This document is supplementary material to “Outplaying Elite Table Tennis Players with an Autonomous Robot.”

Approximate Python Code

Here we provide approximate Python code that replicates the most critical aspects of our work. For the purposes of illustration, the code presented here is implemented in less computationally efficient ways than our actual implementation; has abstracted away many of the system implementation details such as remote process communication and check-pointing; and is written more directly than our actual implementation which supported wide configuration capabilities that otherwise detract from readability.

```python
def train_loop():
    models = Models(
        policy=make_policy_network(),
        af1=make_q_network(),
        af2=make_q_network(),
        af1_target=make_q_network(),
        af2_target=make_q_network(),
    )
    # Sync prediction and target networks before start
    copy_weights(models.af1.trainable_params, models.af1_target.trainable_params)
    copy_weights(models.af2.trainable_params, models.af2_target.trainable_params)

    opts = Optimizers(
        policy=keras.optimizers.Adam(learning_rate=1.0e-4),
        critic=keras.optimizers.Adam(learning_rate=1.0e-4),
    )

    param_server = start_parameter_server(models.policy)
    replay_server = start_experience_replay_server()
    table_datasets = [
        make_dataset_for_table(replay_server, table_name)
        for table_name in TABLE_NAMES
    ]
    task_server = start_task_factory_server()

    # Wait for MINIMUM_STEPS to be recorded to replay before we start any training
    replay_server.block_and_wait(MINIMUM_STEPS)

    epoch = 0
    while True:
        for _ in range(BATCHES_PER_EPOCH):
            transition_data = sample_data(table_datasets)
            # see SAC code listing for implementation
            losses = step_sac(models, opts, transition_data)
            write_metrics(losses)
        param_server.update_params(models.policy)
        epoch += 1
        # do a periodic evaluation of the policy
        if epoch % EPOCHS_PER_EVAL == 0:
            task_server.needs_eval = True

def sample_data(table_datasets: List[tf.data.Dataset]) -> Transition:
    # For simplicity, this code omits handling the edge case when only a subset of
    # the tables have data to sample.
    # Before concat, each dataset has a different batch size based on the table
    # weights
    batches = [next(iter(dataset)) for dataset in table_datasets]
    batches = nested_concat(batches, axis=0)

    return Transition(
        obs=batches.obs,
        action=batches.action,
        reward=batches.reward,
        next_obs=batches.next_obs,
        done=batches.done,
    )
```
Listing 1: Training Loop Code

```python
def step_sac(m: Models , opts: Optimizers , t: Transition) -> Losses:
    critic_params = m.af1.trainable_params + m.af2.trainable_params
    target_critic_params = m.af1_target.trainable_params + m.af2_target.trainable_params

    # these values do not need gradients traced through them
    policy_head_next = m.policy.policy_head(t.next_obs)
    actions_next = m.policy.sample_from_head(policy_head_next)
    log_prob_next = m.policy.log_prob(policy_head_next, actions_next)

    # update policy
    with tf.GradientTape() as tape:
        policy_head = m.policy.policy_head(t.obs)
        sampled_actions = m.policy.sample_from_head(policy_head)
        log_prob = m.policy.log_prob(policy_head, sampled_actions)
        policy_loss = policy_loss(
            log_prob_samples=log_prob,
            q1_sampled=m.af1.action_value(t.obs, sampled_actions),
            q2_sampled=m.af2.action_value(t.obs, sampled_actions),
        )
    opts.policy.minimize(policy_loss, m.policy.trainiable_params, tape)

    # update critic networks
    with tf.GradientTape() as tape:

        critic_loss = critic_loss(
            q1_observed=m.af1.action_value(t.obs, t.action),
            q2_observed=m.af2.action_value(t.obs, t.action),
            q1_sampled_next=m.af1_target.action_value(t.next_obs, actions_next),
            q2_sampled_next=m.af2_target.action_value(t.next_obs, actions_next),
            log_prob_next=log_prob_next,
            reward=t.reward,
            done=t.done,
        )

    opts.critic.minimize(critic_loss, critic_params, tape)

    # move target network params toward prediction networks
    for pred_param, target_param in zip(critic_params, target_critic_params):
        target_param.assign(
            SMOOTH_FACTOR * pred_param + (1.0 - SMOOTH_FACTOR) * target_param
        )

    return Losses(policy_loss, critic_loss)

def policy_loss(
    log_prob_samples: tf.Tensor, # (N, 1)
    q1_sampled: tf.Tensor, # (N, 1)
    q2_sampled: tf.Tensor, # (N, 1)
) -> tf.Tensor:
    min_q = tf.minimum(q1_sampled, q2_sampled)
    return tf.math.reduce_mean(ALPHA * log_prob_samples - min_q)

def td_target(reward: tf.Tensor, done: tf.Tensor, discount: tf.Tensor, next_v: tf.Tensor) -> tf.Tensor:
    return reward + (1 - done) * discount * next_v

def critic_loss(
    q1_observed: tf.Tensor,  # (N, 1)
    q2_observed: tf.Tensor,  # (N, 1)
```

```python
    q1_sampled_next: tf.Tensor,  # (N, 1)
    q2_sampled_next: tf.Tensor,  # (N, 1)
    log_prob_next: tf.Tensor,  # (N, 1)
    reward: tf.Tensor,  # (N, 1)
    done: tf.Tensor,  # (N, 1)
):
    min_q = tf.minimum(q1_sampled_next, q2_sampled_next)
    q_target = tf.stop_gradient(td_target(reward, done, DISCOUNT, min_q - ALPHA *
    log_prob_next))
    q1_loss=tf.keras.losses.MeanSquaredError(q_target, q1_observed),
    q2_loss=tf.keras.losses.MeanSquaredError(q_target, q2_observed),
    return q1_loss + q2_loss
```

Listing 2: SAC Code

```python
def get_task(context: DistributedTaskContext) -> Task:
    # The returned task can be either an experience collection task, or an evaluation
     task
    task_server = get_task_server()
    # The model_state contains information like env_steps, epoch_number, policy model
    # such that the desired logic can be implemented in the tasks
    model_state = context.model_state
    if task_server.needs_eval:
        # Time for periodic evaluation
        task_server.needs_eval = False
        # In evaluation task only the model_state is needed
        return DistributedEvalTask(
            model_state=model_state,
        )
    else:
        # Standard collect task
        # replay_configuration contains the configuration
        # to send data to replay buffer
        replay_configuration = context.replay_config
        # In collect task both model_state and replay_configuration are needed
        return DistributedCollectTask(
            model_state=model_state,
            replay_config=replay_configuration,
        )
```

Listing 3: The task factory is run in a separate process and is remotely queried by rollout workers.

```python
def rollout_loop(context: DistributedTaskContext):
    faoc = make_faoc_controller()
    env_server = get_environment_server()
    replay_server = get_experience_replay_server()
    while True:
        task = get_task(context)
        # Learning starts after collecting a min amount of transitions
        # before that, use Ornstein-Uhlenbeck noise policy
        if task.model_state.epoch.value == 0:
            agent_policy = OrnsteinUhlenbeckPolicy()
        else:
            agent_policy = task.model_state.policy

        env_server.launch_environment() # Start the environment
        obs = env_server.get_first_obs()
        p_obs = preprocess(obs)  # Preprocess observation for policy input
        done = False
        while not done:
            action = agent_policy(p_obs)  # Get action from policy
```

```python
            reference_traj = faoc(action)  # Get reference trajectory from FAOC
            env_server.step(reference_traj) # Step env with robot ref traj
            # Get the next observation 32ms later
            next_obs = env_server.wait_for_next_action_intent()
            reward = compute_reward(obs, next_obs)  # step reward
            done = env_server.check_termination_event() # Check if episode ended

            if not task.is_eval:
                transition = Transition(
                    obs=p_obs,
                    action=action,
                    reward=reward,
                    next_obs=preprocess(next_obs),
                    done=done,
                )
                # Write transition to replay buffer
                replay_server.write(transition, task.replay_config)

            obs = next_obs
            p_obs = preprocess(next_obs)

        write_metrics(env_server.get_episode_metrics()) # Write episode metrics
```

Listing 4: Rollout worker code. 30 instances are run in parallel.

```python
import pygad

def evaluation_wrapper(ga_instance: pygad.GA, solution: np.ndarray, solution_idx: int) -> float:
    """Wrapper function for evaluating a genome"""
    genes = genome_evaluation(solution)

    # Get the serve trajectory from the genes and the MPC
    solution = genes_to_serve(genes)
    if solution is None:
        return 0.0

    # Run the physics on the robot trajectory and record the behavior of the ball
    ball, robot, racket_contacts, robot_contacts, bounces, net_hits = simulate_serve(solution)

    # If the serve is not legal, award a small fitness based on the illegality
    legal, fitness = is_serve_legal(ball, robot, racket_contacts, robot_contacts, bounces,
     net_hits)

    if legal:
        # Compute the fitness of the serve based on the overall trajectory and
        # the ball state (pos, vel, spin) at the second bounce
        fitness += compute_fitness(ball, ball[bounces[1]])

    return fitness

def serve_training() -> np.ndarray:
    gen_alg = pygad.GA(
        # Tunable parameters
        num_generations=100,
        num_parents_mating=6,
        mutation_num_genes=3,
        stop_criteria="saturate_40",
        # Fixed parameters
        random_mutation_min_val=-0.20,
        random_mutation_max_val=0.20,
        num_genes=9,
        fitness_func=evaluation_wrapper,
        init_range_low=0.0,
        init_range_high=1.0,
        gene_space={"low": 0.0, "high": 1.0},
    )
    gen_alg.run()
```

```
    genes, _, _ = gen_alg.best_solution()
    return genes
```

Listing 5: Serve training pipeline

```
class BallContact(Enum):
    AIR = 0  # ball is airbone, i.e., no contact at all
    RACKET = 1  # ball contact with racket
    TABLE = 2  # ball contact with table


class BallPosition:
    """Timestamped external ball position obtained from the overhead perception system triangulations"""

    timestamp: float
    x: float  # x position [m]
    y: float  # y position [m]
    z: float  # z position [m]


class BallVelocity:
    """Timestamped ball velocity estimated from successive ball positions in the ball state estimator"""

    timestamp: float
    x: float  # x velocity [m/s]
    y: float  # y velocity [m/s]
    z: float  # z velocity [m/s]


class GazeControlSystem:
    """System consisting of pan/tilt galvanometer mirrors and an electrically tunable focal power lens"""

    pan_angle: float  # pan angle of the vertical mirror
    tilt_angle: float  # tilt angle of the horizontal mirror
    focal_power: float  # focal power of the liquid lens

    T_gcs_world: np.ndarray  # homogeneous transform from world to gcs

    def track_ball_position(self, position_world: BallPosition):
        pass  # set pan/tilt angles to track the ball and adjust focal power based on distance (pre-calibrated)


class Event:
    """Single definition for clarity; An event can either be in
    * 2D pixel coordinates,
    * 3D metric coordinates, or
    * unit spherical coordinates"""

    timestamp: float  # in seconds
    x: int | float  # x-coordinate [pixel/unit]
    y: int | float  # y-coordinate [pixel/unit]
    polarity: bool  # on/off events

    # optional
    z: int | float  # z-coordinate [pixel/unit]
    polar: float  # polar coordinate [rad]
    azimuth: float  # azimuth coordinate [rad]


class EventCamera:
    # note: in practice, we use a 320x320 pixel center region of interest (ROI) for efficiency
    width: int  # sensor width [pixels]
    height: int  # sensor height [pixels]

    polarities = [0]  # we only accumulate the OFF polarity in practice

    def generate_events(self) -> Generator[Event]:
```

```
        while True:
            yield Event()

    def events_to_time_surface(self, events: list[Event]) -> np.ndarray:
        """a time surface is a (per-polarity) image of the most recent event timestamp at every (x,y) location"""
        time_surface = np.zeros(self.height, self.width, len(self.polarities))
        for timestamp, x, y, polarity in events:
            time_surface[y, x, polarity] = timestamp
        return time_surface

    @classmethod
    def time_surface_to_events(time_surface: np.ndarray) -> Generator[Event]:
        width, height, polarities = time_surface.shape
        for x in range(width):
            for y in range(height):
                for polarity in polarities:
                    timestamp = time_surface[y, x]
                    if timestamp != 0:
                        yield Event(timestamp=timestamp, x=x, y=y, polarity=polarity)


class BallDetection:
    center_x: float  # x-coordinate of the bounding circle center [pix]
    center_y: float  # x-coordinate of the bounding circle center [pix]
    radius: float  # radius of the bounding circle [pix]
    confidence: float  # confidence in the bounding cirlce [0-1]


class BallDetectorCNN:

    tensorrt_model: Callable[[list[np.ndarray]], list[BallDetection]]  # trained with torch and converted using trtexec

    def detect_balls(self, time_surfaces: list[np.ndarray]) -> list[BallDetection]:
        return self.tensorrt_model(time_surfaces)

    def mask_ball(time_surface: np.ndarray, ball_detection: BallDetection):
        """mask the time surface such that only events on the ball surface remain"""
        x, y, radius, _ = ball_detection
        mask = np.zeros_like(time_surface, dtype=bool)
        circle_mask = cv.circle(mask, (x, y), radius, fill=True)
        return time_surface & circle_mask


class SpinEstimate:
    """Timestamped angular velocity (spin) estimate with associated uncertainty (covariance or variance)"""

    timestamp: float
    spin: np.ndarray  # 3d angular velocity

    # depending on the spin esimation method (CNN or CMax) we have different measures of uncertainty
    covariance: float  # heterescedastic uncertainty learned by the spin estimation CNN (lower = better)
    variance: float  # variance/contrast of image of warped events spin spin refinement with CMax (higher = better)

    @property
    def axis(self):
        return self.spin / self.magnitude()

    @property
    def magnitude(self):
        return np.linalg.norm(self.spin)


class SpinEstimatorCNN:
    """CNN trained on pseudo ground-truth labels obtained with CMax that also predicts heteroscedastic uncertainty"""

```

```python
    tensorrt_model: Callable[[list[np.ndarray]], list[SpinEstimate]]  # trained with torch and converted using trtexec

    def estimate_spins(self, time_surface: list[np.ndarray]) -> list[SpinEstimate]:
        """batch inference"""
        return self.tensorrt_model(time_surface)


class BallStateEstimator:
    """Pseudo ball state estimator"""

    position: BallPosition
    velocity: np.ndarray
    spin: SpinEstimate
    ball_physics_model: Callable[[float, float], Any]  # models air drag, gravity, etc.

    # the main loop needs to know if a ball contact occurred; we model this with callbacks
    detect_ball_contact: Callable[[BallPosition, BallVelocity], BallContact]  # detect if ball contact occurred
    contact_callback: Callable[[BallContact]]  # notify caller about the ball contact

    # the latency is estimated using the difference in hardware trigger and triangulation arrival timestamp
    current_latency: float

    def predict(self, timestamp: float) -> Any:
        """predict the state of the ball given a time point"""
        ball_contact = self.detect_ball_contact(self.position, self.velocity)
        self.contact_callback(ball_contact)
        if ball_contact == BallContact.TABLE:
            ...  # apply table contact model to state variables to predict post-contact spin estimate
        elif ball_contact == BallContact.RACKET:
            self.reset()
        elif ball_contact == BallContact.AIR:
            self.ball_physics_model(timestamp, self.current_latency)
        ...
        return self.position, self.spin

    def update(self, observation: BallPosition | SpinEstimate) -> Any:
        """update the filter with a new observation (and its associated uncertainty)"""
        self.current_latency = observation.timestamp - ...
        ...
        return self.position, self.spin

    def reset(self):
        """reset filter"""
        ...

    def register_contact_callback(self, callback: Callable[[BallContact]]):
        self.contact_callback = callback


class MedianFilter:
    """Simple median filter to reject false ball detections"""

    window_size: int
    observations: list[Any]

    def filter(self, observation: Any) -> Any:
        """median filter the observation and immediately return the result"""
        # note: if observation is list-like, the median is computed per component
        if len(self.observations) > self.window_size:
            self.observations.pop(0)
        self.observations.append(observation)
        return np.median(self.observations)

    def reset(self):
        self.observations = []


class ContrastMaximization:
```

```python
    """Note: iterative version for clarity; we have implemented this on the GPU"""

    def events2d_to_3d(events2d: list[Event], ball_detection: BallDetection) -> Generator[Event]:
        """Convert 2d events to 3d events on the unit sphere using the ball detection (orthographic projection)"""
        center_x, center_y, radius, _ = ball_detection
        for event2d in events2d:
            timestamp, x2d, y2d, polarity = event2d
            x3d, y3d = ((x2d, y2d) - (center_x, center_y)) / radius
            z3d = -np.sqrt(1 - (x3d**2 + x3d**2))
            yield Event(timestamp=timestamp, x=x3d, y=y3d, z=z3d, polarity=polarity)

    def warp_events3d_by_spin(events3d: list[Event], spin_estimate: SpinEstimate) -> Generator[Event]:
        """Rotate 3d events to a reference timestamp using a candidate spin estimate"""
        timestamp_ref = events3d[-1].timestamp
        for event3d in events3d:
            timestamp, x, y, z, polarity = event3d
            td = event3d.timestamp - timestamp_ref
            rotation = Rotation.from_rotvec(-spin_estimate * td)
            x_rot, y_rot, z_rot = rotation.apply((x, y, z))
            yield Event(timestamp=timestamp, x=x_rot, y=y_rot, z=z_rot, polariy=polarity)

    def event3d_to_spherical(events3d: list[Event]) -> Generator[Event]:
        for event3d in events3d:
            timestamp, x, y, z, polarity = event3d
            polar, azimuth = np.arcsin(z), np.arctan2(y, x)
            yield Event(timestamp=timestamp, polar=polar, azimuth=azimuth, polarity=polarity)

    def spherical_to_image_of_warped_events(events_spherical: list[Event], size: tuple[int, int]) -> np.ndarray:
        """Generate an image of warped events in spherical coordinates"""
        height, width = size
        image_of_warped_events = np.zeros((height, width))
        for event_spherical in events_spherical:
            polar, azimuth = event_spherical
            x = width * (azimuth / (2 * np.pi) + 0.5)
            y = height * ((0.5 - polar) / np.pi)
            image_of_warped_events[y, x] += 1
        return image_of_warped_events


class SpinRefinerCMax:
    """Returns the spin candidate that best explains the ball rotation by contrast-maximizing (CMax) the image of warped events"""

    contrast_maximizer: ContrastMaximization

    def refine_spin_estimate(
        self, events2d: list[Event], spin_candidates: SpinEstimate, ball_detection: BallDetection
    ) -> SpinEstimate:

        # note: in practice, this for loop is executed in parallel on the GPU (for all spin candidates simultaneously)
        best_spin_candidate = SpinEstimate()
        max_variance = 0
        for spin_candiate in spin_candidates:
            events3d = self.contrast_maximizer.events2d_to_3d(events2d, ball_detection)
            events3d_warped = self.contrast_maximizer.warp_events3d_by_spin(events3d, spin_candiate)
            events_spherical = self.contrast_maximizer.event3d_to_spherical(events3d_warped)
            image_of_warped_events = self.contrast_maximizer.spherical_to_image_of_warped_events(events_spherical)
            variance = np.var(image_of_warped_events)
            if variance > max_variance:
                best_spin_candidate = spin_candiate
                best_spin_candidate.variance = variance

        return best_spin_candidate


class BallTrackingAndSpinEstimation:
```

```python
    """Main class modeling the Gaze Control System operation: ball tracking and spin estimation """

    gaze_control_systems: list[GazeControlSystem]
    event_cameras: list[EventCamera]
    ball_state_estimator: BallStateEstimator

    # time surfaces
    time_surfaces: list[np.ndarray]  # shared between threads

    # CNN-based ball detection
    ball_detector_cnn: BallDetectorCNN
    ball_detections_filtered: list[BallDetection]  # shared from ball detection -> spin estimation thread

    # CNN-based spin estimation
    spin_estimator_cnn: SpinEstimatorCNN
    best_spin_estimate: SpinEstimate  # shared from spin estimation -> spin refinement thread
    best_spin_index: int  # shared from spin estimation -> spin refinement thread

    # CMax-based spin refinement
    spin_refiner_cmax: SpinRefinerCMax
    refined_spin_estimate_cmax: SpinEstimate  # shared from spin refinement -> spin estimation thread

    # publishers/subscribers
    triangulation_subscriber: Subscriber
    spin_estimation_publisher: Publisher

    def init(self, num_systems=3):
        self.gaze_control_systems = [GazeControlSystem() for _ in range(num_systems)]
        self.median_filters = [MedianFilter(window_size=5) for _ in range(num_systems)]

        # start all threads
        self.should_exit = False  # set by user interaction, e.g. Ctrl+C

        # initialize spin estimates
        self.best_spin_estimate = None
        self.refined_spin_estimate_cmax = None

        # register a contact callback (e.g., ball with racket or table) to reset spin estimates
        self.ball_state_estimator.register_contact_callback(self.contact_callback)

        # the ball triangulations are obtained from the overhead perception system at 200Hz in a callback
        # the ball position is filtered and extrapolated in time which allows for smooth as-fast-as-possible tracking
        self.triangulation_subscriber.callback(self.ball_triangulation_callback)
        self.ball_tracking_thread()

        # 1) detect ball, 2) estimate spin, and 3) refine spin (asynchronously but with dependence on each other)
        self.ball_detection_thread()
        self.spin_estimation_thread()
        self.spin_refinement_thread()

    @callback
    def contact_callback(self, ball_contact: BallContact):
        # reset spin estimates when racket or table contact is detected
        if ball_contact is BallContact.RACKET or BallContact.TABLE:
            self.best_spin_estimate = None
            self.refined_spin_estimate_cmax = None

    @callback
    def ball_triangulation_callback(self, ball_position: BallPosition):

        # estimate ball state (and latency); in practice, this is called at 5ms intervals (200Hz)
        self.ball_state_estimator.update(position=ball_position)

    @thread
    def ball_tracking_thread(self):
        """consumes: ball state estimate, produces: per-system mirror/lens control commands"""
```

```python
        while not self.should_exit:

            # predict the current state of the ball (accounting for triangulation latency and aerodynamics)
            ball_position_predicted, _ = self.ball_state_estimator.predict(time.now())

            # track the ball with each gaze control system
            (gcs.track_ball_position(ball_position_predicted) for gcs in self.gaze_control_systems)

    @thread
    def ball_detection_thread(self):
        """consumes: per-system time surfaces, produces: per-system filtered ball detections"""
        while not self.should_exit:
            # note: in practice, we accumulate events directly into the time surface for efficiency
            self.time_surfaces = [
                event_camera.events_to_time_surface(event_camera.generate_events())
                for event_camera in self.event_cameras
            ]

            # detect ball on time surfaces using batched inference
            ball_detections = self.ball_detector_cnn.detect_balls(self.time_surfaces)

            # apply median filter to ball detections
            self.ball_detections_filtered = [
                median_filter(ball_detection)
                for median_filter, ball_detection in zip(self.median_filters, ball_detections)
            ]

    @thread
    def spin_estimation_thread(self):
        """consumes: per-system time surfaces and ball detections, produces: single (possibly refined) spin estimate"""
        while not self.should_exit:

            # mask events outside of the ball bounding circle
            time_surfaces_masked = [
                BallDetectorCNN.mask_ball(time_surface, ball_detection)
                for time_surface, ball_detection in zip(self.time_surfaces, self.ball_detections_filtered)
            ]

            # estimate spin on masked time surfaces using batched inference
            spin_estimates = self.spin_estimator_cnn.estimate_spins(time_surfaces_masked)

            # best spin estimate has lowest covariance/uncertainty
            self.best_spin_index = np.argmin(spin_estimates, key=lambda spin_estimate: spin_estimate.covariance)
            best_spin_estimate = spin_estimates[self.best_spin_index]

            # filter best spin estimate before making it available for refinement
            _, self.best_spin_estimate = self.ball_state_estimator.update(spin=best_spin_estimate)

            # if available with low uncertainty (= high variance), overwrite the CNN's best spin estimate magnitude
            # we assume that the spin axis predicted by the CNN is already accurate, thus only refine the magnitude
            MIN_VARIANCE = 0.01
            if self.refined_spin_estimate_cmax is not None and self.refined_spin_estimate_cmax.variance > MIN_VARIANCE:
                self.best_spin_estimate.magnitude = self.refined_spin_estimate_cmax.magnitude

            # publish for robot control
            self.spin_estimation_publisher.publish(self.best_spin_estimate)

    @thread
    def spin_refinement_thread(self):
        """consumes: single spin estimate, produces: single refined spin estimate"""
        while not self.should_exit:

```

```
            # only refine once we have a spin estimate with high enough magnitude
            MIN_SPIN_MAGNITUDE = 50  # rad/s
            if self.best_spin_estimate is None or self.best_spin_estimate.magnitude < MIN_SPIN_MAGNITUDE:
                continue

            # generate N magnitude-varying spin candidates around the current best estimate
            MAX_ERROR_PERC, MAX_SPIN_MAGNITUDE, MAX_NUM_SPIN_CANDIDATES = 0.7, 1000, 1000
            magnitude = self.best_spin_estimate.magnitude()
            mag_lower_bound = max(0, magnitude * (1 - MAX_ERROR_PERC))
            mag_upper_bound = min(magnitude * (1 + MAX_ERROR_PERC), MAX_SPIN_MAGNITUDE)
            magnitudes = np.linspace(mag_lower_bound, mag_upper_bound, MAX_NUM_SPIN_CANDIDATES)
            spin_candidates = magnitudes * self.best_spin_estimate.axis()

            # convert back to events representation for refinement with contrast maximization
            events: list[Event] = EventCamera.time_surface_to_events(self.time_surfaces[self.best_spin_index])

            # evaluate all spin candidates in parallel on the GPU for efficiency
            # the refined spin estimate is the one that maximizes the variance of the image of warped events
            self.refined_spin_estimate_cmax = self.spin_refiner_cmax.refine_spin_estimate(events, spin_candidates)


def main():
    BallTrackingAndSpinEstimation().init(num_systems=3)  # run the gaze control system(s)
```

Listing 6: Gaze Control System pseudocode