Applications Hautes Performances & Analyse Numérique Avancée pour l'Environnement .NET

par Visual Numerics (Livres blancs)

Date de publication : 12/01/2009

Dernière mise à jour : 12/01/2009

Ce document explorera certaines forces de la plateforme .NET concernant l'analyse numérique. De nombreuses astuces pour augmenter les performances des applications .NET seront présentées. De plus, comme il est développé ou adapté au cadre d'application .NET des programmes qui nécessitent des algorithmes mathématiques et statistiques, il existe un besoin naturel pour des bibliothèques numériques issues de tierces parties. La bibliothèque numérique IMSL® C# sera une pièce clef d'une telle mise en ouvre par de nombreux utilisateurs. Plusieurs caractéristiques de cette bibliothèque qui permettent l'utilisation du cadre d'application .NET pour l'analyse avancée seront démontrées.

0 - Introduction

Ces dernières années, le paradigme de programmation du cadre d'application (*framework*) .NET a été adopté rapidement pour l'analyse numérique avancée avec la communauté des services financiers, connue depuis longtemps comme des développeurs précoces qui montrent la voie aux autres. Le cadre d'application .NET est une plateforme puissante pour de nombreuses raisons, y compris une plus grande productivité des programmeurs, une plus grande sécurité des types, de meilleures politiques de sécurité, mais avec des commodités prêtes à l'emploi. Toutefois, de nombreux programmeurs peuvent se poser des questions quant à l'adéquation de la plateforme pour des applications numériques avancées.

Ce document explorera certaines forces de la plateforme .NET concernant l'analyse numérique. De nombreuses astuces pour augmenter les performances des applications .NET seront présentées. De plus, comme il est développé ou adapté au cadre d'application .NET des programmes qui nécessitent des algorithmes mathématiques et statistiques, il existe un besoin naturel pour des bibliothèques numériques issues de tierces parties. La **bibliothèque numérique IMSL** sera une pièce clef d'une telle mise en ouvre par de nombreux utilisateurs. Plusieurs caractéristiques de cette bibliothèque qui permettent l'utilisation du cadre d'application .NET pour l'analyse avancée seront démontrées.

L'analyse avancée est utilisée dans de nombreuses industries lorsque les organisations évoluent de l'utilisation des outils analytiques de base, qui se contentent d'agréger des données historiques, à l'analyse prédictive, qui permet aux organisations de prédire des résultats par l'utilisation de techniques mathématiques et statistiques. L'analyse avancée équipe les organisations pour mieux planifier, modéliser de nouvelles opportunités et améliorer la précision des budgets et des prévisions. Les commerçants peuvent gérer leur inventaire avec plus de précision. Les laboratoires pharmaceutiques peuvent augmenter la productivité de leur personnel. Les entreprises de service financier peuvent mieux conserver leurs clients.

I - Avantages du cadre d'application .NET

La principale force du cadre d'application .NET est la "productivité du programmeur". Ce terme englobe de nombreuses caractéristiques, mais nous ne nous focaliserons que sur deux d'entre elles. D'abord la souplesse du langage. Le concept est que vous pouvez écrire du code dans n'importe quel langage supporté, utiliser des bibliothèques écrites dans n'importe quel autre langage supporté, et pouvoir encore profiter de tous les aspects de toute la plateforme. Que vous écriviez une application bureautique, le code derrière une application web ASP.NET, ou l'aspect client ou serveur d'un service web, vous pouvez travailler en C#, Visual Basic.NET, J#, Managed C#, ou n'importe lequel des quelque deux douzaines d'autres langages. Ainsi, des experts en langage GUI peuvent écrire en Visual Basic.NET, tandis que la couche logique métier est écrite en C++ et les interactions base de données sont écrites en C#. Chacun de ces bits de code se compilera vers le Microsoft Intermediate Language (MSIL ou juste IL) et ils pourront interagir facilement les uns avec les autres.

L'autre aspect majeur de la productivité du programmeur est la maturité et l'exhaustivité des outils disponibles. Spécifiquement, le Visual Studio IDE a évolué dans l'environnement .NET et est devenu un facteur majeur pour rendre les programmeurs plus productifs. Alors qu'il reste possible de générer des codes Windows natifs en utilisant Visual Studio IDE, ce dernier est totalement intégré dans le cadre d'application .NET. Consistant en un éditeur de code qui met en valeur la syntaxe, un débogueur et un designer GUI, tous intégrés dans différents compilateurs, Visual Studio IDE est un outil puissant. Contrairement à beaucoup d'autres environnements de programmation, les outils Visual Studio sont regroupés tous au même endroit. Un programmeur moyen peut créer n'importe quoi depuis une application bureautique à un service web en quelques minutes. Ceci combiné à la souplesse du langage de la plateforme, les bibliothèques de tierces parties, si elles sont totalement compatibles avec .NET, auront des métadonnées indépendantes du langage qui leur seront associées. Cela signifie qu'un programmeur peut ajouter une référence à une bibliothèque dans son projet Visual Studio, tandis qu'IDE analyse ces informations et fournit à la fois la complétion du code et des informations Intellisense au programmeur à mesure qu'il écrit. Avec des API comme cela à portée de main, les développeurs n'auront plus à se reporter sans arrêt à leur documentation.

Les autres forces de la plateforme incluent des politiques de déploiement et des métadonnées, la sécurité, la robustesse et la fiabilité. Chaque assemblage EXE ou DLL construit sur le cadre d'application .NET contient des

métadonnées qui fournissent des informations de version sur lui-même et sur d'autres assemblages et types qui reposent sur lui. Avec une stratégie de version ou une stratégie d'éditeur, les programmeurs peuvent éviter plus facilement le célèbre "enfer des DLL". Sur le front de la sécurité, les composants vérifiables (générés naturellement quand on utilise C# ou VB.NET et quand on utilise le registre "/clr:safe" pour C++) garantissent que le code ne peut pas violer les réglages de sécurité et que si le code échoue, il le fera d'une façon qui ne sera pas catastrophique. Permettre au Common Language Runtime (CLR) de vérifier les réglages de sécurité à l'exécution autorise une quantité sans précédent de sécurités pour vos applications. De plus, isoler le code CLR du système d'exploitation garantit que si un crash catastrophique devait survenir, ses ramifications pourraient être facilement isolées. Avec toutes ces caractéristiques ci-dessus et d'autres comme la *type safety* (où le code ne peut accéder qu'aux emplacements mémoire qu'il a le droit d'utiliser), des applications écrites pour la plateforme .NET peuvent néanmoins ne pas être exemptes de tout bug, mais elles seront plus robustes et plus sûres que beaucoup de solutions alternatives.

II - Maximiser les performances dans le cadre d'application .NET

Quand on envisage d'écrire un code à calcul intensif sur une nouvelle plateforme, la question des performances finit naturellement par survenir. Alors que le cadre d'application .NET et le CLR ont certaines caractéristiques qui conduiront à l'hypothèse que c'est un environnement de langage interprété, l'avènement de la compilation juste-à-temps (JAT) apporte avec elle des performances proches des implémentations de langage natif. Le code (initialement écrit dans n'importe quel langage supporté) est initialement compilé vers l'IL. Ce langage, alors qu'il se lit comme un langage de type assembleur, n'est pas un code que votre processeur de bureau pourrait comprendre et exécuter. à l'exécution, l'IL est compilé en un code exécutable spécifique de la machine par le compilateur JAT (JITter). Ce code sera compilé pour correspondre à l'architecture sur laquelle le code doit être exécuté (ex. Pentium IV x86, Xeon EM64T, Itanium IA64). C'est grâce à cette étape JITter bien optimisée que les performances sont beaucoup plus proches de celles du code natif que ce que n'importe quel langage interprété pourra jamais faire. Il y a naturellement un léger délai lorsque le JITter compile le code. Ce délai peut se surmonter en utilisant l'utilitaire générateur d'image natif ngen. En "ngenant" votre code, l'assemblage .NET sera précompilé et sera prêt à être exécuté dans l'environnement local.

Les performances "proches de l'environnement natif" dépendent évidemment du suivi de certaines bonnes pratiques quand on écrit des applications .NET. Voici quelques astuces pour tirer les meilleures performances de cet environnement :

II-A - Utilisation précautionneuse des types de valeur et des objets

Les types de valeur et les objets sont traités très différemment en mémoire. Les types de valeur coûtent très peu à créer et sont créés sur la pile. Inversement, instancier un objet est au moins légèrement plus coûteux, même pour des objets simples, et ils sont créés sur le tas. Fondamentalement, quand vous avez le choix d'utiliser un int ou un Integer, choisissez le int tant que c'est possible. Il y aura peut-être des cas où vous aurez besoin de reconvertir un type de valeur en une référence : on appelle cette opération boxing. Boxing et un-boxing sont des opérations relativement coûteuses et doivent être évitées autant que possible. Ce qu'il faut avoir à l'esprit, c'est que les langages .NET de haut niveau comme C# et Visual Basic.NET effectuent certains boxings automatiquement. Si vous n'y prenez pas garde, vous perdrez beaucoup d'optimisations de performances à cause des boxings automatisés dont vous pourriez ne pas avoir conscience. Quand on écrit en Managed C++, toutes les opérations de boxing doivent être faites à la main : on réalise ainsi un niveau supplémentaire de conscience.

II-B - Utilisation d'ensembles

En fonction de l'expérience individuelle du programmeur dans l'écriture de codes haute performance dans d'autres langages, il peut être naturel d'écrire une boucle en utilisant une variable locale invariante au lieu d'une propriété de l'ensemble. Par exemple, considérons le code C# suivant pour boucler dans un ensemble :

```
int len = myArray.length ;
    for(int i=0 ; i<len ; i++) {
    myArray[i] *= 2;
}
```

Ici, nous savons que la longueur de *myArray* ne va pas changer dans la boucle, aussi allons-nous la calculer une fois, conserver sa valeur dans une variable appelée *len*, puis l'utiliser pour la borne supérieure de la boucle. Il peut être surprenant de savoir que le code suivant génère effectivement quatre lignes de code IL de moins :

```
for(int i=0 ; i<len ; i++) {
    myArray[i] *= 2;
}
```

Parce que le compilateur sait quelque chose à propos de *myArray*, une seule vérification de bornes est nécessaire pour cette boucle. Ces contrôles internes profitent de la robustesse de l'utilisation de la plateforme .NET mais n'améliorent pas les performances.

Si vous savez quelque chose à propos de la longueur de *myArray* au moment du développement, il se peut que vous puissiez utiliser une constante au terme de la boucle. Vous réaliserez les meilleures performances sous la forme du nombre minimum de lignes générées par IL (deux de moins que le second exemple ci-dessus). Bien sûr, il faudra connaître la taille de *myArray* au moment de la compilation et il faudra qu'elle n'ait pas changé au moment de l'exécution.

II-B-1 - Minimiser les exceptions

En fonction une fois encore des connaissances du programmeur concernant les concepts de la Programmation Orientée Objet (POO), il peut être tentant d'interjeter des exceptions comme moyen de contrôler le flux d'un programme. à l'époque de la programmation en Fortran et en C, les programmeurs avaient plusieurs méthodes à leur disposition pour imposer des flux dans le code. Sur cette base, certains programmeurs peuvent être tentés de "tirer avantage" de certaines caractéristiques de la POO. Néanmoins, interjeter des exceptions est une opération très coûteuse, et un parcours de la pile d'appels (*stack walk*) est nécessaire. En tant que tel, interjetez des exceptions dans le cadre du design du programme ou de l'interface de programmation mais pas pour contrôler le flux d'une application.

II-D - Éviter le nettoyage manuel

Au fil des lignes, les programmeurs procéduriers ou plus généralement les programmeurs agressifs, novices dans le monde de la programmation POO, peuvent être tentés d'appeler le nettoyeur manuellement. Sous .NET, le nettoyeur est un thread qui contrôle l'utilisation d'objets et qui libère automatiquement l'espace mémoire utilisé par des objets qui ne sont plus référencés. Le schéma est un concept mûr et bien conçu. Si vous venez de libérer un grand nombre d'objets, vous pourriez être tenté d'appeler le nettoyeur pour libérer l'espace mémoire plus tôt. Toutefois, il faut l'éviter, car cela ne fera qu'interrompre le fonctionnement prévu du nettoyeur.

III - La bibliothèque IMSL C# pour applications .NET

à mesure que des applications anciennes sont transférées dans le cadre d'application .NET, leurs utilisateurs ont besoin de versions plus récentes des bibliothèques de tierces parties qui étaient utilisées dans l'application d'origine. De façon similaire, à mesure que de nouvelles applications sont écrites en langage .NET, on a besoin de nouvelles bibliothèques pour remplir les niches où sévit un manque d'expertise ou où il est trop coûteux de développer un code maison. Pour faire face à ces besoins, Visual Numerics a développé la bibliothèque numérique IMSL C#. La bibliothèque IMSL C# couvre une large gamme de fonctionnalités mathématiques et statistiques et est écrite à 100% en C#. C'est un assemblage entièrement géré que l'on peut intégrer de façon homogène dans n'importe quelle application .NET : il n'est pas besoin de se battre avec un code crypté ni d'appeler des DLL C/C++ natifs. En tant qu'assemblage géré avec métadonnées indépendantes du langage, la bibliothèque peut s'utiliser aussi facilement sous C# que sous n'importe quel autre langage .NET tel que Visual Basic.NET.

La bibliothèque IMSL C# consiste en six espaces de noms : Imsl, Imsl.Math, Imsl.Stat, Imsl.Finance, Imsl.DataMining.Neural et Imsl.Chart2D. L'essentiel de la fonctionnalité de la bibliothèque réside dans les espaces de noms Math et Stat, alors que Finance contient quelques routines utilitaires financières et les classes de gestion d'exceptions sont dans l'espace de noms Imsl. L'espace de nom DataMining inclut une implémentation de réseau neuronal et Chart2D contient des classes bureautiques souples et des classes graphiques (*charting classes*) sur

base web. Quand on travaille sur la bibliothèque de classes de base standard .NET, la fonctionnalité fournie par la bibliothèque IMSL C# à la fois étend le cadre d'application .NET, pour permettre aux utilisateurs d'écrire leurs propres algorithmes numériques personnalisés, et utilise des algorithmes éprouvés depuis plus de 35 ans.

La bibliothèque IMSL C# s'étend dans plusieurs domaines clefs pour permettre aux programmeurs d'écrire des routines d'analyse avancée :

III-A - Nombres complexes

Les nombres complexes sont nécessaires pour différentes raisons en calcul numérique, mais il n'y a pas de support pour eux incorporé dans le cadre d'application .NET. La bibliothèque IMSL C# inclut l'implémentation Imsl.Math.Complex en tant que structure inaltérable. Cette structure a été choisie pour permettre à des objets complexes de se comporter autant que possible comme des types de données primitifs. Un certain nombre de méthodes comme Conjugate, Real, Abs, Cos, etc. sont incluses pour effectuer toute une gamme de calculs de base sur un nombre complexe. De plus, des opérateurs standards tels que l'addition, la division, la multiplication, la soustraction, l'égalité, l'inégalité et la valeur négative sont fournis pour tirer avantage des caractéristiques de surcharge d'opérateurs de .NET. Finalement, le constructeur est surchargé pour permettre aux utilisateurs de créer un nombre complexe à partir d'un autre nombre complexe, un double réel, ou en fournissant les parties réelles et imaginaires en tant que valeurs à double décimale.

III-A-1 - Fonctions de distribution

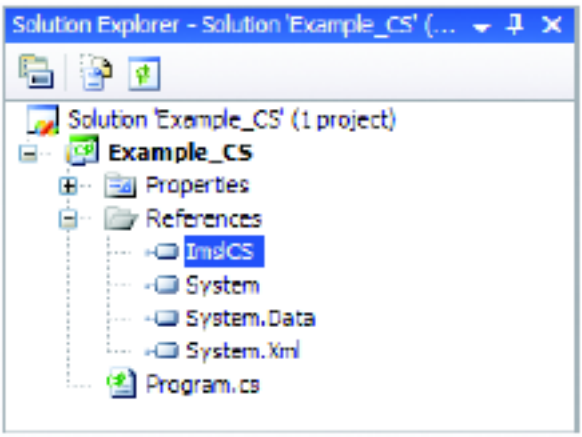
Pour créer de nombreuses espèces de routines d'analyse statistique, il est nécessaire de calculer diverses fonctions de distribution. Cette fonctionnalité est fournie par la classe Imsl.Stat.Cdf avec plus de 50 méthodes différentes qui retournent différentes fonctions de distribution cumulatives, fonctions de densité de probabilité, et leurs réciproques. Dans le même contexte, pour créer des modes statistiques, différentes distributions de nombres aléatoires sont également nécessaires. La classe Imsl.Stat.Random encrypte System.Random pour retourner un "NextDouble", mais a aussi la capacité de retourner le "Next" de plus de 20 autres distributions. Notez qu'il existe aussi une implémentation Mersenne Twister qui peut remplacer le générateur standard System.Random.

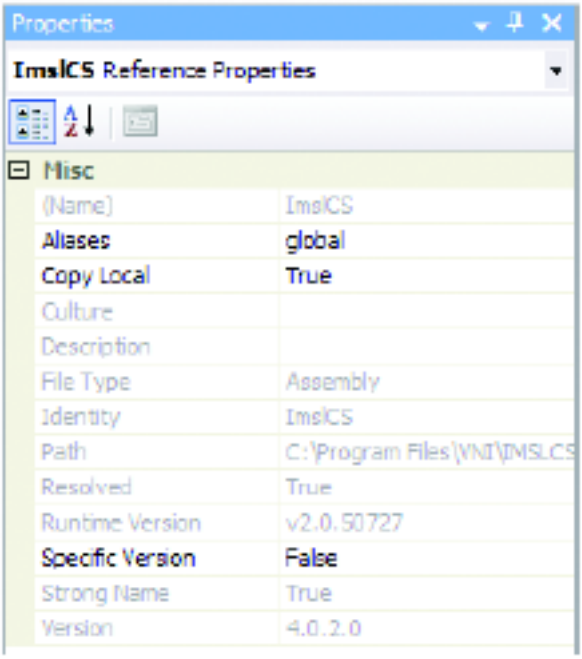
III-C - Manipulations de matrices

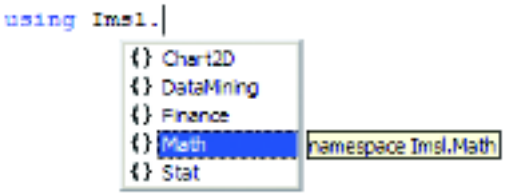
La capacité de manipuler des matrices d'une façon intelligente est également de toute première importance pour de nombreux types différents d'applications numériques. Sur ce point, la bibliothèque IMSL C# inclut la classe Imsl.Math.Matrix qui est entièrement constituée de méthodes statiques publiques pour traiter un ensemble standard double[,] comme une matrice. Les méthodes sont typiques : addition, multiplication, soustraction, transposition, CheckSquareMatrix, ForbeniusNorm, InfinityNorm et OneNorm. La seule autre opération que vous devriez normalement aimer avoir est l'inversion de matrice, et il existe plusieurs options incorporées ici dans d'autres classes (LU, ComplexLU, Cholesky et SVD) puisque l'inversion de matrice est un sous-produit de ces décompositions. Des routines de manipulation de matrices complexes sont également disponibles.

IV - Utilisation de Visual Studio et de la bibliothèque IMSL C#

La plupart des développeurs qui écrivent pour le cadre d'application .NET utiliseront Visual Studio comme leur outil de choix. Visual Studio est un environnement de développement intégré (IDE) robuste avec un excellent support et une excellente connaissance des langages .NET. à partir de cette unique application, les utilisateurs peuvent écrire des notes, concevoir des interfaces, compiler des applications et déboguer le produit final. Utiliser la bibliothèque IMSL C# est très immédiat, et puisqu'elle est purement constituée de code C# et de code géré, Visual Studio est capable de rendre la programmation par cet assemblage plus facile que jamais.





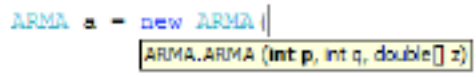


Pour commencer, chargez un projet .NET existant, ou créez-en un nouveau. Peu importe le langage utilisé. L'étape suivante consiste à ajouter une référence à l'assemblage de la bibliothèque IMSL C#, ImslCD.dll, au projet. Cela peut se faire en cliquant avec le bouton droit de la souris sur le projet dans Solution Explorer et en sélectionnant "Add reference.", ou en sélectionnant "Project" dans la barre de menu et en sélectionnant "Add reference.". D'une façon ou d'une autre, la boîte de dialogue "Add reference" apparaîtra. Cherchez l'assemblage ImslCD.dll et cliquez sur OK pour l'ajouter au projet. Notez que si vous avez ajouté l'assemblage au cache de code Global Assembly Cache (GAC), la propriété Copy Local sera réglée sur False et le répertoire des fichiers du projet ne contiendra aucune copie du dll. Sinon, Copy Local sera réglée sur True et ImslCD.dll sera copié dans le répertoire approprié aux termes du projet.

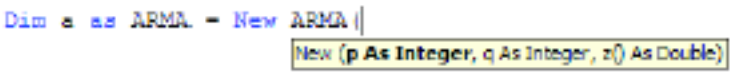
À ce point, toutes les fonctionnalités de la bibliothèque IMSL C# sont à portée de vos doigts. Essayez-les en ajoutant un espace de noms à coder dans votre projet. Tout en haut de votre programme, ajoutez une ligne telle que "using Imsl.Math;" (pour C#), "Imports Imsl.Math" (pour VB.NET) ou "using namespace Imsl::Math;" (pour C++). Cette ligne dépendra bien évidemment du langage de votre projet, mais vous devriez avoir noté qu'après avoir tapé "Imsl.", la boîte de dialogue d'achèvement de code est apparue avec une liste de tous les espaces de noms accessibles dans l'assemblage. Cela indique que la référence a été ajoutée convenablement et que les algorithmes IMSL peuvent être facilement ajoutés à votre application.

Note sur la facilité d'utilisation et les capacités inter-langages : Envisagez le cas où vous auriez besoin de faire des prévisions et aimeriez utiliser un modèle ARMA. La bibliothèque IMSL C# contient une classe appelée ARMA, et une documentation est disponible en copie papier ou électronique à la fois à la fois en version d'aide compilée (chm) et en version PDF. Mais vous êtes en pleine écriture, et vous n'aimeriez pas vous arrêter pour extraire la documentation puis rappeler les paramètres nécessaires à ce constructeur de classe. Ici, une caractéristiques de Visual Studio appelée Intellisense vous viendra en aide. Pour les trois différents langages mentionnés plus haut, voici quelques captures d'écran de ce qu'Intellisense vous montrera quand vous essaierez de créer un objet ARMA :

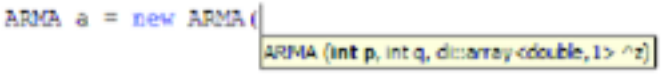
Notez que la liste de paramètres pour le constructeur apparaît dans la langue du projet. Ainsi, quand l'algorithme est écrit en C#, les métadonnées indépendantes du langage permettent à Visual Studio de présenter le code tel qu'attendu dans l'environnement actuel.



En C#



En VB.NET



Dans les extensions gérées de C++

V - Conclusion

La plateforme .NET est une option viable pour l'écriture d'applications d'analyse numérique hautes performances et pour le portage d'applications numériques anciennes. Le haut niveau de productivité des programmeurs pour écrire dans un cadre d'application moderne dans ce genre est très profitable, à la fois aujourd'hui au moment de l'écriture du code, et demain au moment où il faudra l'entretenir. La compréhension de la programmation orientée objet et de certaines implémentations de .NET s'avère être une aide pour tirer les meilleures performances de l'application. Des bibliothèques de tierces parties comme la bibliothèque numérique IMSL C# pour les applications Microsoft .NET permettent aux développeurs de se concentrer sur l'écriture de meilleures applications sans écrire d'algorithmes complexes. La réutilisation de codes de qualité commerciale de cette manière économise du temps à la fois au départ et aussi sur le long terme, en réduisant le temps nécessaire à documenter, à tester et à entretenir le code. Intégrer ce genre de composants dans des IDE tels que Visual Studio est facile et augmente encore la productivité du programmeur.