Ada Lovelace; the world’s first programmer

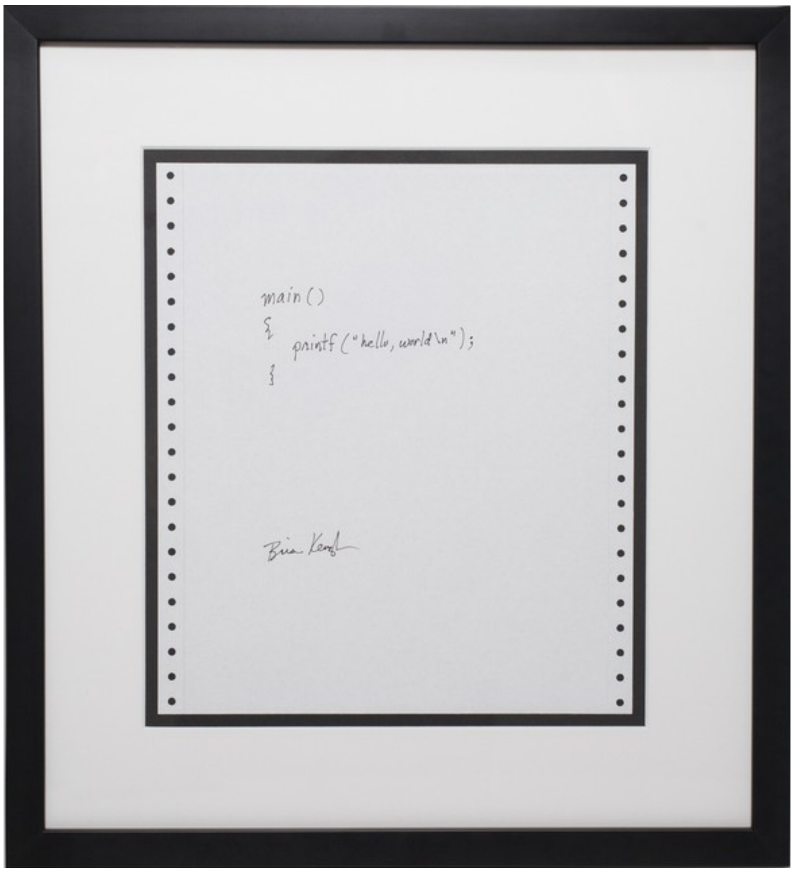
Gavin Conran

The author is assuming the reader has a general knowledge of computing and experience in at least one scripted and one compiled language.

Introduction

Who can capture the imagination of the technically minded more than Ada Lovelace? Daughter of the romantic poet, Lord Byron, and the first to realise the potential of ‘software’ running on a ‘hardware’ platform, in her case Charles Babbage’s Difference Engine. Her insight has fundamentally changed how the world works and the way we live our lives.

To stretch romantic notions of coders inspired by art – ArtNet comes to mind - I could not resist sharing Brian Kernighan’s hand-wriiten “Hello World” image, as shown in Illustration 1, as it will probably bring a smile to the face of anyone who really started their coding journey, like me in 1991, with these few lines of C code.



```
main()
{
    printf("hello, world\n");
}
```

[illegible]

Illustration 1: Brian Kernigahn’s “Hello World”

Even outside the domains of software engineering and computer science, programming is becoming an essential skill for many fields of study and professions, but diverging demands, coupled with the need for specialisation and optimisation, have dictated that domain specific eco-systems have emerged for each of the subject areas. For example, applied mathematicians and engineers have traditionally used Matlab (because of its linear algebra capability), C/C++ and Fortran for their computations; statisticians, business analysts and social & political scientists rely heavily on R (or XPSS or SAS); operational researchers use linear solvers, such as Gurobi, for their work in optimisation; and as for scientists, they are known to use Python for general programming, data analysis and the visualisation of their results. In fact, you can probably find the use of Python in most, if not all, eco-systems.

Software houses are known to use Java and related frameworks, such as JavaFX and Android, to develop business and mobile applications, web developers rely on Javascript and web frameworks, such as Ruby-on-Rails, AngularJS and Django (Python), for their online applications and Game, Computer Graphics and Trading Platform developers make extensive use of C++ due to its reputation for robustness and speed.

With a desire for the reader to make sense of such an array of domains and their supporting languages, the paper starts by taking a quick look at different programming paradigms and languages; bear in mind that each of which could warrant a paper in their own right. This is followed by a short discourse on the more advanced and ever more important features of concurrency and parallel computing.

We then explore the difference between scripted and compiled languages and explain how new programming languages, like Julia, are combining the fast proto-typing benefits of scripting with the performance and efficiency of compiled languages, such as C and Fortran (and their parallel extensions for high performance computing), thus solving the ‘two language’ problem, which is particularly salient in finance, data analysis and many flavours of scientific computing, including deep learning.

The ‘two language’ problem being proto-tying in a scripting language, e.g. Python, but then writing the application in a compiled language, like C++, which is used for production.

As the world is becoming more and more data and cloud centric, the paper ends with a discussion, albeit brief, on structured and unstructured data. For example the ‘unstructured’ big data eco-system utilises platforms, such as Hadoop & Spark, for Big Data Analytics (BDA), while the ‘structured’ data world of Business Intelligence relies on traditional relational databases and SQL. Taday, it usual for the data platforms, structured and ubsrtuctured, to be hosted in the cloud and are accessed by administartors and users remotely over the Internet.

Programming Paradigms

None of the following programming paradigms have a precise, globally unanimous definition, nor official international standard. Nor is there agreement on which paradigm constitues the best method to developing software. The subroutines that implement OOP methods may be ultimately coded in an imperative, functional, or procedural style that may, or may not, directly alter state on behalf of the invoking program. There is some over between paradigms, inevitably, but the main features or identifable differences are summarised in Table 1 below.

Paradigm	Description	Main Trails	Examples
Imperative / Structural / Procedural	Programs are statements that directly ***change computated state*** (datafields) and are based on the ***procedure call***.	- Direct assignments - Common data structures - Global variables - Indentation - Local variables - Sequence - Selection - Iteration - Modularisation	C, C++, Java, Python, Ruby, Pascal, Modula-2, Matlab, Octave, Fortran, Julia, Ada
Functional	Treats computation as the evaluation of ***mathematical functions*** avoiding ***state*** and ***mutable*** data.	- Lambda calculus - Compositionality - Formula - Recursion - Referential transparency - No side effects	Java (>= 8.0), C++, Python, R, Ruby, Scala, Standard ML, Javascript
Event-Driven	Control flow is determined mainly by ***events***, such as mouse clicks or interrupts.	- Main loops - Event handlers - Asynchronous processes	Javascript
Object-Oriented	Treats datafields as ***objects*** manipulated through predefined ***methods*** only.	- Objects - Methods - Message passing - Information hiding - Data abstraction - Encapsulation - Polymorphism - Inheritance - Serialisation-marshalling	C++, Java, Python, Ruby, Scala, Javascript, Ada
Declarative	Defines ***program logic***, but not detailed control flow.	- Fourth generation languages - Spreadsheets - Report program generators	SQL, Regex, OWL, SPARQL
Automata-based programming	Treats programs as a ***model*** of a finite state machine or any other formal ***automata***.	- State enumeration - Control variable - State changes - Isomorphism - State transition table	Abstract State Machine Language

Table 1: Main Programming Paradigms

Concurrency and Parallelism

In general, concurrency is when two or more tasks can start, run, and complete in overlapping time periods. It doesn’t necessarily mean they’ll ever both be running at the same instant, e.g. multi-tasking on a single-core machine. Parallelism, on the other hand, is when tasks literally run at the same time, e.g. on a multi-core processor.

Concurrency

Concurrency is everywhere in modern programming:

- Multiple computers in a network
- Multiple applications running on one computer
- Multiple processors in a computer (multi-core CPU)

In fact, concurrency is essential in modern programming:

- Web sites must handle multiple simultaneous users
- Mobile apps need to do some of the their procesing on servers in the cloud
- Graphical user interfaces almost always require background work that does not interrupt the user, e.g. IDEs compile code while you are still editing it.

There are four basic types of Concurrency found in Java:

1. **Containment**

- Example: Graphical User Interfaces, e.g. JavaFX and Android
- Most GUI applications have a single thread, called the *event dispatch thread*, to handle interactions with the application's GUI components
- Here multi-threading is achieved by using *thread containment*
- There are three important ***Design Patterns*** used in GUI development:
    - ***View Tree***: central feature in the architecture of most GUI toolkits
    - ***Model-View-Controller***: Separates data, output, and input
    - ***Listener***: Decouples the model from the View and Controller

2. **Immutability**

- Ideal for concurrency because it is *not possible to alter the state (Data Fields)* and recursion is a special case of *Re-entrant Code*
- Example: Immutable Lists
- Recursive Data Types in Java 8 are similar to the same feature in Scala
- Two important ***Design Patterns*** used in constructing Recursive Data Types:

    - ***Interpreter Pattern***: pattern of implementing an operation over a recursive data type
    - ***Sentinal Objects Pattern***: to signal the base case or end-point of a data structure

3. **Thread Safe Data Types**

- Example: Library Data Structures use ***Shared Memory Model*** in form of ***Monitor Pattern***
- List, Set, & HashMap are Sequential Data Types but they all have Concurrent versions

4. S**ynchronisation** (syntactic sugar for Locks and Conditions)

- An example of Synchronisation is the ***Message Passing Model*** in the form of the ***Producer / Consumer Pattern***
- Example of the ***Message Passing Model*** is the ***Client / Server Pattern*** found in Networking
- Programmers can apply a ***Monitor*** directly to a data Type by using the keyword Synchronize or by "going under the hood" and using the *Lock and Condition* Interfaces
- Example: Mutable Data Structures use *Coarse-grain locking*
- Example: Operating Systems use *Fine-grain locks* for high-performance and *Lock Ordering* to deal with Deadlocks

Parallelism

Supercomputing and Parallel Computing play an important role in the field of scientific computing and is used for a wide range of computationally intensive tasks in various fields, including quantum mechanics, weather forecasting, climate research, oil and gas exploration , molecular modeling, and physical simulations, such as simulations of the early moments of the universe, airplane and spacecraft aerodynamics, the detonation of nuclear weapons, and nuclear fusion. Throughout their history, they have been essential in the field of cryptography.

Parallel Computing relies on two libraries for parallelising programs:

- OpenMP (Open Multi-Processing)
    - Specification for writing ***shared memory*** multi-process apps, e.g. multi-core CPU
- Message Passing Interface (MPI)
    - Uses ***message passing*** paradigms with no shared memory, e.g. distributed machines

An application built with a hybrid model of parallel programming can run on a computer cluster using both OpenMP and MPI, such that OpenMP is used for parallelism within (multi-core) node while MPI is used for parallelism between nodes.

- CUDA
  A parallel computing platform and programming model developed by Nvidia for general purpose computing on its own GPUs (Graphics Processing Units). CUDA enabled developers to speed up compute-intensive applications by harnesing the power of GPUs for the parallelisable part of the computation, e.g. deep learning – more on this later.

Programming Languages

Scripted Languages

Scripted languages refer to dynamic high-level general purpose languages, such as Python, Javascript, Matlab, Ruby & R, with the term 'script' often used for small programs and executed by ***interpretors*** (step-by-step executors of source code, where no pre-runtime translation takes place). Shell scripts of operating systems are also refered to as 'scripts'.

The main advantages of scripting languages are that they tend to be easy to learn and with ***dynamic typing*** it makes it possible to do quick proto-typing of new applications and solutions. The down side is that they tend to be slow.

Compiled Languages

A programming language whose implementations are typically ***compilers*** (translators that generate machine code from source code), and not interpretors. Example include C and Fortran. This explains why it is necessary to compile a program on each target operating system, i.e. write once and compile everywhere.

The main advantage of compiled languages is that they are fast, explaining why most professional scientific computing is programmed using languages like C and Fortran. The downside is that they are harder to learn and master than scripting languages and with ***static typing*** programs tend to be harder to program and debug.

A ***combination of both solutions*** is also common: a compiler can translate the source code into some intermediate form, ***bytecode***, which is then passed to an interpretor which executes it, e.g. *Java is compiled to bytecode which is then interpreted by the Java Runtime Environment (JRE)*. It is this procedure that allows Java applications to run on multiple operating systems (assuming a JRE is installed), i.e. compile once and run everywhere. It also helps explain why C and Fortran out perform Java for scientific computing and why Java is preferred for desk-top business apps running on multiple Operating Systems.

Beating the Trade-off

Rather than having to trade off productivity (quick programming) with efficiency (fast program execution), it is desirable to have both, i.e. the ease of Python with the performance of C. Such a language is Julia whose creators claim:

"Julia is a high-level, high-performance dynamic programming language for technical computing. Julia features optional typing, multiple dispatch, and good performance, achieved using type inference and just-in-time (JIT) compilation, implemented using LLVM"

- ***Optional Typing:***
  Julia's type system is dynamic, but gains some of the advantages of static type systems by making it possible to indicate that certain values are of a specific type. This can be of great assistance in generating efficient code, but even more significantly, it allows method dispatch on the types of function arguments to be deeply integrated with the language.

- ***Multiple Dispatch:***
  A function or method can be dynamically dispatched on the run-time (dynamic) type; a form of polymorphism. Multiple dispatch routes the dynamic dispatch to the implementing function or method using the combined characteristics of one or more arguments.
- ***Type Inference:***
  Refers to the process of deducing the types of later values from the types of input values
- ***Just-In-Time (JIT) Compilation:***
  Involves compilation during execution of a program – at run time – rather than prior to execution. Most often, this consists of source code or more commonly bytecode translation to machine code, which is then executed directly.
- ***LLVM:***
  A compiler infrastructure project which is a collection of modular and reusable compiler and toolchain technologies used to develop compiler front ends (to internal representation) and back ends (to CPU architecture).
- ***Good Performance:***
  As shown in Illustration 2, the micro-benchmarks indicates that Julia approaches C and Fortran like performance. As expected it beats Java and all the scripted languages.

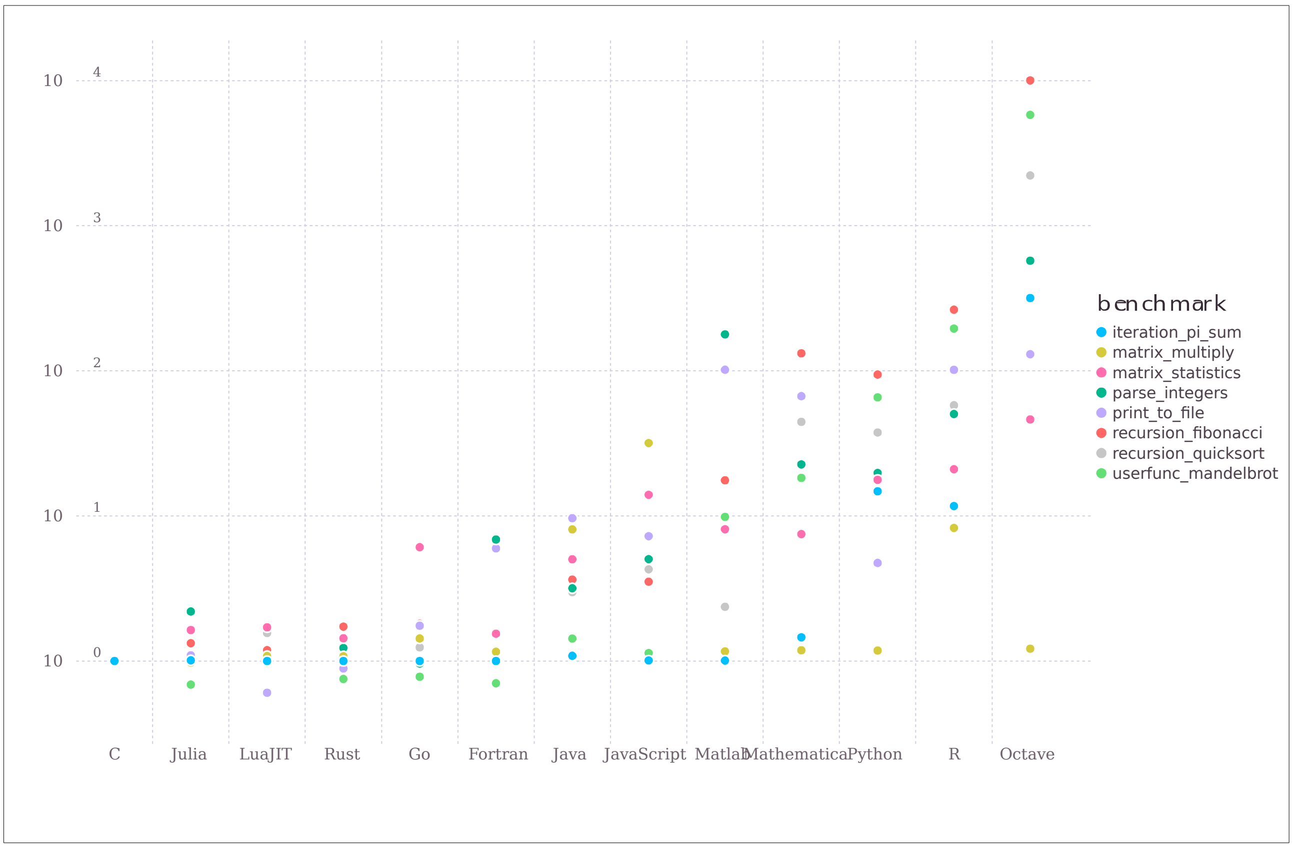


Illustration 2: Good performance of Julia

- ***High Performance Computing***
  Julia provides built-in primitives for parallel computing at every level: instruction level parallelism, multi-threading & distributed computing.

Data

This section outlines the difference between structured and unstructured data coupled with an overview of cloud computing, an online resource, used, more times than not, to host data platforms.

Structured Data: Relational Databases

A relational database is a digital database based on the relational model of data, as proposed by E. F. Codd. A software system used to maintain relational databases is a relational database management system (RDBMS). Virtually all relational database systems use SQL (Structured Query Language) for querying and maintaining the database.

This model organises data into one or more tables (or "relations") of columns or rows, with a unique key identifying each row. Rows are also called records or tuples. Columns are called attributes.

Unstructured Data: Big Data

Big Data is where parallel computing tools are needed to handle data which represents a distinct and clearly defined change in the computer science used, via parallel programming theories, and losses of some of the guarantees and capabilities made by Codd's relational model.

Further more, Big Data usually includes data sets with sizes beyond the ability of commonly used software tools to capture, curate, manage, and process data within a tolerable elapsed time. Big Data philosophy encompasses unstructured, semi-structured and structured data, however the main focus is unstructured data. Big Data requires a set of techniques and technologies with new forms of integration to reveal insights from datasets that are diverse, complex, and of a massive scale. Of interest is that much effort has been put into creating front-end SQL like querying capabilities for Big Data.

Over time a clear delineation between Big Data and Business Intelligence has emerged:

- ***Business Intelligence*** uses ***descriptive statistics***, as found in the Gauss paper, with data with high information density to measure things, detect trends, etc.
- ***Big Data*** uses ***inductive statistics, or machine learning algorithms***, as in the Descartes and Euler papers, to infer laws (regressions and clustering) from large data sets with low information density to reveal relationships and dependencies, or to perform predictions of outcomes and behaviours.

Cloud Computing

Cloud computing is the on demand availability of computer system resources, especially data storage and computing power, without direct active management by the user. The term is generally used to describe data centres available to many users over the Internet. Large clouds, predominant today, often have functions distributed over multiple locations from central servers. If the connection to the user is relatively close, it may be designated an edge server. Cloud computing relies on the sharing of resources to achieve coherence and economies of scale.

References

Papers / Books / Courses

Although not directly referenced in the paper, the following texts were used extensively:

Graduate:

Oppenheim, A. and Schafer, W. (2013). Discrete-Time Signal Processing

Kutz, N. (2013). Data-Driven Modeling & Scientific Computation: Complex Systems & Big Data

Bishop, M. (2006). Pattern Recognition and Machine Learning

Hastie, T., Tibshirani, R. and Friedman, J. (2008). The Elements of Statistical Learning

LeVeque, R. (2013). High Performance Scientific Computing

Barba, L. A. (2014). Practical Numerical Methods with Python

LeCun, Y., Bottou, L., Bengio, Y. and Haffner. (1998). Gradient-Based Learning Applied to Document Recognition

Turek, D. (2004) Design of Efficient Digital Interpolation Filters for Integer Upsampling

Jha, R.G. (2012). On the Numerical Simulations of Feynman's Path Integrals using Markov Chain Monte-Carlo with Metropolis-Hastings Algorithms

Undergraduate:

Calculus:

Fowler, J. and Snapp, B. (2014). MOOCulus

Bonfert-Taylor, P. (2015). Complex Analysis

Blanchard, P. and Devaney, R. (2011). Differential Equations

Linear Algebra:

Klein, P. (2013). Coding the Matrix: Linear Algebra through Applications to Computer Science

Strang, G. (2006). Linear Algebra and Its Applications, 4th Addition

Probability & Statistics:

Conway, A.(2012). Statistics One

Johns Hopkins Specialisation (2014). Data Science

Physics:

Krauth, W. (2006). Statistical Mechanics: Algorithms and Computation

Aspuru-Guzik, A. (2017). The Quantum World

Economics & Financial Mathematics:

Zivot, E. (2016). Computational Finance and Financial Econometrics

Cvitanic, J. and Zapatero, F. (2004). Intro to the Economics and Mathematics of Financial Markets

Rangel, A. (2016). Principles of Economics with Calculus

Electronics:

Prandoni, P. and Vetterli, M. (2008). Signal Processing for Communications

Agarwa, A. and Lang, J. (2005). Foundations and Digital Electronic Circuits

Computation and Algorithms:

Guttag, J. (2013). Introduction to Computation and Programming Using Python, 1st Edition

Sedgewick, R. and Wayne, K. (2011). Algorithms, 4th Edition

Block, J. (2008). Effective Java, 3rd Edition

Deitel, P. and Deitel, H. (2010). Java: How to Program, 8th Edition

Richie, D. and Kernighan, B. (1988). The C Programming Language, 2nd Edition

Boyd, S. and Vandenberge, L. (2018). Intro to Applied Linear Algebra. Julia Language Campanion

Data Science & Cloud Computing

Codd, E.F. (1970). A Relational Model of Data for Large Shared Data Banks

JPL-Caltech (2014). Big Data Analytics Virtual Summer School

University of Maryland (2018). Cloud Computing Micro-Masters

Berkeley University (2015). Data Science and Engineering with Spark XSeries Program

Additional:

Wikipedia (multiple links)

Linear Algebra (scipy.linalg) https://docs.scipy.org/doc/numpy/reference/routines.linalg.html

Signal Processing (scipy.signal) https://docs.scipy.org/doc/scipy/reference/signal.html

Machine Learning: https://scikit-learn.org/stable/auto_examples/decomposition/plot_pca_iris.html

Data Analysis: https://pandas.pydata.org/

Graphs / Networks: https://networkx.github.io/

Simulated Annealing: https://www.fourmilab.ch/documents/travelling/anneal/

Codes

Github codes

As I have written the majority of code for these papers in Python I have only included links to some of my other language repositories in GitHub:

Java:
https://github.com/gavinconran/Concurrency

C/C++:
https://github.com/gavinconran/CPlusPlus

Fortran (with MPI):
https://github.com/gavinconran/numerical-finance/tree/master/07_HighPerformanceComputing

Javascript:
Code: https://github.com/gavinconran/CourseraWebDev/tree/gh-pages/module5-solution
Web Site: https://gavinconran.github.io/CourseraWebDev/module5-solution/

R:
https://github.com/gavinconran/ESSEC

Julia Codes

01_Base_dual.jl
02_Gadfly_Rdatasets_iris.jl
03_Random_Plots_SDE.jl
04_JuMP_basic.jl