Sinussum

Impact d'un nombre partiel de termes dans une somme de sinusoïdes

Table des matières

1. Introduction

Le module2 du cours ICC nous apprend que tout signal peut s'exprimer théoriquement sous forme d'une somme, éventuellement infinie, de sinusoïdes, comme par exemple le signal en dents de scie (Fig1a). Vous verrez plus tard dans votre cursus comment calculer les coefficients de ces termes. Dans ce projet, La Table1 vous fournit l'expression théorique des trois signaux de la Fig1 (dent de scie, carré, triangle) et on veut examiner l'impact du cas pratique pour lequel on calcule seulement un nombre fini **nbN** des termes de ces sommes.

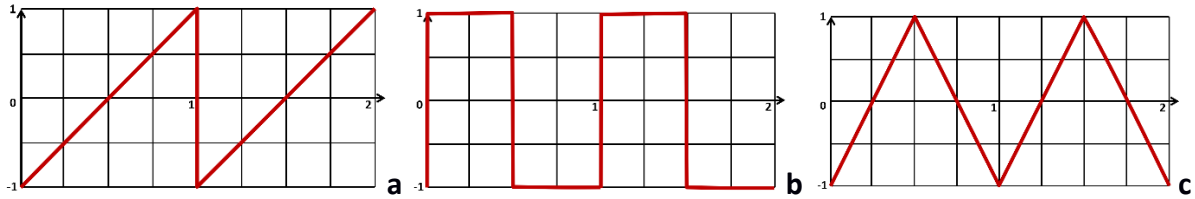

Figure 1 : 2 périodes des signaux en dent de scie (a), carré (b) et triangulaire (c) ; tous ces signaux peuvent être exprimés avec des segments de droites mais aussi avec une somme infinie de sinusoïdes (Table 1)

Dent de scie	$\frac{-2}{\pi}\sum_{k=1}^{\infty}\frac{(-1)^k}{k}\sin\left(2\pi k(t-0.5)\right)$
Carré	$\frac{4}{\pi}\sum_{k=1}^{\infty}\frac{\sin\left(2\pi(2k-1)t\right)}{2k-1}$
Triangulaire	$\frac{-8}{\pi^2}\sum_{k=1}^{\infty}\frac{(-1)^k}{(2k-1)^2}\sin\left(2\pi(2k-1)(t-0.25)\right)$

Table 1 : expression des signaux de fréquence 1Hz et d'amplitude variant entre -1 et 1, avec une somme infinie de sinusoïdes

Pour chaque exécution du programme on indique un des 3 signaux, le nombre de termes **nbN** de la somme et les paramètres précisant comment on veut afficher dans le terminal le signal obtenu avec des segments de droites[1] et celui approché avec une somme partielle. Le programme affiche le signal approché et la valeur maximale de ce signal sur une période.

[1] Pour les valeurs de *t* produisant une discontinuité, on pose que la valeur désirée est la moyenne de celles obtenues par les deux segments de droites voisins. Par exemple, en *t* égal à 0 et 1 cette valeur vaut zéro pour les signaux en dent de scie et carré. Même chose en *t* égal à 0.5 pour le signal carré.

2. Méthode de travail générale

2.1 Mise en oeuvre des grands principes

Le projet représente une quantité de travail bien supérieure aux exercices proposés dans les séries. La mise en oeuvre des principes ***d'abstraction*** et de ***ré-utilisation*** est un élément central dans la stratégie de résolution.

En effet, certaines tâches correspondent à un sous-problème (*abstraction*) dont la solution sous forme d'une fonction peut être utilisée pour résoudre une tâche plus complexe. C'est tout à fait normal qu'une telle fonction ne soit appelée qu'une seule fois car le but est la structuration de la solution d'une manière claire et lisible (un critère fréquent est que la taille maximum d'une fonction ne doit pas dépasser une page écran). Dans certains contextes la conception d'une fonction peut être ajustée à l'aide de paramètres pour pouvoir être *ré-utilisée* à plusieurs endroits dans le code.

2.2 Faire une analyse papier-crayon et pseudocode AVANT de vous lancer dans le codage

Le problème est décrit d'une manière indépendante d'un langage de programmation en section 3 ; c'est ce qu'on appelle les *spécifications*. Une telle description permet de réfléchir, *sans programmer*, à la décomposition du problème en un ensemble de sous-problèmes qui seront réalisés par des fonctions ; c'est ce qu'on appelle la phase *d'analyse*. Cette phase est typiquement faite avec un papier-crayon en pseudocode comme support pour organiser vos idées et faciliter le dialogue avec les assistant.e.s.

Le résultat de cette phase d'analyse est un ensemble de fonctions dont le *but* de chaque fonction est clairement identifié : sur quelles données travaille-t-elle ? Quel(s) résultat(s) fournit-elle ?

Ensuite seulement on peut passer à une méthode de codage rigoureuse qui va vous garantir une progression régulière dans la mise au point du projet. Elle est décrite ci-dessous.

2.3 Vérification précoce par les tests (*scaffolding*)

La clef du succès du codage est de ***vérifier*** que chaque fonction réalise bien son but avec un *solide éventail de tests pour lesquels on connait les résultats attendus*. Grâce à cette méthode un sous-problème est résolu une fois pour toute dès le niveau le plus élémentaire.

L'approche de vérification par des tests implique d'*écrire du code supplémentaire dédié à ces tests* AVANT de commencer à traiter les niveaux supérieurs du projet. Vous serez ainsi amené à écrire un certain nombre de petits programmes dédiés à ces tests. Cette méthode de travail est appelée *scaffolding* (échafaudage) et cherche à rendre votre code robuste à la grande variété des cas possibles.

En effet nous disposons d'outils tels que l'éditeur et le compilateur pour détecter certaines erreurs mais ils ne permettent pas de toutes les trouver. Nous savons déjà qu'il est recommandé de *recompiler très fréquemment* afin réduire au minimum le temps de recherche des ***erreurs syntaxiques*** (fautes d'orthographe/grammaire du langage C++). Le raisonnement est qu'une erreur se trouve dans la portion de code écrite depuis la dernière compilation avec succès, ce qui permet de réduire à très peu de lignes de codes l'espace de recherche de cette erreur. La méthode de l'échafaudage (scaffolding) est destinée à trouver les ***erreurs sémantiques*** que le compilateur ne trouve pas car le programme respecte la syntaxe du

langage ; les erreurs sémantiques vont produire un comportement incorrect du programme. Le point essentiel dans cette méthode est qu'il faut tester *chaque fonction* ***individuellement*** pour *vérifier* qu'elle produit le résultat attendu AVANT de l'utiliser ailleurs. Là aussi cette approche vous rend plus efficace dans le développement de code car l'erreur sémantique se trouve normalement dans la dernière fonction en cours de test car les autres fonctions qu'elle appelle doivent être déjà validées.

2.4 Bottom-up ou top-down ?

La méthode présentée dans la section précédente suggère de vérifier d'abord les fonctions de bas-niveaux avant celles qui les utilisent. C'est l'approche ***bottom-up*****.** C'est en général ce que nous recommandons pour un projet.

A l'inverse, il existe aussi une approche ***top-down*** pour la mise au point des fonctions ; il peut être légitime de vouloir tester une fonction **f()** qui appelle une fonction **g()** avant que **g()** soit écrite en détail. Cette approche *top-down* est possible si le résultat de **g()** est facile à définir, par exemple si elle renvoie `true` ou `false`, car la seule chose utilisée par **f()** est ce résultat.

On peut ainsi vérifier **f()** à l'aide d'une *forme minimale de la fonction* **g()** que l'on appelle un ***stub***. Cette forme minimale respecte le prototype prévu pour **g()** en terme de paramètres et de type de la valeur de retour ; par contre sa définition peut se restreindre à un corps vide si aucune valeur n'est retournée ou très peu de chose, par exemple une instruction return de *true* ou *false* si **g()** est supposée renvoyer un booléen.

2.5 Redirection des entrées-sorties pour automatiser les tests d'un programme

On utilisera **exclusivement** l'entrée standard (*clavier*) pour fournir les données et la sortie standard (*terminal*) pour afficher les résultats. Il ne faut pas écrire de code de lecture-écriture de fichier dans ce projet.

A la place, le mécanisme de *redirection* des entrées-sorties est vu en TP (semaine6) et permet de travailler avec des fichiers de test *sans avoir à changer une seule ligne de code*. En effet les données d'un test peuvent être éditées avec l'éditeur de programme et mémorisées dans un fichier de test. Ensuite il suffit de préciser le nom du fichier de test sur la ligne de commande qui lance l'exécution du programme et celui-ci obtient les données du fichier comme si elles venaient du clavier.

Exemple : si le nom de l'exécutable de votre projet est **proj** et que le nom d'un fichier de test est **t01.txt**, alors vous pouvez lancer votre exécutable dans le terminal de la façon suivante :

```
./proj < t01.txt
```

où le contenu de **t01.txt** est envoyé sur l'entrée standard pour cette exécution de **proj**.

Cette méthode de test est recommandée surtout pour relancer fréquemment et efficacement un ensemble de tests et vérifier que votre programme est toujours correct. C'est aussi en redirigeant des fichiers de test sur l'entrée standard que nous noterons votre programme. De plus nous redirigerons l'affichage dans un fichier que nous comparerons avec le résultat obtenu avec notre version du programme. Voici comment compléter la syntaxe précédente avec une redirection de la sortie vers un fichier **out01.txt** que vous pouvez ensuite ouvrir dans un éditeur de texte pour le consulter:

```
./proj < t01.txt > out01.txt
```

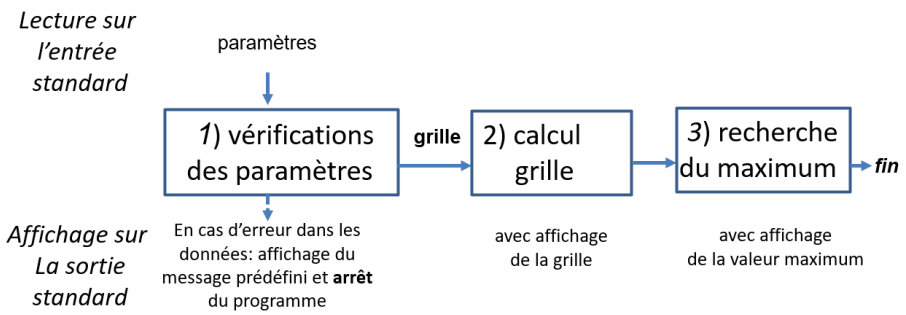

Figure 2: tâches principales à effectuer par le programme : tout d'abord la vérification des paramètres. Les affichages sont effectués sur la sortie standard.

3. Spécifications détaillées

3.1 Buts des tâches

Cette section approfondit les éléments fournis en section 1 en cherchant à identifier des structures de données et des fonctions pouvant être ré-utilisées. La structure générale du programme est séquentielle avec une suite de trois tâches : d'abord la lecture et la vérification des données, puis le remplissage d'une grille qui va servir à dessiner une partie du signal, et enfin une recherche du maximum sur une période du signal approché par le nombre ndN de termes de la somme de sinusoïdes (Fig 2).

3.1.1 Tâche 1 : lecture et vérification des paramètres

Plusieurs types d'erreurs doivent être détectés, dès la lecture de chaque paramètre. Dès qu'une erreur est détectée, le programme doit afficher un message d'erreur prédéfini. Nous désignons les messages d'erreur par une expression en majuscules dans la suite de cette section ; la manière de les afficher est fournie en section 4.4. En particulier, vous devez utiliser le fichier fourni **useful_stuff.txt** pour les recopier dans votre programme.

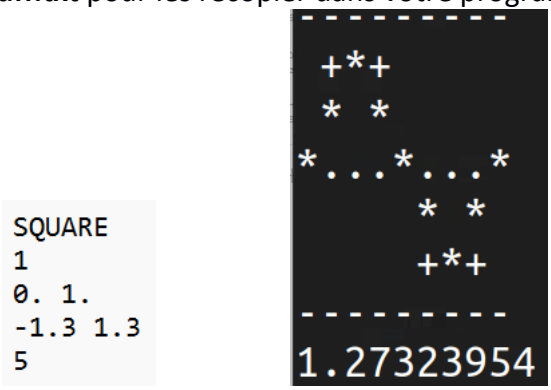

Figure 3: contenu du fichier de test t01.txt (gauche) ; sortie du programme pour ce fichier redirigé sur l'entrée standard : affichage d'un signal carré de fréquence 1Hz et d'amplitude 1 et son approximation avec une seule sinusoïde dans l'intervalle temporel [0,1] et [-1.3, 1.3] sur l'axe vertical (droite)

Lecture des données : les données sont fournies sur l'entrée standard (stdin = clavier) sur des lignes distinctes selon l'ordre indiqué sur l'exemple de la Figure 3. Les données d'une même ligne peuvent être précédées, suivies et séparées par un ou plusieurs espaces ou une ou plusieurs tabulations. Les lignes des données peuvent être séparées par zéro ou plusieurs lignes vides. Les paragraphes suivants décrivent les données fournies au programme :

Type du signal : le premier paramètre est une chaîne de caractères désignant un des 3 signaux : SAWTOOTH, SQUARE, TRIANGLE (détection d'erreur: BAD_SIGNAL).

Nombre de termes **nbN** : entier qui doit être strictement positif (détection d'erreur:

NBN_TOO_SMALL).

Intervalle *temporel* pour l'affichage : nombres à virgule ; d'abord **tmin** puis **tmax** tels que **tmax > tmin** (détection d'erreur: TIME_MIN_MAX) *et* les deux valeurs appartiennent à [0, 1] (détection d'erreur: WRONG_TIME_VAL).

Intervalle *d'amplitudes* pour l'affichage des signaux théorique et approché : nombres à virgule ; d'abord **min** puis **max** tels que **max > min** (détection d'erreur: SIGNAL_MIN_MAX). Il n'y a pas d'autre limitation sur min et max.

La dernière valeur **nbL** est celle du nombre de lignes d'affichages **>2** (détection d'erreur: NBL_TOO_SMALL) et impaire (détection d'erreur : NBL_NOT_ODD). Votre programme doit en déduire le nombre de colonnes d'affichage nbC avec l'équation :

nbC = 2*nbL -1

Si une erreur est détectée, l'affichage du message d'erreur fait quitter le programme avec le code fourni (section 4.4). Si aucune erreur n'est détectée le programme peut continuer avec la tâche2 qui consiste à remplir un tableau à deux indices de **nbL lignes x nbC colonnes** qui va servir de grille pour représenter le signal théorique et le signal approché.

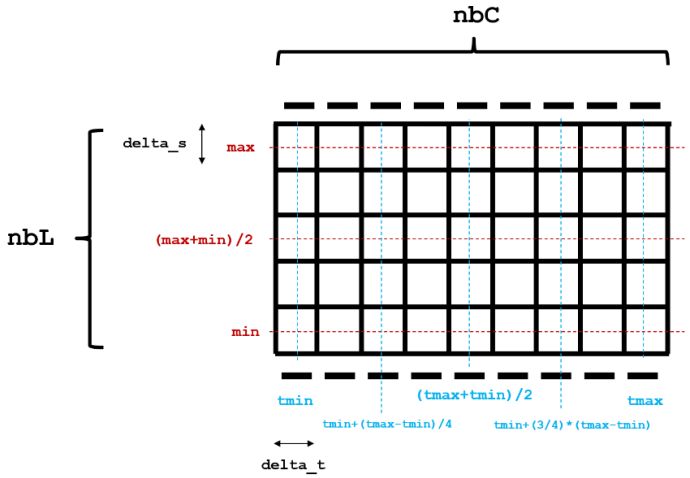

Figure 4: encadrement des valeurs temporelles et d'amplitude des signaux théoriques et approchés. L'affichage de cette grille est précédé et suivi par l'affichage de **nbC** caractères '**–**'

3.1.2 Tâche 2 : calcul et affichage de la grille représentant les signaux

La grille affichant les signaux utilise un petit nombre de caractères (Fig3 droite):

- chaque élément de la grille est initialisé avec le caractère « espace »
- les caractères suivants sont mis dans la grille avec une priorité croissante :
  - le caractère '.' indique que cet élément appartient à l'axe temporel s=0
  - le caractère '+' indique une valeur de signal théorique
  - le caractère '*' indique une valeur de signal approché

La figure 4 montre les spécifications du calcul et de l'affichage des signaux.
Une cellule de la grille selon l'axe temporel vaut **delta_t = (tmax-tmin)/(nbC-1) .**
Une cellule de la grille selon l'axe vertical vaut **delta_s = (max-min)/(nbL-1) .**

La variable temporelle **t** vaut **t=tmin +j*delta_t** ; avec **j** variant de **0** à **nbC-1**. Cette formule de mise à jour de **t** est destinée à minimiser l'accumulation d'erreurs d'arrondis sur sa valeur.

L'indice **j** est l'indice de colonne d'un élément de la grille. Certaines valeurs de **t** produisent une discontinuité (Fig1) pour SQUARE (0., 0.5 et 1.) et pour SAWTHOOTH (0. et 1.). Sachant que **t** est obtenu par calcul en virgule flottante on ne doit PAS faire un test d'égalité sur ces valeurs mais tester si **t** appartient à un intervalle avec une tolerance EPSIL_T.

Pour chaque valeur de **t** la valeur calculée d'un signal est notée **s(t)**. La valeur **s(t)** détermine quel indice de ligne **i** il faut utiliser pour écrire le symbole associé dans la grille. Si la valeur en virgule flottante de l'expression **v= (s(t) – min)/delta_s + 0.5** est positive ou nulle, alors on utilise l'entier **i** obtenu par troncation de **v** (si elle est négative, la valeur n'est pas dans la grille). Ainsi toutes les valeurs de **(s(t) – min)/delta_s** comprises dans l'intervalle **[i -0.5, i+0.5[** donneront **i** comme indice.

Calcul de la grille : la tâche2 consiste donc à initialiser la grille **nbL x nbC** avec des espaces puis à la remplir dans l'ordre suivant : 1) avec les symboles de l'axe temporel s'il est présent dans l'intervalle d'amplitude [min, max], puis 2) avec ceux du signal théorique et finalement 3) avec ceux du signal approché pour **nbN** sinusoïdes.

Affichage de la grille : La dernière étape consiste à afficher le résultat. Pour cela on affiche d'abord une ligne de **nbC** caractères '-' puis le contenu de la grille ligne par ligne, puis une seconde ligne de **nbC** caractères '-' (voir Fig3 droite[2]).

Bon à savoir : voir la section 2.5 pour tester très efficacement votre programme

3.1.3 Tâche3 : recherche du maximum du signal approché sur une période

Dans cette dernière tâche on vous demande de déterminer efficacement la valeur maximum sur une période du signal approché par les **nbN** sinusoïdes. Etant donné les types de signaux étudiés dans ce projet, cette valeur sera dans le voisinage de 1. Elle doit être affichée avec le format d'affichage indiqué en section 4.

Cadrage de l'intervalle temporel dans lequel rechercher le maximum du signal approché :
Ce maximum dépend bien sûr du type de signal mais la valeur de **nbN** peut aussi jouer un rôle. C'est pourquoi nous vous demandons d'effectuer cette recherche dans l'intervalle temporel suivant selon le signal :

- signal en dent de scie (Fig1 gauche) : [1- 1/(2*nbN+1), 1]
- signal carré (Fig1 milieu) : [0 , 1/(2*nbN+1)]
- signal triangulaire (Fig1 droite) : [0.5-1/(2*(2*nbN+1)), 0.5+1/(2*(2*nbN+1))]

Les deux premier cas sont les plus intéressants ; c'est sur eux qu'il faut mettre au point votre algorithme de recherche sachant que le signal approché présente ***un seul*** maximum dans l'intervalle temporel indiqué.

Type d'algorithme de recherche à mettre au point : l'algorithme de recherche doit tirer parti de cette information (un seul maximum dans l'intervalle temporel indiqué) pour mettre au point une approche par dichotomie. A chaque passage dans la boucle de cet algorithme, une valeur approchée du maximum est obtenue. L'algorithme doit s'arrêter lorsque la différence entre les valeurs obtenues lors de deux passages consécutifs est inférieure à EPSIL_DICHO (cette valeur est fournie dans **useful_stuff.txt)**.

[2] La valeur affichée 1.27323954 qui suit la seconde ligne de '-' dans la figure3 est le résultat de la tache3 décrite en section 3.1.3

4. Implémentation en langage C++

4.1 Contraintes spécifiques à ce projet

Indépendamment de ce que doit faire le programme (les spécifications décrites plus haut) nous imposons les contraintes suivantes de mise en oeuvre :

- Ecriture en C++11 : votre code sera compilé avec l'option **`-std=c++11`**
- Tout votre code doit être écrit dans un seul fichier source.
- Il faut utiliser un **vector** de **vector** de **char** pour la grille car **nbL** et **nbC** sont différents d'une exécution à l'autre.
- Il faut utiliser **string** pour lire les types de signaux ; ensuite on peut utiliser un **enum**.
- Les calculs doivent être fait en virgule flottante **double** précision.
- EPSIL_DICHO et EPSIL_T sont précisées dans le fichier **`useful_stuff.txt`.**
- Le format d'affichage du maximum doit être indiqué avant l'affichage du maximum. Il faut d'abord écrire cette instruction :

```
std::cout << setprecision(8) << fixed ;
```

4.2 Clarté et structuration du code avec des fonctions

La clarté de votre code sera prise en compte. Un examen manuel de votre code source sera effectué par une personne chargée d'évaluer le respect des conventions de programmation utilisées dans ce cours. Par souci d'efficacité seuls les codes indiqués dans nos conventions vous seront communiqués avec les numéros des lignes concernées dans votre fichier source.

4.3 Variables locales ou globales ?

4.3.1 Où déclarer les variables et les tableaux ?

La règle de base est qu'une variable (ou un tableau) n'est déclarée que ***localement***, là où elle est utilisée, le plus bas possible dans la hiérarchie des appels de fonctions. **Si et seulement si** une variable **x** (ou un tableau) déclarée dans une fonction **h()** est ***nécessaire*** pour une autre fonction **f(),** alors elle est transmise en paramètre à cette fonction[3] au moment de l'appel **f(x)** avec transmission par valeur ou par référence selon vos besoins**.**

Conséquence: si deux fonctions **f()** et **g()** indépendantes l'une de l'autre doivent travailler sur la même variable **x** (ou un tableau) alors une fonction de niveau supérieur **h()** doit être écrite qui va appeler **f(x)** et **g(x)** avec transmission par valeur ou par référence selon vos besoins.

4.3.2 Qu'en est-il des ***constantes*** ?

Voici les règles que nous nous donnons, en conformité avec nos conventions de programmation :

- Si une constante n'est utilisée qu'à l'intérieur d'*une seule fonction*, alors la déclaration d'une variable avec **constexpr** peut être faite dans cette fonction ou globalement.
- Si la constante est utilisée dans *plus d'une fonction*, alors la déclaration d'une variable avec **constexpr** *doit* être faite de manière globale, en début de fichier comme décrit dans les conventions de programmation.
- L'alternative de la déclaration de symboles avec #define est autorisée pour des

[3] Nous n'acceptons pas l'approche qui consiste à transmettre systématiquement un grand nombre de paramètres à la fonction **f()** et qui restent ensuite inutilisés dans cette fonction car cela la rend moins lisible. Conclusion : *chaque paramètre doit avoir sa justification*.

constantes utilisées dans plusieurs fonctions. Elles sont aussi déclarées en début de fichier comme décrit dans les conventions de programmation.

4.4 Fichiers de test, messages d'erreur :

Nous vous fournissons plusieurs fichiers de tests téléchargeables depuis le folder du projet sur moodle (section 4.4.2). A vous de créer vos propres fichiers plus élaborés ou juste différents. Il suffit d'ouvrir votre éditeur de code source préféré pour écrire les données et de sauvegarder avec l'extension **.txt** .

4.4.1 Message d'erreur

Le fichier **useful_stuff.txt** contient les définitions C++ des constantes, dont celles des messages d'erreur et une fonction **print_error** qu'il faut recopier dans votre fichier source (cf conventions). La fonction **print_error** se charge de quitter le programme en appelant **exit(0)**.

Exemple d'utilisation : si votre code détecte l'erreur BAD_SIGNAL il suffit de faire un appel en passant seulement la constante du message: **print_error(BAD_SIGNAL).**

4.4.2 Vérification anticipée de l'exécution de votre projet

Pour chaque fichier de test public nous fournissons le fichier texte de la sortie attendue. Vous pouvez ainsi valider les tests publics en comparant votre propre sortie avec celle qui est fournie. La section 2.5 montre comment rediriger la sortie de votre exécution vers un fichier.

Nous automatiserons la vérification de l'exécution de votre projet de la même manière, en comparant toutes les sorties attendues avec celle obtenues avec votre projet (après compilation par nos soins sur la VM).

Nous utiliserons aussi des fichiers de tests privés, non-fournis, pour vous stimuler à essayer d'autres scénarios de tests.

5 Proposition d'ACTIONS pour mettre en œuvre votre projet

Les sous-sections « ACTIONS » sont seulement des suggestions pour organiser votre travail.

5.1 Tâche1 : ACTION1 : test de la lecture des paramètres

La section 3.1.1 et la Fig 3 gauche décrivent l'ordre de lecture des paramètres. La taille de la grille n'étant pas connue à l'avance, **vector** s'impose comme choix pour les structures de données de ce projet.

Concevez une première version du programme qui gère seulement la lecture et la détection d'erreur sur les paramètres fournis en entrée. Dans ce premier exercice d'abstraction, rappelez-vous que la fonction principale **main()** joue seulement le role de « table des matières » et est constituée essentiellement d'appels de fonctions. Vous devez donc déléguer cette tâche de lecture a une ou plusieurs fonctions.

Concernant le bon endroit où déclarer vos variables, seules les variables stratégiques ont le droit d'exister au niveau de **main().** Les variables utilisées temporairement par une tâche doivent être définies localement dans la fonction qui gère cette tâche. Une fonction peut modifier un paramètre passé par référence ou renvoyer une valeur lue.

Testez cette première version en tirant parti de la redirection des entrées-sorties (série

TP_s4.2) pour couvrir les cas d'erreur avec les fichiers de test fournis et ceux que vous aurez écrits vous-même. On suppose que le type des entrées est correct (string, type numérique).

5.2 Tâche2 : ACTION2 : Affichage

Cette version minimale de votre programme doit déclarer, initialiser et remplir la grille du signal approché et théorique. Ecrire une fonction indépendante pour l'**affichage** d'une grille passée en paramètre.

Faites des tests rigoureux quand vous manipulez les indices

Grâce à cette étape vous pouvez visualiser les différents signaux avec un nombre **nbN** de termes de plus en plus grand. Utilisez aussi les paramètres de cadrage temporel et en amplitude pour zoomer sur la région intéressante pour la tâche 3. Familiarisez-vous avec la forme du signal approché.

5.3 Tâche3 : ACTION3 : recherche du maximum

Avant de vous lancer dans le code, ébauchez le pseudocode de votre recherche dichotomique avec du papier/crayon. Concentrez-vous sur le cas SQUARE et, grâce à votre expérience avec ACTION2, simulez l'exécution de votre algorithme sur une courbe de ce type.

L'implémentation ne pose pas de problème. Nous acceptons les deux approches : itérative et récursive. Respectez l'indication de format d'affichage donnée en section 4.1.

Ensuite exécutez le programme pour des valeurs de plus en plus grande de **nbN** et notez la valeur trouvée du maximum ; en effet le rapport final demande comment il évolue quand **nbN** augmente.

6. Rendu :

Noms des fichiers : tous les fichiers ont le même nom qui doit être votre numéro de SCIPER ; ils diffèrent seulement par leur extension :

.cc pour le fichier source, **.pdf** pour le rapport, **.zip** pour le fichier archive.

Par exemple pour une personne X de numéro SCIPER **123456**, le nom de fichier source est **123456.cc** son rapport est **123456.pdf** et ces deux fichiers sont dans le fichier archive **123456.zip** .

Téléversement du fichier archive : le fichier archive contient seulement l'unique **fichier source** et le **rapport**. Il DOIT obligatoirement être de type **.zip** ; la VM met à disposition cet outil de compression. Il faudra le téléverser (upload), au plus tard le **22/12 à 17h,** à l'aide du lien sur **moodle** (Topic 14) ; le site de téléversement sera ouvert plusieurs semaines avant la date limite sur moodle.

Vous êtes responsables de *vérifier* que l'upload s'est bien passé en le téléchargeant (download) dans votre compte et en examinant que tout est bien présent.

Vous pouvez toujours faire un nouvel upload jusqu'à la date limite.

Votre code source doit respecter les conventions de programmation du cours dont : au maximum 87 caractères par ligne (commentaire compris) et 40 lignes par fonction.

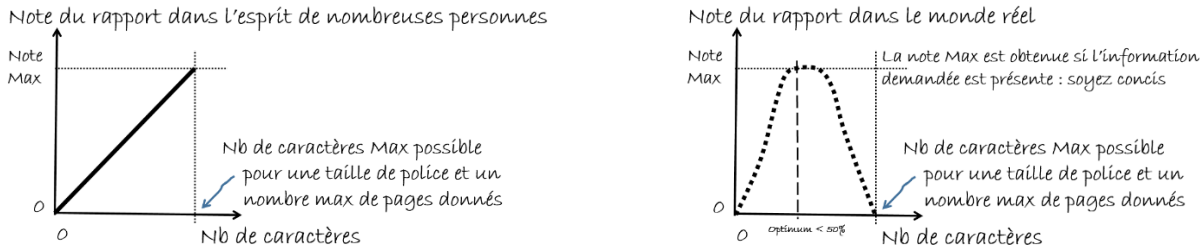

Figure 5 : un mauvais rapport dilue l'information utile jusqu'à atteindre le nombre maximum de caractères possibles sur la page (à cause de la croyance populaire de la figure de gauche) : dans la réalité ce type de rapport très compact sera très pénalisé parce qu'il est peu lisible (figure de droite) ; un bon rapport est celui qui fournit les informations demandées avec concision avec une mise en page aérée et lisible

6.1 Rapport

Le Rapport est d'au maximum **2 pages** écrites avec un traitement de texte. Il ne contient PAS de page de titre, ni de table des matières. Le Rapport contient :

a) Résultat de la phase **d'analyse** (max une demi-page, police de taille 11) :
Décrire la structure générale du programme en faisant ressortir, avec concision (Fig 5), la mise en oeuvre des principes *d'abstraction* et de *ré-utilisation* dans votre projet.

b) justifier **l'ordre de complexité de votre tâche2** en fonction de **nbN** et **nbL** en 2-3 phrases.

c) Fournir le **Pseudocode** de la recherche dichotomique du maximum du signal SQUARE (PAS de code source). Le pseudocode doit tenir en entier sur une seule page (soit sur la p 1 ou la p 2 mais pas entre les deux).

d) Quel est le comportement de la valeur maximale trouvée lorsque **nbN** tend vers l'infini ? Indiquez la valeur limite s'il y en a une. A faire pour les 3 types de signaux. Proposer une explication si une différence qualitative est observée entre les 3 types de signaux.

6.2 Barème indicatif (12pts):

Ce barème est indicatif et provisoire. Il sera éventuellement modifié par la suite.

(2pt) rapport : soyez concis (Fig 5)
(4pt) Lisibilité, structuration du code et conventions de programmation du cours
(3pt) votre programme fonctionne correctement avec les fichiers publics
(3pt) votre programme fonctionne correctement avec les fichiers non-publics

Dernier rappel : le projet d'automne est INDIVIDUEL ; vous pouvez seulement utiliser des outils de type copilot de VSCode. Il est interdit de sous-traiter tout ou partie du travail à un tiers. De plus le détecteur de plagiat sera utilisé selon les recommandations du SAC. Le plagiat inclut la copie de code disponible sur internet.