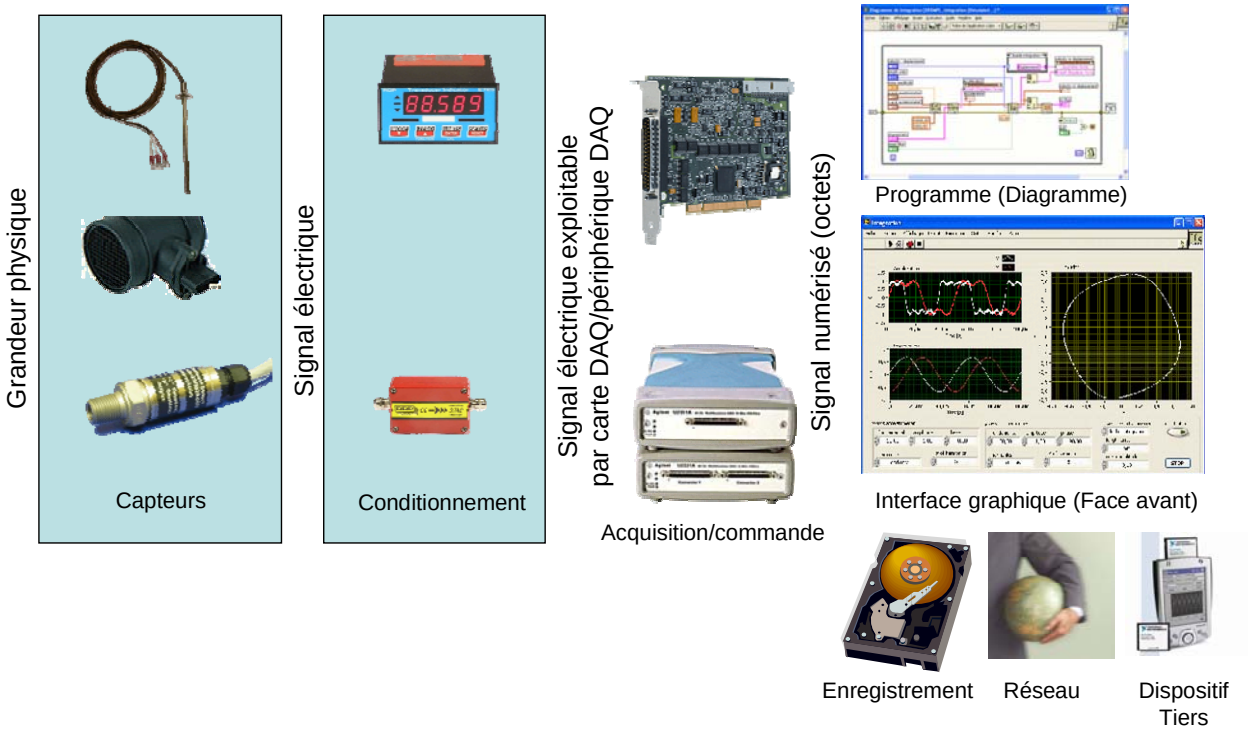
Introduction à LabVIEW

Premiers pas vers l'expérience



Novembre 2007
Emmanuel Grolleau

Table des matières

Table des exercices

1 Origines de LabVIEW

LabVIEW pour ***Lab**oratory **V**irtual **I**nstrumentation **E**ngineering **W**orkbench* est un environnement de développement en langage **G**. Notons que l'on écrit LabVIEW et non pas LabView ou Labview, etc.

Il a été créé en 1986, initialement pour Apple Macintosh, qui était à l'époque l'un des seuls ordinateurs proposant une interface graphique native. L'histoire de LabVIEW explique un vocabulaire spécifique, et explique encore certaines actions. A l'origine, LabVIEW s'exécute sur des écrans noir et blancs, puis sur des écrans 16 couleurs, 256, etc.

LabVIEW est un environnement de développement propriétaire (par opposition à un environnement ouvert, permettant à plusieurs personnes de développer des compilateurs compatibles, comme pour C/C++, Ada, Java, Fortran, etc.) développé et vendu par la société **National Instruments** (**NI**). Le premier métier de NI est de fabriquer du matériel d'acquisition (notamment sur le protocole GPIB au début des années 80) rapidement destiné au marché des micro-ordinateurs (IBM PC, Apple Macintosh). Ainsi, la première version de LabVIEW s'attache à offrir un environnement de développement dont le rôle est de permettre simplement à l'utilisateur de créer des **instruments virtuels** (***virtual instrument***, ou ***vi***) utilisant le matériel d'acquisition NI pour reproduire sur un micro-ordinateur le comportement d'un instrument personnalisé et personnalisable à volonté.

2 Le concept d'instrument virtuel

L'idée de base est d'utiliser une carte d'acquisition ou un périphérique d'acquisition, dont le rôle est d'acquérir un signal électrique provenant de l'extérieur, généralement un **capteur** (thermocouple, débitmètre, voltmètre, etc.) ou un ensemble de capteurs, effectuer un traitement, un enregistrement dans un fichier ou une base de données, une restitution à l'écran via une interface graphique, et éventuellement effectuer un ensemble d'actions sur le monde extérieur à l'aide d'**actionneurs** (électrovanne, moteur, etc.). Les variations possibles sont infinies en fonction des besoins et de leurs évolutions.

L'avantage de l'instrument virtuel sur l'instrument réel est indéniable, puisqu'il est du ressort du programmeur de l'instrument virtuel (par conséquent… vous…) de faire évoluer l'instrument virtuel en fonction des besoins (interrogation via le réseau, tolérance aux pannes, gestion de différentes vues, calculs, etc.).

La Figure 1 présente les étapes classiques du phénomène physique à sa visualisation sur un instrument réel.

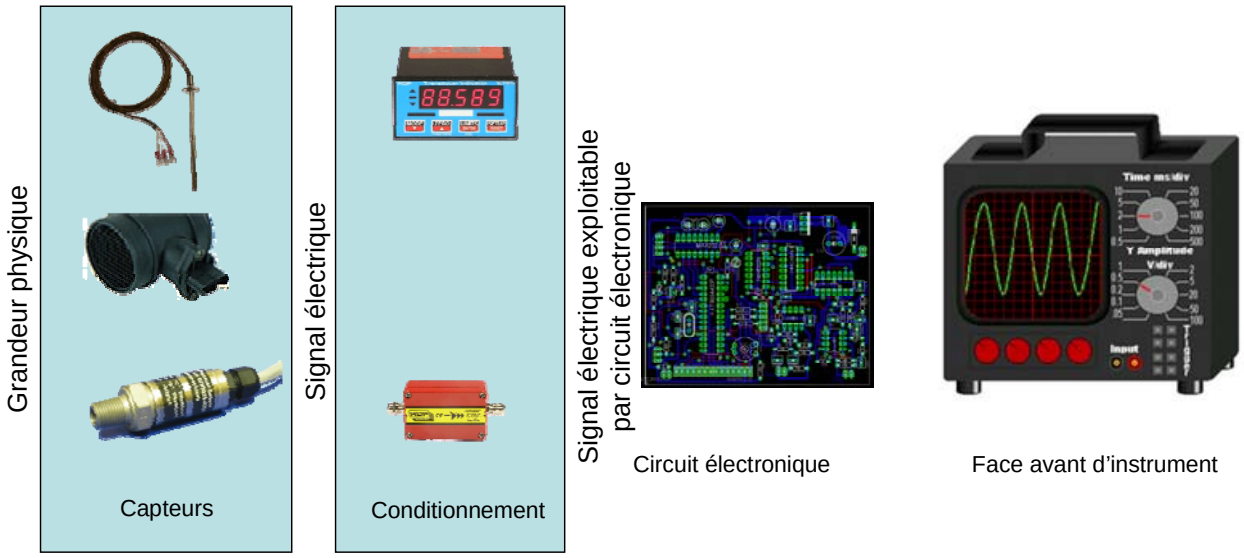


Figure 1 : de la grandeur physique à l'instrument réel

La Figure 2 montre que l'instrument virtuel diffère dans les dernières étapes : en effet, un dispositif d'acquisition (carte d'acquisition, périphérique) permet au micro-ordinateur d'acquérir le signal. En LabVIEW, nous verrons que 2 parties sont utilisées lorsque l'on programme : le programme (**diagramme** dans la terminologie LabVIEW), qui représente le traitement qui sur un instrument réel est pris en charge par un circuit électronique, et l'interface graphique (**face avant** pour LabVIEW) qui, à l'instar de la face avant d'un instrument réel, permet d'afficher à l'utilisateur et d'interagir avec lui. Le programme permet d'effectuer tout traitement automatisable (de l'enregistrement, à l'utilisation d'un réseau, en passant par la commande).

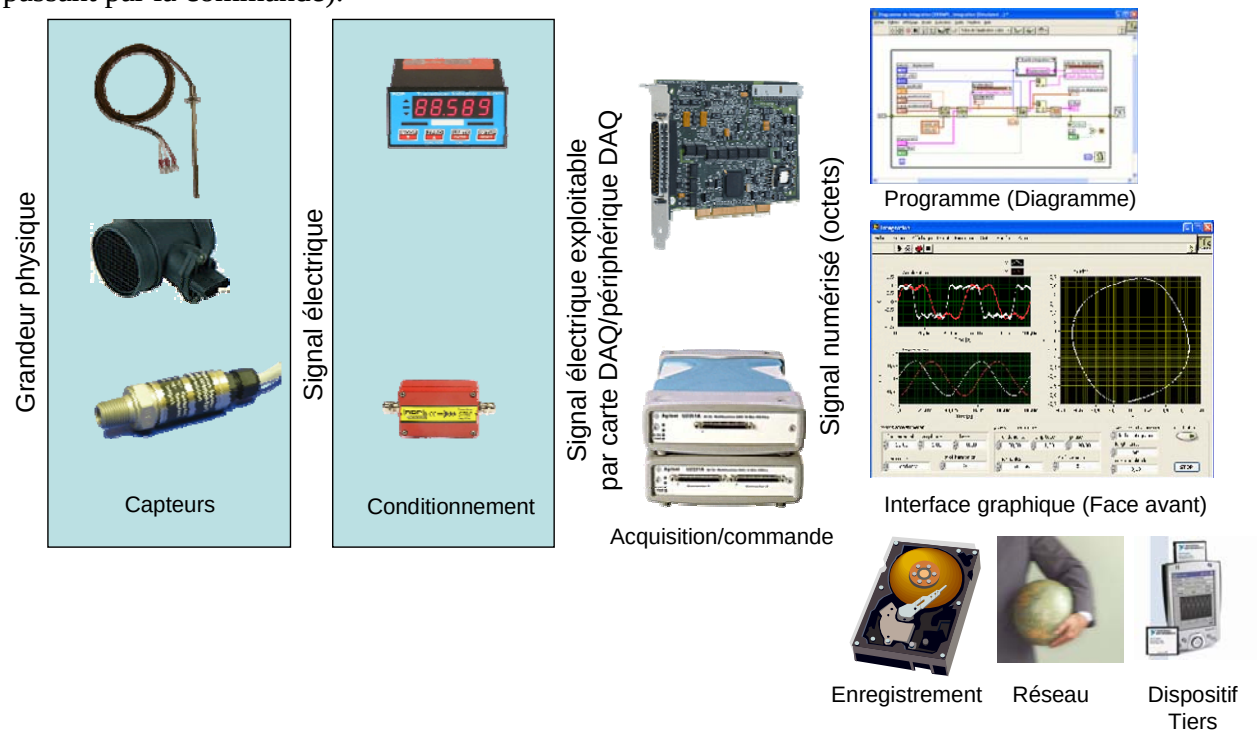


Figure 2 : de la grandeur physique à l'instrument virtuel

3 Premier pas

Lorsqu'on lance LabVIEW  , l'écran de démarrage permet de démarrer toutes les opérations proposées par cet environnement de développement (voir Figure 3).

Figure 3 : écran d'accueil de LabVIEW

Pour créer un nouveau programme on sélectionnera « VI vide ». LabVIEW utilise le nom de ***vi*** (***virtual instrument***), à prononcer à l'anglaise, « vi-aïe » pour les programmes et sous-programmes. Les programmes ou sous-programmes auront l'extension « **.vi** » et seront donc nommés *vi*. Nous verrons qu'il y a peu de différences entre un programme et un sous-programme, on pourra donc parfois utiliser spécifiquement le terme de « sous-*vi* » pour sous-programme, mais cela ne sera pas toujours le cas.

Sur l'écran de démarrage, la partie gauche permet de créer ou d'ouvrir un vi ou un projet (nous verrons la notion de projet, récente dans LabVIEW, plus loin dans la formation). La partie droite, sur laquelle nous reviendrons bientôt, permet d'accéder à la documentation extrêmement riche, incluant notamment divers tutoriaux. De plus, elle donne un pointeur vers les nouveautés de la version courante par rapport à la version précédente, ainsi que vers des sites internets (ainsi Zone LabVIEW est une zone d'échange d'expérience très active). Enfin, cet écran donne un lien vers un outil de recherche d'exemples très variés. Nous aurons l'occasion d'utiliser cet outil indispensable dès que l'on souhaite s'atteler à un type de problème nouveau.

4 Créer un nouveau vi

Que l'on souhaite créer un programme ou un sous-programme, on crée un *vi*. Pour LabVIEW tout *vi* est considéré comme un instrument virtuel. Par conséquent il a un comportement (voir Figure 4) donné sur le **diagramme** (fenêtre blanche) et une interface utilisateur nommée **face-avant** (fenêtre grise). De plus, un *vi* sera symbolisé par son icône. Il est important de retenir qu'un *vi* est stocké dans un unique fichier *.vi* : **2 fenêtres, mais 1 seul fichier**.

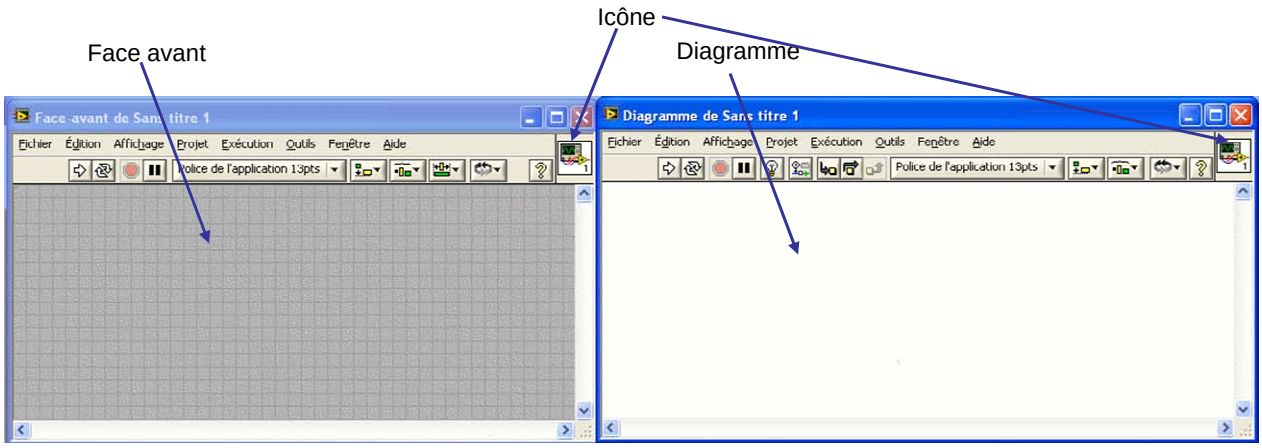


Figure 4 : un *vi* LabVIEW est composé d'un diagramme (fenêtre blanche) et d'une face avant (fenêtre grise) et représenté par une icône

Sur la face-avant, nous serons donc amenés à placer des éléments graphiques (entrées du programme, à l'instar des boutons d'un instrument, sorties à l'instar des éléments affichés sur un instrument, éléments de décoration,…), alors que sur le diagramme nous placerons la logique du programme, en général logique qui relie les entrées aux sorties : comment calculer les sorties à partir des entrées.

4.1 Les palettes

En LabVIEW, toute la programmation se passe de façon graphique, il n'y a pas de syntaxe à connaître (pas de *begin*, *end*, *for*, etc.). Quand on débute, on commence souvent par créer la face-avant, puis on passe au diagramme pour représenter la logique du programme. Nous verrons par la suite qu'il est souvent plus rapide de faire l'inverse (sauf pour le *vi* correspondant à ce qui sera montré finalement à l'utilisateur).

4.1.1 Palette de commandes

Commençons donc comme tout débutant par créer l'interface graphique : supposons que le programme prenne 2 numériques en entrée, et calcule un résultat sous forme d'un numérique. Il nous faudra donc créer 2 entrées numériques, et 1 sortie numérique (affichage). Dans le jargon LabVIEW, les entrées s'appellent des **commandes** et les sorties des **indicateurs** (toujours par analogie avec un instrument). Les commandes, indicateurs et décoration sont disponibles à partir de la **palette de commandes** (voir Figure 5) de LabVIEW. Il y a plusieurs façons d'afficher la palette de commandes :

- Faire un click droit sur la face-avant (attention, un click droit sur le diagramme affiche la palette de fonctions dont nous discuterons après). Remarquer la punaise en haut à gauche de la palette qui apparaît sous la forme d'un menu : en cliquant sur celle-ci, la palette reste affichée sous forme d'une fenêtre.
- Dans le menu déroulant de la face-avant, cliquer sur « Affichage », puis sélectionner « Palettes des commandes ». Cela a pour effet d'afficher la palette sous forme d'une fenêtre (équivalent à utiliser la punaise).

Remarquer que même si elle est affichée, la palette de commandes devient invisible lorsque la fenêtre de la face-avant n'est pas active.

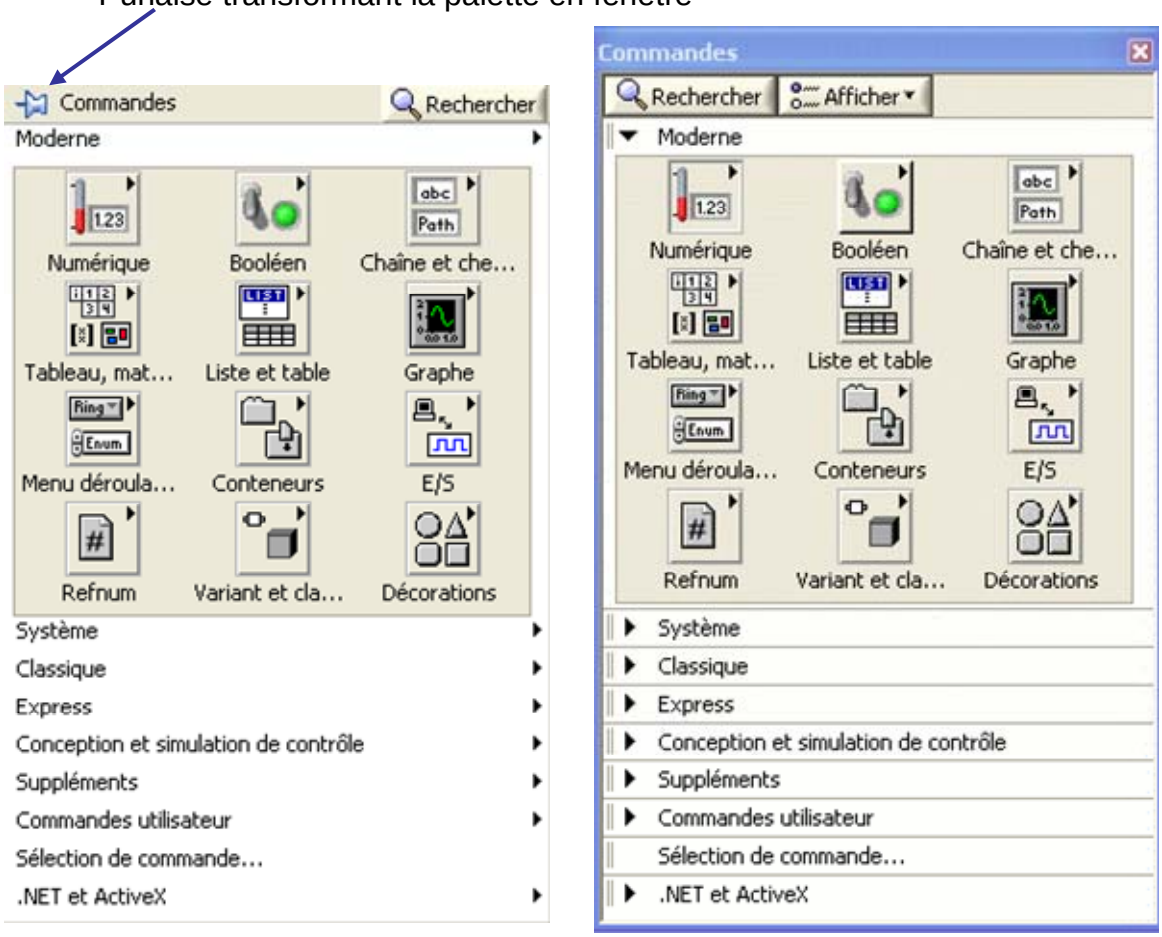


Figure 5 : palette de commandes : à gauche sous forme de menu (click droit sur la face-avant), à droite sous forme de fenêtre (via menu déroulant de la face avant, ou bien à l'aide de la punaise)

Plusieurs palettes existent (« Moderne », « Système », « Classique », « Express », « Commandes utilisateur », « .NET et ActiveX », les autres palettes dépendant des modules supplémentaires installés). La plupart du temps nous utiliserons la palette « Moderne ».

Cette palette organise les éléments par catégorie :

- « Numérique » : offre des commandes et indicateurs permettant de saisir ou afficher un **numérique**. Noter que par défaut, ce numérique est en général un nombre réel (par opposition à un nombre entier). Nous parlerons plus tard de la notion de type.
- « Booléen » : commandes et indicateurs booléens (l'algèbre de Boole, dite algèbre booléenne est l'algèbre de la logique, dans laquelle les variables peuvent prendre la valeur vrai ou la valeur faux). Les **booléens** correspondent aux boutons à 2 états et aux indicateurs à 2 états (LED, etc.).
- « Chaîne et chemin » : commandes et indicateurs permettant de saisir ou d'afficher des **chaînes de caractères** (par chaîne de caractères, on entend des mots ou phrases, i.e. suite de caractères). De même, on trouve ici ce qui permet de saisir ou afficher un nom de fichier ou de répertoire (chemin).
- « Graphe » : propose différents indicateurs de graphes.

Ces quatre sous-palettes sont les plus fréquemment utilisées. Vous êtes invité bien entendu à ouvrir les menus afin de découvrir les éléments que proposent les différentes palettes et sous palettes (par exemple la palette système offre des éléments au « look » du système d'exploitation sous-jacent comme Windows, Mac OS,…).

Il est important de comprendre que l'apparence d'une commande, par exemple pour un numérique, n'a pas de réelle importance : une « commande numérique » et un « bouton

rotatif » ont la même fonction pour le programme : permettre d'entrer un numérique. Le choix d'une apparence n'a donc strictement aucune importance pour un *vi* qui sera utilisé comme un sous-programme (sa face-avant ne sera pas vue par l'utilisateur). Ce choix aura bien entendu son importance sur le ou les *vi* qui seront vus par l'utilisateur du programme final.

La Figure 6 montre le *vi* après création de deux commandes numériques (« Numérique » et « Bouton rotatif ») et d'un indicateur numérique (« vumètre ») sur la face-avant. On peut voir qu'à chaque commande ou indicateur de la face-avant correspond une représentation logique sur le diagramme. La différence entre une commande et un indicateur est fondamentale : la commande fournit une donnée au diagramme. Noter sur l'icône de la commande dans le diagramme la flèche à droite allant vers l'extérieur et représentant le cheminement de la donnée (de la commande vers le programme). L'indicateur, quant à lui, permet d'afficher une donnée, qui lui est fournie par le programme : noter sur son icône la flèche à gauche de l'extérieur vers l'icône (du programme vers l'indicateur). Une autre différence graphique utile dans les versions plus anciennes de LabVIEW (version 6 et antérieure) est que le bord d'un indicateur est en trait gras dans le diagramme, alors que celui d'une commande est en trait fin.

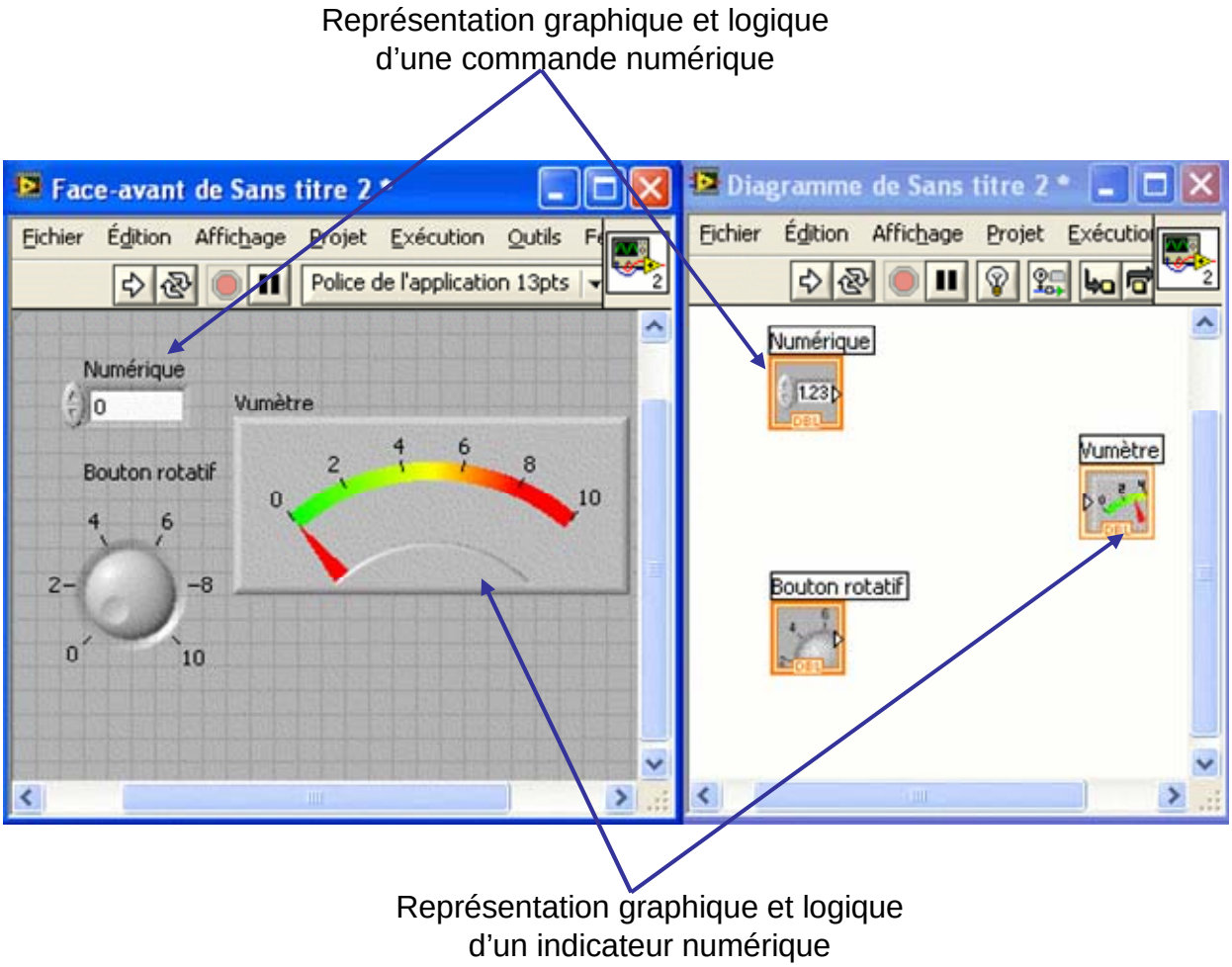


Figure 6 : 2 commandes et un indicateur numériques

4.1.2 Palette d'outils

Avant de rentrer dans la façon d'élaborer le programme (diagramme), il est bon de comprendre la façon dont on donne certaines fonctionnalités à la souris. En effet, tout ou presque étant fait à la souris, il faut comprendre que l'utilisation seule de la souris pourra avoir pour effet de redimensionner, déplacer, actionner (tourner un bouton, cliquer sur un

menu, etc.), colorier, sélectionner du texte, etc. Les principales fonctions de la souris sont les suivantes :

- la « main » (plus formellement la fonctionnalité souris « Actionner les commandes ») permet d'actionner les commandes comme pendant l'exécution du *vi*. Il est ainsi possible de tourner un bouton, bouger une aiguille, cliquer sur une barre de défilement, etc.
- la « flèche » (plus formellement la fonctionnalité souris « Positionner/Dimensionner/Sélectionner ») permet de déplacer un élément, le redimensionner (lorsque l'on passe la souris sur une poignée qui est représentée par un point bleu). Au repos (quand aucune action n'est possible à l'emplacement courant de la souris, typiquement parce-qu'aucun objet n'est à l'emplacement courant de la souris, le curseur de la souris est +.
- l'édition de texte permet de modifier le texte (typiquement le nom des éléments affichés, les valeurs numériques des échelles, etc.) ou bien d'écrire librement du texte. Noter qu'un texte écrit librement correspond, sur la face avant, à de la décoration (ce n'est pas un élément actif), et sur le diagramme à du commentaire (attention à ne pas le confondre avec une chaîne de caractères). Là encore, l'allure du curseur permet de savoir ce qui sera fait : si l'on clique avec le curseur , on créera un nouveau texte libre. Si le curseur est au dessus d'un texte, il prendra la forme , montrant ainsi qu'un click permettra d'éditer le texte situé sous le curseur.
- la bobine (plus formellement « Connecter les terminaux ») permet de créer un câble entre deux terminaux (éléments « connectables »). Notons pour le moment que cette fonctionnalité n'est pas utilisée sur la face avant, mais que c'est une fonctionnalité primordiale au niveau du diagramme.

On peut faire en sorte que le changement de fonctionnalité de la souris s'opère automatiquement (c'est le cas si lorsque le pointeur n'est pas au dessus d'un objet, son apparence est +*). Pour cela, ou pour changer de fonctionnalité, on utilise la **palette d'outils** (voir Figure 7). La palette d'outils apparaît à l'aide du menu déroulant « Affichage » → « Palette d'outils ». L'utilisation de la sélection automatique est conseillée : dans ce cas, en fonction du placement de la souris, LabVIEW choisit automatiquement la fonctionnalité (l'« outil ») adéquat. Déplacez votre souris lentement au-dessus des objets, et voyez comme son aspect change en fonction de l'outil choisi par LabVIEW. La seule subtilité concerne la saisie de texte : dans ce cas, double-cliquer sur un texte existant permet de l'éditer et double-cliquer n'importe où dans la fenêtre permet de créer un texte libre.

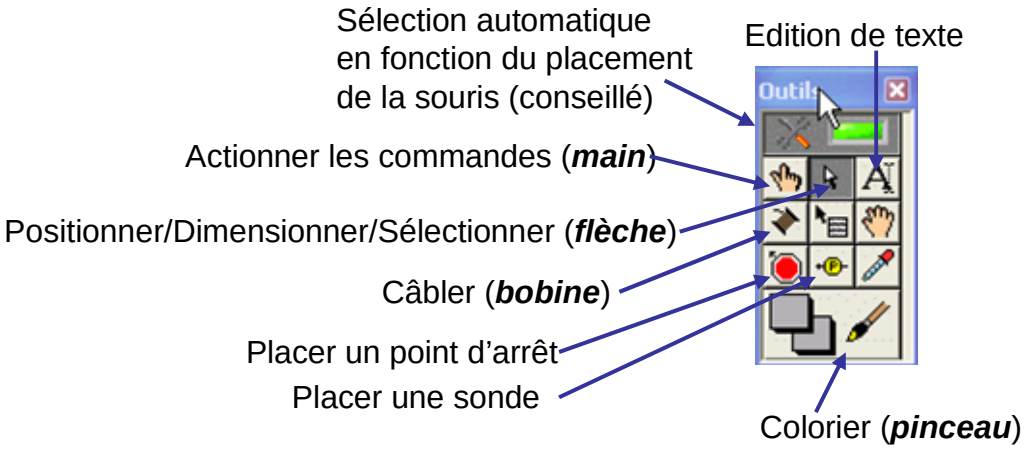


Figure 7 : la palette d'outils permet de choisir la fonctionnalité de la souris

La fonctionnalité pinceau permet de colorier les différents éléments de la face-avant et du diagramme. Son utilisation manque, contrairement au reste, d'intuitivité. Pour l'utiliser, sélectionner l'outil pinceau dans la palette d'outils, puis faire un click droit sur l'objet ou la partie de l'objet à colorier. Chaque partie d'objet peut avoir une seule couleur, ou 2 (une couleur d'avant plan et une couleur d'arrière plan). Si l'objet possède 2 couleurs, l'appui sur la barre espace permet de contrôler la partie qu'on colorie (avant-plan, arrière-plan, les 2 simultanément).

4.1.3 Palette de fonctions

Lorsque la face avant est prête, on peut passer à la définition du programme en lui-même. Notre attention va donc maintenant se focaliser sur le diagramme. Supposons que nous souhaitions réaliser un convertisseur d'unité de température de °C vers °F en utilisant la formule F=1,8xC+32. Nous commençons par créer la face avant (une commande numérique, un indicateur numérique, sous forme d'un thermomètre). Quelques modifications (police, ajout de texte libre, décorations, couleurs) permettent d'obtenir une face avant telle que celle présentée sur la Figure 8.

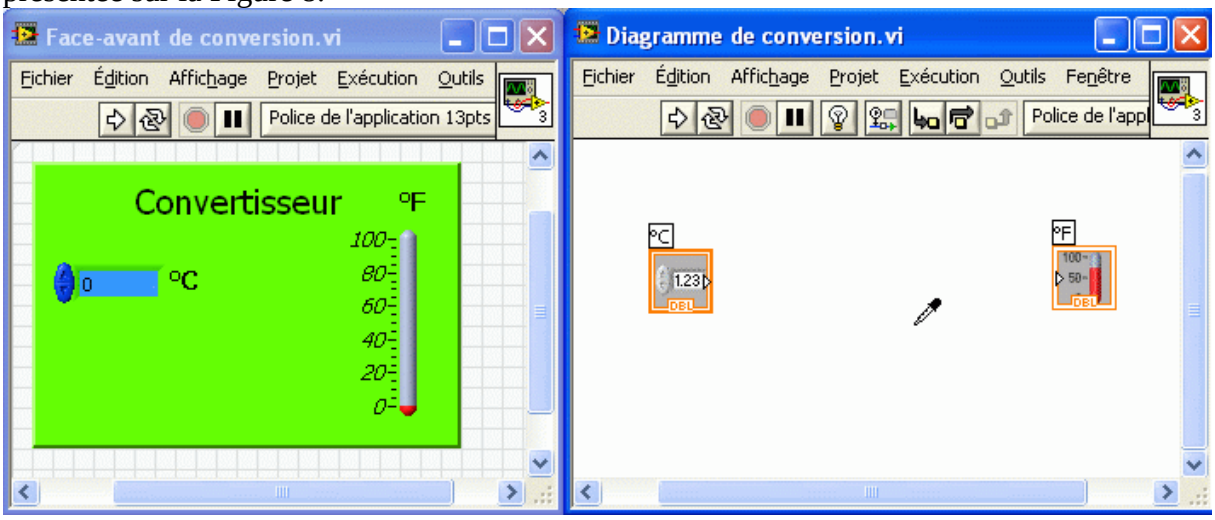


Figure 8 : face avant et diagramme « vide » d'un convertisseur d'unités

A l'instar de la palette de commandes utilisée pour la face-avant, le diagramme offre lui aussi une palette : la **palette de fonctions** (voir Figure 9). Cette palette peut-être obtenue de manière analogue à la palette de commandes (menu déroulant « Affichage » de la fenêtre du diagramme, ou bien click droit dans le diagramme).

Chaque fonction (sous-*vi*) en LabVIEW est représentée par une icône. Les fonctions sont regroupées par catégories : « Structures », « Tableau », « Cluster et variant », « Numérique », « Entrées/Sorties sur fichier », « Booléen », « Chaîne de caractère », « Comparaison », etc. Nous aurons l'occasion de les explorer ensemble.

L'une des fonctionnalités importantes de cette palette est le bouton « **Rechercher** ». En effet, la bibliothèque de fonctions est importante, d'autant plus si différents modules optionnels sont installés. La recherche permet une recherche rapide de *vi* à l'aide d'un mot clé. Ainsi, avant de se lancer dans le convertisseur, aurait-il été utile de rechercher « conver » pour s'apercevoir que LabVIEW peut gérer lui-même les conversions d'unités.

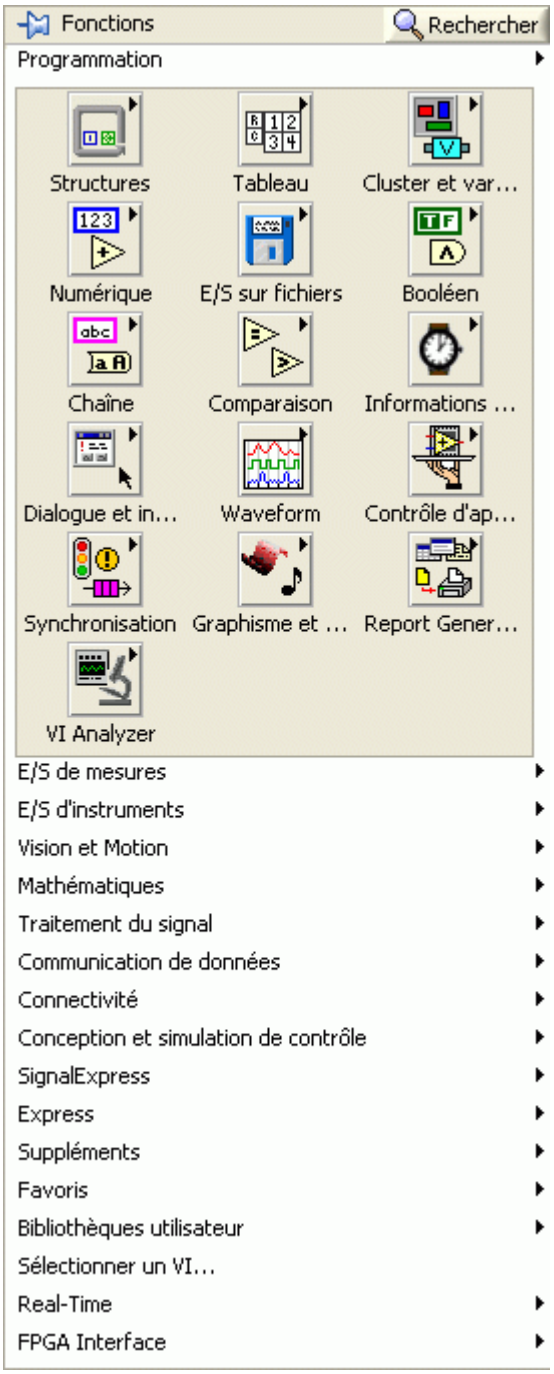


Figure 9 : la palette de fonctions

Exercice d'application 1 : comprendre la notion de diagramme et face-avant, savoir trouver les palettes.
A faire : Lancer LabVIEW, créer un nouveau *vi*. Afficher la palette d'outils, et sélectionner « Sélection automatique de l'outil ». Afficher sous forme de fenêtre la palette de commandes et la palette de fonctions. Fermer la fenêtre diagramme du *vi*. La faire réapparaître en utilisant le menu « Fenêtre » de la fenêtre face-avant ou bien le raccourci clavier **Ctrl+E**.

4.2 Création du diagramme

Il nous faut commencer, pour le convertisseur, à placer les fonctions nécessaires : une addition, et une multiplication (F=1,8xC+32). Toutes les deux sont dans la palette « Numérique » (étape 1 de la Figure 10). Ensuite, on utilise l'outil bobine pour câbler l'entrée (la commande) « °C » à une entrée de la fonction multiplication (étape 2). Il nous faut alors multiplier cela à la constante 1,8. Il y a différents moyens de créer une constante, mais le

plus simple, à utiliser dans 99% des cas (sauf dans le cas de *vi* polymorphe), est de faire un click droit sur l'entrée, puis de choisir « Créer » → « Constante ». Une constante, du bon type, est alors créée et reliée à cette entrée. Le diagramme est enfin complété comme à l'étape 3. Pour exécuter le vi, il faut cliquer sur le bouton « Exécuter »  . Certains utiliseront « Exécuter en continu »  , qui a pour effet d'exécuter le *vi* sans arrêt jusqu'à l'appui sur le bouton « Abandonner l'exécution »  . Cependant, je déconseille de prendre cette habitude : un programme bien fait contient une boucle d'exécution au niveau le plus haut, et son propre bouton d'arrêt sur la face-avant…

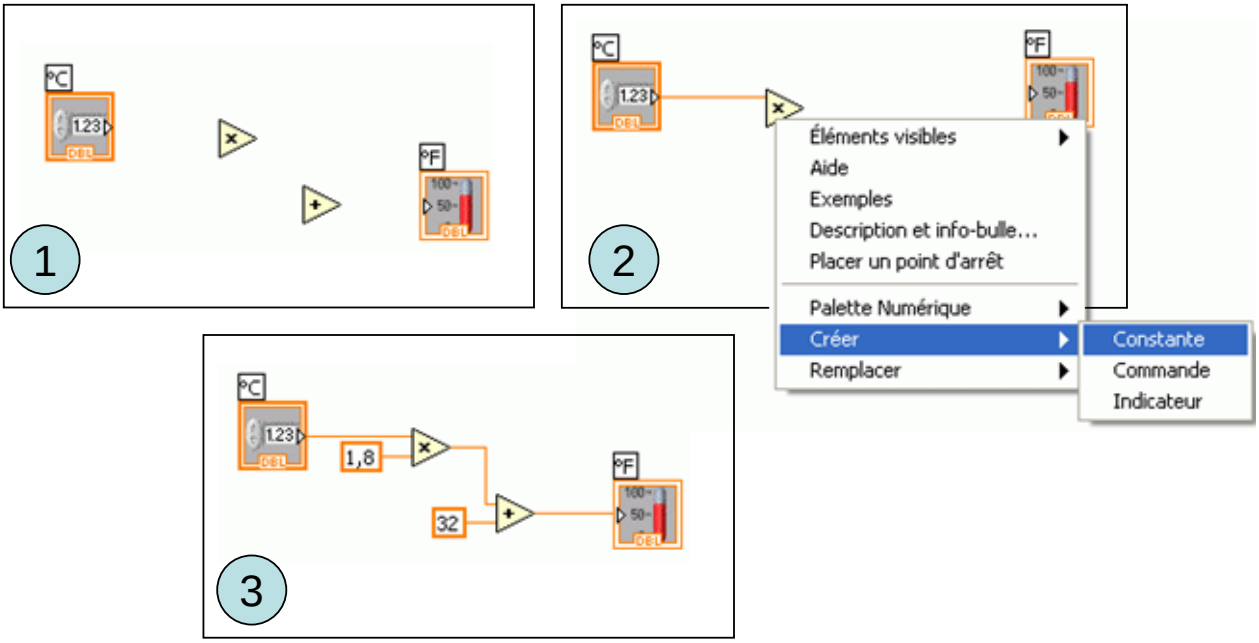


Figure 10 : étapes de la construction du diagramme

A partir d'ici, il convient d'utiliser la fenêtre d'**aide contextuelle** (menu déroulant « Aide » → « Afficher l'aide contextuelle » ou raccourci clavier **Ctrl+H**) afin d'obtenir des descriptions rapides des fonctions, ou bien savoir quel type de données passe sur un fil. Le contenu de l'aide est mis à jour en fonction de l'élément se trouvant sous le curseur de la souris. Ainsi, si l'on place la souris sur le nœud « x » de la Figure 10, on obtient l'aide contextuelle donnée sur la Figure 11.

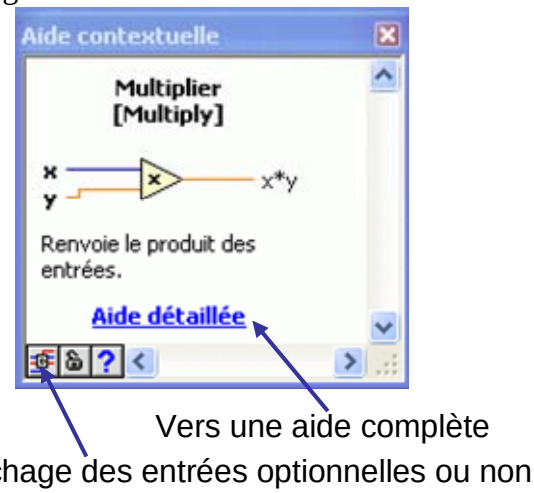


Figure 11 : aide contextuelle

Exercice d'application 2 : comprendre le principe du câblage, utiliser l'aide contextuelle, créer et personnaliser une face-avant simple. Utiliser l'aide en ligne.

A faire : Créer le *vi* de conversion donné sur la Figure 10. Pour cela, s'assurer que l'aide contextuelle est activée. Personnaliser sa face avant comme sur la Figure 8. Rechercher dans l'aide détaillée ce qui se rapporte à la gestion d'unités. Modifier le *vi* afin d'utiliser les unités de mesure et la conversion automatique par LabVIEW.

Note : on peut faire une différence entre *vi* standard et ***vi express***. Les *vi express* sont représentés par une icône dans un carré bleu ciel comme 

. Les *vi* express permettent de programmer très simplement une tâche commune. Cependant, je ne saurais les conseiller : généralement plus lents à l'exécution que des *vi* programmés équivalent, lents à charger, et surtout, en cas de génération de fichier exécutable à partir d'un *vi*, les *vi express* augmentent considérablement la taille de l'exécutable généré. Enfin, ils ont des fonctionnalités limitées aux tâches communes : si le besoin diffère un peu des besoins prévus, alors il faudra de toute façon faire sans. Dans cette formation, il ne sera donc fait aucun cas de *vi express*.

4.2.1 Programmation flots de données

Intuitivement, le comportement est le suivant : lorsque le *vi* est exécuté, les commandes (entrées de données sur la face avant) et les constantes fournissent leur valeur. Les commandes sont d'ailleurs appelées **terminaux d'entrée** du diagramme. Tout fil sortant d'un de ces éléments transporte donc sa valeur vers l'entrée du prochain **nœud**. Lorsqu'un nœud possède une donnée sur chacune de ses entrées, il devient exécutable. A chaque étape de l'exécution, le *runtime* (moteur d'exécution LabVIEW) choisit parmi tous les nœuds exécutables un nœud à exécuter. En s'exécutant, un nœud consomme ses entrées, et génère des sorties sur chacun des fils en sortie. En général, les flots se terminent dans un indicateur, appelé **terminal de sortie** (ici « °F »). Cela a pour effet d'afficher la valeur contenue sur le fil. Afin de bien voir le déroulement de l'exécution d'un *vi*, on pourra actionner  l'animation de l'exécution : cela permet de visualiser les transits des données sur les fils.

Ce type de programmation s'appelle de la programmation **flots de données**. Chaque fil supporte un flot de données, comme si les données s'écoulaient sur les fils de même qu'une rivière s'écoulerait dans un lit. La programmation flot de données est une philosophie très différente des types de programmations que vous connaissez peut-être, comme la programmation impérative (Fortran, C/C++, Ada, Java, Javascript, PHP, etc. en fait la plus grande majorité des langages de programmation), ou la programmation fonctionnelle (comme caml ou XSLT) ou encore de la programmation déclarative (Prolog, SQL). Ce type de programmation correspond au langage G. Très souvent, on confond langage G et LabVIEW puisque cet environnement est le seul représentant très connu du langage G.

La différence majeure par rapport à un langage plus classique (impératif) est que le langage G ne dispose de séquencement implicite. Ainsi, dans un langage impératif, l'ordre des instructions implique un séquencement de celles-ci. En G, il n'en est rien : seule la dépendance liée aux flots de données (une entrée dépendant du flot de sortie) entraîne une notion de séquencement (l'action A a lieu avant l'action B). Cependant, lorsque plusieurs portions de programmes sont indépendantes (pas de dépendance de flots de données), elles peuvent être exécutées en parallèle. Ce parallélisme peut être réel si la machine dispose de plusieurs processeurs ou d'un processeur à plusieurs cœurs, ou bien il se traduit par de l'entrelacement temporel. Ainsi, si l'action A et l'action B ont lieu en parallèle, alors il est possible, par exemple, que le *runtime* commence A, puis commence B, puis continue A, puis

continue B, puis termine A, puis termine B. En fait, le programme est exécuté par différentes tâches du système d'exploitation (on parle de *threads*) ce qui fait de G un langage parallèle naturel.

L'une des réserves que certains opposent à LabVIEW est la lenteur que ce type d'exécution graphique et parallèle peut entraîner. En réalité, à l'exécution, LabVIEW compile les *vi* en code exécutable. Lors de l'exécution, à part quelques optimisations qui pourraient améliorer la vitesse, sa vitesse d'exécution est tout à fait comparable à la vitesse d'exécution d'un programme écrit en C++. Il est d'ailleurs possible, afin d'éviter une compilation des *vi* et ainsi accélérer le temps de chargement de compiler tous les *vi* d'un répertoire (menu « Outils » → « Avancé » → « Recompiler… ».

4.2.2 Correction syntaxique

Vous aurez peut-être remarqué lors de la création de votre *vi* que la flèche d'exécution est parfois **brisée**  . Comme tout langage de programmation, LabVIEW vérifie la syntaxe des *vi*. La différence par rapport à nombre d'autres langages est que la vérification a lieu pendant l'édition des *vi*. Les erreurs les plus classiques, à ce stade, sont :

- Plusieurs flots sont regroupés (un fil connecte plus d'une source de données) : en LabVIEW, un flot de données peut se décomposer, mais on ne peut pas regrouper 2 flots dans un seul, sauf si l'on utilise un constructeur de type (nous verrons cela par la suite). Dans le cas contraire, le flot apparaît en pointillés comme sur la Figure 12.
- On a connecté 2 entrées ensemble ou bien 2 sorties ensemble. Il ne faut pas oublier que chaque connecteur de nœud est orienté. Par convention, c'est orienté de gauche à droite (entrées à gauche, sorties à droite).
- On a connecté des connecteurs de type incompatible (nous verrons très bientôt la notion de type).
- Des entrées obligatoire n'ont pas été câblées (ainsi, si le nœud « + » ne possède qu'une entrée, une erreur de syntaxe a lieu). Chaque entrée de nœud peut être obligatoire ou non (dans le cas où une entrée n'est pas obligatoire et que rien n'est donné en entrée, une valeur par défaut est prise).

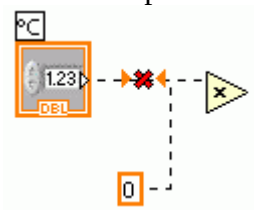


Figure 12 : erreur de syntaxe, des flots sont regroupés

Quand elle concerne des flots de données, une erreur de syntaxe se traduit par le fait que les fils concernés sont brisés. Une commande très utile se trouve dans le menu « Edition » → « Supprimer les fils brisés », ou raccourci clavier Ctrl+B. Sur la Figure 12 cela aura pour effet de supprimer tous les fils brisés.
En cliquant sur la flèche brisée, on obtient une fenêtre donnant la liste des erreurs (voir Figure 13).

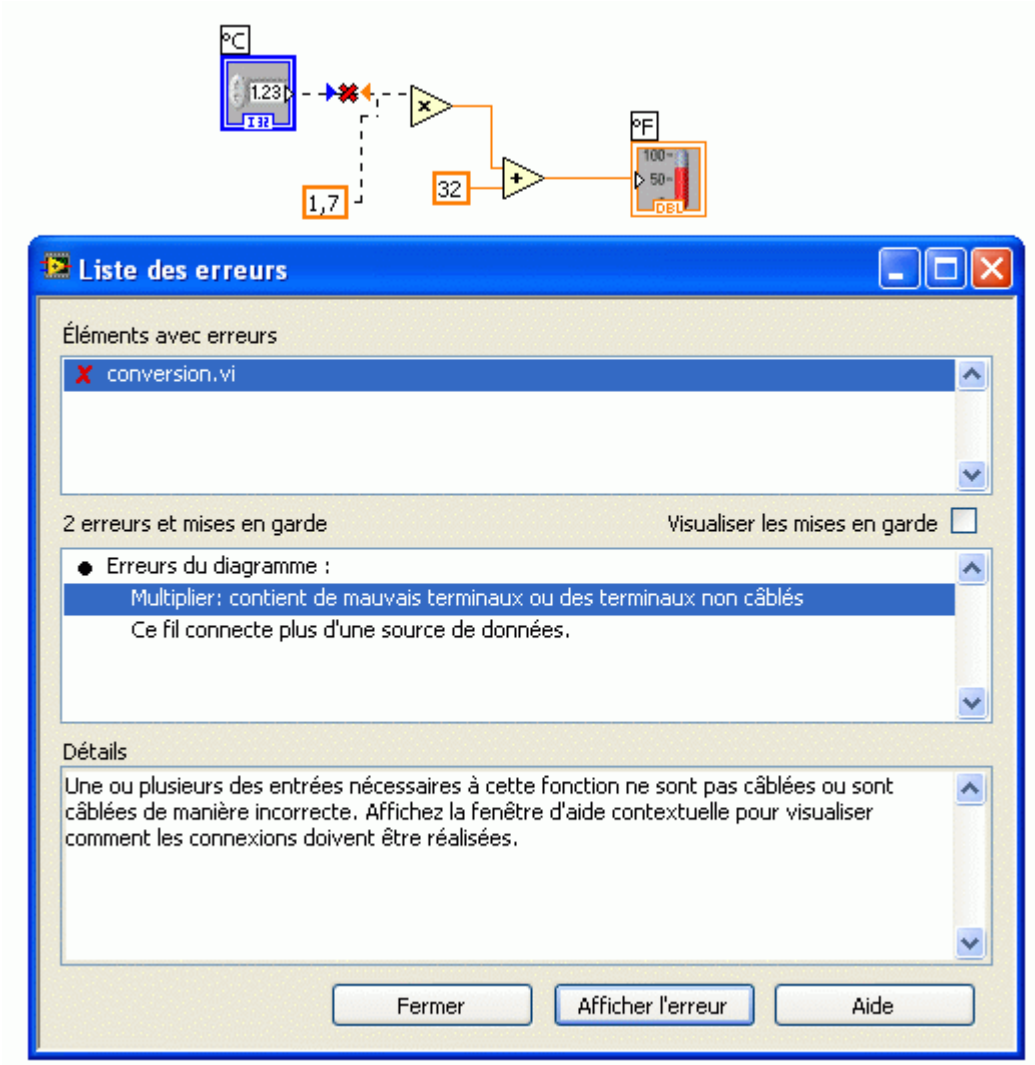


Figure 13 :affichage de la liste des erreurs

4.2.3 Typage

LabVIEW est un langage à typage fort. Toute donnée a un type.

4.2.3.1 Types de base

A chaque type de base correspond une couleur (voir Figure 14). Notons que différents types peuvent être représentés par une même couleur. Ainsi, par exemple les types numériques discrets sont bleus (entiers signés, non signés, sur 8, 16, 32 ou 64 bits, ainsi que les types énumérés).

Représentation	Couleur	Type
	bleu	Nombre entier : U8, U16, U32, U64 (U pour *unsigned*) non signé sur 8, 16, 32 ou 64 bits I8, I16, I32, I64 (I pour *integer*) signé sur 8, 16, 32 ou 64 bits
	bleu	Valeur énumérée, correspond à des entiers de 0 à n-1 (n étant le nombre de valeurs du type)
	orange	Nombre flottant (norme IEEE 764) : SGL (simple précision, codée sur 32 bits), DBL (double précision codée sur 64 bits), EXT (précision étendue dépendant de la plateforme)

	vert	Booléan : prend les valeurs T (*true* – vrai) ou F (*false* – faux)
	rose	Chaîne de caractères
	violet	Variant : représentation de n'importe quel type, sous forme des octets de données correspondant et d'un identifiant de type. Toute donnée d'un type donné peut être transformée en variant. Le variant pourra être re-transformé en son type initial, à condition de connaître le type initial.
	marron	Date.

Figure 14 : types de base

On peut remarquer que lorsqu'on crée un numérique, on ne contrôle pas son type : il s'agit généralement d'un flottant double précision quand on crée une commande ou un indicateur sur la face-avant, et d'un entier signé sur 32 bits lorsque l'on place une constante. On doit alors faire un click droit sur l'élément, et choisir le menu « Représentation » pour changer son type en un autre type numérique (voir Figure 15).

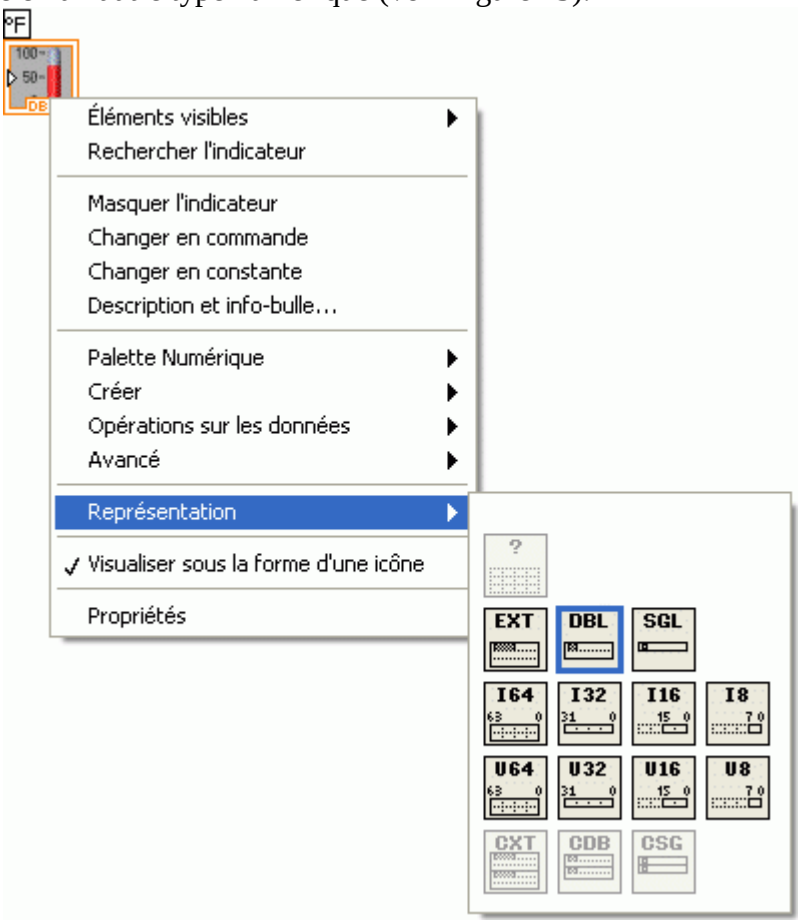


Figure 15 : changement de représentations pour un numérique

Lorsque l'on connecte deux terminaux de type différent mais compatibles (typiquement, deux numériques), le flot entrant va s'adapter automatiquement au type de donnée du connecteur. Cela s'appelle une **coercion** de type. LabVIEW montre graphiquement les coercions à l'aide d'un point rouge (dans les versions antérieures de LabVIEW il était gris) au niveau du connecteur. La Figure 16 montre une coercion : pour s'adapter à la

multiplication, la donnée venant de « °C », qui est un entier, est convertie en flottant double précision (type le plus « large » de l'opération).

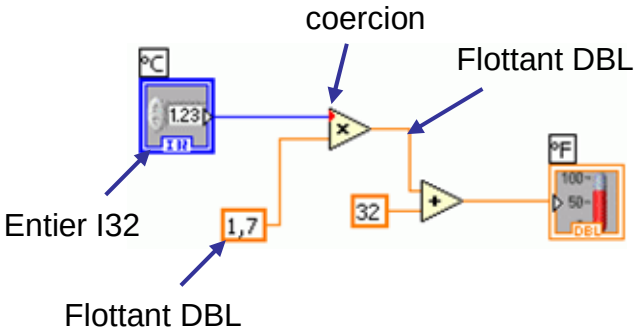


Figure 16 : coercion de type

4.2.3.2 Types composés

Les types peuvent être composés en utilisant des constructeurs : le **cluster** et le **tableau**.

4.2.3.2.1 Cluster

Le cluster correspond à un type enregistrement (nommé *record* ou *struct* dans d'autres langages de programmation). Il s'agit de grouper dans une seule donnée différentes données (appelées champs) pouvant être de type différent. L'un des clusters qui est très souvent utilisé est le **cluster d'erreur**. En effet LabVIEW utilise 3 valeurs pour caractériser une erreur :

- un booléen **status** vrai si une erreur a eu lieu, faux sinon,
- un entier *code* donnant le numéro d'erreur, permettant d'obtenir de plus amples informations sur l'erreur,
- une chaîne de caractère *source* contenant généralement le nom du *vi* dans lequel l'erreur s'est produite.

La Figure 17 montre une utilisation typique du cluster d'erreur, et par cet exemple la façon dont on extrait un champ d'un cluster. Un fichier est ouvert ou créé, et son contenu est remplacé par le texte « test ». Ensuite le fichier est fermé. Tout ceci s'il n'y a pas eu d'erreur : en effet, il y a de multiples possibilités d'erreurs lors de l'ouverture, de l'écriture, de la fermeture : le fichier n'est pas accessible, l'utilisateur annule le choix du nom de fichier, droits insuffisants pour modifier le fichier, déconnexion du réseau si le fichier est sur le réseau, etc. Ainsi, si par exemple, l'ouverture du fichier (premier *vi* de la chaîne de traitement) déclenche une erreur, l'écriture et la fermeture ne doivent rien faire puisque le fichier n'a pas été ouvert. C'est le cas puisque les *vi* de LabVIEW ayant une entrée « Entrée d'erreur » fonctionnent de la façon suivante : si il y a erreur, ne rien faire et propager l'erreur (i.e. recopier l'entrée d'erreur sur la sortie d'erreur). Sinon, faire le traitement, mais si erreur il y a l'envoyer sur la sortie d'erreur, dans le cas contraire (aucune erreur en entrée, aucune erreur lors du traitement) renvoyer « pas d'erreur » (i.e. le champ *status* est à la valeur faux) sur la sortie d'erreur.

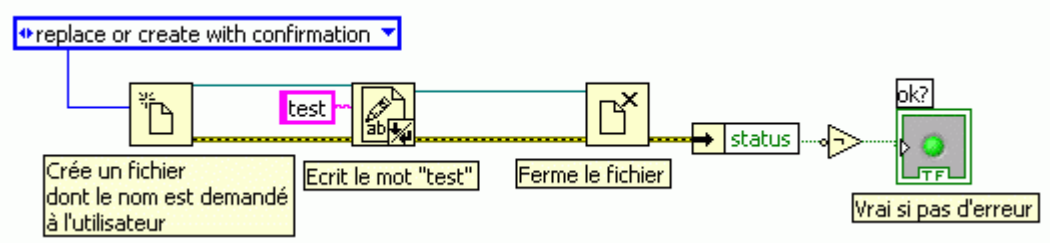


Figure 17 : utilisation typique du cluster d'erreur

Pour ceux qui connaissent d'autres langages de programmation, ce type de gestion d'erreurs peut paraître nouveau. En effet, dans les langages basés sur C, il est fréquent d'avoir à tester la valeur de retour d'erreur après chaque action, afin d'interrompre la chaîne de traitement. En Ada ou dans d'autres langages proposant les exceptions (Java, C++) le traitement est interrompu par une exception et un bloc traite exception peut alors être exécuté. Le traitement des erreurs proposé par LabVIEW est moins poussant que les exceptions, mais plus pratique qu'un traitement à la C.

On peut noter sur la Figure 17 la façon dont on extrait le champ booléen *status* du cluster d'erreur : celui-ci est faux si et seulement si aucune erreur n'a eu lieu pendant le traitement. Sa négation est donc vrai si et seulement si il n'y a pas eu d'erreur, et allume une LED placée sur la face-avant. On utilise pour extraire le champ qui nous intéresse le nœud « Désassembler par nom » trouvé dans la palette « Cluster ». Le *vi* « Désassembler par nom », ainsi que les autres *vi* permettant d'assembler ou de désassembler les *cluster* sont redimensionnables (en terme de nombre d'entrées/ou de sorties).

Noter aussi que le fait que les *vi* de la chaîne de traitement dépendent de leurs prédécesseur (dépendance en terme de flots de données) implique un séquencement des actions : ouvrir puis écrire puis fermer puis afficher la LED.

La création d'un *cluster* se fait à l'aide d'« assembler » (voir Figure 18) ou « Assembler par nom » (voir Figure 19). L'avantage d'« Assembler par nom » est la lecture très claire que l'on pourra faire lorsque l'on désassemblera. Cependant, « Assembler par nom » nécessite de donner un cluster du même type que celui que l'on veut assembler en entrée (« entrée d'erreur » sur l'exemple donné sur la Figure 19).

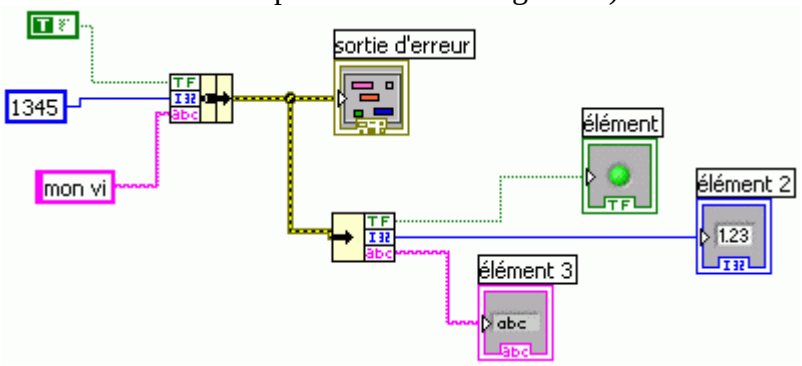


Figure 18 : création d'un cluster à l'aide d'« Assembler »

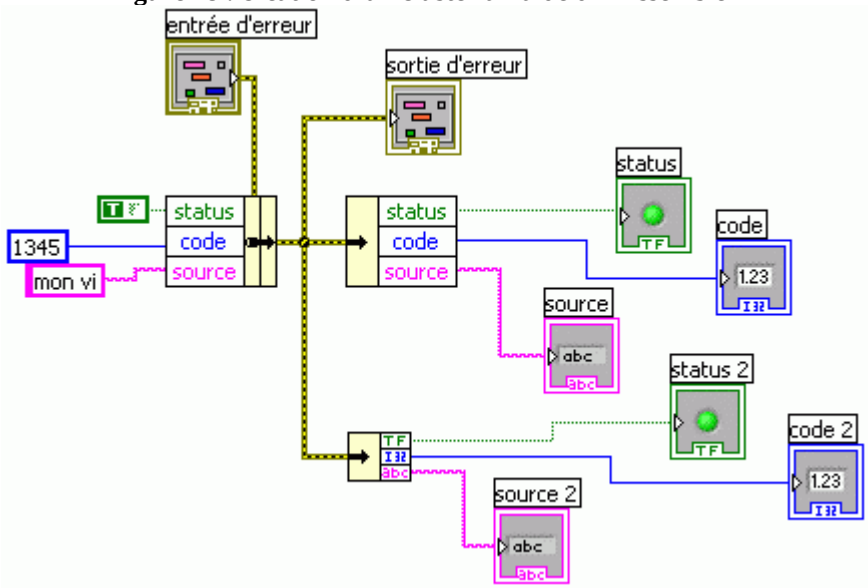


Figure 19 : création d'un cluster à l'aide d'« Assembler par nom »

Bien que les exemples donnés portent sur le *cluster* d'erreur (c'est bien souvent le premier *cluster* manipulé), on peut assembler n'importe quelles données dans un cluster. LabVIEW est très souple quant à l'utilisation de clusters. Ainsi, lorsque des clusters ne sont composés que de numériques, ils gardent la propriété numérique. Les opérations arithmétiques sont alors possibles sur le cluster. Ainsi, sur la Figure 20, on additionne directement deux *clusters*. Noter que les *clusters* numériques ont une couleur marron, alors que les clusters non numériques (comme le *cluster* d'erreur) ont une couleur tirant vers le kaki.

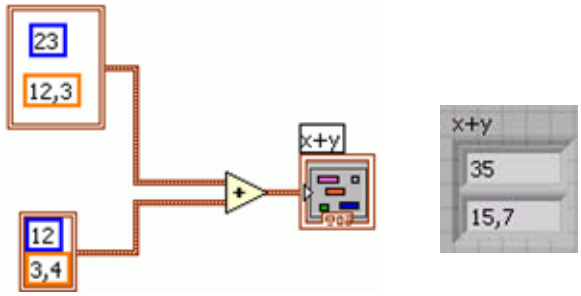
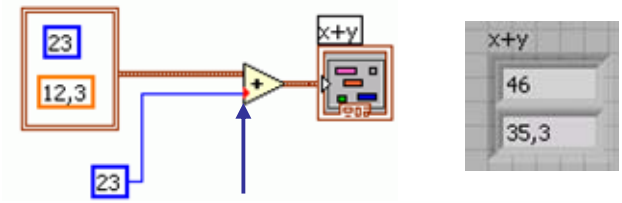


Figure 20 : opération arithmétique sur cluster (addition de 2 *clusters*)

Il est aussi possible d'effectuer une opération *cluster*/variable numérique (voir Figure 21) : dans ce cas, en réalité, une coercion du numérique est effectuée et c'est le même type d'opération que sur la Figure 20.



Coercion en cluster 23|23,0

Figure 21 : opération arithmétique sur cluster (cluster plus scalaire)

4.2.3.2.2 Tableaux

Les tableaux regroupent un nombre variable d'éléments du même type sur 1 ou plusieurs dimensions. Toutes les opérations de manipulation des tableaux se trouvent dans la palette tableau. La taille d'un tableau est donnée par  « Taille d'un tableau ». Notons que ce *vi* est **polymorphe** : il peut avoir n'importe quel tableau en entrée (quel que soit le type des éléments contenus) à 1 ou plusieurs dimensions. Si le tableau en entrée a une dimension, « Taille d'un tableau » renvoie un entier (nombre d'éléments du tableau) alors que s'il possède plusieurs dimensions, il renvoie un tableau d'entiers (1 entier par dimension, donnant sa taille).

Les tableaux sont indexés de 0 à taille_dimension-1 pour chaque dimension. Ainsi, pour obtenir le 1er élément d'un tableau à 1 dimension, on utilise  « Indexer un tableau » en demandant l'élément d'index 0. Notons que ce *vi* change d'apparence en fonction de la dimension du tableau donné en entrée. Ainsi, pour un tableau à 2 dimensions, son apparence est  , ce qui permet de donner l'indice de la 1ère et de la 2nde dimension. Si l'on ne donne en entrée que l'indice de la 1ère dimension, alors LabVIEW comprend que l'on veut accéder au tableau. Cela est généralisable à un tableau de dimension *n* : si l'on donne uniquement la 1ère valeur d'indice (ou rien, ce qui équivaut à entrer la valeur 0), on obtient le tableau à *n*-1 dimensions contenu à cet indice.

En LabVIEW, l'absence de valeur n'existe pas : si un flot est supposé sortir d'un *vi*, alors il y aura une valeur sur le flot. Pour les tableaux, cela signifie qu'accéder à un tableau d'éléments de type T au-delà de la taille du tableau (après l'indice taille_dimension-1), une valeur sera tout de même renvoyée par « Indexer un tableau ». Cette valeur sera la **valeur par défaut du type** (par exemple, 0 pour un numérique, tableau vide si c'est un tableau, etc.).

Un flot de données de type tableau est représenté en fil épais (pour un tableau unidimensionnel) et en double pour un tableau multidimensionnel. La couleur de base du trait est celle du type d'élément du tableau (exemple : orange épais signifie tableau à 1 dimension de nombres flottants).

Exercice d'application 3 : comprendre le fonctionnement des tableaux, voir la correspondance avec un fichier tableur. Prendre conscience des problèmes de localisation du séparateur décimal (utilisation du « . » ou de la « , » comme séparateur décimal dans le tableur). Comprendre la façon dont LabVIEW gère les noms de fichier et les *vi* de manipulation de noms de fichiers.

A faire : lancer une application tableur (MS Office, Ou StarOffice), et créer une feuille de calcul avec 2 colonnes de nombres réels, qui sera enregistrée sous le nom **essai tableau 1.csv**.
Le but du *vi* est d'afficher sur un indicateur combien il y a de lignes et de colonnes, et d'afficher la moyenne des nombres de la 1ère colonne.
Pour la lecture, on utilisera « Lire un fichier tableur ». Attention, peut entraîner le remplissage du tableau par ligne ou par colonne. **Utiliser l'aide détaillée** pour ce *vi*. Pour trouver un *vi* de calcul de moyenne, **utiliser la fonction rechercher** de la palette de fonctions.
Lorsque le traitement fonctionne, essayer de mettre le nom du fichier tableur utilisé en entrée sous la forme d'une constante : il s'agit d'exprimer dans le diagramme que le fichier tableur s'appelle essai tableau 1.csv et se trouve dans le même répertoire que le *vi* (qu'il conviendra donc d'enregistrer au même endroit).

Remarque : le *vi* « Lire un fichier tableur » est polymorphe. En effet, il peut lire des flottants (*Double*), des entiers, ou des chaînes de caractères dans le fichier et ainsi sélectionner le type de données fournies en sortie (le tableau bidimensionnel de valeurs). Certains *vi* polymorphes comprennent la forme qu'ils doivent choisir grâce à leur entrée (c'est le cas par exemple d'« indexer un tableau »), d'autres, au contraire, ne peuvent la déterminer à partir de leur entrée. Avant la version 8 de LabVIEW, il fallait faire un click droit sur un tel *vi* polymorphe afin de trouver le menu et de sélectionner la forme à utiliser. Depuis la version 8, le polymorphisme d'un tel *vi* se traduit par un petit menu déroulant se situant sous l'icône (voir).

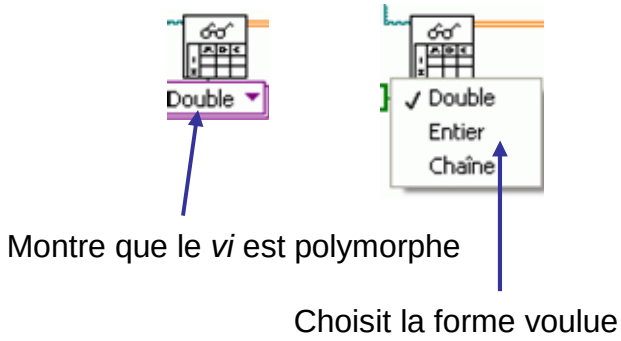


Figure 22 : un *vi* polymorphe ne pouvant définir sa forme à partir du type de ses entrées

4.2.3.2.3 Waveforms

Le type *waveform* est un type correspondant à un *cluster* possédant un champ date (absolue ou relative) t_0, un champ flottant représentant l'intervalle de temps *dt* séparant chaque valeur, et un tableau de valeurs réelles *Y*. Dans les versions récentes de LabVIEW, un champ additionnel de type variant permet à certains périphériques d'ajouter des données spécifiques. Nous verrons les *waveforms* lors de la partie acquisition de données. A notre

niveau actuel, ce qui est intéressant est de voir que LabVIEW utilise lui-même des types composés qui pourront, comme les types de base, être utilisés pour vérifier la cohérence du diagramme.

4.2.3.2.4 Types strictes

A l'instar du *waveform* de LabVIEW, le programmeur peut lui aussi créer ses types de la même façon que l'on crée des types dans des langages de programmation autres. Nous n'en parlerons pas en détail lors de cette formation.

4.2.3.2.5 Classes

La version 8 de LabVIEW a apporté une révolution attendue depuis longtemps par nombre de programmeurs LabVIEW. Il s'agit de la notion de classes. Il est possible de définir un type comme étant une classe (on peut voir cela comme une sorte de *cluster* avec des champs). On définit ensuite des méthodes comme des *vis* qui permettent de manipuler ce type. Le type n'est accessible que par les méthodes données, ce qui permet de contrôler la façon dont les variables de ce type (les objets) seront manipulées. On parle de programmation objet. L'avantage de la programmation objet est la notion d'héritage et de polymorphisme de classe. Illustrons le concept d'héritage par un exemple : supposons qu'on définisse une classe véhicule, avec un certains nombre d'attributs (en fait les champs du *cluster* utilisé pour représenter un véhicule) et de méthodes (calculer le poids, le nombre de roues, etc.). On peut créer une classe automobile et une classe avion qui héritent de la classe véhicule : il est alors inutile dans chacune de ces classes de redéfinir les attributs présents dans véhicule puisqu'ils sont hérités de la classe mère. Les méthodes peuvent être redéfinies si besoin est, ou tout simplement héritées de la classe véhicule. Des attributs et méthodes spécifiques à un avion ou une voiture seront ajoutés. Le polymorphisme de classe consiste alors à pouvoir définir des *vi* prenant un véhicule en entrée : à l'exécution, on pourra en fait passer une automobile en entrée, ou bien un avion, puisque tout ce qui existe, tout ce dont on pourrait avoir besoin, est défini sur ces classes. On peut donc passer indifféremment un véhicule, une automobile ou un avion, ou tout classe qui hérite de véhicule à la place du véhicule : d'où le polymorphisme.

Nous ne parlerons pas de programmation objet pendant cette formation, mais le lecteur intéressé est invité à lire la documentation et regarder les exemples fournis dans LabVIEW 8.

4.2.4 Encapsulation (notion de sous-programme)

Les *vi* que nous avons utilisés jusqu'à présent ont un point commun : ils possèdent un certain nombre d'entrées, et un certain nombre de sorties, on un nom, une icône illustrant leur fonction, et une documentation. Cette partie montre comment créer un sous-programme, lui donner un nom, une icône et le documenter. Pour cela, nous prendrons un exemple de générateur aléatoire.

Exercice d'application 4 : manipuler les types de données, rechercher des *vis*, réaliser un test unitaire

A faire : rechercher les *vi* existant dans LabVIEW concernant le mot « aléatoire ». Cette fonction génère un nombre pseudo-aléatoire uniforme sous forme d'un flottant compris entre 0 et 1. Nous souhaitons créer un générateur aléatoire qui prend en entrée deux nombres entiers I32 que nous appellerons min et max, et génère un nombre pseudo-aléatoire uniforme sur l'intervalle [min..max].

La formule mathématique à utiliser est :

nombre_aléatoire = arrondi(nombre_aleatoire_entre_0_et_1*(max-min))+ min

Sauvegarder le *vi* obtenu sous le nom « Aléatoire.vi » (pas indispensable, mais toujours bon…).

Faire des tests avec différentes valeurs : min et max positifs, négatifs, min positif, max négatif, min=max, max>min…

Remarque : le but est de transformer ce *vi* tel qu'il soit utilisable en sous-*vi*, il serait donc superflu de travailler outre mesure sur l'apparence de la face-avant.

4.2.4.1 Etape indispensable : créer les connecteurs

Il n'y a qu'une étape indispensable à la création d'un sous-vi (bien que 2 autres étapes soient fortement conseillées) : la création des connecteurs du *vi*. Il s'agit en effet de définir par quel biais un *vi* utilisant notre sous-*vi* lui passera les données d'entrées, et obtiendra les données de sorties. Le plus souvent, les données d'entrées sont les commandes définies sur la face-avant (dans notre exemple, min et max), et les données de sortie correspondent le plus souvent aux indicateurs de la face-avant (pour notre exemple, le résultat).

Pour cela :

- se placer sur la face-avant,
- faire un click droit sur l'icône, choisir « Visualiser le connecteur » (voir Figure 23), l'icône est alors remplacé par les connecteurs (vierges au début) du *vi* (voir Figure 24),

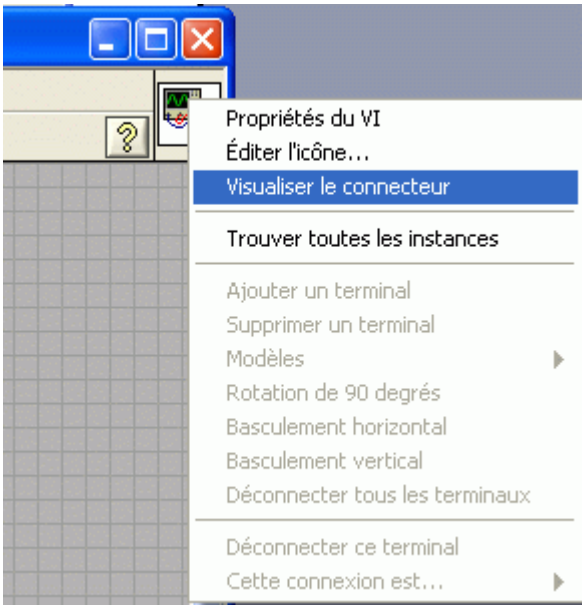


Figure 23 : visualiser le connecteur

Figure 24 : connecteur d'un *vi*

- le cas échéant choisir un autre modèle de connecteur (plus ou moins d'entrées et sorties), se souvenir que par convention, les entrées sont à gauche, les sorties à droite, pour cela faire un click droit sur l'icône/connecteur et choisir dans « Modèles » le modèle de connecteur adéquat. Remarque importante : lorsque le sous-*vi* est utilisé par d'autres *vi*, il est ennuyeux de changer de modèle de connecteur, donc il vaut mieux prévoir (sauf si l'on est absolument sûr de ne pas avoir à ajouter d'entrées ou sorties par la suite) plus d'entrées et sorties que celles qu'on utilise. Pour notre exemple, ce ne sera pas le cas : on peut donc choisir un modèle à 2 entrées et une sortie,
- il reste à relier les commandes et indicateurs de la face-avant avec les connecteurs : pour chacun, faire un click sur un connecteur libre, puis un click sur l'élément de face avant correspondant : la couleur correspondant au type de la commande ou du connecteur apparaît dans le connecteur, montrant qu'ils sont liés.

A partir d'ici, le *vi* peut être utilisé en tant que sous-*vi*. Cependant, il est fortement conseillé de lui donner une icône et une documentation.

Exercice d'application 5 : rendre un *vi* utilisable en tant que sous-*vi*
A faire : éditer les connecteurs d'« Aléatoire.vi », et les rendre obligatoires.

4.2.4.2 Etape conseillée : créer une icône personnalisée

LabVIEW permet d'utiliser un petit outil de dessin d'icônes. Pour cela, il faut faire un click droit sur l'icône, et choisir « Editer l'icône… ». Cet outil est malheureusement peu puissant et peu intuitif (voir Figure 25).

Figure 25 : édition d'icône

Commencer par éditer l'icône en 256 couleurs. Un conseil : on peut utiliser le copier/coller, donc ce qui donne de bons résultats en peu de temps consiste à s'inspirer d'icônes existantes et à les modifier. Ainsi, pour notre *vi*, on pourrait copier l'icône du générateur aléatoire de LabVIEW : le « truc » consiste à sélectionner le vi dans le diagramme, faire « Edition » → « Copier » ou bien Ctrl-C, puis d'ouvrir un éditeur d'images (Paint par exemple), de Coller dans l'éditeur d'image puis de copier cela à partir de l'éditeur d'images. Enfin, d'ouvrir l'éditeur d'icône, et de coller l'image. En résumé, cela donne copier l'icône du diagramme, coller dans l'éditeur d'images, copier à partir de l'éditeur d'images, coller dans l'éditeur d'icônes de LabVIEW… Vous vous demandez sans doutes pourquoi le copier/coller du diagramme vers l'éditeur d'icônes ne fonctionne pas… Moi aussi (en réalité il y a une explication un peu complexe due au type d'élément présent dans le presse-papier, qui n'est pas exactement une image quand on copie à partir du diagramme LabVIEW, alors que c'est exactement une image lorsqu'on copie à partir de l'éditeur d'image)…

Noter que le style de texte courant (taille, alignement, police) peut être modifié en double-cliquant sur le bouton « Entrée de texte ».

Lorsque l'icône est satisfaisante, on peut l'utiliser aussi pour le mode 16 couleurs et le mode noir et blanc. Ces icônes seront utilisées dans le cas où le *vi* est visualisé sur un écran ne disposant que de 16 couleurs ou étant noir et blanc. Pour cela, il suffit de cliquer sur « 16 couleurs » puis « Copier à partir de » « 256 couleurs ». L'icône 256 couleurs est alors recopié avec une diminution du nombre de couleurs dans « 16 couleurs ». Procéder de la même façon pour l'icône noir et blanc.

Voilà, l'icône, qui est là pour suggérer la fonction du *vi*, est créée : voici la mienne 

.

Exercice d'application 6 : créer l'icône d'un *vi*

A faire : créer l'icône du *vi* « Aléatoire.vi ».

4.2.4.3 Etape conseillée : créer une documentation

Lorsqu'on passe la souris sur l'icône du *vi*, on peut voir sa documentation par défaut dans la fenêtre d'aide contextuelle (voir Figure 26).

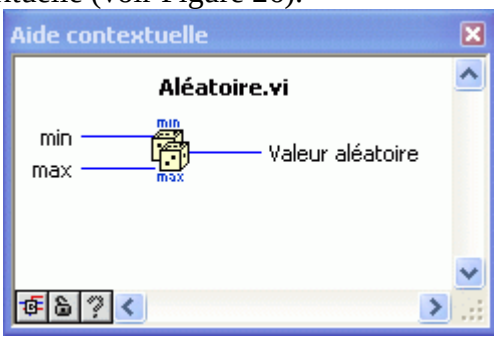


Figure 26 : documentation par défaut d'un *vi*

Une petite description supplémentaire est la bienvenue. Par exemple, au laboratoire, on appelle cela une spécification et au minimum, cela a la forme suivante :
Nécessite : néant (cela pourrait être max>min>=0)
Entraîne : Valeur aléatoire est une valeur aléatoire dans l'intervalle [min..max] tirée de façon uniforme.

Pour entrer cette documentation, faire un click droit sur l'icône du *vi*, et choisir « Propriétés du VI », pour la catégorie, choisir « Documentation », et entrer le texte. A partir de maintenant, passer la souris sur notre icône (en particulier quand il est utilisé en tant que sous-*vi*) affichera une aide contextuelle très utile (voir Figure 27).

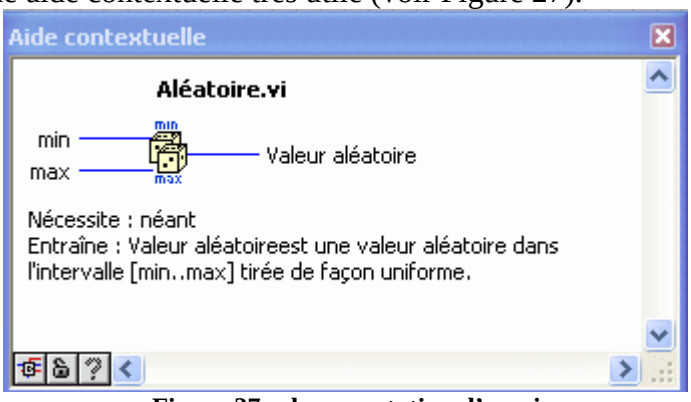


Figure 27 : documentation d'un *vi*

Exercice d'application 7 : créer la documentation d'un *vi*
A faire : créer la documentation du *vi* « Aléatoire.vi ».

4.2.4.4 Utiliser un sous-vi personnel dans un vi

Il y a différentes façons de placer l'icône d'un sous-*vi* dans le diagramme d'un autre *vi* :

- utiliser l'outil flèche de la souris pour déplacer (glisser-déposer) l'icône du *vi* dans un autre diagramme,
- prendre le fichier (par exemple « Aléatoire.vi ») dans l'explorateur de fichiers et faire un glisser-déposer sur le diagramme,
- dans le diagramme, utiliser la palette de fonctions et choisir « Sélectionner un vi… »,
- bien entendu, faire un copier-coller du vi dans un diagramme fonctionne aussi.

Notons au passage que LabVIEW n'autorise pas la récursivité, ainsi un *vi* ne peut s'appeler lui-même. Il existe des moyens détournés mais assez complexes de contourner cette limitation.
Il peut être intéressant, lorsqu'on utilise souvent certains *vi* personnels, de les avoir toujours à portée de main. On pourrait ainsi les placer dans la palette de fonctions. On peut personnaliser de deux façons la palette de fonctions :

- la façon la plus simple consiste à faire apparaître son *vi* dans la palette « Bibliothèques utilisateur ». Pour ce faire, il suffit d'enregistrer son vi dans le dossier « user.lib » de LabVIEW (LabVIEW se trouve généralement, sous Windows, à l'emplacement « C:\Program Files\National Instruments\LabVIEWx.y » où x.y est la version (en ce moment 8.2). Noter qu'il faut pour cela avoir les permissions d'écriture dans ce dossier. LabVIEW doit être redémarré pour que la palette soit mise à jour, et présente les *vi* présents dans ce dossier (voir Figure 28).

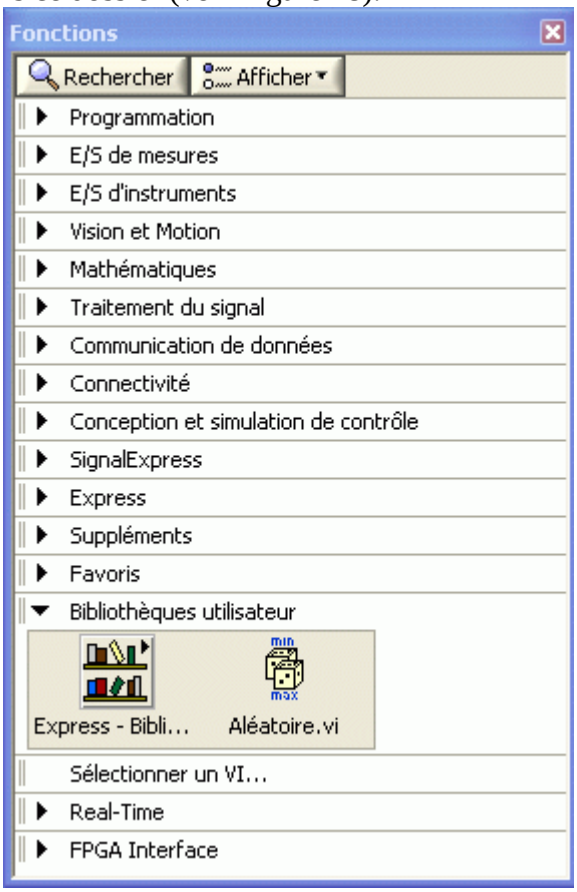


Figure 28 : déposer un *vi* dans « user.lib » le fait apparaître dans la palette « Bibliothèque utilisateur »

- une seconde manière, plus avancé (niveau expert) est utilisée lorsque l'on a créé complètement une bibliothèque. Elle consiste à regrouper sa bibliothèque dans une arborescence de dossiers. Le dossier racine de cette arborescence est alors mis dans le dossier « vi.lib\addons » du dossier de LabVIEW. Ensuite, on pourra utiliser l'édition de menus pour personnaliser le résultat.

Par conséquent, en général, on se contente de déposer les *vi* personnels dans « user.lib » jusqu'à ce qu'on ait à se poser la question de créer une bibliothèque personnelle lorsque la « Bibliothèque utilisateur » contient trop de *vi*.

Exercice d'application 8 : connaître l'arborescence des fichiers LabVIEW
A faire : trouver le dossier LabVIEW avec l'explorateur de fichiers. Regarder ce qui se trouve dans « user.lib », et dans « vi.lib ».

4.2.5 Structures de contrôle

Tout langage de programmation propose des structures de contrôle : l'alternative (structure conditionnelle), des boucles à nombre d'itération connue (boucle Pour), et des boucles conditionnées par une condition d'arrêt (répéter … jusqu'à/répéter … tant que). LabVIEW propose ces structures de contrôle, mais il est important de noter qu'il ne propose pas de boucle de type *while* (Faire tant que …). La structure de contrôle inhérente à la programmation flot de données est la séquence, dont nous avons parlé auparavant.

Toutes les structures de contrôle se trouvent dans la palette « Programmation » → « Structures ».

Afin d'illustrer et d'utiliser ces structures de contrôle, nous créerons un petit jeu de hasard au fil des explications, qui utilisera le générateur pseudo-aléatoire fait en section 4.2.4.

4.2.5.1 La structure conditionnelle

La structure conditionnelle peut correspondre à un *if … then … else …* ou bien à un *case … is*.

4.2.5.1.1 La structure de choix if/then/else

La Figure 29 montre une structure de choix de type *if/then/else*. Dans un langage de programmation textuel, cela correspondrait à (avec || opérateur faire en parallèle) :

Saisir A || Saisir B
Si A = B alors
    Txt ← "A et B sont égaux"
Sinon
    Txt ← "A et B sont différents"
Fin si
Afficher Txt

LabVIEW représente l'alternative à l'aide de plusieurs diagrammes alternatifs (ici le diagramme pour le cas vrai sera exécuté quand le booléen entrée sur le « ? » est vrai, alors que celui du cas faux sera exécuté quand ce booléen est faux).

Important :

- C'est le type de la donnée arrivant sur la condition (« ? ») qui fait que l'on a un *if/then/else* dans le cas où ce type est booléen,
- on peut faire entrer n'importe quelle donnée dans cette structure,
- si une donnée sort, alors elle doit avoir une valeur quel que soit le cas (ici, une chaîne de caractères sort de la structure : on lui a donné une valeur dans le cas vrai, et une valeur dans le cas faux), si un cas ne donne pas de valeur à une sortie (voir Figure 30) c'est une erreur de syntaxe. Notons qu'en sortie d'une structure de contrôle, le « carré » s'appelle un **tunnel**. L'erreur de syntaxe associée s'appelle « il manque un élément relié au tunnel »,
- toute structure est considérée comme un nœud (comme si c'était un sous-*vi*) avec des entrées et des sorties : le nœud devient exécutable quand toutes ses entrées sont disponibles, et ses sorties sont générées quand l'exécution du nœud (donc de tout ce qu'il contient) est terminée. Ainsi, sur la Figure 31, l'exécution de la structure conditionnelle ne se fait que lorsque toutes les valeurs en entrée sont disponibles, c'est-à-dire après exécution parallèle de *Saisir A*||*Saisir B*||*Attendre 3 secondes*. Dans ce cas, la structure conditionnelle ne s'exécutera au mieux que 3 secondes après le lancement. Noter qu'on peut utiliser si on le souhaite toute valeur donnée en entrée (y compris la valeur câblée sur « ? » considérée elle aussi comme un entrée).

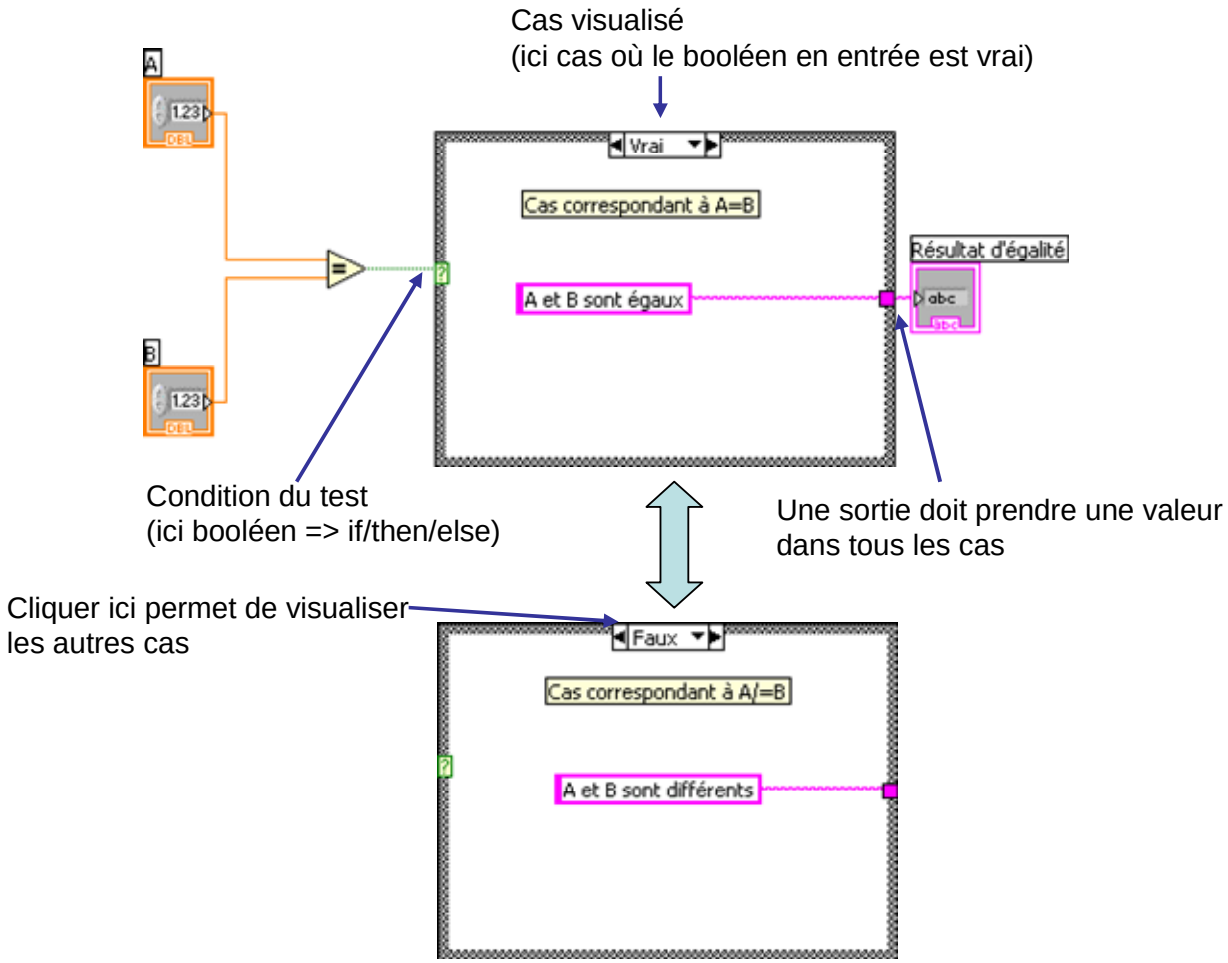


Figure 29 : la structure de choix avec une condition booléenne en entrée est un if/then/else

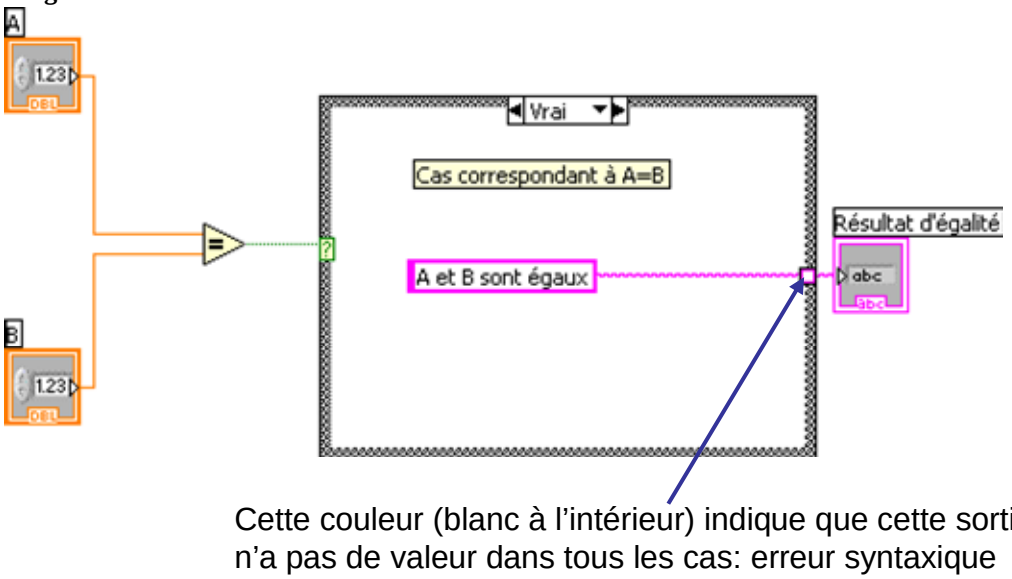


Figure 30 : erreur de syntaxe typique dans une structure conditionnelle : un cas ne donne pas de valeur à une sortie

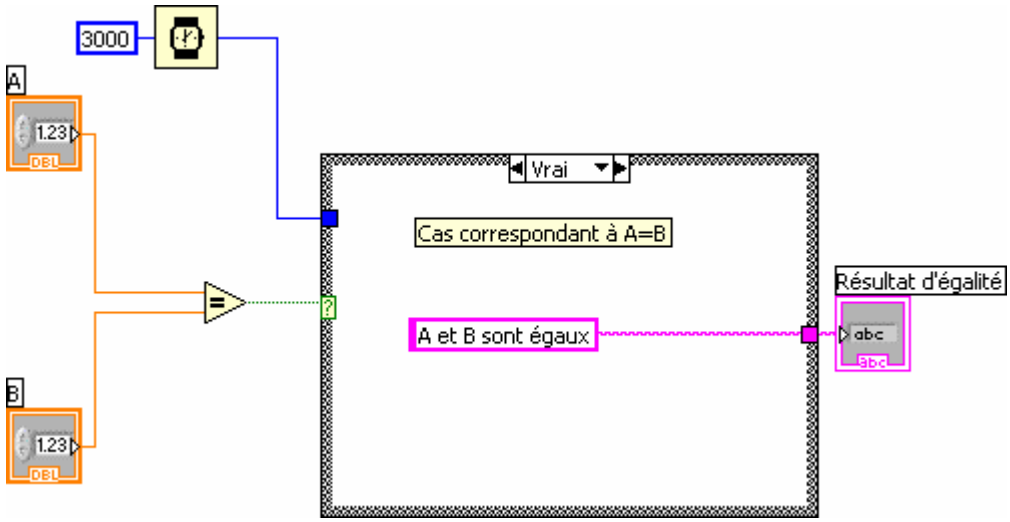


Figure 31 : une structure est comme tout nœud LabVIEW, elle ne s'exécute que lorsque toutes ses entrées sont disponibles

Exercice d'application 9 : créer une structure conditionnelle, utiliser « Animer l'exécution »
A faire : implémenter le *vi* donné sur la Figure 31, exécuter en mode « Animer l'exécution ». Les valeurs de A et B sont-elles lues dès le début, ou après 3 secondes ?

Exercice d'application 10 : utiliser une structure conditionnelle, utiliser un sous-*vi* personnel, utiliser une boîte de dialogue
A faire : créer un nouveau *vi* utilisant « Aléatoire.vi ». Il doit implémenter le code suivant :
Saisir entier Proposition||Tirer un nombre aléatoire entre 1 et 12
Si ces entiers sont égaux alors
    Text ← "Vous avez gagné"
Sinon
    Text ← "Vous avez perdu"
Fin si
Afficher Text dans une boîte de dialogue
Pour la boîte de dialogue on utilisera la fonction de recherche de la palette de fonctions.

4.2.5.1.2 La structure à choix multiple

La seconde forme d'utilisation de la structure de choix est la structure à choix multiple. La seule différence par rapport à la structure conditionnelle est le type de donnée câblé en entrée : cela peut être un type scalaire (numérique, type énuméré, chaîne de caractères). Sur la Figure 32, on a placé une chaîne de caractères en entrée de la structure, elle fonctionne donc comme une structure de choix, noter qu'il y a alors un cas « **Par défaut** » qui sera le cas choisi si aucune des autres conditions n'est satisfaite. Pour ajouter/enlever des conditions ou bien définir la condition par défaut, il suffit de faire un click droit au niveau des choix (en haut de la structure).

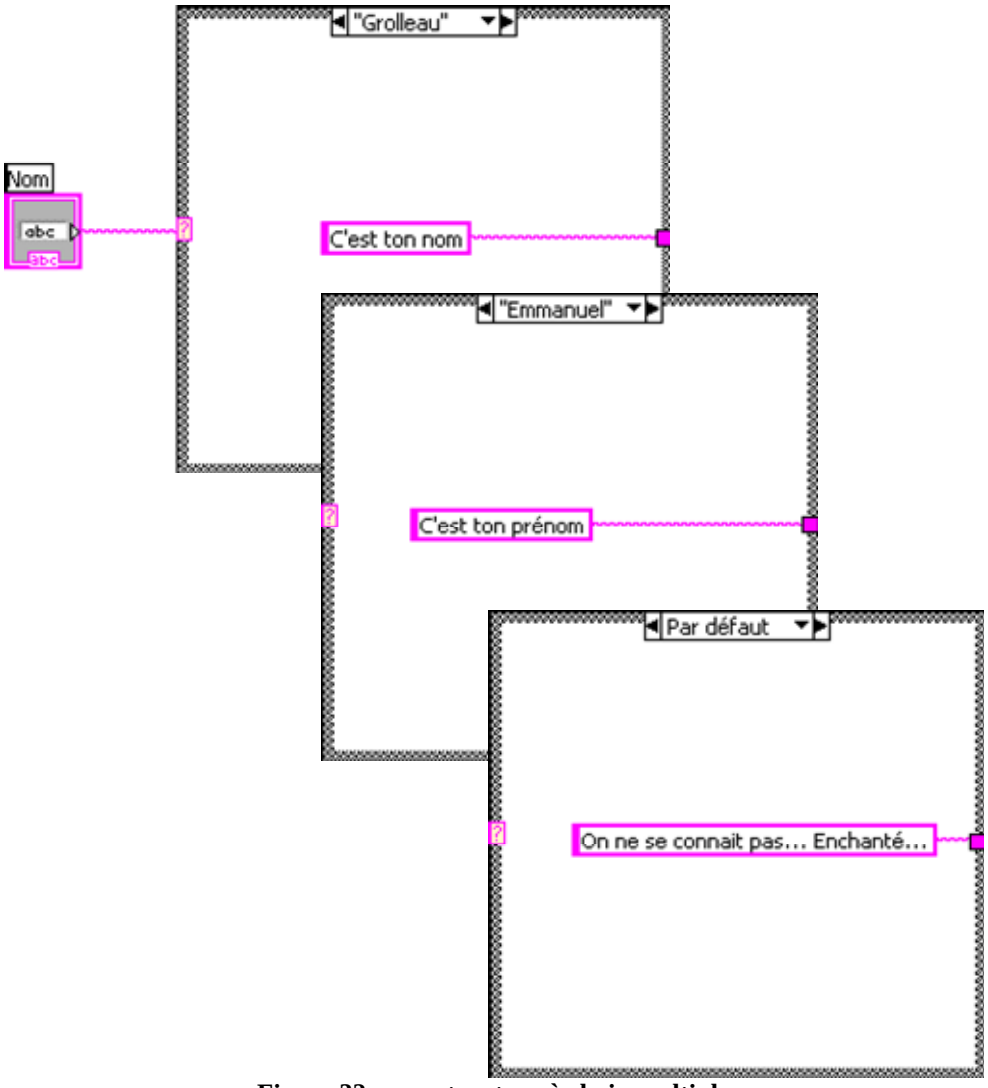


Figure 32 : une structure à choix multiple

4.2.5.2 La boucle For

La boucle *For* (boucle pour) correspond en langage de programmation textuelle à *Pour i de 0 à N-1 faire…* Ainsi, sur la Figure 33, on peut voir le décompte N du nombre d'itérations de la boucle, et l'indice d'itération i donnant le numéro d'itération en cours (variant de 0 à N-1).

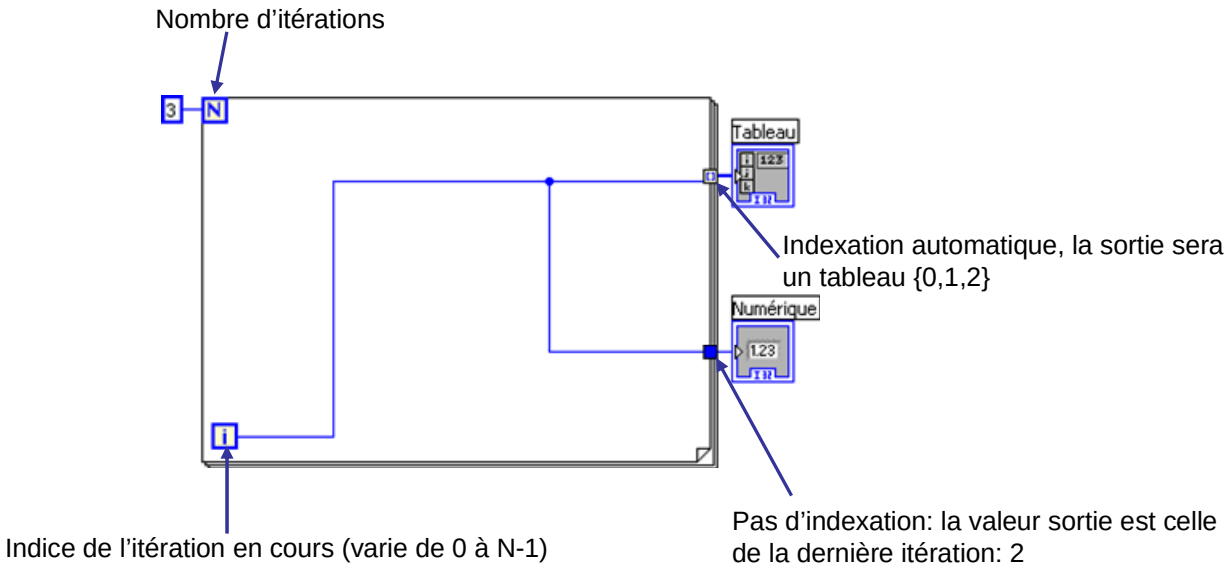


Figure 33 : une boucle For

4.2.5.2.1 Indexation ou non des sorties

Le diagramme contenu dans une boucle est exécuté N fois, et les **sorties ne sont générées qu'à la terminaison de la boucle**, c'est-à-dire à la fin de la 3ème itération pour l'exemple donné ici. On a le choix entre **indexer une sortie** (chaque élément obtenu à chaque itération est mis dans un tableau, à la fin de l'exécution de la boucle, c'est donc le tableau de tous les éléments calculés qui sort de la boucle) et **ne pas indexer une sortie** (dans ce cas, la valeur sortie est la dernière valeur calculée, i.e. celle obtenue lors de la dernière itération). Graphiquement, le diagramme représente le tunnel de sortie indexée avec des crochets []alors qu'une sortie non indexée est représentée par un tunnel (carré) plein. Pour changer le mode d'un tunnel (qu'il soit en entrée ou en sortie) d'indexé à non indexé, faire un click droit sur le tunnel (voir Figure 34).

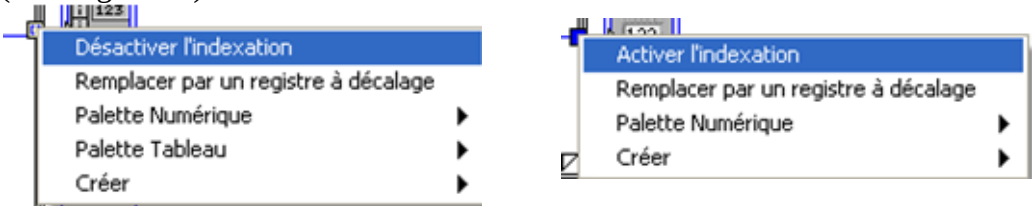


Figure 34 : activer/désactiver l'indexation

Exercice d'application 11 : comprendre le fonctionnement d'une boucle For
A faire : implémenter le *vi* donné sur la Figure 33, exécuter en mode « Animer l'exécution » : constater que les valeurs ne sortent de la boucle qu'à la fin de son exécution.

4.2.5.2.2 Indexation des entrées

De la même façon, les entrées peuvent être automatiquement indexées : ainsi, si un tableau de dimension *n* est indexé en entrée d'une boucle, c'est un tableau de dimension *n-1* qu'on retrouve à l'intérieur de la boucle : à l'itération *i*, la valeur d'entrée est le $i^{ème}$ élément du tableau indexé en entrée. Ainsi, sur la Figure 35, à chaque itération *i* de la boucle, le $i^{ème}$ élément du tableau donné en entrée est utilisé. Noter que dans le cas où un tableau seul est donné en entrée d'une boucle For, **il n'est pas nécessaire de câbler le nombre d'itérations N** : LabVIEW comprend alors que la boucle consiste à indexer tout le tableau (donc implicitement, N est la taille du tableau indexé en entrée). Dans notre exemple, à chaque itération on prend un élément du tableau, auquel on ajoute 5, puis on le met en indexation en

sortie. A la fin de la boucle, le tableau de sortie contient chaque élément du tableau d'entrée auquel on a ajouté 5.

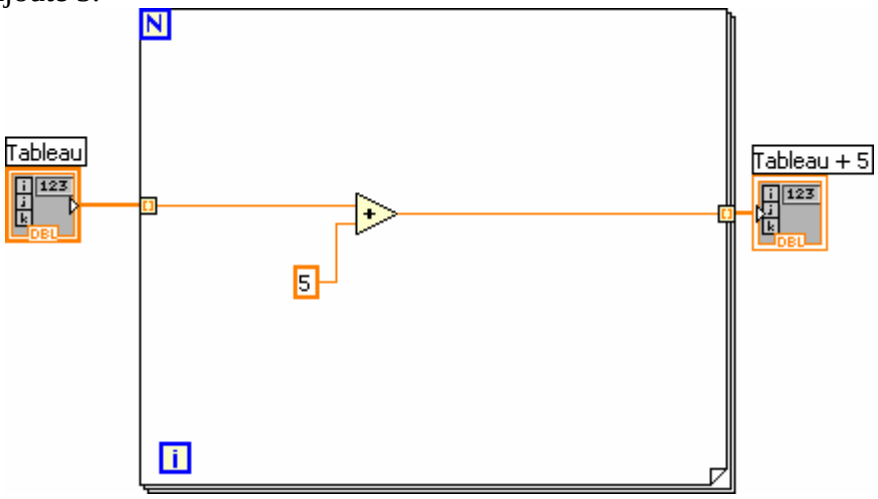


Figure 35 : indexation automatique d'un tableau en entrée

Notons que l'exemple donné ici est stupide puisque LabVIEW autorise les opérations arithmétiques entre scalaires et tableaux (on peut donc directement ajouter 5 à un tableau).

Exercice d'application 12 : utiliser une boucle For

A faire : prendre le *vi* de l'Exercice d'application 10. Le modifier afin de tirer 3 fois un nombre aléatoire, chaque tirage est espacé d'1 seconde, et affiché. Le dernier tirage est alors celui comparé au nombre donné par l'utilisateur.

4.2.5.2.3 Registre à décalage

D'après la philosophie flots de données, il ne peut y avoir aucun cycle dans le flot : en effet, puisqu'un nœud ne s'exécute que lorsque toutes ses entrées sont disponibles, et ne fournit ses sorties qu'après avoir été exécuté, un diagramme contenant un cycle ne serait pas exécutable. Cependant, supposons que l'on souhaite calculer la moyenne des éléments d'un tableau (bien entendu, on pourrait utiliser un *vi* de calcul de moyenne, mais nous supposerons que nous voulons programmer cela). Dans un langage textuel, on écrirait :

Moyenne ← 0
Pour i de 0 à taille(tableau)-1 faire
    *Moyenne ← ((Moyenne * i)+tableau[i])/(i+1)*
    Afficher Moyenne
Fait

Si on souhaite exprimer le fait que la moyenne à l'itération *i-1* est la moyenne utilisée à l'itération *i* comme on souhaiterait le faire sur la Figure 36, on commet une erreur syntaxique puisqu'on crée un cycle. C'est une erreur de syntaxe, certes, mais ce n'est pas une erreur de raisonnement : en effet, nous n'exprimons pas ainsi un cycle. N'oublions pas qu'une boucle correspond à la représentation superposée d'un diagramme qui doit être exécuté plusieurs fois.

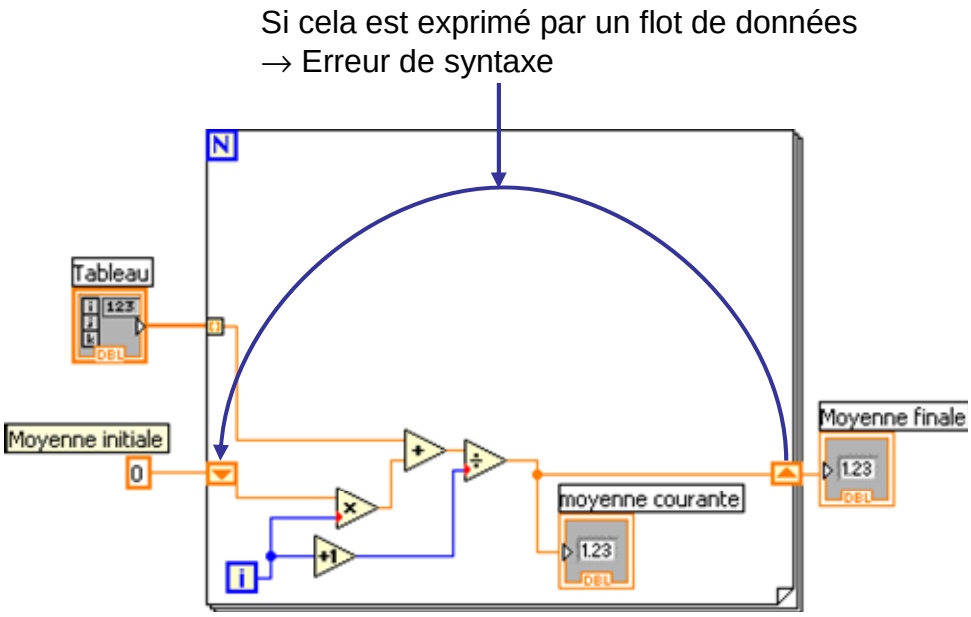


Figure 36 : la rétroaction dans une boucle ne peut se faire via un flot de données classique

En réalité, ce que nous souhaitons représenter correspond à la vue qui est donnée sur la Figure 37. Il n'y a pas de cycle : le calcul d'une itération est donné en entrée de la suivante, comme si le tunnel de sortie de l'itération était relié au tunnel d'entrée de l'autre. Cependant, un tunnel ne sort pas de valeur aux à la fin de chaque itération mais uniquement à la fin de la dernière itération.

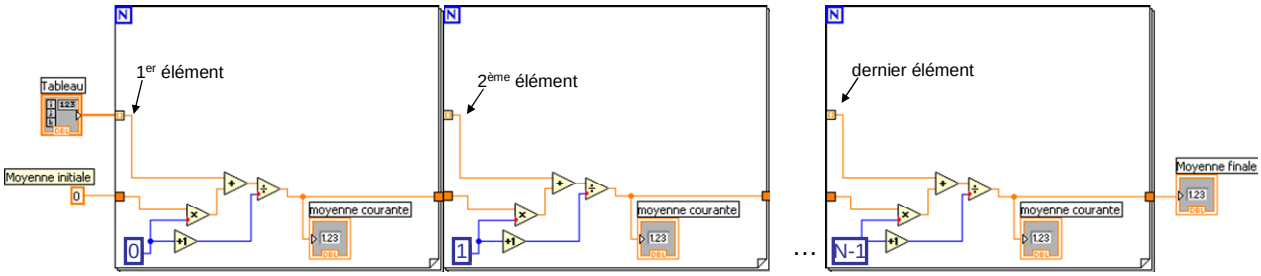


Figure 37 : représentation figurée d'une boucle « déroulée »

LabVIEW offre pour pallier cela le **registre à décalage**. Sous ce nom barbare ne se cache que « relier la sortie d'une itération à l'entrée de la suivante ». D'un point de vue pratique, il suffit pour créer un registre à décalage de faire un click droit sur le tunnel qu'on souhaite transformer en registre à décalage et de choisir « Remplacer par un registre à décalage ». On est alors amené à désigner le tunnel correspondant (l'entrée si on a remplacé la sortie par un registre à décalage, la sortie si on a remplacé l'entrée). Le *vi* correct de calcul de moyenne est donné sur la Figure 38.

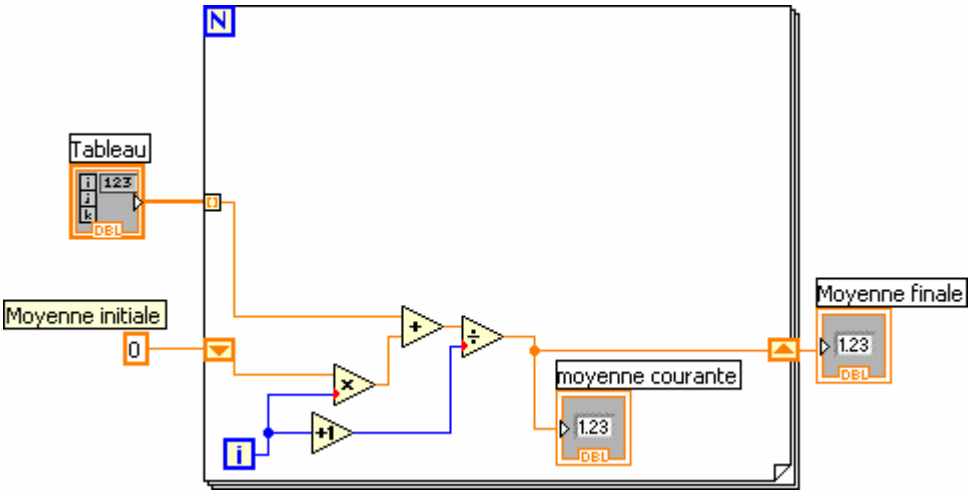


Figure 38 : utilisation d'un registre à décalage

Notons au passage que le 0 en entrée du registre à décalage donne sa valeur initiale, et que la valeur du registre en sortie de boucle est la dernière valeur calculée (comme si c'était un tunnel classique). Il est important de **toujours donner une valeur initiale à un registre à décalage** (sauf dans certains cas spécifiques). En effet, si on ne donne pas de valeur initiale, le comportement est le suivant : lorsque le registre n'a jamais pris de valeur auparavant, il prend la valeur par défaut du type comme valeur initiale. Par contre, si le *vi* est appelé plusieurs fois, sa valeur initiale est la dernière valeur inscrite (c'est-à-dire lors de l'appel précédent).

Exercice d'application 13 : manipuler un registre à décalage, et un nœud de rétroaction

A faire : s'inspirer du *vi* donné sur la Figure 38 pour vérifier que la moyenne du générateur aléatoire est bien au milieu de l'intervalle min..max (faire 1000 itérations). Remplacer le registre à décalage par un **nœud de rétroaction** (c'est juste une représentation alternative du registre à décalage). Ensuite, afin d'avoir le temps de voir « Moyenne courante » converger, temporiser la boucle. Le principe est le suivant : une itération ne peut avoir lieu avant que l'itération précédente ne soit terminée, il suffit donc de ralentir l'exécution d'une itération en mettant une attente (voir Figure 31) en parallèle du reste de l'itération.

4.2.5.3 La boucle faire tant que/faire jusqu'à

Le second type de boucle est la boucle *While* (*faire … tant que* ou *faire … jusqu'à).* Noter qu'il n'existe pas de boucle *faire tant que …*, c'est-à-dire qu'en LabVIEW au moins une itération d'une telle boucle a lieu (on peut toujours utiliser une structure conditionnelle afin de ne rien faire si la condition de boucle n'est pas satisfaite). La plus typique des boucles *tant que* est la boucle que l'on trouve en général au niveau du programme lui-même. En effet, comme cela a été dit précédemment, l'exécution « en continu » d'un *vi* est à proscrire. Si l'on souhaite faire une exécution « en continu » jusqu'à ce que l'utilisateur souhaite arrêter, alors on place une boucle *While* comme sur la Figure 39. Cette figure met en évidence les deux modes d'arrêt possibles pour une boucle *While* : arrêter sur une condition vraie, et continuer sur une condition vraie. Typiquement, si on souhaite conditionner l'arrêt à l'appui sur un bouton « stop », étant donné qu'un tel bouton renvoie faux quand il n'est pas appuyé, et vrai sinon, la boucle doit s'arrêter lorsque le bouton renvoie vrai.

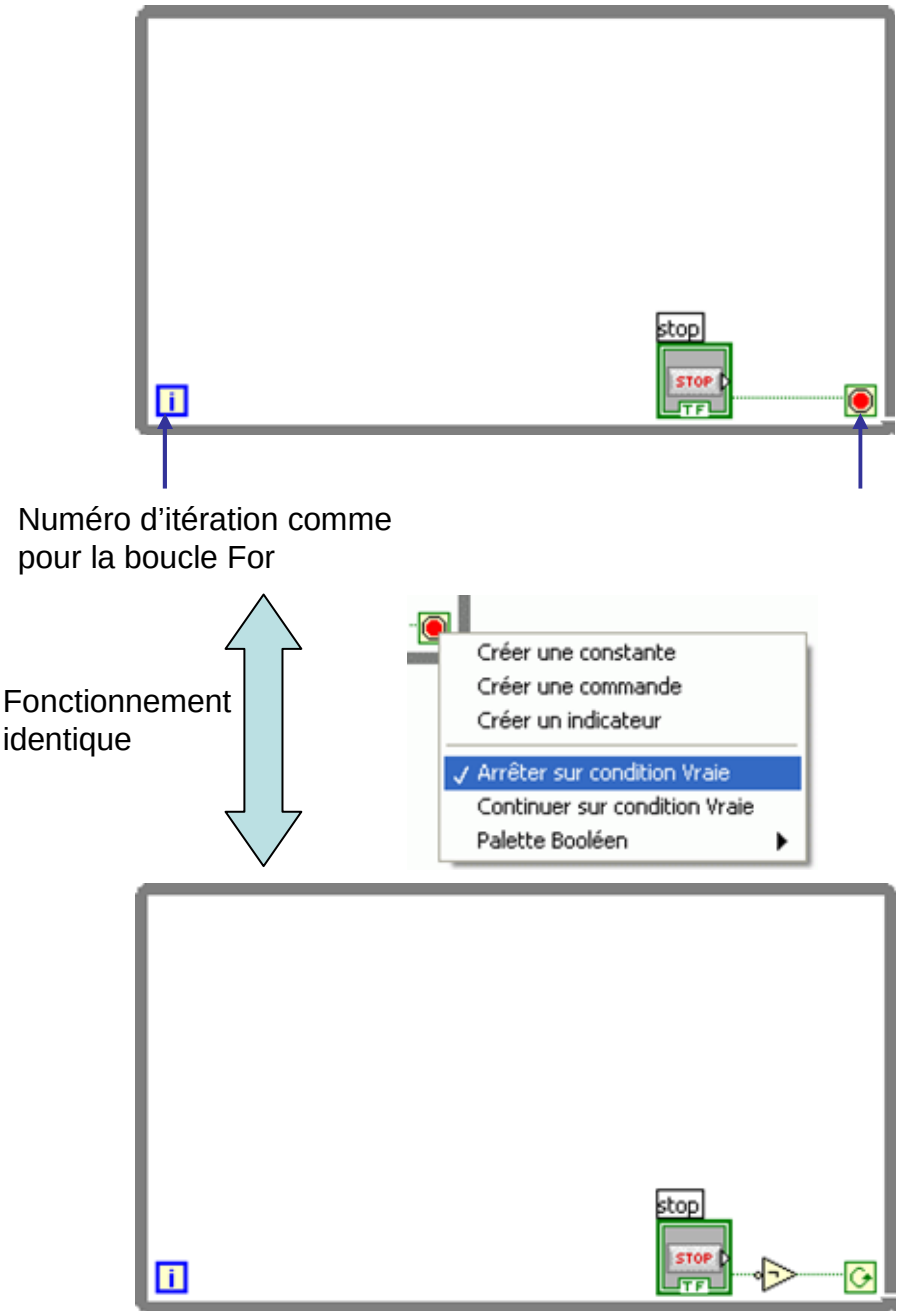


Figure 39 : un boucle *While possède deux possibilités de programmer l'arrêt*

Exercice d'application 14 : créer une boucle *While*, comprendre le fonctionnement du tunnel d'entrée, comprendre l'action mécanique des boutons

A faire : Implémenter le *vi* de la Figure 39. Passer en mode « Animer l'exécution » et tester que l'arrêt s'effectue comme il faut. Sortir le bouton « Stop » de la boucle, et observer le comportement. Qu'en déduisez-vous ?

Aller sur la face avant, faire un click droit sur le bouton, et tester les différentes actions mécaniques possibles (toujours en mode « Animer l'exécution »).

Exercice d'application 15 : utiliser une boucle *While*, comprendre les problèmes liés à l'absence de séquencement

A faire : Placer le contenu de l'Exercice d'application 12 dans une boucle *While*. La condition d'arrêt est donnée par la réponse de l'utilisateur à une boîte de dialogue à 2 boutons lui demandant s'il souhaite recommencer. Le comportement n'est pas le comportement attendu, pourquoi ?

4.2.5.4 La séquence

La structure séquence sert dans des cas comme celui de l'Exercice d'application 15 : lorsque l'on souhaite obliger un certain ordre dans les actions, mais qu'il n'y a pas moyen d'utiliser les flots de données pour le forcer.

LabVIEW propose 2 structures séquence : l'une empilée, l'autre déroulée. Il n'y a pas de différence sémantique, mais uniquement des différences de représentation entre les 2. La séquence déroulée étant plus lisible, et plus simple d'utilisation, je conseillerais de l'utiliser dans toute situation. Pour créer une telle séquence, on place une première étape de séquence, puis à l'aide d'un click droit sur le bord de celle-ci (voir Figure 40) on peut ajouter d'autres étapes. L'idée est qu'une étape ne peut commencer avant que l'étape précédente ne soit terminée.

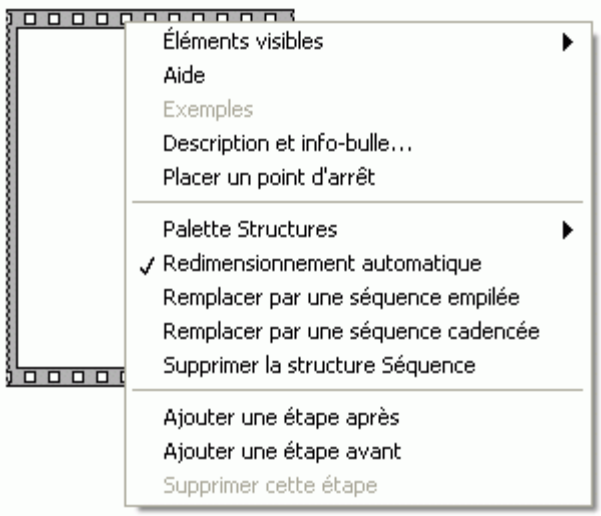


Figure 40 : création d'une structure séquence déroulée

Exercice d'application 16 : utiliser une structure séquence, utiliser un point d'arrêt

A faire : modifier le *vi* de l'Exercice d'application 15 afin que l'affichage de la boîte de dialogue demandant si l'on veut continuer se fasse après qu'on ait affiché le résultat du tirage. Placer un point d'arrêt sur le cas vrai de la structure conditionnelle et exécuter le *vi* (sans être en mode « Animer l'exécution »). Exécuter quelques pas « pas-à-pas ».

4.3 Utilisation avancée

Nous avons maintenant vu les éléments de base de la programmation LabVIEW. Cette partie propose de voir quelques graphes proposés par LabVIEW, la façon dont il gère les fichiers, et surtout la façon dont on peut créer une interface graphique. Au passage, divers autres éléments de programmation seront abordés.

4.3.1 Utilisation de graphes

Les 3 principaux types de graphe proposés par LabVIEW sont le graphe déroulant (affiche des courbes point par point), le graphe (affiche les courbes à partir du/des tableaux contenant tous les points), et le graphe XY (courbes paramétriques).

4.3.1.1 Graphe déroulant

Le **graphe déroulant** est un indicateur 2D représentant une courbe (ou plusieurs courbes) dont les points sont donnés point par point (i.e. on ne donne pas tous les points d'une courbe en une seule fois).

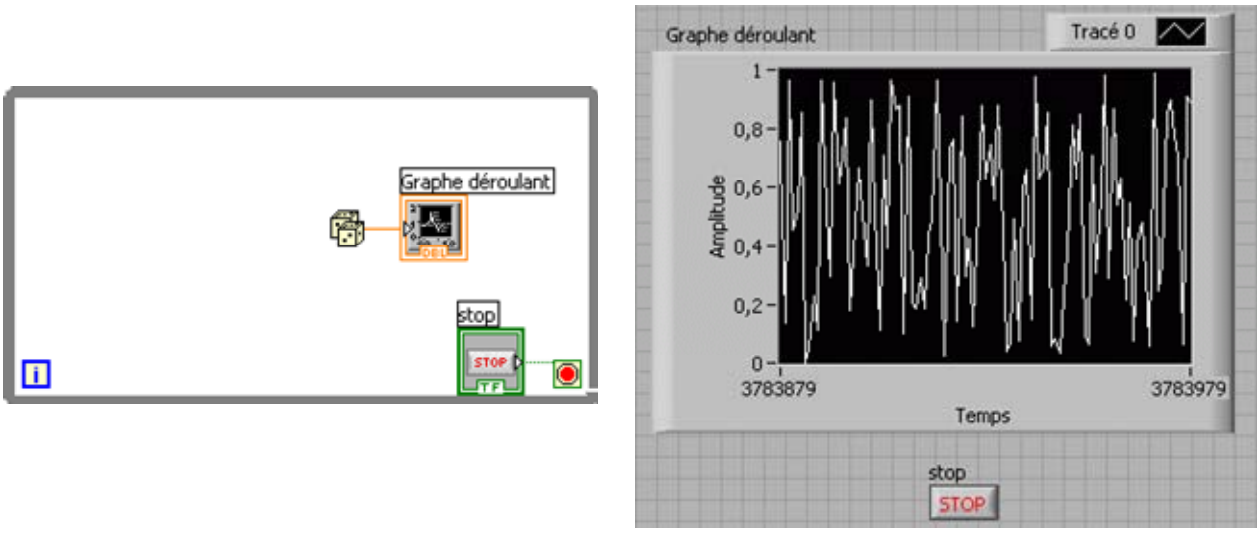


Figure 41 : un graphe déroulant

La Figure 41 représente un graphe déroulant : en abscisse le temps (par défaut, mais tous les éléments peuvent être édités), en ordonnée, l'amplitude. Le nom d'une courbe par défaut est « Tracé 0 », 1, 2, etc. Mais bien entendu tout ceci peut être modifié. Pour modifier l'apparence et le comportement d'un graphe, on fait un click droit sur le graphe, ou sur le tracé concerné. Ainsi, par exemple, on peut mettre une barre de défilement (attention, on ne peut défiler que sur la **longueur de l'historique** que l'on a définie), le graphe peut se mettre à l'échelle automatiquement (pour les X et les Y), etc. Etant donné qu'un graphe déroulant conserve son contenu, on peut l'effacer en faisant click droit sur le graphe → « Opérations sur les données » → « Effacer le graphe déroulant ».

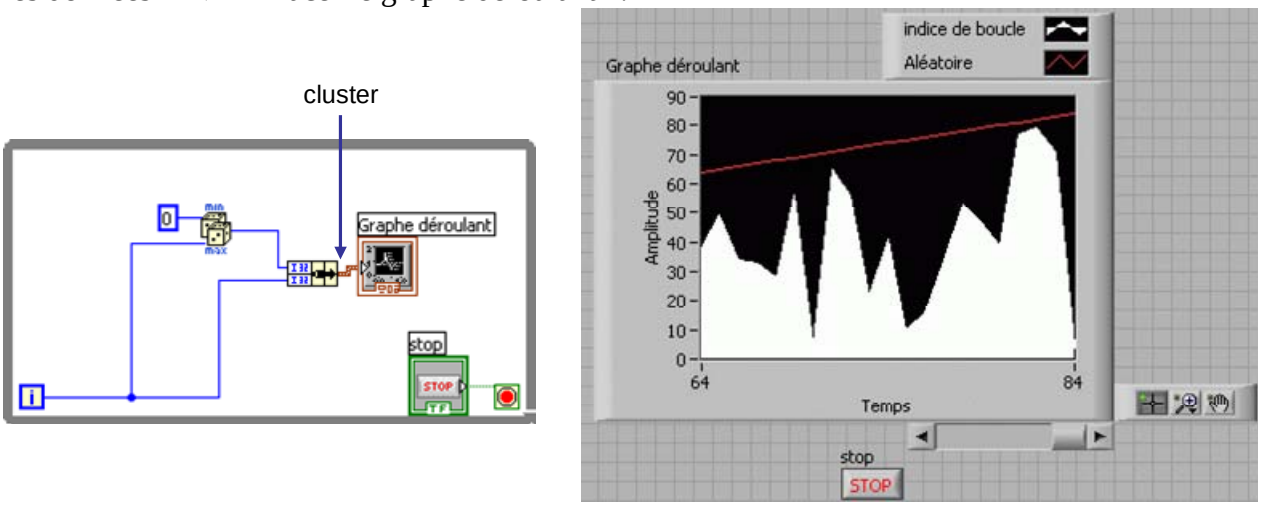


Figure 42 : graphe déroulant multi-courbes

On peut afficher plusieurs courbes comme sur la Figure 42 en plaçant les points de chaque courbe à afficher dans un *cluster*.

Exercice d'application 17 : créer un graphe déroulant multicourbe, personnaliser l'affichage
A faire : Implémenter le *vi* de la Figure 42. Afficher la barre de défilement de l'axe X, configurer l'historique à 10000 valeurs, changer le style de courbes, et leur couleur, donner un nom personnel aux courbes.

4.3.1.2 Graphe

Un graphe nécessite un tableau de points ou un *waveform* par courbe affichée. Sur la Figure 43, on utilise un *vi* de la palette « Traitement du signal » → « Génération de signaux » qui donne un tableau de flottants en sortie.

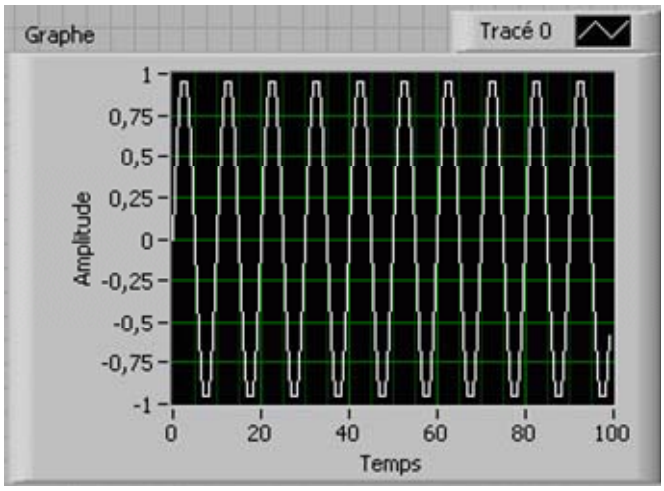
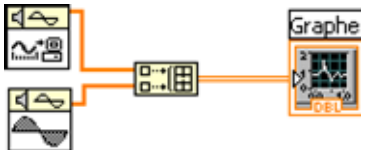


Figure 43 : un graphe

La gestion de plusieurs courbes peut être faite en envoyant un tableau à plusieurs dimensions (« Concaténer des tableaux » prenant les différentes courbes à afficher, voir).



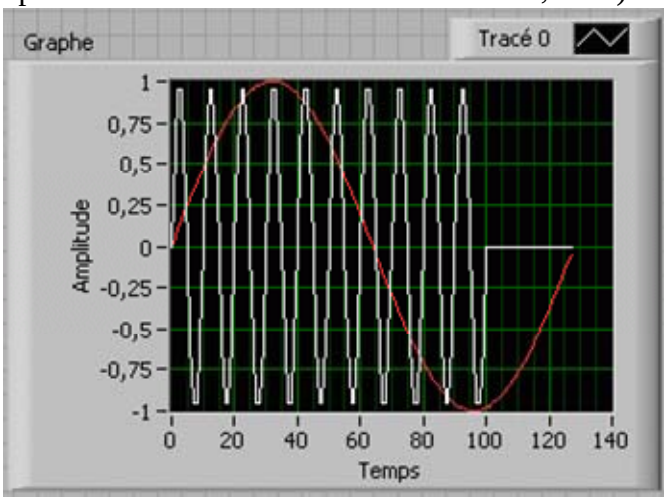


Figure 44 : un graphe multi-courbes

Exercice d'application 18 : créer un graphe, créer un histogramme

A faire : reprendre le *vi* de l'Exercice d'application 13, sortir le tableau des tirages de la boucle, en faire un histogramme général (utiliser la recherche dans la palette de fonctions), et afficher cet histogramme dans un graphe (on choisira bien entendu un tracé de type histogramme). Peut-on en déduire que le générateur aléatoire fonctionne bien ?

Nous pouvons constater que les points sont supposés avoir un espacement d'une unité de temps sur l'axe des X (temps), ce qui n'est pas le cas lors de mesures. Très souvent, les mesures sont horodatées, et la durée séparant 2 mesures est connue. Un groupe constitué d'un ensemble de points, d'une date de début de mesure, et d'une durée entre 2 mesures est généralement exprimée sous forme de *waveforms*. La Figure 45 représente l'utilisation d'un *waveform*, notez l'effet sur le graphe, qui affiche alors des dates on non plus des unités de temps. Dans l'exemple donné, chaque point est supposé être distant d'une seconde du précédent, avec le 1er point ayant liu à la date courante. De nombreux *vi* d'acquisition de données donnent directement les mesures sous forme de *waveforms*.

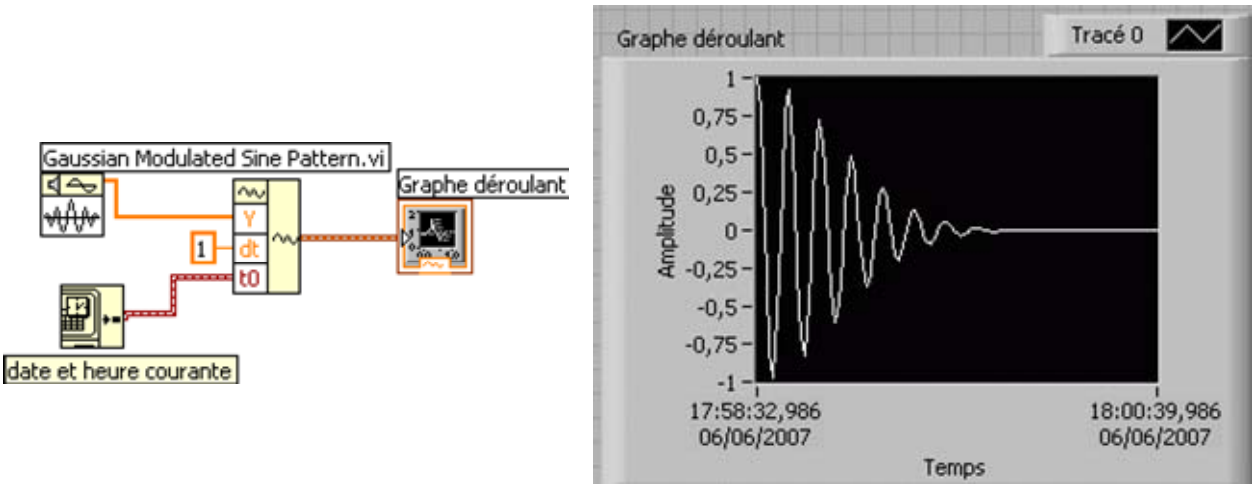


Figure 45 : un graphe construit à partir d'un *waveform*

4.3.1.3 Graphe XY

Le graphe XY permet de tracer des courbes paramétriques. Pour ce faire, il faut envoyer un *cluster* contenant les points pour X, et les points pour Y sur chaque composante du *cluster* en entrée du graphe (voir Figure 46).

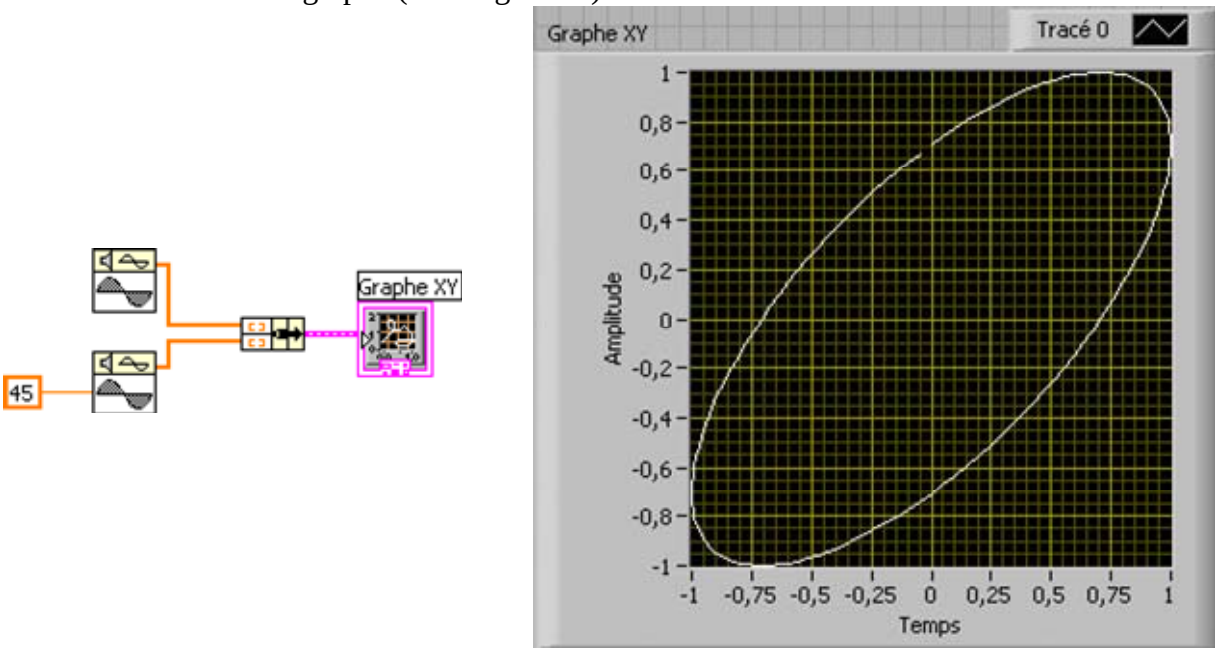


Figure 46 : un graphe XY

Pour afficher plusieurs courbes, on placera les *clusters* de chaque courbe dans un tableau qu'on envoie sur le graphe XY (voir Figure 47).

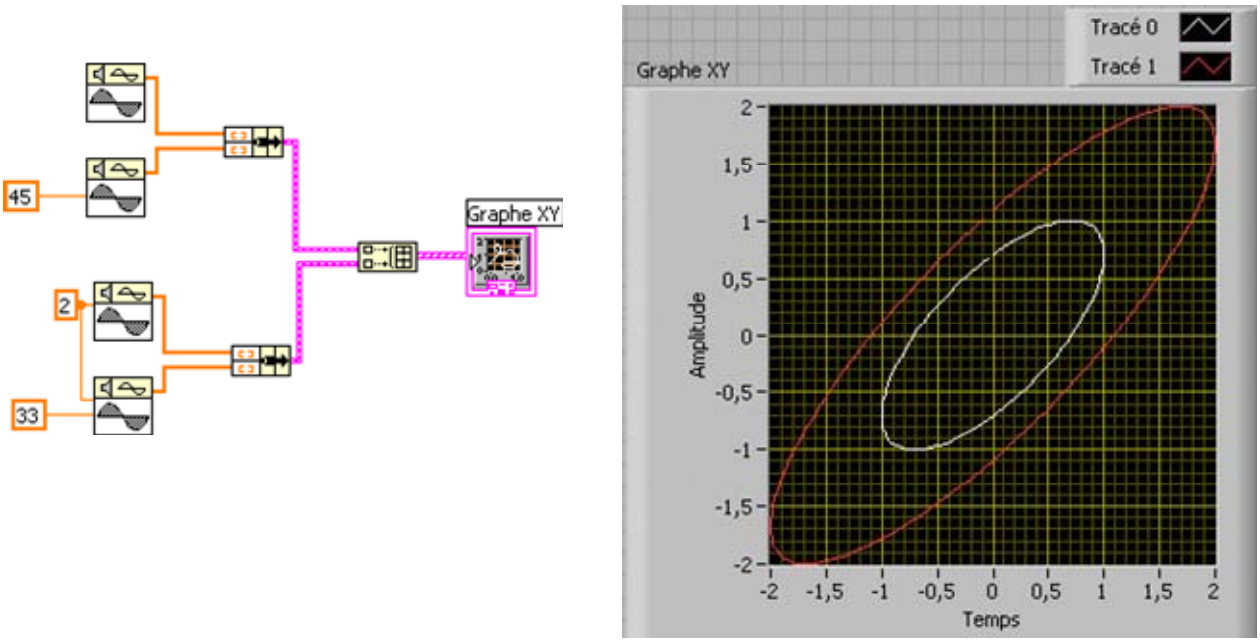


Figure 47 : un graphe XY multi-courbes

4.3.2 Gestion de fichiers

LabVIEW propose plusieurs niveaux de gestion de fichiers dans la palette et les sous-palettes « E/S sur fichiers ». Dans n'importe quel langage de programmation, la gestion de fichier nécessite les étapes suivantes :

- Ouverture du fichier : pour cela il est nécessaire de connaître le nom du fichier pour le système (par exemple sous Windows « c:\dossier\fichier.csv »). Il y a deux possibilités d'action : soit l'on connaît le nom du fichier au moment de l'ouverture, soit on ne le connaît pas et on le demande à l'utilisateur en ouvrant une boîte de dialogue de sélection du fichier. Lors de l'ouverture, on doit choisir le mode d'ouverture parmi :
    - Lecture seule
    - Lecture/écriture
    - Ecriture seule

  De même, le fichier s'il est inexistant peut être créé, ou bien s'il est existant, on peut choisir de le remplacer, ou d'ajouter des choses à la fin du fichier (*append*) ou plus rarement de remplacer des octets contenus. Noter qu'on insère pas dans un fichier : si l'on souhaite insérer des données à partir de l'octet *n*, alors, on doit ouvrir le fichier, se placer à la position *n*, lire les données restantes, inscrire à la position *n* les données à insérer suivies des données restantes (qui sont donc réinscrite).

  On peut noter qu'au plus un processus peut posséder un droit d'écriture sur un fichier ouvert.
- Après l'ouverture, on dispose d'un **descripteur de fichier** ou **identificateur de fichier** ou **référence de fichier**. Ce descripteur est en général utilisé pour toutes les opérations suivantes (lecture/écriture/fermeture). LabVIEW appelle cela un ***refnum***. Donc un *refnum* est obtenu après ouverture d'un fichier, et est valide jusqu'à sa fermeture.
- Les opérations de lecture ou d'écriture se font sur le *refnum*. En général, on distingue 2 types de fichiers : les fichiers binaires et les fichiers textuels. Les fichiers binaires sont par exemple des images, des vidéos, des archives compressées (mais il y a peu de chance que l'on édite de tels fichiers sans utiliser des *vi* spécialisés dans

ces traitements) ou encore des données pures (par exemple, sauvegarde des données d'un programme sous forme de leur type). Les fichiers textuels représentent la plupart des fichiers utilisés couramment (fichier .csv, fichier texte, fichier de configuration, etc.).

- A la fin, on ferme le fichier. Si l'on souhaite l'utiliser ultérieurement, il faudra le ré-ouvrir afin d'obtenir un nouveau *refnum*.

On retiendra donc qu'il y a 3 étapes lors de la gestion d'un fichier : ouverture, lecture/écriture, fermeture. Etant donné que LabVIEW s'adresse à différents types de programmeur, de nombreux *vi* cachent ces 3 étapes en les intégrant directement (c'est le cas de celui que nous avons utilisé dans l'Exercice d'application 3). De plus, LabVIEW propose plus que les fichiers binaires ou textuels afin de faciliter les opérations courantes de manipulation des fichiers :

- Fichier tableur : lecture/écriture de/vers le format .csv
- Fichier de configuration : manipulation de fichiers au format .ini (comme par exemple c:\windows\win.ini). Ces fichiers sont assez typiques pour gérer la configuration des programmes, bien qu'on commence à utiliser le format XML de plus en plus pour cela. Voici un exemple de fichier de configuration :

```
; ceci est un commentaire
[General]
; entre crochet on nomme une section
Language = fr
; les valeurs sont données sous la forme clé = valeur
BackGroundColor = red
Font = Times
Fontsize = 12
[Graph]
HistorySize = 10000
```

- Fichier de *waveforms* enregistrés au format binaire .tdms : permet de gérer simplement des fichiers de mesures. Format de fichier propriétaire LabVIEW (le format .tdm remplace dorénavant le format .tdm).

Exercice d'application 19 : rechercher des exemples, utiliser un fichier .tdms

A faire : ouvrir la fenêtre de démarrage LabVIEW et rechercher un exemple concernant les fichiers .tdms (un exemple d'écriture et un exemple de lecture). L'exécuter.

- Fichier de *waveforms* enregistrés au format textuel .lvm : permet de gérer simplement des fichiers de mesures, de façon moins efficace qu'au format .tdms car tout est converti au format texte. Ces fichiers sont cependant lisibles par un humain, ou un tableur.
- Fichier journal (datalog) : version de fichier binaire compatible avec des plateformes simples (PocketPC). Manipulation similaire à la manipulation de fichiers binaires.
- Notons aussi l'existence d'un module (***toolkit***) nommé *Entreprise Connectivity* qui permet d'utiliser des base de données (MS Access, MySql, DBase…) et d'y accéder via l'ODBC. L'utilisation est extrêmement puissante mais requière des connaissances en bases de données (notamment le langage SQL).

4.3.2.1 Utilisation de fichiers textes

Afin d'illustrer le fonctionnement d'un fichier texte, et de comprendre les chaînes de format (communes à beaucoup de langages basés sur le langage C), cette partie propose d'explorer un peu le fonctionnement de la gestion de ces fichiers.

A partir d'ici, les fonctions de recherche de *vi* et d'exemples doivent être assimilées, ce qui devrait faciliter dorénavant la présentation des exercices d'application.

Exercice d'application 20 : manipuler un fichier texte, formater des chaînes, convertir en chaîne

A faire : créer un *vi* possédant sur sa face avant les éléments suivants :
- une entrée chaîne de caractères c
- une entrée flottant f
- une entrée entier i
- une sortie chaîne de caractères c_sortie
- une sortie flottant f_sortie
- une sortie entier i_sortie

Le but est le suivant : enregistrer dans un fichier texte ces éléments (donc transformés en chaînes de caractères, avec 8 chiffres après la virgule pour le flottant) puis de lire le fichier et de les afficher.
Pour cela nous utiliserons 2 méthodes :
Méthode 1 : utilisation de chaînes de format (utilisation de « Formater dans un fichier », « Balayer un fichier »). Pour les premiers essais, ne pas mettre d'espace dans c, puis tenter en insérant des espaces. Modifier la chaîne de format pour insérer un retour chariot entre chaque donnée.
Méthode 2 : convertir les entrées, puis concaténer les chaînes obtenues afin de les enregistrer.

4.3.2.2 Utilisation de fichiers binaires

Les fichiers binaires se comportent de la même façon que les fichiers textes avec l'exception qu'ils sont typés. Cela signifie que lorsqu'on écrit des éléments, LabVIEW connaît leur type et les transforme en octets (par exemple, un tableau d'entiers I16 aura une taille de 4 octets (taille du tableau)+2 octets (taille des éléments)*nombre d'éléments).

A la lecture, il faut passer un élément du bon type à LabVIEW afin qu'il sache combien d'octets il doit lire, et comment leur redonner leur type (voir).

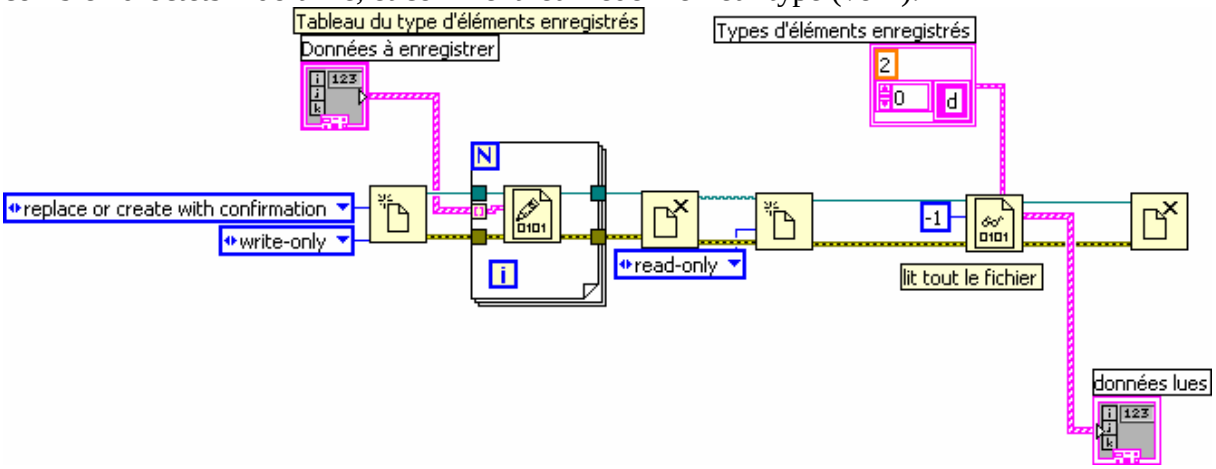


Figure 48 : utilisation de fichier binaire

4.3.2.3 Utilisation de fichiers pour gérer une configuration

Dans ce paragraphe, nous verrons non seulement comment gérer les fichiers de configuration, mais aussi comment conserver une « variable globale » en utilisant quelque chose de plus sûr qu'une variable globale. De plus, nous utiliserons un projet, et la notion de types stricts.

Nous supposons que nous travaillons sur une application qui risque de croître en terme de nombre de *vi*. Nous utiliserons donc un **projet**.

Nous supposons que nous avons une application, qui, à l'heure actuelle, possède des paramètres que nous souhaitons conserver et lire dans un fichier « .ini ». Nous prendrons comme paramètres le *cluster* composé de deux entiers et une couleur donné sur la Figure 49.

Figure 49 : cluster contenant des paramètres de configuration

Nous savons que ce *cluster* risque fortement d'évoluer au cours du développement de l'application (ajout de champs). De plus, il sera utilisé dans beaucoup de *vi* et il est inconcevable d'avoir à tous les modifier si on le modifie. Nous allons donc définir un type. Si nous sommes dans un projet, il faut pour cela créer une nouvelle commande (extension « .ctl »), comme sur la Figure 50.

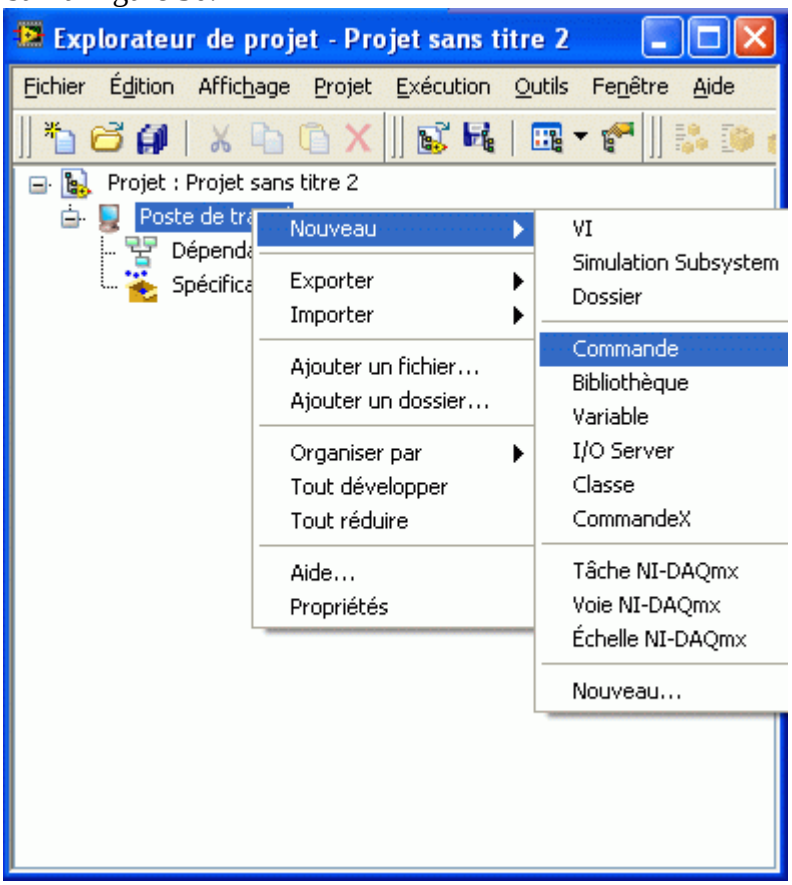


Figure 50 : création d'une nouvelle commande dans un projet

Une commande ne possède qu'une face avant, et permet de créer des commandes à l'allure personnalisée. Elle permet aussi et surtout de créer des types : lorsqu'un **type strict** est utilisé dans un *vi*, que ce soit sous forme de constante, de commande ou d'indicateur, il sera mis à jour si le type est modifié. C'est exactement ce que nous voulons. La Figure 51 montre la définition d'un tel type.

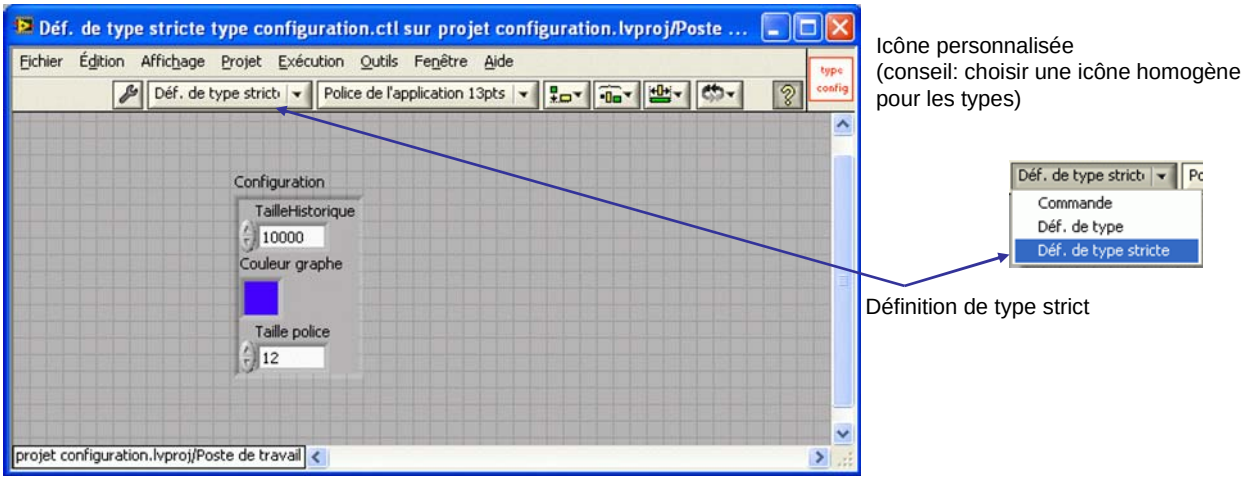


Figure 51 : définition d'un type strict

Exercice d'application 21 : créer un type strict

A faire : créer un projet, et y créer le type strict de la Figure 51.

A différents endroits du programme, nous aurons à lire la configuration, et à la modifier. Certains utiliseraient une variable globale pour y stocker la configuration. Cependant, cela pose des problèmes d'accès concurrent (si une écriture interrompt une lecture, on peut lire des données inconsistante). L'une des méthodes est d'utiliser une propriété du registre à décalage. Une « variable globale » serait conservée dans un registre à décalage. En effet, si il n'est pas initialisé, le registre à décalage contient la dernière valeur inscrite (lors d'un appel précédent). La Figure 52 représente un *vi* permettant de lire ou stocker la configuration (sans problème de concurrence puisque les *vi* sont non-réentrants par défaut). Remarquer que la boucle *While* s'arrête immédiatement après sa première instance (valeur *Faux* connectée à la condition d'arrêt). Par conséquent, elle n'est là que pour servir de support au registre à décalage. Le *vi* possède 2 entrées : l'une, « Config in », optionnelle, étant la configuration à stocker dans le registre à décalage, l'autre, « Action », obligatoire, est l'opération à effectuer : lire la configuration courante, ou bien la stocker (Ecrire). Noter qu'il pourrait y avoir plus d'opérations (initialiser, appliquer, etc.) par conséquent il est très conseillé de créer un type strict pour « Action ».

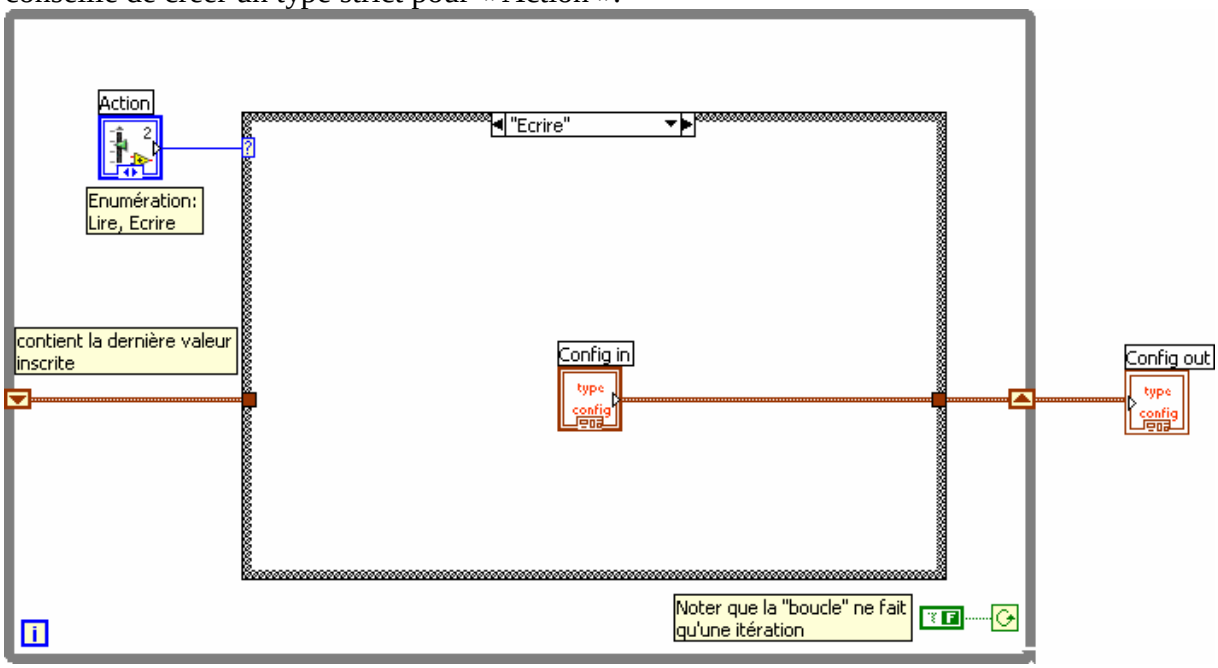


Figure 52 : utilisation du registre à décalage à la place d'une variable globale

Dans tous les cas, la valeur actuelle est envoyée sur « Config out ». Un autre avantage de cette technique est qu'en ouvrant la face avant de ce *vi* à l'exécution, on peut voir son contenu (valeur de « Config out »).

Exercice d'application 22 : créer un *vi* non réentrant de stockage de valeur globale, utiliser un type strict
A faire : créer un type strict énuméré dans le projet, contenant les valeurs lire et écrire. Utiliser les deux types stricts créés pour implémenter le *vi* de la Figure 52 qui sera aussi intégré au projet. Le transformer en sous-vi tel que « Action » est une entrée nécessaire.

La Figure 53 montre l'enregistrement de la configuration dans un fichier appli.ini se trouvant dans le même dossier que le *vi*. Remarquer l'utilisation du *vi* de stoackage.

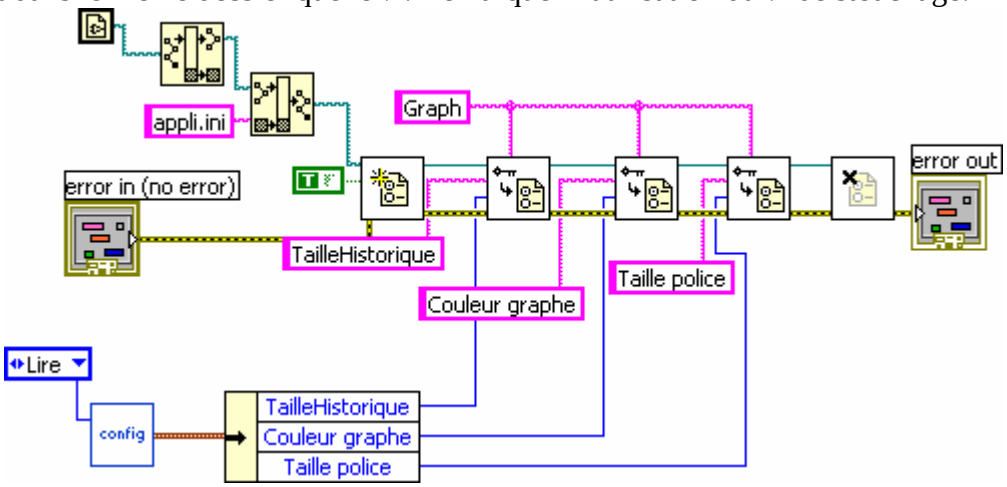


Figure 53 : enregistrement du fichier .ini

Le contenu du fichier ainsi créé est :

```
[Graph]
TailleHistorique=0
Couleur graphe=0
Taille police=0
```

Exercice d'application 23 : utiliser un *vi* non réentrant de stockage, utiliser un fichier .ini
A faire : Implémenter le *vi* donné sur la Figure 53, ainsi que le *vi* symétrique qui lit le fichier appli.ini et, s'il n'y a pas d'erreur, écrit la configuration dans le *vi* non-réentrant de stockage.

Notons que ce type de *vi* peut-être utilisé aussi pour faire communiquer deux parties parallèles d'un programme : l'un écrirait des données, à son rythme, et l'autre les lierait. C'est ce type de *vi* qui est utilisé pour faire de la communication asynchrone.

4.3.3 Interface graphique

Cette partie se focalise sur la création d'interfaces homme-machine (IHM) en LabVIEW. La première partie parle du traitement par scrutation et du traitement par événement des entrées utilisateur. La seconde partie traite de la façon dont on crée des IHM dynamiques à l'aide des **nœuds de propriété**.

4.3.3.1 Réactions aux événements

Généralement, lorsqu'on programme ses premières interfaces graphiques, on utilise de la scrutation : il s'agit, dans une boucle *While*, temporisée (par exemple toutes les 100 ou 200 ms), de lire l'état des boutons/entrées et de les traiter. Cela a pour effet de gaspiller pas mal de temps processeur. Des IHM efficaces se basent sur le traitement des événements utilisateur : quand l'utilisateur clique sur un élément d'une application, ou bien bouge sa souris, ou tape une touche, l'application qui possède le focus se voit avertie par le système d'exploitation grâce à un événement. La programmation d'IHM se basant sur les événements utilise ce principe : grâce à la structure événement, LabVIEW peut les transmettre à notre application.

Le moyen le plus simple de créer une IHM utilisant les événements est d'utiliser le modèle proposé (« Fichier » → « Nouveau… » puis choisir « Application de haut niveau utilisant les événements »). Le *vi* obtenu est alors celui de la Figure 54.

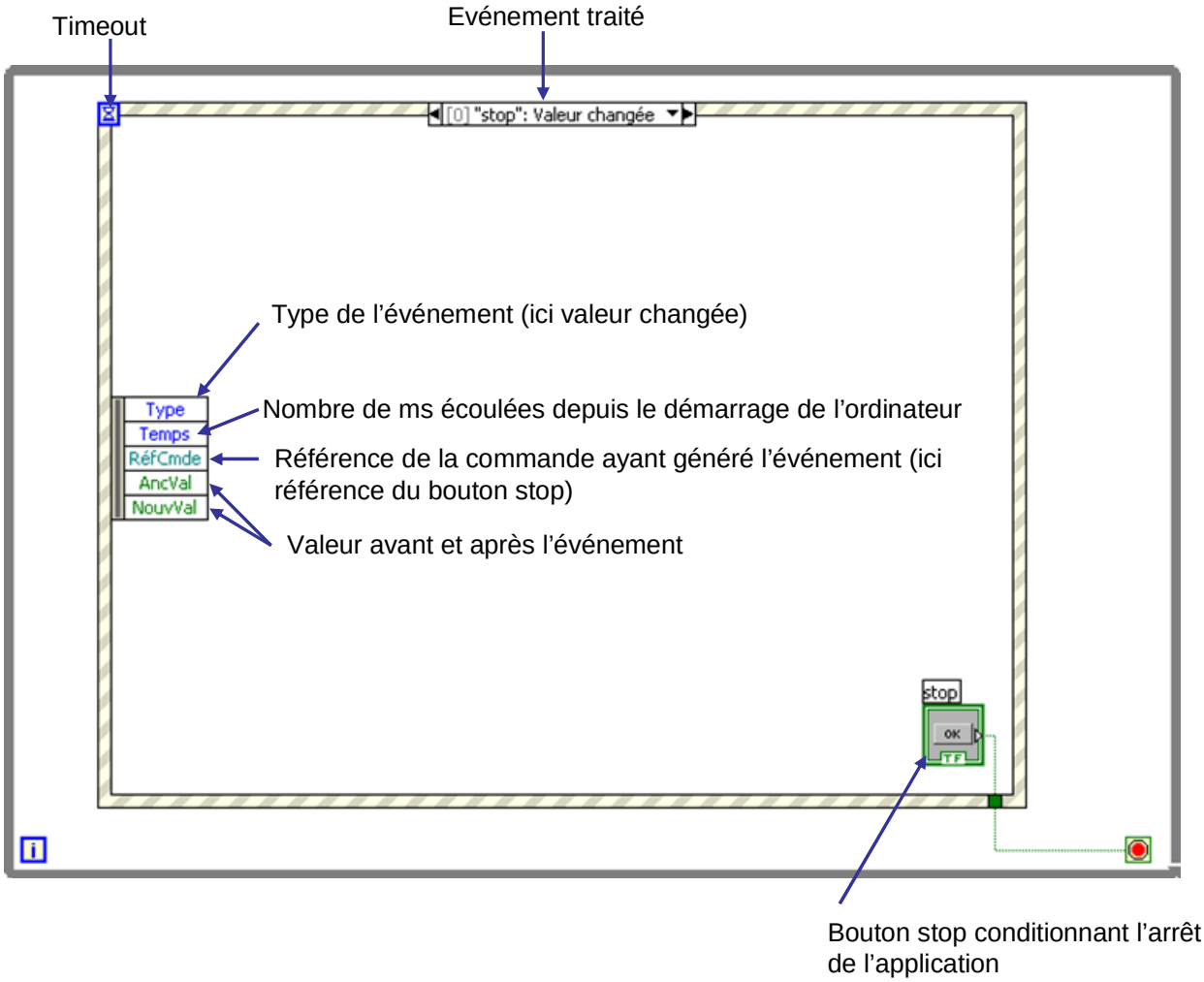


Figure 54 : application de haut niveau utilisant les événements

La boucle montrée est la boucle principale de l'application (on aura la même boucle pour une boîte de dialogue). La structure à événement attend qu'un événement pris en compte arrive, le *vi* est donc bloqué en l'absence d'événement. Si l'on souhaite faire quelque chose même en l'absence d'événement, on câblera le *Timeout* de sorte qu'au bout du nombre de millisecondes en entrée, l'événement *timeout* aura lieu.

On peut récupérer tout événement : click sur un bouton, déplacement de la souris, fermeture de fenêtre, de l'application, etc. On peut éditer et ajouter des événements à traiter à partir d'une boîte de dialogue comme celle présentée sur la Figure 55.

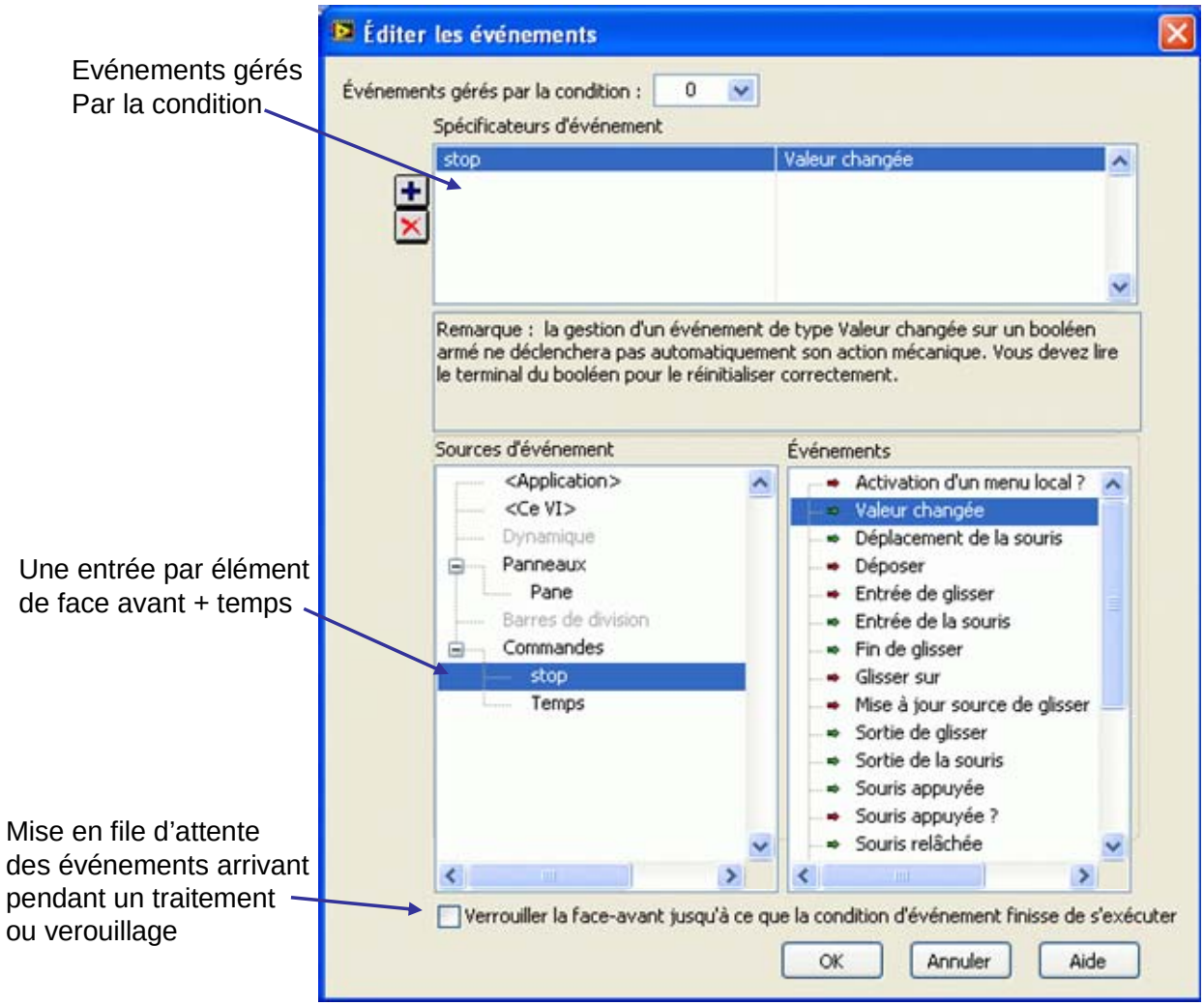


Figure 55 : édition d'un événement

4.3.3.2 IHM d'application / boîte de dialogue

Auparavant, nous avons utilisé des boîtes de dialogue standard dans l'application du jeu de hasard. Nous allons voir ici comment remplacer cela par une boîte de dialogue personnalisée.

Une boîte de dialogue (voir Figure 56) correspond à un sous-*vi* de l'application dont la propriété d'« Apparence de la fenêtre » est une boîte de dialogue (ou encore avec les propriétés personnalisées « Afficher la face-avant à l'appel » et « Fermer après l'exécution » cochée, et surtout un comportement « Modale »).

Attention, si une fenêtre est modale, elle est au-dessus de toute autre fenêtre de l'application et conserve le focus (i.e. les autres fenêtres de l'application ne reçoivent pas les événements). Par conséquent lorsqu'on lance une application comportant des fenêtres modales, il faut que celles-ci soient fermées avant lancement ou bien elles resteront modales dès le lancement.

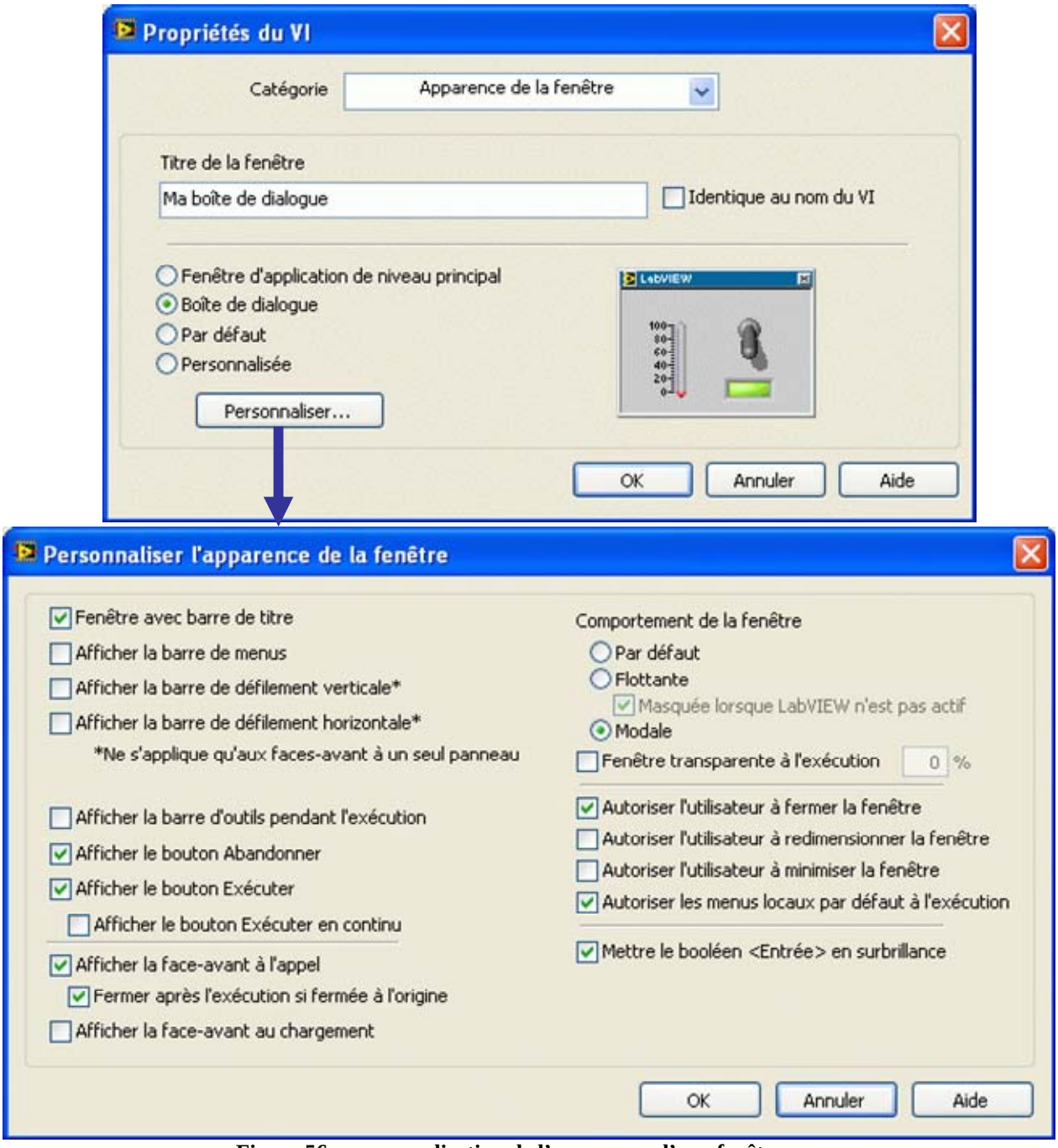


Figure 56 : personnalisation de l'apparence d'une fenêtre

Exercice d'application 24 : créer et personnaliser une boîte de dialogue, gérer une IHM par événements

A faire : placer les *vi* « Aléatoire.vi » et le programme de test que nous avions fait lors de l'Exercice d'application 18 (celui avec l'histogramme) dans le projet créé pour gérer des configurations. Ajouter dans le projet un nouveau *vi* créé à partir du modèle « Boîte de dialogue utilisant les événements », et le modifier de sorte qu'il soit similaire à la Figure 57. Le fonctionnement sous-jacent doit être pour « Recharger » de remettre la valeur de « Configuration » à la configuration courante (à l'aide du *vi* non réentrant). Lors de l'appui sur le bouton « Ok », on modifie la configuration courante, et on l'enregistre dans le fichier .ini.

Remarque : « Configuration » étant une commande, on pourra le modifier en utilisant une variable locale.

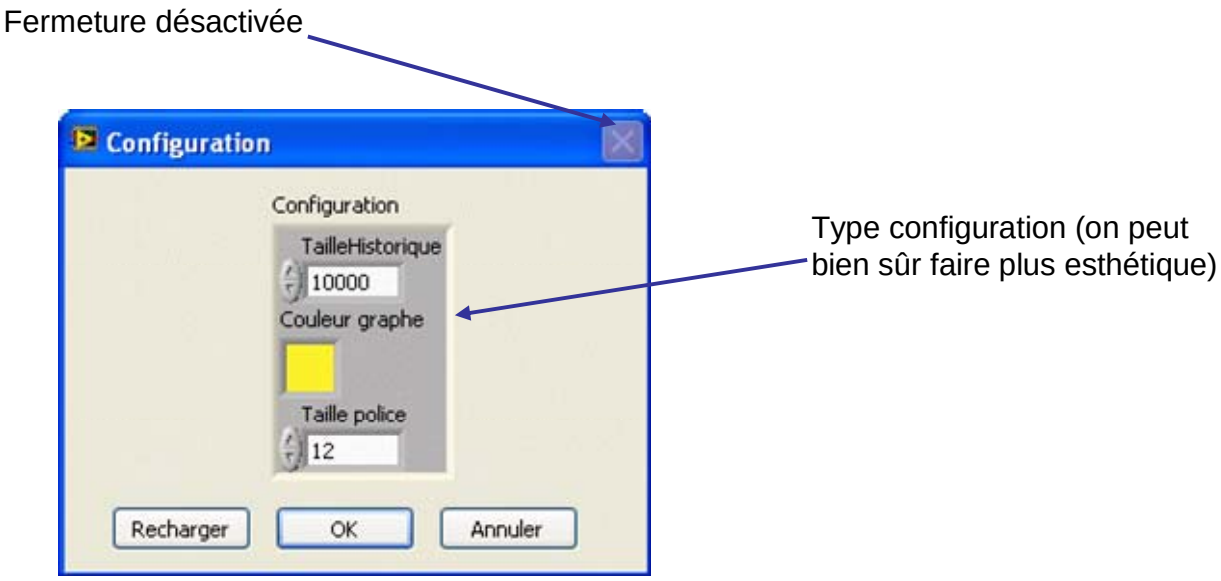


Figure 57 : face-avant de la boîte de dialogue à créer

4.3.3.3 Interface graphique dynamique

4.3.3.3.1 Nœuds de propriétés

Pour l'heure, on ne sait pas comment utiliser le fichier de configuration pour qu'il ait un impact sur l'interface graphique du *vi*. Ceci nécessite d'accéder par programme aux propriétés que nous avons jusqu'à présent modifiées à la main. On peut y accéder en créant des **nœuds de propriétés**. Pour ce faire, on fait un click droit dans le diagramme sur le terminal, on choisit « Créer » → « Nœud de propriété » → la propriété qui nous intéresse. Il n'est pas aisé au début de trouver la bonne propriété et le débutant tâtonnera souvent à sa recherche.

Par exemple, pour rendre visible ou invisible un élément de face-avant, on utilisera la propriété « Visible » 

. On peut voir sur cette icône que la flèche va vers la droite, montrant qu'on accède à la propriété en lecture. Si on veut la modifier, il nous faut faire un click droit sur la propriété et la changer en écriture 

. Il nous suffit alors d'envoyer la valeur voulue pendant le fonctionnement du programme. Un click gauche sur la propriété permet d'en sélectionner une autre, de plus le *vi* est redimensionnable (permettant d'accéder à plusieurs propriétés à la fois).

Exercice d'application 25 : créer une IHM dynamique à l'aide des nœuds de propriétés, utiliser des onglets
A faire : dans le projet utilisé dans l'exercice précédent, se baser sur le *vi* de l'Exercice d'application 18 (histogramme) afin de créer une application basée sur les événements dont la face avant est donnée sur la Figure 58. Noter qu'elle possède un titre personnalisé, pas de barre d'outils (mais conserve un menu déroulant pour pouvoir l'arrêter en cas de « plantage »). Appuyer sur le bouton « Calculer » lance un test du générateur aléatoire, et l'appui sur le bouton « Options » ouvre la *popup* (fenêtre modale) de configuration faite lors de l'exercice précédent. Attention, si le bouton « Ok » est appuyé pour fermer cette fenêtre, on utilise les nœuds de propriété pour changer la couleur de la courbe et la taille de la police de l'étiquette de nom de l'axe des X et des Y en fonction de la configuration (on ignorera le paramètre « TailleHistorique ». La même chose (appliquer les options lues dans le fichier .ini) doit être faite dès le lancement de l'application.

Ajouter ensuite des onglets : le premier panneau affiche le contenu de la face-avant actuelle, alors que le second affiche des éléments statiques « A propos de… » possédant des informations sur le programme et son auteur.

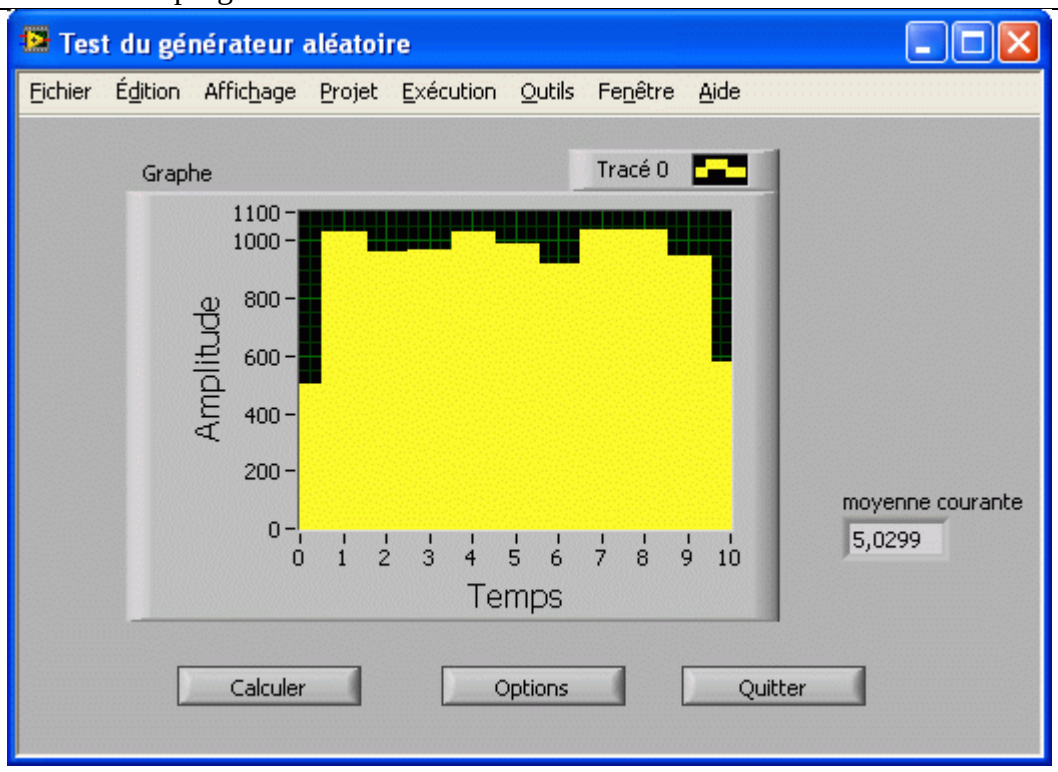


Figure 58 : face-avant de l'application

Le problème est que très rapidement, lorsque beaucoup d'éléments sont configurables, le *vi* possédant l'IHM de l'application devient très volumineux et presque inextricable. Nous allons donc voir 2 techniques permettant de passer par des sous-*vi* : la première consiste à utiliser un conteneur de face-avant secondaire, la seconde consiste à utiliser des références sur les terminaux.

4.3.3.3.2 Face-avant secondaire

(voir Figure 59). Ce qui peut être déroutant est l'utilisation d'une référence de *vi* : il s'agit d'une référence que l'on peut créer dynamiquement en ouvrant une référence sur un *vi* (par exemple en passant son nom), ou bien de façon constante comme sur la figure. Dans ce cas, on crée une référence de *vi* statique sur laquelle on fait un glisser-déposer du *vi* dont on veut afficher la face-avant. Cependant, cela reste assez complexe à manipuler, et souffre d'inconvénients (notamment le fait qu'il faut penser à exécuter le *vi*, ce qui n'est pas le cas sur la Figure 59).

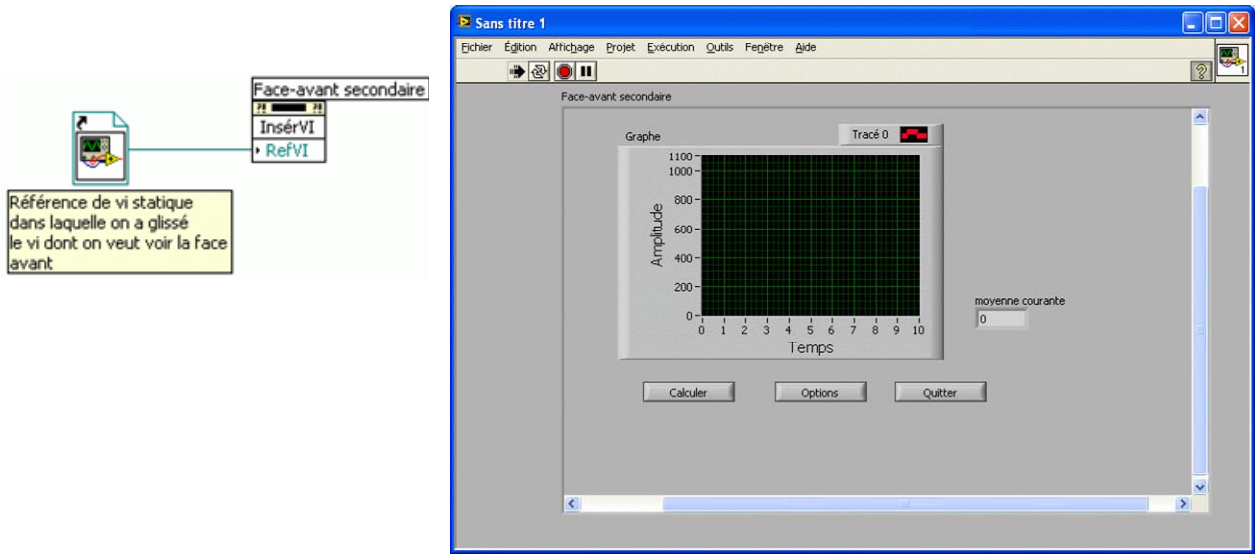


Figure 59 : utilisation d'une face-avant secondaire

On est donc amené à utiliser un nœud de méthode prenant en entrée la référence du *vi* dont on affiche la face-avant, afin de l'exécuter comme cela est montré sur la Figure 60. Noter sur la face-avant l'absence des barres de défilement (elles étaient présentes sur la Figure 59, témoignant du fait que la face-avant secondaire était visible mais non exécutée). On peut remarquer comment on exécute ce *vi* : le paramètre important est « AttendreLaFin » qui précise si le *vi* doit être exécuté de façon synchrone (le *vi* appelant doit attendre la fin) ou asynchrone.

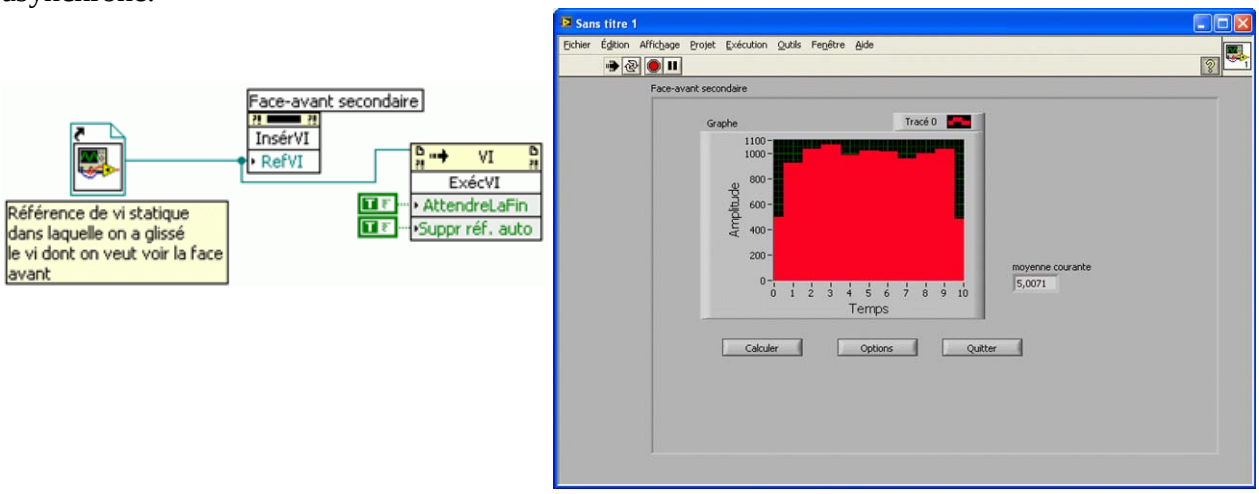


Figure 60 : utilisation correcte d'une face avant secondaire en appel synchrone

4.3.3.3.3 Utilisation de références sur les terminaux

Les références permettent d'utiliser des nœuds de propriété et donc agir dynamiquement sur une face-avant à partir d'autres *vi* auxquels les références seraient passées. Une référence (comme Graphe) est obtenue en faisant un click droit sur un terminal comme sur la Figure 61. Le but étant de créer un sous-*vi* prenant cette référence en paramètre, il nous faut créer une commande à partir de cette référence (qui pour l'instant est une constante). Il y a deux façons de procéder : faire un click droit sur la référence, pour créer une commande, qui pourra être copiée/collée sur la face-avant du sous-*vi*. La seconde façon consiste tout simplement à faire un glisser-déposer de la référence sur la face-avant du sous-*vi*. La commande obtenue ressemble à 

pour un graphe.

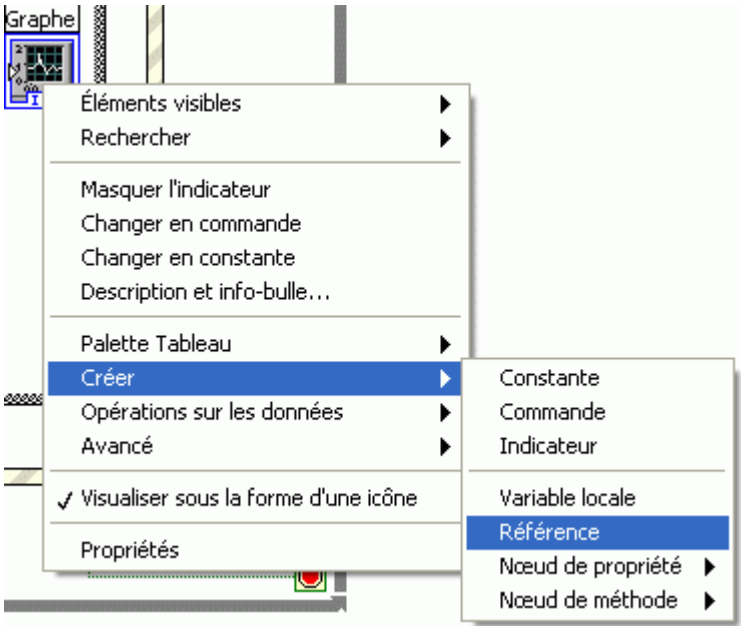


Figure 61 : créer une référence

Il suffit ensuite d'utiliser des nœuds de propriétés acceptant des références en entrée (palette « Contrôle d'applications »), c'est-à-dire des nœuds qui ne sont attachés à aucun élément de face-avant. La modification des propriétés du graphe peut donc être effectuée dans un sous-*vi* comme sur la Figure 62.

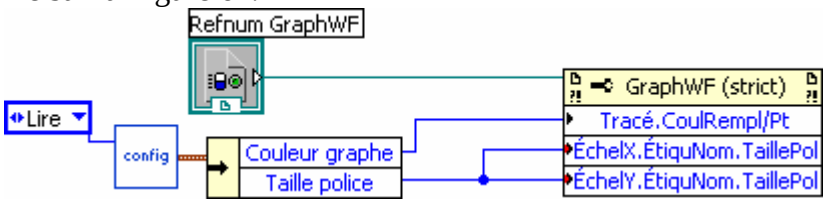


Figure 62 : utilisation d'une référence

Exercice d'application 26 : utiliser les références, modifier un type strict

A faire : modifier le programme de l'Exercice d'application 25 afin que la modification des propriétés du graphe s'effectue dans un sous-*vi* tel que celui de la Figure 62.

Modifier le type « Configuration » afin d'ajouter un paramètre NombreItérations (inutile de modifier le chargement/enregistrement dans les fichiers de configuration). Ce paramètre sera utilisé comme entrée N de la boucle For.

4.3.4 Création de fichier exécutable

LabVIEW professionnel peut créer des fichiers exécutables. Ils s'exécuteront sur une machine qui n'a besoin que du *LabVIEW runtime* qui est gratuit. Le fonctionnement en est simple, à partir de la fenêtre projet, on peut faire un click droit sur « Spécification de construction » (voir Figure 63).

Figure 63 : création d'exécutable

L'étape importante se trouve dans l'onglet « Fichiers sources », dans laquelle on va définir le *vi* de démarrage, c'est-à-dire celui qui serait lancé pour lancer l'application.

Exercice d'application 27 : créer un exécutable

A faire : créer l'exécutable de l'application.

Noter que l'accès au fichier de configuration ne fonctionne pas et que l'application ne se ferme pas à la fin de l'exécution.
Pour que l'application se ferme, il faut utiliser le « Quitter LabVIEW » à la fin de l'application.
Pour que le fichier puisse être accédé, il faut savoir une chose : lorsque des *vi* sont dans un exécutable, tout se passe comme s'ils étaient dans un sous-dossier au nom de l'exécutable. Par conséquent, il faut remonter d'un cran de plus dans l'arborescence pour le nom de fichier.
Problème : l'exécutable et la version LabVIEW ne sont pas compatibles, car il serait plus qu'ennuyeux de voir LabVIEW se fermer à chaque fois que l'on ferme l'application, et le fichier de configuration « Appli.ini » ne serait pas dans le même dossier. Pour gérer les 2 cas de façon homogène, on est donc amenés à tester dans le programme s'il s'exécute dans LabVIEW ou bien dans un exécutable, et de calculer les chemins vers les fichiers ou de quitter LabVIEW à la fin ou non en fonction de cela. Un moyen simple de savoir si on se trouve dans l'exécutable est de tester l'extension du dossier parent d'un *vi* (voir Figure 64) : si c'est « .exe » alors on se trouve dans l'exécutable, sinon, on est dans l'environnement LabVIEW.

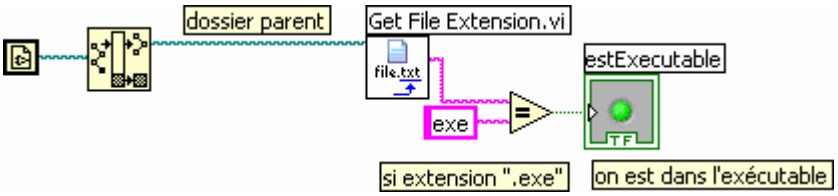


Figure 64 : savoir si l'on est ou non dans un exécutable

4.3.5 Quelques petits trucs

Quelques petits trucs :

- Un double click sur un élément de face-avant amène au terminal du diagramme directement et vice-versa.
- On peut demande à LabVIEW d'« Arranger le câblage » en faisant un click droit sur le câble.
- On peut définir les valeurs par défaut des éléments d'une face avant (click droit sur l'élément, puis « Opérations sur les données » → « Désigner la valeur actuelle par défaut ».
- Lorsqu'on a un programme important, les outils du menu déroulant « Outils » → « Profil » → « Performances et mémoire » permettront de voir sur quel *vi* focaliser ses efforts en terme d'optimisation du temps de traitement ou de la mémoire utilisée (ainsi, très souvent les temps de traitement longs sont liés à l'utilisation de la concaténation de tableaux dans des boucles, avantageusement remplacées par une création initiale d'un tableau de taille suffisante, puis « Insérer dans un tableau » à la place de concaténer).

5 Conclusion

J'espère que ces explications et exercices auront aidé le lecteur, débutant, ou intermédiaire, à progresser dans l'utilisation du langage LabVIEW. De nombreux domaines restent à explorer : utilisation du réseau, optimisations, intégration dans le système (ActiveX,…), création et utilisation de DLL, création de bibliothèques complètes, utilisation des classes, gestion fine du multitâche, gestion des propriétés d'exécution des *vi*, cibles spécifiques, gestion des sources et versions, etc. mais j'espère que l'accent porté sur l'utilisation de la documentation et des recherches d'exemples pousseront les curieux à trouver relativement simplement ce dont ils auront besoin à l'avenir.

Il y a malheureusement relativement peu d'ouvrages en français traitant de LabVIEW. J'encourage le lecteur à se procurer le livre de Francis Cottet « LabVIEW : Programmation et applications » chez Dunod. Il a été écrit à la sortie de la version 6 de LabVIEW, et manque les fonctionnalités introduites aux versions 7 et 8, mais il reste un ouvrage de référence pour les bases de programmation en LabVIEW. Nous travaillons actuellement à la rédaction de la nouvelle version.

De plus, le lecteur intéressé par les aspects programmation multitâche en LabVIEW pourra se procurer le livre de Francis Cottet et moi-même « Systèmes temps réel de contrôle-commande : Conception et Implémentation », chez Dunod. Il est à ma connaissance le seul ouvrage parlant de la façon dont on peut concevoir et programmer un système multitâche à but temps réel en LabVIEW (il traite parallèlement LabVIEW, C et Ada).