année **`a`**, renvoyant le nombre de jours écoulés depuis le premier janvier 1900. Tester votre procédure avec le 5 juillet 2015 (réponse : 42189).
5. En informatique, un problème similaire au bogue de l'an 2000 pourrait perturber le fonctionnement d'ordinateurs 32 bits. Le problème concerne des logiciels qui utilisent la représentation POSIX du temps, dans lequel le temps est représenté comme le nombre de secondes écoulées depuis le premier janvier 1970, 0:00:00). Sur les ordinateurs 32 bits, la plupart des systèmes d'exploitation concernés code ce nombre comme un nombre entier signé de 32 bits, ce qui limite le nombre de secondes à $2^{31} - 1$.

   Déterminer en quelle année pourrait se produire le bogue de la représentation POSIX sur un ordinateur 32 bits, s'il en existe encore.

2018.21 – **Exercice à dominante simulation numérique**

Cet exercice est prévu pour le langage Python (et ses bibliothèques numpy, scipy, matplotlib). À chaque question, les instructions ou les fonctions écrites devront être testées.

Soit Ω_V le sous-ensemble $\{(1,0),(-1,0),(0,1),(0,-1)\}$ de l'espace vectoriel $\mathbb{R}^2$.

Soit (V_n) une suite de variables aléatoires à valeurs indépendantes suivant la même loi uniforme sur Ω_V.

On définit une suite de variable aléatoires (X_n) en posant :

$$X_0 = (0,0), \quad \text{et} \quad \forall n \in \mathbb{N}, \quad X_{n+1} = X_n + V_n \quad .$$

On appelle « *trajectoire de longueur n* » la ligne brisée aléatoire reliant les points de coordonnées $X_0, X_1, \cdots, X_n$.

1. Écrire une fonction `tirage` sans argument et renvoyant une des listes suivantes avec équiprobabilité : `[1,0]`, `[-1,0]`, `[0,1]` et `[0,-1]`.

   On pourra utiliser la fonction `rand` du module `numpy.random` de *Python*.
2. En déduire une fonction `trajectoire` d'argument n renvoyant le tirage d'une trajectoire de longueur n, sous forme d'une liste de couples `[x,y]`.
3. Tracer quelques trajectoires de longueurs 10, 100, 1000, puis 10000.
4. Écrire une fonction `premierRetour` d'argument m renvoyant le plus petit entier non nul $n \leqslant m$ tel que $X_n = (0,0)$ s'il existe, et -1 sinon.
5. On note T le premier retour en $(0,0)$. Au moyen d'un programme, conjecturer si la variable aléatoire T est bien définie.

2018.22 – **Exercice à dominante simulation numérique**

Cet exercice est prévu pour le langage Python (et ses bibliothèques numpy, scipy, matplotlib). À chaque question, les instructions ou les fonctions écrites devront être testées.

---

1. Que fait la fonction **`L2str`** suivante ? L'appliquer à la liste `[1,2,1,1]`.

```
def L2str(L) :
    ch = ""
    for e in L :
        ch = ch + str(e)
    return ch
```

Soit une suite d'entiers naturels $(u_n)_{n\in\mathbb{N}}$ définie par : $u_0 = 1$ et $\forall n \in \mathbb{N}\,,\ u_{n+1} = \phi(u_n)$, où la fonction ϕ est définie comme suit : $\phi(k)$ est le nombre composé des chiffres décrivant le nombre k. Par exemple :

1112	$\mapsto$	3112	(trois uns, un deux)
29	$\mapsto$	1219	(un deux, un neuf)
333	$\mapsto$	33	(trois trois)
1211	$\mapsto$	111221	(un un, un deux, deux uns)

Dans cet exercice, chaque entier naturel sera codé sous forme d'une liste de chiffres.

2. Écrire une fonction **`lire`** d'argument une liste L de chiffres codant un nombre k et renvoyant la liste des chiffres de $\phi(k)$. Par exemple, `lire([1,2,1,1])` doit donner `[1,1,1,2,2,1]`.
3. Afficher les quinze premiers termes de la suite u_n.
4. Quels sont les chiffres présents dans u_n ? Expliquer succinctement pourquoi.
5. Pour tout entier naturel n, on note ℓ_n le nombre de chiffres de u_n. On peut montrer l'existence de k et λ dans $\mathbb{R}_+^*$ tels que $\ell_n \underset{n\to+\infty}{\sim} k\,\lambda^n$.

   Vérifier numériquement ce résultat en donnant des valeurs grossièrement estimées de k et de λ.
6. Étudier la proportion des différents chiffres dans u_n.

2018.23 – **Exercice à dominante simulation numérique**

Cet exercice est prévu pour le langage Python (et ses bibliothèques numpy, scipy, matplotlib). À chaque question, les instructions ou les fonctions écrites devront être testées.

On cherche à étudier numériquement les solutions de l'équation différentielle avec condition initiale suivante :

$$y'(t) = t^2 - y(t)^3 \quad \text{avec} \quad y(-1.5) = a\ . \tag{1}$$

1. Pour $a = 2$ et un pas de discrétisation $h = \frac{1}{4}$, puis $h = \frac{1}{8}$, représenter les solutions approchées du problème (1) obtenues par la méthode d'Euler sur l'intervalle $[-1.5, 2.5]$.
2. En utilisant la fonction **`odeint`** du module **`scipy.integrate`** de *Python*, résoudre numériquement le problème (1) pour $a = 2$. On prendra garde à définir soigneusement les arguments de cette fonction en lisant attentivement l'aide en ligne.
3. Sur la figure existante, rajouter la courbe de la solution numérique obtenue à la question précédente.
4. Rajouter ensuite les courbes des solutions numériques pour $a = 1.3$ et $a = 0.1$. Qu'observe-t-on ?
5. Pour résoudre $y'(t) = f(t, y(t))$ avec $y(t_0) = a$, on utilise maintenant la suite (y_n) définie par $y_0 = a$ et $y_{k+1} = y_k + \dfrac{h}{2}\ (\ f(t_k, y_k) + f(\ t_{k+1}\ ,\ y_k + h\, f(t_k, y_k)\ )\ )$.

   Représenter, pour $a = 2$ et les mêmes pas de discrétisation qu'à la question 1, les solutions approchées obtenues par cette méthode. Expliquer pourquoi le résultat semble meilleur.

2018.24 – **Exercice à dominante simulation numérique**

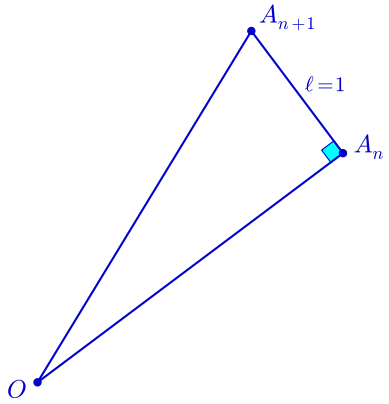
Cet exercice est prévu pour le langage Python (et ses bibliothèques numpy, scipy, matplotlib). À chaque question, les instructions ou les fonctions écrites devront être testées.

Dans le plan affine, muni d'un repère orthonormé direct d'origine O, on considère la suite de points $(A_n)_{n\in\mathbb{N}}$ tels que :

- A_0 a pour coordonnées $(1, 0)$.
- pour tout entier naturel non nul n, le triangle OA_nA_{n+1} est rectangle en A_n, la distance A_nA_{n+1} vaut 1 et l'angle $\left(\overrightarrow{OA_n}, \overrightarrow{OA_{n+1}}\right)$ est direct.

Si x_n et y_n sont les coordonnées de A_n, on a :

$$\begin{cases} x_{n+1} &= x_n - \dfrac{y_n}{\sqrt{x_n^2+y_n^2}} \\ y_{n+1} &= y_n + \dfrac{x_n}{\sqrt{x_n^2+y_n^2}} \end{cases}.$$



1. Écrire une fonction A d'argument N renvoyant la liste des coordonnées des points A_n pour $0 \leqslant n \leqslant N$.
2. Écrire une fonction **`afficher`** d'argument N affichant la figure représentant les $(N+1)$ premiers points A_n. La tester pour $N = 60$.

Les points A_n tournent autour de l'origine $\mathcal{O}$ du repère. On note $T(k)$ la valeur de n telle que le point A_n commence le k-ième tour.

3. Écrire une fonction **`tours`** d'argument m qui renvoie la liste des $T(k)$ pour k variant de 1 à m. Afficher **`tours(5)`**.
4. Faire tracer chacun des cinq premiers tours avec une couleur différente.
5. Déterminer les abscisses des dix premiers points d'intersection de la ligne reliant les A_n avec la partie positive de l'axe des abscisses. Vérifier qu'à chaque tour, on s'est éloigné du centre d'une distance environ égale à π.

2018.25 – **Exercice à dominante simulation numérique**

Cet exercice est prévu pour le langage Python (et ses bibliothèques numpy, scipy, matplotlib). À chaque question, les instructions ou les fonctions écrites devront être testées.

Une méthode pour estimer numériquement l'aire du disque de centre O et de rayon 1 est la suivante : on tire au hasard N points de coordonnées $(x, y) \in [-1, 1]\times[-1, 1]$; parmi ces points, on compte le nombre i de ceux qui appartiennent au disque ; On admet que i/N est une approximation du rapport de l'aire du disque par l'aire du carré.

1. Observer et expliquer ce que fait le code suivant :

```
from numpy.random import rand
Lpts = 2*rand(8,2) - 1
print(Lpts)
```

2. Écrire une fonction **estim1** d'argument N qui renvoie une valeur approchée de l'aire du disque, calculée selon le procédé indiqué ci-dessus.
3. Écrire une fonction **estim2** analogue à **estim1** qui fait tracer en plus les points tirés dans le carré ; ceux dans le disque avec une couleur, et ceux à l'extérieur avec une autre couleur.

   Pour obtenir un repère orthonormé, on pourra utiliser l'instruction **axis("equal")** grâce au module **matplotlib.pyplot** de *Python*.

On s'intéresse maintenant à l'aire du domaine $\mathcal{D}$ d'équation : $(x^2 + y^2)^2 \leqslant x^3$.

4. Faire tracer l'allure de la frontière du domaine $\mathcal{D}$ après avoir résolu « *à la main* » en y dans $\mathbb{R}$ l'équation $(x^2 + y^2)^2 = x^3$.
5. En déduire que le domaine $\mathcal{D}$ est contenu dans un rectangle que l'on déterminera.
6. À l'aide de la méthode précédente, estimez l'aire du domaine $\mathcal{D}$ et faire afficher avec des couleurs différentes les points tirés, selon qu'ils seront dans $\mathcal{D}$ ou pas.