Modélisations de solutions séquentielles destinées à piloter une chaîne d'énergie

publié le 20/07/2020 - mis à jour le 17/11/2020

La simulation d'un système réel ne s'arrête pas à la chaîne d'énergie. Pour l'animer, il convient de créer des stimuli adaptés et, si cela ne suffit pas, de réaliser des systèmes séquentiels chargés de commander la chaîne d'énergie

---

Descriptif :

L'animation réaliste d'un système, demande à connaître les méthodes de création de stimuli adaptés aux besoins, et, si nécessaire, de réalisation de solutions séquentielles, soit à l'aide de composants (pour des séquences très simples), soit à l'aide de scripts (bien plus souples, plus puissants et plus faciles à mettre au point).

Sommaire :

Cinquième volet d'une série de 7 articles amenant à maîtriser la modélisation de composants de la chaîne d'énergie et de la chaîne d'information dans le but d'effectuer la simulation d'un asservissement numérique de position la plus précise possible afin que le comportement réel soit totalement prévisible.

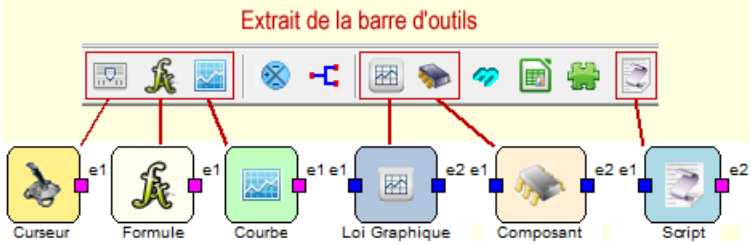
L'animation réaliste d'un système, demande à connaître les méthodes de création de stimuli adaptés aux besoins, et, si nécessaire, de réalisation de solutions séquentielles, soit à l'aide de composants (pour des séquences très simples), soit à l'aide de scripts (bien plus souples, plus puissants et théoriquement plus faciles à mettre au point).

Cet article va aborder, en s'appuyant sur des exemples, les différentes solutions permettant d'animer une chaîne d'énergie, en partant des plus simples pour arriver aux plus abouties (l'utilisateur pourra ainsi choisir son type de solution selon le contexte ou son goût pour la programmation) :

1. Soit à partir d'un **Curseur** paramétrable, dont la valeur peut être modifiée à la souris ou imposée en saisissant une valeur au clavier (l'utilisation des curseurs est essentielle lors de la phase de mise au point de la chaîne d'énergie afin de valider ses performances ou de rechercher ses limites en modifiant, par exemple, la consigne et/ou la charge)
2. Soit à partir d'une **Formule** permettant de définir rapidement un stimulus simple à l'aide d'une équation, linéaire ou pas, fonction du temps (accessible par la variable t)
3. Soit à partir d'une **Courbe** ou d'une **Loi graphique** permettant d'obtenir un stimulus de forme complexe (par exemple, le résultat obtenu lors d'une simulation précédente ou des données tirées d'une documentation constructeur) afin de reproduire une consigne ou une information issue d'un capteur, conforme(s) à ce que le système réel est censé recevoir (le but étant alors de visualiser le comportement du système dans son environnement).
4. Soit à partir d'un **Composant** permettant de compléter la bibliothèque par un nouveau stimulus, ou de réaliser entièrement un séquenceur basique (structure séquentielle généralement limitée à un cas simple, nécessaire et suffisante pour pouvoir commander une chaîne d'énergie en tenant compte d'une consigne, d'ordres ou de

capteurs binaires)

5. Soit à partir d'un **Script** (écrit généralement en Python 2.7, à installer sur la machine ou dans le langage de script LUA, intégré à Sinusphy) permettant de réaliser un séquenceur suffisamment évolué (les scripts permettent, généralement, de réaliser très facilement tous les traitements nécessaires en n'utilisant que quelques équations et quelques tests comme cela va être montré à la fin de cet article)



En complément des diverses démonstrations et explications de cet article, sont fournis en archive zip :

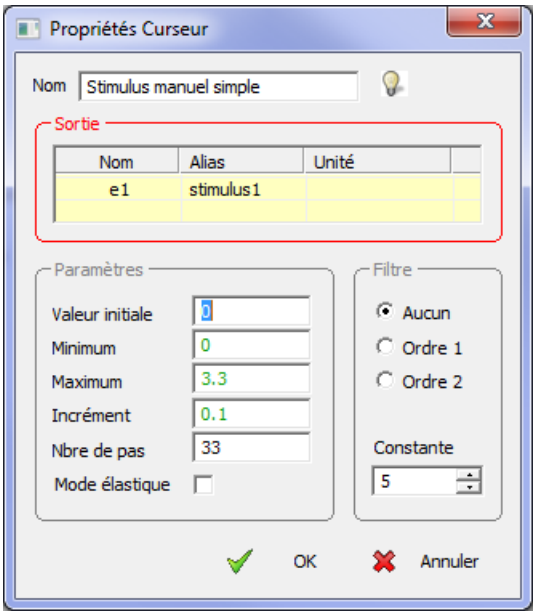
- Les schémas enregistrés en Sinusphy V2 (*pour être compatibles avec toutes les versions postérieures*),
- Les composants modifiés ou créés avec l'éditeur de composants (*compatibles avec toutes les versions*),
- Les scripts Python utilisés (*dotés de commentaires et compatibles avec toutes les versions*),
- Toutes les copies d'écran en qualité originale,
- Quelques modèles et schémas de simulation supplémentaires.

ressourcesarticlesinusphyn6 (Zip de 1.4 Mo)

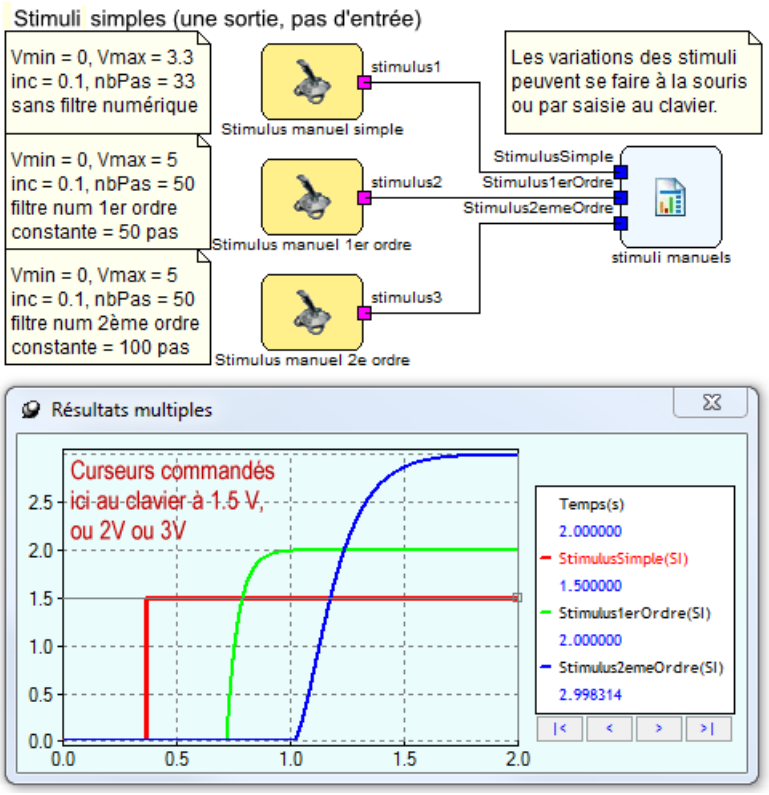
● 1- Utilisation de stimuli simples, prédéfinis, avec une seule sortie

L'outil **Curseur**, avec, ici, son paramétrage expliqué :

- **Nom** : jusqu'à 21 carac. si l'option "Labels externes" est cochée dans Sinusphy, sinon 11 carac. maxi,
- **Alias** : facultatif, lettres et chiffres uniquement !,
- **Unité** : optionnelle dans une chaîne d'information,
- **Valeur initiale, Minimum, Maximum, Incrément, Nbre de pas** : ⚠ cohérence de mise !,
- **Mode élastique** : à enlever en général car ramène automatiquement la valeur du curseur à 0,
- **Filtre** : Le filtrage numérique (à la manière d'une moyenne) permet de lisser (atténuer) les variations brusques de la valeur (gérée à la souris ou au clavier) afin de d'obtenir un signal lentement variable plus conforme à la réalité (constante exprimée par un entier correspondant au nombre de pas de calculs nécessaires : 100 maxi).



On voit, ci-dessous, les résultats obtenus, soit sans filtre, soit en choisissant un type de filtre et sa constante. Les 3 échelons ont été créés par la saisie directe de valeurs au clavier. Il est à remarquer que le filtrage n'entraîne aucun dépassement.



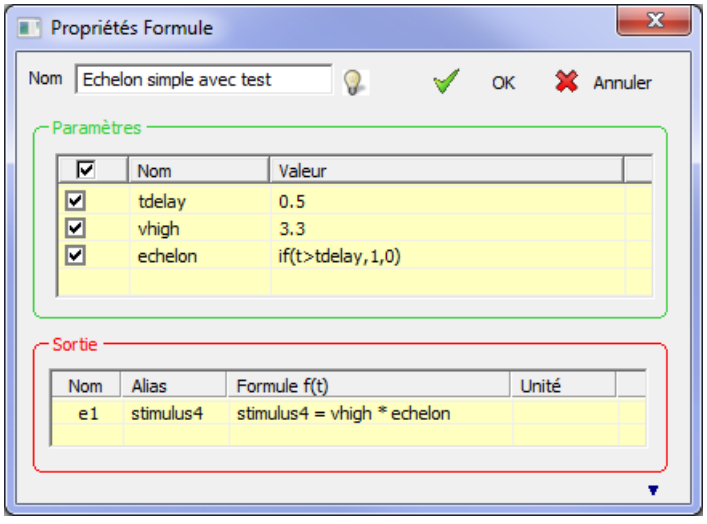
L'outil **Formule**, avec, ici, son utilisation expliquée sur 2 exemples :

Zone "Sortie" : Peut généralement suffire pour définir entièrement la formule mais on y perd en lisibilité

- **Alias** : nom facultatif ne devant comporter que des lettres et des chiffres,
- **Formule f(t)** : t est la variable d'entrée. On peut également définir une sortie constante,
- **Unité** : optionnelle dans une chaîne d'information

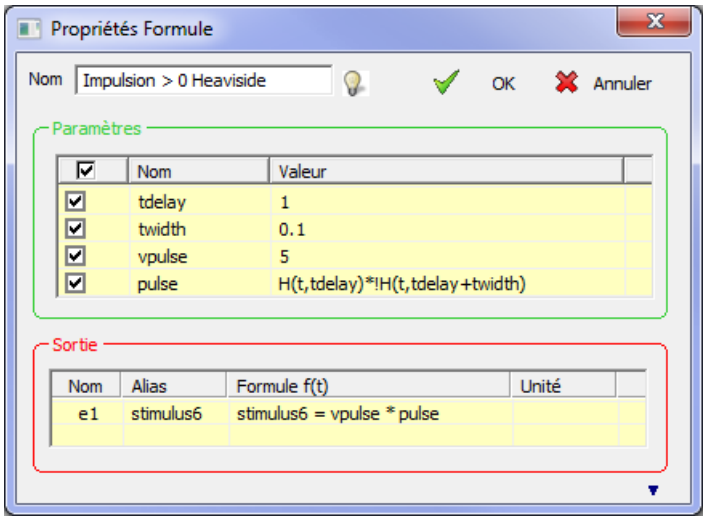
Zone "Paramètres" : Des constantes et un test **if** (les tests sont toujours des alternatives complètes) :

- **tdelay = 0.5** (tdelay représente ici un temps en secondes),
- **vhigh = 3.3** (vhigh représente ici une tension en volts),
- **echelon = if(t>tdelay,1,0)** signifie : si la variable système, notée t, est supérieure à tdelay, alors echelon=1 sinon echelon=0

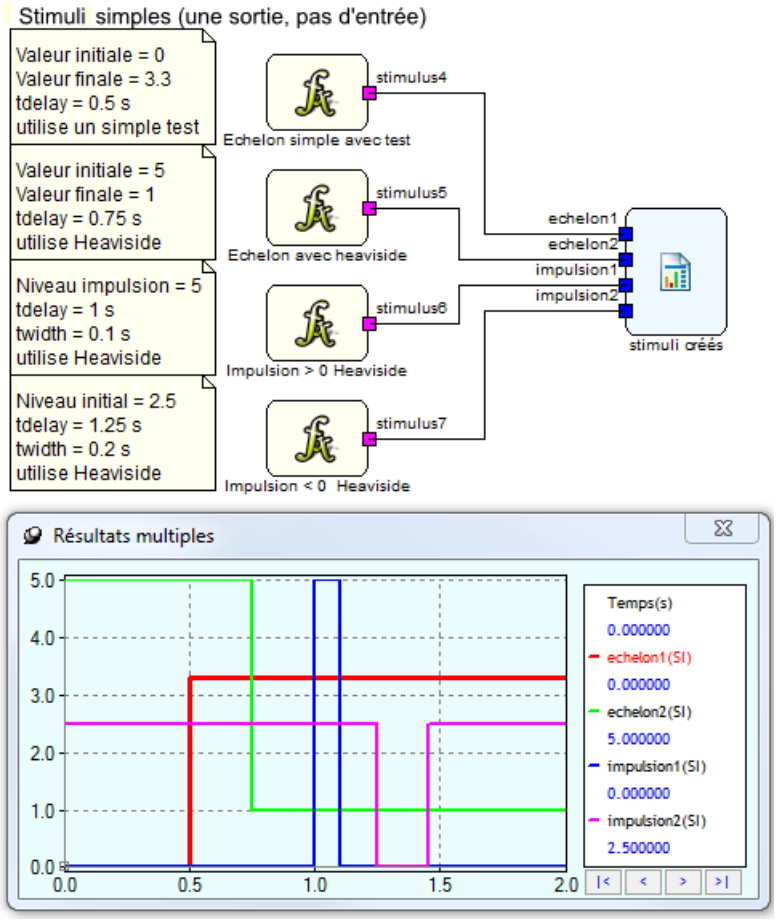


Zone "Paramètres" : Des constantes et la fonction Heaviside, **notée H**, peu connue mais très pratique (Heaviside, également appelée fonction échelon unité) :

- **tdelay = 1** (temps en secondes),
- **twidth = 0.1** (durée en secondes),
- **vpulse = 5** (tension en volts),
- **pulse = H(t,tdelay) * !H(t,tdelay+twidth)** signifie : si la variable système, notée t, est comprise entre tdelay et tdelay+twidth, alors pulse = 1 sinon pulse = 0.



On voit, ci-dessous, les formes des 4 signaux obtenus : 2 échelons et 2 impulsions. Seul le premier échelon est défini par un test **if(t>tdelay,1,0)**. En effet, la syntaxe peut être vite illisible si on associe plusieurs tests, c'est pour cela que l'on utilisera plus volontiers la fonction Heaviside, conçue pour réaliser un échelon.



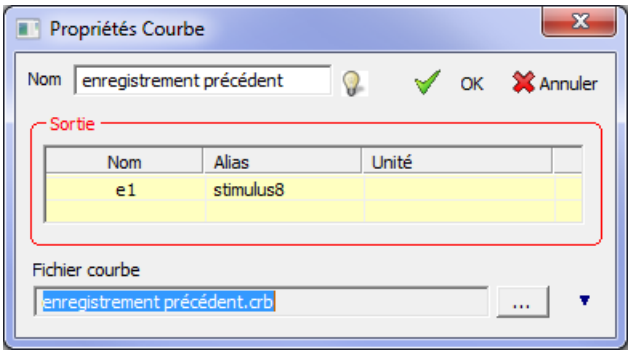
A retenir :

- un échelon d'amplitude high = une alternative complète ou une fonction Heaviside :
  if(t>tdelay, high, 0) <=> **high * H(t, tdelay)**
- une impulsion = 2 tests imbriqués ou le produit de 2 fonctions Heaviside complémentaires :
  if(t>tdelay,if(t<=tdelay+twidth,high,0),0) <=> **high*H(t, tdelay)* !H(t, tdelay+twidth)**
  A noter que le ! réalise la négation, ce qui signifie que lorsque t > tdelay+twidth alors H(t,tdelay+twidth) passe à 1 et !H(t,tdelay+twidth) passe à 0.

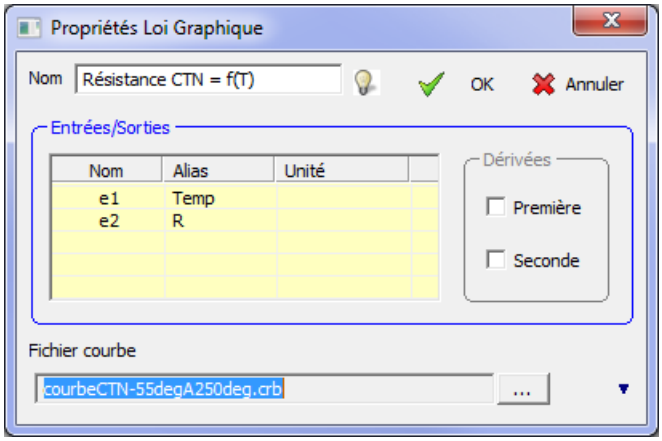
Les outils **Courbe** et **Loi Graphique**, expliqués :

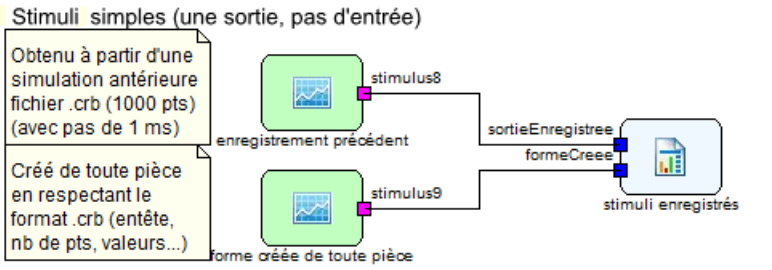
Nom (*Courbe*) : C'est automatiquement celui du fichier .crb. Il ne peut pas être modifié !

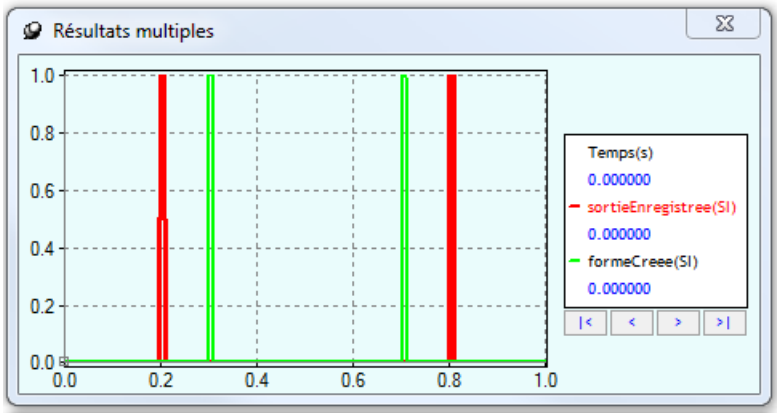
Sortie : Il est conseillé de compléter le champ alias sinon on y perd en lisibilité
Fichier courbe : Un fichier .crb, respectant la syntaxe et judicieusement placé dans le même dossier que le fichier Sinusphy.

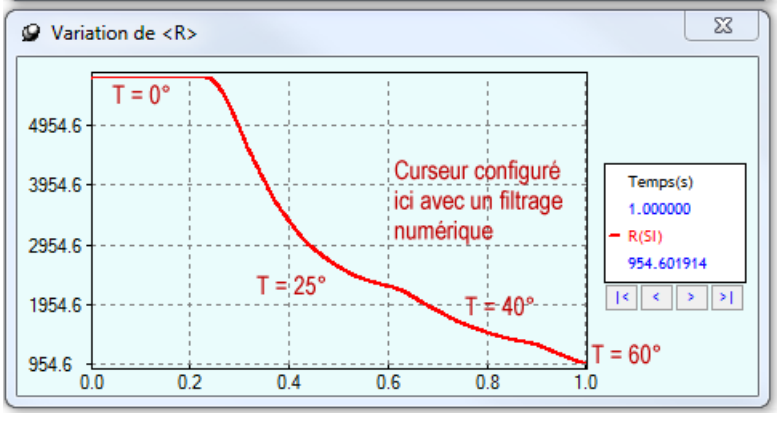


Nom (Loi Graphique) : Champ modifiable. S'il est vide, ce sera celui du fichier .crb utilisé.
Entrées/Sorties : Par défaut, il n'y a qu'une sortie. On peut y ajouter ses dérivées en cochant sur Première et/ou Seconde (*exemple : si le fichier .crb définit la position, on obtient directement la vitesse et l'accélération*). Il est donc recommandé de compléter les champs alias.
Fichier courbe : Un fichier .crb, respectant la syntaxe et judicieusement placé dans le même dossier que le fichier Sinusphy.





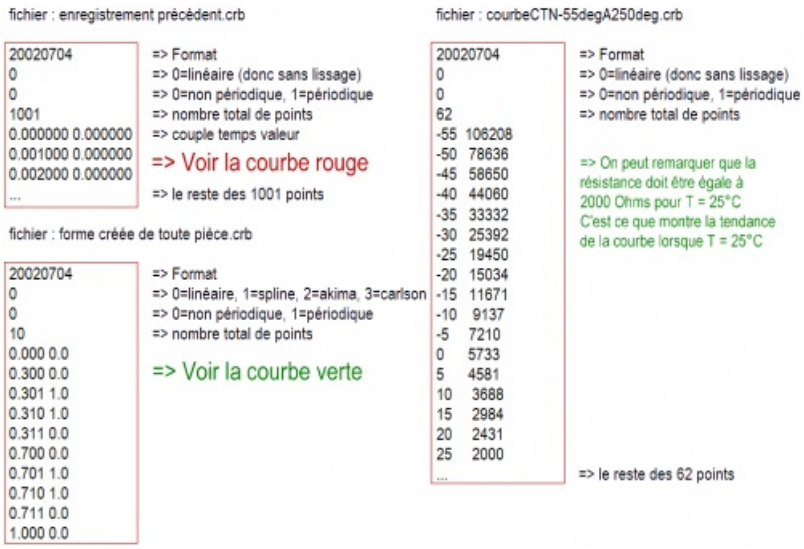




Quelques explications sur le format courbe .crb (lisible par tout éditeur de texte car c'est un simple fichier texte au standard de codage ANSI, classique sous Windows) :

Sinusphy intègre un outil nommé **Editeur de courbes** issu des logiciels de la même société : Mecaplan SW, Meca 3D...
Il est ainsi possible de définir graphiquement des signaux non périodiques ou périodiques.
Mais deux autres possibilités existent :

- L'enregistrement d'une courbe obtenue lors d'une simulation précédente
- La modification d'un fichier .crb existant à l'aide de tout éditeur de textes



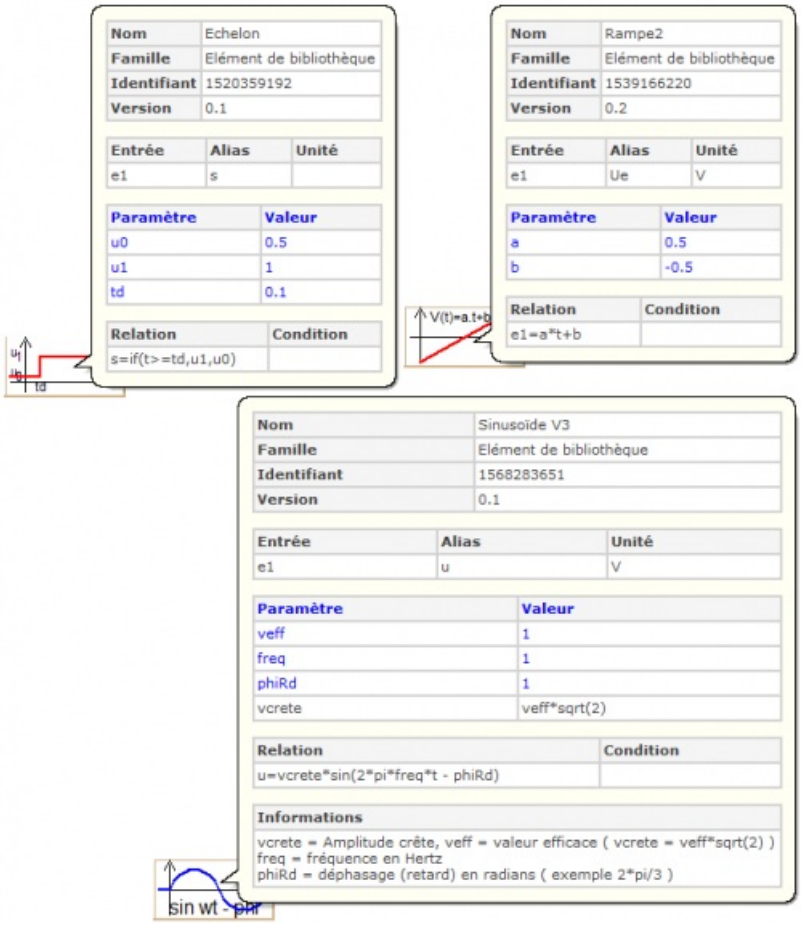
Que ce soit avec l'éditeur de courbes ou à l'aide d'un éditeur de textes, il est ainsi possible de créer toutes formes de signaux (*périodiques ou non*) pour simuler des variations d'une consigne ou de l'état d'un capteur en se passant de toute mise en équation.

● 2- Utilisation, modification ou création de composants de la bibliothèque

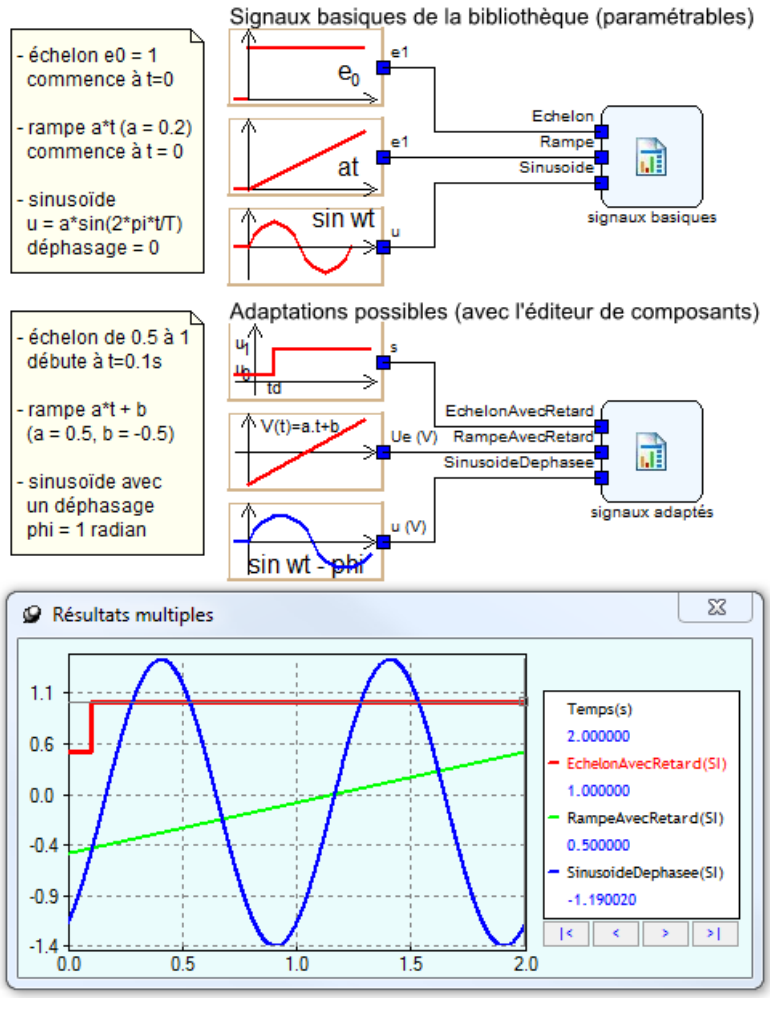
L'objet est, ici, de modifier des stimuli (éléments de la chaîne d'information) déjà existants sous forme de composants et d'en créer d'autres, plus adaptés aux besoins. Il faudra, ici, maîtriser différents types d'équations.

Les composants de la bibliothèque sont des composants standards et, par conséquent, sont modifiables avec l'utilitaire, appelé **Editeur de Composants** fourni avec Sinusphy.

- Dans la bibliothèque **Entrées** de Sinusphy, sont regroupées les 3 formes basiques de stimuli : l'échelon, la rampe et la sinusoïde.
- Il est possible de modifier ces formes basiques (et il est fortement conseillé de les sauvegarder sous d'autres noms), afin de les adapter aux besoins (dans les exemples décrits ci-après, de simples paramètres ont été ajoutés : un retard **td**, un décalage à l'origine **b** ou un déphasage **phiRd** et certains choix ont été effectués : prise en compte de la valeur efficace et de la fréquence au lieu de la période ou choix du volt comme unité).

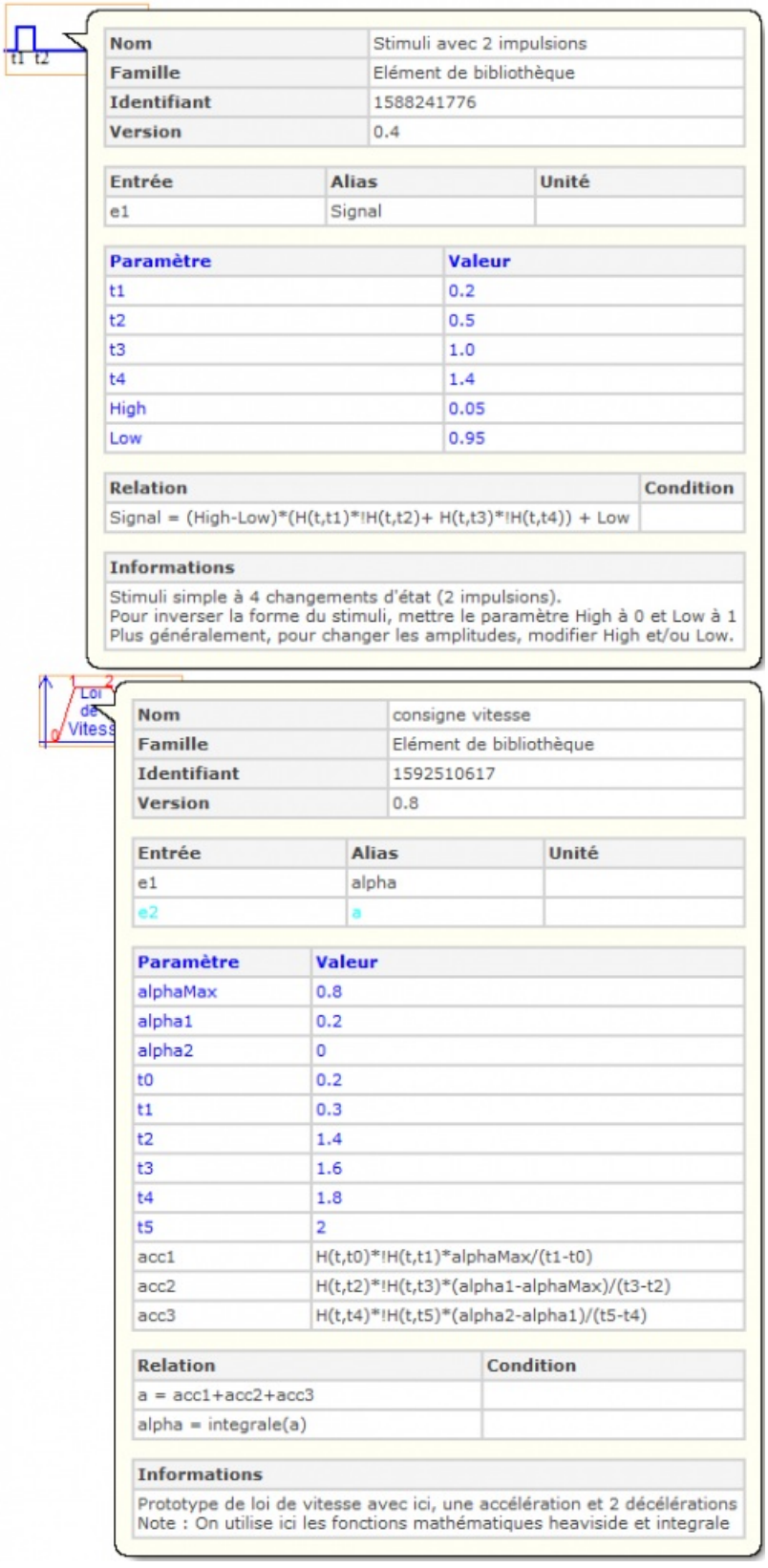


On visualise, ci-dessous, le retard **td** de 0.1 s pour l'échelon, le décalage à l'origine **b** de -0.5 V et le déphasage à l'origine **phiRd** de -1 rad (*-57,3 deg*)



Pour répondre davantage aux besoins, il est également possible de définir totalement des stimuli encore plus spécifiques (une série d'impulsions, une loi de vitesse...) afin de compléter la bibliothèque **Entrées** de Sinusphy. (ceci est détaillé dans les exemples ci-dessous)

- Tout comme pour l'outil "Formule", des paramètres internes peuvent être créés, afin d'être clairement identifiables et modifiables avant de lancer les calculs (pensez à documenter ces nouveaux composants réutilisables)



Nom	Stimuli avec 2 impulsions
Famille	Elément de bibliothèque
Identifiant	1588241776
Version	0.4

Entrée	Alias	Unité
e1	Signal	

Paramètre	Valeur
t1	0.2
t2	0.5
t3	1.0
t4	1.4
High	0.05
Low	0.95

Relation	Condition
Signal = (High-Low)*(H(t,t1)*!H(t,t2)+ H(t,t3)*!H(t,t4)) + Low	

Informations
Stimuli simple à 4 changements d'état (2 impulsions). Pour inverser la forme du stimuli, mettre le paramètre High à 0 et Low à 1 Plus généralement, pour changer les amplitudes, modifier High et/ou Low.

Nom	consigne vitesse
Famille	Elément de bibliothèque
Identifiant	1592510617
Version	0.8

Entrée	Alias	Unité
e1	alpha	
e2	a	

Paramètre	Valeur
alphaMax	0.8
alpha1	0.2
alpha2	0
t0	0.2
t1	0.3
t2	1.4
t3	1.6
t4	1.8
t5	2
acc1	H(t,t0)*!H(t,t1)*alphaMax/(t1-t0)
acc2	H(t,t2)*!H(t,t3)*(alpha1-alphaMax)/(t3-t2)
acc3	H(t,t4)*!H(t,t5)*(alpha2-alpha1)/(t5-t4)

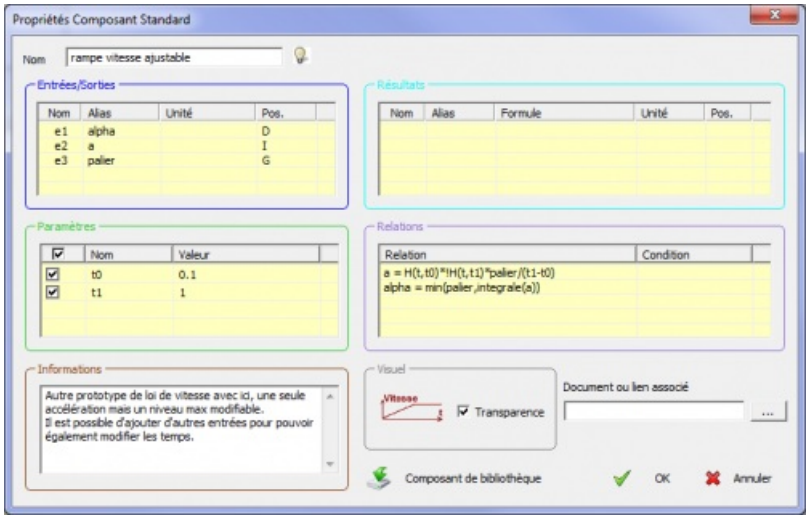
Relation	Condition
a = acc1+acc2+acc3	
alpha = integrale(a)	

Informations
Prototype de loi de vitesse avec ici, une accélération et 2 décélérations Note : On utilise ici les fonctions mathématiques heaviside et integrale

Remarquer ici l'utilisation systématique de la fonction Heaviside et de paramètres modifiables.
D'autre part, pour réaliser la loi de vitesse, il est nécessaire d'utiliser un second signal, ici, l'entrée e2, associée à

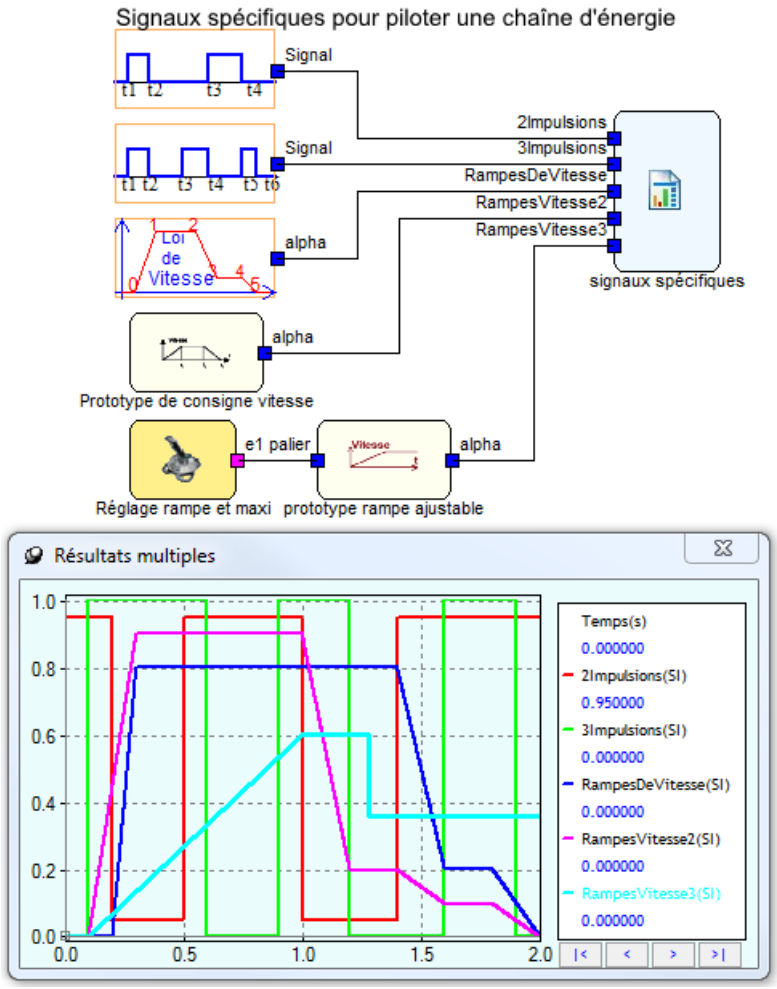
l’alias a. Cette entrée, volontairement cachée (définie comme interne) est absolument nécessaire pour garder en mémoire toutes les valeurs afin de pouvoir effectuer un traitement du signal comme la dérivée ou, ici, l’intégrale. L’entrée e2 étant, ici, une succession de constantes obtenues par Heaviside (représentant des coefficients directeurs : alpha/Δt) et de valeurs nulles, la fonction integrale, appliquée à ce signal, permettra d’obtenir les rampes et les paliers voulus.

- Par ailleurs, l’outil "Composant" permettant d’ajouter plusieurs entrées, il peut être intéressant d’intervenir, en cours de simulation, à l’aide de curseurs, sur les valeurs de certains paramètres (par exemple, l’ajustage de durées ou de niveaux).
- Ci-dessous la boite de dialogue du prototype de composant créé directement dans le schéma. On retrouve e2 (définie comme interne et associée à l’alias a), dont on déduit l’intégrale et une nouvelle entrée qui permet d’intervenir sur le paramètre **palier**.
  accélération : **a = H(t,t0)* !H(t,t1)*palier/(t1-t0)**
  consigne vitesse : **alpha = min(palier, integrale(a))**



Ci-dessous le résultat de la simulation de 3 nouveaux composants et de 2 prototypes de composants (appelés ainsi car créés directement dans le schéma et pas encore convertis en composants réutilisables). Pour le dernier (RampesVitesse3), le curseur a été positionné au départ sur 0,6 puis a été modifié, au clavier, à 0,36 aux alentours de 1.25 s.

A noter que les 2 lois de vitesse, ci-dessus, peuvent, également, être très rapidement obtenues par des fichiers courbe (extension .crb) dans lesquels, il suffira de n’entrer, à l’aide d’un éditeur de texte (le bloc notes peut suffire), que les 3 lignes d’entête, le compteur de points et les points correspondants (ici 8 à 10 points suffiront) ce qui montre que jusqu’ici, il est possible d’éviter l’écriture d’équations, ou la phase programmation/mise au point).



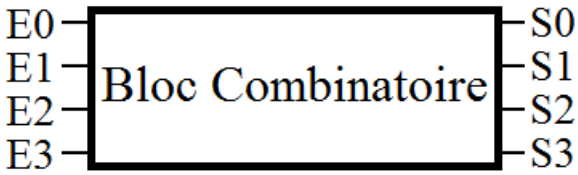
⚠ Pour compléter la bibliothèque de Sinusphy (ou modifier l'un de ses composants existant), il est nécessaire de lancer l'éditeur de composants avec les droits d'administrateur (clic droit puis Exécuter en tant qu'Administrateur), car la bibliothèque est située dans un dossier Programmes protégé. A l'inverse, si ces modèles sont créés pour être partagés, la sauvegarde dans la bibliothèque du poste de travail n'est pas la solution.

● 3- Création de composants de la bibliothèque et de prototypes de composants pour réaliser des séquenceurs

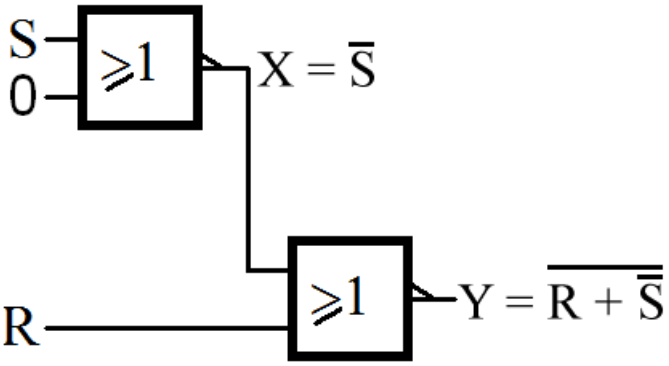
Comme indiqué précédemment, la modélisation d'un séquenceur n'est utile que si la création ou l'utilisation d'un stimulus ne suffit pas. Cela correspond généralement à la nécessité de surveiller l'état du système (prise en compte de la distance ou de la hauteur parcourue ou gestion des butées, par exemple)

Rappel : Différence entre logique combinatoire et logique séquentielle :

- En logique combinatoire, les sorties ne dépendent que de l'état des entrées. Il n'y a pas d'effet mémoire.
- On réalisera donc une fonction combinatoire à l'aide d'une équation logique reliant une sortie à une ou plusieurs entrées, que ce soit dans un composant Sinusphy ou dans un script Python.



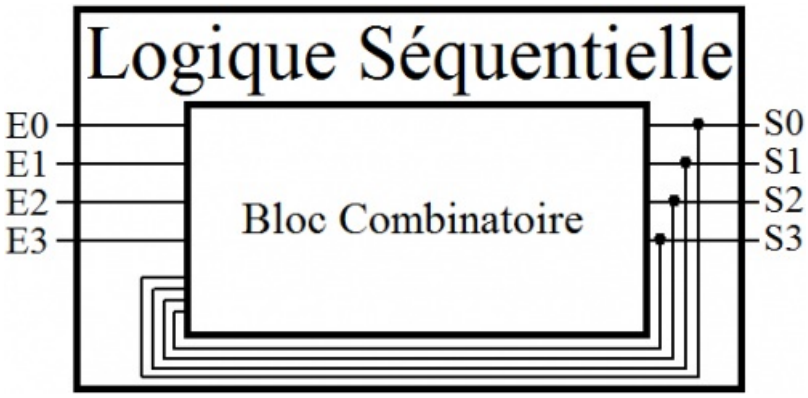
- Ci-dessous, l'équation logique de Y [**Y = /(R or /S) ⇔ Y = S and /R**] permet de forcer Y à 1 si S = 1 lorsque R = 0. Dans tous les autres cas, Y = 0. (Cette fonction logique, dans sa version simplifiée sous forme d'un ET et d'un inverseur, est parfois appelée fonction Inhibition).



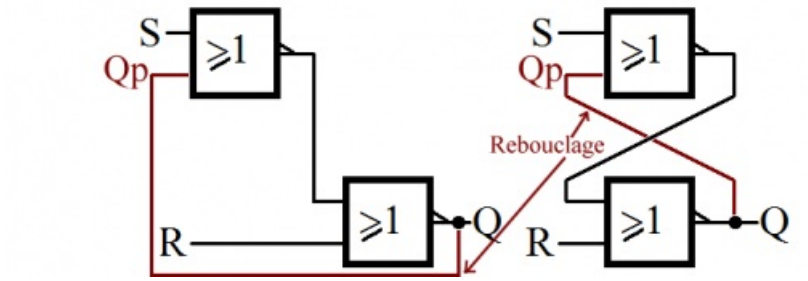
S	R	Y
0	0	0
0	1	0
1	0	1
1	1	0

(ici R prioritaire sur S)

- En logique séquentielle, les sorties dépendent également de l'état précédent du séquenceur, par conséquent, des anciennes valeurs des sorties. Il y a ainsi possibilité de voir différents états des sorties pour une même combinaison des entrées.
- Application : On cherchera à avoir un effet mémoire (voire un cycle de comptage ou autre), non pas par une ou des équations logiques mais par un ou plusieurs tests logiques (si tel événement, modifier, ainsi, telle sortie), que ce soit dans un composant Sinusphy ou dans un script Python.



- Ci-dessous, deux variantes du schéma du plus simple de tous les séquenceurs (*la mémoire RS*) réalisé avec des portes logiques NON OU. Comme on peut le voir, le schéma est une variante de la fonction précédente avec une nouvelle liaison permettant de tenir compte de l'état de la sortie. La sortie Q dépend alors de S, de R et de Qp (l'état précédent de Q) [**Q = /(R or /(S or Qp))**]. Cela permet de tenir compte de l'ancienne valeur de Q lorsque S et R ne sont pas actifs. Cet état sera appelé : **l'état mémoire**. (Qp = 1 si seul S était à 1 avant de passer à 0 ou Qp = 0 si R était seul à 1 avant de passer à 0). Il apparaît ainsi la notion d'état précédent et cette notion servira pour gérer les entrées, les sorties et les variables internes.



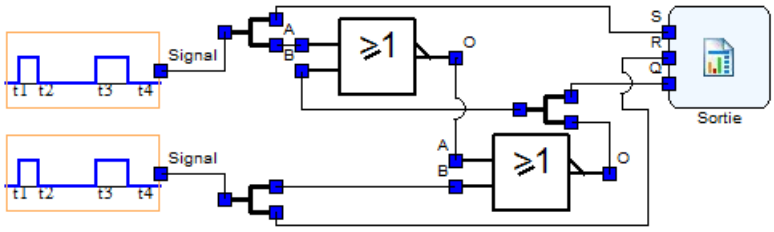
S	R	Q	*Table de la mémoire RS*
0	0	Qp	*(état précédent)*
0	1	0	*(mise à 0)*
1	0	1	*(mise à 1)*
1	1	0	*(ici R prioritaire sur S)*

Modélisation, simulation et analyse de fonctionnement de la mémoire RS :

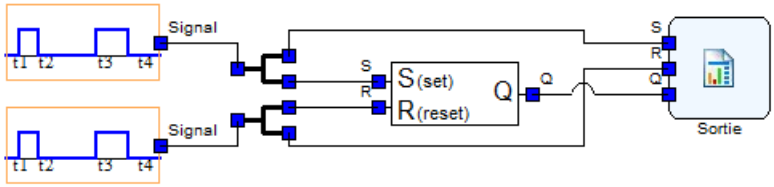
- Soit le cahier des charges : « Démarrer le moteur par un appui sur un bouton **Marche** et arrêter le moteur par un appui sur le bouton **Arrêt d'Urgence** (ou tout autre événement lié à l'état de la machine comme un fin de course et modélisable par un simple composant comparant (utiliser un test ou Heaviside) l'intégrale de la vitesse à une valeur limite...). Le moteur doit rester dans l'état imposé par le dernier ordre reçu, ce qui nécessite d'utiliser le phénomène mémoire vu précédemment. »
- Nous allons modéliser, simuler et analyser le fonctionnement de diverses modélisations d'une mémoire RS. Pour des raisons évidentes de sécurité, il faudra bien vérifier que l'**Arrêt d'Urgence ⇔ Reset**, sera bien prioritaire sur le bouton **Marche ⇔ Set**.
- Pour les 3 modélisations suivantes, les stimuli seront définis ainsi :
  - **entrée S** : impulsion n°1 de 0.2 s à 0.25 s et impulsion n°2 de 1 s à 1.4 s
  - **entrée R** : impulsion n°1 de 0.6 s à 0.65 s et impulsion n°2 de 1.2 s à 1.3 s

Trois exemples :

- La solution câblée classique, vue auparavant, à deux 2 portes logiques **NON OU** (*Nor Gate*). Ce n'est, évidemment, pas la solution préconisée pour la suite mais elle est toujours utilisée, tout comme le câblage auto maintien (qui réalise la même fonction mémoire, à l'aide d'un relais et de contacts de boutons poussoirs, dans la partie commande de machines en électrotechnique) et mérite d'être étudiée pour aider à la compréhension de la suite.



- La réalisation d'un composant spécifique **mémoire RS**. Cette mémoire RS sera commandée par les niveaux "high" des entrées **S** et **R**. Le cœur de la mémoire RS est une imbrication de 2 tests complets (du type si... alors... sinon...) avec l'utilisation de la valeur précédente de la sortie **Q** lorsque **R** et **S** ne sont pas à l'état actif. Pour rendre l'état de **R** prioritaire sur celui de **S**, il suffit, ici, de commencer par tester l'entrée **R** afin de ne tester l'entrée **S** que si l'entrée **R** est inactive.



On voit, ci-dessous, comment se fait la traduction des tensions (niveau haut défini par l'utilisateur) en niveaux logiques 1 ou 0 grâce à la fonction Heaviside ainsi que la solution pour réagir à **R** et à **S** ou maintenir l'état précédent. A noter qu'ici, comme dans d'autres langages, la valeur "1" est équivalente au booléen **True**, et par conséquent :

Q = Vh * if(R_, 0, if(S_, Vh, Q)) ⇔ Q = Vh * if(R_==1, 0, if(S_==1, Vh, Q))

Nom	Mémoire RS (R prioritaire sur S)
Famille	Elément de bibliothèque
Identifiant	1593179745
Version	0.4

Entrée	Alias	Unité
e1	S	
e2	R	
e3	Q	

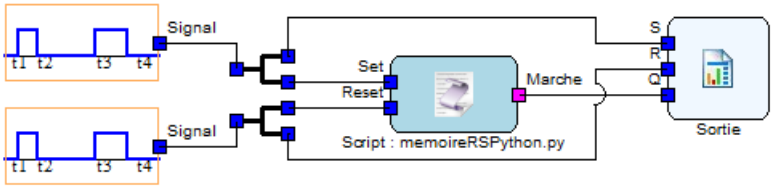
Paramètre	Valeur
Vh	1
S_	H(S,Vh/2)
R_	H(R,Vh/2)

Relation	Condition
Q=Vh*if(R_,0,if(S_,Vh,Q))	

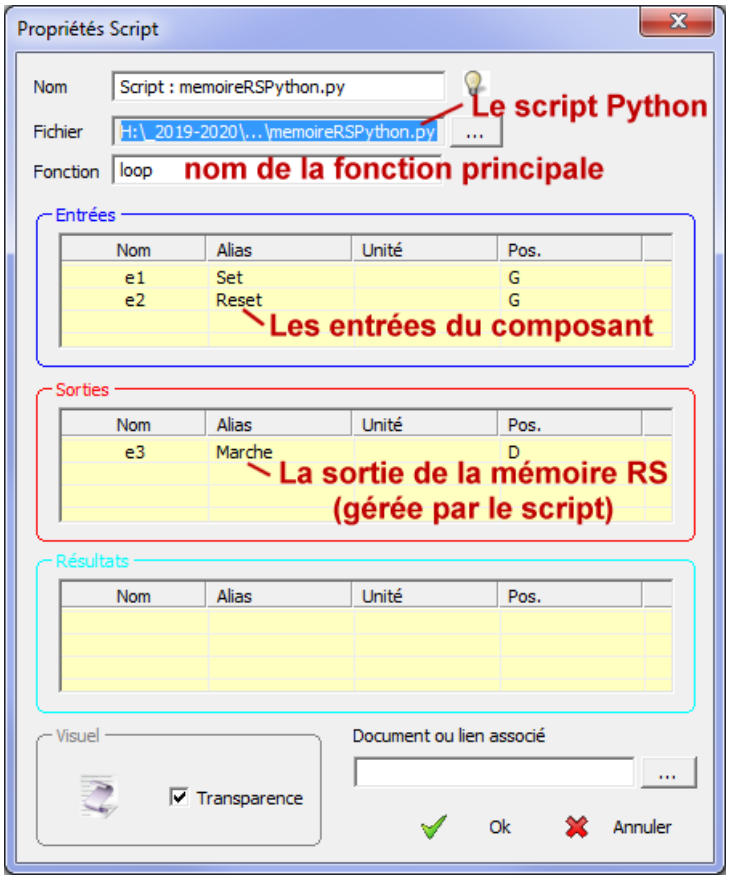
Informations

C'est la fonction séquentielle la plus élémentaire :
R = Reset (mise à 0 de la sortie Q),
S = Set (mise à 1 de la sortie Q).
Deux versions existent. Ici, R est prioritaire sur S
On voit que le phénomène mémoire est obtenu dans l'équation de Q par : Q = f(R_,S_, Q)
Note : S_ et R_ sont des valeurs logiques 0 ou 1

- La réalisation d'un simple script **memoireRSPython.py**. Cette mémoire RS sera commandée par les niveaux "high" des entrées **Set** et **Reset**. Le cœur de ce script est, ici, une succession de 2 alternatives simples. Cette solution a l'avantage d'être très lisible. Pour rendre l'état de **Reset** prioritaire sur celui de **Set**, il suffira, ainsi, de tester **Reset** en dernier. A noter que la solution de deux alternatives imbriquées reste parfaitement valable (en testant, alors, Reset en premier...), surtout, que sur le papier, elle devrait être plus rapide mais cela se fait au prix de plusieurs indentations et d'une perte de lisibilité.



On voit, ci-dessous, comment se déclarent les entrées et sorties dans le composant Script et comment le lien se fait avec le script et la fonction principale appelée à chaque pas de calcul (cette fonction principale a été volontairement appelée, ici, **loop** en référence à la structure de base d'un programme sous Arduino et afin de conserver la notion d'unique "boucle" obtenue par l'appel de Sinusphy à chaque pas).



Et dans le script, ci-dessous, on repère les points importants :

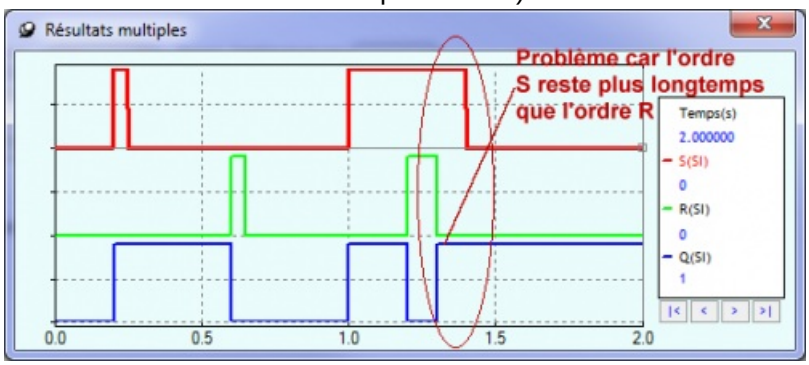
1. La lecture des signaux (*set = value("Set")*)
2. La conversion (En fait, il n'est pas nécessaire de la gérer car c'est automatique en Python car le booléen **True** correspond à une valeur non nulle)
3. Le cœur du script (Les 2 tests des variables **set** et **reset**, sous forme d'alternatives simples, suffisent car le maintien de l'état précédent s'obtient par défaut si aucun test n'est validé).

```
#-*-coding:-latin-1-*-
#-Gestion-d'une-simple-mémoire-RS-(écrit-en-Python-2.7-pour-Sinusphy)
#-But-:-Valider-un-1er-script-avant-de-l'implanter-dans-le-µcontrôleur-du-système

#-Incorporer-les-composants-utilisés-dans-les-bibliothèques
#-from-math-import-...-(exemple-:-ceil,-floor,-pi,-pow,-sqrt...)
from-sinusphy-import-value,-setvalue,-timer-#-Existent-aussi-timeout,trace-et-duration

#-Variable-globale-marche-pour-mémoriser-l'état-du-séquenceur-(la-mémoire-RS)
marche-=-0

#-Fonction-principale-appelée-par-Sinusphy-à-chaque-pas-de-calcul
#-Nommer,-dans-l'ordre-les-entrées-et-les-sorties-en-respectant-la-CASSE
def-loop-(Set,-Reset,-Marche):
····global-marche··············#-Autorise-la-modification-de-la-variable-globale

····#-ENTRÉES-:-Lire-les-valeurs-fournies-par-Sinusphy
····set···=-value("Set")·····#-Lecture-de-la-valeur-fournie-par-Sinusphy
····reset-=-value("Reset")····#····"·············"·····"···········"

····#-***-Si-aucun-test-validé,-l'état-du-séquenceur-restera-inchangé-!
····if-set:···················#-set-commande-la-mise-en-marche-du-moteur
········marche-=-1···········#-Imposer-la-mise-en-marche-du-moteur

····if-reset:·················#-reset-est-prioritaire-car-testé-en-dernier
········marche-=-0···········#-Annuler-la-commande-du-moteur
····#-***-Fin-de-la-partie-transférable-dans-le-µcontrôleur-du-système

····#-SORTIE-:-Placer-la-valeur-sur-la-sortie-pour-Sinusphy
····setvalue("Marche",-marche)-#-Renvoyer-la-valeur-à-Sinusphy
····return-1-#-Renvoie-1-pour-signaler-que-le-traitement-de-loop-s'est-bien-déroulé
```

A noter que les 3 solutions donnent exactement le même résultat (jusqu'à l'état initial de Q qui égal à 0).

- Pour le **composant mémoire RS**, c'est SinusPhy qui initialise la variable Q à 0.
- Pour le **script Python**, la variable globale **marche** associée à la sortie doit être déclarée et initialisée à 0.
- Pour les **portes logiques NON OU**, afin d'obtenir l'état initial voulu, l'une des 2 portes logiques a été simplement optimisée (on peut toutefois vérifier que, s'il y a eu une première simulation, l'état mémoire initial

est logiquement le dernier état obtenu à la simulation précédente).



Note : Comme on peut le voir, à l'instant t = 1.3 s, cette solution basique montre tout de même ses limites car on peut voir que lorsque **R** repasse à 0 avant **S** après qu'ils aient été tous les deux à 1, la sécurité n'est pas assurée car, visiblement, le moteur redémarre... Il est vrai que le cahier des charges n'abordait pas le temps d'appui sur les boutons !
Nous allons voir comment y remédier au paragraphe suivant en prenant en compte les fronts (c'est à dire les variations des entrées) au lieu des niveaux !

● 4- Création d'un séquenceur réagissant sur fronts, sous forme de composant SinusPhy ou de script Python

Modélisation, simulation et analyse de fonctionnement d'un séquenceur commandé sur fronts :

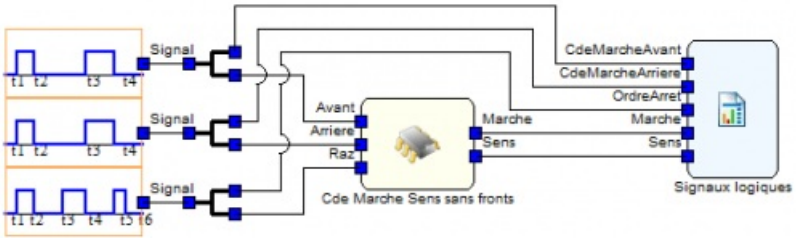
- Soit le nouveau cahier des charges : « Démarrer le moteur dans le sens horaire par un appui sur le bouton **Avant** ou dans le sens anti-horaire par un appui sur le bouton **Arrière**. A tout moment, le moteur peut être arrêté par appui sur le bouton **Raz** (*ou tout autre événement lié à l'état de la machine...*). Le moteur ne peut changer de sens que s'il a été auparavant arrêté. Le temps d'appui sur **Avant** et sur **Arrière** ne devra pas intervenir ; seul le moment où on commence à appuyer sur ces boutons devra être pris en compte. »
- Une détection des fronts est nécessaire mais pour la comprendre, nous allons modéliser, dans un premier temps, ce nouveau séquenceur sans prendre en compte les fronts puis nous allons ajouter la détection des fronts sur **Avant** et sur **Arrière**.
- Afin de visualiser facilement la différence, les stimuli seront les mêmes.
- Un exemple d'évolution de ce séquenceur sera proposé par l'ajout d'une sortie permettant de commander le moteur sans à-coups.

Note : Compte tenu de l'utilisation de l'effet mémoire (réalisé soit par un composant soit un script) et afin de toujours démarrer dans les mêmes conditions, il conviendra de toujours fermer la boite de dialogue de simulation afin de forcer la réinitialisation des états pour une nouvelle simulation.

Description des stimuli communs :

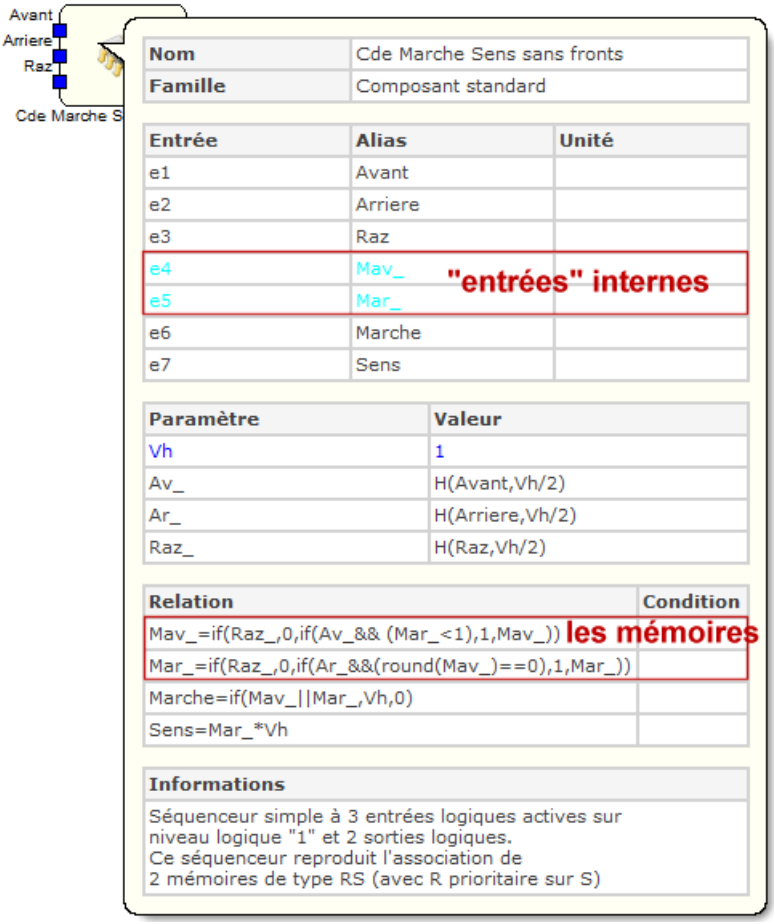
- **Signal Avant** : impulsion n°1 de 1 s à 1.5 s et impulsion n°2 de 6.5 s à 7 s
- **Signal Arriere** : impulsion n°1 de 4.75 s à 5.5 s et impulsion n°2 de 7.5 s à 9.5 s
- **Signal Raz** : impulsion n°1 de 2.5 s à 2.6 s, impulsion n°2 de 5 s à 5.1 s et impulsion n°3 de 8 s à 8.1 s

Première solution : Réalisation, à l'aide d'un composant Sinusphy, du séquenceur sans la prise en compte des fronts :
Schéma épuré utilisant les 3 stimuli :



Propriétés du composant :

- Le choix a été fait, ici, d'utiliser, comme variables intermédiaires, deux entrées internes Mav_ et Mar_ (entrées volontairement invisibles) afin d'obtenir les deux effets mémoire.
- **Equation de Mav**_ : Mav_=if(Raz_,0,if(Av_&& (Mar_<1),1,Mav_))
  => Si Raz est à 1, alors Mav_ est à 0, sinon : si ordre Avant et pas marche arrière, alors Mav_ = 1 sinon Mav_ ne change pas (*noter le choix du test Mar_ < 1. On aurait pu choisir : Mar_ < 0.5*)
- **Equation de Mar**_ : Mar_=if(Raz_,0,if(Ar_&&(round(Mav_)==0),1,Mar_))
  => Si Raz est à 1, alors Mar_ est à 0, sinon : si ordre Arrière et pas marche avant, alors Mar_ = 1 sinon Mar_ ne change pas (*noter le choix d'une autre forme de test équivalente à Mar_ < 1 mais plus technique : round(Mav_)==0 car la valeur initiale de Mav_ n'est pas un entier et la comparaison à l'entier 0 ne fonctionnera pas*)



Nom	Cde Marche Sens sans fronts
Famille	Composant standard

Entrée	Alias	Unité
e1	Avant	
e2	Arriere	
e3	Raz	
e4	Mav_	
e5	Mar_	
e6	Marche	
e7	Sens	

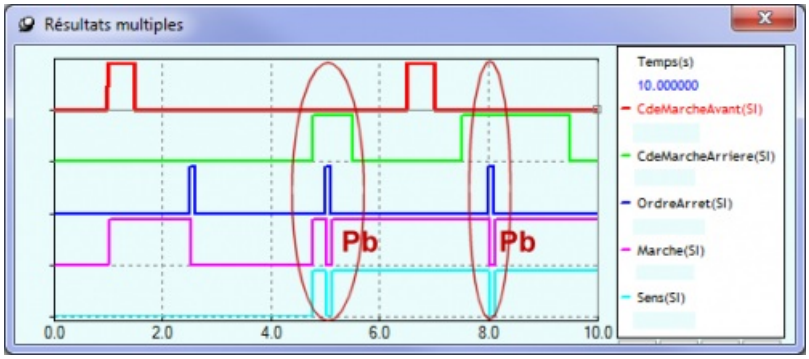
Paramètre	Valeur
Vh	1
Av_	H(Avant,Vh/2)
Ar_	H(Arriere,Vh/2)
Raz_	H(Raz,Vh/2)

Relation	Condition
Mav_=if(Raz_,0,if(Av_&& (Mar_<1),1,Mav_))	
Mar_=if(Raz_,0,if(Ar_&&(round(Mav_)==0),1,Mar_))	
Marche=if(Mav_||Mar_,Vh,0)	
Sens=Mar_*Vh	

Informations

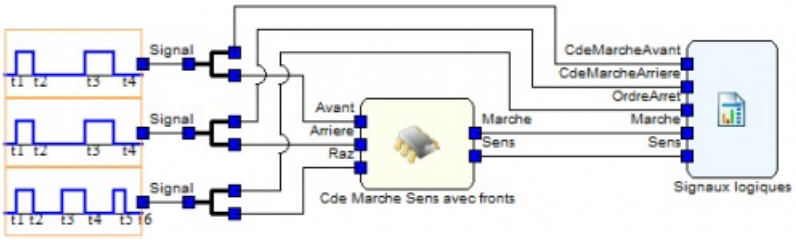
Séquenceur simple à 3 entrées logiques actives sur niveau logique "1" et 2 sorties logiques.
Ce séquenceur reproduit l'association de 2 mémoires de type RS (avec R prioritaire sur S)

Chronogrammes :
On remarque également l'existence du problème (*qui se reproduit ici aux environs 5 s et 8 s*), lié à la durée des signaux. Et ce problème engendre ensuite, vers 6.5 s, l'impossibilité de prendre en compte la commande de marche avant.



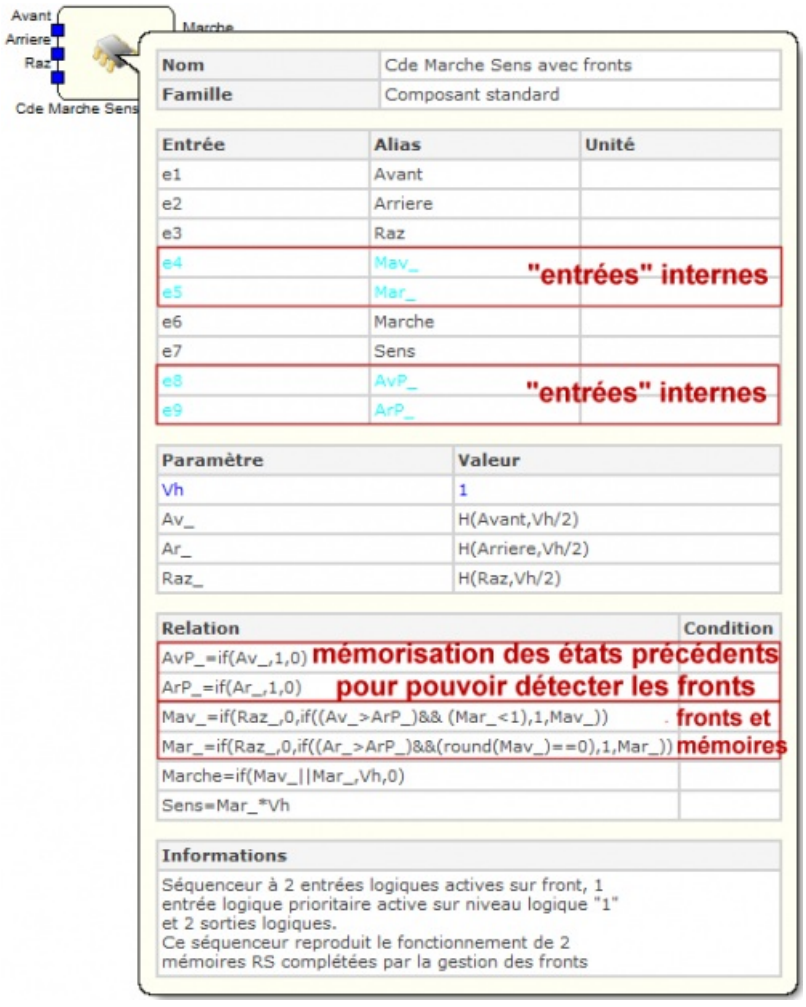
Deuxième solution : Réalisation, à l'aide d'un composant Sinusphy, du séquenceur avec la prise en compte des

fronts :
C'est le même schéma épuré utilisant les 3 stimuli. Quelques aménagements seront faits en interne pour tenir compte des fronts :

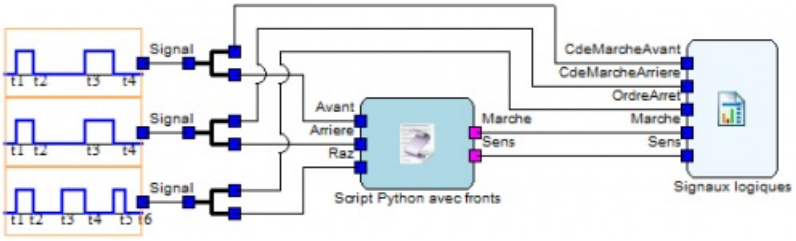


Nouvelles propriétés du composant :

- Le choix a été fait, ici, d'ajouter, comme variables supplémentaires, deux entrées internes AvP_ et ArP_ (*entrées volontairement invisibles*) afin d'obtenir les deux effets mémoire supplémentaires pour détecter des variations que l'on appellera des fronts montants (*la nouvelle valeur de l'entrée devant être plus grande (1) que son ancienne valeur (0)*).
- **Equation de AvP_** : AvP_=if(Av_,1,0)
  => Il peut sembler étrange que AvP_=if(Av_,1,0) soit si différente de AvP_=Av_. Ceci est lié au mode de calcul ACAUSAL qui recherche un équilibre et, par conséquent, ne fait pas de différence entre : AvP_=Av_, Av_=AvP_ ou encore 0=AvP_-Av_... alors que l'utilisation d'un test force la copie de Av_ vers AvP et mémorise ainsi Av_.
- **Equation de ArP_** : ArP_=if(Ar_,1,0)
  => Même remarque liée au mode de calcul ACAUSAL
  -* **Modification des équations de Mar_ et Mav_** : ...if(Av_&&... a été remplacé par if((Av_>AvP_)&&... et ...if(Ar_&&... a été remplacé par if((Ar_>ArP_)&&...
  => Ce n'est plus sur l'état 1 mais sur le passage logique de 0 à 1 (*que l'on appellera front montant*), que le test est validé.

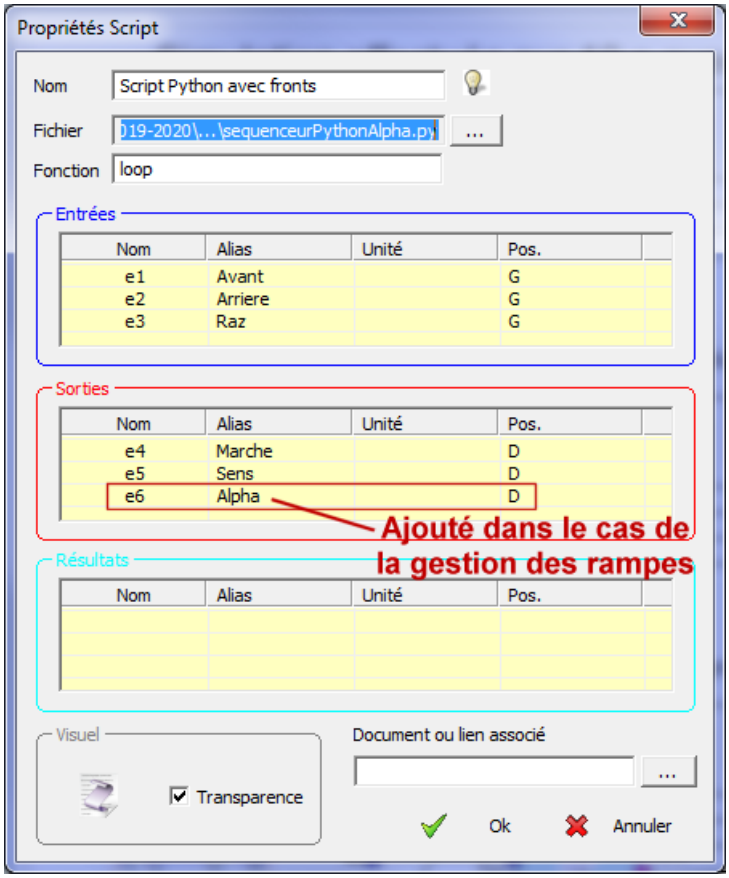


Variante de cette deuxième solution : Réalisation, à l'aide d'un script Python, du séquenceur avec la prise en compte des fronts :
C'est le même schéma épuré utilisant les 3 stimuli et un composant Script :



Propriétés du composant script :
On notera la même organisation que pour l'exemple précédent simulant une mémoire RS.

⚠ Il convient de respecter l'écriture des noms (*différenciation entre MAJUSCULES et minuscules*) et le nombre d'entrées et de sorties. Par exemple, si on ajoute, ici, une sortie Alpha, celle-ci doit être gérée dans le script sous peine de blocage et la réciproque est vraie.

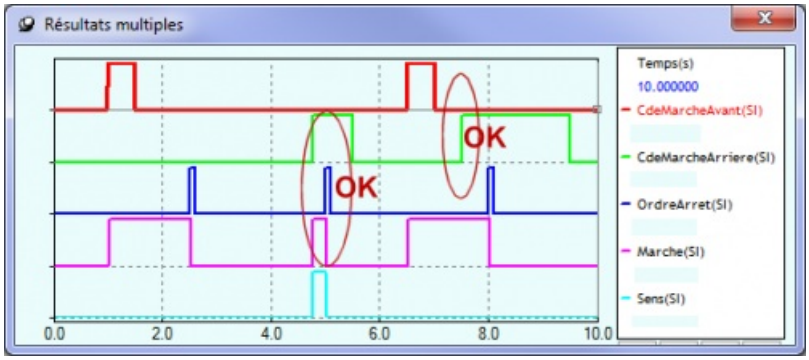


Script associé :
Ce script reprend la structure vue avec la mémoire RS.
Noter :

- La première ligne de la fonction loop : **def loop (Avant, Arriere, Raz, Marche, Sens) :**
- La gestion des fronts montants obtenue par les 2 syntaxes : **if av > avp...** et **if ar > arp...** associées aux 2 lignes situées en fin de script **avp = av** et **arp = ar** qui recopient les valeurs actuelles des entrées afin de pouvoir comparer au prochain pas.

```
#-*- coding: latin-1 -*-
#Gestion d'un simple séquenceur (écrit en Python 2.7 pour Sinusphy)
#But : Valider une portion de code avant de l'implanter dans un µcontrôleur réel

#Incorporer les composants utilisés dans les bibliothèques
#from math import ... (exemple : ceil, floor, pi, pow, sqrt...)
from sinusphy import value, setvalue, timer #et aussi timeout, trace et duration

#Variables globales : av & avp ou ar & arp pour gérer les fronts et les sorties
av, avp, ar, arp, mAvant, mArriere = 0, 0, 0, 0, 0, 0

#Fonction principale appelée par Sinusphy à chaque pas de calcul
#Nommer, dans l'ordre les entrées et les sorties en respectant la CASSE
def loop (Avant, Arriere, Raz, Marche, Sens):
    global av, avp, ar, arp, mAvant, mArriere

    av = value("Avant")          #Lecture de la valeur actuelle dans Sinusphy
    ar = value("Arriere")        #    "            "       "       "
    raz = value("Raz")           #    "            "       "       "

    #*** Début de la partie pouvant être transférée dans un µcontrôleur réel
    if av > avp and not mArriere: #Front montant et pas de marche arrière ?
        mAvant = 1                #Imposer la marche avant

    if ar > arp and not mAvant:   #Front montant et pas de marche avant ?
        mArriere = 1              #Imposer la marche arrière

    if raz:                       #Sur niveau, prioritaire car testé en dernier
        mAvant = 0                #Annuler la marche avant
        mArriere = 0              #Annuler la marche arrière

    #*** Si aucun test validé, l'état du séquenceur est inchangé (mémoire)

    avp = av                      #Mémorise la valeur précédente pour la suite
    arp = ar                      #    "          "       "       "
    #*** Fin de la partie transférable dans un µcontrôleur réel

    #SORTIES : Adapter et placer les valeurs sur les sorties pour Sinusphy
    setvalue("Marche", mAvant or mArriere) #Renvoyer la valeur à Sinusphy
    setvalue("Sens", mArriere)             #    "          "       "
    return 1 #Renvoie 1 pour signaler que loop s'est déroulé jusqu'au bout
```

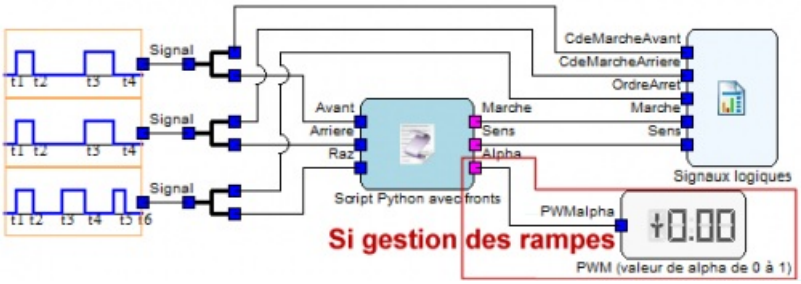
Chronogrammes identiques obtenus avec le composant Sinusphy ou le script Python :
On vérifie que le problème, lié à la durée des appuis, a bien disparu, que la commande de marche avant est bien prise en compte à l'instant 6.5 s et que la marche arrière est logiquement ignorée vers 7.5 s.



Dernière variante de cette solution : La puissance d'un script Python permet logiquement d'ajouter des fonctionnalités :

Pour l'exemple, nous allons ajouter une sortie Alpha qui va pouvoir commander la chaîne d'énergie (variateur -> moteur->...) de manière progressive. Nous allons utiliser le terme de rampes d'accélération et de décélération.
La sortie Alpha pourra varier entre 0 et 1 et correspondra à un rapport cyclique variant entre 0% et 100% qui sera relié à un variateur de vitesse (*un hacheur par exemple*).
Il sera possible de modifier, dans le script, les valeurs alphaMax et duree afin d'obtenir l'effet voulu.

Schéma légèrement modifié pour sortir et afficher indépendamment la valeur Alpha qui sera le seul signal non binaire.



Pour le script correspondant, on notera l'utilisation de :

- La fonction timer() disponible dans la bibliothèque sinusphy pour Python. Elle permet de connaître le temps depuis le début de la simulation et de ne pas dépendre du pas de calcul.
- Les fonctions min et max de Python permettant de limiter la variable alpha à l'intervalle autorisé [0, 1].
- La variable alphap permettant de mémoriser la valeur maximale atteinte par alpha.

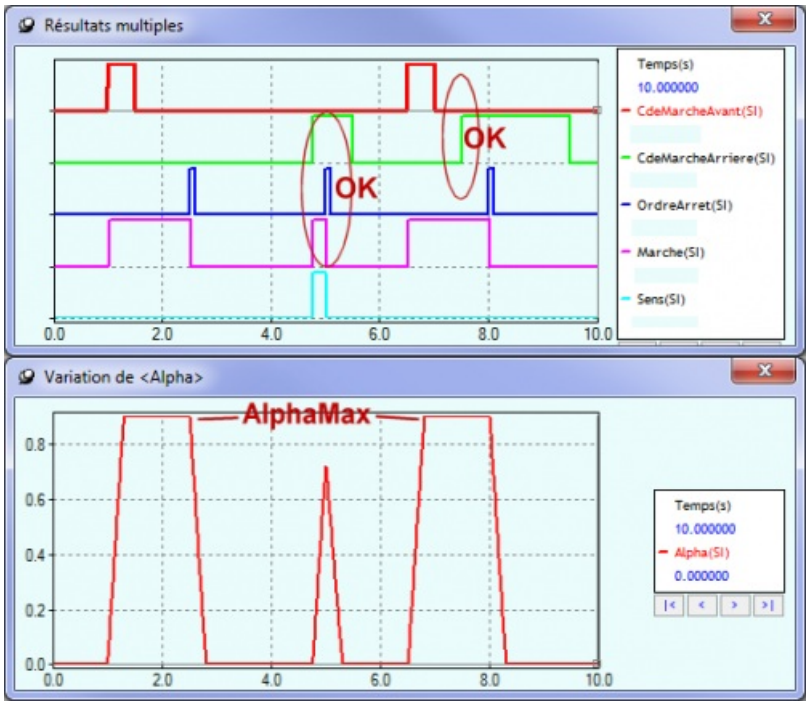
Le script s'étoffe peu à peu tout en gardant la même structure et une logique séquentielle très lisible.

```
# -*- coding: latin-1 -*-
# Gestion d'un simple séquenceur (écrit en Python 2.7 pour Sinusphy)
# But : Valider une portion de code avant de l'implanter dans un µcontrôleur réel

# Incorporer les composants utilisés dans les bibliothèques
# from math import ... (exemple : ceil, floor, pi, pow, sqrt...)
from sinusphy import value, setvalue, timer # et aussi timeout, trace et duration

# Variables globales : Pour gérer les fronts, les sorties TOR et alpha (PWM)
av, avp, ar, arp, mAvant, mArriere, alpha, alphap = 0, 0, 0, 0, 0, 0, 0, 0

# Paramètres pour définir la rampe d'accélération
alphaMax = 0.9                  # Valeur limite de alpha (rapport cyclique max)
duree = 0.3                     # Durée rampes accélération et décélération
tref = 0.0                      # Temps de référence modifié avant chaque rampe

# Fonction principale appelée par Sinusphy à chaque pas de calcul
# Nommer, dans l'ordre les entrées et les sorties en respectant la CASSE
def loop (Avant, Arriere, Raz, Marche, Sens, Alpha):
    global av, avp, ar, arp, mAvant, mArriere, alpha, alphap, tref

    av = value("Avant")         # Lecture de la valeur actuelle dans Sinusphy
    ar = value("Arriere")       #    "            "       "     "           "
    raz = value("Raz")          #    "            "       "     "           "

    # *** Début de la partie pouvant être transférée dans un µcontrôleur réel
    if av>avp and not mArriere: # Front montant et pas de marche arrière ?
        mAvant = 1              # Imposer la marche avant
        tref = timer()          # Initialiser le temps de référence

    if ar>arp and not mAvant:   # Front montant et pas de marche avant ?
        mArriere = 1            # Imposer la marche arrière
        tref = timer()          # Initialiser le temps de référence

    if raz:                     # Sur niveau, prioritaire car testé en dernier
        if mAvant or mArriere:
            mAvant = 0          # Annuler la marche avant
            mArriere = 0        # Annuler la marche arrière
            tref = timer()      # Initialiser le temps de référence
            alphap = alpha      # Mémoriser la valeur initiale de décélération

    # *** Si aucun test validé, l'état du séquenceur est inchangé (mémoire)

    # *** Gestion des rampes d'accélération et de décélération
    if mAvant or mArriere:      # Augmenter alpha tant que le moteur tourne
        alpha = min(alphaMax, (timer()-tref)*alphaMax/duree)
    else:
        alpha = max(0, alphap-(timer()-tref)*alphap/duree)

    avp = av                    # Mémorise la valeur précédente pour la suite
    arp = ar                    #    "           "       "     "     "
    # *** Fin de la partie transférable dans un µcontrôleur réel

    # SORTIES : Adapter et placer les valeurs sur les sorties pour Sinusphy
    setvalue("Marche", mAvant or mArriere) # Renvoyer la valeur à Sinusphy
    setvalue("Sens", mArriere)  #    "           "       "     "
    setvalue("Alpha", alpha)    #    "           "       "     "
    return 1 # Renvoie 1 pour signaler que loop s'est déroulé jusqu'au bout
```

La variable alpha évolue ainsi :

- Première accélération jusqu'à alphaMax fixé à 0.9 (soit 90%)

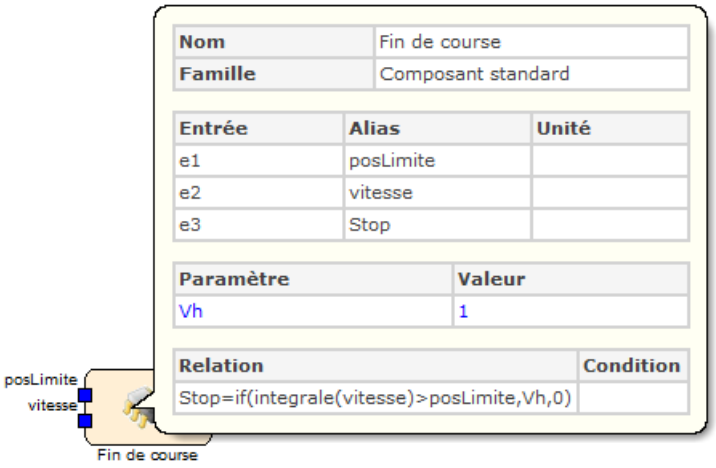
- Décélération jusqu'à arrêter le moteur (0%)
- Seconde accélération jusqu'à environ 70% du fait d'une durée trop courte...

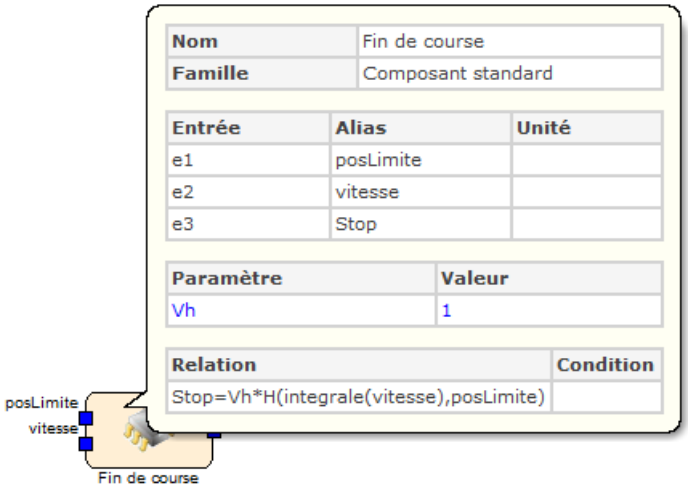


En conclusion, pour l'instant, avec les différentes solutions proposées :

- Il est possible de commander une chaîne d'énergie (*nous avons vu différentes façons de créer des stimuli avec des curseurs, des fichiers .crb définissant des courbes particulières, des composants basiques existants, ou des modifications de composants existant et, enfin, de créations de composants spécifiques*).
- Il est également possible de commander une chaîne d'énergie et de changer de mode de fonctionnement en fonction de l'état du système (*pour cela, il faut commencer par modéliser le capteur adapté et, ensuite, modéliser un séquenceur avec une entrée spécifique pour récupérer l'état de ce capteur*).

Voici, comme exemple de création d'un simple capteur à sortie TOR : un **capteur fin de course** :
De part sa technologie (*mécanique, optique ou magnétique*) et son emplacement, il détermine une position limite et le signale par un changement d'état logique.
Le modèle de ce **capteur fin de course** doit commencer par déduire la position actuelle du système (*que ce soit un angle de rotation ou un déplacement linéaire*) des évolutions de la vitesse correspondante (*on fait appel à la fonction integrale*), puis il doit comparer cette position à une valeur limite (*dans le modèle, cela peut être une constante ou une entrée dont la valeur est modifiable via un curseur*) pour donner un résultat binaire compatible avec les exemples de séquenceurs précédents (*ci-dessous, 2 versions, l'une avec un test, l'autre avec Heaviside. Toutes deux renvoient, ici, un niveau 1 si la limite est atteinte*)





Par contre, nous n'avons pas encore abordé les techniques pour moduler la commande de la chaîne d'énergie afin de maintenir, par exemple, la vitesse du moteur, quelque soit la charge. Ceci sera développé dans le prochain article. On parlera alors de système bouclé. Les notions de consigne, de chaîne directe, de chaîne de retour, d'asservissement, de précision, de stabilité et de correcteur PID seront abordées.

Les modélisations des composants complémentaires pourront être complètement faites sous forme de simples composants Sinusphy. Par exemple, le correcteur PID, chargé d'assurer la précision et la stabilité, pourra être réalisé par un composant Sinusphy paramétrable par des constantes ou par des entrées gérées par des curseurs.

Dans la réalité, les correcteurs PID sont gérés, le plus souvent, numériquement. Nous verrons comment le construire par quelques lignes dans un script Python afin d'approcher les solutions réelles où le microcontrôleur fait les acquisitions des grandeurs analogiques via des CAN et crée le signal de commande PWM. On parlera alors d'asservissement numérique. De nouvelles notions seront abordées, comme la discrétisation en amplitude, la période et la fréquence d'échantillonnage qui interviennent dans la précision et la stabilité.
Le script python, mis au point à l'aide de Sinusphy, peut alors être implanté dans le microcontrôleur en s'assurant que la période d'échantillonnage soit suffisamment petite et, surtout, constante.