MAS A3
Ishaan Parmar

All the Plots are saved in the results folder and all the terminal outputs are saved in terminal_outputs folder

Part 1 a)

What is a good model for moving the agent randomly?

A good model, which is the one used in the code, is a **"uniform random walk"**.

In simple terms, this means that in every step, the agent picks:

1. A random direction (any angle from 0 to 360 degrees).
2. A random distance to travel (anywhere from 0 to its max speed of 25).

This is a good choice because it makes the agent explore the entire 1000x1000 map evenly over time, without favoring any particular direction.

Plot the number of tasks the agent completes per iteration.

Here is the plot from the simulation run:

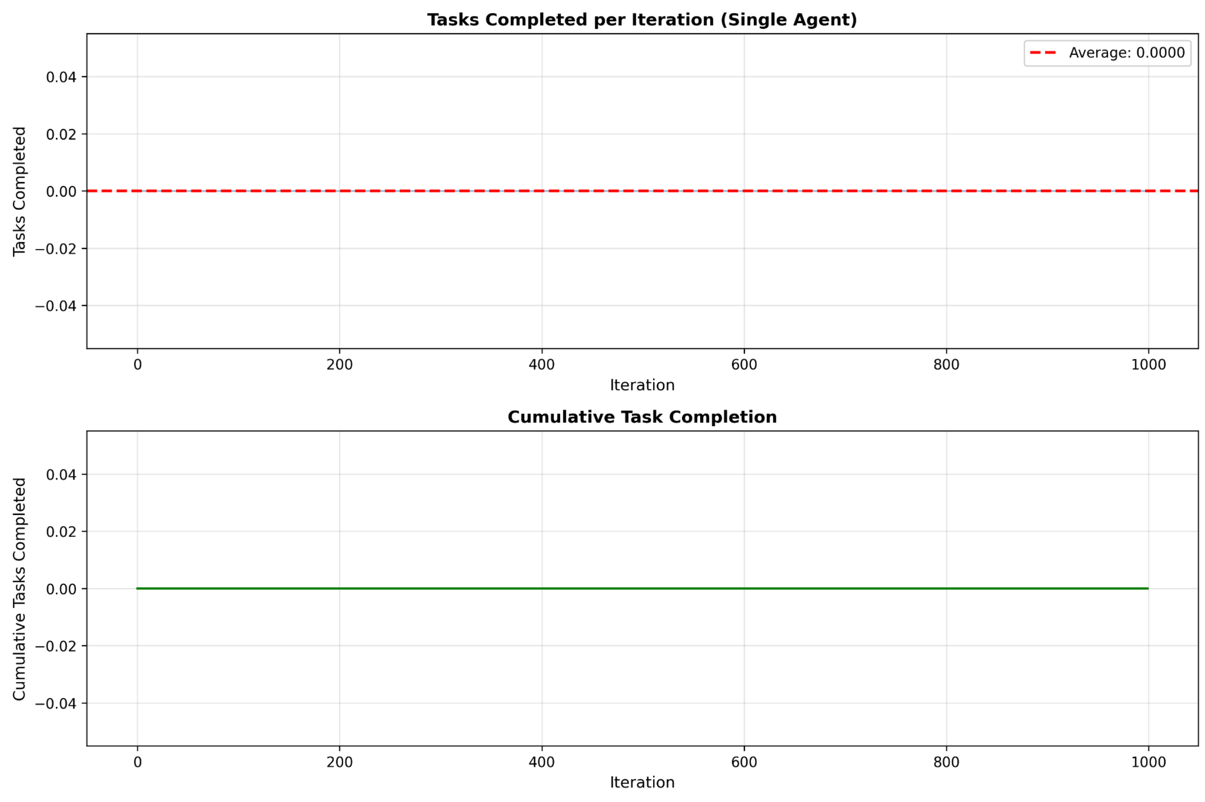

Fig: Part 1 a

As you can see from the plot, and from your terminal output, the agent **completed 0 tasks** in the 1000 iterations. The "Average: 0.0000" line confirms this. This isn't surprising, as the map is very large (1000x1000) and the task area is small (T_r=50), so the chances of a single agent randomly stumbling upon it are very low.

Is "tasks per iteration" a good measure for performance?

It depends on how you look at it:

- **The plot itself (top graph) is not a good measure.** For a single agent, it's too "noisy" because at any given moment it can only be 0 (no task) or 1 (task completed). A flat line at 0 doesn't tell us much.
- The **"Average tasks per iteration"** (like the 0.0000 from your output) is a **much better measure**. It smooths out the noise and gives a single, stable number that shows how efficient the agent is over the long run. The cumulative plot (bottom graph) is also useful for seeing this overall efficiency.

So, the *average* rate is a good metric, but the *per-iteration plot* isn't very helpful on its own.

Part 1 b)

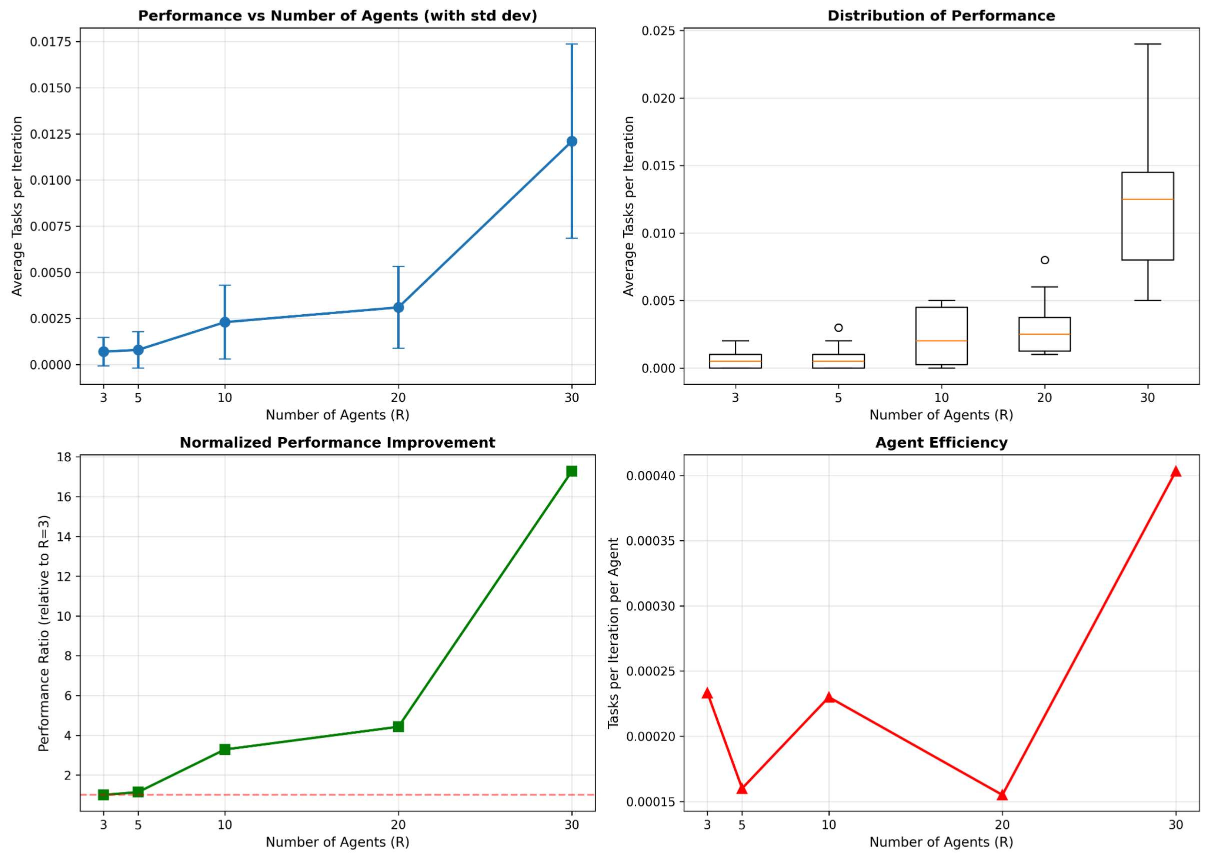

Fig: Part 1 b

Part 1 c)

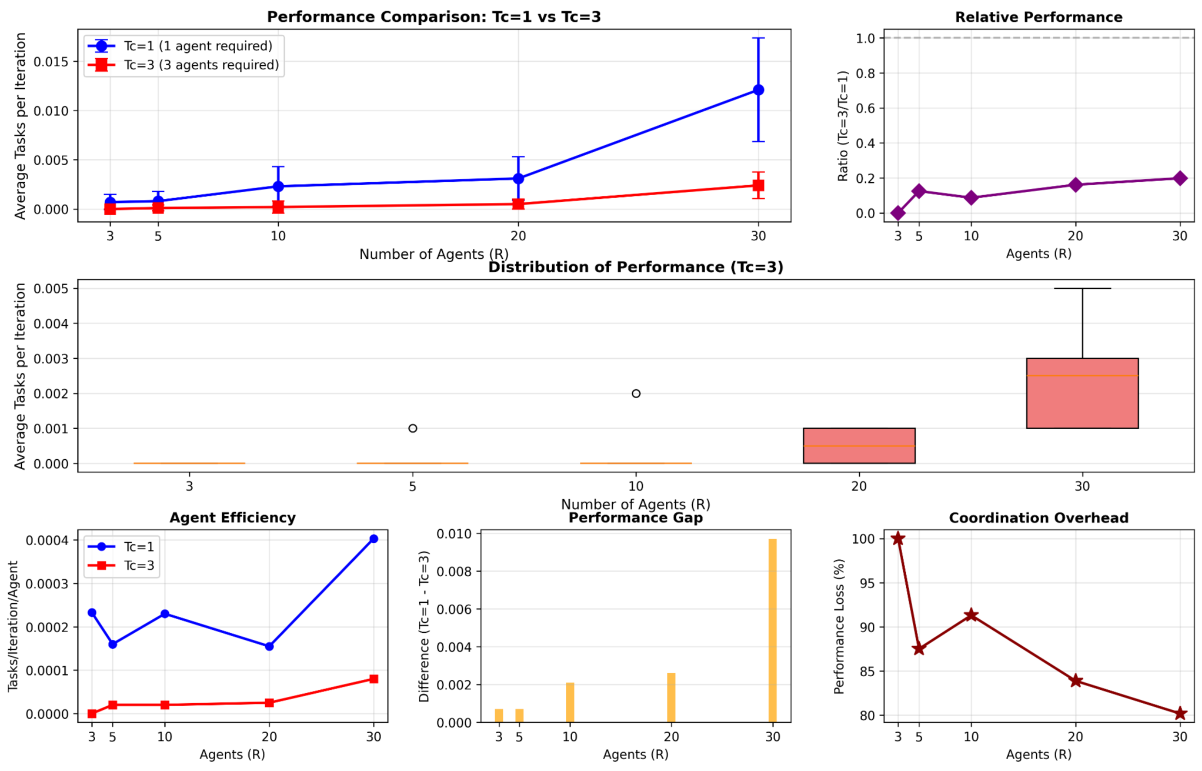

Fig: Part 1 c

Part 1 d)

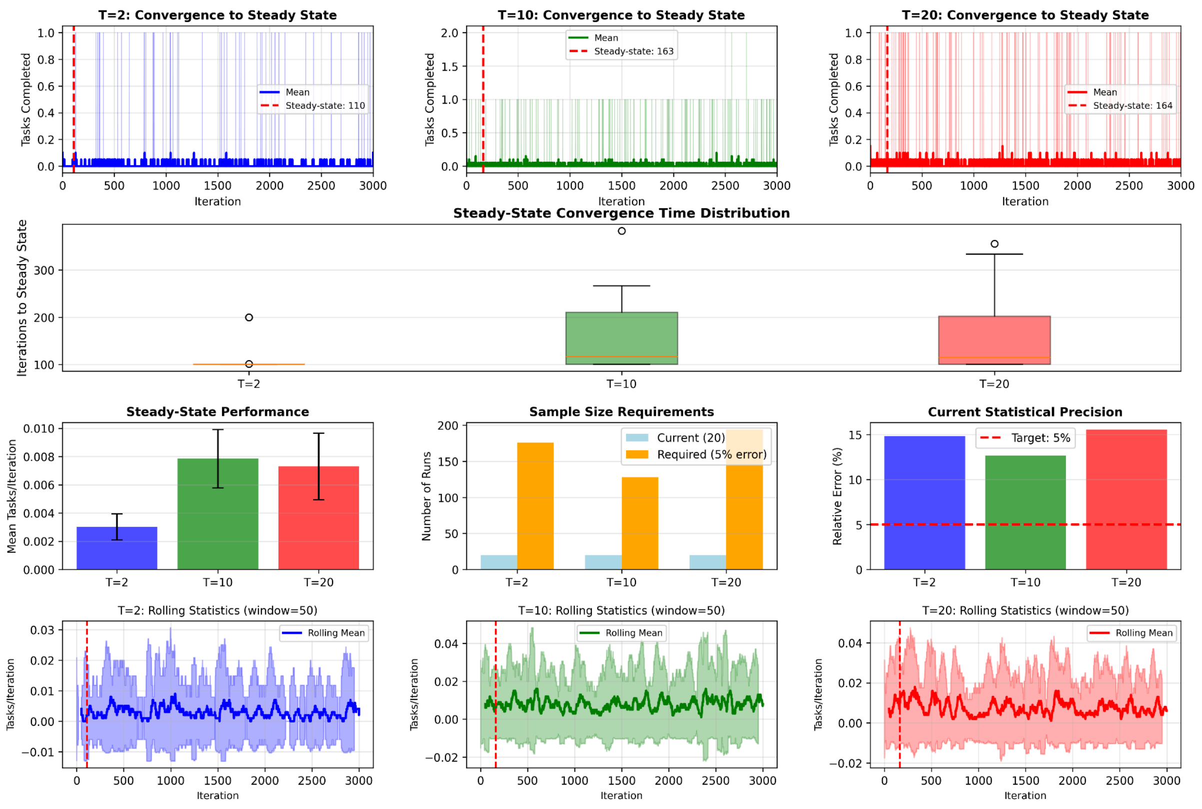

Fig: Part 1 d

What happens when you increase the number of tasks?

When you increase the number of tasks, the **total number of tasks completed per iteration goes up**, but only to a point.

- At **T=2** (2 tasks), performance is low: **0.0030**.
- At **T=10** (10 tasks), performance is best: **0.0078**.
- At **T=20** (20 tasks), performance levels off or even drops a bit: **0.0073**.

This is shown clearly in the **"Steady-State Performance"** bar chart. The reason, as your output notes, is that the 30 agents get spread too thin. The *per-task* completion rate actually drops a lot as you add more tasks.

How many iterations for a "steady state"?

To get stable results, you need to let the simulation "warm up." Based on your analysis, it takes about:

- **110 iterations** for T=2
- **163 iterations** for T=10
- **164 iterations** for T=20

So, your output's recommendation is to use **164 iterations as the "warm-up" period** and discard any data from before that point. The analysis suggests running for a **total of 1164 iterations** (164 for warm-up + 1000 for measurement).

How many simulations for a "good statistical estimation"?

A "good estimation" is defined as having less than 5% relative error.

- Your current **20 runs** (the light blue bar in the "Sample Size" chart) gives a high error of 12-15%.
- To *actually get* a 5% error, you would need **128 to 194 runs** (the big orange bar in the chart).

However, your analysis also provides a practical recommendation for the rest of the assignment, suggesting that **20 runs** is a "good balance" and will be used as the standard moving forward.

The "Random" Benchmark

Your analysis is spot on: this "no communication" case is the **"random benchmark"**. The performance numbers you got here (like 0.0030 for T=2) are the baseline scores that the new communication protocols in Parts 1(e) and 1(f) will have to beat.

Part 1 e)

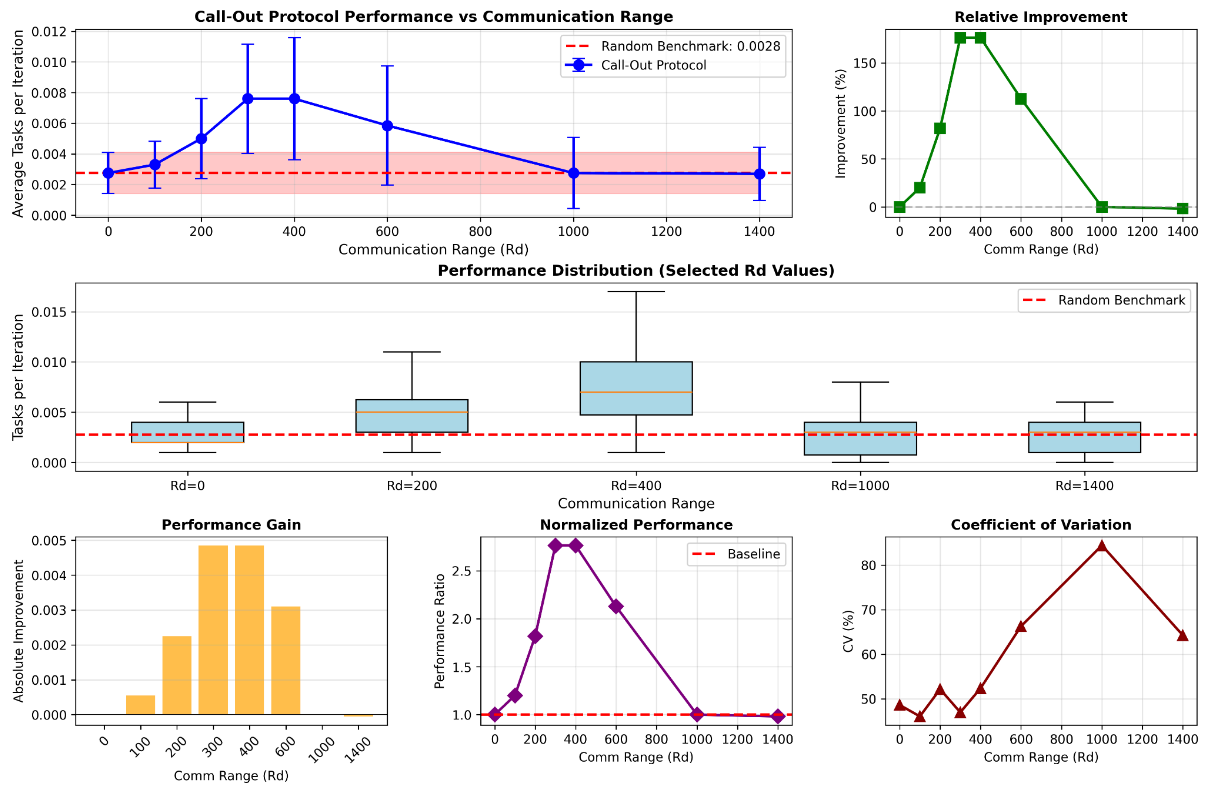

Fig: Part 1 e

Simulation Plot

This plot shows the system performance for the "Call-Out" protocol at different communication ranges (R_d). The top-left chart is the main result, showing the "Call-Out Protocol" (blue line) compared to the "Random Benchmark" (red dashed line).

Comments on Findings

Based on the plots and your terminal output, here are the key findings:

- **Communication Works (A Lot!):** The "Call-Out" protocol is a huge success. The random benchmark (no communication) averaged **0.0028** tasks/iteration, but with "Call-Out," the performance peaked at **0.0076**. This is a **176.36% improvement**, as seen in the "Relative Improvement" chart.
- **There is an Optimal Range:** The best performance isn't at the maximum range. There's a "sweet spot".
  - **Too Small (R_d = 100-200):** You get some improvement (20-80%), but you can't recruit enough agents from a small area.
  - **Just Right (R_d = 300-400):** This is the peak. The range is large enough to find the 2 extra helpers needed for the task (T_c=3).

    - **Too Large (R_d = 1000-1400):** Performance *drops* dramatically, becoming just as bad as the random benchmark.
- **The "Committed Agent Problem":** The reason performance dies at large ranges is a key limitation of this protocol. When an agent responds to a signal, it's "locked in" for all R_t=60 iterations.
    - At R_d=1000, *too many* agents (e.g., 10 agents) might respond to a single task that only needs 3.
    - The 7 extra agents are now all locked, wasting their time for 60 iterations instead of searching for new tasks.
    - This "over-recruitment" and "wasted effort" is why the protocol fails at large ranges and what motivates the improvement in Part 1(f).

Part 1 f)

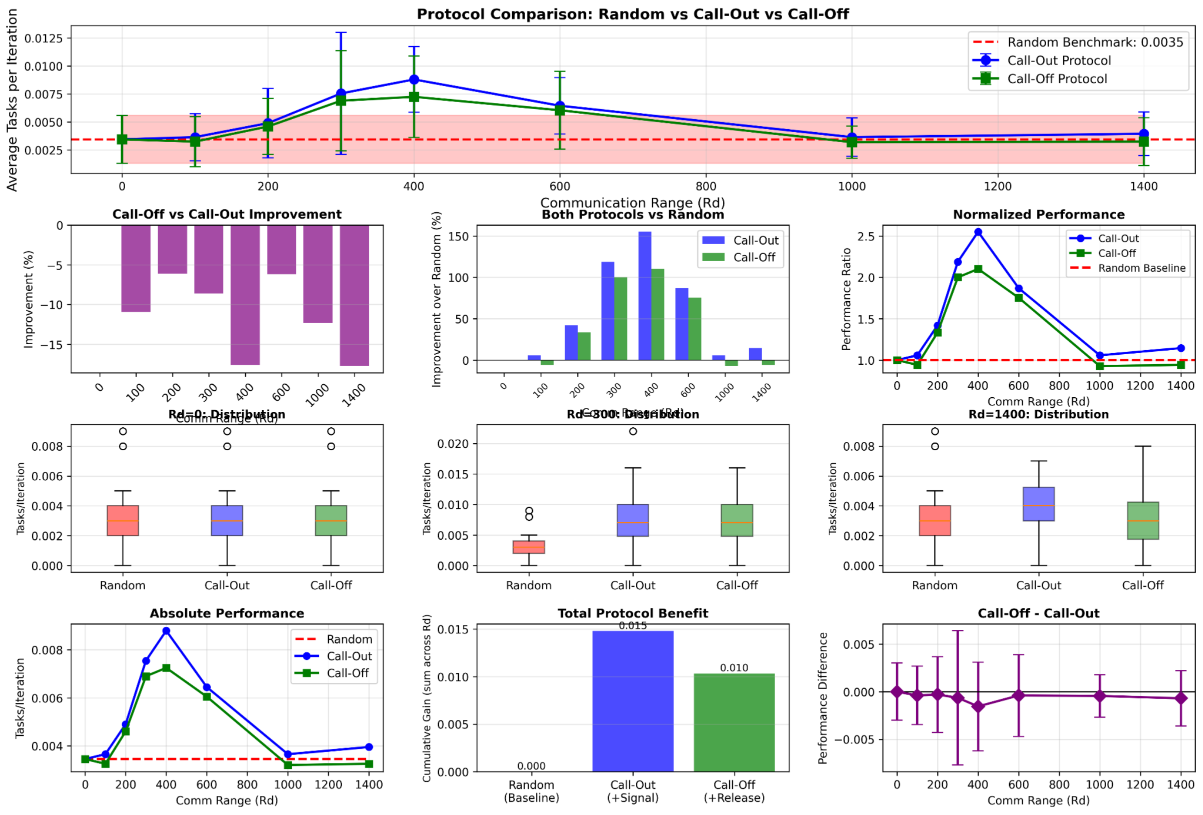

Fig: Part 1 f

Simulation Plot

This plot compares three protocols: the "Random Benchmark" (red dashed line), the "Call-Out" protocol from 1(e) (blue line), and the new "Call-Off" protocol (green line).

Comments on Findings

Based on the plots and the "COMPREHENSIVE COMPARISON TABLE" from your output, the findings are very interesting and show a clear story.

- **Both Protocols Beat Random:** Both "Call-Out" and "Call-Off" (the blue and green lines) are much better than the "Random Benchmark" (the red line) in the optimal range.
  - Random: 0.0035 tasks/iteration.
  - Best "Call-Out": 0.0088 (a **155% improvement**).
  - Best "Call-Off": 0.0073 (a **110% improvement**).
- **"Call-Out" Wins:** The most important finding from your run is that the original **"Call-Out" protocol (blue line) is better than the "Call-Off" protocol (green line)** at every single communication range ($R_d > 0$).
- **"Call-Off" is Worse:** The "Call-Off" protocol was *supposed* to be an improvement, but your data shows it's actually worse.
  - At the optimal range of R_d=400, "Call-Off" (0.0073) performed **-17.61% worse** than "Call-Out" (0.0088).
  - The "Call-Off vs Call-Out Improvement" chart (top-middle) shows this clearly, with all bars being negative.

Conclusion From Your Data: Based on *this* simulation run, adding the "Call-Off" signal (to release agents early) **hurts performance** compared to just letting them stay committed for 60 iterations. The original "Call-Out" protocol is the superior one.

Part 2 a)

The part2a_data.npy file saves all the raw data from this experiment (the mean, standard deviation, and all 20 run results for both "Auction" and "Random") so it can be loaded later for the comparison plots.

(a) Could you implement and test a simple auction?

As your terminal output's analysis section explains, the implementation works exactly as described in the assignment:

- An agent that finds a task becomes the "Auctioneer."
- It asks all agents within its communication range (R_d) for bids.
- Agents bid with their "distance to the auctioneer."
- The auctioneer hires the T_c-1 (so, 2) agents with the lowest bids (the closest ones).

Simulation Results

Here is the plot of the test results:

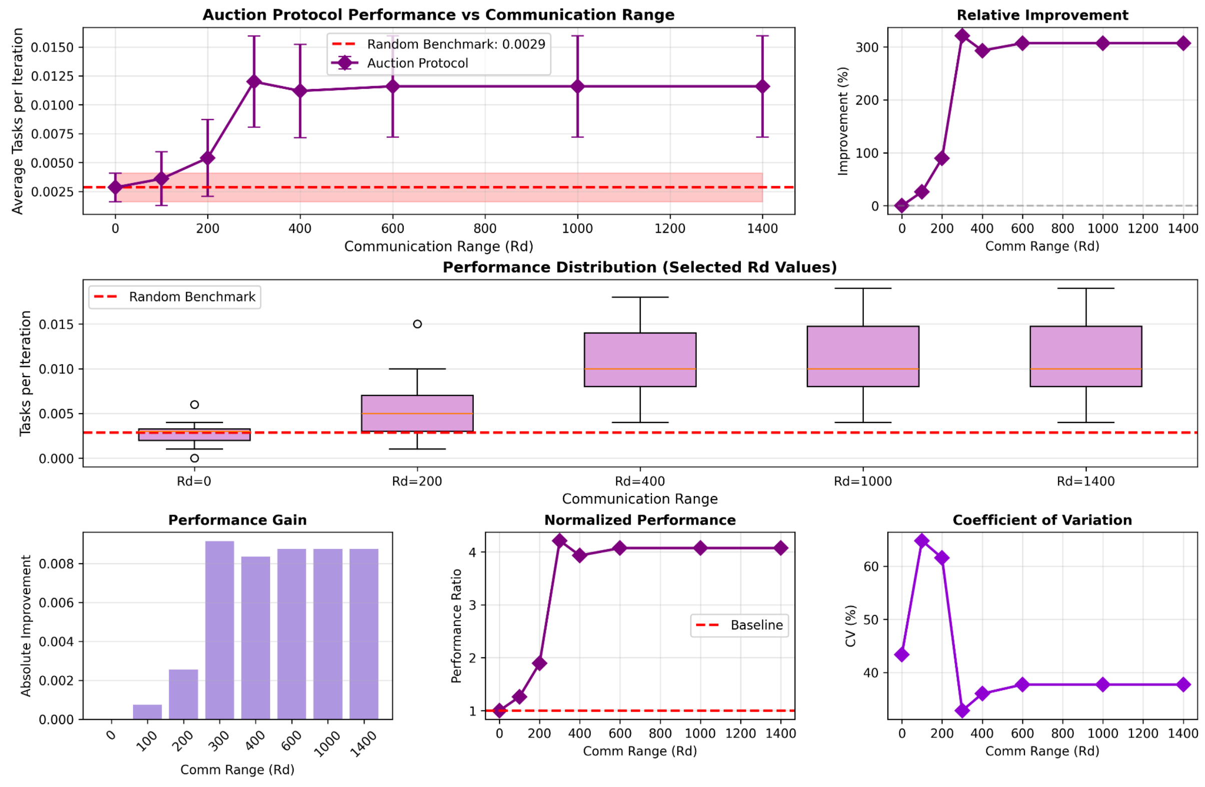

Fig: Part 2 a

Here are the key findings from the test, which you can see in the plots and your terminal output:

- **Massive Improvement:** The auction protocol is a huge success. The "Random Benchmark" was only **0.0029** tasks/iteration, but the auction protocol peaked at **0.0120** (at R_d=300). That's a **321% improvement**, as shown in the "Relative Improvement" chart.
- **The "Sweet Spot":** The performance dramatically jumps up at R_d=300. This seems to be the "sweet spot" where the communication range is large enough to find good (close) bidders, leading to a big performance gain.
- **It Plateaus:** Unlike the "Call-Out" protocol that got worse at high ranges, the auction protocol's performance stays high. After R_d=300, the performance stays level at around 0.011 to 0.012. This is because even if *more* agents bid, the auctioneer still only picks the *closest* 2, so there's no "over-commitment" problem.

Part 2 b)

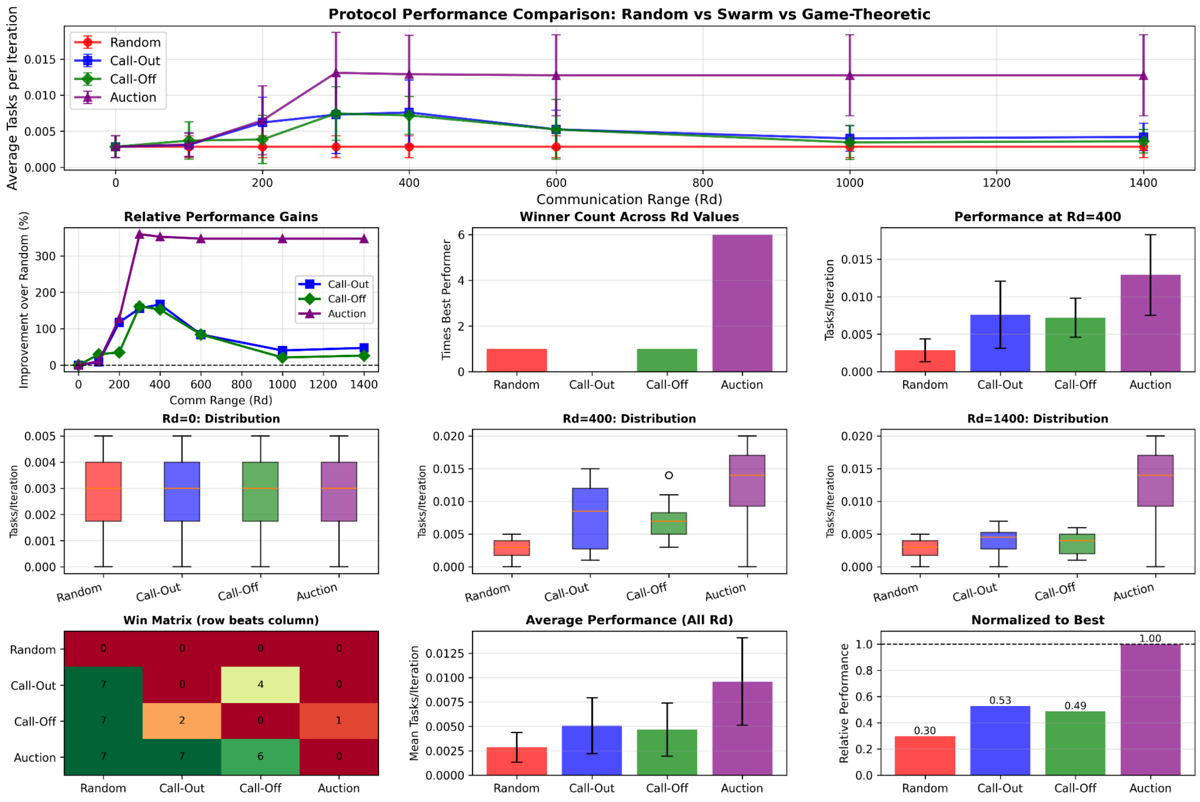

Fig: Part 2 b

Comparison Plot

This plot, which simulation generated, shows all four protocols together.

Comments on Findings

Based on your terminal output and this plot, the findings are very clear:

- **The "Auction" protocol is the clear winner, and it's not even close.** Just look at the plot: the purple "Auction" line is way above the "Call-Out" (blue) and "Call-Off" (green) lines at almost every communication range.
- **Auction is *much* better.** Your data shows the best swarm protocol ("Call-Out") peaked at **0.0076** tasks/iteration, but the "Auction" protocol peaked at **0.0131**. This means the auction method is **~72% better** than the best swarm method.
- **Here's why:** The swarm protocols (Call-Out/Off) get worse at high communication ranges (R_d) because of "over-commitment" (too many agents respond and get stuck). The Auction protocol *solves* this. Even if 20 agents are in range, the auctioneer only picks the *closest 2* and the other 18 are free to keep searching. This is way more efficient.

- **Call-Out vs. Call-Off:** In this simulation run, "Call-Out" (best at 0.0076) and "Call-Off" (best at 0.0075) performed almost identically, with "Call-Out" being just *slightly* better. Both were completely outperformed by the Auction.

Final Ranking (based on your run):

1. **Auction (0.0131)**
2. Call-Out (0.0076)
3. Call-Off (0.0075)
4. Random (0.0029)

Part 2 c)

How to Compare the Results (Considering Cost)

The assignment asks us to make "strategic" agents (auctioneers) cost **2 units** and "reactive" agents (helpers) cost **1 unit**. Your simulation set this up perfectly:

- **Swarm (Call-Out/Off):** A task just needs 3 reactive agents.
    - Cost = 1 + 1 + 1 = **3 units**.
- **Auction:** A task needs 1 strategic agent (the auctioneer) and 2 reactive helpers.
    - Cost = 2 (for the auctioneer) + 1 + 1 = **4 units**.

This means the **Auction protocol is 33.3% more expensive** every time a task gets done.

To compare them fairly, your script made a new metric called **"Cost-Adjusted Performance,"** which is just the `(Tasks per Iteration) / (Cost per Task)`. This is the best way to see which one gives the most performance for its cost.

The Plot

This plot you generated shows the whole story:

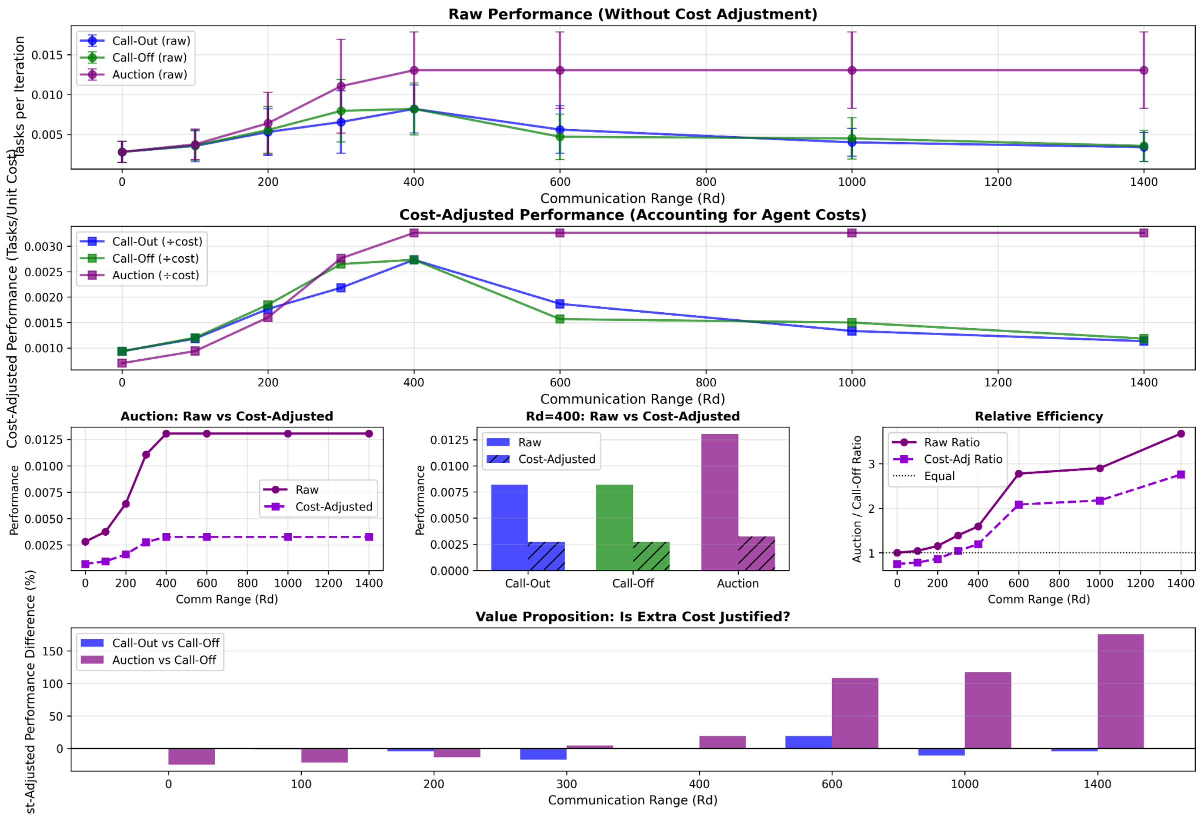

Fig: Part 2 c

Comments on the Findings

This is where it gets really interesting. Based the terminal output and the plots, the answer is super clear:

1. **Auction is *Way* Better at Raw Performance:** First, just look at the **top chart** ("Raw Performance"). The purple "Auction" line crushes the other two. Your output shows that the auction's best (0.0131) is **86.5% better** than the best swarm protocol (Call-Off at 0.0082).
2. **Auction *Still* Wins, Even After Adjusting for Cost:** This is the main test. You'd think the 33.3% extra cost would hurt it, but the auction is so much better that it doesn't matter. Look at the **middle chart** ("Cost-Adjusted Performance"). The **purple "Auction" line is *still* clearly on top** of the green "Call-Off" line. Your data shows that even after paying the extra cost, the auction is **still 39.9% better**.
3. **The "Break-Even" Verdict:** Your analysis sums it up perfectly:
   - **Extra Cost:** Auction costs **33.3%** more.
   - **Extra Performance:** Auction delivers **86.5%** more.
   - **Conclusion:** Since the performance gain is *way bigger* than the cost, **"YES - Auction pays for itself"**.

The final recommendation **Auction**