



Chain of Code: Reasoning with a Language Model- Augmented Code Emulator

Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman,
Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia[†], Brian Ichter[†]

chengshu@stanford.edu, xiafei@google.com, brian@physicalintelligence.company

ICML 2024 Oral

Google DeepMind

π

Stanford

Berkeley
UNIVERSITY OF CALIFORNIA

LLM can leverage code-writing capabilities to solve complex algorithmic/mathematical tasks.

Consider two triangles formed by (0, 0), (2, 5), (3, 4) and (1, 2), (4, 10), (3, 7) respectively. Which triangle has a larger area? Feel free to write python code to solve the problem if necessary.

Self-define new functions

```
python Analyzed Always show details Copy code

import numpy as np

# Function to calculate area of triangle using vertices (x1, y1), (x2, y2), (x3, y3)
def triangle_area(x1, y1, x2, y2, x3, y3):
    return abs(x1*(y2 - y3) + x2*(y3 - y1) + x3*(y1 - y2)) / 2

# Coordinates of the first triangle
triangle1_coords = [(0, 0), (2, 5), (3, 4)]

# Coordinates of the second triangle
triangle2_coords = [(1, 2), (4, 10), (3, 7)]

# Calculate the areas of the triangles
area_triangle1 = triangle_area(*triangle1_coords[0], *triangle1_coords[1], *triangle1_coords[2])
area_triangle2 = triangle_area(*triangle2_coords[0], *triangle2_coords[1], *triangle2_coords[2])

area_triangle1, area_triangle2

Result
(3.5, 0.5)
```

The area of the first triangle is 3.5 square units, while the area of the second triangle is 0.5 square units. Therefore, the first triangle has a larger area. [-]

Leverage existing libraries

```
python Analyzed Always show details Copy code

from sympy import Polygon, Point

# Define the points of the triangles
triangle1 = Polygon(Point(0, 0), Point(2, 5), Point(3, 4))
triangle2 = Polygon(Point(1, 2), Point(4, 10), Point(3, 7))

# Calculate the areas of the triangles
area1 = triangle1.area
area2 = triangle2.area

area1, area2

Result
(-7/2, -1/2)
```

The calculated areas are given as negative due to the order of points provided. The absolute values represent the actual areas.

- Area of Triangle 1: $|-7/2| = 3.5$
- Area of Triangle 2: $|-1/2| = 0.5$

Thus, Triangle 1 (formed by points (0, 0), (2, 5), and (3, 4)) has a larger area of 3.5 square units compared to Triangle 2's area of 0.5 square units. [-]

However, many semantic tasks are challenging to express through code.

How many times do you detect sarcasm in the paragraph?

“Oh, isn't it delightful how AI advances so fast? One minute, you're deep in your Ph.D. studies, the next, everything you know is outdated. With ICML spewing out 3000 new papers a year, it's like trying to drink from a firehose. What should I do...?”

```
num_sarcasm = 0
for sentence in paragraph:
    num_sarcasm += detect_sarcasm(sentence)
return num_sarcasm

def detect_sarcasm(sentence):
```



```
>>> detect_sarcasm("Oh, isn't it delightful how AI advances so fast?")
True
>>> detect_sarcasm("What should I do...?")
False
```

- Chain-of-Thought decomposes problems into intermediate reasoning steps via natural language, excelling at semantic tasks yet struggling with precise numerical or symbolic operations.
- Program of Thoughts generates code to represent its reasoning steps, excelling at arithmetic tasks yet struggling with semantic tasks.
- ScratchPad maintains a program state of intermediate steps when simulating the output of code, enabling a LM to act as a code interpreter.

(a) Chain of Thought

```
Q: Roger has 5 balls. He buys 2 more packs, each with 3. How many balls does he have now?  
  
Roger starts with 5 balls.  
2 packs of 3 balls is 6.  
5 + 6 = 11.  
  
A: 11
```

(b) Program of Thoughts

```
Q: Roger has 5 balls. He buys 2 more packs, each with 3. How many balls does he have now?  
  
num_balls = 5  
num_balls += 2 * 3  
answer = num_balls  
  
A: 11
```

(c) ScratchPad

```
Q: Roger has 5 balls. He buys 2 more packs, each with 3. How many balls does he have now?  
  
num_balls = 5 state: {num_balls = 5}  
num_balls += 2 * 3 state: {num_balls = 11}  
answer = num_balls state: {answer = 11}  
  
A: 11
```

Blue highlight indicates LM generation.

Red highlight indicates LM generated code being executed by an interpreter

Purple highlight indicates an LMulator (LM code emulator) simulating the code via a program state in green.

Chain of Code (CoC)

Given a problem, CoC generates reasoning substeps in the structure of code. CoC runs the code line by line then.

For each line:

- If executable, the interpreter updates the program state with precise calculations.
- If not executable, the LMulator simulates the expected output, and the program state is updated accordingly.

This interplay enables CoC to handle a wide range of tasks that mix numerical precision with semantic reasoning.

Chain of Code

```
Q: How many countries have I been to? I've been to Mumbai,
London, Washington, Grand Canyon, Baltimore, ...
1  places, countries = ["Mumbai", ...], set()
2  for place in places:
3      country = get_country(place)
4      countries.add(country)
5  answer = len(countries)
A:
...
```

Blue highlight indicates LM generation.

Red highlight indicates LM generated code being executed by an interpreter.

Purple highlight indicates an LMulator (LM code emulator) simulating the code via a program state in green.

Method	Key Idea	Executes Code?	Handles Non-Executable Steps?	Improvement in Reasoning
Chain of Thought (CoT)	Uses step-by-step natural language reasoning	No	No	Limited on numeric/symbolic tasks
Program of Thoughts (PoT)	Writes and executes code for problem-solving	Yes	No	Effective primarily for arithmetic tasks
ScratchPad	Tracks intermediate steps through text-based simulation	No	Yes	Works only in text-based reasoning
Chain of Code (CoC)	Combines code execution with LM-based simulation	Yes	Yes	Integrates the best of both approaches

Evaluation

Dataset: BIG-Bench Hard [1]

- Subset of the 23 most challenging tasks from BIG-Bench [2]
- Across diverse problem domains



Commonsense



Logic



Arithmetic



Geometry



Language

Models: *text-davinci-003* from OpenAI

BIG-Bench Hard Task

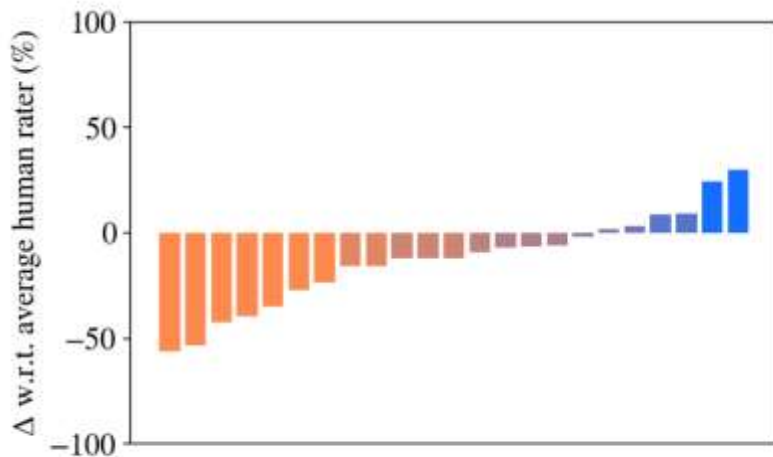
- Boolean Expressions^{λ+}
- Causal Judgement^{κ*}
- Date Understanding^{κ-}
- Disambiguation QA^{κ/}
- Dyck Languages^{λ+}
- Formal Fallacies^{κ*}
- Geometric Shapes^{λ+}
- Hyperbaton^{κ/}
- Logical Deduction^{λ*}
- Movie Recommendation^{κ/}
- Multi-Step Arithmetic^{λ+}
- Navigate^{λ*}
- Object Counting^{λ-}
- Penguins in a Table^{κ-}
- Reasoning about Colored Objects^{κ-}
- Ruin Names^{κ/}
- Salient Translation Error Detection^{κ/}
- Snarks^{κ/}
- Sports Understanding^{κ/}
- Temporal Sequences^{λ*}
- Tracking Shuffled Objects^{λ-}
- Web of Lies^{λ-}
- Word Sorting^{λ+}

Task Averages

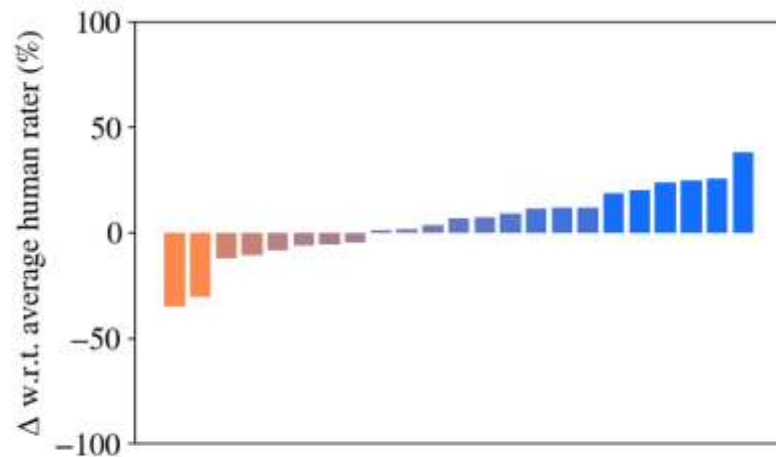
- NLP Task (avg)^κ
- Algorithmic Task (avg)^λ
- All Tasks (avg)

1. [\[2210.09261\] Challenging BIG-Bench Tasks and Whether Chain-of-Thought Can Solve Them](#)
2. [\[2206.04615\] Beyond the Imitation Game: Quantifying and extrapolating the capabilities of language models](#)

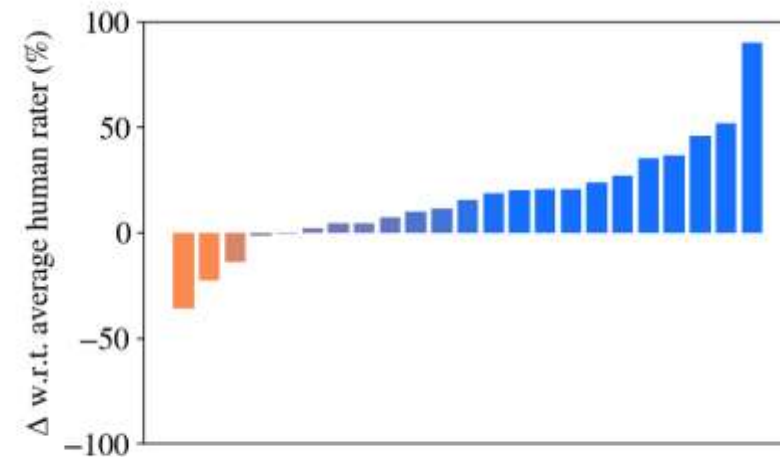
Evaluation



(a) Direct answer only

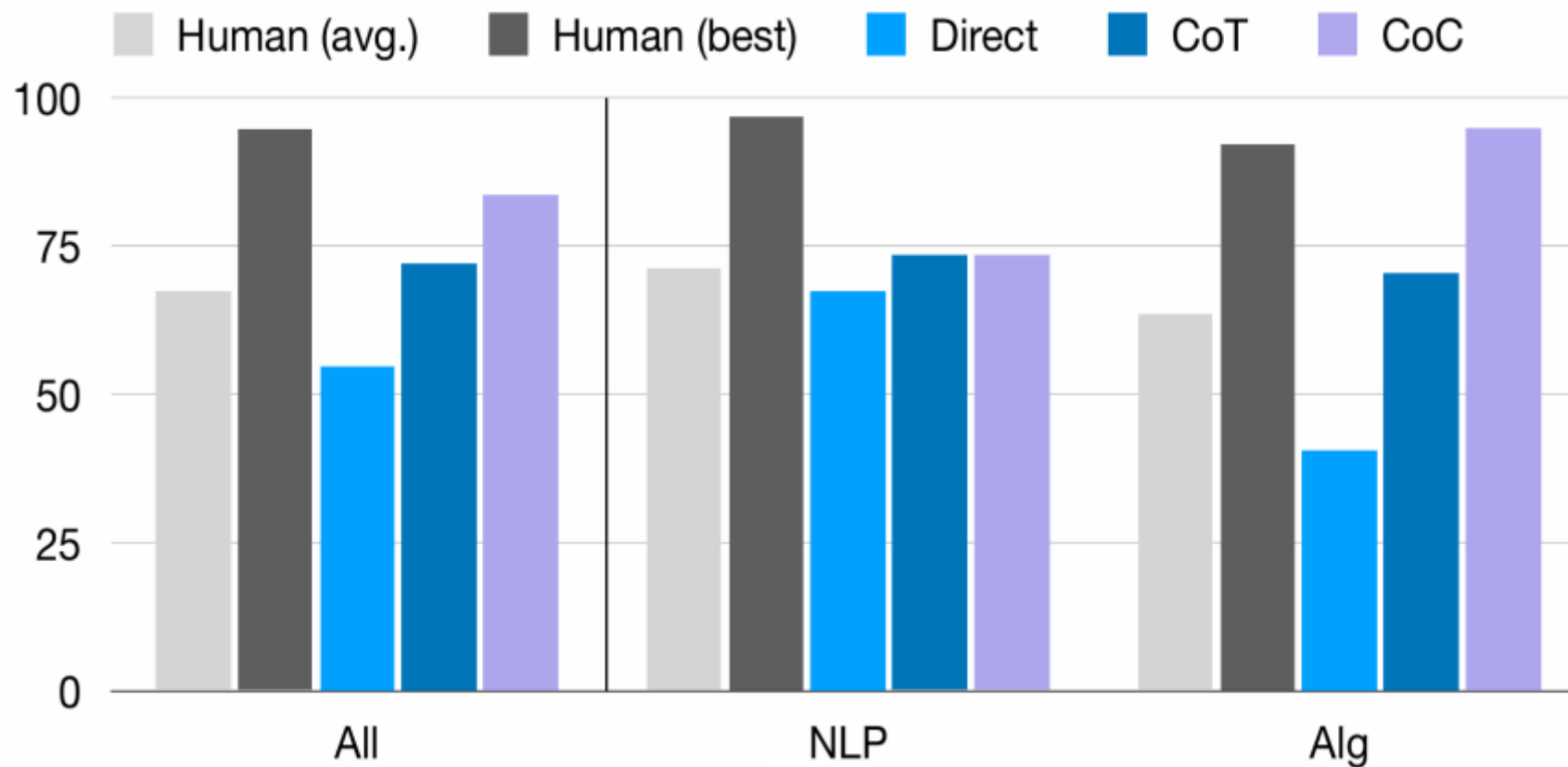


(b) Chain of Thought



(c) Chain of Code (Ours)

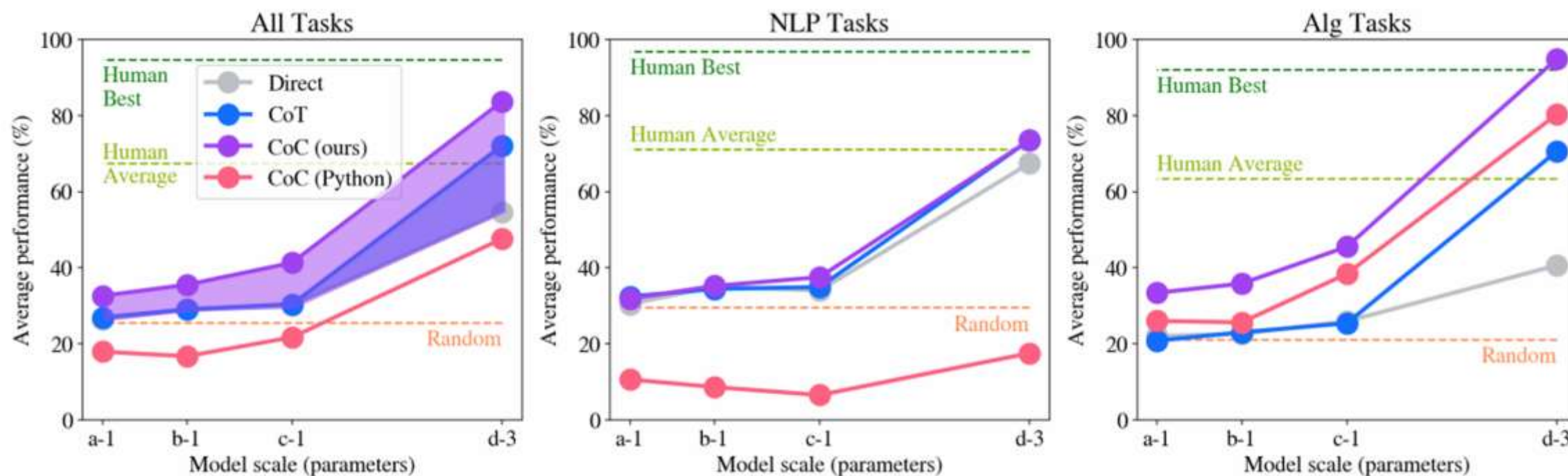
Chain of Code outperforms the average human raters in 18 out of 23 tasks.



Chain of Code performs on par with Chain of Thought for the NLP subset, and outperforms even the best human raters for the algorithmic subset.

Evaluation

a-1 refers to model *text-ada-001* from OpenAI
b-1 refers to model *text-babbage-001* from OpenAI
c-1 refers to model *text-curie-001* from OpenAI
d-3 refers to model *text-davinci-003* from OpenAI



Unlike CoT, CoC brings benefits even for smaller-sized models.

CoC is well fit for solving robotics tasks because

- They require both semantic and algorithmic reasoning.
- They also involve interfacing with other APIs through code (e.g., control or perception APIs) and with users through natural language.

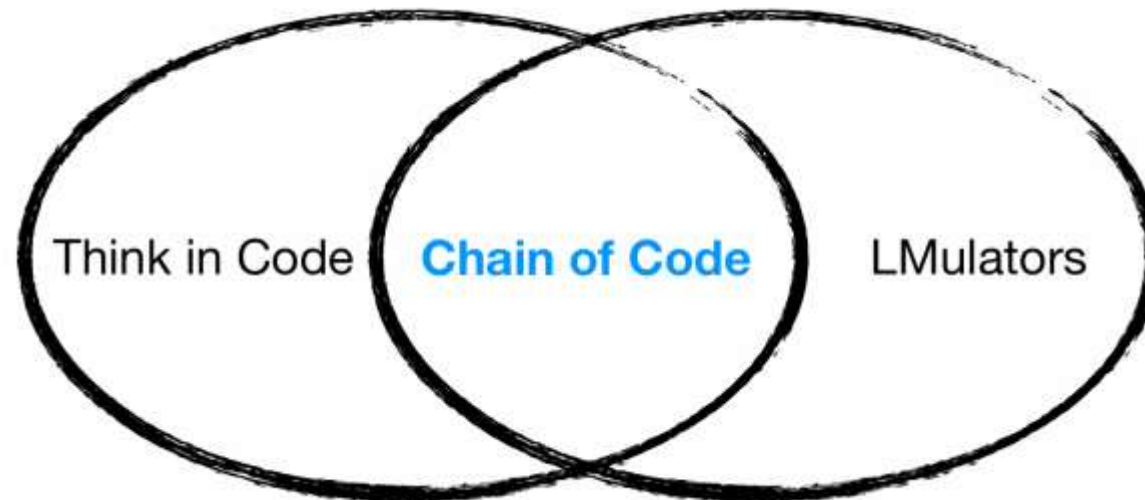
Q: Sort the objects on the table into the compost bin and the recycle bin.

```
_, _, objs, _ = robot.detect_objects()
compost_bin_obj = None
recycle_bin_obj = None
for obj in objs:
    is_match_description = match_description(obj, "compost bin", ret_type="bool")
    if is_match_description:
        compost_bin_obj = obj
        break
for obj in objs:
    is_match_description = match_description(obj, "recycle bin", ret_type="bool")
    if is_match_description:
        recycle_bin_obj = obj
        break
if compost_bin_obj is None or recycle_bin_obj is None:
    robot.say("No compost bin or recycle bin found.")
else:
    for obj in objs:
        if obj == compost_bin_obj or obj == recycle_bin_obj:
            continue
        is_compostable = is_compostable(obj, ret_type="bool")
        if is_compostable:
            robot.pick_place(obj, compost_bin_obj)
        else:
            robot.pick_place(obj, recycle_bin_obj)
```



While Chain of Code offers a powerful approach, it also faces some challenges:

1. Running both a code interpreter and an LMulator can be **slower** than direct text-based reasoning.
2. Some tasks might still be difficult to simulate accurately with code-based reasoning.
3. Future work could focus on refining how the LM simulates non-executable steps to further boost performance.



Thanks