Sci♯: User Manual

Alejandro Piad

March 13, 2013

Contents

Preface

Sci♯ is a C# library for scientific computing applications. It contains a set of frameworks for a number of areas of scientific computing, such as optimization, simulation, graph searching, probabilities and statistics, language processing, numerical algorithms, computational geometry and number theory. It also provides tools for testing and comparison of algorithms and for the generation of reports in several formats.

The main goal of **Sci♯** is to provide a set of tools for the scientific community to develop algorithms in .NET for a very broad range of applications. Despite being written entirely in C# it aims to provide efficient implementations, based on parallel, distributed and GPU programming techniques.

This report describes the main functionalities built into the library, and provides a guide for its use and extension. It is by no mean a complete nor updated guide. **Sci♯** is always changing, and the best place to find up to date and extensive information on the nasty details of the library is on its website.

The rest of the report is divided into chapters, one for each of the **Sci♯** main *namespaces*. Each chapter builds upon one or more examples in a tutorial-like fashion. There are of course many details that escape these examples. For some of the most important details, footnotes or other indications will be given.

Prerequisites

Sci♯ is a library written in C# for scientific computing. As such, this manual considers on behalf of the reader a working understanding both of the C# language and the scientific topics covered by the library.

The algorithms and data structures implemented in the library are in most cases well known in the scientific community. In such cases, the first time a given algorithm, data structure or any other important concept is mentioned in this manual, a reference will be provided to the corresponding Wikipedia page. All non-standard modifications will be explained in detail proportional to how strange or new is the given modification, and when possible references will be provided from the corresponding papers or websites.

About the Source Code

All code used throughout the manual is written in C#, adhering to the style used in by the developers of the library. Being a .NET library, all functionality can be used in any CLR compliant language. Most the examples used in the

manual are shipped with the source code for the library. Source code will be formatted in `monospace` font, with the corresponding syntax highlight.

The **Sci♯** library has been designed following modern design patterns and coding styles. Throughout the library we have tried to be consistent with a unified style, which will be used in the code examples for this manual. Being a scientific library, we have tried also to provide support for some flavoured syntax styles widely used in some scientific packages, such as Matlab and Octave. We have also tried to supply a syntax as close as possible to that used by academicians in the corresponding areas.

We have implemented this syntax sugar with heavy use of operators overloading and other fluid interfaces techniques. This means that there are almost always two ways of doing things, with the usual C# syntax (dot notation and such) and with the flavoured syntax which may involve using some strange operators and method names, in order to get a code that looks familiar to the mathematician eye.

We have designed the library this way because we feel it is more natural for the scientist, but it could be weird for the programmer used to the clean dot notation syntax. If you ever feel uncomfortable using the flavoured syntax, you can always find a way of doing the same thing using standard C# syntax.

We particularly prefer the syntax sugar [1] whenever possible, because we designed the library this way, it has become a common language for the library developers, and in the code examples you'll find it anywhere. In any case, in this manual we will always provide hints of how to circumvent the syntax sugar in favour of a more C#-ist syntax.

What this Manual Is Not

This manual is **not** a reference for algorithms and data structures. You will not find implementation details for the algorithms, nor proves of asymptotic complexity, or any other theoretical details of the implemented tools, other than the very basic definitions.

This manual does **not** explain which algorithm or technique is better in which case, nor provides comparisons between algorithms other than (perhaps) some information about our particular implementation that we think you may need to know. For instance, we will never say that *QuickSort* is faster than *MergeSort* (well, I just did, but that is not the point). We could say, however, that **our** *QuickSort* implementation is faster than **our** *MergeSort* implementation.

You **will** find, however, links to relevant places where you can start doing some research about the techniques implemented. You can also look at the source code in any moment, where you'll find all implementation details you want (is all in there, isn't it?). Also, the examples have been designed trying to choose interesting or at least classic problems that can be solved using the corresponding techniques.

But always keep in mind that we are just giving you a hammer, it is your choice how to use it: to build a house, to bake some cookies, or to break some-

[1] There is a common joke that says: *Syntax sugar causes cancer of the semicolon.*

one's head. The examples we provide will almost always be about house building, in fewer cases about cooking, but *almost* never about head smashing.

Chapter 1

Core

The root *namespace* in **Sci♯** contains basic tools used by the rest of the framework. This includes the basic matrix and vector operations, function definition and composition, definition of exceptions specific to the framework, some exension methods for `IEnumerable<T>`, logging and debugging tools, and some basic data types. The `Vector` and `Matrix` classes provide algebraic operations on these data types. The `Function` static class and related classes and interfaces provide the functionality for function definition, composition and differentiation.

Other important classes in the root *namespace* are `Engine`, `Debug`, `Logger` and `Serializers`. These are static classes which store configuration parameters for the rest of the framework. The `Engine` class deals with numerical comparisons and the configurations for the parallel, distributed and GPU implementations of the algorithms which support these features. The `Debug` class provides methods to track the execution of algorithms. The `Logger` class provides methods to generate logging data. The `Serializers` class provides methods for saving and loading data in various formats.

1.1 Algebraic Operations

The `Vector` and `Matrix` classes provide an object-oriented abstraction of the corresponding mathematical entities. Both classes provide the corresponding definitions for arithmetic operations (using static methods, instance methods and operator overloading) and indexers.

1.2 The `Engine` Class

1.3 Debugging and Logging

1.4 Serialization

1.5 Configuration Contexts

The four classes seen in the previous sections store configuration values in static properties. To help setting and restoring these values, the classes are built around the concept of *contexts*. A context is an instance of the `IContext` interface, which stores internal data necessary to restore configuration values to the specific class. Contexts are created using the static method `OpenContext` in each of the specified classes, and destroyed using the `CloseContext` method of the `IContext` instance. When a context is closed, it restores all configuration properties to the values they had before the context creation.

As an example, take a look at the following line of code.

```
Serializers.PathPrefix = "FolderA";

var context = Serializers.OpenContext();

// All changes after this line will be undo
Serializers.PathPrefix = "FolderB"

// Do your stuff

context.CloseContext();

Debug.Assert(Serializers.PathPrefix == "FolderA");
```

As you can see if you run this code, the previous value for the `PathPrefix` property is restored after the call to `CloseContext`. This functionality becomes increasingly useful if you nest context creations. The basic idea is that if you need to change some configuration data inside your algorithm, you do it inside a context, so that the configuration setted by the caller is restored before your algorithm ends.

The `IContext` interface implements `IDisposable`, which allows the `using` idiom to be employed, for easier manipulation of contexts. The previous example can be rewritten in the following form;

```
Serializers.PathPrefix = "FolderA";

using(Serializers.OpenContext()}
{
    // All changes after this line will be undo
    Serializers.PathPrefix = "FolderB"

    // Do your stuff
```

```
}

Debug.Assert(Serializers.PathPrefix == "FolderA");
```

Notice that the `using` instruction takes care of calling `Dispose`, which in return calls `CloseContext`. Since the `context` variable is unused inside the `using` block, there is no need to declare it. This idiom also makes explicit the contexts nesting which avoids the common mistake of closing contexts in the wrong order. It is also cleaner since you don't get to see any alien `IContext` instance in your code.

1.6 Extension Methods

1.7 The `Scanner` Class

Chapter 2

Benchmarks

Chapter 3

Classification

Chapter 4

Clustering

Chapter 5

Collections

Chapter 6

Game Theory

Chapter 7

Geometry

Chapter 8

Language

The `Language` *namespace* contains a framework for the definition, parsing and processing of languages, built around the concepts of grammars, automata and regular expressions.

8.1 Grammars

In the `Grammars` namespace you'll find an embedded DSL for the definition of context-free attributed grammars. The basic class is the `Grammar<T>` class (and its non-generic counterpart `Grammar`). This class is used to define a context-free grammar whose rule attributes are defined by the generic parameter type.

The grammar building is based on types that encapsulates the concepts of non-terminal and terminal symbols, with defined operations for the construction of production rules. On top of this classes, making a heavy use of operator overloading and other fluid interfaces techniques you'll find an embedded DSL which aims to provide a syntax as near as possible to the EBNF notation used in the formal languages community.

To use this DSL, you only interact directly with the `Grammar<T>` class.

8.1.1 Defining Symbols and Productions

Let's begin with the construction of a parser for the very simple language $L = \{w = a^n b^n | n > 0\}$. This is a classic context free language that will serve as base to show how to define symbols and productions. Since we don't require the use of attributes, we'll be using the non-generic `Grammar` class.

We first need a `Grammar` instance, which is created simply by calling its constructor.

```
var G = new Grammar();
```

Now that we have a grammar, let's define the symbols (both non-terminal and terminals). Non-terminal symbols are called *rules*, while terminal symbols are called *tokens*. To define symbols we can use the methods `Rule` and `Token` respectively.

```
var S = G.Rule("S");
var a = G.Token('a');
var b = G.Token('b');
```

In the case of rules, you can pass in a name as an argument, just for debugging purposes. In the case of tokens you need to provide either a simple `char` or a `string` which will be the regular expression used to parse the token. You can provide an optional name as well, but when you define tokens using a `char` it is used for the name too.

Now it's time to define the production rules. For this very simple grammar there is only very simple rule. Productions are defined using the `%=` operator for definition (like the arrow $\rightarrow$ in EBNF) [1], the `+` operator for symbol concatenation [2], and the `|` operator for branches, just like in EBNF.

```
S %= a + S + b | a + b;
```

Now that the grammar is complete, we can test it by building a parser for it or generating a lot of strings out of it (more on these topics later).

You could have skipped the definition of tokens `a` and `b`, and use strings directly in the definition for the production, like this:

```
S %= "a" + S + "b" | "ab";
```

This way you don't have to define dummy token just for a character or a string. The `Grammar` class takes care of creating the token. Also, if you use the same string more than once, you'll get the same `Token` instance.

[1] We would have prefer to use the `>>` operator, but unfortunately it can only be redefined with an `int` argument in C♯

[2] In EBNF symbol concatenation is defined just by placing symbols side of each other, but this is not possible in C♯

Chapter 9

Number Theory

Chapter 10

Numerics

Chapter 11

Optimization

Chapter 12

Path Finding

Chapter 13

Probabilities

Chapter 14

Reporting

Chapter 15

Simulation

Chapter 16

Sorting

Chapter 17

Tools

Reference

This chapter lists all public type members in **Sci♯** . Types are organized by *namespace*. All references are cross-linked and linked to the corresponding index entry.