GMac: The Next Generation (2 of 2)

by Ahmad Eid - Friday, February 10, 2017

https://gacomputing.info/2017/02/10/gmac-tng-2/

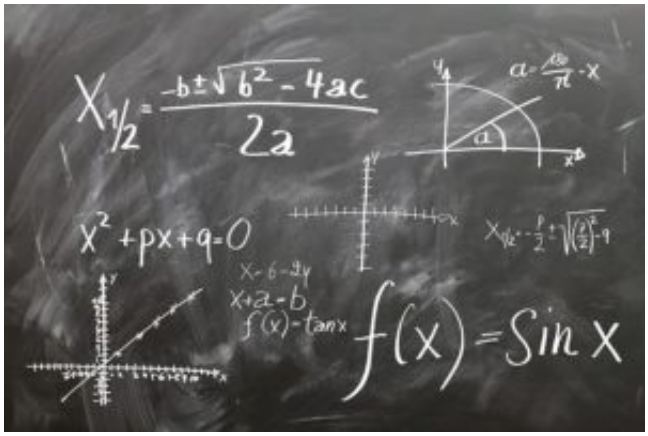
There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult. -- C. A. R. Hoare.

[dropcap]P[/dropcap]lanning for the next generation of GMac began in August 2011. I started to design the new version of GMac from scratch by reading significant parts of Terence Parr's book "Language Implementation Patterns" [1. Terence Parr is the creator of **ANTLR (AN**other **T**ool for **L**anguage **R**ecognition**)**; a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees.], Robert W. Sebesta's classic book "Concepts of Programming Languages", and the second edition of the bestseller Dragon Book " Compilers: Principles, Techniques, and Tools". I had learned many lessons during developing the first GMac prototype. These books provided a solid conceptual framework for designing the new version of GMac containing all the lessons I'd learned before.

In the previous post, I talked about the first part of my journey developing GMac, the fascinating discoveries I made, and the difficulties I faced along the way. In this final part, I explain the design decisions I made for GMac and how I came to select them, in addition to the developments I hope to make in the future.

Having a Symbolic Value

> ... with proper design, the features come cheaply. This approach is arduous, but continues to succeed. --Dennis Ritchie.



The first line of code I wrote was in the most basic of its foundations; namely the interface between GMac and a mathematical symbolic processing system like Mathematica. In addition, I created a library of utilities and basic data structures that grew over time called **UtilsLib**. I initially started out trying to design a single library that is capable of unifying and handling symbolic processing for a number of **Computer Algebra Systems (CAS)** like Mathematica, Maple, Mathcad, SageMath, Maxima, etc. In order to create such class library, I would have to learn the basics of all these systems to abstract a common interface. This was a huge project by itself that would delay my dream considerably. I eventually concentrated on designing an Object Oriented (OO) interface to Mathematica's NET Link API in the hope that I would have the time to expand it later to other symbolic processing systems.

Computer algebra systems mostly use homogeneous trees of objects, like Mathematica's Expr class, to store information about the structure of all their expressions. These homogenous trees are suitable for

computer algebra but not easy to handle using OO design. This is a similar problem to needing an OO interface between tables in relational databases and the OO applications interacting with them, we call that the Data Access Layer **(DAL)**.

The final result is my SymbolicInterface component class library, a kind of **Symbolic Processing Access Layer (SPAL)** to Mathematica. The main purpose of this library is to add low-level symbolic processing capabilities to GMac on symbolic scalars. A symbolic scalar is the most basic unit of computation that GMac can handle; it represents an expression that can be evaluated into a real number after substituting values into its symbolic variables if any are present. For example, all the following are basic symbolic scalars in GMac based on Mathematica syntax for symbolic expressions:

- -109
- 2.3456
- Rational[6,11]
- Pi
- Sin[2*Pi/3]
- Exp[-t/5]
- x + 3 * Power[y, 2] - 5 x * y

The restrictions on the Mathematica values and functions used in scalar expressions is that each value or function should be convertible into the target language in which code generation is desired. Each symbolic variable, like x, y, and t in the above expressions, should be associated with a data store in the target code; for example, a local variable, an array element, a class member, a named constant etc. Additional kinds of symbolic expressions with unified OO interfaces can be defined and evaluated like matrices, vectors, boolean expressions, rules, etc.

Using the SymbolicInterface classes I could now define basic operations on multivectors in any selected GA frame. I could define symbolic basis blades and their linear combinations using an approach similar to the one described in "Geometric Algebra for Computer Science". I could create and manipulate symbolic multivectors as linear combinations of basis blades with any given signature, even non-orthogonal ones like the 5D Conformal GA. The coefficients of these symbolic multivectors are symbolic scalars, not just numeric floating point numbers. I began implementing all standard operations on multivectors and blades like the geometric, outer, and inner products, inversions and involutions, outermorphisms, etc.

The SymbolicInterface classes are used extensively across all layers and components of GMac to communicate with Mathematica through a unified OO interface. This interface can be used in similar projects. I hope I could make it an open source component in the near future, if I could obtain proper funding for GMac, to be further developed by interested people. I also need to add automatic symbolic multivector differentiation to increase the range of applications that GMac can be used in.

Being Specific

> It's OK to figure out murder mysteries, but you shouldn't need to figure out code. You should be able to read it. -- Steve McConnell.

The next logical step to make was to design and implement the main user interface to the GMac system, its Domain Specific Language **(DSL)**. One of the lessons I'd learned during my Ph.D. was that using sophisticated software systems requires a well-designed interface close to the user's specific domain of knowledge, not the software engineer's. Two approaches, DSLs and Visual Programming Languages **(VPLs)**, are the best candidates for the job.
A VPL can always be built around a DSL but would require an additional layer to visually interface with the user, So I decided to start with a simple Geometric Algebra based DSL [2. For more information about DSLs and alternative approaches, you can read my post "Computing: Please, Mind Your Language! ".]. I hope in the future I could implement a visual environment for GMac that can be used to define GA-models using visual GA-notation and relate the DSL's components together visually.

It took a lot of experimentation and re-design, but the final result is **GMacDSL**. The language design of GMacDSL is simple and focused on the domain; namely GA-based modeling. GMacDSL is not Turing-complete because it doesn't need to be. The main elements of GMacDSL are:

1. **Namespaces:** These are simple logical named containers for the other elements of GMacDSL similar ti Java packages and C# namespaces.
2. **Frames:** Using GMacDSL, the user can define any fixed set of named basis vectors along with their relative metric relations including Euclidean, Orthonormal, Orthogonal, and Non-orthogonal frames. Frames are the basic abstraction on which all subsequent symbolic computations occur inside GMac.
3. **Subspaces:** A subspace is a set of basis blades of a given frame. They are useful in many cases for selecting or defining parts of multivectors during symbolic computations.
4. **Pre-defined Data Types:** Two types are present in GMacDSL. Scalars are symbolic expressions of real values, including number literals. In addition, each frame automatically defines an associated Multivector type; a list of scalars defining a symbolic linear combination of the Frame's basis blades.
5. **Structures:** User-defined combinations of named members having given types including scalars, multivectors, or other structures. These are similar to user-defined types and structs in other languages. The main purpose of structures is to implement the pattern of Geometric Generators in GMacDSL.
6. **Constants:** These are constant-valued named data containers of any type including scalars, multivectors, and structures. The scalar values inside constants can be large symbolic expressions; they needn't be number literals.
7. **Fixed Outermorphisms:** These are simple outermorphisms with fixed scalar elements that are

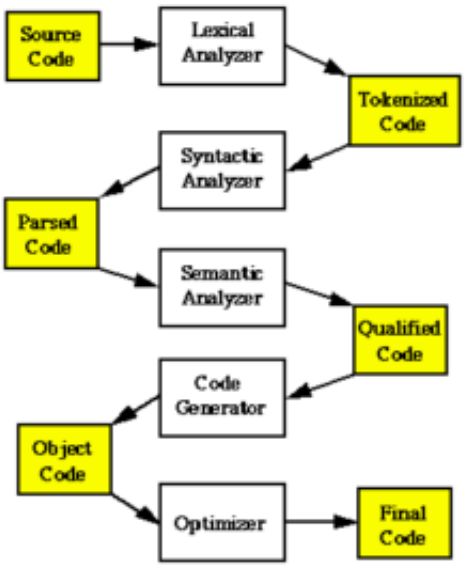
mainly used to transform related frames. For example, we can define a non-orthogonal frame using an orthogonal one and an invertible change of basis matrix. Then GMac automatically defines two fixed outermorphisms to transform multivectors between the two frames.

8. **Multivector Expressions:** The main function of GMac is to handle standard GA operations on symbolic multivectors. Multivector expressions provide this capability to define macros and construct constants.
9. **Macros:** A macro in GMacDSL is the most important unit of computation. Simply speaking, a macro is a list of GA-based high-level computations on scalars, multivectors, and structures that produce a single value of any type from computations performed on a number of inputs, also of any type. A macro is not a procedure or function in the familiar way of common programming languages. A macro can only call other macros defined before it in the DSL code and allows no recursion.

This structure is simple to understand and use for non-programmers working with GA models while being sufficient for covering many practical GA models. In the future, I hope to develop GMacDSL using the Functional Programming Paradigm. This would provide elegant and powerful syntax and semantics for GA-based models that would take great benefit from GA's unified and elegant mathematical abstractions.

Finding Irony

> When debugging, novices insert corrective code; experts remove defective code. -- Richard Pattis.



Construction of a typical compiler (source)

In order to experiment with various designs of GMacDSL I had to implement a compiler to parse and translate GMacDSL code into a suitable intermediate data structure. In 2012, I began writing the **GMacCompiler** front-end for that purpose. The first step in creating a typical compiler is to implement its parser. I initially tried to code my own parser using the elegant functional capabilities of F# but I later

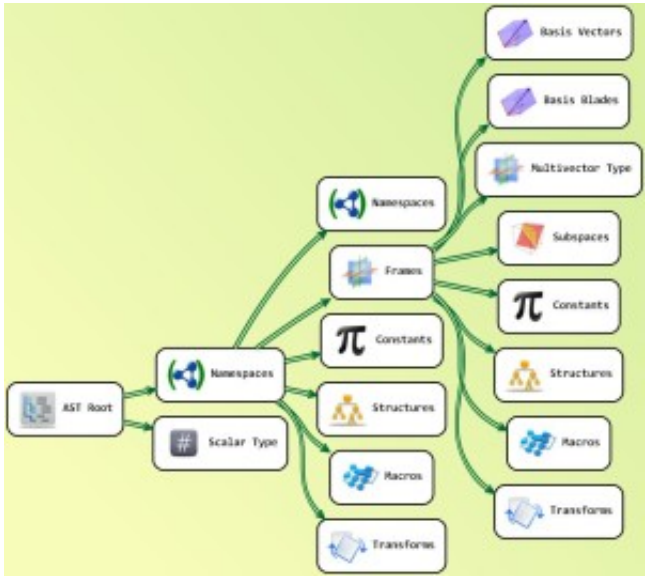
found a much better solution called Irony by Roman Ivantsov [3. You can find good Irony tutorials here and here, in addition to the many samples coming with the Irony source code.]. Irony has many good features that attract computer language designers to use it as a base for their language tools. Using Irony I could easily parse GMacDSL code into a homogeneous **Parse Tree**. This parse tree is not useful by itself because it's based on a context-free Irony Grammar, but it can be converted into a more useful Abstract Syntax Tree **(AST)** through semantic analysis.

Based on Irony, I developed an infrastructure for creating simple DSLs. I call this infrastructure **IronyGrammars** [4. The IronyGrammars class library is not yet documented. I hope I could release it as an open-source project in the future.]; it provides many services needed by typical DSL compilers like source code handling, error and warning messages reporting, symbol table management, semantic analysis, AST construction and navigation, interpretation services, etc. Naturally, the most difficult part of implementing the IronyGrammars infrastructure is the debugging. Many cycles of refactoring, testing, and debugging were made. The implementation finally settled in a relatively satisfying state after removing much code.

I used IronyGrammars to create the GMacCompiler front-end as a specific compiler of GMacDSL code. The GMacCompiler front-end parses and translates the GMacDSL code into the desired intermediate data structure; the GMacAST structure used as the primary GA-based high-level information source for further symbolic processing at later stages. I also created mini-compilers that can translate partial GMacDSL code like a single multivector expression, a single user defined structure, or a single macro. This kind of small compilers is very useful during code composition for adding temporary structure to an already created GMacAST.

Caring for the Environment, Talking to Trees

> Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away. -- Antoine de Saint-Exupery.



GMacAST Main Components

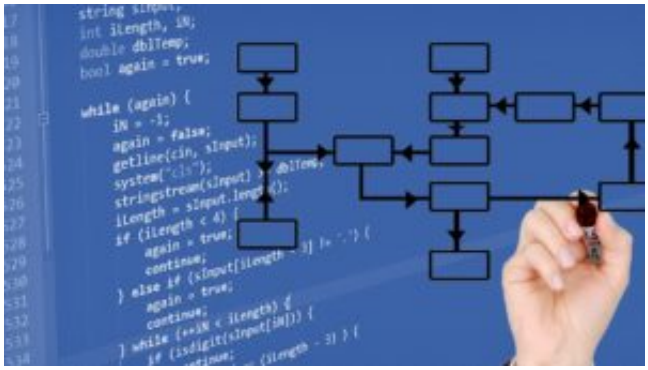
By 2013 I had a solid perception of the main components that GMac should have. I also was almost finished with two of GMac's core components: GMacDSL and the GMacCompiler front-end that translated GMacDSL code into the high-level GMacAST intermediate structure. It was about time to create a proper, but simple, Integrated Development Environment **(IDE)**; the GMacIDE.

The first thing I needed is a simple colored TextBox control for Syntax Highlighting. I soon found the nice FastColoredTextBox control by Pavel Torgashov [5. You can find the FCTB source code here.]. I implemented a simple interface for GMacDSL code management to handle projects containing multiple code files, to compile and report errors and warnings, explore the GMacAST, generate sample code from a single macro, and perform other functions.

The current GMacIDE is very simple, I hope in the future it can be extended to contain a set of Visual Editors for creating and modifying GMacDSL code and manage the details of code composition from GA-models. In addition, I need to add some form of interactive multivector visualization component to enable exploring GA-based geometry visually.

Scripting Your Next Play

> Talk is cheap. Show me the code. -- Linus Torvalds.



The GMacIDE enables the GMac user to perform many functions. One of the most important of these functions is interfacing with the GMac Scripting Engine. Implementing a scripting engine for a system like GMac is essential. GMacDSL is not Turing-complete, the user can describe GA-based models but can't properly interact with them unless he uses more programming structure like loops, conditional execution, expression evaluation, etc. These functions require an interpreter, at least, to be implemented properly. In addition, the powerful symbolic processing capabilities supporting GMac can be accessed using such scripting engine.

GMac scripting is intended for exploration of geometric ideas through GA-based geometric models and algorithms. Three languages can be integrated into a single GMac script, each for a specific purpose:

1. C# code comprises the main body of the script. C# is not originally designed for scripting purposes, nevertheless, C# is a very powerful compiled statically typed object-oriented language that can be used for scripting. To simplify using C# as a scripting interface for GMac some syntactic sugar is used to "sweeten" the process of reading and writing GMac scripts.
2. GMacDSL code can be executed on multivector values and user-defined structures. The scalar

coefficients of multivectors can be numerical, symbolic, or a mix of both. GMac code is passed, as C# public method calls, to GMac internal services in string form to be automatically compiled and executed by the GMacCompiler.

3. Mathematica code can be executed on the Mathematica kernel used in the background by GMac. The Mathematica text code is passed through C# service method calls to the Mathematica kernel through the SymbolicInterface classes. This can be used to exploit the full power of Mathematica through GMac to perform many symbolic manipulations and graphing tasks related to the purpose of the GA-based script exploration.

The main input to the script is always a GMacAST structure compiled from the main GMacDSL code. This combination of GMacAST structure, C#, GMac, and Mathematica code, make GMac scripting a very powerful method for GA-models exploration from the simple to the complex.

Scripting in GMac is not intended for efficient execution of GA-based algorithms. If the user is satisfied by the final algorithm, GMacAPI can be used by a good software designer to implement an efficient, well-structured version of the script in any desired programming language, with specific types of multivectors and GA algorithms.

Composing Text Art

> In science, if you don't do it, somebody else will. Whereas in art, if Beethoven didn't compose the 'Ninth Symphony,' no one else before or after is going to compose the 'Ninth Symphony' that he composed; no one else is going to paint 'Starry Night' by van Gogh. -- Neil deGrasse Tyson.

The final stage in any compiler is code generation from some intermediate representation of the input source code. This stage is the most demanding for creativity because it relies on designing and implementing many optimizations to generate target code suitable for its specific consumer. Traditional compilers typically generate machine code for native hardware or byte code for virtual machine frameworks. This form of code is machine-oriented, cold and repetitive code not intended for reading or understanding by humans. GMac, being a DSL based source-to-source compiler, produces programmer-oriented textual source code in some high-level human-readable language. The difference between designing the code generator of any typical compiler vs. the code generator of a system like GMac is similar to the difference between designing a machine that creates mechanical parts and designing a tool-set for an artist.

If we look up the verb ***compose*** in the Merriam-Webster online dictionary we'll find the following meanings:

1. **A) to form by putting together**: fashion; 'a committee composed of three representatives'
   B) to form the substance of: constitute; 'composed of many ingredients'
   C) to produce (as columns or pages of type) by composition
2. **A) to create by mental or artistic labor**: produce; 'compose a sonnet'
   B) to formulate and write (a piece of music): to compose music for
3. **to deal with or act on so as to reduce to a minimum** 'compose their differences'
4. **to arrange in proper or orderly form**
5. **to free from agitation**: calm, settle 'composed himself'

My personal view of coding is a form of:

> Creative composition of highly structured, human-understandable, and machine-compilable text holding all the meanings of the verb ***compose*** stated above.

To me writing code is not just about execution efficiency or blind implementation of algorithms; writing code is fundamentally artistic such that no two skilled software developers may produce the same code for a single problem. Like there can be no machine that may creatively produce music or paintings, there can be no single code generator that can, by itself, write human-understandable code with all its rich content of information, creativity, and beauty. Nevertheless, we can certainly make many smaller tool sets to help the skilled code developer layout code in the way humanly and creatively desired, while automatically generating machine-oriented code from the intermediate representation to free the ***coding artist*** from its repetitive cold nature.

The text generation tool set specifically created for GMac, but independently usable otherwise, is a C# library called **TextComposerLib**. I made the decision of creating my own text generation component in 2013 after an extensive search for many similar libraries. I found no libraries that satisfied my vision for GMac. The TextComposerLib library is like a set of paint brushes to artists. The user can combine several of its text composition tools to create highly structured text. The structure can be generated on the internal level of a single document, or on the external level of files and folders.

In the future, I hope to make TextComposerLib more versatile and reliable by introducing Multithreading capabilities and designing a Service-Oriented interface. I also hope to release it as an open-source component for general text composition tasks.

Building A Factory

> Make everything as simple as possible, but not simpler. -- Albert Einstein.

In 2014 I began implementing the GMac part I enjoyed the most; the GMacAPI code assembly component. If TextComposerLib is like a set of tools for artists to compose with, then GMacAPI is like a fully configurable production line for assembling structured code. Using this extensive and sophisticated Application Programming Interface **(API)**, you can perform a full process of assembling code libraries in any desired target programming language. Using the GMacAPI you can:

- Supply raw materials to the assembly line; mainly a compiled GMacAST data structure that is independent of any particular target language, like C#, Python, C++, Java, etc., from any particular external computing libraries, and from any particular software architecture required.
- Use the full composition capabilities of TextComposerLib for producing the final code.
- Define sub-processes for assembling subsets of the final code library by inheriting from the GMacAPI core classes.
- Monitor and log each step of the code assembly process for debugging and maintenance.
- Use GMacDSL macros to produce a series of optimized computationally equivalent low-level code in the target language of choice.

I'm very proud of my work in the GMacAPI as it contains many interesting techniques from computer science and software engineering; especially techniques related to code block optimization. The tasks in which GMacAPI can be used are diverse and wide-ranging. I've tried my best to make it as simple as possible for a good software Engineer to use a given GA-model and create a full code library out of it. Because such goal is not easy to reach, using the GMacAPI component requires a good deal of learning and experimentation. I hope in the future to make it more accessible through some form of Visual Interface in which the software engineer can lay out the components, connect, and configure them visually with few lines of code as possible.

Seeking Guidance

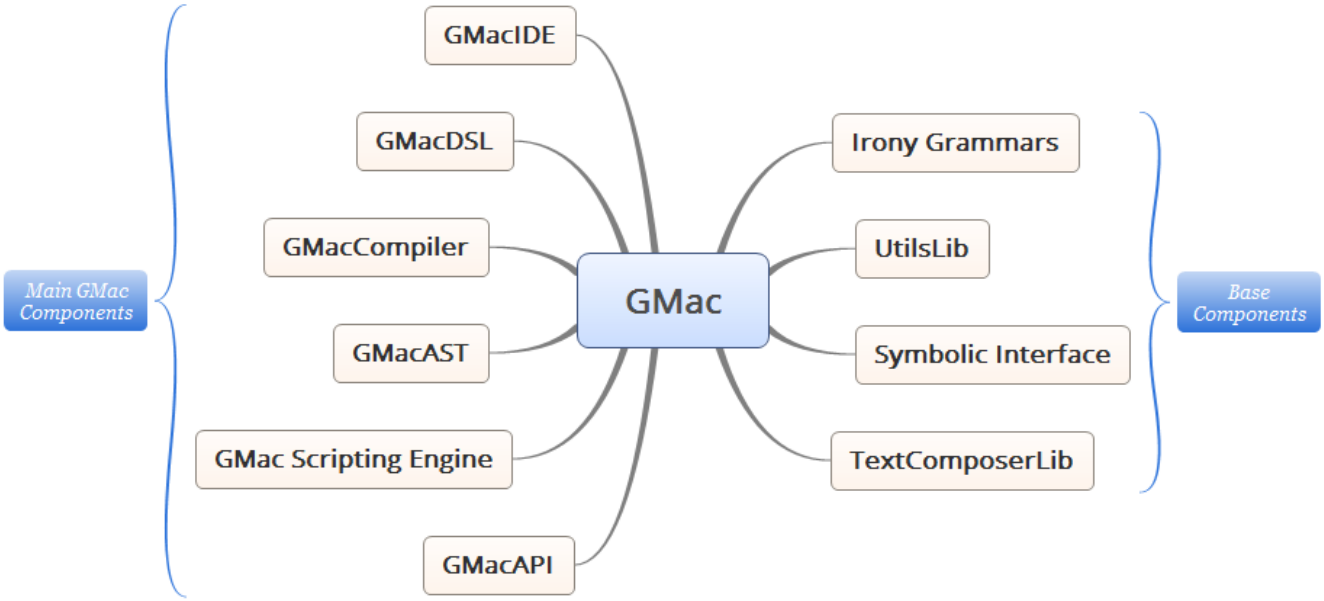
> Conceptual integrity is the most important consideration in system design. -- Fred Brooks, "The Mythical Man-Month"

Software documentation is a hard but necessary part of any software system. GMac is a sophisticated system with lots of interacting components. For a good part of 2015 and 2016, I concentrated on writing the online GMac Guides. The GMac Guides explain in good details the structure of the main components of GMac and how to use them as intended. While writing these guides I discovered some design deficiencies in some parts of GMac that needed fixing or enhancing. This is simply because I couldn't explain them to myself in plain English, or with a direct example when I tried to write the GMac Guides.

Naturally, the full power of GMac can only be illustrated through practical examples. I've created the GMac Samples page on this website for this purpose. In time this page should contain enough examples that I hope would motivate users to explore the computational space using Geometric Algebra and GMac.

The Next Generation

> Space: the final frontier. These are the voyages of the starship Enterprise. Its continuing mission: to explore strange new worlds, to seek out new life and new civilizations, to boldly go where no one has gone before. -- Captain Jean-Luc Picard, Star Trek: The Next Generation



The main components of GMac

I'm writing these words on the 9th of February 2017; it is my 40th birthday. This post is my final thoughts on the 8 years journey creating my own vessel for exploration, GMac. I don't know if I will actually get to use it, as I hope, to explore the computational universe. I still need to add a visualization component and a multivector differentiation component to the mix. I also hope I could modify the design to use other CAS like SymPy or Maple and add a Visual Interface to create GMacDSL code that should be developed to apply functional programming techniques. Many other enhancements and components are possible. Nevertheless, I have learned a lot as I initially expected. I'm now ready to explore applications and algorithms with my own tool. I'm ready to contact other people to exchange ideas and investigate possibilities. I'm ready to begin a new journey as this one ends.

تمت بحمد الله الجمعة ١٣ جماد أول ١٤٣٨هـ

PDF generated by Kalin's PDF Creation Station