

High Performance Applications with Advanced Numerical Analysis on the .NET Framework

A White Paper by Visual Numerics, Inc.
September 2007

Visual Numerics, Inc.
2500 Wilcrest Drive, Suite 200
Houston, TX 77042
USA

www.vni.com

High Performance Applications with Advanced Numerical Analysis on the .NET Framework

by Visual Numerics, Inc.

Copyright © 2005-2007 by Visual Numerics, Inc. All rights reserved.
Printed in the United States of America.

Publishing History:

September 2007 – Updated

September 2005 – Initial publication

Trademark Information

Visual Numerics, IMSL and PV-WAVE are registered trademarks of Visual Numerics, Inc. in the U.S. and other countries. JMSL, JWAVE, TS-WAVE and Knowledge in Motion are trademarks of Visual Numerics, Inc. All other company, product or brand names are the property of their respective owners.

The information contained in this document is subject to change without notice. Visual Numerics, Inc., makes no warranty of any kind with regard to this material, included, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Visual Numerics, Inc., shall not be liable for errors contained herein or for incidental, consequential, or other indirect damages in connection with the furnishing, performance, or use of this material.

TABLE OF CONTENTS

Introduction	4
Advantages of the .NET Framework	4
Maximizing Performance in the .NET Framework	5
The IMSL C# Library for .NET Applications	7
Using Visual Studio and the IMSL C# Library	8
Conclusion	9

Introduction

In recent years, the programming paradigm of the .NET Framework has quickly been adopted for advanced numerical analysis with the financial services community, long known as early adopters leading the movement. The .NET Framework is a strong platform for many reasons, including advanced programmer productivity, type safety, security policies, but with off-the-shelf convenience. However, many programmers may question the suitability of the platform for advanced numerical applications.

This paper will explore some of the strengths of the .NET platform with respect to numerical analysis. Many tips for increasing the performance of .NET applications will be presented. Additionally, as programs that require advanced mathematical and statistical algorithms are developed or ported to the .NET Framework, there is a natural need for third party numerical libraries. The IMSL® C# Numerical Library will be a key part of such an implementation for many users. Several features of this library that allow the .NET Framework to be used for advanced analytics will be demonstrated.

Advanced analytics are being used across multiple industries as organizations evolve from using basic analytics that simply summarize historical data to predictive analytics, which enable organizations to forecast outcomes through the use of mathematical and statistical techniques. Advanced analytics equip organizations to plan better, model new opportunities and improve the accuracy of budgets and forecasts. Retailers can more accurately manage inventory, healthcare companies can increase staff productivity, and financial services companies can improve customer retention.

Advantages of the .NET Framework

The major strength of the .NET Framework is “programmer productivity.” This term encompasses many features, but we will focus on just two. First is language flexibility. The concept is that you can write code in any supported language, use libraries written in any other supported language, and still be able to take advantage of all the aspects of the entire platform. Whether you are writing a desktop application, code behind an ASP.NET web application, or either client or server side of a Web Service, you can work in C#, Visual Basic.NET, J#, Managed C++, or any of the over two dozen third party languages. So, GUI experts can write in Visual Basic.NET, while the business logic tier is written in C++, and database interaction is written in C#. Each of these bits of code will compile to the Microsoft Intermediate Language (MSIL or just IL) and can interoperate with each other seamlessly.

The other major aspect of programmer productivity is the maturity and completeness of the toolset available. Specifically, the Visual Studio IDE has evolved into the .NET world and become a major factor in making programmers more productive. While the power is still there to generate native Windows code using the Visual Studio IDE, it is fully integrated with the .NET Framework. Consisting of a syntax-highlighting code editor, a debugger, and a GUI designer all integrated with various compilers, Visual Studio is a powerful tool. Unlike many other programming environments, Visual Studio tools are consolidated in one place. An average programmer can create anything from a desktop application to a Web Service in just a matter of minutes. Combined with the language flexibility of the platform, third party libraries, if fully .NET-compliant, will have language-independent metadata associated with them. This means a programmer can add a reference to a library in their Visual Studio project, while the IDE parses this information and provides both code-completion and Intellisense information to the programmer as they write. With APIs at their fingertips like this, developers will no longer be pouring over documentation repeatedly.

Other strengths of the platform include deployment policies and metadata, security, robustness and reliability. Each EXE or DLL assembly built on the .NET Framework contains metadata that provides versioning information about itself and other assemblies and types it relies on. Together with a Version Policy or Publisher Policy, developers can more easily avoid the infamous “DLL Hell” syndrome. On the security front, verifiable components (generated naturally when using C# or VB.NET, and using the “/clr:safe” flag for C++) guarantees that code cannot violate security settings and that if code fails, it does so in a non-catastrophic manner. Allowing the Common Language Runtime (CLR) to verify security settings at runtime allows an unprecedented amount of security to your applications. Further, having the CLR isolate code from the operating system level ensures that should a disastrous crash occur, its ramifications can be easily isolated. With all of the above features and others such as type-safety (where code can only access memory locations it is authorized to use), applications written for the .NET platform may still not be bug-free, but will be more robust and more secure than many alternatives.

Maximizing Performance in the .NET Framework

As one considers writing computationally intensive code on a new platform, the question of performance will naturally arise. While the .NET Framework and the CLR have some features that will lead to the assumption that it is an interpreted language environment, the advent of Just-in-time (JIT) compilation brings with it performance close to native language implementations. Code (written in any supported language) is initially compiled to IL. This language, while it reads like assembly language, is not code that your desktop processor could understand and execute. At runtime, IL is compiled to machine-specific natively executable code by the just-in-time compiler (JITter). This code will be compiled to match the architecture on which the code is to be executed (for example, Pentium IV x86 versus Xeon EM64T versus Itanium IA64). It is through this well-optimized JITter step that performance is much closer to native code than any interpreted language could ever be. There is of course a slight delay as the JITter compiles the code. This delay can be overcome by using the Native Image Generator Utility called ngen. By “ngening” your code, the .NET assembly will be pre-compiled and is ready to be executed on the local environment.

Performance “close to native” of course depends on following some best practices when writing .NET applications. The following are some tips to get the best performance out of this environment.

- 1. Careful Use of Value Types and Objects:** Value types and objects are treated very differently in memory. Value types are very inexpensive to create and are created on the stack. Conversely, instantiating an object is at least slightly more costly, even for simple objects, and they are created on the heap. Basically, when you have the choice of using an int or an Integer, choose the int whenever possible. There may be times you need to then convert a value type to a reference; this operation is called boxing. Boxing and un-boxing is a relatively expensive operation and should be avoided if possible. The catch is that the high-level .NET languages like C# and Visual Basic.NET will do some boxing automatically. If you are not careful, many performance optimizations are lost because of automated boxing that you may not be aware of. When writing in Managed C++, all boxing operations must be done by hand, and so an extra level of awareness is realized.

2. Using Arrays: Depending on a programmer's individual experience writing high-performance code in other languages, it may be natural to write a loop using an invariant local variable instead of using a property of the array. For example, consider the following C# code to loop through an array:

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

Here we know the length of *myArray* is not going to change inside the loop, so we'll compute it one time, store the value in a variable called *len*, and then use that for the loop's upper bound. It may be surprising to know that the following code actually generates four fewer lines of IL code:

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

Because the compiler knows something about *myArray*, there is only a single bounds check required for this loop. In the first case, the variable *len* may change inside the loop, and so extra work is required to be sure *myArray* is not addressed outside of its bounds. These internal checks benefit the robustness of using the .NET platform, but do not enhance performance.

If you know something about the length of *myArray* at development time, you may be able to use a constant as the endpoint of the loop. Optimal performance will be realized here as the fewest lines of IL generated (two fewer than the second example above). Of course, the size of *myArray* will have to be known at compile time and could not change at runtime.

3. Minimize Throwing Exceptions: Depending again on a programmer's knowledge of Object Oriented Programming (OOP) concepts, it may be tempting to throw Exceptions as a way of controlling the flow of a program. In the procedural programming days of C and Fortran, programmers had many methods available to them to dictate flow through the code. Coming from this background, some developers may be tempted to "take advantage" of some of the characteristics of OOP. However, throwing Exceptions is a very expensive operation, and a stack walk is required. As such, throw Exceptions as program design or API designates, but not to control the flow of an application.

4. Avoid Manual Garbage Collection: Further along the lines of procedural programmers, or aggressive programmers in general, new to an OO world, one may be tempted to call the garbage collector manually. The garbage collector in .NET is a separate thread that monitors the use of objects and will automatically free memory used by objects that are no longer referenced. The scheme is a mature and well designed concept. If you have just released a large number of objects, there may be temptation to call the garbage collector to release their memory sooner. However, this should be avoided as it will only serve to interrupt the garbage collector's designed operation.

The IMSL C# Library for Microsoft .NET Applications

As legacy applications are ported for the .NET Framework, users will need newer versions of any third party libraries that were used in the original application. Similarly, as new applications are written in .NET languages, libraries will be required to fill the niches where expertise is lacking or where it is too costly to develop code in-house. To meet these needs, Visual Numerics has developed the IMSL C# Numerical Library. The IMSL C# Library covers a broad range of mathematical and statistical functionality and is written in 100% C#. It is a fully managed assembly that can be integrated seamlessly with any .NET application: there is no need to wrangle with wrapped code or to call native C/C++ DLLs. As fully managed code with language-independent metadata, the library can be used as easily from C# as any other .NET language like Visual Basic.NET.

The IMSL C# Library consists of six namespaces: `Imsl`, `Imsl.Math`, `Imsl.Stat`, `Imsl.Finance`, `Imsl.DataMining.Neural`, and `Imsl.Chart2D`. The bulk of the library's functionality resides in the `Math` and `Stat` namespaces, while some utility financial routines are in `Finance` and Exception-handling classes are in the `Imsl` namespace. The `DataMining` namespace includes a Neural Network implementation and `Chart2D` contains flexible desktop and web-based charting classes. Building on the standard .NET Base Class Library, the functionality provided by the IMSL C# Library both extends the .NET Framework to enable users to write their own custom numerical algorithms and also utilizes more than 35 years worth of proven algorithms.

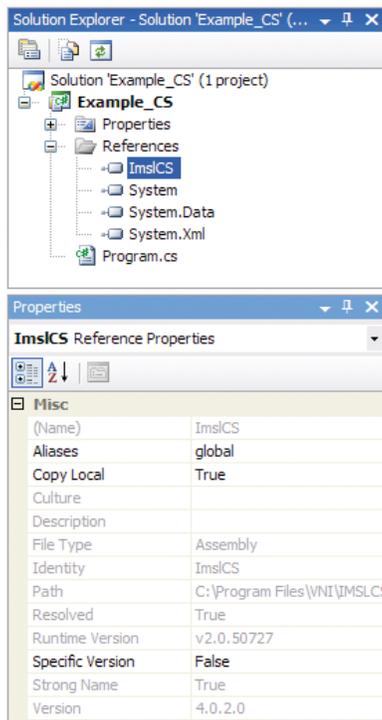
The IMSL C# Library is extended in several key areas to allow programmers to write advanced analysis routines:

- 1. *Complex Numbers:*** Complex numbers are required for a variety of reasons in numerical computing, but there is no support for them built into the .NET Framework. The IMSL C# Library includes the `Imsl.Math.Complex` implementation as an immutable struct. This structure was chosen to allow Complex objects to behave as much like primitive data types as possible. A number of methods like `Conjugate`, `Real`, `Abs`, `Cos`, etc. are included to perform a range of basic computations of a complex number. Additionally, standard operators like `Addition`, `Division`, `Multiplication`, `Subtraction`, `Equality`, `Inequality`, and `Unary Negation` are provided taking advantage of .NET's operator overloading features. Finally the constructor is overloaded to allow users to create a complex number from another complex number, a real double, or by providing the real and imaginary parts as double precision values.
- 2. *Distribution Functions:*** To build many kinds of statistical analysis routines, it is necessary to compute various distribution functions. This functionality is provided by the `Imsl.Stat.Cdf` class with over 50 different methods that return different Cumulative Distribution Functions, Probability Density Functions, and their inverses. Along the same lines, to create statistical models, various distributions of random numbers are also required. The `Imsl.Stat.Random` class wraps `System.Random` to return a "NextDouble" but also has the capability to return the "Next" of more than 20 other distributions. Note that there is also a Mersenne Twister implementation that can replace the standard `System.Random` generator.
- 3. *Matrix Manipulation:*** The ability to manipulate matrices in a meaningful way is also paramount to many different kinds of numerical applications. To this point, the IMSL C# Library includes the `Imsl.Math.Matrix` class which is made entirely of public static methods to treat a standard `double[,]` array as a matrix. The methods are typical: `Add`, `Multiply`, `Subtract`, `Transpose`, `CheckSquareMatrix`, `ForbeniusNorm`, `InfinityNorm`, and `OneNorm`. The only other operation you would normally like to

have is a matrix inversion, and there are several options here embedded in other classes (LU, ComplexLU, Cholesky and SVD) since the inverted matrix is a byproduct of these decompositions. Complex Matrix manipulation routines are available as well.

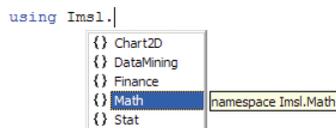
Using Visual Studio and the IMSL C# Library

Most developers writing for the .NET framework will be using Visual Studio as their tool of choice. Visual Studio is a robust and mature integrated development environment (IDE) with excellent support and knowledge about the .NET languages. From this single application, users can write code, design interfaces, compile applications, and debug the final product. Using the IMSL C# Library is very straightforward, and because it is pure C# and managed code, Visual Studio is able to make programming using the assembly easier than ever.



To start, load an existing .NET project, or create a new one; the language used does not matter. The next step is to add a reference to the IMSL C# library assembly, `ImslCS.dll`, to the project. This can be accomplished by right-clicking on the project in the Solution Explorer and selecting “Add Reference...” or selecting “Project” off the menu bar and selecting “Add Reference...”. Either way, the “Add Reference” dialog will appear. Browse to the `ImslCS.dll` assembly and click OK to add it to the project. Note that if you have added the assembly to the Global Assembly Cache (GAC) the Copy Local property will be set to false, the project’s bin directory will not contain a copy of the dll. Otherwise, Copy Local will be true, and `ImslCS.dll` will be copied to the appropriate bin directory under the project.

At this point, all of the functionality of the IMSL C# Library is at your fingertips. Try it out by adding a namespace to code in your project. At the top of your program, add a line like “using `Imsl.Math`;” (for C#) “Imports `Imsl.Math` (for VB.NET) or “using namespace `Imsl::Math`;” (for C++). This line will obviously depend on the language of the project, but you should have noticed that after typing “`Imsl.`” the code-completion dialog appeared with a list of all the namespaces accessible in the assembly. This signals the reference has been added properly and the IMSL algorithms can be easily added to your application.



One note on ease-of-use and cross-language capabilities: consider the case where you need to do some forecasting and would like to use an ARMA model. The IMSL C# Library contains a class called

ARMA, and documentation is available in hardcopy or electronically in both compiled help (chm) and PDF versions. But you are in the middle of writing and would rather not stop to pull up the documentation to recall the parameters required for this class’s constructor. Here, a Visual Studio feature called Intellisense will help out. For the three different languages mentioned above, here are some screenshots of what Intellisense will show as you try to create an ARMA object.

In C#: `ARMA a = new ARMA(|
ARMA.ARMA (int p, int q, double[] z)`

In VB.NET: `Dim a as ARMA = New ARMA(|
New (p As Integer, q As Integer, z() As Double)`

In Managed Extensions for C++: `ARMA a = new ARMA(|
ARMA (int p, int q, cli::array<double,1> ^z)`

Notice how the parameter list for the constructor appears in the language of the project. So while the algorithm is written in C#, the language-independent metadata allows Visual Studio to present code as expected in the current environment.

Conclusion

The .NET platform is a viable option for writing high-performance numerical analysis applications and for porting legacy numerical applications. The high level of programmer productivity writing for such a modern framework is very valuable both as code is written and in the future as code is maintained. Understanding of object-oriented programming and some implementations of .NET prove helpful in wringing additional performance out of applications. Third party libraries such as the IMSL C# Numerical Library for Microsoft .NET Applications allow developers to concentrate on writing better applications without writing complex algorithms. The reuse of commercial-quality code in this manner saves time both initially and also in the long run by reducing the time needed to document, test and maintain the code. Integrating such components into IDEs like Visual Studio is easy and further adds to the programmer's productivity.