Conformal Prediction for STL Runtime Verification

Lars Lindemann*
llindema@usc.edu
University of Southern California
Los Angeles, California, USA

Xin Qin*
xinqin@usc.edu
University of Southern California
Los Angeles, California, USA

Jyotirmoy V. Deshmukh
jyotirmoy.deshmukh@usc.edu
University of Southern California
Los Angeles, California, USA

George J. Pappas
pappasg@seas.upenn.edu
University of Pennsylvania
Philadelphia, Pennsylvania, USA

Presented By :

Praneat Data (210740)
Bhavaj Singla (210265)
Srushti Srivastava (211060)
Sajal Jain (210903)
Danish Mehmood (210297)
Jayant Soni (210468)

ABSTRACT

Problem:

- The paper addresses how to predict failures in cyber-physical systems during operation
- Specifically focuses on calculating the probability that a system trajectory will violate specifications
- Uses Signal Temporal Logic (STL) to express system specifications

Solution:

This paper presents two novel predictive runtime verification algorithms to tackle the above issues.

OVERVIEW

- Problem Formulation
- STL
- Trajectory Predictors
- Predictive Runtime Verification
- Conformal Predictions
- Direct STL Method
- Indirect STL Method
- Implementation
- Conclusion

PROBLEM FORMULATION

Distribution D

- D represents unknown distribution over system trajectories
- $X = (X_0, X_1, \ldots) \sim D$ is a random trajectory
- X_t represents system state at time t
- System follows Markov decision process: $X_{t+1} = f(X_t, w_t)$
    - w_t is random variable
    - f describes system dynamics

Key Assumptions & Data

- Access to K independent realizations
- Training dataset $D = \{x^{(1)},\ldots,x^{(K)}\}$
- System specifications φ provided

STL

What is STL? :

- STL is a formal language for defining time-based specifications on system signals.
- Used to encode conditions that a dynamic system's behavior must satisfy over time, especially in real-time systems like autonomous vehicles or drones.

Core Components of STL :

- Predicates: Basic building blocks, representing conditions that can be true or false.
- Defined by $h : \mathbb{R}^n \to \mathbb{R}$, with predicate $\mu(x_\tau)$ being True if $h(x_\tau) \geq 0$ and False otherwise.
- STL Syntax:
  - Formulas (ϕ) are constructed recursively using:

$$\phi ::= \text{True} \mid \mu \mid \neg\phi' \mid \phi' \wedge \phi'' \mid \phi' U_I \phi''$$

    - **Negation** ($\neg$): Represents "not."
    - **Conjunction** ($\wedge$): Represents "and."
    - **Until** (U_I): Ensures that ϕ' holds until ϕ'' becomes true within time interval I

STL

STL Operators and Semantics:

1. Derived STL Operators:

- **Disjunction** ($\phi' \vee \phi'' := \neg(\neg\phi' \wedge \neg\phi'')$): Represents "or."
- **Eventually** ($F_I\phi := \top U_I \phi$): ϕ will be true at some point within interval I.
- **Always** ($G_I\phi := \neg F_I \neg\phi$): ϕ holds throughout interval I.

2. Satisfaction and robust semantics:

- To check if a signal x satisfies an STL formula ϕ at time τ_0: $(x, \tau_0) \models \phi$.
- **Robust Semantics** $\rho_\phi(x, \tau_0)$: Quantifies how strongly ϕ is satisfied or violated.
  - $\rho_\phi(x, \tau_0) > 0$ implies $(x, \tau_0) \models \phi$.
  - Positive values mean stronger satisfaction, while negative values indicate a violation

TRAJECTORY PREDICTORS

Goal:

Given an observed partial sequence $(x_0, \ldots, x_t)$ at the current time $t \geq 0$, we want to predict the states $(x_{t+1}, \ldots, x_{t+H})$ for a prediction horizon of $H > 0$.

Assumptions:

We assume that the $\mathrm{PREDICT}$ function, say $\mathcal{Y}$, is a measurable function that maps observations $(x_0, \ldots, x_t)$ to predictions $(\hat{x}_{t+1|t}, \ldots, \hat{x}_{t+H|t})$ of $(x_{t+1}, \ldots, x_{t+H})$.

PREDICT functions are typically learned, In this paper we are using Recurrent Neural Networks (RNNs) as the PREDICT function.

TRAJECTORY PREDICTORS

For $\tau \leq t$, the recurrent structure of an RNN is given as

$$a_\tau^1 := \mathcal{A}(x_\tau, a_{\tau-1}^1),$$

$$a_\tau^i := \mathcal{A}(x_\tau, a_{\tau-1}^i, a_\tau^{i-1}), \quad \forall i \in \{2, \ldots, d\}$$

$$y_{\tau+1|\tau} := \mathcal{Y}(a_\tau^d),$$

where x_τ is the input that is sequentially applied to the RNN and where $\mathcal{A}$ is a function that can parameterize different types of RNNs,

Here we will be using Long Short-Term Memory (LSTM) Network.

Furthermore, d is the RNN's depth and $a_1^\tau, \ldots, a_d^\tau$ are the hidden states. The output $y_{t+1|t} := (\hat{x}_{t+1|t}, \ldots, \hat{x}_{t+H|t})$ provides an estimate of $(x_{t+1}, \ldots, x_{t+H})$ via the function Y, which typically parameterizes a linear last layer.

PREDICTIVE RUNTIME VERIFICATION

- **Predictive runtime verification (PRV) is an approach to system monitoring.**
- **Observes current system state (prefix)**
- **Predicts future states (suffix)**
- **Estimates probability of future specification violations**

$(x_0, x_1, \ldots)$ denotes a realization of $X := (X_0, X_1, \ldots) \sim D$.

$\mathbf{x}_{\text{obs}} := (x_0, \ldots, x_t)$ at time t, i.e., all states up until time t are known.

$x_{\text{un}} := (x_{t+1}, x_{t+2}, \ldots)$ are not known yet.

$$X := (X_{\text{obs}}, X_{\text{un}})$$

PREDICTIVE RUNTIME VERIFICATION

Problem 1. *Given a distribution* $(X_0, X_1, \ldots) \sim \mathcal{D}$, *the current time* t, *the observations* $x_{obs} := (x_0, \ldots, x_t)$, *a bounded STL formula* ϕ *that is enabled at* τ_0, *and a failure probability* $\delta \in (0, 1)$, *determine if* $P\big((X, \tau_0) \models \phi\big) \geq 1 - \delta$ *holds.*

$$P\big((X, \tau_0) \models \neg\phi\big) \leq \delta,$$

We obtain a probabilistic lower bound $\bar{C} \in \mathbb{R}$ on the robust semantics $\rho^{\phi}(X, \tau_0)$, i.e.,

$$P(\rho^{\phi}(X, \tau_0) \geq \bar{C}) \geq 1 - \delta.$$

CONFORMAL PREDICTION – INTRODUCTION

Definition

Conformal prediction is a statistical method used to determine how confidently a model can predict future states without needing strict assumptions on data distribution. It provides a confidence level for our prediction region

Prediction Region

Prediction Region defines the area where the actual outcome is likely to fall within a specified probability (e.g., 95%).

95% is chosen as it is a good balance between speed and accuracy.

CONFORMAL PREDICTION – QUANTILE LEMMA

Non Confomity Scores

Let $R^{(0)}, \ldots, R^{(k)}$ be $k+1$ independent and identically distributed random variables. The variable $R^{(i)}$ is usually referred to as the nonconformity score. In supervised learning, it may be defined as $R^{(i)} := \|Y^{(i)} - \mu(X^{(i)})\|$ where the predictor μ attempts to predict an output $Y^{(i)}$ based on an input $X^{(i)}$. A large nonconformity score indicates a poor predictive model.

Suppose we have Non comformality scores R1 to Rk and we need to calculate R0 such that

$$P(R^{(0)} \leq C) \geq 1 - \delta.$$

By a surprisingly simple quantile argument, see [74, Lemma 1], one can obtain C to be the $(1-\delta)$th quantile of the empirical distribution of the values $R^{(1)}, \ldots, R^{(k)}$ and ∞. By assuming that $R^{(1)}, \ldots, R^{(k)}$ are sorted in non-decreasing order, and by adding $R^{(k+1)} := \infty$, we can equivalently obtain $C := R^{(p)}$ where $p := \lceil (k+1)(1-\delta) \rceil$, i.e., C is the pth smallest nonconformity score.

CONFORMAL PREDICTION – APPLICATION

STL Compliances

STL is used to express rules that CPS must follow over time (e.g., maintaining speed within a limit). Conformal prediction assesses the likelihood of these requirements being met.

Direct Method

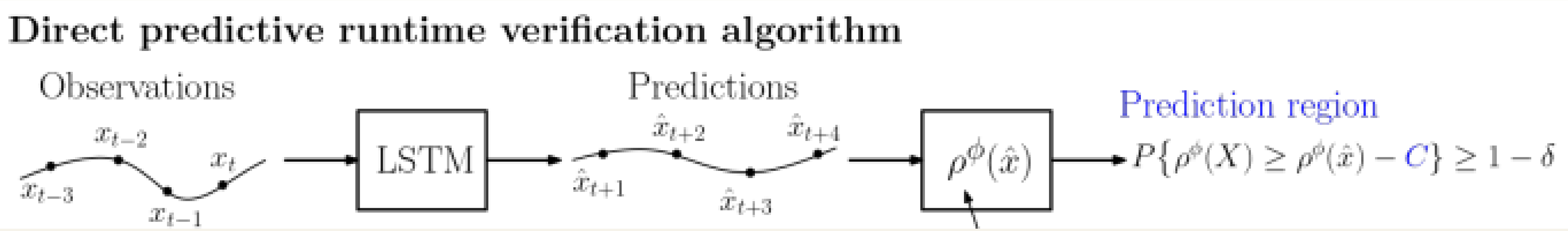
Applies prediction directly to the STL requirement, creating a confidence interval around the satisfaction measure.

Indirect Method

Predicts the system's future states first, then uses these predictions to indirectly verify STL compliance.

DIRECT STL PRV

- In Direct STL PRV, we aim to use predictive modeling to ensure that STL specifications are met over future horizons in real-time systems
- We can obtain the predictions for all future times from the PREDICT function that we obtain by training RNN models on the dataset



CONFORMAL PREDICTION EQUATION

We define H as the maximum prediction horizon that is needed to estimate the satisfaction of bounded STL specification

$$H := \tau_0 + L^{\phi} - t$$

From the equation of conformal prediction we know

$$P\big(\rho^{\phi}(\hat{x}, \tau_0) - \rho^{\phi}(X, \tau_0) \leq C\big) \geq 1 - \delta.$$

For the real trajectory to satisfy the STL constrainsts, we have

$$P(\rho^{\phi}(X, \tau_0) > 0) \geq 1 - \delta$$

From the above two eqautions we get

$$\rho^{\phi}(\hat{x}, \tau_0) > C$$

DERIEVING CONSTANT "C"

- To obtain the constant C we consider the non conformilty score R for each of the datapoint in the Calibration Dataset
- We sort the scores in non decreasing order and also add (Dcal +1) non conformity score ∞
- Using the Quantile Lemma from before we can get our desired value of C

$$R^{(i)} := \rho^{\phi}(\hat{x}^{(i)}, \tau_0) - \rho^{\phi}(x^{(i)}, \tau_0)$$

$$C := R^{(p)} \quad where \;\; p := \lceil (|D_{cal}| + 1)(1 - \delta) \rceil$$

$$P((X, \tau_0) \models \phi) \geq 1 - \delta \; if \, \rho^{\phi}(\hat{x}, \tau_0) > C$$

STEPS

- **We have a partial observation of a trajectory till time t**
- **We use our trained trajectory predictors to predict the trajectory till the horizon**
- **We calculate the non-conformity scores on the calibration dataset till the horizon**

$$R^{(i)} := \rho^{\phi}(\hat{x}^{(i)}, \tau_0) - \rho^{\phi}(x^{(i)}, \tau_0)$$

- **Using quantile lemma we sort these scores and find the (1- delta) quantile score represented as C**
- **If the predicted trajectory robustness value is greater than C that means the system is safe and we can move forward**

$$C := R^{(p)} \quad where \;\; p := \lceil (|D_{cal}| + 1)(1 - \delta) \rceil$$

$$P((X, \tau_0) \models \phi) \geq 1 - \delta \; if \, \rho^{\phi}(\hat{x}, \tau_0) > C$$

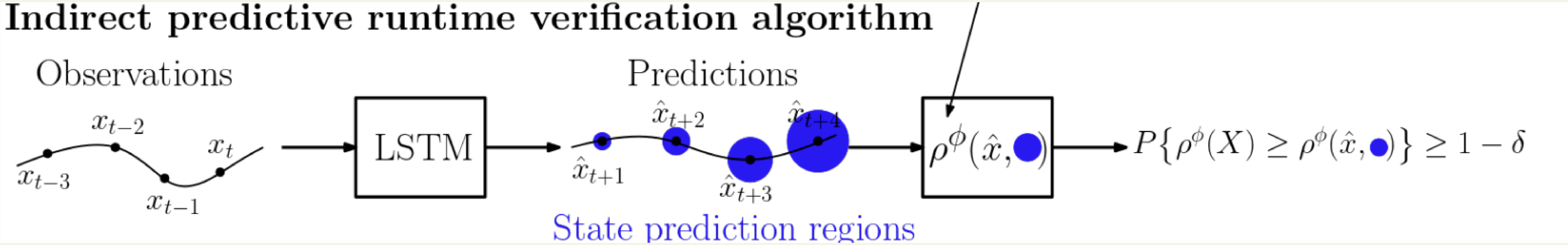
INDIRECT STL PRV

- **The indirect approach first generates prediction regions for future states and then uses these regions to verify the STL compliance.**
- **Prediction regions are derived from predicted states.**

For a failure probability of $\delta \in (0, 1)$, our first goal is to construct prediction regions defined by constants C_τ so that

$$P\big(\|X_\tau - \hat{x}_{\tau|t}\| \leq C_\tau,\ \forall \tau \in \{t+1, \ldots, t+H\}\big) \geq 1 - \delta, \qquad (5)$$

$C\tau$ **should be such that the state** $X\tau$ **is** $C\tau$ **-close to our predictions** $x\tau\ |t$ **for all relevant times** $\tau \in \{t + 1, \ldots, t + H\}$ **with a probability of at least 1 -** Δ



STEPS TO GENERATE PREDICTION REGIONS

1. Collect Historical Data:
   - Use a dataset of past observations where both predicted states $x_{\tau \mid t}$ and their corresponding true states X_τ are known.
   - For example, for time $t_1, t_2, \ldots, t_N$, you have: $\{(x_{t1}, X_{t1}), (x_{t2}, X_{t2}), \ldots, (x_{tN}, X_{tN})\}$
2. Calculate Past Prediction Errors:
   - Compute the prediction error (or deviation) for each past prediction: $e_i = ||X_{ti} - x_{ti}||$
   - This gives a distribution of errors $e_1, e_2, \ldots, e_N$.

STEPS TO GENERATE PREDICTION REGIONS

3. Determine the Quantile for Cτ:

- For a desired confidence level 1–Δ, calculate the (1–Δ)-quantile of the past errors.

$$C\tau = \text{Quantile}_{1\text{-}\delta}(e1, e2, \ldots, eN)$$

- This value Cτ is the threshold that ensures future prediction errors will be within this limit with probability 1–Δ.

4. Apply Cτ to New Predictions:

- Use the calculated Cτ for new predictions xτ ∣ t to construct prediction regions for future time steps:

$$P(||X\tau - x\tau \mid t|| \leq C\tau) \geq 1-\Delta$$

You are predicting the **altitude** of an aircraft.

- **Historical Data**: Suppose you have past data:

$$(\hat{x}_{t_1} = 10,300, X_{t_1} = 10,500), (\hat{x}_{t_2} = 10,400, X_{t_2} = 10,600), \ldots$$

  From this, you calculate past prediction errors:

$$e_1 = |10,500 - 10,300| = 200, \quad e_2 = |10,600 - 10,400| = 200, \ldots$$

- **Quantile Calculation**: For a **95% confidence level** ($\delta = 0.05$), calculate the **95th percentile** of the error distribution $\{e_1, e_2, \ldots\}$. Suppose this value is:

$$C_\tau = 250$$

- **Future Prediction**: If your model predicts $\hat{x}_{t+1|t} = 10,450$, the prediction region becomes:

$$[10,450 - 250, 10,450 + 250] = [10,200, 10,700]$$

  You are **95% confident** that the true future altitude X_{t+1} will lie within this range.

NEXT STEPS

1. **Define a Prediction Region:**
   - For each future time step τ, the true state Xτ is uncertain.
   - A prediction region Bτ is constructed around the predicted state xτ ∣ t, ensuring:

P(Xτ∈Bτ)≥1-Δ

$$\mathcal{B}_\tau = \{\zeta \in \mathbb{R}^n \mid \|\zeta - \hat{x}_{\tau|t}\| \leq C_\tau\}$$

2. **Worst-Case Evaluation of Robust Semantics (ρ̄ϕ):**
   - Robust semantics ρϕ measures how well the system satisfies the STL formula ϕ.
   - Since Xτ is unknown, the worst-case robust semantics ρ̄ϕ is computed by evaluating ρϕ over the entire prediction region Bτ

$$\bar{\rho}^\mu(\hat{x}, \tau) = \begin{cases} h(x_\tau) & \text{if } \tau \leq t \text{ (known states)} \\ \inf_{\zeta \in \mathcal{B}_\tau} h(\zeta) & \text{if } \tau > t \text{ (future states)} \end{cases}$$

NEXT STEPS

Component	Explanation
$\bar{\rho}^{\mu}(\hat{x}, \tau)$	Robust satisfaction measure for the predicate μ at time τ. It considers either the exact state (for known times) or worst-case scenarios (for future times).
$h(x_\tau)$	Satisfaction measure for the known state x_τ. For example, it could measure how far a car's speed is from exceeding a limit.

- **Predicate μ defines constraints on the system state at a given moment.**
- **STL Formula ϕ extends this by adding temporal logic to specify how those constraints should be satisfied over time.**

- **Predicate**: "The car's speed must remain below 80 km/h."

$$\mu(x) = (x < 80), \quad h(x) = 80 - x$$

1. **For Known Time $\tau = t$:**
   - Actual speed at time t: $x_t = 75$ km/h.
   - Satisfaction measure:
   $$\bar{\rho}^{\mu}(\hat{x}, t) = h(x_t) = 80 - 75 = 5$$
   - The system satisfies the requirement by a margin of 5 km/h.
2. **For Future Time $\tau = t + 1$:**
   - Predicted speed: $\hat{x}_{t+1|t} = 78$ km/h.
   - Prediction region: $\mathcal{B}_{t+1} = [77, 79]$ km/h (based on uncertainty bounds).
   - Worst-case satisfaction:
   $$\bar{\rho}^{\mu}(\hat{x}, t+1) = \inf_{\zeta \in \mathcal{B}_{t+1}} (80 - \zeta)$$
   Evaluate for the worst-case speed:
   $$\text{For } \zeta = 79 : \quad 80 - 79 = 1$$
   - Worst-case satisfaction margin is **1 km/h**. The system still satisfies the requirement but by a smaller margin.

IMPLEMENTATION

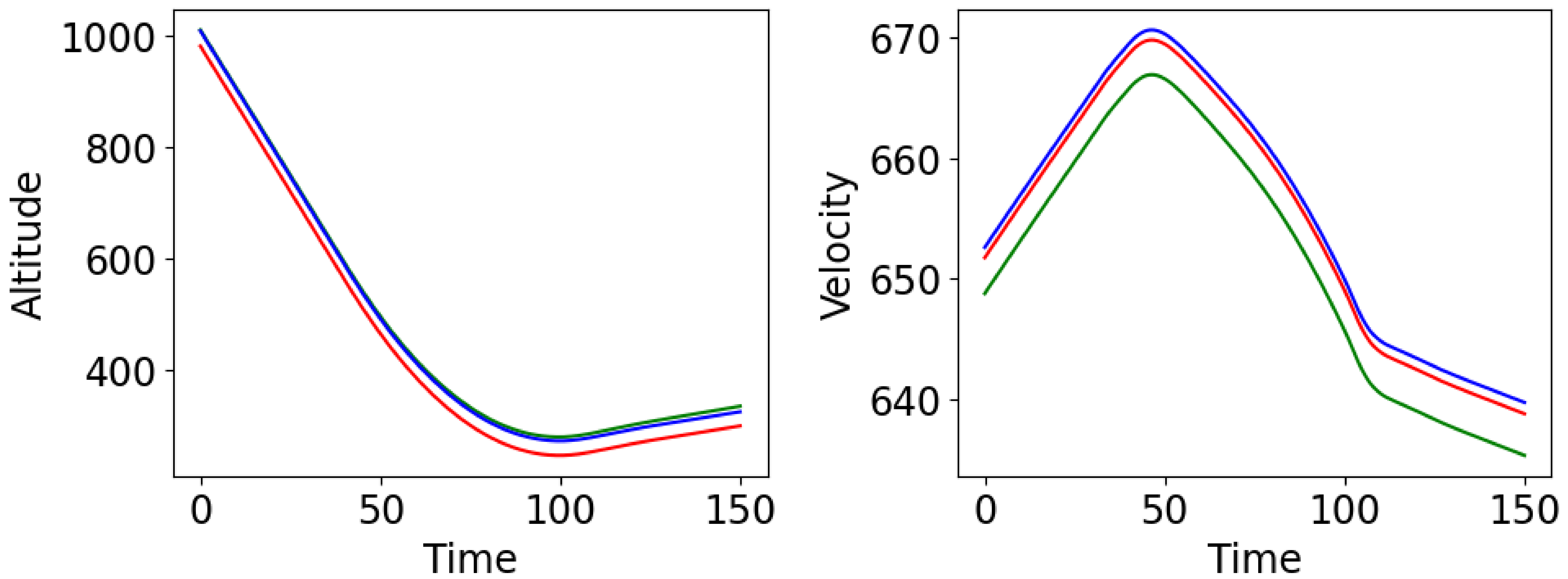
```python
# Set a seed.
selected_seed = 100
random.seed(selected_seed)

# Codes for parameters of plotting.
mpl.rcParams.update(mpl.rcParamsDefault)
font = {'size' : 17}
mpl.rc('font', **font)


# Define hyperparameters for the simulation.
power = 9  # engine power level (0-10)
alpha = deg2rad(2.1215)  # Trim Angle of Attack (rad)
beta = 0  # Side slip angle (rad)
phi = -math.pi / 8  # Roll angle from wings level (rad)
theta = (-math.pi / 2) * 0.3  # Pitch angle from nose level (rad)
psi = 0  # Yaw angle from North (rad)
tmax = 5  # simulation time
simulation_step = 1 / 30

# distributional information.
nominal_alt_mean = 1000
nominal_alt_std = 10
nominal_vel_mean = 650
nominal_vel_std = 5


# Write a class for simulation purpose.
class Plane:
    def __init__(self, power, alpha, beta, phi, theta, psi, tmax, step, nominal_alt_mean, nominal_alt_std,
    nominal_vel_mean, nominal_vel_std):
        # This function initializes the system.
        self.power = power # engine power level (0 - 10)
        self.alpha = alpha # Trim Angle of Attack (rad)
        self.beta = beta # Side slip angle (rad)
        self.phi = phi # Roll angle from wings level (rad)
        self.theta = theta # Pitch angle from nose level (rad)
        self.psi = psi # Yaw angle from North (rad)
        self.tmax = tmax
        self.step = step

        # Distributional info.
        self.nominal_alt_mean = nominal_alt_mean
        self.nominal_alt_std = nominal_alt_std
        self.nominal_vel_mean = nominal_vel_mean
        self.nominal_vel_std = nominal_vel_std

    def generate_nominal_trajectory(self, tmax, step):
        # First, randomly select an initial altitude based on the distribution info.
        alt = random.normal(self.nominal_alt_mean, self.nominal_alt_std)
        vel = random.normal(self.nominal_vel_mean, self.nominal_vel_std)
        init_state = [vel, self.alpha, self.beta, self.phi, self.theta, self.psi, 0, 0, 0, 0, 0, alt, self.power]
        ap = GcasAutopilot(init_mode='roll', stdout=True, gain_str='old')
        res = run_f16_sim(init_state, tmax, ap, step=step, extended_states=True)
        altitude =  list(plot.return_single(res, 'alt')[1])
        velocity = list(plot.return_single(res, 'vt')[1])
        return altitude, velocity
```

Let's consider the setup for simulating F-16 Fighter jet. You can find the codes in https://github.com/stanleybak/AeroBenchVVPython.git, which we use in this example.

To start with, let's construct a simulatable plane system. For our verification purpose, let's assume that the system comes from a distribution with the initial altitude of N(1000, 10^2) and initial velocity of N(650, 5^2), with all other hyperparameters remain the same. We observe the states of altitudes and velocities (air speeds).

<u>Code for the implementation can be found here</u>

IMPLEMENTATION



Example Trajectories from Nominal Distribution

IMPLEMENTATION

```python
current_time = 80
training_size = 100
validation_size = 50
n_epochs_alt = 1000
n_epochs_vel = 5000


class Predictor(nn.Module):
    # Write a class for the predictor.
    def __init__(self, set_inputsize, set_outputsize):
        super().__init__()
        self.lstm = nn.LSTM(input_size=set_inputsize, hidden_size=50, num_layers=1, batch_first=True)
        self.linear = nn.Linear(50, set_outputsize)

    def forward(self, x):
        x, _ = self.lstm(x)
        x = self.linear(x)
        return x

def train_predictor(train_x, train_y, validation_x, validation_y, n_epochs):
    predictor = Predictor(len(train_x[0]), len(train_y[0]))
    optimizer = optim.Adam(predictor.parameters())
    loss_fn = nn.MSELoss()
    loader = data.DataLoader(data.TensorDataset(train_x, train_y), shuffle=True, batch_size=8)
    for epoch in range(n_epochs):
        predictor.train()
        for X_batch, y_batch in loader:
            y_pred = predictor(X_batch)
            loss = loss_fn(y_pred, y_batch)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        # Validation
        if epoch % 100 != 0:
            continue
        predictor.eval()
        with torch.no_grad():
            y_pred = predictor(train_x)
            train_rmse = np.sqrt(loss_fn(y_pred, train_y))
            y_pred = predictor(validation_x)
            validation_rmse = np.sqrt(loss_fn(y_pred, validation_y))
        print("Epoch %d: train RMSE %.4f, validation RMSE %.4f" % (epoch, train_rmse, validation_rmse))
    return predictor


def generate_predictions(altitude_prefixes, velocity_prefixes, alt_predictor, vel_predictor, normalization_constant):
    # First, normalize the prefixes.
    altitude_prefixes_normalized = array(altitude_prefixes) / normalization_constant
    velocity_prefixes_normalized = array(velocity_prefixes) / normalization_constant
    # Generate predictions.
    with torch.no_grad():
        alt_y_pred = array(alt_predictor(torch.FloatTensor(np.array(altitude_prefixes_normalized)))) * normalization
    with torch.no_grad():
        vel_y_pred = array(vel_predictor(torch.FloatTensor(np.array(velocity_prefixes_normalized)))) * normalization
    # Piece together the predicted trajectories and the predictions.
    for i in range(len(altitude_prefixes)):
        altitude_prefixes[i].extend(list(alt_y_pred[i]))
        velocity_prefixes[i].extend(list(vel_y_pred[i]))
    return altitude_prefixes, velocity_prefixes

normalization = 1000 # For training purpose, let's use a normalization constant.
```

Neural Network Structure

- LSTM layer (hidden size: 50, single layer)
- Linear layer for output prediction
- Handles both altitude and velocity predictions

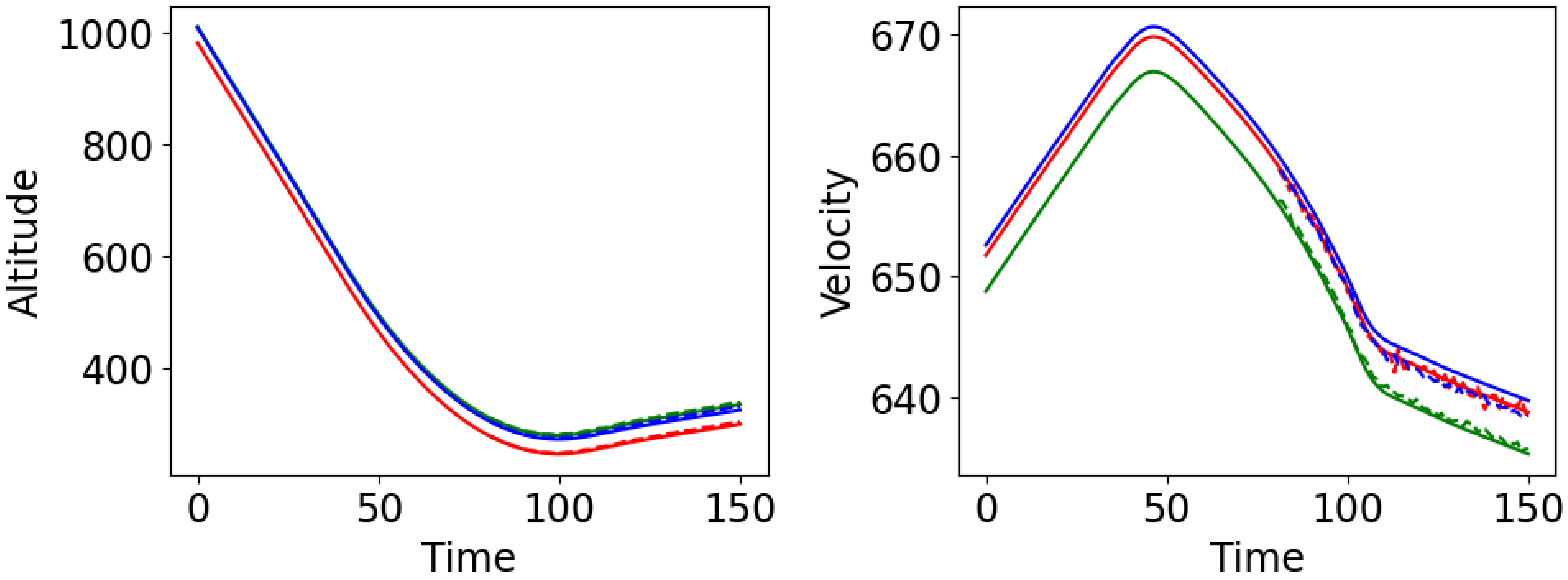
Key Parameters

- Training examples: 100
- Validation examples: 50
- Training epochs:
  - Altitude: 1,000 epochs
  - Velocity: 5,000 epochs
- Batch size: 8
- Normalization factor: 1,000

Training Implementation

- Uses Adam optimizer
- Mean Squared Error (MSE) loss function
- Validation check every 100 epochs
- Tracks both training and validation RMSE

IMPLEMENTATION



Example Predicted Trajectories from Nominal Distribution

IMPLEMENTATION

```
# Generate an additional dataset.
alpha_computation_size = 100
alpha_alts = []
alpha_vels = []
for _ in range(alpha_computation_size):
    altitude, velocity = plane.generate_nominal_trajectory(tmax, simulation_step)
    alpha_alts.append(altitude)
    alpha_vels.append(velocity)
# Next, we generate predictions.
alpha_alts_prefixes = [alpha_alts[j][:current_time + 1] for j in range(len(alpha_alts))]
alpha_vels_prefixes = [alpha_vels[j][:current_time + 1] for j in range(len(alpha_vels))]
alpha_pred_alts, alpha_pred_vels = generate_predictions(alpha_alts_prefixes, alpha_vels_prefixes, alt_predictor,
vel_predictor, normalization)
# Now, compute the alphas.
alphas = dict()
for tau in range(current_time + 1, terminal_time + 1):
    error_list = []
    for i in range(alpha_computation_size):
        ground_alt = alpha_alts[i][tau]
        pred_alt = alpha_pred_alts[i][tau]
        ground_vel = alpha_vels[i][tau]
        pred_vel = alpha_pred_vels[i][tau]
        error = ((ground_alt - pred_alt) ** 2 + (ground_vel - pred_vel) ** 2) ** (1 / 2)
        error_list.append(error)
    alphas[tau] = 1 / max(error_list)
```

We are generating some extra dataset from the same normal distribution as earlier.

IMPLEMENTATION

In the following example, the specification is to ensure safety where we monitor if there is no low-altitude flying and if the plane is below an altitude threshold, the speed should be slow enough to ensure safety.

Globally_[0, T](p >= zeta_1 and (p < zeta_2 implies v <= zeta_3))

$$\varphi = \Box[0,T]\ (\mu_1 \wedge (\mu_2 \rightarrow \mu_3))$$

equivalent to

$$\varphi = \Box[0,T]\ (\mu_1 \wedge (\neg\mu_2 \vee \mu_3))$$

IMPLEMENTATION

```
experimental_size = 400
calibration_size = 700
test_size = 200
zeta_1 = 100
zeta_2 = 300
zeta_3 = 650
delta = 0.05


# Let's first write a function in computing the robust semantics given a trajectory.
def compute_robust_semantics(alt, vel):
    final_robustness = float("inf")
    for tau in range(len(alt)):
        time_robustness = min(alt[tau] - zeta_1, max(alt[tau] - zeta_2, zeta_3 - vel[tau]))
        final_robustness = min(final_robustness, time_robustness)
    return final_robustness

# We also need a function in computing the worst-case semantics given a trajectory and the prediction regions.
def compute_worst_robust_semantics(alt, vel, prediction_regions):
    collision_robustnesses = []
    height_robustnesses = []
    speed_robustnesses = []
    for tau in range(0, terminal_time + 1):
        if tau <= current_time:
            collision_robustness = alt[tau] - zeta_1
            height_robustness = alt[tau] - zeta_2
            speed_robustness = zeta_3 - vel[tau]
        else:
            # Since the predicate is affine, the minimum predicate robustness happens only when the dimension of
            interest changes to the maximal amount while the other dimensions remain the same.
            collision_robustness = (alt[tau] - prediction_regions[tau]) - zeta_1
            height_robustness = (alt[tau] - prediction_regions[tau]) - zeta_2
            speed_robustness = zeta_3 - (vel[tau] + prediction_regions[tau])
        collision_robustnesses.append(collision_robustness)
        height_robustnesses.append(height_robustness)
        speed_robustnesses.append(speed_robustness)
    # Compose the robustnesses together.
    final_robustness = float("inf")
    for tau in range(len(alt)):
        time_robustness = min(collision_robustnesses[tau], max(height_robustnesses[tau], speed_robustnesses[tau]))
        final_robustness = min(final_robustness, time_robustness)
    return final_robustness
```

We now formulate the robustness semantics and the worst case scenario for the STL, for the direct and the indirect method.

$$\rho(\mu_1, t) = p(t) - 100$$

$$\rho(\mu_2, t) = 300 - p(t)$$

$$\rho(\mu_3, t) = 650 - v(t)$$

$$\rho(\varphi, t) = \min\{ \min(\rho(\mu_1, t)),\ \max(-\rho(\mu_2, t), \rho(\mu_3, t)) \}$$

IMPLEMENTATION

```python
for i in range(experimental_size):
    print("Conducting Experiment ", i + 1)
    # First, collect the calibration data.
    calib_alts = []
    calib_vels = []
    for _ in range(calibration_size):
        altitude, velocity = plane.generate_nominal_trajectory(tmax, simulation_step)
        calib_alts.append(altitude)
        calib_vels.append(velocity)
    # Next, we generate predictions on the calibration data.
    calib_alts_prefixes = [calib_alts[j][:current_time + 1] for j in range(len(calib_alts))]
    calib_vels_prefixes = [calib_vels[j][:current_time + 1] for j in range(len(calib_vels))]
    calib_pred_alts, calib_pred_vels = generate_predictions(calib_alts_prefixes, calib_vels_prefixes, alt_predictor,
    vel_predictor, normalization)
    # Then, we generate the test set for each experiment.
    test_alts = []
    test_vels = []
    for _ in range(test_size):
        altitude, velocity = plane.generate_nominal_trajectory(tmax, simulation_step)
        test_alts.append(altitude)
        test_vels.append(velocity)
    # We generate the predictions on the test data (This is done online in validation, but we generate here to avoid
    redundant executions).
    test_alts_prefixes = [test_alts[j][:current_time + 1] for j in range(len(test_alts))]
    test_vels_prefixes = [test_vels[j][:current_time + 1] for j in range(len(test_vels))]
    test_pred_alts, test_pred_vels = generate_predictions(test_alts_prefixes, test_vels_prefixes, alt_predictor,
    vel_predictor, normalization)
```

- Calibration
  - Generate calibration trajectories
  - Compute predictions
  - Calculate nonconformity scores
  - Determine threshold c_direct
- Validation
  - Test coverage on test trajectories
  - Plot histograms and scatter plots
  - Save experimental results

DIRECT METHOD

```
# Let's first experiment with the direct method.
print("Performing Experiment with the Direct Method.")
# Now, let's perform conformal prediction.
calib_robustnesses = [compute_robust_semantics(calib_alts[j], calib_vels[j]) for j in range(calibration_size)]
calib_pred_robustnesses = [compute_robust_semantics(calib_pred_alts[j], calib_pred_vels[j]) for j in range
(calibration_size)]
direct_nonconformity_scores = [calib_pred_robustnesses[j] - calib_robustnesses[j] for j in range(calibration_size)]
direct_nonconformity_scores.sort()
direct_nonconformity_scores.append(float("inf"))
p = int(np.ceil((calibration_size + 1) * (1 - delta)))
c_direct = direct_nonconformity_scores[p - 1]
# Measure EC by generating an additonal trajectory.
ec_altitude, ec_velocity = plane.generate_nominal_trajectory(tmax, simulation_step)
ec_altitude_prefix = ec_altitude[:current_time + 1]
ec_velocity_prefix = ec_velocity[:current_time + 1]
ec_prediction = generate_predictions([ec_altitude_prefix], [ec_velocity_prefix], alt_predictor, vel_predictor,
normalization)
ec_altitude_pred, ec_velocity_pred = ec_prediction[0][0], ec_prediction[1][0]
ec_ground_robustness = compute_robust_semantics(ec_altitude, ec_velocity)
ec_lowerbound_robustness = compute_robust_semantics(ec_altitude_pred, ec_velocity_pred) - c_direct
ec_count_direct += (ec_ground_robustness >= ec_lowerbound_robustness)
# We plot the histogram of nonconformity scores for illustration if this is the first trial.
if i == 0:
    plt.hist(direct_nonconformity_scores[:-1], bins = 20)
    plt.xlabel("Nonconformity Score")
    plt.ylabel("Frequency")
    plt.axvline(c_direct, label = "c", color = "g")
    plt.legend()
    plt.title("Nonconformity Scores from the Direct Method")
    plt.savefig("plots/nonconformity_scores_direct.pdf")
    plt.show()
    # Let's save the experimental results.
    with open("results/direct_nonconformity_scores.json" , "w") as file:
        json.dump(direct_nonconformity_scores, file)
    with open("results/c_direct.json", "w") as file:
        json.dump(c_direct, file)
```

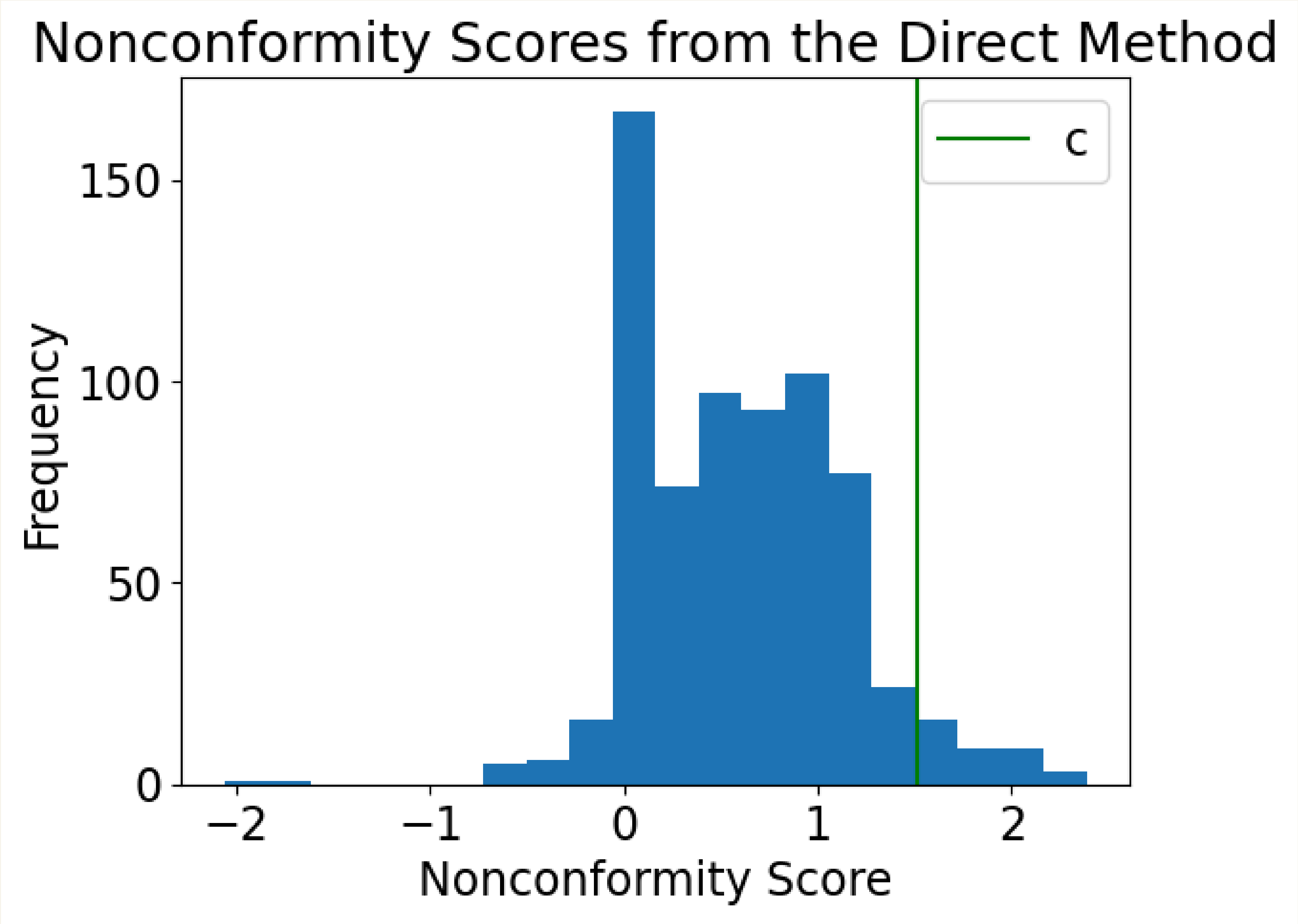
DIRECT METHOD

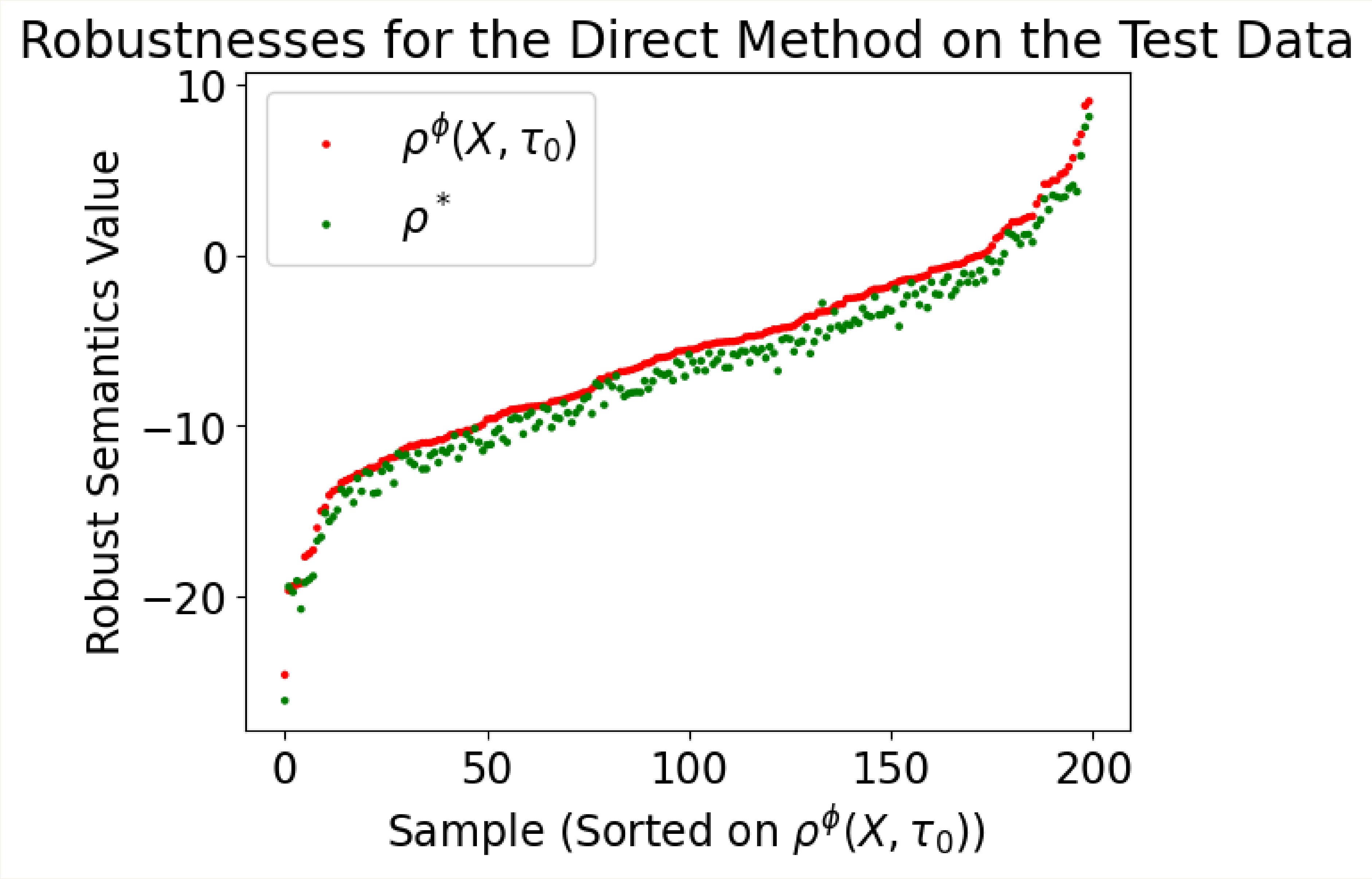
```
# Now, we are ready to validate the results.
direct_correct_count = 0
direct_test_robustnesses = []
direct_test_lowerbound_robustnesses = []
for j in range(test_size):
    direct_test_robustness = compute_robust_semantics(test_alts[j], test_vels[j])
    direct_test_pred_robustness = compute_robust_semantics(test_pred_alts[j], test_pred_vels[j])
    direct_test_lowerbound_robustness = direct_test_pred_robustness - c_direct
    if direct_test_robustness >= direct_test_lowerbound_robustness:
        direct_correct_count += 1
    direct_test_robustnesses.append(direct_test_robustness)
    direct_test_lowerbound_robustnesses.append(direct_test_lowerbound_robustness)
# We plot the scatter plot of the robustnesses for testing.
if i == 0:
    sorted_direct_test_robustnesses, sorted_direct_test_lowerbound_robustnesses = zip(*sorted(zip
    (direct_test_robustnesses, direct_test_lowerbound_robustnesses)))
    dot_sizes = [5 for j in range(test_size)]
    plt.scatter([j for j in range(test_size)], sorted_direct_test_robustnesses, s=dot_sizes, color = "r", label=
    "$\\rho^\phi(X, \\tau_0)$")
    plt.scatter([j for j in range(test_size)], sorted_direct_test_lowerbound_robustnesses , s=dot_sizes, color =
    "g", label= "$\\rho^*$")
    plt.xlabel("Sample (Sorted on $\\rho^\phi(X, \\tau_0)$)")
    plt.ylabel("Robust Semantics Value")
    plt.legend()
    plt.title("Robustnesses for the Direct Method on the Test Data")
    plt.savefig("plots/direct_robustnesses_scatter.pdf")
    plt.show()
    # Save the experimental results.
    with open("results/direct_test_robustnesses.json", "w") as file:
        json.dump(direct_test_robustnesses, file)
    with open("results/direct_test_lowerbound_robustnesses.json", "w") as file:
        json.dump(direct_test_lowerbound_robustnesses, file)
direct_coverage = direct_correct_count / test_size
direct_coverages.append(direct_coverage)
print("The Coverage of the Direct Method is: ", direct_coverage)
```

Now lets validate the results for the direct method.

We define coverage as the correct_count / test_size
This will give us a metric to check the accuracy of the prediction.

DIRECT METHOD



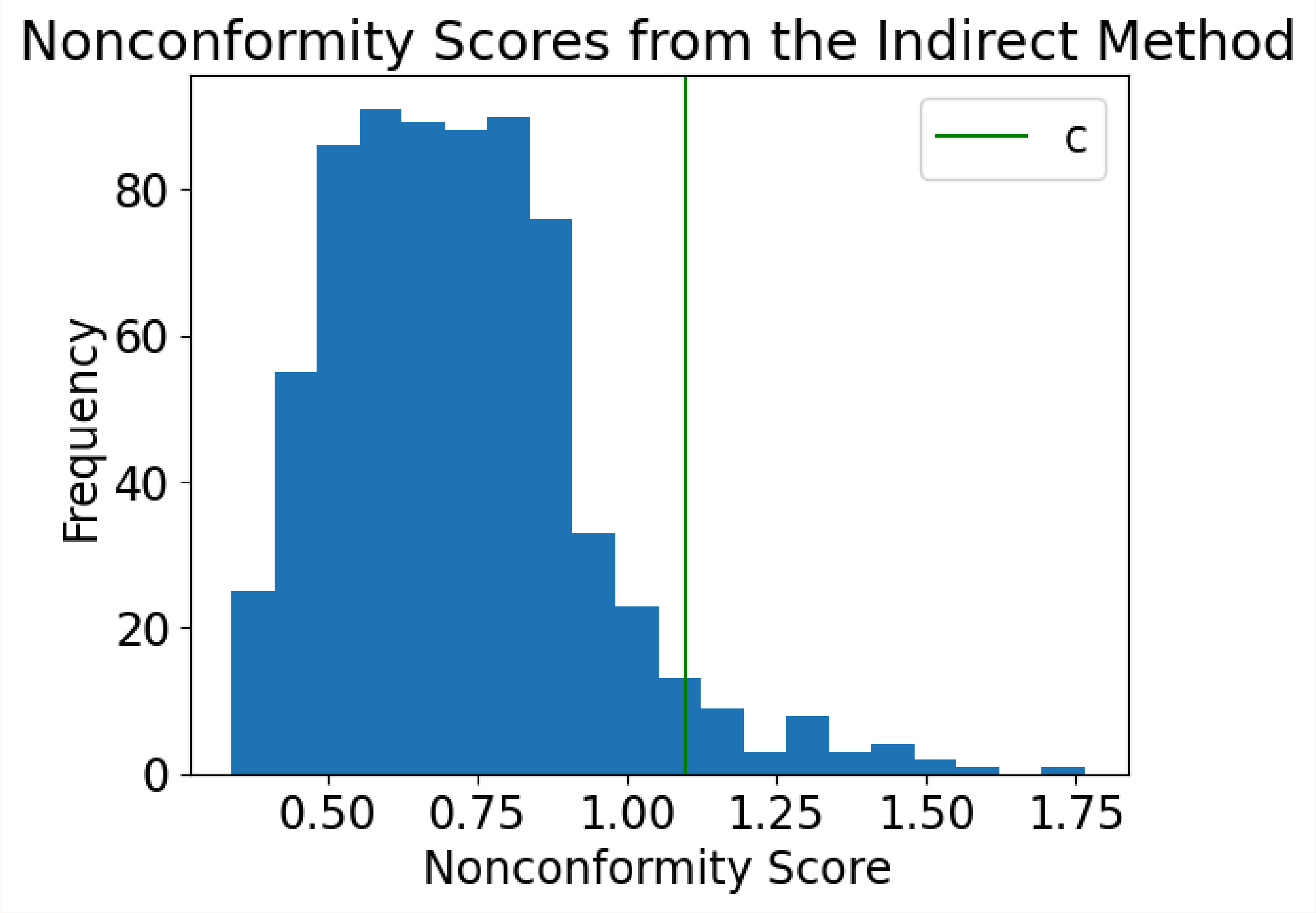


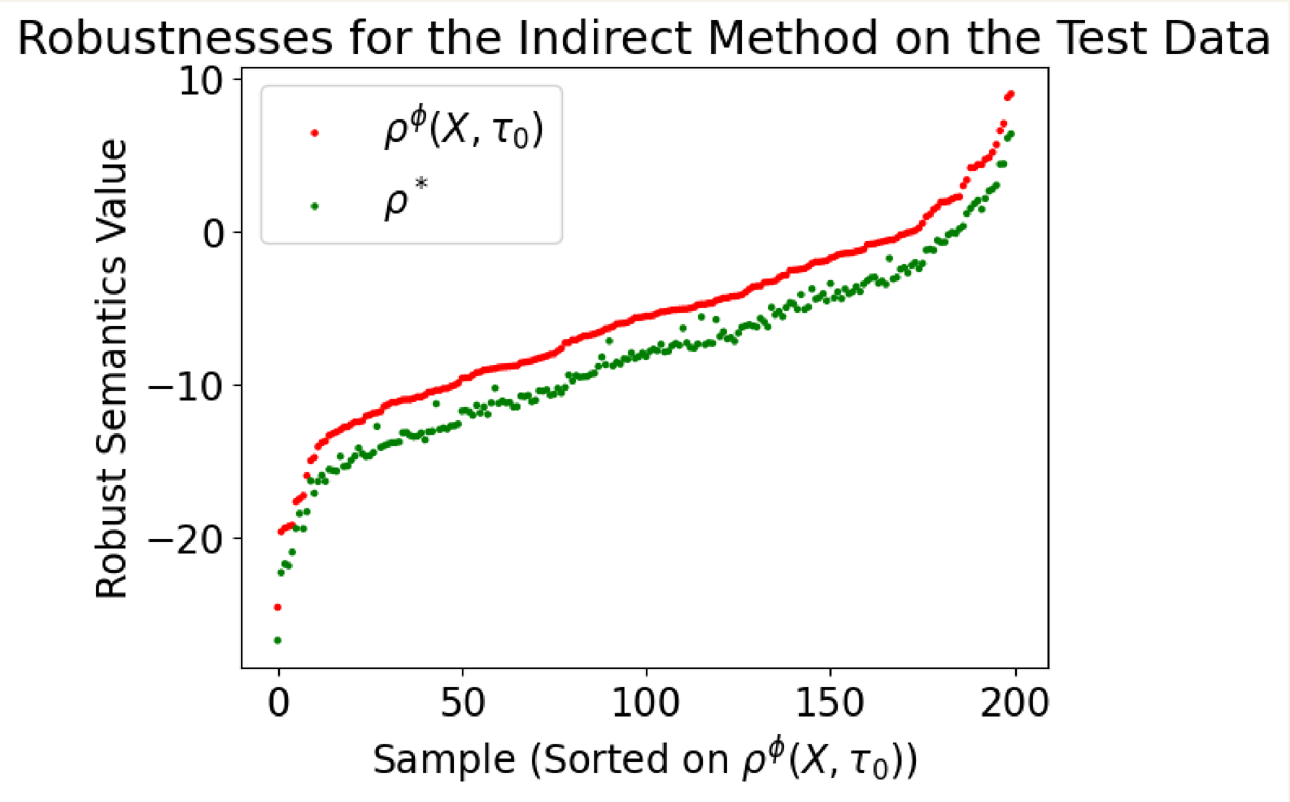
INDIRECT METHOD

```python
print("Performing Experiment with the Indirect Method.")
# Now, let's perform conformal prediction.
indirect_nonconformity_scores = []
for j in range(calibration_size):
    max_score = 0 - float("inf")
    for tau in range(current_time + 1, terminal_time + 1):
        ground_alt = calib_alts[j][tau]
        pred_alt = calib_pred_alts[j][tau]
        ground_vel = calib_vels[j][tau]
        pred_vel = calib_pred_vels[j][tau]
        error = ((ground_alt - pred_alt) ** 2 + (ground_vel - pred_vel) ** 2) ** (1 / 2)
        max_score = max(max_score, alphas[tau] * error)
    indirect_nonconformity_scores.append(max_score)
indirect_nonconformity_scores.sort()
indirect_nonconformity_scores.append(float("inf"))
p = int(np.ceil((calibration_size + 1) * (1 - delta)))
c_indirect = indirect_nonconformity_scores[p - 1]
```

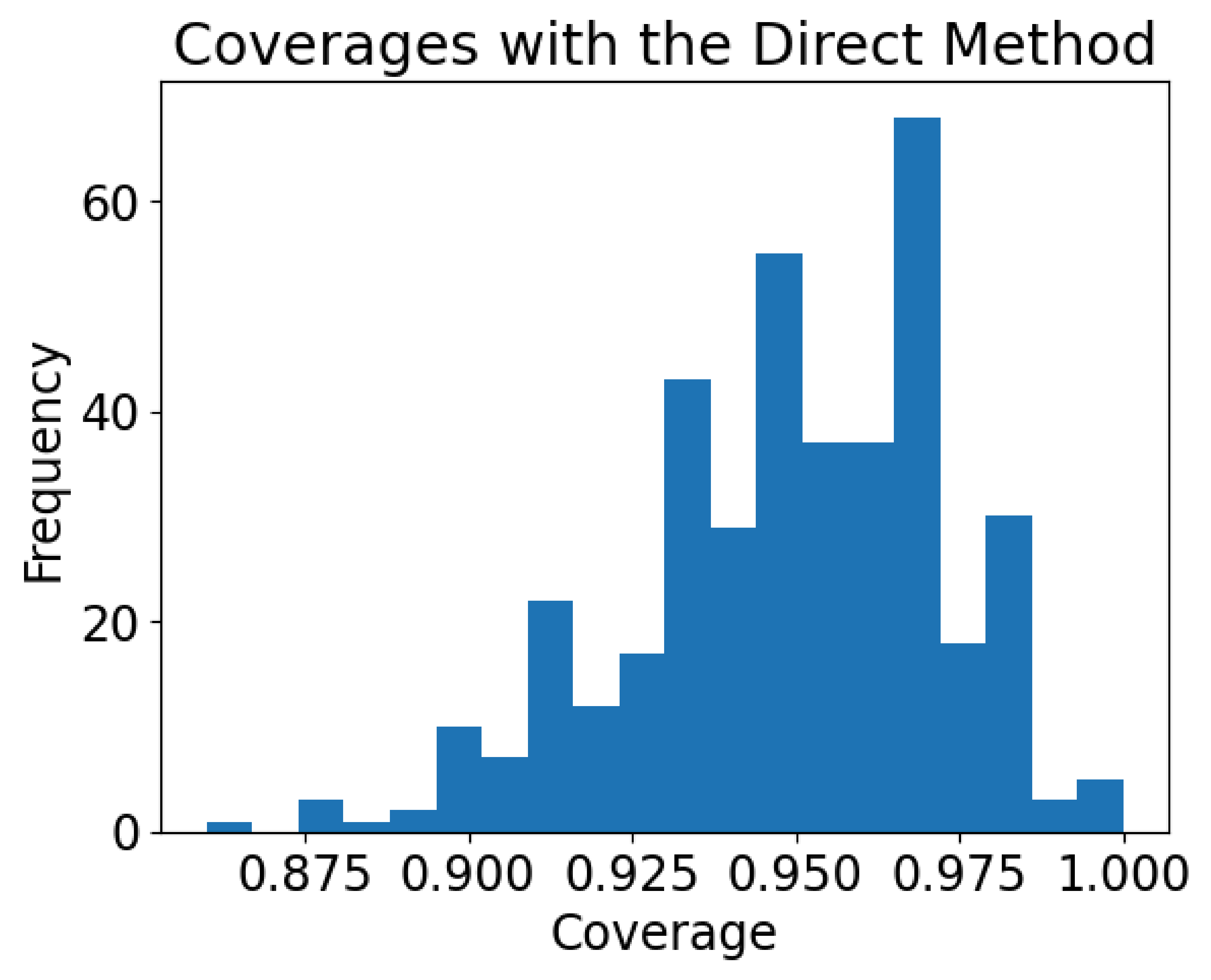
INDIRECT METHOD

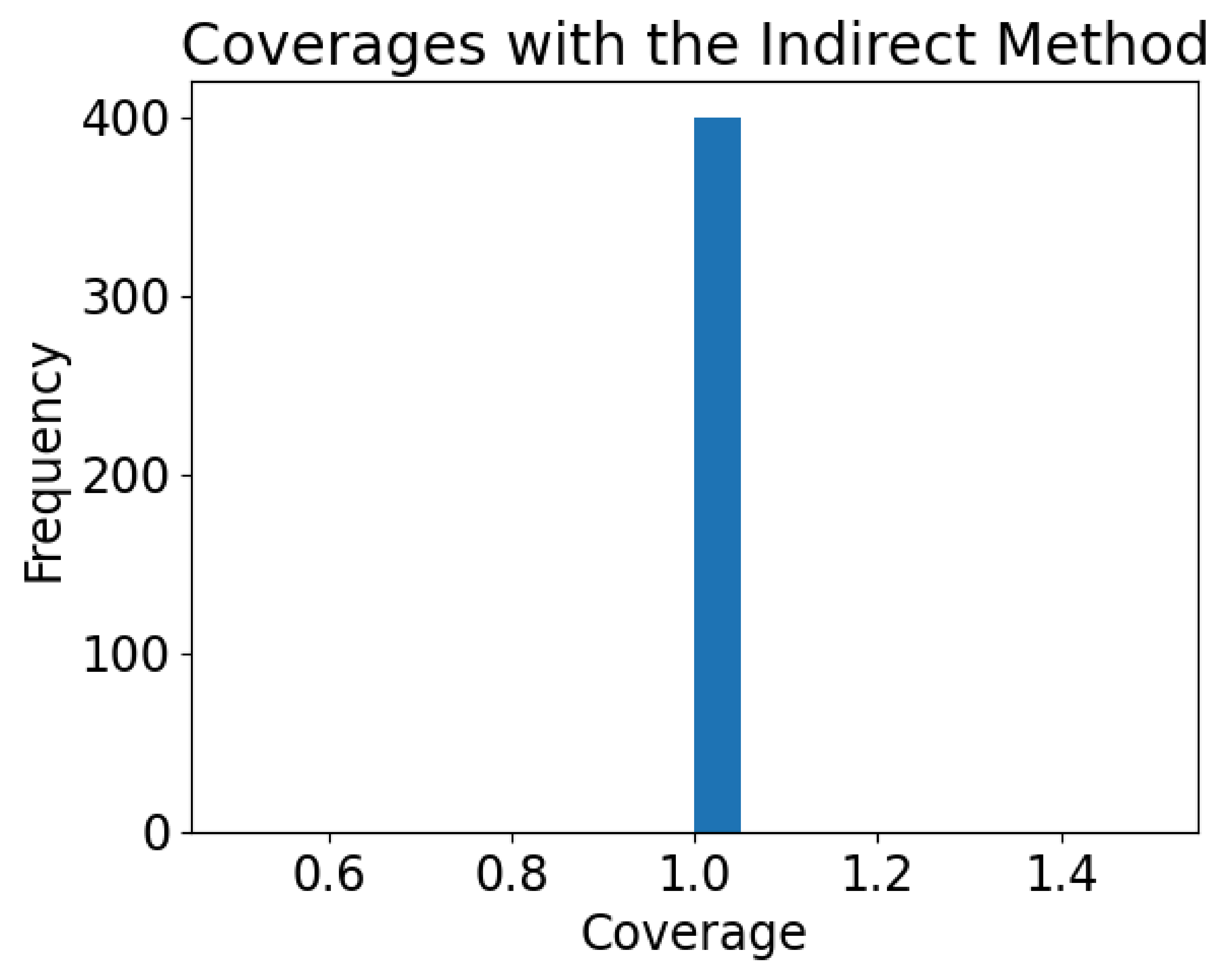
```
# Let's illustrate one example of nonconformity scores.
if i == 0:
    plt.hist(indirect_nonconformity_scores[:-1], bins = 20)
    plt.xlabel("Nonconformity Score")
    plt.ylabel("Frequency")
    plt.axvline(c_indirect, label = "c", color = "g")
    plt.legend()
    plt.title("Nonconformity Scores from the Indirect Method")
    plt.savefig("plots/nonconformity_scores_indirect.pdf")
    plt.show()
    # Let's save the experimental results.
    with open("results/indirect_nonconformity_scores.json" , "w") as file:
        json.dump(indirect_nonconformity_scores, file)
    with open("results/c_indirect.json", "w") as file:
        json.dump(c_indirect, file)
# Compute prediction regions.
prediction_regions = dict()
for tau in range(current_time + 1, terminal_time + 1):
    prediction_regions[tau] = c_indirect / alphas[tau]
# Now, we are ready to validate the results.
indirect_correct_count = 0
indirect_test_robustnesses = []
indirect_test_lowerbound_robustnesses = []
for j in range(test_size):
    indirect_test_robustness = compute_robust_semantics(test_alts[j], test_vels[j])
    # We now call the previously written function for the worst-case semantics.
    indirect_test_lowerbound_robustness = compute_worst_robust_semantics(test_alts[j], test_vels[j],
    prediction_regions)
    if indirect_test_robustness >= indirect_test_lowerbound_robustness:
        indirect_correct_count += 1
    indirect_test_robustnesses.append(indirect_test_robustness)
    indirect_test_lowerbound_robustnesses.append(indirect_test_lowerbound_robustness)
# We plot the scatter plot of the robustnesses for testing.
if i == 0:
    sorted_indirect_test_robustnesses, sorted_indirect_test_lowerbound_robustnesses = zip(*sorted(zip
    (indirect_test_robustnesses, indirect_test_lowerbound_robustnesses)))
    dot_sizes = [5 for j in range(test_size)]
    plt.scatter([j for j in range(test_size)], sorted_indirect_test_robustnesses, s=dot_sizes, color = "r", label=
    "$\\rho^\phi(X, \\tau_0)$")
    plt.scatter([j for j in range(test_size)], sorted_indirect_test_lowerbound_robustnesses , s=dot_sizes, color =
    "g", label= "$\\rho^*$")
    plt.xlabel("Sample (Sorted on $\\rho^\phi(X, \\tau_0)$)")
    plt.ylabel("Robust Semantics Value")
    plt.legend()
    plt.title("Robustnesses for the Indirect Method on the Test Data")
    plt.savefig("plots/indirect_robustnesses_scatter.pdf")
    plt.show()
    # Save the experimental results.
    with open("results/indirect_test_robustnesses.json", "w") as file:
        json.dump(indirect_test_robustnesses, file)
    with open("results/indirect_test_lowerbound_robustnesses.json", "w") as file:
        json.dump(indirect_test_lowerbound_robustnesses, file)
indirect_coverage = indirect_correct_count / test_size
indirect_coverages.append(indirect_coverage)
print("The Coverage of the Indirect Method is: ", indirect_coverage)
```





COVERAGES OF BOTH THE METHODS





Aspect	Direct Method	Indirect Method
Evaluation Approach	Directly evaluates the satisfaction of the STL formula ϕ using predicted states.	Evaluates worst-case satisfaction over prediction regions for future states.
Uncertainty Handling	Quantifies uncertainty in the satisfaction measure for a single predicted trajectory.	Considers uncertainty by evaluating all possible states within prediction regions.
Conservatism	Less conservative; focuses on the most likely predicted trajectory.	More conservative; accounts for worst-case scenarios within prediction regions.
Computation Complexity	Lower computational cost, as only one trajectory is evaluated.	Higher computational cost, due to evaluation over multiple possible states in prediction regions.
Use Case	Suitable for systems with low uncertainty and where fast verification is sufficient.	Ideal for safety-critical systems or those with high uncertainty, requiring worst-case guarantees.

CONCLUSION

We presented two predictive runtime verification algorithms to compute the probability that the current system trajectory violates a signal temporal logic specification. Both algorithms use i) trajectory predictors to predict future system states and ii) conformal prediction to quantify prediction uncertainty. Conformal prediction enables us to obtain valid probabilistic runtime verification guarantees. To the best of our knowledge, these are the first formal guarantees for a predictive runtime verification algorithm that applies to widely used trajectory predictors such as RNNs and LSTMs while being computationally simple and making no assumptions on the underlying distribution. An advantage of our approach is that a changing system specification does not require expensive retraining as in existing works. We concluded with experiments of an F-16 aircraft and a self-driving car equipped with LSTMs.

Future Advancements - It offers a practical solution for runtime verification across various CPS applications, enhancing safety and performance predictability in uncertain environments.

THANK YOU