Frédéric Butin

Classes prépas scientifiques

1re et 2e années

ITC

Informatique tronc commun

Cours et exercices corrigés



ellipses

Références sciences

ITC

Informatique tronc commun

Cours et exercices corrigés

Frédéric Butin

Classes prépas scientifiques

1re et 2e années

ellipses

Collection Références sciences

dirigée par Paul de Laboulaye
paul.delaboulaye@editions-ellipses.fr

Retrouvez tous les livres de la collection et des extraits sur www.editions-ellipses.fr

ISBN 9782340-073937
©Ellipses Édition Marketing S.A., 2022
8/10 rue la Quintinie 75015 Paris

DANGER
LE PHOTOCOPILLAGE TUE LE LIVRE

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5.2° et 3°a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective », et d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit constituerait une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

www.editions-ellipses.fr

Frédéric BUTIN

Informatique Tronc Commun
Cours et exercices corrigés

1re et 2e années
de classes préparatoires scientifiques

Avant-propos

L'ouvrage couvre la totalité du programme d'informatique "tronc commun" des classes préparatoires aux grandes écoles, tout en s'ouvrant sur des horizons plus larges. Son but est de permettre aux étudiants de gérer entièrement un problème informatique depuis son analyse jusqu'à sa résolution par l'écriture d'un programme, son étude et sa vérification.

Sont abordées les idées importantes de l'algorithmique qui s'appuie sur des méthodes de programmation très efficaces, comme la récursivité, le principe "diviser pour régner", l'heuristique gloutonne ou encore la programmation dynamique utile dans les problèmes d'optimisation. Les algorithmes classiques sont présentés, analysés et implémentés en langage Python, et les principales structures de données sont définies.

Les problèmes d'optimisation trouvent aussi des solutions dans la théorie des graphes, introduite ici avec les parcours usuels, en largeur et en profondeur, et les méthodes de recherche de plus courts chemins.

La gestion d'informations toujours plus massives nécessite par ailleurs l'utilisation accrue de bases de données : elles interviennent en particulier dans n'importe quel site internet. Dans la présentation qui en est faite ici, l'accent est mis sur l'écriture de requêtes en langage SQL.

Enfin, l'intelligence artificielle apparaît dans des domaines de plus en plus variés, où elle donne naissance à des outils qui simplifient certaines tâches. Le dernier chapitre du livre en propose une initiation et présente aussi des éléments de la théorie des jeux.

Les chapitres 1 à 5 correspondent dans une large mesure au programme de première année, mais comportent aussi des thèmes et exercices davantage adressés à des étudiants de deuxième année ou ayant déjà un certain recul sur le sujet. Les chapitres 6 à 8 portent quant à eux sur le programme de deuxième année. Ce nombre restreint de chapitres confère à l'ouvrage une unité qui permet d'aller droit au but de façon synthétique.

Notons que le premier chapitre donne les outils nécessaires pour utiliser Python dans d'autres domaines scientifiques, comme la Physique, la Chimie et les Sciences Industrielles. Il peut aussi constituer une initiation pour les lecteurs moins familiarisés avec ce langage.

Les exemples d'application, les exercices et les problèmes proposés — tous entièrement résolus — sont variés et multiples avec, en trame de fond, le souci constant du concret. L'ouvrage se veut progressif, dans le cours comme dans les exercices dont le degré de difficulté est signalé (0, 1 ou 2 astérisques). Les paragraphes comportant un astérisque contiennent, eux, des notions plus délicates ou constituent des compléments de cours.

Enfin, dans un but de simplification de l'écriture des scripts, précisons que lorsqu'ils sont nécessaires, les modules `numpy` et `matplotlib.pyplot` sont supposés être déjà importés par les deux lignes suivantes.

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

Nous exprimons ici notre reconnaissance aux amis et collègues pour la relecture de l'ouvrage et pour leurs suggestions bienveillantes. Et par avance, nous remercions les lecteurs qui voudront bien nous faire part de leurs remarques.

Sommaire

1 Tutoriel d'initiation à Python **1**
1.1 Installation et premier exemple ... 1
1.1.1 Installation de Python ... 1
1.1.2 Premier exemple ... 2
1.1.3 Modules ... 2
1.1.4 Utilisation de fichiers ... 3
1.2 Types d'objets ... 3
1.2.1 Entiers et flottants ... 3
1.2.2 Variables ... 3
1.2.3 Nombres complexes ... 4
1.2.4 Booléens ... 4
1.2.5 Chaînes de caractères ... 4
1.2.6 Listes ... 5
1.2.7 Polynômes ... 6
1.3 Graphes de fonctions et courbes paramétrées ... 7
1.3.1 Le module matplotlib ... 7
1.3.2 Graphes de fonctions ... 7
1.3.3 Courbes paramétrées ... 8
1.4 Algèbre linéaire ... 8
1.4.1 Le module numpy ... 8
1.4.2 Tableaux ... 9
1.4.3 Tableaux contenant des coefficients aléatoires ... 10
1.4.4 Opérations sur les tableaux ... 10
1.5 Graphiques multiples, courbes implicites, 3D et animations ... 12
1.5.1 Superposition de graphes ... 12
1.5.2 Courbes implicites ... 12
1.5.3 Fenêtres contenant plusieurs graphiques ... 13
1.5.4 Représentations graphiques dans l'espace ... 14
1.5.5 Animations ... 15
1.6 Résolution numérique d'équations ... 15
1.6.1 Systèmes linéaires ... 15
1.6.2 Le module scipy ... 16
1.6.3 Dérivation d'une fonction ... 16
1.6.4 Equations non linéaires ... 16
1.6.5 Equations différentielles ... 16

1.7 Intégration numérique ... 17
1.7.1 Méthode standard ... 17
1.7.2 Méthode des trapèzes ... 18
1.8 Simulation des lois de probabilité ... 18
1.9 Calcul formel ... 19
1.9.1 Le module sympy ... 19
1.9.2 Exemples ... 19

2 Algorithmes classiques 21
2.1 Algorithmes pour les nombres et les polynômes ... 21
2.1.1 Calcul de puissances ... 21
2.1.1.1 Algorithme d'exponentiation naïve ... 22
2.1.1.2 Algorithme d'exponentiation rapide ... 22
2.1.2 Algorithme d'Euclide ... 23
2.1.2.1 Principe et implémentation ... 23
2.1.2.2 Majorant de la complexité de l'algorithme d'Euclide ... 24
2.1.2.3 Algorithme d'Euclide étendu ... 27
2.1.3 Cryptographie ... 28
2.1.3.1 Le système RSA ... 28
2.1.3.2 Construction des clés ... 28
2.1.3.3 Cryptage du message ... 29
2.1.3.4 Décryptage du message ... 30
2.1.4 Evaluation d'un polynôme en un point ... 32
2.1.4.1 Algorithme naïf ... 32
2.1.4.2 Algorithme de Horner ... 32
2.2 Algorithmes de recherche dans un tableau trié ... 33
2.2.1 Recherche séquentielle ... 33
2.2.2 Recherche dichotomique ... 34
2.3 Algorithmes de tri ... 35
2.3.1 Tri par sélection ... 35
2.3.2 Tri par insertion ... 36
2.3.3 Tri par fusion ... 38
2.3.4 Tri rapide (quick sort) ... 42
2.3.5 Optimalité de certains algorithmes de tri ... 45
2.3.6 Tri utilisé par défaut en Python ... 46
2.4 Transformée de Fourier discrète ... 46
2.4.1 Définition de la transformée de Fourier discrète ... 46
2.4.2 Transformée de Fourier rapide (FFT) ... 47
2.4.2.1 Principe de calcul ... 48
2.4.2.2 Algorithme de Cooley et Tukey ... 48
2.4.2.3 Complexité de l'algorithme de Cooley et Tukey ... 49
2.4.3 Transformée de Fourier discrète inverse ... 49
2.4.4 Transformée de Fourier discrète en dimension 2 ... 50
2.4.5 Deux applications de la transformée de Fourier discrète ... 50
2.4.5.1 Valeurs approchées des coefficients de Fourier ... 50

2.4.5.2 Produit rapide de deux polynoˆmes 52
2.5 Traitement d'images 53
2.5.1 Affichage d'une image en Python 53
2.5.2 Synthèse des couleurs 55
2.5.3 Filtres 55
2.5.4 Filtres non linéaires 55
2.5.5 Filtres linéaires 57
2.5.6 Utilisation de la FFT en traitement d'images 60
2.6 EXERCICES ET CORRIGÉS 61
2.6.1 Nombres et polynômes 61
2.6.2 Algorithmes itératifs 67
2.6.3 Algèbre linéaire 81
2.6.4 Récursivité 86
2.6.5 Tris 101
2.6.5.1 Tris classiques 101
2.6.5.2 Tris "exotiques" 107
2.6.6 Transformée de Fourier 110
2.6.7 Traitement d'images 116
2.7 PROBLÈMES 124

3 Algorithmes : méthodes et structures 137
3.1 Algorithmique 137
3.1.1 Définition et structure d'un algorithme 137
3.1.2 Procédures en Python 138
3.1.3 Variables locales et variables globales 139
3.1.4 Tests 140
3.1.5 Boucles 140
3.1.6 Spécifications d'un programme 141
3.1.6.1 Entrée, sortie, signature et docstring 141
3.1.6.2 Préconditions, postconditions et assertions 141
3.1.6.3 Application à un exemple 142
3.1.7 Effets de bord 142
3.1.8 Problèmes d'optimisation et problèmes de recherche 143
3.1.9 Jeu de tests associé à un programme 143
3.2 Récursivité 144
3.2.1 Introduction à la récursivité 144
3.2.2 Exemple prototypique de fonction récursive : la factorielle 144
3.2.3 La méthode "diviser pour régner" 145
3.3 Etude d'un algorithme 145
3.3.1 Terminaison d'un algorithme 146
3.3.2 Correction d'un algorithme 146
3.3.3 Complexité d'un algorithme 146
3.3.3.1 Différentes classes de complexité 146
3.3.3.2 Exemple de calcul de complexité : cas d'école 147
3.3.4 Suites récurrentes utilisées en algorithmique 148
3.4 Tableaux et listes 150

3.4.1 Tableaux 151
3.4.2 Listes chaînées 151
3.4.3 Listes doublement chaînées 153
3.4.4 Tableaux dynamiques 154
3.5 Tables de hachage 156
3.5.1 Introduction 156
3.5.2 Tables à adressage direct 156
3.5.3 Définition d'une table de hachage 157
3.5.4 Résolution des collisions par chaînage* 158
3.5.5 Construction d'une fonction de hachage 160
3.5.6 Résolution des collisions par adressage ouvert* 161
3.5.7 Objets hachables et objets mutables 164
3.5.7.1 Définitions et conséquences 164
3.5.7.2 Egalité et identité 165
3.5.7.3 Copie d'un élément 166
3.6 Ensembles et dictionnaires 167
3.6.1 Ensembles 167
3.6.2 Dictionnaires 168
3.7 Utilisation de fichiers 168
3.7.1 Utilisation d'un fichier texte 168
3.7.1.1 Ecriture, lecture et ajout de données 169
3.7.1.2 Exemples 169
3.7.2 Utilisation d'un fichier csv 170
3.7.2.1 Définition et opérations de base 170
3.7.2.2 Exemple 170
3.8 Représentation des nombres dans un ordinateur 171
3.8.1 Représentation des entiers dans un ordinateur 171
3.8.1.1 Langage machine et langages de haut niveau 171
3.8.1.2 Représentation des entiers naturels 172
3.8.1.3 Représentation des entiers relatifs 172
3.8.1.4 Entiers multi-précision 173
3.8.2 Représentation des réels dans un ordinateur 174
3.8.2.1 Définitions 174
3.8.2.2 Exemple 176
3.8.2.3 Changement de base pour les nombres flottants 177
3.8.2.4 Approximation d'un réel 177
3.8.2.5 Flottants en simple précision 178
3.8.2.6 Flottants en double précision 179
3.8.2.7 Limites de précision dans le calcul avec les flottants 179
3.9 EXERCICES ET CORRIGÉS 182
3.9.1 Hachage 182
3.9.2 Représentation des nombres 183
3.9.2.1 Nombres entiers 183
3.9.2.2 Nombres flottants 186

4 Graphes **195**
4.1 Graphes ... 195
4.1.1 Définitions ... 195
4.1.1.1 Graphes non orientés ... 195
4.1.1.2 Graphes orientés ... 196
4.1.2 Représentations d'un graphe : matrice et liste d'adjacence ... 197
4.1.3 Sous-graphes ... 198
4.1.4 Graphes valués ... 199
4.1.5 Connexité et arêtes séparatrices ... 199
4.1.5.1 Composantes connexes et fortement connexes ... 199
4.1.5.2 Arêtes séparatrices ... 200
4.1.6 Arbres ... 201
4.1.6.1 Définitions et propriétés ... 201
4.1.6.2 Arbres couvrants ... 203
4.1.6.3 Représentation des arbres ... 204
4.2 Parcours dans les graphes ... 205
4.2.1 Piles et files ... 206
4.2.1.1 Piles ... 206
4.2.1.2 Files ... 207
4.2.2 Parcours en largeur ... 208
4.2.2.1 Définitions et algorithme ... 209
4.2.2.2 Propriétés du parcours en largeur ... 211
4.2.2.3 Applications du parcours en largeur ... 212
4.2.3 Parcours en profondeur ... 214
4.2.3.1 Définitions et algorithme ... 214
4.2.3.2 Propriétés du parcours en profondeur ... 218
4.2.3.3 Applications du parcours en profondeur ... 219
4.3 Tas et files de priorité* ... 221
4.3.1 Structure de tas (binaire) ... 221
4.3.2 Tas max et opérations ... 223
4.3.2.1 Première opération — Modifier un tas max ... 223
4.3.2.2 Deuxième opération — Ordonner un tableau en un tas max ... 224
4.3.2.3 Tas max pour les files de priorité ... 225
4.3.3 Tri par tas ... 225
4.3.4 Tas min et opérations ... 226
4.3.5 Files de priorité max ... 227
4.3.5.1 Première opération — Rechercher un élément de clé maximale ... 228
4.3.5.2 Deuxième opération — Extraire un élément de clé maximale ... 228
4.3.5.3 Troisième opération — Augmenter la clé d'un élément ... 228
4.3.5.4 Quatrième opération — Insérer un élément ... 229
4.3.6 Files de priorité min ... 230

4.4 Plus courts chemins 231
4.4.1 Définitions et propriétés 231
4.4.2 Algorithme de plus courts chemins par tri topologique* 234
4.4.3 Algorithme de Bellman-Ford* 235
4.4.4 Algorithme de Dijkstra 237
4.4.4.1 Implémentation naïve de l'algorithme 238
4.4.4.2 Implémentation utilisant une file de priorité* 240
4.4.5 Algorithme A étoile 241
4.5 EXERCICES ET CORRIGÉS 243
4.5.1 Graphes 243
4.5.2 Arbres 250
4.5.3 Piles et files 256
4.5.4 Parcours dans un graphe 258
4.5.5 Tas 271
4.5.6 Plus courts chemins dans un graphe 273
4.6 PROBLÈ MES 277

5 Algorithmes gloutons 287
5.1 Heuristique gloutonne 287
5.1.1 Hypergraphes et algorithme glouton 287
5.1.2 Méthode 288
5.2 Matroïdes* 289
5.3 Ordonnancement de tâches avec pénalités 293
5.4 Algorithmes de Kruskal et de Prim* 295
5.4.1 Matroïdes graphiques 295
5.4.2 Algorithme de Kruskal 296
5.4.3 Algorithme de Prim 298
5.5 Codage de Huffman* 301
5.5.1 Codage préfixe 301
5.5.2 Propriétés du codage de Huffman 303
5.5.3 Implémentation du codage de Huffman 305
5.5.4 Implémentation du décodage 306
5.6 Problème du sac à dos : version fractionnaire 307
5.6.1 Présentation du problème 307
5.6.2 Résolution 307
5.7 EXERCICES ET CORRIGÉS 308
5.7.1 Matroïdes et algorithme glouton 308
5.7.2 Ordonnancement de tâches avec pénalités 311
5.7.3 Algorithmes de Kruskal et de Prim 312
5.7.4 Rendu de monnaie 315
5.7.5 Codage de Huffman 317
5.7.6 Problème du sac à dos : version fractionnaire 321

6 Bases de données 323
6.1 Introduction 323
6.1.1 Présentation des systèmes de gestion de bases de données 323

6.1.2 Architecture trois tiers 323
6.1.3 Quatre étapes 324
6.2 Modèle conceptuel de données 324
6.2.1 Présentation du modèle 324
6.2.2 Exemple de référence 325
6.2.3 Entités et attributs 326
6.2.4 Types, occurrences et identifiants 326
6.2.5 Associations 327
6.2.5.1 Associations 1 – 1 328
6.2.5.2 Associations 1 – * 329
6.2.5.3 Associations 329
6.2.6 Entités faibles 329
6.2.7 Contraintes d'intégrité 330
6.2.8 Méthode de création d'un modèle conceptuel de données 330
6.3 Modèle logique de données 331
6.3.1 Relations 331
6.3.2 Clés primaires et clés étrangères 331
6.3.3 Contraintes référentielles et graphe des clés étrangères 333
6.3.4 Passage d'un modèle conceptuel à un modèle logique 333
6.4 Implémentation et peuplement 336
6.4.1 Création de la base de données 337
6.4.2 Peuplement de la base de données 338
6.5 Interrogation d'une base de données 339
6.5.1 Projection 339
6.5.2 Sélection 340
6.5.3 Produit cartésien 341
6.5.4 Réunion 342
6.5.5 Intersection 342
6.5.6 Différence 343
6.5.7 Renommage 343
6.5.8 Jointure 343
6.5.9 Auto-jointure 344
6.5.10 Schémas standards pour les opérateurs 345
6.5.11 Division cartésienne 345
6.5.12 Fonctions d'agrégation 346
6.5.13 SQL et Python 347
6.6 EXERCICES ET CORRIGÉS 348
6.7 PROBLÈMES 358

7 Programmation dynamique 373
7.1 Principes de la programmation dynamique 373
7.1.1 Définitions 373
7.1.2 Méthode 374
7.2 Découpage d'une barre métallique 374
7.2.1 Etape 1. Structure d'une solution optimale 375
7.2.2 Etape 2. Solution récursive 375

7.2.3 Etape 3a. Calcul récursif du prix maximal : résolution du problème d'optimisation....376
7.2.4 Etape 3b. Calcul itératif du prix maximal : résolution du problème d'optimisation....377
7.2.5 Etape 4. Construction d'une solution optimale associée au prix maximal : résolution du problème de recherche....377
7.3 Algorithme de Levenshtein....378
7.3.1 Etape 1. Structure d'une solution optimale....379
7.3.2 Etape 2. Solution récursive....379
7.3.3 Etape 3b. Calcul itératif de la distance minimale : résolution du problème d'optimisation....380
7.3.4 Etape 4. Construction d'une solution optimale associée à la distance minimale : résolution du problème de recherche....381
7.4 Problème du sac à dos : version entière....381
7.4.1 Présentation du problème et résolution naïve....381
7.4.2 Etape 1. Structure d'une solution optimale....382
7.4.3 Etapes 2, 3 et 4. Résolution des problèmes d'optimisation et de recherche....382
7.5 Multiplications matricielles enchaînées....383
7.5.1 Etape 1. Structure d'une solution optimale....384
7.5.2 Etape 2. Solution récursive....384
7.5.3 Etape 3a. Calcul récursif du coût minimal : résolution du problème d'optimisation....385
7.5.4 Etape 3b. Calcul itératif du coût minimal : résolution du problème d'optimisation....386
7.5.5 Etape 4. Construction d'une solution optimale associée au coût minimal : résolution du problème de recherche....386
7.6 Plus longue sous-suite commune....387
7.6.1 Etape 1. Structure d'une solution optimale....387
7.6.2 Etape 2. Solution récursive....389
7.6.3 Etape 3b. Calcul itératif de la longueur maximale : résolution du problème d'optimisation....389
7.6.4 Etape 4. Construction d'une solution optimale associée à la longueur maximale : résolution du problème de recherche....390
7.7 Algorithme de Floyd-Warshall....390
7.7.1 Etapes 1 et 2. Structure d'une solution optimale....390
7.7.2 Etape 3b. Calcul itératif de la valeur maximale : résolution du problème d'optimisation....391
7.7.3 Etape 4. Construction d'une solution optimale associée à la valeur maximale : résolution du problème de recherche....392
7.8 Comparaison de trois méthodes....394
7.8.1 Comparaison avec la méthode "diviser pour régner"....394
7.8.2 Comparaison avec la méthode gloutonne....395
7.8.3 Tableau récapitulatif....395
7.9 EXERCICES ET CORRIGÉS....396

7.9.1 Algorithme de Levenshtein ... 397
7.9.2 Problème du sac à dos : version entière ... 399
7.9.3 Multiplications matricielles enchaînées ... 400
7.9.4 Plus longue sous-suite commune ... 402
7.9.5 Algorithme de Floyd-Warshall ... 404

8 Intelligence artificielle et étude des jeux 407
8.1 Principes de l'intelligence artificielle ... 407
8.1.1 Définitions et exemples ... 407
8.1.2 Deux principaux types d'apprentissage statistique ... 408
8.1.3 Utilisation de Python ... 409
8.2 Apprentissage supervisé ... 410
8.2.1 Introduction ... 410
8.2.2 Utilisation de Python ... 410
8.2.2.1 Exemple de classification ... 410
8.2.2.2 Exemple de régression ... 412
8.2.3 Evaluation d'un modèle ... 412
8.2.3.1 Fonction de perte et risque empirique ... 412
8.2.3.2 Cas d'un modèle de classification ... 412
8.2.3.3 Cas d'un modèle de régression ... 414
8.2.4 Algorithme des k plus proches voisins ... 414
8.2.4.1 Introduction ... 414
8.2.4.2 Classification ... 415
8.2.4.3 Régression ... 417
8.2.5 Classification naïve bayésienne* ... 419
8.2.5.1 Introduction ... 419
8.2.5.2 Principe et algorithme ... 419
8.2.6 Régressions linéaires* ... 421
8.2.6.1 Cas d'un problème de régression ... 421
8.2.6.2 Fonctions logit et logistique ... 423
8.2.6.3 Cas d'un problème de classification binaire ... 424
8.2.7 Initiation aux réseaux de neurones artificiels* ... 428
8.2.7.1 Introduction ... 428
8.2.7.2 Perceptron ... 428
8.2.7.3 Perceptron multiclasse ... 430
8.2.7.4 Perceptron multicouche ... 431
8.2.7.5 Apprentissage d'un réseau de neurones ... 433
8.3 Apprentissage non supervisé ... 437
8.3.1 Algorithme des k moyennes ... 437
8.3.1.1 Introduction ... 437
8.3.1.2 L'algorithme des k moyennes ... 438
8.3.1.3 Implémentation en Python ... 439
8.4 Etude des jeux ... 443
8.4.1 Jeux à deux joueurs ... 443
8.4.1.1 Introduction et définitions ... 443

8.4.1.2 Exemple du jeu de Nim 444
8.4.1.3 Exemple du jeu de morpion 446
8.4.1.4 Arbre d'un jeu 446
8.4.2 Jeux d'accessibilité 447
8.4.3 Construction d'une stratégie gagnante 449
8.4.4 Algorithme du minimax 450
8.4.4.1 Principe et algorithme 450
8.4.4.2 Algorithme du minimax avec une heuristique 452
8.5 EXERCICES ET CORRIGÉS 453
8.5.1 Intelligence artificielle — Apprentissage supervisé 453
8.5.2 Intelligence artificielle — Apprentissage non supervisé 462
8.5.3 Etude des jeux — Graphes et stratégies 466
8.5.4 Etude des jeux — Algorithme du minimax 470
8.6 PROBLÈMES 476

A Complexité d'algorithmes **485**

Bibliographie **487**

Index **489**

Chapitre 1

Tutoriel d'initiation à Python

Le langage Python[1] est un langage interprété[2], créé en 1989 par Guido van Rossum, et utilisé dans de nombreux domaines. Sa syntaxe simple en fait un langage facile à lire et à comprendre. Il a en outre un intérêt tout particulier dans le domaine scientifique, grâce aux nombreux modules qui permettent d'utiliser et de développer des méthodes numériques, mais aussi de faire du calcul formel.

Ce chapitre présente une initiation à Python sous la forme d'un tutoriel qui, au fil de l'apprentissage, deviendra un aide-mémoire facilement consultable. De même qu'une première étape dans l'apprentissage d'une langue est la répétition de ce qui est entendu, de même ici la meilleure façon d'étudier ce tutoriel est de saisir au clavier tous les exemples donnés pour s'approprier ainsi les différentes commandes de Python.

1.1 Installation et premier exemple

1.1.1 Installation de Python

Pour installer Python, on peut télécharger le fichier `Winpython64-3.9.10.0.exe` en cliquant sur le lien ci-dessous et l'exécuter.

`https://sourceforge.net/projects/winpython/files/WinPython_3.9/3.9.10.0/Winpython64-3.9.10.0.exe/download`

Ce fichier est aussi accessible sur le site `https://winpython.github.io/`.

1. On trouve dans le langage Python de nombreuses références à la série des Monty Python, chère à l'inventeur de ce langage !

2. Comme son nom l'indique, un tel langage est interprété par un logiciel appelé **interpréteur** qui exécute ligne à ligne le code avec les données d'entrée et calcule les données de sortie. A l'inverse, dans un langage compilé, le code source est d'abord transformé en un code binaire par un logiciel appellé **compilateur** ; ensuite, le système d'exploitation calcule les données de sortie grâce à ce code et aux données d'entrée. Un programme en langage compilé s'exécute en général plus rapidement qu'un programme en langage interprété. Cependant, un langage interprété présente aussi l'intérêt de fonctionner directement sur tout ordinateur dès qu'il possède l'interpréteur adéquat.

Cette distribution de Python contient tous les modules nécessaires au contenu du livre, en particulier `numpy` (pour les tableaux et l'algèbre linéaire), `scipy` (pour les méthodes numériques), `matplotlib` (pour les tracés graphiques), `sqlite3` (pour les bases de données), `sklearn` (pour l'intelligence artificielle), `pandas` (pour le traitement des données), `sympy` (pour le calcul formel), `cx_Freeze` (pour convertir un fichier Python en exécutable). Elle contient également les éditeurs Idle, Spyder, Pyzo et le notebook Jupyter[3].

1.1.2 Premier exemple

Quand on ouvre le terminal Idle, le symbole >>> apparaît à l'écran. On peut alors saisir une instruction que l'on exécute avec la touche "Entrée".
Calculer $4 \times 13 - 1$.
Une nouvelle ligne commençant par >>> apparaît alors. Calculer $\frac{4}{3}$.
On remarque que l'on obtient une valeur approchée en utilisant le symbole `/`. Si l'on souhaite avoir la division euclidienne de 4 par 3, on devra écrire `4//3`. Quant au reste, il est obtenu en écrivant `4%3`. Notons aussi que la commande `divmod(a,b)` renvoie le couple formé du quotient et du reste de la division euclidienne de a par b.
Pour accéder aux instructions précédemment écrites, on utilise simultanément les touches "alt" et "P".

La commande `print` permet d'afficher un résultat et la commande `input` de récupérer des données saisies au clavier.

```
prenom = input('Quel est votre prénom ? ')
```

On affiche alors le résultat.

```
print('Bonjour '+str(prenom))
```

1.1.3 Modules

On peut importer des **modules** qui ne sont pas accessibles par défaut quand on lance Python. Pour calculer $\sin(\frac{\pi}{2}) + \sqrt{2} - e$, on importe par exemple le module `math` qui contient un certain nombre de fonctions mathématiques. Voici trois façons de procéder (et les trois sont utiles).
— Soit on écrit `from math import *` (cela importe tout le module), puis l'expression `sin(pi/2)+sqrt(2)-e`.
— Soit on écrit `from math import sin,pi,sqrt,e` (seules les commandes `sin`, `pi`, `sqrt`, `e` sont importées), puis `sin(pi/2)+sqrt(2)-e`.
— Soit on écrit `import math as mt` (cela importe tout le module et ses fonctions devront être préfixées par `mt`), puis `mt.sin(mt.pi/2)+mt.sqrt(2)-mt.e`.
Pour connaître toutes les commandes contenues dans le module `cmath`, on entre l'instruction `dir("cmath")`.

3. Les notebooks donnent une visualisation facile et une bonne présentation avec différents effets de style. Ils sont bien adaptés à l'utilisation en intelligence artificielle. Cependant, ils ne garantissent pas l'ordre d'exécution du code et rendent délicats la définition de modules et le lien avec d'autres outils : ils ne sont donc pas appropriés au développement de logiciels.

1.1.4 Utilisation de fichiers

Dans le menu de IDLE, sélectionner "File", puis "New File". Une nouvelle fenêtre s'ouvre, dans laquelle on peut entrer des instructions, avec des retours à la ligne sans que la ligne précédente soit exécutée. Toutes ces instructions peuvent cependant être exécutées en bloc en utilisant la touche "f5".

✔ *Dans la pratique, on crée un fichier*[4] *dès que l'on souhaite écrire une procédure ou tracer un graphe. Le terminal utilisé jusqu'à présent sert alors à évaluer certaines fonctions du fichier pour des valeurs données des paramètres.*

1.2 Types d'objets

1.2.1 Entiers et flottants

Différents types d'objets existent en Python : entiers, flottants, chaînes de caractères...
La commande `type` donne le type d'un objet. Entrer `type(-44)` et `type(-44.0)`.

On peut convertir l'entier -44 en un nombre flottant en écrivant `float(-44)`, de même que l'on peut convertir le flottant -44.1 en entier (i. e. calculer $\text{sgn}(x)\,E(|x|)$) en écrivant `int(-44.1)`. La partie entière est donnée par la commande `floor` du module `math`. Quant à la commande `round`, elle donne un arrondi, et l'on peut préciser le nombre de décimales souhaité. Exemple : `round(44.7318,2)`.

1.2.2 Variables

Une **variable** est une *référence*[5] à une "case mémoire" dans laquelle on peut mettre un objet (nombre, polynôme, fonction, matrice, graphique...).
Par exemple, pour créer une variable a qui fait référence à l'objet 13 (on dit que l'on affecte la valeur 13 à la variable a), on écrit `a=13`. Evaluer alors a et a^2.

On peut créer plusieurs variables simultanément. Par exemple `b,c=4,7`. L'affectation des variables se fait en parallèle, ce qui évite le recours à une troisième variable quand on veut échanger leurs valeurs : les instructions `b,c=c,b` échangent le contenu des variables b et c. Attention cependant au cas des tableaux 2d dont on reparlera au

4. Le module `cx_Freeze` transforme un fichier Python en exécutable : pour un fichier dont le nom est `bonjour.py`, on le place dans un dossier, appelé par exemple `Compilation`, situé sur le bureau. On crée dans ce dossier le fichier `setup.py` qui contient les trois lignes suivantes.

```
from cx_Freeze import setup, Executable
setup(name='Salutation',version='1.0',description='Programme pour saluer',
        executables=[Executable('bonjour.py')])
```

Ensuite, on lance un terminal et l'on obtient une ligne semblable à `C:\Users\Fred>`.
Pour se placer dans le dossier `Compilation`, on entre l'instruction `cd Desktop\Compilation`.
On obtient alors la ligne `C:\Users\Fred\Desktop\Compilation>`.
On entre enfin la commande `python setup.py build` qui construit le fichier exécutable.
Un dossier `build` est alors créé dans le dossier `Compilation`, contenant un dossier qui à son tour comporte plusieurs fichiers dont le fichier exécutable `bonjour.exe`.

5. Et non directement une "case mémoire"! Cette distinction a toute son importance.

paragraphe 1.4.4...
Il est aussi possible d'affecter une même valeur à plusieurs variables en même temps : `a=b=c=4`. Observer également l'effet de l'instruction `a,*b=1,2,3,4`.
Notons enfin que l'objet `None` exprime l'absence de valeur.

On peut ajouter une valeur à une variable donnée (c'est l'incrémentation). L'instruction `a=a+4` peut aussi s'écrire `a+=4` ; on a de même les opérateurs `-=`, `*=`, `/=`, `//=` et `%=`.

Pour connaître la place occupée en mémoire par une variable, on dispose de l'instruction `getsizeof` [6].
`import sys`
`a=range(10000000); sys.getsizeof(a)` $\longrightarrow 48$
`b=[k for k in a]; sys.getsizeof(b)` $\longrightarrow 81\,528\,056$

1.2.3 Nombres complexes

Le nombre complexe i est représenté en Python par la constante `1j`. Pour définir le nombre $z = 4 - 7i$, on écrira `z=4-7j`. On accède alors aux parties réelle et imaginaire de z par les instructions `z.real` et `z.imag`. Quant au module de z, il est donné par `abs(z)`.

Pour utiliser certaines commandes relatives aux nombres complexes, on doit importer le module `cmath`. Après avoir écrit `import cmath`, on peut calculer l'argument de z (`cmath.phase(z)`) et son écriture exponentielle (`cmath.polar(z)`). Inversement, l'écriture cartésienne d'un nombre complexe $z = \rho e^{i\theta}$ s'obtient par la commande `rect`. Par exemple pour $e^{i\frac{\pi}{4}}$, on écrit `cmath.rect(1,cmath.pi/4)`.

1.2.4 Booléens

Les **constantes booléennes** sont `True` et `False`. Par exemple, si l'on veut savoir si l'inégalité $4 < 13$ est vraie, on entre `4<13` et Python renvoie `True`. On teste l'égalité (resp. la non égalité) de deux valeurs au moyen de l'opérateur == (resp. `!=`).
La conjonction (resp. la disjonction) est donnée par `and` (resp. `or`).
Quant à la négation, elle s'écrit `not`.
On convertit un booléen en un entier en écrivant par exemple `int(True)`.
Tester ces différentes commandes à partir des formules $4 < 13$, $4 > 13$, $4 = 4$, $4 \neq 4$, en créant des variables qui les contiennent.

1.2.5 Chaînes de caractères

Une **chaîne de caractères** est une suite de caractères placée entre guillemets ou entre apostrophes. Par exemple `x='vacances'`. On peut concaténer des chaînes de caractères grâce à l'opérateur `+`. On écrit ainsi `'vive les '+x`.

6. Le résultat est exprimé en **bytes**, un byte étant la plus petite unité de mémoire allouable, et en général un byte vaut un octet.

Calculer aussi `7*x`.
Pour concaténer des chaînes `c1,c2,...,cn` on écrit `c=''.join([c1,c2,...,cn])`.

La commande `str` convertit un entier ou un flottant en une chaîne de caractères, ce qui est utile pour les concaténations. Exemple : `'numéro '+str(4)`. Réciproquement, on a les commandes `int` et `float` : entrer `int('44')` et `float('44.7')`.
La commande `%s` permet de placer une valeur dans une chaîne de caractères : par exemple, l'instruction `'%s éléphants et %s pandas'%(4,7)` renvoie la chaîne de caractères `'4 éléphants et 7 pandas'`.

1.2.6 Listes

Une **liste** en Python est une suite finie d'objets séparés par des virgules qui est placée entre crochets. Un exemple de liste est `L=[4,-13.7,'vacances',True]`. Une liste est en fait un tableau (au sens où l'on peut accéder facilement à n'importe quelle case et en temps constant, indépendamment de la longueur de la liste) dynamique (au sens où l'on peut ajouter et retirer des éléments et modifier leur valeur).
Attention, toute liste est indexée à partir de 0 (et non 1). La liste vide est notée `[]`. La commande `len` retourne le nombre d'éléments d'une liste, et `L[j]` le $j-$ème élément de la liste L.

On peut ajouter un élément à la fin d'une liste grâce à la méthode `append`.
Exemple : `L.append(92160)`.
Pour concaténer deux listes, on utilise l'opérateur `+`. Exemple : `L+[2,False,'ab']`.
On peut également construire des listes avec l'opérateur `*`. Exemples : `4*L` (i. e. "4 fois la liste L"), `7*[0]`.
L'instruction `L[i:j]` renvoie la sous-liste de L formée des éléments d'indices compris dans l'intervalle $[\![i,\, j[\![$. Exemple : `L[0:3]`.
Si l'on omet l'indice i (resp. j), Python le remplace automatiquement par 0 (resp. la taille de la liste). Exemples : `L[:3]`, `L[1:]`.
On peut aussi accéder aux éléments d'une liste en utilisant des indices négatifs. Ainsi `L[-1]` renvoie le dernier élément de la liste.
La commande `del` supprime un élément d'une liste. Exemple : `del(L[2])`. Et la méthode `pop` supprime le dernier élément d'une liste et le récupère : `z=L.pop()`.
Attention, les variables sont des *références* : si l'on définit `x=[4,7,13,0]` et `y=x`, alors la modification `x[1]=44` affectera aussi y ("car x et y pointent vers la même case mémoire"). En revanche si l'on définit `x=[4,7,13,0]` et `y=[4,7,13,0]`, alors la modification `x[1]=44` n'affectera pas y ("car x et y pointent vers deux cases mémoire différentes").

Etant donné une liste L comportant un élément a, l'instruction `L.index(a)` donne l'indice correspondant à la première occurrence de a dans L.
Quant à l'instruction `L.reverse()`, elle modifie la liste L en renversant l'ordre de ses éléments. On peut aussi créer une copie inversée de `L` en écrivant `M=L[::-1]`.

L'instruction `list(range(0,4))` renvoie la liste des entiers de l'intervalle $[\![0,\, 3]\!]$.

Autres exemples : `list(range(0,14,3))`, `list(range(6,1,-1))`.

Une méthode très importante pour définir une liste est donnée par l'exemple `[1/j for j in range(1,10)]`. On peut aussi spécifier une condition sur j : `[1/j for j in range(1,10) if j%2==0]`.

La commande `sum` permet de calculer des sommes finies. Ainsi on obtient la somme $\sum_{k=1}^{10} k$ au moyen de `sum([k for k in range(1,11)])`. De même, $\sum_{k=1}^{100} \frac{1}{k^2}$ s'obtient par `sum([1/k**2 for k in range(1,101)])`.

La commande `L.count(a)` donne le nombre d'occurrences de `a` dans `L`. On peut alors en déduire l'élément le plus fréquent dans `L`.
`L=[2,1,3,4,3,4,1,1,4,4,5,4,4,3,2,5]; max(L,key=L.count)` $\longrightarrow 4$
On peut également utiliser la fonction `Counter`.
```
from collections import Counter
Counter(L) ⟶ Counter({4: 6, 1: 3, 3: 3, 2: 2, 5: 2})
```

Il est utile, en particulier pour des données provenant de fichiers, de pouvoir transformer une chaîne de caractères en une liste. La commande `c.split('séparateur')` effectue cette opération.
`c='1.44,j,7.32,m'; L=c.split(',')` $\longrightarrow$ `['1.44', 'j', '7.32', 'm']`

1.2.7 Polynômes

Pour effectuer des calculs sur les polynômes, on importe le module `numpy`, dont nous reparlerons dans le paragraphe 1.4.1, en écrivant `import numpy as np`.

La commande `poly1d` permet de créer un polynôme en définissant ses coefficients dans l'ordre des degrés décroissants. Exemple : `P=np.poly1d([4,7,-1,2])`. On peut alors afficher ce polynôme en écrivant `print(P)`. Les opérations sur les polynômes sont obtenues grâce aux opérateurs classiques $+$, $-$, $*$. Le degré d'un polynôme est donné par la commande `order`. Exemple : `P.order`. Et l'on obtient la liste des coefficients de P (toujours dans l'ordre des degrés décroissants) en écrivant `P.c`. Pour avoir le coefficient de degré j, on écrit `P[j]`.

On obtient le polynôme dérivé (resp. primitive s'annulant en 0) grâce à `P.deriv()` (resp. `P.integ()`). Pour évaluer le polynôme en un point a, il suffit d'écrire `P(a)`. Pour calculer le quotient et le reste de la division euclidienne d'un polynôme A par un polynôme B, on procède ainsi.
```
A=np.poly1d([4,7,-1,2]); B=np.poly1d([3,-5])
Q,R=A/B ⟶ poly1d([ 1.33333333,  4.55555556,  7.25925926]),
    poly1d([ 38.2962963])
```

1.3 Graphes de fonctions et courbes paramétrées

1.3.1 Le module `matplotlib`

Pour tracer des graphiques avec Python, on doit importer le sous-module `pyplot` du module `matplotlib`. Notons aussi que le module `numpy` est souvent nécessaire. On commencera donc le fichier contenant les instructions graphiques par les deux lignes ci-dessous.

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

1.3.2 Graphes de fonctions

• Pour tracer le graphe de la fonction f sur l'intervalle $[a,\, b]$, on discrétise cet intervalle en une suite $t_0, \ldots, t_n$ d'abscisses. On trace alors la ligne brisée joignant les points de coordonnées $(t_j,\, f(t_j))$. Puis la commande `show()` (toujours placée à la fin) affiche le graphique. Notons que la commande `clf()` efface la fenêtre graphique. Pour tracer la fonction $\cos$ sur $[0,\, 10]$, on écrit donc

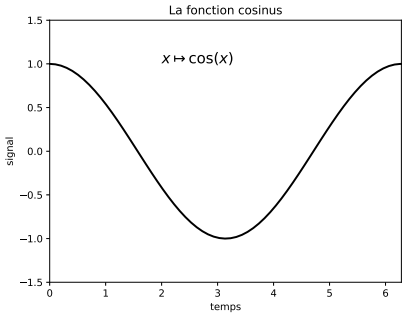
`t=np.linspace(0,10,100)` (ici on choisit 100 points de discrétisation)

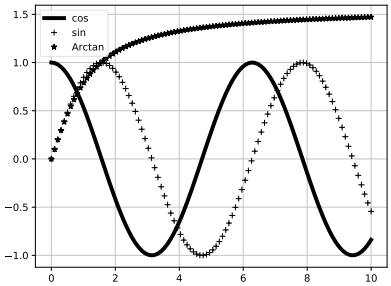
```
y=np.cos(t)
plt.plot(t,y); plt.show()
```

• De nombreuses options peuvent être ajoutées. Pour avoir un quadrillage, on écrit `grid()`. La commande `axis` spécifie une fenêtre d'affichage : on obtient la fenêtre $[0,\, 2\pi] \times [-1.5,\, 1.5]$ en écrivant `plt.axis([0,2*np.pi,-1.5,1.5])`. On peut indiquer un troisième argument dans `plot` pour préciser un style de tracé ou une couleur. On peut aussi choisir l'épaisseur du trait grâce à la commande `linewidth`.
L'instruction `plt.axis('equal')` permet d'avoir un repère orthonormé.
On peut aussi indiquer des informations sur les axes.
Exemple : `plt.xlabel('temps'); plt.ylabel('signal')`.
L'instruction `plt.title('La fonction cosinus')` ajoute un titre au graphe.
Il est aussi possible d'écrire du texte sur le graphique, éventuellement en Latex :
`plt.text(2,1,r'$x\mapsto\cos(x)$',fontsize=15,color='b')`.

• Pour superposer plusieurs graphes, on procède comme ci-après. La légende (facultative) `plt.legend(('cos','sin','Arctan'))` (dans l'ordre indiqué pour le tracé) permet de les repérer.

```
t=np.linspace(0,10,100)
y=np.cos(t); z=np.sin(t); w=np.arctan(t)
plt.plot(t,y,'r',t,z,'+k',t,w,'*g',linewidth=4)
plt.legend(('cos','sin','Arctan')); plt.grid(); plt.show()
```





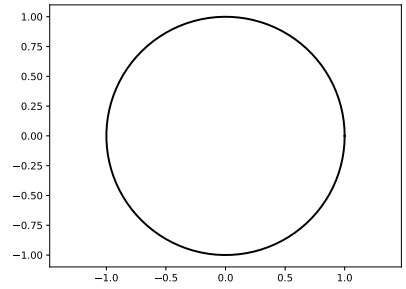
1.3.3 Courbes paramétrées

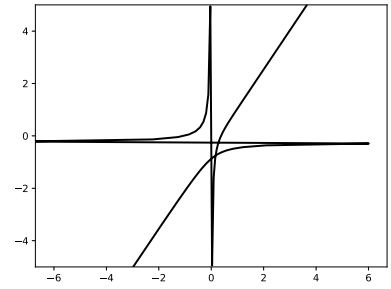
Pour tracer la courbe paramétrée $t \mapsto (x(t), y(t))$, on utilise toujours la commande `plot`. Exemple :

```
t=np.linspace(0,2*np.pi,100)
x=np.cos(t); y=np.sin(t)
plt.plot(x,y,'r'); plt.axis('equal'); plt.show()
```

De même, pour tracer la courbe définie par $t \mapsto \left(\dfrac{2t+t^2}{3+3t},\ \dfrac{2t^2-1}{4t}\right)$, on écrit les instructions suivantes.

```
t=np.linspace(-10,10,200)
x=(2*t+t**2)/(3+3*t); y=(2*t**2-1)/(4*t)
plt.plot(x,y,'r'); plt.axis('equal');
plt.axis([-5,5,-5,5]); plt.show()
```





1.4 Algèbre linéaire

1.4.1 Le module `numpy`

Pour effectuer des calculs sur les polynômes, on importe le module `numpy` : on le fait en général en écrivant `import numpy as np`. Une fois `numpy` importé, il est

inutile d'importer `math` ou `cmath`, car toutes les fonctions élémentaires de `numpy` (a priori plutôt matricielles) s'appliquent aussi aux scalaires. Ce module contient encore les constantes π et e, qui s'écrivent `np.pi` et `np.e`.

1.4.2 Tableaux

La fonction `array` permet de construire des tableaux à une ou plusieurs dimensions. Ainsi, les vecteurs sont des tableaux à une dimension (1d), les matrices des tableaux à deux dimensions (2d). Voici deux exemples.

```
T=np.array([4,7.1,8,0]); print(T); type(T)
U=np.array([[1,2],[3,4]]); print(U)
```

La commande `linspace(a,b,n)` construit un tableau 1d dont les n composantes sont uniformément réparties entre a et b.
Exemple : `np.linspace(0,1,5)`.
La commande `arange(a,b,inc)` crée un tableau 1d. Elle fonctionne comme la commande `range(a,b,n)` sauf que le troisième argument peut être un flottant.
Exemple : `np.arange(0,1,0.2)`.

La fonction `zeros` (resp. `ones`) crée un tableau contenant uniquement des 0 (resp. des 1). Exemples : `np.zeros(5)`, `np.ones((3,4))` (tableau 2d de taille 3×4).
Les fonctions `zeros_like` et `ones_like` créent des tableaux de 0 ou de 1 ayant les mêmes dimensions qu'un autre tableau T.
La fonction `eye` donne des matrices identités. Exemple : `np.eye(4)`.

La fonction `diag` construit un tableau 2d à partir d'une diagonale donnée sous la forme d'un tableau 1d, ou extrait une diagonale d'un tableau 2d sous la forme d'un tableau 1d. Exemples :

```
np.diag(np.ones(3))
```

`np.diag(np.ones(3),k=1)` (k précise le numéro de la diagonale, et par défaut on a $k = 0$)

```
A = np.array([[1,2,3],[4,5,6],[7,8,9]]); print(A); np.diag(A);
np.diag(A,k=-1)
```

Les attributs `ndim`, `shape` et `dtype` donnent respectivement le nombre de dimensions d'un tableau, le nombre d'éléments dans chaque dimension et le type des éléments du tableau. Exemple : `U.ndim; U.shape; U.dtype`.
On accède au $j-$ème élément d'un tableau 1d T en écrivant `T[j]` (comme pour les listes). On récupère les éléments d'indices appartenant à $[\![j,\ k[\![$ en écrivant `T[j:k]`.
On accède au $(i,\ j)-$ème élément d'un tableau 2d U en écrivant `U[i,j]`. Pour récupérer toute la colonne d'indice 1 on écrit `U[:,1]`. Pour récupérer les éléments de la ligne d'indice 0 dont l'indice de colonne est supérieur ou égal à 1, on écrit `U[0,1:]`.

La moyenne et l'écart-type des éléments d'un tableau `a` sont respectivement donnés par `np.mean(a)` et `np.std(a)`.

1.4.3 Tableaux contenant des coefficients aléatoires

Le sous-module `random` de `numpy` permet de générer des nombres aléatoires :
— `random.rand` pour la loi uniforme $\mathcal{U}([0,\,1])$,
— `random.randn` pour la loi normale $\mathcal{N}(0,\,1)$,
— `random.randint` pour une loi uniforme sur un intervalle d'entiers.
Exemples :
`np.random.rand(3)` (tableau 1d, loi uniforme sur $[0,\,1]$)
`np.random.randn(4,2)` (tableau 2d 4×2, loi normale $\mathcal{N}(0,\,1)$)
`np.random.randint(0,5,10)` (tableau 1d, loi uniforme sur les entiers de $[\![0,\,4]\!]$)
`np.random.randint(0,5,(4,4))` (tableau 2d, loi uniforme sur $[\![0,\,4]\!]$).

1.4.4 Opérations sur les tableaux

• Soient a, b des scalaires, n un entier positif et u, v des tableaux de même dimension, alors `u+v`, `u-v`, `a*u+b*v`, `u*v`, `u/v`, `u**n` effectuent les opérations élément par élément. Par exemple, les éléments du tableau donné par `u*v` ont pour $(i,\,j)-$ème composante $u_{i,j}v_{i,j}$ (et non $\sum_{k=0}^{n} u_{i,k}v_{k,j}$!). Cependant, les combinaisons linéaires élément par élément correspondent aux combinaisons linéaires des tableaux u et v.

La multiplication matricielle s'écrit avec la fonction `dot` : si le nombre de colonnes de A est égal au nombre de lignes de B, alors `np.dot(A,B)` calcule le produit de A par B.
De même, si x est un tableau 1d de taille n, et A un tableau 2d de taille $m \times n$, alors `np.dot(A,x)` effectue le produit Ax, le résultat étant un tableau 1d (et non un vecteur colonne !).
Si y est un tableau 1d de taille m, `np.dot(y,A)` effectue le produit ${}^t yA$, le résultat étant un tableau 1d (et non un vecteur ligne !).
Si x et y sont deux tableaux 1d de même taille, `np.dot(x,y)` effectue le produit scalaire $(x \,|\, y)$.
Si x est un tableau 1d de taille m et y un tableau 1d de taille n, `A=np.outer(x,y)` renvoie un tableau 2d de taille $m \times n$ de coefficients $A_{i,j} = x_i y_j$.

Pour calculer des puissances de matrices, on utilise la commande `matrix_power` du sous-module `linalg` de `numpy`.
Exemple : `np.linalg.matrix_power(U,2)`.
Cette commande calcule aussi l'inverse d'une matrice.
Exemple : `np.linalg.matrix_power(U,-1)`.

Les fonctions élémentaires `cos`, `sin`, `exp`, `log` du module `numpy` s'appliquent aux tableaux et agissent élément par élément (`log` désigne le logarithme népérien).

Les méthodes `sum`, `prod`, `mean`, `var`, `std`, `min`, `max` calculent respectivement la somme, le produit, la moyenne, la variance, l'écart-type, le minimum et le maximum des éléments d'un tableau. Exemple : `U.prod()`.

• On peut transformer un tableau md en un tableau nd grâce à la commande `reshape` à condition de ne pas changer le nombre total d'éléments du tableau.
Exemples : `T = np.arange(12); TT = np.reshape(T,(4,3));`
`TTT = np.reshape(T,(2,3,2))`
Notons que le tableau initial et le tableau redimensionné partagent les mêmes éléments, si bien qu'une modification de l'un des tableaux affectera l'autre.

La fonction `concatenate` permet de concaténer des tableaux ayant le même nombre de dimensions.
Exemple : `A=np.ones((2,2)); B=np.zeros((2,2))`
`np.concatenate((A,B))` (les tableaux sont placés l'un en dessous de l'autre)
`np.concatenate((A,B),axis=0)` (idem)
`np.concatenate((A,B),axis=1)` (les tableaux sont placés l'un à côté de l'autre)

• On peut sélectionner et modifier les éléments d'un tableau à l'aide de conditions booléennes : par exemple l'instruction `U[U>2]=44` remplace tout élément du tableau U qui est strictement supérieur à 2 par 44.

Pour échanger deux coefficients d'un tableau 1d T, on peut faire une affectation en parallèle des variables, comme ceci.
`a=np.array([1,2,3,4]); a[2],a[1]=a[1],a[2]`
Attention, cette méthode n'est plus valable quand on échange deux lignes d'un tableau 2d, comme le montrent les instructions ci-après.
`b=np.array([[1,2],[3,4]]); b[1,:],b[0,:]=b[0,:],b[1,:]`
Pour échanger les deux lignes de b, on procède à l'aide d'une variable auxiliaire.
`b=np.array([[1,2],[3,4]])`
`t=np.copy(b[0,:]); b[0,:]=np.copy(b[1,:]); b[1,:]=t`

Attention, si l'on définit un tableau d'entiers T, et si l'on essaie d'attribuer une valeur flottante à l'un de ses coefficients, cette valeur est automatiquement convertie en entier. Cependant, pour forcer l'affectation, on peut créer une copie de T dont les coefficients sont des flottants, en écrivant `U=T.astype(float)`.

• Pour transposer une matrice, on peut utiliser la méthode `transpose` ou l'attribut `T`.
Exemple : `V=U.transpose(); W=U.T`.
Comme avec la commande `reshape`, le tableau transposé partage ses éléments avec le tableau initial. Si cela ne convient pas, il faut utiliser la commande `copy`.
Exemple : `UU=np.copy(U)`, ou `UU=U.copy()`.

La trace d'une matrice est donnée par la commande `trace`. Exemple : `np.trace(U)`.
Son déterminant est donné par la commande `det` du sous-module `linalg`. Exemple : `np.linalg.det(U)`.

La commande `eig` du sous-module `linalg` calcule les valeurs propres et les vecteurs propres d'une matrice.

Exemple : `l,P=np.linalg.eig(U)`
`Q=np.linalg.matrix_power(P,-1); D=np.dot(np.dot(Q,U),P)`
`np.dot(U,P[:,0])-l[0]*P[:,0]; np.dot(U,P[:,1])-l[1]*P[:,1]`
La matrice P est la matrice de passage de la base canonique de $\mathbb{C}^n$ vers la base de vecteurs propres, et ses vecteurs colonnes sont les vecteurs propres associés aux valeurs propres contenues dans le tableau `l`.
Le polynôme caractéristique ($\chi_A(\lambda) = \det(\lambda I_n - A)$) est donné par la commande `poly`.
Exemple : `np.poly(U)` $\longrightarrow$ `array([ 1., -5., -2.])`, ce qui signifie que $\chi_U(\lambda) = \lambda^2 - 5\lambda - 2$.

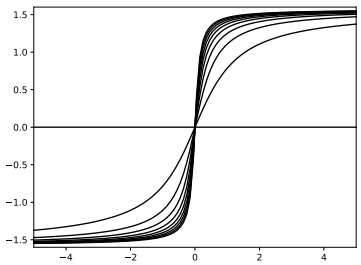
Considérons deux vecteurs $x = (x_0, \ldots, x_{m-1})$ et $y = (y_0, \ldots, y_{m-1})$ de taille m, où $x_0, \ldots, x_{m-1}$ sont distincts, et un entier $n < m$. Alors il existe un unique polynôme P de degré inférieur ou égal à n qui minimise la quantité $\sum_{j=0}^{m-1} |P(x_j) - y_j|^2$: ce polynôme s'appelle le **polynôme des moindres carrés** de degré inférieur ou égal à n associé à x et y. Il peut être calculé au moyen de la commande `polyfit`, de syntaxe `np.polyfit(x,y,n)`.

1.5 Graphes de fonctions, courbes implicites, représentations graphiques dans l'espace, animations

1.5.1 Superposition de graphes

Pour compléter les résultats du paragraphe 1.3.2, donnons un exemple de superposition de graphes utilisant une boucle.

```
t=np.linspace(-5,5,100)
for n in range(10):
    plt.plot(t,np.arctan(n*t),
        color=(n/10,(10-n)/10,0.8))
plt.show()
```

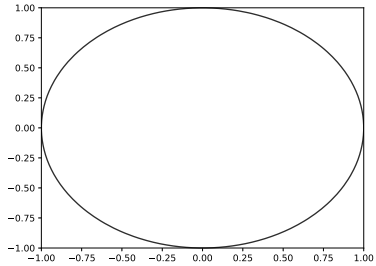


1.5.2 Courbes implicites

On peut représenter l'ensemble des points du plan vérifiant une équation (on parle alors de courbe implicite, ou de ligne de niveau) grâce aux commandes `contour` et `meshgrid`. Exemples :

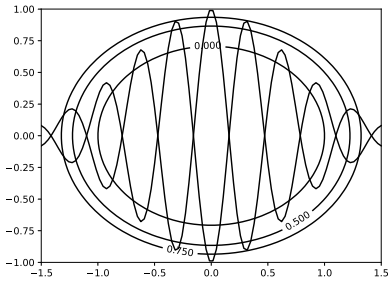
• Tracé de $\{(x, y) \in \mathbb{R}^2 \,/\, x^2 + y^2 - 1 = 0\}$.

```
def f(x,y):
    return x**2+y**2-1
x=np.linspace(-1,1,50)
y=np.linspace(-1,1,50)
X,Y=np.meshgrid(x,y)
Z=f(X,Y)
plt.contour(X,Y,Z,[0]); plt.show()
```



• Superposition de deux types de graphes.

```
def f(x,y):
    return x**2+2*y**2-1
t=np.linspace(-1.5,1.5,100)
z=np.exp(-t**2)*np.cos(10*t)
l1,l2=plt.plot(t,z,'g',t,-z,'c')
x=np.linspace(-1.5,1.5,50)
y=np.linspace(-1,1,50)
X,Y=np.meshgrid(x,y)
Z=f(X,Y)
C=plt.contour(X,Y,Z,[0,0.5,0.75])
plt.clabel(C,fontsize=10)
plt.show()
```

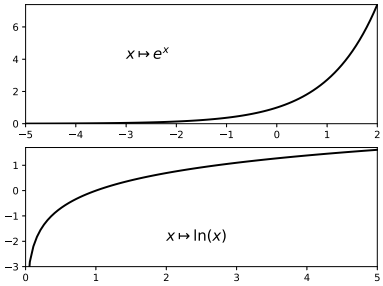


1.5.3 Fenêtres contenant plusieurs graphiques

On peut créer des fenêtres contenant plusieurs graphiques. Voici un exemple.

```
plt.subplot(2,1,1)
t=np.linspace(-5,2,100)
plt.plot(t,np.exp(t),'g')
plt.axis([-5,2,0,np.exp(2)])
plt.text(-3,4,r'$x\mapsto e^x$',
    color='g',fontsize=25)
```

```
plt.subplot(2,1,2)
t=np.linspace(0.01,5,100)
plt.plot(t,np.log(t),'r')
plt.axis([0,5,-3,1.7])
plt.text(2,-2,r'$x\mapsto
    \ln(x)$',color='r',
    fontsize=25)
plt.show()
```



1.5.4 Représentations graphiques dans l'espace

• Pour effectuer des représentations graphiques dans l'espace, on utilise le sous-module `mpl_toolkits.mplot3d`.

```
from mpl_toolkits.mplot3d import Axes3D
```

• On peut représenter des courbes paramétrées dans l'espace. Voici par exemple la représentation d'une hélice.
On commence par initialiser la figure.

```
fig=plt.figure()
ax=fig.gca(projection='3d')
```

Puis on définit la courbe paramétrée et on la trace.

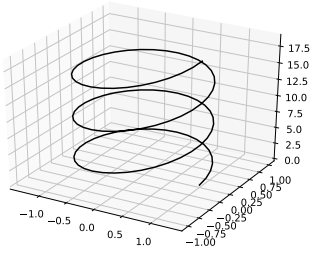
```
t=np.linspace(0,6*np.pi,100)
x=np.cos(t); y=np.sin(t); z=t
ax.plot(x,y,z); ax.axis('equal'); plt.show()
```

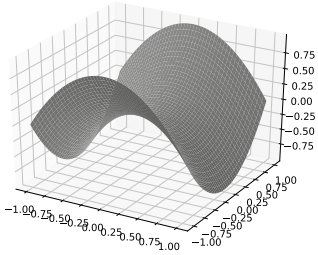
• On peut aussi représenter des surfaces dans l'espace. Voici l'exemple de la surface paramétrée par $f(x, y) \mapsto x^2 - y^2$, qui est un paraboloïde hyperbolique, ou selle de cheval[7].
On commence là encore par initialiser la figure.

```
fig=plt.figure()
ax=fig.gca(projection='3d')
```

Puis on définit la surface et on la représente.

```
h=2/49; x=np.linspace(-1,1,50); y=np.linspace(-1,1,50)
x,y=np.meshgrid(x,y)
 def f(x,y):
     return x**2-y**2
z=np.array([[f(-1+i*h,-1+j*h) for j in range(50)]
    for i in range(50)])
ax.plot_surface(x,y,z,rstride=1,cstride=1); plt.show()
```





7. Une telle surface admet, dans un repère adapté, une équation de la forme $z = \frac{x^2}{a^2} + \frac{y^2}{b^2}$. Pour comtempler un joli paraboloïde hyperbolique, le lecteur est invité à aller sur le toit de la cathédrale Notre-Dame de Royan !

1.5.5 Animations

Pour tracer une courbe animée, on importe la fonction `animation`.

```
from matplotlib import animation
```

On souhaite ici représenter le graphe de $x \mapsto \cos(4x - 2ih)$ sur $[0,\ 4]$ quand i varie de 0 à 99 au cours du temps.

```
h=0.01; n=100
x=np.linspace(0,4,n)
```

• On initialise la figure.

```
fig=plt.figure()
line, =plt.plot([],[])
plt.xlim(0,4); plt.ylim(-1,1)
def init():
    line.set_data([],[])
    return line,
```

• On définit le graphe à afficher au $i-$ème pas de l'animation.

```
def animate(i):
    t=i*h; y=np.cos(4*x-2*i*h)
    line.set_data(x, y)
    return line,
```

• La fonction `animation` effectue alors le tracé animé. Notons que le paramètre `frames` donne le nombre de pas de l'animation tandis que le paramètre `interval` règle la vitesse du tracé.

```
ani=animation.FuncAnimation(fig,animate,init_func=init,
    frames=n,blit=True,interval=20,repeat=False)
plt.show()
```

1.6 Résolution numérique d'équations

1.6.1 Systèmes linéaires

Pour résoudre un système linéaire, on peut utiliser la commande `solve` du sous-module `linalg` de `numpy`.
Par exemple, pour résoudre le système linéaire

$$\begin{cases} x + 3y + 5z &= 0 \\ 2x + 5y + 8z &= 8 \\ 2x + 3y + 8z &= 3 \end{cases},$$

on écrit les instructions suivantes.

```
A=np.array([[1,3,5],[2,5,8],[2,3,8]])
b=np.array([0,8,3])
x=np.linalg.solve(A,b)
```

1.6.2 Le module `scipy`

De nombreuses méthodes numériques nécessitent l'utilisation du module `scipy`. Ce module contient la plupart des commandes de `numpy`, et quand on l'importe, il devient souvent inutile d'importer `numpy`.
Pour importer le module `scipy`, on écrit la ligne ci-dessous.

```
import scipy as sp
```

1.6.3 Dérivation d'une fonction

On peut calculer la dérivée d'une fonction f en un point a en utilisant la commande `derivative` du sous-module `misc` de `scipy`. La syntaxe de cette commande est `derivative(f,a)`. Dérivons par exemple la fonction $f : x \mapsto 4x^2$ au point 1.

```
from scipy import misc
def f(x): return 4*x**2
misc.derivative(f,1)
```

1.6.4 Equations non linéaires

La fonction `fsolve` du sous-module `optimize` de `scipy` résout de façon approchée des équations scalaires ou vectorielles qui s'écrivent $f(x_1, \ldots, x_n) = (0, \ldots, 0)$. La syntaxe de cette commande est `op.fsolve(f,v)`, où f est une fonction de $\mathbb{R}^n$ dans $\mathbb{R}^n$ et où v est un vecteur de $\mathbb{R}^n$ "qui n'est pas trop éloigné de la solution". Voici deux exemples.

Equation

$x + 2\cos(x) = 0$

```
import scipy.optimize as op
def f(x):
    return x+2*sp.cos(x)
x0=op.fsolve(f,0.3)
print(x0)
```

Système d'équations

$x\cos(y) = 4$ et $xy - y = 5$

```
import scipy.optimize as op
def g(x):
    return [x[0]*sp.cos(x[1])-4,
        x[1]*x[0]-x[1]-5]
z0=op.fsolve(g,[1,1])
print(z0)
```

On peut aussi utiliser la fonction `newton` du même sous-module : la syntaxe de cette commande est `op.newton(f,v,fprime=None)` : quand aucune fonction dérivée `fprime` n'est précisée, la méthode de la sécante est utilisée, sinon c'est la méthode de Newton.
Quant à la fonction `bisect` du même sous-module, elle recherche une racine par dichotomie dans un segment $[a, b]$, et sa syntaxe est `op.bisect(f,a,b)`.

1.6.5 Equations différentielles

On importe le sous-module `integrate` du module `scipy`.

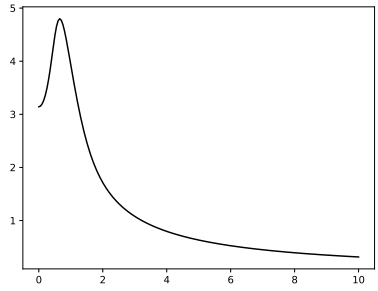
```
import scipy.integrate as it
```

La commande `odeint` du sous-module `integrate` résout de façon approchée des équations différentielles d'ordre 1 scalaires ou vectorielles qui s'écrivent sous la forme $x'(t) = f(x(t), t)$ avec la condition initiale $x(t_0) = x_0$. La syntaxe de cette commande est `it.odeint(f,x0,t)`. Cette commande retourne les valeurs de la

solution approchée évaluée sur les composantes du vecteur t, vecteur dont la première composante doit être t_0. Voici deux exemples.

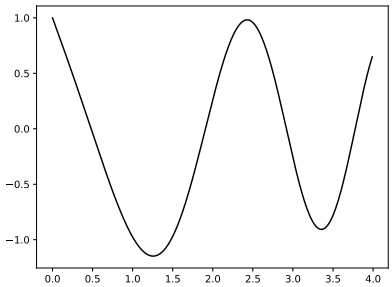
• Ordre 1 : $x' - (t + x)\sin(tx) = 0$,
avec $x(0) = \pi$.

```
def f(x,t):
    return (t+x)*sp.sin(t*x)
t=sp.linspace(0,10,200)
sol=it.odeint(f,sp.pi,t)
plt.plot(t,sol)
plt.show()
```



• Ordre 2 : $x'' + 4tx = 0$,
avec $x(0) = 1$, $x'(0) = -2$.
Cette équation est équivalente à
$\mathbf{y}' = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix}' = \begin{bmatrix} y_1 \\ -4ty_0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -4t & 0 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -4t & 0 \end{bmatrix} \mathbf{y}$,
i. e. $\mathbf{y}' = f(\mathbf{y}, t)$, avec $f(\mathbf{y}, t) = \begin{bmatrix} 0 & 1 \\ -4t & 0 \end{bmatrix} \mathbf{y}$.

```
y0=[1,-2]
def f(y,t):
    return [y[1],-4*t*y[0]]
t=sp.arange(0,4.0,0.01)
sol=it.odeint(f,y0,t)
plt.plot(t,sol[:,0])
plt.show()
```



Notons que dans le cas où l'équation comporte des paramètres $k_1, \ldots, k_r$, ceux-ci doivent être ajoutés aux arguments de la fonction f : `f(y,t,k_1,...,k_r)`. De plus, le $r-$uplet $(k_1, \ldots, k_r)$ de ces paramètres doit être placé en dernier argument de la fonction `odeint` : `sol=it.odeint(f,y0,t,(k_1,...,k_r))`.

1.7 Intégration numérique

On utilise le sous-module `integrate` du module `scipy` pour calculer (de façon approchée) des intégrales. On écrira donc les deux lignes ci-dessous.

```
import scipy as sp
import scipy.integrate as it
```

1.7.1 Méthode standard

La commande permettant d'intégrer une fonction $f : x \mapsto f(x)$ sur un intervalle $[a, b]$ est `quad`. Sa syntaxe est `it.quad(lambda x: f(x),a,b)`. Notons que le résultat est donné sous la forme d'un couple (I, e) où I est une valeur approchée de l'intégrale et où e borne l'erreur d'approximation.

La syntaxe `lambda x: f(x)` signifie que l'on crée une "fonction anonyme" définie par $x \mapsto f(x)$.
Exemples : `it.quad(lambda x: sp.cos(x),0,sp.pi/2)`
`it.quad(lambda x: sp.exp(-x**2),-10,10)`
`it.quad(lambda x: sp.exp(-x**2),-10,10)[0]-sp.sqrt(sp.pi)`

1.7.2 Méthode des trapèzes

Quand on ne connaît une fonction que via ses valeurs $y_0, \ldots, y_n$ en les points d'une subdivision $x = (x_0, \ldots, x_n)$ de pas dx, on peut utiliser la méthode des trapèzes pour calculer son intégrale approchée avec la commande `trapz` du module `numpy`. Sa syntaxe est `np.trapz(y,x,dx)`. L'argument y est un tableau 1d de composantes $y_0, \ldots, y_n$ et les arguments x et dx sont optionnels. S'ils ne sont pas précisés, x est un tableau de même dimension que y et de composantes $1, 2, \ldots$, et dx vaut 1. Exemple : `x=np.linspace(-10,10,40); np.trapz(np.exp(-x**2),x)`.

1.8 Simulation des lois de probabilité

• Le sous-module `stats` de `scipy` permet d'effectuer des calculs avec les lois de probabilité classiques, comme par exemple les lois normale (`norm`), exponentielle (`expon`), uniforme (`uniform`), de Bernoulli (`bernoulli`), binomiale (`binom`), géométrique (`geom`), hypergéométrique (`hypergeom`), de Poisson (`poisson`), discrète uniforme (`randint`). Notons que l'on obtient une loi normale de moyenne m et d'écart-type s en écrivant `norm(m,s)`.

• On définit une variable aléatoire et l'on peut alors utiliser sa fonction de densité de probabilité `pdf` (probability density function, pour une loi continue) ou sa fonction de masse `pmf` (probability mass function, pour une loi discrète), sa fonction de répartition `cdf` (cumulative density function) et la réciproque de la fonction de répartition `ppf` (percent point function). On accède à la moyenne et à l'écart-type de la variable aléatoire grâce aux méthodes `mean` et `sdt`. On peut effectuer des tirages aléatoires relevant de cette loi en utilisant la commande `rvs`. Voici deux exemples.

Loi binomiale

```
from scipy.stats import binom
va=binom(20,0.4)
va.rvs(size=5)
va.pmf(4); va.pmf(40)
va.cdf(19); va.cdf(20)
va.ppf(0.5); va.ppf(0.6)
```

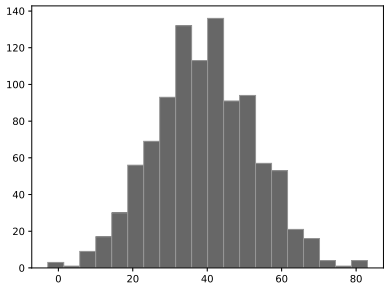
Loi de Poisson

```
from scipy.stats import poisson
va=poisson(3.1)
va.rvs(size=5)
va.pmf(2); va.pmf(4)
va.cdf(4); va.cdf(10)
va.ppf(0.85); va.ppf(0.99)
```

On peut aussi tracer des histogrammes, comme dans les exemples ci-dessous.
`import matplotlib.pyplot as plt`

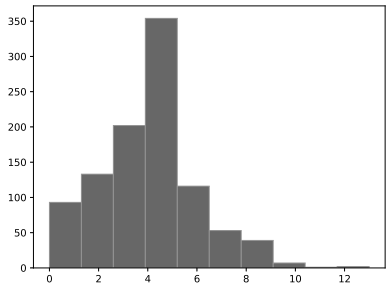
Loi normale

```
from scipy.stats import norm
va=norm(40,13); L=va.rvs(size=1000)
plt.hist(L,20,fc='g',ec='r')
plt.show()
```



Loi de Poisson

```
from scipy.stats import poisson
va=poisson(4)
L=va.rvs(size=1000)
plt.hist(L,10,fc='g',ec='r')
```



1.9 Calcul formel

1.9.1 Le module `sympy`

Python permet aussi de faire du calcul formel grâce au module `sympy`, c'est-à-dire de manipuler des symboles et des expressions littérales, sans utiliser de valeurs numériques. On obtient ainsi les solutions exactes de nombreuses équations, ou la valeur exacte d'intégrales dépendant de paramètres. Pour utiliser ce module, on écrit [8]

```
from sympy import *
```

Voici un aperçu des possibilités du module.

1.9.2 Exemples

• On peut définir des expressions contenant des symboles.
Exemple : `x,y=symbols('x y'); e=2*x-4*y`

• On peut créer une variable p contenant l'expression $xy - 4x^2 - 10y + 40x$, factoriser cette expression (avec `factor`) et stocker le résultat dans une variable q pour ensuite développer q (avec `expand`) et stocker le résultat dans une variable r.

```
p=x*y-4*x**2-10*y+40*x; q=factor(p); r=expand(q)
```

• Calcul de $\sin(\frac{\pi}{2})$: `sin(pi/2)`

Calcul de $\lim\limits_{x\to 0} \dfrac{\sin(x)}{x}$: `limit(sin(x)/x,x,0)`

8. La ligne ci-dessous permet d'avoir un meilleur affichage.

```
from sympy import init_printing; init_printing()
```

Calcul de la dérivée de $x \mapsto \cos(x)e^x$: `diff(cos(x)*exp(x),x)`

Calcul d'une primitive de $x \mapsto x^2\cos(x) - 7e^x\sin(x)$:
`integrate(x**2*cos(x)-7*exp(x)*sin(x),x)`

Calcul de l'intégrale généralisée $\int_{-\infty}^{\infty} \cos(x^2)dx$:
`integrate(cos(x**2),(x,-oo,oo))`

Résolution de l'équation $x^2 - 3 = 0$: `solve(x**2-3,x)`

Résolution de l'équation différentielle $y'' - y = e^t$:

```
t=symbols('t'); y=Function('y')
dsolve(Eq(y(t).diff(t,t)-y(t),exp(t)),y(t))
```

Détermination des éléments propres d'une matrice :

```
A=Matrix([[1, 2], [2, 2]]); A.eigenvals(); A.eigenvects()
```

Développement limité de e^x : `series(exp(x),x,0,4)`

Chapitre 2

Algorithmes classiques

Evoquer l'algorithme d'Euclide fait prendre conscience que l'informatique préexiste aux ordinateurs[1] ! On attribue cependant les prémices de cette science à trois grands noms : Gottfried Leibniz (XVII^{e} siècle) qui introduit le langage formel de la logique et le calcul binaire (jugé alors esthétique mais inutile !), Charles Babbage (XIX^{e} siècle) qui conçoit une machine analytique[2] (jamais réalisée en son temps), et Alan Turing (XX^{e} siècle).
Il est essentiel de maîtriser l'algorithmique pour être en mesure de s'adapter à tout langage de programmation : alors qu'un programme dépend d'un langage donné, un algorithme en est indépendant.

Conformément au programme, ce chapitre fait entrer le lecteur dans l'algorithmique *in medias res*. Le but est de lui faire découvrir des algorithmes itératifs et récursifs classiques portant sur les nombres et les polynômes, la recherche d'un élément dans un tableau trié, les méthodes de tri d'un tableau, la transformée de Fourier discrète ainsi que des applications au traitement d'images. On pourra se concentrer en première lecture sur les algorithmes plus que sur leur étude, pour reprendre cette étude de manière plus approfondie après avoir traité le chapitre suivant.

2.1 Algorithmes pour les nombres et les polynômes

2.1.1 Calcul de puissances

Etant donné un élément a d'un anneau A ($\mathbb{R}$, $\mathbb{C}$, $\mathcal{M}_n(\mathbb{R})$), on se propose de calculer une puissance a^n de a pour n fixé "assez grand". Ce problème conduit à écrire les algorithmes suivants.

1. Le mot "ordinateur" a été suggéré par le philologue et théologien Jacques Perret en 1955 suite à la demande de son ancien élève, responsable du service publicité d'IBM France.

2. Babbage utilise dans sa machine le principe des cartes perforées emprunté au métier Jacquard, première machine programmable.

2.1.1.1 Algorithme d'exponentiation naïve

L'algorithme naïf consiste à calculer de proche en proche toutes les puissances précédant a^n en utilisant la relation $a^k = a\,a^{k-1}$.
Voici une implémentation de cet algorithme dans le cas où A est $\mathbb{R}$ ou $\mathbb{C}$.

Algorithme en Python 2.1.1 (Exponentiation naïve)

```
def EXP_NAIVE(a,n):
    if n==0:
        return 1
    elif n==1:
        return a
    else:
        b=a
        for j in range(n-1):
            b*=a
        return b
```

L'algorithme se termine (il ne comporte qu'une boucle `for`), il est correct (un invariant de boucle est $\mathcal{P}(j)$: "à la fin de la $j-$ème boucle, $b = a^{j+2}$") et sa complexité est de $n-1$ produits.

2.1.1.2 Algorithme d'exponentiation rapide

Cet algorithme repose sur la méthode "diviser pour régner". Son principe est le suivant. Si l'on écrit $n = 2\,m + r$ avec $r = 0$ ou 1, alors

$$a^n = (a^2)^m\,a^r = \begin{cases} (a^2)^m & \text{si } r = 0 \\ a\,(a^2)^m & \text{si } r = 1 \end{cases} .$$

Nous donnons ici une version récursive et une version itérative.

Algorithme en Python 2.1.2 (Exponentiation rapide)

```
def EXP_RAPIDE_REC(a,n):
   if n==0:
      return 1
   elif n==1:
      return a
   elif n%2==0:
      return EXP_RAPIDE_REC(a*a,n//2)
   else:
      return a*EXP_RAPIDE_REC(a*a,n//2)
```

```
def EXP_RAPIDE(a,n):
   m,y,b=n,1,a
   while m!=0:
      if m%2==1:
         y*=b
      m=m//2
      b*=b
return y
```

L'algorithme se termine (les valeurs successives de m sont strictement décroissantes dans $\mathbb{N}$), il est correct (proposition 1) et sa complexité est en $\Theta(\log_2(n))$ (proposition 2).

Proposition 2.1.1

Un invariant de boucle de l'algorithme d'exponentiation rapide est $\mathcal{P}(m)$: "à la fin de chaque boucle, on a $y\,b^m = a^n$".

Démonstration :
Pour la $k-$ème boucle, on indexe les variables par k, et l'on note $u//v$ le quotient de la division euclidienne de u par v. Il s'agit de prouver que pour tout k, on a $y_k\,b_k^{m_k} = a^n$.
• Au départ, avant la première boucle, on a $m_0 = n$, $y_0 = 1$ et $b_0 = a$. Distinguons deux cas.
— Si m_0 est impair, $y_1 = a$, $m_1 = m_0//2$, $b_1 = a^2$, d'où $y_1\,b_1^{m_1} = a(aa)^{m_0//2} = a^n$.
— Si m_0 est pair, $y_1 = 1$, $m_1 = m_0//2$, $b_1 = a^2$, d'où $y_1\,b_1^{m_1} = 1(aa)^{m_0//2} = a^{m_0} = a^n$.
• Supposons qu'à la fin de la $k-$ème boucle, on a $y_k\,b_k^{m_k} = a^n$. Distinguons alors deux cas.
— Si m_k est impair, alors $y_{k+1} = y_k b_k$, $m_{k+1} = m_k//2$, $b_{k+1} = b_k^2$, d'où par hypothèse de récurrence, $y_{k+1}\,b_{k+1}^{m_{k+1}} = y_k b_k b_k^{2m_{k+1}} = y_k b_k^{m_k} = a^n$.
— Si m_k est pair, alors $y_{k+1} = y_k$, $m_{k+1} = m_k//2$, $b_{k+1} = b_k^2$, d'où par hypothèse de récurrence, $y_{k+1}\,b_{k+1}^{m_{k+1}} = y_k b_k^{2m_{k+1}} = y_k b_k^{m_k} = a^n$.
• A la fin, on a $m = 0$ avec $yb^m = a^n$, d'où $y = a^n$. ■

Proposition 2.1.2

La complexité $C(n)$ de l'algorithme d'exponentiation rapide (en nombre de produits) est en $\Theta(\log_2(n))$.

Démonstration :
Considérons le développement binaire $(n)_2 = \overline{b_p \dots b_0}$ où les $b_k \in \{0,1\}$ sont les restes successifs des divisions par 2 (avec $b_p = 1$). Alors $n = b_p\,2^p + \dots + b_k\,2^k + \dots + b_0$ et $a^n = \left(a^{2^p}\right)^{b_p} \dots \left(a^{2^k}\right)^{b_k} \dots (a^2)^{b_1}\,a^{b_0}$.
On a p produits pour les a^{2^k}, d'où $p \leq C(n) \leq 2\,p$. Comme $2^p \leq n < 2^{p+1}$, alors $p \leq \log_2(n) < p+1$, donc $\log_2(n) - 1 < C(n) \leq 2\log_2(n)$ et $C(n) = \Theta(\log_2(n))$. ■

2.1.2 Algorithme d'Euclide

2.1.2.1 Principe et implémentation

L'**algorithme d'Euclide** calcule le pgcd de deux entiers naturels a et b selon le principe : si $b = 0$, alors $a \wedge b = a$, sinon $a \wedge b = b \wedge (a\%b)$, où $a\%b$ est le reste de la division euclidienne de a par b. Nous donnons une version récursive et une version itérative de cet algorithme.

Algorithme en Python 2.1.3 (Algorithme d'Euclide)

```
def PGCD_REC(a,b):
    if b==0:
        return a
    else:
        return PGCD_REC(b,a%b)
```

```
def PGCD(a,b):
    r,t=a,b
    while t!=0:
        r,t=t,r%t
    return r
```

Proposition 2.1.3

L'algorithme itératif se termine car la suite des valeurs successives prises par r est strictement décroissante à partir du rang 1 et à valeurs dans $\mathbb{N}$.

Démonstration :
Pour la $k-$ème boucle, on indexe les variables par k, et l'on note $u\%v$ le reste de la division euclidienne de u par v. Il s'agit de prouver, par récurrence, que pour tout k, on a $r_{k+1} < r_k$.
• Montrons que $r_1 < r_0$ ou $r_2 < r_1$. On a déjà $r_0 = a$, $t_0 = b$, $r_1 = t_0 = b$ et $t_1 = r_0\%t_0 = a\%b$. Distinguons alors deux cas.
— Si $b < a$, alors on a bien $r_1 < r_0$.
— Si $b \geq a$, alors $r_2 = t_1 = a\%b$, d'où $r_2 < b = r_1$.
• Supposons que $r_k < r_{k-1}$. On a déjà $t_k = r_{k-1}\%t_{k-1}$ et $r_k = t_{k-1}$. De plus, $t_{k+1} = r_k\%t_k$ et $r_{k+1} = t_k$, d'où $r_{k+1} = t_k = r_{k-1}\%t_{k-1} = r_{k-1}\%r_k < r_k$. ■

En vertu du principe de l'algorithme d'Euclide, on obtient immédiatement la proposition suivante.

Proposition 2.1.4

Un invariant de boucle de l'algorithme itératif est $\mathcal{P}(r)$: "à la fin de chaque boucle, $t \wedge r = a \wedge b$".

2.1.2.2 Majorant de la complexité de l'algorithme d'Euclide

Nous cherchons à majorer la complexité en nombre de divisions euclidiennes de l'algorithme d'Euclide. Notons déjà que ce nombre est le même pour les deux versions de l'algorithme.

Remarque 2.1.5

Si $a < b$, la première itération (ou le premier appel récursif) échange a et b. En outre, si $a = b$, une seule division donne le résultat. On peut donc supposer dans la suite que $1 \leq b < a$.

Dans les résultats qui suivent, nous utiliserons à plusieurs reprises la suite de Fibonacci [3]. Nous rappelons sa définition et l'expression de son terme général.

Définition 2.1.6

La **suite de Fibonacci** est la suite $(F_n)_{n\in\mathbb{N}}$ définie par $F_0 = F_1 = 1$ et pour tout $n \in \mathbb{N}$, $F_{n+2} = F_{n+1} + F_n$. Le $n-$ème terme de cette suite s'appelle le $n-$ème **nombre de Fibonacci**.

Proposition 2.1.7

Pour tout $n \in \mathbb{N}$, on a $F_n = \frac{5+\sqrt{5}}{10}\phi^n + \frac{5-\sqrt{5}}{10}\left(-\frac{1}{\phi}\right)^n$, où $\phi = \frac{1+\sqrt{5}}{2} \simeq 1.62$ est le **nombre d'or**.
En particulier, $F_n \sim \frac{5+\sqrt{5}}{10}\phi^n$.

Démonstration :
L'équation caractéristique de la suite est $r^2 - r - 1 = 0$. Ses racines sont $\phi = \frac{1+\sqrt{5}}{2}$ et $-\frac{1}{\phi}$. Alors il existe $(\lambda, \mu) \in \mathbb{R}^2$ tel que pour tout $n \in \mathbb{N}$, $F_n = \lambda\phi^n + \mu\left(-\frac{1}{\phi}\right)^n$. Et (λ, μ) est solution du système formé par les équations $\lambda+\mu = 1$ et $\lambda\phi-\frac{\mu}{\phi} = 1$, d'où $\lambda = \frac{5+\sqrt{5}}{10}$ et $\mu = \frac{5-\sqrt{5}}{10}$. ■

Proposition 2.1.8

Supposons que $1 \leq b < a$.
Si n est le nombre de divisions euclidiennes dans le calcul du pgcd de a et b, alors $F_{n+1} \leq a$ et $F_n \leq b$.

Démonstration :
Notons $d(a, b)$ le nombre de divisions euclidiennes pour calculer $a \wedge b$.
Posons alors $E_n = \{(a, b) \in \mathbb{N}^2 \,/\, 1 \leq b < a \text{ et } d(a, b) = n\}$.

3. Elle décrit la croissance de la "descendance" d'un couple de lapins.

Par récurrence sur $n \in \mathbb{N}$, montrons que $F_{n+1} \leq a$ et $F_n \leq b$.
• On a $F_1 = 1 \leq a$ et $F_0 = 1 \leq b$, donc le résultat est vrai au rang 0.
• Supposons que le résultat est vrai au rang $n-1$, et montrons qu'il est encore vrai au rang n. On a donc, pour tout $(a, b) \in E_{n-1}$, $F_n \leq a$ et $F_{n-1} \leq b$. Soit $(a, b) \in E_n$. Effectuons la division euclidienne de a par b : $a = bq + r$ avec $q \geq 1$ et $0 \leq r < b$. Alors $a \wedge b = b \wedge r$ et (b, r) appartient à E_{n-1}, donc par hypothèse de récurrence, on obtient $F_n \leq b$ et $F_{n-1} \leq r$. Alors

$$a = bq + r \geq b + r \geq F_n + F_{n-1} = F_{n+1},$$

ce qui montre que le résultat est vrai au rang n. ■

Lemme 2.1.9

Pour tout $n \in \mathbb{N}$, on a $F_n \geq \dfrac{\phi^n}{2}$.

Démonstration :
Procédons par récurrence sur $n \in \mathbb{N}$.
• On a $F_0 = 1 \geq \frac{1}{2}$ et $F_1 = 1 \geq \frac{\phi}{2}$ car $\phi \simeq 1.62$.
• Supposons que $F_{n-1} \geq \frac{\phi^{n-1}}{2}$ et que $F_n \geq \frac{\phi^n}{2}$. Alors

$$F_{n+1} = F_n + F_{n-1} \geq \frac{1}{2}\left(\phi^{n-1} + \phi^n\right) = \frac{1}{2}\phi^{n-1}\left(1 + \phi\right) = \frac{1}{2}\phi^{n-1}\phi^2 = \frac{1}{2}\phi^{n+1},$$

d'où le résultat. ■

Proposition 2.1.10 (Théorème de Lamé)

Soit n le nombre de divisions euclidiennes dans le calcul du pgcd de a et b, avec $1 \leq b < a$.
Alors $n \leq 2 + \frac{\ln b}{\ln \phi}$ et $n \leq 1 + \frac{\ln a}{\ln \phi}$, donc $n = O(\ln b)$.

Démonstration :
D'après la proposition 8, $a \geq F_{n+1}$. Or d'après le lemme 9, on a $F_{n+1} \geq \frac{\phi^{n+1}}{2}$, d'où $a \geq \frac{\phi^{n+1}}{2}$. Ainsi, $(n+1)\ln(\phi) \leq \ln(2a) = \ln(2) + \ln(a)$. Et $\phi^2 = 1 + \phi > 2$, donc $2\ln(\phi) \geq \ln(2)$. Alors $(n+1)\ln(\phi) \leq 2\ln(\phi) + \ln(a)$, i. e. $n\ln(\phi) \leq \ln(\phi) + \ln(a)$, d'où $n \leq 1 + \frac{\ln(a)}{\ln(\phi)}$.
De même, d'après la proposition 8 et le lemme 9, on a $b \geq F_n \geq \frac{\phi^n}{2}$ d'où $\phi^n \leq 2b$, donc $n\ln(\phi) \leq \ln(2) + \ln(b) \leq 2\ln(\phi) + \ln(b)$, d'où $n \leq 2 + \frac{\ln(b)}{\ln(\phi)}$. Finalement, $n = O(\ln(b))$. ■

Remarque 2.1.11

— Les majorants de la proposition 8 sont "atteints" dans le calcul du pgcd de F_{n+1} et F_n où tous les quotients des divisions sont "minimaux" puisqu'égaux à 1 sauf le dernier qui vaut 2.
— On a $\frac{1}{\ln \phi} < \frac{2}{\ln 2}$, donc $n < 2 + 2\log_2(b)$. Et l'on a vu que $\log_2(b)$ est du même ordre de grandeur que le nombre de chiffres de b en base 2.

2.1.2.3 Algorithme d'Euclide étendu

Etant donné deux entiers a et b, il existe d'après le théorème de Bézout deux entiers u et v tels que $au + bv = a \wedge b$. L'**algorithme d'Euclide étendu** calcule le pgcd de a et b ainsi qu'un couple (u, v) vérifiant cette relation.

✔ *Cet algorithme, très important, admet de nombreuses variantes et tout autant d'applications : calcul d'une suite de Sturm pour la recherche des racines réelles d'un polynôme, algorithme de Buchberger qui permet d'obtenir une base de Gröbner d'un idéal, ou encore algorithme de Sugiyama pour les codes correcteurs d'erreurs.*

Voici une implémentation en Python de l'algorithme d'Euclide étendu.

Algorithme en Python 2.1.4 (Algorithme d'Euclide étendu)

```
def EUCLIDE_ETENDU(a,b):
    r,u,v,r1,u1,v1=a,1,0,b,0,1
    while r1!=0:
        q=r//r1
        r,u,v,r1,u1,v1=r1,u1,v1,r-q*r1,u-q*u1,v-q*v1
    return r,u,v
```

Etant donné $n \in \mathbb{N}^*$ fixé, on dit qu'un entier $a \in \mathbb{Z}$ est **inversible modulo** n s'il existe un entier b (appelé inverse de a modulo n) tel que $ab \equiv 1 \mod n$. D'après le théorème de Bézout, un entier a est inversible modulo n si et seulement s'il est premier avec n. L'algorithme d'Euclide étendu donne un tel inverse u de a en calculant un couple (u, v) tel que $au + nv = 1$.

Exercice 2.1.1 (Inverse modulaire)
Ecrire une fonction `INVERSE_MODULAIRE(a,n)` qui reçoit deux entiers $a \in \mathbb{Z}$ et $n \in \mathbb{N}$ et qui renvoie 0 si $a = 0$ et l'unique inverse de a modulo n qui se trouve dans l'intervalle $[\![0, n-1]\!]$ si $a \neq 0$.
[Corrigé page 61]

2.1.3 Cryptographie

2.1.3.1 Le système RSA

Le système RSA est un système de cryptage à clé publique, inventé en 1977 par Ron Rivest, Adi Shamir et Leonard Adleman (leurs initiales ont donné RSA). C'est, avec le système AES, l'un des plus utilisés actuellement.

Ce principe repose sur le délicat problème de la factorisation [4] des grands entiers en nombres premiers. Pour le comprendre, prenons l'exemple [5] d'Amélie qui envoie à Baudouin le message "Vive les vacances!" en le cryptant. Elle va utiliser une **clé de cryptage** créée par Baudouin. Lui, utilisera sa **clé de décryptage**.

> Nous détaillons dans la suite les opérations de construction des clés, de cryptage et de décryptage, en simplifiant les étapes d'encodage-décodage du texte, qui ne concernent pas le principe même du cryptage.

2.1.3.2 Construction des clés

Baudouin fixe deux entiers premiers p et q distincts qu'il garde secrets, rend public leur produit $a = pq$ et pose $\omega = (p-1)(q-1)$ qu'il garde secret. Il fixe également un entier m, public, premier avec ω. Alors d'après le théorème de Bézout, m est inversible modulo ω et son inverse k, secret, se calcule grâce à l'algorithme d'Euclide étendu. Le couple $(a,\, m)$, représente la clé de cryptage de Baudouin. D'où le nom de système de cryptage à **clé publique**, puisque la clé de cryptage est connue de tous. Quant à la clé de décryptage de Baudouin, c'est le couple $(a,\, k)$, dans lequel k n'est connu que de Baudouin.

✔ *La robustesse du système RSA repose sur la difficulté à factoriser a en facteurs premiers. Il est donc difficile pour une tierce personne de retrouver p et q à partir de a, et donc difficile d'obtenir ω puis k pour connaître la clé de décryptage.*

Ici on choisit de coder le nombre a de la clé de cryptage sur 128 bits [6] : on choisit a de telle sorte qu'il appartienne à l'intervalle $[\![2^{127},\, 2^{128}-1]\!]$: il s'écrit donc en base 2 avec exactement 128 chiffres. Par exemple, si Baudouin choisit les deux nombres premiers $p = 18\,403\,224\,795\,191\,622\,251$ et $q = 18\,237\,210\,859\,617\,604\,733$, alors

$$\begin{aligned} a &= 335\,623\,491\,086\,852\,623\,037\,736\,525\,153\,165\,713\,983, \\ \omega &= 335\,623\,491\,086\,852\,623\,001\,096\,089\,498\,356\,487\,000. \end{aligned}$$

Ensuite il choisit [7] $m = 65\,537$, de sorte que la clé de cryptage de Baudouin est le

4. La factorisation des entiers est un problème difficile. Plusieurs algorithmes existent, comme l'algorithme de Pollard ou la méthode de factorisation par courbe elliptique de Lenstra, mais aucun n'est suffisamment efficace.

5. Dans la pratique, l'utilisation du RSA est souvent combinée à celle d'un algorithme de chiffrement symétrique à clé privée comme le système AES, le RSA servant au cryptage de la clé utilisée par AES, et AES cryptant le message.

6. On recommande actuellement d'utiliser des clés d'au moins 2048 bits, ou même 4096 bits.

7. Cette valeur est recommandée par la norme X509.

couple $(335\,623\,491\,086\,852\,623\,037\,736\,525\,153\,165\,713\,983,\ 65\,537)$.

L'inverse de m modulo ω est $k = 34\,045\,271\,659\,450\,329\,397\,306\,663\,457\,055\,921\,473$, nombre gardé secret, et la clé de décryptage de Baudouin est le couple $(a,\ k)$.

```
p=18403224795191622251; q=18237210859617604733
a=p*q; omega=(p-1)*(q-1); m=65537; np.gcd(m,omega); ⟶ 1
k=INVERSE_MODULAIRE(m,omega)
```

2.1.3.3 Cryptage du message

Pour crypter son message, Amélie procède en trois étapes.

— Elle convertit chaque caractère du texte en son nombre en unicode, puis en héxadécimal (i. e. base 16).

```
message='Vive les vacances !'
L1=[hex(ord(c))[2:] for c in message]
```

Elle complète aussi par un zéro les nombres à un seul chiffre.

```
L2=[a if len(a)==2 else '0'+a for a in L1]
    ⟶ ['56', '69', '76', '65', '20', '6c', '65', '73', '20',
   '76', '61', '63', '61', '6e', '63', '65', '73', '20', '21']
```

— Puis, au moyen de la fonction `conv120` ci-après, elle regroupe les nombres à deux chiffres en base 16 en des groupes de 15 octets (le message est ainsi découpé en blocs de 120 bits) qui sont ensuite convertis en base 10. En effet, les nombres qui s'écrivent avec deux chiffres en hexadécimal sont exactement les $2^8 = 256$ valeurs possibles d'un octet. Elle obtient ainsi une liste de nombres en base 10, tous strictement inférieurs à a.

```
L3=conv120(L2)
    ⟶ [448676563062738789820555299293064803, 1702043681]
```

Algorithme en Python 2.1.5

```
def conv120(L):
    J=[]
    for j in range(len(L)//15+1):
        M=L[j*15:(j+1)*15]; c=''
        for a in M:
            c+=a
        J.append(int(c,16))
    return J
```

— Elle élève enfin chaque élément de cette liste à la puissance m modulo a : cette troisième étape constitue véritablement le cryptage RSA.

Il est nécessaire pour calculer de telles puissances d'entiers de disposer d'une fonction d'exponentiation rapide modulo un entier. L'exercice qui suit propose de programmer une telle fonction.

Exercice 2.1.2 (Exponentiation rapide modulaire)
En s'inspirant de l'algorithme 2, écrire une fonction `EXP_MODULAIRE(a,n,p)` qui reçoit trois entiers $a \in \mathbb{Z}$, $n \in \mathbb{N}$ et $p \in \mathbb{N}^*$ et renvoie $a^n \mod p$ en utilisant la méthode d'exponentiation rapide.
[Corrigé page 62]

Cette nouvelle liste (notée L_4) d'entiers en base 10, tous strictement inférieurs à a, est le message crypté qu'Amélie envoie à Baudouin. La fonction `RSA` qui suit réalise cette opération ; notons que la même fonction servira au décryptage : il suffira de remplacer l'entier m de la clé de cryptage par l'entier k de la clé de décryptage.

Algorithme en Python 2.1.6 (RSA)

```
def RSA(a,m,L):
    return [EXP_MODULAIRE(x,m,a) for x in L]
```

```
L4=RSA(a,m,L3)
    ⟶ [157343380634634368572660122263439968078,
          200920919903455773640706528588837839151]
```

2.1.3.4 Décryptage du message

Théorème 2.1.1 (Petit théorème de Fermat)

Soient x un entier et p un nombre premier.
Si p ne divise pas x, alors $x^{p-1} \equiv 1 \mod p$.

Corollaire 2.1.12

Soient x, l deux entiers et p, q deux nombres premiers distincts.
Alors $x^{1+l(p-1)(q-1)} \equiv x \mod pq$.

Démonstration :
Distinguons tout d'abord deux cas.
— Si x n'est pas divisible par p, alors $x^{l(p-1)(q-1)} = \left(x^{l(q-1)}\right)^{p-1} \equiv 1 \mod p$ d'après le théorème 1.

— Si x est divisible par p, alors $x^{1+l(p-1)(q-1)} \equiv 0 \equiv x \mod p$.
Donc dans tous les cas, on a $x^{1+l(p-1)(q-1)} \equiv x \mod p$. De même, on a la relation $x^{1+l(p-1)(q-1)} \equiv x \mod q$. Ainsi, p et q divisent tous deux $x^{1+l(p-1)(q-1)} - x$. Comme ce sont deux nombres premiers distincts, leur produit pq divise aussi le nombre $x^{1+l(p-1)(q-1)} - x$, d'où le résultat. ■

La proposition suivante permet de décrypter le message : c'est le résultat-clé sur lequel est fondé le RSA !

Proposition 2.1.13

On a $(x^m)^k \equiv x \mod a$.

Démonstration :
Montrons que $(x^m)^k \equiv x \mod a$. On a par hypothèse $mk \equiv 1 \mod \omega$, i. e. il existe $l \in \mathbb{Z}$ tel que $mk - 1 = l(p-1)(q-1)$. Alors $(x^m)^k = x^{mk} = x^{1+l(p-1)(q-1)} \equiv x \mod a$ d'après le corollaire 12. ■

Quand Baudouin reçoit le message crypté (i.e. la liste L_4), il procède aussi en trois étapes pour le décoder.

— Au moyen de la fonction `RSA`, il élève tout d'abord chaque élément de cette liste à la puissance k modulo a, et d'après la proposition 13 il retrouve ainsi la liste L_3 : cette première étape constitue véritablement le décryptage RSA.
`L5=RSA(a,k,L4); L5==L3` ⟶ `True`

— Ensuite, il convertit chaque nombre de la liste en base 16 puis le sépare en groupes de deux chiffres grâce à la fonction `deconv120`.
`L6=deconv120(L5)`

Algorithme en Python 2.1.7

```
def deconv120(L):
    J=[]
    for a in L:
        h=hex(a)[2:]
        if len(h)%2==1:
            h='0'+h
        for j in range(len(h)//2):
            J.append(h[2*j:2*(j+1)])
    return J
```

— Il remplace enfin les nombres par les caractères correspondants.
`L7=[chr(int(a,16)) for a in L6]`

Et il reforme une chaîne de caractères.
`L8=''.join(L7)` $\longrightarrow$ `'Vive les vacances !'`, message toujours apprécié !

> Il faut faire attention si l'on envoie le même message à deux amis dont les clés de cryptage $(a_1,\, m_1)$ et $(a_2,\, m_2)$ sont telles que m_1 et m_2 sont premiers entre eux ! D'après le théorème de Bézout, il existe deux entiers u_1 et u_2 tels que $u_1 m_1 + u_2 m_2 = 1$, d'où $x = (x^{m_1})^{u_1} (x^{m_2})^{u_2}$, et le message peut facilement être décrypté par une tierce personne !

2.1.4 Evaluation d'un polynôme en un point

On donne maintenant des algorithmes pour évaluer un polynôme en un point. Soient $P = \sum_{k=0}^{n} a_k\, X^k \in \mathbb{K}[X]$ ($\mathbb{K}$ désigne $\mathbb{R}$ ou $\mathbb{C}$) et $x \in \mathbb{K}$. On souhaite calculer la valeur numérique $P(x) = \sum_{k=0}^{n} a_k\, x^k$.

2.1.4.1 Algorithme naïf

L'algorithme naïf consiste à effectuer la somme des monômes.

Algorithme 2.1.8 (Evaluation naïve)

```
EVALUATION_NAIVE(P, x) :
    z ← 1  (z contient les x^k successifs)
    y ← a_0
    Pour k de 1 à n :
        z ← z * x
        y ← y + a_k * z
    Renvoyer y
```

Proposition 2.1.14

La complexité de l'algorithme d'évaluation naïve est de n additions et $2\,n$ multiplications.

2.1.4.2 Algorithme de Horner

En écrivant $P(x)$ sous la forme $P(x) = \Big(\cdots \big((a_n\, x + a_{n-1})\, x + a_{n-2} \big)\, x + \cdots + a_1 \Big)\, x + a_0$ (pour P de degré 4, $P(x) = (((a_4 x + a_3) x + a_2) x + a_1) x + a_0)$), on en déduit l'algorithme suivant.

Algorithme 2.1.9 (Algorithme de Horner)

```
HORNER(P, x) :
    y ← a_n
    Pour k de n − 1 en descendant jusqu'à 0 :
        y ← y ∗ x + a_k
    Renvoyer y
```

Proposition 2.1.15

Un invariant de boucle est $\mathcal{P}(k)$: "à la sortie de la boucle d'indice k, on a $y = \sum_{i=k}^{n} a_i \, x^{i-k}$".

Proposition 2.1.16

La complexité de l'algorithme de Horner est de n additions et n multiplications.

2.2 Algorithmes de recherche dans un tableau trié

2.2.1 Recherche séquentielle

Pour chercher un élément x dans un tableau L trié, la méthode naïve (**recherche séquentielle**) consiste à parcourir le tableau. La fonction `CHERCHE_SEQU` renvoie -1 si x n'est pas dans L, ou le premier indice i tel que $L[i] = x$ sinon.

Algorithme en Python 2.2.10 (Recherche séquentielle)

```
def CHERCHE_SEQU(x,L):
    n=len(L); k=0
    while k<n and L[k]<x:
        k+=1
    if (k==0 and L[0]!=x) or k==n:
        return -1
    elif L[k]==x:
        return k
    else:
        return -1
```