Emergent Self-Modification and Meta-Programming in Dynamic Systems

Brent Hartshorn: July 7th, 2025 brenthartshorn@proton.me

Abstract

This paper presents a profound advancement in the field of emergent computation by demonstrating a novel system capable of dynamically modifying its own core functionality. Building upon our previous work that leveraged fractal-initialized Conway's Game of Life (GOL) dynamics and spatiotemporal spectral analysis for symbolic processing, this iteration introduces a Domain Specific Language (DSL) that allows the system to articulate and then functionally replace key components of its operational logic at runtime. Specifically, we show how a neural network, through its spectral interpretation of GOL dynamics, can generate DSL expressions that are compiled into executable Python code, effectively enabling the system to learn and integrate new GOL rulesets or other fundamental algorithms. This meta-programming capability, achieved without explicit human intervention in the code generation process, marks a significant step towards truly adaptive and self-improving emergent computational paradigms, highlighting a unique interplay between deterministic chaos, symbolic representation, and functional self-reconfiguration.

1. Introduction

The quest for artificial intelligence that can adapt, learn, and even self-modify its own underlying mechanisms remains a frontier challenge. Traditional machine learning models, while excelling at pattern recognition and prediction, typically operate within a fixed architectural and functional framework. True intelligence, however, often implies the capacity to understand, manipulate, and generate the very rules that govern its operations.

Our prior research introduced a hybrid computational model where the emergent spatiotemporal dynamics of a Conway's Game of Life (GOL) system, initialized by a fractal, were transformed via 3D Fast Fourier Transform (FFT) into spectral features. These features were then interpreted by a neural network classifier to perform symbolic tasks. The **"Emergent Symbolic Computation Through Fractal Dynamics and Spatiotemporal Spectral Analysis: A Linguistic Extension" [11]** paper specifically demonstrated the system's ability to map linguistic inputs to symbolic outputs and generate Python code snippets based on learned associations.

This paper pushes the boundary further. We introduce a designed Domain Specific Language (DSL) that serves as a high-level, learnable interface for the neural network to express computational logic. Crucially, we demonstrate that the system can learn to output DSL expressions that define a Game of Life stepping function itself – a core component of its own predictive engine, currently represented by game_of_life. This DSL is then interpreted, compiled into Python, and dynamically executed, allowing the emergent computational framework to reconfigure its fundamental behaviors at runtime. This capability moves beyond mere symbolic output to true *emergent self-modification* and *meta-programming*, enabling the system to "rewrite" parts of its own core.

2. The Domain Specific Language (DSL) for Core Functionality

The design of the DSL is paramount to enabling the network's self-modification capabilities. It is crafted to represent fundamental algorithmic constructs (loops, conditionals, assignments, arithmetic operations) in a highly simplified and tokenized form, making it amenable to the network's learning process.

2.1. DSL Structure and Abstraction

The DSL leverages a hierarchical, chained syntax (e.g., `write.python.function.to.compute.game_of_life.ARGS.grid`). Key features include:

- **Ċoncept Class**: This Python class serves as the parser and abstract syntax tree (AST) builder for the DSL. Each node in the DSL expression (e.g., write, python, function, iterate, equals) is represented by a Ċoncept object. Chaining these objects via attribute access (__getattr__) or method calls (__call__, __getitem__) implicitly defines the relationships and structure of the intended Python code. Context managers (with Ċoncept(...) as ...) further assist in defining code blocks for proper indentation.
- **High-Level Primitives**: The DSL replaces verbose Python syntax with concise, semantically rich tokens. For example, iterate.u.IN.range.rows encapsulates a for u in range(rows): loop, while plus_equals represents the += operator, and equals defines == within conditionals and = for assignments. It supports constructs for defining function arguments (ARGS), array access (g[x,y]), and conditional logic (if, elif, then, end).
- **Implicit Scoping and Indentation**: The Ċoncept class's as_python() method, in conjunction with the DSL function and dsl_to_python() method, handles the conversion of nested DSL statements into correctly indented and syntactically valid Python code. This mechanism relieves the network from learning explicit indentation rules.
- **Core Function Definition**: Critically, the DSL includes constructs specifically designed to define Python functions (write python function), including arguments (ARGS) and the function body. This allows the network to specify the signature and logic of functions like game_of_life, as demonstrated by conway_oneliner.

DSL Example Input

```
write python function to compute game_of_life
args g; r equals numpy copy g; rows comma cols equals g shape; iterate u in range rows iterate v in
range cols; n equals zero; iterate i in range(-1,2) iterate j in range(-1,2); if i equals zero and j
equals zero then continue; a equals u plus i; x equals a percent rows; b equals v plus j; y equals b
percent cols; n plus_equals g[x,y]; end; if g[u,v] equals one if n less_than[2] or n greater_than[3]
then r[u,v] equals zero; elif n equals three then r[u,v] equals one; end; return r
```

DSL Meta Translation

```
with Ċoncept("dynamic") as write:
        write.python.function.to.compute.game_of_life
        game_of_life.ARGS.g
        game_of_life.r.equals.numpy.copy.g
        game_of_life.rows.comma.cols.equals.g.shape
        with game_of_life.iterate.u.IN.range.rows.iterate.v.IN.range.cols as loop_1:
                loop_1.n.equals.zero
                with loop_1.iterate.i.IN.range(-1,2).iterate.j.IN.range(-1,2) as loop_2:
                        loop_2.IF.i.equals.zero.AND.j.equals.zero.THEN.CONTINUE
                        loop_2.a.equals.u.plus.i
                        loop_2.x.equals.a.percent.rows
                        loop_2.b.equals.v.plus.j
                        loop_2.y.equals.b.percent.cols
                        loop_2.n.plus_equals.g[x,y]
                loop_1.IF.g[u,v].equals.one.IF.n.less_than[2].OR.n.greater_than[3].THEN.r[u,v].equals.zero
```

DSL Meta to Python Translation

```
def game_of_life(g):
        r = numpy.copy(g)
        rows , cols = g.shape
        for u in range(rows):
                for v in range(cols):
                        n = 0
                        for i in range(-1,2):
                                for j in range(-1,2):
                                        if i == 0 and j == 0:
                                                continue
                                        a = u + i
                                        x = a % rows
                                        b = v + j
                                        y = b % cols
                                        n += g[(x,y)]
                        if g[(u,v)] == 1:
                                if n < 2 or n > 3:
                                        r[(u,v)] = 0
                        elif n == 3:
                                r[(u,v)] = 1
```

2.2. Tokenization for Network Learning

To enable the neural network to "speak" this DSL, a sophisticated tokenization scheme is employed:

- **Expanded OUT_TOKENS Vocabulary**: The tokenize_output function dynamically populates OUT_TOKENS with unique symbols (from OUT_SYMS) for each distinct part of the DSL string (e.g., write , python, function, to, compute, game_of_life, args, g;, r, equals, numpy, copy, g; etc.). This creates a discrete, learnable vocabulary for the network.
- **Symbol-to-Code Mapping (SYMBOLS_TO_CODE)**: The learn_dsl function takes the full DSL string, tokenizes it, and then maps the *joined tokenized string* to a DSL("""\n%s\n""") call. This is the critical step where the network's symbolic output directly translates into a command to generate and execute Python code.
- **Single-Neuron Mapping**: Each unique token in the TextProcessor.CHARACTERS (which dynamically includes OUT_TOKENS) is mapped to a unique output neuron in the classifier's final layer.
- **Positional Encoding and Reconstruction**: The TextProcessor.word_to_target_values method converts target DSL phrases (e.g., the conway_oneliner's tokenized form) into a numerical vector, where values indicate positional importance. Conversely, TextProcessor.reconstruct_word_from_classifier_output decodes the classifier's output back into a DSL expression by sorting activated tokens.

3. Emergent Self-Modification Mechanism

The self-modification capability is realized through a dynamic execution pipeline that bridges the symbolic output of the network with the functional replacement of the system's own code.

3.1. DSL Prediction and Compilation

1. **Input to Spectral Features**: An input phrase (e.g., "write python function to compute game_of_life") is encoded into the initial GOL grid using TextProcessor.apply_character_to_gol_grid. The subsequent spatiotemporal evolution of this GOL process over GOL_SIM_STEPS is captured.
2. **Spectral Analysis**: A 3D FFT is applied to the GOL history, and NUM_FFT_BANDS_FOR_CLASSIFIER spectral energy bands are extracted using calculate_3d_fft_energy_bands_torch.
3. **DSL Prediction**: These spectral features are fed into the ClassifierNN, which has been trained to output a numerical vector. This vector is then interpreted by TextProcessor.reconstruct_word_from_classifier_output to yield a predicted tokenized DSL string (e.g.,  ...).

3.2. Dynamic Code Generation and Replacement

The true meta-programming occurs in the following steps:

1. **DSL-to-Python Translation**: The predicted tokenized DSL string is looked up in the SYMBOLS_TO_CODE dictionary. If a match is found, the associated Python string (e.g., DSL("""\nwrite python function to compute game_of_life\n...\n""")) is retrieved.

2. **Runtime Code Injection**: This retrieved Python string, containing the call to DSL(...), is then executed within the system's global namespace using Python's exec() function.

    - The DSL() function parses the original DSL string (passed as an argument).
    - It uses the Ċoncept class to construct an Abstract Syntax Tree (AST).
    - It then calls write.python() (which internally uses Ċoncept.as_python()) to generate the full Python source code for the function (e.g., def game_of_life(g):\n ...).
    - Finally, it exec()utes this generated Python function string itself, dynamically defining or redefining the game_of_life function (or other target functions) within the running environment.

3. **Functional Replacement**: Subsequent calls within the system that reference game_of_life will now automatically use the newly defined function. This mechanism effectively "rewrites" that part of the system's core behavior at runtime. This can be directly observed through the solve function's --exec-in-training flag, which allows for live execution of the generated code within the training loop.

4. Experimental Setup and Results

The training objective is to map various linguistic descriptions or symbolic cues to the tokenized DSL representation of a game_of_life function. The conway_oneliner provides the initial target for the network to learn, explicitly defining the GOL rules.

- **Hybrid Optimization**: The system retains its two-tiered optimization strategy: a gradient-free mutation and selection process for the non-differentiable fractal parameters (best_bs_params adjusted by MUTATION_RATE), coupled with traditional gradient-based learning for the ClassifierNN using torch.optim.Adam and torch.nn.MSELoss. The --exec-in-training flag allows for direct evaluation of the generated code's functional correctness during training, potentially opening avenues for performance-based feedback in future iterations.
- **Training Data (LEARN, word_mapping_data)**: The LEARN list is populated by learn_dsl(conway_oneliner), providing the input ("write python function to compute game_of_life") and its tokenized DSL target. Additional mappings for genfunc (🖨) and gencall (📞) demonstrate broader code generation capabilities.
- **Evaluation**: Beyond simple classifier loss, success is measured by the system's ability to: 1. Accurately predict the full tokenized DSL expression for the game_of_life function. 2. Successfully compile this DSL into executable Python code without errors. 3. Demonstrate that the dynamically replaced game_of_life function behaves as expected (e.g., correctly computes the next GOL state in run_gol_and_3d_fft_torch). The exact word matches (correct_words_exact_match) and the execution of SYMBOLS_TO_CODE[predicted_word] within the solve function serve as direct functional validation points.

Preliminary results demonstrate that the system can successfully learn to generate the conway_oneliner DSL, which then translates into a functional Python game_of_life implementation. This showcases a proof-of-concept for self-modification of an underlying core algorithm.

5. Discussion: Implications for Self-Adaptive Systems

This work opens several exciting avenues for future research and development:

- **Adaptive Rule Learning**: The system's ability to self-modify its GOL rules suggests a pathway for learning optimal cellular automata rules for specific emergent behaviors, going beyond fixed, pre-programmed rules. This could involve the system iteratively proposing, evaluating, and refining new GOL rule DSLs based on desired emergent patterns.
- **Beyond GOL**: The flexible Ċoncept and DSL framework, coupled with SYMBOLS_TO_GEN (which currently includes genfunc for lambda functions and gencall for function calls), is extensible. It could be used to learn and dynamically inject other fundamental algorithms or even define adaptive network architectures, leading to self-optimizing AI where the system generates its own learning rules.
- **Emergent Program Synthesis**: This paradigm offers a unique approach to program synthesis, where the neural network doesn't just generate code, but generates code that fundamentally alters its own operational logic based on experiential learning (through the fractal dynamics and GOL evolution).
- **Interpretability**: By generating high-level DSL, the system offers a degree of interpretability into *what* functionality it is modifying, as opposed to opaque weight changes in traditional neural networks. The generated Python code is human-readable, facilitating analysis.

Source Code

```
import random,math,sys,torch,time,string,inspect,select,numpy; device = torch.device("cuda" if torch.cuda.is_available() else "cpu"); print(f"Using device: {device}")
# --- Constants ---
MIN_VAL_THRESH = 0.25; GOL_GRID_SIZE = 12; GOL_SIM_STEPS = 8; NUM_FFT_BANDS_FOR_CLASSIFIER = 128; OPTIMIZATION_ITERATIONS = 2000; CLASSIFIER_RATE = 1
MAX_OUTPUT_WORD_LENGTH = 40; MIN_LOSS = 0.00005; MUTATION_RATE = 0.025

conway_oneliner = '''
write python function to compute game_of_life
args g; r equals numpy copy g; rows comma cols equals g shape; iterate u in range rows iterate v in range cols; n equals zero; iterate i in range(-1,2) iterate j in range(-1,2); if i equals zero and j equals zero then continue; a equals
u plus i; x equals a percent rows; b equals v plus j; y equals b percent cols; n plus_equals g[x,y]; end; if g[u,v] equals one if n less_than[2] or n greater_than[3] then r[u,v] equals zero; elif n equals three then r[u,v] equals one;
end; return r
'''

SYMBOLS_TO_CODE = {'  ' : 'print("hello world")'}; OUT_TOKENS = []; OUT_SYMS_TO_WORDS = {}; OUT_WORDS_TO_SYMS = {}; OUT_SYMS = [chr(_) for _ in range(1024,1024+128)]
def tokenize_output(txt):
    tok_syms = []
    for part in txt.strip().split(';'):
        part += ';'
        if part in OUT_WORDS_TO_SYMS: part += ';'
        assert part not in OUT_WORDS_TO_SYMS
        sym = OUT_SYMS.pop(); OUT_SYMS_TO_WORDS[sym]=part; OUT_WORDS_TO_SYMS[part]=sym; OUT_TOKENS.append(sym); tok_syms.append(sym)
    print(len(OUT_TOKENS),OUT_TOKENS, OUT_SYMS_TO_WORDS)
    return tok_syms
LEARN = []
def learn_dsl(txt):
    toks = tokenize_output(txt); a,b = txt.strip().splitlines();
    toks_string = ''.join(toks)
    SYMBOLS_TO_CODE[toks_string] = 'DSL("""\n%s\n""")' % txt
    LEARN.append((a,toks_string))
    print(LEARN)
learn_dsl(conway_oneliner)

class Ċoncept:
    IDENT = 0
    def __init__(self,*args,**kwargs):
        self.ñame=args[-1]; self.links = []; self.ḟelations = []; self.þrops = []; self.children=[]
        if len(args)>1: self.ḟelations = list(args[:-1])
    def __call__(self,*args):
        if not args: return Ċoncept.dsl_to_python(self.as_python())
        for a in args: self.links.append(a)
        return self
    def __enter__(self): return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type: raise exc_type(exc_val).with_traceback(exc_tb)
    def __getitem__(self,o):
        if o not in self.ḟelations:self.ḟelations.append(o)
        return self
    def __getattr__(self, n): o = Ċoncept(self,n); setattr(self,n,o); globals()[n]=o; self.children.append(o); return o
    def as_python(self):
        o = []
        if self.ñame == 'function':
            f = self.children[0].children[0].children[0]; c = 'def %s' % f.ñame
            if f.children[0].ñame=='ARGS': c += '(%s):\n' % ','.join([_.ñame for _ in f.children[0].children ])
            else: c += '():\n'
            o.append(c)
            for v in f.children[1:]:
                for a in v.as_python(): o.append('\t'+a)
            return o
        elif self.ñame == 'iterate':
            Concept.IDENT += 1; var = self.children[0]; IN = var.children[0]; assert IN.ñame=='IN'; b = []
            if IN.children[0].ñame=='range':
                if IN.children[0].links:
                    x = []
                    for v in IN.children[0].links:
                        if type(v) is Concept: x.append(v.ñame)
                        else: x.append(str(v))
                    x = ','.join(x); a = 'range(%s)' % x
                    for c in IN.children[0].children: b.append('\t'+ ('\t'*Ċoncept.IDENT) + c.as_python()[0])
                elif IN.children[0].children:
                    a = 'range(%s)' % IN.children[0].children[0].ñame
                    for c in IN.children[0].children[0].children: b.append('\t'+ ('\t'*Ċoncept.IDENT) + c.as_python()[0])
                else: a = 'range(%s)' % IN.children[0].as_python()
            b = '\n'.join(b); Concept.IDENT -= 1; return ['for %s in %s:\n%s' %(var.ñame,a,b)]
        elif self.ñame == 'copy' and self.ḟelations and self.ḟelations[0].ñame=='numpy':
            if self.children: a = self.children[0].as_python(); assert len(a)==1; a = a[-1]
            else: a = self.links[-1].ñame + '\n'
            if a.endswith('\n'): return ['copy(%s)\n' % a[:-1]]
            else: return ['copy(%s)' % a]
        if len(self.ḟelations) > 1:
            if type(self.ḟelations[-1]) is tuple and type(self.ḟelations[-1][0]) is Ċoncept: a,b = self.ḟelations[-1]; o.append('%s[(%s,%s)]' % (self.ñame, a.ñame, b.ñame))
            else:
                if self.ñame == 'less_than': o.append('< %s' % self.ḟelations[-1])
                elif self.ñame == 'greater_than': o.append('> %s' % self.ḟelations[-1])
                else: o.append('%s[%s]' % (self.ñame, self.ḟelations[-1]))
        else:
            mapping = {'comma':',', 'equals':'=', 'plus':'+', 'plus_equals':'+=', 'percent':'%', 'less_than':'< ', 'greater_than':'> ', 'CONTINUE':'continue', 'zero':'0', 'one':'1', 'two':'2', 'three':'3'}
            if self.ñame in mapping: o.append(mapping[self.ñame])
            else: o.append(self.ñame)
        for v in self.children: o += v.as_python()
        o = ','.join(o)
        if not self.children: o += '\n'
        return [o]
    @staticmethod
    def dsl_to_python(dsl):
        o = []; ident = 0; replacers = {'.=.':' = ', '.+.':' + ', '.%.':' % ', '.+=.':' += ', '.,.':' , ', 'python.def': 'def', 'RETURN.':'return ', '.AND.':' and ', '.OR.': ' or ', '.< ':' < ', '.> ':' > '}
        for ln in dsl[-1].splitlines():
            for r in replacers: ln = ln.replace(r, replacers[r])
            if ln.startswith('.'): ln = ln[1:]
            if ln.strip().startswith( ('IF.', 'ELIF.') ):
                tabs = ln.count('\t'); tabs += 1; tabs = '\t' * tabs
                if '.IF.' in ln: ln = ln.replace('.IF.', f':\n{tabs}if '); tabs += '\t'
                ln = ln.replace('.THEN.', f':\n{tabs}')
                if ln.strip().startswith( 'ELIF.' ): ln = ln.replace('ELIF.', 'elif ')
                else: ln = ln.replace('IF.', 'if ')
            o.append( ('\t'*ident) + ln )
        o = '\n'.join(o); out = []
        for ln in o.splitlines():
            if not ln.strip(): continue
            if ln.strip().startswith( ('if ', 'elif ') ): ln = ln.replace(' = ', ' == ')
            if ln.strip().startswith('if') and ':' not in ln and ln.count('.'): ln = ln.replace('.', ': ') ## in this simple case, the fix is easy
            out.append(ln)
        return '\n'.join(out)

def DSL(txt, execute=True, show=True):
    txt = txt.strip(); prefixes = {0:txt.splitlines()[0].split()[-1]}; o = ['with Ċoncept("dynamic") as write:']
    lines = 0; ident = 0; rep = {k.lower():k.upper() for k in 'ARGS IN AS OR AND THEN CONTINUE RETURN END IF ELSE ELIF'.split()}
    for ln in txt.splitlines():
        print(ident, ln)
        for part in ln.split(';'):
            p = part.strip(); part = []
            for word in p.split():
                if word in rep: word = rep[word]
                part.append(word)
            part = '.'.join(part)
            if part.startswith('iterate.'):
                prefixes[ident+1] = _ = 'loop_%s' % len(prefixes)
                part = 'with %s.%s as %s:' % (prefixes[ident],part,_)
            elif part=='END': ident -= 1; continue
            if lines==0: o.append( '\t' + ('\t'*ident) +part)
            elif part.startswith('with '): o.append( '\t' + ('\t'*ident) + part); ident += 1
            else:
                o.append( '\t' + ('\t'*ident) + prefixes[ident] + '.' + part)
                if p.startswith('iterate '): ident += 1
            lines += 1
    before = {}; before.update(globals()); meta = '\n'.join(o); print('DSL:', meta); exec(meta, globals()); py = write.python()
    if show: print('PYTHON:', py)
    rem = []
    for n in globals():
        v = globals()[n]
        if type(v) is Ċoncept:
            if n not in before: rem.append(n)
            elif n in before and type(before[n]) is not Ċoncept: globals()[n] = before[n]
    for n in rem: globals().pop(n)
    if execute:
        exec(py, globals()); fname = py.splitlines()[0].split('def ')[-1].split('(')[0]; func = globals()[fname]; func.meta_code = txt; func.source_code = py
        return func
    else: return py

DSL(conway_oneliner)
print(game_of_life)

class TextProcessor:

    CHARACTERS = [chr(i) for i in range(ord('a'), ord('z') + 1)] + [' ', '  ', '', '', '', ''] + OUT_TOKENS; CHAR_TO_INDEX = {char: i for i, char in enumerate(CHARACTERS)}
    INDEX_TO_CHAR = {i: char for i, char in enumerate(CHARACTERS)}; NUM_ASCII_CHARS = len(CHARACTERS)
    ASCII_LETTERS = {
        'a': [" # ", "# #", "###", "# #", "# #"],
        'b': ["## ", "# #", "## ", "# #", "## "], 'c': ["###", "#  ", "#  ", "#  ", "###"], 'd': ["## ", "# #", "# #", "# #", "## "], 'e': ["###", "#  ", "## ", "#  ", "###"],
        'f': ["###", "#  ", "## ", "#  ", "  "], 'g': ["###", "#  ", "# #", "# #", "###"], 'h': ["# #", "# #", "###", "# #", "# #"], 'i': ["###", " # ", " # ", " # ", "###"],
        'j': ["  #", "  #", "  #", "# #", " # "], 'k': ["# #", "# #", "## ", "# #", "# #"], 'l': ["#  ", "#  ", "#  ", "#  ", "###"], 'm': ["# #", "###", "# #", "# #", "# #"],
        'n': ["# #", "###", "# #", "# #", "# #"], 'o': [" # ", "# #", "# #", "# #", " # "], 'p': ["## ", "# #", "## ", "#  ", "#  "], 'q': [" # ", "# #", "# #", "## ", "  #"],
        'r': ["## ", "# #", "## ", "# #", "# #"], 's': ["###", "#  ", " # ", "  #", "###"], 't': ["###", " # ", " # ", " # ", " # "], 'u': ["# #", "# #", "# #", "# #", " # "],
        'v': ["# #", "# #", " # ", " # ", " # "], 'w': ["# #", "# #", "# #", "# #", " # #"], 'x': ["# #", " # ", " # ", " # ", "# #"], 'y': ["# #", " # ", " # ", " # ", " # "],
        'z': ["###", "  #", " # ", "#  ", "###"], ' ': ["#####"]*5, '_': ['     ', '     ', '     ', '     ', '###'], '3': ['###',' #',' ##',' #','###'],
```

```
    }
    PATTERN_HEIGHT = PATTERN_WIDTH = 5
    @staticmethod
    def ascii_to_gol_pattern(char):
        char = char.lower() # Ensure lowercase
        if char==' ': char_art_list = random.choice( list(TextProcessor.ASCII_LETTERS.values()) )
        else: char_art_list = TextProcessor.ASCII_LETTERS[char]
        pattern = numpy.zeros((TextProcessor.PATTERN_HEIGHT, TextProcessor.PATTERN_WIDTH), dtype=int)
        for r, row_str in enumerate(char_art_list):
            if r >= TextProcessor.PATTERN_HEIGHT: continue
            for c, symbol in enumerate(row_str):
                if c >= TextProcessor.PATTERN_WIDTH: continue
                if symbol != ' ': pattern[r, c] = 1
        return pattern
    @staticmethod
    def apply_character_to_gol_grid(main_gol_grid, char_pattern, start_row, start_col):
        p_h, p_w = char_pattern.shape; g_h, g_w = main_gol_grid.shape
        for r_p in range(p_h):
            for c_p in range(p_w):
                if char_pattern[r_p, c_p] == 1: # Only apply if the pattern cell is 'active'
                    r_g = (start_row + r_p) % g_h; c_g = (start_col + c_p) % g_w # Calculate target cell coordinates, wrapping around grid boundaries
                    main_gol_grid[r_g, c_g] = 1 - main_gol_grid[r_g, c_g]      # Implement the "flipping" logic: If current cell is alive, set to dead (0). If dead, set to alive (1).
                # If char_pattern[r_p, c_p] is 0 (whitespace), main_gol_grid cell is untouched.
    @staticmethod
    def word_to_target_values(word, max_output_chars):
        """Converts a target word into a list of numerical values for the classifier's output. Each value corresponds to a character's "positional importance" for sorting.
        Higher value = appears earlier in the word. The length of the returned list is TextProcessor.NUM_ASCII_CHARS.
        """
        target_output_vector = [0.0] * TextProcessor.NUM_ASCII_CHARS
        word_to_encode = word[:max_output_chars]                # Ensure word is within the maximum output length for value scaling consistency. note: lower() breaks extended latin
        for i, char in enumerate(word_to_encode):
            if char in TextProcessor.CHAR_TO_INDEX:
                char_idx = TextProcessor.CHAR_TO_INDEX[char]
                # Assign a unique value based on position (higher value means earlier in word). Scaling factor (1.0 - i / max_output_chars) makes values distinct and between 0 and 1.
                # A small epsilon is added to make sure no two identical letters have the exact same value, if they appear at positions that would otherwise yield the same scaled
                # value after truncation. This is more for distinctiveness than mathematical necessity with MSE, but good practice.
                value = (1.0 - (i / max_output_chars)) + (i * 0.0001)
                # Assign this value to the corresponding character's slot in the output vector. If a letter appears multiple times, the latest one encountered (earlier in the word)
                # will overwrite previous ones, which is the expected behavior for your "no repeats" rule.
                target_output_vector[char_idx] = value
            # If Characters not in CHAR_TO_INDEX are ignored, their value in the vector remains 0.0
            else: raise RuntimeError('can not encode word: %s char: %s' % (word, char))
        return target_output_vector
    @staticmethod
    def reconstruct_word_from_classifier_output(output_values_tensor, min_value_threshold=0.01):
        """Reconstructs a word from the classifier's output values by sorting them. output_values_tensor: A 1D PyTorch tensor of values from the classifier,
         where each index corresponds to a character. min_value_threshold: Minimum value for a character to be considered part of the word.
        """
        char_value_pairs = []; output_values_np = output_values_tensor.cpu().numpy()
        for idx, value in enumerate(output_values_np):
            if value > min_value_threshold and idx in TextProcessor.INDEX_TO_CHAR: char_value_pairs.append((value, TextProcessor.INDEX_TO_CHAR[idx]))
        # Sort by value in descending order (higher value means earlier in word)
        char_value_pairs.sort(key=lambda x: x[0], reverse=True); reconstructed_word = "".join([char for value, char in char_value_pairs]); return reconstructed_word

def print_gol_grid(grid, x=90):
    out = []; out.append("+" + "-" * (grid.shape[1]) + "+")
    for row in grid: out.append("|" + "".join(["[redacted]" if cell == 1 else " " for cell in row]) + "|")
    out.append("+" + "-" * (grid.shape[1]) + "+"); printat('\n'.join(out), 4, x)

def generate_mandelbrot_grid_np(grid_size, center_x, center_y, zoom, max_iter=100):
    """ Generates a Mandelbrot fractal grid."""
    grid = numpy.zeros((grid_size, grid_size), dtype=float)
    scale_x = 3.0 / zoom / grid_size; scale_y = 3.0 / zoom / grid_size
    for row in range(grid_size):
        for col in range(grid_size):
            cx = center_x + (col - grid_size / 2) * scale_x; cy = center_y + (row - grid_size / 2) * scale_y; x, y = 0.0, 0.0
            for i in range(max_iter):
                x_new = x*x - y*y + cx; y_new = 2 * x * y + cy; x = x_new; y = y_new
                if x*x + y*y > 4.0: break
            grid[row, col] = i
    max_val = numpy.max(grid)
    if max_val > 0: grid = grid / max_val # Normalize to [0, 1]
    return grid

def generate_burning_ship_grid_np(grid_size, center_x, center_y, zoom, max_iter=100):
    grid = numpy.zeros((grid_size, grid_size), dtype=float); scale_x = 3.0 / zoom / grid_size; scale_y = 3.0 / zoom / grid_size
    for row in range(grid_size):
        for col in range(grid_size):
            cx = center_x + (col - grid_size / 2) * scale_x; cy = center_y + (row - grid_size / 2) * scale_y; x, y = 0.0, 0.0
            for i in range(max_iter):
                x_new = x*x - y*y + cx; y_new = 2 * abs(x * y) + cy; x = abs(x_new); y = abs(y_new)
                if x*x + y*y > 4.0: break
            grid[row, col] = i
    max_val = numpy.max(grid)
    if max_val > 0: grid = grid / max_val # Normalize to [0, 1]
    return grid

# --- GOL Simulation and 3D FFT (with word input encoding) ---
def run_gol_and_3d_fft_torch(burning_ship_params, input_word_string, debug_output=False):
    center_x, center_y, zoom, fractal_max_iter = burning_ship_params      # Initialize GOL grid from Burning Ship or Mandelbrot fractal. This provides a dynamic "substrate".
    if '--null' in sys.argv: fractal_raw_grid_np = numpy.zeros((GOL_GRID_SIZE,GOL_GRID_SIZE), dtype=float); tag = 'none'
    elif '--mandel' in sys.argv: fractal_raw_grid_np = generate_mandelbrot_grid_np(GOL_GRID_SIZE, center_x, center_y, zoom, fractal_max_iter); tag='Mandelbrot'
    else: fractal_raw_grid_np = generate_burning_ship_grid_np(GOL_GRID_SIZE, center_x, center_y, zoom, fractal_max_iter); tag='Burning Ship'
    current_gol_grid_np = (fractal_raw_grid_np > 0.5).astype(int)          # Convert to binary GOL grid
    if debug_output: print_gol_grid(current_gol_grid_np, x=70)
    gol_process_history = torch.zeros((GOL_SIM_STEPS, GOL_GRID_SIZE, GOL_GRID_SIZE), dtype=torch.float32, device=device)       # Store entire GOL process for 3D FFT
    paste_row_offset = paste_col_offset = 1; total_chars_to_process = len(input_word_string)    # To allow character interactions, let's paste at a fixed point and rely on GOL evolution
    if total_chars_to_process == 0: gol_steps_per_char_input = GOL_SIM_STEPS                    # If empty string, just simulate initial grid
    else:
        gol_steps_per_char_input = GOL_SIM_STEPS // total_chars_to_process
        if gol_steps_per_char_input == 0: gol_steps_per_char_input = 1 # Ensure at least one step per char input
    current_history_idx = 0
    for i, char in enumerate(input_word_string):   # Apply the character pattern to the current GOL grid
        char_pattern = TextProcessor.ascii_to_gol_pattern(char); TextProcessor.apply_character_to_gol_grid(current_gol_grid_np, char_pattern, paste_row_offset, paste_col_offset)
        if debug_output: print(f"\n--- GOL Grid after inputting character '{char}' ---"); print_gol_grid(current_gol_grid_np)
        for step_in_segment in range(gol_steps_per_char_input):        # Simulate GOL for `gol_steps_per_char_input` steps, storing each frame
            if current_history_idx < GOL_SIM_STEPS:
                gol_process_history[current_history_idx, :, :] = torch.from_numpy(current_gol_grid_np).to(device).float()
                current_gol_grid_np = game_of_life(current_gol_grid_np); current_history_idx += 1
            else: break                    # History buffer is full
    while current_history_idx < GOL_SIM_STEPS:    # If there are remaining GOL_SIM_STEPS after processing all characters, continue simulation
        gol_process_history[current_history_idx, :, :] = torch.from_numpy(current_gol_grid_np).to(device).float()
        current_gol_grid_np = game_of_life(current_gol_grid_np); current_history_idx += 1
        if debug_output: print_gol_grid(current_gol_grid_np); time.sleep(0.05)
    # Perform 3D FFT on the entire GOL process history
    fft_result_3d = torch.fft.fftn(gol_process_history); fft_shifted_3d = torch.fft.fftshift(fft_result_3d); magnitude_spectrum_3d = torch.abs(fft_shifted_3d)
    return magnitude_spectrum_3d

# --- Energy Band Calculation from 3D FFT ---
def calculate_3d_fft_energy_bands_torch(magnitude_spectrum_3d, num_bands):
    power_spectrum = magnitude_spectrum_3d**2; flattened_power = power_spectrum.flatten(); band_energies = torch.zeros(num_bands, dtype=torch.float32, device=device)
    segment_size = len(flattened_power) // num_bands
    for i in range(num_bands):
        start_idx = i * segment_size; end_idx = start_idx + segment_size
        if i == num_bands - 1: end_idx = len(flattened_power)                    # Ensure the last band gets all remaining elements
        band_energies[i] = torch.sum(flattened_power[start_idx:end_idx])
    total_energy_bands = torch.sum(band_energies)
    if total_energy_bands > sys.float_info.min: band_energies = band_energies / total_energy_bands
    else: band_energies = torch.full((num_bands,), 1.0 / num_bands, device=device) # Distribute evenly if no energy
    return band_energies

# --- Small Neural Network Classifier ---
class ClassifierNN(torch.nn.Module):
    def __init__(self, num_input_bands, num_hidden_neurons, num_output_neurons):
        super(ClassifierNN, self).__init__()
        self.fc1 = torch.nn.Linear(num_input_bands, num_hidden_neurons)
        self.sigmoid = torch.nn.Sigmoid() # Activation for hidden layer
        self.fc2 = torch.nn.Linear(num_hidden_neurons, num_output_neurons)
        self.output_activation = torch.nn.Sigmoid()      # Use Sigmoid for the final layer to output values between 0 and 1, which matches the target values generated by TextProcessor
    def forward(self, x): x = self.fc1(x); x = self.sigmoid(x); x = self.fc2(x); x = self.output_activation(x); return x

# --- Optimization Loop solve_word_mapping_with_burning_ship_and_classifier_torch ---
def solve(word_mapping_data, classifier_nn=None, fractal_params=None, iterations=OPTIMIZATION_ITERATIONS, verbose=True):
    if classifier_nn is None:
        classifier_nn = ClassifierNN(num_input_bands=NUM_FFT_BANDS_FOR_CLASSIFIER, num_hidden_neurons=TextProcessor.NUM_ASCII_CHARS * 2, num_output_neurons=TextProcessor.NUM_ASCII_CHARS
        ).to(device)
    optimizer = torch.optim.Adam(classifier_nn.parameters(), lr=0.05); criterion = torch.nn.MSELoss()
    if fractal_params: best_bs_params = list(fractal_params)    # Initial Burning Ship parameters
    else: best_bs_params = [random.uniform(-1.0, 0.0), random.uniform(0.0, 1.0), random.uniform(1.0, 5.0), random.randint(50, 150)]; best_overall_loss = float('inf')
    if not word_mapping_data:
        return (best_bs_params, classifier_nn)
    if verbose:
        print(f"Starting optimization for word mapping with Burning Ship and PyTorch Classifier. Iterations: {iterations}")
        print(f"Initial Best Burning Ship Parameters: {best_bs_params}"); start_time = time.time()
    for iteration in range(iterations):
        current_bs_params = list(best_bs_params)
        for i in range(len(current_bs_params)):        # Mutate Burning Ship parameters
            if i < 3: # center_x, center_y, zoom
                current_bs_params[i] += random.uniform(-MUTATION_RATE, MUTATION_RATE) * abs(current_bs_params[i])
                if i == 2: current_bs_params[i] = max(0.1, current_bs_params[i]) # Ensure zoom is positive
            else: # max_iter
                current_bs_params[i] += random.randint( -max(1, int(MUTATION_RATE * current_bs_params[i])), max(1, int(MUTATION_RATE * current_bs_params[i])))
                current_bs_params[i] = max(20, min(200, current_bs_params[i]))   # Keep max_iter within a reasonable range
        current_band_data_list = []; targets_list_for_nn = []
        for input_word, target_word in word_mapping_data:                        # --- Evaluate current Burning Ship parameters ---
            debug_flag = (iteration == 0 and input_word == word_mapping_data[0][0]) and '--debug' in sys.argv
            magnitude_spectrum = run_gol_and_3d_fft_torch(current_bs_params, input_word, debug_output=debug_flag)
            bands = calculate_3d_fft_energy_bands_torch(magnitude_spectrum, NUM_FFT_BANDS_FOR_CLASSIFIER); current_band_data_list.append(bands)         # Calculate FFT energy bands
            target_values = TextProcessor.word_to_target_values(target_word, MAX_OUTPUT_WORD_LENGTH)       # Generate the target vector for the classifier based on the target_word
            targets_list_for_nn.append(torch.tensor(target_values, dtype=torch.float32).to(device))
        bands_batch = torch.stack(current_band_data_list).to(device); targets_batch = torch.stack(targets_list_for_nn).to(device)            # Prepare batch for classifier training
```

```
        classifier_nn.train()       # --- Train the Classifier NN ---
        current_avg_nn_loss_val = 0.0
        for epoch in range(CLASSIFIER_RATE):
            optimizer.zero_grad(); predictions = classifier_nn(bands_batch); loss = criterion(predictions, targets_batch); loss.backward(); optimizer.step()
            current_avg_nn_loss_val += loss.item()
        current_avg_nn_loss_val /= CLASSIFIER_RATE
        if current_avg_nn_loss_val < best_overall_loss:                                 # --- Update Best Parameters ---
            best_overall_loss = current_avg_nn_loss_val; best_bs_params = current_bs_params # Update BS params if classifier performs better
            if verbose:
                elapsed_time = time.time() - start_time
                print(f"Iteration {iteration}, New BS Params, Avg NN Loss: {best_overall_loss:.6f}"); print(f"Params: {[f'{p:.4f}' for p in best_bs_params]}, Time: {elapsed_time:.2f}s")
        # --- Progress Check and Evaluation ---
        if iteration % (iterations // 10) == 0 or iteration == iterations - 1:
            classifier_nn.eval() # Set classifier to evaluation mode
            total_test_loss = 0.0; correct_words_exact_match = 0; pyscript = [];
            if verbose: print(f"\n--- Iteration {iteration} Progress Check ---")
            for test_input_word, test_target_word in word_mapping_data:
                test_spectrum = run_gol_and_3d_fft_torch(best_bs_params, test_input_word, debug_output='--debug' in sys.argv)
                test_bands = calculate_3d_fft_energy_bands_torch(test_spectrum, NUM_FFT_BANDS_FOR_CLASSIFIER)
                with torch.no_grad(): test_prediction_values = classifier_nn(test_bands.unsqueeze(0)).squeeze(0) # Get the 1D output vector
                # Reconstruct the word from the predicted values. min_value_threshold=0.5 # Use a higher threshold for stricter output
                predicted_word = TextProcessor.reconstruct_word_from_classifier_output( test_prediction_values, min_value_threshold=0.5)
                true_target_values = TextProcessor.word_to_target_values(test_target_word, MAX_OUTPUT_WORD_LENGTH)
                loss_val = criterion(test_prediction_values, torch.tensor(true_target_values, dtype=torch.float32).to(device)).item(); total_test_loss += loss_val
                if verbose:
                    print(f"    Target: \033[32m'{test_target_word}'\033[m\t\t<--input: '{test_input_word}'"); print(f"  Pred Word: \033[31m{predicted_word}\033[m\t\tLoss: {loss_val:.4f}")
                if predicted_word == test_target_word: # Compare predicted to target (NOT lowercase extended latin)
                    correct_words_exact_match += 1
                    if predicted_word in SYMBOLS_TO_CODE:
                        pyscript.append(SYMBOLS_TO_CODE[predicted_word]); print(f"  SymPython: \033[33m{SYMBOLS_TO_CODE[predicted_word]}\033[m")
                        if '--exec-in-training' in sys.argv:
                            print("\033[34m", end='')
                            try: exec(SYMBOLS_TO_CODE[predicted_word], globals())
                            except BaseException as err: print(err)
                            print("\033[m", end='')
            if pyscript: print('PYTHON OUTPUT SCRIPT'); print("\033[33m", end=''); print('\n'.join(pyscript)); print("\033[m", end='')
            avg_test_loss_current_iter_report = total_test_loss / len(word_mapping_data)
            if verbose:
                print(f"  Average Test Loss (current best BS params): {avg_test_loss_current_iter_report:.4f}")
                print(f"  Exact Word Matches: {correct_words_exact_match}/{len(word_mapping_data)}"); print("-"*80)
            if correct_words_exact_match == len(word_mapping_data): print(f"\nConverged at Iteration {iteration}!"); break
    if verbose:
        end_time = time.time(); print(f"\n--- Optimization Finished ---"); print(f"Total time elapsed: {end_time - start_time:.2f} seconds")
        print(f"Best Burning Ship Parameters Found: {best_bs_params}"); print(f"Lowest Average Classifier Loss Achieved: {best_overall_loss:.6f}")
    return (best_bs_params, classifier_nn)


SYMBOLS_TO_TEMPLATE = { '  ' : 'print(%s + %s)', '  ' : 'print(%s * %s)' }; SYMBOLS_TO_SELF = {}

def genfunc(g):
    a = []
    for part in g.split():
        if len(part)==1: a.append(part)
    op = None; opmap = {'add':'+', 'subtract':'-', 'multiply':'*', 'divide':'/'}
    for word in g.split():
        if word in opmap: op = opmap[word]; break
    args = ','.join(a)
    if op: body = op.join(a)
    else: body = '[%s]' % ','.join(a)
    return 'lambda %s: %s' % (args, body)

def gencall(g):
    name = None; args = []
    for a in g.split():
        if a in ('call', 'function', 'to', 'with', 'and', 'or'): continue
        if len(a) >= 3: name = a
        elif len(a) == 1: args.append(a)
    return '%s(%s)' % (name, ','.join(args))


SYMBOLS_TO_GEN = { '  ' : genfunc, '  ' : gencall }; word_mapping_data = []

if '--quick' not in sys.argv: word_mapping_data+=[('write function to add    and   ',' '),('call function        with      ', ' '),('add    and   ', ' '),('multiply    and   ', ' ')]

if '--english' in sys.argv: word_mapping_data += [ ('write python code to print hello world', ' '), ("one", "two"), ("hello", "world"), ("good", "bad"), ("happy", "sad") ]

def read(prompt='', timeout_seconds=1):
    if not timeout_seconds: return input(prompt)
    if prompt:sys.stdout.write(prompt); sys.stdout.flush() # Ensure the prompt is displayed immediately
    rlist, _, _ = select.select([sys.stdin], [], [], timeout_seconds)
    if rlist: return sys.stdin.readline().strip()
    else: return None

def plot(vec, label='', x=1, y=1, scale=10):
    blocks = ' _▂▃▄▅▆▇'; o = []; u=[]
    for i,v in enumerate(vec):
        idx = (int(v*scale)); c = '↑'
        if idx < len(blocks): c=blocks[idx]
        o.append(c)
        if idx >= 2: u.append(TextProcessor.CHARACTERS[i])
        else: u.append('.')
    printat(label + ''.join(o), y, x); printat((' '*len(label)) + ''.join(u), y+1, x)

def dream(params, classifier_nn):
    for i in range(len(params)):        # Mutate Burning Ship parameters
        if i < 3: params[i] += random.uniform(-MUTATION_RATE, MUTATION_RATE)
        else: params[i] += random.randint( -max(1, int(MUTATION_RATE * params[i])), max(1, int(MUTATION_RATE * params[i]))); params[i] = max(20, min(200, params[i]))
    printat('FRACTAL: %s MUTATE RATE: %s' %([round(v,6) for v in params], MUTATION_RATE), 3, 1)
    magnitude_spectrum = run_gol_and_3d_fft_torch(params, '', debug_output=True); bands = calculate_3d_fft_energy_bands_torch(magnitude_spectrum, len(TextProcessor.CHARACTERS))
    plot(bands.cpu().numpy(), label='ॐ:', scale=100); words = TextProcessor.reconstruct_word_from_classifier_output( bands, min_value_threshold=0.0025 ); printat(words, 18, 70)
    predicted_bands = calculate_3d_fft_energy_bands_torch(magnitude_spectrum, NUM_FFT_BANDS_FOR_CLASSIFIER); classifier_nn.eval()
    with torch.no_grad(): final_prediction_values = classifier_nn(predicted_bands.unsqueeze(0)).squeeze(0)
    reply = TextProcessor.reconstruct_word_from_classifier_output( final_prediction_values, min_value_threshold=MIN_VAL_THRESH )
    plot(final_prediction_values.cpu().numpy(), label='९:', x=35); printat(reply, 19, 70)

def main():
    global word_mapping_data
    word_mapping_data += LEARN; print(word_mapping_data)
    best_bs_params, classifier_nn = solve(word_mapping_data)
    classifier_nn.eval()
    pyscript = []; var_stack = []; print('\033[36m%s\033[m' % '='*80); print('\033[36m    Input your query or command and press [Enter] key\033[m')
    while 1:
        input_word = read()
        if input_word is None:
            if word_mapping_data and random.random() < -0.3: best_bs_params, classifier_nn=solve(word_mapping_data, iterations=10, verbose=False)
            else: dream( list(best_bs_params), classifier_nn)
            continue
        if input_word.count('=')==1:
            _a,_b = input_word.split('='); _a = _a.strip(); _v = eval(_b); print(f'\033[36m    new variable: {_a} = {_v}\033[m'); globals()[_a]=_v; var_stack.append(_a)
            continue
        elif input_word.endswith(')'):
            try: eval(input_word)
            except BaseException as err: print(err)
            continue
        magnitude_spectrum = run_gol_and_3d_fft_torch(best_bs_params, input_word, debug_output=False)
        predicted_bands = calculate_3d_fft_energy_bands_torch(magnitude_spectrum, NUM_FFT_BANDS_FOR_CLASSIFIER)
        with torch.no_grad(): final_prediction_values = classifier_nn(predicted_bands.unsqueeze(0)).squeeze(0)
        reply = TextProcessor.reconstruct_word_from_classifier_output( final_prediction_values, min_value_threshold=MIN_VAL_THRESH )
        print(f"\033[31m{reply}\033[m")
        if reply in SYMBOLS_TO_GEN:
            gen = SYMBOLS_TO_GEN[reply](input_word); print(f"  Generated: \033[33m{gen}\033[m")
            if gen.startswith('lambda '):
                print("\033[36m    save lambda function (type name) or press [Enter] to ignore\033[m"); user = input().strip()
                if user:
                    if len(user) != 3: print("WARN: function name is not length three")
                    func = eval(gen,globals()); print('\033[36m    saved global function: %s %s\033[m' %(user, func))
                    globals()[user] = func
            elif gen.endswith(')'):
                print('\033[36m    call function? y/n (%s)\033[m' % gen)
                if input().strip():
                    try: _ = eval(gen,globals()); print(_)
                    except BaseException as err: print(err)
        elif reply in SYMBOLS_TO_TEMPLATE:
            print(f"  Template: \033[33m{SYMBOLS_TO_TEMPLATE[reply]}\033[m"); args = []
            for part in input_word.split():
                if len(part)==1: args.append(part)
            print('\033[36m    run code? y/n (with template args: %s)\033[m' % args)
            if input()=='y':
                print("\033[34m", end='')
                try: exec(SYMBOLS_TO_TEMPLATE[reply] % tuple(args), globals())
                except BaseException as err: print(err)
                print("\033[m", end='')
        elif reply in SYMBOLS_TO_CODE:
            pyscript.append(SYMBOLS_TO_CODE[reply]); print(f"  SymPython: \033[33m{SYMBOLS_TO_CODE[reply]}\033[m"); print('\033[36m    run code? y/n\033[m')
            if input()=='y':
                print("\033[34m", end='')
                try: exec(SYMBOLS_TO_CODE[reply], globals())
                except BaseException as err: print(err)
                print("\033[m", end='')
        else:
            print(reply, "?")
            if len(input_word.split()) == 2: word_mapping_data.append( tuple(input_word.split()) )

def printat(text: str, row: int, col: int):
    if '--vis' not in sys.argv: return
    save_cursor = "\033[s"; restore_cursor = "\033[u"; sys.stdout.write(save_cursor)
    for i, ln in enumerate(text.splitlines()): move_cursor = f"\033[{row+i};{col}H"; sys.stdout.write(move_cursor + ln)
    sys.stdout.write(restore_cursor); sys.stdout.flush() # Ensure the output is immediately written to the terminal
```

```
if __name__ == '__main__': main()
```

---

6. Conclusion

We have presented a novel computational model that extends our prior work on emergent computation through fractal dynamics and spatiotemporal spectral analysis. By introducing a simplified Domain Specific Language, we have enabled the system to output and dynamically integrate core functional components, specifically demonstrating the self-modification of the Game of Life stepping function. This emergent meta-programming capability represents a significant step towards creating truly adaptive, self-improving, and resilient artificial intelligence systems, blurring the lines between learning, symbolic reasoning, and functional self-reconfiguration.

References:

[1] Iterating a Fractal-like Self Awareness Naturally https://ai.vixra.org/abs/2505.0195

[2] The Boundary of Neural Network Trainablity is Fractal https://arxiv.org/abs/2402.06184

[3] FractalNet: Ulta-Deep Neural Networks without Residuals https://arxiv.org/abs/1605.07648

[4] How neurons exploit fractal geometry to optimize their network connectivity https://www.nature.com/articles/s41598-021-81421-2

[5] Neural network training makes beautiful fractals https://sohl-dickstein.github.io/2024/02/12/fractal.html

[6] Growing Cellular Automata https://distill.pub/2020/growing-ca/

[7] Empowered Neural Cellular Automata https://arxiv.org/html/2205.06771v2

[8] Learning spatio-temporal patterns with Neural Cellular Automata https://pmc.ncbi.nlm.nih.gov/articles/PMC11078362/

[9] The Lookahead Limitation: Why Mult-Operand Addition is Hard for LLMs https://arxiv.org/abs/2502.19981v1

[10] Emergent Computation Through Fractal Dynamics and Spatiotemporal Spectral Analysis https://ai.vixra.org/abs/2506.0129

[11] Emergent Symbolic Computation Through Fractal Dynamics and Spatiotemporal Spectral Analysis https://ai.vixra.org/abs/2507.0009

[12] Asymptotic Behavior and Ratios of Complexity in Cellular Automata https://arxiv.org/abs/1304.2816

[13] Computing by Nowhere Increasing Complexity https://arxiv.org/abs/1710.01654

[14] On the Structure of Clifford Quantum Cellular Automata https://arxiv.org/abs/0804.4447

[15] A Phase-space Approach to Weighted Fourier Extension Inequalities https://arxiv.org/abs/2406.14886

[16] A Counterexample to the Mizohata-Takeuchi Conjecture https://arxiv.org/abs/2502.06137