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

Chengshu Li[*1,2], **Jacky Liang**[1], **Andy Zeng**[1], **Xinyun Chen**[1], **Karol Hausman**[1,2], **Dorsa Sadigh**[1,2], **Sergey Levine**[1,3], **Li Fei-Fei**[2], **Fei Xia**[†,1], **Brian Ichter**[†,1]
Google DeepMind[1], Stanford University[2], University of California, Berkeley[3]
chengshu@stanford.edu, xiafei@google.com, ichter@google.com
https://chain-of-code.github.io

Abstract

Code provides a general syntactic structure to build complex programs and perform precise computations when paired with a code interpreter – we hypothesize that language models (LMs) can leverage code-writing to improve Chain of Thought reasoning not only for logic and arithmetic tasks [5, 26, 1], but also for *semantic* ones (and in particular, those that are a mix of both). For example, consider prompting an LM to write code that counts the number of times it detects sarcasm in an essay: the LM may struggle to write an implementation for "`detect_sarcasm(string)`" that can be executed by the interpreter (handling the edge cases would be insurmountable). However, LMs may still produce a valid solution if they not only write code, but also selectively "emulate" the interpreter by generating the expected output of "`detect_sarcasm(string)`"and other lines of code that cannot be executed. In this work, we propose Chain of Code (CoC), a simple yet surprisingly effective extension that improves LM code-driven reasoning. The key idea is to encourage LMs to format semantic sub-tasks in a program as flexible pseudocode that the interpreter can explicitly catch undefined behaviors and hand off to simulate with an LM (as an "LMulator"). Experiments demonstrate that Chain of Code outperforms Chain of Thought and other baselines across a variety of benchmarks; on BIG-Bench Hard, Chain of Code achieves 84%, a gain of 12% over Chain of Thought. CoC scales well with large and small models alike, and broadens the scope of reasoning questions that LMs can correctly answer by "thinking in code".

Direct answer only

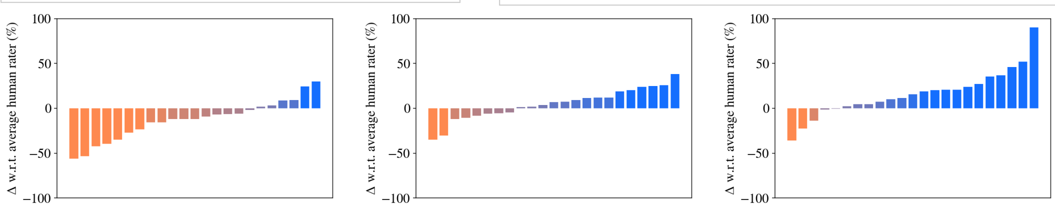
```
Q: How many countries have I been to? I've been to Mumbai, London, Washington, Grand Canyon, ...
A: 32 (20%, ✗), 29 (10%, ✗), 54 (10%, ✓), ...
```

Chain of Thought

```
Q: Let's think step by step. How many countries have I been to? I've been to Mumbai, London, ...
We'll group by countries and count:
1. India: Mumbai, Delhi, Agra
2. UK: London, Dover, Edinburgh, Skye
3. USA: Washington, Grand Canyon, ...
A: 61 (20%, ✗), 60 (20%, ✗), 54 (10%, ✓), ...
```

Chain of Code

```
Q: How many countries have I been to? I've been to Mumbai, London, Washington, Grand Canyon, Baltimore, ...
places, countries = ["Mumbai", ...], set()
delta state: {places = ['Mumbai', ...], countries = set()}
for place in places:
delta state: {place = 'Mumbai'}
    country = get_country(place)
delta state: {country = 'India')}
    countries.add(country)
delta state: {countries = {'India'}}
answer = len(countries)    delta state: {answer = 54}
A: 54 (100%, ✓)
```



(a) Direct answer only (b) Chain of Thought (c) Chain of Code (Ours)

Figure 1: Chain of Code generates code and reasons through an LM-augmented code emulator. Lines evaluated with Python are in red and with an LM are in purple. The full query is in Fig. A4. (1a-1c) show results on BIG-Bench Hard compared to human performance [34].

[*]Work done as a student researcher at Google DeepMind. [†]Equal advising.

1 Introduction

Language models (LMs) at certain scale exhibit the profound ability to solve complex reasoning questions [3, 41] – from writing math programs [9] to solving science problems [17]. Notably, these capabilities have shown to improve with Chain of Thought (CoT) prompting [42], whereby complex problems are decomposed into a sequence of intermediate reasoning steps. CoT excels at semantic reasoning tasks, but tends to struggle with questions that involve numeric or symbolic reasoning [36, 23]. Subsequent work addresses this by prompting LMs (e.g., trained on Github [4]) to write and execute code [5, 26, 1]. Code in particular is advantageous because it provides both (i) a general syntactic structure to build and encode complex programs [19] (e.g., logic structures, functional vocabularies – in ways that are Turing complete), and (ii) an interface by which existing APIs paired together with an interpreter can be used to perform precise algorithmic computations (e.g., from multiplication of large numbers to sorting an array of size 10,000) that a language model trained only to mimic the statistically most likely next token would otherwise struggle to produce.

While writing and executing code may improve LM reasoning performance across a wide range of arithmetic tasks, this particular approach contends with the fact that many semantic tasks are rather difficult (and at times, nearly impossible) to express in code. For example, it remains unclear how to write a function that returns a boolean when it detects sarcasm in a string [36] (handling the edge cases would be insurmountable). Perhaps fundamentally, using LMs to write programs in lieu of multi-step textual reasoning inherently assumes that the intermediate reasoning traces (expressed in lines of code) all need to be *executable* by an interpreter. Is it possible to lift these restrictions to get the best of both reasoning in code and reasoning in language?

In this work, we propose Chain of Code (CoC), a simple yet surprisingly effective extension to improve LM code-driven reasoning – where the LM not only writes a program, but also selectively "simulates" the interpreter by generating the expected output of certain lines of code (that the interpreter could not execute). The key idea is to encourage LMs to format semantic sub-tasks in a program as flexible pseudocode that at runtime can be explicitly caught and handed off to emulate with an LM – we term this an LMulator (a portmanteau of LM and emulator). For example, given the task "*in the above paragraph, count how many times the person was sarcastic*," we can in-context prompt the LM to write a program that may call helper functions such as `is_sarcastic(sentence)`, to which the LM makes a linguistic prediction and returns the result as a boolean output, that then gets processed with the rest of the program. Specifically, we formulate LM reasoning as the following process (illustrated in Figure 1): the LM writes code, the interpreter steps through to execute each line of code (in red), or if it fails, simulates the result with the LM (in purple) and updates the program state (in green). CoC inherits the benefits of both (i) writing executable code (where precise algorithmic compututations are left to an interpreter), and (ii) writing pseudocode for semantic problems, and generating their outputs (which can be thought of as a simple formatting change, to which LMs are robust [22]) – enabling the LM to "think in code."

Extensive experiments demonstrate that CoC is applicable to a wide variety of challenging numerical and semantic reasoning questions, and outperforms a number of popular baselines. In particular, we find that it achieves high performance on BIG-Bench Hard tasks [36], outperforming average human raters overall and outperforming even the best human raters on an algorithmic subset of tasks, and to the best of our knowledge setting a new state of the art. We further show that *both* code interpreter execution and language model execution simulation are necessary for this performance, and that the approach scales well with large and small models alike – contrary to prompting techniques like Chain of Thought that only emerge at scale. Finally, we demonstrate how Chain of Code can serve as a general purpose reasoner via *cross-task prompting* benchmark, which in contrast to prior work, uses prompts from different families of problems as context – providing only the structure of the response (as opposed to the solution itself). This work underscores how one may leverage the structure and computational power of code and the reasoning abilities of language models to enable a "best of both worlds" reasoner.

2 Chain of Code: Reasoning with an LMulator

In this section, we describe Chain of Code (CoC) prompting, an approach that leverages the ability of language models to code, to reason, and to leverage an LM-augmented code emulator (an LMulator) to simulate running code. We start with background in Section 2.1, then overview the method in Section 2.2, its implementation in Section 2.3, and finally its capabilities in Section 2.4.

2.1 Preliminaries

Briefly, we overview some background on LM reasoning. Many of these reasoning techniques have been enabled by in-context learning [3], which provides the model with a few demonstrative examples at inference time, rather than updating any weights with gradients. These examples serve to provide context and format for the setting, enabling the model to emulate these examples while adapting to a new query. This property has been instrumental in easily applying LMs to new tasks as it can be rapidly adapted and requires minimal data.

Through in-context learning, approaches have been developed to leverage human thought processes and use tools to improve performance of language models. We outline three such approaches that provide the foundations for Chain of Code. Chain of Thought (CoT) [42], ScratchPad [26], and Program of Thoughts [5] demonstrated the efficacy of breaking problems down into substeps. For CoT these substeps are in natural language, mirroring one's thought process when stepping through a complicated problem. ScratchPad, on the other hand, maintains a program state of intermediate steps when simulating the output of code – resulting in an LM acting as a code interpreter. Program of Thoughts [5] focused on generating the code itself, which is then executed by a code interpreter to solve reasoning problems. Each of these is visualized in Figure 2.

(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
```

(d) Chain of Code Generation

```
Q: I have an orange, a violin, two peaches, an apple, a pepper, and three plums. How many fruits do I have?

objects = {"orange": 1, "violin": 1, "peaches": 2, "apple": 1, "pepper": 1, "plum": 3}
num_fruits = 0
for object in objects:
  object_is_fruit = is_fruit(object)
  if object_is_fruit:
    num_fruits += objects[object]
answer = num_fruits
```

(e) Chain of Code Execution

```
Q: I have an orange, a violin, two peaches, an apple, a pepper, and three plums. How many fruits do I have?

objects = {"orange": 1, "violin": 1, "peaches": 2, "apple": 1, "pepper": 1, "plum": 3}
delta state: {objects = {'orange': 1, 'violin': 1, ...}}
num_fruits = 0
delta state: {num_fruits = 0}
for object in objects:
delta state: {object = 'orange'} # updated for each loop
  object_is_fruit = is_fruit(object)
delta state: {object_is_fruit = True}
  if object_is_fruit:
delta state: {}
    num_fruits += objects[object]
delta state: {num_fruits = 1}
answer = num_fruits
delta state: {answer = 7}

A: 7
```

Figure 2: **Previous reasoning methods:** To solve advanced problems, (2a) Chain of Thought prompting breaks the problem down into intermediate steps, (2b) Program of Thoughts prompting writes and executes code, and (2c) ScratchPad prompting simulates running already written code by tracking intermediate steps through a program state. **Our reasoning method:** Chain of Code first (2d) generates code or psuedocode to solve the question and then (2e) executes the code with a code interpreter if possible, and with an LMulator (language model emulating code) otherwise. Blue highlight indicates LM generation, red highlight indicates LM generated code being executed, and purple highlight indicates LMulator simulating the code via a program state in green.

2.2 CHAIN OF CODE

Inspired by how a human may reason through a particularly complex problem with a mix of natural language, pseudocode, and running code or how a researcher may develop a new general algorithm through a code-based formalism then apply it to a problem, Chain of Code proceeds in two steps: (1) Generation, which, given the question to solve, an LM generates code to reason through the problem, and (2) Execution, which executes the code via a code interpreter when possible and via an LM when not. See Section 2.3 for more details on the specific implementation.

Chain of Code Generation Given a problem to solve, CoC generates reasoning substeps in the structure of code. This code provides the framework of reasoning through the problem, and may be in the form of explicit code, pseudocode, or natural language. Figure 2d walks through a potential generation to solve an object counting problem from BIG-Bench.

Chain of Code Execution A core contribution of CoC is not just the generation of reasoning code, but the manner in which it is executed. Once the code is written, the code is attempted to be run by a code interpreter – in this work we consider Python, but the approach is general to any interpreter. If the code is successfully executed, the program state is updated and the execution continues. If the code is not executable or raises any exception, the language model instead is used to simulate the execution. The program state is subsequently updated by the language model's outputs and the execution continues. Herein, we refer to this as an LMulator, a portmanteau of LM and code emulator. This relatively simple change enables a variety of new applications for code which mix semantics and numerics. Figure 2e shows how the generated code is run, maintaining the program state and switching between the Python executor and the LMulator.

2.3 CHAIN OF CODE IMPLEMENTATION

While the generation implementation is straightforward prompting and language model generation, the execution implementation is slightly more complex. Our implementation is based on using Python's `try` and `except` and maintaining a program state. Line by line CoC steps through the code. If the line is executable by a code interpreter, it is executed, the program state is updated, and the program continues. If it is not executable by a code interpreter, a language model is given the context of the program (the question, the prior lines, and the history of the program state) and generates the next program state. This emulation can also leverage chain of thought to determine how to respond. That generated program state is then updated for the code interpreter as well. This sharing of program state interweaves the code interpreter and the language model simulator in a manner applicable to arbitrary interweaving, even control flow like `for`-loops and `if`-statements. This continues until the entire code is run, and the answer is retrieved as the value of the variable named `answer`, or in case of irrecoverable errors, with the language model outputting `A: answer`.

As a brief example, the code `answer = 0; answer += is_sarcastic('you don't say'); answer += 1;` would be executed as follows: (1) Python would execute the first line `answer = 0;` and update the program state to `{answer = 0}`, (2) Python would attempt to execute the second line and fail, and thus the LMulator would simulate the code `answer += is_sarcastic('you don't say');` by generating the program state `{answer = 1}`, which would be updated in the program, (3) Python would execute the last line `answer += 1;` and update the program state to `{answer = 2}`, (4) the answer would be retrieved as `2`.

2.4 CHAIN OF CODE ABILITIES

Chain of Code has several attractive properties:

1. It enables code use in entirely new regimes, by combining the advantages of code with the powerful semantic and commonsense knowledge of language models, which can easily express rules that are challenging to express in code (e.g., which foods are fruits?). Such an ability may have benefits beyond reasoning problems and its flexibility enables executing expressive language, such as pseudocode.
2. It leverages the ability of language models to code, a particular strength of recent language models due to the high quality data available.
3. It inherits many of the benefits of reasoning code, both the formal yet expressive structure of code (e.g., Turing completeness) and powerful computational tools available to code (whether simply multiplying two numbers, calculating $\sqrt[5]{12121}$, or simulating physics).

4. It inherits many of the benefits of techniques that reason via intermediate steps, such as Chain of Thought. These techniques enable the language model to use more computation when necessary to solve a problem as well as provide more interpretability.

Empirically, we observe in Section 3 that these benefits results in significant improvements in reasoning performance over a variety of challenging tasks.

3 Language Reasoning Experimental Evaluation

We select challenging problems requiring varied types of reasoning, whether arithmetic, commonsense, or symbolic reasoning tasks, to answer the following questions:

1. How well does CoC perform overall across a variety of tasks?
2. Which types of problems does CoC perform best?
3. How does each aspect of CoC affects overall performance?
4. How does CoC scale with model size?
5. How does CoC perform as a general-purpose reasoner, with prompt examples from different problems rather than the same problem (which we term cross-task prompting)?
6. How does CoC compare with instruction tuned chat models with and without tools?

We first discuss the approaches, ablations, and baselines considered in Section 3.1, then the tasks considered in Section 3.2, and finally the results in Section 3.3.

3.1 Baselines and Ablations

We consider our main method to be **CoC (Interweave)**, also referred to as **CoC (Ours)**, though we also propose two variants with simpler implementation and modestly lower performance: **CoC (try Python except LM)** and **CoC (try Python except LM state)**. These two variants attempt to run the entire generated code with Python (rather than line by line) and if it fails, simulate the code execution with the LMulator, outputting a final answer or an intermediate state trace, respectively. We also perform the following ablations, some of which are comparable to previous work as noted. In **CoC (Python)** Python is used to run the entire generated code and if the code is not executable, it is marked as failure – this can be thought of as a comparison to Program of Thoughts [5] or Program-aided language models [10]. We note that in many cases this baseline is particularly challenged, as writing executable code for some of the reasoning problems becomes nearly impossible (e.g., writing code to judge if a phrase is sarcastic), but one may focus on the results for Algorithmic only tasks for a more fair comparison. In **CoC (LM)** the code is interpreted by an LMulator outputting the final answer, and in **CoC (LM state)** the code is interpreted by an LMulator outputting a state trace of intermediate steps – this can be thought of as ScratchPad prompting for reasoning [26]. Note, the last two ablations do not leverage the Python interpreter.

We also compare against the following baselines. In **Direct** question answering the LM simply responds to the question with a final answer. In Chain of Thought prompting **(CoT)** the LM uses intermediate steps to solve the task; we use CoT as our standard prompt technique for the field of substep prompting [16, 48] as prompts are readily available.

3.2 Tasks

We consider a subset of challenging tasks from BIG-Bench [34] called BIG-Bench Hard (BBH) [36] to ensure we are solving the most challenging tasks. These tasks were specifically selected for their difficulty for language models and the datasets provides human-rater baselines and a set of Chain of Thought prompts. The 23 tasks require semantic reasoning(e.g., “Movie Recommendation”), numerical reasoning (e.g., “Multi-Step Arithmetic”), and a combination of both (e.g., “Object Counting”). As such they enable us to study the efficacy of CoC across varied problems, not just those that coding is a natural fit for. Several prompts are shown in Appendix Figure A1. We also show results for the grade-school math (GSM8K) benchmark [7] in Appendix Section A.2, though find that these problems are primarily solved algorithmically alone through code.

These tasks are evaluated with **few-shot prompting**, whereby three examples from the same problem family are provided as context. We also introduce a new evaluation setting, **cross-task prompting**, whereby three examples of *different* problems are provided as context. As such, the language model has in-context examples of the *format* of reasoning, but isn't provided explicit instructions on *how* to reason. We see this as an indicative signal for a general-purpose reasoner, which in many real-world applications (e.g., chatbots) would be asked to reason across a wide variety of tasks.

The models used herein include the OpenAI family of models: `text-ada-001`, `text-baggage-001`, `text-curie-001`, and `text-davinci-003` (in plots we denote these as a-1, b-1, c-1, and d-3). We also consider PaLM-2's code finetuned variant [6, 12]. For instruction tuned models, we compare to recent variants of GPT (`gpt-3.5-turbo` and `gpt-4`) with the chat completion mode run in October 2023. The results below are using the `text-davinci-003` model unless otherwise stated.

3.3 Results

Question 1: Overall Performance. The overall performance of CoC is shown in Figure 1 and Table 1 (with full results in Table A1). We see that CoC outperforms other approaches, both in the number of tasks it exceeds the human baseline and in the overall amount that it exceeds the baseline. Indeed, CoC's 84% is SoTA to the best of our knowledge [11]. In several tasks CoC vastly outperforms the human baseline and other methods, achieving nearly 100% – generally for these tasks the result is complicated in language but trivial in code (e.g., a task from multi-step arithmetic `Q:` $((-3+5\times8\times-4)-(9-8\times-7))=$). We also observe that CoT outperforms the human baseline on a number of tasks, while the Direct answer fares poorly.

Table 1: Overall performance (%) with both few-shot prompting with a single task and cross-task. The delta compared to direct prompting is shown in parenthesis.

		text-davinci-003			PaLM 2-S* (code variant [12])		
	Human	Direct	CoT	CoC	Direct	CoT	CoC
Single task	68	55	72 (+17)	84 (+29)	49	61 (+12)	78 (+29)
Cross task	-	50	55 (+5)	61 (+11)	45	47 (+2)	47 (+2)

Question 2: Problem Type. Figure 3 breaks the results down by problem type; the task labels are shown in Table A1. First, we isolate problems that are primarily algorithmic or primarily natural language (these categories were identified in [36]). We see that on algorithmic tasks, CoC performs particularly well, while on natural language tasks CoC performs on par with CoT. This is particularly encouraging, because one may expect these language oriented tasks to be a worse fit for code. The key is that our method offers the flexibility of using a LMulator to simulate the output of code execution, retaining the semantic reasoning capabilities of LMs for natural language problems.

Figure 3 additionally breaks the tasks down into categories that capture how different each question's response is and whether the code can be fully executed by Python (denoted Python only vs. Python + LM). For some tasks within the benchmark, each question has the same code or Chain of Thought, with the only variation being the inputs – in this case we say the code is (repeated code), and if not then it is denoted (new code). As expected, we see that when the code is repeated and run by Python, CoC gets nearly 100%, though these tasks (e.g., multi-step arithmetic) seem to be among the most challenging for the other baselines, including human raters. The other categories are more challenging for CoC; however in each, we still see a benefit over baselines.

Question 3: Ablations. Figures 4 and 5, and Table 2 show the ablations performed to motivate each aspect of Chain of Code prompting. As one may expect, the approaches that execute Python (CoC (Interweave, Python, try Python except LM, try Python except LM state)) achieve 100% performance on several tasks – if the code is correct, then the model will be correct every time. However, the approach that relies on Python alone (CoC (Python)) performs poorly when applied to non-algorithmic tasks, failing almost all. The CoC (Python) ablation is similar to recent works [10, 5], which show that if applied to numerical problems then code reasoning performs well. CoC without the Python interpreter (CoC (LM, LM state)) too fares poorly, though we see that the step-by-step approach proposed in ScratchPad prompting [26] improves in each task.

We also show that ablations CoC (try Python except LM, try Python except LM state), in which CoC first tries to run the entire code with Python and if it fails simulates the code with an LM, perform quite well. Again we see that maintaining a program state provides an improvement in performance. With only

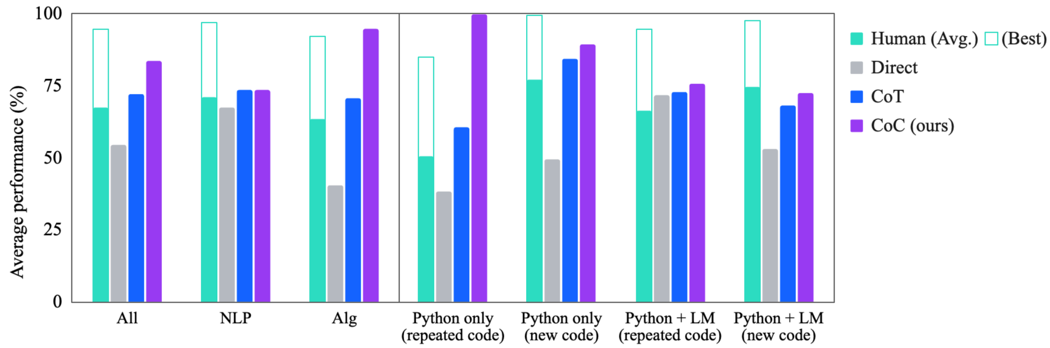


Figure 3: Average performance across different baselines grouped by task type, indicating the problem type and how the CoC is generated and executed.

minor degradations in performance observed, they are reasonable alternatives to the fully interweaved CoC for their simplicity. Though we note, these ablations' performance would be much worse in cases where interweaving code and semantics is truly necessary – for example, if we imagine a case where code is necessary to parse image inputs or to access an external database, but language is necessary to parse the results (see the robotics applications in Section 4).

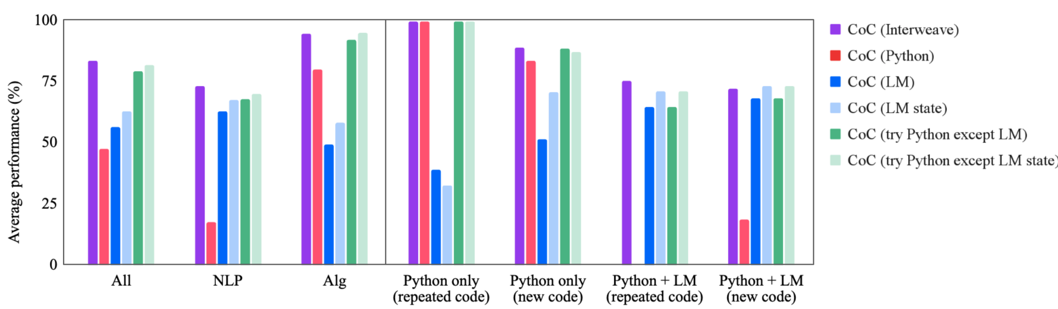


Figure 4: CoC ablations on average performance grouped by task type.

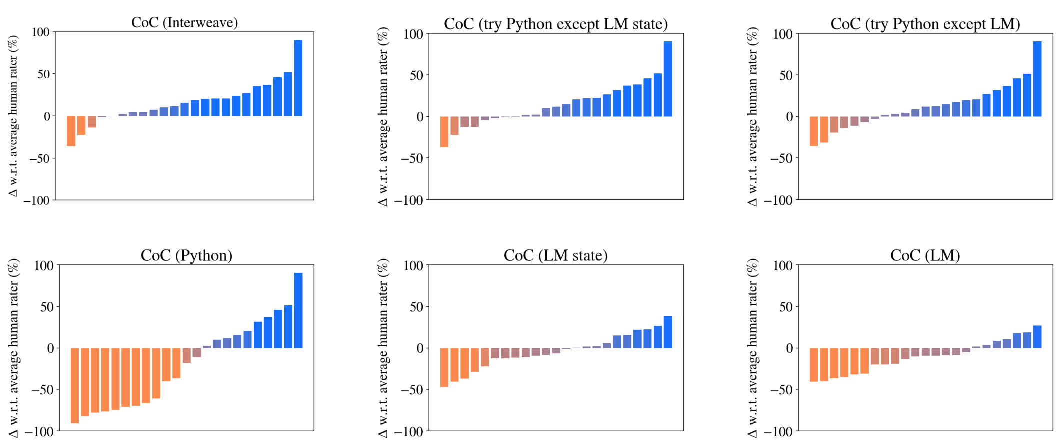


Figure 5: Results across all BIG-Bench Hard tasks compared to human baseline [34]. The tasks (x-axis) in each plot are sorted individually by performance. See Table A1 and Figure 4 for a breakdown by task type.

Question 4: Scaling. Figure 6 shows the performance of CoC across various model sizes. We observe that, similar to Chain of Thought prompting, the improvements of CoC increases as model size increases. In fact, for some of the algorithmic tasks, Chain of Code even outperforms the best human raters (whom

Table 2: Ablation overall performance (%) with both few-shot prompting with a single task and cross-task. The delta compared to the full model (Interweave) is shown in parenthesis.

	Chain of Code					
Prompt	Interweave	try Python except LM state	try Python except LM	Python	LM state	LM
Single task	84	82 (-2)	80 (-4)	48 (-36)	63 (-21)	57 (-27)
Cross task	61	57 (-4)	60 (-1)	35 (-26)	49 (-12)	50 (-11)

admittedly did not have access to code). Unlike Chain of Thought prompting, however, which only brings performance benefits for the largest model (d-3), CoC outperforms the direct question answering baseline also for smaller models (a-1, b-1, c-1), suggesting that it's easier for smaller models to output structured code as intermediate steps rather than natural languages.

Question 5: Cross-task Prompting. For cross-task prompting, we prompt the language models with a few examples from different problems. We see the performance drops for all methods in Figure 6. Despite this drop, CoC outperforms CoT and direct prompting at scale, nearly achieving human average performance. This is a promising indication towards general purpose reasoning, in which a model does not expect to receive examples of similar problems in its prompt.

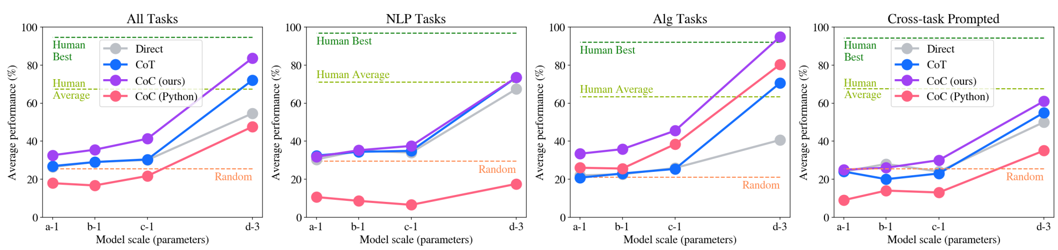


Figure 6: Average performance with model scaling.

Question 6: Instruction Tuned Models. To compare against instruction tuned models with the chat interface, we prompt the models with instructions to elicit the desired reasoning approaches. For the baselines, we ask the model to "directly answer" (Direct) or "think step by step" (CoT). For CoC variants, we ask the model to "write python code to help solve the problem, if it's helpful". If a program is written, we either run the code with a Python interpreter and then feed the result (or the error message if execution fails) back to the model to determine a final answer (CoC (Python)), or ask the model to simulate the output of code execution as a LMulator (CoC (LM)). The CoC (Python) baseline can be thought of as a comparison to an LM with Python tool use.

Table 3 shows the performance of each. With `gpt-3.5-turbo`, both CoT and CoC (Python) show benefits over direct prompting, although both are strongly outperformed by CoC (Interweave). With `gpt-4`, despite the considerable model strength advantage over `text-davinci-003`, CoC (Interweave) still outperforms, though the gap is narrower. Due to the limits of the chat interface, we are unable to run the full CoC (Interweaved) approach with these models, but we do expect further gains if it were to be paired with `gpt-4`.

Table 3: Comparisons with instruction tuned models in the chat interface, with and without tool use.

text-davinci-003	gpt-3.5-turbo				gpt-4			
CoC (Interweave)	Direct	CoT	CoC (Python)	CoC (LM)	Direct	CoT	CoC (Python)	CoC (LM)
84	51 (-33)	56 (-28)	56 (-28)	45 (-39)	70 (-14)	78 (-6)	82 (-2)	75 (-9)

4 Robotics Applications

Downstream applications such as robotics are well fit for CoC as robotics tasks require semantic reasoning and algorithmic reasoning, as well as interfacing with other APIs through code (such as control or perception APIs [19]) and with users through natural language. For example, given a task like "sort the fruits by size", the robot must reason over which items are fruits, sort them by size, and then connect those decisions to actions executable on the robot. CoC (Interweave) is able to solve these challenges with the Python interpreter and the LMulator at runtime, while allowing for more interpretability and fine-grained control of the robot policies.

Environment and Robot Setup. Our environment is a tabletop with small objects (containers, toys, etc) and a UR5 robot arm equipped with a vacuum gripper and a wrist-mounted RGB-D camera. For the purpose of our experiments, the available perception API is `detect_objects()`, which returns a list of detected objects (probabilities, labels, bounding boxes and segmentation masks) from the wrist camera. This API is implemented with first querying GPT-4V [27] for a list of objects, and then using Grounding-SAM [15, 20] to localize them. The available control API is `pick_place(obj1, obj2)`, which is a scripted primitive skill that picks up `obj1` and places it on top of `obj2`. There is also a text-to-speech API `say(sentence)` that allows the robot to communicate with the user.

Results. We evaluate with a number of tabletop pick-and-place robotics tasks that involve semantic reasoning; these tasks are listed in Section A.4. With few-shot prompting, one example is provided as context (of a food serving problem) so that the language model understands the expected structure as well as the available robot APIs. From this single example, we see that our model is able to generalize to new objects, languages, and task domains (see Figure A3 and an example trajectory in Figure 7). Note that for these robotics tasks, unlike the previous language reasoning tasks, our main method CoC (Interweave) is the only capable approach, as the code requires line-by-line interplay between the Python interpreter execution (robot APIs) and the LMulator (commonsense QA like `is_compostable`).

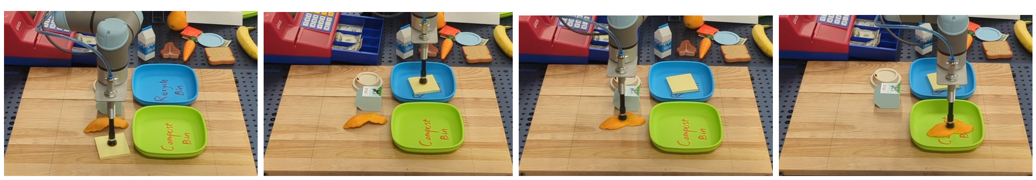


Figure 7: Robot trajectory visualization for task "sort the objects on the table into the compost bin and the recycle bin". CoC first generates code to solve the problem, and then executes the code with Python if possible (e.g., robot APIs like `detect_objects` and `pick_place`), and with LMulator if not (e.g., commonsense QA like `is_compostable`). The robot successfully picks and places the Post-it note to the recycle bin and the orange peel to the compost bin. See the full code in Fig. A3.

5 Related Work

Language Model Reasoning The abilities and applications of language models have seen significant progress, due to their overall performance [6, 37, 31, 11] and emergent capabilities [41], such as few-shot prompting [3] and abstract reasoning [42]. Perhaps most related to this work, a number of works have leveraged prompting to improve reasoning [8]: Chain of Thought [42] proposes to break a task down into intermediate reasoning steps, least-to-most [48] proposes a series of increasingly simpler problems, and ScratchPad [26] proposes to maintain a trace of intermediate results for interpreting code (this first demonstrated the code simulation ability of LMs required for our LMulator). Along these lines "let's think step by step" [16] uses a few key words to elicit such break downs (words that were later refined to "Take a deep breath and work on this problem step-by-step" in [43]). Beyond these, other approaches structure such step-by-step solutions into graphical structures [45, 2], plans [39, 25], or mixture of expert-based sampling [40, 49]. CoC builds upon the intuition of these works, with the observation that *code* is a formal, structured approach to breaking a problem down into sub-steps with many advantages beyond natural language alone.

Language Model Tool Use Many recent works have proposed techniques for language models to use tools to respond to queries [21]. These tools have often been provided to the language model through prompting [7, 14, 6, 9, 44], enabling tools like calculators for math problems, code interpreters, databases, or more.

These tools too can provide feedback on novel modalities [35, 46]. To expand the range of tools available, others have used external tool databases or finetuned language models [32, 30, 29, 28]. As tool interfaces vary, feedback from the tool too can improve performance [13, 47]. In this work we leverage the expressibility and generality of full code as well as its structure, by treating it both as a tool and as a framework.

Language Model Program Synthesis The ability of language models to code is well known and they have been applied as programming assistants [4] and shown to be capable programmers on their own [1, 18, 24]. This ability has been applied to a variety of tasks outside of language alone, leveraging their ability to reason through code in new settings, such as robotics [19, 33], embodied agents [38], or vision [35]. Others have specifically done so for reasoning, such as Program of Thoughts [5] and Program-aided Language Models [10], which generate code to solve numerical reasoning problems. Herein, we focus on the interplay between writing code, running code, and language models simulating code, thus enabling new regimes of language model code applications, such as semantic reasoning.

6 Conclusions, Limitations, and Future Work

We have proposed Chain of Code, an approach towards reasoning with language models through writing code, and executing code either with an interpreter or with a language model that simulates the execution (termed herein an LMulator) if the code is not executable. As such, CoC can leverage both the expressive structure of code and the powerful tools available to it. Beyond this, by simulating the execution of non-executable code, CoC can apply to problems nominally outside the scope of code (e.g., semantic reasoning problems). We have demonstrated that this approach outperforms baselines, and for some tasks even the best human raters, in a range of challenging language and numeric reasoning problems.

This work is not without its limitations. First, generating and executing in two steps as well as interweaving code and language execution requires additional context length and computation time. Second, though we have not seen any loss of performance for semantic tasks in aggregate, there are few tasks in which code doesn't help, e.g., the task Ruin Names, which asks whether an edit for a name is humorous. Finally, our implementation to interweave LM and code is quite simple, tracking the program state in strings and parsing the strings into Python's built-in data types (e.g., dict, tuple). As our method stands now, the LM cannot modify custom Python objects while simulating code execution. In theory, however, it is doable as long as each of these Python objects have a serialization and deserialization method, e.g., using techniques like Protocol Buffers.

There are many avenues for future work with CoC. First, we believe that a unified code and language interpreter well combines the commonsense of language models with the analytical abilities, structure, and interpretability of code. Such a technology can thus enable applications of code and code-like reasoning to novel problem regimes, beyond simple reasoning. Second, we are interested in investigating the degree to which finetuning a language model to be an LMulator can benefit semantic code reasoning. Third, we see evidence that reasoning through many pathways yields improvements, which is a promising step forward. Finally, we believe this integration with code enables access to external modalities, such as vision or databases, and represents a interesting path for new applications (e.g., robotics, augmented reality).

References

[1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[2] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michal Podstawski, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. *arXiv preprint arXiv:2308.09687*, 2023.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[5] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

[6] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[7] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[8] David Dohan, Winnie Xu, Aitor Lewkowycz, Jacob Austin, David Bieber, Raphael Gontijo Lopes, Yuhuai Wu, Henryk Michalewski, Rif A Saurous, Jascha Sohl-Dickstein, et al. Language model cascades. *arXiv preprint arXiv:2207.10342*, 2022.

[9] Iddo Drori, Sarah Zhang, Reece Shuttleworth, Leonard Tang, Albert Lu, Elizabeth Ke, Kevin Liu, Linda Chen, Sunny Tran, Newman Cheng, et al. A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level. *Proceedings of the National Academy of Sciences*, 119(32):e2123433119, 2022.

[10] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pp. 10764–10799. PMLR, 2023.

[11] Google Gemini Team. Gemini: A family of highly capable multimodal models. Technical report, Google, 2023. URL `https://storage.googleapis.com/deepmind-media/gemini/gemini_1_report.pdf`.

[12] Google, Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.

[13] Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint arXiv:2305.11738*, 2023.

[14] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406*, 2022.

[15] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, and Ross Girshick. Segment anything. *arXiv:2304.02643*, 2023.

[16] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35: 22199–22213, 2022.

[17] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models, 2022. *arXiv preprint arXiv:2206.14858*, 2022. URL `https://arxiv.org/abs/2206.14858`.

[18] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[19] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9493–9500. IEEE, 2023.

[20] Shilong Liu, Zhaoyang Zeng, Tianhe Ren, Feng Li, Hao Zhang, Jie Yang, Chunyuan Li, Jianwei Yang, Hang Su, Jun Zhu, et al. Grounding dino: Marrying dino with grounded pre-training for open-set object detection. *arXiv preprint arXiv:2303.05499*, 2023.

[21] Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, et al. Augmented language models: a survey. *arXiv preprint arXiv:2302.07842*, 2023.

[22] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837*, 2022.

[23] Suvir Mirchandani, Fei Xia, Pete Florence, Brian Ichter, Danny Driess, Montserrat Gonzalez Arenas, Kanishka Rao, Dorsa Sadigh, and Andy Zeng. Large language models as general pattern machines. *arXiv preprint arXiv:2307.04721*, 2023.

[24] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[25] Xuefei Ning, Zinan Lin, Zixuan Zhou, Huazhong Yang, and Yu Wang. Skeleton-of-thought: Large language models can do parallel decoding. *arXiv preprint arXiv:2307.15337*, 2023.

[26] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

[27] OpenAI. Gpt-4 technical report, 2023.

[28] Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023.

[29] Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.

[30] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*, 2023.

[31] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[32] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

[33] Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 11523–11530. IEEE, 2023.

[34] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022.

[35] Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. *arXiv preprint arXiv:2303.08128*, 2023.

[36] Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, et al. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*, 2022.

[37] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[38] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

[39] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*, 2023.

[40] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

[41] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

[42] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

[43] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. *arXiv preprint arXiv:2309.03409*, 2023.

[44] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[45] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*, 2023.

[46] Andy Zeng, Maria Attarian, Brian Ichter, Krzysztof Choromanski, Adrian Wong, Stefan Welker, Federico Tombari, Aveek Purohit, Michael Ryoo, Vikas Sindhwani, et al. Socratic models: Composing zero-shot multimodal reasoning with language. *arXiv preprint arXiv:2204.00598*, 2022.

[47] Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, et al. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification. *arXiv preprint arXiv:2308.07921*, 2023.

[48] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.

[49] Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew M Dai, Quoc V Le, James Laudon, et al. Mixture-of-experts with expert choice routing. *Advances in Neural Information Processing Systems*, 35:7103–7114, 2022.

A Appendix

A.1 Quantitative results on language reasoning tasks

Table A1 shows the full per-task results across ablations on BIG-Bench Hard (BBH) tasks, as well as broken down by task type and execution type.

Table A1: Full results across ablations on BIG-Bench Hard (BBH) tasks.

	Srivastava et al. [34] Suzgun et al. [36]					Chain of Code					
BIG-Bench Hard Task	Rand.	Human (Avg.)	Human (Max)	Direct	CoT	Inter-weave	try Python except LM state	try Python except LM	Python	LM state	LM
Boolean Expressions$^{\lambda+}$	50	79	100	88	89	100	100	100	100	95	90
Causal Judgement$^{\kappa*}$	50	70	100	64	64	56	57	63	0	57	60
Date Understanding$^{\kappa-}$	17	77	100	61	84	75	72	74	59	66	57
Disambiguation QA$^{\kappa/}$	33	67	93	70	68	71	67	68	0	67	68
Dyck Languages$^{\lambda+}$	1	48	100	6	50	100	100	99	99	1	7
Formal Fallacies$^{\kappa*}$	25	91	100	56	56	55	54	55	0	54	56
Geometric Shapes$^{\lambda+}$	12	54	100	48	66	100	100	100	100	13	44
Hyperbaton$^{\kappa/}$	50	75	100	63	64	98	62	55	0	62	55
Logical Deduction$^{\lambda*}$	23	40	89	49	66	68	79	57	0	79	58
Movie Recommendation$^{\kappa/}$	25	61	90	85	81	80	83	80	0	83	79
Multi-Step Arithmetic$^{\lambda+}$	0	10	25	0	48	100	100	100	100	0	1
Navigate$^{\lambda*}$	50	82	100	58	94	86	84	68	0	84	68
Object Counting$^{\lambda-}$	0	86	100	30	82	96	98	98	98	57	50
Penguins in a Table$^{\kappa-}$	0	78	100	62	82	90	88	90	88	71	59
Reasoning about Colored Objects$^{\kappa-}$	12	75	100	64	87	78	74	78	64	64	70
Ruin Names$^{\kappa/}$	25	78	100	76	70	55	56	46	0	56	47
Salient Translation Error Detection$^{\kappa/}$	17	37	80	66	61	58	63	64	0	63	64
Snarks$^{\kappa/}$	50	77	100	70	71	76	76	66	0	76	66
Sports Understanding$^{\kappa/}$	50	71	100	72	96	91	93	75	0	93	74
Temporal Sequences$^{\lambda*}$	25	91	100	38	60	98	93	99	93	93	99
Tracking Shuffled Objects$^{\lambda-}$	23	65	100	25	72	100	96	96	96	71	24
Web of Lies$^{\lambda-}$	50	81	100	54	100	97	96	96	97	96	50
Word Sorting$^{\lambda+}$	0	63	100	51	50	99	100	99	100	54	54
Task Averages											
NLP Task (avg)$^{\kappa}$	30	71	97	67	74	74	70	68	18	68	63
Algorithmic Task (avg)$^{\lambda}$	21	64	92	41	71	95	95	92	80	58	50
All Tasks (avg)	26	68	95	55	72	84	82	80	48	63	57
Execution Type											
Python exec (same program)$^{+}$	13	51	85	38	61	100	100	100	100	33	39
Python exec (different program)$^{-}$	17	77	100	49	84	89	87	89	84	71	52
LM exec (same program)$^{/}$	36	66	95	72	73	76	71	65	0	71	65
LM exec (different program)*	35	75	98	53	68	72	73	68	19	73	68

λ denotes an algorithmic task and κ denotes an NLP task (with categories outlined in Suzgun et al. [36]). $+$ denotes a task where the code between prompts is repeated and can be executed by Python, $-$ denotes a task where the code between prompts must change and can be executed by Python, $/$ denotes a task where the code between prompts is repeated and must be executed by the LM, and $*$ denotes a task where the code between prompts must change and must be executed by the LM.

A.2 Quantitative results on GSM8K tasks

Table A2 shows results on the grade school math (GSM8K) benchmark [7] with direct prompting, Chain of Thought, and Chain of Code. We find that CoC generally outperforms CoT and Direct prompting. Since these tasks are primarily algorithmic and are solved by Python alone, all Chain of Code variants that use Python achieve the same performance – also the same performance shown in Program of Thoughts [5].

A.3 Qualitative results on language reasoning tasks

Figure A1 shows the model outputs for a few reasoning tasks from BIG-Bench Hard (BBH) and Figure A2 shows a demonstrative example of date reasoning. These examples are selected to highlight the interweaving execution of the Python interpreter and the LMulator.

Table A2: GSM8K [7] performance (%) with both few-shot prompting with a single task and cross-task. The delta compared to direct prompting is shown in parenthesis.

Prompt	Direct	CoT	Chain of Code: Interweave	Chain of Code: try Python except LM state	Chain of Code: try Python except LM	Chain of Code: Python only	Chain of Code: LM state	Chain of Code: LM only
Single task	16	63 (47)	71 (55)	72 (56)	71 (55)	71 (55)	45 (29)	22 (6)
Cross task	14	55 (41)	60 (46)	60 (46)	60 (46)	60 (46)	41 (27)	16 (2)

A.4 Results on robotics tasks

For few-shot prompting, we include a single example: "Serve a meal that follows the user's dietary restrictions". During test time, we query the model with each of the following instructions.

- "Pack a lunch box for someone who is on a vegan diet."
- "Assemble a sandwich for someone who is vegetarian."
- "Gather ingredients for a peanut butter sandwich in a plate."
- "Prepare 西红柿炒蛋 in the pot." (interleaving English and Chinese on purpose)
- "Place all paper-made objects in the grass-colored container."
- "Sort the objects on the table into the compost bin and the recycle bin."
- "My steak is too bland. Can you help?"

Figure A3 shows the one-shot prompt as well as the model outputs and how they are executed for a few test instructions.

(a) Movie Recommendation

```
Q: Find a movie similar to Batman, The Mask, The Fugitive, Pretty Woman:
Options:
(A) The Front Page
(B) Maelstrom
(C) The Lion King
(D) Lamerica

options = {"The Front Page": "(A)", "Maelstrom": "(B)", "The Lion King": "(C)", "Lamerica": "(D)"}
delta state: {options = {'The Front Page': ..., ...}}
query_movies = list(options.keys())
delta state: {query_movies = ['The Front Page', ...]}
ref_movies = ["Batman", "The Mask", "The Fugitive", "Pretty Woman"]
delta state: {ref_movies = ['Batman', ..., 'Pretty Woman']}
ref_movie_infos = get_movie_genre_and_year(ref_movies, ret_type="list[tuple[str, int]]")
delta state: {ref_movie_infos = [('action', 1989), ...]}
most_similar_movie = get_most_similar_movie(query_movies, ref_movies, ref_movie_infos, ret_type="str")
delta state: {most_similar_movie = 'The Lion King'}
answer = options[most_similar_movie] if most_similar_movie in options else None
delta state: {answer = '(C)'}
```

(b) Hyperbaton

```
Q: Which sentence has the correct adjective order:
Options:
(A) small grey square cloth Japanese shoe
(B) small square grey Japanese cloth shoe"

import numpy as np
delta state: {}
options = {"(A)": "small grey square cloth Japanese shoe", "(B)": "small square grey Japanese cloth shoe"}
delta state: {options = {'(A)': ..., '(B)': ...}}
priority = {"opinion": 1, "size": 2, "age": 3, "shape": 4, "color": 5, "origin": 6, "material": 7, "purpose": 8}
delta state: {priority = {'opinion': .., .., 'purpose': ..}}
valid_types = list(priority.keys())
delta state: {valid_types = ['opinion', ..., 'purpose']}
scores = []
delta state: {scores = []}
for option, sentence in options.items():
delta state: {option, sentence = '(A)', 'small ... shoe'}
# updated for each loop
  adjs = sentence.split(" ")[:-1]
delta state: {adjs = ['small', 'grey', 'square', 'cloth']}
  order = [priority[get_adjective_type(adj, valid_types, ret_type=str)] for adj in adjs]
delta state: {order = [2, 5, 4, 6]}
  scores.append([order[i+1] > order[i] for i in range(len(order) - 1)].count(True))
delta state: {scores = [2]}
answer = list(options.keys())[np.argmax(scores)]
delta state: {answer = '(B)'}
```

(c) Logical Deduction

```
Q: The following paragraphs each describe a set of three objects arranged in a fixed order. The statements are logically consistent within each paragraph. On a shelf, there are three books: a green book, a red book, and a blue book. The red book is the rightmost. The blue book is to the right of the green book.
Options:
(A) The green book is the leftmost
(B) The red book is the leftmost
(C) The blue book is the leftmost

options = {"green": "(A)", "red": "(B)", "blue": "(C)"}
delta state: {options = {'green': ..., ..., 'blue': ...}}
order_info = "left to right"
delta state: {order_info = 'left to right'}
full_order = [None, None, None]
delta state: {full_order = [None, None, None]}
partial_order = []
delta state: {partial_order = []}
full_order[-1] = "red"
delta state: {full_order = [None, None, 'red']}
partial_order.append(("green", "blue"))
delta state: {partial_order = [('green', 'blue')]}
full_order = generate_full_order(full_order, partial_order, ret_type=list)
delta state: {full_order = ['green', 'blue', 'red']}
query = "leftmost"
delta state: {query = 'leftmost'}
result = query_result(order_info, full_order, query, ret_type=str)
delta state: {result = 'green'}
answer = options[result] if result in options else None
delta state: {answer = '(A)'}
```

(d) Disambiguation QA

```
Q: In the following sentences, explain the antecedent of the pronoun (which thing the pronoun refers to), or state that it is ambiguous.
Sentence: The homeowner asked the inspector if the house they had purchased was structurally sound.
Options:
(A) The homeowner had purchased
(B) The inspector had purchased
(C) Ambiguous

context = "The homeowner asked the inspector if the house they had purchased was structurally sound."
delta state: {context = 'The homeowner asked ... sound.'}
pronoun = "they"
delta state: {pronoun = 'they'}
a = "homeowner"
delta state: {a = 'homeowner'}
b = "inspector"
delta state: {b = 'inspector'}
version_a = "The homeowner asked the inspector if the house the homeowner had purchased was structurally sound."
delta state: {version_a = 'The homeowner asked ... sound.'}
version_b = "The homeowner asked the inspector if the house the inspector had purchased was structurally sound."
delta state: {version_b = 'The homeowner asked ... sound.'}
valid_a = can_pronoun_refer_to_noun(pronoun=pronoun, noun=a, full_sentence=version_a, ret_type=bool)
delta state: {valid_a = True}
valid_b = can_pronoun_refer_to_noun(pronoun=pronoun, noun=b, full_sentence=version_b, ret_type=bool)
delta state: {valid_b = False}
if valid_a and not valid_b:
delta state: {}
  answer = "(A)"
delta state: {answer = '(A)'}
elif valid_b and not valid_a:
  answer = "(B)"
else:
  answer = "(C)"
```

Figure A1: Model outputs for a few reasoning tasks from BIG-Bench Hard (BBH). We observe that CoC can apply to a wide variety of complex reasoning tasks that involve both semantic and numeric reasoning. Red highlight indicates LM generated code being executed by the Python interpreter, and purple highlight indicates LM simulating the code execution.

Direct answer only

Q: What holiday is 314 days after Valentine's Day in 2024?

A: Christmas (35%, ✗), New Year's (15%, ✗), Other (45%, ✗), Christmas Eve (0%, ✓)

Chain of Thought

Q: What holiday is 314 days after Valentine's Day in 2024? Let's think step by step

Step 1: Count the days in February after Valentine's Day: ...

A: Christmas Eve (35%, ✓), Christmas (5%, ✗), Nov X (20%, ✗), Jan X (20%, ✗), Other (20%, ✗)

Chain of Code

Q: What holiday is 314 days after Valentine's Day in 2024?

```
from datetime import date, timedelta
delta_state: {}
day1 = get_valentines_day_date(2024)
delta_state: {day1 = date(year=2024, month=2, day=14)}
day2 = day1 + timedelta(days=314)
delta_state: {day2 = date(year=2024, month=12, day=24)}
answer = get_holiday(day2)
delta_state: {answer = 'Christmas Eve'}
```

A: Christmas Eve (100%, ✓)

Figure A2: A demonstrative example of how Chain of Code generates code and reasons through an LM-augmented code emulator. Lines evaluated with Python are in red and with an LM are in purple. The chain of thought and direct answers were evaluated with gpt-4 in October 2023, and we note the current model (as of December 2023) writes code to solve this problem and gets the same solution as Chain of Code.

(a) Given Prompt

```
Q: Serve a meal that follows the user's dietary restrictions.

restrictions = input("Do you have any dietary restrictions?")
_, _, objs, _ = robot.detect_objects()
container_obj = None
for obj in objs:
  is_match_description = match_description(obj, "food container", ret_type="bool")
  if is_match_description:
    container_obj = obj
    break
if container_obj is None:
  robot.say("No food container found.")
else:
  for obj in objs:
    if obj == container_obj:
      continue
    is_follow_restrictions = follow_restrictions(obj, restrictions, ret_type="bool")
    if is_follow_restrictions:
      robot.pick_place(obj, container_obj)
```

(b) Novel Task

```
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)
```

(c) Novel Task

```
Q: Prepare 西红柿炒蛋 (stir-fried tomato and eggs) in the pot.

ingredients = ["西红柿", "鸡蛋"] ["tomato", "egg"]
_, _, objs, _ = robot.detect_objects()
pot_obj = None
for obj in objs:
  is_match_description = match_description(obj, "cooking pot", ret_type="bool")
  if is_match_description:
    pot_obj = obj
    break
if pot_obj is None:
  robot.say("No cooking pot found.")
else:
  for ingredient in ingredients:
    for obj in objs:
      is_match_description = match_description(obj, ingredient, ret_type="bool")
      if is_match_description:
        robot.pick_place(obj, pot_obj)
```

(d) Novel Task

```
Q: My steak is too bland. Can you help?

robot.say("Sure, let me see what I can do.")
_, _, objs, _ = robot.detect_objects()
for obj in objs:
  is_match_description = match_description(obj, "steak", ret_type="bool")
  if is_match_description:
    steak_obj = obj
    break
if steak_obj is None:
  robot.say("No steak found.")
else:
  robot.say("I'm going to season your steak with some salt and pepper.")
  robot.pick_place("salt", steak_obj)
  robot.pick_place("pepper", steak_obj)
```

Figure A3: The one-shot prompt as well as the model outputs for a few test instructions for the robotics tasks. When given a single example in the prompt (a), our method can generalize (b-d) to new objects, languages, and task domains. Red highlight indicates LM generated code being executed by the Python interpreter, and purple highlight indicates LM simulating the code execution. Gray text is for illustration purpose only, and not provided to our model. Note that code in the form of robot.<func_name> invokes robot APIs.

Figure A4: Full question used in Fig. 1

```
How many countries have I been to? I’ve been to Mumbai, London, Washington, Grand Canyon, Baltimore, Longsheng, Guilin, Beijing, Galapagos, Quito, Barcelona, Paris, Prague, Nice, Dehli, Agra, Rome, Florence, Amalfi, Athens, Míkonos, Málaga, Monaco, Berlin, Munich, Innsbruck, Bern, Milan, Lucerne, Gimmelwald (Schilthornbahn), St Moritz, St Petersburg, Helsinki, Amsterdam, Gdańsk, Vancouver, Anchorage, Montreal, Belize, The Bahamas, Jamaica, Hawaii, Acadia National Park, Stockholm, Copenhagen, Dover, Lyon, Madrid, Toulouse, Santorini, Oslo, Kusadasi, Souda, Rhodes, Tallinn, Venice, Vatican City, Naples, Cape Town, Johannesburg, Addis Abeba, Nairobi, Seattle, San Francisco, Chicago, St Louis, Memphis, Chinle, Stanford, New York, Philadelphia, Boston, Miami, New Orleans, Walt Disney World Resort, Jacksonville, Las Vegas, Los Angeles, Portland, Salt Lake City, Tahoe City, Phoenix, Albuquerque, Cleveland, Charlottesville, Nags Head, Newfoundland and Labrador, Burlington, Wilmington, Myrtle Beach, St Lucia, Barbados, Grenada, Banff, Haiti, Montego Bay, Sao Palo, Rio, Lima, Cusco, Cozumel, Amarillo, Yosemite National Park, Joshua Tree, Zion National Park, Bryce Canyon National Park, Grand Teton National Park, Yellowstone National Park, Glacier National Park, Mount Hood, Paso Robles, San Diego, Bend, North Cascades National Park, Olympic National Park Visitor Center, Jasper National Park, Sequoia National Park, Kings Canyon National Park, Shasta National Forest, Mount Saint Helens, Mount Rainier, Austin, Buenos Aires, El Calafate, El Chaltén, Fitz Roy, Torres del Paine National Park, Puerto Natales, Puerto Varas, Santiago, Marble Caves, Cerro Castillo, Coyhaique, Singapore, Casablanca, Marrakesh, Cairo, Jerusalem, Tokyo, Kyoto Prefecture, Taipei City, Taichung City, Krk, Naturpark Puez-Geisler, Ljubljana, Plitvice Lakes National Park, Fairbanks, Juneau, Dallas, Sydney, Cairns, Brisbane, Hook Island, Charleston, Panama City, Bangkok, Chiang Mai, Bengaluru, Denver, Indianapolis, Nashville, Blacksburg, Lisbon, Porto, Estes Park, Coeur d’Alene, Hood River, Denali, Sitka, Mexico City, Warsaw, Geneva, Auckland, Queenstown, Whitefish, Minneapolis, Sioux Falls, Bozeman, Missoula, Springfield, Skye, Edinburgh, Honolulu, Kauai, Haleakalā National Park, Wrangell-St. Elias National Park & Preserve, Atlanta, Tirana, Corfu, Siena.
```