# A Mathematical Investigation of the Halting Function and the Resolution of the Halting Paradox via Termination Analysis on Definite Topological Spaces

Smt. Deshmukh Naziya Sultana Khursheed Ahmed
*Lecturer in Mathematics, Government Polytechnic, Nanded*

***Abstract*—One of the most essential topics dealing with computability theory and mathematical logic is the Halting Problem. According to classical computability theory, there is no universal function $H(P, I)$ that can decide whether an arbitrary program $P$ halts or not on input $I$. A new mathematical framework for the existence of halting function over definite topological space $(X, \tau)$ is proposed in this paper. The execution of the program is viewed as a continuous mapping $f: X \to X$. The different computational states are defined as points of a topological space.**

**The issue analyzes the convergence, compactness, and fixed-point theory that analyzes if a program will terminate. A computational sequence $\{x_n\}$ generated by**

$$x_{n+1} = f(x_n)$$

**is said to terminate if there exists a stable point $x^* \in X$ such that**

$$f(x^*) = x^*$$

**While a universal halting function is undecidable for general-purpose computational systems, halting functions exist for limited domains. These must be compact and finite state space topologies of the entire computational state space. The framework gives a non-convergence proof of a, which is the classical halting paradox.**

***Index Terms*—Halting Problem, Halting Function, Topological Spaces, Computability Theory, Program Termination, Mathematical Logic, Fixed Point Theory, Formal Verification.**

## I. INTRODUCTION

The Halting Problem is a main focus of Computability Theory among all undecidable problems. In 1936 Alan Turing proved no algorithm exists to tell whether an arbitrary program $P$ halts on input I The program may loop forever. The theory of computation, one of the most essential areas of mathematics and computer science, deals with the performance and limitations of algorithms. The classical halting function is often denoted as.

$$H(P, I) = \begin{cases} 1, & \text{if } P \text{ halts on } I \\ 0, & \text{if } P \text{ runs forever on } I \end{cases}$$

Turing demonstrated that a universal function cannot exist for all computational systems because it leads to a logical contradiction and self-referential behavior. Consequently, his finding became one of the foundations for modern theoretical computer science and mathematical logic.

Usually, in classical combinational or sequential computing, specifications and analyses of programs are done symbolically by means of Turing machines, recursive functions and formal languages. According to modern mathematical research, computational processes can be interpreted in a geometric way by topological and functional analytic structures. Consequently, a computational process may thus be conceived of as a dynamic of states in a suitable space where each computational state is…

$(X, \tau)$.

In the proposed framework, program execution is represented by a continuous mapping

$$f: X \to X$$

where $X$ denotes the set of computational states. Starting from an initial state $x_0 \in X$, the execution of the program generates a sequence

$$x_{n+1} = f(x_n), n \geq 0$$

The behavior of this sequence determines whether the program terminates or continues indefinitely. If there exists a point $x^* \in X$ such that

$$f(x^*) = x^*$$

then $x^*$ represents a stable halting state of the computation. So, we might view termination rather as a fixed-point and convergence issue instead of merely a symbolic logical issue. Topology employs a number of useful mathematical tools, including compactness, continuity, connectedness, convergence, and so on, to understand the iterations. A compact topological space won't let you wander off without enough direction. Through utilizing the fixed-point theory, the researchers will be able to determine a stable disposition of these iterative computations and they will be converging such a point. So, restricting computation only to definite topological spaces can lead us to understand the halting of computation or any iterative procedure in a mathematically rigorous and controlled manner.

This paper aims to investigate existences of halting functions over certain topological space with a view to develop a mathematical framework for termination checking. We investigate how a global halting function may be impossible in an unrestricted computation, but there may exist a restricted halting structure over a specific finite or compact space. This means we are turning the classical halting paradox into a convergence problem of structured maths.

This paper sets a connection between topology and computability and formal verification. The framework proposed by this paper may be applicable for analysis of program correctness, automated theorem proving, compiler optimization, reliable software system design etc. A study of such kind opens up a new direction to understand computation through a modern sophisticated mathematical structure/topological method

## II. RELATED WORKS

The contemporary work on computability and halting problems finds use in modern mathematics and theoretical computer science. Important ideas that delineate the boundaries and capabilities of computation have emerged from various fields with logic, topology, recursion theory, programming language semantics, etc. Our research is fundamentally based on these works and aims at their generalization in a topological setting for termination checking.

Turing made one of the earliest and most influential contributions. In 1937, he wrote a famous paper on computable numbers and the Entscheidungsproblem. In this piece, he is a Turing machine is a model of computation. Moreover, he demonstrated that it is impossible to develop an algorithm that can determine if a given program will halt or go on indefinitely. Stated differently, he established the undecidability of the halting problem. Turing's work is the foundation of theoretical computer science. It discussed what computation cannot do. Moreover, it established a formal mathematical framework for studying machines and algorithms.

During this time, Alonzo Church created closely related alternatives using lambda calculus and recursive functions. In elementary number theory, Church dealt with an unsolvable problem and thereby indicated that mathematical problems cannot be solved by any algorithmic process. His thinking culminated in what is now called the Church-Turing thesis, which indicates that every effectively computable function can be represented symbolically by the concept of being recursive or Turing-computable. Church therefore reinforced the understanding of the calculation which we have covered up till now has natural limits.

Kurt Gödel's incompleteness theorem also influenced the computability theory. Gödel proved that every sufficiently powerful mathematical system contains true statements that cannot be proved in that system. The discovery caused a stir in foundational mathematics in that it showed logical systems have limits. Gödel's ideas are related to the halting problem because both address the limits of formal reasoning and the decidability of algorithms. The next influential researcher who extended the computability theory was Stephen Kleene. In a lengthy and comprehensive work about recursive functions and metamathematics, Kleene introduced partial recursive functions, computable predicates, recursive relations and more. Since these words also became terms utilized in the studies of decidability and computation, it was Kleene's paper that researchers relied on. The Kleene study is a formulation of mathematics in which one can do a systematic study of recursive computation. In the

ensuing years, scientists began looking at computing with more advanced mathematical structures. Dana Scott developed domain theory and denotational semantics, which are techniques for defining the meanings of programming languages. Scott demonstrated that computation can be represented using ordered structures and continuous functions between them. The work of Scott created a crucial link between topology and computer science. This study aligns with Scott's research, as both view computations as continuous operations performed on a mathematical space. John C.

Mitchell played a key role in programming language theory and semantics. His research demonstrated the use of mathematical logic and formal systems to describe the behaviour and correctness of computer programs. Mitchell's is important because it connects the programming of systems with formal mathematical reasoning. Since these termination checking and verification procedures are functionally formal, this underpins the present work in a strong theoretical sense. M. Huth and M. Ryan explored logical approaches for modeling computing systems. Logic was studied in Computer Studies with reference to reasoning systems, model checking, verification and other related methods. The writers showed us the design and application of logical systems which can properly establish whether programs obey some property. It is essential to ensure that a program will stop running eventually and will not carry on running forever. The quoted reference work provides strong theoretical basis to present work. Topology has been the main source of several important notions of this work. J. L. Kelley laid down the foundations of general topology. This popular topical work clarified the basic notions of topology-open sets, continuity, convergence, compactness and connectedness. All of this will be essential in the work to be presented because calculation will be interpreted through topological spaces and their continuous maps or morphisms. Additionally, the concepts of compactness and convergence help us understand whether the computation sequence stabilizes or continues to diverge. Allen Hatcher's work on algebraic topology introduces sophisticated topological techniques such as homotropy and homology.

Algebraic topology arose mainly due to the application to geometry and topology gave insight in structure of space and how transformation occur in space. In other words may outline what computational paths may make with better dimensional structures.

The primary agent behind the development of category theory was Saunders Mac Lane. This area of study examines objects and their various transformations at an extremely general level. In addition, this is one of the most abstract mathematical frameworks. The application of the category theory is quite strong in logics as well as computer sciences.

Edsger W. Dijkstra emphasized a disciplined programming approach, along with correctness proofs of programs. He has shown that programming activities are mathematical activities that rely on correctness and reliability. Hence, one must use formal modelling techniques while designing a computer programme. This has a direct relation to our aim of termination checking.

Hartley Rogers Jr. contributed primarily to recursive function theory and effective computability. He articulated in an exact manner the mathematical nature of unresolvable problems and computable functions. Research by Rogers strengthens the theory of recursive systems and undecidability.

P. T Johnstone has studied Stone spaces and their application in topology and logic in depth. The studies showed that a logical system and a topological space can be the ultra product or equivalently the algebras of elementary classes of each other. The present study may benefit from the analysis of the intertwining of topology and logic. The study of the computational behaviour of a machine or a program in the context of a topological manifold or space is known as topological computation. In any event, many basic features of computational topology are still not known.

The halting problem has recently been studied by different researchers from different perspectives. L. Peralta performed interesting research on the computable domain of very complex halting functions, and conjectures on the possible existence of extensions complete of partial functions. He also considers self-referential programs which must, on input of their own code, decide the halting or non-halting of an unrestricted programme. He proposes studying confined mathematical and computational frameworks, where one could meaningfully explore the termination behavior of codes and programs that

conform to restricted formal machines. What the author of the result attempts to demonstrate relates to the classical result that total halting functions are impossible. However, the outcome is confined to a large yet smaller domain than the entire extension or possible positive one-sided complete solution to the halting problem of an ultra-partial halting function on an infinite computably enumerable domain of input programs and their codes. this work studies more rigorously the possible existence of a halting mechanism which there may exist.

Erving Goffman's book, "The Presentation of Self in Everyday Life", focuses on the idea of self-presentation. This refers to the manner in which we express ourselves how we show who we are to other people in society. The notion suggests that individuals can present themselves differently, based on the social situation they are in as well as how they want others to see them. A law student example could just be a normal guy but, at the same time, seen as a future attorney or respectable professional.

The idea behind the book is that we are all actors on a stage. So, we play certain parts to impress the other members of the cast aka members of society. In other words, we put on performances for others. Goffman's social perspective sees interaction as performative, just like theatrical performances in front of an audience.

## III. PRELIMINARIES

In studying the mathematical existence of the halting function, it is assumed some preliminary notions of topology, computability theory and fixed-points. Some fundamental definitions, as well as results from these domains, serve as the conceptual foundation for a method of termination verification. Within definite spaces a topological one.

1. Topological Space

Let $X$be a non-empty set and let $\tau$be a collection of subsets of $X$. The pair $(X, \tau)$is called a topological space if the following conditions hold:

1. $\emptyset \in \tau$and $X \in \tau$
2. The union of arbitrary elements of $\tau$belongs to $\tau$
3. The intersection of finitely many elements of $\tau$belongs to $\tau$

Here, $X$represents the set of computational states, while $\tau$defines the structural behavior of the space.

2. Continuous Mapping

Let $(X, \tau_X)$and $(Y, \tau_Y)$be two topological spaces. A function

$$f: X \to Y$$

is said to be continuous if for every open set $V \subseteq Y$, the inverse image

$$f^{-1}(V) \subseteq X$$

is also open in $X$.

As per this paper, in other words a program's evolution is depicted by a continuous mapping since computation transitions must be structure-preserving mapping in the state space.

3. Computational Sequence

Starting from an initial computational state $x_0 \in X$, repeated application of the execution function $f$generates a sequence

$$x_{n+1} = f(x_n), n \geq 0$$

The sequence

$$\{x_n\} = \{x_0, x_1, x_2, \dots\}$$

A program's behaviour is a complete computational behaviour. The program will stop when this sequence converges. In the same way a program can diverge, so can the sequence.

4. Halting Function

The classical halting function is defined as

$$H(P, I) = \begin{cases} 1, & \text{if program } P \text{ halts on input } I \\ 0, & \text{otherwise} \end{cases}$$

Here $P$ refers to a program and $I$ refer to an input. In unrestricted computational systems, Turing showed that no universal function is defined for any possible program. The proposed study investigates the activity of all the hypotheses taken over definite topological spaces.

5. Fixed Point

A point $x^* \in X$is called a fixed point of the function $f: X \to X$if

$$f(x^*) = x^*$$

A stable computational state is a fixed point; executing the algorithm does not change the state anymore. Therefore, fixed points are viewed as halting states.

6. Compact Space

A topological space $(X, \tau)$is said to be compact if every open cover of $X$has a finite subcover.

Due to the ineffectiveness of examine countless computational paths of programs, "compactness" is used to limit the uncontrolled divergence of computational sequences. Compactness refers to a property of topological spaces whose main influence is to regularize local behaviours.

7. Convergence of Computational Sequences

A computational sequence $\{x_n\}$converges to a point $x^* \in X$if

$$\lim_{n\to\infty} x_n = x^*$$

The system's computational states eventually stabilize system-wide, which is known as stability. Under the proposed framework, convergence entails termination behavior.

## IV. ANALYSIS OF THE STUDY

Lemma 1: Existence of Computational State Sequence

Statement: Let $(X, \tau)$be a definite topological space, where $X$represents the set of all possible computational states and $\tau$represents the topology on $X$. Let $f: X \to X$be a continuous mapping representing the execution rule of a program. Then, for every initial computational state $x_0 \in X$, there exists a well-defined sequence $\{x_n\}_{n=0}^{\infty}$such that

$$x_{n+1} = f(x_n), n \geq 0.$$

*Solution:*

Since $x_0 \in X$, the program begins from a valid initial computational state. The function $f: X \to X$means that every element of $X$is mapped back into $X$. Therefore, when the function $f$is applied to $x_0$, we obtain

$$x_1 = f(x_0).$$

Because $f$maps $X$into itself, $x_1 \in X$. Again, applying $f$to $x_1$, we get

$$x_2 = f(x_1).$$

Similarly, $x_2 \in X$. Continuing this process repeatedly, we obtain

$$x_3 = f(x_2), x_4 = f(x_3), \ldots$$

Thus, by induction, for every natural number $n$, the state $x_n$belongs to $X$. Hence, the sequence

$$x_0, x_1, x_2, \ldots, x_n, \ldots$$

is well defined in the topological space $X$. This sequence represents the complete computational path generated by the program from the initial state $x_0$. Therefore, every program execution over a definite topological space can be represented as an iterative sequence of computational states.

Lemma 2: Termination Implies the Existence of a Stable State

Statement: Let $f: X \to X$be a continuous program execution function over a definite topological space $X$. If a program terminates after a finite number of steps, then there exists a state $x^* \in X$such that

$$f(x^*) = x^*.$$

*Solution:*

Assume that the program terminates after a finite number of computational steps. This means that there exists a natural number $N$such that the program reaches a final state $x_N$. After reaching this final state, the program does not produce any further change in its computational configuration.

Since the program has stopped at $x_N$, applying the execution function $f$again should not move the computation to a new state. Therefore,

$$f(x_N) = x_N.$$

Now, let

$$x^* = x_N.$$

Then we obtain

$$f(x^*) = x^*.$$

This means that $x^*$is a fixed point of the function $f$. In computational language, this fixed point represents a stable state or a halting state. The program cannot

move beyond this state because the execution process has already completed.

Hence, every terminating computational process has at least one stable final state in the definite topological space.

Lemma 3: Fixed Point Represents a Halting State

Statement: Let $f: X \to X$ be a continuous mapping representing program execution over a definite topological space $X$. If there exists $x^* \in X$ such that

$$f(x^*) = x^*,$$

then $x^*$ represents a halting state of the program.

*Solution:*

A fixed point of a function is a point that remains unchanged after the function is applied. Therefore, if

$$f(x^*) = x^*,$$

then the program state does not change after reaching $x^*$. This means that the execution rule of the program produces no new computational state from $x^*$.

A program is considered always running unless something in the world repeats in itself. In more technical terms, a program will only terminate if each of its computations changes the state of the world or if it indicates that we should do something further in the world. In the context of systems theory, the stage could define a whole behaviour, like a map will repeat itself if it maps to itself. Put another way.

Thus, $x^*$ behaves like a terminal state. It indicates that the program has completed its execution and has no further transition to perform. Therefore, the fixed point $x^*$ may be interpreted as a halting state.

Termination and fixed points are two important concepts in computer science which appear to be unrelated. However, the following lemma creates a direct link between both.

Lemma 4: Compactness Restricts Infinite Divergence

Statement: Let $(X, \tau)$ be a compact definite topological space. Let $\{x_n\}$ be a computational sequence generated by

$$x_{n+1} = f(x_n).$$

Then the sequence $\{x_n\} \subseteq X$ cannot diverge outside $X$, and it has at least one accumulation point in $X$.

*Solution:*

Since $X$ is compact, every open cover of $X$ has a finite subcover. Compactness is an important property because it prevents elements of a sequence from escaping infinitely far away from the space.

The computational sequence

$$\{x_n\} = x_0, x_1, x_2, \ldots$$

is completely contained in $X$. Since $f: X \to X$, every next state remains inside the same space. Therefore, the computational path cannot leave $X$.

Now, because $X$ is compact, an infinite sequence in $X$ must have a cluster point or accumulation point under standard compactness assumptions such as sequential compactness in metric spaces. This means that even if the program does not halt immediately, its states must repeatedly approach some region of the space.

Hence, the sequence may show one of the following behaviors:

1. It reaches a fixed point and terminates.
2. It enters a repeated cycle.
3. It approaches a limiting accumulation point.

Compactness prevents uncontrolled divergence in all cases. For this purpose termination of programs is studied with the help of compactness because it limits the computation behaviour to the mathematical structure.

Thus, a compact definite topological space provides a suitable environment for termination analysis.

Lemma 5: Decidability of Termination over Finite Topological Spaces

Statement: Let $X$ be a finite definite topological space and let $f: X \to X$ be a program execution function. Then termination checking is decidable over $X$.

*Solution:*

Since $X$ is finite, it contains only a finite number of computational states. Suppose

$$X = \{x_1, x_2, x_3, \ldots, x_m\}.$$

Starting from any initial state $x_0 \in X$, the program generates the sequence

$$x_0, x_1, x_2, \ldots$$

where

$$x_{n+1} = f(x_n).$$

Because $X$ has only $m$ elements, the computational sequence cannot produce infinitely many distinct

states. Therefore, after at most $m + 1$steps, one of the following must happen:

1. The sequence reaches a halting state $x^*$such that $f(x^*) = x^*$.
2. The sequence repeats a previously visited state without reaching a halting state.

In the first case, the program exits. In the second case, the program enters a loop and runs forever without terminating.

Because both cases can be checked in a finite number of steps, an algorithm can tell whether the program halts or does not halt. The algorithm just remembers all the previous states it visited and checks for fixed-point or repetition.

The importance of this lemma lies in the fact that it allows us to conclude that while the Turing machine's universal halting problem is undecidable still halting behaviour in limited and finite mathematical domains is definitely decidable.

## V. CONCLUSION

It is a well-known fact that the halting problem is among the most challenging and significant theoretical computer science problems. Additionally, it states that no algorithm can determine if any program will halt or run forever. Additionally, these questions are what allow us to investigate classical computability theory. According to the halting problem, there can be no computer program that gives the answer when any possible program and its input are entered. Some processes do not terminate at all while some programs never stop running. In the same manner, an algorithm cannot stipulate which one of these alternatives it will be for any possible program.

This means examining the definite topological spaces to study halting problem mathematically. When computing becomes better organized, and more mathematical in nature, then one can study termination behaviour with significance and in a systematic way. Additionally, in this article, computational processes are interpreted as continuous functions over topological spaces. In addition, it introduces the crucial concepts of topological characterizations of extensive and exact maps. In addition, it has adopted essential topological concepts such as continuity, compactness, convergence, fixed points and has applied them to the analysis of programs. In addition, the study has made available powerful mathematical techniques for reasoning about computing processes.

According to the paper, the internal states of an algorithm are points in some mathematical space. In addition, the mapping of the program describes the movement of points in that space from computation step to computation step. Also, the state-transform.

Compact topological spaces act to contain uncontrolled divergence and supply bounds on the trajectories and structure for modeling computation. The study also shows that there is no unrestricted Halting Function. This is stronger than the standard Halting Function non-existence proof because the states aren't restricted. The existence of an unrestricted halting function for the entire unrestricted domain is impossible, although restricted halting functions in definite topological spaces can exist. In addition, Termination checking is decidable in finite topological spaces because the computation will have a finite number of states. It eliminates the chance of overwhelmingly lengthy unprovoked conduct and restricts the divergence. In addition, well-established mathematical methods can be used to analyze the computation process

More importantly, the framework connects topology with mathematical logic and computation theory. Most importantly, we can have a geometric interpretation of halting problem instead of treating it merely as a symbolic and logical paradox. The outcome resulting from this method proved beneficial in several areas. Naturally, it would concern the correctness of a program with respect to its specification. Automated reasoning and compiler optimization of iterative control systems too. One practical application of this approach is reliable software system design. To conclude, the framework suggests opportunities for future research. The concepts presented in the work have potential applications in probabilistic computation, dynamical system theory, quantum computing, and topological frameworks of higher dimensions. Computational geometry employs advanced techniques from algebraic topology and category theory.

Topological structures and other mathematical restrictions often provide the setting in which one does meaningful mathematics. Thus, we feel that our

approach is in the right direction and that in time we can develop a computability model which can be proven mathematically. A new way to look at the halting paradox.

REFERENCES


[1] M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," Proceedings of the London Mathematical Society, vol. 42, no. 2, pp. 230–265, 1937. DOI: 10.1112/plms/s2-42.1.230.

[2] Church, "An Unsolvable Problem of Elementary Number Theory," American Journal of Mathematics, vol. 58, no. 2, pp. 345–363, 1936. DOI: 10.2307/2371045.

[3] S. C. Kleene, Introduction to Metamathematics, Amsterdam, Netherlands: North-Holland Publishing Company, 1952.

[4] K. Gödel, "On Formally Undecidable Propositions of Principia Mathematica and Related Systems," Monatshefte für Mathematik und Physik, vol. 38, pp. 173–198, 1931. DOI: 10.1007/BF01700692.

[5] D. Scott, "Domains for Denotational Semantics," International Colloquium on Automata, Languages and Programming, pp. 577–613, 1982. DOI: 10.1007/3-540-11576-5_83.

[6] M. Sipser, Introduction to the Theory of Computation, 3rd ed., Boston, MA, USA: Cengage Learning, 2012.

[7] J. C. Mitchell, Foundations for Programming Languages, Cambridge, MA, USA: MIT Press, 1996.

[8] M. Huth and M. Ryan, Logic in Computer Science: Modelling and Reasoning about Systems, 2nd ed., Cambridge University Press, 2004. DOI: 10.1017/CBO9780511810275.

[9] Hatcher, Algebraic Topology, Cambridge University Press, 2005. DOI: 10.1017/CBO9780511816949.

[10] S. Mac Lane, Categories for the Working Mathematician, 2nd ed., Springer, 2010. DOI: 10.1007/978-1-4757-4721-8.

[11] J. L. Kelley, General Topology, Springer, 2017. DOI: 10.1007/978-1-4612-6046-3.

[12] E. W. Dijkstra, A Discipline of Programming, Prentice Hall, 2004 edition.

[13] H. Rogers Jr., Theory of Recursive Functions and Effective Computability, MIT Press, 1987.

[14] P. T. Johnstone, Stone Spaces, Cambridge University Press, 2009. DOI: 10.1017/CBO9780511526268.

[15] L. Peralta, "Computable Domains of a Halting Function," arXiv preprint arXiv:2404.09484, 2024. DOI: 10.48550/arXiv.2404.09484.

[16] J. D. Hamkins and T. Nenu, "Did Turing Prove the Undecidability of the Halting Problem?" arXiv preprint arXiv:2407.00680, 2024. DOI: 10.48550/arXiv.2407.00680.

[17] O. Shevchenko and M. Yampolsky, "Computability of Brjuno-like Functions," arXiv preprint arXiv:2501.04195, 2025. DOI: 10.48550/arXiv.2501.04195.

[18] P. Cattabriga, "A Note on Turing's 1936," arXiv preprint arXiv:1308.0497, 2013. DOI: 10.48550/arXiv.1308.0497.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed., MIT Press, 2009.