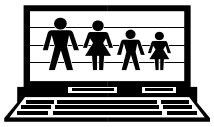
BDSPS 

Guide de programmation

Le présent guide décrit la façon d'utiliser la BDSPS en mode boîte de verre. Ce mode permet à l'utilisateur d'ajouter des variables et des paramètres à la base de données ainsi que d'en modifier les algorithmes ou d'en appliquer de nouveaux. Le mode exige l'utilisation du compilateur C++ de Microsoft Visual Studio.net 2008.

Statistics Canada Statistique Canada

Canada

Table des matières

Introduction

Le *Guide de programmation* décrit la façon dont l'utilisateur peut modifier la BDSPS pour modéliser des systèmes d'imposition/transfert ou des options stratégiques non traitables directement par la BDSPS actuelle, par exemple pour modifier la logique du système d'imposition/transfert afin d'évaluer les effets distributifs statiques associés à une proposition stratégique.

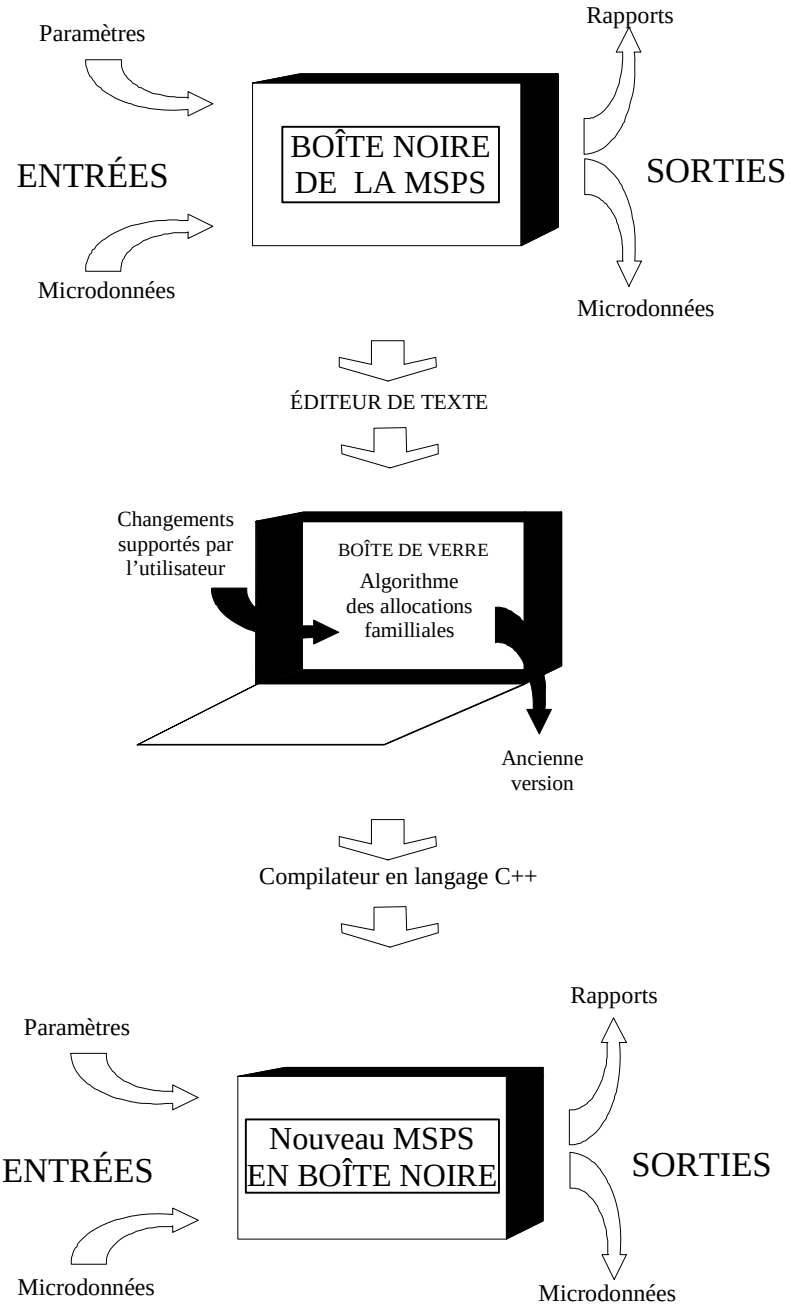
Ce chapitre traite de différents sujets préparatoires essentiels à une bonne compréhension de la façon d'utiliser la BDSPS en mode boîte de verre. Il aborde spécifiquement les sujets suivants :

(1) Description du mode boîte de verre, particulièrement par rapport au mode boîte noire,
(2) Exigences matérielles et logicielles liées à l'utilisation du mode boîte de verre, et
(3) Niveau requis de connaissance en programmation.

Les sections subséquentes du Guide portent sur les aspects détaillés liés spécifiquement au développement d'applications en mode boîte de verre. La section qui suit décrit une procédure de « *démarrage rapide* », axée sur une modification simple de la BDSPS en mode boîte de verre, qui permet de vérifier que la base de données actuelle a été installée de manière appropriée. La section *BDSPS et structure du répertoire de la boîte de verre* énonce les caractéristiques de la structure de sous-répertoires associée aux différents modes d'exploitation de la boîte de verre. La section *Structure particuliers-ménages de la BDSPS* donne des détails importants sur les principales structures de données utilisées par la BDSPS. La section *Structure d'appel de la BDSPS* décrit la structure d'appel des modules BDSPS d'un modèle particulier. La section *Développement en boîte de verre : ajout de paramètres scalaires ordinaires*, qui traite des formes les plus courantes de paramètres scalaires, aborde les mécanismes utilisés pour l'ajout à un modèle BDSPS de paramètres de modèle définis par l'utilisateur. La section *Développement en boîte de verre : ajout de paramètres inhabituels* traite de l'ajout de types de paramètres scalaires moins courants et de vecteurs et de matrices associés à de nouveaux paramètres de modèle. La section *Développement en boîte de verre : ajout de nouvelles variables* décrit l'ajout de nouvelles variables à un modèle. Enfin, la section *Modification des variables de données de base et de variante* définit formellement le mode de gestion des algorithmes standard et de rechange dans le contexte des opérations en mode boîte de verre.

OBJET DU MODE BOÎTE DE VERRE

Le diagramme ci-dessous donne un aperçu global simplifié du processus de simulation des impôts et des transferts :



L'utilisateur précise une suite d'entrées (paramètres et données) traitées par un système d'algorithmes (boîte noire) qui produit des sorties système (tableaux et microdonnées). L'utilisateur peut créer plusieurs simulations différentes en variant les entrées puis en analysant les sorties. Il peut même déduire une partie du contenu de la boîte noire par des essais répétés. Cependant, le contenu de la boîte noire limite les simulations possibles. Par exemple, si les règles d'application de la taxe sur les biens et services (incluant les dispositions qui permettent de déterminer si les paramètres et les données d'entrée sont appropriés) ne sont pas incluses dans le système d'algorithmes, le programme ne peut être simulé sans effectivement ouvrir et modifier la boîte noire. La capacité d'examiner l'intérieur de la boîte noire et d'en modifier le contenu équivaut à transformer la boîte noire

en boîte de verre.

Le Guide explique la façon d'utiliser la BDSPS en mode boîte de verre. Nommément, le terme « mode boîte de verre » désigne une méthode de modification des versions exécutables du programme BDSPS pour effectuer des analyses que ne permettrait pas une BDSPS originale non modifiée. Ce mode peut servir à ajouter ou modifier des paramètres, des variables et des algorithmes. Son utilisation nécessite toujours la modification du code source C++ et la compilation d'une nouvelle version exécutable du programme. Le « mode boîte noire » désigne l'exécution subséquente d'une version exécutable transmise par Statistique Canada ou modifiée par l'utilisateur en mode « boîte de verre ». L'utilisateur se sert toujours du mode boîte noire pour effectuer, avec des paramètres modifiés, des variables d'utilisateur et des expressions de totalisation, diverses simulations liées à des politiques.

Compte tenu des étapes supplémentaires que nécessite l'utilisation du mode boîte de verre, l'utilisateur devrait éviter de l'utiliser dans la mesure du possible. La BDSPS offre un certain nombre de mécanismes qui permettent aux analystes d'obtenir les nombreux résultats recherchés sans reprogrammation. La technique la plus courante consiste à modifier les ensembles de paramètres de programme par défaut qui gouvernent la BDSPS. L'analyste peut simuler les répercussions d'une augmentation ou de l'abolition des allocations familiales en modifiant les valeurs numériques des paramètres concernés. Ou il peut définir ses propres variables dans le fichier de paramètres de contrôle et utiliser les variables ainsi obtenues dans une vaste gamme de sorties BDSPS. Le *Guide d'introduction* inclut un exemple détaillé dans lequel un analyste utilise les variables définies par l'utilisateur pour simuler un crédit d'impôt sur les revenus salariaux. L'analyste peut également créer « à la volée » des variables sous forme d'expressions et les exporter ou les entrer dans des tableaux comme s'il s'agissait de vraies variables; il peut facilement représenter les écarts qu'il y aura pour une variable donnée dans les systèmes d'imposition/transfert de base et de variante. Le *Guide d'utilisation des tableaux croisés* inclut plusieurs exemples de ce type de création à la volée.

Le mode boîte de verre doit être utilisé dans les situations suivantes :

(1) Ajout de nouveaux paramètres.
(2) Ajout de nouvelles variables qui exigent un renvoi à d'autres membres particuliers de la famille.
(3) Nouveaux projets conçus pour interagir avec le système d'imposition/transfert. Par exemple, des allocations imposables pour les nouveau-nés.
(4) Nouveaux projets qui modifient la logique des programmes existants de façons qui n'ont pas encore été paramétrées.

Si l'utilisateur doit apporter de telles modifications à la BDSPS pour refléter d'autres systèmes d'imposition/transfert, il doit connaître à fond les techniques décrites dans le présent Guide.

EXIGENCES MATÉRIELLES ET LOGICIELLES DU MODE BOÎTE DE VERRE

Le document *Introduction et aperçu* énonce les exigences détaillées relatives au matériel et au logiciel.

Les principaux aspects des exigences logicielles sont les suivantes :

1. L'utilisation de la BDSPS en mode boîte de verre exige l'utilisation de Visual C++ pour compiler les instructions du code source C de l'utilisateur en langage machine requis par la BDSPS. Pour cette version de la BDSPS, vous devez utiliser Microsoft Visual Studio .net 2008.

2. La BDSPS requiert un système d'exploitation compatible avec la version de Visual C++.

CONNAISSANCE REQUISE DE LA PROGRAMMATION

Puisque l'utilisation de la BDSPS en mode boîte de verre exige une certaine programmation, l'utilisateur devra posséder davantage de connaissances qu'un utilisateur habituel. Cette section souligne les connaissances l'utilisateur devra soit posséder, soit consentir à apprendre.

Connaissances requises du système d'exploitation

L'utilisation de la BDSPS en mode boîte de verre exige de l'utilisateur qu'il maîtrise un certain nombre d'aspects liés au système d'exploitation. Il doit connaître le concept d'environnement DOS et ses variables, telle la variable PATH. Pour bien utiliser le mode boîte de verre, il doit également utiliser efficacement un certain nombre de commandes DOS.

L'interface Visual SPSD/M peut exécuter des modèles en mode boîte de verre. Les utilisateurs doivent indiquer dans la boîte de sélection de scénario qu'il faut utiliser un autre code de base et accéder au fichier exécutable de la boîte de verre. Une fois que les paramètres de la boîte de verre ont été chargés, on peut accéder aux nouveaux paramètres dans les listes hiérarchiques. Les nouveaux paramètres d'utilisateur se trouvent dans la dernière section de chaque hiérarchie.

Consultez le document *Comment exécuter le guide du MSPS* pour plus de détails sur l'utilisation de SPSD/M Visuel et MSPS Classique.

Concepts de base de programmation (non liés à un langage)

Vous ne devez pas utiliser le mode boîte de verre de la BDSPS comme base d'apprentissage de votre premier langage de programmation. Les utilisateurs doivent posséder une bonne connaissance d'au moins un langage informatique de haut niveau (p. ex., FORTRAN, BASIC, PASCAL et SAS) avant d'utiliser la boîte de verre. Puisque les applications en boîte de verre exigent de la programmation en langage compilé, il est souhaitable que les utilisateurs possèdent déjà une bonne connaissance des concepts clés. L'utilisateur doit se sentir à l'aise d'utiliser un éditeur de texte pour rédiger ou réviser le code source et un compilateur pour produire le fichier exécutable correspondant. Il lui sera utile de posséder une connaissance approfondie des notions de bibliothèques, de macros, de programmation modulaire et de validation de programme.

Il est essentiel que l'utilisateur applique l'expérience acquise avec ces concepts. Il est préférable, avant de s'attaquer à des applications en boîte de verre, qu'il ait déjà rédigé et mis au point plusieurs programmes inhabituels en d'autres langages que le langage C++. Il est

possible pour un utilisateur d'apprendre à programmer avec la BDSPS, mais cela n'est pas recommandé. Les futurs utilisateurs de la BDSPS qui souhaitent acquérir ou renforcer leurs compétences de base en programmation peuvent consulter une grande variété d'ouvrages disponibles sur le sujet.

Connaissance du langage de programmation C

Initialement en langage C, le code de base de la BDSPS a depuis évolué vers le langage C++. Dans le présent document, le langage de programmation fait référence aux deux langages, sans distinction. Les utilisateurs actuels de la boîte de verre doivent programmer en langage C++. Bien que la structure de la BDSPS prévoie que certaines fonctions sont effectuées pour l'utilisateur, ce dernier sera plus efficace s'il connaît bien au préalable les notions sous-jacentes. Les utilisateurs doivent comprendre le pourquoi de la définition des constantes et de la déclaration des variables, et en saisir la portée. Ils doivent comprendre les variables et les types de variables, particulièrement les variables pointeurs et les variables structurées, ainsi que la façon dont le langage C les utilise. Ils doivent comprendre la nature et la structure des fonctions et la variété des instructions qui les composent. Ils doivent connaître à fond les principaux flux d'instructions de commande du langage C (if-else, switch, while, for, do-while), ainsi que les tableaux d'attributions et d'opérateurs du langage C, y compris les opérateurs d'incrémentation. Les utilisateurs qui ont déjà travaillé avec d'autres langages de programmation, et qui sont en mesure d'absorber cette information sous forme concentrée, peuvent consulter le livre de référence standard de Kernighan et Ritchie « The C Programming Language ». De plus, le manuel du langage C++ livré avec le produit C++ Optimizing Compiler de Microsoft est une source très utile et fiable d'information sur ce langage et sa mise en œuvre.

Enfin, il va de soi que les utilisateurs de la BDSPS doivent comprendre les notions de base du compilateur C++ de Microsoft et saisir tout ce qu'il permet d'accomplir; ils doivent connaître les différents messages d'erreur que le compilateur peut produire lorsqu'il traite le code d'utilisateur. L'ensemble des manuels de Microsoft livrés avec le compilateur C++ font autorité dans ce domaine.

Exemple de démarrage rapide

Comme le titre l'indique, ce chapitre permet de commencer à utiliser rapidement la BDSPS en mode boîte de verre. Le chapitre remplit trois grandes fonctions. En premier lieu, il permet à l'utilisateur de vérifier l'installation du compilateur et de la BDSPS. S'il réussit à exécuter l'exemple simple fourni dans ce chapitre, c'est que toutes les grandes étapes de l'installation ont été exécutées correctement. En deuxième lieu, l'exemple sert d'introduction à la terminologie et aux principaux concepts de la boîte de verre. Finalement, l'exemple illustre, de manière intégrée, le flux général des applications du mode boîte de verre.

La technique utilisée dans ce chapitre est essentiellement narrative. Le chapitre met l'accent sur l'approche générale et fait franchir au lecteur toutes les étapes d'élaboration d'une application en boîte de verre simplifiée. Il décrit les principaux détails de l'exercice sans viser l'exhaustivité. L'illustration particulière utilisée ici a été choisie pour sa simplicité; elle tient compte des aspects les plus critiques des applications en boîte de verre sans s'attarder

aux exigences supplémentaires propres aux applications plus complexes.

Pour illustrer une application en boîte de verre relativement simple, nous reprenons un exemple de simulation utilisé précédemment en mode boîte noire, soit l'ajout d'un nouveau crédit d'impôt sur les revenus salariaux initialement présenté à la session 3 des exemples de simulations du document *Introduction et aperçu*. La conception de base de la prestation est reprise ci-dessous.

Admissibilité :	Les personnes âgées de 21 ans et plus sont individuellement admissibles si elles résident dans des familles de recensement qui incluent des enfants âgés de moins de 21 ans.
Prestation maximale :	Les prestations maximales sont de 1 200 $ par personne admissible.
Test de revenu :	Le test de revenu est basé sur le revenu d'emploi d'une personne, augmenté du revenu d'emploi de son conjoint, le cas échéant.
Points tournants :	Les prestations commencent dès le premier dollar de revenu gagné. Elles atteignent le maximum à 8 000 $ de revenu d'emploi et se maintiennent à ce niveau jusqu'à 12 000 $ de revenu gagné, maximum après lequel elles commencent à diminuer.
Taux de réduction :	Une prestation égale à 15 % du revenu gagné est payable jusqu'à un maximum de 1 200 $. Au delà de 12 000 $ de revenu familial gagné, les prestations maximales sont réduites de 10 ¢ pour chaque dollar supplémentaire de revenu gagné.

Comme l'indique l'exposé narratif, le lecteur ne devrait pas s'interroger sur les « pourquoi » de la mise en œuvre. Il trouvera dans les sections subséquentes du Guide de programmation des explications plus détaillées. Cependant, on incite fortement l'utilisateur à exécuter l'exemple au complet et à effectuer réellement toutes les tâches décrites. Ce n'est que de cette manière que le premier but de l'exercice, vérification du processus d'installation, peut être atteint.

PRÉLIMINAIRES

Pour commencer, l'utilisateur doit sélectionner un sous-répertoire de travail, soit un répertoire du disque dur dans lequel il modifiera les copies pertinentes des fichiers de code source en langage C et décrira la nature du nouveau système. **Nous recommandons fortement d'utiliser un répertoire autre que ceux créés pour la BDSPS elle-même lors de son installation.** L'utilisateur peut créer un nouveau sous-répertoire le cas échéant. Aux fins du présent exposé, nous présumons qu'il peut utiliser comme sous-répertoire de travail le sous-répertoire nommé `GLASSEX1`.

L'utilisateur commence le processus en copiant, du sous-répertoire `GLASS` de la BDSPS au sous-répertoire de travail `GLASSEX1`, tous les fichiers de gabarit pertinents. Ces fichiers sont ceux qui contiennent déjà, pour une application en boîte de verre, la majeure partie de l'information nécessaire que l'utilisateur modifiera pour créer les versions finales requises par l'application. Dans cet exemple, les fichiers de gabarit pertinents sont les suivants :

1. `Adrv.cpp`, gabarit « pilote » de rechange qui invoque dans l'ordre approprié toutes les fonctions d'imposition/transfert de la BDSPS. Livré avec la BDSPS, il est en réalité une réplique de la fonction pilote de base (**l'utilisateur doit le copier dans son sous-répertoire de travail**).

2. `Agai.cpp`, gabarit de rechange qui permet de calculer les nouveaux crédits remboursables et les nouvelles prestations garanties. Livré comme élément de la BDSPS, il est une sous-fonction dont peuvent se servir les utilisateurs de la boîte de verre qui désirent intégrer un nouveau programme n'ayant aucune incidence sur les programmes actuels du système d'imposition. (**L'utilisateur doit le copier dans son sous-répertoire de travail**).

3. `SPSMGL.sln` et `SPSMGL.vcproj`, gabarits qui permettent d'effectuer la compilation et les liaisons du nouveau modèle de l'utilisateur (**il faut copier ces fichiers du répertoire /spsm/glass dans votre sous-répertoire de travail**).

Pour d'autres applications en boîte de verre, l'utilisateur devra peut-être également copier d'autres gabarits d'imposition/transfert et fichiers d'en-tête en langage C. Dans cet exemple, toutefois, l'utilisateur n'est pas tenu de modifier de fichiers d'en-tête puisque le modèle ne crée aucune nouvelle variable et n'utilise aucun nouveau paramètre officiel.

La procédure générale de l'application en boîte de verre utilisée aux fins d'illustration est simple.

Dans les COPIES inscriptibles des fichiers `Adrv.cpp et Agai.cpp`, nous effectuons les petites modifications décrites ci-dessous.

1. Nous ouvrons ensuite l'utilitaire de solution `SPSMGL.sln` dans Visual Studio .net.

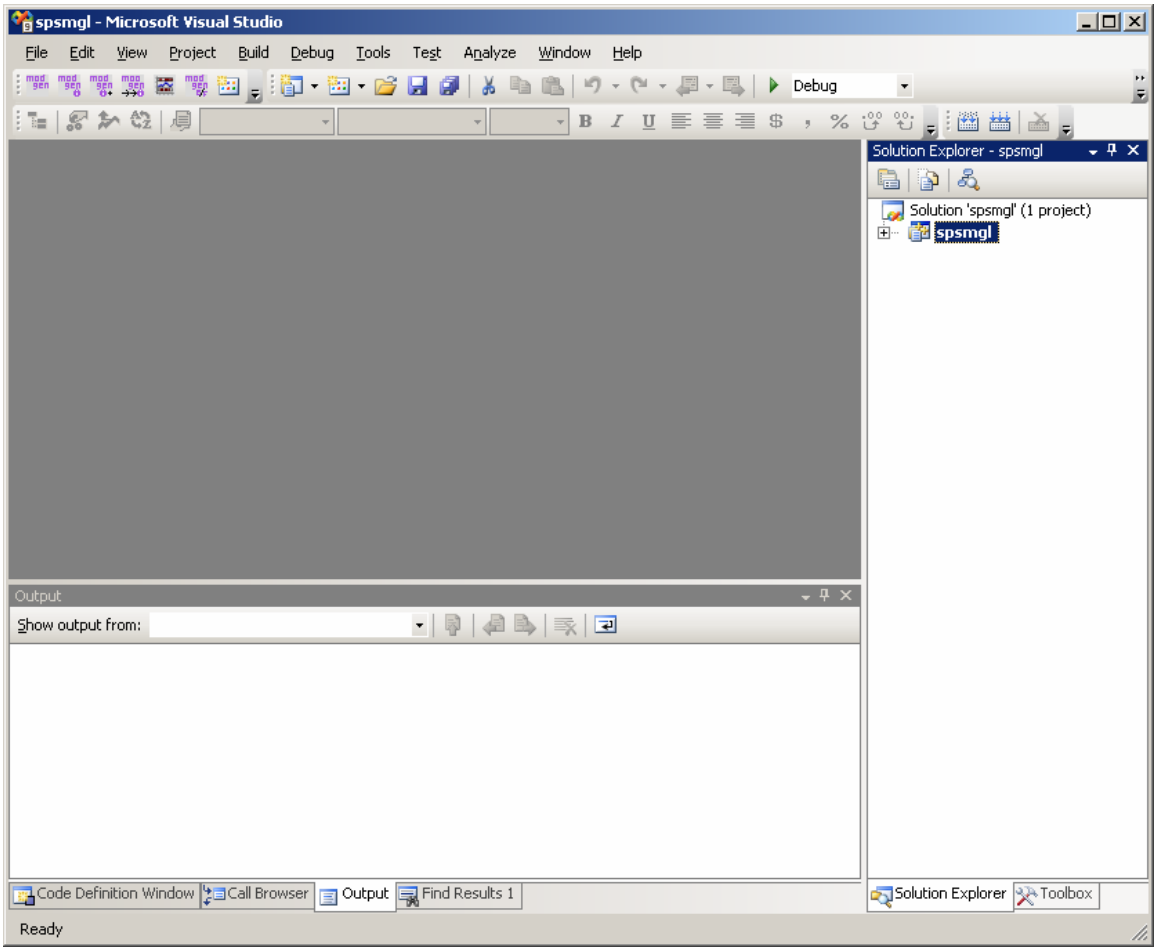
MODIFICATION DE L'ENVIRONNEMENT DU PROJET

On doit ajouter le sous-répertoire clé `\SPSM\DEFS` dans **Tools: Options: Projects: VC++ Directories: Include files** puisqu'il contient les définitions relatives aux applications en boîte de verre.

On doit ajouter le répertoire clé `\SPSM\WIN32` dans **Tools: Options: Projects: VC++ Directories: Library files**, puisqu'il contient les bibliothèques pertinentes de la boîte de verre.

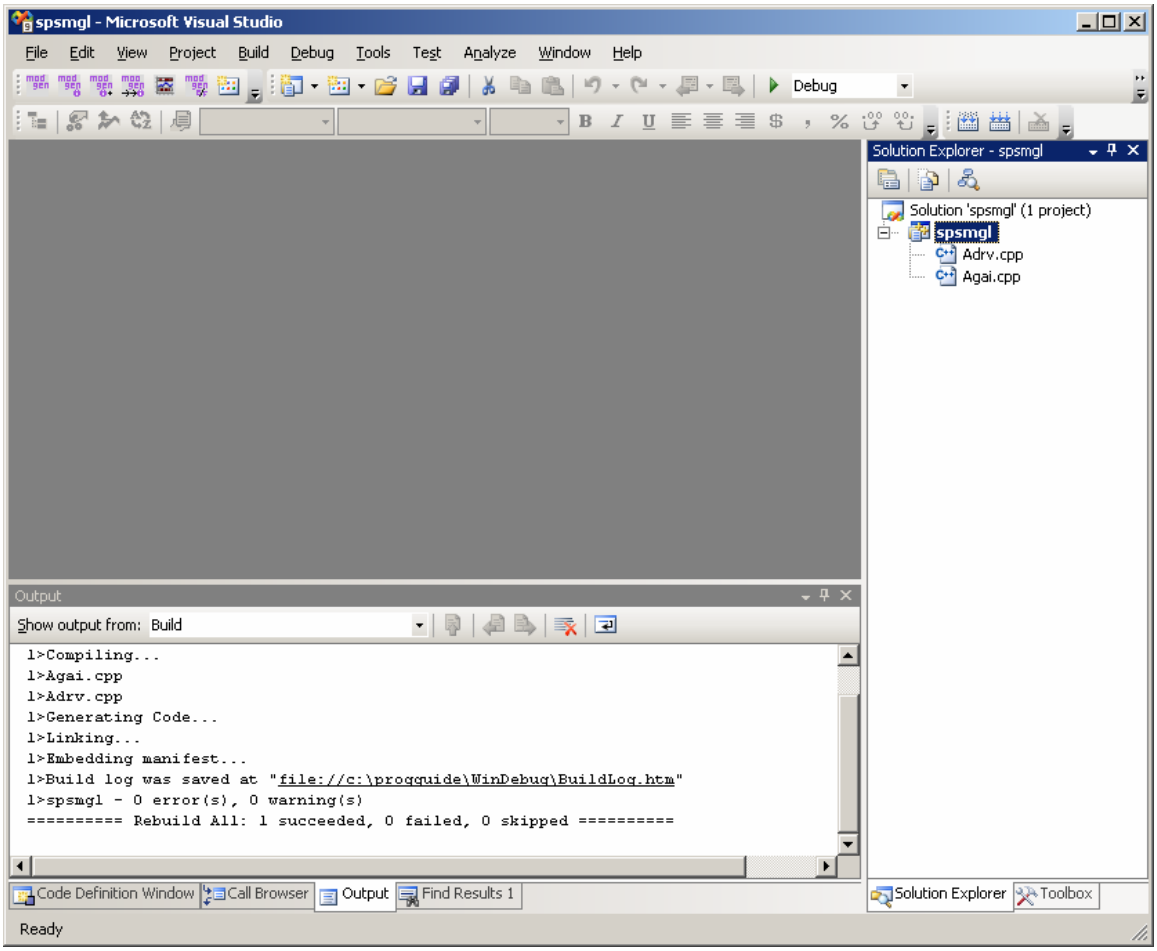
AJOUT DE FICHIERS AU PROJET

Pour ajouter des fichiers au projet, il faut d'abord ouvrir le fichier de solution spsmgl.exe dans le répertoire de travail. Lorsque vous double-cliquez sur ce fichier, le système affiche l'écran suivant :



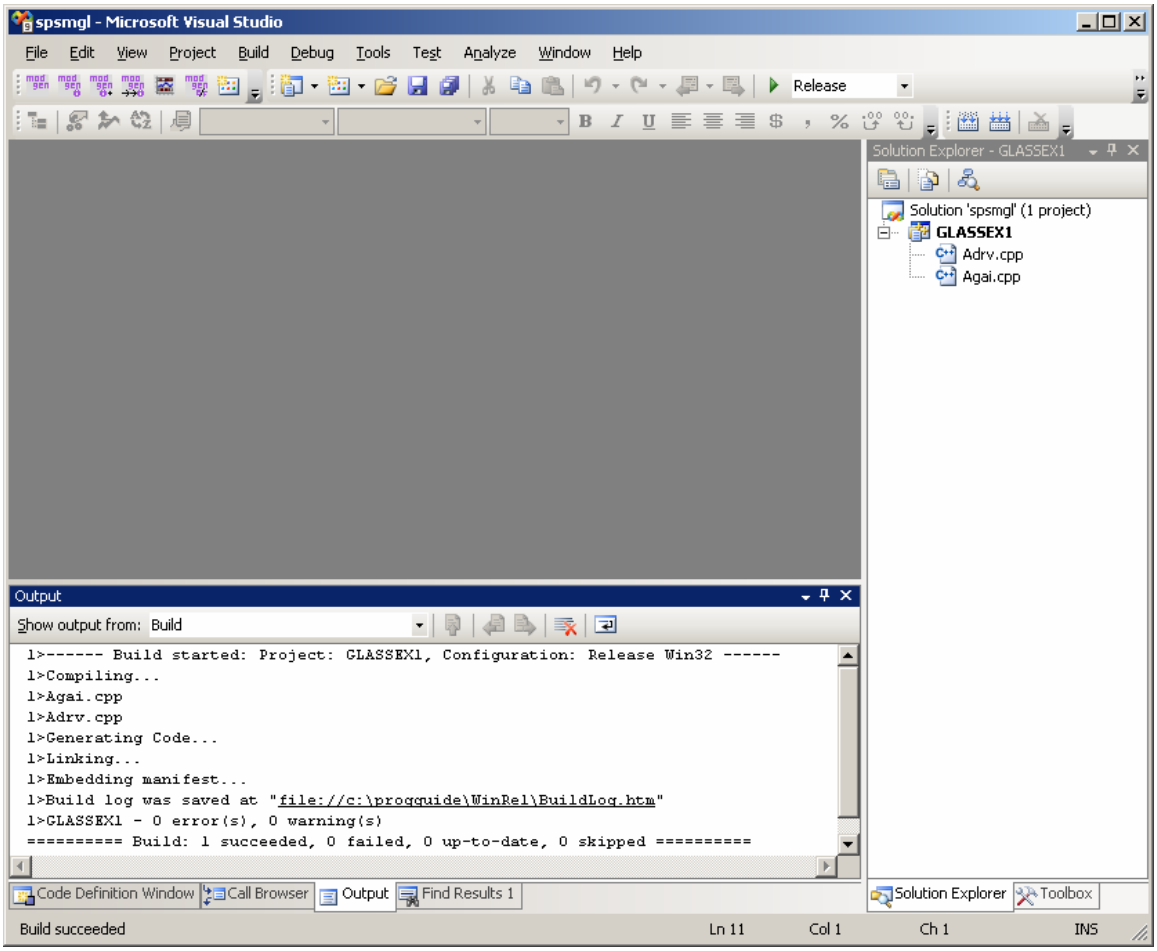
Avant d'ajouter les fichiers sélectionnés, assurez-vous que la solution spsmgl est en surbrillance (voir le panneau de droite ci-dessus) et que l'onglet Solution est sélectionné au bas du panneau solution.

Pour ajouter les fichiers Adrv.cpp et Agai.cpp, sélectionnez Project: Add Existing Item et naviguez vers le répertoire de travail, sélectionnez les deux dossiers, puis cliquez sur Open.



Si les fichiers sont ajoutés correctement, ils doivent figurer sous la solution spsmgl, tel que ci-dessus. À ce stade-ci, les utilisateurs doivent construire le modèle afin de s'assurer que tout fonctionne comme il se doit. Ils peuvent utiliser à cette fin le menu déroulant Build: Build spsmgl.

Dans ce petit exemple, on remplace par glassex1 le nom de fichier exécutable par défaut spsmgl du modèle compilé. À cette fin, on peut modifier les propriétés de solution sous le panneau Solution du côté droit de l'écran. Les utilisateurs doivent également s'assurer que la solution spsmgl est en surbrillance. Ensuite, dans le panneau des propriétés près du champ (Name), ils entrent le nouveau nom exécutable, soit GLASSEX1, et sauvegardent les modifications. Les étapes de Build doivent ensuite afficher le nouveau nom exécutable. La sélection du menu déroulant Build doit afficher le changement de nom, tel que ci-dessous :



MODIFICATION DE LA FONCTION DE PILOTE DE RECHANGE (ADRV.CPP)

`Le fichier Adrv.cpp` contient deux types d'information que l'utilisateur du mode boîte de verre voudra modifier. Le premier concerne l'information d'étiquetage que la BDSPS utilise dans ses rapports et ses messages d'erreur. Lorsque l'utilisateur modifie cette information de manière appropriée, la sortie fournit une information plus pertinente. Le second type concerne les appels de fonction qui effectuent l'essentiel des calculs de système d'imposition/transfert du modèle.

L'utilisateur modifie l'information d'étiquetage dans la partie du code semblable à ce qui suit et qui débute approximativement à la ligne 79 :

```
==================== GLOBAL VARIABLE DEFINITIONS ============== */
/*global*/ char ALTNAME[IDSIZE+1] = "Unnamed";
/* Give global string describing version of this module */
/*global*/ char FAR Tdrv[] = "Untitled"
```

La chaîne `ALTNAME[IDSIZE+1]` fournit un identificateur pour le pilote de rechange; l'utilisateur remplace le paramètre fictif « Unnamed » par le nom plus significatif « EITC Quick Start ». La longueur du nouveau nom ne doit pas être supérieure à 20 caractères. Ce nom s'affichera dans l'écran de bienvenue. La chaîne `Tdrv[]` attribue un titre au pilote de rechange; l'utilisateur remplace le paramètre fictif « Untitled » par le titre plus significatif « EITC Quick Start ». La longueur du nouveau titre ne doit pas être supérieure à

20 caractères. Le contenu de TDrv figure dans le fichier de paramètres de commande à titre de description d'algorithme. Une fois ces substitutions effectuées, la « section étiquetage » se présente comme suit :

```
/* ================== GLOBAL VARIABLE DEFINITIONS ============== */
/*global*/ char ALTNAME[IDSIZE+1] = "EITC Quick Start";
/* Give global string describing version of this module */
/*global*/ char FAR Tdrv[] = " EITC Quick Start"
```

Dans le corps du code, l'utilisateur doit effectuer une seule modification pour indiquer que le calcul des prestations du système de variante doit utiliser un autre calcul pour les crédits.

La ligne pertinente du code, près de la ligne 164, est la suivante :

```
gai(hh);     /* compute new guarantees, refundable credits */
```

Si elle n'est pas modifiée, cette ligne invoque simplement une fonction qui attribue la valeur 0 à la variable imiosa pour tous les membres du ménage. L'utilisateur modifie la ligne pour qu'elle invoque plutôt l'autre calcul de crédit que nous allons décrire brièvement ci-dessous. Cette modification consiste simplement à substituer le nouveau nom de fonction; le code source révisé se présente comme suit :

```
Agai(hh);    /* compute new guarantees, refundable credits */
```

Dans cet exemple, ces trois simples modifications suffisent à modifier la fonction Adrv.cpp.

AJOUT DU NOUVEAU CRÉDIT D'IMPÔT SUR LES REVENUS SALARIAUX (AGAI.CPP)

La fonction Agai.cpp est conçue pour permettre aux utilisateurs de la boîte de verre d'ajouter de nouveaux programmes n'ayant aucune incidence sur les autres programmes du système actuel d'imposition du modèle de rechange. Tout comme c'est le cas pour celles du fichier Adrv.cpp, les modifications apportées par l'utilisateur sont de deux ordres : modifications de l'étiquetage et modifications du corps du programme.

Les modifications d'étiquette sont très simples. Près de la ligne 48, la fonction attribue au module un titre, Tgai[], qui est utilisé dans le rapport dans lequel la BDSPS indique les fonctions utilisées pour calculer les impôts et les transferts. Comme c'est le cas pour le titre du pilote, ce titre figure comme description d'algorithme dans le fichier de paramètres de commande. La partie pertinente du code est la suivante :

```
=============== GLOBAL VARIABLE DEFINITIONS ================== */
/* Give global string describing version of this module */
/*global*/ char FAR Tgai[] = "Untitled"
```

L'utilisateur modifie la chaîne « Untitled » de manière à donner davantage d'information. La section obtenue se présente comme suit :

```
=============== GLOBAL VARIABLE DEFINITIONS ================== */
/* Give global string describing version of this module */
/*global*/ char FAR Tgai[] = "EITC Quick Start"
```

Les modifications de la partie substantielle de la fonction Agai.cpp sont un peu plus complexes, mais pas extrêmement compliquées. Cette fonction inclut une variable

(imiosa – Garanties ou autre montant d'AS) dont la valeur en mode boîte noire est toujours « zéro »; toutefois, on peut l'utiliser dans les algorithmes de « boîte de verre » pour modéliser de nouveaux programmes. La valeur de « imiosa » est incluse dans les programmes de transfert fédéral (imftran), qui sont également inclus dans le revenu disponible (immdisp). Cette configuration est pratique pour les utilisateurs de la boîte de verre et leur permet de modéliser de nouveaux programmes pour lesquels il est possible de mesurer l'incidence sur le revenu disponible.

Le fichier `Agai.cpp` inclut une petite boucle de traitement de ménage qui attribue à la variable imiosa la valeur 0 pour toutes les personnes.

```
      register P_in in;
      register int ini;

      DEBUG_ON("Agai");

      /* process persons in household */
      for (ini=0, in=&hh->in[0]; ini<hh->hhnin; ini++, in++) {
            in->im.imiosa = ZERO;
      }
      DEBUG_OFF("Agai");
```

Ce bloc de code peut être supprimé (ou mis en commentaires) et remplacé par ce qui suit :

```
      register P_in in;
      register int ini;
      register P_in ineld;
      register P_in inspo;
      register P_cf cf;
      register int cfi;
      int nceitc;
      NUMBER cfempinc;
      NUMBER eitc;

      DEBUG_ON("Agai");
      /* process persons in household – currently commented out*/
/*    for (ini=0, in=&hh->in[0]; ini<hh->hhnin; ini++, in++) {
            in->im.imiosa = ZERO;
      }
*/

    /* process each census family in household */
    for (cfi=0, cf=&hh->cf[0];  cfi<hh->hhncf;  cfi++, cf++) {
         /* initialise elder's pointer */
         ineld = cf->cfineld;

         /* calculate elder's contribution to family net income */
         cfempinc = ineld->id.idiemp;


         if (cf->cfspoflg) {
               DEBUG1("%s spouse present\n");
               inspo = cf->cfinspo;    /* spouse's in pointer */

               /* add spouse's net income to family net income */
```

```
                cfempinc += inspo->id.idiemp;
        }

        nceitc= 0;

        /* process children in census family */
        for (ini=0, in=cf->cfinch;  ini<cf->cfnchild; ini++, in++) {

                if (in->id.idage >= 21) {
                   DEBUG2("%s discarding old child, aged %d\n", in-
>id.idage);
                   continue;
                }

                /* Count up remaining children */
                nceitc++;
        }

        eitc = 0;
        if (nceitc > 0 ) {
              if ( cfempinc < 8000 ) {
                        eitc = .15 * cfempinc;
              }
              else if ( cfempinc <= 12000 ) {
                        eitc = 1200;
              }
              else if ( cfempinc < 24000) {
                   eitc = 1200 - ((cfempinc - 12000) * .10);
              }
              else {
                     eitc = 0;
              }
           }

       /* process persons in census family */
       for (ini=0, in=cf->cfin;  ini<cf->cfnpers;  ini++, in++) {

                if (in->id.idage > 20) {
                      in->im.imiosa = eitc;
                }
           }
     }

     DEBUG_OFF("Agai");
```

La logique de base du code consiste à traiter chaque famille de recensement individuellement. Dans un premier temps, on identifie le chef de la famille de recensement et on additionne son revenu gagné à la variable temporaire cfempinc (revenu gagné de la famille de recensement). Puis, on détermine s'il y a un conjoint ou une conjointe et on additionne son revenu gagné à la variable cfempinc. Ensuite, on doit créer une boucle pour dénombrer les enfants de la famille de recensement âgés de 0 à 20 ans. Pour les familles qui comptent au moins un enfant âgé de moins de 21 ans, on calcule le montant de prestation sur le revenu gagné. Pour les familles dont le revenu gagné est inférieur à 8 000 $, la prestation équivaut à 15% du revenu gagné. Les familles dont le revenu est supérieur à 8 000 $ mais inférieur ou égal à 12 000 $ reçoivent une prestation maximale de 1 200 $. Les familles dont

le revenu est supérieur à 12 000 $ mais inférieur à 24 000 $ reçoivent une prestation réduite égale au montant maximum de 1 200 $, moins 10% du revenu gagné au delà de 12 000 $. Enfin, les familles dont le revenu gagné est supérieur à 24 000 $ ne sont pas admissibles au crédit d'impôt sur les revenus salariaux et on leur attribue une valeur de 0. On exécute une dernière boucle sur les personnes de la famille de recensement pour attribuer la valeur du crédit d'impôt sur les revenus salariaux de la famille à chaque personne âgée de 21 ans ou plus dans les familles qui comptent des enfants âgés de moins de 20 ans.

Une fois les modifications du fichier **`Agai.cpp`** menées à terme, le travail de l'utilisateur est essentiellement terminé. Toutes les modifications de substance et d'étiquetage pertinentes sont terminées et, si aucune erreur n'a été commise durant leur entrée, il ne reste qu'à compiler et valider le nouveau modèle. Le plus important, toutefois, demeure le fichier exécutable produit par la compilation de C++, dans cet exemple le fichier **`GLASSEX1.EXE,`** que l'utilisateur exécute pour analyser les incidences du changement modélisé.

ESSAI DU MODÈLE DE RECHANGE RÉSULTANT DE LA BDSPS

Il y a deux façons pour les utilisateurs d'exécuter le modèle en boîte de verre.

1. MSPS Visuel – L'interface de ce système est conçue pour permettre la lecture du code du modèle de rechange – lorsque l'utilisateur ouvre l'interface visuelle et choisit d'exécuter une nouvelle simulation, il peut sélectionner l'option de boîte de verre et naviguer vers le sous-répertoire où se trouve le fichier exécutable concerné. Il peut également choisir d'exécuter un scénario de base et (ou) de variante en utilisant soit les modules de la BDSPS standard et (ou) le modèle en boîte de verre.

2. MSPS Classique – Les utilisateurs qui sont à l'aise d'utiliser le MSPS Classique peuvent exécuter les modèles en boîte de verre essentiellement de la même manière en substituant simplement le nom du modèle exécutable (et le chemin de répertoire, au besoin) dans le fichier de traitement en différé.

Les utilisateurs sont invités à consulter le document *Comment exécuter le guide du MSPS* pour plus de détails.

Une fois les modifications apportées, le fichier résultant compilé et les liens établis de manière à créer le nouveau fichier exécutable, nous sommes prêts à tester le nouveau modèle. Les deux buts connexes de cette étape sont les suivants :

1. Faire la preuve que la modification souhaitée est réussie, et

2. Produire les sorties qui aideront à diagnostiquer les erreurs, le cas échéant.

Un moyen tout à fait naturel d'établir la preuve consiste à utiliser les tableaux croisés d'une exécution comparative entre le système d'imposition/transfert non modifié (système de base) et la forme modifiée (système de variante). Plus loin dans la présente section vous trouverez des exemples de deux tableaux croisés de ce type.

Pour effectuer l'exécution comparative du nouveau modèle et obtenir la sortie dont nous avons besoin, il faut modifier les paramètres de commande du modèle. Le *Guide des paramètres* donne une description complète des paramètres de commande de la BDSPS; à ce stade-ci, nous dressons simplement la liste des valeurs des paramètres clés utiles aux fins de l'exemple. (La partie « `glassx1a` » des deux noms de fichier est l'acronyme de « Glass box example 1, version a » (exemple boîte de verre 1, version a).)

```
OUTCPR glassx1a.cpr  # Name of control parameter file (out)
VARALG EITC Quick Start    # Name of variant algorithm
VARMETH       3      # Method of creating variant variables
BASMETH       2      # Method of creating base variables
OUTTBL       glassx1a.tbl         # Name of report file (out)
```

Remarquez, si on utilise le MSPS Classique, qu'il est extrêmement important de fournir le paramètre de commande qui permet d'exécuter l'algorithme de rechange en donnant la valeur 3 à la variable VARMETH.

Quelques tableaux suffisent à valider cet exemple :

1. Totaliser le montant de prestation de crédit d'impôt sur les revenus salariaux, le nombre de ceux qui bénéficient d'un changement positif de leur revenu disponible compte tenu des scénarios de base et de variante, et le nombre de personnes non touchées par la nouvelle prestation (aucune différence entre les revenus disponibles de base et de variante), par groupes d'âges, pour confirmer que les prestations sont versées uniquement aux personnes concernées, et

2. Totaliser, par groupe de revenus d'emploi, la prestation moyenne de crédit d'impôt sur les revenus salariaux de ceux qui y ont droit pour confirmer que nous accordons le nouveau crédit d'impôt aux seuls types d'unité concernés, et

3. Produire un total similaire aux deux premiers par groupe de revenus disponibles de base.

Si on utilise l'analyse de l'exemple de la session 3, légèrement modifié puisque la variable imiosa est déjà incluse dans le revenu disponible, les instructions UVAR de cette validation ressemblent à ce qui suit :

```
gainer = @immdisp > 0;
label(gainer) = "Received EITC Flag (Gainer)";
nochange = (@immdisp==0);
label(nochange)="Unaffected by EITC Flag";
agegrp=split(idage,20,64);
label(agegrp)="Age";
empigrp=split(idiemp,0,8000,12000,24000);
dispgrp=split(_immdisp,5000,10000,15000,20000,25000,
                                  30000,35000,40000,45000);
label(dispgrp)="Base disposable income group";
```

Remarquez que les résultats de la boîte de verre ont été produits avec une version antérieure du modèle de la BDSPS.

Le paramètre XTSPEC qui sert à produire ces tableaux ressemble à ce qui suit :

```
XTSPEC
IN:{imiosa, gainer:S=3, nochange:S=3} * agegrp+;
IN:empigrp+ * {imiosa, imiosa/gainer:L="Average Benefits",
               gainer:S=3, nochange:S=3};
IN:dispgrp+ * {imiosa, gainer:S=3, nochange:S=3, scfrecs};
```

Les principales composantes de cette demande sont les suivantes :

(Compte tenu que la validation des résultats est semblable à celle de l'exemple de la session 3 du guide ***Introduction et aperçu***, nous abrégeons l'explication des tableaux.)

1. Le premier tableau confirme que le programme a été mis en oeuvre correctement puisqu'aucune prestation n'est payée aux enfants de 20 ans et moins.

2. Le tableau 2U montre que le crédit d'impôt, comme prévu, n'est pas versé aux personnes dont le revenu d'emploi est supérieur à 24 000 $ et que certaines personnes qui n'ont gagné aucun revenu reçoivent effectivement le crédit puisqu'elles appartiennent à une famille de recensement admissible (présence d'enfants) et que le revenu gagné du chef et (ou) du conjoint ou de la conjointe est inférieur à 24 000 $.

3. Le troisième tableau totalise simplement la répartition des prestations de crédit d'impôt sur les revenus salariaux par personnes et par groupe de revenus disponibles. Les personnes dont le revenu disponible est élevé et qui reçoivent le montant de crédit d'impôt sur les revenus salariaux tirent des revenus d'autres sources que l'emploi.

Les tableaux produits par l'exécution du nouveau modèle `GLASSEX1` sont les suivants :

```
Table 1U: Selected Quantities for Individuals by Age
```

Quantity	Age			
	Min-20	21-64	65-Max	All
Other SA or guarantees (M)	0.0	804.1	3.7	807.9
Received EITC Flag (Gainer) (000)	0.0	1226.7	7.4	1234.0
Unaffected by EITC Flag (000)	8146.0	17946.5	4184.0	30276.6

```
Table 2U: Selected Quantities for Individuals by Wages & salaries Group
```

Wages & salaries Group	Other SA or guarantees (M)	Average Benefits	Received EITC Flag (Gainer) (000)	Unaffected by EITC Flag (000)
Min-0	201.0	641.2102	313.4	14751.3
1-8000	246.1	594.5951	413.9	3191.6
8001-12000	143.0	1006.2308	142.1	1073.4
12001-24000	214.6	597.1696	359.3	2556.5
24001-Max	3.2	608.0466	5.2	8703.8
All	807.9	654.6415	1234.0	30276.6

```
+-----------+-------------+-------------+-------------+-------------+
```

Table 3U: Selected Quantities for Individuals by Base disposable income group

Base disposable income group	Other SA or guarantees (M)	Received EITC Flag (Gainer) (000)	Unaffected by EITC Flag (000)	SLID Records
Min-5000	73.7	139.3	9226.9	19744
5001-10000	94.3	151.2	2547.6	5722
10001-15000	136.1	202.1	2987.8	7607
15001-20000	173.8	237.2	2987.4	8349
20001-25000	163.0	241.8	2440.6	6443
25001-30000	72.7	118.2	2309.0	5650
30001-35000	39.2	59.9	1888.1	4544
35001-40000	17.5	27.8	1421.2	3378
40001-45000	9.5	12.0	1041.9	2378
45001-Max	28.2	44.5	3426.1	7311
All	807.9	1234.0	30276.6	71126

À partir des valeurs contenues dans ces tableaux, nous concluons que les modifications apportées ci-dessus ont fort probablement réussi à produire les résultats recherchés.

Le test décrit ci-dessus met un terme à l'exemple de démarrage rapide. Compte tenu de l'objectif visé, nous n'avons peut-être pas été aussi précis et méthodique dans notre approche que nous aurions dû l'être dans le cas d'une application réelle. Aussi, nous mentionnons brièvement un certains nombre d'étapes que nous aurions pu choisir pour mettre en œuvre le changement hypothétique.

Nous aurions pu inclure des commentaires de type « historique de révision » dans les fichiers `Adrv.cpp` et `Agai.cpp` pour documenter la nature des modifications et les raisons qui ont motivé la manière dont nous les avons mises en œuvre. Cette forme de documentation est une composante d'une saine pratique professionnelle en matière de développement et de maintenance de logiciel.

Nous aurions pu créer un paramètre pour la valeur du montant maximum de crédit d'impôt sur les revenus salariaux montant (1 200 $) pour le cas où nous aurions voulu répéter ultérieurement l'analyse avec une valeur de crédit d'impôt différente. Nous aurions également pu créer un paramètre pour les limites d'âge des enfants d'une famille de recensement qui déterminent l'admissibilité au montant de crédit d'impôt. On aurait pu appliquer la même logique pour les paramètre de taux et de points tournants de revenu.

De manière générale, le style de modification et le niveau de test effectués ici sont appropriés compte tenu des objectifs restreints de cet exemple. Toutefois, dans le cas d'une

application en boîte de verre plus sérieuse, l'utilisateur voudra probablement apporter les modification requises de manière plus méthodique et accorder plus d'attention aux aspects telle la documentation, l'étiquetage, la validation et l'efficacité des calculs.

RÉSUMÉ

Ce chapitre propose une description de premier niveau des applications en boîte de verre de la BDSPS, incluant une illustration axée sur un exemple particulier. Les différentes sections abordent des sujets telle la modification des calculs substantiels d'un nouveau crédit d'impôt sur les revenus salariaux, la modification de la fonction pilote de la BDSPS qui coordonne le calcul des impôts et des transferts, et l'utilisation du compilateur C++ pour créer une nouvelle version du modèle. Une courte section sur la validation illustre la production de tableaux pour évaluer la réussite du changement.

BDSPS et structure du répertoire de la boîte de verre

De prime abord, un commentaire d'ordre général s'impose : **L'UTILISATEUR NE DOIT D'AUCUNE FAÇON MODIFIER LE CONTENU DES SOUS-RÉPERTOIRES DE LA BDSPS CI-DESSOUS**. (1) Les applications en boîte de verre requièrent de toujours travailler avec des **COPIES** de certains des fichiers de ces sous-répertoires. (2) L'utilisateur effectue tous ses travaux dans l'un des **SOUS-RÉPERTOIRES DISTINCTS** qu'il a créés pour stocker les fichiers de travail destinés aux applications en boîte de verre.

DEFS — Ce sous-répertoire contient un certain nombre de fichiers d'en-tête qui définissent les structures et les constantes utilisées dans l'ensemble de la BDSPS. L'utilisateur s'intéressera surtout au fichier `vs.h` qui définit la structure hiérarchique des données qui contient l'information sur les ménages et les personnes. Il faut toutefois se rappeler que l'utilisateur ne doit jamais modifier cette structure. Pour ajouter les variables qu'il définit, il doit utiliser une COPIE du fichier `vsu.h`.

EXAMPLE — Ce sous-répertoire contient divers fichiers d'INCLUSION qui servent à préciser les paramètres utilisés pour les exemples d'exécution décrits dans la partie didactique du document *Introduction et aperçu*. Ces fichiers peuvent être utiles pour vérifier si l'installation de la BDSPS a été réussie et pour apprendre comment utiliser les modèles déjà développés; toutefois, ils ne sont pas particulièrement pertinents pour le développement des modèles en boîte de verre et nous n'en tenons pas compte dans la présente discussion.

GLASS — Ce sous-répertoire contient les gabarits de départ dont l'utilisateur se servira pour rédiger le code de création des modèles et des systèmes d'imposition/transfert de variante. (1) Il contient le code source de toutes les fonctions fiscales et de prestation de la BDSPS; l'utilisateur trouvera probablement plus efficace de créer les nouvelles fonctions dont il a besoin en modifiant des COPIES de ces éléments. (2) Il contient les fonctions qui permettent de mettre à la disposition de l'ensemble de la BDSPS les variables et les paramètres définis par l'utilisateur, incluant les fichiers d'en-tête connexes qui définissent les structures qui servent à conserver ces

paramètres et variables.

MODEL Ce sous-répertoire contient des exemples de définition de paramètres et de variables de modèle. Les éléments qu'il contient doivent être utilisés SEULEMENT comme exemples concrets de définitions pour les nouvelles variables et les nouveaux paramètres des applications en boîte de verre. L'utilisateur ne doit jamais modifier le contenu de ces fichiers, ni même en utiliser ou en modifier les copies.

WIN32 Ce sous-répertoire contient quelques « fichiers d'exécution de commande » 32 bits de Windows qui régissent la forme de la structure de recouvrement utilisée par la BDSPS. De manière très générale, ces éléments sont semblables à ceux de la bibliothèque LIB; ils sont requis par la fonction SPSMGL.sln, qui est en mesure de les utiliser pour compiler une nouvelle version du modèle. Il contient aussi certains fichiers exécutables utilisés pour la modification de la BDSPS dans un projet.

Dans l'exemple de démarrage rapide, les fichiers `ADRV.CPP` et `AGAI.CPP` sont les fichiers de code source C++ copiés du sous-répertoire `GLASS puis modifiés` pour refléter la logique du nouveau programme; leurs contreparties `OBJ` sont les fichiers objet produits comme sorties lors de la compilation des fichiers « .CPP » dans WINDEBUG et WINREL. Les fichiers `GLASSEX1.EXE` et `GLASSEX1.ncb` ont été créés par la commande de compilation.

L'information essentielle de ce chapitre peut se résumer comme suit :

1. **L'utilisateur de la BDSPS ne doit d'aucune façon modifier le contenu du sous-répertoire `SPSM` ni l'un ou l'autre de ses sous-répertoires créés pendant l'installation de la BDSPS**.

2. L'utilisateur crée des sous-répertoires « de travail » distincts pour les applications en boîte de verre. Il est préférable que ces sous-répertoires ne soient pas des sous-répertoires de la BDSPS.

3. **L'utilisateur de la boîte de verre copie les éléments pertinents du répertoire `SPSM\GLASS`, qu'il utilise comme gabarits pour effectuer les modifications.** Les modifications elles-mêmes sont effectuées dans ces COPIES. Les sections suivantes du *Guide de programmation* indiquent de manière très détaillée ce que l'utilisateur doit modifier ainsi que l'endroit où se trouvent les gabarits concernés.

4. Les sous-répertoires clés `\SPSM\DEFS` doivent être ajoutés dans **Tools: Options: Directory** où se trouvent les définitions pertinentes des applications.

Structure particuliers-ménages de la BDSPS

Le présent chapitre vise trois buts principaux, abordés dans des sections distinctes, qui traitent tous de manière générale de la structure des données de la BDSPS et de son utilisation.

La première section donne un aperçu du cadre de travail de la BDSPS lié au stockage des données sur les ménages, les familles et les personnes. Il est essentiel que l'utilisateur de la boîte de verre connaisse bien cette structure pour être en mesure de modifier les valeurs des variables modélisées et des variables de données existantes, ou de s'y référer, ainsi que pour créer, le cas échéant, de nouvelles variables jugées nécessaires à la production d'une version personnalisée de la BDSPS.

La deuxième section donne des explications sur l'utilisation des variables pointeurs comme principal outil permettant à l'utilisateur d'accéder à des éléments particuliers des données. Elle décrit également les principales règles d'attribution de noms utilisées pour les applications en boîte de verre. Ces sujets intéressent à la fois les utilisateurs qui créent leurs propres applications en boîte de verre et ceux qui cherchent à comprendre les algorithmes standard de la BDSPS. La « philosophie » sous-jacente de ce développement respecte l'orientation générale du présent guide; à plusieurs égards, il est beaucoup plus important pour l'utilisateur de la boîte de verre de maîtriser, mécaniquement, la façon d'effectuer son travail de manière robuste et dans le respect des normes que de comprendre tous les motifs conceptuels qui sous-tendent les structures et les techniques. Autrement dit, l'orientation de la section est résolument pratique et met davantage l'accent sur la mécanique du « comment faire » que sur les détails du « pourquoi ».

La troisième section inclut un « bestiaire » de fragments de code servant à exécuter les tâches courantes en boîte de verre, particulièrement celles qui ont trait aux structures de données. En fait, l'utilisateur doit non seulement être en mesure de pouvoir reproduire une solution existante plutôt que de la réinventer, mais il doit également s'assurer qu'elle respecte les normes et ne nécessite aucune mise au point. Les fragments de code offerts dans la section incluent les suivants : a) traitement des familles et des personnes appropriées par des instructions « for », b) référence à d'autres membres de la famille, c) accès à la base de données et aux variables modélisées existantes et d) attribution de nouvelles valeurs à ces variables.

STRUCTURE DES DONNÉES DE LA BDSPS

La BDSPS est un fichier ordonné fixe dont l'utilisateur ne peut trier le contenu. Il est essentiel de comprendre l'ordre de tri de la base de données lorsque l'on tente de traiter les ménages en boucles. La base regroupe les ménages en grappes triées au hasard de façon stratifiée. Chaque ménage est trié comme suit :

Ménage

    Familles économiques

        Familles de recensement

            Familles nucléaires

                Chefs de famille

                Conjoint, le cas échéant

Enfants, des plus jeunes aux plus âgés

À l'intérieur d'un ménage, les personnes sont regroupées en familles économiques. À l'intérieur d'une famille économique, les personnes sont regroupées en familles de recensement. À l'intérieur de la famille de recensement, les personnes sont regroupées en familles nucléaires. À l'intérieur de la famille nucléaire, le chef figure toujours en premier, suivi du conjoint, le cas échéant. Les enfants suivent ensuite et sont triés selon leur âge.

Chaque ménage complet est versé dans la structure de données précisée ci-dessus. Les boucles peuvent ensuite être établies de manière à permettre le traitement de l'une ou l'autre des unités d'analyse à l'intérieur d'un ménage.

Le *Guide des variables* inclut des descriptions détaillées de la substance des variables individuelles de la BDSPS . La majorité des détails sur le contenu des nombreuses structures se trouve dans le fichier `vs.h`. Les éléments clés nécessaires à la définition des variables se trouvent dans le fichier `spsm.h`. Certaines des macros permettent à l'utilisateur d'intervenir symboliquement pour clarifier leur signification, ou pour assurer l'uniformité de la précision numérique :

```
#define LOGICAL int     /* type used to store true or false values       */
#define TRUE  1         /* manifest constants to make code more readable */
#define FALSE 0
#define NUMBER float
#define ZERO (float) 0.0
#define HALF (float) 0.5
#define ONE (float) 1.0
#define THOUSAND (float) 1000.0
#define MILLION (float) 1000000.0
```

PRÉSENTATION DES POINTEURS DANS LA BDSPS

La structure uv est une structure dont le contenu est défini par l'utilisateur, tant au plan de la substance que des noms de variables. Un chapitre décrit la façon dont l'utilisateur crée de nouvelles variables, par exemple, pour définir un nouveau programme d'imposition/transfert. L'utilisateur contrôle le contenu de la structure avec le fichier d'en-tête `vsu.h et le fichier vsdu.c` et peut modifier toutes les valeurs des éléments définis dans le fichier `Adrv.cpp. Ces capacités d'attribution et de définition sont au cœur même des applications en boîte de verre auxquelles l'utilisateur doit ajouter de nouvelles variables. Évidemment, pour que le cumul fonctionne correctement dans l'ensemble de la BDSPS, l'utilisateur doit prendre soin d'attribuer les nouvelles variables et les nouveaux impôts aux bonnes personnes.`

Le langage C++ fait une grande utilisation des variables pointeurs, variables qui pointent vers un secteur particulier de la mémoire et, plus précisément, vers une structure de données spécifique. Les parties du code source de la BDSPS qui traitent des algorithmes d'imposition/transfert utilisent moins les pointeurs et les règles mathématiques qui les gouvernent que les parties auxquelles l'utilisateur n'a pas accès; toutefois, l'utilisateur de la boîte de verre doit néanmoins se servir de ces pointeurs. L'emploi de ces outils est essentiel et la BDSPS a été conçue de manière à faciliter au maximum leur utilisation par les

concepteurs. Le produit offre une variété de fragments de code et de macros qui permettent une utilisation plus simple et plus mécanique. La section bestiaire décrit brièvement la façon d'appliquer ces pointeurs aux tâches courantes en boîte de verre, telles les boucles et les références. N'oubliez pas, cependant, que la section ne constitue d'aucune manière un cours exhaustif sur l'utilisation générale des pointeurs ailleurs que dans la BDSPS.

BESTIAIRE

Un bestiaire est une « collection de descriptions d'animaux réels ou imaginaires ». Les « animaux » dont il est question, et qui sont décrits ici, sont réels. Il s'agit de fragments de code source en langage C++ susceptibles d'être utiles à l'utilisateur de la boîte de verre lors de la lecture et de l'écriture de code pour les programmes d'imposition/transfert. Les fragments décrits ici ont tous été versés dans la base de code des algorithmes de la BDSPS (dans le répertoire `GLASS`) afin que l'utilisateur puisse copier les segments sans les retaper.

Les éléments de bestiaire fournis aident à soutenir l'approche proposée dans l'ensemble du guide. En effet, on ne doit pas s'attendre à ce que les utilisateurs réinventent la roue; ils doivent recevoir toute l'aide possible pour tirer avantage de ce que la BDSPS peut déjà leur offrir. Ils peuvent copier et éventuellement modifier le code existant pour bénéficier des quatre principaux avantages suivants :

1. Le code source existant est réputé exact et ne doit donc pas être débogué.

2. Il est possible d'obtenir une plus grande uniformité entre le code de l'utilisateur et celui livré avec la BDSPS.

3. Copier le code est beaucoup plus rapide que de le retaper.

4. Souvent, l'utilisateur peut exécuter le travail requis sans être tenu de comprendre tous les détails sous-jacents. Il lui suffit de respecter le format général, c'est-à-dire utiliser une en-tête, suivie du code lui-même et, parfois, d'une courte explication ou d'un bref commentaire.

Exemples de boucles

Une des tâches les plus couramment utilisées pour lire, modifier ou rédiger du code est la création de boucles à l'intérieur des unités pertinentes d'un ménage ou de l'une de ses structures. Le jeu de segments de code source ci-dessous, probablement le jeu le plus complet qui permet à l'utilisateur d'effectuer les boucles dont il a besoin, inclut les définitions pertinentes nécessaires. Par exemple, dans le premier exemple, l'utilisateur doit déclarer le pointeur « in » du type « P_in, » et le nombre entier « ini » qui pourront servir au fonctionnement de la boucle (itération). De manière générale, les définitions figurent dans le code source, avant la boucle elle-même.

```
/** * PROCESS ALL INDIVIDUALS IN HOUSEHOLD hh **/

register P_in in;
int ini;

for    (ini=0, in=&hh->in[0]; ini<hh->hhnin; ini++, in++) {
       DEBUG2("%s processing individual %d in household\n", ini);
```

```
	/* code here, using pointer 'in' */
	}
```

Dans la boucle ci-dessus, et dans celles qui suivent, on utilise l'instruction « for » du langage C++. Les éléments qui précèdent le point virgule initial initialisent les variables de l'itération. La condition entre les deux points-virgules précise quand la boucle doit se poursuivre. Les éléments à l'intérieur des parenthèses, après le deuxième point-virgule, spécifient l'incrémentation nécessaire pour l'itération suivante. Dans le fragment de code se trouve également le commentaire « code here ». Il indique l'endroit où insérer le code de la BDSPS, ou celui de l'utilisateur, pour agir sur l'unité à l'intérieur de laquelle se produit le cycle d'itération. Ce commentaire indique également le pointeur que la boucle commande à l'intérieur de l'unité.

```
/*** PROCESS ALL INDIVIDUALS IN ECONOMIC FAMILY ef   **/

	register P_in in;
	int ini;     for (ini=0, in=ef->efin; ini<ef->efnpers; ini++, in++) {
	DEBUG2("%s processing individual %d in economic family\n", ini);
	/* code here, using pointer 'in' */
	}

	/*** PROCESS ALL INDIVIDUALS IN CENSUS FAMILY cf     **/

	register P_in in;
	int ini;
	for (ini=0, in=cf->cfin; ini<cf->cfnpers; ini++, in++) {
	DEBUG2("%s processing individual %d in census family\n", ini);
	/* code here, using pointer 'in' */
	}

	/*** PROCESS ALL CHILDREN (including 18+) IN CENSUS FAMILY cf     **/

	register P_in in;
	int ini;
	for (ini=0, in=cf->cfinch; ini<cf->cfnchild; ini++, in++) {
	DEBUG2("%s processing child (including 18+) %d in census family\n", ini);
	/* code here, using pointer 'in' */
	}

	/*** PROCESS YOUNG CHILDREN IN CENSUS FAMILY cf      **/

	register P_in in;
	int ini;
	for (ini=0, in=cf->cfinch; ini<cf->cfnkids; ini++, in++) {
	DEBUG2("%s processing child (<18) %d in census family\n", ini);
	/* code here, using pointer 'in' */
	}

	/*** PROCESS ALL INDIVIDUALS IN NUCLEAR FAMILY nf    **/

	register P_in in;
	int ini;
	for (ini=0, in=nf->nfin; ini<nf->nfnpers; ini++, in++) {
	DEBUG2("%s processing individual %d in nuclear family\n", ini);
	/* code here, using pointer 'in' */

	}

	/*** PROCESS CHILDREN IN NUCLEAR FAMILY nf    **/

	register P_in in;
```

```
int ini;
for (ini=0, in=nf->nfinch; ini<nf->nfnkids; ini++, in++) {
DEBUG2("%s processing child %d in nuclear family\n", ini);
/* code here, using pointer 'in' */

}


/*** PROCESS ALL ECONOMIC FAMILIES IN HOUSEHOLD hh   **/

P_ef ef;
int efi;
for (efi=0, ef=&hh->ef[0]; efi<hh->hhnef; efi++, ef++) {
DEBUG2("%s processing economic family %d\n", efi);
/* code here, using pointer 'ef' */

}

/*** PROCESS ALL CENSUS FAMILIES IN HOUSEHOLD hh     **/

P_cf cf;
int cfi;
for (cfi=0, cf=&hh->cf[0]; cfi<hh->hhncf; cfi++, cf++) {
DEBUG2("%s processing census family %d\n", cfi);
/* code here, using pointer 'cf' */

}

/*** PROCESS ALL NUCLEAR FAMILIES IN HOUSEHOLD hh    **/

P_nf nf;
int nfi;
for (nfi=0, nf=&hh->nf[0]; nfi<hh->hhnnf; nfi++, nf++) {
DEBUG2("%s processing nuclear family %d\n", nfi);
/* code here, using pointer 'nf' */

}
```

Références relatives à une personne

Une autre tâche couramment utilisée en boîte de verre concerne les références à d'autres personnes dans une structure ou une sous-structure, ou à des unités d'analyse de rang « plus élevé » dans la structure. Ces références permettent à l'utilisateur de faire référence aux caractéristiques telle la province de résidence d'une personne, le revenu du conjoint du membre le plus âgé de la famille de recensement (le cas échéant), ou l'âge du deuxième enfant le plus âgé vivant dans une famille de recensement, à l'intérieur d'une famille économique commune.

```
/*** REFERENCE SPOUSE OF INDIVIDUAL in  **/

if (in->id.idspoflg) {
P_in inspo;
inspo = in->id.idinspo;
/* code here, using pointer 'inspo' */
}
```

```
Notice here that there will not always exist a spouse.
```

```
/*** REFERENCE HOUSEHOLD OF INDIVIDUAL in     **/

P_hh hh;
hh = in->id.idhh;
```

```
/* code here, using pointer 'hh' */
```

Lorsqu'il a récupéré le pointeur vers le ménage, l'utilisateur peut ensuite accéder aux caractéristiques du ménage telle la province de résidence. Contrairement à la situation où il est question du conjoint d'une personne, il existe toujours un ménage.

```
/*** REFERENCE ECONOMIC FAMILY OF INDIVIDUAL in      **/
P_ef ef;
ef = in->id.idef;
/* code here, using pointer 'ef' */
```

De la même manière, la famille économique de la personne existe toujours et permet de vérifier si la personne vit dans une unité en dessous du seuil de pauvreté.

```
/*** REFERENCE CENSUS FAMILY OF INDIVIDUAL in  **/
P_cf cf;
cf = in->id.idcf;
/* code here, using pointer 'cf' */
/*** REFERENCE NUCLEAR FAMILY OF INDIVIDUAL in       **/
P_nf nf;
nf = in->id.idnf;
/* code here, using pointer 'nf' */
```

Ces références clés, combinées aux fragments d'itération mentionnés à la section précédente, permettent à l'utilisateur d'effectuer relativement sans inconvénient presque toutes les tâches jugées nécessaires à la simulation d'un système d'imposition/transfert.

RÉSUMÉ

La première partie du présent chapitre décrit la structure des données utilisées dans la BDSPS. Elle indique aussi les macros de fonction et les constantes manifestes les plus importantes que l'utilisateur peut trouver dans le code source de la BDSPS. Les parties suivantes décrivent le rôle des variables pointeurs et établissent les caractéristiques des principaux types de pointeurs utilisés. Enfin, la dernière partie inclut un bestiaire de fragments de code utilisés pour les tâches les plus courantes en mode boîte de verre, soit l'utilisation de boucles dans les unités de personnes et de familles et les références au conjoint d'une personne ou aux unités d'analyse qui le concerne.

Le prochain chapitre utilise ces éléments de base et décrit la façon dont la BDSPS traite les ménages pour calculer les impôts et les transferts. Cette description sert ensuite de base aux derniers chapitres où il est question des méthodes qui permettent d'ajouter des variables et des paramètres définis par l'utilisateur pour modifier la logique du système d'imposition/transfert.

Structure d'appel des fonctions de la BDSPS

Le calcul des impôts et des transferts en espèces d'un ménage est contrôlé par une fonction dont la seule tâche consiste à appeler toutes les autres fonctions individuelles de l'algorithme d'imposition/transfert. L'ordre des appels est critique pour la simulation compte tenu des besoins d'information des fonctions. Par exemple, il faut connaître le revenu net avant de pouvoir calculer le SRG. La liste suivante inclut les fonctions appelées par les paramètres `drv` et `adrv`, dans l'ordre où elles sont appelées.

Fonction	Description
ui(hh)	Calcul des prestations d'assurance-emploi
famod(hh)	Calcul des allocations familiales
oas(hh)	Calcul de la sécurité de vieillesse
dem(hh)	Calcul des nouvelles subventions démographiques
txinet(hh)	Calcul du revenu net
gis(hh)	Calcul du supplément de revenu garanti pour les personnes âgées
gist(hh)	Calcul du supplément provincial de revenu pour les personnes âgées
samod(hh)	Calcul de l'aide sociale
txitax(hh)	Calcul du revenu imposable
txhstr(hh)	Calcul des déductions pour enfants et conjoint
txcalc(hh)	Calcul de l'impôt fédéral
txctc(hh)	Calcul du crédit d'impôt pour enfants
txfstc(hh)	Calcul du crédit fédéral sur la taxe de vente
txprov(hh)	Calcul des crédits des impôts provinciaux
gai(hh)	Calcul de nouveaux crédits remboursables, de nouvelles garanties
memo1(hh)	Calcul du revenu disponible, etc.
ctmod(hh)	Calcul des taxes à la consommation et attribution aux personnes
memo2(hh)	Calcul du revenu consommable, etc.
cceopt(hh, drv)	Valeur zéro pour les frais de garde d'enfants pour les jeunes enfants, si cela se révèle optimal
classu(hh)	Calcul des variables d'établissement de rapports définies par l'utilisateur

L'ordre d'appel des fonctions d'éléments de `drv` reflète l'ordre logique entre eux.

- Les premières fonctions, ui, famod et oas, simulent des programmes dont les prestations sont déterminées par des facteurs autres que le revenu; elles sont donc appelées en premier.
- dem est un sous-programme d'application en boîte de verre qui exige l'exécution des calculs avant l'entrée dans les programmes de système d'imposition.
- txinet calcule le revenu net avant certains transferts.
- gis calcule les transferts aux personnes âgées.
- samod calcule l'aide sociale ou les transferts de revenus garantis.
- Les impôts fédéraux et provinciaux sont ensuite calculés par les quatre fonctions suivantes dont le préfixe est tx (txitax, txhstr, txcalc et txprov).
- gist, txctc et txfstc calculent les programmes de transfert en fonction du revenu.
- gai est un autre sous-programme destiné aux utilisateurs de la boîte de verre qui désirent simuler des options qui requièrent de l'information sur tous les impôts sur le revenu et tous les transferts en espèces personnels. Par exemple, l'utilisateur peut utiliser cette fonction pour simuler un programme de supplément de revenu.
- Les fonctions memo1 et memo2 créent des variables agrégées pour les fins de rapport.
- Dans la fonction ctmod, la taxe d'accise et la taxe de vente sont calculées par l'application des taux réels de taxe de vente basés sur les entrées et les sorties de dépenses des familles observées.
- cceopt optimise le revenu en optimisant le crédit pour frais de garde d'enfants et le crédit

pour enfants.

- classu est un sous-programme qui permet à l'utilisateur de calculer et d'attribuer des valeurs à des variables nouvelles ou redéfinies.

Les fonctions appelées par le paramètre `drv` appellent d'autres fonctions et sous-fonctions pour terminer leurs calculs. Veuillez consulter la fonction désirée dans le *Guide des algorithmes* pour obtenir une description plus détaillée. Les sous-fonctions peuvent figurer dans la liste de la fonction qui les appelle. Pour bien comprendre le calcul du revenu net, l'utilisateur doit donc consulter les deux fonctions txinet et txccea.

Les sous-fonctions sont définies à l'intérieur de la fonction (fichier) qui les appelle. L'exemple suivant est un appel de la sous-fonction uiclm() dans ui.cpp, où uiclm est définie dans une section de ui.cpp.

```
valid_claim = uiclm(in, &in->id.uc1, in->id.uc1.ucy1, &in-
>im.ub1,
                hh->hd.hdprov, hh->hd.hdurb, wctb, in-
>id.uc1.ucstart);
```

Développement en boîte de verre : ajout de paramètres scalaires ordinaires

Comme son titre l'indique, ce chapitre explique la mécanique des tâches de programmation associées à l'ajout de paramètres scalaires ordinaires pendant le développement d'applications en boîte de verre. Au plan de la structure, le chapitre communique cette information à l'aide d'un exemple pratique détaillé. La première section revoit la procédure générale utilisée pour le développement des applications et décrit les étapes essentielles à toute modification de modèle, que ce soit la modification de codes ou l'ajout de paramètres ou de variables. La deuxième section aborde plusieurs mesures préliminaires pour l'ajout de paramètres. Elle décrit aussi la nature de l'exemple proposé, soit une introduction à l'exemple portant sur le crédit d'impôt sur les revenus salariaux utilisé dans le chapitre Démarrage rapide du Guide. Les sections suivantes utilisent ensuite cet exemple pour expliquer en détail les étapes qui permettent d'ajouter à un modèle les types de paramètres scalaires les plus courants. Enfin, la dernière section résume les principaux points concernant l'ajout de ces paramètres de formes les plus courantes.

PROCÉDURE GÉNÉRALE DE MODIFICATION EN BOÎTE DE VERRE : RÉCAPITULATION

La section précédente décrit la procédure générale de développement d'applications en boîte de verre, incluant la logique sous-jacente aux différentes étapes. Nous résumons ici les principaux points :

- Créer un sous-répertoire de travail

- Déterminer les fichiers à modifier

- Copier les fichiers pertinents dans le sous-répertoire de travail et supprimer les propriétés de « lecture seule »

- Modifier les fichiers pertinents

- Compiler la nouvelle version

- Tester la nouvelle version du modèle

- Effectuer les analyses prévues

Créer un sous-répertoire de travail

L'utilisateur crée un nouveau « sous-répertoire de travail » pour conserver les fichiers concernés de la nouvelle application en boîte de verre. Il modifie les fichiers du sous-répertoire de travail en prenant soin de ne modifier aucun autre fichier de la BDSPS.

Déterminer les fichiers à modifier

L'utilisateur détermine les fichiers du répertoire `c:\spsm\glass` pour lesquels il doit créer des variantes. Dans l'exemple de démarrage rapide, nous avons déterminé qu'il s'agissait des fichiers `Agai.cpp, Adrv.cpp, SPSMGL.vcproj` et `SPSMGL.sln`. Dans l'exemple utilisé ici, nous indiquons de quelle manière d'autres fichiers, par exemple, `Mpu.h` et `Ampd.cpp`, participent également à l'ajout de paramètres dans une application en boîte de verre. Une autre section explique en plus comment d'autres fichiers, `Vsu.h` et `Vsdu.cpp`, peuvent intéresser l'utilisateur qui désire ajouter des variables à un modèle. De toute évidence, les fichiers de la fonction d'imposition/transfert qui utilisent les nouveaux paramètres doivent aussi être modifiés. À l'occasion, l'utilisateur pourra juger plus efficace d'utiliser comme gabarits des fichiers déjà développés dans une application antérieure plutôt que de repartir à zéro avec les fichiers de gabarit du sous-répertoire `glass`.

Copier les fichiers pertinents dans le sous-répertoire de travail

L'utilisateur copie tous les fichiers jugés pertinents dans le sous-répertoire de travail. Il travaillera seulement avec des copies, sans toucher aux originaux. Il doit supprimer les propriétés de « lecture seule » des fichiers pour être en mesure de les modifier.

Modifier les fichiers pertinents

L'utilisateur apporte les modifications appropriées à chacun des fichiers jugés pertinents. Nous recommandons d'apporter les changements dans l'ordre suivant :

1. Inclure tous les fichiers pertinents dans le projet et changer le nom du fichier de sortie dans Project: Setting: Link.

2. Modifier le fichier `Adrv.cpp`, au besoin.

3. Modifier les fichiers `Mpu.h et Ampd.cpp`, le cas échéant, pour ajouter de nouveaux paramètres au modèle.

4. Modifier les fichiers `Vsu.h` et `Vsdu.cpp`, le cas échéant, pour ajouter toute nouvelle variable de sortie au modèle.

5. Modifier les fichiers de code source pour ajouter la nouvelle logique au code du système d'imposition/transfert.

Nous suivrons cet ordre dans les exemples présentés ici et dans les sections suivantes.

Compiler la nouvelle version

L'utilisateur doit activer la fonction Debugging dans Build: Set Active Configuration puis effectuer la mise au point du projet. Lorsque les modifications appropriées ont été apportées au programme, le nouveau modèle doit être compilé.

Tester la nouvelle version du modèle

L'utilisateur teste la nouvelle version avec un ensemble d'analyses de validation conçues de manière à révéler tout problème inhérent de la logique ajoutée ou modifiée. Cette étape peut exiger le retour à une version antérieure pour corriger les lacunes décelées.

Effectuer l'analyse prévue

Enfin, une fois la validation terminée, l'utilisateur peut procéder à des « exécutions de production » du nouveau code exécutable afin de simuler les conséquences du changement modélisé.

PRÉSENTATION DE L'AJOUT DES PARAMÈTRES

Cette section aborde quelques éléments préliminaires critiques pour la procédure d'ajout de paramètres scalaires habituels. En premier lieu, elle illustre la raison pour laquelle l'utilisateur peut désirer ajouter un ou plusieurs paramètres à un modèle. En plus, elle décrit la substance des nouveaux paramètres utilisés pour illustrer l'ajout de paramètres ordinaires.

Comme on l'a noté à la fin de l'exemple de démarrage rapide, l'analyste hypothétique a pris quelques raccourcis pour accomplir ce qui peut être fait différemment au cours d'un exercice réel d'élaboration de politiques, plus particulièrement lorsque le nouveau modèle est destiné à être utilisé de façon répétitive ou par plusieurs analystes. Un de ces raccourcis consiste à « implanter » le crédit d'impôt et les niveaux de revenu du crédit d'impôt sur les revenus salariaux dans la fonction `Agai.cpp`. Bien que cela puisse être acceptable si l'utilisateur ne souhaite pas essayer une autre valeur comme montant de prestation, cette approche n'est pas particulièrement efficace s'il désire examiner les incidences d'autres valeurs. L'utilisateur doit modifier de nouveau le code puis recompiler le modèle pour chaque valeur distincte à examiner; par exemple, il peut chercher à vérifier s'il est vrai que les incidences sont habituellement proportionnelles au montant de la prestation et désirer valider cette énoncé en essayant plusieurs valeurs. En ajoutant des paramètres appropriés au modèle, aucune modification supplémentaire n'est nécessaire et l'utilisateur peut étudier les effets de plusieurs valeurs sans recompiler le modèle, simplement en fournissant de nouvelles valeurs de paramètres au modèle modifié.

Les diverses sections du présent chapitre décrivent donc les étapes nécessaires à l'ajout de

paramètres au modèle et illustrent de manière particulière le nettoyage de l'exemple utilisé pour le démarrage rapide. Ce chapitre se limite aux formes les plus couramment utilisées de paramètres scalaires. Nous croyons que les genres d'ajout décrits répondent au moins à 80 % des besoins d'ajouts de paramètres. Nous gardons pour les dernières sections la définition plus ésotérique des paramètres scalaires, des vecteurs et des matrices de paramètres. Quel que soit le type des nouveaux paramètres ajoutés à un modèle, ils deviennent accessibles à toutes les fonctions appelées par `Adrv.cpp` et non seulement aux fonctions d'un programme de transfert spécifique.

Essentiellement, nous ajoutons sept paramètres à une variante du modèle utilisé pour le démarrage rapide. Ces ajouts correspondent aux trois formes les plus courantes de paramètre que l'utilisateur aura l'occasion d'utiliser.

1. Le premier type de paramètre, scalaire « à virgule flottante » ou à valeur « réelle », fournit la valeur du montant maximum du crédit d'impôt sur les revenus salariaux, le seuil de revenu qui définit la conception du crédit et les taux d'instauration progressive et de retrait progressif utilisés pour le calcul du crédit. Nous appelons ces paramètres :

   EITCMAX – Montant maximum du crédit d'impôt sur les revenus salariaux

   EITCPIR – Taux d'instauration progressive du crédit d'impôt sur les revenus salariaux

   EITCPOR – Taux de retrait progressif du crédit d'impôt sur les revenus salariaux

   EITCTPMX – Point tournant du revenu pour un crédit maximum d'impôt sur les revenus salariaux

   EITCTPRC – Point tournant du revenu pour un crédit réduit d'impôt sur les revenus salariaux

2. Le deuxième paramètre, une valeur entière scalaire, indique l'âge d'admissibilité des enfants de la famille recensement et élimine la valeur « 20 » implantée directement dans l'exemple de démarrage rapide. Nous appelons ce paramètre `EITCAGE` (âge des enfants pour l'admissibilité de la famille recensement au crédit d'impôt sur les revenus salariaux).

3. Le troisième paramètre, variable « drapeau » qui sert de commutateur booléen, indique si l'on doit tenir compte des deux premiers paramètres. À ce titre, sa fonction est semblable à celle de nombreuses variables drapeau utilisées dans l'ensemble de la BDSPS. Lorsqu'il est « activé », il permet le calcul du crédit d'impôt; lorsqu'il est « désactivé », le modèle n'effectue aucun calcul associé au nouveau crédit d'impôt sur les revenus salariaux. Nous appelons ce paramètre `EITCFLAG` (drapeau d'activation du crédit d'impôt sur les revenus salariaux).

La description présume que l'utilisateur a choisi d'utiliser comme répertoire de travail, en le créant au besoin, le répertoire `\glassex2`.

COPIER LES FICHIERS ADRV.CPP, MPU.H, AMPD.CPP, AGAI.CPP, SPSMGL.VCPROJ ET SPSMGL.SLN

L'utilisateur copie dans le nouveau sous-répertoire de travail tous les fichiers qui doivent être modifiés. Il voudra également modifier le fichier `Adrv.cpp` pour mettre à jour la description utilisée dans les fichiers de code modifiés (dans ce cas-ci, seulement le fichier `Agai.cpp`). On doit donc copier le fichier `Adrv.cpp.`

Deux autres fichiers, `Mpu.h` et `Ampd.cpp`, demeurent pertinents pour l'ajout d'un nouveau paramètre de modèle. Le fichier `Mpu.h` (paramètres de modèle, utilisateur) est un fichier d'en-tête en langage C qui définit la nature du nouveau paramètre; le fichier `Ampd.cpp` (autres définitions de paramètre de modèle) contient les appels de fonctions qui font connaître les paramètres de l'utilisateur à l'ensemble de la BDSPS afin qu'ils puissent, par exemple, faire l'objet de références et permettre des modifications de valeurs « à la volée » lorsque l'utilisateur exécute le fichier exécutable de la BDSPS.

L'utilisateur doit copier ces fichiers du sous-répertoire `glass`. Par exemple, s'il a déjà modifié ces fichiers ailleurs pour définir d'autres paramètres et désire conserver ces modifications, il peut copier les gabarits de `Mpu.h` et `Ampd.cpp` du sous-répertoire où ils se trouvent. Le terme « gabarits » désigne des fichiers, des éléments de texte ou du code existants, utilisés comme point de départ pour apporter les modifications désirées. Ainsi, il est tout à fait inutile que l'utilisateur crée à partir de zéro une nouvelle version des fichiers pertinents. Dans le présent exemple, nous présumons que ces paramètres sont les premiers que l'on ajoute et nous copions les gabarits du répertoire `glass`.

Pour terminer, l'utilisateur doit copier la ou les fonctions d'imposition/transfert essentielles qui utiliseront le nouveau paramètre. Pour nos fins, la seule fonction vraiment pertinente est la fonction `Agai.cpp`. Plutôt que de la copier du répertoire `glass` et de reprendre ensuite le travail depuis le début, nous la copions du répertoire `glassex1`; il suffit de trouver l'endroit où il faut attribuer l'augmentation.

L'utilisateur doit copier les fichiers SPSMGL.vcproj et SPSMGL.sln qui décrivent l'environnement du projet.

MISE À JOUR DU PROJET

Tous les fichiers requis doivent être inclus dans le projet et le nom du fichier exécutable à la sortie doit être remplacé par glassex2.exe dans Project: Setting: Link to.

MISE À JOUR DE LA DESCRIPTION DE L'ALGORITHME DANS LE FICHIER ADRV.CPP

N'oubliez pas que l'on a attribué, dans l'exemple de démarrage rapide, de nouvelles valeurs aux variables globales `altname[]`et `Tdrv[]`pour identifier et documenter la nature des modifications apportées. À ce stade-ci, compte tenu de la nouvelle version de modèle que l'on a créée, une substitution correspondante s'impose. Les deux substitutions, qui consistent exclusivement à définir le contenu de deux chaînes, produisent le code suivant :

```
=================== GLOBAL VARIABLE DEFINITIONS ============== */
/*global*/ char ALTNAME[IDSIZE+1] = "Parameterized EITC";
```

```
/* Give global string describing version of this module */
/*global*/ char FAR Tdrv[] = "Parameterized EITC"
```

MODIFIER LE FICHIER MPU.H POUR DÉFINIR LES NOUVEAUX PARAMÈTRES

L'utilisateur doit ensuite modifier le fichier Mpu.h afin de définir le type des nouveaux paramètres. Lorsque la modification est apportée dans la version boîte de verre de Mpu.h, la ligne qui contient la chaîne « UMDUMMY » est remplacée par la définition des nouveaux paramètres. Le nom « UMDUMMY » renvoie à « paramètre de modèle d'utilisateur fictif ». Nous appelons le premier nouveau paramètre EITCMAX pour indiquer qu'il s'agit du montant maximum de crédit d'impôt sur les revenus salariaux. Avant le changement, la ligne indiquée (près de la ligne 77) est semblable à ce qui suit :

```
int UMDUMMY;      /* dummy entry                              */
```

Puisque, comme l'indique l'étiquette, cette entrée n'est qu'un paramètre fictif qui permet à la BDSPS de travailler lorsque l'utilisateur n'a pas encore défini de paramètres, nous supprimons cette ligne, que nous remplaçons par les lignes suivantes :

```
NUMBER EITCMAX;    /* Earned Income Tax Credit Maximum Amount  */
NUMBER EITCPIR;    /* Earned Income Tax Credit Phase In Rate    */
NUMBER EITCPOR;    /* Earned Income Tax Credit Phase Out Rate    */
NUMBER EITCTPMX;   /* EITC Turning Point for Maximum Credit */
NUMBER EITCTPRC;   /* EITC Turning Point for Reduced Credit */
int    EITCAGE;    /* Earned Income Tax Credit Child Age for Census Family
Eligibility   */
int    EITCFLAG;  /* Earned Income Tax Credit Activation Flag        */
```

Sur la première ligne, « NUMBER » est une macro utilisée par la BDSPS pour assurer la portabilité d'un ordinateur à l'autre; elle correspond au type « à virgule flottante ». EITCMAX est le nom du nouveau paramètre. Par convention, dans la BDSPS, les noms de paramètres sont en majuscules. Les autres paramètres « NUMBER » définissent les taux d'instauration progressive et de retrait progressif et les points tournants de revenu gagné qui déterminent le niveau de crédit. Les deux autres paramètres sont évidemment des nombres entiers. Aux fins de lisibilité, nous avons également ajouté des commentaires à la droite afin d'indiquer la nature des valeurs de paramètre.

Ces simples ajouts mettent un terme à la modification du fichier Mpu.h. Normalement, lorsque nous ajoutons de nouveaux paramètres à un ensemble de paramètres existants définis par l'utilisateur, il suffit d'ajouter les nouvelles définitions à la fin de la liste actuelle dans le fichier.

La BDSPS prévoit de l'espace pour un maximum de 500 nouveaux paramètres, suffisamment pour les applications types d'utilisateur du mode boîte de verre. Il est possible d'ajouter encore plus de paramètres lorsque certains sont du type « int », moins long. **Toute tentative de dépasser cette limite entraîne un message d'erreur lors de la compilation.**

MODIFIER LE FICHIER AMPD.CPP DE MANIÈRE À RENDRE LES PARAMÈTRES DISPONIBLES À LA BDSPS

L'utilisateur doit également modifier le fichier Ampd.cpp pour rendre le nouveau paramètre accessible aux composantes de la BDSPS qui peuvent en avoir besoin. À cette fin, la BDSPS offre la fonction « pmaddent » (module de paramètre, ajout d'une entrée).

L'utilisateur appelle cette fonction une fois pour chaque nouveau paramètre, immédiatement avant l'instruction « DEBUG_OFF (Ampd », vers la fin du fichier Ampd.cpp, près de la ligne 190.

Si l'utilisateur travaille avec une copie du fichier Ampd.cpp qui contient déjà l'appel de pmaddent pour d'autres paramètres, il peut utiliser ces appels comme gabarits. Dans l'exemple, toutefois, aucun autre paramètre n'a été ajouté pour le moment et nous copions le gabarit pmaddent du fichier C:\SPSM\MODEL\Mpd1.cpp (fichier 1 de définition de paramètres de modèle). Nous reconnaissons que le premier paramètre EITCMAX, paramètre de type NOMBRE, s'apparente au paramètre BOAS, près de la ligne 530. Nous copions simplement cet appel, que nous modifions de manière appropriée. Cette pratique est standard en développement en boîte de verre. L'appel copié est semblable à ce qui suit :

```
pmaddent(pcp, "BOAS",    (char *)&MP.BOAS,     NULL, P_SCL, C_NUM,  0,     0, NULL, 0);
```

Nous le modifions à nos fins en changeant les deux références à BOAS de manière que la ligne renvoie au nouveau paramètre. En remplaçant « BOAS » par « EITCMAX » et «(char *)&MP.BOAS » par « (char *)&MP.UM.EITCMAX », puisque le nouveau paramètre est un élément de la structure UM (modèle d'utilisateur) qui se trouve dans la structure MP (paramètres de modèle), nous obtenons le résultat suivant :

```
pmaddent(pcp, "EITCMAX",    (char *)&MP.UM.EITCMAX, NULL,   P_SCL, C_NUM,  0,    0, NULL, 0);
```

Pour le moment, nous conservons tous les autres arguments de la fonction sans égard à ce qu'ils représentent. Cela ne pose aucun problème dans la mesure où nous avons choisi un gabarit approprié. Ultérieurement, nous étudierons la signification de chacun des arguments de pmaddent de manière à pouvoir mieux évaluer les sources appropriées de gabarits pmaddent et à corriger plus efficacement tout choix inapproprié.

On peut également utiliser ce gabarit pour les 2 paramètres de point tournant de revenu, EITCTPMX et EITCTPRC, dont le code révisé de pmaddent se présente comme suit :

```
pmaddent(pcp, "EITCTPMX",    (char *)&MP.UM.EITCTPMX, NULL,   P_SCL, C_NUM,  0,  0, NULL, 0);
pmaddent(pcp, "EITCTPRC",    (char *)&MP.UM.EITCTPRC, NULL,   P_SCL, C_NUM,  0,  0, NULL, 0);
```

Les autres paramètres de type NUMBER sont les taux utilisés pour le calcul des taux d'instauration progressive et de retrait progressif du crédit d'impôt sur les revenus salariaux. Dans l'exemple tiré du fichier mpd1.cpp, nous constatons que nous pouvons utiliser le paramètre OASRR puisqu'il représente un taux de réduction. L'appel copié ressemble à ce qui suit :

```
pmaddent(pcp, "OASRR",   (char *)&MP.OASRR,      F_FRACT, P_SCL, C_NUM,  0,      0, NULL, 0);
```

Nous le modifions à nos fins en changeant les deux références à OASRR pour qu'elles correspondent au nouveau paramètre. En remplaçant « OASRR » par « EITCPIR » et « (char *) &MP.OASRR » par « (char *) &MP.UM.EITCPIR », nous obtenons ce qui suit :

```
pmaddent(pcp, "EITCPIR",   (char *)&MP.UM.EITCPIR,      F_FRACT, P_SCL, C_NUM,  0,  0, NULL,
0);
```

Nous pouvons copier cette ligne et la modifier pour le paramètre de taux de retrait progressif et obtenir le résultat suivant :

```
pmaddent(pcp, "EITCPOR",   (char *)&MP.UM.EITCPOR,      F_FRACT, P_SCL, C_NUM,  0,  0, NULL,
```

```
0);
```

Nous choisissons UIWAITWKS (nombre entier de semaines dans la période de carence de l'assurance-emploi) comme gabarit pour le paramètre à nombre entier représentant l'âge des enfants de la famille de recensement qui détermine l'admissibilité de la famille au crédit. L'appel copié est semblable à ce qui suit :

```
pmaddent(pcp, "UIWAITWKS",   (char *)&MP.UIWAITWKS,   NULL, P_SCL, C_INT,    0,      0, NULL,
0);
```

Comme dans le cas des paramètres ci-dessus, nous modifions le gabarit à deux endroits; nous remplaçons le nom du paramètre par celui du nouveau paramètre et sa relation avec la structure MP en incluant l'indicateur UM. L'appel de pmaddent modifié ressemble à ce qui suit :

```
pmaddent(pcp, "EITCAGE",   (char *)&MP.UM.EITCAGE,   NULL, P_SCL, C_INT,    0,  0, NULL, 0);
```

De même, nous choisissons un paramètre drapeau existant qui sert de gabarit pour notre drapeau de nouveau crédit d'impôt sur les revenus salariaux, OASFLAG, qui détermine s'il faut calculer ou non les prestations de la Sécurité de la vieillesse. Avant les modifications, le dernier appel ressemble à ce qui suit :

```
pmaddent(pcp, "OASFLAG",   (char *)&MP.OASFLAG,    NULL, P_SCL, C_INT,  E_FLAG, 0, NULL, 0);
```

Comme dans les exemples précédents, nous modifions le gabarit à deux endroits. L'appel de pmaddent modifié ressemble à ce qui suit :

```
pmaddent(pcp, "EITCFLAG",   (char *)&MP.UM.EITCFLAG, NULL, P_SCL, C_INT, E_FLAG, 0, NULL, 0);
```

Ces simples ajouts mettent un terme à la modification du fichier `Ampd.cpp` et, une fois que nous les avons attribuées, rendent les VALEURS des nouveaux paramètres disponibles à l'ensemble de la BDSPS. Plus loin dans la présente section, nous traitons de certains des mécanismes qui permettent à l'utilisateur d'effectuer les attributions. Il nous faut également fournir des étiquettes de paramètre claires afin que la BDSPS puisse les utiliser pour documenter de manière significative les paramètres de modèle, le cas échéant.

Ici encore, la conception de la BDSPS facilite le travail en offrant la fonction `stradd`, qui permet d'effectuer l'étiquetage. Immédiatement après les instructions de pmaddent, nous insérons les lignes suivantes pour appeler cette fonction :

```
stradd("EITCMAX", "Earned Income Tax Credit Maximum Amount");
stradd("EITCPIR", "Earned Income Tax Credit Phase In Rate");
stradd("EITCPOR", "Earned Income Tax Credit Phase Out Rate");
stradd("EITCTPMX", "EITC Income Turning Point for Maximum Credit");
stradd("EITCTPRC", "EITC Income Turning Point for Reduced Credit");
stradd("EITCAGE", "EITC Child Age for Census Family Eligibility");
stradd("EITCFLAG", "Earned Income Tax Credit Activation Flag");
```

La fonction `stradd` (ajout de chaîne) permet de « lier » la chaîne de descripteur au paramètre de manière que le descripteur figure automatiquement dans toute la documentation et tout l'étiquetage pertinents de la BDSPS. Les arguments de la fonction sont à ce point simples (une chaîne identifie le nom du nouveau paramètre et une autre fournit la description) que nous n'avons pas besoin de recourir à un gabarit.

L'élément final de cette étape, la compilation partielle de la fonction `Ampd.cpp`, `est facultatif, mais nous le recommandons puisqu'il peut assurer un développement ordonné des applications en boîte de verre.`

Ce type de compilation permet à l'utilisateur de demander au compilateur de vérifier les erreurs de syntaxe pendant qu'il a fraîchement en mémoire la nature de la modification effectuée. Toutefois, cette opération ne vérifie pas que le code source modifié est aligné sur le reste de la BDSPS. Pour que la compilation avec Debug fonctionne, l'utilisateur doit avoir modifié tous les fichiers d'en-tête pertinents, dans ce cas-ci, le fichier Mpu.h.

MODIFIER LES FONCTIONS QUI UTILISENT LES NOUVEAUX PARAMÈTRES

Pour terminer les modifications de la programmation nécessaires à l'ajout de paramètres, nous devons modifier la fonction Agai.cpp de manière à lui faire utiliser les nouveaux paramètres symboliques plutôt que les valeurs implantées directement dans l'exemple de démarrage rapide. Pour commencer, nous ajustons l'étiquette définie pour cette fonction, c'est-à-dire le code de définition de l'étiquette, de la manière suivante :

```
/*global*/ char FAR Tfa[] = "Agai.cpp Parameterized"
```

La BDSPS peut utiliser cette étiquette chaque fois qu'elle doit utiliser la description de fonction dans sa documentation.

Les modifications de code à apporter à la fonction Agai.cpp sont faciles à appliquer.

Nous remplaçons dans l'exemple de démarrage rapide la valeur « 1200.0 » par la représentation symbolique « MP.UM.EITCMAX ». Les règles d'attribution des noms, exactement identiques à celles utilisées pour l'appel de la fonction « pmaddent » dans la modification apportée ci-dessus au fichier Ampd.cpp, reflètent l'emplacement de EITCMAX dans la sous-structure UM (modèle d'utilisateur) de la structure MP (paramètre de modèle) que la BDSPS utilise pour stocker tous les paramètres de modèle.

Nous remplaçons dans l'exemple de démarrage rapide la valeur « 20 », qui représente l'âge que doivent avoir les enfants de la famille pour que celle-ci soit jugée admissible au crédit d'impôt, par la valeur MP.UM.EITCAGE. Toutes les formules pertinentes sont ajustées en conséquence. Nous pouvons également utiliser ce paramètre pour indiquer la limite d'âge des personnes qui peuvent recevoir le crédit une fois que l'admissibilité de la famille a été déterminée.

Donc, le code source de l'exemple de démarrage rapide est révisé et remplacé par ce qui suit :

```
void Agai(
      P_hh hh
)
{
      register P_in in;
      register int ini;
      register P_in ineld;
      register P_in inspo;
      register P_cf cf;
      register int cfi;
      int nceitc;
```

```
      NUMBER cfempinc;
      NUMBER eitc;

      DEBUG_ON("Agai");

      /* process persons in household – currently commented out*/
/*    for (ini=0, in=&hh->in[0]; ini<hh->hhnin; ini++, in++) {
            in->im.imiosa = ZERO;
      }
*/
      if (MP.UM.EITCFLAG) {
            /* process each census family in household */
            for (cfi=0, cf=&hh->cf[0];  cfi<hh->hhncf;  cfi++, cf++) {
               /* initialise elder's pointer */
               ineld = cf->cfineld;

               /* calculate elder's contribution to family net income */
               cfempinc = ineld->id.idiemp;


               if (cf->cfspoflg) {
                     DEBUG1("%s spouse present\n");
                     inspo = cf->cfinspo;    /* spouse's in pointer */

                     /* add spouse's net income to family net income */
                     cfempinc += inspo->id.idiemp;
               }

               nceitc = 0;

               /* process children in census family */
               for (ini=0, in=cf->cfinch;  ini<cf->cfnchild; ini++, in++)
{

                        if (in->id.idage > MP.UM.EITCAGE) {
                           DEBUG2("%s discarding old child, aged %d\n",
in->id.idage);
                           continue;
                        }

                        /* Count up remaining children */
                        nceitc++;
               }

               eitc = 0;
               if (nceitc > 0 ) {
                     if ( cfempinc < MP.UM.EITCTPMX ) {
                              eitc = MP.UM.EITCPIR * cfempinc;
                     }
                     else if ( cfempinc <= MP.UM.EITCTPRC ) {
                           eitc = MP.UM.EITCMAX;
                     }
                     else {
                           eitc = nneg(MP.UM.EITCMAX - ((cfempinc -
MP.UM.EITCTPRC) * MP.UM.EITCPOR));
                     }
                  }
```

```
                  /* process persons in census family */
                  for (ini=0, in=cf->cfin;  ini<cf->cfnpers;  ini++, in++)
{
                        if (in->id.idage > MP.UM.EITCAGE) {
                              in->im.imiosa = eitc;
                        }
                  }
            }
      }

      DEBUG_OFF("Agai");
}
```

La logique sous-jacente, qui demeure inchangée, est maintenant définie par l'intermédiaire de paramètres. En rédigeant le code source de cette façon, nous présumons que les utilisateurs du modèle attribueront uniquement des valeurs raisonnables aux paramètres. Par exemple, nous sommes confiants qu'aucun utilisateur n'attribuera par inadvertance la valeur zéro (0) au paramètre MP.UM.EITCAGE, créant ainsi accidentellement un crédit d'impôt pour les familles dont l'âge des enfants est 0 pour ensuite attribuer le crédit à tous les membres de la famille admissible dont l'âge est supérieur à 0. Plus loin, nous montrerons comment l'utilisateur peut se servir des fonctions de vérification-correction de la BDSPS pour s'assurer que les valeurs attribuées aux paramètres sont raisonnables.

Encore une fois, nous exécutons une compilation avec Debug pour identifier les erreurs de syntaxe avant de compiler le nouveau modèle.

VALIDER ET EFFECTUER DES EXÉCUTIONS DE PRODUCTION EN BOÎTE NOIRE

Comme dans le cas de l'exemple de démarrage rapide, nous devons encore vérifier si les résultats obtenus sont raisonnables. Puisque le coût et le temps requis pour l'exécution de la BDSPS sont négligeables, deux exécutions de validation particulières s'imposent d'office.

Les utilisateurs peuvent soit lire un fichier d'inclusion de paramètres de modèle (.mpi) s'ils utilisent le MSPS Classique, soit entrer à la volée des valeurs de paramètres de boîte de verre s'ils utilisent MSPS Visuel. Dans MSPS Visuel, ces paramètres se trouvent dans la dernière section "Paramètres pour les transferts et l'impôt créer par l'utilisateur" de l'onglet "Impôts et transferts" (le cas échéant) pour le scénario de base et de variante, selon le choix de modèles standard et (ou) de remplacement effectué par l'utilisateur dans la boîte de définition de scénario pour l'utilisation de MSPS Visuel.

1. La première exécution, avec les paramètres `EITCMAX et EITCPIR` réglés à zéro, utilise une exécution de base et de variante et vérifie les résultats du revenu disponible. Nous attribuons la valeur 0 aux paramètres `EITCMAX` et `EITCPIR` et la valeur 1 au paramètre `EITCFLAG`. Nous nous attendons à ce qu'il n'y ait aucune différence entre les systèmes de base et de variante puisque la valeur zéro attribuée au paramètre du montant de crédit d'impôt annule la modification.

2. Nous modifions le premier test comme suit : attribution de la valeur 1200 au paramètre `EITCMAX, maintien de la valeur 1 dans le paramètre`

EITCFLAG, et attribution des valeurs 0.15 au paramètre EITCPIR, 0.10 au paramètre EITCPOR, 8000 au paramètre EITCTPMX et 12000 au paramètre EITCTPRC. Encore une fois, nous demandons à la sortie les tableaux de base et de variante du test précédent et nous nous attendons à constater les mêmes résultats que ceux obtenus dans l'exemple de démarrage rapide avec les valeurs implantées directement.

3. Nous attribuons la valeur 2400 au paramètre EITCMAX; nous nous attendons que cela augmente considérablement le coût de l'option hypothétique puisque nous doublons le montant maximum de crédit. Les tableaux spécifiques nous permettent de vérifier facilement, au moins pour le montant brut de crédit, si les bons montants de crédit d'impôt ont été calculés pour chaque famille admissible compte tenu du revenu gagné.

4. Enfin, nous ajoutons un quatrième test pour désactiver le crédit d'impôt avec le paramètre EITCFLAG. En effectuant ce test de validation, nous conservons les valeurs de tous les autres paramètres afin de nous assurer que les effets notés soient uniquement dus à l'attribution de la valeur 0 au paramètre drapeau. Comme dans le cas de la première exécution de validation décrite ci-dessus, nous nous attendons à ce qu'il n'y ait aucune différence entre les revenus disponibles de base et de variante puisque le calcul du crédit d'impôt a été supprimé.

Pour effectuer les tests de validation, il ne reste qu'à attribuer les valeurs désirées aux nouveaux paramètres. La conception de la BDSPS facilite cette opération. Si nous exécutons simplement le nouveau modèle sans nous préoccuper de préciser la valeur de paramètre requise, la BDSPS relève l'omission et nous permet de fournir la valeur « à la volée » avec la fonction de modification de paramètres. Le nouveau fichier de paramètres peut également avoir été précisé dans un fichier MPI (inclusion de paramètres de modèle). La description précise de ces deux dernières méthodes se trouve dans le *Guide d'utilisation*.

En effectuant les tests ci-dessus, nous constatons que la modification, soit l'ajout des nouveaux paramètres, a été effectuée correctement puisque tous les ensembles de sorties sont tels que prévus. Les résultats du troisième test, dans lequel nous doublons la valeur du paramètre EITCMAX (montant maximum de crédit d'impôt) sont particulièrement importants. Ils nous permettent de vérifier si les montants appropriés de prestations supplémentaires sont ajoutés aux mêmes familles et qu'ils correspondent aux montants prévus. Maintenant que les modifications apportées au modèle sont validées, nous pouvons effectuer l'ensemble des exécutions de production. Par exemple, un client peut demander d'attribuer la valeur 18 au paramètre EITCAGE afin de vérifier si le nombre de familles admissibles au crédit d'impôt est inférieur à ce qu'il serait si l'on attribuait la valeur 20 au paramètre, et si les coûts agrégés sont légèrement inférieurs. Nous pouvons également attribuer une valeur beaucoup plus grande, p. ex., 5000.0, au paramètre EITCMAX afin de confirmer qu'un transfert d'une telle importance a pour effet de modifier de manière substantielle le revenu disponible des personnes concernées.

RÉSUMÉ/CONCLUSION

Pour conclure, il convient de souligner, sans les redévelopper, les points clés de l'ajout de

paramètres scalaires ordinaires à un modèle. À cet égard, nous convenons que l'analyste utilise des COPIES des fichiers pertinents et qu'il exécute toutes les modifications dans un sous-répertoire de travail réservé à l'analyse en cours. Nous présumons également que l'utilisateur a mis à jour le projet de manière à inclure tous les fichiers de code source pertinents et que, du point de vue technique, il récupère le plus souvent des éléments de codes existants (gabarits), qu'il modifie au besoin.

1. Modifier le fichier d'en-tête `Mpu.` en ajoutant une instruction pour chaque nouveau paramètre. L'instruction indique le nom du paramètre et son type, et utilise la valeur NUMBER pour les valeurs à virgule flottante.

2. Modifier le fichier de code source `Ampd.cpp` en ajoutant deux instructions pour chaque nouveau paramètre.

    - Ajouter un appel de « pmaddent » pour chaque paramètre de manière que la BDSPS puisse mettre sa valeur à la disposition de toutes les fonctions appelées par `Adrv.cpp`. Il suffit normalement de copier un appel existant puis de le modifier à deux endroits – nom et adresse du paramètre.

    - Ajouter un appel de stradd pour chaque paramètre de manière que la BDSPS lie l'étiquette de paramètre à ce nouveau paramètre.

3. Modifier les fonctions essentielles pertinentes de manière à utiliser les nouveaux paramètres en modifiant l'étiquette ainsi que la logique interne de la fonction.

4. Mettre au point et compiler le nouveau programme. Faire les « exécutions de production » requises avec le modèle, puis interpréter les résultats.

Développement en boîte de verre : ajout de paramètres inhabituels

Ce chapitre décrit plus en détail les arguments de la fonction pmaddent et son utilisation lorsque l'utilisateur ajoute des paramètres matriciels, vectoriels et scalaires à des applications en boîte de verre. Pour ce faire, il s'appuie sur la base établie à la section précédente (ajout des paramètres scalaires ordinaires) et élabore de nouvelles considérations pour des paramètres scalaires inhabituels, des tableaux vectoriels et de recherche, et des matrices. La dernière section résume les points essentiels concernant l'ajout de ces paramètres moins courants à un modèle.

La première section du chapitre présente l'ensemble des arguments pour la fonction clé pmaddent et décrit les principales caractéristiques de chacun d'eux. La section suivante présente une liste des types de paramètres scalaires que l'utilisateur peut désirer ajouter. Pour chaque type, elle indique brièvement l'objet, décrit les arguments clés de pmaddent et propose un gabarit pmaddent approprié pour créer un paramètre de ce type. Suivent des sections qui traitent des considérations spéciales liées à l'ajout de vecteurs de paramètres, puis de paramètres de recherche et de matrices de paramètres.

FONCTION PMADDENT ET SES ARGUMENTS

Rappelez-vous que, dans la section portant sur la description de l'ajout de paramètres ordinaires, l'aspect le plus complexe de la mise de nouveaux paramètres à la disposition d'un modèle réside dans la modification du fichier Ampd.cpp, les changements du fichier Mpu.h se limitant à des définitions très simples des types de paramètre. Pour modifier le fichier Ampd.cpp, le seul défi important, et non particulièrement difficile à relever, consiste à appeler la fonction pmaddent. Nous avons noté que l'utilisateur de la boîte de verre peut habituellement contourner la complexité de cette fonction en choisissant simplement un gabarit d'appel « approprié », copié d'un paramètre déjà défini « suffisamment similaire ». Dans cette section, nous expliquons plus en détail la signification des divers arguments de pmaddent de manière que l'utilisateur puisse utiliser la fonction en toute confiance, même s'il n'existe aucun gabarit simple à copier et modifier.

Le point de départ de la description des arguments de pmaddent est le commentaire explicatif qui se trouve dans le fichier Ampd.c (près de la ligne 150 de la version BOÎTE DE VERRE). Nous traitons chacun des dix arguments dans l'ordre, tout en soulignant que l'utilisateur doit rarement recourir à cette information. La plupart du temps, le paramètre ajouté est facile à comprendre et l'utilisateur peut facilement trouver un gabarit de paramètre à peu près semblable. Dans tous ces cas, l'utilisateur doit simplement modifier le gabarit pertinent et poursuivre la modélisation en laissant les aspects complexes de pmaddent aux personnes chargées des tâches inhabituelles.

Voici un résumé des arguments de pmaddent dans Ampd.cpp.

```
/**
* pmaddent(
*    pcp,                  <= parameter chain being extended (leave as is)
*    "XXXXX",              <= name by which the parameter will be known
*    (char *)&MP.UM.XXXXX, <= address of the parameter
*    Format,               <= printing information for the parameter
*    Agg_Type,             <= Aggregate type (scalar, vector, etc.)
*    C_Type,               <= C-type (integer, number, string)
*    Edit,                 <= Edits to be performed
*    Row_max,              <= Maximum number of rows, or option edit limit.
*    Rows_addr,            <= Address of int holding current number of rows
*    Limit                    <= Number of columns);
**/
```

Le premier argument (pcp) est particulièrement simple; l'utilisateur entre TOUJOURS la variable pcp. L'argument identifie la chaîne de paramètres spécifique que l'utilisateur prolonge. Bien que la BDSPS utilise d'autres chaînes de paramètres dans ses opérations, l'utilisateur peut ajouter des paramètres SEULEMENT à la chaîne pcp.

Le deuxième argument, représenté par la chaîne fictive « XXXXX » dans le commentaire, est le nom de l'utilisateur du paramètre. Le nom sera le même que celui utilisé dans la définition Mpu.h. L'utilisateur doit prendre soin de choisir des mnémoniques raisonnables pour ces noms, par exemple EITCFLAG que nous avons utilisé précédemment. La règle de la BDSPS stipule que ces noms doivent commencer par une majuscule et contenir seulement des majuscules et des chiffres.

Le troisième argument, représenté par la chaîne fictive (char *)&MP.UM.XXXXX est l'adresse du paramètre. La partie initiale (« cast » dans le langage C) de l'argument,

« (char * », est invariable. La partie « MP.UM » est également invariable puisque les paramètres de l'utilisateur sont toujours ajoutés à la structure « paramètre de modèle, modèle d'utilisateur ». La partie « XXXXX » représente le nom du paramètre d'utilisateur; elle prend la valeur de la chaîne utilisée comme deuxième argument, sans les guillemets. Conformément à la manière dont le langage C traite les adresses de variable, la perluète (&) est présente si le paramètre est un paramètre scalaire, et habituellement absente s'il ne l'est pas (c.-à-d., dans le cas d'un vecteur, d'un paramètre de recherche de tableau ou d'une matrice). On aborde plus loin dans une rubrique particulière le mécanisme du langage C qui permet de référer spécifiquement au premier élément d'un tableau. Dans le cas spécial d'un paramètre « FICTIF » décrit ci-dessous, ce troisième argument prend la valeur « NULL ».

Le quatrième argument, représenté dans la description ci-dessus par « Format », est une chaîne. Il contient l'information sur la façon dont la BDSPS doit afficher la valeur du paramètre dans la documentation. Habituellement, l'utilisateur choisit le format prédéfini « NULL » pour indiquer que la BDSPS doit imprimer le paramètre de la façon qui lui semble correcte. Un autre format prédéfini, « F_FRACT », contient la chaîne « 8.5 » particulièrement appropriée pour l'impression des valeurs de fraction. L'utilisateur peut aussi entrer une chaîne explicite, par exemple « 8.0 », qui indique que la valeur a une longueur de huit caractères et ne peut inclure de fraction. L'argument « 7.2 » indique une chaîne de sept caractères, suivie de deux décimales. Le cas échéant, l'argument peut inclure plusieurs indicateurs de format (p. ex., « 8.0 8.2 8.2 »), par exemple pour les paramètres de type recherche de tableau. Le format prédéfini F_LKTUR, utilisé pour les paramètres de type P_LKPXY, est un exemple concret de cette utilisation.

Le cinquième argument, représenté dans la description ci-dessus par « Agg_Type », indique le type de paramètre. Cet argument reflète un choix forcé entre les six valeurs entières de 0 à 5. Chacune a une contrepartie mnémonique dont l'utilisateur peut se servir, aux fins de clarté, au lieu de la valeur numérique elle-même. Les six valeurs, leur contrepartie mnémonique, et leur interprétation, sont les suivantes :

La valeur 0, représentée par la mnémonique P_SCL, est la valeur la plus courante. Elle est utilisée pour un paramètre qui a une valeur scalaire (nombre entier, flottant, fraction, etc.).

La valeur 1, représentée par la mnémonique P_VCT, est utilisée lorsque le paramètre est un vecteur. D'autres renseignements clés au sujet du vecteur, par exemple le nombre d'éléments qu'il contient, sont donnés par d'autres arguments de pmaddent.

Les valeurs 2 et 3, représentées par les mnémoniques P_LKPXY et P_LKPSL, sont utilisées à l'intérieur de la BDSPS pour deux types spéciaux de tableaux dans lesquels les recherches sont effectuées, une en format X-Y et l'autre en format plage-pente. Dans le cas où l'utilisateur désire créer ces types de paramètre, les paramètres GISST et FTX fournissent des exemples opérationnels. Ces deux types définissent des tableaux qui correspondent aux fonctions LKUP1 et LKUP2, respectivement; les fonctions LKUP1 et LKUP2 elles-mêmes sont documentées dans le *Guide des algorithmes*. L'utilisation de tableaux dans la BDSPS est documentée plus en détail dans ce chapitre. La valeur 4, représentée par la mnémonique P_TBL, est utilisée lorsque le paramètre est une matrice bidimensionnelle (tableau). D'autres renseignements clés sur la matrice, par exemple, le nombre de rangées et de

colonnes, sont donnés dans d'autres arguments de pmaddent. La matrice des taxes à la consommation, CTTXRM, est un bon exemple.

La valeur 5, représentée par la mnémonique P_DUMMY, n'est habituellement pas utilisée. Ce type de paramètre correspond à une entrée fictive utilisée pour conserver le nom d'une chaîne d'en-tête aux fins de documentation.

Le sixième argument, représenté dans la description ci-dessus par « C_Type », indique le type de paramètre. Il y a trois entrées possibles pour cet argument. La valeur C_INT est appropriée lorsque la valeur inhérente du paramètre est un nombre entier, c.-à-d. un nombre sans partie fractionnaire, dont la valeur se situe entre les valeurs limites de nombre entier du langage C. On utilise une valeur C_INT pour cet argument lorsque l'entrée du paramètre dans `Mpu.h` a utilisé une déclaration « int ». Les paramètres « drapeaux » ou « options » sont habituellement des nombres entiers.

La valeur C_NUM est appropriée lorsque la valeur du paramètre peut inclure une partie fractionnaire ou qu'elle est trop grande pour être stockée comme nombre entier. L'utilisateur utilise une valeur C_NUM pour cet argument lorsque l'entrée du paramètre dans `Mpu.h` a utilisé une déclaration « NUMBER ».

La valeur C_STR est utilisée lorsque la valeur de paramètre est une entrée fictive utilisée pour une chaîne d'en-tête. Les utilisateurs n'ont habituellement pas l'occasion d'utiliser la valeur C_STR.

Le septième argument, représenté dans la description ci-dessus par « edit », indique les fonctions de vérification qui seront appliquées à la valeur du paramètre. L'activation du contrôle de vérification force la valeur du paramètre à respecter diverses contraintes jugées appropriées. En plus, ces fonctions peuvent limiter la capacité de l'utilisateur de modifier les valeurs de paramètre avec la fonction de modification des paramètres de la BDSPS au moment de l'exécution. L'argument de pmaddent qui régit les contrôles de vérification est une valeur à nombre entier. Habituellement, l'utilisateur choisit une valeur en entrant un élément d'un ensemble de valeurs mnémoniques prédéfinies (décrites ci-dessous).

Les codes et leur interprétation sont les suivants :

E_NONE (valeur 0) indique qu'aucun contrôle de vérification ne doit être exécuté sur ce paramètre.

E_FIXL (valeur 1) s'applique seulement lorsque le paramètre est un vecteur, un tableau de recherche ou un tableau (dont le nombre maximum de rangées est connu). Le code de vérification empêche l'utilisateur de tenter de modifier le nombre réel de rangées de la valeur maximale. La mnémonique ici indique que la limite de nombre de rangées est fixe.

E_FLAG (valeur 2) indique que le paramètre est un drapeau. Selon les règles de la BDSPS, cela signifie que le paramètre est traité comme une variable binaire (définie comme nombre entier) qui droit prendre la valeur 0 (zéro) ou 1 (un).

E_FRCT (valeur 4) indique que le paramètre est une valeur fractionnaire qui doit se situer

entre 0,0 et 1,0.

E_NOCH (valeur 8) indique que l'utilisateur n'a pas le droit de modifier la valeur du paramètre avec l'éditeur de paramètres de la BDSPS. Ce contrôle de vérification peut s'appliquer à n'importe quel type de paramètre, C_INT, C_NUM ou C_STR.

E_OPT (valeur 16) indique que le paramètre est un type spécial d'« option », correspondant à un choix forcé (nombre entier) de valeur se situant entre 1 et le nombre maximal facultatif permis. Le nombre maximal lui-même est fourni, pour les paramètres d'options, par le huitième argument de pmaddent.

Lorsqu'il y a lieu d'utiliser plusieurs codes, l'utilisateur peut simplement additionner les valeurs des éléments pertinents. Par exemple, la valeur 12 indique un paramètre qui doit être fractionnaire et que l'utilisateur ne peut modifier à la volée au moment de l'exécution.

Le huitième argument, représenté dans la description ci-dessus par « Row_max », indique le nombre maximal de rangées pour certains types de paramètres (P_VEC, P_LKPXY, P_LKPSL, ou P_TBL). (Notez que la souplesse de la BDSPS permet d'utiliser dans une application particulière un nombre réel de rangées inférieur à cette valeur maximale.) Pour les autres types de paramètres (P_SCL et P_DUMMY) cet argument doit contenir la valeur 0 (zéro), sauf pour les paramètres d'OPTION, où il indique le nombre de valeurs d'option légitimes. (La valeur N dans un paramètre d'OPTION indique que les valeurs légitimes se situent entre 1 et N inclusivement.) Comme les paramètres scalaires (P_SCL) sont les plus fréquemment utilisés, cet argument contient le plus souvent la valeur 0.

Le neuvième argument, représenté dans la description ci-dessus par « Rows_addr », contient l'adresse de la variable, en nombre entier, correspondant au nombre réel de rangées de certains types de paramètre, P_VEC, P_LKPXY, P_LKPSL et P_TBL. Lorsque le nombre de rangées n'est pas pertinent, par exemple pour un paramètre scalaire ou fictif, l'utilisateur entre la valeur « NULL »; l'argument contient donc habituellement cette valeur.

Le dixième et dernier argument de pmaddent, représenté dans la description ci-dessus par « Limit », indique, pour les paramètres de type P_TBL, le nombre de colonnes du tableau. Contrairement aux rangées, dont le nombre réel peut être plus petit que le nombre maximum, la BDSPS exige que le nombre réel de colonnes soit fixé à l'avance. Pour tous les autres types de paramètre, cet argument contient la valeur 0 (zéro).

Deux autres arguments facultatifs, non utilisés actuellement dans le code de la boîte de verre, peuvent être ajoutés à la liste. Le onzième est une étiquette utilisée pour les rangées des tableaux et les paramètres vecteurs. Le douzième est une étiquette utilisée pour les colonnes des paramètres tableaux. Ces étiquettes contrôlent l'affichage des paramètres dans MSPS Visuel.

DESCRIPTION DES PARAMÈTRES SCALAIRES

Après la description des arguments de pmaddent, nous étudions dans un premier temps les types de paramètre scalaire que l'utilisateur peut désirer ajouter, en ordre décroissant de leur fréquence d'utilisation prévue. Pour chaque type, la description précise 1) la nature générale

du paramètre, 2) les principaux arguments de pmaddent, et 3) un gabarit de pmaddent approprié. Même si le chapitre traite principalement des types de paramètre plus spécialisés, nous avons inclus dans cette section sur les paramètres scalaires, aux fins d'exhaustivité, des cas de types de paramètre plus courants déjà décrits à la section précédente du présent Guide.

Paramètres REAL/float/NUMBER

L'analyste utilise ce type de paramètre pour fournir une valeur réelle, par exemple certaines garanties de programme exprimées en dollars et en cents. La définition dans `Mpu.h` utilise la spécification NUMBER. Dans l'appel de pmaddent, l'argument clé est l'entrée C_NUM pour C_Type. Le gabarit suivant est approprié :

```
pmaddent(pcp, "FCBBAS",    (char *)&MP.FCBBAS,     NULL, P_SCL, C_NUM,  0,     0, NULL, 0);
```

Paramètres INTEGER/int

L'analyste utilise ce paramètre pour fournir une valeur qui est essentiellement un nombre entier, par exemple le nombre habituel de semaines de la période de carence aux fins de l'assurance-emploi. La définition dans `Mpu.h` utilise la spécification int dans l'appel de pmaddent, l'argument clé est l'entrée C_INT pour C_Type. Le gabarit suivant est approprié :

```
pmaddent(pcp, "UIWAITWKS",   (char *)&MP.UIWAITWKS,   NULL, P_SCL, C_INT,    0,     0, NULL, 0);
```

Paramètres FLAG

L'analyste utilise ce type de paramètre pour fournir une valeur de « commutation », par exemple un indicateur qui précise si certains autres calculs doivent être exécutés ou non. La définition de `mpu.h` utilise la spécification int pour un paramètre de ce genre. Dans l'appel de pmaddent, les arguments clés sont l'entrée C_INT pour le C_Type et l'entrée E_FLAG pour edit. Le gabarit suivant est approprié :

```
pmaddent(pcp, "FAFLAG",   (char *)&MP.FAFLAG,    NULL, P_SCL, C_INT,  E_FLAG, 0, NULL, 0);
```

Paramètres FRACTION

L'analyste utilise ce type de paramètre lorsqu'il désire fournir une valeur qui est essentiellement une fraction, donc moins élevée que la valeur d'un nombre à virgule flottante. Les taux d'imposition et les taux de cotisation sont de bons exemples de ce type de paramètre. La définition dans `Mpu.h` utilise la spécification NUMBER dans ces paramètres. Dans l'appel de pmaddent, les arguments clés sont l'entrée C_NUM pour C_Type et l'entrée F_FRACT pour Format. Dans l'appel de gabarit suggéré pour ce type de paramètre, l'utilisateur a choisi de NE PAS exiger de contrôle de vérification qui limite la valeur entre zéro et un; le gabarit est le suivant :

```
pmaddent(pcp, "UIPF",   (char *)&MP.UIPF, F_FRACT,P_SCL, C_NUM, 0,     0, NULL, 0);
```

Paramètres OPTION

L'analyste utilise ce type de paramètre lorsque le paramètre reflète une sélection imposée parmi un nombre fixe et limité de choix possibles; une valeur numérique permet d'indiquer une sélection nominale ou qualitative. Comme exemple de distinction qualitative de ce type on peut prendre le cas d'un paramètre qui indique si le montant du crédit d'impôt aux aidants naturels doit être déterminé soit 1) en accordant la valeur maximale du crédit, 2) en appliquant la valeur du livre vert au crédit, ou 3) en modélisant le montant avec le test de revenu approprié. Dans l'appel de pmaddent, les arguments clés sont l'entrée C_INT pour

C_TYPE, l'entrée E_OPT pour Edit et l'entrée numérique donnant le nombre de catégories légitimes pour l'argument Row-max. Le gabarit suivant est approprié :

```
pmaddent(pcp, "CGTCOPT",  (char *)&MP.CGTCOPT,    NULL, P_SCL, C_INT, E_OPT,  3, NULL, 0);
```

Paramètres EDIT-FRACTION

L'analyste utilise ce type de paramètre lorsqu'il désire limiter toute valeur fournie par l'utilisateur à un intervalle se situant entre zéro et un. Par exemple, le paramètre peut représenter un taux de réduction des taxes jugé déraisonnable s'il correspond à un taux inférieur à 0 % ou supérieur à 100 %. La définition utilisée dans `Mpu.h` pour un paramètre de fraction modifiable utilise une spécification NUMBER. Dans l'appel de pmaddent, les arguments clés sont l'entrée C_NUM pour C_Type et l'entrée E_FRCT pour Edit. L'utilisateur peut également désirer préciser une spécification de Format de F_FRACT. Le gabarit suivant est approprié :

```
pmaddent(pcp, "CHATR1",  (char *)&MP.CHATR1,      NULL, P_SCL, C_NUM, E_FRCT, 0, NULL, 0);
```

Paramètres DUMMY

L'utilisateur ne précise pas habituellement de paramètres DUMMY (fictifs) prévus pour transmettre de l'information sur l'étiquetage et la répartition en sections lors de la documentation des configurations de paramètre. Le gabarit suivant est représentatif :

```
pmaddent(pcp, "2.3.1",     NULL,                NULL, P_DUMMY, C_STR, 0,    0, NULL, 0);
```

L'utilisateur peut choisir des mécanismes qui permettent de fournir des valeurs pour tous les types de paramètres scalaires :

1. spécification par présence dans un fichier de paramètres d'inclusion supplémentaire (fichiers MPI, CPI et API), et

2. spécification par la fonction de modification de paramètres de MSPS Visuel. Vous trouverez des paramètres définis par l'utilisateur dans la dernière section de la hiérarchie sous les onglets de paramètre contrôle/ajustement/modèle.

VECTEURS DE PARAMÈTRES DÉFINIS PAR L'UTILISATEUR

Les parties précédentes du chapitre portent principalement sur les paramètres scalaires, types les plus courants et les plus faciles à décrire. La BDSPS offre aussi à l'utilisateur la possibilité de créer des vecteurs de paramètres. Ces vecteurs sont les plus appropriés pour créer un ensemble de paramètres connexes dont les membres se présentent dans un ordre « indexable » naturel d'une seule dimension.

Prenons le cas d'un analyste qui modélise un programme quelconque de supplément au logement que l'on veut proposer. Pour chaque taille de famille, de un à dix, ce programme hypothétique a une limite de revenu au-delà de laquelle une famille devient catégoriquement inadmissible aux prestations. Malheureusement, ces limites, bien qu'elles augmentent avec la taille de la famille, ne sont pas liées à la taille de manière uniforme ou facile à calculer. L'utilisateur opte plutôt pour dix paramètres différents, correspondant aux tailles un à dix et plus, pour représenter les seuils de prestation. Il est beaucoup plus logique d'utiliser un vecteur de paramètres, indexé selon la taille de la famille, que d'élaborer un code qui traite chacune des dix possibilités comme un cas distinct.

Dans cette section, nous définissons les principaux aspects que l'utilisateur doit comprendre pour définir des vecteurs de paramètres définis par l'utilisateur pour les modèles de la BDSPS. Bien que les commentaires formulés précédemment sur l'ajout général de paramètres demeurent valables (ordre des modifications à apporter aux fichiers, utilisation de mnémoniques, validation, etc.), nous mettons l'accent sur les aspects propres à l'utilisation efficace de vecteurs de paramètres définis par l'utilisateur.

Ajouts aux fichiers `Mpu.h`, `Cpu.h` ou `Apu.h`

L'utilisateur déclare des paramètres scalaires dans le fichier `Mpu.h` (ou `Cpu.h` ou `Apu.h`) et il doit également déclarer les vecteurs de paramètres définis par l'utilisateur dans ces fichiers. Bien que les déclarations de valeurs scalaires et vectorielles soient très similaires, les déclarations de vecteurs utilisent une expression entre crochets pour indiquer la longueur du vecteur. Puisque la BDSPS traite les vecteurs de paramètres comme des vecteurs de colonnes, la longueur du vecteur désigne le nombre de rangées.

Dans l'exemple de programme de subvention au logement, présumons que l'utilisateur a déclaré une constante manifeste HHPYCOMR (programme de logement hypothétique, nombre maximum de rangées de seuil de revenu). Il lui a attribué la valeur dix parce qu'il y aura un seuil distinct pour chaque taille de famille, de un à dix et plus. La constante est définie par l'instruction suivante :

```
#define HHPYCOMR 10   /* maximum # of number of rows in the HHPYCO vector  */
```

Consultez le fichier `Mp.h` dans le sous-répertoire `SPSM\DEFS` (à partir approximativement de la ligne 400) pour obtenir des illustrations de vecteurs de paramètres propres à la BDSPS en boîte noire et qui ne sont pas définis par l'utilisateur.

Le vecteur lui-même doit être nommé HHPYCO et inclure la $i^{ème}$ entrée correspondant au seuil d'une famille de taille i+1. (Il faut se rappeler que tous les vecteurs dans le langage C débutent par l'entrée zéro.) L'entrée du fichier `Mpu.h` pour le nouveau vecteur s'apparente à la suivante :

```
NUMBER HHPYCO[HHPYCOMR];  /* Hypothetical Housing Program Income Cutoffs */
```

Bien qu'il soit possible d'entrer la longueur directement dans la déclaration, par exemple en utilisant une constante telle HHPYCO[10], nous déconseillons fortement cette approche. Nous recommandons plutôt l'utilisation de la constante manifeste décrite ci-dessus puisque dans l'appel correspondant de pmaddent dans `Ampd.cpp`, on doit préciser une entrée pour le nombre maximum de rangées. L'utilisation de ce type de constante dans les deux fichiers `Mpu.h` (ou `Apu.h` ou `Cpu.h`) et dans `Ampd.c` évite toute possibilité que l'on utilise deux valeurs différentes lors d'une révision ultérieure. Les erreurs produites dans ce dernier cas sont telles qu'il devient très difficile d'en identifier la cause.

N'oubliez pas que le nombre réel de rangées d'un vecteur (colonne) lors d'une exécution de la BDSPS peut différer du nombre maximum possible (lui être inférieur). L'utilisateur doit donc également déclarer, dans le même fichier d'en-tête, une variable dans laquelle la BDSPS stockera le nombre réel de rangées utilisées (valeur qui peut varier d'une exécution à l'autre d'une version exécutable en mode boîte de verre). L'utilisateur, en insérant une déclaration supplémentaire dans le fichier d'en-tête, fournit une variable dans laquelle la

BDSPS stockera le nombre réel de rangées. Par convention, dans la BDSPS, les variables de longueur sont nommées comme suit : nom du paramètre suivi du suffixe « rows ». Le fichier `Mpu.h` doit aussi contenir la déclaration suivante :

```
int HPPYCOrows;          /*   number of rows in HPPYCO */
```

Le fichier mp.h du sous-répertoire `SPSM\DEFS` inclut de nombreux exemples dans la section sur les limites des tableaux (près de la ligne 2500). Ultérieurement, l'appel de HPPYCO par pmaddent dans le fichier `Ampd.cpp` fera référence à l'adresse de la variable HPPYCOrows.

Ajouts au fichier `Ampd.cpp`

Pour que la BDSPS puisse rendre les valeurs du nouveau vecteur de paramètres disponibles au code final de l'utilisateur, comme c'est le cas avec les paramètres scalaires, l'utilisateur doit établir les liens appropriés par un appel de pmaddent. L'appel doit ressembler à ce qui suit :

```
pmaddent(pcp, "HHPYCO",  (char *)MP.UM.HHPYCO,  NULL, P_VCT, C_NUM, E_NONE,
HHPYCOMR, &MP.UM.HHPYCOrows, 0);
```

ou

```
pmaddent(pcp, "HHPYCO",  (char *)&MP.UM.HHPYCO[0],  NULL, P_VCT, C_NUM, E_NONE,
HHPYCOMR, &MP.UM.HHPYCOrows, 0);
```

Dans le premier cas, le troisième argument n'utilise pas la perluète puisque la référence porte sur le nouveau vecteur de paramètres; le langage C traite ce type de référence comme adresse du premier élément. Dans le deuxième cas, l'utilisateur a choisi de référer plus explicitement à l'adresse du premier élément en incluant la perluète et l'index [0]. Les fichiers `MpdX.cpp` du sous-répertoire `SPSM\MODEL` contiennent des exemples des deux types de référence.

Dans la description des aspects particuliers des vecteurs de paramètres définis par l'utilisateur, trois autres arguments de pmaddent méritent des commentaires particuliers. L'argument Agg_Type (n° 5) prend obligatoirement la valeur P_VCT. L'argument Row-max (n° 8) est la constante manifeste créée dans le fichier `Mpu.h` pour préciser le nombre maximum de rangées; dans l'exemple du programme de subvention au logement, cette constante correspond à l'entrée HHPYCOMR. Enfin, l'entrée Rows-addr (n° 9) correspond au nom de la variable déclarée qui doit contenir le nombre réel de rangées, précédé de la perluète; dans le même exemple, cette constante correspond à l'entrée &MP.UM.HHPYCOrows.

Remarquez que d'autres capacités activées par les arguments de pmaddent demeurent accessibles à l'utilisateur. On utilise donc C_Type pour indiquer si la variable contient une valeur à virgule flottante ou un nombre entier. S'il le désire, l'utilisateur peut se servir de l'argument Format pour préciser un format pour chacune des valeurs individuelles du vecteur. Il utilise l'argument Edit pour imposer les vérifications de modification pertinentes.

Comme c'est le cas avec les paramètres scalaires, l'utilisateur voudra aussi modifier le fichier `Ampd.cpp` afin d'ajouter l'appel de stradd pour chaque nouveau vecteur de paramètres définis par l'utilisateur. Ainsi, la BDSPS inclura dans la documentation sur les nouveaux paramètres la description textuelle de l'utilisateur.

Références aux vecteurs de paramètres définis par l'utilisateur dans le code source

Une fois que l'utilisateur a fini d'apporter aux fichiers d'en-tête et `Ampd.cpp` les modifications nécessaires pour rendre le vecteur de paramètres accessibles aux fonctions de base, il doit créer les références aux valeurs de paramètre pertinentes dans ces fonctions. Pour poursuivre avec l'exemple hypothétique précédent, présumons que l'utilisateur rend accessible une variable à nombre entier, HHPFS (taille de la famille pour le programme hypothétique de subvention au logement), qui indique la taille de la famille définie par les règlements qu'il prévoit appliquer pour régir le programme. Présumons également que l'utilisateur est absolument certain que la valeur de HHPFS se situera entre 1 et 9, inclusivement. Pour référer au seuil de revenu pertinent pour les prestations du programme hypothétique, comme le langage C numérote toujours les éléments d'un vecteur en commençant à zéro, l'utilisateur se servira d'une expression ayant la forme suivante :

```
MP.UM.HHPYCO[HHPFS-1]
```

Spécification des valeurs de vecteur de paramètres

Afin que le nouveau code de l'utilisateur soit vraiment utile, les valeurs des éléments du vecteur doivent être mises à la disposition de la BDSPS de manière qu'elle puisse, à son tour, les rendre accessibles au code de l'utilisateur. Habituellement, l'utilisateur précise ces valeurs dans un fichier « .MPI » (ou son équivalent « `.CPI` » ou « .API »). Le vecteur FLVCRT, qui précise le pourcentage du coût du fonds de travailleurs alloué comme crédit par province, est un bon exemple.

```
FLVCRT       10    # Percent of labour sponsored funds costs allowed as a credit
     0.15
     0.15
     0.15
     0.15
     0.15
     0.15
     0.15
     0.15
     0.15
     0.15
```

Le format est clair. La première ligne contient le nom du paramètre, suivi du nombre RÉEL d'éléments à utiliser; au choix, on peut ajouter un commentaire pour que quiconque consulte le fichier sache à quoi sert le paramètre. Les lignes qui suivent précisent, à raison d'une valeur par ligne, les valeurs du vecteur. Il est important que l'entrée du nombre d'éléments ne dépasse pas la valeur du nombre maximum de rangées précisé dans l'entrée pmaddent et que le nombre de lignes supplémentaires du fichier de paramètres soit égal au nombre figurant sur la première ligne des paramètres; la BDSPS vérifie que ces exigences sont respectées.

Pour poursuivre avec le programme hypothétique de subvention au logement, l'utilisateur peut entrer, dans le fichier « .MPI », une déclaration semblable à ce qui suit :

```
HHPYCO         10   # Income cutoffs for housing program, by family size
      5000.0
      6120.0
      7250.0
      8400.0
      9500.0
```

```
10600.0
11600.0
12500.0
13300.0
13900.0
```

Résumé

Les étapes clés de l'ajout de vecteurs de paramètres définis par l'utilisateur à un modèle boîte de verre de la BDSPS peuvent se résumer ainsi :

1. Apporter les modifications appropriées au fichier d'en-tête (p. ex., `Mpu.h`).
    - Utiliser une constante manifeste pour la longueur maximale du vecteur, p. ex.,
    - #define HHPYCOMR 10 /* Nombre maximum de rangées pour HHPYCO */.
    - Déclarer le vecteur lui-même,
    - NUMBER HHPYCO[HHPYCOMR]; /* commentaire */.
    - Déclarer une variable pour recevoir la longueur réelle du vecteur, p. ex.,
    - int HPPYCOrows; /* Nombre réel de rangées dans HPPYCO */.
2. Apporter les modifications appropriées au fichier `Ampd.cpp;` ne pas oublier les avantages d'une compilation partielle.
    - Insérer un appel approprié de pmaddent, habituellement par la modification d'une copie d'un appel existant.
    - Entrer un appel de stradd de manière que la BDSPS puisse étiqueter les nouveaux paramètres, le cas échéant.
3. Rédiger en langage C le code source qui utilise les paramètres. Ne pas oublier que le langage C commence à numéroter les vecteurs en utilisant l'élément zéro. À ce stade-ci, il est souvent utile d'effectuer une compilation de mise au point.
4. Fournir la valeur des éléments du vecteur par une entrée multilignes dans un fichier de paramètres approprié.
5. Ne pas oublier qu'il faut valider et tester le nouveau code pour s'assurer qu'il accomplit ce pourquoi il a été créé.

TABLEAUX DÉFINIS PAR L'UTILISATEUR POUR LES RECHERCHES

Les paramètres, sous forme de tableaux, sont principalement utiles pour effectuer certaines recherches, par exemple, trouver la valeur y correspondante d'une valeur x donnée. La présente section utilise à titre d'exemples deux tableaux actuels de la BDSPS et un nouveau tableau hypothétique défini par l'utilisateur et qui doit être ajouté comme paramètre. Ensemble, ces trois exemples couvrent les trois grandes formes de paramètres de tableau qui répondent habituellement aux besoins d'un utilisateur du mode boîte de verre.

Le premier exemple de tableau existant concerne les impôts fédéraux – calculer, pour un revenu imposable donné, l'impôt correspondant à partir du tableau/barème d'imposition.

Le deuxième exemple concerne les taux de participation à un programme – en présumant que la décision de demander ou non une prestation dans le cadre d'un programme pourrait être

liée à la prestation demandée (plus la prestation éventuellement reçue sera élevée, plus il est probable qu'une unité en fasse la demande), rechercher la probabilité que l'unité fasse une demande de prestation (ou de participation au programme), compte tenu de l'avantage qu'elle pourrait en tirer.

Le troisième exemple, le nouveau paramètre, concerne un supplément de revenu totalement hypothétique basé grosso modo sur un crédit d'impôt sur le revenu gagné aux États-Unis, mais appliqué à des gains individuels. Dans ce cas, le programme hypothétique de supplément de revenu accorde une prestation au revenu initial, jusqu'à 10 000 $ par année, au taux de 15 %, n'accorde aucune prestation aux gains de 10 000 $ à 15 000 $, puis, au-delà de 15 000 $, réduit la prestation donnée auparavant au taux de 10 % des gains au-dessus de 15 000 $, de manière qu'aucune prestation ne soit versée à des personnes gagnant 30 000 $ ou plus. Le nouveau paramètre décrit la prestation payable en fonction des gains de la personne. Les paires de coordonnées pertinentes sont donc (0, 0), (10000, 1500), (15000, 1500) et (30000, 0).

Pour ce qui est de leur spécification comme paramètres de la BDSPS, les tableaux sont très semblables aux vecteurs. La principale exception est que le tableau compte un nombre fixe de colonnes, trois, plutôt qu'une seule colonne dans le cas du vecteur. (Pour fonctionner, les tableaux utilisent les fonctions lkup1 et lkup2 de la BDSPS.) Donc, compte tenu des exceptions relativement mineures soulignées dans la présente section, l'ajout d'un tableau à une application en boîte de verre est à peu près similaire à l'ajout d'un vecteur de paramètres. Par conséquent, les prescriptions concernant les noms mnémoniques des vecteurs, les étiquetages stradd, les compilations partielles, les validations, etc. ne sont pas reprises ici.

Types de tableaux et fonctions de recherche

Il est essentiel d'établir immédiatement une distinction entre deux dichotomies pour utiliser efficacement les tableaux dans la BDSPS.

La première dichotomie concerne le type de tableaux. L'utilisateur choisit le type dans le cinquième argument de l'appel de pmaddent.

Si l'argument est P_LKPXY, alors les recherches dans le tableau s'effectuent dans le format X-Y en utilisant la première colonne (valeur x) du tableau puis la seconde colonne (valeur y); la valeur de pente de la troisième colonne (la pente des segments successifs du tableau) est présente, mais n'est pas utilisée (cette information étant redondante puisqu'elle est calculée à partir de la paire X-Y). Si le cinquième argument de pmaddent est P_LKPSL, alors les recherches dans le tableau s'effectuent dans le format de la pente en utilisant l'information de la première colonne (valeur x) et de la troisième colonne (pentes), plus la première valeur de la deuxième colonne (valeur y). Les autres valeurs de la deuxième colonne ne sont pas prises en compte puisqu'elles sont redondantes et pourraient être calculées avec les autres données contenues dans le tableau.

La deuxième dichotomie concerne l'application ou non de l'interpolation au calcul lorsque l'utilisateur effectue une recherche connexe à l'intérieur du tableau. Lorsqu'il désire utiliser l'interpolation (quand la valeur recherchée peut se situer ENTRE les entrées de la colonne des valeurs y), l'utilisateur invoque la fonction lkup1 de la bibliothèque d'algorithmes de la

BDSPS. S'il ne désire pas utiliser l'interpolation, il invoque la fonction sœur lkup2. Le Guide des algorithmes donne une description précise de ces deux algorithmes.

Présence dans les fichiers d'en-tête de la BDSPS

Comme dans le cas des vecteurs de paramètres, les paramètres définis par l'utilisateur présentés sous forme de tableaux exigent certaines entrées dans un fichier d'en-tête approprié (`Mpu.h, Cpu.h`, ou `Apu.h`).

Une de ces entrées est (habituellement) une constante manifeste qui sert à définir la longueur maximale du tableau. Le tableau de barème d'impôt fédéral (FTX) utilise la longueur maximale (FTXMAX). Le tableau de participation des célibataires prestataires de pension de retraite touchant le SRG (GISST) utilise GISSTMAX. Pour le tableau de supplément des gains, ESS, nous utilisons ESSMAX. Les définitions correspondantes (dans le fichier Mp.h pour FTXMAX et GISSTMAX, et dans le fichier `Mpu.h` pour ESSMAX) sont les suivantes :

```
#define FTXMAX   15     /* maximum of number of rows in FTX table       */
#define GISSTMAX  8     /* maximum of number of elements in GISST table */
```

et

```
#define ESSMAX    5     /* maximum number of rows in ESS schedule       */
```

La deuxième entrée est une variable dans laquelle la BDSPS stocke le nombre réel de rangées utilisées dans le tableau pour une exécution donnée. Ce nombre doit évidemment être inférieur ou égal au nombre maximal. Selon les règles de la BDSPS, les définitions des variables dans le fichier `Mp.h` qui doivent contenir le nombre réel d'éléments sont les suivantes :

```
int   GISSTrows;         /* number of rows in GISST table  */
int   FTXrows;           /* number of rows in FTX          */
```

Dans le fichier `mpu.h`, nous suivons cette règle et définissons une variable ESSrows pour le nombre réel de rangées dans ESS :

```
int   ESSrows;           /* number of rows in ESS schedule */
```

`Les fichiers Mp.h` (pour les tableaux de FTX et GISST) et `Mpu.h` (pour le tableau ESS) doivent également inclure la définition des tableaux eux-mêmes. Habituellement, ils sont définis avec des constantes manifestes établies antérieurement. La BDSPS fournit une constante, LKP_COLS, qui indique clairement son rôle dans la définition du nombre de colonnes des tableaux de recherche. Les définitions elles-mêmes sont simples :

```
NUMBER FTX[FTXMAX][LKP_COLS]; /* Federal tax table [taxable income,basic federal
tax] */
NUMBER GISST[GISSTMAX][LKP_COLS]; /* GIS take-up rate: single pensioner by benefit
level [benefit,rate] */
NUMBER ESS[ESSMAX][LKP_COLS]; /* Earnings supplement schedule [earnings, benefit
level] */
```

Présence dans les appels de pmaddent dans `Ampd.cpp`

L'utilisateur qui définit des paramètres de tableau doit modifier le fichier `Ampd.cpp` et ajouter des appels de pmaddent pour permettre à la BDSPS de mettre le paramètre à la disposition du code source réel. Nous examinons d'abord les entrées pertinentes de

pmaddent pour les tableaux existants FTX et GISST de la BDSPS.

L'exemple FTX, tiré du fichier `Mpd2.cpp`, se présente comme suit :

```
pmaddent(pcp, "FTX",     (char *)&MP.FTX[0][0],  NULL, P_LKPSL, C_NUM, 0,
FTXMAX, &MP.FTXrows, 0);
```

Remarquez que le troisième argument indique clairement que le tableau contient à la fois des rangées et des colonnes et que le cinquième argument indique qu'il s'agit d'un tableau axé sur les pentes; le huitième et le neuvième arguments utilisent les entrées de constante manifeste et de nombre réel de rangées définies dans le fichier `Mp.h`.

L'exemple du GISST, tiré du fichier `Mpd1.cpp`, se présente comme suit :

```
pmaddent(pcp, "GISST",   (char *)&MP.GISST[0][0],F_LKTUR, P_LKPXY, C_NUM, E_FRCT,
GISSTMAX, &MP.GISSTrows, 0);
```

Ici, le cinquième argument indique qu'il s'agit d'un tableau du type X-Y. Encore une fois, le huitième et le neuvième arguments utilisent les éléments définis pour le tableau dans le fichier `mp.h`.

Pour le programme hypothétique de supplément de gains, nous ajoutons au fichier `Ampd.cpp` un appel de pmaddent (copié d'un appel existant modifié, le cas échéant) semblable à ce qui suit :

```
pmaddent(pcp, "ESS", (char *)&MP.UM.ESS[0][0], NULL, P_LKPXY, C_NUM, 0, ESSMAX,
&MP.UM.ESSrows, 0);
```

Les forts parallèles avec le tableau GISST existant devraient être évidents. Remarquez toutefois les principales différences propres à un tableau de paramètres défini par l'utilisateur : le qualificatif UM dans le troisième et le neuvième arguments, la constante définie par l'utilisateur (nombre maximum de rangées) et l'adresse de variable (nombre réel de rangées) pour les huitième et neuvième arguments de pmaddent.

Emploi des références au tableau dans le code utilisateur

Les applications en boîte de verre se servent presque exclusivement des deux fonctions de recherche de la BDSPS, lkup1 et lkup2, pour faire référence aux tableaux qu'elles utilisent. Cette approche simplifie beaucoup les déclarations du code source qui utilisent les paramètres concernés. Nos trois exemples illustrent la nature de ces références.

La fonction `ATXCALC.CPP` du sous-répertoire `GLASS` est utilisée pour le calcul de l'impôt fédéral sur le revenu. Pour une personne donnée, ce calcul exige la recherche de l'impôt en fonction de son revenu imposable. L'utilisateur choisit d'appliquer ou non l'interpolation (en choisissant lkup1 ou lkup2), fournit le tableau, le nombre de rangées et la valeur x pertinente; la fonction de recherche s'occupe automatiquement du reste du traitement. Dans ce cas-ci, l'utilisateur choisit d'appliquer l'interpolation à un tableau. Le code source utilisé est semblable à ce qui suit :

```
if (isnzero(in->im.imitax)) {
/* calculate federal tax    */
in->im.imfedtax = (NUMBER) lkup1(MP.FTX, MP.FTXrows, in->im.imitax);
DEBUG2("%s fedtax =%.2f\n", in->im.imfedtax);
}
```

La fonction AGIS.CPP du sous-répertoire GLASS calcule les prestations de SRG. Ce calcul exige la recherche, en fonction des prestations payables possibles, de la probabilité que l'unité adhérera au programme de prestations (c.-à-d. qu'elle en fera la demande). Dans ce cas-ci, l'utilisateur choisit de ne pas invoquer l'interpolation - le taux de participation désiré est celui qui figure sur la dernière rangée dans laquelle la prestation éventuelle est au moins aussi élevée que la valeur x de la rangée. L'utilisateur fournit le tableau, le nombre réel de rangées et les prestations de SRG possibles, et la fonction de recherche retourne la probabilité de participation. (Le tableau lui-même se trouve à la sous-section suivante.) L'expression servant à vérifier la probabilité de participation est semblable à ce qui suit

```
lkup2(MP.GISST, MP.GISSTrows, (double) gis))
```

Pour illustrer le supplément de gains, présumons que l'utilisateur a attribué la définition appropriée de gains d'une personne à une variable (double) nommée iearn. Alors, l'expression de recherche du supplément de gains correspondant pour la personne est la suivante :

```
lkup1(MP.UM.ESS, MP.UM.ESSrows, iearn)
```

Remarquez qu'il faut utiliser le qualificatif UM pour indiquer que la valeur ESS est un tableau défini par l'utilisateur.

Présence dans les fichiers de paramètres

Comme c'est le cas pour tous les autres paramètres, l'utilisateur a la responsabilité de définir les paramètres de tableau dans le fichier de paramètres approprié (.MPI, .CPI, .API ou à la volée). Parallèlement à la spécification d'un vecteur de paramètres, la première ligne fournit le nom du paramètre et le nombre de rangées, ainsi qu'un commentaire identifiant le paramètre. Les rangées suivantes du tableau incluent les valeurs x et y, et les triplets de la pente. L'insertion entre parenthèses des articles redondants (non utilisés pour le calcul) est la seule caractéristique qui n'est probablement pas évidente.

Le tableau FTX axé sur la pente décrit l'impôt à payer (avant la réforme fiscale) en fonction du revenu imposable :

```
FTX       10          #  Federal  tax
                      table
          0       0        0.150
          40726   (6109)   0.220
          81452   (15069)  0.260
          12626   (26720)  0.290
          4
```

Le tableau GISST du type X-Y décrit les probabilités de participation en fonction du montant des prestations de SRG disponibles. L'utilisation de la fonction lkup2 dans le tableau indique que les taux de participation subissent une croissance abrupte aux niveaux des prestations clés.

```
GISST   5             # GIS take-up rate: single pensioner by benefit
                      level
```

```
        0               0.365           (0.0009)
        169             0.510           (0.0006)
        419             0.660           (0.0003)
        919             0.820           (0.0001)
        3169            1.000           (0.0001)
```

Le tableau ESS de type X-Y décrit la prestation de supplément de gains en fonction des gains de la personne; il est utilisé avec la fonction lkup1 puisqu'on a opté pour l'interpolation.

```
ESS     4               # Hypothetical earnings supplement schedule
        0               0               (0.15)
        10000           1500            (0.00)
        15000           1500            (-0.10)
        30000           0               (0.00)
```

Étapes clés de l'ajout de paramètres de tableau

La plupart des étapes clés concernant les paramètres de tableau sont identiques à ceux des paramètres de vecteur.

1. Modifier le fichier d'en-tête pertinent de manière à inclure une constante manifeste pour le nombre maximum de rangées, une variable à nombre entier pour stocker le nombre réel de rangées, et la définition du tableau lui-même.

2. Modifier le fichier `Ampd.cpp` de manière à inclure les appels de pmaddent et de stradd appropriés, généralement importés d'ailleurs puis modifiés.

3. Fournir le tableau dans un fichier de paramètres ou un fichier d'inclusion de paramètres approprié, et ne pas oublier de valider l'ajout.

Deux autres étapes clés sont propres aux paramètres de tableau.

1. S'assurer, dans le fichier de paramètres, que la colonne des valeurs x du tableau contient des valeurs strictement classées en ordre ascendant.

2. Ne pas oublier de « marquer » les valeurs redondantes dans le tableau en les mettant entre parenthèses.

AJOUT DE MATRICES DE PARAMÈTRES

Dans certaines situations très précises touchant des groupes de paramètres, il arrive que même les vecteurs ou les tableaux de paramètres ne conviennent pas vraiment. Par exemple, plutôt que d'utiliser en parallèle plusieurs vecteurs d'égale longueur, il peut s'avérer beaucoup plus efficace d'effectuer une recherche dans une matrice de valeurs. La conception de la BDSPS permet de définir et d'utiliser des matrices, même si elle limite à deux le nombre de dimensions (rangées et colonnes). Cette section décrit l'utilisation des matrices de paramètres en utilisant deux exemples; le premier est tiré de la version boîte noire de la BDSPS, et l'autre exige la spécification d'une nouvelle matrice de paramètres définie par l'utilisateur. Compte tenu de l'étroite relation entre les vecteurs de paramètres et les matrices de paramètres, la présente section ne souligne aucun aspect spécifique.

L'illustration en boîte noire utilise la matrice CTPRST qui concerne les capacités de traitement de la taxe à la consommation de la BDSPS. Ce paramètre fournit une grande matrice de facteurs (48 éléments de consommation (rangées) par 10 provinces (colonnes)) utilisés pour le calcul de la taxe de vente provinciale.

Le second exemple, dans lequel l'utilisateur ajoute à la BDSPS une nouvelle matrice de paramètres, utilise une matrice de niveaux de seuil de revenu pour une mesure expérimentale (hypothétique) de la pauvreté. Afin de faciliter la classification de l'état de pauvreté des familles, l'utilisateur désire se servir d'une matrice indiquant les seuils pertinents en fonction de variables à nombre entier qui précisent les structures des familles (rangées) et la taille du lieu de résidence (colonnes). Ainsi, l'entrée (3,2) de la matrice inclut le seuil de pauvreté d'une famille dont l'index de structure est trois et l'index de la taille du lieu de résidence est deux. L'utilisateur a choisi de nommer sa matrice EPMCO (seuils expérimentaux de mesure de la pauvreté). Aux fins de cet exemple, nous présumons que l'utilisateur choisit une mesure définie en fonction de 18 structures de familles (combinaisons de nombre de membres de la famille et de leur âge) et de quatre catégories de taille de lieux de résidence.

Présence dans le fichier `Mpu.h`

Si nous examinons d'abord l'exemple de la matrice de la boîte noire, il n'y a rien d'étonnant à ce que l'information d'en-tête pertinente pour CTPRST se trouve dans le fichier `Mp.h` du sous-répertoire `DEFS`. Le fichier inclut donc une définition de nombre entier qui permet de définir le nombre réel de rangées (éléments de consommation) CTNUMCOM, de la façon suivante :

```
int CTNUMCOM;    /* number of rows for commodity dimension parms     */
```

Le fichier inclut en plus une définition de la matrice elle-même :

```
NUMBER CTPRST[NUMCOM][NUMREG];    /* Provincial retail sales tax [com x prov] */
```

Cependant, le fichier `Mp.h` ne contient pas de constantes manifestes pour les dimensions de la matrice (NUMCOM et NUMREG) puisqu'elles sont étroitement liées à la conception de la fonction de taxe à la consommation dans la BDSPS et qu'elles ont été définies ailleurs pour que le module de taxe à la consommation puisse les utiliser plus facilement.

Quant à la matrice expérimentale de seuils de mesure de la pauvreté, il va de soi que nous devons fournir l'information « de définition » pertinente à la BDSPS en insérant dans le fichier `Mpu.h` les entrées obligatoires suivantes : (1) les constantes manifestes pour les dimensions, (2) une variable pour le nombre réel de rangées et (3) la matrice elle-même. Les lignes correspondant à ces éléments dans le fichier`Mpu.h` sont semblables à ce qui suit :

```
#define EPMFAMMAX   18    /* maximum of number of family structures (rows) for EPMCO matrix  */

#define EPMSIZE     4    /* number of size of place of residence categories for EPMCO matrix */

int EPMCOrows;   /* number of rows for EPMCO matrix    */
NUMBER EPMCO[EPMFAMMAX][EPMSIZMAX]; /* experimental poverty measure cutoffs [fam x size] */
```

Présence dans le fichier `Ampd.cpp`

En plus du besoin de vecteurs de paramètres, la BDSPS requiert pour chaque matrice de

paramètres un appel de pmaddent afin de mettre les valeurs des paramètres à la disposition du code source de l'utilisateur.

Dans l'exemple en boîte noire, cet appel, qui se trouve dans le fichier `Mpd4.cpp`, est semblable à ce qui suit (évidemment, il existe un appel de stradd correspondant) :

```
pmaddent(pcp, "CTPRST", (char *)MP.CTPRST, NULL, P_TBL, C_NUM, E_FIXL,
NUMCOM, &MP.CTNUMCOM, NUMREG);
```

À ce stade-ci, les seuls arguments qui nous intéressent sont l'entrée P_TBL pour le cinquième argument (Agg_Type) et l'entrée NUMREG pour l'argument final (nombre de colonnes). Les huitième et neuvième entrées (maximum et adresse du nombre réel de rangées) sont exactement telles que prévues compte tenu des descriptions précédentes des vecteurs et des tableaux.

Si nous examinons l'exemple de mesure de la pauvreté en boîte de verre, nous savons qu'il faut ajouter un appel de pmaddent dans le fichier `Ampd.cpp` afin de permettre à la BDSPS de donner au code source de l'utilisateur l'accès à la matrice de paramètres. Cet appel est semblable à ce qui suit :

```
pmaddent(pcp, "EPMCO", (char *)MP.UM.EPMCO, NULL, P_TBL, C_NUM, E_NONE, EPMFAMMAX, &MP.UM.EPMCOrows,
EPMSIZE);
```

On présume que l'utilisateur ajoutera aussi au fichier `Ampd.cpp` un appel de stradd afin de permettre à la BDSPS de produire l'information documentaire appropriée.

Référence aux éléments de la matrice dans le code source

La référence aux éléments de la matrice de paramètres est facile. Si on présume que la variable i contient la catégorie d'élément de consommation (nombre entier) et la variable j, le code de la province (nombre entier), alors le facteur de récupération correspondant de cette combinaison est le suivant :

```
MP.CTPRST[i][j]
```

De même, si la variable à nombre entier fstruct contient le code de structure de la famille et la variable à nombre entier sizecode, la catégorie de la taille du lieu de résidence, alors le seuil expérimental de mesure de la pauvreté pour la combinaison structure/taille est exprimé comme suit :

```
MP.UM.EPMCO[fstruct][sizecode]
```

Le facteur principal dont il faut tenir compte dans ces références est la règle du langage C qui stipule que chaque dimension commence par un élément zéro; par exemple, le tableau de 18 entrées sur 4 utilise des indices de 0 à 17 et de 0 à 3, respectivement. Un utilisateur doit décider du meilleur compromis entre l'utilisation, comme indices dans les matrices, de nombres entiers « naturels et positifs », et l'économie d'un bloc de mémoire fixe pour les paramètres définis par l'utilisateur (incluant les variables d'adresse de rangée requises).

Présence dans les fichiers de paramètres

Comme c'est le cas pour les autres formes de paramètre, l'utilisateur doit fournir des valeurs de paramètre. Habituellement, on attribue ces valeurs par des entrées dans les fichiers

d'inclusion de paramètres appropriés (p. ex., `.MPI, .CPI ou .API`) ou à la volée dans le cas des utilisateurs de SPSM Visuel . Pour les matrices de paramètres, une entrée de fichier de paramètres comprend une première ligne, qui précise le nom du paramètre et le nombre réel de rangées et, normalement, un commentaire de documentation. Les lignes suivantes constituent les rangées de la matrice. Dans l'exemple, nous fournissons seulement la première ligne d'identification et les premières des lignes de valeurs numériques.

Entrées de fichier de l'exemple en boîte noire :

```
CTPRST        40                  # Provincial retail sales tax
   0.01326 0.01326 0.01326 0.01326 0.01316 0.01406 0.02242 0.00626 0.00010 0.00550
   0.15257 0.15257 0.15257 0.15257 0.13057 0.24354 0.15684 0.13914 0.00013 0.29100
   0.17538 0.17538 0.17538 0.17538 0.16338 0.22635 0.13837 0.08953 0.00010 0.00605
   0.08125 0.08125 0.08125 0.08125 0.08424 0.07750 0.06300 0.08521 0.00009 0.07406
   0.08029 0.08029 0.08029 0.08029 0.07239 0.06953 0.05715 0.07306 0.00010 0.06512
   0.08293 0.08293 0.08293 0.08293 0.06684 0.05282 0.05581 0.00305 0.00008 0.06866
   0.00296 0.00296 0.00296 0.00296 0.00359 0.00197 0.00130 0.00171 0.00001 0.00141
   0.00997 0.00997 0.00997 0.00997 0.00934 0.00753 0.01018 0.01073 0.00024 0.01057
   0.00886 0.00886 0.00886 0.00886 0.01140 0.01421 0.00969 0.00879 0.00022 0.01017
   0.08363 0.08363 0.08363 0.08363 0.06777 0.00206 0.02368 0.04331 0.00004 0.00662
   0.08283 0.08283 0.08283 0.08283 0.35376 0.00201 0.02646 0.00544 0.00004 0.02263
   0.09406 0.09406 0.09406 0.09406 0.06143 0.00733 0.01685 0.01645 0.00064 0.02582
   0.08515 0.08515 0.08515 0.08515 0.07698 0.09175 0.07097 0.06762 0.00011 0.08368
   0.08160 0.08160 0.08160 0.08160 0.09371 0.08702 0.06739 0.06646 0.00008 0.07739
   0.08086 0.08086 0.08086 0.08086 0.08141 0.08654 0.06925 0.06538 0.00009 0.07740
   0.08238 0.08238 0.08238 0.08238 0.08320 0.08203 0.06751 0.05395 0.00011 0.07746
   0.08331 0.08331 0.08331 0.08331 0.09420 0.01711 0.07477 0.01461 0.00009 0.01935
   0.00067 0.00067 0.00067 0.00067 0.00054 0.00464 0.00740 0.00678 0.00006 0.00690
   0.05967 0.05967 0.05967 0.05967 0.05408 0.04822 0.02270 0.01925 0.00017 0.01865
   0.00821 0.00821 0.00821 0.00821 0.01031 0.00618 0.00623 0.00397 0.00011 0.00738
   0.00043 0.00043 0.00043 0.00043 0.00034 0.00124 0.00145 0.00173 0.00002 0.00059
   0.01581 0.01581 0.01581 0.01581 0.00875 0.10256 0.01323 0.00799 0.00025 0.01145
   0.02112 0.02112 0.02112 0.02112 0.02389 0.04246 0.03516 0.00786 0.00013 0.01465
   0.07207 0.07207 0.07207 0.07207 0.06970 0.08270 0.07019 0.04924 0.00005 0.10050
   0.07667 0.07667 0.07667 0.07667 0.07584 0.08081 0.06841 0.03319 0.00014 0.04053
   0.14145 0.14145 0.14145 0.14145 0.14506 0.01002 0.00841 0.00897 0.00012 0.01248
   0.04574 0.04574 0.04574 0.04574 0.04843 0.08112 0.03185 0.02851 0.00021 0.02790
   0.03739 0.03739 0.03739 0.03739 0.04921 0.01000 0.02035 0.01185 0.00019 0.01653
   0.08336 0.08336 0.08336 0.08336 0.08897 0.07353 0.06346 0.06354 0.00003 0.04449
   0.07581 0.07581 0.07581 0.07581 0.08182 0.07966 0.05424 0.06289 0.00007 0.07054
   0.07746 0.07746 0.07746 0.07746 0.08965 0.04561 0.05949 0.03563 0.00009 0.04247
   0.04765 0.04765 0.04765 0.04765 0.04967 0.02692 0.02058 0.02111 0.00016 0.01419
   0.00489 0.00489 0.00489 0.00489 0.00411 0.00745 0.00795 0.00733 0.00017 0.00929
   0.08402 0.08402 0.08402 0.08402 0.11465 0.08444 0.06428 0.06551 0.00008 0.07433
   0.07875 0.07875 0.07875 0.07875 0.07826 0.08018 0.07052 0.06623 0.00015 0.07777
   0.04826 0.04826 0.04826 0.04826 0.04245 0.00867 0.00918 0.00758 0.00008 0.01028
   0.06598 0.06598 0.06598 0.06598 0.07010 0.05898 0.07703 0.01556 0.00707 0.02343
   0.02430 0.02430 0.02430 0.02430 0.02547 0.02539 0.00705 0.00708 0.00018 0.01004
   0.01002 0.01002 0.01002 0.01002 0.01255 0.00805 0.00822 0.00735 0.00029 0.01300
   0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
```

Entrées de fichier de l'exemple de mesure de la pauvreté :

```
EPMCO         18          # Experimental poverty measure cutoffs
              5600.0
              6210.0
              6530.0
              7050.0
```

RÉSUMÉ/CONCLUSION

Pour conclure, il convient de souligner, sans les redévelopper, les points clés principaux de l'ajout à un modèle de paramètres scalaires inhabituels et de paramètres non scalaires. Ainsi, nous présumons que l'analyste suit les procédures générales décrites pour les paramètres scalaires, c'est-à-dire qu'il travaille avec des COPIES de tous les fichiers pertinents et qu'il

apporte toutes les modifications dans le sous-répertoire de travail réservé à l'analyse en cours. Nous présumons également que l'utilisateur a mis l'environnement de projet à jour et qu'il utilise les « listes de vérification » appropriées fournies pour les paramètres inhabituels.

1. Nous recommandons la « copie » comme approche générale pour effectuer le travail. Tout au long du chapitre, nous avons donné des exemples concrets des éléments qu'un utilisateur peut utiliser comme gabarits. Rarement l'utilisateur aura-t-il besoin d'utiliser tous les éléments détaillés des fichiers `Mpu.h/Cpu.h/Apu.h` (définitions, constantes manifestes pour le nombre maximum de rangées et nombre réel de rangées) et `Ampd.cpp` (pmaddent et stradd).

2. L'utilisateur expert peut souhaiter être au fait des « services » spéciaux qu'offrent les arguments de pmaddent : capacité de préciser les formats d'impression et d'effectuer le contrôle de vérification, et nombre maximum permis de rangées ou d'options.

3. Les vecteurs peuvent parfois s'avérer beaucoup plus efficaces que plusieurs paramètres scalaires individuels. La BDSPS offre cette capacité, mais l'utilisateur doit s'assurer de fournir l'information supplémentaire dans l'appel de pmaddent, de prévoir une autre variable pour le nombre de rangées pertinents ainsi qu'une constante pour les dimensions. Nous avons offert plusieurs gabarits possibles pour faciliter cette approche.

4. À de nombreux égard, les tableaux représentent un cas de vecteurs spécial que l'on applique lorsque l'on a besoin d'effectuer, à partir d'une relation fixe, une recherche de valeurs y en fonction de valeurs x.

5. On peut également utiliser des matrices (bidimensionnelles). Il faut alors fournir une information supplémentaire, le nombre de colonnes; toutefois, cette méthode peut s'avérer beaucoup plus efficace que la manipulation de plusieurs vecteurs parallèles. Encore une fois, la méthode copier-modifier est recommandée.

Développement en boîte de verre : ajout de nouvelles variables

Le présent chapitre décrit la façon d'ajouter des variables dépendantes définies par l'utilisateur à une application en boîte de verre de la BDSPS. Il illustre la façon de relever des défis, comme ceux que pose l'exemple de démarrage rapide à l'utilisateur qui aimerait compter sur une variable distincte pour l'hypothétique crédit d'impôt sur les revenus salariaux. La disponibilité de variables dépendantes est encore plus importante si l'utilisateur modélise un nouveau programme, par exemple un supplément de gains qui ne peut facilement être combiné à une variable dépendante d'un modèle existant.

La structure du chapitre permet d'aborder toutes les étapes et les questions importantes concernant l'ajout de nouvelles variables dépendantes dans un modèle. Le chapitre inclut une introduction générale au processus et une section qui définit les principaux types de variables que l'utilisateur peut souhaiter ajouter. Suit un exemple de la fonction critique vardef qui établit les liens entre le code source de l'utilisateur et le reste de la BDSPS; il décrit également la façon d'utiliser la fonction stradd pour rendre l'étiquetage des nouvelles variables accessible à l'ensemble de la BDSPS. Le chapitre présente ensuite une extension

de l'exemple de crédit d'impôt sur les revenus salariaux utilisé en mode démarrage rapide et qui définit les nouvelles variables qui sont mises à la disposition des nombreuses fonctions de sortie de la BDSPS. Finalement, il inclut des exemples de modification du code source que l'utilisateur doit effectuer ainsi que des descriptions de la compilation et de la validation du modèle obtenu.

APERÇU DE L'AJOUT DE VARIABLES

En résumé, les principales étapes de l'ajout de nouvelles variables sont les suivantes :

1. Décider des nouvelles variables dépendantes requises, choisir des noms et des descriptions qui leur conviennent, et copier tous les fichiers de code source et d'en-tête concernés dans un sous-répertoire où le nouveau modèle sera construit.

2. Apporter les changements pertinents à l'environnement de projet (identifier tous les fichiers de code source appropriés associés aux nouvelles variables dépendantes) et mettre à jour le fichier `Adrv.cpp` (fournir les chaînes de texte documentaire).

3. Apporter les changements nécessaires aux fichiers `vsu.h` et `vsdu.cpp` pour rendre les nouvelles variables dépendantes accessibles à l'ensemble du modèle de la BDSPS qui sera créé.

4. Fournir le nouveau code source (dans des modules nouveaux ou existants) pour le calcul des valeurs des nouvelles variables dépendantes.

5. Compiler le nouveau modèle et le valider afin d'en vérifier l'exactitude.

Les étapes ci-dessus ne constituent évidemment qu'un simple aperçu. La section portant sur l'ajout de paramètres et la section de récapitulation fournissent une description beaucoup plus complète du processus global de création de modèles. Et le présent chapitre porte essentiellement sur les détails particuliers qui concernent l'ajout de nouvelles variables dépendantes.

CARACTÉRISTIQUES ET TYPES DE VARIABLES DÉPENDANTES

La BDSPS donne à l'utilisateur la possibilité de créer trois types distincts de variables dépendantes définies par l'utilisateur. Il s'agit de trois types de variable scalaire. La BDSPS ne permet pas de définir des variables dépendantes de vecteurs et de matrices. Les types de variable sont les suivants :

1. Analyse numérique – Type le plus courant de variable dépendante définie par l'utilisateur. Il consiste en une valeur numérique (à virgule flottante) qui sert de variable d'analyse, par exemple, une variable totalisée comme entrée de cellule dans le paramètre de commande XTSPEC. Un bon exemple de ce type de variable est la valeur d'une nouvelle prestation en fonction du revenu qui peut être versée à une famille.

2. Analyse à nombre entier – Moins fréquemment utilisé, ce type de variable dépendante consiste en un nombre entier (int) qui sert de variable d'analyse. Ce type de variable est principalement utilisé pour l'exportation en format SAS puisqu'une variable à nombre

entier occupe moins d'espace qu'une variable d'analyse numérique. Des exemples de ce type de variable incluent les nombres minimum et maximum de semaines pendant lesquelles une famille peut ne gagner aucun revenu au cours de l'année (tel que déduit à partir des variables de la population active pour les membres de la famille, p. ex., les semaines sans travail et à la recherche de travail).

3. Classe à nombre entier – Ce type de variable dépendante consiste en un nombre entier (int) qui sert de variable de classe, p. ex., pour définir les catégories d'une variable de classe dans le paramètre XTSPEC. Ce type de variable est particulièrement intéressant lorsque sa valeur représente des catégories purement nominales, p. ex., une classe de famille par types.

Quelques autres caractéristiques des variables dépendantes définies par l'utilisateur, individuellement et collectivement, ont une importance considérable pour l'utilisateur du mode boîte de verre.

En premier lieu, toutes les variables dépendantes définies par l'utilisateur sont définies au niveau individuel. L'utilisateur doit donc prendre soin d'attribuer des valeurs aux « bonnes » personnes de manière que, lorsque l'unité d'analyse se situe à un niveau supérieur (p. ex., au niveau de la famille de recensement), les algorithmes de cumul de la BDSPS produisent les résultats désirés.

En second lieu, l'espace alloué pour ces variables peut contenir environ 50 variables. Tout dépassement de cette limite peut entraîner des erreurs obscures qu'il sera difficile de repérer.

FONCTIONS VARDEF ET STRADD ET LEURS ARGUMENTS

Les fonctions vardef et stradd sont absolument essentielles à la capacité de créer de nouvelles variables définies par l'utilisateur et de faire en sorte qu'elles soient utilisées correctement dans l'ensemble de la BDSPS. L'information communiquée par les appels de ces fonctions permet à l'ensemble de la BDSPS de connaître la nature des nouvelles variables et du texte documentaire qui les accompagne. Cette section traite en premier de la fonction vardef, suivie de la fonction stradd.

La fonction vardef joue le même rôle général pour les variables définies par l'utilisateur que celui de pmaddent pour les paramètres définis par l'utilisateur. Il y a un appel de vardef pour chaque variable que l'utilisateur définit. La fonction vardef définit les caractéristiques de la nouvelle variable de manière que la BDSPS puisse utiliser la même structure de variables que celle de sa propre base de données et de ses propres variables d'analyse et de classe. Les appels de vardef sont toujours effectués dans la fonction `vsdu.c`. La courte description suivante des arguments de la fonction se trouve près de la ligne 100 de cette fonction :

```
*    vardef("_uvew",       <= the name of the variable, quoted, with '_'
*           IN,            <= home structure (leave at 'IN')
*           im.uv.ew,      <= variable location (always in im.uv)
*           C_INT,         <= C-type (C_INT or C_NUM)
*           V_CLAS         <= type of variable (V_CLAS or V_ANAL)
*           );
```

Nous décrivons la nature des arguments de vardef un à la fois, dans l'ordre. Les sections subséquentes de ce chapitre incluent des illustrations précises de l'utilisation des deux

fonctions vardef et stradd.

Argument « Name » de vardef (et définition du nom de la souche de la variable)

Le premier argument inclut le nom de la variable dans une chaîne de texte entre guillemets doubles. L'utilisateur doit toujours inclure un caractère de soulignement comme premier caractère après les guillemets doubles d'ouverture, suivi des caractères « uv », pour indiquer le statut « variable d'utilisateur ». Le reste du nom, c'est-à-dire tout ce qui suit le préfixe « _uv », est désigné souche du nom de la variable. En général, la partie souche devrait être la plus significative et mnémonique que possible.

Argument « Home Structure » de vardef

Le deuxième argument indique que la structure dans laquelle la nouvelle variable réside. Puisque les variables définies par l'utilisateur sont TOUJOURS définies au niveau de la personne, l'utilisateur doit toujours entrer cet argument sous la forme « IN » (sans guillemets).

Argument « Variable Location » de vardef

Le troisième argument indique l'endroit où se trouve la variable (par rapport aux structures de données de la BDSPS). L'emplacement est précisé par trois éléments, dont deux invariables. Plus précisément, la première partie de l'endroit contient TOUJOURS « im.uv » (sans guillemets). Cette information indique à la BDSPS que la nouvelle variable est à l'intérieur de la partie variable d'utilisateur (uv) de la structure im (variables de modèle du niveau individuel). La partie finale de la spécification est la souche du nom de la nouvelle variable, telle que définie ci-dessus pour le premier argument.

Argument « Type-C » de vardef (C_NUM & C_INT)

Le quatrième argument précise le type de variable dans le langage C. Il prend l'une ou l'autre de deux valeurs. Les variables d'analyse numérique utilisent l'entrée « C_NUM » (sans les guillemets). Les variables de classe et d'analyse à nombre entier utilisent la valeur « C_INT » (sans les guillemets).

Argument « Usage » (Type) de vardef (V_ANAL & V_ Clas)

Le cinquième et dernier argument indique si la BDSPS doit traiter la variable comme variable d'analyse (totalisable) ou variable de classe (catégorie). Il prend l'une ou l'autre de deux valeurs. Les variables d'analyse numérique et à nombre entier utilisent l'entrée « V_ANAL » (sans les guillemets). Les variables de classe à nombre entier utilisent la forme « V_CLAS » (sans les guillemets).

La combinaison des quatrième et cinquième entrées indique à la BDSPS le nombre d'octets de mémoire qu'elle doit attribuer aux variables. Tel qu'indiqué ci-dessus, il faut prévoir six octets pour une variable d'analyse numérique, trois pour une variable d'analyse à nombre entier et un pour une variable de classe à nombre entier.

Nous avons déjà étudié des applications simples de la fonction stradd quand nous avons discuté de la documentation des paramètres définis par l'utilisateur. Cette fonction joue le même rôle ici, mais de manière plus sophistiquée; elle sert à définir à la fois une courte

description des variables d'utilisateur elles-mêmes et, dans le cas spécial des variables de classe à nombre entier et des variables d'analyse à nombre entier, la gamme de valeurs et les étiquettes associées à des valeurs particulières des variables. Le fichier `vsdu.cpp` inclut près de la ligne 110 une courte documentation des utilisations de la description des variables et des étiquettes de valeur.

```
*    stradd("uvew",        <= the name of the variable, quoted
*           "Region"       <= a printing label for the variable
*           );
**    stradd("ew",          <= the stem name of the variable, quoted
*           "\tEast\tWest" <= string containing a label for each valid
*           );                level, preceded by a tab '\t' character.
```

Comme dans le cas de la fonction vardef ci-dessus, nous poursuivons l'analyse des arguments dans l'ordre. La tâche ici est complexe puisque le NOMBRE D'APPELS et la structure des arguments de stradd dépendent du type de variables pour lesquelles la fonction est invoquée. Heureusement, il n'y a toujours que deux d'arguments pour cette fonction. Nous favorisons donc la clarté plutôt que la brièveté et nous décrivons chacun des trois types (analyse numérique, analyse à nombre entier et classe à nombre entier) individuellement.

Appels de stradd pour les variables d'analyse numérique

Les variables d'analyse numérique exigent un seul appel de la fonction stradd Le premier argument précise le nom de la variable. Il est identique à celui utilisé pour le premier argument de vardef, SAUF LE CARACTÈRE DE SOULIGNEMENT HABITUEL AU DÉBUT QUI EST OMIS DANS CE CAS-CI.

Le deuxième argument de la variable d'analyse numérique est la chaîne (entre guillemets) que la BDSPS utilise pour imprimer une description de la variable.

Exemple :

```
stradd("uvnewben",   "New Hypothetical Benefit");
```

Appels de stradd pour les variables d'analyse à nombre entier

L'ajout de variable d'analyse à nombre entier exige deux appels distincts de stradd. Le premier appel définit l'étiquette pour l'ensemble de la variable. Le second définit, par un ensemble d'étiquettes pour chaque valeur entière, la gamme des valeurs de la variable.

Dans le premier appel (étiquette de variable), le premier argument précise le nom de la variable. Il est identique à celui qui est utilisé pour le premier argument de vardef, SAUF LE CARACTÈRE DE SOULIGNEMENT HABITUEL AU DÉBUT QUI EST OMIS DANS CE CAS-CI.

Dans le premier appel (étiquette de variable), le second argument est la chaîne (entre guillemets) que la BDSPS utilise lorsqu'elle besoin d'une description pour l'ensemble de la variable, par exemple pour la documentation d'un tableau.

Dans le second appel (étiquette de valeur), le premier argument est la souche de nom de la variable; le caractère de soulignement et la chaîne de début « uv » ne doivent pas être présents.

Dans le second appel (étiquette de valeur), le second argument est une chaîne entre guillemets qui indique à la BDSPS le nombre de catégories pertinentes. La chaîne est formée de la répétition du gabarit « tx », où x varie toujours entre 0 et « un moins le nombre total de catégories». Ainsi, pour une variable qui compte quatre catégories, le second argument prend la forme « t0\t1\t2\t3 ». La notation /t est la façon dont le langage C indique normalement un caractère de tabulation.

Exemple :

```
stradd("uvnputpp",   "Number persons unemployed 2+ periods");
stradd("nputpp",       "\t0\t1\t2\t3\t4");
```

Appels de stradd pour les variables de classe à nombre entier

L'appel de stradd pour les variables de classe à nombre entier est identique à celui des variables d'analyse à nombre entier, À UNE EXCEPTION CRITIQUE PRÈS. Dans le second appel (étiquette de valeur), le second argument est une chaîne entre guillemets qui fournit les étiquettes textuelles pour les nombreuses catégories de la variable. En fait, le nombre d'étiquettes fournies par l'utilisateur correspond aux nombres entiers entre 0 et le « nombre de catégories moins un » du second appel de stradd pour une variable d'analyse à nombre entier. Par exemple, les étiquettes pour la documentation de « région » peuvent être semblables à ce qui suit :

```
\tAtlantic\tQuebec\tOntario\tPrairies\tBritish Columbia
```

Ces étiquettes, qui peuvent contenir des espaces vides encastrés (puisque le caractère de tabulation sert de délimiteur) sont utilisées lorsque l'utilisateur se sert de la fonction de tableaux croisés de la BDSPS ou exporte la nouvelle variable dans un fichier SAS.

Exemple :

```
stradd("uvfamcat",   "Nominal Family Income Category");
stradd("famcat",     "\tVery Poor\tPoor\tNear Poor\tNon-Poor\tRich");
```

En plus des définitions descriptives des arguments de vardef et de stradd, que l'on retrouve près des lignes 100 à 115 de la fonction `vsdu.cpp`, cette dernière inclut également des combinaisons de gabarits d'appel de vardef et de stradd pour les trois types de nouvelles variables. Comme c'est habituellement le cas avec la BDSPS, l'utilisateur peut modifier des copies de ces gabarits lorsqu'il définit de nouvelles variables. Les gabarits figurent près des lignes 125 à 145 de la fonction `vsdu.cpp.`

```
*  ------------------
*  A numeric variable:
*  ------------------
vardef("_xxxxxxxx",  IN, im.uv.xxxxxxxx,    C_NUM, V_ANAL);
stradd("xxxxxxxx",   "Variable label");

*  ------------------------------------------------------
*  An integer analysis variable, with values 0 through 4:
*  ------------------------------------------------------
vardef("_yyxxxxxx",  IN, im.uv.yyxxxxxx,    C_INT, V_ANAL);
stradd("yyxxxxxx",   "Variable label");
stradd("xxxxxx",     "\t0\t1\t2\t3\t4");
*  ------------------------------------------------
*  An integer class variable, with values 0 through 4:
*  ------------------------------------------------
vardef("_yyxxxxxx",  IN, im.uv.yyxxxxxx,    C_INT, V_CLAS);
```

```
stradd("yyxxxxxx",   "Variable label");
stradd("xxxxxx", "\tLABEL0\tLABEL1\tLABEL2\tLABEL3\tLABEL4");
```

EXTENSION DE L'EXEMPLE DE CRÉDIT D'IMPÔT SUR LES REVENUS SALARIAUX

L'explication ci-dessus concernant l'ajout de variables définies par l'utilisateur est complète du point de vue de la définition; toutefois, il est utile d'examiner en quoi consiste les diverses étapes. Dans la présente section, nous résumons l'exemple concret qui est expliqué plus en détail dans les autres sections. Essentiellement, l'exemple est une extension de l'exemple de crédit d'impôt sur les revenus salariaux utilisé dans le démarrage rapide, puis amélioré par l'ajout de paramètre définis par l'utilisateur.

À ce stade-ci, nous visons à fournir un exemple pratique qui illustre concrètement les trois types de variables définies par l'utilisateur sans imposer au lecteur toutes les tâches courantes inévitables associées à la préparation d'un exemple entièrement nouveau. Pour atteindre cet objectif, nous n'hésitons pas à sacrifier légèrement le réalisme (quant aux pratiques et motifs institutionnels) au profit d'un exemple clair et précis.

Nous procédons à l'extension de l'exemple de crédit d'impôt sur les revenus salariaux en ajoutant trois variables définies par l'utilisateur :

1. Variable d'analyse numérique : la nouvelle variable représente le montant brut de crédit d'impôt sur les revenus salariaux reçu; nous l'appelons « uveitc » (variable utilisateur, crédit d'impôt sur les revenus salariaux). Nous attribuons cette variable aux membres concernés des familles de recensement admissibles.

2. Variable d'analyse à nombre entier : la nouvelle variable représente le nombre d'enfants de la famille de recensement qui ont l'âge prescrit pour l'admissibilité de la famille au crédit d'impôt sur les revenus salariaux. Nous appelons la variable « uvnceitc » (variable d'utilisateur, nombre d'enfants pour le crédit d'impôt sur les revenus salariaux). Nous attribuons cette variable au chef de la famille de recensement.

3. Variable de classe à nombre entier : la nouvelle variable établit la catégorie du niveau du crédit d'impôt sur les revenus salariaux reçu par les membres admissibles de la famille; nous appelons la variable « uveitclvl » (variable d'utilisateur, niveau du crédit d'impôt sur les revenus salariaux). Nous l'utiliserons principalement comme variable de catégorie pour les tableaux conçus pour les extensions du code de crédit d'impôt sur les revenus salariaux. Nous attribuons cette variable aux membres admissibles de la famille en fonction de l'âge d'admissibilité.

Au moment d'apporter les modifications et de rédiger le code nécessaire à la mise en œuvre de ces nouvelles variables définies par l'utilisateur, nous présumons que les fichiers pertinents (`Adrv.cpp, vsu.h, vsdu.cpp, Agai.cpp, SPSMGL.vcproj, SPSMGL.sln`, etc.) ont été COPIÉS dans un nouveau sous-répertoire approprié; nous présumons aussi qu'il s'appelle `GLASSEX3`, puisqu'il s'agit du troisième exemple en boîte de verre. Nous avons également besoin, en raison de la nature de l'exemple, d'un autre fichier, Amemo1.cpp, que l'on doit copier du sous-répertoire GLASS. Nous donnerons un aperçu des modifications requises plus loin.

MODIFICATIONS DES FICHIERS DE PROJET ET DU FICHIER ADRV.CPP

Pour commencer, nous incluons tous les fichiers pertinents dans le projet et remplaçons par glassex3.exe le nom du fichier exécutable dans Project: Setting: Links.

Les modifications à apporter au fichier `Adrv.cpp` sont simples et incluent a) la mise à jour des courtes descriptions textuelles du modèle et b) la sélection du fichier Agai (plutôt que gai) pour le calcul du crédit d'impôt sur les revenus salariaux. L'utilisateur doit également modifier la fonction Adrv.cpp pour qu'elle utilise Amemo1 (plutôt que memo1); nous discuterons plus loin des modifications à apporter à Amemo1.

La BDSPS affiche la première des deux descriptions dans l'écran d'ouverture pour indiquer à l'utilisateur la nature du nouveau système. Elle utilise la seconde description comme élément de documentation « `.CPR` » (paramètre de commande) produit lors de l'exécution du modèle. N'oubliez pas que le positionnement du texte (à l'écran et dans le fichier de sortie) empêche d'utiliser des descriptions d'une longueur supérieure à 20 caractères. Après l'ajout des nouvelles descriptions, la partie concernée du fichier Adrv.cpp (près de la ligne 80) est semblable à ce qui suit :

```
================ GLOBAL VARIABLE DEFINITIONS ================== */
/*global*/ char ALTNAME[IDSIZE+1] = "EITC New Vars Ex";
/* Give global string describing version of this module */
/*global*/ char FAR Tdrv[] = "EITC New Vars Ex"
#ifdef MSC
" [" __TIMESTAMP__ "]"
#endif
;
```

La ligne modifiée (près de la 165), qui indique que le pilote de rechange utilise le fichier Agai.cpp plutôt que le fichier gai.cpp, est semblable à la suivante :

```
Agai(hh);         /* compute new guarantees, refundable credits  */
```

L'utilisateur doit également modifier la ligne ci-dessous pour rediriger le pilote vers la fonction de rechange memo1 :

```
Amemo1(hh);         /* compute disposable income, etc.  */
```

Enfin, il faut compiler une version Debug dans Build:Start:Debug. Les liens requis et les compilations sont indiqués.

MODIFICATIONS DU FICHIER VSU.H

Le fichier vsu.h sert à définir en langage C la structure qui contient les variables définies par l'utilisateur. La partie pertinente du fichier, copiée du sous-répertoire SPSM\GLASS, est semblable à ce qui suit :

```
typedef struct uv_ {
        NUMBER   uvdummy;          /* dummy variable */
} uv_;
```

Nous remplaçons la ligne uvdummy par trois lignes qui définissent les nouvelles variables, uveitc, uvnceitc et uveitclvl. Ces nouvelles lignes indiquent le type des nouvelles variables. Après les changements, la nouvelle partie du fichier `vsu.h` est semblable à ce qui suit :

```
typedef struct uv_ {
    NUMBER   uveitc;    /* Earned Income Tax Credit amount */
    int      uvnceitc;  /* Number Children for EITC eligibility */
    int      uveitclvl; /* Earned Income Tax Credit level */
} uv_;
```

Remarquez les règles d'attribution de nom de fichier utilisées ici. L'instruction typedef exige que les variables soient précédées du préfixe uv, mais N'UTILISE PAS le caractère de soulignement de début utilisé dans les instructions vardef illustrées plus loin dans les modifications apportées au fichier `vsdu.cpp`.

Il n'est pas toujours nécessaire, comme nous l'avons fait ici, de modifier la version GLASS de `vsu.h`. S'il existe déjà une version (définie par l'utilisateur) du fichier `vsu.h` qui inclut des variables définies par l'utilisateur qui peuvent être conservées, il suffit de copier le fichier existant et de le modifier au besoin. N'oubliez pas la limite globale de 200 octets par personne pour les variables définies par l'utilisateur.

MODIFICATIONS DU FICHIER VSDU.CPP

Les modifications requises à la copie du fichier `vsdu.cpp` incluent les appels de vardef et de stradd pour permettre à la BDSPS d'accéder aux nouvelles variables et à leur documentation. Compte tenu de leur simplicité, nous utilisons les gabarits d'exemple au début du fichier. Nous ferons ces appels à la fin du fichier `vsdu.cpp`, juste avant l'instruction « DEBUG_OFF("vsdu"); ». Les ajouts sont les suivants :

```
/* uveitc: (Analysis) Earned Income Tax Credit amount */
   vardef("_uveitc",  IN, im.uv.uveitc,    C_NUM, V_ANAL);
   stradd("uveitc",   "Earned Income Tax Credit amount");
/* uvnceitc: (Analysis) number of children for EITC eligibility */
   vardef("_uvnceitc",  IN, im.uv.uvnceitc,    C_INT, V_ANAL);
   stradd("uvnceitc",   "# Children for EITC eligibility");
   stradd("nceitc",     "\t0\t1\t2\t3\t4\t5\t6\t7");
/* uveitclvl: (Class) Earned Income Tax Credit level */
   vardef("_uveitclvl", IN, im.uv.uveitclvl,    C_INT, V_CLAS);
   stradd("uveitclvl",  "Earned Income Tax Credit level");
   stradd("eitclvl",     "\t0 EITC\tReduced EITC\tFull EITC");
```

Remarquez le second appel de stradd pour chacune des deux variables à nombre entier et l'omission du préfixe uv dans cet appel qui définit le nombre de cas (variable d'analyse à nombre entier) ou les étiquettes de catégorie (variable de classe à nombre entier).

MODIFICATIONS DU FICHIER AGAI.CPP (OU, PLUS GÉNÉRALEMENT, DE TOUT NOUVEAU CODE SOURCE DE BASE)

Les tâches préliminaires précédentes débouchent sur la tâche principale, soit la révision du fichier `Agai.cpp` afin qu'il reflète le nouveau calcul du crédit d'impôt sur les revenus salariaux. Nous utilisons le fichier `Agai.cpp`, mais en général, à ce stade-ci, l'utilisateur est en mesure de rédiger et (ou) de modifier le code source nécessaire pour apporter les changements désirés au calcul des variables de la BDSPS, quel que soit le module visé par ces modifications. Nous illustrons les modifications apportées à l'exemple du crédit d'impôt sur les revenus salariaux une partie à la fois; nous montrons le fichier `Agai.cpp` à son état non modifié, puis après que nous lui avons ajouté les variables désirées.

Identification des chaînes

La documentation est importante. Pendant que nous modifions le fichier `Agai.cpp`, nous mettons d'abord la description à jour. Là où la version GLASS du fichier `Agai.cpp` donne la description fictive (près de la ligne 48) :

```
/*global*/ char FAR Tgai[] = "Untitled"
```

nous entrons une description plus significative :

```
/*global*/ char FAR Tgai[] = "EITC Vars Version"
```

Variables locales

Les variables intermédiaires (locales) peuvent être très utiles. Lorsque la version GLASS originale du fichier `Agai.cpp` n'utilise pas de variables locales, nous ajoutons quelques unes de ces variables dans l'exemple de démarrage rapide, tel qu'illustré ci-dessous. On recommande d'initialiser à ZÉRO les variables NUMBER ou à virgule flottante.

```
/* user-defined intermediate (local) variables in support of glass box example 3
(user-defined SPSM variables) */
   NUMBER eitc = ZERO; /* amount of earned income tax credit */
   int    nceitc;       /* number of children for EITC eligibility */
```

Calcul et attribution des nouvelles variables de modèle

Nous sommes maintenant prêts à calculer les nouvelles variables et à les attribuer aux variables appropriées de la BDSPS définies par l'utilisateur. Dans l'exemple du fichier `Agai.cpp`, nous cherchons à calculer le montant du crédit d'impôt remboursable éventuel. Nous effectuons ce calcul immédiatement après avoir défini tous les impôts fédéraux et provinciaux dans la BDSPS, mais avant que le revenu disponible ne soit attribuer comme sortie de la fonction Amemo1.

Nous conservons également la boucle initiale qui attribue à la variable imiosa la valeur 0 pour tous les membres du ménage. Nous n'avons plus besoin de cette variable puisque nous avons créé la variable spéciale uveitc pour représenter le montant du crédit d'impôt sur les revenus salariaux. Si l'utilisateur à mis en commentaire (désactivé) la boucle initiale dans les premières étapes de l'exemple du fichier `Agai.cpp`, il lui suffit de réactiver le code. S'il a supprimé la boucle dans les anciennes versions, il lui suffit de la copier de la version GLASS du fichier `Agai.cpp`.

```
void Agai(
      P_hh hh
)
{
      register P_in in;
      register int ini;
      register P_in ineld;
      register P_in inspo;
      register P_cf cf;
      register int cfi;
      int nceitc;
      NUMBER cfempinc;
      NUMBER eitc;

      DEBUG_ON("Agai");
```

```
      /* process persons in household */
      for (ini=0, in=&hh->in[0]; ini<hh->hhnin; ini++, in++) {
            in->im.imiosa = ZERO;
      }

      if (MP.UM.EITCFLAG) {
            /* process each census family in household */
            for (cfi=0, cf=&hh->cf[0];  cfi<hh->hhncf;  cfi++, cf++) {
               /* initialise elder's pointer */
               ineld = cf->cfineld;

               /* calculate elder's contribution to family net income */
               cfempinc = ineld->id.idiemp;


               if (cf->cfspoflg) {
                     DEBUG1("%s spouse present\n");
                     inspo = cf->cfinspo;    /* spouse's in pointer */

                     /* add spouse's net income to family net income */
                     cfempinc += inspo->id.idiemp;
               }

               nceitc = 0;

               /* process children in census family */
               for (ini=0, in=cf->cfinch;  ini<cf->cfnchild; ini++, in++)
{

                        if (in->id.idage > MP.UM.EITCAGE) {
                           DEBUG2("%s discarding old child, aged %d\n",
in->id.idage);
                           continue;
                        }

                        /* Count up remaining children */
                        nceitc++;
               }

               eitc = 0;
               if (nceitc > 0 ) {
                     if ( cfempinc < MP.UM.EITCTPMX ) {
                              eitc = MP.UM.EITCPIR * cfempinc;
                     }
                     else if ( cfempinc <= MP.UM.EITCTPRC ) {
                           eitc = MP.UM.EITCMAX;
                     }
                     else {
                           eitc = nneg(MP.UM.EITCMAX - ((cfempinc -
MP.UM.EITCTPRC) * MP.UM.EITCPOR));
                     }
                  }

            /* process persons in census family */
            for (ini=0, in=cf->cfin;  ini<cf->cfnpers;  ini++, in++) {
```

```
                        if (in->id.idage > MP.UM.EITCAGE) {
                              in->im.uv.uveitc = eitc;
                              if (eitc > 0) {
                                    if (eitc == MP.UM.EITCMAX) {
                        // individual assigned a maximum EITC amount
                                          in->im.uv.uveitclvl = 2;
                                    }
                                    else {
                              // individual assigned a reduced EITC amount
                                          in->im.uv.uveitclvl = 1;
                                    }
                              }
                              else {
                        // individual did not receive any EITC amount
                                    in->im.uv.uveitclvl = 0;
                              }
                        }
                        // assign the number of children under the age
cutoff that determines family eligibility
                        // to the head of the census family
                        if (in->id.idcfrh == 0) {
                              in->im.uv.uvnceitc = nceitc;
                        }
                  }
               }
         }

         DEBUG_OFF("Agai");
}
```

Modifications au fichier Amemo1.cpp

Nous devons faire un ajout au fichier `Amemo1.cpp` et inclure le crédit d'impôt sur les revenus salariaux dans la valeur imftran des transferts fédéraux, qui est également incluse dans le revenu disponible. Dans la section d'attribution de code du fichier `Amemo1.cpp`, l'attribution initiale du paramètre imftran est effectuée comme suit :

```
in->im.imftran  = in->im.imioas + in->im.imiotg
                  + in->im.imigis + in->im.imispa
                  + in->im.imfcben + in->im.imicqp
                  + in->im.imfoth + in->im.imiuib
                  + in->im.imfstc + in->im.imqtar + in->im.imiosa
                  + in->im.imfortc + in->im.imiuccbr ;
```

Nous devons ajouter au paramètre imftran la valeur de la nouvelle variable d'utilisateur « uveitc » puisque le programme de crédit d'impôt sur les revenus salariaux inclut sa propre variable spéciale dans le code de boîte de verre. Il n'était pas nécessaire de modifier le fichier `Amemo1.cpp` aux premiers stades de l'exemple puisque nous avons utilisé la variable intégrée imiosa, déjà incluse dans imftran. La nouvelle variable est donc ajoutée immédiatement après la variable « imiosa », qui est ensuite réinitialisée à 0 pour tous les membres.

```
in->im.imftran  = in->im.imioas + in->im.imiotg
                  + in->im.imigis + in->im.imispa
                  + in->im.imfcben + in->im.imicqp
                  + in->im.imfoth + in->im.imiuib
                  + in->im.imfstc + in->im.imqtar + in->im.imiosa
                  + in->im.imfortc + in->im.imiuccbr + in->im.uv.uveitc ;
```

Compilation

Nous devons faire la mise au point du modèle et vérifier s'il fonctionne correctement, puis compiler le nouveau modèle GLASSEX3.EXE.

VALIDATION

Lorsque la compilation est terminée et que le fichier GLASSEX3.EXE a été créé, l'utilisateur peut le valider pour vérifier si la logique s'exécute tel que prévu. Puisque nous avons déjà illustré la validation de manière très détaillée dans cet exemple, nous incluons ici seulement un ensemble représentatif de sorties de tableaux croisés. Au cours de ses activités quotidiennes, l'utilisateur s'assurera de l'exactitude du modèle avant d'effectuer une exécution de production des tableaux désirés.

La mini-validation effectuée ici consiste en un nouvel ensemble de tableaux utilisés pour tester les variables de la boîte de verre. Cette validation permet de tester la définition initiale du crédit, soit un crédit d'impôt de 1 200 $ pour les membres de familles de recensement admissibles, c'est-à-dire qui comptent des enfants dont l'âge se situe sous la limite d'âge prescrite. Le paramètre XTSPEC pertinent figure ci-dessous :

```
XTSPEC
IN:{uveitc, gainer:S=3, nochange:S=3} * agegrp+;
IN:empigrp+ * {uveitc, uveitc/gainer:L="Average Benefits",
               gainer:S=3, nochange:S=3};
IN:dispgrp+ * {uveitc, gainer:S=3, nochange:S=3, scfrecs};
CF:uvnceitc+ * {uveitc};
IN:uveitclvl+ * {uveitc, gainer:S=3, nochange:S=3} * agegrp+;
```

Les spécifications des 3 premiers tableaux sont semblables à celles des tableaux antérieurs de glassex1, incluant les instructions UVAR; le paramètre uveitc est remplacé par le paramètre imiosa afin d'illustrer l'utilisation d'une variable définie par l'utilisateur en guise de variable d'analyse. Il s'agit de la même utilisation que si les variables avaient fait partie de la BDSPS initiale, y compris la possibilité d'utiliser le qualificatif « + » pour indiquer l'agrégation d'une dimension de variable catégorique. Les tableaux de résultats doivent être identiques à ceux de glassex1.

Remarquez que les résultats de la boîte de verre ont été produits avec une version antérieure du modèle de la BDSPS.

```
Table 1U: Selected Quantities for Individuals by Age

                                                    Age
+-----------------------------------+--------+--------+--------+--------+
|Quantity                           | Min-20 | 21-64  | 65-Max |  All   |
```

Earned Income Tax Credit amount (M)	0.0	804.1	3.7	807.9
Received EITC Flag (Gainer) (000)	0.0	1226.7	7.4	1234.0
Unaffected by EITC Flag (000)	8146.0	17946.5	4184.0	30276.6

Table 2U: Selected Quantities for Individuals by Wages & salaries Group

Wages & salaries Group	Earned Income Tax Credit amount (M)	Average Benefits	Received EITC Flag (Gainer) (000)	Unaffected by EITC Flag (000)
Min-0	201.0	641.2102	313.4	14751.3
1-8000	246.1	594.5951	413.9	3191.6
8001-12000	143.0	1006.2308	142.1	1073.4
12001-24000	214.6	597.1696	359.3	2556.5
24001-Max	3.2	608.0466	5.2	8703.8
All	807.9	654.6415	1234.0	30276.6

Table 3U: Selected Quantities for Individuals by Base disposable income group

Base disposable income group	Earned Income Tax Credit amount (M)	Received EITC Flag (Gainer) (000)	Unaffected by EITC Flag (000)	SLID Records
Min-5000	73.7	139.3	9226.9	19744
5001-10000	94.3	151.2	2547.6	5722
10001-15000	136.1	202.1	2987.8	7607
15001-20000	173.8	237.2	2987.4	8349
20001-25000	163.0	241.8	2440.6	6443
25001-30000	72.7	118.2	2309.0	5650
30001-35000	39.2	59.9	1888.1	4544
35001-40000	17.5	27.8	1421.2	3378
40001-45000	9.5	12.0	1041.9	2378
45001-Max	28.2	44.5	3426.1	7311
All	807.9	1234.0	30276.6	71126

Le quatrième tableau illustre l'utilisation d'une variable définie par l'utilisateur en guise de variable de classification. Nous utilisons ici le qualificatif « CF: » pour produire les montants de crédit d'impôt sur les revenus salariaux versés aux familles de recensement en fonction du nombre d'enfants utilisés pour déterminer l'admissibilité de la famille. Dans le module `Agai.cpp`, nous avons attribué uvnceitc au chef de la famille de recensement et il est plus logique d'utiliser ce tableau au niveau de la famille de recensement. Le tableau

accroît la confiance en la structure de code puisque aucun crédit d'impôt sur les revenus salariauxs n'est versé aux familles de recensement qui ne comptent pas d'enfants admissibles.

Table 4U: Earned Income Tax Credit amount (M) for Census Families by # Children for EITC eligibility

# Children for EITC eligibility	Earned Income Tax Credit amount (M)
0	0.0
1	340.4
2	296.3
3	116.9
4	37.8
5	13.1
6	3.4
7	0.0
All	807.9

Le dernier tableau illustre l'utilisation d'une variable définie par l'utilisateur en guise de variable de classification à niveaux définis. Le niveau de crédit d'impôt sur les revenus salariaux est indiqué par niveau de crédit (0, réduit, complet) et au niveau individuel. Dans ce cas-ci, nous reprenons la structure du premier tableau et utilisons une variable de classification définie par l'utilisateur comme dimension supplémentaire. Cela permet de confirmer encore une fois que le code fonctionne tel que prévu. Le premier tableau de la série confirme qu'aucune prestation de crédit d'impôt sur les revenus salariaux n'est versée aux personnes dont la variable de classification « uveitclvl » contient la valeur « 0 EITC ». Le second tableau indique le nombre de personnes qui ont reçu une prestation réduite de crédit d'impôt; il ne montre aucune personne qui n'a rien reçu puisque toutes ont au moins reçu un certain montant de crédit d'impôt. Le troisième tableau inclut uniquement les personnes qui ont reçu le montant complet de crédit d'impôt. Nous constatons que la majorité des bénéficiaires sont admissibles à un montant réduit de crédit d'impôt et non à la prestation maximum de 1 200 $. Le dernier tableau est identique au tableau 1U, soit un cumul des 3 niveaux de montants de prestation de crédit d'impôt.

Table 5U: Selected Quantities for Individuals by Earned Income Tax Credit level and Age

Earned Income Tax Credit level = 0 EITC

	Age			
Quantity	Min-20	21-64	65-Max	All
Earned Income Tax Credit amount (M)	0.0	0.0	0.0	0.0
Received EITC Flag (Gainer) (000)	0.0	0.0	0.0	0.0
Unaffected by EITC Flag (000)	8146.0	17946.5	4184.0	30276.6

```
+-----------------------------------+--------+--------+--------+--------+
```

Table 5U (suite) : Selected Quantities for Individuals by Earned Income Tax Credit level and Age

Earned Income Tax Credit level = Reduced EITC

Quantity	Age Min-20	21-64	65-Max	All
Earned Income Tax Credit amount (M)	0.0	608.7	2.7	611.4
Received EITC Flag (Gainer) (000)	0.0	1063.8	6.6	1070.4
Unaffected by EITC Flag (000)	0.0	0.0	0.0	0.0

Table 5U (suite) : Selected Quantities for Individuals by Earned Income Tax Credit level and Age

Earned Income Tax Credit level = Full EITC

Quantity	Age Min-20	21-64	65-Max	All
Earned Income Tax Credit amount (M)	0.0	195.4	1.0	196.4
Received EITC Flag (Gainer) (000)	0.0	162.8	0.8	163.7
Unaffected by EITC Flag (000)	0.0	0.0	0.0	0.0

Table 5U (suite) : Selected Quantities for Individuals by Earned Income Tax Credit level and Age

Earned Income Tax Credit level = All

Quantity	Age Min-20	21-64	65-Max	All
Earned Income Tax Credit amount (M)	0.0	804.1	3.7	807.9
Received EITC Flag (Gainer) (000)	0.0	1226.7	7.4	1234.0
Unaffected by EITC Flag (000)	8146.0	17946.5	4184.0	30276.6

Une fois la validation terminée, l'utilisateur commence la production des tableaux désirés et des autres sorties.

RÉSUMÉ/CONCLUSIONS

Nous résumons les points clés de ce chapitre dans une liste de contrôle des principaux éléments requis pour l'ajout de nouvelles variables définies par l'utilisateur à un modèle de la BDSPS.

1. Planifier les modifications désirées « sur papier ». Choisir le nom des nouvelles variables et élaborer la logique qui permet de les calculer. Identifier les fichiers de code source spécifiques concernés (p. ex., `Agai.cpp, Amemo1.cpp`). Choisir un sous-répertoire

pour le nouveau modèle ou en créer un, le cas échéant.

2. Copier tous les fichiers pertinents dans le sous-répertoire de travail.
   - On doit toujours utiliser les fichiers `SPSMGL.sln, SPSMGL.vcproj, Adrv.cpp, vsu.h` et `vsdu.cpp` et les fichiers de code source pertinents tels les fichiers `Agai.cpp` et `Amemo1.cpp` utilisés dans l'exemple.
   - Les fichiers `mpu.h` et `Ampd.cpp` peuvent aussi être nécessaires si l'on doit ajouter simultanément des paramètres.
3. Mettre le `projet à jour et modifier le nom du fichier de sortie.`
4. Mettre à jour le fichier `Adrv.cpp`.
   - Insérer de brèves descriptions pertinentes pour les deux arguments de documentation (chaînes ALTNAME et Tdrv).
   - Modifier les appels de fonction de manière à invoquer les deux versions de remplacement des fonctions de calcul d'imposition/transfert, p. ex., Agai(hh) plutôt que gai(hh).
5. Mettre à jour le fichier `vsu.h`. À l'intérieur de la structure « uv_ », indiquer le type et le nom des nouvelles variables définies par l'utilisateur. Ne pas oublier d'utiliser le préfixe « uv » sans utiliser le caractère de soulignement.
6. Mettre à jour le fichier `vsdu.cpp.`
   - Pour chaque nouvelle variable, prévoir un appel de fonction vardef afin de définir la nature de la variable à la BDSPS.
   - Pour chaque nouvelle variable définie par l'utilisateur, appeler la fonction stradd pour fournir une description (chaîne de texte) de la variable.
   - Pour chaque variable à nombre entier, d'analyse ou de classification, appeler la fonction stradd une seconde fois (en utilisant juste la souche du nom) pour fournir la liste des étiquettes des valeurs entières de la variable. Ne pas oublier que les variables d'analyse indiquent seulement le nombre de catégories (de 0 à n) et que dans les variables de classe à nombre entier définies par l'utilisateur, les étiquettes contiennent du texte choisi par l'utilisateur.
7. Faire les modifications nécessaires aux sous-programmes d'imposition/transfert de base. Envisager d'utiliser des variables intermédiaires pour simplifier le travail. Prendre soin d'effectuer les initialisations appropriées et d'attribuer les valeurs calculées à la personne appropriée.
8. Compiler le nouveau modèle. Ne pas oublier de le valider avant de l'utiliser pour tout travail de production important.

Modification des variables de données de base et de variante

Le présent chapitre décrit la façon dont les utilisateurs peuvent, le cas échéant, modifier les

valeurs dans la base de données de la BDSPS pour analyser des choix stratégiques. Ces modifications sont différentes de celles apportées aux variables dépendantes, aux paramètres et à la logique du modèle décrites dans les chapitres précédents. Il s'agit ici de modifications apportées aux données d'entrée des algorithmes d'imposition/transfert plutôt qu'à logique de ces algorithmes. Les genres de modification étudiés ici sont temporaires. Ils affectent les valeurs que le modèle de l'utilisateur « voit » au cours d'une exécution particulière, mais n'ont aucune incidence sur les valeurs réelles stockées dans la BDSPS elle-même.

Habituellement, mais non exclusivement, les modifications de la base de données effectuées par l'utilisateur concernent des montants exprimés en dollars, éléments de revenu ou de déduction. L'utilisateur peut souhaiter accroître ou réduire le revenu d'une source particulière, par exemple, réduire un revenu d'intérêts pour tenir compte d'une possible chute des taux d'intérêt. Toutefois, l'utilisateur peut aussi souhaiter modifier une variable autre que celle du revenu, par exemple, la variable de fréquentation scolaire des enfants plus âgés de certaines familles.

Il existe une distinction importante dans le cas des modèles de la BDSPS qui simulent deux systèmes d'imposition/transfert (base et variante) : les modifications affectent-elles les valeurs que « voit » l'ensemble du modèle de l'utilisateur, ou uniquement celles d'un seul de ses systèmes (base ou variante). Cette distinction est importante au point que nous avons organisé la structure du chapitre pour en tenir compte. Il faut cependant noter que cette distinction ne s'applique pas aux modèles qui simulent un seul système d'imposition/transfert. La procédure recommandée ici incite l'utilisateur à opter, dans la mesure du possible, pour l'approche de système unique.

La section suivante décrit la façon de modifier les données liées à l'exécution d'un modèle immédiatement après leur lecture par la BDSPS. Évidemment, les modifications qui y sont abordés ont une incidence sur TOUS les systèmes d'imposition/transfert du modèle. La section décrit deux cas auxiliaires; dans le premier, l'utilisateur modifie les données en utilisant la fonction d'ajustement des données intégrée à la BDSPS et, dans le second, plus complexe, il crée sa propre logique d'ajustement. Ce dernier cas peut exiger la définition de nouveaux paramètres d'ajustement de données. La première section indique à quel endroit et de quelle façon apporter les modifications de système unique et inclut un exemple pratique détaillé.

La section suivante décrit les modifications qui touchent un seul système (base ou variante) au cours d'une exécution de la BDSPS. Elle donne des explications sur la façon dont l'utilisation de la fonction « fichier de résultats » de la BDSPS peut souvent permettre de convertir ce cas en « système unique » plus simple, tel que décrit ci-dessus. Toutefois, lorsqu'il est impossible ou inapproprié d'utiliser l'approche du fichier de résultats, la section prévoit également une description de l'endroit où apporter les modifications nécessaires et de la façon de s'y prendre. Elle se termine par un exemple pratique de la façon d'apporter des ajustements de bases de données du système.

MODIFICATIONS QUI TOUCHENT TOUS LES SYSTÈMES D'IMPOSITION/TRANSFERT D'UN MODÈLE

Cette section décrit la façon de modifier les données qui touchent tous les systèmes

d'imposition/transfert d'un modèle de la BDSPS. La méthode s'applique à la modification des données tant d'un modèle qui inclut un seul système d'imposition/transfert que d'un modèle à deux systèmes.

En premier lieu, la section examine les fonctions d'ajustement des données de la BDSPS qui permettent à l'utilisateur d'attribuer des valeurs à des paramètres d'ajustement existants en utilisant des fichiers API (inclusion de paramètres d'ajustement de données).

Ensuite, elle aborde l'ajout de nouveaux algorithmes d'ajustement de données. Pour ce type d'ajustement, l'utilisateur définit la nouvelle logique d'ajustement dans le fichier `adju.cpp` et définit, le cas échéant, de nouveaux paramètres en apportant des modifications aux fichiers `apu.h` et `apdu.cpp`. Il peut également souhaiter définir de nouvelles variables indépendantes pour aider à la validation du modèle.

Enfin, la section donne, pour le deuxième cas auxiliaire, un exemple pratique détaillé, suivi d'une liste de vérification pour la modification de type ajustement « global » des données.

Modification type de la croissance du revenu et de la population avec des fichiers API

La conception de la BDSPS prévoit déjà les besoins de l'utilisateur concernant l'ajustement type des données. Le sous-répertoire `\SPSD` contient un certain nombre de fichiers, dont le nom respecte la forme `BAxx_yy.APR`, qui indiquent à la BDSPS de faire vieillir les données, autres que la structure démographique sous-jacente, des années XX à YY. Par exemple, le fichier `BA09_10.APR` contient les paramètres de vieillissement qui permettent de faire vieillir les variables non démographiques de la BDSPS de 2009 à 2010.

Si la substance des paramètres de ces fichiers est appropriée aux besoins de l'utilisateur, l'ajustement des données est simple. L'utilisateur entre le nom du fichier, dont les dates correspondent « le plus à ses besoins », dans le paramètre INPAPR du fichier de paramètres de commande. Toutes les modifications nécessaires de la valeur de ces paramètres sont alors apportées soit par l'intermédiaire d'un fichier « .API » (inclusion de paramètres d'ajustement de données), soit à la volée lorsqu'on utilise MSPS Visuel.

Bien que le *Guide des paramètres* fournisse une description exacte de ces paramètres, il convient d'expliquer sommairement le contrôle étendu qu'ils permettent.

Certains paramètres précisent la façon dont les revenus imputés/convertis doivent être traités (non prise en compte, ou adoption de l'une de deux méthodes de synthèse). Un grand nombre de paramètres régissent l'« élimination » des taxes à la consommation des dépenses de la famille.

Un autre ensemble de paramètres définit les seuils de pauvreté des familles. Il permet à l'utilisateur de préciser un ensemble de « seuils de pauvreté » pour les familles économiques, les seuils particuliers variant selon la taille de la famille et celle du lieu de résidence. L'utilisateur type trouvera probablement des plus utiles le vaste ensemble de facteurs de croissance prévus pour les variables de la BDSPS contenant des valeurs en dollars : revenus, déductions et dépenses. Chacune de ces variables possède virtuellement son propre facteur de croissance.

La BDSPS fournit un mécanisme pratique de vieillissement démographique de sa population sous-jacente. Les fichiers « .WGT » du répertoire SPSD permettent aux utilisateurs d'ajuster la base de population des années modélisées.

Changements nécessitant une nouvelle logique pour le fichier `adju.cpp`

La souplesse qu'offrent les paramètres de vieillissement (« `.APR` » `et` « `.API` ») et les fichiers (« `.WGT` ») de vieillissement de la population suffit souvent à répondre aux besoins de l'utilisateur. Cependant, dans certains cas, celui-ci peut désirer ou requérir un contrôle plus direct sur les données de simulation. Les quelques exemples qui suivent indiquent la mesure dans laquelle ce contrôle est possible. Il convient de rappeler au lecteur que les exemples donnés ici mettent davantage l'accent sur le besoin de comprendre rapidement la façon d'utiliser ce type de contrôle plutôt que sur le strict respect de l'utilisation réaliste des politiques.

1. L'utilisateur peut augmenter le niveau d'éducation moyen en ajustant la variable « idedlev » de certaines personnes, ce qui peut entraîner une distribution des niveaux d'éducation atteints qui respectent l'esprit d'une quelconque prévision exogène.

2. L'utilisateur peut désirer accroître un montant de revenu ou de transfert particulier avec un facteur lié aux caractéristiques de l'unité. Par exemple, en se basant sur l'hypothèse que les portefeuilles diffèrent en fonction de l'âge et du revenu des investisseurs, un utilisateur peut ne pas désirer modéliser l'effet d'une augmentation des taux d'intérêt en haussant le revenu d'intérêt de chacun dans la même proportion. Il peut plutôt appliquer un facteur plus petit aux investisseurs susceptibles, selon lui, d'être conservateurs et (ou) dont le portefeuille se renouvelle plus lentement. Ce genre d'hypothèse traite ces familles comme si elles ne pouvaient pas bénéficier aussi rapidement de taux d'intérêt plus élevés.

3. Un utilisateur peut désirer modéliser une participation plus grande de la population active en changeant le tableau pertinent de variables de la population active pour les personnes incluses dans la BDSPS (semaine de travail, gains d'emploi rémunéré, assurance-emploi, etc.). Les modifications d'une telle variété de variables intereliées ne doivent être effectuées qu'après une planification importante et exhaustive.

4. À la limite, un utilisateur de la BDSPS expérimenté et bien informé peut même modifier la structure des ménages ou des familles en modélisant une hausse spectaculaire des naissances et en ajoutant des « enfants synthétiques » aux familles concernées dans la base de données.

La fonction `adju.cpp`, dans le sous-répertoire `\SPSM\GLASS`, permet à l'utilisateur d'ajouter une nouvelle logique de vieillissement des données aux modèles de la BDSPS. Cette fonction est appelée immédiatement après que la BDSPS a lu les données de chaque ménage et avant le calcul de toute variable de transfert ou pour mémoire. L'utilisateur peut indiquer la logique de ses propres modifications immédiatement après l'invocation « adj(hh) » que la BDSPS utilise pour effectuer son propre vieillissement des données, c'est-à-dire son mécanisme intégré d'application des paramètres de croissance de revenu précisés dans les fichiers « `.APR` » et « `.API` » pertinents.

Pour appliquer une nouvelle logique de vieillissement de données, les utilisateurs devront peut-être définir de nouvelles variables intermédiaires (variables de compteur, de pointeur, etc.) et (ou) de nouveaux paramètres d'ajustement de données personnalisés. La sous-section suivante décrit la méthode générale d'ajout de ces paramètres, incluant les modifications spécifiques connexes développées dans l'exemple pratique subséquent.

Ajout de nouveaux paramètres d'ajustement de la base de données

L'ajout de nouveaux paramètres de base de donnée définis par l'utilisateur, très similaire à celui des nouveaux paramètres de modèle, tel que décrit aux chapitres précédents, comporte toutefois quelques légères différences.

1) Les modèles de la BDSPS utilisent un seul fichier de paramètres d'ajustement de base de données (extension « .APR »); ils peuvent utiliser un ou deux fichiers de paramètres de modèle (extension « .MPR ») selon qu'ils modélisent un ou deux systèmes de transfert. 2) De ce fait, les utilisateurs fournissent les valeurs des paramètres d'ajustement d'utilisateur dans des fichiers « .API » (inclusion de paramètres d'ajustement de données), qui modifient les fichiers « .APR » standard, plutôt que dans des fichiers « .MPI » (inclusion de paramètres de modèle) qui modifient les fichiers « .MPR » standard. 3) De nouveaux paramètres d'ajustement sont définis dans le fichier `apu.h` (en-tête) plutôt que dans le fichier d'en-tête `mpu.h` utilisé pour les paramètres de modèle. 4) De la même manière, les appels de fonction qui mettent les paramètres à la disposition du reste du modèle sont effectués dans le fichier `apdu.cpp`, plutôt que dans le fichier ampd.cpp utilisé pour les paramètres de modèle. Toutefois, les structures des appels de pmaddent et de stradd concernés demeurent exactement les mêmes. Il ne faut pas oublier que certains arguments de ces fonctions diffèrent selon qu'il s'agit de paramètres d'ajustement ou de paramètres de modèle. L'exemple pratique souligne ces différences. 5) Enfin, les modifications apportées à la logique elle-même sont définies dans le fichier `adju.cpp` plutôt que dans les fonctions d'imposition/transfert individuelles, tel le fichier `Atxcalc.cpp`, qui concernent les modifications de la logique de calcul d'imposition/transfert d'un modèle.

Nous notons en passant que les paramètres de commande de la BDSPS utilisent une structure parallèle similaire; toutefois, même dans les applications en boîte de verre, l'utilisateur n'a pas besoin de DÉFINIR de nouveaux paramètres de commande. Il lui suffit de modifier les valeurs des paramètres de commande existants.

Exemple pratique

Notre utilisateur hypothétique, qui cherche à traduire l'effet d'un changement particulier apporté au traitement de l'impôt fédéral sur le revenu, veut accroître les cotisations aux RÉER dans un modèle. Il désire appliquer la croissance soit à un système unique, soit aux deux systèmes, de base et de variante, d'un modèle comparatif. Toutefois, il ne désire pas présumer que les cotisations de chaque contribuable croissent au même rythme et il veut simuler une croissance non proportionnelle en fonction du revenu. On présume que l'objet principal du modèle se trouve ailleurs dans le système d'imposition/transfert. Autrement dit, l'utilisateur n'est pas particulièrement intéressé aux répercussions des augmentations de cotisations aux RÉER elles-mêmes. Il désire plutôt obtenir simplement de « meilleures » représentations des montants de déductions à utiliser dans tous les calculs des systèmes de

transfert pertinents.

Pour obtenir un exemple plus précis, présumons que l'utilisateur désire augmenter les cotisations actuelles de x % pour chaque tranche (entière ou partielle) de 10 000 $ de gains d'emploi rémunéré et de gains d'emploi autonome sur un montant de base initial de 20 000 $. Ainsi, une personne qui gagne 45 000 $ verrait ses cotisations aux RÉER augmenter d'un facteur (1,0 + 3x), où x est le nouveau paramètre défini par l'utilisateur. On obtient une croissance ADDITIONNÉE ET COMPOSÉE avec la croissance induite par le paramètre GFRRSP de croissance standard des cotisations aux RÉER.

Selon un scénario NON développé ici, l'utilisateur pourrait aussi induire la présence de cotisations aux RÉER dans le cas de personnes ayant déclaré 0 $ de cotisations. L'exemple développé plus loin illustre ce genre de synthèse de montants exprimés en dollars.

Dans le reste de la présente sous-section, nous suivons pas à pas la méthode utilisée pour appliquer cette croissance conditionnelle (au-delà de la croissance appliquée par le paramètre d'ajustement GFRRSP). Nous présumons que l'utilisateur a créé le sous-répertoire GLASSEX4 pour cet exemple et y a « COPIÉ » tous les fichiers concernés (`SPSMGL.sln`, `SPSMGL.vcproj`, `apu.h`, `apdu.cpp`, et `adju.cpp`, en plus des fichiers de paramètres BDSPS nécessaires à l'exécution du nouveau modèle). Dans ce répertoire, l'utilisateur crée un fichier « `.API` » afin de fournir une valeur pour le nouveau paramètre défini par l'utilisateur.

Puisque le processus d'ajout de paramètres d'ajustement et de modèle est très similaire à la procédure décrite aux chapitres précédents, nous réduisons au minimum nos observations concernant ces modifications. On présume que l'utilisateur a modifié le projet de manière à inclure tous les fichiers pertinents et qu'il a donné le nom `GLASSEX4.EXE` à la sortie de la compilation.

Nous incluons la documentation d'ajustement dans la chaîne concernée définie dans le fichier adju.cpp, tel que décrit ci-dessous.

(A) Modifications au fichier `apu.h`

Pour commencer, nous établissons un paramètre défini par l'utilisateur pour le facteur de croissance des cotisations aux RÉER défini par l'utilisateur, facteur « x » dans la description ci-dessus. Comme moyen mnémonique, UDGFRRSP (facteur de croissance défini par l'utilisateur, cotisations aux RÉER) semble convenir.

Les ajouts au fichier `apu.h`, qui indiquent le genre de paramètre que l'on définit, sont insérés juste avant la spécification de prototype de fonction et remplace la valeur fictive du paramètre d'ajustement d'utilisateur UADUMMY dans le code `apu.h`.

```
typedef struct UA_ {
int UADUMMY;         /* dummy entry */
 }
 UA_;
```

Dans notre exemple, nous remplaçons la ligne UADUMMY par ce qui suit :

```
NUMBER UDGFRRSP;    /* User-defined growth factor for RRSP Contr.  */
```

(B) Modifications au fichier `apdu.cpp`

Dans la fonction apdu.cpp, nous ajoutons des appels aux fonctions pmaddent et stradd afin de permettre un accès plus large de la BDSPS aux valeurs du nouveau paramètre. Vous trouverez les détails de ces fonctions aux chapitres précédents. Nous faisons les ajouts à la fin de la fonction `apdu.cpp`, juste avant la déclaration :

```
DEBUG_OFF("apdu");
```

Les deux appels se présentent comme suit :

```
pmaddent(pap, "UDGFRRSP", (char *)&AP.UA.UDGFRRSP, NULL, P_SCL, C_NUM, 0, 0, NULL, 0);
```

et

```
stradd("UDGFRRSP", "User-defined growth factor for RRSP Contr.");
```

Le texte explicatif au début de la fonction `apdu.cpp` décrit les arguments AXÉS SUR LE VIEILLISSEMENT des fonctions pmaddent et stradd. Il inclut également les gabarits que nous allons utiliser ici (un paramètre scalaire).

Deux différences essentielles caractérisent l'utilisation de la fonction pmaddent en lieu et place de la définition de nouveaux paramètres de modèle. 1) Le premier argument est pap plutôt que pcp. 2) Le troisième argument diffère; le nouveau paramètre réside dans la structure UA (ajustement par l'utilisateur) à l'intérieur de la structure AP (paramètre d'ajustement) de la BDSPS. Cette approche est fort différente de la référence « &MP.UM » utilisée pour les paramètres de modèle définis par l'utilisateur (modèle d'utilisateur à l'intérieur des paramètres de modèle).

(C) Modifications au fichier `adju.cpp`

La première modification permet de mettre à jour de la chaîne de texte de documentation sur l'ajustement des données. La fonction originale `SPSM\GLASS` définit cette chaîne aux environs de la ligne 53, comme suit :

```
/*global*/ char AGENAME[IDSIZE+1] = "Unnamed";
```

Nous la modifions de la façon suivante :

```
/*global*/ char AGENAME[IDSIZE+1] = "RRSP Contr(Earnings)";
```

En utilisant la valeur de paramètre dans l'ensemble de la BDSPS, nous effectuons des ajouts de code source de manière à appliquer la croissance des cotisations aux RÉER. En premier lieu, nous avons besoin de quelques variables locales qui nous aident à passer d'une personne du ménage analysé à l'autre et éventuellement à attribuer les cotisations modifiées aux RÉER. Nous ajoutons donc les quatre déclarations qui suivent à la fonction `adju.cpp` en les insérant immédiatement après l'accolade d'ouverture de la fonction :

```
NUMBER earn;       /* total paid and self-employment earnings */
int group;         /* number of UDGFRRSP multiples to use */
register P_in in;  /* pointer to data for current person */
int ini;           /* persons processed */
```

Pour ce qui est des attributions d'ajustement elles-mêmes, l'endroit approprié se trouve vers de la toute fin de la fonction `adju.cpp`, à l'intérieur du segment de code suivant :

```
        DEBUG_ON("adju");
        /* Just call the standard adjustment algorithm */
        adj(hh);
        DEBUG_OFF("adju");
```

L'ajout est inséré entre les instructions « `adj(hh);` » et « `DEBUG_OFF("adju");` ».

```
/* Grow RRSP contributions as a function of total earnings */

for (ini=0, in=&hh->in[0]; ini<hh->hhnin; in++, ini++) {
        if (in->id.idrrsp == (NUMBER)0.0) {
                continue;
        }
        earn = in->id.idiemp + in->id.idise;
        if (earn <= (NUMBER)20000.0) {
                continue;
        }
        group = (int)(ONE+(earn-(NUMBER)20000.0)/(NUMBER)10000.0);
        in->id.idrrsp*=(ONE+AP.UA.UDGFRRSP*(float)group);
}
```

Le nouveau code, précédé d'une en-tête contenant un commentaire explicatif, est réparti en éléments relativement simples.

(1) La partie commande de l'instruction « for » a été entièrement copiée de la fonction `memo1.cpp` (calcul des totaux des personnes) dans le sous-répertoire `SPSM\GLASS`. Elle traite toutes les personnes du ménage. Les variables locales définies auparavant sont utilisées dans cette opération.

(2) La croissance par multiplication des cotisations aux RÉER n'est pas significative s'il n'y en a pas au départ. Ainsi, l'instruction « if-continue » des trois lignes suivantes n'exécute pas les quatre instructions suivantes si la personne n'a pas cotisé. Le type « NUMBER », ici et plus loin, indique les intentions de l'utilisateur relativement aux types de variables; il empêche le compilateur d'émettre des avertissements.

(3) Si les cotisations aux RÉER sont positives, la ligne suivante calcule les gains de la personne pour un emploi rémunéré et un emploi autonome. Si le total ne dépasse pas 20 000 $, le système ne traite pas le reste de l'instruction « for »; une autre instruction « if-continue » remplit cette fonction.

(4) L'attribution à la variable « de groupe » calcule le nombre de multiples de UDGFRRSP pertinents pour la croissance. L'instruction finale dans le corps de la boucle applique la croissance par une attribution multiplicatrice. Ces deux instructions seront exécutées seulement si une certaine croissance est appropriée. Les types (int) et (NUMBER) qu'ils contiennent indiquent l'intention explicite de l'utilisateur concernant les conversions de types de variables; ils servent à éviter les avertissements inutiles à l'étape de la compilation.

(D) Compilation du modèle amélioré

Il faut déboguer le modèle avant de compiler le fichier exécutable `GLASSEX4.EXE` pour être en mesure d'utiliser le modèle pour des essais de validation et les travaux de production.

(E) Entrée d'une valeur de paramètre

Pour toute exécution particulière du modèle, l'utilisateur doit fournir une valeur pour le nouveau paramètre, par exemple 0,01. Habituellement, il le fait « à la volée » pendant l'exécution du nouveau modèle, ou en utilisant un fichier « `.API` » (inclusion de paramètres d'ajustement de données) qui modifie le contenu du fichier `APR` précisé dans le fichier de commande du modèle (« `.CPR` »). Dans l'exemple, si aucun paramètre d'ajustement existant ne doit être modifié, le fichier « .API » contient une seule ligne :

```
UDGFRRSP      0.01
```

(F) Validation du modèle

Avant d'utiliser réellement le modèle, l'utilisateur voudra le valider pour s'assurer qu'il fonctionne de la façon prévue. Nous n'effectuons pas en détail cette validation à ce moment-ci pour des raisons d'espace; normalement, il faudrait produire quelques tableaux spécifiques pour différentes exécutions afin de vérifier si le modèle produit les résultats prévus. Par exemple, si l'on conserve la valeur zéro comme facteur UDGFRRSP, le montant total du RÉER demeure inchangé. De la même manière, une valeur faible, p. ex., 0,01, aura peu ou pas d'effet sur les unités à faible revenu, mais aura un effet plus important sur les unités à revenu plus élevé. Un tableau, défini au niveau de la personne, qui indique une augmentation de la variable de cotisations aux RÉER comme fonction des gains de la personne permet facilement de déterminer si l'algorithme donne le bon montant d'augmentation du RÉER. Le tableau peut être produit en utilisant un fichier de résultats basé sur la base de données non modifiée puis en comparant le nombre de personnes et les montants des cotisations aux RÉER aux équivalents de ces variables après le nouveau vieillissement des cotisations.

Exemple de résultats obtenus en utilisant la valeur 0,01 comme paramètre d'ajustement UDGFRRSP :

	Croissance avant	Croissance après	Écart
Cotisations aux RÉER (M $)	11 134,3	11 329,2	194,9
Impôt sur le revenu fédéral (M $)	41 173,3	41 118,0	55,3
Impôt sur le revenu provincial (M $)	24 190,6	24,160,5	30,1

Le total des cotisations aux RÉER s'est accru d'environ 1,75 % et les impôts sur le revenu fédéral et provincial ont subi une baisse correspondante légèrement inférieure au montant des nouvelles cotisations aux RÉER.

Liste de vérification pour la modification « globale » des variables de base de données

(A) Vérifier si les fonctions existantes de la BDSPS suffisent pour appliquer l'ajustement souhaité des données, rendant ainsi inutile toute nouvelle logique.

Peut-on appliquer l'ajustement souhaité de la population en choisissant parmi les fichiers de pondération de cas existants? Si c'est le cas, il suffit de préciser le fichier pertinent (avec l'extension « `.WGT` ») en utilisant le paramètre de commande INPWGT (pondération à l'entrée). Utilisez un fichier « `.CPI` » pour fournir la valeur INPWGT souhaitée, ou

entrez-la à la volée.

Peut-on effectuer l'ajustement des valeurs de données en modifiant les valeurs des paramètres d'ajustement de données de la BDSPS, de pair avec l'algorithme d'ajustement des données normal de la BDSPS (adj(hh))? Si c'est le cas, fournissez les valeurs de paramètre d'ajustement pertinents en utilisant un fichier « .API » ou à la volée. Cette opération peut être faite de manière interactive ou en utilisant un fichier de traitement par lots servant à coordonner l'exécution du modèle.

(B) Si les ajustements que l'on souhaite apporter aux données ne peuvent l'être en utilisant les procédures intégrées d'ajustement des données, il faut prévoir une nouvelle logique. Les étapes d'ajout de cette nouvelle logique sont les suivantes :

1. Copier tous les fichiers pertinents dans un nouveau répertoire créé pour l'analyse. Les fichiers `\SPSM\GLASS\adju.cpp, SPSMGL.vcproj et SPBMGL.sln` sont toujours pertinents. Les fichiers `\SPSM\GLASS apu.h` et `\SPSM\GLASS\apdu.cpp` sont pertinents lorsque de nouveaux paramètres d'ajustement sont requis.

2. Modifier l'environnement du projet de manière à inclure tous les fichiers concernés et changer le nom du modèle compilé. Modifiez le fichier `apu.h` si vous définissez de nouveaux paramètres d'ajustement de données.

3. Modifier le fichier `apdu.cpp` s'il y a définition de nouveaux paramètres d'ajustement de données. Les modifications incluent de nouveaux appels de pmaddent et de stradd afin que la substance des nouveaux paramètres soit disponible dans l'ensemble de la BDSPS. Mettre le modèle au point.

4. Modifier le fichier `adju.cpp`. Modifier d'abord la chaîne de texte de documentation de la fonction, `AGENAME[IDSIZE+1].` Appliquer ensuite la nouvelle logique d'ajustement des données. Cette étape consiste souvent à déclarer des variables locales utiles et à traiter à tour de rôle toutes les personnes ou toutes les familles du ménage.

5. Compiler et valider le modèle avant de l'utiliser en production. On recommande d'effectuer une totalisation parallèle des personnes et des montants pertinents avant et après les modifications d'ajustement des données.

6. Faire des exécutions de production avec la nouvelle logique d'ajustement après l'avoir validée.

MODIFICATIONS QUI TOUCHENT SEULEMENT SOIT LA BASE, SOIT LA VARIANTE

La construction d'un modèle dans lequel l'ajustement des données diffère entre le système de base et le système de variante est plus complexe que celle d'un modèle dans lequel les deux systèmes sont traités de manière identique. Dans la mesure du possible, l'utilisateur devrait éviter ce genre de complexité. La capacité de la BDSPS d'utiliser les fichiers de résultats (extension « .MRS ») fournit le principal mécanisme pour éviter l'ajustement des données soumis à des conditions de système.

L'approche de base consiste à diviser le problème en deux volets, un pour chaque système. Ensuite, on applique à chacun un seul algorithme d'ajustement de données en utilisant les méthodes décrites plus haut dans le présent chapitre. L'utilisateur crée d'abord un fichier de résultats pour l'un des deux systèmes en choisissant les variables nécessaires pour les totalisations propres au système et les comparaisons. Lorsqu'il crée le premier système, l'utilisateur applique les hypothèses d'ajustement de données qui lui sont appropriées. Ensuite, il effectue une simulation du second système en lui appliquant l'ajustement de données de rechange approprié. Le système lit le fichier de résultats en parallèle avec le traitement du second système, de manière que les deux systèmes, et leurs hypothèses d'ajustement de données différentes, soient disponibles simultanément pour toutes les comparaisons à effectuer. La publication ***Introduction et aperçu*** inclut une illustration de l'utilisation des fichiers de résultats.

La dernière partie de cette section traite des situations où la méthode des fichiers de résultats est jugée inappropriée ou inadéquate pour la tâche à accomplir. Voici quelques exemples à titre d'illustration :

1. L'utilisateur peut juger très important d'avoir à sa disposition un modèle intégré qui, une fois validé, est relativement facile à utiliser en direct.

2. L'application prévue du modèle peut exiger une analyse de sensibilité qui nécessite plusieurs fichiers MRS, ce qui peut créer une certaine confusion. Par exemple, l'analyse peut exiger une évaluation des répercussions de la modification de l'ajustement d'une variable particulière pendant que diverses autres variables sont modifiées en parallèle de manière répétitive entre les systèmes de base et de variante.

3. L'application prévue peut exiger des comparaisons complexes nécessitant de volumineux fichiers .MRS.

Nous croyons toutefois que de telles situations, qui peuvent se présenter à l'occasion, sont l'exception plutôt que la règle. Nous incitons donc l'utilisateur à tenter d'éviter les modèles de systèmes parallèles dans lesquels l'ajustement des données est différent.

De manière générale, la méthode utilisée pour modifier les données propres à un système est semblable à celle utilisée pour modifier la LOGIQUE D'IMPOSITION/TRANSFERT d'un système. Tout nouveau paramètre d'ajustement de données propres à un système est ajouté, par les fichiers `mpu.h` et `mpdu.cpp`, comme paramètre de MODÈLE et NON comme paramètre d'ajustement de données. Tel que décrit ci-dessous, l'utilisateur peut souhaiter ajouter de nouvelles variables dépendantes de MODÈLE pour pister les modifications apportées. Même lorsque de nouveaux paramètres et de nouvelles variables dépendantes ne sont pas nécessaires, la procédure peut s'appliquer à la fois aux modèles de base et de variante; toutefois, les explications données ci-dessous sont axées sur les situations plus courantes d'utilisation de modèles de variante.

La méthode axée sur le MODÈLE résumée ci-dessus tient à la conception de la BDSPS. Puisqu'il n'y a qu'un seul fichier « .APR », ses paramètres ont inévitablement une incidence sur l'ajustement de données de tous les systèmes compris dans le modèle. D'autre part, les

modifications apportées par les fichiers « .MPI » et les fonctions `Adrv.cpp` et `drv.cpp` propres au système s'appliquent à un seul système d'imposition/transfert particulier. L'utilisateur peut tirer avantage de cette caractéristique pour appliquer des ajustements de données propres au système.

La solution pour effectuer des modifications d'ajustement des données propres au système est liée aux modifications apportées au fichier Acall.cpp. Il s'agit, essentiellement, pour l'utilisateur d'« intercepter » l'enregistrement de données du ménage juste avant son utilisation par les fonctions de la procédure, d'effectuer les modifications désirées, puis de rétablir l'enregistrement à son état original juste avant que l'exécution de la procédure cesse. La section suivante explore plus en profondeur les étapes liées à l'utilisation du fichier `Acall.cpp.`

Mise en œuvre des modifications dans `Acall.cpp`

La présente section traite presque exclusivement des détails des modifications apportées au fichier `Acall.cpp`. En raison de la similitude des ajustements de données propres au système effectués lors des types de révision de systèmes d'imposition/transfert décrits précédemment dans ce document, certains sujets ne sont pas repris ici. En particulier, on s'attend que les utilisateurs ajoutent eux-mêmes les nouveaux paramètres et les nouvelles variables dépendantes nécessaires en utilisant les méthodes développées aux sections précédentes. Par exemple, un utilisateur peut souhaiter ajouter une nouvelle variable de modèle pour vérifier si la valeur originale d'une variable incluse dans la base de données a été modifiée par des ajustements propres au système.

Nous étudions les modifications requises dans l'ordre où le lecteur en prend connaissance en lisant le code source du fichier `Acall.cpp`. L'exemple pratique qui suit illustre une application concrète des modifications.

(A) Déclarer de nouvelles variables locales (dans le fichier `Acall.cpp`)

N'oubliez pas que la procédure générale exige de l'utilisateur qu'il sauvegarde les valeurs des variables à ajuster. La sauvegarde permet de rétablir les valeurs avant de sortir du fichier `Acall.cpp`. L'utilisateur doit donc inclure dans le fichier Acall.cpp des déclarations locales appropriées pour créer l'espace de sauvegarde nécessaire. Habituellement, les variables à ajuster seront définies au niveau de la personne. En général, les nouvelles variables doivent donc être définies comme des vecteurs de longueur MAXPERS. (MAXPERS représente le nombre maximal de personnes dans un ménage.) L'utilisateur peut également souhaiter définir d'autres variables de travail locales. Habituellement, il déclare ces variables juste avant l'accolade d'ouverture de cette fonction, près de la ligne 90 de la version non modifiée du fichier `Acall.cpp`

(B) Sauvegarder les valeurs à modifier

La toute première étape de la partie exécutable du fichier `Acall.cpp` consiste à sauvegarder les valeurs originales des variables qui seront modifiées. Ainsi, aucune des autres fonctions invoquées dans le fichier Acall.cpp ne peut ni modifier immédiatement la

valeur, ni utiliser la valeur non modifiée. Habituellement, la sauvegarde est effectuée par une instruction « for » qui traite toutes les personnes d'un ménage à tour de rôle et les copie, une à la fois, dans les éléments d'un vecteur déclaré à l'étape (A). Un des éléments du bestiaire fournit les commandes pertinentes du traitement par étapes. L'utilisateur accomplit cette opération près de la ligne 90 du code non modifié, immédiatement après l'instruction suivante :

```
DEBUG_ON("Acall");
```

(C) Modifier les valeurs de la base de données

Immédiatement après la sauvegarde des valeurs, mais avant que le pointeur de ménage ne soit transmis à l'une ou l'autre des fonctions d'imposition/transfert ou de cumul, l'utilisateur doit apporter les modifications nécessaires aux valeurs des variables concernées. Ces modifications constituent l'essentiel de la « programmation réelle », c'est-à-dire la logique qui ne peut en pratique s'inspirée de toute autre logique utilisée ailleurs dans la BDSPS.

(D) Utiliser les valeurs nouvellement ajustées

Cette étape, la plus facile de toutes, ne requiert aucun effort spécial de l'utilisateur. Elle consiste à RETENIR les appels des diverses fonctions d'imposition/transfert et pour mémoire. Puisque, à ce stade-ci, les valeurs des variables pertinentes ont déjà été ajustées, toutes ces fonctions effectuent leurs calculs en utilisant les ménages ajustés.

(E) Restaurer les valeurs originales

L'étape finale consiste à restaurer les valeurs originales des variables ajustées. Cette opération est habituellement effectuée à proximité de la ligne 90 de la version non modifiée du fichier `Acall.cpp`, immédiatement avant que le contrôle échappe à la fonction, c'est-à-dire avant l'instruction suivante :

```
DEBUG_OFF("Acall");
```

La restauration est importante au plan de la globalité, de la maintenabilité et de la réutilisabilité du code. L'utilisateur programme la modification sans savoir si le système visé sera un système de base ou de variante. En restaurant les valeurs, l'utilisateur peut minimiser les éventuels effets secondaires non désirés ailleurs dans le modèle. En outre, cette procédure réduit au minimum la possibilité d'effets secondaires non désirés si jamais les nouveaux ajustements sont utilisés dans un autre modèle.

Exemple pratique

(A) Substance à modéliser

Pour commencer, nous décrivons la logique fondamentale utilisée dans l'exemple. Il va de soi que nous pouvons atteindre les mêmes buts d'ajustement de données en utilisant les techniques d'« évitement » décrites précédemment; toutefois, puisque l'objectif de la documentation est d'illustrer des techniques d'ajustement de données propres au système, nous présumons arbitrairement que ces techniques d'évitement ne servent pas nos objectifs premiers.

Présumons qu'une analyse exogène quelconque des nouveaux besoins de déclaration de revenu donnent à penser que des personnes déclareront davantage de revenus d'emploi autonome. Plus particulièrement, présumons que 5 % de ces personnes, (1) qui ne déclarent pas plus de 100 $ de revenu d'emploi autonome (agricole et non agricole combinés), (2) qui sont âgés dans les deux cas de plus de 25 ans et de moins de 60 ans et (3) qui, en outre, ont été sans emploi durant une demie année ou plus et qui sont en recherche de travail, ont effectivement retirer des revenus d'emploi autonome non agricole qu'ils n'ont pas déclarés antérieurement mais qu'ils déclareront à l'avenir. Présumons, en plus, que l'on croit que les montants de ces « nouveaux » revenus d'emploi autonome seront répartis uniformément, entre zéro et 4 000 $ par année.

L'utilisateur tente d'estimer les impôts sur le revenu supplémentaires que le gouvernement peut percevoir de ces personnes et d'évaluer les répercussions de ce « nouveau » revenu sur la réduction du taux de pauvreté mesuré par la fonction du seuil de la pauvreté. Pour effectuer cette étude, l'utilisateur prévoit utiliser le système d'imposition/transfert de variante pour imputer les montants appropriés associés à ces nouveaux revenus à des personnes choisies au hasard et qui satisfont à ces trois conditions.

(B) Nouvelles variables et nouveaux paramètres pertinents

Selon les pratiques recommandées de la BDSPS, qui visent à éviter l'utilisation de valeurs directement insérées dans un modèle, l'utilisateur établit les nouveaux paramètres d'ajustement d'utilisateur, comme suit :

Paramètre	Description :	Valeur :
NSEFLAG	« Drapeau de nouveau revenu d'emploi autonome »	1
NSEAMT	« Montant de base du nouveau revenu d'emploi autonome »	100,0
		0,05
NSEFRC	« Fraction du nouveau revenu d'emploi autonome »	26
NSEWKS	« Exigence de semaines de nouvel emploi autonome »	25
NSEMINAGE	« Âge minimum du nouvel emploi autonome »	60
NSEMAXAGE	« Âge maximum de nouvel emploi autonome »	4 000,0
NSEMAXINC	« Nouveau revenu maximum du nouvel emploi autonome »	

L'utilisateur définit également de nouvelles variables qui permettent de dénombrer facilement les personnes admissibles et d'identifier celles pour lesquelles de nouveaux revenus seront synthétisés. De plus, la nouvelle variable supplémentaire pour les montants de revenu synthétisés pourra s'avérer utile.

Description des variables :

uvnseef	« Nouvel emploi autonome admissible »
uvnsesf	« Nouvel emploi autonome reçu »
uvnseamt	« Montant du nouvel emploi autonome »

(C) Préparation de l'analyse

L'utilisateur crée d'abord un nouveau sous-répertoire d'analyse, `GLASSEX5`, dans lequel il copie les fichiers de gabarit requis suivants : `SPSMGL.vcproj` et `SPSMGL.sln` (pour contrôler la compilation), `mpu.h` et `Ampd.cpp` (pour assurer la disponibilité des nouveaux paramètres), `vsu.h` et `vsdu.cpp` (pour assurer la disponibilité des nouvelles variables) et `Acall.cpp` (pour appliquer les nouveaux ajustements de base de données propres au système).

Les modifications sont examinées dans l'ordre que l'on a recommandé à l'utilisateur.

(D) Changements à apporter au projet

Tous les fichiers pertinents doivent être inclus dans le projet et le nom du modèle de sortie doit être remplacé par GLASSEX5.EXE.

(E) Changements apportés au fichier `mpu.h`

L'utilisateur fournit les déclarations pour tous les nouveaux paramètres décrits ci-dessus.

```
int    NSEFLAG;    /* New Self-Employment Income Flag */
NUMBER NSEAMT;     /* New Self-Employment 'Trivial Amount' */
NUMBER NSEFRC;     /* New Self-Employment Fraction */
NUMBER NSEWKS;     /* New Self-Employment Weeks Requirement */
NUMBER NSEMINAGE;  /* New Self-Employment Minimum Age */
NUMBER NSEMAXAGE;  /* New Self-Employment Maximum Age */
NUMBER NSEMAXINC;  /* New Self-Employment Maximum New Income */
```

(F) Changements apportés au fichier `Ampd.cpp`

L'utilisateur modifie le fichier `Ampd.c` en fournissant les appels de pmaddent et de stradd pour tous les nouveaux paramètres. Les nouveaux appels de pmaddent sont les suivants :

```
pmaddent(pcp, "NSEFLAG",   (char *)&MP.UM.NSEFLAG,    NULL, P_SCL, C_INT,  E_FLAG, 0, NULL,
0);
pmaddent(pcp, "NSEAMT",    (char *)&MP.UM.NSEAMT,     NULL, P_SCL, C_NUM,  0,     0, NULL,
0);
pmaddent(pcp, "NSEFRC",    (char *)&MP.UM.NSEFRC,     NULL, P_SCL, C_NUM,  E_FRCT, 0, NULL,
0);
pmaddent(pcp, "NSEWKS",    (char *)&MP.UM.NSEWKS,     NULL, P_SCL, C_NUM,  0,     0, NULL,
0);
pmaddent(pcp, "NSEMINAGE", (char *)&MP.UM.NSEMINAGE,  NULL, P_SCL, C_NUM,  0,     0, NULL,
0);
pmaddent(pcp, "NSEMAXAGE", (char *)&MP.UM.NSEMAXAGE,  NULL, P_SCL, C_NUM,  0,     0, NULL,
0);
pmaddent(pcp, "NSEMAXINC", (char *)&MP.UM.NSEMAXINC,  NULL, P_SCL, C_NUM,  0,     0, NULL,
0);
```

Les appels de stradd connexes sont les suivants :

```
stradd("NSEFLAG",   "New Self-Employment Income Flag");
stradd("NSEAMT",    "New Self-Employment 'Trivial Amount'");
stradd("NSEFRC",    "New Self-Employment Fraction");
stradd("NSEWKS",    "New Self-Employment Weeks Requirement");
stradd("NSEMINAGE", "New Self-Employment Minimum Age");
stradd("NSEMAXAGE", "New Self-Employment Maximum Age");
stradd("NSEMAXINC", "New Self-Employment Maximum New Income");
```

(G) Changements apportés au fichier `vsu.h`

Dans ce fichier, l'utilisateur déclare les nouvelles variables qui permettent une totalisation et

une validation plus pratique des personnes pour lesquelles le nouveau revenu est retenu ou réellement synthétisé.

```
int      uvnseef;  /* Eligible for New Self-Empl Synthesis */
int      uvnsesf;  /* Received New Self-Empl Income */
NUMBER   uvnseamt; /* New Self-Empl Amount */
```

(H) Changements apportés au fichier `vsdu.cpp`

Dans le fichier `vsdu.c,` l'utilisateur appelle les fonctions vardef et stradd pour rendre les nouvelles variables disponibles à l'ensemble du modèle. Tel qu'indiqué précédemment, on doit utiliser deux variables de classe pour les sorties de tableaux croisés et une valeur à virgule flottante NUMBER pour le montant du revenu d'emploi autonome synthétisé.

```
/* uvnseef: (Class) Flag: Individual eligible for NSE synthesis? */
vardef("_uvnseef", IN, im.uv.uvnseef,    C_INT, V_CLAS);
stradd("uvnseef",  "Eligibility for Synth Self-Empl");
stradd("nseef",    "\tNot Eligible\tEligible");

/* uvnsesf: (Class)  Flag: Individual Got Synth. NSE? */
vardef("_uvnsesf",  IN, im.uv.uvnsesf,    C_INT, V_CLAS);
stradd("uvnsesf",   "Synth Self-Empl Receipt");
stradd("nsesf",     "\tNo Receipt\tReceipt");

/* uvnseamt: (Analysis) NUMBER: Amount of synthesized NSE */
vardef("_uvnseamt",  IN, im.uv.uvnseamt,    C_NUM, V_ANAL);
stradd("uvnseamt",   "Synth Self-Empl Amount");
```

(I) Changements apportés au fichier `Acall.cpp`

(i) Les premières modifications concernent la déclaration de nouvelles variables essentielles au processus d'ajustement des données. Nous utilisons la notation standard de la BDSPS pour le pointeur vers une personne et pour le nombre de personnes soumises au traitement (pour les règles d'arrêt à l'intérieur des ménages). En plus, on déclare un vecteur pour conserver les valeurs originales du revenu d'emploi autonome des personnes.

```
register P_in in;         /* pointer to data for current person */
int ini;                  /* persons processed */
NUMBER orignfse[20];       /* original self-empl income */
```

(ii) La modification suivante concerne le code qui permet de stocker le revenu d'emploi autonome non agricole existant et de le rétablir par la suite à son état initial. Nous utilisons un des éléments standard du bestiaire, et traitons à tour de rôle les personnes à l'intérieur du ménage, pour appliquer cette fonction d'archivage.

```
/* Archive original database values for self-employment */

for (ini=0, in=&hh->in[0]; ini<hh->hhnin; in++, ini++) {
      orignfse[ini]=in->id.idise;
}
```

Une version légèrement plus efficace de ce code permet une exécution conditionnelle de l'instruction de sauvegarde basée sur l'attribution de la valeur 1 au paramètre NSEFLAG pour activer la fonction de synthèse. La présente version est plus simple et légèrement plus sûre.

(iii) Application du revenu d'emploi autonome augmenté sous conditions

Nous incitons les utilisateurs à se servir des générateurs de nombres aléatoires de la SPSDM (idrand) pour assurer la reproductibilité des résultats. L'utilisation des nombres aléatoires de C++ peut générer des résultats différents pour chaque exécution.

```
       /* Selectively synthesize self-employment income */

for (ini=0, in=&hh->in[0]; ini<hh->hhnin; in++, ini++) {
       in->im.uv.uvnseef=0;  /* assign values to new vars */
       in->im.uv.uvnsesf=0;
       in->im.uv.uvnseamt=(NUMBER)0.0;
       if (MP.UM.NSEFLAG==0) {
              continue; /* don't synthesize if facility is off */
       }

if ( (in->id.idise>MP.UM.NSEAMT) ||
       (in->id.idnage<MP.UM.NSEMINAGE) ||
       (in->id.idnage>MP.UM.NSEMAXAGE) ||
       (in->id.idlyun<(int)MP.UM.NSEWKS) ) {
              continue;           /* ignore ineligible individuals */
       }

       in->im.uv.uvnseef=1; /* mark indiv. as potentially eligible */

if (in->id.idrand[2]>MP.UM.NSEFRC) {
       continue;      /* individual was not selected to get income */
}

in->im.uv.uvnsesf=1; /* mark indiv. as recipient */
in->im.uv.uvnseamt=in->id.idrand[3]*MP.UM.NSEMAXINC; /*synthesize amt */
in->id.idise+=in->im.uv.uvnseamt; /* add syn amt to self-empl */
}
```

Le code ci-dessus, malgré sa longueur, est simple. La boucle de traitement itératif des personnes permet d'effectuer ce qui suit :

On attribue des valeurs par défaut aux nouvelles variables définies par l'utilisateur.

Le reste des instructions de la boucle n'est pas exécuté si la fonction n'est pas activée.

Le reste des instructions de la boucle n'est pas exécuté si la personne ne respecte pas les conditions d'admissibilité pour la synthèse du nouveau revenu d'emploi autonome.

On indique que la personne est potentiellement admissible aux calculs de synthèse; le reste des instructions de la boucle n'est pas exécuté si la personne n'est pas « retenue » pour recevoir le revenu.

Si l'exécution atteint cette étape, on indique que la personne peut bénéficier du revenu synthétisé et on impute le montant en additionnant le nouveau montant à la variable d'emploi autonome de la personne.

Une fois la boucle exécutée, la synthèse du nouveau revenu d'emploi autonome est complète pour tous les membres du ménage. À ce stade-ci, les instructions « régulières » du fichier `Adrv.cpp` sont exécutées et calculent les montants d'imposition/transfert et les divers articles pour mémoire.

(iv) Enfin, une fois que le ménage ajusté a été traité par toutes les fonctions d'imposition/transfert et pour mémoire, le nouveau code rétablit les valeurs initiales de revenu d'emploi autonome.

```
   /* Restore original database values for self-employment */

       for (ini=0, in=&hh->in[0]; ini<hh->hhnin; in++, ini++) {

in->id.idise=orignfse[ini]; }
```

Une version légèrement plus efficace de ce code permet une exécution conditionnelle de l'instruction de restauration basée sur l'attribution de la valeur 1 au paramètre NSEFLAG pour activer la fonction de synthèse. La présente version est plus simple et légèrement plus sûre.

(J) Nouveaux fichiers MPI et CPI

Il reste encore à fournir les valeurs aux divers paramètres de manière que la BDSPS, pendant une exécution particulière, puisse appliquer les ajustements désirés. Un « fichier d'inclusion » de paramètres (extension « .MPI ») remplit cette fonction avec les entrées ci-dessous. Au choix, ceux qui utilisent MSPS Visuel peuvent modifier les paramètres à la volée à partir de la hiérarchie.

```
NSEFLAG          1
NSEAMT           100.0
NSEFRC           0.05
NSEWKS           26.0
NSEMINAGE        25.0
NSEMAXAGE        60.0
NSEMAXINC        4000.0
```

Il faut également veiller à ce que des séries indépendantes pertinentes de variables pseudo-aléatoires soient générées pour servir d'entrées aux choix « aléatoires » de bénéficiaires de revenus synthétiques et aux montants connexes de revenu synthétisé.

(K) Compilation et validation du modèle

Lorsqu'il a terminé toutes les modifications du code source, l'utilisateur doit d'abord mettre le modèle au point puis le compiler dans le fichier exécutable concerné, `GLASSEX5`. Nous concluons cet exemple pratique par une explication rapide et expéditive d'un ensemble de tableaux de validation. Dans le cas d'une application réelle, l'utilisateur doit normalement effectuer une validation beaucoup plus rigoureuse des modifications. N'oubliez pas que vous pouvez effectuer ce genre d'ajustement de données propres au système plus facilement en utilisant des fichiers de résultats (« `.MRS` »). Cette approche consiste à appliquer une logique d'attribution de revenu équivalente en utilisant le fichier adju.cpp et à fournir à la volée, ou à partir d'un fichier `API`, les paramètres pertinents.

Présumons, dans cette illustration de validation rapide et expéditive, que la source exogène de l'utilisateur a déjà fourni, peut-être sous forme de fonction d'une variable quelconque de politique pertinente, une indication sommaire du nombre de personnes visées par ce nouveau revenu d'emploi autonome.

L’utilisateur doit d’abord totaliser les nombres de personnes selon les valeurs des deux variables de classe uvnsef et uvnsesf définies par l’utilisateur. Les entrées de ce tableau peuvent alors être comparées à la source exogène afin de confirmer (1) que les nombres de personnes admissibles correspondent à ceux précisés par la « source exogène » et (2) que l’on a attribué à une proportion appropriée de ces personnes un nouveau revenu d’emploi autonome.

Ensuite, l’utilisateur doit vérifier que le montant moyen du revenu du nouvel emploi autonome imputé est approprié (c.-à-d. la moitié de la valeur du paramètre NSEMAXINC 4 000 $). Il convient de totaliser le montant du nouveau revenu imputé de manière à pouvoir le comparer aux augmentations d’impôt fédéral et provincial sur le revenu. L’utilisateur peut donc vérifier qu’une proportion appropriée de nouveau revenu revient au gouvernement à titre d’impôt sur le revenu.

Dans le cas des exécutions de validation, il semble également logique de vérifier le niveau de modification des incidences des unités situées sous le seuil de pauvreté pertinent. Compte tenu des conditions d’admissibilité au revenu synthétisé relativement strictes et de la portion relativement faible de la population admissible choisie pour recevoir le nouveau revenu d’emploi autonome, l’utilisateur ne doit s’attendre qu’à de petites modifications de cette incidence.

Dans l’exemple qui suit, nous illustrons la première partie de cette validation en vérifiant les montants du nouveau revenu d’emploi autonome. Les modifications de « taux de pauvreté », non indiquées ici, sont calculées à l’aide des variables de la BDSPS « efpovthr » (seuil de pauvreté) et « impovinc » (revenu pour fins de comparaison avec le seuil de pauvreté pertinent). Pour effectuer la validation, il est plus pratique d’utiliser des tableaux croisés. Les paramètres de commande pertinents, extraits du fichier « .CPI », sont les suivants :

```
XTFLAG    1
XTSPEC
        IN: { units }
            * uvnseef
            * uvnsesf;
         IN: { uvnseamt,
            uvnseamt/units }
            * uvnsesf;
         IN: { uvnseamt,
            imtxf-_imtxf,
            imtxp-_imtxp }
            * uvnsesf;
```

Remarquez que les résultats de la boîte de verre ont été produits avec une version antérieure du modèle de la BDSPS.

Les tableaux produits sont les suivant :

Table 1U: Unit Count (000) for Individuals by Eligibility for Synth Self-Empl and Synth Self-Empl Receipt

Synth Self-Empl Receipt

Eligibility for Synth Self-Empl	No Receipt	Receipt
Not Eligible	23351.7	0.0
Eligible	809.6	47.2

Table 2U: Selected Quantities for Individuals by Synth Self-Empl Receipt

Synth Self-Empl Receipt

Quantity	No Receipt	Receipt
Synth Self-Empl Amount (M)	0.0	92.5
uvnseamt/units	0	1962

Table 3U: Selected Quantities for Individuals by Synth Self-Empl Receipt

Synth Self-Empl Receipt

Quantity	No Receipt	Receipt
Synth Self-Empl Amount (M)	0.0	92.5
imtxf-_imtxf (M)	1.0	12.9
imtxp-_imtxp (M)	0.5	9.5

Pour ce qui est de la substance de ces tableaux, nous présumons que les 809,6 milliers de personnes du tableau 1U correspondent raisonnablement bien à ce que prévoit la « source de données exogène » hypothétique. Puisque 47,2 milliers de ces personnes reçoivent un nouveau revenu d'emploi autonome quelconque, l'objectif de 5 % a été sommairement atteint..

Le tableau 2U confirme que le nouvel algorithme attribue le nouveau revenu d'emploi autonome seulement à des personnes admissibles. Le montant total du nouveau revenu et les montants moyens connexes confirment que les montants prévus de nouveau revenu sont synthétisés (environ 2 000 $ par personne choisie).

La table 3U indique la partie du nouveau revenu, un peu plus que le quart, qui est retenue par le système fiscal. Comme on s'y attendait, la plus grande partie de ces montant récupérés l'est directement des individus qui l'ont reçue. Certains montants sont récupérés des personnes qui n'ont pas reçu le nouveau revenu - principalement du fait que ces personnes ne peuvent plus réclamer le crédit d'impôt pour époux car leurs époux reçoivent le nouveau revenu. Il est clair que, avec un revenu de moins de 100 M $ distribués dans l'ensemble du secteur personnel, nous ne nous attendons pas à de grandes répercussions sur la proportion de la population en dessous du seuil de pauvreté.

Finalement, lorsque l'utilisateur est satisfait de l'exactitude des procédures d'ajustement, il exécute la BDSPS au complet dans le modèle en effectuant une ou plusieurs exécutions de production. Pour atteindre les objectifs de l'exemple décrit au début de la présente section, la sortie doit inclure le total des impôts sur le revenu du fédéral et du provincial ainsi que le

nombre de familles qui se situent au-dessus et au-dessous des seuils de pauvreté; la sortie est produite à la fois avec et sans la synthèse du nouveau revenu d'emploi autonome. Habituellement, l'utilisateur inclut également la ventilation de ces variables en fonction des variables de classe pertinentes, tel le type de famille.

Liste de vérification pour les modifications apportées à la base de données propres au système

(A) Créer un nouveau sous-répertoire d'analyse et y copier les gabarits de tous les fichiers requis pour l'analyse. Les éléments possibles incluent les fichiers `SPSMGL.sln`, `SPSMGL.vcproj`, `mpu.h`, `Ampd.cpp`, `vsu.h`, `vsdu.cpp`, `Acall.cpp et un fichier de commande` (« `.CPR` »). Dans ce sous-répertoire, l'utilisateur crée également les autres fichiers d'analyse requis pour lesquels il n'existe aucun gabarit particulier, par exemple le fichier « .MPI » qui fournit les valeurs aux paramètres d'ajustement des données propres au système, ou un fichier de traitement par lots pour contrôler la session de la BDSPS.

(B) Changer l'environnement du projet et inclure tous les fichiers pertinents, et modifier le nom du fichier de sortie exécutable.

(C) Modifier les fichiers `mpu.h` et `Ampd.cpp` pour déclarer les nouveaux paramètres d'ajustement de données propres au système et, par des appels de pmaddent et de stradd, les rendre accessibles à l'ensemble de la BDSPS.

(D) Modifier au besoin les fichiers `vsu.h` et `vsdu.cpp` pour déclarer toutes les nouvelles variables de modèle propre au système et, par des appels de vardef et de stradd, les rendre accessibles à l'ensemble de la BDSPS.

(E) Modifier le fichier `Acall.cpp` pour permettre la sauvegarde des valeurs originales des variables à ajuster, effectuer les modifications et, une fois le traitement du ménage terminé, rétablir les valeurs originales avant de quitter la procédure. Ces étapes exigent habituellement la définition de VECTEURS locaux de valeurs dont les dimensions correspondent aux nombres possibles de personnes dans un ménage.

(F) Compiler le nouveau modèle et corriger tout problème identifié par le compilateur.

(G) Fournir, avec des fichiers « .MPI » ou à la volée, les valeurs des nouveaux paramètres d'ajustement de données propres au système. Lorsque l'ajustement fait appel à des variables pseudo-aléatoires, fournir un fichier « .CPI » qui inclut les modifications appropriées du paramètre SEED (ou modifier les paramètres à la volée lorsqu'on utilise MSPS Visuel). Le modèle pourra utiliser ces valeurs de paramètres de modèle et de contrôle au moment de l'exécution du modèle soit en direct, soit par un fichier de traitement par lots de la BDSPS.

(H) Valider soigneusement le modèle, puis faire des exécutions de production.