PyOED: An Open Source, Backend-Agnostic, Bayesian OED Toolbox for Rapid Development

Abhijit Chowdhary[1], Shady Ahmed[2], Ahmed Attia[3]

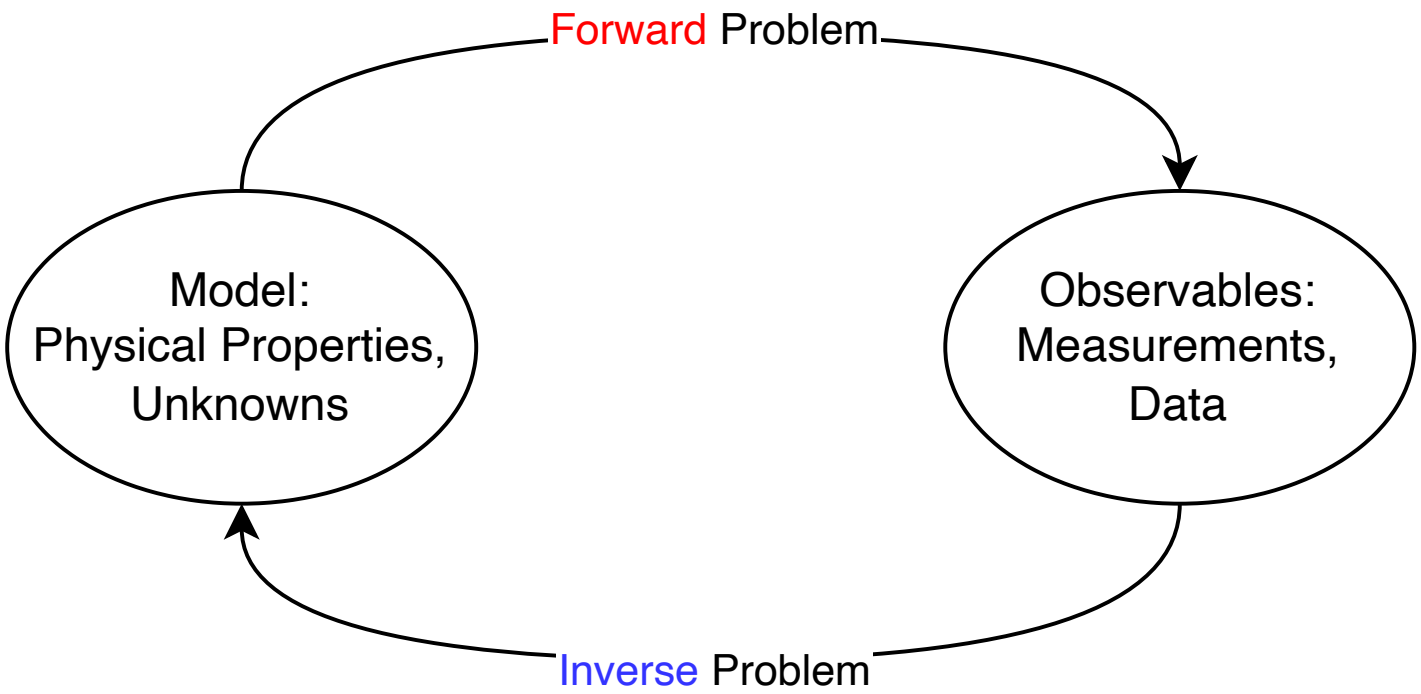
[1]North Carolina State University

[3]Pacific Northwest National Laboratory

[2]Argonne National Laboratory

July 29, 2024

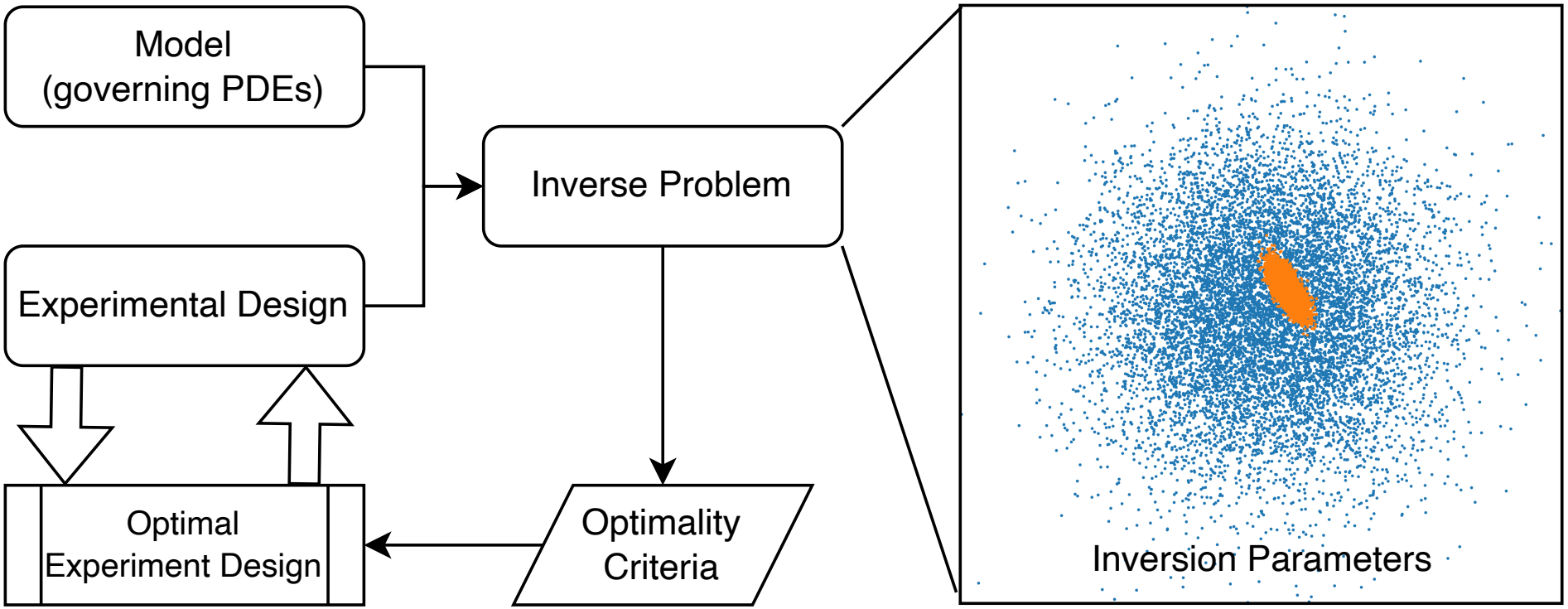
Introduction: Motivation



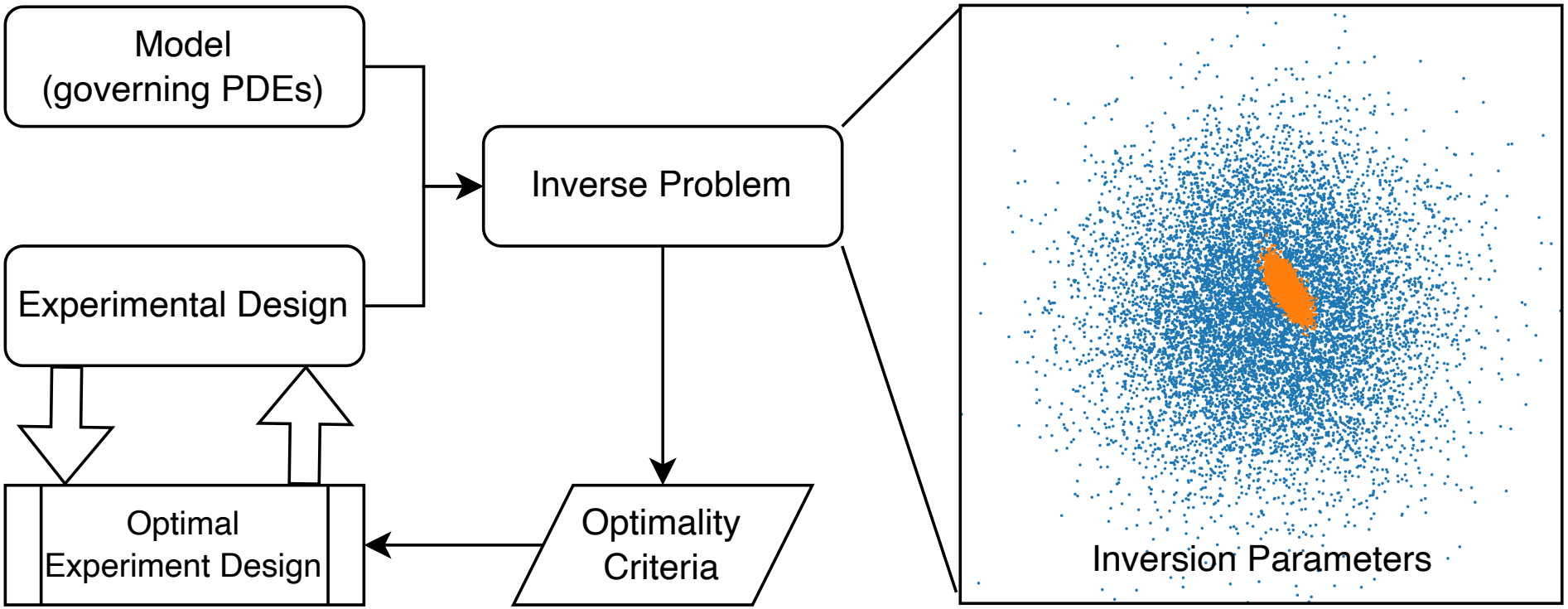
Applications

1. Fault-slip inference
2. Contaminant-source identification
3. Permeability field inversion in porous medium flow
4. Epidemic model calibration

Optimal Experimental Design of Bayesian Inverse Problems



Optimal Experimental Design of Bayesian Inverse Problems



This is hard!

PyOED in a Nutshell

PyOED

Vision

Enable the rapid development and testing of model-constrained OED problems.

1. Provide a tested framework for OED
2. Use a simple interface to minimize developer friction
3. Ensure each component is extensible and/or optional

PyOED in a Nutshell

PyOED

Vision

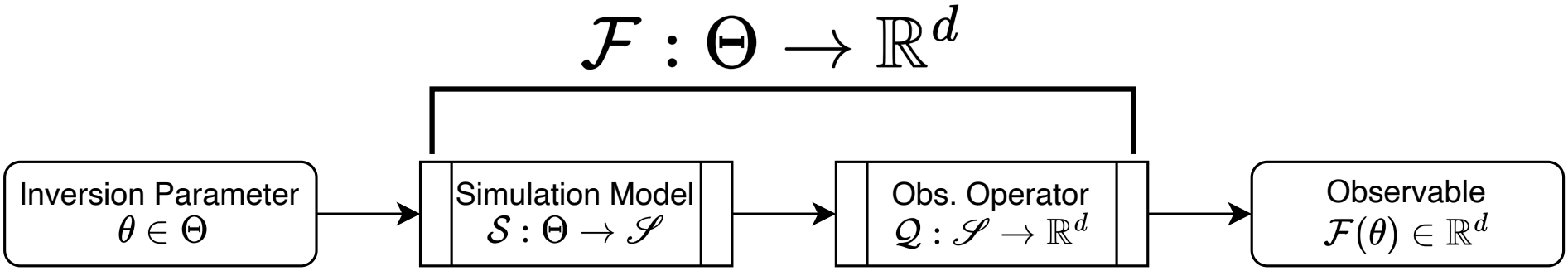
Enable the rapid development and testing of model-constrained OED problems.

1. Provide a tested framework for OED
2. Use a simple interface to minimize developer friction
3. Ensure each component is extensible and/or optional

Features

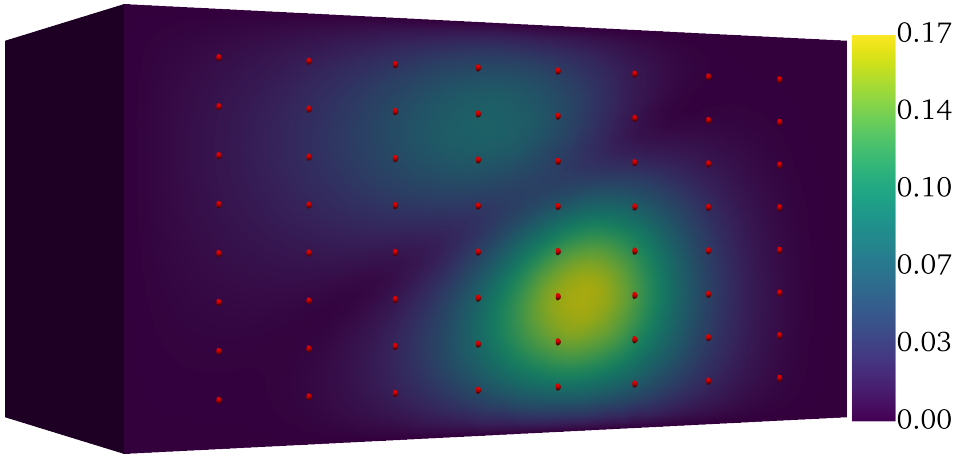
1. Implementations of classical and new OED algorithms
2. Flexible assimilation subsystem
3. Multiple standard simulation models built in & flexibility to use your own

Governing Equations



Definition

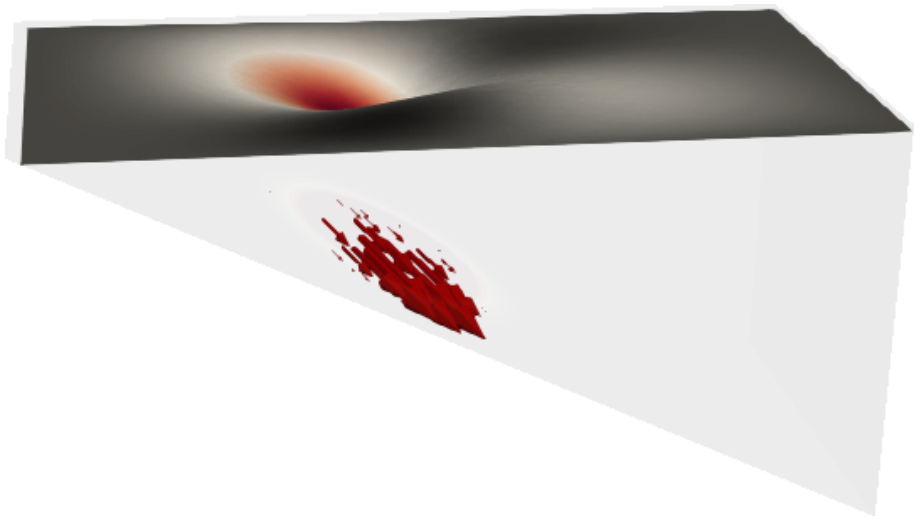
We refer to the map $\mathcal{F} : \Theta \to \mathbb{R}^d$ as the parameter to observable map.



Simulation Model

$$\mathcal{S} : \Theta \to \mathscr{S}$$

- Built-in
  - Toy Models
  - PDEs (Advection-Diffusion, Poisson, Bateman-Burgers, etc.)
  - Imaging
- Bring your own model
  - FEniCS(x)
  - Jax / PyTorch



Both the forward and adjoint (Jacobian transpose) action must be supplied!

Simulation Model: Example Structure

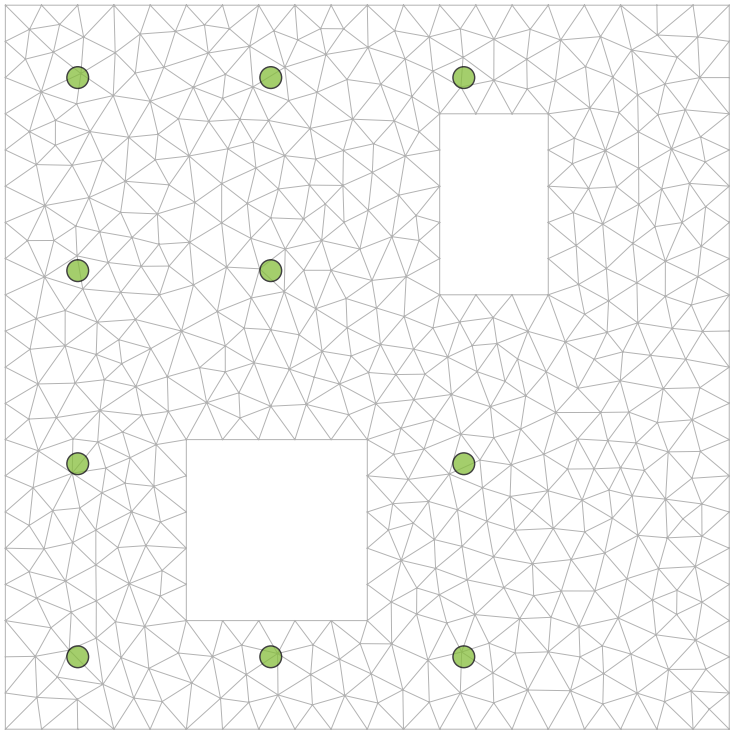
```python
class TimeIndependentModel(SimulationModel):

    @abstractmethod
    def state_vector(self, init_val=0, **kwargs):
        ...

    @abstractmethod
    def parameter_vector(self, init_val=0, **kwargs):
        ...

    @abstractmethod
    def solve_forward(self, state, verbose=False):
        ...

    def solve_adjoint(self, adjoint):
        ...

    def Jacobian_T_matvec(self, state, eval_at):
        ...

```

```python
model = ToyLinearTimeIndependent({'nx':5, 'np':5}) # S(theta) = S * theta

```

Observation Operator

$$\mathcal{B} : \mathscr{S} \rightarrow \mathbb{R}^d$$

- Identity
- Selection
- Interpolation
- FEniCS Compatible
- Time Dependent

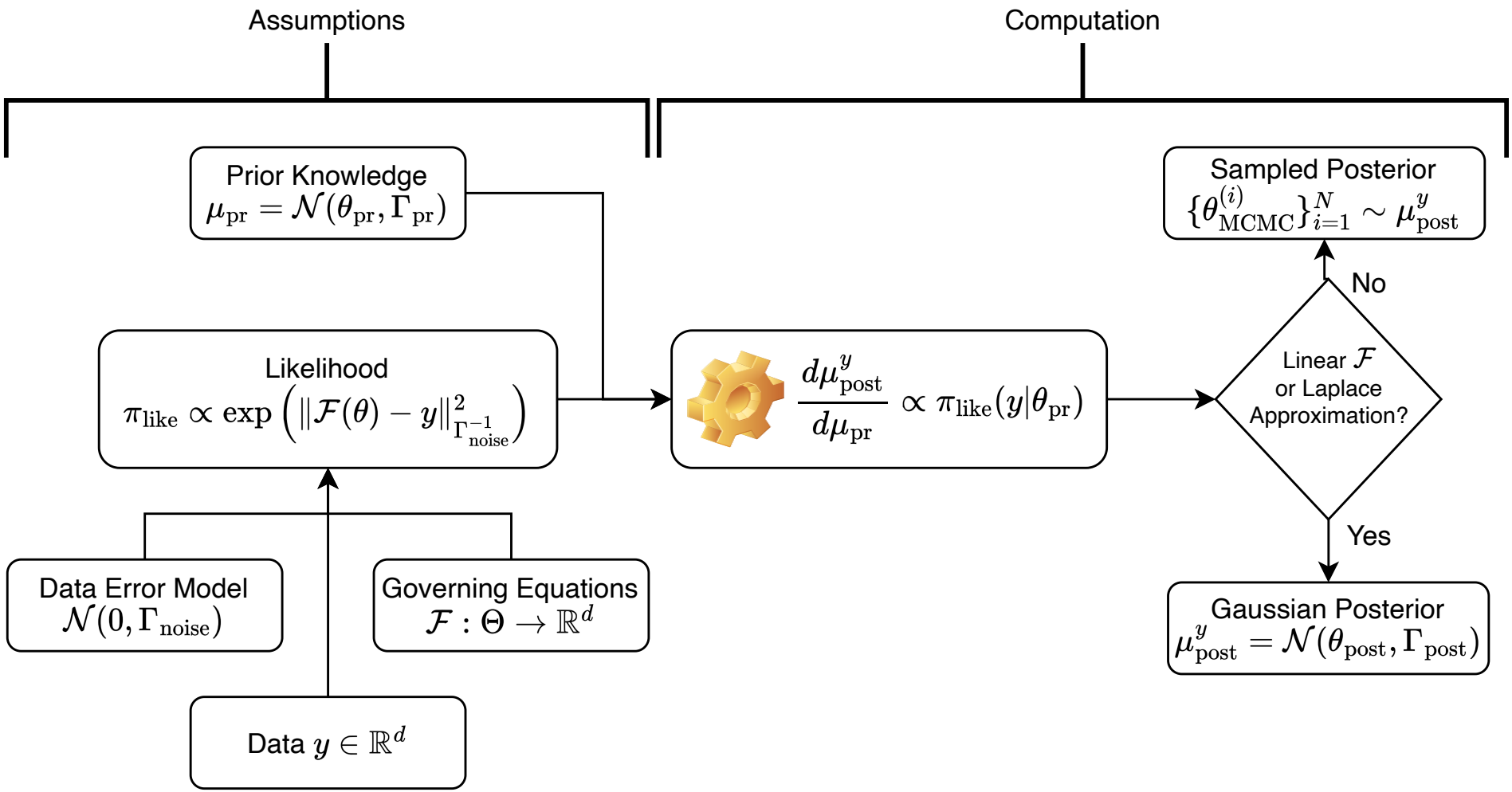


```
obs_oper = Identity({'model':model})
```

Bayesian Inverse Problem Framework

Goal

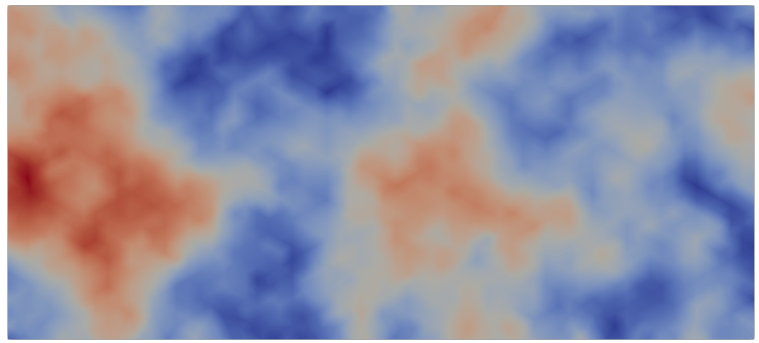
Infer a distribution for θ encoding data and prior information.



Observation Error and Prior Distributions

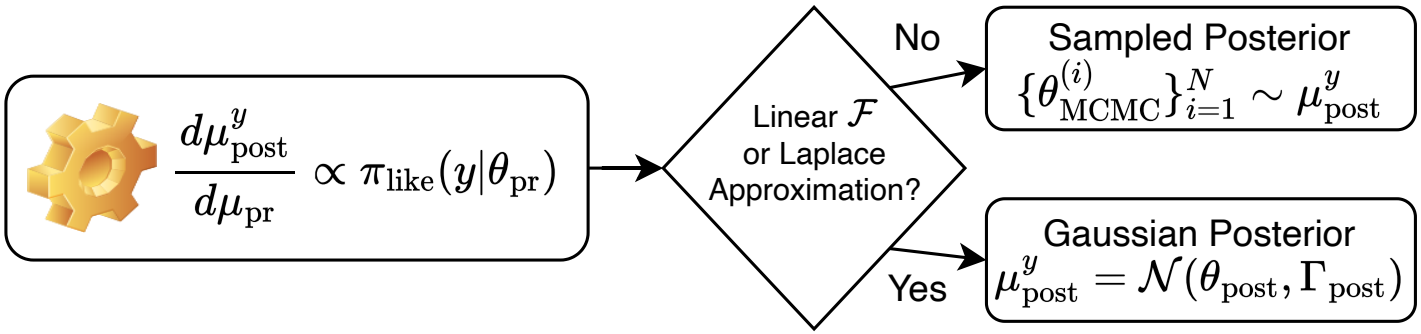
Supported Gaussian Error Models

- $\mathcal{N}(\mu, \Sigma)$
- Gaussian with (Bi-)Laplacian Covariance
- Complex Gaussian



```
obs_noise = GaussianErrorModel({'size':obs_oper.shape[0], 'variance':1e-3})
prior = GaussianErrorModel({'size':model.state_size, 'mean':0, 'variance':1})

```

Inverse Problem

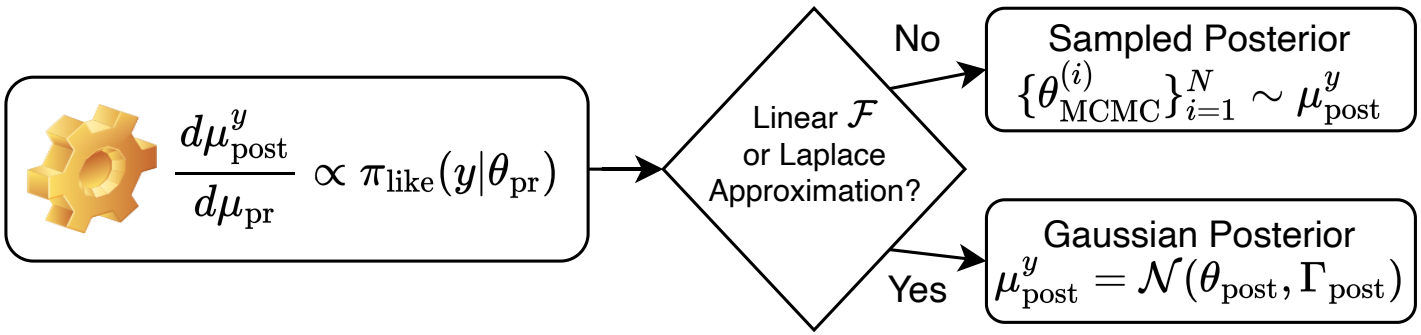


Posterior is represented by a Gaussian with covariance determined by

- Closed form expression (linear models)
- Laplace + Gauss-Newton Approximation (non-linear models)

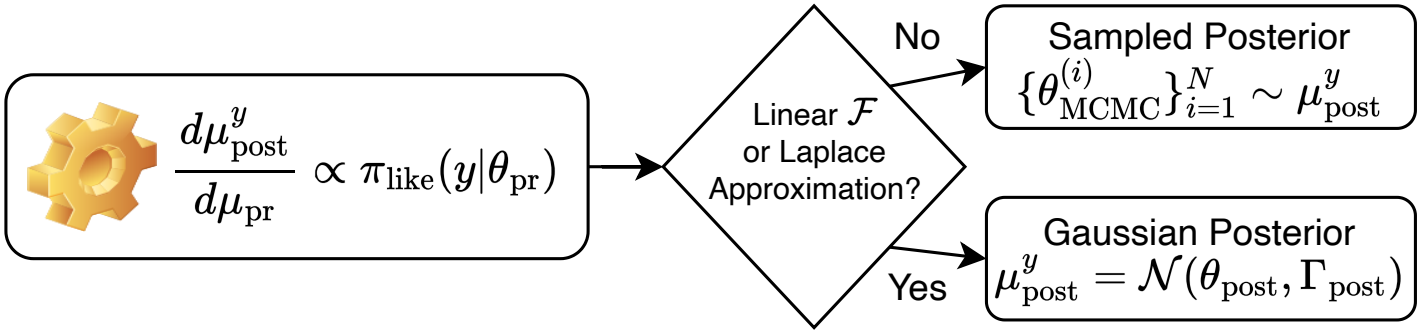
We also have MCMC (Hamiltonian).

Inverse Problem



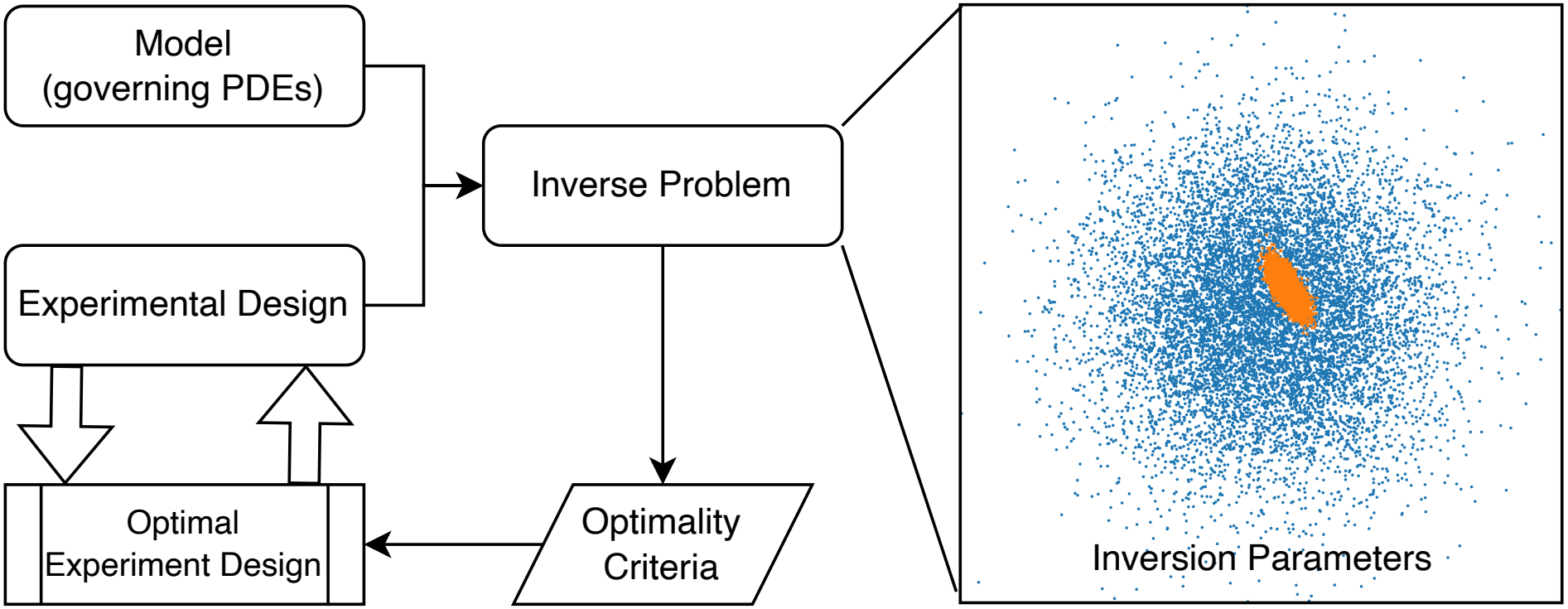

```python
class Smoother(ABC):

    @_abstractmethod
    def solve_inverse_problem(self):
        ...

    @_abstractmethod
    def apply_forward_operator(self, *args, **kwargs):
        ...

    def apply_forward_operator_adjoint(self, *args, **kwargs):
        ...

```

Inverse Problem




```
1  inverse_problem = VanillaThreeDVar()
2
3  inverse_problem.register_model(model)
4  inverse_problem.register_observation_operator(obs_oper)
5  inverse_problem.register_prior_model(prior)
6  inverse_problem.register_observation_error_model(obs_noise)
7
8  observations = ... # Acquire data
9  inverse_problem.register_observation(observations)
10
11 inverse_problem.solve_inverse_problem(update_posterior=True)
12
```

Optimal Experimental Design of Bayesian Inverse Problems



OED Problem

Let ζ be a binary vector that parametrizes the experiment. Then we seek:

$$\zeta^{\text{opt}} = \underset{\zeta \in \{0,1\}^d}{\arg\max}\ \mathcal{U}(\zeta)$$

Optimal Experimental Design: Utility Criteria

Built-in Options

Exact and randomized formulations available where applicable!

- Alphabetic Criteria
  - A-Optimal
  - D-Optimal
- Information-Theoric Criteria
  - (Expected) Information Gain

```
1  class UtilityFunction(ABC):
2
3      @abstractmethod
4      def evaluate(self, *args, **kwargs):
5          pass
6
```

Optimal Experimental Design

Relaxed OED:

$$\zeta^{\text{opt}} = \underset{\zeta \in [0,1]^d}{\arg\max}\, \mathcal{U}(\zeta)$$

Stochastic OED:

$$\mathbf{p}^{\text{opt}} = \underset{\mathbf{p} \in [0,1]^d}{\arg\max}\, \mathbb{E}_{\zeta \sim \mathbb{P}(\zeta|\mathbf{p})} \left[\mathcal{U}(\zeta)\right]$$

```
oed_problem = SensorPlacementBinaryOED(
    configs=dict(
        inverse_problem=inverse_problem,
        criterion='D-opt',
        use_FIM=True,
        optimization_settings={...}
    )
)

```

Optimal Experimental Design

Relaxed OED:

$$\zeta^{\text{opt}} = \underset{\zeta \in [0,1]^d}{\arg\max}\, \mathcal{U}(\zeta)$$

Stochastic OED:

$$\mathbf{p}^{\text{opt}} = \underset{\mathbf{p} \in [0,1]^d}{\arg\max}\, \mathbb{E}_{\zeta \sim \mathbb{P}(\zeta|\mathbf{p})} \left[\mathcal{U}(\zeta)\right]$$

```
oed_problem = SensorPlacementBinaryOED(
    configs=dict(
        inverse_problem=inverse_problem,
        criterion='D-opt',
        use_FIM=True,
        optimization_settings={...}
    )
)

```

```
budget = 2
oed_problem.register_penalty_term(
    penalty_weight=-20,
    penalty_function=lambda design: (np.sum(design)-budget)**2,
)

```

Optimal Experimental Design

Relaxed OED:

$$\zeta^{\text{opt}} = \underset{\zeta \in [0,1]^d}{\arg\max}\, \mathcal{U}(\zeta)$$

Stochastic OED:

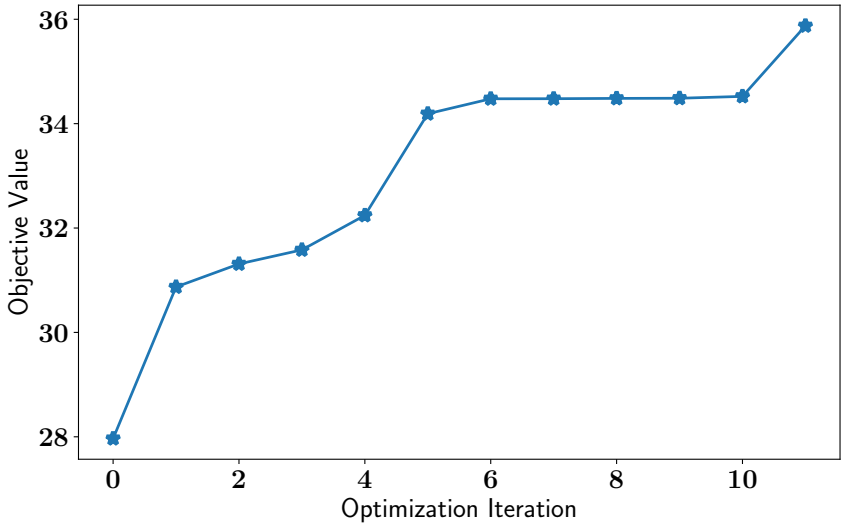
$$\mathbf{p}^{\text{opt}} = \underset{\mathbf{p} \in [0,1]^d}{\arg\max}\, \mathbb{E}_{\zeta \sim \mathbb{P}(\zeta|\mathbf{p})} \left[\mathcal{U}(\zeta)\right]$$

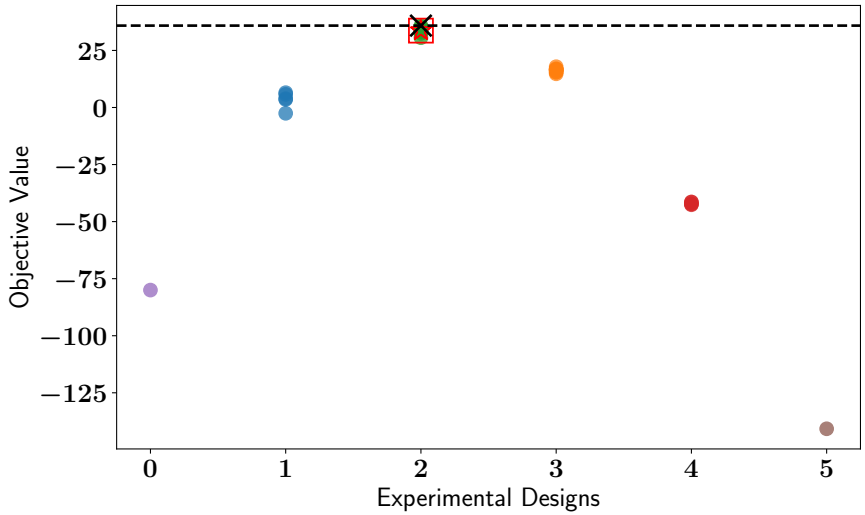
```
oed_problem = SensorPlacementBinaryOED(
    configs=dict(
        inverse_problem=inverse_problem,
        criterion='D-opt',
        use_FIM=True,
        optimization_settings={...}
    )
)

```

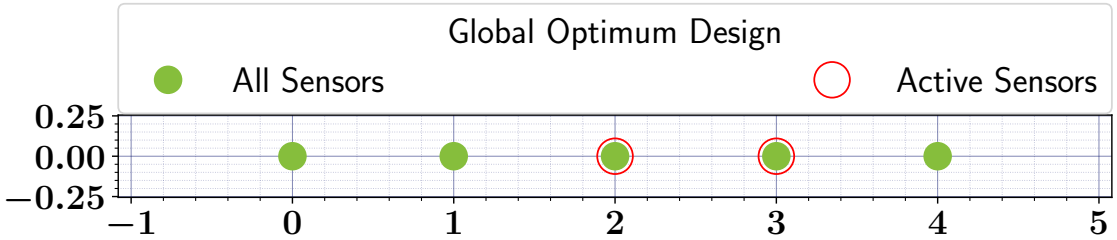
```
budget = 2
oed_problem.register_penalty_term(
    penalty_weight=-20,
    penalty_function=lambda design: (np.sum(design)-budget)**2,
)

```

```
designs = itertools.product([0, 1], repeat=oed_problem.design_size)
obj_vals = map(oed_problem.oed_objective, designs)
```

And that's OED in 25 lines!

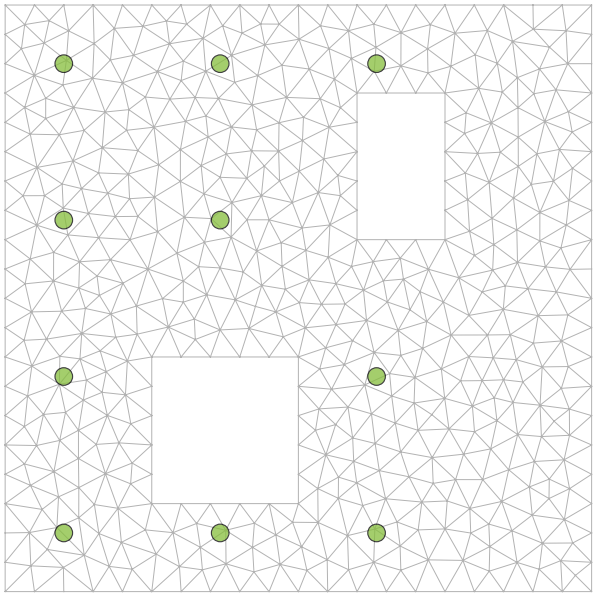


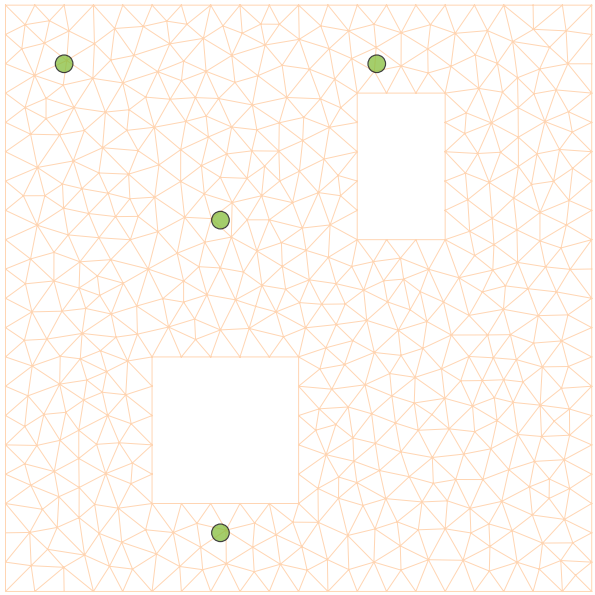


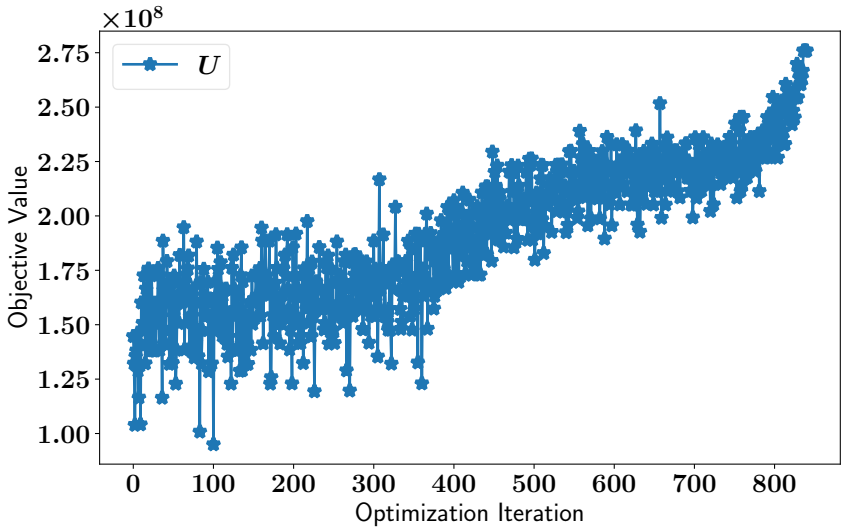


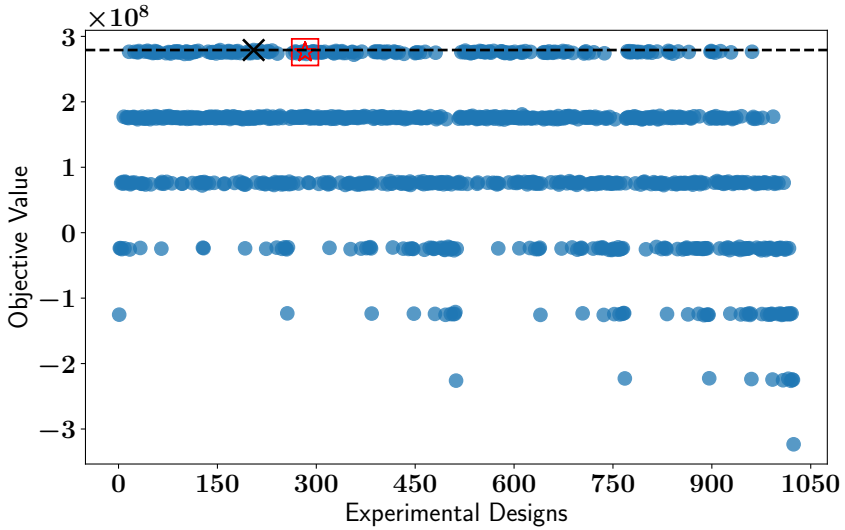

```
stochastic_oed_results = oed_problem.solve_oed_problem()
stochastic_oed_results.update_bruteforce_results()
stochastic_oed_results.plot_results()

```

Advection-Diffusion Experiment









Conclusion

- Source
  - `https://gitlab.com/ahmedattia/pyoed/`
- Feature / Bug Tracker
  - `https://gitlab.com/ahmedattia/pyoed/-/issues`
- Documentation
  - `https://web.cels.anl.gov/~aattia/pyoed`
- Next Steps
  - Publication (article in review)
  - Expanded DA and OED formulations
  - Performance Effort

```
$ pip install git+https://gitlab.com/ahmedattia/pyoed.git
```

Paper in review

Chowdhary, A., Ahmed, S. E., & Attia, A. (2023). PyOED: An extensible suite for data assimilation and model-constrained optimal design of experiments (arXiv:2301.08336). arXiv.