**Remarque :**

- Tous les exercices doivent être programmés sous forme de fonction.
- Ensuite, dans le code principal (ou même dans la console), vous faites appel à votre fonction en lui passant les valeurs en paramètres. Plus besoin de **scan()** donc.

## 1. PGCD - Plus grand commun diviseur

**Entrée** : a, b (entiers > 0)

**Sortie** : PGCD

**Calcul** : (1) Calculer la valeur absolue de l'écart entre a et b (cf. ABS(.))

(2) Affecter le résultat au plus grand des deux

(3) Continuer ainsi jusqu'à ce que a == b

(4) Le PGCD = a (ou b qu'importe)

Voici la signature de la fonction : `pgcd <- function(a,b)`

## 2. Calcul amplitude d'intervalles – Intervalles de largeurs égales

Ecrire une fonction qui prend en entrée a, b et k.

Si a > b ou k = 0 → la fonction renvoie la valeur NA

Autrement, elle doit renvoyer (b-a)/k

Voici l'en-tête de la fonction : `intervalle <- function (a,b,k=3)`

## 3. Simulation

On veut placer son capital 100.000 euros sur un produit financier sur 3 ans. L'évolution annuelle est (+10.000 ; 0 ; -5.000) avec les probabilités (0.2 ; 0.5 ; 0.3). Quelle sera la valeur de votre capital au bout de la 3ème année ? A un horizon de 1 an, le calcul est facile. A horizon de 3 ans, ça devient compliqué. On veut calculer le capital final par simulation.

Ecrire une application qui renvoie par simulation la valeur de capital obtenue au bout de 3 ans. Appeler alors 1000 fois cette fonction et calculer la moyenne des valeurs obtenues. Voir dans l'aide l'utilisation de la fonction **runif** qui génère un nombre aléatoire suivant une loi uniforme.

Vous aurez à écrire deux fonctions :

- `simulation <- function()` se charge de réaliser une simulation
- `allsimulation <- function(n = 1000)` se charge de répéter n fois la simulation et d'en déduire la moyenne.

## 4. Fonction de Répartition de la loi de Poisson

Ecrire une fonction prenant en entrée x et lambda, elle doit renvoyer la valeur de la fonction de répartition de la loi de Poisson.

Pour rappel, la formule s'écrit : $$P(X \leq x) = \sum_{i=0}^{x} e^{-\lambda} \frac{\lambda^i}{i!}$$

Voici l'en-tête de la fonction : `fpoisson <- function(x,lambda)`

Pour vérifier votre fonction, voyez du côté de la fonction >> **ppois** << de R.

## 5. Aire entre deux bornes de la fonction de densité de la loi normale

On souhaite calculer la surface située entre 2 bornes d'une fonction continue.

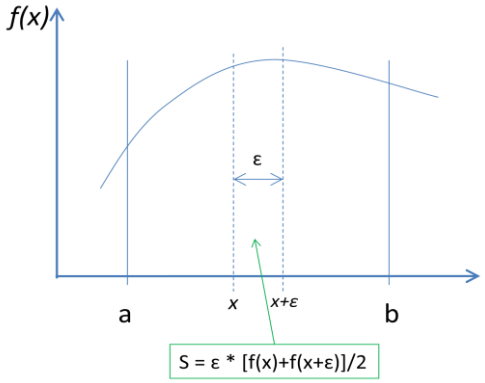
**Entrée** : a et b (réels)

**Sortie** : Surface (réel)

**Stratégie** : L'idée consiste à subdiviser l'espace allant de « a » à « b » en une série de trapèzes de largeur ε (paramètre de l'algorithme). On sait que la surface du trapèze est égale à S = ε * [f(x) + f(x + ε)]/2.

Il suffit alors d'additionner les surfaces de trapèzes pour obtenir la surface totale.

Schématiquement, nous aurions ceci :




Dans notre cas, il s'agit dans notre exemple de calculer l'intégrale de la fonction de densité de la loi normale centrée et réduite dont la fonction de densité s'écrit :

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

L'en-tête de la fonction s'écrit : `loi_normale <- function(a,b,epsilon=0.001)`

Pour vérifier votre fonction, faites : >> **pnorm(b)-pnorm(a)** << dans R.

## 6. Fonction de répartition de la loi normale

Exploitez le fait que la fonction de répartition de la loi normale est symétrique autour de 0, et que F(0) = 0.5 pour l'obtenir à partir de la fonction réalisée à la question précédente.

Votre fonction maintenant s'écrit : `loi_normale_plus <- function(b,epsilon=0.001)`

Il s'agit donc de calculer la surface allant de $-\infty$ à b maintenant en s'appuyant sur les propriétés particulières de la loi normale décrite ci-dessus.

Votre fonction devrait faire appel à la fonction **loi_normale()** de la question précédente.

Pour vérifier votre fonction, faites >> **pnorm(b)** << sous R.

## 7. Fonction de répartition de la loi du KHI-2

Programmez la fonction de répartition de la loi du KHI-2. La description de sa fonction de densité est accessible sur le web. Voici son en-tête :

`loi_khideux <- function(b,ddl,epsilon=0.001)`.

**Attention :**

- La loi du KHI-2 est définie sur [0, $+\infty$ [
- Le principe est le même que pour la question 5, sauf que la fonction de calcul de la surface à l'aide de la méthode de trapèze doit être réécrite pour qu'elle puisse prendre en paramètre la fonction de densité (loi normale ou loi du khi-2) !

Réécrivez en ce sens la fonction **loi_normale**, puis définissez la fonction **loi_khideux**.

Voir >> **pchisq** << pour vérifier votre fonction.

## 8. Méthode de Simpson

L'approximation des surfaces locales à l'aide de la méthode des trapèzes ci-dessus a le mérite de la simplicité, mais peut être considérée un peu fruste parfois.

On se propose d'améliorer le processus en programmant l'approximation par la méthode de Simpson (https://fr.wikipedia.org/wiki/M%C3%A9thode_de_Simpson) c.-à-d. vous mettez en place un dispositif similaire à ci-dessus, mais au lieu d'utiliser la formule des trapèzes pour calculer la surface de chaque tuyau d'orgue de largeur epsilon, vous utilisez la formule de Simpson.

Redéfinissez alors les fonctions ci-dessus (loi_normale, loi_normale_plus, loi_khdeux) de manière à ce que l'on puisse lui passer en paramètre le calcul local des surfaces trapeze() [par défaut] ou simpson(). Comparez les résultats pour différentes valeurs de epsilon. La nouvelle approche est-elle plus précise ?

Voici les nouvelles signatures des fonctions :

```
loi_normale <- function(a,b,epsilon=0.001,surface=trapeze)
loi_normale_plus <- function(b,epsilon=0.001,surface=trapeze)
loi_khideux <- function(b,ddl,epsilon=0.001,surface=trapeze)
```

## 9. Suite de Fibonacci

Ecrire une fonction qui renvoie la valeur du nombre de Fibonacci pour n (entier >= 0).

```
fibonacci <- function(n)
```

Passez par la formule de récurrence fibonacci(n) = fibonacci(n-1) + fibonacci(n-2), sachant que fibonacci(0) = 0 et fibonacci(1) = 1.

Remarques :

- Vous pouvez vous essayer éventuellement à une écriture récursive de votre fonction.
- Si l'utilisateur entre une valeur n non entière (avec une partie décimale) ou négative, vous interrompez l'exécution de la fonction avec l'instruction stop("Message d'erreur") (cf. https://stat.ethz.ch/R-manual/R-devel/library/base/html/stop.html)

## 10. Résolution d'équation – Méthode de dichotomie

La méthode de dichotomie est un algorithme de recherche d'un zéro d'une fonction dans un intervalle donné (https://fr.wikipedia.org/wiki/Méthode_de_dichotomie). A partir du pseudo-code disponible sur Wikipédia, programmez une fonction qui prend en entrée une fonction à analyser à un paramètre (FONCTION), les extrémités de l'intervalle à traiter (a, b), et le degré de précision souhaité (epsilon). Elle renvoie la valeur de l'abscisse correspond au zéro de la fonction à analyser.

Voici le prototype : `dichotomie <- function(FONCTION, a, b, epsilon=0.0001)`

**Attention** : vérifiez au préalable que (b > a), et que FONCTION(a) et FONCTION(b) sont de signes opposés. Si ce n'est pas le cas, stoppez les calculs en renvoyant la valeur NA.

**Remarque** : Quelques exemples de fonction à analyser et d'appel de dichotomie() [vous pouvez tester d'autres fonctions à analyser bien sûr]

```r
#fonctions à analyser
FONCTION_ID <- function(x){
  return(x)
}

#autre fonction à analyser
FONCTION_CARRE <- function(x){
  return(x^2-1)
}

#dichotomie
dichotomie <- function(FONCTION,a,b,epsilon=0.0001){
 … à vous de programmer …
}

#exemle d'appel
print(dichotomie(FONCTION_ID,-10,10)) # -7.629395e-05
print(dichotomie(FONCTION_CARRE,0,5)) # 0.9999847
```

## 11. Seuils des bâtons brisés en ACP

**Remarque :** Nous ne sommes pas censés connaître les vecteurs à ce stade de notre cours.

Le test des bâtons brisés identifie les axes pertinents dans une ACP normée. Sont acceptés les composantes portées par des valeurs propres supérieures à un seuil, lequel est défini par le nombre total de facteurs et le numéro de la composante à évaluer (voir la doc sur le drive [Séance 2] **ACP.pdf** ; **page 25**).

Ecrire une fonction qui prend en entrée le nombre total de facteurs (où le nombre de variables présentées à l'ACP, ces informations sont équivalentes) « p » et affiche directement les valeurs seuils pour chaque composante (k = 1, …, p).

Voici la signature de la fonction : `batons_brises <- function(p)`

## 12. Probabilité simulée (méthode de Monte Carlo)

Ecrire une fonction qui génère « n » fois une valeur aléatoire selon une loi normale (0, 1), elle doit ensuite calculer la proportion des valeurs supérieures à un seuil.

Voici le prototype de la fonction : `proba_simulee <- function(n, seuil)`

Où : « n » est le nombre de valeurs à générer

« seuil » est le seuil de référence

Remarque :

- La fonction **`rnorm(1)`** permet de générer « une » valeur aléatoire selon une loi normale centrée et réduite.
- Votre résultat devrait être proche de **`1 – pnorm(seuil) ;`** plus n sera élevé, plus votre simulation sera précise.
- Je rappelle que nous ne sommes pas censés savoir manipuler les vecteurs pour l'instant, il faudra passer par des boucles.

## 13. Anagrammes

Ecrire une fonction qui prend en entrée une chaîne de caractères de longueur dont on considère qu'elle ne possède pas de doublons (pas de caractères qui se répètent, il n'est pas nécessaire de tester, on part du principe que l'utilisateur joue le jeu). Votre fonction doit afficher les chaînes formées à partir de toutes les permutations possibles de ses caractères.

Par ex. : pour « ABC », nous avons « ABC », « ACB », « BAC », …, « CBA » (3 ! = 6 chaînes).

Voici la signature de la fonction : `combinatoire <- function(chaine)`

Remarque : substr() permet d'extraire une sous-chaîne d'une chaîne via des indices, nchar() permet de comptabiliser le nombre de caractères dans une chaîne.

## 14. Optimisation

Une entreprise fabrique des poutres métalliques de longueur de 3 m, que l'on découpe en tronçons de trois longueurs prédéfinies selon les commandes clients : 0.5 m, 1.5 m et 2 m.

Les chutes posent problème car elles doivent être refondues à la charge de l'entreprise, ce qui s'avère très coûteux. L'objectif est donc d'optimiser la découpe c.-à-d. produire le moins de poutres possibles afin de minimiser les chutes.

Prenons un exemple. L'entreprise reçoit une commande de de (n1 = 2 tronçons de 0.5 m ; n2 = 0 de 1.5m ; n3 = 1 tronçon de 2m). La meilleure solution est de produire (N = 1) seule poutre de 3 m que l'on découpera en une fraction de 2 m, et 2 de 0.5 m. Le total des chutes est égal à (chute = 0). La pire solution est de produire (N = 3) poutres, le total des chutes serait alors de (chute = 1 + 2.5 + 2.5 = 6 m).

Implémentez un algorithme qui renvoie le nombre minimal (N) de poutres à produire pour une commande (n1, n2, n3) donnée. La fonction doit également afficher la longueur totale des chutes. Voici la signature de la fonction : **`optim_production <- function(n1,n2,n3)`**