Module Interface Specification for TPG

Team 3, Tangle
Calvyn Siong
Cyruss Allen Amante
Edward Gao
Richard Li
Mark Angelo Cruz

April 2, 2025

1 Revision History

Date	Version	Notes
01/16/2025	0.0	Revision 0
03/13/2025	0.1	Modified uses to be consistent among modules based on peer feedback
03/29/2025	0.2	Added more details about .rslt files based on peer feedback
03/30/2025	0.3	Revised variable definition for TPG Experiment Module
04/02/2025	1	Revision 1

2 Symbols, Abbreviations and Acronyms

See SRS Documentation at this link. This section records information for easy reference.

symbol	description
AC	Anticipated Change
DAG	Directed Acyclic Graph
DNN	Deep Neural Network
FR	Functional Requirement
M	Module
MG	Module Guide
MIS	Module Interface Specification
Multi-Task RL	Multi-Task Reinforcement Learning
NFR	Non-Functional Requirement
OS	Operating System
R	Requirement
RL	Reinforcment Learning
SC	Scientific Computing
SRS	Software Requirements Specification
TPG	Tangled Program Graphs
UC	Unlikely Change

Contents

3 Introduction

The following document details the Module Interface Specifications for TPG. The Tangled Program Graphs (TPG) framework is developing an interface to test the evolutionary machine learning framework Tangle Programming Graphs (TPG) in a robotic simulation engine called MuJoCo created by Google Deepmind. The TPG project is written in C++, and C++ notation will be utilized within this document as such.

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at https://github.com/TPGEngine/tpg

4 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995), with the addition that template modules have been adapted from Ghezzi et al. (2003). The mathematical notation comes from Chapter 3 of Hoffman and Strooper (1995). For instance, the symbol := is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1|c_2 \Rightarrow r_2|...|c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by TPG.

Data Type	Notation	Description
character	char	a single symbol or digit
string	*string*	a sequence of characters
integer	*int*, $\mathbb{Z}$	a number without a fractional component in (-∞, ∞)
double	*double*	used to store high-precision floating-point numbers [1.7E-308, 1.7E+308]
natural number	$\mathbb{N}$	a number without a fractional component in [1, ∞)
real	$\mathbb{R}$	any number in (-∞, ∞)
pi	π	a mathematical constant, approximately equal to 3.14159
file	*file*	a collection of data that is stored on a computer

The specification of TPG uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, TPG uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.

5 Module Decomposition

The following table is taken directly from the Module Guide document for this project.

Level 1	**Level 2**
Hardware-Hiding Module	N/A
Behaviour-Hiding Module	Classic Control Module MuJoCo Module Visualization Module Logging Module
Software Decision Module	TPG Experiment Module

Table 1: Module Hierarchy

6 MIS of Classic Control Module

6.1 Module

ClassicControlEnv

6.2 Uses

The Classic Control Module is responsible for simulating and visualizing classic control tasks, such as cart-pole balancing or pendulum. These tasks provide a standardized environment for evaluating and training the TPG framework, allowing for a baseline assessment of the TPG algorithm. It relies on OpenGL and GLUT for rendering and visualizing the control environments. The module also uses the Task Environment (TaskEnv.h) to define the structure and behavior of the control tasks

6.3 Syntax

6.3.1 Exported Constants

None

6.3.2 Exported Access Programs

Name	Input	Output	Exceptions
ClassicControlEnv	-	-	-
~ClassicControlEnv	-	-	-
Bound	{x: double, m: double, M: double}	double	-
DisplayFunction	{window_width: int, window_height: int, zoom: double}	void	-
SaveScreenshotToFile	{filename: string, window_width: int, window_height: int}	void	File write error
DrawBitmapText	{string: char*, x: float, y: float, z: float}	void	-
DrawStrokeText	{string: char*, x: float, y: float, z: float}	void	-
DrawTrace	{idx: int, label: string, action_processed: double, y_action_trace: double}	void	-
DrawEpisodeStepCounter	{episode: int, step: int, x: float, y: float}	void	-
Linspace	{a: double, b: double, N: size_t}	std::vector<double>	

6.3.3 State Variables

Name	Type	Description
dis_reset	std::uniform_real_distribution<>	Distribution for reset values
dis_noise	std::uniform_real_distribution<>	Distribution for noise values
action_trace	std::vector<std::deque<double>>	History of actions taken

6.3.4 Environment Variables

OpenGL frame buffer

6.3.5 Assumptions

- OpenGL and GLUT are properly initialized before any drawing functions are called
- The action_trace vector is initialized with 3 deques of size 200
- The noise distribution is initialized with range

$$-\pi, \pi$$

6.3.6 Access Routine Semantics

ClassicControlEnv():

- transition: initializes action_trace with 3 deques of 200 zeros, sets dis_noise range to

$$-\pi, \pi$$

- exception: none

~ClassicControlEnv():

- transition: self $\rightarrow$ destroyed
- exception: none

Bound(x, m, M):

- output: $out := min(max(x, m), M)$
- exception: none

DisplayFunction(*window_width*, *window_height*, *zoom*):

- transition: virtual function to be implemented by derived classes
- exception: none

SaveScreenshotToFile(*filename*, *window_width*, *window_height*):

- transition: saves current OpenGL frame buffer to file
- exception: file write error

DrawBitmapText(*string*, x, y, z):

- transition: renders bitmap text at specified 3D coordinates
- exception: none

DrawStrokeText(*string*, x, y, z):

- transition: renders stroke text at specified 3D coordinates with scaling
- exception: none

DrawTrace(*idx*, *label*, *action_processed*, *y_action_trace*):

- transition: updates and renders action trace history
- exception: none

DrawEpisodeStepCounter(*episode*, *step*, *x*, *y*):

- transition: renders episode and step counter at specified coordinates
- exception: none

Linspace(*a*, *b*, *N*):

- output: *out* := vector of N evenly spaced values between a and b
- exception: none

6.3.7 Local Functions

None

7 MIS of MuJoCo Module

7.1 Module

MujocoEnv

7.2 Uses

The MuJoCo Module is developed to simulate complex robotic tasks and physical environments using the MuJoCo physics engine. This module enables the TPG framework to train and evaluate policies in more dynamic environments, such as robotic arm manipulation or humanoid movements. The MuJoCo Module is essential for testing TPG in scenarios that require precise physics simulation and interactions between agents and their environment. The Module uses the MuJoCo Framework (mujoco/mujoco.h) to simulate physical environments and model robotic systems. It also uses the Task Environment (TaskEnv.h) like ClassicControl to define the tasks and objectives within the MuJoCo environment.

7.3 Syntax

7.3.1 Exported Constants

None

7.3.2 Exported Access Programs

Name	Input	Output	Exceptions
mujocoEnv	-	-	-
~mujocoEnv	-	-	-
reset	{rng : mt19937&}	-	-
terminal	-	bool	-
sim_step	{action : std::vector<double>}	Results	-
initialize_simulation	-	-	Unable to load binary model
set_state	{qpos : std::vector<double>, qvel : std::vector<double>}	-	-
do_simulation	{ctrl : std::vector<double>, n_frames: int}	-	-
GetObsSize	-	int	-

7.3.3 State Variables

Name	Type	Description
m_	mjModel*	Pointer to the MuJoCo model
d_	mjData*	Pointer to the MuJoCo data
cam_	mjvCamera	Camera used for visualization
opt_	mjvOption	Visualization options
scn_	mjvScene	Visualization scene
con_	mjrContext	Custom GPU rendering context
init_qpos_	std::vector<double>	Initial positions
init_qvel_	std::vector<double>	Initial velocities
model_path_	string	Absolute path to model xml file
frame_skip_	int	Number of frames per simulation step
obs_size_	int	Size of the observation space

7.3.4 Environment Variables

None

7.3.5 Assumptions

- The MuJoCo environment is properly installed on the user's device.

7.3.6 Access Routine Semantics

mujocoEnv():

- output: *out* := self
- exception: none

~mujocoEnv():

- transition: self $\rightarrow$ destroyed
- exception: none

reset(*rng*):

- transition: resets the environment to initial state using a random number generator, *rng*.
- exception: none

terminal():

- transition: environment in terminal state ? terminal := true : terminal := false
- output: *out* := terminal
- exception: none

sim_step(*action*):

- transition: advances the simulation by one step using the given *action*.
- output: *out* := Results object containing simulation outcomes.
- exception: none

initialize_simulation():

- transition: initializes the simulation by loading the MuJoCo model and creating its data structure.
- exception: unable to load binary model error.

set_state(*qpos*, *qvel*):

- transition: sets the positions and velocities, *qpos* and *qvel*, in the simulation state.
- exception: none

do_simulation(*ctrl*, *n_frames*):

- transition: executes *n_frames* of simulation steps with the given control inputs, *ctrl*.
- exception: none

7.3.7 Local Functions

None

8 MIS of Visualization Module

8.1 Module

The visualization module implements visualization functionality of a TPG experiment with graphs, charts, video playback, and a realtime simulation.

8.2 Uses

The visualization module is used to visualize the results of a TPG experiment while it is running or after the experiment has completed. This information contains data pertaining to what is going on in an experiment, and performance metrics.

8.3 Syntax

8.3.1 Exported Constants

None

8.3.2 Exported Access Programs

Name	Input	Output	Exceptions
plot-stats	{fitnessValues: int, generationTime: int, programCount: int, teamCount: int}	{PDF file}	Returns a PDF file of plot statistics for an experiment
run-mpi	{fitnessValues: int, generationTime: int, programCount: int, teamCount: int}	{OpenGL animation}	An OpenGL animation starts to run

8.4 Semantics

8.4.1 State Variables

None

8.4.2 Environment Variables

Name	Type	Description
tpg.[experimentId].std	File(s)	output log files generated for each experiment

8.4.3 Assumptions

- Format of the log files is known and will not change.

8.4.4 Access Routine Semantics

None

8.4.5 Local Functions

Name	Input	Output	Description
extractMetrics	{logFile: file}	{dataFile: file}	Saves extracted data into .rslt files
transformData	{logFile: file}	{dataFile: file}	Reverse processes the log files, pattern matches, reformatting, and aggregating the data and saving to a .rslt file
plotData	{logFile: file}	{plotFile: file}	Plots the data from the .rslt file and saves the plot to a .pdf file

Note: A ".rslt" file is a type of results file that contains information about the execution of a test case, script, or test plan.

9 MIS of Logging Module

9.1 Overview

The Logging module implements logs key information and debugging related information

9.2 Uses

The log module is used to view information regarding the commands transmitted to a satellite. This information contains data pertaining to the commands sent during a particular schedule, their run time and the response received.

9.3 Syntax

9.3.1 Exported Constants

None

9.3.2 Exported Access Programs

Name	Input	Output	Exceptions
LogMetrics	{experimentId: string, metrics: map⟨string, double⟩}	void	File write error
LogGeneration	{experimentId: string, generation: int, stats: map⟨string, double⟩}	void	File write error
LogError	{experimentId: string, error: string}	void	File write error

9.4 Semantics

9.4.1 State Variables

Name	Type	Description
logFiles	map⟨string, File*⟩	Map of experiment IDs to their log file handles

9.4.2 Environment Variables

Name	Type	Description
tpg.[experimentId].std	File	Output log files for each experiment

9.4.3 Assumptions

- The logging module must be initialized before any logging operations
- The file system has write permissions for creating log files
- Each experiment has a unique experimentId
- Log file format follows the pattern "tpg.[experimentId].std"

9.4.4 Access Routine Semantics

LogMetrics(*experimentId*, *metrics*):

- transition: writes metrics data to the corresponding experiment log file
- exception: FILE_WRITE_ERROR if file cannot be written

LogGeneration(*experimentId*, *generation*, *stats*):

- transition: writes generation statistics to the corresponding experiment log file

- exception: FILE_WRITE_ERROR if file cannot be written

LogError(*experimentId*, *error*):

- transition: writes error message to the corresponding experiment log file
- exception: FILE_WRITE_ERROR if file cannot be written

9.4.5 Local Functions

Name	Input	Output	Description
getLogFile	{experimentId: string}	File*	Gets or creates log file handle for given experiment

10 MIS of TPG Experiment Module

10.1 Module

TPGExperimentMPI

10.2 Uses

The TPG Experiment module is responsible for managing and evolving different policies, evaluating different tasks, and tracking experiments. It is used during the execution of experiments through the commands: **tpg-run-slurm** and **tpg-run-mpi**, which are used for executing tasks in virtual and local machines respectively.

10.3 Syntax

10.3.1 Exported Constants

None

10.3.2 Exported Access Programs

None

10.4 Semantics

10.4.1 State Variables

Name	Type	Description
world_rank	int	The rank of the MPI process within the environment. It is used to distinguish between the master and evaluator processes.
n_task	int	The current number of tasks (environments) running.
active_task	int	The index of the current active task.
phase	int	The numerical representation of the current phase of the program.
t_current	int	The current generation or iteration of the task.
t_start	int	The starting generation or iteration of the task.
task_to_replay	int	The index of the task to replay.
mj_model_path	str	The path to the MuJoCo environment object.

10.4.2 Environment Variables

Name	Type	Description
COMET_API_KEY	string	The API key used for Comet experiment tracking service

10.4.3 Assumptions

- Experimental parameters such as **mj_model_path** and **active_tasks** are well-defined. The definition for these terms are defined in the state variables.

10.4.4 Access Routine Semantics

This module is public and compiled into an executable, which is then run as a script.

10.4.5 Local Functions

None

References

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL http://citeseer.ist.psu.edu/428727.html.

11 Appendix

Appendix — Reflection

1. What went well while writing this deliverable?

   Mark - Everyone on the team was able to accomplish their part of the deliverable without any critical blockers. The team was able to work cohesively, leveraging their diverse knowledge on certain topics which reflects the outcome of the deliverable. Constant communication and updates on every member's progress allowed the team to keep track and stay informed with convenience and ease.

   Cyruss - An aspect that went well while writing this deliverable was the easy division of parts, specifically deciding who would be doing which sections of the deliverable. Each member leaned towards specific parts, which minimized any conflict between members. The discussion amongst the team during the winter break allowed each team member to focus on specific areas throughout the few couple of weeks, making the process more efficient. Another aspect that went well was our ability to set deadlines for ourselves. Even though we faced some minor delays, the deliverable was able to be completed.

   Richard - When writing this deliverable, the team was very transparent, cohesive, and organized. We were all able to divide our tasks in a quick manner, thus allowing each group member to work asynchronously. When there were errors regarding merge conflicts, all the members involved answered the group chat quickly, and the merge conflicts were able to get resolved.

   Calvyn - I believe that the team was able to coordinate effectively, ensuring each member understood which modules we would be including and why we needed these modules. The team maintained a consistent line of communication, which allowed us to address any issues that came up.

2. What pain points did you experience during this deliverable, and how did you resolve them?

   Mark - A few pain points during this deliverable include deciding what and what not to cover within the design document and the process of keeping everything consistent to what is declared as notations. Since TPG was already implemented prior to Capstone, it has a complex codebase with a great amount of components. With this, the team dissected and looked through every module that we can cover that are relevant to our Capstone project and must be included in the document. For consistency, the codebase mainly uses abstract classes, which may cause confusion for readers who are not familiar with the syntax. To reduce confusion, the team had to expand the notation section with relevant symbols and mathematical terms that were used for our MIS.

Cyruss - A major pain point experienced during this deliverable was the difficulties in determining how to structure the design of the capstone project. It was difficult to decide due to working on an existing codebase, and figuring out how to abstract all this data from a higher level than from a C++ point of view. However, the team worked together to resolve this issue by having a detailed meeting, focusing on understanding our goals for the project and ensuring that the design of the existing codebase is minimally obstructed. With detailed conversations with everyone on the team, a consensus was determined on how to format the contents of the project's code within the deliverable with a high degree of satisfaction from each of the members. There was somewhat of another pain point when it came to the communication between some members, with us not being on track with everything; however, the team resolved this through consistent follow-ups and discussion amongst everyone.

Richard - A pain point our team dealt with was consistency. Since we all have different styles towards notation, standardizing this process took some communication and adjustment. For instance, a small example was a debate using "None" vs "N/A". When doing a code review, we found that there was an inconsistency in this notation. This sparked a conversation between some of us that we needed to standardize this notation because in previous deliverables we got warning about these inconsistencies.

Calvyn - A pain point during this deliverable was dealing with modules that have not yet been created/fully fleshed out. This brought a level of uncertainty in planning out functions and their interactions, as they might be subject to change as the project evolves. To address this, the team had to make informed assumptions based on our understanding of the project's goals and the existing codebase. In addition, our team held discussions to ensure our MIS would align as best as it could with our project's purpose.

3. Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. your peers, stakeholders, potential users)? For those that were not, why, and where did they come from?

   Pretty much all of the design decisions for the deliverable and the project stemmed from the team supervisor, Dr. Stephen Kelly (and indirectly his graduate students). This was due to the fact that TPG already had an existing codebase and is concurrently being worked on by graduate students of Dr. Kelly throughout the capstone project. This required the team to have consistent (weekly) meetings with the supervisor, ensuring that no delays or blockers would arise due to the number of people being worked on. For all documentation and code changes, the team would consistently check in with Dr. Kelly to ensure that it aligned with his team's goals in addition to the capstone project goals.

4. While creating the design doc, what parts of your other documents (e.g. requirements, hazard analysis, etc), it any, needed to be changed, and why?

For this deliverable, there were no parts of other documents that needed to be changed.

5. What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)

   Some of the limitations of our project come from the constraints of using free and open-source resources. They are easy to use and setup for development, it might limit the robustness and scalability of our CI/CD pipeline. Furthermore, since the the CI/CD integrations we are building are from scratch with limited experience, it might lack the rigor and standards seen in professional solutions. With unlimited resources, we could leverage enterprise-grade tools and services to build a more robust and scalable pipeline, reducing manual intervention and improving reliability.

   Another limitation is our computational capacity. If we have access to higher end performance servers. With extensive computations for the TPG algorithm being ran across various environments, we would be able to test and iterate more frequently if we wanted to. With more powerful servers, we could run larger-scale experiments, explore more diverse scenarios, and optimize the algorithm to a greater extent.

6. Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)

   With our project, a lot of the design choices were influenced by our supervisor Dr. Kelly. This is mainly due to the fact that he is a domain expert and he is also constantly working on the same codebase as us (he is also pushing changes and building alongside us). Our team has weekly meetings with him as we discuss future plans and give him updates on what we work on. A design that we had been contemplating for a while is related to the logging mechanism, and we decided to include this as a module because we all noticed while running experiments that the scripts Dr. Kelly used to generate logs were inefficient. We selected this option of refactoring the logging system because it is a crucial part of the workflow. Automating and making this easier by converting the scripts from bash to Python would make development and analysis easier. Now, on our roadmap we will be focusing on the logging module and integrating/refactoring this system to fit within the broader ecosystem of Dr. Kelly's vision for his TPG project.