17

MATHÉMATIQUES

Calcul et algorithmique

BTS | Mathématiques | Groupements D1, D2

Objectifs du chapitre

- Comprendre la notion d'algorithme et analyser un algorithme donné
- Maîtriser les variables, les types de données et l'affectation
- Utiliser les structures conditionnelles (SI…ALORS…SINON) et les boucles (POUR, TANT QUE)
- Définir et utiliser des fonctions et procédures ; aborder la récursivité
- Appliquer des algorithmes numériques : méthodes des rectangles, des trapèzes, Newton-Raphson, dichotomie
- Connaître les algorithmes de tri classiques et évaluer leur complexité
- Traduire des algorithmes en code Python et utiliser les bibliothèques `math` et `numpy`
- Mettre en œuvre un algorithme de régulation PID dans un contexte industriel

Situation professionnelle

Maxime est technicien supérieur en génie climatique dans une entreprise de maintenance industrielle. Son responsable lui demande de programmer un script d'automatisation pour calculer la consommation énergétique journalière d'une installation de chauffage à partir de relevés de température enregistrés toutes les 15 minutes. Maxime doit écrire un algorithme qui lit un tableau de mesures, calcule l'intégrale numérique de la puissance consommée et génère une alerte si la consommation dépasse un seuil. Il s'appuie sur ses connaissances en algorithmique et en Python pour automatiser entièrement ce traitement.

1. Notions de base d'algorithmique

Définition Un **algorithme** est une suite finie, non ambiguë et ordonnée d'instructions permettant de résoudre un problème ou d'effectuer un calcul. Il prend des *entrées*, effectue un *traitement* et produit des *sorties*. Un algorithme doit toujours se terminer en un nombre fini d'étapes.

1.1 Variables et types de données

Une **variable** est un emplacement mémoire identifié par un nom, contenant une valeur susceptible d'évoluer au cours de l'exécution de l'algorithme.

Type	Description	Exemples de valeurs
Entier	Nombre entier relatif	−5, 0, 42, 1000
Réel	Nombre décimal (flottant)	3,14 ; −0,5 ; 2,718
Booléen	Valeur logique	VRAI, FAUX
Chaîne	Suite de caractères	« Bonjour », « température »
Tableau	Collection indexée de valeurs de même type	T[1..10] de réels

1.2 Affectation

Définition L'**affectation** attribue une valeur à une variable. Notation : `variable ← expression`. La valeur calculée à droite du symbole ← est stockée dans la variable à gauche. L'ancienne valeur de la variable est définitivement écrasée.

Exemples d'affectations

```
x ← 5
y ← 3.14
z ← x + y          // z prend la valeur 8,14
compteur ← 0
compteur ← compteur + 1  // incrémentation : compteur vaut 1
nom ← "Alice"
valide ← VRAI
```

Attention L'affectation n'est **pas** une équation mathématique. L'instruction `x ← x + 1` est parfaitement valide : elle lit la valeur actuelle de `x`, lui ajoute 1 et range le résultat dans `x`. En mathématiques, l'équation $x = x + 1$ n'a pas de solution.

1.3 Entrées et sorties

Un algorithme interagit avec l'utilisateur (ou des fichiers) via deux instructions :

- `Saisir(variable)` — lit une valeur fournie et la stocke dans la variable
- `Afficher(expression)` — affiche la valeur de l'expression à l'écran

Exemple complet : conversion °C → °F

```
Algorithme ConversionTemperature
Variables :
    celsius, fahrenheit : réel
Début
    Afficher("Entrez la température en °C : ")
    Saisir(celsius)
    fahrenheit ← celsius * 9 / 5 + 32
    Afficher("Température en °F : ", fahrenheit)
Fin
```

Test avec `celsius = 100` : $f = 100 \times \frac{9}{5} + 32 = 212\,^{\circ}$F. Correct.

2. Structures de contrôle

2.1 Structure conditionnelle : SI ... ALORS ... SINON ... FIN SI

Syntaxe

```
SI condition ALORS
    instructions si VRAI
SINON
    instructions si FAUX
FIN SI
```

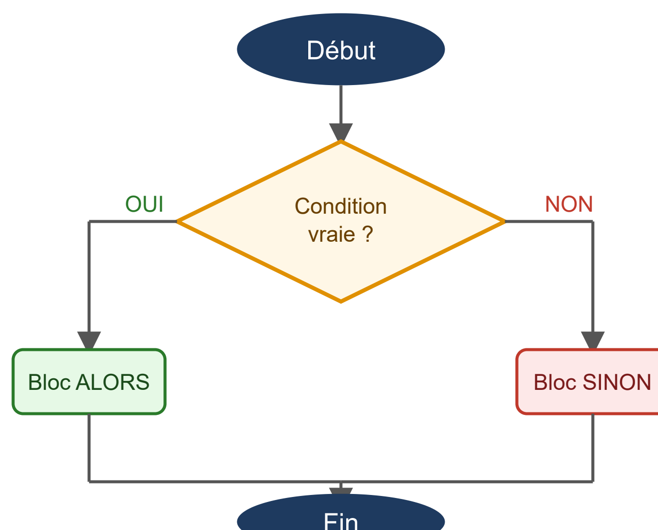
La clause `SINON` est facultative. On peut chaîner des tests avec `SINON SI`.

Exemple : classification d'une pièce mécanique

Un technicien métrologue mesure le diamètre d'une pièce. La cote nominale est 50 mm avec une tolérance de ±0,05 mm.

```
Algorithme ControleQualite
Variables :
    diametre : réel
Début
    Afficher("Diamètre mesuré (mm) : ")
    Saisir(diametre)
    SI (diametre >= 49.95) ET (diametre <= 50.05) ALORS
        Afficher("Pièce ACCEPTÉE")
    SINON SI diametre < 49.95 ALORS
        Afficher("Pièce REJETÉE — sous-dimensionnée")
    SINON
        Afficher("Pièce REJETÉE — sur-dimensionnée")
    FIN SI
Fin
```

Organigramme de la structure SI…ALORS…SINON



2.2 Boucle POUR (nombre d'itérations connu)

Syntaxe

```
POUR variable DE valeur_debut A valeur_fin [PAS DE pas] FAIRE
    instructions
FIN POUR
```

La variable prend successivement chaque valeur de *valeur_debut* à *valeur_fin* avec l'incrément *pas* (1 par défaut).

Exemple : somme des N premiers entiers

```
Algorithme SommeEntiers
Variables :
    n, i : entier
    somme : entier
Début
    Saisir(n)
    somme ← 0
    POUR i DE 1 A n FAIRE
        somme ← somme + i
    FIN POUR
    Afficher("Somme = ", somme)
    // Vérification : formule close n(n+1)/2
Fin
```

Pour $n = 5 : 1 + 2 + 3 + 4 + 5 = 15$. La formule donne $\frac{5 \times 6}{2} = 15$. ✓

Exemple : calcul de la moyenne d'un tableau de mesures

```
Algorithme MoyenneMesures
Variables :
    n, i : entier
    T : tableau de réels
    somme, moyenne : réel
Début
    Saisir(n)
    POUR i DE 1 A n FAIRE
        Afficher("Mesure ", i, " : ")
        Saisir(T[i])
    FIN POUR
    somme ← 0
    POUR i DE 1 A n FAIRE
        somme ← somme + T[i]
    FIN POUR
    moyenne ← somme / n
    Afficher("Moyenne = ", moyenne)
Fin
```

2.3 Boucle TANT QUE (condition d'arrêt)

Syntaxe

```
TANT QUE condition FAIRE
    instructions
FIN TANT QUE
```

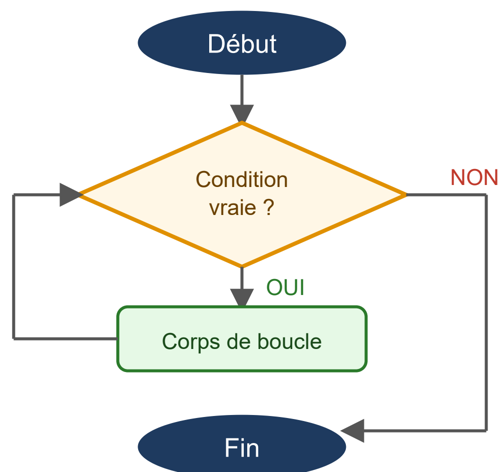
La boucle s'exécute **tant que** la condition est vraie. Si la condition est fausse dès le départ, le corps n'est jamais exécuté.

Attention Il faut s'assurer que la condition devient un jour fausse, sinon on obtient une **boucle infinie** qui bloque le programme. Vérifier qu'une variable du test est bien modifiée dans le corps.

Exemple : saisie valide d'une pression

```
Algorithme SaisiePression
Variables :
    pression : réel
Début
    pression ← -1
    TANT QUE pression < 0 FAIRE
        Afficher("Pression (Pa, valeur > 0) : ")
        Saisir(pression)
        SI pression < 0 ALORS
            Afficher("Erreur : la pression doit être positive.")
        FIN SI
    FIN TANT QUE
    Afficher("Pression enregistrée : ", pression, " Pa")
Fin
```

Organigramme de la boucle TANT QUE



3. Fonctions et procédures

Définitions

- **Procédure** : bloc d'instructions nommé réutilisable, ne renvoyant aucune valeur.
- **Fonction** : bloc d'instructions nommé renvoyant une valeur via l'instruction `RETOURNER`.
- **Paramètres** : valeurs transmises à la fonction lors de l'appel.
- L'**appel** d'une fonction interrompt le programme principal, exécute la fonction, puis reprend après l'appel.

Exemple : fonction factorielle (itérative)

```
Fonction Factorielle(n : entier) : entier
Variables :
    i, resultat : entier
Début
    resultat ← 1
    POUR i DE 2 A n FAIRE
        resultat ← resultat * i
    FIN POUR
    RETOURNER resultat
Fin

// Programme principal :
k ← Factorielle(6)     // k = 6! = 720
Afficher(k)             // affiche 720
```

3.1 Récursivité

Définition Une fonction est **récursive** si elle s'appelle elle-même dans sa définition.

Toute fonction récursive doit posséder :

- Un **cas de base** : condition d'arrêt qui ne fait pas d'appel récursif.
- Un **cas récursif** : ramène le problème à une instance strictement plus petite.

Factorielle récursive

```
Fonction FactoriellRec(n : entier) : entier
Début
    SI n <= 1 ALORS
        RETOURNER 1                          // cas de base
    SINON
        RETOURNER n * FactoriellRec(n - 1)  // cas récursif
    FIN SI
Fin
```

Déroulement de `FactoriellRec(4)` :

```
FactoriellRec(4) = 4 × FactoriellRec(3)
                = 4 × 3 × FactoriellRec(2)
                = 4 × 3 × 2 × FactoriellRec(1)
                = 4 × 3 × 2 × 1 = 24
```

Suite de Fibonacci récursive

Définition : $F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$.

```
Fonction Fibonacci(n : entier) : entier
Début
    SI n = 0 ALORS RETOURNER 0
    SINON SI n = 1 ALORS RETOURNER 1
    SINON RETOURNER Fibonacci(n-1) + Fibonacci(n-2)
    FIN SI
Fin
```

Premières valeurs : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

Attention — inefficacité de la récursivité naïve La version récursive naïve de Fibonacci calcule plusieurs fois les mêmes valeurs. `Fibonacci(30)` effectue plus d'un million d'appels ! On préfèrera une version itérative pour les grandes valeurs de n.

4. Algorithmes numériques

4.1 Intégration numérique — méthode des rectangles

On veut approcher l'intégrale définie $I = \int_a^b f(x)\,dx$ lorsqu'on ne dispose pas d'une primitive analytique.

Méthode des rectangles à gauche On divise $[a, b]$ en n sous-intervalles de largeur $h = \frac{b-a}{n}$. Sur chaque sous-intervalle $[x_k, x_{k+1}]$, on approche f par la constante $f(x_k)$.

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

L'erreur est de l'ordre de $O(h)$ (converge lentement).

Pseudo-code : méthode des rectangles

```
Fonction IntegraleRectangles(a, b : réel ; n : entier) : réel
Variables :
    h, somme, x : réel
    k : entier
Début
    h ← (b - a) / n
    somme ← 0
    POUR k DE 0 A n-1 FAIRE
        x ← a + k * h
        somme ← somme + f(x)      // f est une fonction définie par a
    FIN POUR
    RETOURNER h * somme
Fin
```

4.2 Méthode des trapèzes

Formule des trapèzes On approche f par un segment de droite sur chaque sous-intervalle.

$$I \approx \frac{h}{2}\left[f(a) + 2\sum_{k=1}^{n-1} f(a+kh) + f(b)\right]$$

L'erreur est en $O(h^2)$: deux fois plus d'intervalles divise l'erreur par 4.

Pseudo-code : méthode des trapèzes

```
Fonction IntegraleTrapèzes(a, b : réel ; n : entier) : réel
Variables :
    h, somme, x : réel
    k : entier
Début
    h ← (b - a) / n
    somme ← f(a) + f(b)
    POUR k DE 1 A n-1 FAIRE
        x ← a + k * h
        somme ← somme + 2 * f(x)
    FIN POUR
    RETOURNER (h / 2) * somme
Fin
```

4.3 Méthode de Newton-Raphson

On cherche une racine de $f(x) = 0$ en itérant à partir d'une approximation initiale x_0.

Formule d'itération de Newton-Raphson

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

La tangente à la courbe de f en x_n coupe l'axe des abscisses en x_{n+1}. On arrête quand $|x_{n+1} - x_n| < \varepsilon$.

Pseudo-code : Newton-Raphson

```
Algorithme Newton
Variables :
    x0, x1, epsilon : réel
Début
    Saisir(x0)
    epsilon ← 1e-8
    x1 ← x0 - f(x0) / f_prime(x0)
    TANT QUE |x1 - x0| >= epsilon FAIRE
        x0 ← x1
        x1 ← x0 - f(x0) / f_prime(x0)
    FIN TANT QUE
    Afficher("Racine approchée : ", x1)
Fin
```

Exemple numérique : résoudre $x^2 - 2 = 0$ (trouver $\sqrt{2}$)

$f(x) = x^2 - 2$, $f'(x) = 2x$, $x_0 = 1{,}5$.

It. n	x_n	$f(x_n)$	$\lvert x_{n+1} - x_n \rvert$
0	1,500 000 000	0,250 000	0,083 333
1	1,416 666 667	0,006 944	0,002 451
2	1,414 215 686	0,000 006	0,000 002
3	1,414 213 562	$< 10^{-12}$	$< 10^{-8}$

On retrouve $\sqrt{2} \approx 1{,}414\,213\,562$.

4.4 Méthode de dichotomie

Méthode de dichotomie **Hypothèses :** f continue sur $[a, b]$ et $f(a) \cdot f(b) < 0$. Il existe alors au moins une racine dans $[a, b]$ (théorème des valeurs intermédiaires).

Principe : on calcule le milieu $m = \frac{a+b}{2}$, puis on remplace a ou b par m selon le signe de $f(m)$, réduisant l'intervalle de moitié à chaque étape. On s'arrête quand $\frac{b-a}{2} < \varepsilon$.

Pseudo-code : dichotomie

```
Algorithme Dichotomie
Variables :
    a, b, m, epsilon : réel
Début
    Saisir(a, b, epsilon)
    TANT QUE (b - a) / 2 >= epsilon FAIRE
        m ← (a + b) / 2
        SI f(m) = 0 ALORS
            a ← m  ;  b ← m   // racine exacte trouvée
        SINON SI f(a) * f(m) < 0 ALORS
            b ← m              // racine dans [a, m]
        SINON
            a ← m              // racine dans [m, b]
        FIN SI
    FIN TANT QUE
    Afficher("Racine ≈ ", (a + b) / 2)
Fin
```

Convergence de la dichotomie Après n itérations, l'erreur est majorée par $\frac{b-a}{2^{n+1}}$. Le nombre d'itérations nécessaires pour atteindre la précision ε est :

$$n \geq \frac{\ln\left(\frac{b-a}{\varepsilon}\right)}{\ln 2} = \log_2\left(\frac{b-a}{\varepsilon}\right)$$

Exemple : pour $[0, 2]$ et $\varepsilon = 10^{-6}$, il faut $n \geq \log_2(2 \times 10^6) \approx 21$ itérations.

5. Algorithmes de tri

5.1 Tri à bulles

Principe On parcourt le tableau en comparant les éléments adjacents et en les échangeant si nécessaire. Après chaque passage, le plus grand élément non trié « remonte » en fin de tableau. On répète jusqu'à ce qu'aucun échange n'ait lieu.

Pseudo-code : tri à bulles optimisé

```
Procédure TriBulles(T : tableau d'entiers ; n : entier)
Variables :
    i, j, temp : entier
    echange : booléen
Début
    POUR i DE 1 A n-1 FAIRE
        echange ← FAUX
        POUR j DE 1 A n-i FAIRE
            SI T[j] > T[j+1] ALORS
                temp   ← T[j]
                T[j]   ← T[j+1]
                T[j+1] ← temp
                echange ← VRAI
            FIN SI
        FIN POUR
        SI NON echange ALORS
            Sortir de la boucle   // tableau déjà trié
        FIN SI
    FIN POUR
Fin
```

Trace du tri à bulles sur [5, 2, 8, 1, 4]

Passage	Tableau après passage	Nb échanges
Initial	[5, 2, 8, 1, 4]	—
1	[2, 5, 1, 4, **8**]	3
2	[2, 1, 4, **5**, **8**]	2
3	[1, 2, **4**, **5**, **8**]	1
4	[**1**, **2**, **4**, **5**, **8**]	0 → arrêt

5.2 Tri par sélection

Principe À chaque étape i, on cherche le minimum dans la partie non triée $T[i..n]$ et on l'échange avec $T[i]$. La partie triée $T[1..i-1]$ croît d'un élément à chaque étape.

Pseudo-code : tri par sélection

```
Procédure TriSelection(T : tableau d'entiers ; n : entier)
Variables :
    i, j, i_min, temp : entier
Début
    POUR i DE 1 A n-1 FAIRE
        i_min ← i
        POUR j DE i+1 A n FAIRE
            SI T[j] < T[i_min] ALORS
                i_min ← j
            FIN SI
        FIN POUR
        SI i_min ≠ i ALORS
            temp      ← T[i]
            T[i]      ← T[i_min]
            T[i_min]  ← temp
        FIN SI
    FIN POUR
Fin
```

6. Complexité algorithmique

Définition La **complexité temporelle** mesure le nombre d'opérations élémentaires en fonction de la taille n des données. La notation *grand O* indique l'ordre de grandeur dans le pire cas. Elle permet de prévoir le comportement d'un algorithme à grande échelle.

O(1)

Complexité constante

Accès à T[k] par indice

O(log n)

Logarithmique

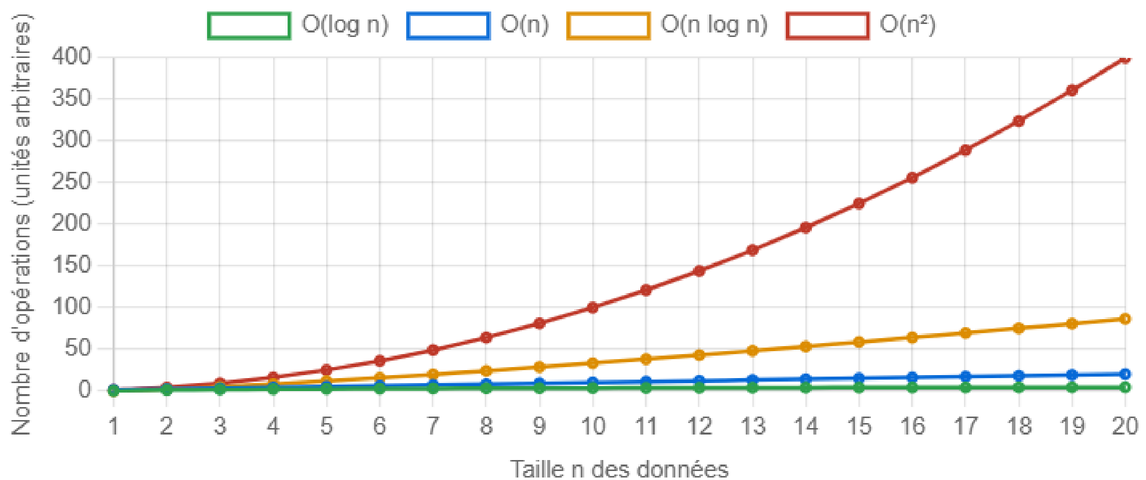
Dichotomie, recherche binaire

O(n)

Linéaire

Parcours de tableau, recherche séquentielle

$O(n \log n)$	**$O(n^2)$**	**$O(2^n)$**
Quasi-linéaire	Quadratique	Exponentielle
Tri rapide, tri fusion (en moyenne)	Tri à bulles, tri par sélection	Fibonacci récursif naïf



Complexité des tris classiques

Algorithme	Meilleur cas	Cas moyen	Pire cas	Mémoire
Tri à bulles	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Tri par sélection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Tri par insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Tri rapide (quicksort)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Tri fusion (mergesort)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

7. Programmation Python

Python est un langage interprété, à typage dynamique, très utilisé en sciences appliquées et en automatisation industrielle. La correspondance entre pseudo-code et Python est quasi directe.

7.1 Variables et opérateurs

Affectation, types, affichage

```
# Python détecte automatiquement le type
temperature = 20.5        # flottant (réel)
compteur = 0              # entier
nom = "Maxime"            # chaîne de caractères
actif = True              # booléen (True/False)

# Affichage
print("Température :", temperature, "°C")
print(f"Bonjour {nom}, compteur = {compteur}")  # f-string

# Saisie utilisateur
valeur = float(input("Entrez une valeur : "))
```

7.2 Structure conditionnelle en Python

Contrôle qualité d'une pièce (traduction du pseudo-code)

```
diametre = float(input("Diamètre mesuré (mm) : "))

if 49.95 <= diametre <= 50.05:
    print("Pièce ACCEPTÉE")
elif diametre < 49.95:
    print("Pièce REJETÉE — sous-dimensionnée")
else:
    print("Pièce REJETÉE — sur-dimensionnée")
```

7.3 Boucles en Python

Boucle for — somme des entiers

```
n = int(input("Entrez n : "))
somme = 0
for i in range(1, n + 1):   # range(1, n+1) génère 1, 2, ..., n
    somme += i
print(f"Somme des {n} premiers entiers = {somme}")
print(f"Vérification formule n(n+1)/2 = {n*(n+1)//2}")
```

Boucle while — saisie valide

```
pression = -1
while pression < 0:
    try:
        pression = float(input("Pression (Pa, > 0) : "))
    except ValueError:
        print("Veuillez entrer un nombre.")
    if pression < 0:
        print("Erreur : valeur négative.")
print(f"Pression enregistrée : {pression} Pa")
```

7.4 Fonctions en Python

Factorielle et Fibonacci

```
def factorielle(n):
    """Calcule n! de façon itérative."""
    resultat = 1
    for i in range(2, n + 1):
        resultat *= i
    return resultat

def fibonacci(n):
    """Retourne F(n) de façon itérative (efficace)."""
    if n == 0:
        return 0
    a, b = 0, 1
    for _ in range(1, n):
        a, b = b, a + b
    return b

# Tests
for k in range(8):
    print(f"F({k}) = {fibonacci(k)}")
print(f"10! = {factorielle(10)}")
```

Méthode des trapèzes en Python

```
import math

def integrale_trapèzes(f, a, b, n):
    """Intègre f sur [a,b] par la méthode des trapèzes avec n sous-i
    h = (b - a) / n
    somme = f(a) + f(b)
    for k in range(1, n):
        somme += 2 * f(a + k * h)
    return (h / 2) * somme

# Exemple : ∫₀^π sin(x)dx = 2 (exact)
result = integrale_trapèzes(math.sin, 0, math.pi, 1000)
print(f"∫₀^π sin(x)dx ≈ {result:.8f}  (exact = 2)")
# Affiche : ∫₀^π sin(x)dx ≈ 1.99999836  (exact = 2)
```

Méthode de Newton-Raphson en Python

```
def newton_raphson(f, df, x0, epsilon=1e-8, max_iter=100):
    """
    Résout f(x) = 0 par Newton-Raphson.
    f       : fonction dont on cherche la racine
    df      : dérivée de f
    x0      : approximation initiale
    epsilon : précision souhaitée
    """
    x = x0
    for i in range(max_iter):
        fx = f(x)
        if abs(fx) < epsilon:
            print(f"Convergé en {i} itérations")
            return x
        dfx = df(x)
        if abs(dfx) < 1e-14:
            raise ValueError("Dérivée nulle — méthode ne converge pa
        x = x - fx / dfx
    print("Avertissement : nombre max d'itérations atteint")
    return x

# Calcul de sqrt(2) : résoudre x² - 2 = 0
f  = lambda x: x**2 - 2
df = lambda x: 2*x
racine = newton_raphson(f, df, x0=1.5)
print(f"√2 ≈ {racine:.10f}")
```

7.5 Bibliothèque math

Fonctions mathématiques courantes

```
import math

print(math.pi)               # 3.141592653589793
print(math.e)                # 2.718281828459045
print(math.sqrt(144))        # 12.0
print(math.log(100, 10))     # 2.0  — log base 10
print(math.log(math.e))      # 1.0  — log naturel
print(math.sin(math.pi/6))   # 0.5  — sin(30°)
print(math.cos(math.pi/3))   # 0.5  — cos(60°)
print(math.factorial(6))     # 720
print(math.gcd(12, 8))       # 4    — PGCD
print(math.ceil(3.2))        # 4    — arrondi supérieur
print(math.floor(3.9))       # 3    — arrondi inférieur
```

7.6 Introduction à NumPy

Tableaux NumPy et calcul vectoriel

```
import numpy as np

# Création de tableaux
temperatures = np.array([18.5, 20.1, 22.3, 19.8, 21.0])
print("Tableau :", temperatures)
print("Moyenne :", np.mean(temperatures))          # 20.34
print("Écart-type :", np.std(temperatures))        # 1.27
print("Maximum :", np.max(temperatures))           # 22.3

# Génération de points
t = np.linspace(0, 24, 97)    # 96 intervalles de 15 min sur 24h
puissance = 2.5 * np.sin(np.pi * t / 12) + 3.0  # kW — modèle sinuso

# Intégrale numérique avec np.trapz
energie_kWh = np.trapz(puissance, t)
print(f"Énergie consommée : {energie_kWh:.2f} kWh")

# Opérations matricielles
A = np.array([[2, 1], [1, 3]])
b = np.array([5, 10])
x = np.linalg.solve(A, b)   # résolution du système Ax = b
print("Solution :", x)
```

8. Application industrielle : régulateur PID numérique

Contexte industriel

Un régulateur PID (Proportionnel-Intégral-Dérivé) est utilisé pour maintenir une grandeur physique (température, pression, débit) à une valeur consigne malgré les perturbations. On l'implante numériquement en calculant la commande toutes les T_e secondes (période d'échantillonnage).

Loi de commande PID discrète À l'instant k, l'erreur est $e_k = \text{consigne} - \text{mesure}_k$.

$$u_k = \underbrace{K_p\, e_k}_{\text{proportionnel}} + \underbrace{K_i\, T_e \sum_{j=0}^{k} e_j}_{\text{intégral}} + \underbrace{K_d\, \frac{e_k - e_{k-1}}{T_e}}_{\text{dérivé}}$$

La commande u_k est saturée dans l'intervalle $[u_{\min},\, u_{\max}]$.

Pseudo-code : algorithme PID numérique

```
Algorithme RegulationPID
Variables :
    Kp, Ki, Kd, Te : réels
    consigne, mesure, e, e_prec, somme_e, u : réels
Début
    Kp ← 2.0 ; Ki ← 0.5 ; Kd ← 0.1 ; Te ← 0.1
    somme_e ← 0 ; e_prec ← 0

    TANT QUE VRAI FAIRE
        LireConsigne(consigne)
        LireCapteur(mesure)
        e ← consigne - mesure
        somme_e ← somme_e + e * Te
        u ← Kp * e + Ki * somme_e + Kd * (e - e_prec) / Te
        u ← MAX(0, MIN(100, u))       // saturation 0-100 %
        AppliquerCommande(u)
        e_prec ← e
        Attendre(Te)
    FIN TANT QUE
Fin
```

Classe PID en Python avec simulation

```
class PID:
    """Régulateur PID numérique à pas fixe Te."""

    def __init__(self, Kp, Ki, Kd, Te, u_min=0, u_max=100):
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd
        self.Te = Te
        self.u_min = u_min
        self.u_max = u_max
        self._somme_e = 0.0
        self._e_prec  = 0.0

    def calculer(self, consigne, mesure):
        """Calcule la commande u pour un instant d'échantillonnage."
        e = consigne - mesure
        self._somme_e += e * self.Te
        derivee = (e - self._e_prec) / self.Te
        u = self.Kp * e + self.Ki * self._somme_e + self.Kd * derive
        u = max(self.u_min, min(self.u_max, u))
        self._e_prec = e
        return u

    def reset(self):
        """Réinitialise l'état interne (ex. : changement de consigne
        self._somme_e = 0.0
        self._e_prec  = 0.0


# --- Simulation de la régulation de température ---
pid = PID(Kp=2.0, Ki=0.5, Kd=0.1, Te=0.1)
consigne    = 60.0     # °C cible
temperature = 20.0     # °C initiale (ambiance)
```

```
print(f"{'t(s)':>5} | {'Mesure °C':>9} | {'Erreur':>7} | {'Commande 
print("-" * 40)
for k in range(25):
    u = pid.calculer(consigne, temperature)
    # Modèle thermique du premier ordre simplifié
    temperature += u * 0.05 - 0.02 * (temperature - 20.0)
    t = k * 0.1
    print(f"{t:>5.1f} | {temperature:>9.2f} | {consigne-temperature:
```

Méthode

Écrire et lire un algorithme — démarche systématique

1. **Analyser le problème** : identifier clairement les entrées, les sorties et les contraintes.
2. **Déclarer les variables** : choisir des noms significatifs et préciser le type.
3. **Décomposer en sous-problèmes** : définir des fonctions pour les traitements répétitifs.
4. **Choisir la bonne structure** : SI pour un choix, POUR quand le nombre d'itérations est connu, TANT QUE sinon.
5. **Vérifier les cas limites** : $n = 0$, liste vide, dénominteur nul, division par zéro…
6. **Tracer à la main** : suivre l'exécution sur un exemple simple en notant chaque valeur de variable.
7. **Estimer la complexité** : l'algorithme est-il utilisable pour de grands n ?
8. **Traduire en Python** : respecter l'indentation (4 espaces), commenter, tester.

À retenir — Calcul et algorithmique

- **Affectation** : `x ← expression` (pseudo-code), `x = expression` (Python)
- **SI/ALORS/SINON** ⇔ `if / elif / else`
- **POUR i DE a À b** ⇔ `for i in range(a, b+1)`
- **TANT QUE cond** ⇔ `while cond`
- **Méthode des trapèzes** : $I \approx \frac{h}{2}\left[f(a) + 2\sum_{k=1}^{n-1} f(a+kh) + f(b)\right]$, erreur $O(h^2)$
- **Newton-Raphson** : $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$, convergence quadratique
- **Dichotomie** : $n \geq \log_2\left(\frac{b-a}{\varepsilon}\right)$ itérations pour la précision ε
- **Tri à bulles / sélection** : complexité $O(n^2)$ — éviter pour $n > 10^4$
- **Hiérarchie** : $O(1) \ll O(\log n) \ll O(n) \ll O(n \log n) \ll O(n^2) \ll O(2^n)$
- **PID discret** : $u_k = K_p e_k + K_i T_e \sum e_j + K_d \frac{e_k - e_{k-1}}{T_e}$

17

MATHÉMATIQUES

Calcul et algorithmique

BTS | Mathématiques | Durée : 40 min | /20

Nom : _____ Prénom : _____ Date : _____

Exercice 1 — Affectation et trace (4 pts)

On exécute l'algorithme suivant :

```
a ← 3
b ← 5
a ← a + b
b ← a - b
Afficher(a, b)
```

a. Donner la valeur de a puis de b après chaque affectation. *(3 pts)*

b. Que vaut le couple affiché (a, b) ? *(1 pt)*

Exercice 2 — Boucle et somme (3 pts)

On considère l'algorithme :

```
somme ← 0
POUR i DE 1 A 4 FAIRE
    somme ← somme + i * i
FIN POUR
Afficher(somme)
```

Donner la valeur affichée, en détaillant le calcul. *(3 pts)*

Exercice 3 — Récursivité (4 pts)

On rappelle la factorielle récursive : $\text{Fact}(n) = n \times \text{Fact}(n-1)$ avec $\text{Fact}(0) = 1$.

a. Dérouler le calcul de $\text{Fact}(4)$ et donner le résultat. *(2 pts)*

b. La suite de Fibonacci vérifie $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$. Donner F_2, F_3, F_4, F_5, F_6. *(2 pts)*

Exercice 4 — Newton-Raphson (5 pts)

On cherche $\sqrt{2}$ en résolvant $f(x) = x^2 - 2 = 0$ par Newton-Raphson :
$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$, avec $f'(x) = 2x$ et $x_0 = 1$.

a. Montrer que la formule se simplifie en $x_{n+1} = \frac{x_n}{2} + \frac{1}{x_n}$. *(2 pts)*
b. Calculer x_1 puis x_2 (valeur exacte sous forme de fraction, puis valeur décimale). *(3 pts)*

Exercice 5 — Complexité (4 pts)

a. Classer du plus rapide au plus lent (pour n grand) : $O(n^2)$, $O(\log n)$, $O(n)$, $O(n \log n)$. *(2 pts)*
b. Donner la complexité dans le pire cas du tri à bulles, et celle de la recherche par dichotomie. *(2 pts)*