Computer – Máy Tính

Nguyễn Quản Bá Hồng*

Ngày 7 tháng 11 năm 2024

Tóm tắt nội dung

This text is a part of the series *Some Topics in Advanced STEM & Beyond*:
URL: https://nqbh.github.io/advanced_STEM/.
Latest version:

- *Computer – Máy Tính.*
  PDF: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/computer/NQBH_computer.pdf.
  TEX: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/computer/NQBH_computer.tex.

Mục lục

*A Scientist & Creative Artist Wannabe. E-mail: nguyenquanbahong@gmail.com. Bến Tre City, Việt Nam.

1 Wikipedia

1.1 Wikipedia/abstraction (computer science)

"In software engineering & computer science, *abstraction* is the process of generalizing concrete details, e.g. attributes, away from the study of objects & systems to focus attention on details of greater importance. Abstraction is a fundamental concept in computer science & software engineering, especially within the object-oriented programming paradigm. E.g.:

- the usage of abstract data types to separate usage from working representations of data within programs.
- the concept of functions or subroutines which represent a specific way of implementing control flow;
- the process of reorganizing common behavior from groups of non-abstract classes into abstract classes using inheritance & subclasses, as seen in object-oriented programming languages.

1.1.1 Rationale

> "The essence of abstraction is preserving information that is relevant in a given context, & forgetting information that is irrelevant in that context." – John V. Guttag

Computing mostly operates independently of the concrete world. The hardware implements a model of computation that is interchangeable with others. The software is structured in architectures to enable humans to create the enormous systems by concentrating on a few issues at a time. These architectures are made of specific choices of abstractions. Greenspun's 10th rule is an aphorism on how such an architecture is both inevitable & complex.

Language abstraction is a central form of abstraction in computing: new artificial languages are developed to express specific aspects of a system. *Modeling languages* help in planning. *Computer language* from the machine language to the assembly language & the high-level language. Each stage can be used as a stepping stone for the next stage. The language abstraction continues e.g. in scritping languages & domain-specific programming languages.

Within a programming language, some features let the programmer create new abstractions. These include subroutines, modules, polymorphism, & software components. Some other abstractions such as software design pattens & architectural styles remain invisible to a translator & operate only in the design of a system.

Some abstractions try to limit the range of concepts a programmer needs to be aware of, by completely hiding the abstractions they are built on. The software engineer & writer Joel Spolsky has criticized these efforts by claiming that all abstractions are *leaky* – that they can never completely hide the details below; however, this does not negate the usefulness of abstraction.

Some abstractions are designed to inter-operate with other abstractions – e.g., a programming language may contain a foreign function interface for making calls to the lower-level language.

1.1.2 Abstraction features

Programming languages. Main article: Wikipedia/programming language. Different programming languages provide different types of abstraction, depending on the intended applications for the language. E.g.:

- In OOP languages e.g. C++, Object Pascal, or Java, the concept of *abstraction* has itself become a declarative statement – using the syntax `function(parameters) = 0;` (in C++) or the keywords `abstract` & `interface` (in Java). After such a declaration, it is the responsibility of the programmer to implement a class to instantiate the object of the declaration.
- Functional programming languages commonly exhibit abstractions related to functions, e.g. lambda abstractions (making a term into a function of some variable) & higher-order functions (parameters are functions).
- Modern members of the Lisp programming language family e.g. Clojure, Scheme, & Common Lisp support macro systems to allow syntactic abstraction. Other programming languages such as Scala also have macros, or very similar metaprogramming features (e.g., Haskell has Template Haskell, & OCaml has MetaOCaml). These can allow a programmer to eliminate boilerplate code, abstract away tedious function call sequences, implement new control flow structures, & implement Domain Specific Languages (DSLs), which allow domain-specific concepts to be expressed in concise & elegant ways. All of these, when used correctly, improve both the programmer's efficiency & the clarity of the code by making the intended purpose more explicit. A consequence of syntactic abstraction is also that any Lisp dialect & in fact almost any programming language can, in principle, be implemented in any modern Lisp with significantly reduced (but still nontrivial in most cases) effort when compared to "more traditional" programming languages such as Python, C, or Java.

Specification methods. Main article: Wikipedia/formal specification. Analysts have developed various methods to formally specify software systems. Some known methods include:

- Abstract-model based method (VDM, Z);
- Algebraic techniques (Larch, CLEAR, OBJ, ACT ONE, CASL);
- Process-based techniques (LOTOS, SDL, Estelle);
- Trace-based techniques (SPECIAL, TAM);
- Knowledge-based techniques (Refine, Gist).

Specification languages Main article: Wikipedia/specification language. Specification languages generally rely on abstractions of 1 kind or another, since specifications are typically defined earlier in a project, (& at a more abstract level) than an eventual implementation. The UML specification language, e.g., allows the definition of *abstract* classes, which in a waterfall project, remain abstract during the architecture & specification phrase of the project.

1.1.3 Control abstraction

Main article: Wikipedia/control flow. Programming languages offer control abstraction as 1 of the main purposes of their use. Computer machines understand operations at the very low level such as moving some bits from 1 location of the memory to another location & producing the sum of 2 sequences of bits. Programming languages allow this to be done in the higher level. E.g., consider this statement written in a Pascal-like fashion: `a := (1 + 2) * 5`. To a human, this seems a fairly simple & obvious calculation. However, the lower-level steps necessary to carry out this evaluation, & return the value 15, & then assign that value to the variable `a`, are actually quite subtle & complex. The values need to be converted to binary representation (often a much more complicated task than one would think) & the calculations decomposed (by the compiler or interpreter) into assembly instructions (again, which are much less intuitive to the programmer: operations such as shifting a binary register left, or adding the binary complement of the contents of 1 register to another, are simply not how humans think about the abstract arithmetical operations of addition or multiplication). Finally, assigning the resulting value of 15 to the variable labeled `a`, so that `a` can be used later, involves additional 'behind-the-scenes' steps of looking up a variable's label & the resultant location in physical or virtual memory, storing the binary representation of 15 to that memory location, etc.

Without control abstraction, a programmer would need to specify *all* the register/binary-level steps each time they simply wanted to add or multiply a couple of numbers & assign the result to a variable. Such duplication of effort has 2 serious negative consequences:

1. it forces the programmer to constantly repeat fairly common tasks every time a similar operation is needed.
2. it forces the programmer to program for the particular hardware & instruction set.

Structured programming. Main article: Wikipedia/structured programming. Structured programming involves the splitting of complex program tasks into smaller pieces with clear flow-control & interfaces between components, with a reduction of the complexity potential for side-effects.

In a simple program, this may aim to ensure that loops have single or obvious exit points & (where possible) to have single exit points from functions & procedures.

In a larger system, it may involve breaking down complex tasks into many different modules. Consider a system which handles payroll on ships & at shore offices:

- The uppermost level may feature a menu of typical end-user operations.
- Within that could be standalone executables or libraries for tasks such as signing on & off employees or printing checks.
- Within each of those standalone components there could be many different source files, each containing the program code to handle a part of the problem, with only selected interfaces available to other parts of the program. A sign on program could have source files for each data entry screen & the database interface (which may itself be a standalone 3rd party library or a statically linked set of library routines).
- Either the database or the payroll application also has to initiate the process of exchanging data with between ship & shore, & that data transfer task will often contain many other components.

These layers produce the effect of isolating the implementation details of 1 component & its assorted internal methods from the others. Object-oriented programming embraces & extends this concept.

1.1.4 Data abstraction

Main article: Wikipedia/abstract data type. Data abstraction enforces a clear separation between the *abstract* properties of a data type & the *concrete* details of its implementation. The abstract properties are those that are visible to client code that makes use of the data type – the *interface* to the data type – while the concrete implementation is kept entirely private, & indeed can change, e.g. to incorporate efficiency improvements over time. The idea is that such chances are not supposed to have any impact on client code, since they involve no difference in the abstract behavior.

E.g., one could define an abstract data type called *lookup table* which uniquely associates *keys* with values, & in which values may be retrieved by specifying their corresponding keys. Such a lookup table may be implemented in various ways: as a hash table, a binary search tree, or even a simple linear list of `(key:value)` pairs. As far as client code is concerned, the abstract properties of the type are the same in each case.

Of course, this all relies on getting the details of the interface right in the 1st place, since any changes there can have major impacts on client code. As 1 way to look at this: the interface forms a *contract* on agreed behavior between the data type & client code; anything not spelled out in the contract is subject to change without notice.

1.1.5 Manual data abstraction

While much of data abstraction occurs through computer science & automation, there are times when this process is done manually & without programming intervention. 1 way this can be understood is through data abstraction within the process of conducting a systematic review of the literature. In this methodology, data is abstracted by 1 or several abstractors when conducting a meta-analysis, with errors reduced through dual data abstraction followed by independent checking, known as adjudication.

1.1.6 Abstraction in OOP

Main article: Wikipedia/object (computer science). In OOP theory, *abstraction* involves the facility to define objects that represent abstract "actors" that can perform work, report on & change their state, & "communicate" with other objects in the system. The term encapsulation refers to the hiding of state details, but extending the concept of *data type* from earlier programming languages to associate *behavior* most strongly with the data, & standardizing the way that different data types interact, is the beginning of *abstraction*. When abstraction proceeds into the operations defined, enabling objects of different types to be substituted, it is called polymorphism. When it proceeds in the opposite direction, inside the types or classes, structuring them to simplify a complex set of relationships, it is called delegation or inheritance.

Various OOP languages offer similar facilities for abstraction, all to support a general strategy of polymorphism in object-oriented programming, which includes the substitution of 1 type for another in the same or similar role. Although not as generally supported, a configuration or image or package may predetermine a great many of these bindings at compile-time, link-time, or loadtime. This would leave only a minimum of such bindings to change at run-time.

Commmon Lisp Object System or Self, e.g., feature less of a class-instance distinction & more use of delegation for polymorphism. Individual objects & functions are abstracted more flexibly to better fit with a shared functional heritage from Lisp.

C++ exemplifies another extreme: it relies heavily on templates & overloading & other static bindings at compile-time, which in turn has certain flexibility problems.

Although these examples offer alternate strategies for achieving the same abstraction, they do not fundamentally alter th need to support abstract nouns in code – all programming relies on an ability to abstract verbs as functions, nouns as data structures, & either as processes.

Consider e.g. a sample Java fragment to represent some common farm "animals" to a level of abstraction suitable to model simple aspects of their hunger & feeding. It defines an `Animal` class to represent both the state of the animal & its functions:

```
public class Animal extends LivingThing
{
    private Location loc;
    private double energyReserves;

    public boolean isHungry() {
        return energyReserves < 2.5;
    }

    public void eat(Food food) {
        // Consume food
        energyReserves += food.getCalories();
    }

    public void moveTo(Location location) {
        // Move to new location
        this.loc = location;
    }
```

```
}
```

With the above definition, one could create objects of type `Animal` & call their methods like this:

```
thePig = new Animal();
theCow = new Animal();
if (thePig.isHungry()) {
    thePig.eat(tableScraps);
}
if (theCow.isHungry()) {
    theCow.eat(grass);
}
theCow.moveTo(theBarn);
```

In the above example, the class `Animal` is an abstraction used in place of an actual animal, `LivingThing` is a further abstraction (in this case a generalization) of `Animal`.

If one requires a more differentiated hierarchy of animals – to differentiate, say, those who provide milk from those ho provide nothing except meat at the end of their lives – that is an intermediary level of abstraction, probably `DairyAnimal(cows,goats)` who would eat foods suitable to giving good milk, & `MeatAnimal(pigs,steers)` who would eat foods to give the best meat-quality.

Such an abstraction could remove the need for the application coder to specify the type of food, so they could concentrate instead on the feeding schedule. The 2 classes could be related using inheritance or stand alone, & the programmer could define varying degrees of polymorphism between the 2 types. These facilities tend to vary drastically between languages, but in general each can achieve anything that is possible with any of the others. A great many operation overloads, data type by data type, can have the same effect at compile-time as any degree of inheritance or other means to achieve polymorphism. The class notation is simply a coder's convenience.

Object-oriented design. Main article: Wikipedia/object-oriented design. Decisions regarding what to abstract & what to keep under the control of the coder become the major concern of object-oriented design & domain analysis – actually determining the relevant relationships in the real world is the concern of object-oriented analysis or legacy analysis.

In general, to determine appropriate abstraction, one must make many small decisions about scope (domain analysis), determine what other systems one must cooperate with (legacy analysis), then perform a detailed object-oriented analysis which is expressed within project time & budget constraints as an object-oriented design. In our simple example, the domain is the barnyard, the live pigs & cows & their eating habits are the legacy constraints, the detailed analysis is that coders must have the flexibility to feed the animals what is available & thus there is no reason to code the type of food into the class itself, & the design is a single simple `Animal` class of which pigs & cows are instances with the same functions. A decision to differentiate `DairyAnimal` would change the detailed analysis but the domain & legacy analysis would be unchanged – thus it is entirely under the control of the programmer, & it is called an abstraction in object-oriented programming as distinct from abstraction in domain or legacy analysis.

1.1.7 Considerations

When discussing formal semantics of programming languages, formal methods or abstract interpretation, *abstraction* refers to the act of considering a less detailed, but safe, definition of the observed program behaviors. E.g., one may observe only the final result of program executions instead of considering all the intermediate steps of executions. Abstraction is defined to a *concrete* (more precise) model of execution.

Abstraction may be *exact* or *faithful* w.r.t. a property if one can answer a question about the property equally well on the concrete or abstract model. E.g., if one wishes to know what the result of the evaluation of a mathematical expression involving only integers $+, -, \cdot$, is worth module n, then one needs only perform all operations module n (a familiar form of this abstraction is casting out nines).

Abstractions, however, though not necessarily *exact*, should be *sound*. I.e., it should be possible to get sound answers from them – even though the abstraction may simply yield a result of undecidability. E.g., students in a class may be abstracted by their minimal & maximal ages; if one asks whenever a certain person belongs to that class, one may simply compare that person's age with the minimal & maximal ages; if his age lies outside the range, one may safely answer that the person does not belong to the class; if it does not, one may only answer "I don't know".

The level of abstraction included in a programming language can influence its overall usability. The Cognitive dimensions framework includes the concept of *abstraction gradient* in a formalism. This framework allows the designer of a programming language to study the trade-offs between abstraction & other characteristics of the design, & how changes in abstraction influence the language usability.

Abstractions can prove useful when dealing with computer programs, because nontrivial properties of computer programs are essentially undecidable (Rice's theorem). As a consequence, automatic methods for deriving information on the behavior of computer programs either have to drop termination (on some occasions, they may fail, crash or never yield out a result), soundness (they may provide false information), or precision (they may answer "I don't know" to some questions).

Abstraction is the core concept of abstract interpretation. Model checking generally takes place on abstract versions of the studied systems.

1.1.8 Levels of abstraction

Main article: Wikipedia/abstraction layer. Computer science commonly presents *levels* (or, less commonly, *layers*) of abstraction, wherein each level represents a different model of the same information & processes, but with varying amounts of detail. Each level uses a system of expression involving a unique set of objects & compositions that apply only to a particular domain. Each relatively abstract, "higher" level builds on a relatively concrete, "lower" level, which tends to provide an increasingly "granular" representation. E.g., gates build on electronic circuits, binary on gates, machine language on binary, programming language on machine language, applications & operating systems on programming languages. Each level is embodied, but not determined, by the level beneath it, making it a language of description that is somewhat self-contained.

Database systems. Main article: Wikipedia/database management system. Data abstraction levels of a database system. Since many users of database systems lack in-depth familiarity with computer data-structures, database developers often hide complexity through the following levels"

- **Physical level.** The lowest level of abstraction describes *how* a system actually stores data. The physical level describes complex low-level data structures in detail.
- **Logical level.** The next higher level of abstraction describes *what* data the database stores, & what relationships exist among those data. The logical level thus describes an entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical level structures, the user of the logical level does not need to be aware of this complexity. This is referred to as physical data independence. Database administrators, who must decide what information to keep in a database, use the logical level of abstraction.
- **View level.** The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of a database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide may views for the same database.

Layered architecture. Main article: Abstraction layer. The ability to provide a design of different levels of abstraction can

- simplify the design considerably
- enable different role players to effectively work at various levels of abstraction
- support the portability of software artifacts (model-based ideally)

System designs & business process design can both use this. Some design processes specifically generate designs that contain various levels of abstraction.

Layered architecture partitions the concerns of the application into stacked groups (layers). It is a technique used in designing computer software, hardware, & communications in which system or network components are isolated in layers so that changes can be made in 1 layer without affecting the others." – Wikipedia/abstraction (computer science)

1.2 Wikipedia/data structure

A data structure known as a hash table. "In computer science, a *data structure* is a data organization & storage format that is usually chosen for efficient access to data. More precisely, a data structure is a collection of data values, the relationships among them, & the functions or operations that can be applied to the data, i.e., it is an algebraic structure about data.

1.2.1 Usage

Data structures serve as the basis for abstract data types (ADT). The ADT defines the logical form of the data type. The data structure implements the physical form of the data type.

Different types of data structures are suited to different kinds of applications, & some are highly specialized to specific tasks. E.g., relational databases commonly use B-tree indexes for data retrieval, while compiler implementations usually use hash tables to look up identifiers.

Data structures provide a means to manage large amounts of data efficiently for uses such as large databases & internet indexing services. Usually, efficient data structures are key to designing efficient algorithms. Some formal design methods & programming language emphasize data structures, rather than algorithms, as the key organizing factor in software design. Data structures can be used to organize the storage & retrieval (thu hồi) of information stored in both main memory & secondary memory.

1.2.2 Implementation

Data structures can be implemented using a variety of programming languages & techniques, but they all share the common goal of efficiently organizing & storing data. Data structures are generally based on the ability of a computer to fetch & store data at any place in its memory, specified by a pointer – a bit string, representing a memory address, that can be itself stored in memory & manipulated by the program. Thus, the array & record data structures are based on computing the addresses of

data items with arithmetic operations, while the linked data structures are based on storing addresses of data items within the structure itself. This approach to data structuring has profound implications for the efficiency & scalability of algorithms. E.g., the contiguous memory allocation in arrays facilitates rapid access & modification operations, leading to optimized performance in sequential data processing scenarios.

The implementation of a data structure usually requires writing a set of procedures that create & manipulate instances of that structure. The efficiency of a data structure cannot ne analyzed separately from those operations. This observation motivates the theoretical concept of an abstract data type, a data structure that is defined indirectly by the operations that may be performed on it, & the mathematical properties of those operations (including their space & time cost).

1.2.3 Examples

The standard type hierarchy of the programming language Python 3. Main article: Wikipedia/list of data structures. There are numerous types of data structures, generally built upon simpler primitive data types. Well known examples are:

- An array is a number of elements in a specific order, typically all of the same type (depending on the language, individual elements may either all be forced to be the same type, or may be of almost any type). Elements are accessed using an integer index to specify which element is required. Typical implementations allocate contiguous memory words for the elements of arrays (but this is not always a necessity). Arrays may be fixed-length or resizable.

- A linked list (also just called *list*) is a linear collection of data elements of any type, called *nodes*, where each node has itself a value, & points to the next node in the linked list. The principal advantage of a linked list over an array is that values can always be efficiently inserted & removed without relocating the rest of the list. Certain other operations, e.g. random access to a certain element, are however slower on lists than on arrays.

- A record (also called *tuple* or *struct*) is an aggregate data structure. A record is a value that contains other values, typically in fixed number & sequence & typically indexed by names. The elements of records are usually called *fields* or *members*. In the context of OOP, records are known as plain old data structures to distinguish them from objects.

- Hash tables, also known as hash maps, are data structures that provide fast retrieval of values based on keys. They use a hashing function to map keys to indexes in an array, allowing for constant-time access in the average case. Hash tables are commonly used in dictionaries, caches, & database indexing. However, hash collisions can occur, which can impact their performance. Techniques like chaining & open addressing are employed to handle collisions.

- Graphs are collections of nodes connected by edges, representing relationships between entities. Graphs can be used to model social networks, computer networks, & transportation networks, among other things. They consist of vertices (nodes) & edges (connections between nodes). Graphs can be directed or undirected, & they can have cycles or be acylic. Graph traversal algorithms include breadth-1st search & depth-1st search.

- Stacks & queues are abstract data types that can be implemented using arrays or linked lists. A stack has 2 primary operations: push (adds an element to the top of the stack) & pop (removes the topmost element from the stack), that follow the Last In, First Out (LIFO) principle. Queues have 2 main operations: enqueue (adds an element to the rear of the queue) & dequeue (removes an element from the front of the queue) that follow the First In, First Out (FIFO) principle.

- Trees represents a hierarchical organization of elements. A tree consists of nodes connected by edges, with 1 node being the root & all other nodes forming subtrees. Trees are widely used in various algorithms & data storage scenarios. Binary trees (particularly heaps), AVL trees, & B-trees are some popular types of trees. They enable efficient & optimal searching, sorting, & hierarchical representation of data.

A trie, or prefix tree, is a special type of tree used to efficiently retrieve strings. In a trie, each node represents a character of a string, & the edges between nodes represent the characters that connect them. This structure is especially useful for tasks like autocomplete, spell-checking, & creating dictionaries. Tries allow for quick searches & operations based on string prefixes.

1.2.4 Language support

Most assembly languages & some low-level languages, such as BCPL (Basic Combined Programming Language), lack built-in support for data structures. On the other hand, many high-level programming languages & some higher-level assembly languages, e.g. MASM, have special syntax or other built-in support for certain data structures, e.g. records & arrays. E.g., the C (a direct descendant of BCPL) & Pascal languages support structs & records, respectively, in addition to vectors (1D arrays) & multi-dimensional arrays.

Most programming languages feature some sort of library mechanism that allows data structure implementations to be reused by different programs. Modern languages usually come with standard libraries that implement the most common data structures. E.g., C++ Standard Template Library, Java Collections Framework, & Microsoft .NET Framework.

Modern languages also generally support modular programming, the separation between the interface of a library module & its implementation. Some provide apaque data types that allow clients to hide implementation details. OOP languages, e.g. C++, Java, & Smalltalk, typically use classes for this purpose.

Many known data structures have concurrent versions which allow multiple computing threads to access a single concrete instance of a data structure simultaneously." – Wikipedia/data structure

1.3 Wikipedia/level set (data structures)

"In computer science, a level set data structure is designed to represent discretely sampled dynamic level sets functions. A common use of this form of data structure is in efficient image rendering. The underlying method constructs a signed distance field that extends the boundary, & can be used to solve the motion of the boundary in this field.

1.3.1 Chronological developments

The powerful level-set method is due to Osher & Sethian 1988. However, the straightforward implementation via a dense d-dimensional array of values, results in both time & storage complexity of $O(n^d)$, where n is the cross sectional resolution of the spatial extents of the domain & d is the number of spatial dimensions of the domain.

1. **Narrow band.** The narrow band level set method, introduced in 1995 by Adalsteinsson & Sethian, restricted most computations to a thin band of active voxels immediately surrounding the interface, thus reducing the time complexity in 3D to $O(n^2)$ for most operations. Periodic updates of the narrowband structure, to rebuild the list of active voxels, were required which entailed an $O(n^3)$ operation in which voxels over the entire volume were accessed. The storage complexity for this narrowband scheme was still $O(n^3)$. Differential constructions over the narrow band domain edge require careful interpolation & domain alternation schemes to stabilize the solution.

2. **Sparse field.** This $O(n^3)$ time complexity was eliminated in the approximate "sparse field" level set method introduced by Whitaker in 1998. The sparse field level set method employs a set of linked list to track the active voxels around the interface. This allows incremental extension of the active region as needed without incurring any significant overhead. While consistently $O(n^2)$ efficient in time, $O(n^3)$ storage space is still required by the sparse field level set method.

3. **Sparse block grid.** The sparse block grid method, introduced by Bridson in 2003, divides the entire bounding volume of size n^3 into small cubic blocks of m^3 voxels each. A coarse grid of size $\left(\frac{n}{m}\right)^3$ then stores pointers only to those blocks that intersect that narrow band of the level set. Block allocation & deallocation occur as the surface propagates to accommodate to the deformations. This method has a suboptimal storage complexity of $O(\left(\frac{n}{m}\right)^3 + m^3n^2)$, but retains the constant time access inherent to dense grids.

4. **Octree.** The octree level set method, introduced by Strain in 1999 & refined by Losasso, Gibu, & Fedkiw, & more recently by Min & Gibou uses a tree of nested cubes of which the leaf nodes contain signed distance values. Octree level sets currently require uniform refinement along the interface (i.e., the narrow band) in order to obtain sufficient precision. This representation is efficient in terms of storage, $O(n^2)$, & relatively efficient in terms of access queries, $O(\log n)$. A advantage of the level method on octree data structures is that one can solve the PDEs associated with typical free boundary problems that use the level set method. The CASL research group has developed this line of work in computational materials, CFD, electrokinetics, image guided surgery & controls.

5. **Run-length encoded.** The run-length encoding (RLE) level set method, introduced in 2004, applies the RLE scheme to compress regions away from the narrow band to just their sign representation while storing with full precision the narrow band. The sequential traversal of the narrow band is optimal & storage efficiency is further improved over the octree level set. The additional of an acceleration lookup table allows for fast $O(\log r)$ random access, where r is the number of runs per cross section. Additional efficiency is gained by applying the RLE scheme in a dimensional recursive fashion, a technique introduced by Nielsen & Museth's similar DT-Grid.

6. **Hash Table Local Level Set.** The Hash Table Local Level Set method, introduced in 2011 by Eyiyurekli & Breen & extended in 2012 by Brun, Guittet, & Gibou, only computes the level set data in a band around the interface, as in the Narrow Band Level-Set Method, but also only stores the data in that same band. A hash table data structure is used, which provides an $O(1)$ access to the data. However, Brun et al. conclude that their method, while being easier to implement, performs worse than a quadtree implementation. They find that

   > as it is, [...] a quadtree data structure seems more adapted than the hash table data structure for level-set algorithms.

   3 main reasons for worse efficiency are listed:

   (a) to obtain accurate results, a rather large band is required close to the interface, which counterbalances the absence of grid nodes far from the interface;

   (b) the performances are deteriorated by extrapolation procedures on the outer edges of the local grid &

   (c) the width of the band restricts the time step & slows down the method.

   (d) **Point-based.** Corbett in 2005 introduced the point-based level set method. Instead of using a uniform sampling of the level set, the continuous level set function is reconstructed from a set of unorganized point samples via moving least squares." – Wikipedia/level set (data structures)

2 Donald Knuth. The Art of Computer Programming. Volume 1: Fundamental Algorithms

Resources – Tài nguyên.

1. [Knu97]. Donald Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms.*

   Với bản dịch tiếng Việt chứa nhiều lỗi chính tả đến mức đáng bị cảnh sát ghé thăm:

2. [Kho06]. Nguyễn Văn Khoa. *Nghệ Thuật Lập Trình Máy Tính.*

This series of books is affectionately dedicated to the Type 650 computer once installed at Case Institute of Technology, in remembrance of many pleasant evenings.

The author & publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind & assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

2.1 Preface

"Here is your book, the one your thousands of letters have asked us to publish. It has taken us years to do, checking & rechecking countless recipes to bring you only the best, only the interesting, only the perfect. Now we can say, without a shadow of a doubt, that every single 1 of them, if you follow the directions to the letter, will work for you exactly as well as it did for us, even if you have never cooked before." – McCall's Cookbook (1963)

"The process of preparing programs for a digital computer is especially attractive, not only because it can be economically & scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music. This book is the 1st volume of a multi-volume set of books that has been designed to train the reader in various skills that go into a programmer's craft.

The following chapters are *not* meant to serve as an introduction to computer programming; the reader is supposed to have had some previous experience. The prerequisites are actually very simple, but a beginner requires time & practice in order to understand the concept of a digital computer. The reader should possess:

1. Some idea of how a stored-program digital computer works; not necessarily the electronics, rather the manner in which instructions can be kept in the machine's memory & successively executed.

2. An ability to put the solutions to problems into such explicit terms that a computer can "understand" them. (These machines have no common sense; they do exactly as they are told, no more & no less. This fact is the hardest concept to grasp when one 1st tries to use a computer.)

3. Some knowledge of the most elementary computer techniques, e.g. looping (performing a set of instructions repeatedly), the use of subroutines, & the use of indexed variables.

4. A little knowledge of common computer jargon – "memory," "registers," "bits," "floating point," "overflow," "software." Most words not defined in the text are given brief definitions in the index at the close of each volume.

These 4 prerequisites can perhaps be summed up into the single requirement that the reader should have written & tested at least, say, 4 programs for at least 1 computer.

I have tried to write this set of books in such a way that it will fill several needs. In the 1st place, these books are reference works that summarize the knowledge that has been acquired in several important fields. In the 2nd place, they can be used as textbooks for self-study or for college courses in the computer & information sciences. To meet both of these objectives, I have incorporated a large number of exercises into the text & have furnished answers for most of them. I have also made an effort to fill the pages with facts rather than with vague, general commentary.

This set of books is intended for people who will be more than just casually interested in computers, yet it is by no means only for the computer specialist. Indeed, 1 of my main goals has been to make these programming techniques more accessible to the many people working in other fields who can make fruitful use of computers, yet who cannot afford the time to locate all of the necessary information that is buried in technical journals.

We might call the subject of these books "nonumerical analysis." Computers have traditionally been associated with the solution of numerical problems e.g. calculation of roots of an equation, numerical interpolation & integration, etc., but such topics are not treated here except in passing. Numerical computer programming is an extremely interesting & rapidly expanding field, & many books have been written about it. Since the early 1960s, however, computers have been used even more often for problems in which numbers occur only by coincidence; the computer's decision-making capabilities are being used, rather than its ability to do arithmetic. We have some use for addition & subtraction in nonnumerical problems, but we rarely feel any need for multiplication & division. Of course, even a person who is primarily concerned with numerical computer programming will benefit from a study of the nonnumerical techniques, for they are present in the background of numerical programs as well.

The results of research in nonnumerical analysis are scattered throughout numerous technical journals. My approach has been to try to distill this vast literature by studying the techniques that are most basic, in the sense that they can be applied to many types of programming situations. I have attempted to coordinate the ideas into more or less of a "theory," as well as to show how the theory applies to a wide variety of practical problems.

Of course, "nonnumerical analysis" is a terribly negative name for this field of study; it is much better to have a positive, descriptive term that characterizes the subject. "Information processing" is too broad a designation for the material I am considering, & "programming techniques" is too narrow. Therefore I wish to propose *analysis of algorithms* as an appropriate name for the subject matter covered in these books. This name is meant to imply "the theory of the properties of particular computer algorithms."

The complete set of books, entitled *The Art of Computer Programming*, has the following general outline:

1. *Vol. 1: Fundamental Algorithms.* Chap. 1: Basic Concepts. Chap. 2: Information Structures.
2. *Vol. 2: Seminumerical Algorithms.* Chap. 3: Random Numbers. Chap. 4: Arithmetic.
3. *Vol. 3: Sorting & Searching.* Chap. 5: Sorting. Chap. 6: Searching.
4. *Vol. 4: Combinatorial Algorithms.* Chap. 7: Combinatorial Searching. Chap. 8: Recursion.
5. *Vol. 5: Syntactical Algorithms.* Chap. 9: Lexical Scanning. Chap. 10: Parsing.

Vol. 4 deals with such a large topic, it actually represents several separate books (Vols. 4A, 4B, etc.). 2 additional volumes on more specialized topics are also planned: Vol. 6: *The theory of Languages* (Chap. 11)' Vol. 7: *Compilers* (Chap. 12).

I started out in 1962 to write a single book with this sequence of chapters, but I soon found that it was more important to treat the subjects in depth rather than to skim over them lightly. The resulting length of the text has meant that each chapter by itself contains more than enough material for a 1-semester college course; so it has become sensible to publish the series in separate volumes. I know that it is strange to have only 1 or 2 chapters in an entire book, but I have decided to retain the original chapter numbering in order to facilitate cross references. A shorter version of Vols. 1–5 is planned, intended specifically to serve as a more general reference &/or text for undergraduate computer courses; its contents will be a subset of the material in these books, with the more specialized information omitted. The same chapter numbering will be used in the abridged edition as in the complete work.

The present volume may be considered as the "intersection" of the entire set, in the sense that it contains basic material that is used in all the other books. Vols. 2–5, on the other hand, may be read independently of each other. Vol. 1 is not only a reference book to be used in connection with the remaining volumes; it may also be used in college courses or for self-study as a text on the subject of *data structures* (emphasizing the material of Chap. 2), or as a text on the subject of *discrete mathematics*, or as a text on the subject of *machine-language programming*.

The point of view I have adopted while writing these chapters differs from that taken in most contemporary books about computer programming in that I am not trying to teach the reader how to use somebody else's software. I am concerned rather with teaching people how to write better software themselves.

My original goal was to bring readers to the frontiers of knowledge in every subject that was treated. But it is extremely difficult to keep up with a field that is economically profitable, & the rapid rise of computer science has made such a dream impossible. The subject has become a vast tapestry[1] with $a \cdot 10^4$ of subtle results contributed by $a \cdot 10^4$ of talented people all over the world. Therefore my new goal has been to concentrate on "classic" techniques that are likely to remain important for many more decades, & to describe them as well as I can. In particular, I have tried to trace the history of each subject, & to provide a solid foundation for future progress. I have attempted to choose terminology that is concise & consistent with current usage. I have tried to include all of the known ideas about sequential computer programming that are both beautiful & easy to state.

A few words are in order about the mathematical content of this set of books. The material has been organized so that persons with no more than a knowledge of high-school algebra may read it, skimming briefly over the more mathematical portions; yet a reader who is mathematically inclined will learn about many interesting mathematical techniques related to discrete mathematics. This dual level of presentation has been achieved in part by assigning ratings to each of the exercises so that the primarily mathematical ones are marked specifically as such, & also by arranging most sections so that the main mathematical results are stated *before* their proofs. The proofs are either left as exercises (with answers to be found in a separate section) or they are given at the end of a section.

A reader who is interested primarily in programming rather than in the associated mathematics may stop reading most sections as soon as the mathematics becomes recognizably difficult. On the other hand, a mathematically oriented reader will find a wealth of interesting material collected here. Much of the published mathematics about computer programming has been faulty, & 1 of the purposes of this book is to instruct readers in proper mathematical approaches to this subject. Since I profess to be a mathematician, it is my duty to maintain mathematical integrity as well as I can.

A knowledge of elementary calculus will suffice for most of the mathematics in these books, since most of the other theory that is needed is developed herein. However, I do need to use deeper theorems of complex variable theory, probability theory, number theory, etc., at times, & in such cases I refer to appropriate textbooks where those subjects are developed.

The hardest decision that I had to make while preparing these books concerned the manner in which to present the various techniques. The advantages of flow charts & of an informal step-by-step description of an algorithm are well known; for a discussion of this, see the article "Computer-Drawn Flowcharts" in the ACM *Communications*, Vol. 6 (Sep. 1963), pp. 555–563. Yet a formal, precise language is also necessary to specify any computer algorithm, & I needed to decide whether to use an algebraic language, e.g. ALGOL or FORTRAN, or to use a machine-oriented language for this purpose. Perhaps many of today's computer experts will disagree with my decision to use a machine-oriented language, but I have become convinced that it was definitely the correct choice, for the following reasons:

[1] tấm thảm.

1. A programmer is greatly influenced by the language in which programs are written; there is an overwhelming tendency to prefer constructions that are simplest in that language, rather than those that are best for the machine. By understanding a machine-oriented language, the programmer will tend to use a much more efficient method; it is much closer to reality.

2. The programs we require are, with a few exceptions, all rather short, so with a suitable computer there will be no trouble understanding the programs.

3. High-level languages are inadequate for discussing important low-level details e.g. coroutine linkage, random number generation, multi-precision arithmetic, & many problems involving the efficient usage of memory.

4. A person who is more than casually interested in computers should be well schooled in machine language, since it is a fundamental part of a computer.

5. Some machine language would be necessary anyway as output of the software programs described in many of the examples.

6. New algebraic languages go in & out of fashion every 5 years or so, while I am trying to emphasize concepts that are timeless.

From the other point of view, I admit that it is somewhat easier to write programs in higher-level programming languages, & it is considerably easier to debug the programs. Indeed, I have rarely used low-level machine language for my own programs since 1970, now that computers are so large & so fast. Many of the problems of interest to us in this book, however, are those for which the programmer's art is most important. E.g., some combinatorial calculations need to be repeated a trillion times, & we save about 11.6 days of computation for every microsecond we can squeeze out of their inner loop. Similarly, it is worthwhile to put an additional effort into the writing of software that will be used many times each day in many computer installations, since the software needs to be written only once.

Given the decision to use a machine-oriented language, which language should be used? I could have chosen the language of a particular machine X, but then those people who do not possess machine X would think this book is only for X-people. Furthermore, machine X probably has a lot of idiosyncrasies[2] that are completely irrelevant to the material in this book yet which must be explained; & in 2 years the manufacturer of machine X will put out machine $X+1$ or machine $10X$, & machine X will no longer be of interest to anyone.

To avoid this dilemma, I have attempted to design an "ideal" computer with very simple rules of operation (requiring, say, only an hour to learn), which also resembles actual machines very closely. There is no reason why a student should be afraid of learning the characteristics of > 1 computer; once 1 machine language has been mastered, others are easily assimilated. Indeed, serious programmers may expect to meet different machine languages in the course of their careers. So the only remaining disadvantage of a mythical machine is the difficulty of executing any programs written for it. Fortunately, that is not really a problem, because many volunteers have come toward to write simulators for the hypothetical machine. Such simulators are ideal for instructional purposes, since they are even easier to use than a real computer would be. I have attempted to cite the best early papers in each subject, together with a sampling of more recent work. When referring to the literature, I use standard abbreviations for the names of periodicals, except that the most commonly cited journals are abbreviated as follows:

1. CACM = Communications of the Association for Computing Machinery
2. CACMJACM = Journal of the Association for Computing Machinery
3. CACMComp. J. = The Computer Journal (British Computer Society)
4. CACMMath. Comp. = Mathematics of Computation
5. CACMAMM = American Mathematical Monthly
6. CACMSICOMP = SIAM Journal on Computing
7. CACMFOCS = IEEE Symposium on Foundations of Computer Science
8. CACMSODA = ACM–SIAM Symposium on Discrete Algorithms
9. CACMSTOC = ACM Symposium on Theory of Computing
10. CACMCrelle = Journal für die reine und angewandte Mathematik

I also use "CMath" to stand for the book *Concrete Mathematics*.

Preface to the 3rd Edition. After having spent 10 years developing the TEX & METAFONT systems for computer typesetting, I am now able to fulfill the dream that I had when I began that work, by applying those systems to *The Art of Computer Programming*. At last the entire text of this book has been captured inside my personal computer, in an electronic form that will make it readily adaptable to future changes in printing & display technology. The new setup has allowed me to make literally thousands of improvements that I've been wanting to incorporate for a long time.

In this new edition I have gone over every word of the text, trying to retain the youthful exuberance[3] of my original sentences while perhaps adding some more mature judgment. Dozens of new exercises have been added; dozens of old exercises have been given new & improved answers.

[2] đặc điểm riêng.
[3] sự hồ hởi.

The Art of Computer Programming is, however, still a work in progress. Therefore some parts of this book are headed by an "under construction" icon, to apologize for the fact that the material is not up-to-date. My files are bursting with important material that I plan to include in the final, glorious, 4th edition of Vol. 1, perhaps 15 years from now; but I must finish Vols. 4–5 1st, & I do not want to delay their publication any more than absolute necessary.

My efforts to extend & enhance these volumes have been enormously enhanced since 1980 by the wise guidance of Addison–Wesley's editor PETER GORDON. He has become not only my "publishing partner" but also a close friend, while continually nudging me to move in fruitful directions. Indeed, my interactions with dozens of Addison–Wesley people during > 3 decades have been much better than any author deserves. The tireless support of managing editor JOHN FULLER, whose meticulous attention to detail has maintained the highest standards of production quality in spite of frequent updates, has been particularly praiseworthy.

"Things have changed in the past 2 decades." – BILL GATES (1995)

"Woe be to him that reads but one book." – GEORGE HERBERT, *Jacula Prudentum*, 1144 (1640)

"Le défaut unique de tous les ouvrages c'est d'être trop longs." – VAUVENARGUES, *Réflexions*, 628 (1746)

"Books are a triviality[4]*. Life alone is great."* – THOMAS CARLYLE, *Journal* (1839)

Notes on the Exercises. The exercises in this set of books have been designed for self-study as well as for classroom study. It is difficult, if not impossible, for anyone to learn a subject purely by reading about it, without applying the information to specific problems & thereby being encouraged to think about what has been read. Furthermore, we all learn best the things that we have discovered for ourselves. Therefore the exercises form a major part of this work; a definite attempt has been made to keep them as informative as possible & to select problems that are enjoyable as well as instructive.

In many books, easy exercises are found mixed randomly among extremely difficult ones. A motley[5] mixture is, however, often unfortunate because readers like to know in advance how long a problem ought to take – otherwise they may just skip over all the problems. A classic example of such a situation is the book *Dynamic Programming* by RICHARD BELLMAN; this is an important, pioneering work in which a group of problems is collected together at the end of some chapters under the heading "Exercises & Research Problems," with extremely trivial questions appearing in the midst of deep, unsolved problems. It is rumored that someone once asked Dr. BELLMAN how to tell the exercises apart from the research problems, & he replied, "If you can solve it, it is an exercise; otherwise it's a research problem."

Good arguments can be made for including both research problems & very easy exercises in a book of this kind; therefore, to save the reader from the possible dilemma of determining which are which, *rating numbers* have been provided to indicate the level of difficulty. These numbers have the following general significance: Rating: Interpretation

- 00: An extremely easy exercise that can be answered immediately if the material of the text has been understood; such an exercise can almost always be worked "in your head."
- 10: A simple problem that makes you think over the material just read, but is by no means difficult. You should be able to do this in one minute at most; pencil & paper may be useful in obtaining the solution.
- 20: An average problem that tests basic understanding of the text material, but you may need about 15 or 20 minutes to answer it completely.
- 30: A problem of moderate difficulty &/or complexity; this one may involve > 2 hours' work to solve satisfactorily, or even more if the TV is on.
- 40: Quite a difficult or lengthy problem that would be suitable for a term project in classroom situations. A student should be able to solve the problem in a reasonable amount of time, but the solution is not trivial.
- 50: A research problem that has not yet been solved satisfactorily, as far as the author knew at the time of writing, although many people have tried. If you have found an answer to such a problem, you ought to write it up for publication; furthermore, the author of this book would appreciate hearing about the solution as soon as possible (provided that it is correct).

By interpolation in this "logarithmic" scale, the significance of other rating numbers becomes clear. E.g., a rating of 17 would indicate an exercise that is a bit simpler than average. Problems with a rating of 50 that are subsequently solved by some reader may appear with a 40 rating in later editions of the book, & in the errata posted on the Internet.

The remainder of the rating number divided by 5 indicates the amount of detailed work required. Thus, an exercise rated 24 may take longer to solve than an exercise that is rated 25, but the latter will require more creativity. All exercises with ratings of 46 or more are open problems for future research, rated according to the number of different attacks that they've resisted so far.

The author has tried earnestly to assign accurate rating numbers, but it is difficult for the person who makes up a problem to know just how formidable it will be for someone else to find a solution; & everyone has more aptitude for certain types of problems than for others. It is hoped that the rating numbers represent a good guess at the level of difficulty, but the should be taken as general guidelines, not as absolute indicators.

This book has been written for readers with varying degrees of mathematical training & sophistication; as a result, some of the exercises are intended only for the use of more mathematically inclined readers. The rating is preceded by an M if the

[4] sự tầm thường.

[5] sặc sỡ.

exercise involves mathematical concepts or motivation to a greater extent than necessary for someone who is primarily interested only in programming the algorithms themselves. An exercise is marked with the letters "HM" if its solution necessarily involves a knowledge of calculus or other higher mathematics not developed in this book. An "HM" designation does *not* necessarily imply difficulty.

Some exercises are preceded by an arrowhead, ▷; this designates problems that are especially instructive & especially recommended. Of course, no reader/student is expected to work *all* of the exercises, so those that seem to be the most valuable have been singled out. (This distinction is not meant to detract from the other exercises!) Each reader should at least make an attempt to solve all of the problems whose rating is 10 or less; & the arrows may help to indicate which of the problems with a higher rating should be given priority.

Solutions to most of the exercises appear in the answer section. Please use them wisely; do not turn to the answer until you have made a genuine effort to solve the problem by yourself, or unless you absolutely do not have time to work this particular problem. *After* getting your own solution or giving the problem a decent try, you may find the answer instructive & helpful. The solution given will often be quite short, & it will sketch the details under the assumption that you have earnestly tried to solve it by your own means 1st. Sometimes the solution gives less information than was asked; often it gives more. It is quite possible that you may have a better answer than the one published here, or you may have found an error in the published solution; in such a case, the author will be pleased to know the details. Later printings of this book will give the improved solutions together with the solver's name where appropriate.

When working an exercise you may generally use the answers to previous exercises, unless specifically forbidden from doing so. The rating numbers have been assigned with this in mind; thus it is possible for exercise $n + 1$ to have a lower rating than exercise n, even though it includes the result of exercise n as a special case.

Summary of codes. ▷: Recommended. M: Mathematical oriented. HM: Requiring "higher math". 00: Intermediate. 10: Simple (1 min). 20: Medium (15 mins). 30: Moderately hard. 40: Term project. 50: Research problem.

2. [10] Of what value can the exercises in a textbook be to the reader? **3.** Generalize your answer. [This is an example of a horrible kind of problem that the author has tried to avoid.]

> *"We can face our problem. We can arrange such facts as we have with order & method."* – Hercule Poirot, in *Murder on the Orient Express* (1934)

2.2 Chap. 1: Basic Concepts

> *"Many persons who are not conversant with mathematical studies imagine that because the business of [Babbage's Analytical Engine] is to give its results in numerical notation, the nature of its processes must consequently be arithmetical & numerical, rather than algebraical & analytical. This is an error. The engine can arrange & combine its numerical quantities exactly as if they were letters or any other general symbols; & in fact it might bring out its results in algebraical notation, were provisions made accordingly."* – Augusta Ada, *Countess of Lovelace* (1843)
>
> *"Practice yourself, for heaven's sake, in little things; & thence proceed to greater."* – Epictetus, *Discourses* IV.i

2.2.1 Algorithms

The notion of an *algorithm* is basic to all of computer programming, so we should begin with a careful analysis of this concept.

The word "algorithm" itself is quite interesting; at 1st glance it may look as though someone intended to write "logarithm" but jumbled up the 1st 4 letters. The word did not appear in *Webster's New World Dictionary* as late as 1957; we find only the older form "algorism" with its ancient meaning, the process of doing arithmetic using Arabic numerals. During the Middle Ages, abacists computed on the abacus & algorists computed by algorism. By the time of the Renaissance, the origin of this word was in doubt, & early linguists attempted to guess at its derivation by making combinations like *algiros* [painful] + *arithmos* [number]; others said no, the word comes from "King Algor of Castile." Finally, historians of mathematics found the true origin of the word algorism: It comes from the name of a famous Persian textbook author, Abu 'Abd Allah Muh ammad ibn Musa al-Khwarizmi (c. 825) – literally, "Father of Abdullah, Mohammed, son of Moses, native of Khwarizm." The Aral Sea in Central Asia was once known as Lake Khwarizm, & the Khwarizm region is located in the Amu River basin just south of that sea. Al-Khwarizmi wrote the celebrated Arabic text *Kitab al-jabr wa'l-muqabala* ("Rules of restoring & equating"); another word, "algebra," stems from the title of that book, which was a systematic study of the solution of linear & quadratic equations.

Gradually the form & meaning of *algorism* became corrupted; as explained by the *Oxford English Dictionary*, the word "passed through many pseudo-etymological perversions, including a recent *algorithm*, in which it is learnedly confused" with the Greek root of the word *arithmetic*. This change from "algorism" to "algorithm" is not hard to understand in view of the fact that people had forgotten the original derivation of the word. An early German mathematical dictionary, *Vollständiges mathematisches Lexicon* (Leipzig: 1747), gave the following definition for the word *Algorithmus*: "Under this designation are combined the notions of the 4 types of arithmetic calculations, namely addition, multiplication, subtraction, & division." The Latin phrase *algorithmus infinitesimalis* was at that time used to denote "ways of calculation with infinitely small quantities, as invented by Leibniz."

By 1950, the word algorithm was most frequently associated with Euclid's algorithm, a process for finding the greatest common divisor of 2 numbers appearing in Euclid's *Elements* (Book 7, Props. 1–2), see [Knu97, Algorithm E: Euclid's algorithm, p. 2].

Each algorithm we consider has been given an identifying letter (E in the preceding example), and the steps of the algorithm are identified by this letter followed by a number (E1, E2, E3).

Each step of an algorithm, e.g. step E1 above, begins with a phrase in brackets that sums up as briefly as possible the principal content of that step. This phrase also usually appears in an accompanying *flow chart* so that the reader will be able to picture the algorithm more readily.

After the summarizing phrase comes a description in words & symbols of some *action* to be performed or some decision to be made. Parenthesized *comments*, like the 2nd sentence in step E1, may also appear. Comments are included as explanatory information about that step, often indicating certain invariant characteristics of the variables or the current goals. They do not specify actions belonging to the algorithm, but are meant only for the reader's benefit as possible aids to comprehension.

2.3 Chap. 2: Information Structures

3 Linux

Resources – Tài nguyên.

1. [Sho19]. William Shotts. *The Linux Command Line: A Complete Introduction.*

I used SUSE & OpenSUSE in WIAS Berlin but I do not like them & the like, so I go back to Ubuntu.

4 Programming

4.1 C/C++

Resources – Tài nguyên.

1. [Ngọ02]. Quách Tuấn Ngọc. *Ngôn Ngữ Lập Trình C.*
2. [Ngọ03]. Quách Tuấn Ngọc. *Ngôn Ngữ Lập Trình C++.*
3. [Str13]. Bjarne Stroustrup. *The C++ Programming Language.*
4. [Str18]. Bjarne Stroustrup. *A Tour of C++.*

4.2 Pascal

Resources – Tài nguyên.

1. [Ngọ08]. Quách Tuấn Ngọc. *Ngôn Ngữ Lập Trình Pascal.*
2. [Ngọ09]. Quách Tuấn Ngọc. *Bài Tập Ngôn Ngữ Lập Trình Pascal.*
3. [DT06]. Lê Văn Doanh, Trần Khắc Tuấn. *101 Thuật Toán & Chương Trình Bài Toán Khoa Học Kỹ Thuật & Kinh Tế Bằng Ngôn Ngữ Turbo-Pascal.*

4.3 Python

Resources – Tài nguyên.

1. [Đức22]. Nguyễn Tiến Đức. *Tuyển Tập 200 Bài Tập Lập Trình Bằng Ngôn Ngữ Python.*
2. [Huy24]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 1.*
3. [**Huy_sang_tao_thuat_toan_lap_trinh_tap_2**]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 2.*
4. [**Huy_sang_tao_thuat_toan_lap_trinh_tap_3**]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 3.*
5. [**Huy_sang_tao_thuat_toan_lap_trinh_tap_4**]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 4.*
6. [**Huy_sang_tao_thuat_toan_lap_trinh_tap_5**]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 5.*
7. [**Huy_sang_tao_thuat_toan_lap_trinh_tap_6**]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 6.*
8. [**Huy_sang_tao_thuat_toan_lap_trinh_tap_7**]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 7.*

5 Software

5.1 FeNiCS

Resources – Tài nguyên.

1. [Dok20]. Jørgen S. Dokken. *Automatic shape derivatives for transient PDEs in FEniCS & Firedrake.*
2. [LL16]. Hans Petter Langtangen, Anders Logg. *Solving PDEs in Python.*

5.2 Firedrake

5.3 Fireshape

Resources – Tài nguyên.

1. [PW20]. Alberto Paganini, Florian Wechsung. *Fireshape Documentation, Release 0.0.1.*
2. [PW21]. Alberto Paganini, Florian Wechsung. *Fireshape: a shape optimization toolbox for Firedrake.*

5.4 Git

Resources – Tài nguyên.

1. [CS14]. Scott Chacon, Ben Straub. *Pro Git.*

5.5 Gmsh

Resources – Tài nguyên.

1. [GR09]. Christophe Geuzaine, Jean-François Remacle. *Gmsh: A 3D finite element mesh generator with built-in pre- & post-processing facilities.*

5.6 OpenFOAM

Resources – Tài nguyên.

1. There are 3 variants of OpenFOAM:
   (a) OpenFOAM.com: Commercial.
   (b) OpenFOAM.org: Open-source with a large community.
   (c) Extended OpenFOAM.
2. [GW22]. Christopher Greenshields, Henry Weller. *Notes on Computational Fluid Dynamics: General Principles.*
3. [TN13]. M. Towara, U. Naumann. *A Discrete Adjoint Model for OpenFOAM.*

5.7 ParMooN

Resources – Tài nguyên.

1. [Wil+17]. Ulrich Wilbrandt, Clemens Bartsch, Naveed Ahmed, Volker John. *ParMooN – a modernized program package based on mapped finite elements.*

5.8 SU2

5.9 Sublime Text

Resources – Tài nguyên.

1. [Bos14]. Wes Bos. *Sublime Text Power User: A Complete Guide.*
2. [Pel13]. Dan Peleg. *Mastering Sublime Text*

6 Miscellaneous

Tài liệu

[Bos14] Wes Bos. *Sublime Text Power User: A Complete Guide*. 2014, p. 202.

[CS14] Scott Chacon and Ben Straub. *Pro Git*. 2nd. Apress, 2014, p. 458.

[Dok20] Jørgen S. Dokken. "Automatic shape derivatives for transient PDEs in FEniCS and Firedrake". In: (2020). URL: https://arxiv.org/abs/2001.10058.

[DT06] Lê Văn Doanh and Trần Khắc Tuấn. *101 Thuật Toán & Chương Trình Bài Toán Khoa Học Kỹ Thuật & Kinh Tế Bằng Ngôn Ngữ Turbo-Pascal*. In lần thứ 10. Nhà Xuất Bản Khoa Học & Kỹ Thuật, 2006, p. 268.

[Dức22] Nguyễn Tiến Dức. *Tuyển Tập 200 Bài Tập Lập Trình Bằng Ngôn Ngữ Python*. Nhà Xuất Bản Đại Học Thái Nguyên, 2022, p. 327.

[GR09] Christophe Geuzaine and Jean-François Remacle. "Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities". In: *Internat. J. Numer. Methods Engrg.* 79.11 (2009), pp. 1309–1331. ISSN: 0029-5981. DOI: 10.1002/nme.2579. URL: https://doi.org/10.1002/nme.2579.

[GW22] Christopher Greenshields and Henry Weller. *Notes on Computational Fluid Dynamics: General Principles*. Reading, UK: CFD Direct Ltd, 2022.

[Huy24] Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 1*. Tái bản lần 10. Nhà Xuất Bản Thông Tin & Truyền Thông, 2024, p. 371.

[Kho06] Nguyễn Văn Khoa. *Nghệ Thuật Lập Trình Máy Tính*. Nhà Xuất Bản Giao Thông Vận Tải, 2006, p. 1050.

[Knu97] Donald Ervin Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*. 3rd edition. Addison-Wesley Professional, 1997, pp. xx+652.

[LL16] Hans Petter Langtangen and Anders Logg. *Solving PDEs in Python*. Vol. 3. Simula SpringerBriefs on Computing. The FEniCS tutorial I. Springer, Cham, 2016, pp. ix+148. ISBN: 978-3-319-52461-0; 978-3-319-52462-7. DOI: 10.1007/978-3-319-52462-7. URL: https://doi.org/10.1007/978-3-319-52462-7.

[Ngọ02] Quách Tuấn Ngọc. *Ngôn Ngữ Lập Trình C*. Nhà Xuất Bản Thống Kê, 2002, p. 425.

[Ngọ03] Quách Tuấn Ngọc. *Ngôn Ngữ Lập Trình C++*. Nhà Xuất Bản Thống Kê, 2003, p. 476.

[Ngọ08] Quách Tuấn Ngọc. *Ngôn Ngữ Lập Trình Pascal*. Nhà Xuất Bản Thống Kê, 2008, p. 338.

[Ngọ09] Quách Tuấn Ngọc. *Bài Tập Ngôn Ngữ Lập Trình Pascal*. Nhà Xuất Bản Giáo Dục, 2009, p. 187.

[Pel13] Dan Peleg. *Mastering Sublime Text*. Packt Publishing, Birmingham - Mumbai, 2013, pp. iv+94.

[PW20] Alberto Paganini and Florian Wechsung. "Fireshape Documentation, Release 0.0.1". In: (2020), pp. ii+31. URL: https://fireshape.readthedocs.io/en/latest/index.html.

[PW21] Alberto Paganini and Florian Wechsung. "Fireshape: a shape optimization toolbox for Firedrake". In: *Struct. Multidiscip. Optim.* 63.5 (2021), pp. 2553–2569. ISSN: 1615-147X. DOI: 10.1007/s00158-020-02813-y. URL: https://doi.org/10.1007/s00158-020-02813-y.

[Sho19] William Shotts. "The Linux Command Line: A Complete Introduction". In: (2019), p. 640.

[Str13] Bjarne Stroustrup. *The C++ Programming Language*. 4th edition. Pearson Addison-Wesley, 2013, pp. xiv+1346.

[Str18] Bjarne Stroustrup. *A Tour of C++*. 2nd edition. Pearson Addison-Wesley, 2018, pp. xii+240.

[TN13] M. Towara and U. Naumann. "A Discrete Adjoint Model for OpenFOAM". In: *Procedia Computer Science* 18 (2013), pp. 429–438. DOI: 10.1016/j.procs.2013.05.206. URL: https://doi.org/10.1016/j.procs.2013.05.206.

[Wil+17] Ulrich Wilbrandt, Clemens Bartsch, Naveed Ahmed, and et al. "ParMooN—a modernized program package based on mapped finite elements". In: *Comput. Math. Appl.* 74.1 (2017), pp. 74–88. ISSN: 0898-1221. DOI: 10.1016/j.camwa.2016.12.020. URL: https://doi.org/10.1016/j.camwa.2016.12.020.