Sujet Projet Informatique:
Evolution du courant d'un circuit.

2012

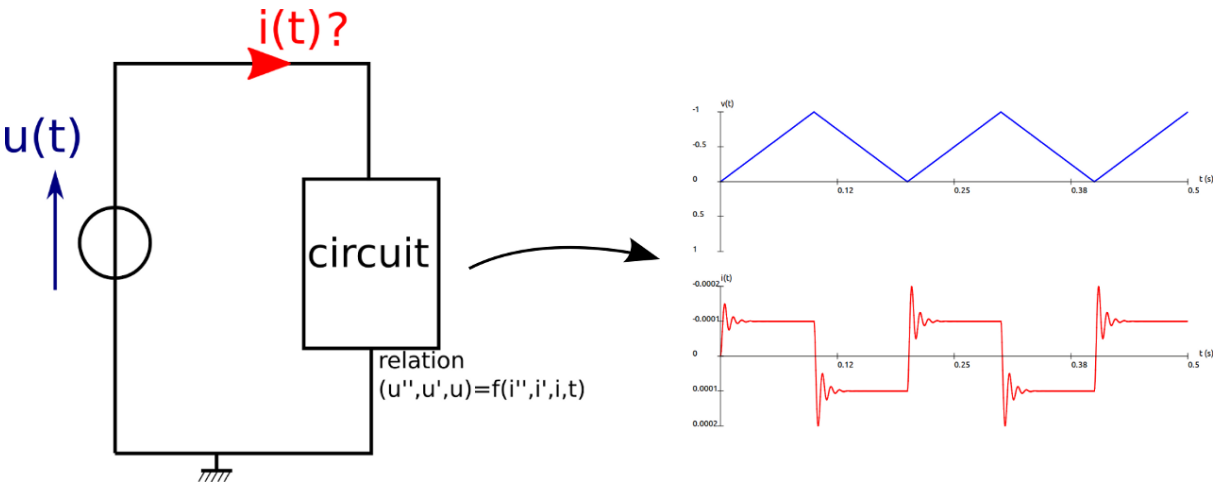

FIGURE 1 – Principe du projet. Étant donné un générateur de tension délivrant $u(t)$ et un circuit modélisé par sa relation entre $u(t)$ et $i(t)$, on simule et on affiche le courant $i(t)$ passant dans le circuit.

1 But

Le but de ce projet consiste à programmer la résolution numérique de l'équation régissant l'évolution du courant passant dans un circuit électronique comme illustré en fig. 1.

2 Introduction

Considérons les circuits montrés en fig. 2 :
Dans le premier cas, l'évolution du courant est donnée par l'équation

$$i(t) = \frac{1}{R}\, u(t)\ .$$

Dans le second cas, on a

$$RC\, i'(t) + i(t) = C\, u'(t)\ .$$

Alors que le troisième cas est donné par

$$CR_2\, u'(t) + LC\, u''(t) = R_2\, i(t) + (L + R_1 R_2 C)\, i'(t) + LC(R_1 + R_2)\, i''(t)\ .$$

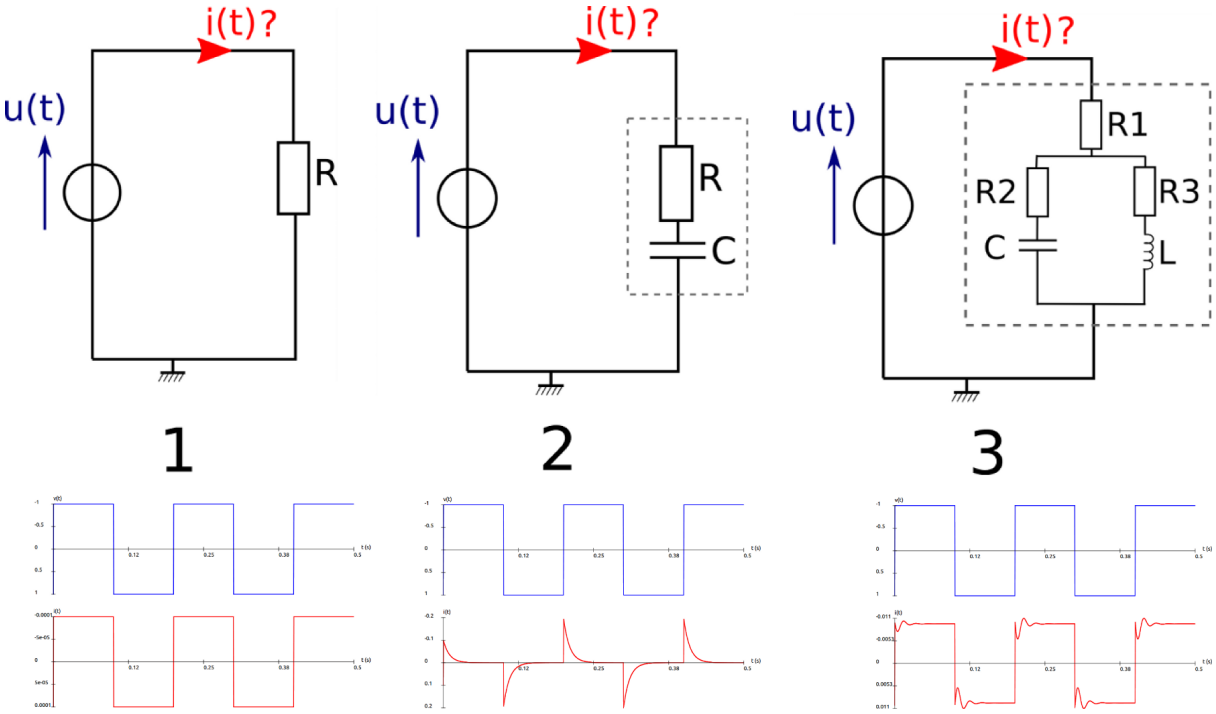

FIGURE 2 – Trois exemples de circuits électroniques modélisable par ce projet ainsi que les courbes de réponses résultantes.

Les solutions analytiques du courant passant dans ces circuits peuvent être connues dans des cas simples (u constant ou sinusoïdal), mais reste plus difficile à exprimer dans le cas général.
On a alors recours à des simulateurs, qui vont résoudre ces équations de manière numérique pour un signal $u(t)$ quelconque donné. On pourra utiliser ce type d'approche pour des signaux $u(t)$ complexes ou pour connaitre le régime transitoire. C'est cette résolution que nous allons mettre en oeuvre dans ce TP.

3 Principe général

3.1 Modélisation du circuit

Nous modélisons un circuit par son équation différentielle. Dans le cas d'assemblage d'éléments passifs linéaires (résistance, condensateur, bobine), cette équation est linéaire. On suppose de plus dans un premier temps que le circuit peut se modéliser par une équation d'ordre 2. C'est-à-dire que l'on peut écrire la relation courant-tension, sous la forme :

$$a_0\, u(t) + a_1\, u'(t) + a_2\, u''(t) = b_0\, i(t) + b_1\, i'(t) + b_2\, i''(t)\ . \tag{1}$$

On en déduit que la modélisation d'un circuit électrique à cette échelle peut se résumer par la donnée des 6 coefficients $[a_0, a_1, a_2, b_0, b_1, b_2]$. On appellera ce vecteur le *modèle*.

Ainsi, par identification, le *modèle* associé aux 3 circuits est donné respectivement par les vecteurs [1] :

- $[1, 0, 0, R, 0, 0]$
- $[0, C, 0, 1, RC, 0]$
- $[0, R_2C, LC, R_2, L + R_1R_2C, LC(R_1, R_2)]$

1. Notez le degré de liberté liée à la normalisation

3.2 Résolution numérique

Étant donné un modèle de circuit, on peut résoudre numériquement l'équation (1), on appelle cela *integrer l'équation*. Pour cela, on considère l'équation à temps discret avec $t = k\Delta t$. Soit i_k et u_k les courants et tensions à l'instant $t = k\Delta t$. En approximant les dérivées par leurs valeurs discrètes avec $i'_k = (i_k - i_{k-1})/\Delta t$, et $i''_k = (i_k - 2\,i_{k-1} + i_{k-2})/(\Delta t)^2$, on peut montrer (voir annexe) que l'équation (1) se réécrit de manière discrète sous la forme :

$$i_k = K(A\,u_k + B\,u_{k-1} + C\,u_{k-2} + D\,i_{k-1} + E\,i_{k-2})\,. \tag{2}$$

Résoudre l'équation différentielle numériquement, revient ainsi alors dans ce cas à itérer sur cette relation récurrente. Les valeurs de (K, A, B, C, D, E) sont données en fonction du *modèle* dans l'annexe.

Notons que dans le cas d'un circuit au repos, on partira des états initiaux

$$i_{-2} = i_{-1} = u_{-2} = u_{-1} = 0\,.$$

4 Implémentation

4.1 Structure générale

Nous allons implémenter cette résolution suivant une structure objet. Les circuits que nous étudions se mettrons toujours sous la forme de deux *éléments électriques* :

- Un *générateur* de tension qui contiendra un *signal* discret, et pourra générer des tensions diverses (rectangulaire,sinusoïdales...).
- Un *composant* passif qui sera caractérisé par un *modèle*. Un composant pourra se diversifier en des éléments classiques (condensateur, circuit RC série, parallèle, RLC, ... etc).

Étant donné ces deux éléments, nous construirons un *simulateur*. Celui-ci, ayant connaissance du signal u_k ainsi que du *modèle* de composant viendra appeler un *intégrateur* pour calculer i_k d'après la relation (1). Il effectuera ce calcul N fois, où N est la taille du signal u.

Une fois que le courant sera calculé. Nous utilisons un widget Qt afin d'afficher le courant $u(t)$ et la tension $i(t)$ sur le même schéma.

4.2 Détail des structures

4.2.1 Element électrique

Un élément électrique est un dipôle au sens large. Il peut posséder des attributs généraux tels qu'un nom, un revendeur, etc.

4.2.2 Générateur

Un générateur est un élément électrique permettant de générer un signal d'un type donné. Il pourra générer un signal sinusoïdal, rectangulaire, triangulaire, ou autre non périodique par exemple. La création de ces signaux seront paramétrés par : leur amplitude, leur fréquence s'ils sont périodiques, le pas de temps entre deux échantillons Δt, le nombre de points total. Il stockera la tension généré sous forme d'une classe *signal*.

4.2.3 Signal

Un signal est un conteneur pour un ensemble de valeur discrète. Il contiendra le vecteur des valeurs, ainsi que le pas d'échantillonnage Δt. On pourra également utiliser cette classe pour simplifier l'analyse du signal : ex. trouver le maximum, etc.

4.3 Composant

Un composant est un élément électrique dont l'étude peut être réalisée. Il contient un *modèle* qui le caractérise.

4.4 Composant spécifique

Un composant spécifique est un type particulier de composant pour lequel on connait explicitement sa composition. Par exemple, on pourra définir une classe *RLCserie* dont les paramètres d'entrées seront (R, L, C) et qui sera transcrit de manière automatique dans le modèle du composant. Chaque composant spécifique pourra également avoir des caractéristiques propres : ex. précision d'une résistance, couleur d'un composant, type de condensateur (chimique, ...), fréquence de coupure d'un filtre, etc.

4.5 Simulateur

Le simulateur est une classe qui, étant donné un composant et un générateur, va pouvoir calculer le courant i passant dans le circuit. Cette classe va appeler l'intégrateur N fois, où N est la longueur du signal.

4.6 Intégrateur

L'intégrateur à pour rôle de réaliser une étape de la relation de récurrence (2). L'intégrateur est implémenté sous forme de machine à état. C'est-à-dire que les valeurs précédentes $u_{k-1}, u_{k-2}, i_{k-1}, i_{k-2}$ sont mises à jour à chaque évaluation de i_k. Vue de l'extérieur de la classe, on aura donc ce type d'appel :

```
u <- signal
integrateur mon_integrateur(composant,delta_t);
for i=0:size(u)
  i[k]=mon_integrateur(u[k]);
```

L'intégrateur en lui-même précalculera les coefficients (K, A, B, C, D, E) dès sont initialisation en fonction du modèle de composant et de Δt. Il stockera et mettra à jour les 2 valeurs précédentes des courants et tensions nécessaires pour la récurrence.

5 Travail demandé

On suivra l'ordre de travail suivant :

5.1 Comprehension de la structure

- Réalisez le diagramme de classes des différents éléments de votre projet en suivant les indications données.

5.2 Implémentation : Modélisation du circuit

- Finir l'implémentation de la classe *signal* et celle du *générateur*.
- Vérifiez cette implémentation sur des exemples simples.
- Créez la classe de *composant* qui intègrera un *modèle*.
- Créez des classes simples de composant spécifiques telles que *resistance*, *RCserie*, ou ceux donnés en exemples.
- Vérifiez que votre classe est bien associée au bon modèle.

5.3 Implémentation : Intégration de l'équation

- Finir l'implémentation de la classe intégrateur sous forme de machine à états en précalculant les coefficients (K, A, B, C, D, E) et en replissant l'opérateur parenthèse afin de pouvoir appeler l'intégration en suivant la syntaxe : $i[k] = monintegrateur(u[k])$. Les valeurs des courants et tensions précédentes seront mises à jour lors de cet appel.
- Vérifiez l'intégration sur un exemple simple où on comparera les étapes de récurrences à celles calculées à la main.
- Insérez l'intégrateur dans une classe simulateur. Celle-ci aura connaissance du composant et du générateur, et proposera une méthode qui calculera et renverra le courant calculé sous forme de *signal*.

5.4 Test d'integration

- Vérifiez que l'ensemble de vos classes (modélisation du circuit + intégration de l'équation) fonctionnent bien entre elles. Pour cela, on considèrera des exemples simples et on affichera le résultat : soit sur ligne de commande pour des exemples très simples, soit sous forme de courbe à l'aide d'un outil externe (Matlab, Octave par ex) pour des exemples de circuits connus.
  À ce stade, votre code d'appel global devra ressembler à celui proposé en fig. 3 :

```
unsigned int N=5e4;double delta_t=1e-5; //parametres de la simulation
double u_max=1;double frequency=5; //generateur de tension
generateur generateur1;generateur1.generate_rectangle(u_max,frequency,delta_t,N);
double R=20,L=0.1,C=1e-4; //composant
RLC_serie composant1(R,L,C); //exemple d'un circuit de type RLC en serie
simulateur simu(generateur1,composant1);
signal i=simu.simulate(); //lancement de la simulation
```

FIGURE 3 – Exemple de code d'appel de la simulation pour un signal $u(t)$ rectangulaire et un circuit RLC.

Notez qu'il est très important de valider cette étape avec soin. Une fois validé, le coeur de vos classes n'aura plus à être modifié à postériori.

5.5 Implémentation : Affichage sous forme de courbe

- En utilisant les widget Qt, mettez en place la visualisation de vos signaux. Notez qu'un exemple vous est fourni pour une sinusoïde dans la classe *mywidget*. Celui-ci devra être modifié et recevra en paramètre votre signal de tension et de courant.
- Vérifiez vos courbes par rapport à des exemples connus (ex. Circuit RLC, etc.)

5.6 Extensions

On n'implémentera pas d'extension tant que le travail de base n'est pas réalisé et vérifié.

5.6.1 Extension à des ordres supérieurs

La mise en série de 3 circuits RC parallèle est modélisée par une équation différentielle d'ordre 3. Il est donc nécessaire d'adapter le modèle et la machine à état d'intégration pour gérer ces circuits plus complexes.

Proposez une implémentation pouvant s'adapter à des ordres supérieurs. Essayez de proposer une solution générique.

5.6.2 Modèle graphique

Implémentez une représentation graphique de votre composant. Par exemple, un modèle graphique (formé de rectangles et lignes) pourra dériver de chaque composant spécifique. Lors de l'affichage, le modèle graphique correspondant sera alors appelé par le *QPainter*.

5.6.3 Analyse de l'intégration

Observez l'influence de Δt sur votre résultat. Que pouvez-vous dire ?

Notez que la discrétisation proposée en annexe n'est pas unique. Ainsi $u(t)$ pouvant être connu à l'avance, on aurait pu utiliser la relation non causale : $u''_k = u_{k+1} - 2\,u_k + u_{k-1}$. Proposez d'autres intégrateurs et observez les modifications sur le résultat.

5.6.4 Spécification de composants

Les composants discrets ont généralement des plages de variations importantes (10 - 20$\%$). Prenez en compte ces valeurs dans vos modèles, et affichez les courbes moyennes et extrêmes liées à ces variations.

Annexe : Discrétisation de l'équation différentielle

On montre ici que l'équation (1) peut être discrétisé pour obtenir la relation donnée en (2). On rappelle que l'on part de

$$a_0\, u(t) + a_1\, u'(t) + a_2\, u''(t) = b_0\, i(t) + b_1\, i'(t) + b_2\, i''(t)\ .$$

En discrétisant les signaux u et i tout les Δt, on note $u_k = u(k\,\Delta t)$, avec $k \in [\![0, N-1]\!]$, et de même pour i_k. u et i peuvent alors être vues comme des vecteurs de taille N :

$$u = [u_0, u_1, \ldots, u_{N-1}]\ .$$

Pour discrétiser les dérivées, on rappelle que

$$u'(t) = \lim_{h\to 0} \frac{u(t) - u(t-h)}{h}\ .$$

De manière discrète, on approxime la limite de h avec Δt, le plus petit écart accessible. On a alors

$$u'(k\Delta t) = \frac{u(k\Delta t) - u(k\Delta t - \Delta t)}{\Delta t} = \frac{u(k\Delta t) - u((k-1)\Delta t)}{\Delta t}\ ,$$

C'est à dire

$$u'_k = \frac{u_k - u_{k-1}}{\Delta t}\ .$$

Pour obtenir la dérivée seconde, nous itérons deux fois cette étape : On a alors

$$u''(k\Delta t) = \frac{u'(k\Delta t) - u'(k\Delta t - \Delta t)}{\Delta t} = \frac{\frac{u(k\Delta t) - u(k\Delta t - \Delta t)}{\Delta t} - \frac{u(k\Delta t - \Delta t) - u(k\Delta t - 2\,\Delta t)}{\Delta t}}{\Delta t}$$

$$\Rightarrow u''(k\Delta t) = \frac{u(k\Delta t) - 2\,u((k-1)\Delta t) + u((k-2)\Delta t)}{(\Delta t)^2}\ ,$$

Ce qui nous donne

$$u''_k = \frac{u_k - 2\,u_{k-1} + u_{k-2}}{(\Delta t)^2}\ .$$

En injectant maintenant la discrétisation dans l'expression de l'équation différentielle, on obtient

$$a_0\, u_k + a_1\, \frac{u_k - u_{k-1}}{\Delta t} + a_2\, \frac{u_k - 2\,u_{k-1} + u_{k-2}}{(\Delta t)^2} = b_0\, i_k + b_1\, \frac{i_k - i_{k-1}}{\Delta t} + b_2\, \frac{i_k - 2\,i_{k-1} + i_{k-2}}{(\Delta t)^2}\ .$$

À ce stade, on rappelle que l'inconnue à trouver est i à l'instant courant, c'est à dire i_k, les autres valeurs de la tension ou du courant aux instants précédents étant connues.
En rassemblant les termes, on obtient alors

$$i_k = \frac{1}{b_0 + \frac{b_1}{\Delta t} + \frac{b_2}{(\Delta t)^2}} \Bigg[\left(a_0 + \frac{a_1}{\Delta t} + \frac{a_2}{(\Delta t)^2}\right) u_k - \left(\frac{a_1}{\Delta t} + 2\frac{a_2}{(\Delta t)^2}\right) u_{k-1} + \frac{a_2}{(\Delta t)^2}\, u_{k-2} + \left(\frac{b_1}{\Delta t} + 2\frac{b_2}{(\Delta t)^2}\right) i_{k-1} - \frac{b_2}{(\Delta t)^2}\, i_{k-2} \Bigg]\ .$$

Nous pouvons donc écrire par identification cette relation de récurrence sous la forme

$$i_k = K(A\, u_k + B\, u_{k-1} + C\, u_{k-2} + D\, i_{k-1} + E\, i_{k-2})\ ,$$

avec

$$\begin{cases} K = \frac{1}{b_0 + \frac{b_1}{\Delta t} + \frac{b_2}{(\Delta t)^2}} \\ A = a_0 + \frac{a_1}{\Delta t} + \frac{a_2}{(\Delta t)^2} \\ B = -\left(\frac{a_1}{\Delta t} + 2\frac{a_2}{(\Delta t)^2}\right) \\ C = \frac{a_2}{(\Delta t)^2} \\ D = \frac{b_1}{\Delta t} + 2\frac{b_2}{(\Delta t)^2} \\ E = -\frac{b_2}{(\Delta t)^2} \end{cases}$$

Aide : Premiers pas

L'énoncé est proposé avec un premier squelette de programme à compléter.

5.7 Compilation et documentation

Les fichiers sources du programme sont disponibles dans le répertoire *src/*, alors que les headers sont dans le répertoire *header/*. Le programme étant lié à Qt, il est nécessaire de faire appel à des générateurs de Makefile automatique. Deux outils sont proposés :

- QMake : avec le fichier *build makefile.pro*.
- CMake : avec le fichier *CMakeLists.txt*.

Notez que QMake est simple d'utilisation et spécialisé dans la gestion de projets Qt. CMake est plus générique, s'adapte à tout type de librairie et à l'avantage de pouvoir générer des Makefile pour Unix et également des projets Visual pour l'environnement Windows.
Vous modifierez l'un ou l'autre des deux fichiers proposés pour ajouter vos propres classes.

Veuillez vous référer au fichier *README* pour le détail des commandes de compilations.

Une documentation *Doxygen* est fournie et accessible par le fichier *doc/html/index.html*. Il s'agit d'une documentation générée automatiquement par analyse des fichiers d'en tête du projet. Ce type de documentation est standard pour de nombreuses librairies. Notez l'importance de la documentation des fichiers d'en-tête.

Ce type de documentation a pour but de renseigner sur l'utilisation des fonctions proposées vue de l'extérieur (arguments à donner, type de retour, rôle de la méthode, etc), et non sur le détail de l'implémentation de la fonction en elle-même.

Comme cette documentation est générée automatiquement, il est possible de la mettre à jour au fur et à mesure de la complétion de votre programme. Le détail de ces appels est fourni dans le fichier *README*. Notez que le diagramme des classes est automatiquement généré de manière visuelle.

Les mots clés (brief, param, return, ...) des commentaires des fichiers d'en tête sont spécifiques à la documentation Doxygen. Plus de détails sont disponibles sur le site de l'outil [2]

Dans un premier temps, assurez vous que vous être en mesure de compiler et d'exécuter le programme. Familiarisez-vous également avec la documentation.

5.8 Notation

Considérons un exemple de définition d'une classe avec le fichier *signal.hpp* (voir fig. 4).

- Ligne 9 : *namespace cpe* { } indique que cette classe est définie dans l'espace de nom *cpe*. C'est à dire que vue de l'extérieur, il est nécessaire d'indiquer le nom de la classe précédé de son espace de nom :
  ex. *cpe : :signal monSignal ;*
  Définir ainsi un espace de nom évite les collisions avec d'autres librairies qui peuvent avoir défini elles-mêmes un type *signal* (ce qui est le cas avec la librairie standard). Notez que la définition de la classe dans le fichier *cpp* est également englobé dans cet espace de nom.
  Dans le cas où de nombreuses classes appartenant à l'espace de nom *cpe* sont utilisées, il est possible de s'éviter le rappel de celui-ci en déclarant en début de fichier : *using namespace cpe ;*. C'est ce qui est fait dans le fichier *main.cpp*. Notez qu'on évitera tant que

2. www.doxygen.org

```
09 namespace cpe
10 {
12     class signal
13     {
14     public:
15
16         // ********************************************************* //
17         // Constructeur
18         // ********************************************************* //
19
21         signal();
23         signal(const std::vector<double>& data,double dt);
24
25         // ********************************************************* //
26         // Acces aux elements du signal
27         // ********************************************************* //
28
30         unsigned int size() const;
32         const double& operator[](unsigned int k) const;
34         const double& dt() const;
35
36         // ********************************************************* //
37         // Calcul sur les elements du signal
38         // ********************************************************* //
39
41       double max_signal() const;
42
43
44     private:
45
47         std::vector<double> data_interne;
49         double dt_interne;
50     };
51 }
```

FIGURE 4 – Exemple de la définition d'une classe et des notations utilisées.

possible d'utiliser un *using namespace* dans les fichiers d'en-tête qui amène à des effets de bords non souhaités.

- Ligne 23 : *const std : :vector<double> & data*. Le signe & indique que l'on passe le vecteur en tant que référence, et non par copie. Cela évite la création et la duplication entière de donnée pour un vecteur lors de cet appel du constructeur. Cela économise donc de l'espace mémoire et du temps.
  Comme le passage d'argument permet de modifier l'argument, le mot clé *const* permet d'assurer à l'utilisateur de la méthode que le vecteur qu'il envoie en paramètre ne sera pas modifié par la méthode.
  Notez que tout passage d'arguments non modifiés par la méthode doit se faire soit par référence constante (en C++), soit par pointeur constant (en C).

- Ligne 34 : *const double& dt() const ;*. Méthode qui renvoie la valeur *dt interne*. On renvoie ici directement la valeur et non une copie, car la méthode retourne une référence. (Notez qu'ici la copie serait valable, car il s'agit d'un simple *double*. Dans le cas d'un vecteur, il serait impératif de ne pas copier les données).
  Le mot clé *const* en début de ligne indique que la référence retournée ne pourra par être modifiée. Il n'est ainsi pas possible de réaliser d'affectation de *dt interne* par ce biais.
  Le mot clé *const* en fin de ligne indique que l'appel à cette méthode ne modifie pas les paramètres internes de la classe. C'est-à-dire que *data interne* et *dt interne* ne sont pas modifiés.

- Ligne 33 : *const double& operator[](unsigned int k) const*. Il s'agit de la définition de l'opérateur crochet [] permettant de l'extérieur d'obtenir la kième valeur du signal en appelant : *mon-*

Signal[k].
Notez qu'en définissant un retour de référence constant et en déclarant que la méthode ne modifie pas les paramètres internes, il est possible d'obtenir la valeur en écrivant par exemple :
double a=monSignal[5] ;
Mais pas de réaliser une affectation : *monSignal[5]=3 ;//interdit par le mot clé const*