POUR LE PIRE ET POUR LE MEILLEUR…

Jean-Michel CHALONS

SAPHIR
Rue du mail
38530 BARRAUX, France
jmc@saphir.fr

1 LABVIEW PRO OU LABVIEW PROTO ?

Un vieux débat et beaucoup de stériles ébats ; LabVIEW est-il :

- la porte ouverte à l'informatique pour tous ?
- un environnement pour professionnel du développement logiciel ?

J'imagine qu'en tant que produit, il se donne d'abord comme vocation de s'imposer dans son domaine et pour ce faire, il ne puisse éviter les deux axes, apparemment contradictoires, en réalité complémentaires :

1 : Pour maintenir un volume propre à justifier son existence, c'est-à-dire à assurer son financement, LabVIEW doit préserver et cultiver son aspect « easy to use », en restant accessible au plus grand nombre des techniciens de tout poil, sauf informaticien qui peut y percevoir plus de menace que d'intérêt ; et c'est à présent LabVIEW 7 Express, le logiciel de paillasse, la démystification et l'indépendance, tout simplement efficace pour les petits besoins du plus grand nombre ! LabVIEW pour le proto.

2 : Pour prétendre à ses lettres de noblesse, LabVIEW s'efforce de s'adresser également à l'ingénierie logicielle, avec un réel succès témoigné par une infiltration croissante au sein de grands projets qui confirment reconnaissance et crédibilité… et c'est LabVIEW pour les pros !

Pourtant la même boîte à outils proposée à tous, offrant autant de facilité au bricoleur que de précision à l'artisan. Attention à ce que l'un n'effraie pas l'autre.

1.1 Pas faux mais injuste

Combien de fois ai-je entendu, vous-même avez-vous peut-être entendu, de critiques, de remontrances, pas vraiment dénuées de bon sens, mais exprimée avec tellement de malveillante partialité…

"C'est un vrai paquet de nouilles !"

Oui le « G programming » peut conduire à la complexité inextricable d'un « code spaghetti »

Non, la faute n'en revient pas à l'outil, mais bien à celui qui le met en oeuvre !

Et même pourrai-je défendre : LabVIEW, intransigeant avec lui-même, dénonce explicitement ce possible travers, tant il est visuellement évident qu'un mauvais code est mauvais ! alors qu'en équivalent de codage textuel, le même niveau de maladresse, le même défaut d'architecture passera volontiers inaperçu si on ne s'attarde pas à l'observer attentivement.

Ainsi, le reproche devrait-il se retourner en compliment :

On peut mal faire, mais cela se voit, alors il convient de s'obliger à faire mieux !

1.2 Ingratitude

J'ai aussi bien entendu :

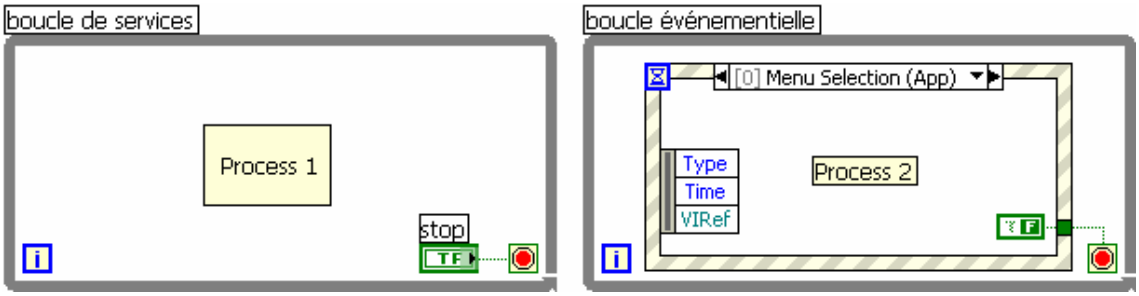
"LabVIEW ? Ah oui ! je connais cet outil de prototypage d'IHM !"

C'est passablement réducteur.

Oui, on peut rapidement préfigurer les dialogues, messages et autres interfaces utilisateurs ; oui, un jeu de « panels » réalistes constitue un précieux support pour la spécification détaillée par le demandeur, de même que pour la conception proposée par l'équipe développement ; oui, ces prototypes d'IHM facilitent la compréhension réciproque des intéressés, confirmant qu'un bon schéma vaut mieux qu'un long discours…

Mais non, il n'est pas nécessaire de passer ensuite au langage textuel pour animer les objets visuels ; c'est juste une question de choix des armes. Et le « G programming » est particulièrement efficace pour gérer des processus parallèles, besoin de base courant s'il en est : acquisition continue avec traitement à la volée, interactivité Opérateur, dialogue simultané au déroulement de tâche de fond, dispositifs serveurs-clients et autres asynchronismes…

Quoi de plus intuitif en effet que de dessiner deux structures côte à côte pour partager les ressources d'un processeur à deux tâches simultanées ?



C'est à vrai dire, exactement cette facilité-là qui en 1988 a laissé LabVIEW 1.2 me convaincre de ses vertus, alors qu'on devait déployer des efforts diaboliques en Fortran, puis en Pascal, Basic… pour obtenir des effets similaires aux médiocres performances de réactivité sur des stations de travail autrement plus coûteuses que les « ordinateurs personnels » émergeants.

Cela étant, et après plus de 15 ans d'évolution, il reste que plus on vise une IHM interactive confortable, plus l'architecture du code doit mettre en œuvre des notions pas réellement complexes mais non triviales, plus le code peut prendre du volume et nécessite rigueur et précision… Cette inévitable implication insuffle quelquefois de l'air au moulin à paroles des détracteurs ; ainsi à vouloir trop bien faire, le développeur logiciel peut, pour finaliser une interface irréprochable, se retrouver vite confronté à une charge supérieure à celle évaluée initialement pour simplement traiter le problème posé.

C'est en ce sens qu'il convient de rester vigilant à proportionner l'effort de qualité d'IHM au véritable enjeu, sans se laisser entraîner par un perfectionnisme excessif et d'autant plus ingrat qu'il sera difficile à valoriser.

En effet la rigueur à un prix : le temps qu'il convient d'y consacrer !

1.3 Respect mutuel

Si l'on oppose l'amateur au professionnel, je tiens personnellement à défendre cette notion de respect :

- que le spécialiste respecte l'amateur, lequel peut et doit prétendre à une autonomie suffisante pour couvrir un « premier niveau de besoins » ; cet amateur deviendra ainsi un excellent interlocuteur sachant parler du même langage et capable d'apprécier les évaluations de charge de celui qui en fait un métier ;
- que l'amateur respecte l'approche méthodologique du professionnel et en apprécie les apports ; qu'il sache en reconnaître la charge de travail et la valeur ;

Une raisonnable modestie devra limiter l'ardeur de l'amateur à chausser trop vite les bottes du spécialiste ; à l'inverse ce dernier trouvera couramment avantage à emprunter impasses et raccourcis sans craindre ni renier d'agir en amateur.

Il importe d'adapter l'effort et les moyens à l'enjeu, de rester capable de choisir consciemment la casquette du bon bricoleur pour parvenir droit au but chaque fois que la modicité de l'enjeu ne justifie pas la mise en œuvre de toutes les bonnes résolutions professionnelles. C'est typiquement le cas des « bouts de code » pour test, pour lever de doute, pour comparaison d'algorithmes, pour évaluation de méthode, pour une application à courte durée de vie, bref pour tout développement assimilable à un prototypage. Attention alors à bien faire distinguer par les tiers utilisateurs, les limites du résultat si économiquement obtenu, de ce que devrait être une application finalisée… à bien évaluer l'effort résiduel pour passer de l'un à l'autre.

1.4 Nominal et singulier

J'appelle « le nominal » ce que doit faire le logiciel, « le singulier » ce qu'il doit ne pas faire, autrement dit « se planter ». C'est avec un humour assez réaliste qu'on cite couramment la loi du 80/20 : "Il faut 80% du temps pour développer 80% de l'application, et encore 80% pour les 20% restant…"

Faites le compte ! Ma perception de cette triste réalité du métier, au demeurant pas spécifique à LabVIEW, est que l'évaluation de charge est essentiellement focalisée sur ce que doit faire le logiciel, en conditions nominales d'exploitation ; or le cycle effectif d'exploitation ne manquera pas d'être jonché de conditions anormales face auxquelles le logiciel montrera sa robustesse ou sa fragilité, selon le niveau de protection contre les singularités que l'analyse et la conception lui auront concédé, à charge d'un développement plus ou moins rigoureux.

Or la rigueur a un prix : et bla-bla… (on avait compris !)

1.5 Saint Axe, priez pour eux

Il apparaît nettement que la mémoire visuelle renforce considérablement l'apprentissage de la technique de programmation, sans s'encombrer de fastidieuses syntaxes. En témoignent les enfants qui acquièrent rapidement une étonnante efficacité, ce qui ne fait pas pour autant de LabVIEW un jouet, mais un outil puissant et accessoirement ludique.

Un universitaire de mes amis, initialement récalcitrant à LabVIEW et finalement bon adepte, me citait l'effet « cliquet » en commentant ainsi :

"Quand je me remets après quelques mois dans un code textuel, je dois non seulement y reprendre mes marques, mais aussi réviser la syntaxe propre au langage ; avec LabVIEW, je retrouve les objets à leur place logique, je remémore immédiatement ce que j'ai fait et ce que j'ai appris précédemment et j'en apprends un peu plus à chaque fois sans m'embarrasser de détails syntaxiques !"

1.6 C’est en forgeant

…qu’on devient forgeron !

« Easy to use », « ready to go », mon œil ! il y a du travail, et c’est un vrai métier qui fera la différence entre les véritables facilités offertes au plus grand nombre pour réaliser de modestes ouvrages d’une part et d’autre part la rigueur nécessaire pour assurer fiabilité, robustesse et évolutivité aux projets plus ambitieux.

Malgré la pertinence indéniable des nombreux « wizards » qui l’enrichissent, LabVIEW n’est pas « magique », mais tout simplement ramené au rang des excellents outils de développement.

2 ILLUSTRATION PAR L’EXEMPLE

Définissons un objectif simple correspondant à un besoin courant :

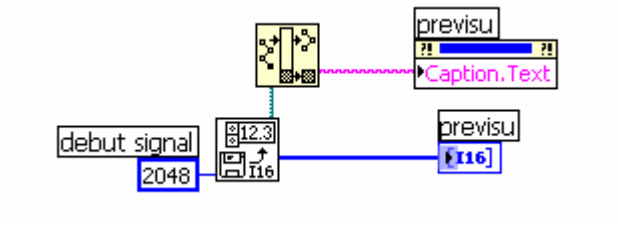
« visualiser un signal échantillonné »

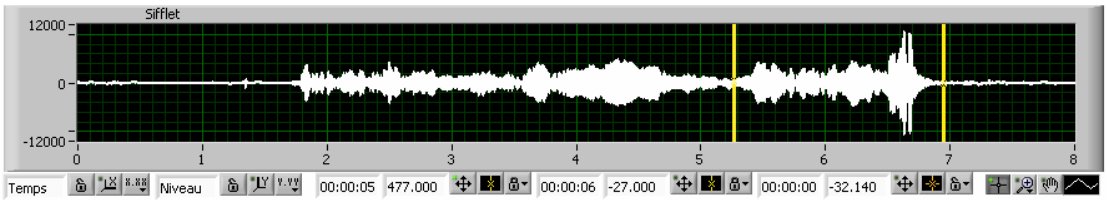
Ce signal pourra arriver en continu d’une source physique pour contrôle à la volée ou avoir été enregistré en fichier numérique ; on souhaite une vue totale de l’ensemble du signal compressé dans un graphe, afin de zoomer ensuite sur des zones d’intérêt… Pour simplifier, on considérera un fichier contenant une seule voie de signal selon un format binaire décrit par un en-tête de 2048 octets qu’on pourra ignorer en première approche. Le signal est borné par la taille du fichier et détermine la fenêtre de visualisation ; dans l’hypothèse d’un signal continu, on déterminerait a priori la durée de visualisation défilante…

Notons que dans les deux cas, le produit de la fréquence d’échantillonnage par la durée de fenêtre à visualiser n’est pas explicitement bornée ; oublions un instant.

2.1 la magie du prototype ;-)

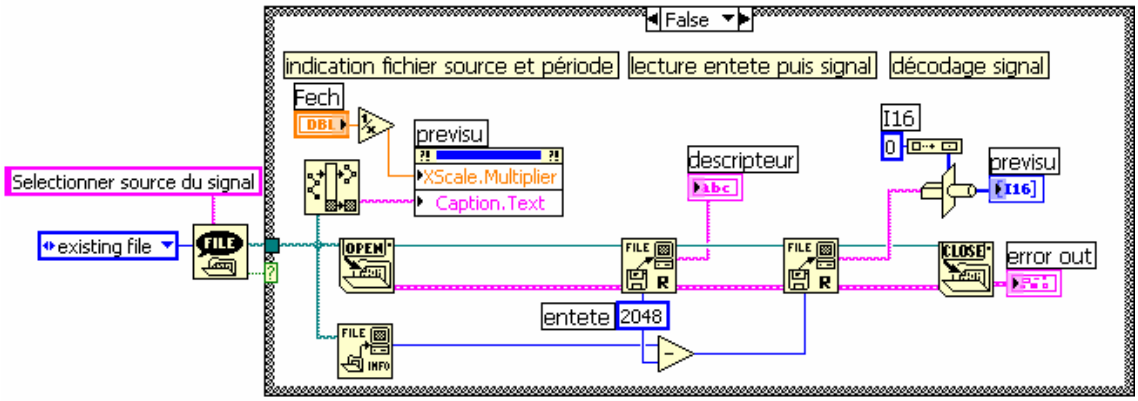
Facile :



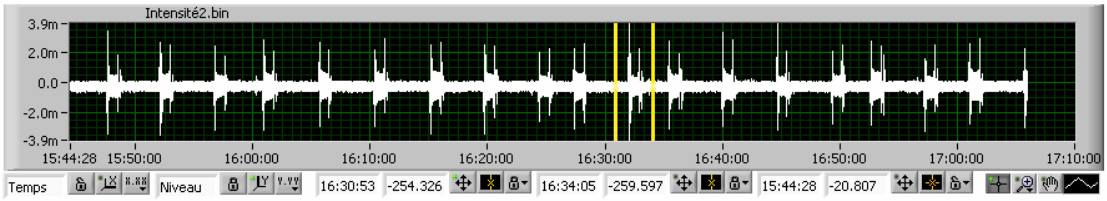


Et voici en un temps trois mouvements, la vue d’un son échantillonné à 11kHz durant 8 secondes, soit près de 90k points résumés dans un graphe qui n’offre pas plus de 1000 pixels en largeur.

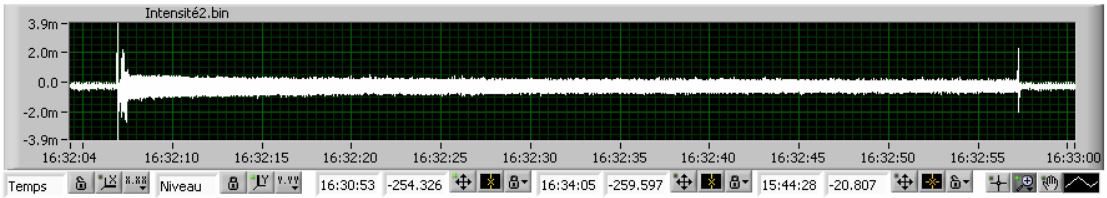
Le développeur averti n’hésitera pas à consacrer trois minutes à user des fonctions bas niveau pour obtenir le même résultat un peu enrichi avec un code apparemment plus volumineux…



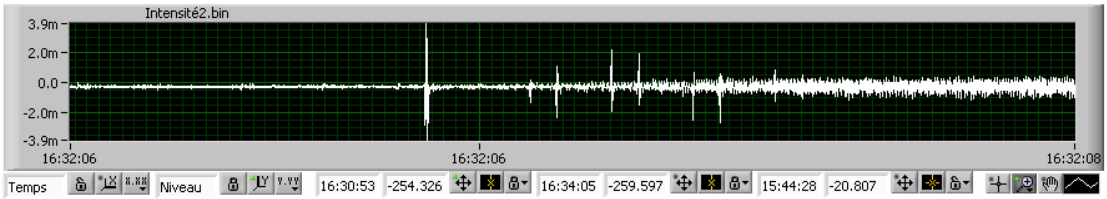
…en réalité très allégé, le détail du précédant étant masqué derrière l'icône 32x32 d'un bien commode sous-programme complexe qui traite de nombreuses options.



On observe de même le contenu de divers enregistrements ; ici 1H¼ de mesure à 3kHz soit 13.5M échantillons ! issus d'un fichier de 27Mb.



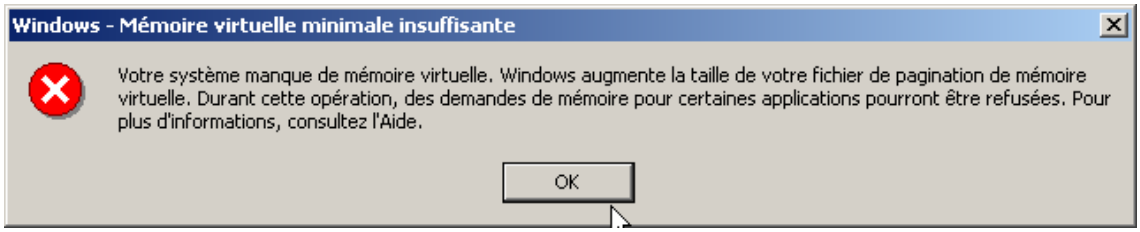
L'objet graphe offre toutes facilités de Zoom, ci dessus sur environ 1 minute de signal, et ci-dessous le détail de 2 secondes, soit 6000 échantillons



Il apparaît extrêmement commode d'observer de telles quantités de données avec aussi peu d'effort ; cependant, on commence à subir de nets défauts de réactivité de l'interface…

2.2 pas de miracle ;-(

Et si le fichier sélectionné dépasse une taille "raisonnable", il faut s'attendre à de forts désagréments, dans le meilleur des cas justifiés par un message clair :



La taille critique dépend bien sûr de la configuration du système, en particulier de la mémoire disponible ; pour un besoin borné, on peut toujours repousser la limite par l'ajout d'une barrette mémoire mais là n'est pas la solution puisque l'utilisateur ne manquera pas d'outrepasser la nouvelle borne.

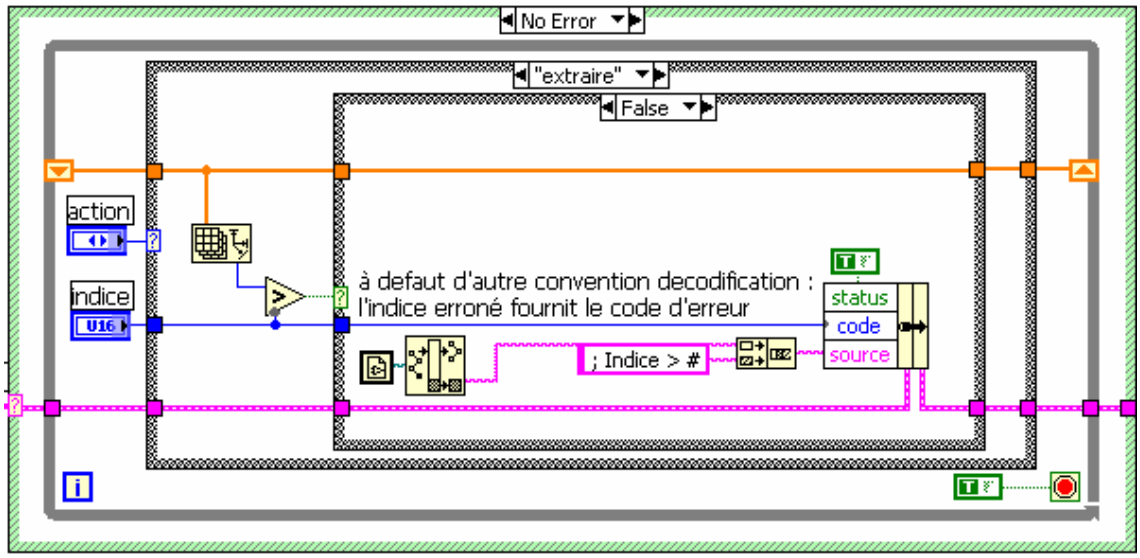
2.3 un petit mot sur le Test

Il convient d'y songer dès la conception ; et l'exemple précédant montre que le prototype peut tromper et doit, avec un minimum d'inspiration, aider à borner le champ d'exploitation, en cherchant systématiquement les conditions extrêmes.

S'il fallait définir trois points fondamentaux relatifs au test du logiciel développé, je serai tenté de proposer :

- premièrement : tester
- deuxièmement : tester
- et troisièmement : faire tester

Premièrement commencer par la fin, c'est-à-dire tester les conditions d'exploitation par l'embarquement au sein du code lui-même d'une stratégie de détection et report d'erreur, thème qui mériterait un livre à lui seul ; pour simple illustration, la réponse à une requête de lecture d'une cellule de tableau sera précédée d'un contrôle de l'indice demandé ; un indice supérieur à la taille reportera un code d'erreur plutôt qu'une cellule erronée…



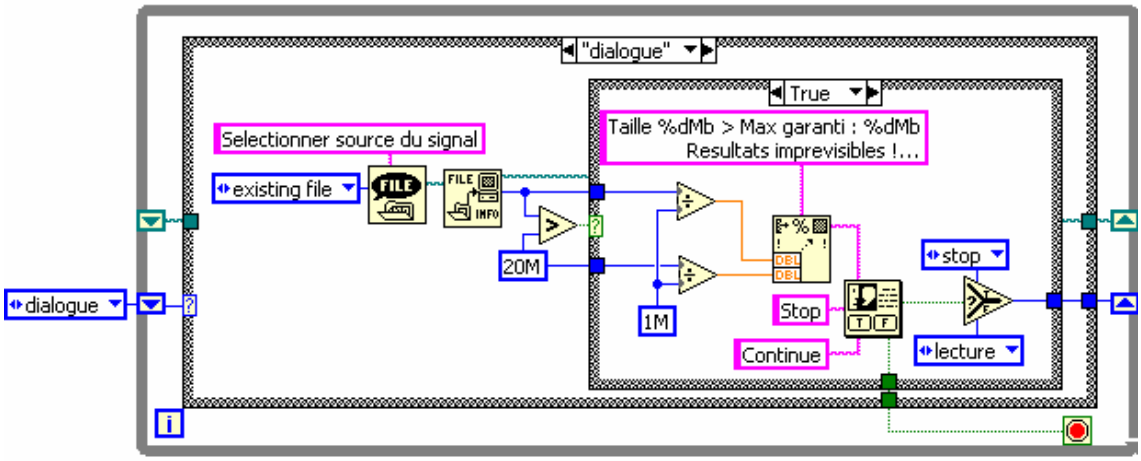
Deuxièmement, appliquer à chaque composant une campagne de test unitaire ; mieux qu'une procédure systématique et fastidieuse, un peu de bon sens et un soupçon d'humilité ne seront pas de trop, et un second livre pourrait enrichir la bibliothèque…

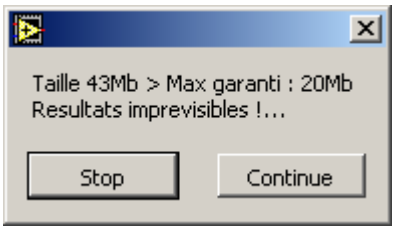
Troisièmement, la véritable mise à l'épreuve se fera au travers de l'utilisation par des tiers, mais d'abord selon une convention de béta-test plutôt que par une mise en exploitation prématurée ; les tiers seront des collègues développeurs, internes ou externes, pour l'intégration de composants logiciels ou des exploitants pour une application finale, avec les réserves de précautions qui s'imposent et une confirmation d'humilité…

2.4 quel champ d'application ?

Ce discours semblera sans doute évident, peut-être même déplacé ! mais combien est-il étonnant de constater la fréquence de ce genre de lacune ? et ce qui va sans dire va tellement mieux en le rappelant…

Si on souhaite proposer un module logiciel réutilisable à partir de ce prototype, il conviendra au minimum de lui demander d'avertir l'utilisateur que sa demande est peut-être excessive :





Au-delà, si l'objectif est de permettre une prévisualisation compressée d'un enregistrement non borné, une stratégie logicielle adaptée devra résulter d'une véritable analyse conceptuelle.

3 UN VISUALISEUR A GEOMETRIE VARIABLE

Les exigences du module à concevoir sont alors :

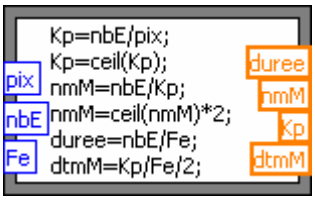
- ne pas limiter la durée du signal (i.e. la taille du fichier) à observer au sein d'un graphe
- être exploitable sur toute configuration informatique, sans requérir d'ajout de RAM
- permettre le zoom sans perte d'information jusqu'au facteur 100

Pour ce faire, une première réflexion conduit à ne pas charger en mémoire l'intégralité des données, mais un échantillonnage de celles-ci, décimées d'un facteur égal au rapport du nombre d'échantillons de la source à celui de la cible, soit au moins 100 fois le nombre de pixels du champ d'affichage du graphe. Afin que la décimation conserve la vue de tout évènement du signal, aussi bref soit-il, on retiendra de chaque bloc de taille égale au facteur de décimation, non pas une valeur représentative, mais le couple des valeurs minimale et maximale de ce bloc ; c'est en fait très proche de la technique pratiquée par l'objet graphe lui-même, à ceci près qu'il traite l'intégralité de ce que la source lui fournit, tandis qu'on s'attache à réduire la quantité de données circulant en mémoire par une technique amont de lecture par blocs séquentiels. Pour un graphe affichant typiquement 1000 pixels (à rapprocher d'une résolution courante d'écran) le nombre d'échantillons retenu sera ainsi limité à 200k ; l'expérience montre qu'une telle taille est parfaitement supportée et constitue un compromis confortable sur un calculateur courant, tandis que 10 ou 20M explosent ma modeste machine.

La réduction sera déterminée par une première fonction qui peut être :

où les entrées sont :

- pix : # pixels dédiés à l'affichage
- nbE : # échantillons déduits de la taille du fichier
- Fe : Fréquence d'échantillonnage issue du descripteur

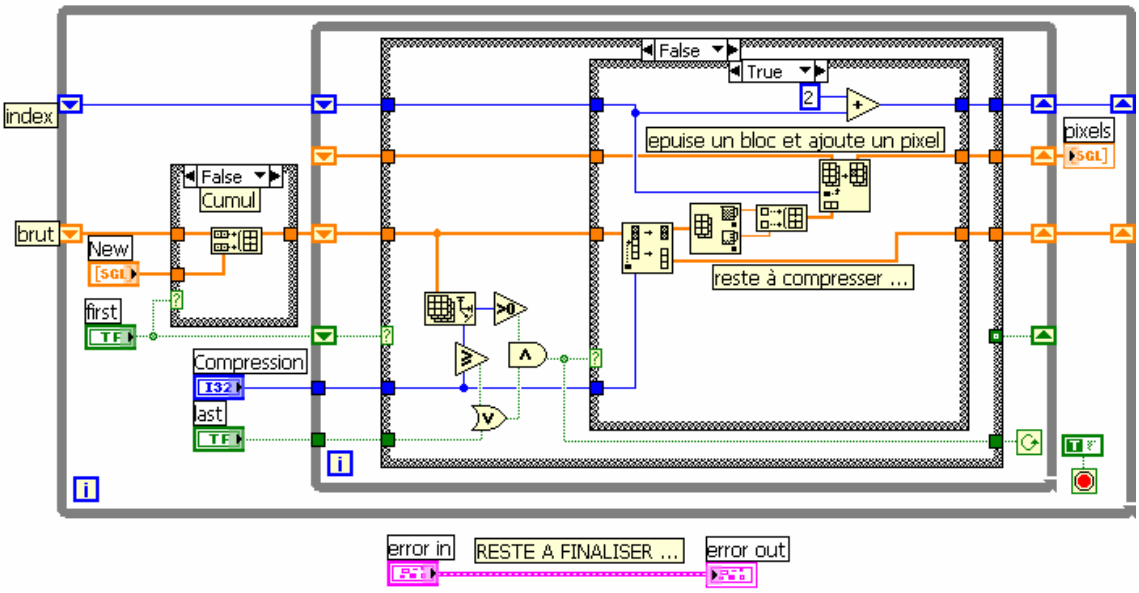


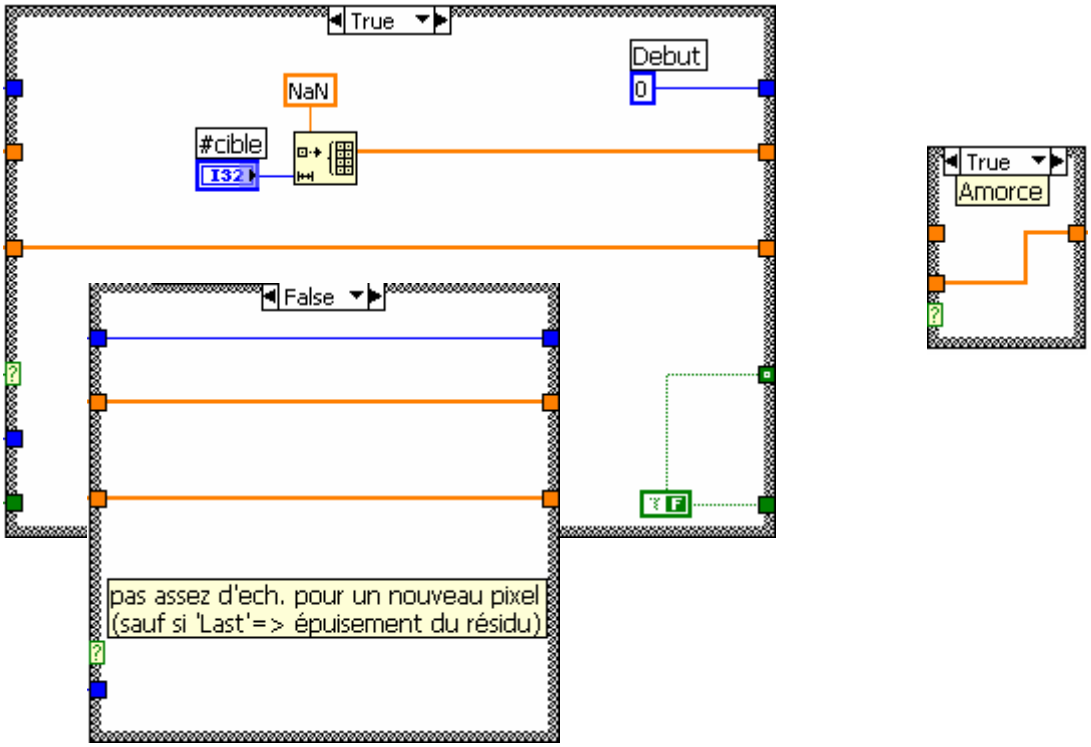
et les sorties :

- durée : durée en secondes de l'enregistrement
- nmM : # points min et Max après réduction
- Kp : rapport de réduction, ou taille du bloc à réduire par un couple min&Max
- dtmM : période équivalente de la série réduite

Si le rapport de réduction Kp est inférieur à 2, alors la lecture et l'affichage se feront directement comme à la première approche. Au-delà, la fonction 'Array Max & Min' sera appliquée à chaque bloc de Kp échantillons ainsi réduits à un couple accumulé dans la série dont la taille finale parfaitement maîtrisée est pré initialisée, pour ne pas exploser la mémoire. Il convient par ailleurs que cette décimation soit totalement découplée du flux d'arrivée des données sources ; d'une part pour respecter le bon principe de cohésion qui s'attache à séparer les entrées-sorties qui peuvent être très diverses et à contraintes variées, des traitements dont les paramètres doivent rester indépendamment maîtrisés. En l'occurrence, la lecture d'un gros fichier par blocs trouvera son optimum en fonction de la configuration (type de disque, cache mémoire, processeur,…) typiquement avec des tailles de 128k à 512k octets, soit 32k à 128k échantillons flottants codés sur 4 octets. Or un fichier de 4M octets soit 1M échantillons conduirait à un rapport de réduction de 10 pour 100 fois 1000 pixels et on imagine mal la performance de cent milles lectures de petits blocs de 10 échantillons ; à l'inverse si un enregistrement de 40Gb devait être affiché dans un graphe de 100 pixels avec un objectif de zoom limité à 50, la réduction serait de 2 millions et les blocs à lire de 8M octets, risquant un autre débordement mémoire.

Cela étant, l'extrême simplicité initiale fait place à un code un peu moins trivial…

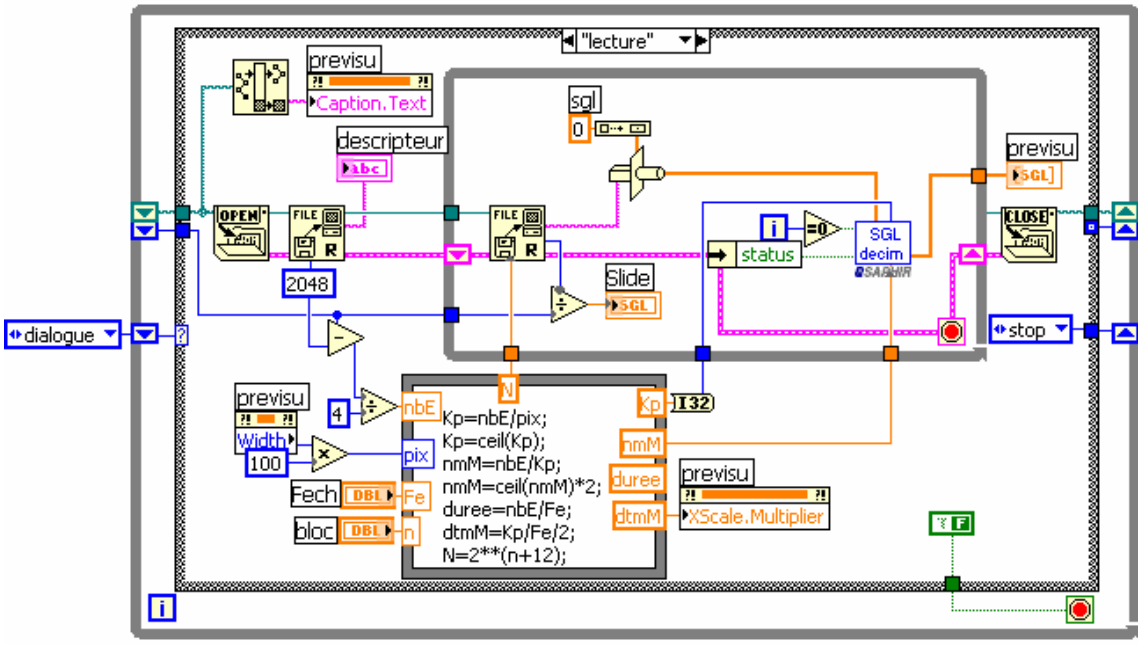




…qui ne doit pas décourager les indécis, mais seulement remettre les difficultés à leur place, la solution équivalente à cette problématique étant ni plus ni moins évidente au moyen de quelque langage que ce soit.

Dit autrement : "LabVIEW ne dispense pas de la réflexion avant codage !"

L'utilisation d'une telle fonction se fait alors aisément par un module appelant pouvant compléter ainsi le dialogue d'ouverture précédent, sans plus besoin de l'avertissement relatif à une taille qui ne présente plus de danger :



Ce « bout de code » n'est qu'un modeste proto destiné à tester et mettre en valeur le module précédent, lequel peut trouver un large champ d'utilité à condition d'être précisément documenté, enrichi d'autres exemples commentés de mise en application, et testé…

4 LA MULTIPLICATION DES PETITS PAINS

A la conception qui précède le codage d'un logiciel doivent se poser les questions de son champ d'application et de sa durée de vie probable, avec un objectif de capitalisation au moins à deux titres :

- y a-t-il un intérêt, immédiat ou potentiel, à gérer plusieurs instances simultanées ? en nombre borné ou en dimension dynamique ?

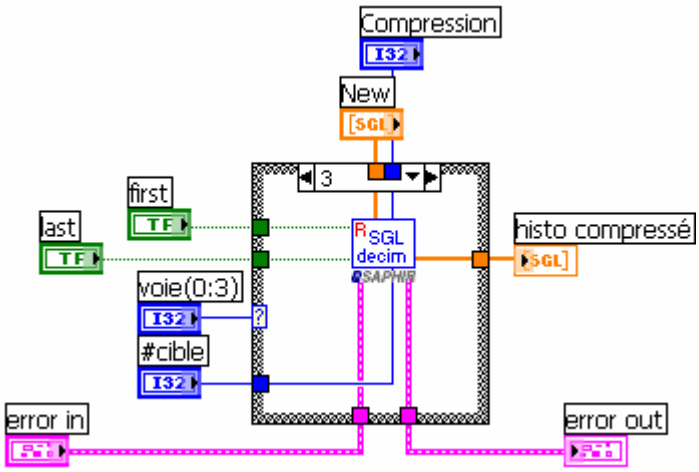
- plus généralement, ce composant pourra-il être étendu et adapté à d'autres champs d'applications ?

Une telle réflexion ne s'improvise pas après avoir finalisé une version « mono » d'un composant logiciel ; seul l'éventuel prototype, vu sa courte durée de vie, devrait bénéficier de tous les passes droits et s'autoriser les raccourcis. Immédiatement après la preuve faite, il convient de changer de casquette.

Concernant le présent exemple, on imagine aisément dès sa conception l'intérêt d'instancier les composants de sorte à exploiter de multiples canaux identiquement en parallèle.

4.1 une médiocre multiplication qui marche

Une construction appelant un nombre prédéfini d'instances de ce même module satisfera un besoin borné, à condition toutefois de bien songer à configurer le sous-VI pour une exécution réentrante que l'on prendra la bonne habitude de repérer par la nomination du VI d'une part et par une marque distinctive sur l'icône d'autre part ; chaque appel duplique alors le code et les données abritées, à défaut de quoi les registres circulaires ne distingueraient pas les différents canaux ; bien sûr la réentrance impose un appel unique pour chaque voie et il ne faudra surtout pas se laisser tenter à séparer la phase d'initialisation du cycle de lecture.



4.2 Pour ne pas conclure…

Associer à ce « Graphical programming » la culture « Object Oriented » s'avère alors tout à fait pertinent

Je laisse à mes confrères le soin d'introduire la pratique de GOOP et d'en justifier l'intérêt professionnel…