SA - EXERCICE 2

REACTION BIMOLECULAIRE PURE

1. Forme intégrée	2. Forme différentielle
bimolpur_1.cpp	bimolpur_2.cpp

Nous supposons maintenant que vous êtes un peu familiarisé avec la manière de travailler avec *Sa*. Nous en résumons simplement les principales étapes :

1) Ecrire le fichier programme, ou modèle (extension **.cpp**) qui contient les équations et les méthodes de calcul correspondant au modèle de la réaction.

2) Compiler ce fichier pour obtenir une version exécutable (extension **.exe**).

3) Exécuter Sa et compléter les paramètres, les conditions initiales et la plage de variation de la variable indépendante (temps en général), ou charger un fichier de commande (extension **.sac**) préexistant et l'adapter au problème actuel. Enregistrer le fichier de commande.

4) Lancer une simulation, avec ce modèle et ces paramètres, afin d'obtenir un fichier de données calculées (extension **.sad**) et différents graphiques d'illustration.

Nous avons volontairement occulté beaucoup d'autres possibilités de Sa, et nous continuerons pour l'instant à le faire, afin de nous concentrer sur l'essentiel.

Nestor est un assistant pour accomplir ces tâches de façon aisée et coordonnée. Une fois l'exécutable construit, toutefois, il peut être utilisé pour le lancer mais ce n'est pas absolument nécessaire : l'exécutable est autonome et peut être lancé en cliquant simplement sur son nom.

En cas de besoin : retour à l'exercice 1 (prise en main)

1. Forme intégrée

télécharger bimolpur_1.cpp télécharger bimolpur_1.sac

La première chose à faire, avec Nestor, est de **choisir un dossier de travail** (voir cette tâche dans exercice 1). Il n'est pas strictement indispensable d'en choisir (ou créer) un nouveau : on peut décider de regrouper une famille de modèles dans le même dossier, c'est une affaire de goût et d'organisation.

Pour écrire un nouveau fichier **.cpp**, il y a plusieurs possibilités :

1) Partir de zéro et utiliser les *Templates*, comme dans l'exercice 1

2) Partir d'un fichier existant déjà et le modifier, éventuellement en utilisant aussi les *Templates*.

La deuxième méthode est souvent très commode. C'est celle que nous adopterons ici.

Avec l'éditeur de Nestor, ouvrons donc le fichier de l'exercice 1 : monomol_1.cpp.

> Vous avez sûrement remarqué, à gauche du bouton *Edit*, que vous avez la possibilité de choisir votre éditeur : *Sa Editor* (par défaut), *Notepad* (l'éditeur texte pur de Windows) ou *PFE* (éditeur orienté programme, que vous pouvez trouver sur http://www.lancs.ac.uk/staff/steveb/cpaap/pfe/). A vous de choisir bien sûr... mais nous vous conseillons plutôt *Sa Editor* !

Afin de ne pas risquer de l'écraser par mégarde, sauvez-le immédiatement sous le nouveau nom **bimolpur_1.cpp**.

Editer ce fichier de façon à obtenir le programme suivant (les éléments modifiés sont en rouge) :

```
// bimolpur_1.cpp // name of this file (simple comment)
//-----------------------------------------------------
#include"global.h"
//-----------------------------------------------------
void Identification(Modele& modele)
{
modele.Fichier = String(__FILE__);
modele.Version = String(__DATE__) + String(" ") + String(__TIME__);
modele.Auteur = "Jane";    // Author (optional)
nom_syst = "bimolpur_1"; // name of the model
n_diff = 0;                // number of differential equations
first_var = 0;             // number of the 1st variable to be integrated
nv_mod = 2;                // number of variables of the model
nexp = 1;                  // number of experiments
}
//-----------------------------------------------------
void eqdiff (Sa_data x, Sa_data* y, Sa_data* dy)
{
}
//-----------------------------------------------------
```

```
void fappel()
{
   for (int i = 0; i < npt; ++i)
   {
      // calcul par l'équation intégrée :
      ca[0][i] = ca[0][0]/(2*ca[0][0]*p[0]*ind[i] + 1);  // [A]
      // calcul par l'équation de conservation :
      ca[1][i] = (ca[0][0] - ca[0][i])/2;  // [B]
   }
}
//---------------------------------------------------
```

La ligne

```
   ca[0][i] = ca[0][0]/(2*ca[0][0]*p[0]*ind[i] + 1);  // [A]
```

ne fait que traduire l'équation *(8)* du cours, et la ligne

```
   ca[1][i] = (ca[0][0] - ca[0][i])/2;  [B]
```

l'équation de conservation, sous la forme $B = (A_0 - A) / 2$.

N'oubliez pas de **sauver** ces modifications.

Compilez.

Lancez.

Sa s'ouvre avec *none* comme fichier de commande (.sac).

Pour faire un fichier de commande, il y a aussi plusieurs manières :

1) Directement à l'aide de l'éditeur : *Sa Editor* est disponible également depuis *Sa* (souligné 2 fois en rouge dans l'écran 5, exercice 1). Vous pouvez soit y ouvrir un fichier .sac existant déjà, soit utiliser le *Template* "*.sac file*" pour avoir une base, le modifier comme il convient, le sauver et le faire lire par *Sa*, dans l'onglet *Général*, volet *Fichier de commande*. Cette méthode peut être plus efficace lorsque les modifications à effectuer sont importantes, car on peut, en particulier, utiliser le copier-coller. Mais il faut faire très attention à respecter le format strict de ces fichiers, sinon *Sa* aura des comportements inattendus !

> Certains éléments des fichiers *.sac* sont obsolètes dans la version actuelle 3.3. Ils doivent cependant y figurer tels quels, pour des raisons de compatibilité avec les travaux effectués antérieurement.

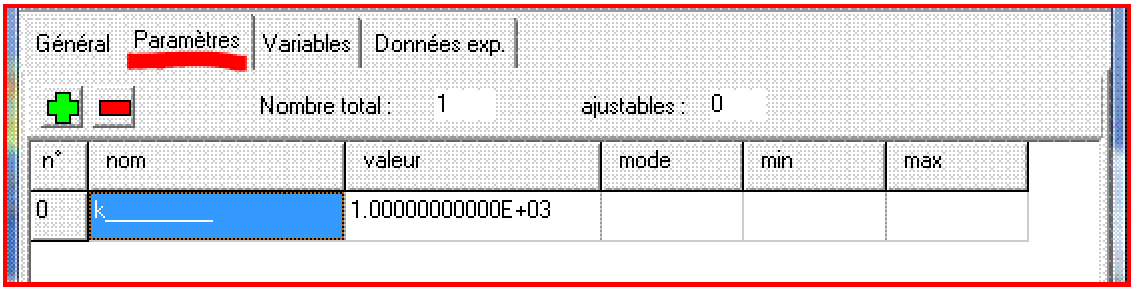
2) Directement par les onglets *Général*, *Paramètres* et *Variables*, comme dans l'exercice 1 (voir le paragraphe).

3) Faire lire un fichier .sac existant déjà, et le modifier comme il convient par la méthode 2. Il est bon dans ce cas de commencer par le sauver sous un nouveau nom, avant même de l'avoir modifié, afin d'éviter d'écraser l'original.

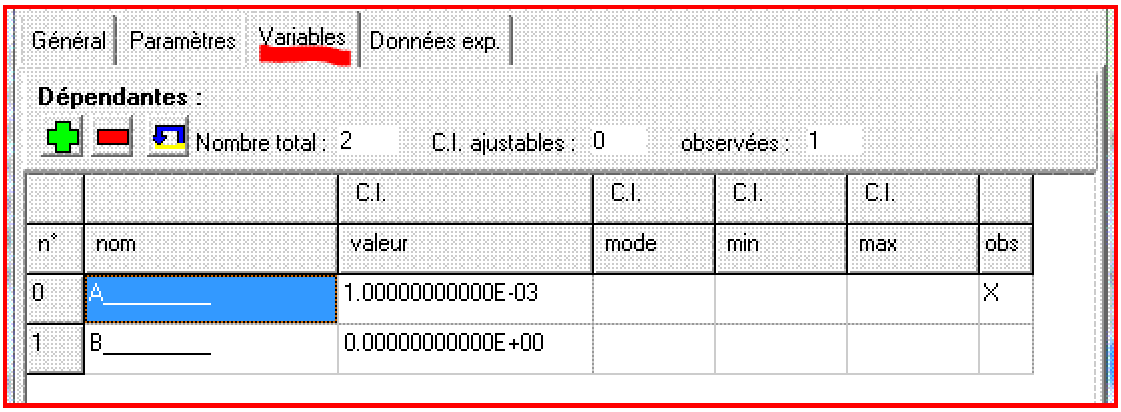
Nous vous conseillons les méthodes 2 ou 3, qui ont l'avantage de garantir le bon format des fichiers enregistrés, et sont commodes tant que le nombre de variables et de paramètres n'est pas trop important.

Utilisez donc une de ces deux méthodes pour créer un fichier **bimolpur_1.sac**.

Les captures d'écrans 1 et 2 suivantes montrent ce que vous devez avoir :



écran 1



écran 2

Mettez *Fin* de la variable indépendante à 20 (s).

Sauvez le fichier de commande. Puis lancez une simulation.

Vous devez obtenir des courbes semblables à la figure III.3.. Ajoutez d'abord la courbe de B, puis modifiez éventuellement les *Echelles* pour mieux comparer.

Vérifiez que l'échelle de temps varie en sens inverse de la concentration initiale de A_0, par exemple en multipliant celle-ci par 2, k étant maintenu constant.

> Tant que vous ne les avez pas détruites, vous pouvez consulter les pages graphiques émanant d'un calcul précédent, en cliquant simplement sur leur onglet. Cela permet, en particulier, de comparer des courbes avec des paramètres différents.
>
> Cependant, les données correspondant à ces pages antérieures ne sont plus présentes en mémoire : vous ne pouvez donc pas modifier ces tracés.

En gardant A_0 à sa valeur double, divisez maintenant k par 2. Comparez avec la toute première simulation.

2. Forme différentielle

télécharger bimolpur_2.cpp télécharger bimolpur_2.sac

Il s'agit maintenant d'effectuer l'intégration numérique de l'équation *(5)* du cours. Choisissez votre méthode, la démarche doit commencer à vous devenir familière. La fonction `eqdiff` doit ressembler à ceci :

```
void eqdiff (Sa_data x, Sa_data* y, Sa_data* dy)
 {
    dy[0] = - 2*p[0]*y[0]*y[0]; // dA/dt
    dy[1] = p[0]*y[0]*y[0];     // dB/dt
 }
```

Et dans `fappel`, il faut appeler l'intégrateur numérique `srkvi` :

```
srkvi(n_diff, &ca[first_var], ind, npt, h0, tol, iset, jacob, h_compt,
c_min);
```

Vous êtes peut-être encore troublé par le fait que les même choses s'appellent `y[0]`, par exemple, dans `eqdiff` et `ca[0][i]` dans le reste du programme, en particulier dans `fappel`. C'est que `y[0]` représente la variable courante d'intégration (la fonction `eqdiff` peut être appelée des millions de fois sans que vous vous en aperceviez), tandis que `ca[0][i]` représente une valeur mémorisée, que vous pouvez récupérer à la fin de l'intégration. Le fait que telle valeur soit mémorisée est déterminé par les paramètres *Début*, *Fin* et *Incrément* de la variable indépendante.

Revoir les explications plus détaillées dans l'exercice 1.

Noter la façon d'élever au carré : `y[0]*y[0]`.

Affectez les valeurs convenables à `n_diff` et à `first_var`.

Maintenant, nous allons ajouter à ce qui précède, **après l'intégration numérique**, le calcul de la vitesse et de l'avancement normalisés (nous avons ajouté des numéros pour nous repérer) :

```
  srkvi(...);
1. Sa_data r_ini;
2. r_ini = 2*p[0]*ca[0][0]*ca[0][0]; // vitesse initiale
3.
4. for (int i = 0; i < npt; ++i)
5. {
6.    ca[2][i] = 2*p[0]*ca[0][i]*ca[0][i]/r_ini; // vitesse normalisée
7.    ca[3][i] = (ca[0][0]-ca[0][i])/ca[0][0]; // avancement normalisé
8. }
```

La ligne 1 *déclare* une variable *locale* `r_ini` (accessible seulement à l'intérieur de la fonction qui la contient, ici `fappel`) dans le type `Sa_data`. Ce type est celui de toutes les variables numériques autres que des entiers utilisées par *Sa*. Par défaut, il est équivalent à la *double précision* (format flottant, 15 chiffres significatifs, valeurs absolues comprises entre 1.7×10^{-308} et 3.4×10^{308}). Il peut être changé par la fonction

Limits de *Nestor*... mais ce n'est vraiment pas nécessaire, et cela nécessiterait une recompilation complète de Sa.

La ligne 2 *affecte* à `r_ini` la valeur de la vitesse initiale. On aurait pu combiner la déclaration et l'affectation en écrivant directement :

```
Sa_data r_ini = 2*p[0]*ca[0][0]*ca[0][0];
```

La raison d'être de cette variable est de sortir de la boucle qui suit ce qui peut être calculé en dehors. Ce ne serait pas dramatique ici d'effectuer `npt` fois la même opération, mais c'est une bonne pratique lorsque les programmes deviennent plus lourds. De plus, cela rend ces derniers souvent beaucoup plus lisibles.

La ligne 4 initie une boucle sur le nombre de points, et les lignes 6 et 7 calculent la vitesse normalisée et l'avancement normalisé et les affecte aux variables 2 et 3, respectivement.

Les vitesses, au signe près, ont déjà été calculées au cours de l'intégration numérique, à chaque appel de `eqdiff`. Mais ces valeurs n'ont servi qu'à la progression de l'intégration et ne sont pas enregistrées (sinon, la mémoire de l'ordinateur serait rapidement saturée). C'est pourquoi il est nécessaire de les recalculer à part, une fois l'intégration terminée.

Affectez la valeur convenable à `nv_mod`. Le programme est terminé. Sauvez (sous le nom bimolpur_2.cpp, par exemple). Compilez. Lancez , construisez un fichier de commande (bimolpur_2;sac), et faites une simulation.

Si vous avez utilisé la même constante de vitesse et les même valeurs initiales que dans la version intégrée, vous devez obtenir évidemment le même résultat.

Sur une nouvelle page graphique, tracez (bouton Y/X) la vitesse normalisée en fonction de l'avancement normalisé : vous devez obtenir quelque chose qui ressemble à la figure III.4. Cependant, vous remarquez que la fin de la courbe n'est pas au point de coordonnées (1, 0) comme elle le devrait. C'est que la réaction n'est pas complètement terminée, et d'ailleurs... elle ne le sera jamais ! Si vous augmentez la valeur de *Fin*, vous vous rapprocherez un peu plus.

En augmentant la valeur de *Fin*, il faut augmenter en conséquence la valeur de *Incrément*., afin de ne pas dépasser le nombre de points maximum, `NPT`, qui vaut 4005 par défaut (c'est très suffisant pour le genre de problèmes que nous traitons pour l'instant). Sinon, vous aurez un message d'information et la simulation sera bloquée. Ce nombre est également modifiable par la fonction*Limits* de *Nestor*.