Efficiently predicting and repairing failure modes via sampling

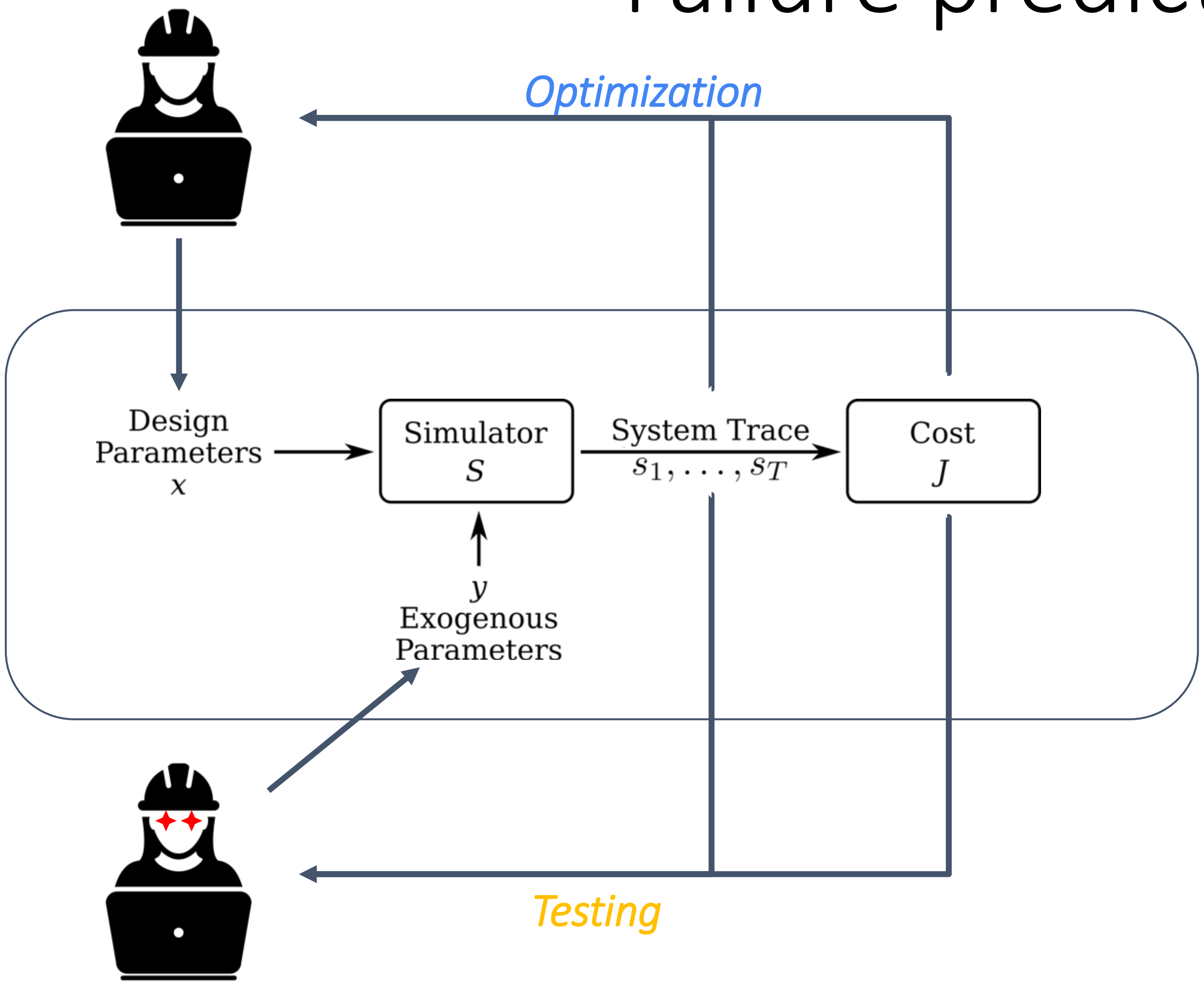
and updates on the GUAM Python version

Chuchu Fan
Assistant Professor of AeroAstro and LIDS
REALM Lab: REliable Autonomous systems Lab at MIT
chuchu@mit.edu

Joint work with Charles Dawson

Presented at the NASA ULI AVIATE Seminars

Failure prediction and repair

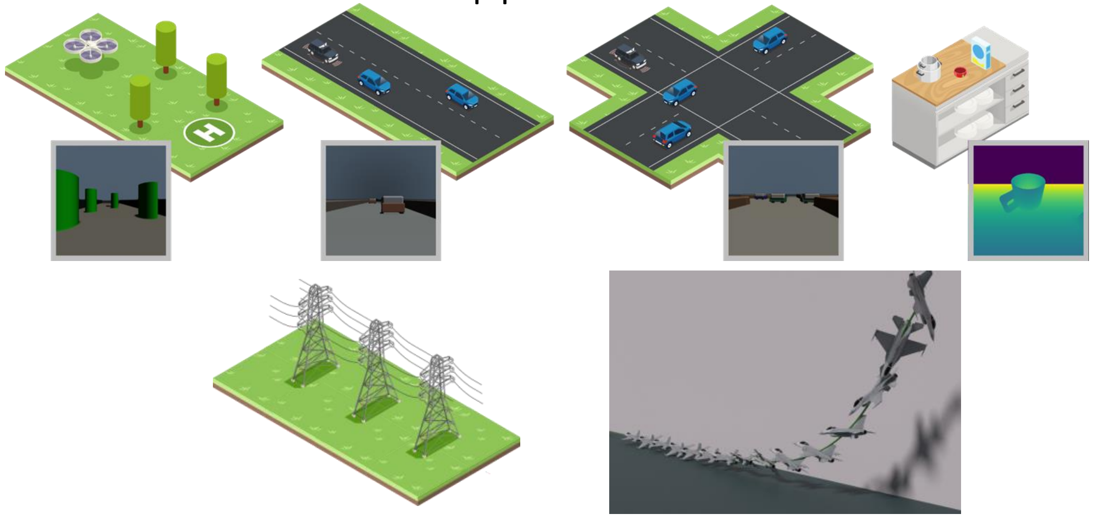


Problem formulation: $\underset{x}{\operatorname{argmin}}\, \underset{y}{\max}\, J \circ S(x, y)$

Our framework answers the following questions

1. How can we predict likely failures ***prior to deployment?***
2. How can we ***understand*** the causes of those failures?
3. What can we do to ***mitigate*** those failures?

Current applications



Failure prediction and repair

$$\underset{x}{\operatorname{argmin}}\ \max_{y} J \circ S(x, y)$$

To solve this problem, we need to find a generalized Nash equilibrium between the optimizer and the adversary:

$$\boxed{x^* = \operatorname{argmin}_x \mathbb{E}_y[J \circ S(x, \phi)]}$$

$$\boxed{y^* = \operatorname{argmax}_x J \circ S(x, \phi)}$$

This avoids the risk of "overfitting" to a particular value of y^*.

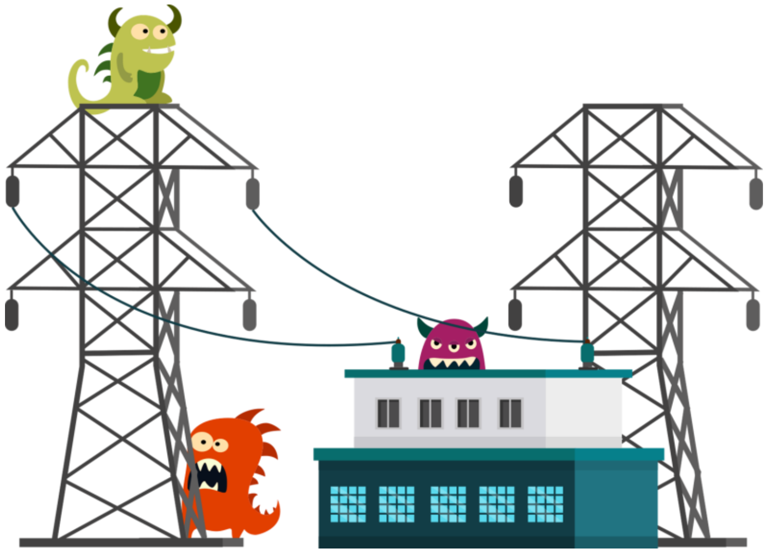
We can use the values of y^* found during successive iterations as high-quality counterexamples to guide the optimization of x.

Prior work: prediction repair

Model-based verification	Black-box verification	Adversarial training
- SAT/SMT - Reachability - Hamilton-Jacobi	- RL - Bayesian optimization - Importance sampling	- Zero-sum games - Domain randomization
☑ Formal guarantees	OK Few guarantees (statistical)	✗ No guarantees
✗ Symbolic model required	☑ Model-free	OK Model need not be symbolic
✗ Computationally expensive	✗ Computationally expensive	☑ Computationally cheap

Ideally, we'd like a method that…

- Explores without getting stuck in local minima,
- Runs faster than black-box verification,
- Doesn't require a symbolic model,
- Combines prediction and repair.



Existing methods overfit to easy test cases

Leads to false confidence

Instead, our method prioritizes diversity

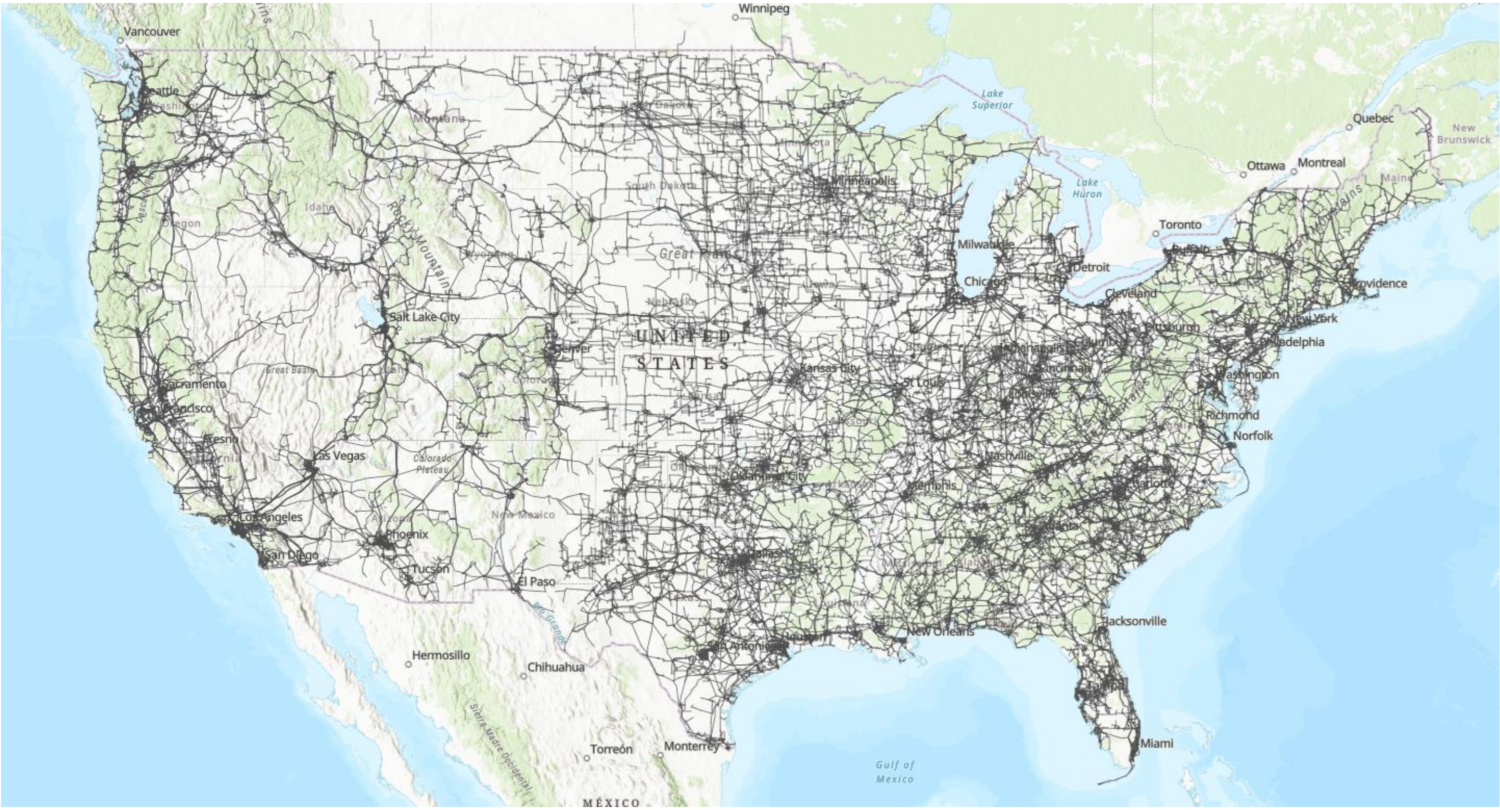
Sampling-based algorithms can help!

Test-case diversity leads to safer operations

Fewer surprises

Images by macrovector and svstudioart on Freepik

Dilemma in failure prediction



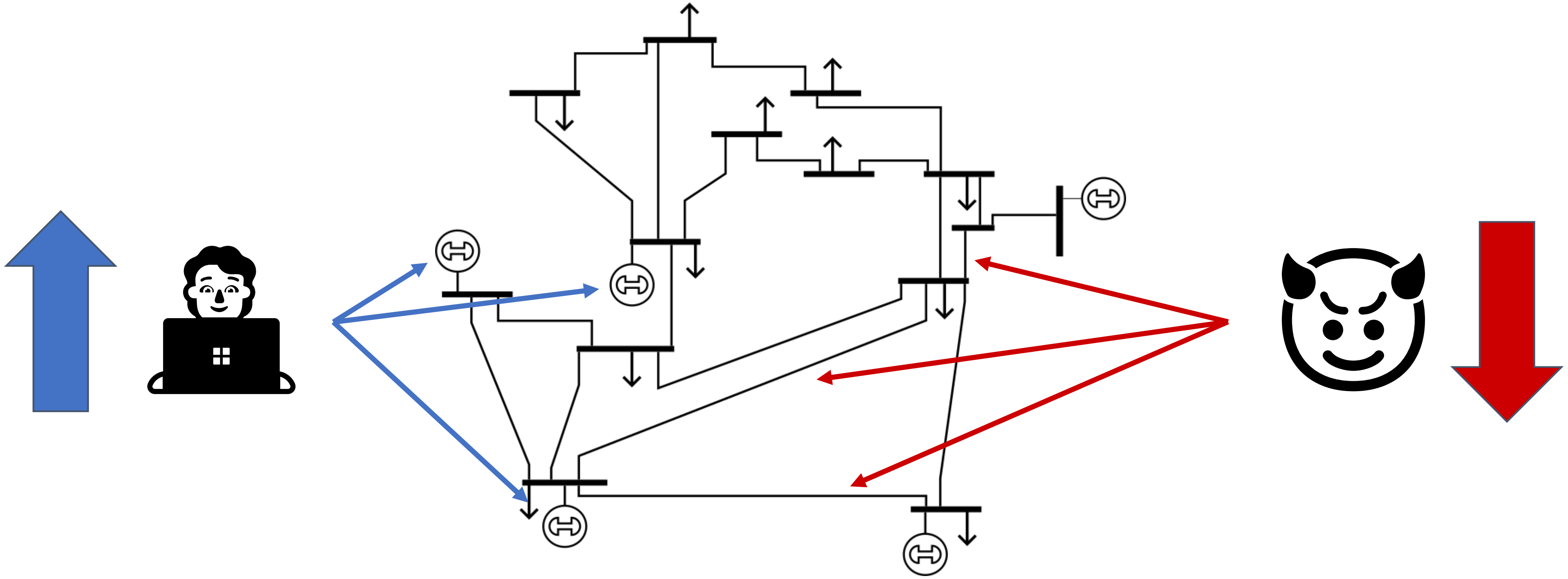
Option 1: check all possible failures

Takes too much time

Option 2: check only worst-case failures

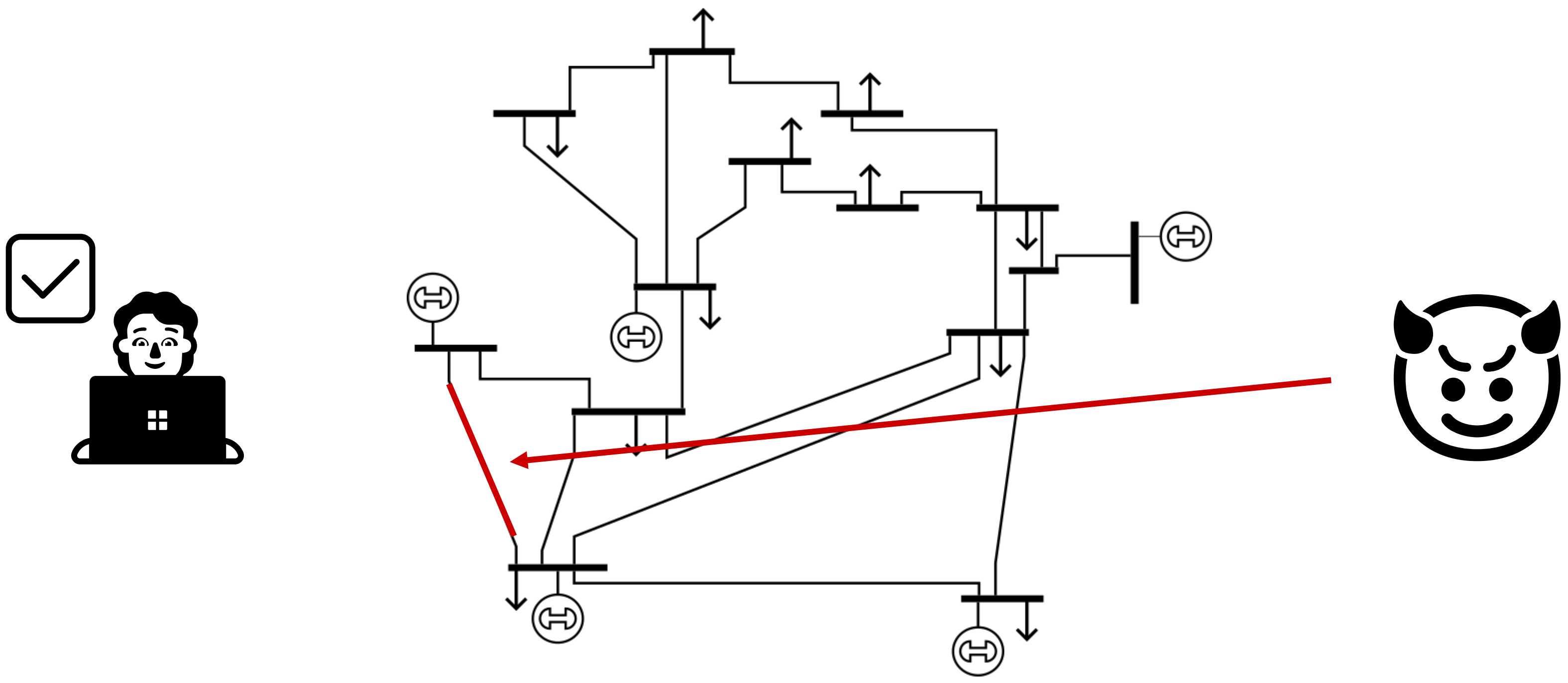
How do we find those worst-case failures?

A common approach to secure dispatch



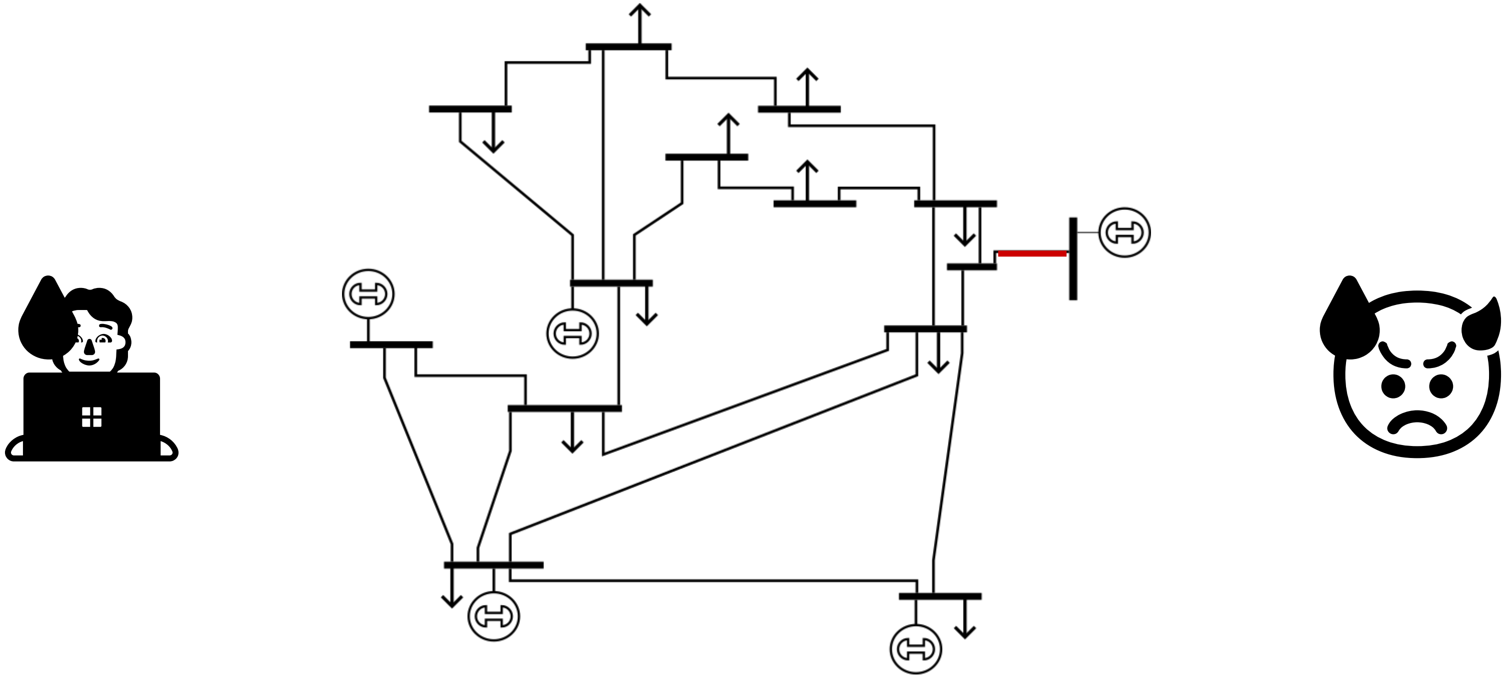
Optimization tug-of-war with the adversary

A common approach to secure dispatch - what can go wrong?



You’ve successfully mitigated the worst failure found by the adversary…

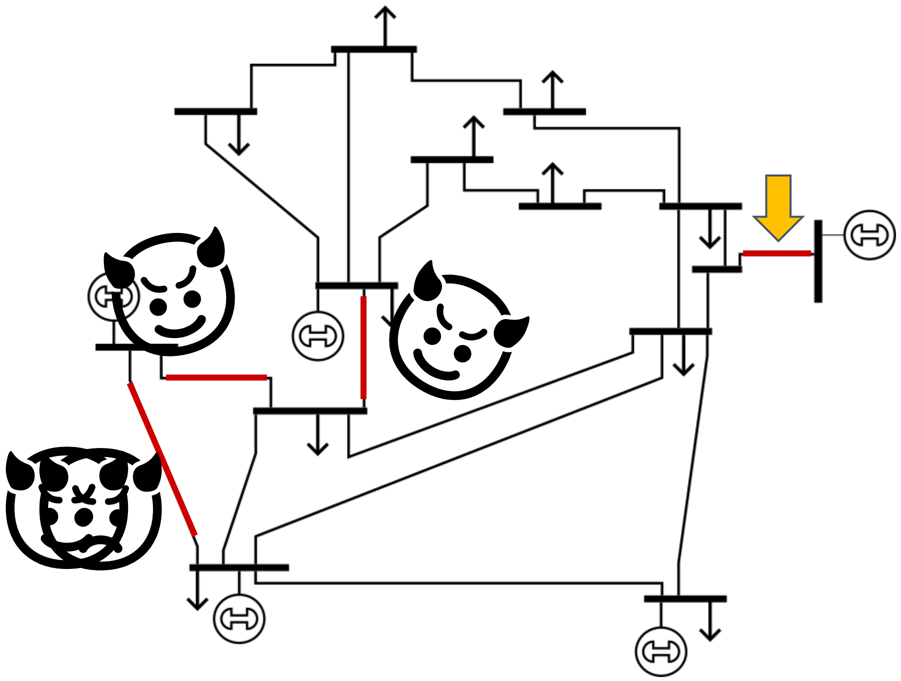
A common approach to secure dispatch - what can go wrong?

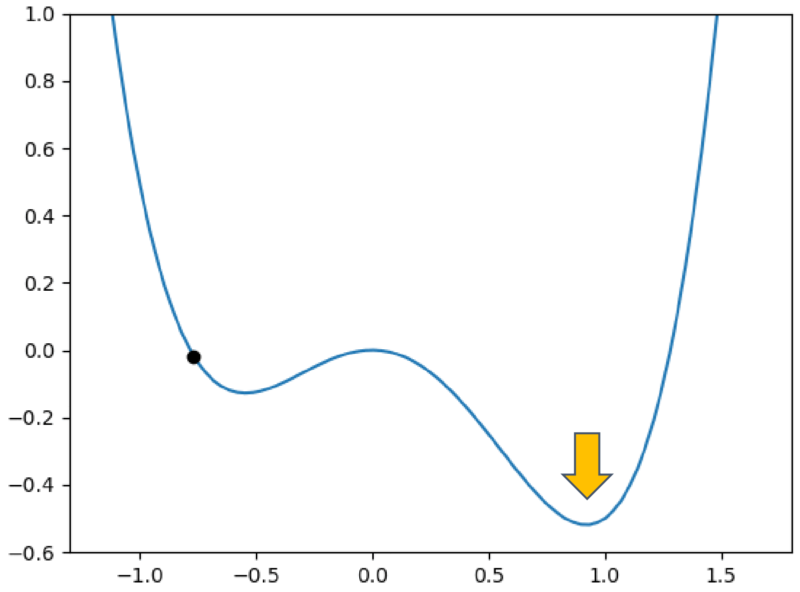


You've successfully mitigated the worst failure found by the adversary…

But what if the adversary didn't find the true worst case?

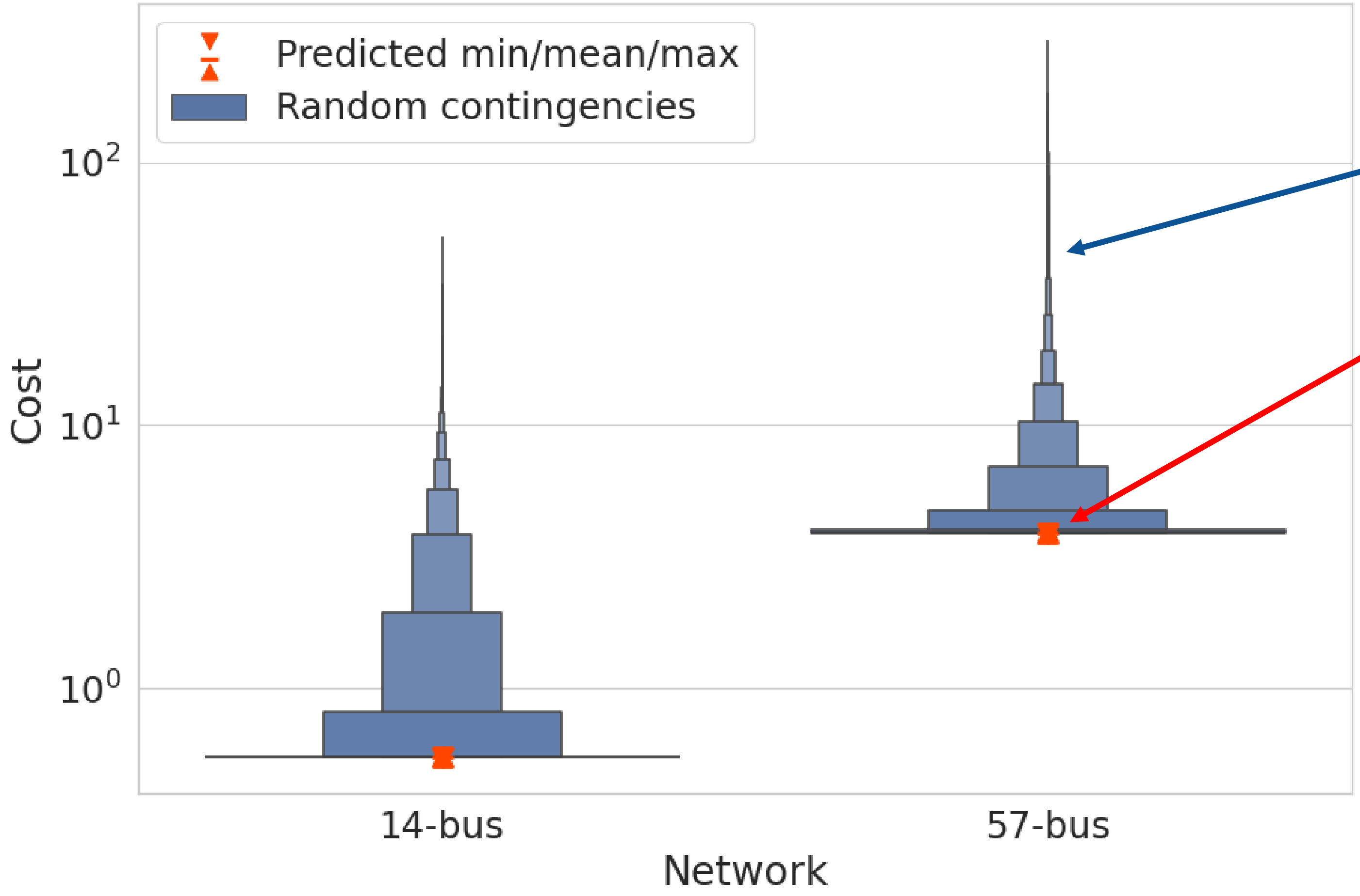
Looking for the worst-case can lead to dead-ends



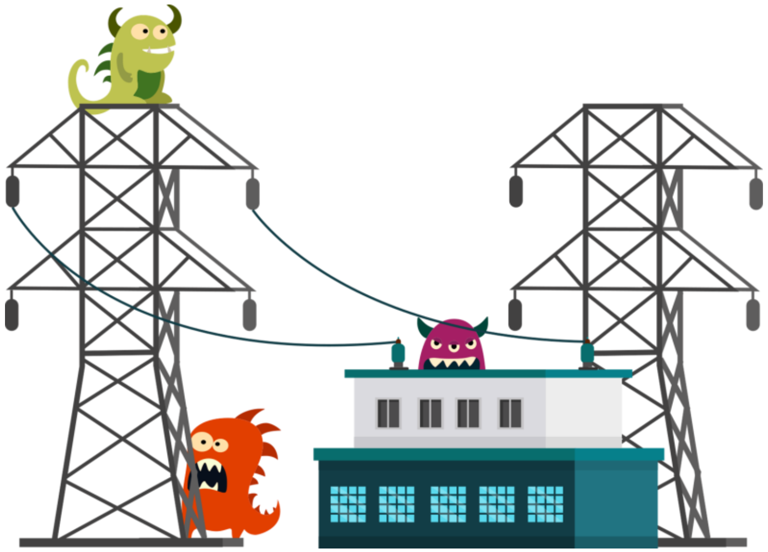


$$x = \operatorname{argmin}\ U(x)$$

Adversarial optimization converges *in theory*…
but it misses many failures in practice.



17% of failures are worse than the predicted worst case



Existing methods overfit to easy test cases

Leads to false confidence

Instead, our method prioritizes diversity

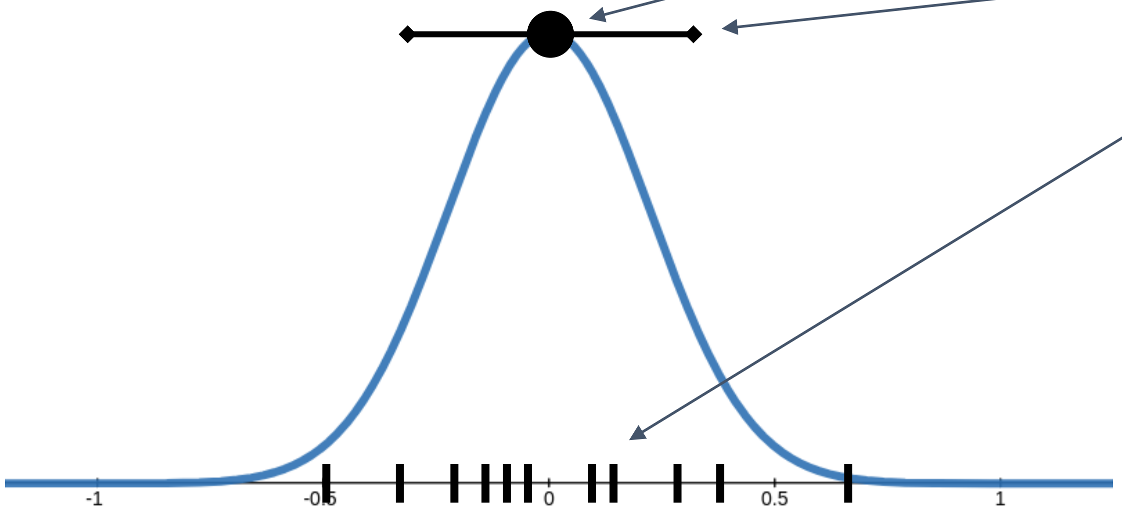
Sampling-based algorithms can help!

Test-case diversity leads to safer operations

Fewer surprises

Images by macrovector and svstudioart on Freepik

Inspiration from computational statistics: sampling



What's more representative?

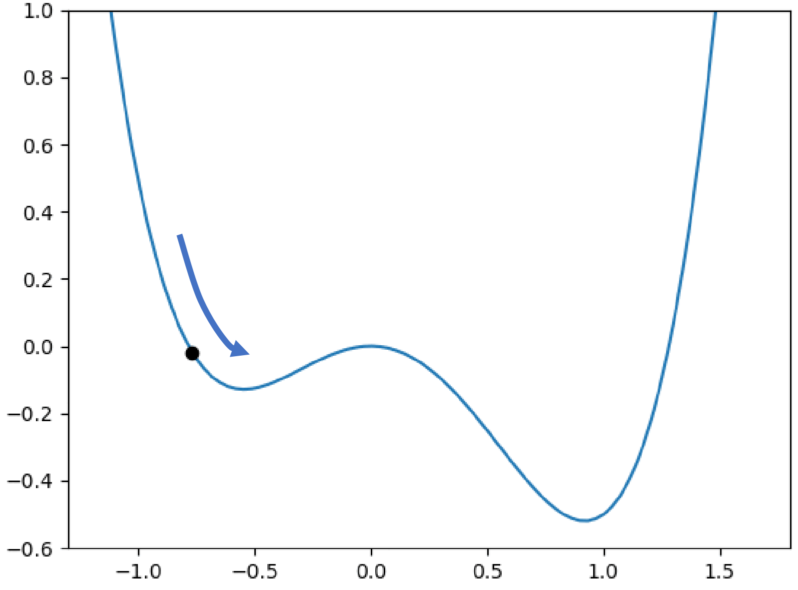
The mean?

Mean + standard deviation?

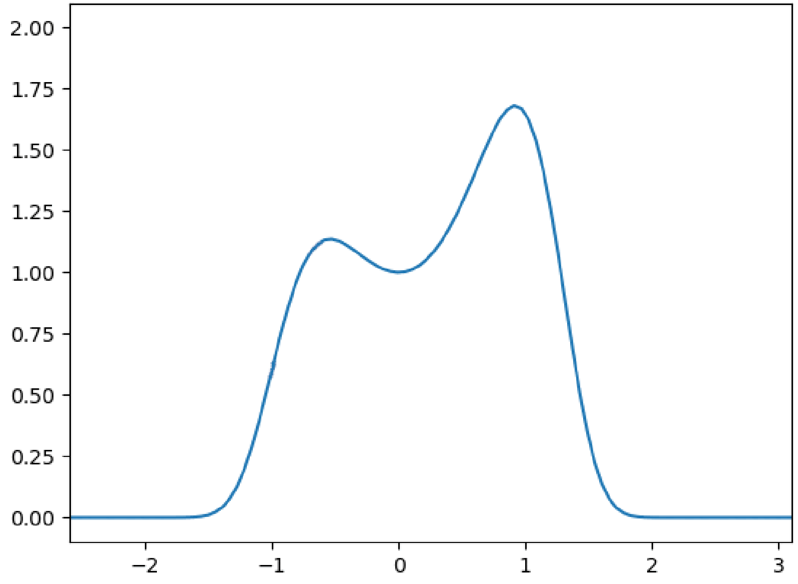
Samples from the distribution!

- Can represent any distribution
- Tells us about risk
- Covers extreme events

Looking for the worst-case = looking for local minima
Sampling (instead of optimizing) gives the full picture



$$x = \operatorname{argmin}\, U(x)$$



$$p(x) \propto e^{-U(x)}$$

How to find diverse failure modes?

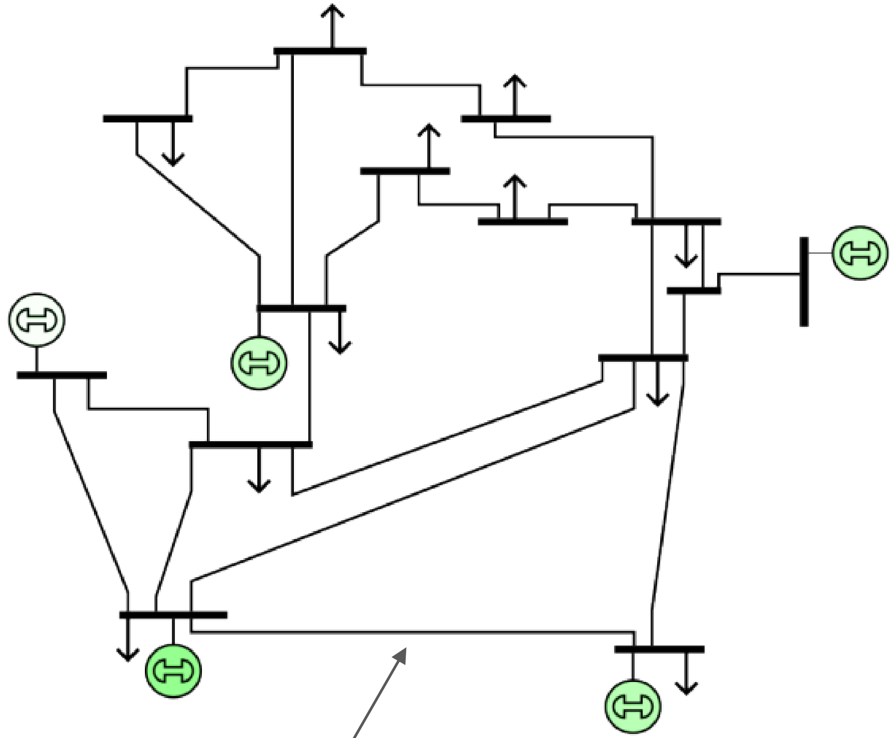
Step 1: balance failure likelihood with severity

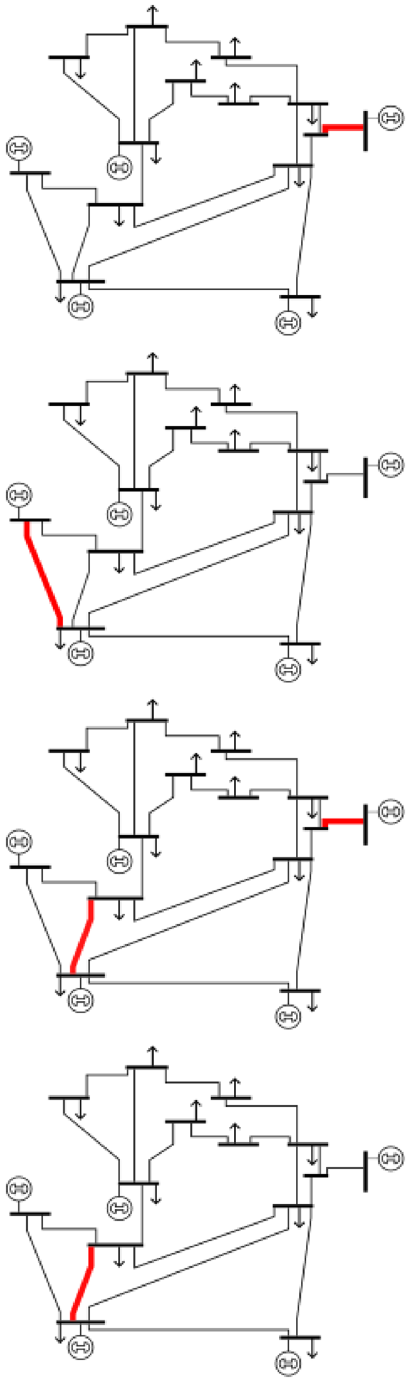
$$J_r(x, y) = J \circ S(x, y) + \log p_{y,0}(y)$$ (Risk-adjusted severity)

Step 2: instead of optimizing, sample *y* from the posterior:

$$y^* = \arg\max_y J_r(x, y) \qquad y \sim p(y|x) \propto p_{y,0}(y) e^{J \circ S(x,y)}$$

Sampling test cases leads to more diversity





$p_{y,0}(y)$ = mixture model with 5% chance of failure for each line

Failure repair as Bayesian inference

Failure mode ***repair*** is also a sampling problem

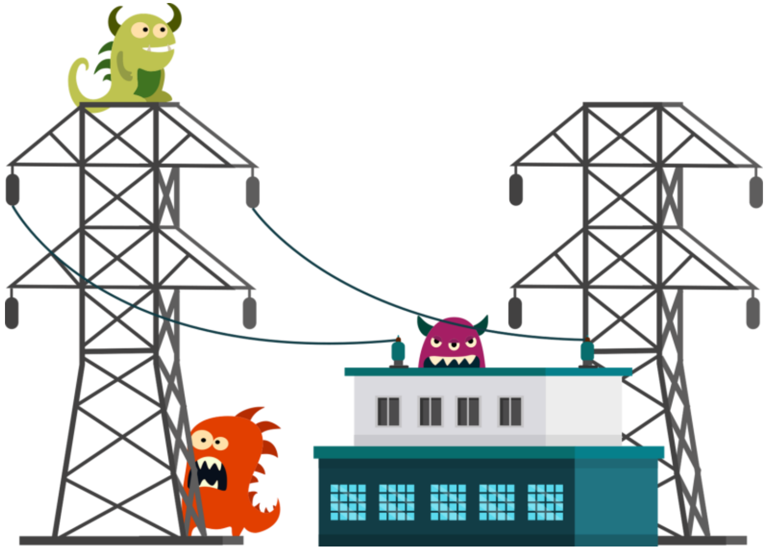
$$x^* = \min_x \left[\mathbb{E}_y\, J(x,y) + \log p_{x,0}(x)\right]$$

expectation over predicted failures

Regularize using prior distribution for *x*

$$x^* \sim p(x|y_1^*, \ldots, y_{n_y}^*) \propto p_{x,0}(x) e^{-\sum_i J \circ S(x, y_i^*)/n_y}$$

"Distribution of good designs given expected failure modes"



Existing methods overfit to easy test cases

Leads to false confidence

Instead, our method prioritizes diversity

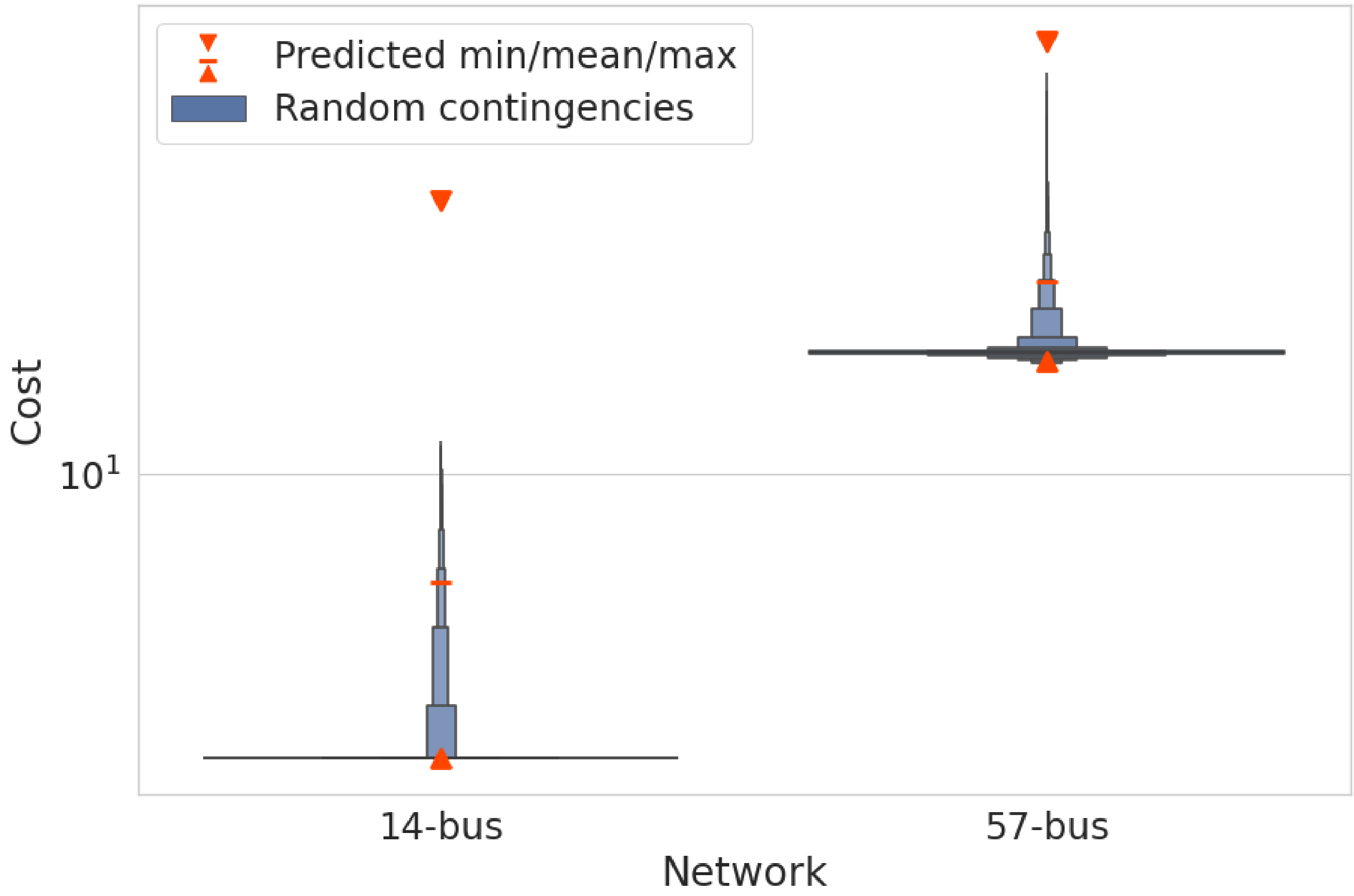
Sampling-based algorithms can help!

Test-case diversity leads to safer operations

Fewer surprises

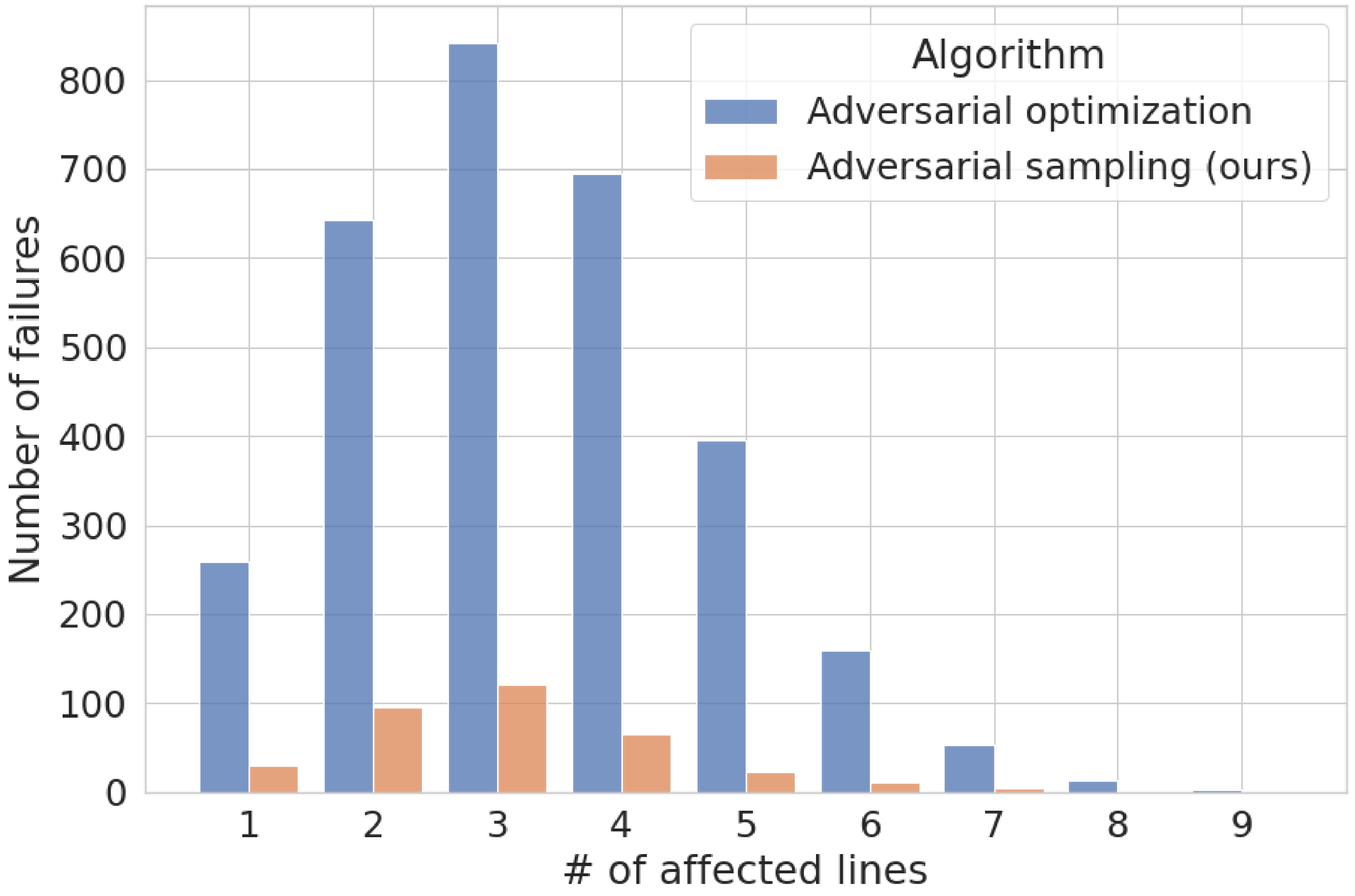
Images by macrovector and svstudioart on Freepik

Diverse test cases yield better coverage of possible failures



None of 10^6 random trials exceed predicted worst case

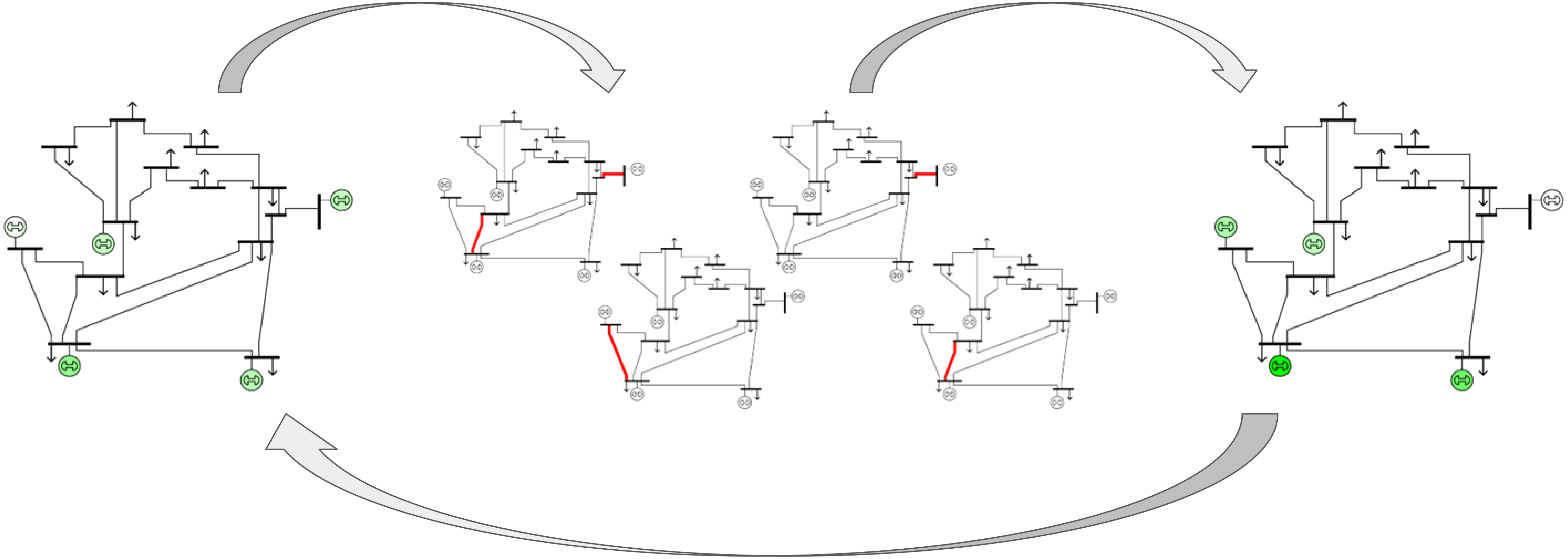
Dispatch using our test cases leads to 10x fewer outages



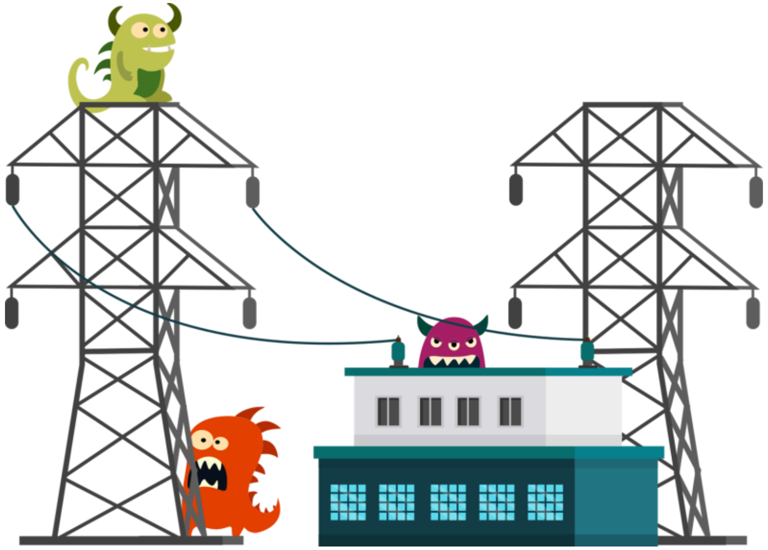
Sequential Adversarial Inference

Predict a diverse set of failure modes for the current design

Use failure modes to inform further design iteration



Loop until design performance has converged



Existing methods overfit to easy test cases

- Leads to false confidence

Instead, our method prioritizes diversity

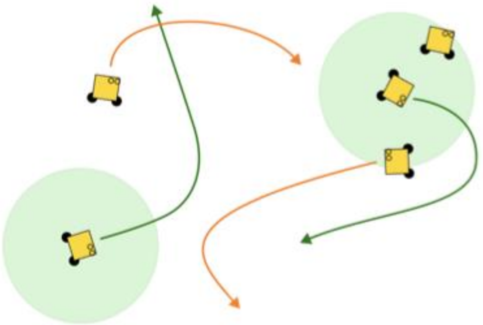
- Sampling-based algorithms can help!

Test-case diversity leads to safer operations

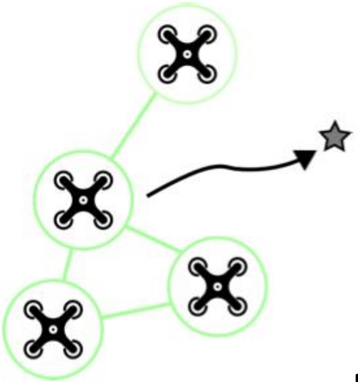
- Fewer surprises

Images by macrovector and svstudioart on Freepik

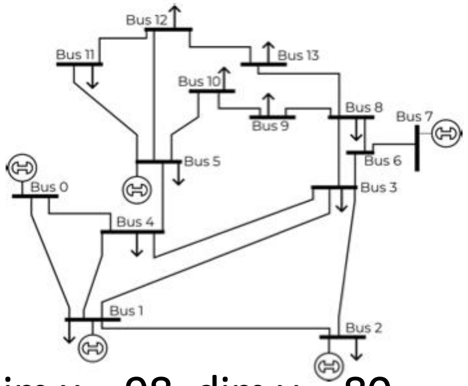
A uniform framework that works across different applications



dim x = 120, dim y = 200



dim x = 100, dim y = 1280



dim x = 98, dim y = 80

dim x = 1800, dim y = 5

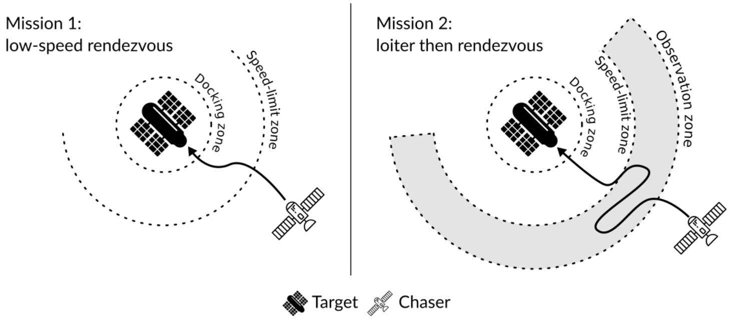
dim x = 1200, dim y = 7

MIT engineers are on a failure-finding mission

The team's new algorithm finds failures and fixes in all sorts of autonomous systems, from drone teams to power grids.

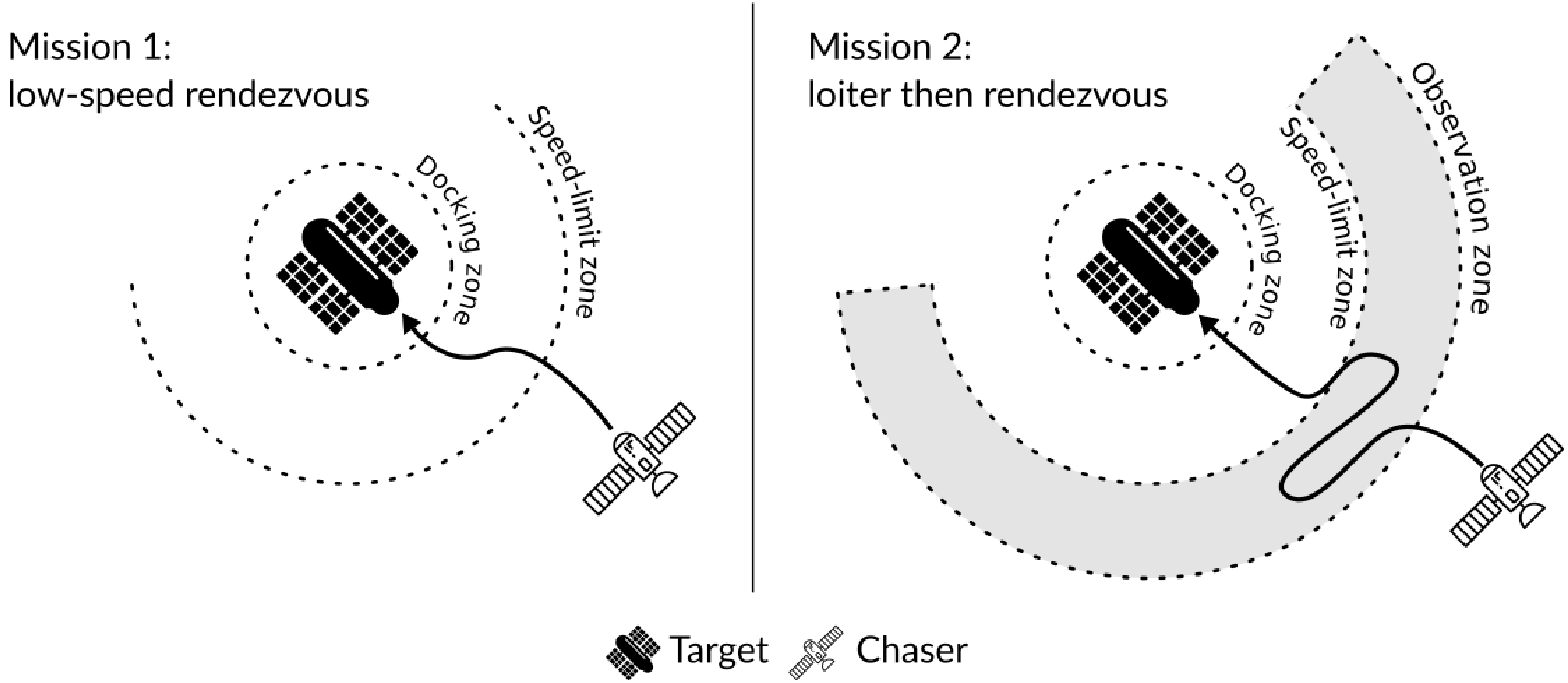
[Full story via MIT News](#)→

Featured on MIT News



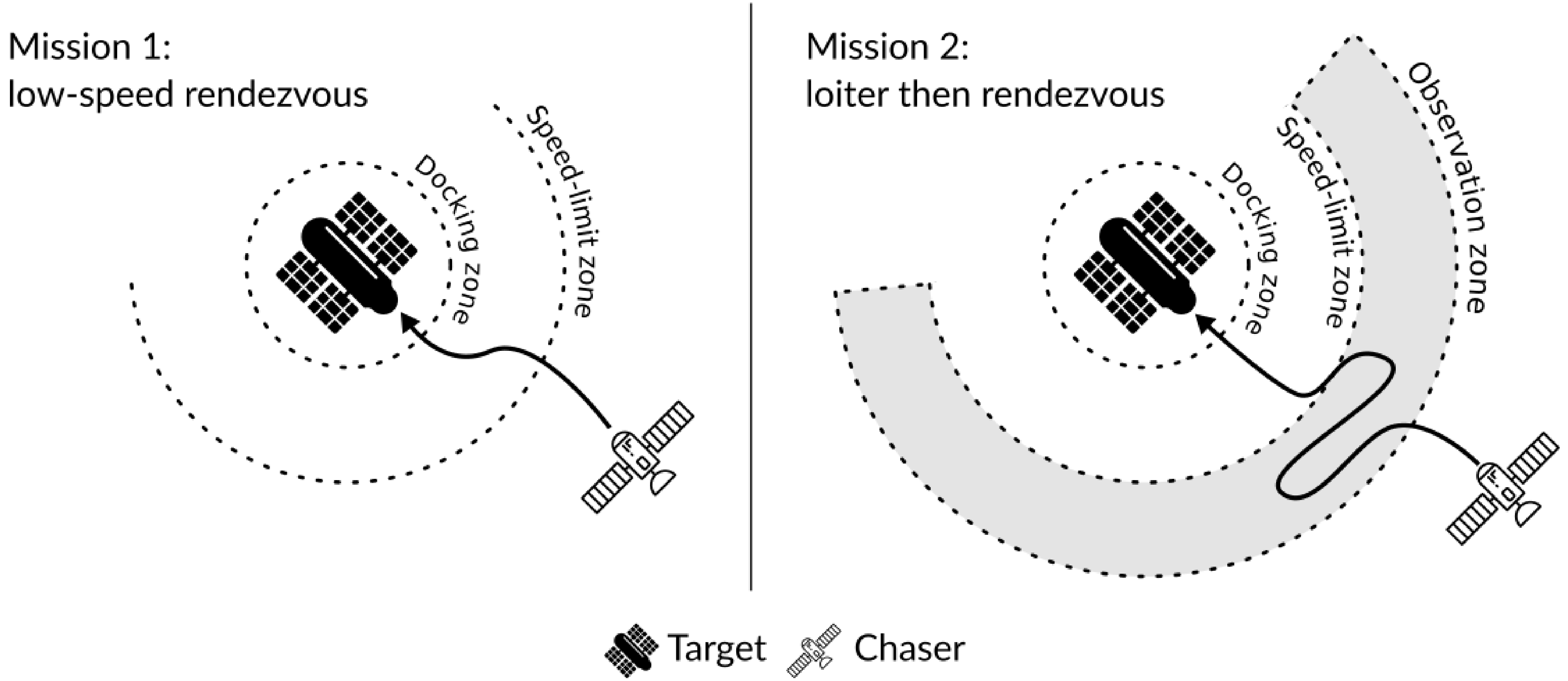
dim x = 318, dim y = 6

Example on satellite rendezvous



Left: the chaser satellite must eventually reach the target while respecting a maximum speed constraint in the region immediately around the target. Right: the chaser must still reach the target and obey the speed limit, but it must also loiter in an observation region for some minimum time before approaching.

Example on satellite rendezvous

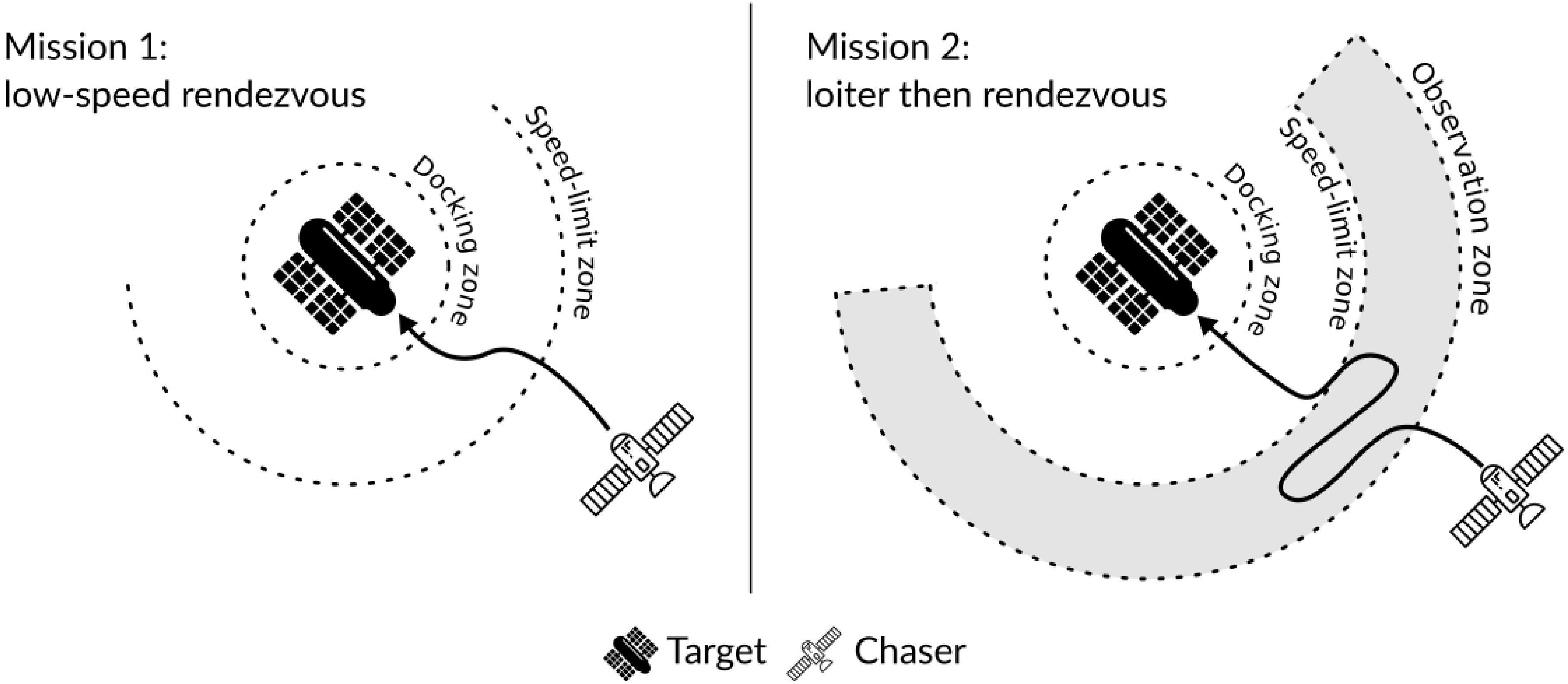


$$\varphi_1 = \varphi_{\text{reach}} \wedge \varphi_{\text{speed-limit}},\ \varphi_2 = \varphi_{\text{reach}} \wedge \varphi_{\text{speed-limit}} \wedge \varphi_{\text{loiter}}$$

$$\varphi_{\text{reach}} = \mathbf{F}(r \leq 0.1), \varphi_{\text{speed-limit}} = \mathbf{G}(r \leq 2 \Rightarrow v \leq 0.1),$$

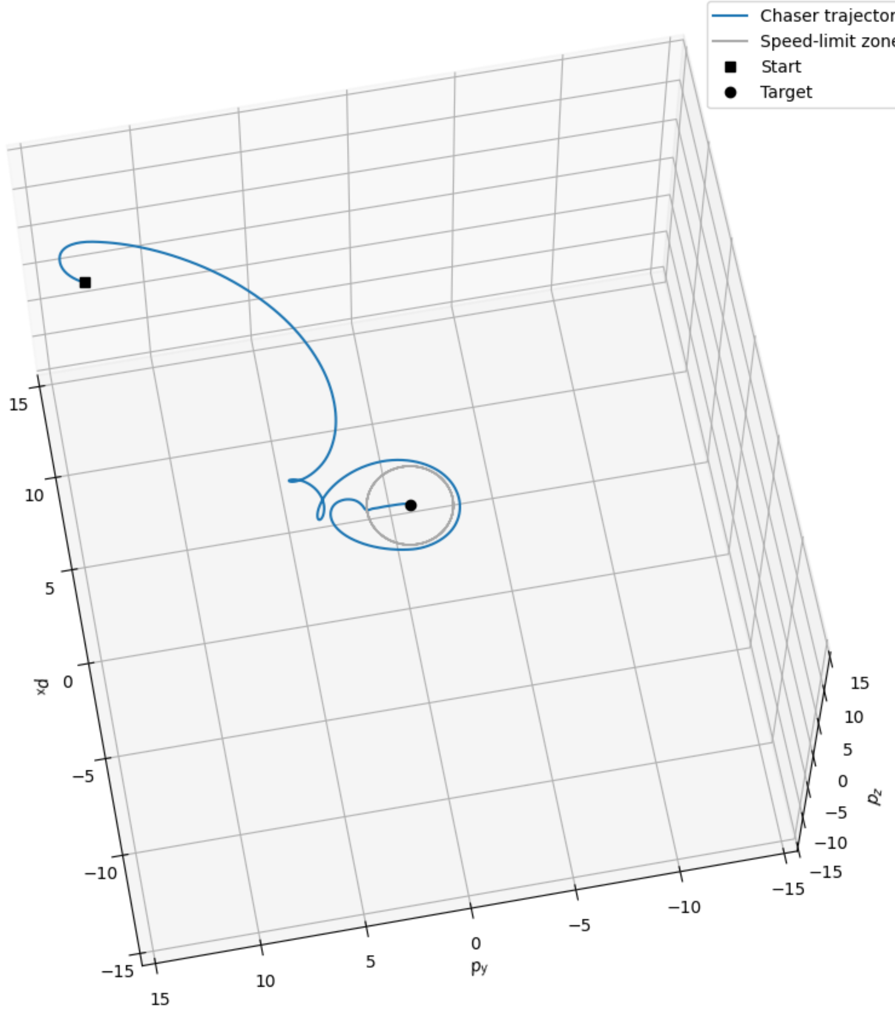
$$\varphi_{\text{loiter}} = \mathbf{FG}_{[0,T_{obs}]}(2 \leq r \wedge r \leq 3)$$

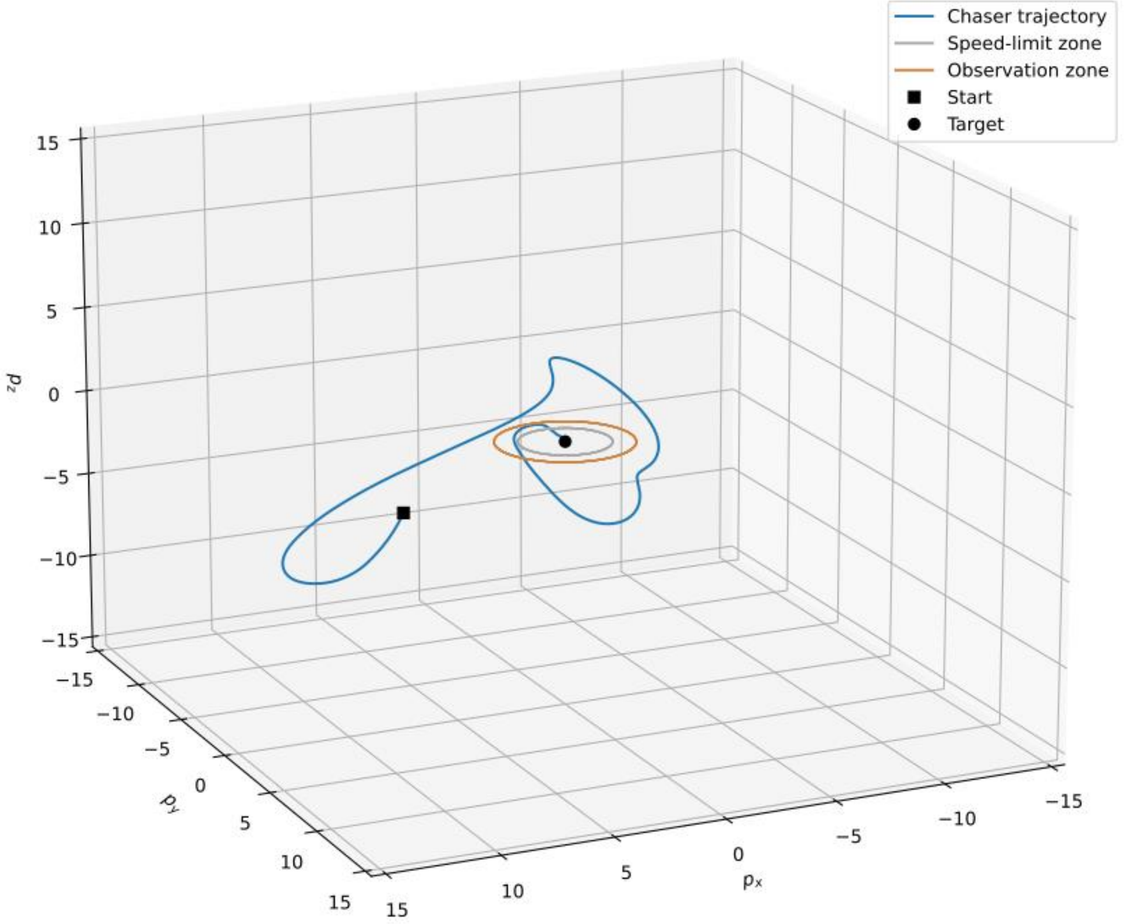
Example on satellite rendezvous



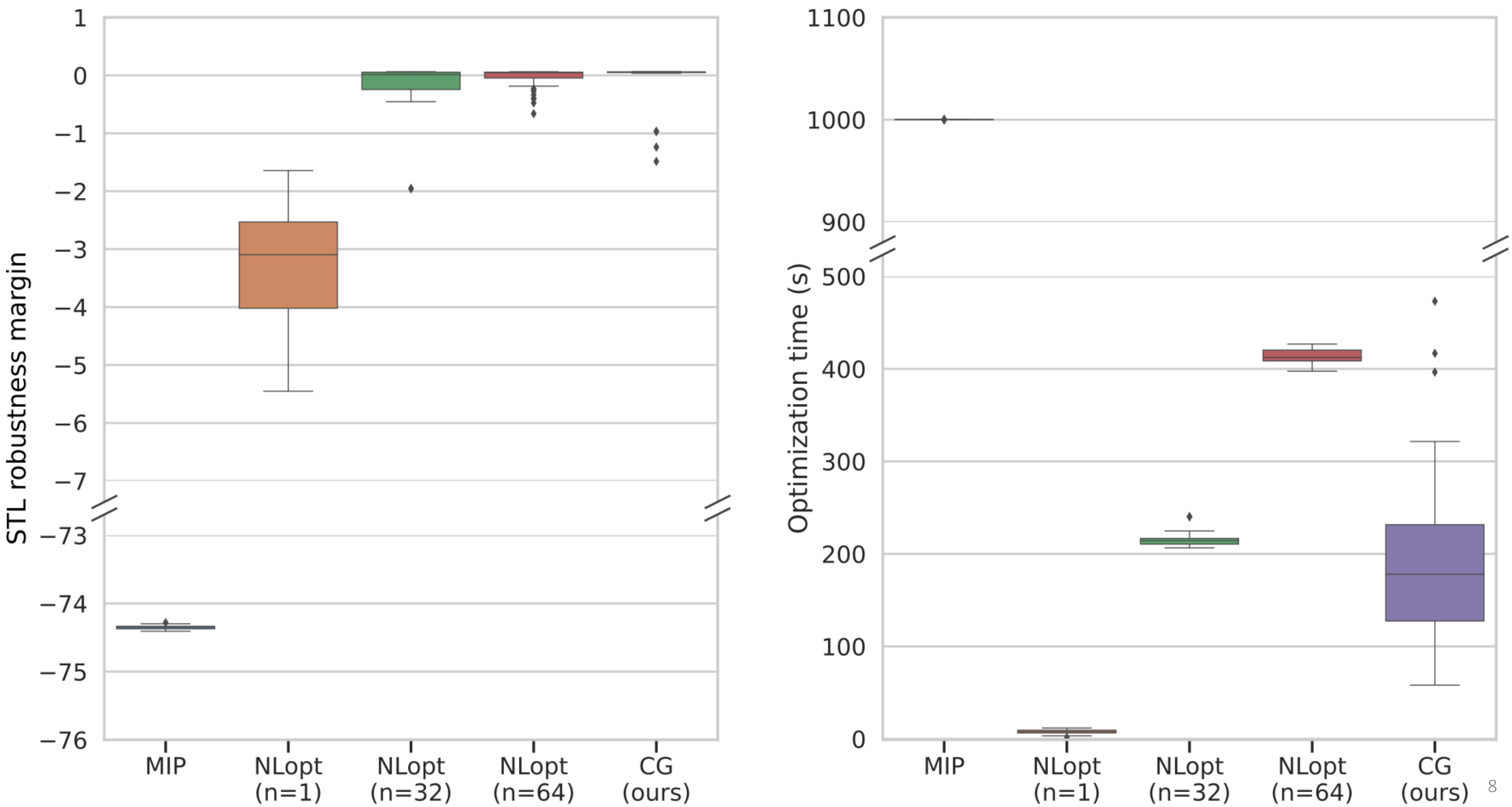
Design parameters x include both state/input waypoints along a planned trajectory and the feedback gains used to track that trajectory, and the exogenous parameters y represent bounded uncertainty in the initial states of the chaser (relative position, relative velocity).

Example on satellite rendezvous

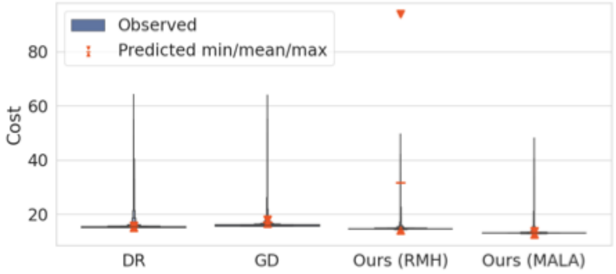




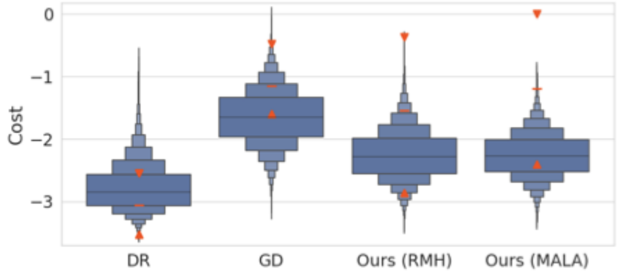
Our algorithm finds the optimized trajectories for both missions.


STL robustness margin
1
0
−1
−2
−3
−4
−5
−6
−7
−73
−74
−75
−76
MIP
NLopt (n=1)
NLopt (n=32)
NLopt (n=64)
CG (ours)
Optimization time (s)
1100
1000
900
500
400
300
200
100
0
MIP
NLopt (n=1)
NLopt (n=32)
NLopt (n=64)
CG (ours)

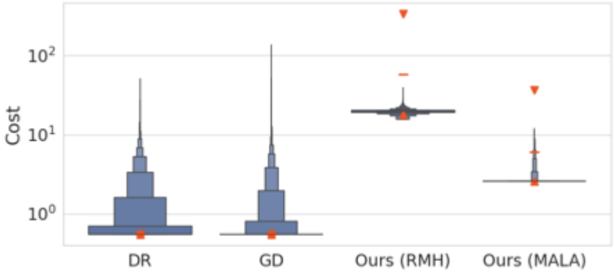
A uniform framework that works across different applications



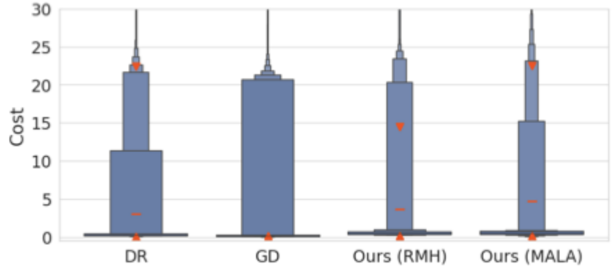
(a) Formation, 5 agents



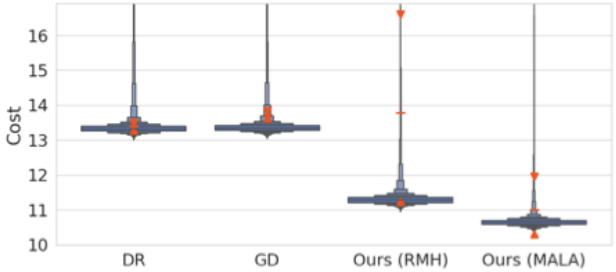
(b) Search, 6 vs. 10



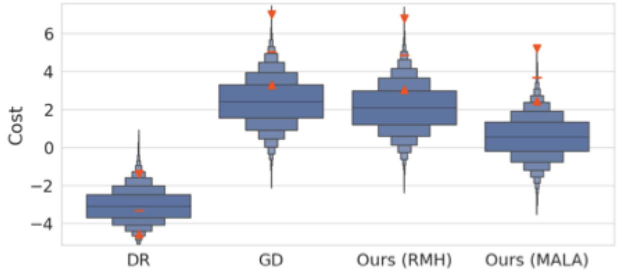
(c) Power grid, 14-bus



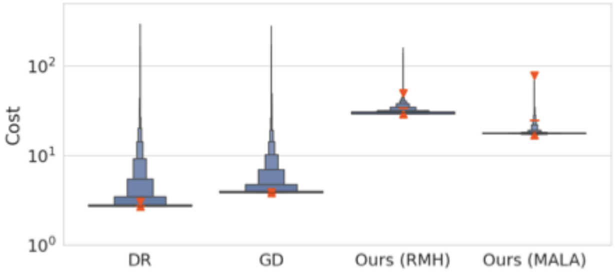
(d) F16 GCAS



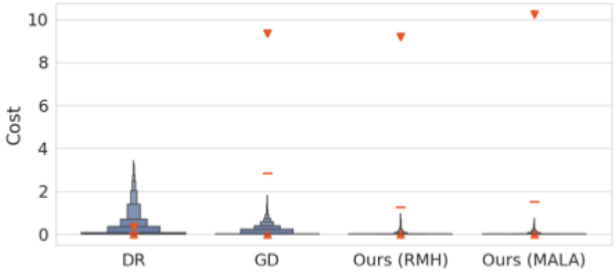
(e) Formation, 10 agents



(f) Search, 12 vs. 20

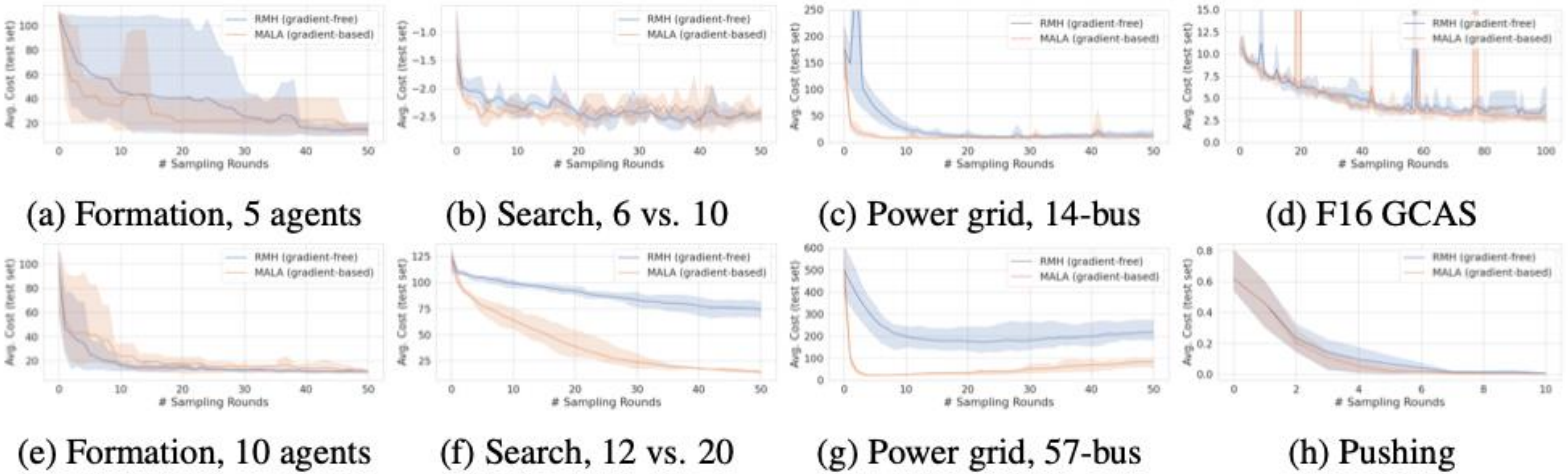


(g) Power grid, 57-bus



(h) Pushing

A uniform framework that works across different applications



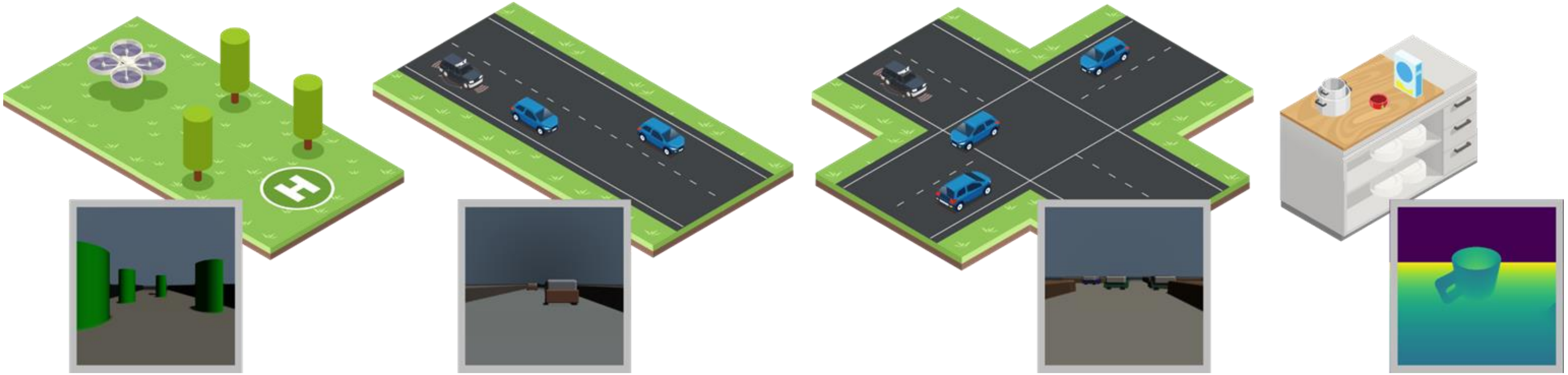
(a) Formation, 5 agents (b) Search, 6 vs. 10 (c) Power grid, 14-bus (d) F16 GCAS

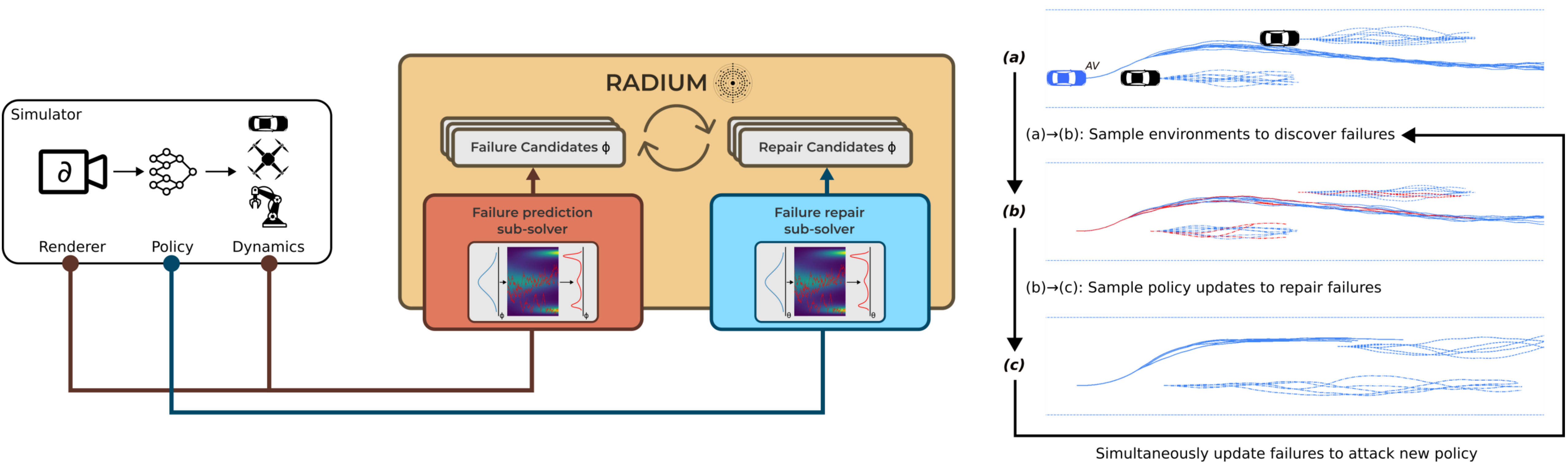
(e) Formation, 10 agents (f) Search, 12 vs. 20 (g) Power grid, 57-bus (h) Pushing

Convergence rates of gradient-based (orange) and gradient-free (blue) samplers

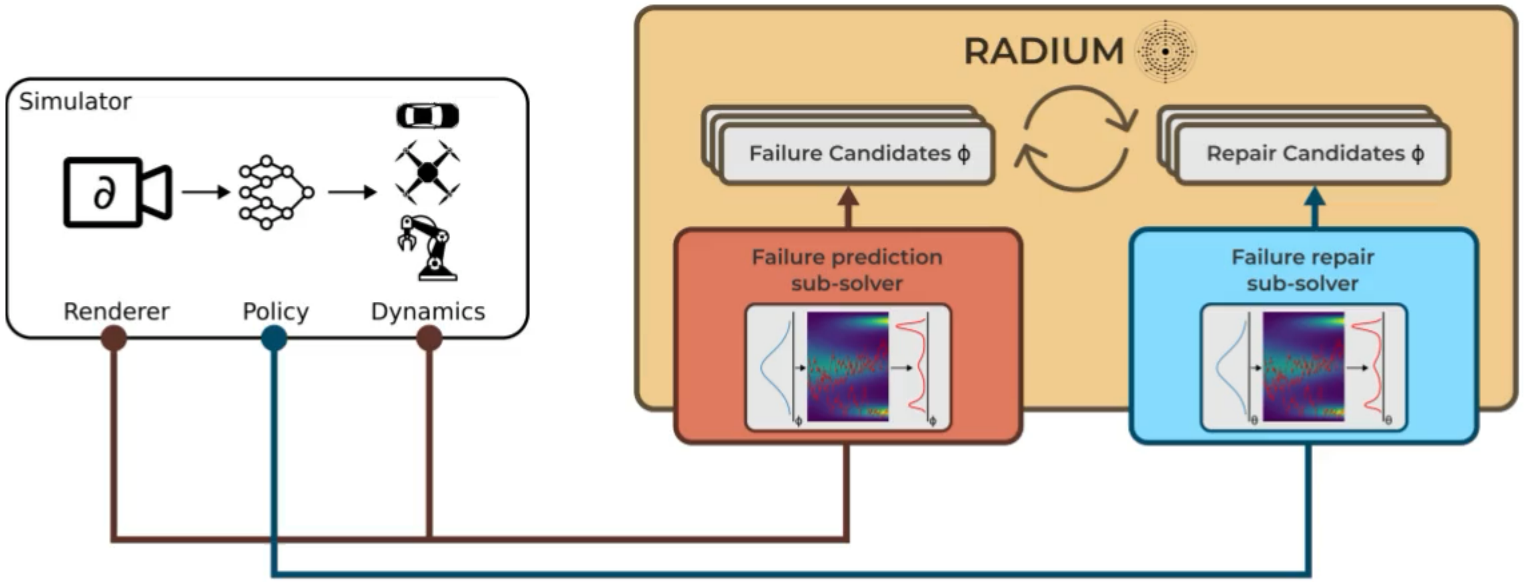
Automatically *predicting* and *mitigating* likely failure modes

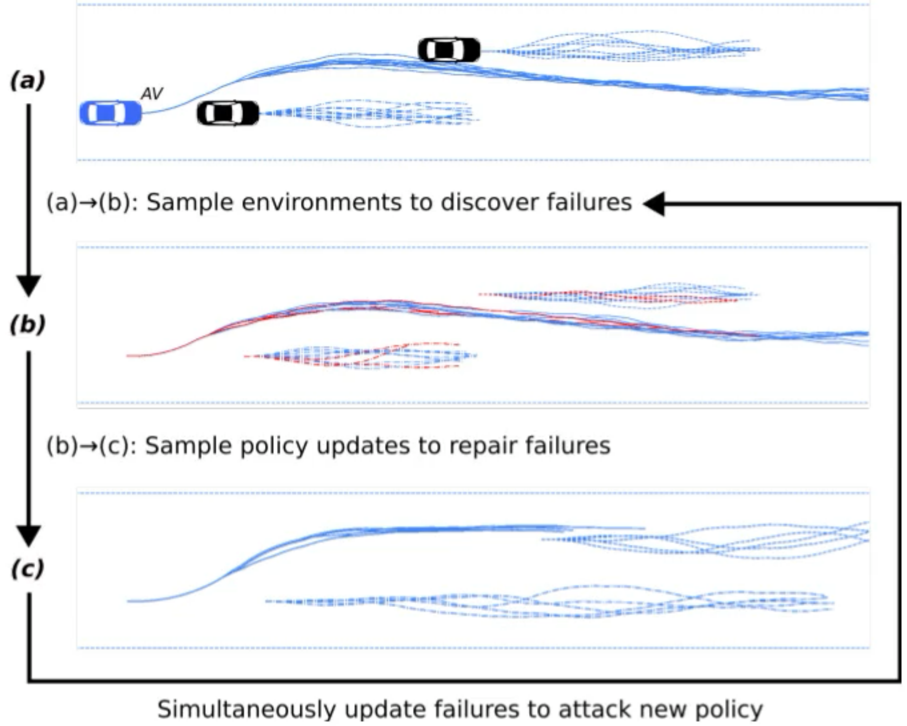
Case studies with perception-based control



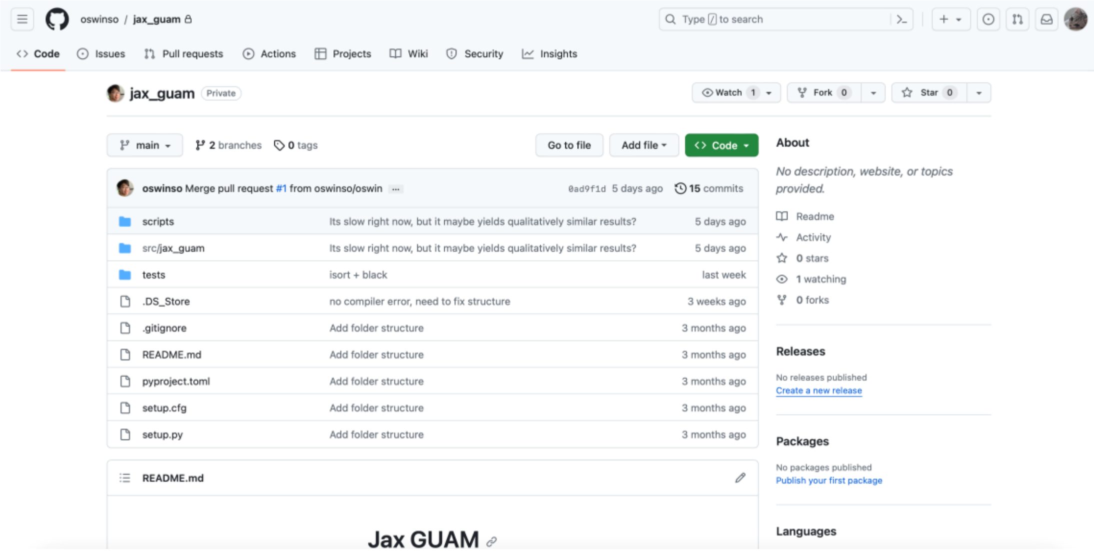

Simulator
∂
Renderer
Policy
Dynamics
RADIUM
Failure Candidates ϕ
Repair Candidates ϕ
Failure prediction
sub-solver
Failure repair
sub-solver
(a)
AV
(a)→(b): Sample environments to discover failures
(b)
(b)→(c): Sample policy updates to repair failures
(c)
Simultaneously update failures to attack new policy

RADIUM
Predicting and repairing failures in learning-based autonomous systems

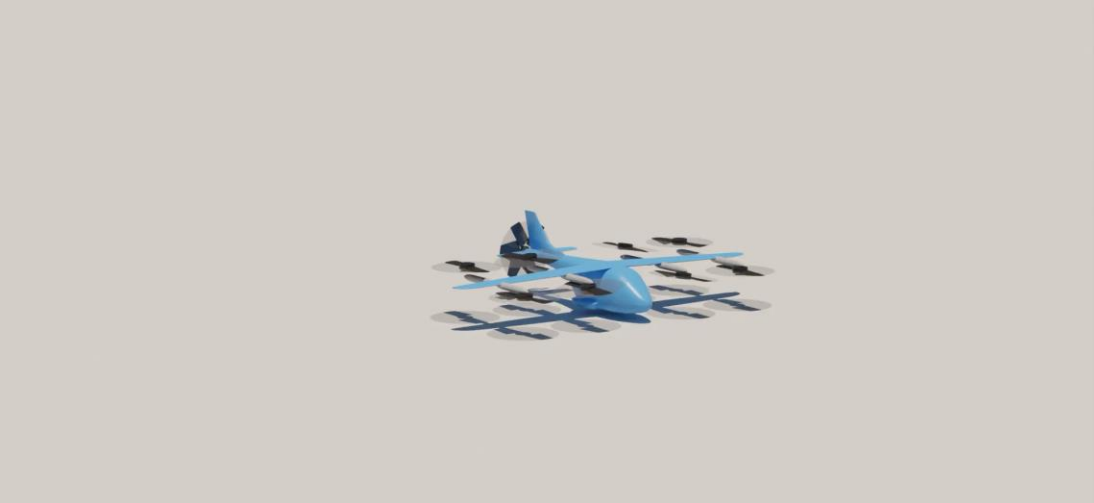




GUAM - JAX Version Translation



A python (differentiable) implementation



A visualizer

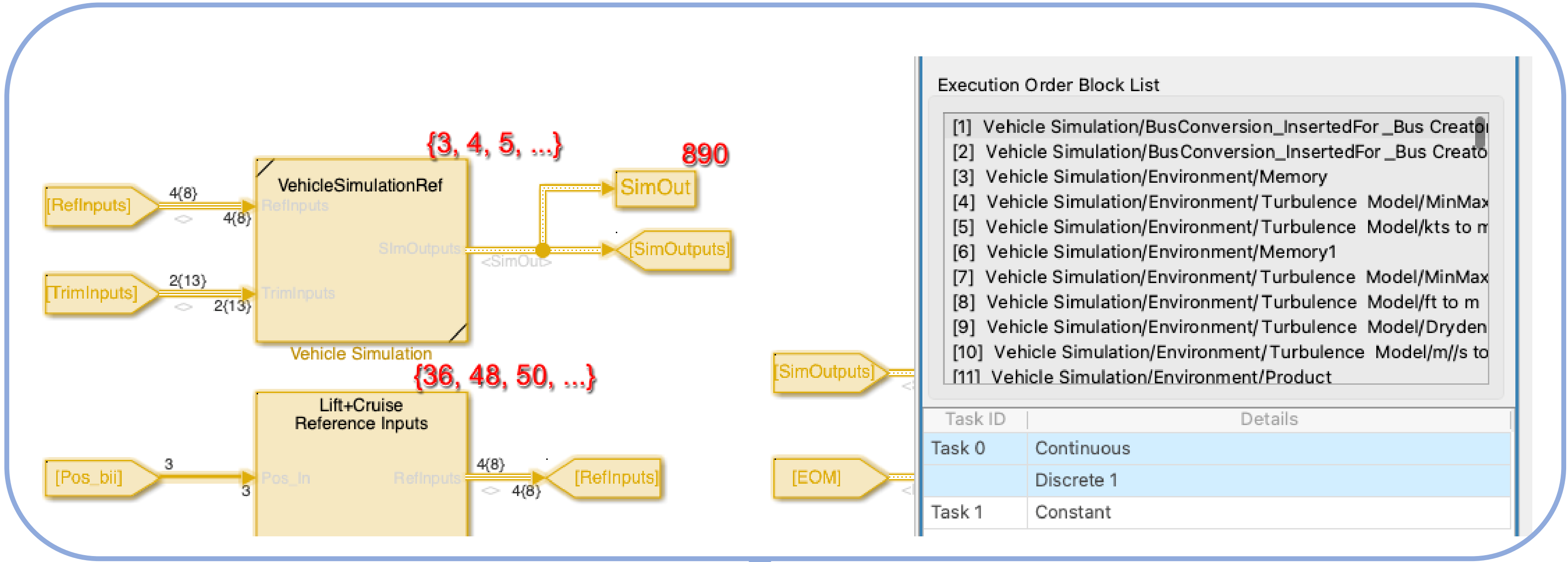
Implementation and testing

- ➢ Translation example: (2) Hover to Transition Timeseries

- ➢ Some important implementation details including:
  - Execution order
  - Integration function
  - Function without direct python equivalent

- ➢ Debugging and result comparison

Execution order



➢ Based on the execution order information we found in ‘Information Overlays’, we mapped our JAX pipeline with exact order.

```
def deriv(self, state: GuamState, ref_inputs: RefInputs) -> GuamState:
    ref_inputs.assert_shapes()

    sensor, aeroprop_body_data, alt_msl = self.veh_eom.get_sensor_aeroprop_altmsl(state.aircraft)
    atmosphere = self._environment.get_env_atmosphere(alt_msl)
    env_data = self.env_data._replace(Atmosphere=atmosphere)
    control, cache = self.controller.get_control(state.controller, sensor, ref_inputs)

    d_state_controller = self.controller.state_deriv(cache)
    pwr_cmd = PwrCmd(CtrlSurfacePwr=control.Cmd.CtrlSurfacePwr, EnginePwr=control.Cmd.EnginePwr)
    power = power_system(pwr_cmd)

    surf_act, prop_act = self.surf_eng.get_surf_prop_act(state.surf_eng, control.Cmd, power)
    d_state_surf_eng = self.surf_eng.surf_engine_state_deriv(control.Cmd, state.surf_eng)

    fm = self.aero_prop.aero_prop(prop_act, surf_act, env_data, aeroprop_body_data)
    fm_total = self.veh_eom.get_fm_with_gravity(state.aircraft, fm)
    d_state_aircraft = self.veh_eom.state_deriv(fm_total, state.aircraft)

    return GuamState(d_state_controller, d_state_aircraft, d_state_surf_eng)
```

Integration

- We used *scipy.integrate.solve_ivp* for integration.

- Since GUAM Simulink is using ode3, which is Bogacki-Shampine, we use *extensisq*, a package that extends *scipy.integrate* that supports Bogacki-Shampine.

```
sol = solve_ivp(deriv_fn_wrapped, [0.0, dt], state13, t_eval=[dt], method=BS5)
```

Function without direct python equivalent

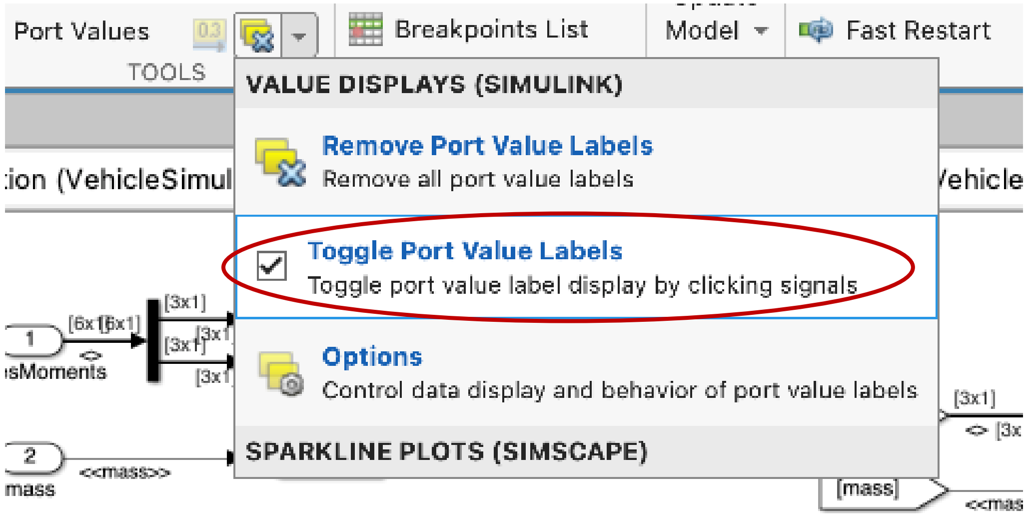
- The *mkpp*, *unmkpp*, and *ppval* functions in MATLAB do not have exact python equivalent, so we created translation by digitizing data with given breaks and calculating piecewise polynomial manually.
- Similarly, we also created functions to match Simulink blocks such as matrix interpolation.

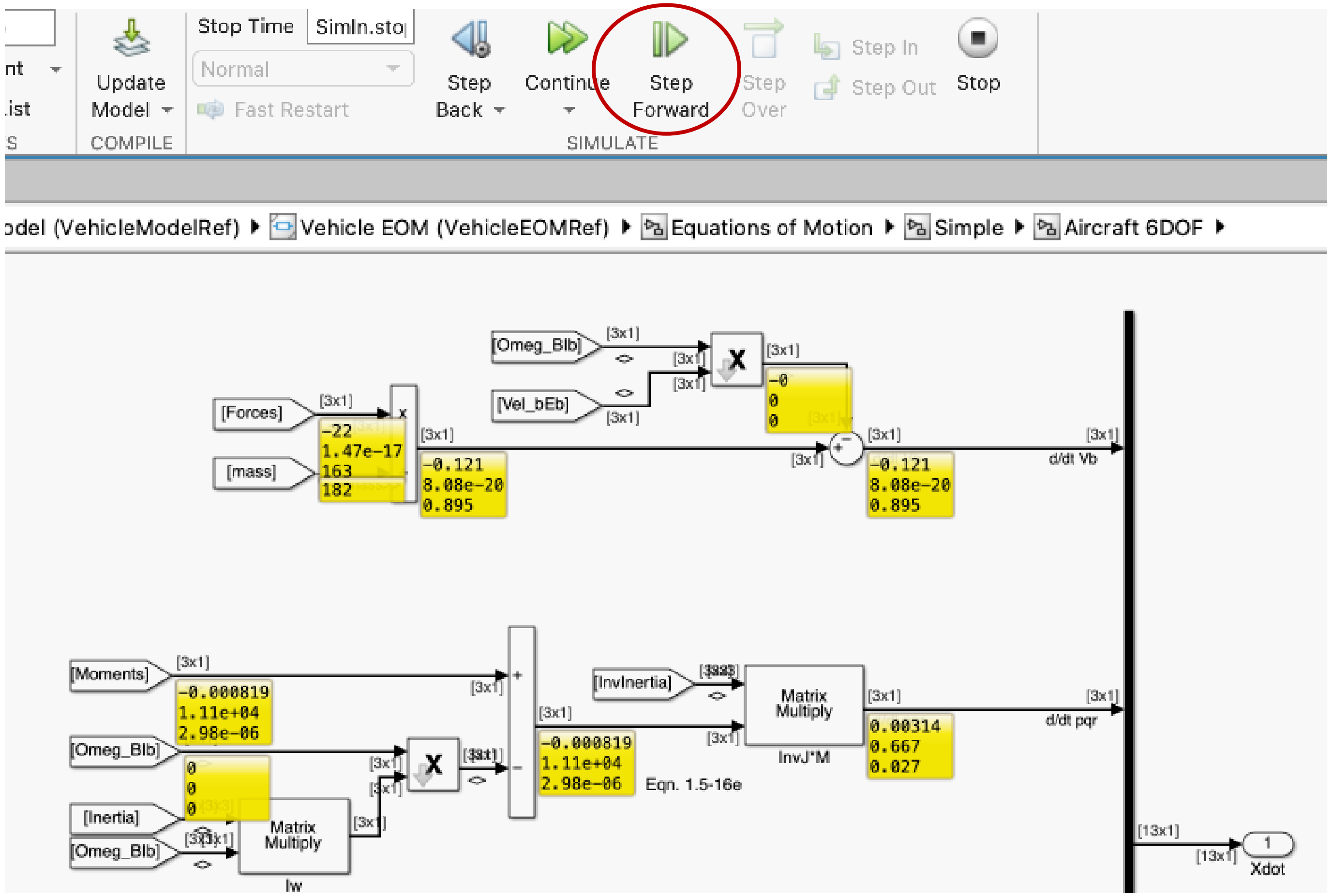
```
% drag coefficient
%   need to use unmkpp/mkpp for codegen
[pp_brk, pp_coef, pp_L, pp_order, pp_dim] = unmkpp(aero_coefs_pp.cd);
pp_mk = mkpp(pp_brk, pp_coef);
cd = ppval(pp_mk, alff);

% outputs
y = cd;
```

```
n_strips = len(x)
assert x.shape == (n_strips, 6) and alff.shape == (n_strips, 1)

breaks = aero_coefs_pp[0][0][2][0][0][1]
assert breaks.shape == (1, 541)

coeffs = aero_coefs_pp[0][0][2][0][0][2]
assert coeffs.shape == (540, 4)

inds = jnp.digitize(alff, breaks[0], right=True) - 1
inds = jnp.clip(inds, 0, len(coeffs) - 1)
assert inds.shape == (n_strips, 1)

coeffs = jnp.array(coeffs)[inds]
assert coeffs.shape == (n_strips, 1, 4)

alff_min_breaks = alff - jnp.array(breaks)[:, inds]
assert alff_min_breaks.shape == (1, n_strips, 1)

y = (
    coeffs[:, :, 0] * alff_min_breaks ** 3
    + coeffs[:, :, 1] * alff_min_breaks ** 2
    + coeffs[:, :, 2] * alff_min_breaks ** 1
    + coeffs[:, :, 3]
)
assert y.shape == (1, n_strips, 1)
```

Debugging

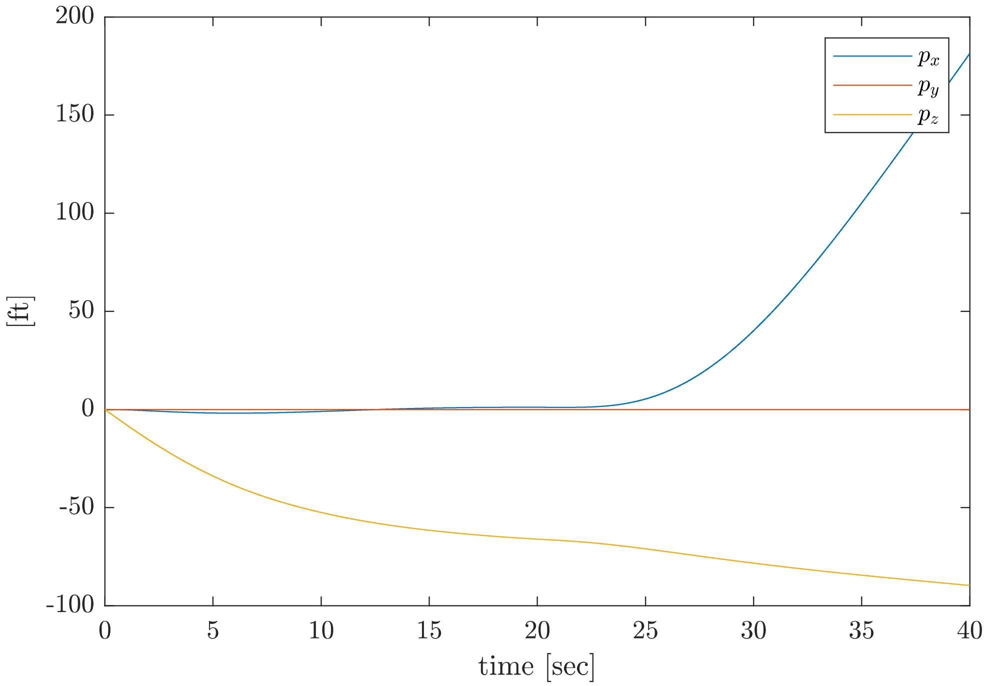
- We started debugging and testing for every single function by enabling 'Toggle port value labels' and step forward step by step to check if values match.



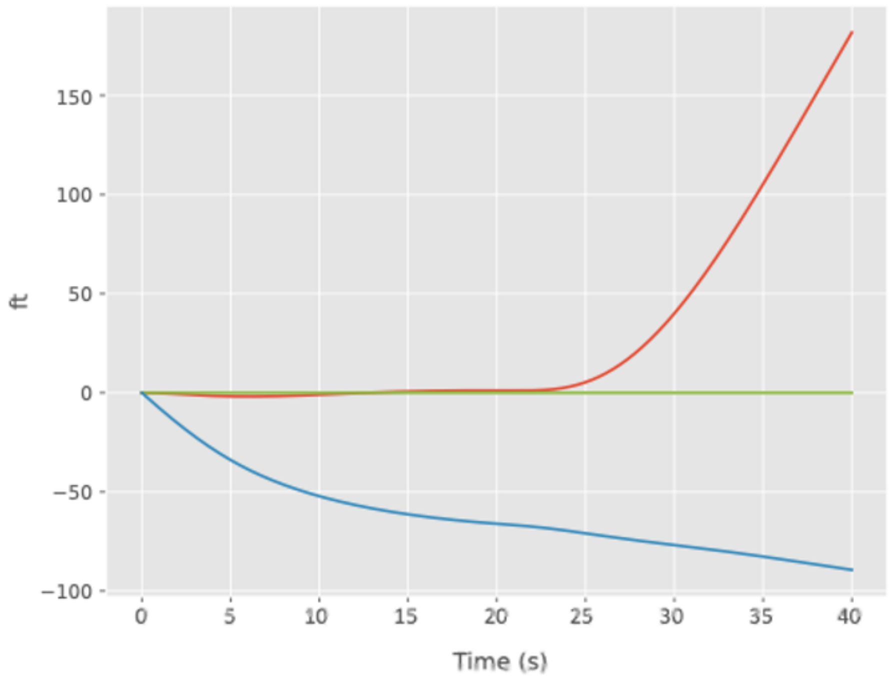


Result comparison

- We compared our result with *SimOut* struct output of Simulink model.
- Here is a result comparison for [*SimOut.Vehicle.EOM.InertialData.Pos_bii*]() field:



GUAM Simulink



GUAM JAX

Trajectories of a learning-based controller on the JAX GUAM