

A Appendix

A.1 Algorithm Details

We provide additional implementation details for EurekaVerse below. We first describe the core algorithm loop in pseudocode. Next, we describe “soft” policy selection, a more robust way to select the best policy using evaluation on the proxy (all generated training) environments.

Algorithm 1 EurekaVerse

```

1: Require: RL algorithm  $\mathcal{A}$ , coding LLM  $\text{LLM}$ , performance criteria  $F$ , LP transformation  $G$ 
2: // Generate initial environments
3:  $\theta_j^i \sim \Lambda_{\text{LLM}}^{\text{init}}(\theta_{\text{incontext}})$ 
4: // Run iterations of co-evolution
5: for generation iteration  $t = 1$  to  $T$  do
6:   // Train multiple policies on environments
7:   for run  $i = 1$  to  $M$  do
8:      $\pi_i = \mathcal{A}(\{\theta_j^i\}_{j=1}^J)$ 
9:   // Construct evaluation environment and select best policy
10:   $\Theta_{\text{proxy}} = \cup_i \{\theta_j^i\}_{j=1}^J$  across iterations
11:   $\pi_{\text{best}} = \arg \max_{\pi_i} F(\pi_i, \Theta_{\text{proxy}})$ 
12:  // Evolve environments
13:   $\hat{\theta}_j^{\text{best}} \sim \Lambda_{\text{LLM}}^{\text{evol}}(\theta_j^{\text{best}})$ 
14:  // Sample environments with learning progress
15:   $\text{LP}(\theta_j^i) := G(V^{M_j^i}(\pi^i), V^{M_j^i}(\pi^{\text{initial}}))$ 
16:   $\theta_j \sim \text{LP}(\theta)$ 
17: Output: best final policy  $\pi^{\text{best}}$ 

```

Soft Selection. During co-evolution, we select the best policy π^{best} via performance on all generated environments Θ_{proxy} , which tests the policies’ generalization and serves as a proxy for the true testing environment. Empirically, we find that this proxy accurately orders the policies, where the best policy in Θ_{proxy} also performs the best in our simulation benchmark. Nonetheless, there are rare cases where the best-performing benchmark policy is not the best one chosen by the proxy. To increase robustness against these inaccuracies, we “softly” select the best policy during each iteration of co-evolution: the selected policy is chosen randomly, where the best policy has probability p_1 , second-best has probability $p_2 < p_1$, and so on. Each agent in the next iteration performs this selection independently (so that Np_1 are initialized to the best policy, Np_2 are initialized to the second-best, etc.). In practice, we select $p_1 = 0.75, p_2 = 0.25, p_3, \dots, p_N = 0$.

A.2 Prompts and Examples

```

You are a reinforcement learning engineer trying to write environment functions as effectively
as possible for a quadruped robot parkour task. Please keep in mind that this robot will
be trained in simulation and deployed in a real world obstacle course, so we want the
obstacles to be realistic and challenging.

To do so, you should perform the following steps:
1. Carefully read the problem statement, specifications, and tips below.
2. Explain what your obstacle course will look like. Keep in mind that your obstacle sizes
   should be relative to the quadruped’s size, and the entire obstacle should fit within the
   terrain bounds. You should be creative when designing your obstacles, and You may draw
   inspiration from features seen in dog parks, playgrounds, and urban environments.
3. Write a function that generates the obstacle course according to your plan.

Problem Statement:
You are given a quadruped robot that needs to navigate through an obstacle course in 3D space.
The floor of the course is represented as a 2D numpy array, height_field, where each
element represents the height of the ground at that point, in meters. To direct the
quadruped, the course has 8 goals that the quadruped must reach in order. The goal
locations should be stored in a 2D numpy array, goals, as (x, y) indices in height_field.

To create your course, you will write a function set_terrain(length, width, field_resolution,
difficulty) that creates and returns the height_field and goals arrays. The function
takes in the length and width of the course, in meters, the scale of quantization for the

```

```

515     obstacle course, in meters, and the difficulty of the obstacle course, a float between 0
516     (easiest) and 1 (hardest) inclusive. Please follow the template below to write your
517     function:
518     ```python
519     import numpy as np
520     import random
521
522     def set_terrain(length, width, field_resolution, difficulty):
523         """Description of your course and the tested skill."""
524
525         def m_to_idx(m):
526             """Converts meters to quantized indices."""
527             return np.round(m / field_resolution).astype(np.int16) if not (isinstance(m, list) or
528             isinstance(m, tuple)) else [round(i / field_resolution) for i in m]
529
530         height_field = np.zeros((m_to_idx(length), m_to_idx(width)))
531         goals = np.zeros((8, 2))
532
533         # Your code here
534
535         return height_field, goals
536     ```
537
538     You can use any function from the numpy and random libraries as well as any in-built Python
539     functions. Please write everything as Python code and annotate your code with comments,
540     including a one-line docstring after the function definition that summarizes your
541     obstacle course.
542
543     Environment Specifications:
544     1. height_field quantizes the ground plane into a grid of points. field_resolution = 0.05, in
545     meters, is the quantization of the ground's (x, y) axes, which represents the distance
546     between adjacent points in the height_field array.
547     2. The ground plane size is 12 x 4 meters, so the height_field.shape = (12 / field_resolution,
548     4 / field_resolution) = (240, 80). Please make sure that your obstacle course spans
549     exactly these dimensions.
550     3. The quadruped's standing size (length, width, height) is 0.645 x 0.28 x 0.4 meters. Keep
551     these dimensions in mind when designing the size of your obstacles.
552     4. The quadruped will spawn with its center at (x, y, z) = (1, width / 2, 0) meters. You must
553     place obstacles at indices with x >= 2 / field_resolution to avoid the quadruped spawning
554     inside an obstacle.
555     5. Please make sure that your obstacles have a width of at least 1 meter. However, in rare
556     cases, we will allow narrow obstacles with length and width of at least 0.4 meters and no
557     smaller, as long as the quadruped is not expected to climb up or down.
558     6. The goal coordinates should be within the bounds of the course. Even if you have fewer than
559     8 obstacles, you must set all 8 goal positions in the goals array.
560     7. The quadruped will be instructed to travel in a straight line from one goal to the next.
561     Thus, if you want the quadruped to turn, you must place a goal at the turning point.
562     8. Your course should test a particular one of the robot's skills, for example by repeating a
563     single type of obstacle. Make the course relatively consistent throughout. There will be
564     other courses to test other skills and different obstacles.
565     9. The heights in field_height can be negative. The quadruped's spawning area will always be
566     at a height of 0 meters, so you can use negative heights to create pits or gaps in the
567     terrain. Use this to force the quadruped to walk or jump across the tops of your
568     obstacles without climbing up or down.
569     10. Since each index in the terrain has one height value, it is impossible to create overhangs
570     or tunnels. Do not try to create these features in your course.
571
572     Tips:
573     1. Do not call your function. Only write the function definition and the code inside it.
574     2. To broadcast a 1D array to a 2D array (or 2D slice), you must first add a new axis to the 1
575     D array.
576     3. You should not add noise directly to height_field. Random noisy terrain is not necessary
577     for our task, and we do not want it as an obstacle.
578     4. When slicing into height_field, make sure to convert from meters to quantized units
579     beforehand. For example, a 2 x 2 meter slice looks like [x - m_to_idx(1):x + m_to_idx(1),
580     y - m_to_idx(1):y + m_to_idx(1)].
581     5. If you write helper functions, please make them nested functions of set_terrain. Your
582     response should be a self-contained function definition following our template.
583

```

Listing 1: System Prompt. In the function template, we provide a conversion function `m_to_idx` to simplify the conversion from coordinates, in meters, to the index in the array.

```

584
585     The following is an example of a terrain generation function. Please reference the example
586     provided, but make your terrain different.
587
588     ```
589     import numpy as np
590
591     def set_terrain(length, width, field_resolution, difficulty):

```

```

592     """Multiple platforms traversing a pit for the robot to climb on and jump across."""
593
594     def m_to_idx(m):
595         """Converts meters to quantized indices."""
596         return np.round(m / field_resolution).astype(np.int16) if not (isinstance(m, list) or
597                               isinstance(m, tuple)) else [round(i / field_resolution) for i in m]
598
599     height_field = np.zeros((m_to_idx(length), m_to_idx(width)))
600     goals = np.zeros((8, 2))
601
602     # Set up platform dimensions
603     # We make the platform height near 0 at minimum difficulty so the quadruped can learn to
604     # climb up
605     platform_length = 1.0 - 0.3 * difficulty
606     platform_length = m_to_idx(platform_length)
607     platform_width = np.random.uniform(1.0, 1.6)
608     platform_width = m_to_idx(platform_width)
609     platform_height_min, platform_height_max = 0.0 + 0.2 * difficulty, 0.05 + 0.25 *
610     difficulty
611     gap_length = 0.1 + 0.7 * difficulty
612     gap_length = m_to_idx(gap_length)
613
614     mid_y = m_to_idx(width) // 2
615
616     def add_platform(start_x, end_x, mid_y):
617         half_width = platform_width // 2
618         x1, x2 = start_x, end_x
619         y1, y2 = mid_y - half_width, mid_y + half_width
620         platform_height = np.random.uniform(platform_height_min, platform_height_max)
621         height_field[x1:x2, y1:y2] = platform_height
622
623     dx_min, dx_max = -0.1, 0.1
624     dx_min, dx_max = m_to_idx(dx_min), m_to_idx(dx_max)
625     dy_min, dy_max = -0.4, 0.4
626     dy_min, dy_max = m_to_idx(dy_min), m_to_idx(dy_max)
627
628     # Set spawn area to flat ground
629     spawn_length = m_to_idx(2)
630     height_field[0:spawn_length, :] = 0
631     # Put first goal at spawn
632     goals[0] = [spawn_length - m_to_idx(0.5), mid_y]
633
634     # Set remaining area to be a pit
635     # We do this to force the robot to jump from platform to platform
636     # Otherwise, the robot can just jump down and climb back up
637     height_field[spawn_length:, :] = -1.0
638
639     cur_x = spawn_length
640     for i in range(6): # Set up 6 platforms
641         dx = np.random.randint(dx_min, dx_max)
642         dy = np.random.randint(dy_min, dy_max)
643         add_platform(cur_x, cur_x + platform_length + dx, mid_y + dy)
644
645         # Put goal in the center of the platform
646         goals[i+1] = [cur_x + (platform_length + dx) / 2, mid_y + dy]
647
648         # Add gap
649         cur_x += platform_length + dx + gap_length
650
651     # Add final goal behind the last platform, fill in the remaining gap
652     goals[-1] = [cur_x + m_to_idx(0.5), mid_y]
653     height_field[cur_x:, :] = 0
654
655     return height_field, goals
656
657

```

Listing 2: Initial Example. This example demonstrates the main structure of a terrain function, including helper functions and a for loop over multiple obstacles.

```

658
659 We trained a quadruped policy to perform parkour on the obstacle course created by the
660 generation code above (as well as others), and we tracked the values of individual reward
661 components as well as other metrics such as the number of goals reached (out of 8),
662 episode length, and the rate of edge violations (feet too close to edge of obstacles):
663 <INSERT POLICY STATISTICS HERE>
664
665 We have also computed statistics for the terrain height of the direct path between goals
666 across multiple difficulties. Note that this is a heuristical shortest-path between goals
667 that approximates the locations an optimal quadruped would traverse. The statistics do

```

```

668     not reflect the actual path taken by the quadruped policy, nor does it include the height
669     of gaps, pits, and other obstacles that the quadruped would not step on:
670     <INSERT TERRAIN STATISTICS HERE>
671
672     Please carefully analyze the statistics above and provide a new, improved generation function.
673     You should pay attention to which parts of the course the quadruped successfully learned
674     and which parts it struggled with. The goal of your course is to balance difficulty and
675     feasibility for the quadruped robot, allowing it to learn and perform better. Thus, if
676     the robot is getting stuck on a certain obstacle or goal, you should consider changing or
677     removing it. You should also follow the guidelines below:
678     - If the number of reached goals is over 80%, please create a harder course while ensuring
679     that it is feasible and safe for a real robot. You should consider adding more obstacles,
680     increasing the complexity of the course, and increasing the difficulty of existing
681     obstacles. For example, you can make climbing obstacles taller or jumping gaps wider.
682     - If the number of reached goals is below 20%, please create an easier course by decreasing
683     the difficulty of existing obstacles or simplifying the course layout. For example, you
684     can make climbing obstacles shorter or jumping gaps narrower. Please also double-check
685     that the course obstacles are fair and feasible for the quadruped.
686     - Otherwise, please create a variation of the current course with the same difficulty but
687     different obstacles. If the quadruped seems stuck on a certain obstacle, please change it
688     .
689
690     Again, please be creative when designing your course as we want to provide a diverse set of
691     training environments for the quadruped. Here is a list of the courses and skills that
692     the quadruped was already trained on. You may use them as inspiration, but please make
693     sure yours is different:
694     <INSERT TERRAIN DESCRIPTIONS HERE>
695
696     Please use the same template for the course generation function and provide a detailed
697     reasoning of the changes you made. The function signature should remain the same.

```

Listing 3: Evolution Prompt. Here, we insert policy training statistics including reward term values, episode length, number of goals reached, and number of edge violations (feet too close to edge) both before and after training. We also compute height field (terrain) statistics, including maximum value, maximum difference between consecutive indices, and standard deviation. Finally, we provide the LLM with docstrings of previous terrains used for training.

```

699     The following is an example of an initial terrain generation function.
700
701     '''
702     (SAME AS INITIAL EXAMPLE)
703     '''
704
705     And the following is an example of a new, improved terrain generation function that has more
706     complex obstacles. You can reference the example provided, but please make your terrain
707     different. This example serves to illustrate how terrains can be made more complex, but
708     you should tune the difficulty according to the previous instructions.
709
710     '''
711     import numpy as np
712
713     def set_terrain(length, width, field_resolution, difficulty):
714         """Multiple sideways-facing ramps traversing a pit for the robot to climb on and jump
715         across."""
716
717         def m_to_idx(m):
718             """Converts meters to quantized indices."""
719             return np.round(m / field_resolution).astype(np.int16) if not (isinstance(m, list) or
720             isinstance(m, tuple)) else [round(i / field_resolution) for i in m]
721
722         height_field = np.zeros((m_to_idx(length), m_to_idx(width)))
723         goals = np.zeros((8, 2))
724
725         # Set up platform and ramp dimensions
726         # We make the platform height near 0 at minimum difficulty so the quadruped can learn to
727         # climb up
728         platform_length = 1.0 - 0.3 * difficulty
729         platform_length = m_to_idx(platform_length)
730         platform_width = np.random.uniform(1.0, 1.1) # Decrease platform width
731         platform_width = m_to_idx(platform_width)
732         platform_height_min, platform_height_max = 0.0 + 0.2 * difficulty, 0.05 + 0.25 *
733         difficulty
734         ramp_height_min, ramp_height_max = 0.0 + 0.5 * difficulty, 0.05 + 0.55 * difficulty
735         gap_length = 0.1 + 0.5 * difficulty # Decrease gap length
736         gap_length = m_to_idx(gap_length)
737
738         mid_y = m_to_idx(width) // 2

```

```

740
741 def add_platform(start_x, end_x, mid_y):
742     half_width = platform_width // 2
743     x1, x2 = start_x, end_x
744     y1, y2 = mid_y - half_width, mid_y + half_width
745     platform_height = np.random.uniform(platform_height_min, platform_height_max)
746     height_field[x1:x2, y1:y2] = platform_height
747
748 def add_ramp(start_x, end_x, mid_y, direction):
749     half_width = platform_width // 2
750     x1, x2 = start_x, end_x
751     y1, y2 = mid_y - half_width, mid_y + half_width
752     ramp_height = np.random.uniform(ramp_height_min, ramp_height_max)
753     slant = np.linspace(0, ramp_height, num=y2-y1)[::direction]
754     slant = slant[None, :] # Add a dimension for broadcasting to x
755     height_field[x1:x2, y1:y2] = slant
756
757 dx_min, dx_max = -0.1, 0.1
758 dx_min, dx_max = m_to_idx(dx_min), m_to_idx(dx_max)
759 dy_min, dy_max = 0.0, 0.4 # Polarity of dy will alternate instead of being random
760 dy_min, dy_max = m_to_idx(dy_min), m_to_idx(dy_max)
761
762 # Set spawn area to flat ground
763 spawn_length = m_to_idx(2)
764 height_field[0:spawn_length, :] = 0
765 # Put first goal at spawn
766 goals[0] = [spawn_length - m_to_idx(0.5), mid_y]
767
768 # Set remaining area to be a pit
769 # We do this to force the robot to jump from platform to platform
770 # Otherwise, the robot can just jump down and climb back up
771 height_field[spawn_length:, :] = -1.0
772
773 # Add first platform
774 cur_x = spawn_length
775 dx = np.random.randint(dx_min, dx_max)
776 dy = np.random.randint(dy_min, dy_max)
777 add_platform(cur_x, cur_x + platform_length + dx, mid_y + dy)
778 goals[1] = [cur_x + (platform_length + dx) / 2, mid_y + dy]
779 cur_x += platform_length + dx + gap_length
780
781 for i in range(1, 6): # Set up 5 ramps
782     dx = np.random.randint(dx_min, dx_max)
783     dy = np.random.randint(dy_min, dy_max)
784     direction = (-1) ** i # Alternate left and right ramps
785     dy = dy * direction # Alternate positive and negative y offsets
786
787     add_ramp(cur_x, cur_x + platform_length + dx, mid_y + dy, direction)
788
789     # Put goal in the center of the ramp
790     goals[i+1] = [cur_x + (platform_length + dx) / 2, mid_y + dy]
791
792     # Add gap
793     cur_x += platform_length + dx + gap_length
794
795 # Add final goal behind the last ramp, fill in the remaining gap
796 goals[-1] = [cur_x + m_to_idx(0.5), mid_y]
797 height_field[cur_x:, :] = 0
798
799 return height_field, goals
800

```

Listing 4: Evolution Example. The initial example is exactly the initial example above, and we provide an example of evolving it; in our example, we replace some flat platforms with slanted ramps.

```

802
803 Here's an alternative terrain generation function that focuses on creating a series of ramps
804 for the quadruped robot to navigate through. This course will test the robot's ability to
805 handle inclined surfaces and maintain balance.
806
807 ```python
808 import numpy as np
809 import random
810
811 def set_terrain(length, width, field_resolution, difficulty):
812     """Ramps of varying angles for the robot to climb and descend."""
813
814     def m_to_idx(m):
815         """Converts meters to quantized indices."""

```

```

816         return np.round(m / field_resolution).astype(np.int16) if not (isinstance(m, list) or
817                               isinstance(m, tuple)) else [round(i / field_resolution) for i in m]
818
819     height_field = np.zeros((m_to_idx(length), m_to_idx(width)))
820     goals = np.zeros((8, 2))
821
822     # Initialize parameters
823     ramp_length = 1.0 + difficulty * 2.0 # Make ramps longer and steeper with higher
824     difficulty
825     ramp_length = m_to_idx(ramp_length)
826     ramp_height = 0.1 + difficulty * 0.4 # Ramp height varies with difficulty
827     ramp_height_change = ramp_height / ramp_length # Height change per quantized step
828     ramp_width = 1.0 # Set ramp width to 1 meter
829     ramp_width = m_to_idx(ramp_width)
830
831     mid_y = m_to_idx(width) // 2
832
833     def add_ramp(start_x, end_x, mid_y, upward=True):
834         """Add a ramp either upward or downward starting from start_x."""
835         for i in range(start_x, end_x):
836             height = (i - start_x) * ramp_height_change if upward else (end_x - 1 - i) *
837             ramp_height_change
838             height_field[i, mid_y - ramp_width // 2: mid_y + ramp_width // 2] = height
839             if not upward:
840                 height_field[i, mid_y - ramp_width // 2: mid_y + ramp_width // 2] -=
841                 ramp_height # Adjust for downward ramp
842
843     dx_min, dx_max = -0.1, 0.1
844     dx_min, dx_max = m_to_idx(dx_min), m_to_idx(dx_max)
845     dy_min, dy_max = -0.4, 0.4
846     dy_min, dy_max = m_to_idx(dy_min), m_to_idx(dy_max)
847
848     # Set flat ground at the start
849     spawn_length = m_to_idx(2)
850     height_field[0:spawn_length, :] = 0
851     goals[0] = [spawn_length - m_to_idx(0.5), mid_y] # First goal at spawn
852
853     # Initial height is 0
854     cur_x = spawn_length
855     cur_upward = True # Start with an upward ramp
856
857     for i in range(7): # Set up 7 ramps
858         dx = np.random.randint(dx_min, dx_max)
859         dy = np.random.randint(dy_min, dy_max)
860         end_x = cur_x + ramp_length + dx
861
862         add_ramp(cur_x, end_x, mid_y + dy, upward=cur_upward)
863
864         # Place goal in the center of the ramp
865         goals[i+1] = [(cur_x + end_x) / 2, mid_y + dy]
866
867         # Switch direction (up or down)
868         cur_upward = not cur_upward
869         cur_x = end_x
870
871     # Final goal at the end of the last ramp
872     goals[-1] = [cur_x + m_to_idx(0.5), mid_y]
873     height_field[cur_x:, :] = 0
874
875     return height_field, goals
876
877

```

Listing 5: Example LLM Output. This output generates multiple ramps in a configuration that's distinctly different from any previously-seen terrains.

```

878 Here's an alternative terrain generation function that focuses on creating a series of ramps
879 for the quadruped robot to navigate through. This course will test the robot's ability to
880 handle inclined surfaces and maintain balance.
881
882
883 ```python
884 import numpy as np
885 import random
886
887 def set_terrain(length, width, field_resolution, difficulty):
888     """Ramps of varying angles for the robot to climb and descend."""
889
890     def m_to_idx(m):
891         """Converts meters to quantized indices."""

```

```

892     return np.round(m / field_resolution).astype(np.int16) if not (isinstance(m, list) or
893         isinstance(m, tuple)) else [round(i / field_resolution) for i in m]
894
895     height_field = np.zeros((m_to_idx(length), m_to_idx(width)))
896     goals = np.zeros((8, 2))
897
898     # Initialize parameters
899     ramp_length = 1.0 + difficulty * 2.0 # Make ramps longer and steeper with higher
900     difficulty
901     ramp_length = m_to_idx(ramp_length)
902     ramp_height = 0.1 + difficulty * 0.4 # Ramp height varies with difficulty
903     ramp_height_change = ramp_height / ramp_length # Height change per quantized step
904     ramp_width = 1.0 # Set ramp width to 1 meter
905     ramp_width = m_to_idx(ramp_width)
906
907     mid_y = m_to_idx(width) // 2
908
909     def add_ramp(start_x, end_x, mid_y, upward=True):
910         """Add a ramp either upward or downward starting from start_x."""
911         for i in range(start_x, end_x):
912             height = (i - start_x) * ramp_height_change if upward else (end_x - 1 - i) *
913             ramp_height_change
914             height_field[i, mid_y - ramp_width // 2: mid_y + ramp_width // 2] = height
915             if not upward:
916                 height_field[i, mid_y - ramp_width // 2: mid_y + ramp_width // 2] -=
917                 ramp_height # Adjust for downward ramp
918
919     dx_min, dx_max = -0.1, 0.1
920     dx_min, dx_max = m_to_idx(dx_min), m_to_idx(dx_max)
921     dy_min, dy_max = -0.4, 0.4
922     dy_min, dy_max = m_to_idx(dy_min), m_to_idx(dy_max)
923
924     # Set flat ground at the start
925     spawn_length = m_to_idx(2)
926     height_field[0:spawn_length, :] = 0
927     goals[0] = [spawn_length - m_to_idx(0.5), mid_y] # First goal at spawn
928
929     # Initial height is 0
930     cur_x = spawn_length
931     cur_upward = True # Start with an upward ramp
932
933     for i in range(7): # Set up 7 ramps
934         dx = np.random.randint(dx_min, dx_max)
935         dy = np.random.randint(dy_min, dy_max)
936         end_x = cur_x + ramp_length + dx
937
938         add_ramp(cur_x, end_x, mid_y + dy, upward=cur_upward)
939
940         # Place goal in the center of the ramp
941         goals[i+1] = [(cur_x + end_x) / 2, mid_y + dy]
942
943         # Switch direction (up or down)
944         cur_upward = not cur_upward
945         cur_x = end_x
946
947     # Final goal at the end of the last ramp
948     goals[-1] = [cur_x + m_to_idx(0.5), mid_y]
949     height_field[cur_x:, :] = 0
950
951     return height_field, goals
952 '''

```

Listing 6: Example LLM Output. This output generates multiple ramps in a configuration that’s distinctly different from any previously-seen terrains.

954 A.3 Generated Terrain Examples

955 In Figure 6, we visualize some of the terrains generated by EurekaVerse. We see that the
956 complexity and difficulty generally increases each iteration, and the quadruped learns to pass through
957 progressively more difficult obstacle courses. We also see some cases of resampling for courses that
958 are difficult but learnable.

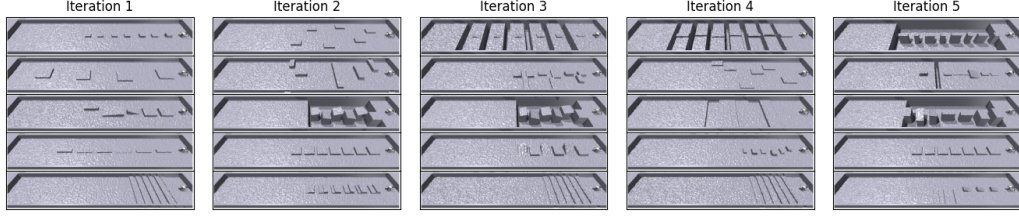


Figure 6: Visualizing a subset of our generated terrains, collected across 5 iterations within one run of EurekaVerse.

A.4 Environment Check and Automatic Fixing

For every LLM-generated terrain function, we first check that it is executable and feasible before using it for training. We check the former by running the training script and the latter with simple heuristics: whether the maximum height is below 3 meters whether the maximum height difference between goals is below 0.8 meters (double the Unitree Go1’s standing height).

To save token usage and query costs, we also automatically fix invalid terrains: moving out-of-bounds goals, setting the quadruped spawn area to flat ground, and expanding obstacles that are too skinny. Note that these fixes can be easily replaced with assertions and additional LLM queries.

A.5 Learning Progress Reweighting

We compute learning progress with the function G , which serves to prioritize progress in difficult tasks (low V) over easy tasks (high V). Following Kanitscheider et al. [58], we use the function $G(x, y) = f(x) - f(y)$ where

$$f(x) = \frac{(1 - x_\theta)x}{x + x_\theta(1 - 2x)} \quad (3)$$

with $x_\theta = 0.25$. As seen in Figure 7, taking $f(x) - f(y)$ leads to a larger learning progress estimate between 0.1 to 0.3 than 0.8 to 1.0.

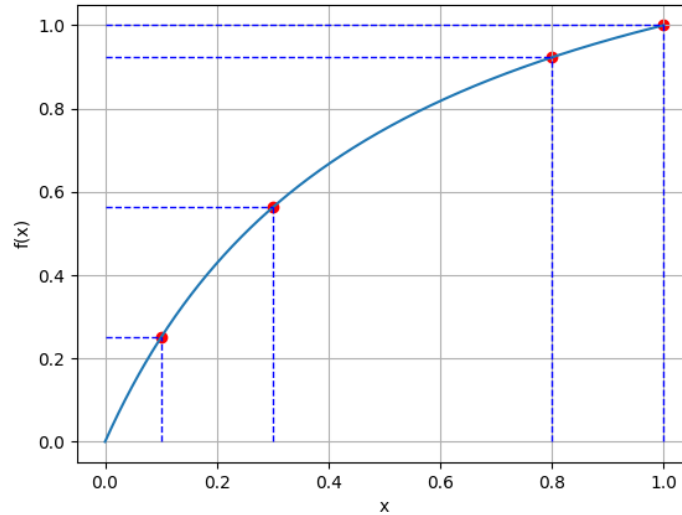


Figure 7: Our learning progress reweighting function G .

A.6 Simulation Setup

Environment Specification. We adopt the simulation framework from Cheng et al. [21], which constructs an obstacle course terrain by quantizing the ground plane into a 2D grid and specifying the height at each index in the grid. Along with the terrain heights, the course includes 8 goal positions indicating both progression and heading the policy should follow. Due to the difficult-but-progressive nature of parkour skills, the course includes an in-built inner curriculum where obstacles are scaled by a difficulty parameter between 0 and 1; during training, policies are initialized at lower difficulties and progress to higher difficulties if they pass the course. This specification is completely defined via a Python function, which takes the difficulty as input and outputs a 2D array for terrain heights and a list of 8 (x, y) coordinates for the goal positions. In *Eurekaverse*, the LLM is instructed to output precisely this function, which defines terrains θ_j modulated by the difficulty argument.

Policy Training. We train the policy on multiple terrains in parallel; specifically, one round of training consists of 10 terrain classes, each with 10 difficulties. Following Cheng et al. [21], we first train a teacher policy using privileged scandot sensing. After fully training a privileged teacher, we then distill a student policy that takes in depth frames from a front-facing camera. Apart from the scandot or depth, the policy also receives proprioception and heading and speed commands. We train the policy with PPO [60] with heading and velocity tracking rewards, regularizations like action rate and torque, and penalties for foot placements near terrain edges. We use this same reward across all methods and ablations.

Sim-To-Real. To facilitate sim-to-real transfer, as in Cheng et al. [21], we use regularized online adaptation (ROA) [67], which trains an adaptation module that estimates environment properties from observation history. We also domain randomize over physical properties like friction, mass, and motor strength. During distillation, we introduce an action delay and depth sensing delay of 1 simulation step (0.02s), and we update the depth frame only once every 5 steps (10Hz). Finally, we introduce noise in the depth input, adding Gaussian noise to its true value and randomly setting some pixels to 0.

Simulation Benchmark. In Figure 8, we render each of the 20 obstacles in our parkour benchmark. Each of these obstacles is instantiated with 10 difficulties during evaluation, and these renders capture the obstacles at medium difficulty.

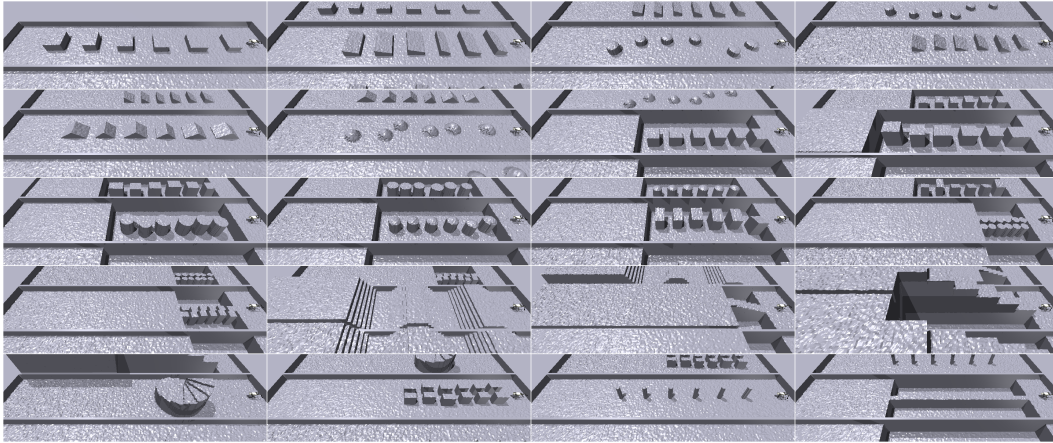


Figure 8: Visualizing our simulated parkour benchmark, roughly organized by skill: climbing boxes, walking on slopes, jumping, traversing stepping stones, climbing stairs, navigating narrow hallways, moving zig-zag through agility poles, and balancing.

A.7 Deployment Details

We deploy on the Unitree Go1, a quadrupedal robot with 12 degrees of freedom. When standing, the robot is 64.5 cm long, 28 cm wide, and 40 cm tall. We use the 3D camera mount introduced

by Zhuang et al. [20] to attach an Intel RealSense D435 camera onto the Go1’s head. Our vision policy runs onboard the Nvidia Jetson Xavier NX, with the depth encoder running asynchronously at 10 Hz and the policy at 50 Hz. Before depth inputs are passed through the depth encoder, we crop the left and right edges to remove dead pixels; we then apply hole-filling and temporal filters and down-sample the resolution from 270x480 to 60x90.

A.8 Ablations

In Figure 9, we plot the performance of ablations from Figure 4 (right) across training steps. All ablations plateau except for EurekaVerse and **Random Resampling**, and we see that **Random Resampling** exhibits much higher variance than Ours.

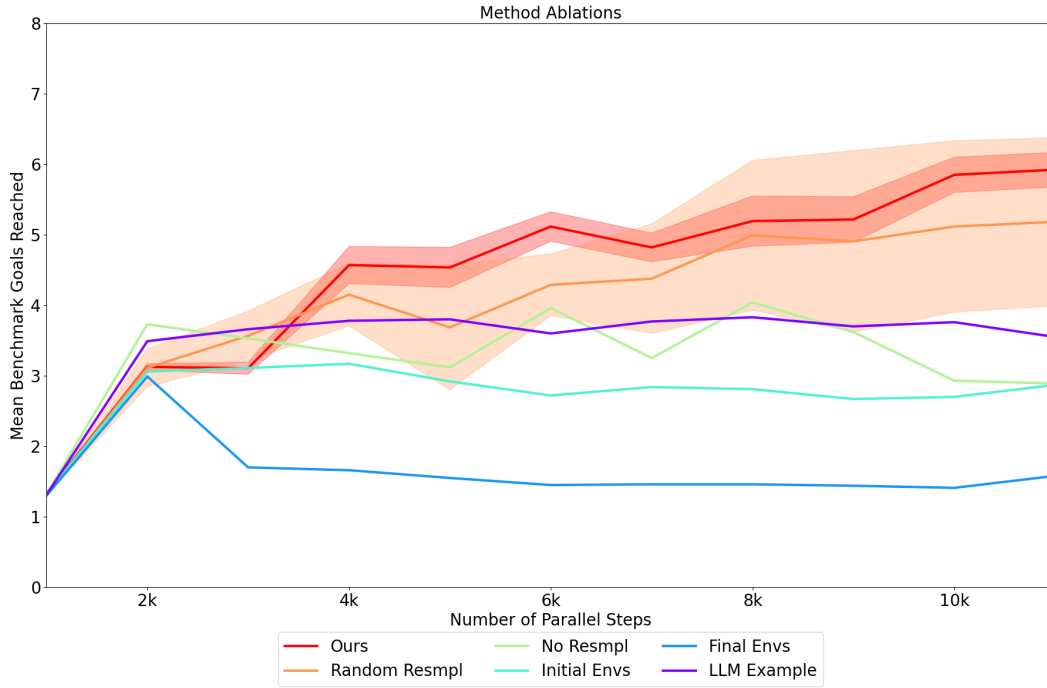


Figure 9: Comparing sim benchmark performance across training steps for EurekaVerse and ablations.