ChatGPT

Building a Genesis-Powered Robot Simulator (Simforge)

Overview:
We are designing **Simforge**, a Python tool to simulate and control multiple robot arms (Meca500, Staubli TX2 series, UR e-series, etc.) using the **Genesis** physics engine. The goal is to achieve **real-world-like robot motion** with robust collision checking (self-collision and environment collision), similar to ROS2 MoveIt2's safety. The user will interact through a GUI (cross-platform on Ubuntu 24.04 and macOS) that allows switching between **joint-space control** and **Cartesian (TCP) control**, much like a real teach pendant. This report covers setting up Genesis on Ubuntu (and macOS), structuring the Simforge project, and implementing key features (environment configs, simulation control, GUI, logging).

Installation and Setup of Genesis Simulation

To begin, set up an isolated Python environment (Python 3.10 or 3.11 is recommended per Genesis requirements [1]). Then follow these steps on each platform:

1. **Create a Virtual Environment:** Use `python3 -m venv venv` (or Conda if preferred) and activate it. This keeps dependencies isolated.
2. **Install PyTorch:** Genesis uses PyTorch for GPU acceleration. Install PyTorch *with the appropriate backend* for your system:
3. On **Ubuntu (NVIDIA GPU)**: Ensure the NVIDIA driver is installed, then install PyTorch with CUDA support (e.g., via the PyTorch website's pip command for CUDA 11/12). For example, you might run `pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118` (if CUDA 11.8 is appropriate).
4. On **Ubuntu (CPU only)**: You can simply `pip install torch` to get the CPU build of PyTorch.
5. On **macOS (Apple Silicon M1/M2)**: PyTorch's pip package includes Apple Metal (MPS) support. Just run `pip install torch torchvision torchaudio` (ensure PyTorch $\geq$1.12 for MPS).
6. *Verify:* After installation, in Python run `import torch; print(torch.cuda.is_available(), torch.backends.mps.is_available())` to ensure CUDA or MPS is detected.
7. **Install Genesis:** Once PyTorch is ready, install Genesis via pip: `pip install genesis-world`. This pulls the Genesis simulation library (requires Python $\geq$3.10) [2] . Genesis is cross-platform – it supports CPU, NVIDIA CUDA, and non-CUDA GPUs (AMD, Apple Metal) [3] . On Linux with NVIDIA GPU, make sure the CUDA drivers are installed for best performance [4] .
8. **(Optional) Motion Planning:** For advanced Cartesian control with path planning, install the OMPL library. Genesis integrates OMPL (Open Motion Planning Library) for sampling-based planning [5] . You can install OMPL's Python bindings via pip: `pip install ompl` [6] . This will allow the tool to compute collision-free paths for the robot when moving in Cartesian mode.
9. **(Optional) Other Features:** If needed, install extras like `splashsurf` or Genesis's ray-tracing renderer as per Genesis docs [7] [8] (not critical for basic operation). Also install a GUI library if not using Tkinter (for example, `pip install PyQt5` or `pip install PySide6` for a Qt GUI).

Backend Selection: In code, Genesis allows choosing the compute backend at init. We will use `gs.init(backend=gs.gpu)` which auto-selects the best GPU backend or falls back to CPU [9]. This means on Linux it will prefer CUDA if available, and on Apple Silicon it will use Metal (MPS) [10]. We can also explicitly choose `gs.cuda` or `gs.metal` if needed, but `gs.gpu` covers both. We'll default to GPU for performance, but ensure the code can run with `gs.cpu` if no GPU is present.

Project Structure and Workspace Initialization

Organize the Simforge project as a **professional Python package**. A suggested structure is:

```
simforge/               # Project root
├── README.md           # Documentation and usage examples
├── setup.py or pyproject.toml  # Package configuration (optional if just a
script)
├── requirements.txt    # Dependencies (genesis-world, PyTorch, GUI libs, etc.)
├── assets/             # Robot assets (URDFs and meshes as given)
│   ├── meca500/...
│   ├── tx2_60/...
│   └── ...             # (All the provided robot subfolders and files)
├── env_configs/        # Template environment configuration files
│   ├── meca500_env.yaml
│   ├── tx2_60_env.yaml
│   ├── ur5e_env.yaml
│   └── ...             # etc., one per robot (and possibly combined scenarios)
└── simforge/           # Python package directory
    ├── __init__.py
    ├── main.py         # CLI entry-point (e.g., `simforge init` or run
simulation)
    ├── simulator.py    # Genesis simulation setup and control logic
    ├── controller.py   # High-level robot control logic (IK, motion planning)
    ├── gui.py          # GUI application (Tkinter or PyQt implementation)
    ├── config.py       # Config file parsing and data structures
    └── logging_utils.py# Logging configuration (for debug mode)
```

Key points about the structure:

- **Assets Directory:** We keep the provided URDFs and mesh files under `assets/`. The tool will load these at runtime. The URDFs contain joint limits, link frames, and collision geometry for each robot. Genesis will parse these files to construct the robot models [11]. (We will ensure the file paths are correctly referenced, e.g. by relative path or an environment variable pointing to the assets folder.)
- **Environment Configs:** The `env_configs/` folder contains YAML files that define simulation scenarios. When the user initializes a workspace (e.g., running `simforge init`), the tool will generate these template configs. Each robot platform has a base config file describing that robot in a default scene. Users can combine or modify them to simulate multiple robots or custom environments. More details on config content are below.

- **Python Package Code:** The `simforge` package contains the source code. This includes modules for simulation handling (`simulator.py`), robot control logic (`controller.py` for IK, path planning, etc.), GUI elements (`gui.py`), and configuration management (`config.py`). Splitting these concerns makes the architecture clean:
- `simulator.py`: Wraps Genesis API calls (initialization, scene creation, stepping the simulation). It might define a `Simulator` class with methods to load robots and objects based on a config.
- `controller.py`: Implements higher-level operations like switching control modes, computing IK solutions via Genesis, calling motion planner, and applying joint controls each step.
- `gui.py`: Builds the user interface (windows, buttons, sliders) and handles user inputs. It interacts with the simulator/controller (possibly via callbacks or a shared state).
- `config.py`: Handles reading the YAML config files into Python objects (using PyYAML). It can also provide functions to initialize a new config with default values.
- `logging_utils.py`: Sets up Python's `logging` module formatting and levels. We'll use this to provide debug output when the user runs the tool in a verbose/debug mode.

Initializing a Workspace: When the user first runs `simforge init`, the tool should create the necessary directories (if not present) and populate default config files. For example, `meca500_env.yaml` might be created with content for a single Meca500 robot simulation. Similar files for each UR and TX2 model will be generated. This gives users a starting point for each robot. The initialization can also copy the `assets/` folder if needed or verify it exists (in our case, assets are already present). The CLI entry (`main.py`) would implement commands like `init` and `start` (to launch the simulation with a chosen config). Using a CLI library (like Python's built-in argparse or Click) can help parse such commands.

Environment Configuration Files (YAML Templates)

Each environment config YAML describes the **simulation setup** for one scenario. This allows users to tweak parameters without modifying code. A typical `*_env.yaml` might look like:

```yaml
scene:
  dt: 0.01              # simulation time step (s)
  gravity: [0, 0, -9.81]  # gravity vector
  backend: "gpu"         # or "cpu" to override auto GPU selection if needed
  show_viewer: true      # whether to open the 3D viewer
robots:
  - name: "UR5e_1"
    urdf: "assets/ur5e/ur5e.urdf"
    base_position: [0.0, 0.0, 0.0]   # (x, y, z) in world coordinates
    base_orientation: [0, 0, 0]     # (roll, pitch, yaw) in degrees
    fixed_base: true                # fix the base link to world (no free-fall)
    initial_joint_positions: [0, -90, 0, -90, 0, 0]  # in degrees (or radians)
  # Additional robots can be listed in the array for multi-robot scenarios
objects:
  - type: "plane"
    position: [0, 0, 0]
    size: [5, 5]      # e.g., plane size or other params
  # Additional static objects or obstacles can be added here
control:
  joint_speed_limit: 0.5     # (rad/s) max joint speed for manual control
  cartesian_speed_limit: 0.1 # (m/s) max TCP linear speed
  default_tcp: [0, 0, 0, 1, 0, 0, 0]  # default Tool Center Point offset (x,y,z
and quaternion)
```

Explanation: The config separates concerns: - **Scene settings:** physics timestep (`dt`), gravity, and viewer options. The user can disable the GUI viewer for headless runs by setting `show_viewer: false` (e.g., in automated testing). The backend is usually auto-chosen by code, but can be forced via config if needed (e.g., `"cpu"` for debugging without GPU). - **Robots:** A list of robot instances. In a simple scenario there's one robot, but this structure lets us define multiple robots (with different or same URDF). Each robot entry has: - `name`: an identifier (used for logging or selecting in the UI if multiple robots are present). - `urdf`: path to the URDF file for that robot model. - `base_position` and `base_orientation`: the pose of the robot's base in the world. This allows placing robots at different locations or mounting angles. - `fixed_base`: boolean. **Important:** For arms mounted on the ground or a table, we set `fixed_base: true` so the base link is anchored to world [12]. If false, Genesis would treat the robot base with a free 6-DOF joint and gravity would cause it to fall. Most stationary robots should be fixed. - `initial_joint_positions`: an array of joint angles for the starting pose. URDF does not natively specify initial joint states, so we define them here (often a "home" position). These values will be applied after loading the robot. - **Objects:** This can include a ground plane or any environment obstacles. In the example, we add a large plane at z=0.0 as the floor. We can extend this to include boxes, walls, etc., by specifying object type, size, pose. - **Control settings:** Parameters related to robot motion control. For instance, speed limits can cap how fast the user's commands move the robot (preventing unrealistic jumps). We also define a default TCP (tool center point) offset – for example, if a gripper or tool is attached, the TCP might be a few centimeters from the robot's flange. This TCP is used in Cartesian mode calculations.

The Simforge tool will **parse this YAML** (using `yaml.safe_load` in `config.py`) into a Python structure (e.g., dictionaries or dataclasses). During simulation startup, we use these settings to configure Genesis.

Genesis Simulation Setup in Code

With the config loaded, the `Simulator` class (in `simulator.py`) will initialize Genesis and build the scene:

- **Initialize Genesis:** We call `import genesis as gs` and then `gs.init()` at the program start. The backend is chosen based on config or defaults. For example:

```
if config.scene.get("backend", "gpu") == "gpu":
    gs.init(backend=gs.gpu)   # auto GPU or CPU selection [10]
else:
    # allow explicit override
    backend = gs.cuda if config.scene["backend"] == "cuda" else gs.cpu
    gs.init(backend=backend)
```

We might also set `precision='32'` (default) and a logging level. In debug mode, we let Genesis log verbosely; otherwise we may set `logging_level='warning'` to reduce console noise [13]. Genesis will print some info about device and version on init.

- **Create Scene:** Next, we create a `gs.Scene`. We pass in simulation options like time step and gravity from the config:

```
scene = gs.Scene(
    sim_options = gs.options.SimOptions(dt=config.scene.get("dt", 0.01),

gravity=tuple(config.scene.get("gravity", (0,0,-9.81)))),
    viewer_options = gs.options.ViewerOptions(
        camera_pos=(3.0, 0.0, 2.0), camera_lookat=(0,0,0.5),  # default
camera
        max_FPS=config.scene.get("max_fps", 60)
    ),
    show_viewer = config.scene.get("show_viewer", True)
)
```

This sets up the physics simulator and optionally an interactive viewer window [14]. The viewer will allow us to see the robot and environment in 3D. (The camera settings can be adjusted; in a later iteration we might expose them in config as well.)

- **Add Environment Objects:** We then add static objects like the ground plane. For example:

```
for obj in config.objects:
    if obj["type"] == "plane":
        plane_entity = scene.add_entity(gs.morphs.Plane(
                           pos=tuple(obj.get("position", (0,0,0))),
                           size=obj.get("size", (5.0, 5.0))
                       ))
    # ... handle other object types like Box, Sphere if needed
```

Genesis provides primitive shapes (Plane, Box, Cylinder, etc.) as morphs [15]. Here we add a ground plane at z=0.0. By default, the plane extends infinitely, but we could limit its visual size via the `size` parameter.

- **Load Robot from URDF:** For each robot in the config, we use `gs.morphs.URDF` to load the model:

```
for robot_cfg in config.robots:
    robot_entity = scene.add_entity(
        gs.morphs.URDF(file=robot_cfg["urdf"],
                       pos=tuple(robot_cfg.get("base_position", (0,0,0))),
                       euler=tuple(robot_cfg.get("base_orientation", (0,0,
0))),
                              fixed=robot_cfg.get("fixed_base", True))
     )
     # store robot_entity in a list for later control
)
```

This parses the URDF file and creates an articulated `Entity` in the scene [16]. We supply the base position/orientation from the config. If `fixed=True`, the base link will be rigidly fixed to the world frame (important for stationary robots) [12]. If `fixed=False`, the robot could fall under gravity or be movable (useful for, say, a drone or a free object). We store the returned `robot_entity` (perhaps in a dictionary by name) so we can reference it for control. Genesis supports multiple robots in one scene; each call to `add_entity` adds another robot object.

- **Set Initial Joint Positions:** After adding the robot, but before starting simulation, we set its joints to the desired initial angles from config. URDF itself doesn't specify initial joint states (aside from default 0 or calibration, per ROS conventions) [17]. We have a couple of options:

- Use **hard reset**: Genesis provides `entity.set_dofs_position(array, dofs_idx)` to directly set joint angles ignoring physics [18]. We can use this to initialize the pose instantly before simulation runs. For example:

```
joint_positions = robot_cfg["initial_joint_positions"]
if joint_positions:
    # Genesis expects radians for revolute joints (assuming URDF uses
radians; convert degrees if needed)
    q0 = np.array([np.deg2rad(q) for q in joint_positions],
dtype=np.float32)
    # Get all joint indices of this robot
    dofs_idx = [j.dof_idx_local for j in robot_entity.get_joints()]
    robot_entity.set_dofs_position(q0, dofs_idx_local=dofs_idx)
```

  This will "teleport" the joints to the start angles. Since we haven't started stepping the physics yet, it's safe to do this initialization.
- Alternatively, we could allow the robot to start from whatever default the URDF defines (usually zero positions) and then command it to move to the start pose via the controller (PD control) over a few simulation steps. However, using `set_dofs_position` for initial setup is straightforward.

The reason we might prefer a controlled move instead of an instant set is to avoid any initial penetration if the start pose is colliding. But our chosen home positions should be collision-free by design. If needed, we ensure the initial pose in config is a valid one (like a typical home pose).

- **Configure Joint Controllers:** Genesis by default sets up a PD controller for each joint (dof) of an articulated robot [19]. It also typically parses some dynamics parameters from URDF/MJCF, like joint limits and possibly damping, friction, etc. [20]. However, for optimal "realistic" control, we often need to tune the controller gains. The documentation notes that **high-quality URDF files may include inertia and friction values which Genesis can parse, but manual tuning of PD gains is**

recommended for best behavior [20] . For example, in the Genesis examples, the Franka arm's gains were set to specific values for stable and responsive control [21] [22] . In our tool, we can define default gain values per robot (perhaps stored in the config or a separate lookup) and apply them:

```python
# Example: set PD gains if provided in config
if "kp" in robot_cfg:
    robot_entity.set_dofs_kp(np.array(robot_cfg["kp"], dtype=np.float32))
    robot_entity.set_dofs_kv(np.array(robot_cfg["kv"], dtype=np.float32))
if "force_limits" in robot_cfg:
    low, high = robot_cfg["force_limits"]
    robot_entity.set_dofs_force_range(np.array(low), np.array(high))
```

If the config doesn't explicitly list gains, we rely on Genesis's defaults or URDF values. We will document how to refine these for each robot (for instance, using values similar to the manufacturer's or MoveIt's). Proper gains ensure the robot moves smoothly and holds position with stiffness, improving realism. They also affect collision response – overly stiff gains might cause instability on collision, so tuning is key.

- **Build the Scene:** After adding all entities (robots and objects) and setting initial states, we finalize the scene with `scene.build()`. This triggers Genesis to compile necessary physics kernels and allocate buffers [23] . It's a required step before simulation loop. When `show_viewer=True`, this call will also open the GUI viewer window to render the scene [24] . (Note: The first build can take some time for JIT compilation, but Genesis caches compiled kernels for reuse [25] .)

- **Simulation Loop:** We then enter the main simulation loop. Typically, we will run `scene.step()` in a loop to advance physics by `dt` each step [26] . Since we want a *live simulation*, we'll run this loop continuously in a separate thread or asynchronous manner so that the GUI remains responsive. Each `scene.step()` will update the physics (positions, velocities, handle collisions, etc.). We might aim for real-time stepping (e.g., 100 Hz if dt=0.01). Genesis can run much faster than real-time (it can simulate a simple scene at millions of FPS on GPU) [27] , but we will throttle it to real-time or a reasonable speed for user interactivity. The viewer by default caps at `max_FPS` (60 by default) to avoid rendering too fast [28] .

- **Collision Handling:** Genesis uses an optimized physics engine with fast collision detection algorithms [29] . All robot links that have collision geometry (as specified in the URDF's `<collision>` tags and mesh files) are automatically considered in the physics simulation. This means that if the robot's arm hits itself or an external object, Genesis will detect the contact and generate collision forces to prevent interpenetration. We don't need to manually enable collisions – by adding entities to the scene, collisions are active by default. To ensure **active collision checking**, we will:

- Use the physics simulation (with PD controllers) rather than teleporting joints during normal operation. This way, if a user command tries to move the robot into a wall or itself, the physics engine will either stop the motion or cause a realistic interaction (e.g., the arm pushes against the obstacle and experiences reaction forces).

- Optionally monitor collisions: We can use Genesis APIs to check contact forces. For instance, after each step, `robot_entity.get_dofs_force()` gives the actual force on each joint, including any collision reaction forces [30]. By comparing this to the commanded forces (`get_dofs_control_force`), we can detect if a collision or other disturbance is happening [30]. In debug mode, we might log a warning if a sudden spike in force is detected, indicating a collision event.
- We can also query contacts at the link level if needed (Genesis likely has functions to get contact points between entities; if not directly, this can be deduced from physics forces). This can be used to implement visual indicators or stop motions if a collision is detected beyond a threshold (similar to a real robot's protective stop).

Accuracy and realism: We will set the physics time-step (`dt`) small enough (0.01s or 0.005s) to ensure stable contact simulation. A smaller step yields more accurate physics at the cost of computing resources. We'll also consider enabling *sufficient solver iterations* or other Genesis options if available to handle heavy contacts without penetration (this might be something to explore in Genesis documentation if issues arise). The combination of well-tuned PD controllers and Genesis's collision handling will give a realistic feel close to a physical robot.

Robot Control Modes and Operations

We support two primary control modes for the robot: **Joint Space** and **Cartesian Space**. The user can switch between these modes via the GUI, and the system will reset the robot to the known initial pose upon mode switch (as specified). This reset ensures a consistent starting point in each mode and avoids confusion if the robot was left in some arbitrary pose. We will implement mode switching by commanding the robot back to the initial joint configuration whenever the user toggles modes (or simply reloading the scene, but that's heavier). Logging will note mode changes and resets for transparency.

Joint-Space Control (Manual Joint Jogging)

In joint mode, the user can directly manipulate individual joint angles, similar to jogging a real robot's joints one by one:

- **UI Controls:** We will provide sliders or numeric input fields for each joint. For example, a UR5e has 6 joints (J1…J6). Each slider's range is set from the joint's minimum to maximum limits (these can be read from the URDF model via Genesis after loading: e.g., `robot_entity.get_joint('shoulder_pan_joint').limits` etc., or we store known limits). The user can drag a slider or input a value (in degrees or radians) for each joint angle. We might also include buttons for small incremental moves (e.g., +1° or -1°) for fine control.

- **Sending Commands:** When the user adjusts a joint slider, the program will send a position command to that joint's controller. Genesis's **PD controller** allows us to set a target position for each joint (or a subset of joints) and it will continuously try to move the joint towards that angle [31]. We use `robot_entity.control_dofs_position(target_array, dofs_idx)` for this [32]. For example, if only joint 2 is changed, we could either:

- Update the entire target array of all joints (keeping other joints at their last targets) and call `control_dofs_position(full_target, all_dofs_idx)`.

- Or use the method to control specific dof: e.g., `robot.control_dofs_position(np.array([new_angle]), [index_of_joint2])` while leaving others as-is (Genesis will retain previous targets for other joints since they are not updated, due to the internal state retention [31]).

Genesis's design means **we do not need to send the command every single step** – once a joint target is set, it persists until changed [31] . So our loop can simply update targets when the user makes a change. The PD controller will continuously apply forces to approach that target, and as long as we keep stepping the simulation (`scene.step()`), the robot will move smoothly to the new angle.

- **Speed and Safety:** In joint mode, movements are typically small and user-driven. We can incorporate a joint velocity limit (from config) to avoid moving too fast. If the user drags a slider from 0° to 90° instantly, the PD controller will command a fast motion to close the gap. We might cap the speed by segmenting the motion: for example, if the jump is large, internally interpolate intermediate targets or set a lower proportional gain so that it doesn't overshoot. Another approach is to use **velocity control** for continuous jogging: e.g., a "+" button could call `control_dofs_velocity` with a small constant velocity for a joint until released [32] . However, using position targets is simpler and sufficient for now, because the user can increment in small steps or we can slow down the controller response via gains.

- **Feedback:** We will display the current joint angles (perhaps next to each slider). This can be read from the simulation state via `robot_entity.get_joint(name).qpos` each loop. This gives the actual angle, which should closely track the commanded angle (discrepancies may occur if a collision prevents the joint from reaching the target – the PD controller will try but a collision force might hold it back). If a collision is happening, we might highlight the joint in red or display a warning, and the actual angle will stop changing even if commanded, until the obstruction is removed or the command changes.

Cartesian Space Control (TCP / End-Effector Control)

In Cartesian mode, the user specifies a desired pose or movement of the robot's Tool Center Point (TCP) in space. This is akin to using a robot's teach pendant in world or tool coordinates.

- **Frames of Reference:** The user can choose the reference frame for Cartesian commands:
- **Robot Base Frame:** Coordinates are relative to the robot's base link (origin usually at the robot's base). Moving in X moves the TCP along the robot's X-axis, etc.
- **World Frame:** Coordinates relative to the simulation world (global axes). If the robot base is not aligned with world axes, moving in world frame will feel different (e.g., world Z is always up).
- **Object Frame:** (Optional) If the user selects an object in the scene (say a workpiece), coordinates can be relative to that object's frame. This is useful for tasks like moving the TCP to a specific point on an object regardless of robot orientation.

We will implement this by transforming the target coordinates into the robot's base frame before solving IK. Genesis's IK solver likely expects targets in world coordinates by default (since it gets the link's global pose). To clarify, if the user chooses "base frame", we take the user input (dx,dy,dz relative to base) and transform it to world coordinates using the robot base's transform (which for a fixed base at some orientation might just be a translation/rotation). For "object frame", we take the object's world pose and apply the relative offset to get a world target. Essentially:

$$ ^{world}T_{target} = \; ^{world}T_{frame} \cdot \; ^{frame}T_{target}, $$
where frame could be base or an object. We can get the base or object transform from Genesis (e.g., `robot_entity.get_link('base_link').pose()` if available, or track base position which we know). For an object, if it's static, we stored its position in config; if it's dynamic, Genesis can give its current pose each step.

- **User Input:** The GUI will provide fields to input either absolute coordinates or incremental moves:
- We may allow the user to enter an absolute target position (X, Y, Z) and orientation (perhaps roll-pitch-yaw or quaternion) for the TCP.
- We can also have jog buttons for Cartesian axes (e.g., "move +X 1 cm" or a continuous move while button pressed), similar to real pendants. To start, implementing absolute moves is simpler, but we'll consider small incremental moves for fine control.

- The user can also specify which frame these coords are in (via a dropdown or radio buttons as mentioned). And they can set or select the TCP offset if different tools are used (e.g., a drop-down of preset TCPs or numeric fields for TCP XYZ offset).

- **Inverse Kinematics (IK):** When the user requests a new TCP pose, we need to compute the corresponding joint angles. Genesis offers a built-in IK solver per robot model: `qpos = robot.inverse_kinematics(link=end_effector_link, pos=target_pos, quat=target_quat)` [33] . We will use this directly. For example:

```
end_link = robot_entity.get_link(robot_cfg.get("end_effector_link",
"tool0"))
qpos_solution = robot_entity.inverse_kinematics(
    link = end_link,
    pos  = np.array(target_xyz, dtype=np.float32),
    quat = np.array(target_quat, dtype=np.float32)
)
```

  Here, `end_effector_link` might be specified in config (for instance, URDF for UR5e might have a link named "wrist_3_link" or "tool0" as the flange). If not specified, we choose the last link in the kinematic chain as default. The IK solver returns an array of joint positions (`qpos_solution`). If the target pose is unreachable or there are multiple solutions, Genesis's solver typically returns one feasible solution (possibly the nearest to current). We should handle if it fails (it might return None or an error if no solution).

- **Motion Planning:** Instead of instantly commanding the robot to the IK solution, we should plan a smooth, collision-free trajectory. Genesis integrates the **OMPL** planner accessible via `robot.plan_path(qpos_goal, ...)` [34] . Using OMPL ensures the motion avoids collisions by sampling waypoints. We will do:

```
path = robot_entity.plan_path(qpos_goal=qpos_solution, num_waypoints=200)
```

The `num_waypoints` can be tuned (more waypoints = finer trajectory). In the example, 200 waypoints with dt=0.01 yields a 2-second motion [34]. OMPL will consider the robot's kinematics and the environment (the Genesis integration uses the collision checking of the physics engine under the hood, presumably, to validate waypoints). This should result in a collision-free path if one exists. If no collision-free path is found (e.g., obstacle blocking), the planner might throw an exception or return an empty path. We should handle that (notify the user that the move is not possible).

- **Execute Path:** Once we have the planned waypoints (a list of joint positions), we execute them in sequence. For execution, we can simply loop and at each simulation step set the next waypoint as the joint target:

```
for waypoint in path:
    robot_entity.control_dofs_position(waypoint)  # set PD targets to this
waypoint
    scene.step()  # step the simulation once
    # Optionally, small sleep to sync real-time if needed (Genesis can
simulate faster than real time)
```

Because our simulation thread likely runs continuously, we might integrate this into that loop, or temporarily take over the loop to execute the trajectory. After sending the final waypoint, we might let the simulation run a few extra steps with the final target to let the controller settle [35] (the PD controller might need ~0.1–0.2s to eliminate any small error [36]).

During execution, the Genesis viewer will show the arm moving smoothly. If any unexpected collision happens (maybe a dynamic obstacle moved in the way), the robot might deviate or stop due to physics – the planner doesn't account for moving obstacles, only static ones at plan time. Our collision monitoring can flag if a collision occurs during execution (e.g., via joint forces as described).

- **Direct Cartesian Jogging:** In addition to point-to-point moves via IK+planning, we might support a more direct "jog" mode for Cartesian control: e.g., user holds a button to move +X in world frame continuously. Implementing this can be done by continuously updating a small incremental target:
- We could compute a small step in Cartesian space (say 1 cm) and use IK to get a joint delta, then use velocity control on joints to achieve a steady motion. However, a simpler approach: periodically call IK for a slightly advanced target (like current TCP position + delta) and set that as the joint target.
- Another approach is to use the Jacobian to map a small Cartesian velocity to joint velocities, but that's more complex and not directly provided by Genesis (Genesis doesn't expose Jacobian in the high-level API as far as docs show).

- Given the complexity, for initial version, we might omit continuous hold-to-move, and rely on user specifying small increments and hitting a move command repeatedly (which effectively yields incremental motion).

- **TCP Orientation:** The UI should allow controlling orientation too. This can be through setting specific angles or a convenient option like "maintain current orientation" while moving linearly. For example, a user might want to move the tool in XYZ without rotating it – we can default to hold the orientation constant (which the IK solver will try to respect if that quaternion is given). We can also allow yaw adjustments, etc. In practice, controlling orientation via sliders (roll, pitch, yaw) and then computing IK for that exact orientation works. We just have to be mindful of quaternion vs Euler representation. Genesis expects a quaternion in w,x,y,z format [37] (they used `[0,1,0,0]` as an example which represents a 180° rotation about X-axis, presumably). We can use a utility (maybe `scipy.spatial.transform.Rotation`) to convert Euler angles from UI to quaternions.

- **Multi-Robot Consideration:** If the scene has multiple robots, the GUI should provide a way to select which robot to control in Cartesian mode (and maybe in joint mode too). We could have a dropdown for "Active Robot" if multiple are loaded. When switched, the IK and planning will apply to that selected robot. Each robot would have its own config and maybe different capabilities (e.g., a smaller robot might have different reachable workspace). For simplicity, we can limit to one robot at a time in the UI, but keep the backend capable of handling several.

Graphical User Interface (GUI Design)

We need a **cross-platform GUI** that runs on Linux and macOS. Two good options are **Tkinter** (built-in, lightweight) and **PyQt/PySide** (more powerful, requires installing Qt bindings). Tkinter is likely sufficient for a control panel style interface and is guaranteed to work on both platforms without extra dependencies, so we can start with that for simplicity.

GUI Layout and Elements

We will design a window (or multiple windows) that includes the following elements:

- **Mode Selection:** Radio buttons or a toggle to switch between **Joint Control** and **Cartesian Control** modes. When the user switches mode, an event handler will invoke our mode-switch logic (reset robot to initial pose, update the UI panels accordingly, and log the mode change).
- **Joint Control Panel:** (Visible when Joint mode is active) A set of sliders (or dial/spinbox controls) for each joint:
- Each slider is labeled with the joint name (or number) and shows its current value. The range is set from joint min to max (from URDF limits).
- The user can drag to a new angle; as they do so, we either continuously update the robot target or wait until release – we likely want immediate response, so we'll update continuously (binding the slider's command to send the new angle).
- There can also be small "+" and "–" step buttons for each joint to nudge angles by a fixed increment (e.g., 1°) for fine control.
- A **Reset Joints** button to return all joints to the initial pose (this essentially calls the same function as mode-switch reset, in case user wants to quickly go back).
- A **Joint Values Display:** either the slider itself shows numeric value or we place a read-only textbox next to each slider showing the exact angle (in degrees). This updates live as the robot moves (which in joint mode is basically as commanded, unless an external force moves it off slightly).
- **Cartesian Control Panel:** (Visible in Cartesian mode) Inputs for the target TCP pose:
- Numeric fields for X, Y, Z position (in meters), and orientation (could be three fields for roll/pitch/yaw in degrees, or four for quaternion w,x,y,z; using Euler angles is more user-friendly).
- Dropdown or radio to select the **reference frame** (Base, World, or pick from list of objects). If objects exist, we can populate a list of object names.

- A field or dropdown for **TCP offset** selection: e.g., "default TCP" vs maybe predefined tool profiles. Initially, we can use one default (as given in config). In future, allow editing TCP offsets.
- **Execute Move** button: When clicked, the system will take the XYZ and orientation inputs, compute IK and plan a path, then execute it. We should provide feedback:
    - If IK fails or no plan is found, show an error popup or message label (e.g., "Target not reachable or path obstructed!").
    - Otherwise, maybe disable inputs while the motion is executing (to prevent new commands mid-trajectory), or allow queuing but that's complex. Simpler: ignore new commands until done, or provide a "Stop" button to abort.
- **Incremental Jog Buttons:** Optionally, for each Cartesian axis and rotation, small buttons like "X+", "X-", "Y+", etc. Pressing one could immediately command a small step (like 1 cm or 5°). This essentially would take the current TCP pose, offset it slightly, and either directly set as target or do a short plan. We have to be careful to not queue too many fast commands – but since simulation is continuous, doing one step at a time is fine.

- **Current Pose Display:** It's useful to show the robot's current TCP position and orientation. We can compute forward kinematics from the joint states via Genesis: e.g., `end_link.get_pose()` if available. If Genesis does not provide a direct FK call, we could attach a dummy object to the TCP or compute via transformation of each link (Genesis likely has something like `link.get_pose()` in world coordinates). We update this display every few steps. The user then sees where the robot is in space, which helps in deciding the next move.

- **Status/Log Output:** A text area or status bar can show runtime messages. This includes debug info like "Mode switched to Cartesian, robot reset to home position.", "Executing motion plan with 200 waypoints…", "Collision detected between link_5 and object Table at time 12.3s" (if we log such events). In debug mode, this could be very verbose. We might implement logging to both console and a scrollable text widget in the GUI for convenience. Using the Python `logging` module, we can attach a handler to feed log records to the GUI.

- **Menu/Toolbar:** If using Tkinter, we might keep it simple (no complex menu). If using Qt, maybe a menubar for options (like loading a different config or toggling debug). But initially, not needed.

GUI and Simulation Integration

The main challenge is keeping the simulation running while the GUI is responsive. We have two approaches:

1. **Background Thread for Simulation:** Start a separate thread for the physics loop. This thread will continuously call `scene.step()` at the set interval. It will also check for any new commands from the GUI (which could be stored in thread-safe shared variables or a queue). For example, when a user moves a slider, the GUI thread updates a target angle variable; the sim thread reads it and calls `control_dofs_position`. Because Genesis is thread-safe for the kind of operations we do (we should confirm, but controlling and stepping should be done on the same thread ideally), we might actually do **all Genesis calls in the sim thread**. The GUI thread simply sends requests.
2. We must ensure thread safety: Tkinter doesn't allow GUI updates from a non-main thread, so the sim thread cannot directly create windows or update widgets. But it can log data or push events. For instance, if the sim thread detects a collision, it could use a `queue.Queue` to send a message to the

main thread, which the main thread periodically polls (via Tkinter `.after()` scheduling) to display in the GUI.

3. We will run the sim thread at (roughly) real-time speed. We can use `time.sleep(dt)` each loop iteration to sync to real time, since Genesis itself can run faster. If we want to allow faster-than-real (for quick simulations), we could skip sleeping, but for a user-interactive tool, real-time is expected.
4. When the user hits “Execute Move”, we might temporarily take control in the sim thread to perform the planned trajectory execution. The sim thread can handle this sequentially (for loop through waypoints). The GUI can show a progress or just be disabled during that short period (e.g., 2 seconds).

5. If the user closes the GUI or hits a stop, we need to signal the thread to break out of the loop and clean up (stop viewer, etc.). We can set a flag that the sim thread checks.

6. **Periodic Step in Main Thread:** Alternatively, we can avoid threads by using Tkinter’s `after` method to schedule `scene.step()` calls in the main loop. For example, call `scene.step()` every 10 ms via `root.after(10, step_once)`. This way, the GUI loop is essentially driving the simulation. This can work since 10 ms is quick, but if the simulation step takes longer (unlikely, Genesis is very fast for few bodies), the GUI might become sluggish. Also, executing a planned path would mean scheduling a series of steps; this is doable but a bit complex to manage state (we’d need to keep track of waypoint index between calls).

7. A hybrid approach: use `after` to achieve ~100 Hz stepping and also check a queue of control commands, similar to how one would integrate with an event loop.
8. Since the user emphasized *live simulation and responsiveness*, the threaded approach might be safer to keep the GUI snappy. We just have to carefully coordinate data.

Given complexity, we lean towards using a **background simulation thread**. We will implement thread-safe communication: - A shared data structure (protected by a lock) for current joint targets. Or even simpler, we can call Genesis control functions directly from the GUI events (since those just set internal targets, which are fine to call outside the simulation step as long as the scene exists). But to avoid race conditions, it’s better if all Genesis modifications happen in one thread. We may instead send a message “set joint i to X” to the sim thread. - We can use Python’s `queue.Queue`: GUI thread `queue.put(("set_joint", i, angle))`, the sim thread in its loop checks `while not queue.empty(): process command`. - Similar for starting a Cartesian move: GUI thread sends (“plan_and_execute”, target_pose) and the sim thread handles the IK and planning (which are heavy computations) and then performs it. This keeps all those operations in the sim thread, avoiding concurrency issues with Genesis internal state.

The GUI thread mostly manages widgets and sends commands, the sim thread manages physics and robot state.

Logging and Debugging

We incorporate robust logging to monitor the tool’s behavior: - Use Python’s `logging` module, configured in `logging_utils.py` to output timestamps and levels. We’ll have a global logger for Simforge. - **Debug Mode:** If the user starts the tool with a `--debug` flag or sets a config, we set logger to DEBUG level, otherwise INFO or WARNING by default. - Throughout the code, log important events: - On startup: log Genesis version (we can get via `gs.__version__`) and which backend was selected (Genesis likely logs that too, but we can explicitly log the `gs.get_active_backend()` if available). - When adding robots, log "Loaded UR5e robot from UR5e.urdf at position (0,0,0)." - Mode switches: "Switched to Joint Control mode – resetting robot to home position." - When user changes a joint: "Joint2 target set to 45.0°". - Cartesian commands: "Received target TCP (x=0.5,y=0.0,z=0.3 in world frame) – solving IK…", then "IK solution found, planning path…", "Executing trajectory with N waypoints." - Collisions: if detected, "Collision detected: Joint3 experiencing 50N internal force – possible contact with environment." - Etc. - These logs can go to the console, and if debug GUI is open, also show on the GUI's log panel. This transparency helps the user understand what's happening under the hood, much like seeing a teach pendant's log or ROS console. - We will also ensure **accurate position reporting**: for example, if in debug, each loop we could print the robot's current joint angles or TCP coordinates. This could be too verbose at 100 Hz, so maybe throttle or on demand. Alternatively, a "Show state" button could dump the state.

Cross-Platform GUI Considerations

Tkinter is quite stable across Linux and macOS. We should be aware of a couple of things: - On macOS, to use Tkinter's main loop, we may need to ensure we're not running in a virtual environment that lacks the Tcl/Tk frameworks. Usually the Python provided in macOS or installed via python.org includes Tkinter support. PyQt/PySide would require a Qt installation, which is heavier. - The Genesis viewer on macOS will use Metal for rendering. It opens a separate window (GL/GUI) which should work alongside the Tkinter window. Users will have two windows: one for controls (our Tkinter) and one for the 3D view (Genesis). We should test that interaction (sometimes multiple GUI frameworks can conflict, but since Genesis viewer is internal C++/OpenGL/Metal, it should be fine). - We must instruct users on macOS to allow the Python app to use the screen (sometimes macOS has security for screen recording when using Metal windows – but likely not an issue). - On Ubuntu, ensure an X display is available (if running without display, set `show_viewer: false`). For GUI, just need an X environment, which is typical in desktop Ubuntu.

Summary and Next Steps

In this research, we outlined how to set up Genesis on Ubuntu 24.04 (and macOS) and how to architect the Simforge tool for realistic robot simulation. In summary:

- **Genesis Setup:** Install PyTorch and genesis-world via pip, choose appropriate backend (CUDA vs MPS) at runtime. Genesis is cross-platform and extremely fast, with optimized collision detection [29] [3].
- **Project Structure:** Organize code into a clean Python package with modules for simulation, control, GUI, and configs. Use a workspace initialization to provide template YAML configs for each robot.
- **Simulation Details:** Use Genesis's API to load URDF models (which include joint and collision data) [16], add environment objects, and configure physics. Keep the robot base fixed unless a floating base is needed [12]. Set initial joint positions from config and tune PD controllers for realistic movement [20]. Leverage Genesis's physics engine to handle collisions and gravity so the robot behaves naturally.
- **Control Implementation:** Provide joint-level control via direct position targets on the built-in PD controller (simulating motor control) [31]. Provide Cartesian control by solving IK for end-effector moves and using OMPL-based planning for collision-free trajectories [33] [34]. Ensure the user can operate in different reference frames by coordinate transforms.
- **GUI and UX:** Create a responsive GUI (e.g., with Tkinter) with a mode toggle, joint sliders, Cartesian input fields, and actionable buttons. Use a background simulation loop to update physics

continuously. Offer feedback through on-screen displays of current positions and logging of events. The experience should mimic a real teach pendant – the user can jog joints or move the tool while seeing the robot move live, and the system prevents or reacts to collisions realistically.
- **Collision Robustness:** By using physics-based motion (rather than teleporting), any collision will manifest as resistance and force feedback in the simulation. Our tool will actively check for such collisions (via joint torques or planned path validity) and could stop movements if necessary, akin to a safety stop. This is analogous to how MoveIt2 continuously checks planned trajectories for self or environment collisions – our implementation uses Genesis's real-time collision checking to achieve the same effect during execution.

Following these guidelines, one can start implementing the Simforge tool from scratch. Begin with the virtual environment and Genesis installation, then incrementally develop the components (perhaps start with loading a robot and manually controlling joints in a small script, then build out the GUI). The result will be a robust simulation platform where controlling a virtual UR5e or Meca500 feels close to controlling the real robot, complete with realistic physics and collision response. By structuring the code well and using Genesis's powerful features, Simforge will be maintainable and extensible (e.g., adding new robot models or integrating sensors in the future).

Sources:

- Genesis Documentation – *Installation and Features* [3] [2] [9] [11] [12]
- Genesis Documentation – *Control and IK/Planning* [20] [31] [30] [33] [34]
- Genesis Website – *Physics Engine Overview* [29] (highlights optimized collision and performance)
- PyPI – *OMPL Library* [6] (for motion planning integration)

---

[1] [2] [3] [4] [7] [8] ◌ Installation — Genesis 0.3.3 documentation
https://genesis-world.readthedocs.io/en/latest/user_guide/overview/installation.html

[5] [21] [22] [28] [33] [34] [35] [36] [37] Inverse Kinematics & Motion Planning — Genesis 0.3.3 documentation
https://genesis-world.readthedocs.io/en/latest/user_guide/getting_started/inverse_kinematics_motion_planning.html

[6] ompl · PyPI
https://pypi.org/project/ompl/

[9] [10] [11] [12] [13] [14] [15] [16] [23] [24] [25] [26] Hello, Genesis — Genesis 0.3.3 documentation
https://genesis-world.readthedocs.io/en/latest/user_guide/getting_started/hello_genesis.html

[17] Can I set initial joint positions via URDF? - ROS Answers archive
https://answers.ros.org/question/392592

[18] [19] [20] [30] [31] [32] ◌ Control Your Robot — Genesis 0.3.3 documentation
https://genesis-world.readthedocs.io/en/latest/user_guide/getting_started/control_your_robot.html

[27] [29] Genesis
https://genesis-embodied-ai.github.io/