

PySensors 2.0: A Python Package for Sparse Sensor Placement

Niharika Karnik^{*1}, Yash Bhangale^{†1}, Mohammad G. Abdo², Andrei A. Klishin³,
Joshua J. Cogliati², Bingni W. Brunton⁶, J. Nathan Kutz⁷, Steven L. Brunton¹, and
Krithika Manohar^{‡1}

¹*Department of Mechanical Engineering, University of Washington*

²*Idaho National Laboratory*

³*Department of Mechanical Engineering, University of Hawai'i at Mānoa*

⁶*Department of Biology, University of Washington*

⁷*Department of Applied Mathematics, University of Washington*

Abstract

PySensors is a Python package for selecting and placing a sparse set of sensors for reconstruction and classification tasks. In this major update to **PySensors**, we introduce spatially constrained sensor placement capabilities, allowing users to enforce constraints such as maximum or exact sensor counts in specific regions, incorporate predetermined sensor locations, and maintain minimum distances between sensors. We extend functionality to support custom basis inputs, enabling integration of any data-driven or spectral basis. We also propose a thermodynamic approach that goes beyond a single “optimal” sensor configuration and maps the complete landscape of sensor interactions induced by the training data. This comprehensive view facilitates integration with external selection criteria and enables assessment of sensor replacement impacts. The new optimization technique also accounts for over- and under-sampling of sensors, utilizing a regularized least squares approach for robust reconstruction. Additionally, we incorporate noise-induced uncertainty quantification of the estimation error and provide visual uncertainty heat maps to guide deployment decisions. To highlight these additions, we provide a brief description of the mathematical algorithms and theory underlying these new capabilities. We demonstrate the usage of new features with illustrative code examples and include practical advice for implementation across various application domains. Finally, we outline a roadmap of potential extensions to further enhance the package’s functionality and applicability to emerging sensing challenges.

1 Introduction

Sensor placement is critical for efficient monitoring, control, and decision-making in modern engineering systems. Sensors play a crucial role in characterizing spatio-temporal dynamics in high-dimensional, non-linear systems such as fluid flows [1], manufacturing [2], geophysical [3] and nuclear systems [4]. Optimal sensor placement ensures accurate, real-time tracking of key system variables with minimal hardware and enables cost-effective, real-time system analysis and control. In general, sensor placement optimization is NP-hard and cannot be solved in polynomial time. There are $\binom{n}{p} = n!/((n-p)p!)$ possible combinations of choosing p sensors from an n -dimensional state. Common approaches to optimizing sensor placement include maximizing the information criteria [5], Bayesian Optimal Experimental Design [6], compressed sensing [7], and heuristic methods. Many sensor placement methods have submodular objective form, which sets guarantees on how close an efficient greedy placement can be to the unknown true optimum [8]. Sub-modular objectives can be efficiently optimized for hundreds or thousands of candidate locations using convex [9] or greedy optimization approaches [8].

PySensors is a Python package [10] dedicated to solving the complex challenge of optimal sensor placement in data-driven systems. It implements advanced sparse optimization algorithms that use dimensionality reduction techniques to identify the most informative measurement locations with

^{*}nkarnik@uw.edu

[†]yash6599@uw.edu

[‡]kmanohar@uw.edu

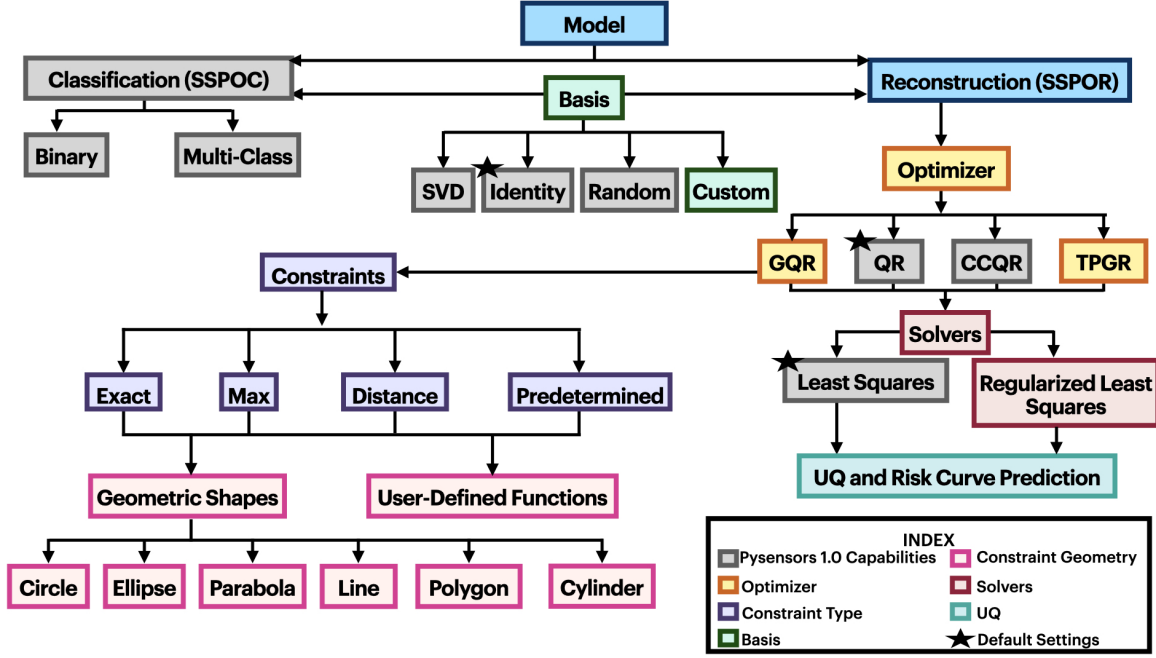


Figure 1: PySensors 2.0 expands its capabilities by introducing custom basis functions, optimizers, constraints, solvers, and uncertainty quantification, enabling constrained sensing, over- and under-sampling, and uncertainty quantification in the presence of noisy sensor measurements.

remarkable efficiency [11], [12], [13]. It helps users identify the best locations for sensors when working with complex high dimensional data, focusing on both reconstruction [11] and classification [12] tasks. The package follows `scikit-learn` conventions for user-friendly access while offering advanced customization options for experienced users. Designed with researchers and practitioners in mind, PySensors provides open-source, accessible tools that support model discovery across various scientific applications.

This new version of PySensors focuses specifically on practical engineering applications where measurement data is inherently noisy and spatial deployment constraints are unavoidable. Key improvements include constraint-aware optimization methods that handle spatial restrictions and sensor density limitations. In addition, the framework introduces uncertainty quantification metrics that track how measurement noise propagates through reconstruction algorithms, enabling robust error estimation in sensor outputs. This version of PySensors implements methodologies introduced by Klishin et al. [14] and Karnik et al. [4] to make them accessible to scientists and engineers in all domains. These enhancements transform PySensors from a purely academic tool into a practical platform for solving real-world sensing challenges while maintaining mathematical rigor.

Other sensor placement packages such as Chama [15], Polire [16], and OSPS toolbox [17], focus primarily on event detection, Gaussian process modeling, and structural health monitoring respectively, while PySensors specifically targets signal reconstruction and classification applications. PySensors 2.0 represents an advancement in sensor optimization software, establishing a distinct position in the field through its novel focus on constrained optimization and uncertainty quantification as shown in Figure 1.

In this work, we first establish the theoretical foundation of sensor placement optimization by examining mathematical techniques for signal reconstruction and classification, while introducing the dimensionality reduction approaches and optimization algorithms that underpin effective sensor selection. We then explore the novel functionalities implemented in PySensors 2.0, including two newly developed optimizers for hard spatial constraints for realistic deployment scenarios, adaptive strategies for sensor oversampling and undersampling, and support for custom basis functions. To demonstrate practical applications, we present a detailed case study applying our constraint-aware optimization framework to a nuclear fuel rod prototype, complete with uncertainty quantification heatmaps and estimation error metrics that provide crucial reliability insights. The paper concludes with practical

implementation guidelines to help users maximize the effectiveness of these advanced sensor placement tools across diverse application domains.

2 Background

This section first describes the two main objectives of **PySensors**: reconstruction and classification, and their implementation using basis functions and optimization techniques for sensor placement.

2.1 Reconstruction

PySensors implements **Sparse Sensor Placement Optimization for Reconstruction (SSPOR)** [11] of full fields $\mathbf{x} \in \mathbb{R}^n$ from p noise-corrupted sensor measurements $\mathbf{y} \in \mathbb{R}^p$

$$\mathbf{y} = \mathbb{S}\mathbf{x} + \boldsymbol{\eta}, \quad (1)$$

where $\boldsymbol{\eta}$ consists of zero-mean, Gaussian independent and identically distributed (i.i.d.) components, and $\mathbb{S} \in \mathbb{R}^{p \times n}$ is the desired sensor (measurement) selection operator. This measurement selection operator \mathbb{S} encodes point measurements with unit entries in a sparse matrix

$$\mathbb{S} = [\mathbf{e}_{\gamma_1} \quad \mathbf{e}_{\gamma_2} \quad \dots \quad \mathbf{e}_{\gamma_p}]^T, \quad (2)$$

where \mathbf{e}_j are canonical basis vectors for \mathbb{R}^n , with a unit entry in component j (where a sensor should be placed) and zeroes elsewhere. Here, $\gamma = \{\gamma_1, \gamma_2, \dots, \gamma_p\} \subset \{1, 2, \dots, n\}$ denotes the index set of sensor locations with cardinality p . Sensor selection then corresponds to the components of \mathbf{x} that were chosen to be measured:

$$\mathbb{S}\mathbf{x} = [x_{\gamma_1} \quad x_{\gamma_2} \quad \dots \quad x_{\gamma_p}]^T. \quad (3)$$

The **SSPOR** class selects these sensors through a cohesive computational framework for strategically minimizing sensor deployment while maintaining reconstruction fidelity in high-dimensional systems. When initialized with measurement data $\mathbf{x} \in \mathbb{R}^n$, **SSPOR** first applies dimensional reduction through basis identification, then employs the computationally efficient QR factorization algorithm to determine optimal sensor locations. This approach, which has demonstrated efficacy in both reduced-order modeling and sparse sensing applications, strategically leverages the inherent structure of the identified basis to prioritize the most informative measurement points.

2.2 Classification

The **Sparse Sensor Placement Optimization for Classification (SSPOC)** framework identifies minimal sensor configurations that can classify high dimensional signals $\mathbf{x} \in \mathbb{R}^n$ as one of c classes. Unlike traditional compressed sensing [7], [18], [19] approaches that focus on signal reconstruction, **SSPOC** specifically targets the preservation of classification boundaries in feature space Ψ_r by identifying the sparsest sensor set capable of reconstructing discriminant hyperplanes.

The **SSPOC** architecture accepts any combination of basis representation and linear classifier, defaulting to Linear Discriminant Analysis (LDA) and Identity basis when not otherwise specified. The optimization pipeline proceeds through multiple stages: dimensional reduction via basis fitting, classifier training within the reduced space, sparse optimization incorporating both classifier weights and basis structures, sensor selection based on optimization outputs, and optional classifier retraining using only measurements from the selected locations.

The framework adapts its optimization strategy according to classification complexity. For binary problems, it employs Orthogonal Matching Pursuit from the **scikit-learn** library, while multi-class scenarios trigger multi-task Lasso regularization. This mathematical flexibility enables the framework to address diverse classification challenges while maintaining computational efficiency and sparse sensing requirements.

2.3 Basis

High dimensional field dynamics $\mathbf{x} \in \mathbb{R}^n$ can be represented as a linear combination of spatial basis modes Ψ weighted by time-varying coefficients \mathbf{a}

$$\mathbf{x} = \Psi_r \mathbf{a}.$$

This basis, which can be built from spectral or data-driven decomposition methods, is typically chosen so that the embedding dimension is as small as possible, i.e., $r \ll n$. Different basis functions can significantly impact sensor selection effectiveness and reconstruction quality [11]. PySensors offers several interchangeable **basis** options for sparse sensor selection:

1. **Identity**: Processes raw measurement data directly without transformation.
2. **SVD**: Utilizes truncated singular value decomposition's $\mathbf{X} = \mathbf{U}_r \Sigma_r \mathbf{V}_r^*$ left singular vectors \mathbf{U}_r , computing only the specified number of modes to reduce computational demands. A randomized SVD option further enhances efficiency.
3. **RandomProjection**: Projects measurements into a new space by multiplying them with random Gaussian vectors, connecting to compressed sensing methodologies from established literature [7], [18], [19].
4. **Custom**: This is a new option included in **PySensors 2.0** that enables users to leverage custom basis functions beyond **PySensors'** built-in options. Researchers can transform their data into an alternative basis such as dynamic mode decomposition modes [20], before passing it to a PySensors instance configured with the Custom basis.

2.4 Optimizers

When a user specifies p sensors, SSPOR first fits a basis Ψ_r to the data and optimizes sensor placement by minimizing the following objective function:

$$\gamma_* = \underset{\gamma, |\gamma|=p}{\operatorname{argmax}} \log \det((\mathbf{S}\Psi_r)^T (\mathbf{S}\Psi_r)). \quad (4)$$

where γ_* denotes the index set of optimized sensor locations with cardinality p . When $p = r$, Equation 4 is equivalent to the maximizer of $\log |\det(\mathbf{S}\Psi_r)|$. Direct optimization of this criterion leads to a brute force combinatorial search. This sensor placement approach builds upon the empirical interpolation method (EIM) [21] and discrete empirical interpolation method (DEIM) [22] to develop a greedy strategy for optimizing sensor selection built upon the pivoted QR factorization [2], [11], [23], [24].

PySensors implements the **QR** optimizer for optimal sensor selection in unconstrained scenarios where the number of sensors p equals the number of modes r . The framework incorporates heterogeneous cost functions into the optimization process to accommodate practical deployment constraints. For example, when monitoring oceanographic parameters, the system can account for the substantially higher costs of deep-sea sensors relative to coastal installations [13]. This capability is implemented through the **Cost-Constrained QR (CCQR)** algorithm in the optimizers submodule, allowing users to balance information capture against resource limitations when designing sensor networks for complex physical systems.

Traditional QR factorization presents challenges in under-sampling $p < r$ and over-sampling $p > r$ scenarios, as well as when spatial constraints must be considered. **PySensors 2.0** addresses these limitations through two new optimization algorithms: **Generalized QR (GQR)** and **Two Point Greedy (TPGR)**.

3 New Functionality

PySensors 2.0 has been enhanced to address critical challenges in sensor placement by incorporating spatial constraints, noise-induced uncertainties, over/under sampling and sensor interactions. These advancements optimize reconstruction performance for complex processes across nuclear energy, fluid dynamics, biological systems, and manufacturing applications, enabling more accurate modeling, prediction, and control capabilities.

3.1 Hard Constraints

The previous version of `Pysensors` implements two sensor placement approaches: (1) an unconstrained optimization formulated as `QR` that allows sensors to be placed anywhere in the domain, and (2) a cost-constrained optimization formulated as `CCQR` that incorporates variable placement costs, making certain regions more expensive for sensor deployment.

Many engineering applications have extreme operating conditions, high costs, limited accessibility and safety regulations that impose significant constraints on spatial locations of sensors. Implementing spatially constrained sensor placement requires a deeper intervention in the underlying `QR` optimization framework. To address this requirement, we have developed a new optimization functionality called **General QR (GQR)** based on Karnik et. al. [4], which provides the architectural flexibility needed to handle complex spatial constraints. In `Pysensors 2.0` we enhance the algorithm’s capabilities by incorporating diverse spatial constraints defined by users through `_norm.calc.py` in `utils`. The three types of spatial constraints handled by the algorithm are:

1. **Region constrained:** This type of constraint arises when we can place either a *maximum* of or *exactly* s sensors in a certain region, while the remaining $r - s$ sensors must be placed outside the constraint region.
 - **Maximum:** This case deals with applications in which the number of sensors in the constraint region should be less than or equal to s . This functionality has been implemented through `max_n` in `_norm.calc.py`.
 - **Exact:** This case deals with applications in which the number of sensors in the constraint region should equal s . This functionality has been implemented through `exact_n` in `_norm.calc.py`.
2. **Predetermined:** This type of constraint occurs when a certain number of sensor locations s are already specified, and optimized locations for the remaining sensors are desired. This functionality has been implemented through `predetermined` in `_norm.calc.py`.
3. **Distance constrained:** This constraint enforces a minimum distance d between selected sensors. This functionality has been implemented through `distance` in `_norm.calc.py`.

Code 1: Constrained sensor selection workflow: (1) Define circular constraint with center at (20,5) and radius 5, (2) Apply Generalized QR (GQR) optimization to select optimal sensor locations within the constraint, (3) Fit SSPOR model with SVD basis, and (4) Visualize selected sensors with constraint overlay.

```
circle = ps.utils._constraints.Circle(center_x = 20, center_y = 5, radius =
    5, loc = 'in', data = X_train)
circle.draw_constraint()
circle.plot_constraint_on_data(plot_type='image')
circle.plot_grid(all_sensors=all_sensors)
const_idx, rank = circle.get_constraint_indices(all_sensors =
    all_sensors_unconst, info= X_train)
s = 4
optimizer = ps.optimizers.GQR()
optimizer_kwargs={'idx_constrained':const_idx,
    'n_sensors': r,
    'n_const_sensors': s,
    'all_sensors':all_sensors,
    'constraint_option':"exact_n"}
basis = ps.basis.SVD(n_basis_modes=n_sensors)
model = ps.SSPOR(basis = basis, optimizer = optimizer, n_sensors = r)
model.fit(X_train,**optimizer_kwargs)
top_sensors = model.get_selected_sensors()
dataframe = circle.sensors_dataframe(sensors = top_sensors)
circle.plot_constraint_on_data(plot_type='image')
circle.plot_selected_sensors(sensors = top_sensors, all_sensors =
    all_sensors)
circle.annotate_sensors(sensors = top_sensors, all_sensors = all_sensors)
```

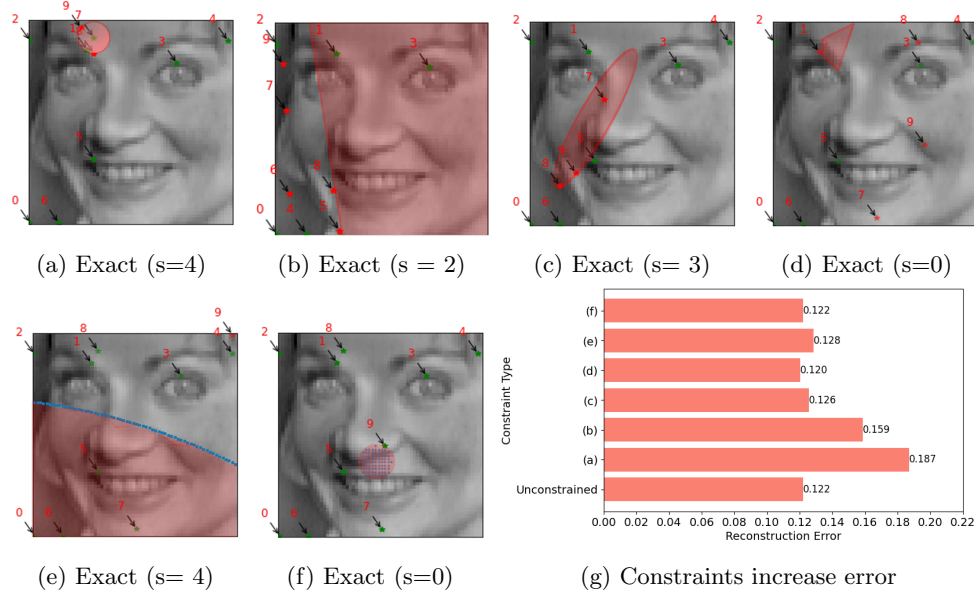


Figure 2: Constraint region definition, visualization and plotting capabilities of the algorithm. The algorithm incorporates various geometric constraint regions including circles (a), lines (b), ellipses (c), polygons (d), parabolas (e), that adapt to user’s specific requirements. Comprehensive visualization capabilities are integrated directly into the codebase, with green stars indicating original QR sensor positions and red stars highlighting sensors that have been repositioned to satisfy constraints. This visual distinction enables users to make more informed decisions. Additionally, the system supports user-defined custom functions (f) for creating specialized constraint regions tailored to specific applications. Reconstruction error on the test set increases when sensors are constrained to specific locations (g), with crowded sensor configurations exhibiting higher errors due to inter-sensor interference.

To implement spatial constraints in our sensor placement methodology, we must first define the shape of our constraint region and then identify which grid points fall within the designated constraint regions. We accomplish this using the specialized classes and functions provided in the `_constraints.py` module in `utils`. The classes define constraint regions in various geometric shapes including `Circle`, `Ellipse`, `Polygon`, `Parabola`, `Line`, and `Cylinder`. Additionally, we provide `UserDefinedConstraints` options that allow users to supply their own constraints either as a Python (`.py`) file containing their constraint shape definition or as an equation string. These classes work with two distinct data formats: image data defined by pixel coordinates and tabular data stored in dataframes with explicit `x`, `y`, and `z` values. Upon instantiation of the class, users must provide the input `data` in one of two formats: a matrix representation for images or a structured dataframe for tabular data. In the latter case, the class requires additional keyword arguments (`kwargs`) specifying the columns corresponding to `X_axis`, `Y_axis`, `Z_axis`, and `Field` parameters, thereby enabling the extraction of relevant data from their respective columns for subsequent analysis.

Once an instance of these classes have been initiated, as seen in the example code snippet above [Code 1](#), the functions `draw_constraint()`, `plot_constraint_on_data()` and `plot_grid()` provide visualization of the constraint region, the constraint region on a snapshot of the data and all possible sensor locations respectively. The function `get_constraint_indices()` is used to identify sensors within constrained regions across multiple data formats and returns indices of sensors located within the specified spatial constraints. These can then be used in subsequent optimization routines to ensure that sensor placement adheres to physical or practical limitations.

The code snippet demonstrates how `General QR (GQR)` optimizer can then be used to handle spatial constraints with the `constraint_option = exact_n`, which ensures exactly $s = 4$ sensors are placed within the constrained region. The `optimizer_kwargs` dictionary passes crucial parameters: constrained indices (`idx_constrained`), total number of sensors (`n_sensors`), number of constrained sensors (`n_const_sensors`), total available sensor locations (`all_sensors`) and the constraint option.

The SSPOR model integrates the basis, optimizer and the model is fit on the input data `X_train` with the specified optimizer key word arguments, yielding an optimal sensor placement that balances reconstruction accuracy with the imposed spatial constraints.

Following model optimization, the algorithm identifies the optimal sensor placement using the function `get_selected_sensors()`, which returns a subset of sensors (`top_sensors`) determined to be most effective for the given constraints. These sensors can subsequently be extracted into a structured dataframe via the `sensors_dataframe()` method for detailed analysis. The spatial distribution of constraints is then visualized through the `plot_constraint_on_data()` method, which generates a representation of the constraint boundaries superimposed on the underlying data field. To facilitate comparative assessment, the `plot_selected_sensors()` method graphically differentiates between the optimal sensor locations and the complete sensor array, while the `annotate_sensors()` method applies appropriate metadata labels to each sensor position, enabling efficient interpretation of the optimization results. Figure 2 illustrates the diverse constraint geometries, constraint typologies, and visualization capabilities implemented in PySensors 2.0, demonstrating the framework’s enhanced flexibility for sensor placement optimization.

3.2 Sensor Landscapes and the TPGR Optimizer

The D-optimal objective in Eqn. 4 suffers from two limitations: it is not defined for the under-sampling case $p < r$, and it is hard to interpret and visualize directly. Ref. [14] resolves these limitations by adding a prior regularization and decomposing the resulting objective into sums over the placed sensors:

$$\mathcal{H} \equiv -\log \det(\mathbf{S}^{-2} + (\mathbb{S}\Psi_r)^T(\mathbb{S}\Psi_r)/\eta^2) \approx E_b + \sum_{i \in \gamma} h_i + \sum_{i \neq j \in \gamma} J_{ij}, \quad (5)$$

where \mathbf{S} is the assumed prior covariance matrix of the coefficients \mathbf{a} and η is the assumed sensor noise magnitude. The typical prior covariances are $\mathbf{S} \propto \mathbf{I}$ (isotropic Gaussian) or $\mathbf{S} = \Sigma_r/\sqrt{N}$ (normalized singular values of training data). The series expansion over terms with more sensors is in principle exact, but we approximate it with the first two terms. In the approximation, E_b is a constant term that does not affect sensor selection, h_i, J_{ij} are the interaction landscapes computed from the basis and the prior. The objective in Eqn. 5 involves only summation over the selected sensors and is thus cheaper to evaluate and update than the original objective in Eqn. 4.

The approximate objective is used in the Two Point Greedy (TPGR) optimizer that can return a user-specified number of sensors p for any mode number r . In contrast, the QR algorithm returns exactly $p = r$ sensors in order of decreasing importance through pivoting, and all following sensors are random. The sensor sets returned by TPGR are nearly equivalent to QR for isotropic prior $\mathbf{S} \propto \mathbf{I}$ and small noise η .

After a SSPOR model is fit using the TPGR Optimizer, the one point and two point energy landscapes can be computed as seen in Code 2. For prior values, the input can either be a numpy array, corresponding to the diagonal part of the covariance matrix (e.g. all equal for an isotropic prior), or the string 'decreasing', which computes the normalized singular values from the training data.

Code 2: Implementation of the Two Point Greedy Optimizer

```
basis = ps.basis.SVD(n_basis_modes=r)
optimizer = ps.optimizers.TPGR(n_sensors, noise, prior)
model = ps.SSPOR(basis=basis, optimizer=optimizer)
model.fit(data)
sensors=model.get_selected_sensors()
one_pt_landscape = model.one_pt_energy_landscape(prior, noise)
two_pt_landscape = model.two_pt_energy_landscape(prior, noise, sensors)
```

3.3 Reconstruction Solvers

Once the set of p sensors has been determined using any of the methods, the sensor measurements \mathbf{y} can be used to determine the state coefficients \mathbf{a} . Under the assumption of linearity, the reconstruction always takes the shape $\hat{\mathbf{a}} = \mathbf{A}\mathbf{y}$ for some matrix $\mathbf{A} : r \times p$.

The first version of the reconstruction matrix corresponds to the Least Squares solution via the Moore-Penrose pseudoinverse:

$$\mathbf{A}_{LS} = (\mathbb{S}\Psi_r)^\dagger. \quad (6)$$

PySensors 2.0 adds the Regularized Least Squares solution derived in Ref. [14]:

$$\mathbf{A}_{RLS} = (\mathbf{S}^{-2} + (\mathbb{S}\Psi_r)^T(\mathbb{S}\Psi_r)/\eta^2)^{-1} (\mathbb{S}\Psi_r)^T/\eta^2, \quad (7)$$

where similarly to the TPGR optimizer, \mathbf{S} is the assumed prior covariance matrix and η is the assumed sensor noise magnitude. The relative magnitude of the coefficient prior and the noise determines whether the reconstruction primarily relies on the measurements or the prior information. This Regularized Least Squares solution is now the default reconstruction solver for PySensors 2.0.

Code 3: Reconstruction using Regularized Least Squares

```
|| model.predict(x_test, prior, noise=noise)
```

The 'unregularized' method will use the Least Squares method using Moore-Penrose pseudoinverse reconstruction solver.

```
|| model.predict(x_test, method='unregularized')
```

3.4 Uncertainty Quantification and Heatmaps

For any choice of the reconstruction matrix \mathbf{A} , the reconstruction $\hat{\mathbf{a}} = \mathbf{A}\mathbf{y}$ predicts the *most likely* coefficients of the reconstructed state, from which the full state can be obtained via projection $\hat{\mathbf{x}} = \Psi_r\hat{\mathbf{a}}$. However, the reconstruction is sensitive to the sensor noise. The expected error of the reconstruction is captured by the covariance matrix derived in Ref. [14]:

$$\mathbf{K} = \Psi_r\mathbf{B}\mathbf{B}^T\Psi_r^T; \quad \mathbf{B} = \eta\mathbf{A}. \quad (8)$$

In practice, the whole covariance matrix in Eqn. 8 is both hard to store as it takes $n \times n$ space in computer memory, and hard to interpret as it has a low rank. Instead, PySensors 2.0 computes a reduced metric: the *uncertainty heatmap* $\sigma = \sqrt{\text{diag}(\mathbf{K})}$ or the marginal standard deviation of each pixel of the reconstruction. Note that for linear reconstructions, the error metrics do not depend on the system state, and thus can be computed with fairly fast numerical linear algebra in time that does not scale with the number of train or test snapshots N .

```
|| x_test = X_test[:, sensors]
|| predicted_state = model.predict(x_test, prior, noise)
|| sigma = model.std(prior, noise)
```

4 Examples

In this section, we demonstrate the enhanced capabilities of PySensors 2.0 through illustrative examples that highlight key advancements in the framework. We present case studies focused on three critical areas: constraint-aware optimization for realistic deployment scenarios using the GQR optimizer, one and two point sensor landscapes, and rigorous uncertainty quantification with the TPGR optimizer for sensor networks subject to measurement noise. For additional functionalities readers are directed to the examples available on the PySensors documentation site ^{*}.

4.1 Functional constraints for a Nuclear simulation Example

Consider the challenge of monitoring a test capsule with an electric heater at the center that mimics the neutronics effect of a nuclear fuel rod. The goal is to study heat transfer from the rod to circulating water before reactor installation. Since the test capsule is axisymmetric, computational fluid dynamics (CFD) simulations model half the domain with the heater at the center. The goal is to strategically place point thermocouples within the capsule to capture the temperature profile. For additional details on this experiment, see Section 4.3 "Steady-state simulation of the OPTI-TWIST prototype" in Karnik et al. [4].

^{*}<https://github.com/dynamicslab/pysensors>

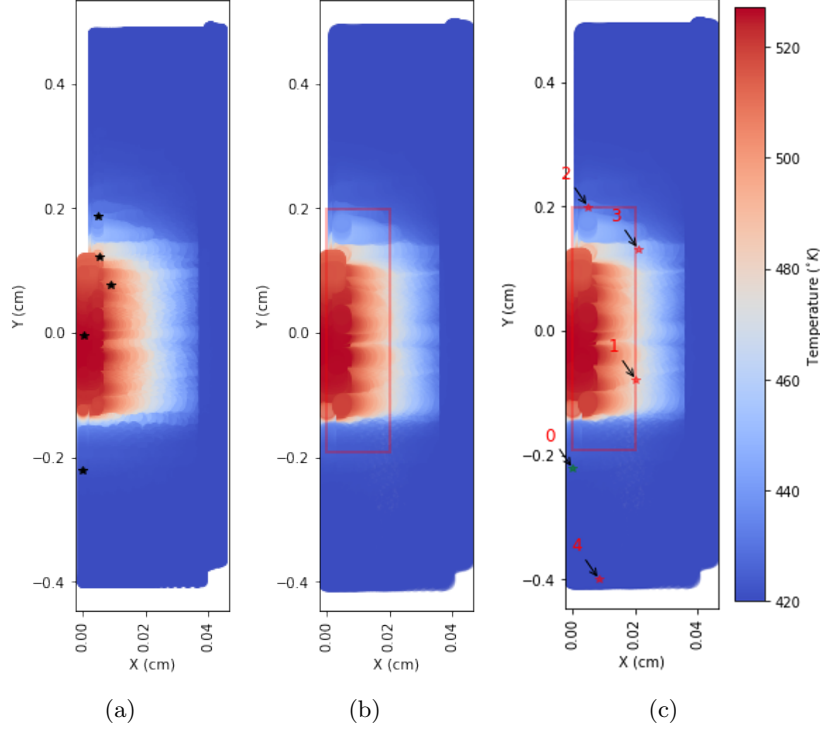


Figure 3: (a) Unconstrained sensor placement optimization results in placing sensors in heater adjacent locations. Implementation of spatial constraints includes defining an exclusion region around the heater, plotting the constraint region overlaid on experimental data (b), and final constrained sensor locations positioned away from the heater (c).

Code 4: Implementation of Sparse Sensor Placement Optimization for Reconstruction (SSPOR) using PySensors. The code demonstrates setting up a rank-5 SVD basis, QR optimizer, and fitting the model to identify optimal sensor locations from the available data. The model selects $r=5$ sensors from all possible sensor locations to minimize reconstruction error.

```
r = 5
basis = ps.basis.SVD(n_basis_modes=r)
optimizer = ps.optimizers.QR()
model = ps.SSPOR(basis=basis, optimizer=optimizer, n_sensors=r)
model.fit(data)
all_sensors = model.get_all_sensors()
sensors = model.get_selected_sensors()
```

The proximity to the heater element creates significant space constraints that render sensor placement in heater-adjacent locations experimentally infeasible. We first look at the placement of $r = 5$ unconstrained sensors using QR as the optimizer, and the SVD as the `basis` as seen in Code 4. Figure 3a shows that 3 of the 5 sensors are adjacent to the heater (black stars). Based on this we want to constrain the domain around the heater between $x_1 = 0, x_2 = 0.02, y_1 = -0.19, y_2 = 0.19$ cm to have no sensors.

A polygon class is initialized to identify sensor locations within the specified rectangular domain as shown in Code 5. A polygonal constraint region is first defined using four corner coordinates $(x_1, y_1), (x_2, y_1), (x_2, y_2), (x_1, y_2)$, with the constraint configured to operate on interior points (`loc = in`). As the data is stored within a dataframe (df), this case differs from subsection 3.1, as we now provide `data = df` and must specify the `Y_axis = 'Y (m)'`, `X_axis = 'X (m)'` and `Field = 'Temperature (K)'`, as opposed to working with images where spatial coordinates are implicit. The column names `'X (m)'`, `'Y (m)'`, and `'Temperature (K)'` correspond to the X-axis, Y-axis, and the temperature variable measured by the sensors, respectively.

Code 5: Constrained sensor placement optimization using PySensors with a rectangular polygon constraint. The code defines spatial boundaries, applies the General QR (GQR) optimizer with SVD basis to select r optimal sensors within the constraint region, and visualizes the results with sensor annotations on the temperature field contour map.

```

polygon = ps.utils._constraints.Polygon([(x1, y1), (x2, y1), (x2, y2), (x1, y2)
], loc = 'in', data = df, Y_axis = 'Y (m)', X_axis = 'X (m)', Field = '
Temperature (K)')
polygon.draw_constraint(plot = (fig, ax))
polygon.plot_constraint_on_data(plot_type='contour_map', plot = (fig, ax), s
= 20)
const_idx, rank = polygon.get_constraint_indices(all_sensors=all_sensors,
info =df)
s = 0
optimizer = ps.optimizers.GQR()
optimizer_kwargs={'idx_constrained':const_idx,
'n_sensors':r,
'n_const_sensors':s,
'all_sensors':all_sensors,
'constraint_option':"max_n"}
basis = ps.basis.SVD(n_basis_modes=r)
model = ps.SSPOR(basis = basis, optimizer = optimizer, n_sensors = r)
model.fit(data,**optimizer_kwargs)
top_sensors = model.get_selected_sensors()
data_sens = polygon.sensors_dataframe(sensors = top_sensors)
polygon.plot_constraint_on_data(plot_type='contour_map', plot = (fig, ax), s
= 20)
polygon.annotate_sensors(sensors = top_sensors, all_sensors=all_sensors)

```

The constraint visualization is implemented through two complementary approaches. First, the geometric boundary of the constraint region is rendered onto the specified figure and axis objects through `draw_constraint`. Next, an overlay visualization of the constraint region onto the underlying data field using contour mapping is achieved by the function `plot_constraint_on_data` as shown in Figure 3b. The constraint indices are extracted from the complete sensor array using the `get_constraint_indices` method, which returns both the constrained sensor indices and their corresponding rank ordering. The optimization framework utilizes the General QR (GQR) algorithm configured with constraint-aware parameters, including the constrained sensor indices `const_idx`, total number of sensors r and number of constrained sensors s . As we want zero sensors in the constrained region, both `max_n` and `exact_n` will give us the same sensor configuration in this case.

The Sparse Sensor Placement Optimization for Reconstruction (SSPOR) model combines SVD basis functions with the GQR optimizer to identify near-optimal sensor locations within the constrained domain. After training on the dataset with specified optimizer keyword arguments (`kwargs`), the model extracts the highest-ranked sensor positions through `get_selected_sensors`. They can be formatted into a structured dataframe revealing the locations of the sensors using the function `sensors_dataframe`. The visualization pipeline overlays these optimized sensor locations on the constraint region's contour map through `plot_constraint_on_data` and `annotate_sensors` as seen in Figure 3c. The plot annotates sensors with red stars for locations that changed from the unconstrained case and green stars for locations that remained the same.

Thus, based on spatial constraints and their geometric shapes that might arise in engineering applications, the framework can initialize various constraint classes and obtain the corresponding constraint indices. The system handles data in both dataframe and image formats, making it applicable to simulation and experimentation data. Sensor location visualization reveals how the constrained optimization shifts sensor positions from their unconstrained near-optimal configuration. Additional analysis tools include reconstruction error studies and noise-induced uncertainty heatmaps that quantify the impact of these spatial constraints on sensing performance [4] as described in the next section.

4.2 Two-point Greedy Optimizer Example

We use the SST dataset to showcase the TPGR optimizer, which uses two-point greedy optimization introduced in Ref. [14] for sensor placement.

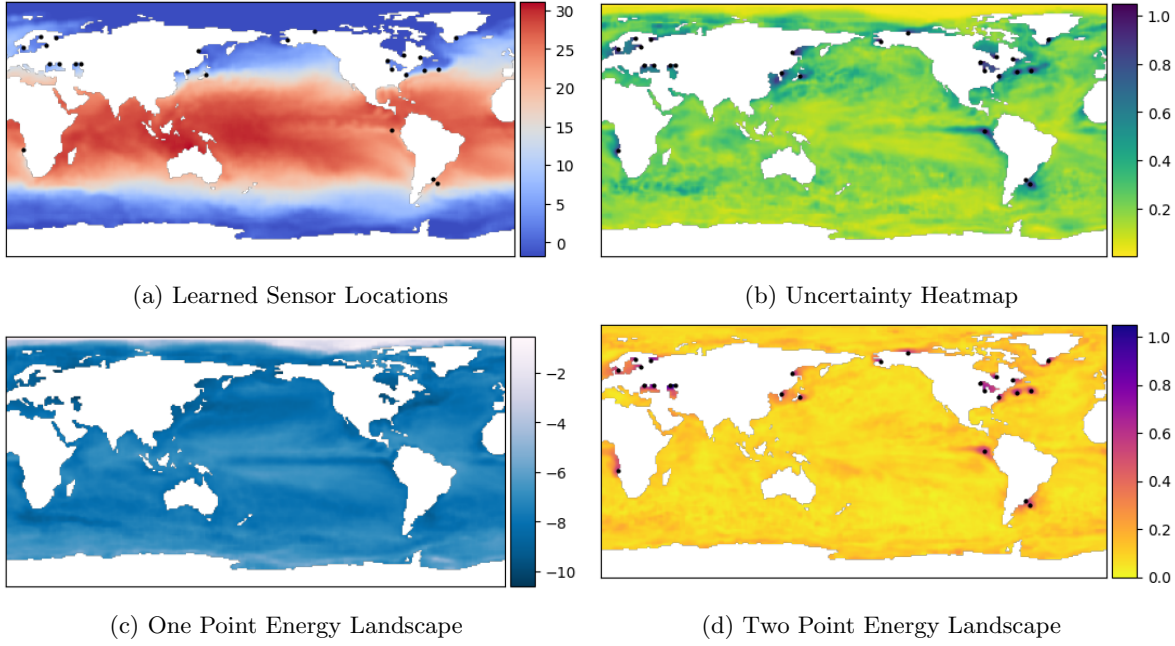


Figure 4: Sensor placement and reconstruction using Two-Point Greedy Optimization results in (a) learned sensor locations on a snapshot of the SST dataset (units in degrees Celcius), (b) Uncertainty heatmap of the reconstructed state, (c) One-point energy landscape at each location, (d) Two-point energy landscape at each location, as a sum of two point interactions with all placed sensors.

```

r = 100
prior = np.full(r, 1000)
noise = 1
p = 25
model = ps.SSPOR(
    basis=ps.basis.SVD(n_basis_modes=r),
    optimizer=ps.optimizers.TPGR(n_sensors=p, prior, noise)
)
model.fit(data_train)

```

We place $p = 25$ sensors using TPGR as the optimizer and SVD as the basis, as shown in Figure 4a. For the TPGR optimizer to place sensors, prior and noise are required arguments. In this example, we are using a Isotropic Gaussian prior (flat prior). Other options of prior can be used: such as a prior constructed from normalized singular values of the training set of the data (decreasing prior), truncated to the number of basis modes r , or a user defined prior.

Next, we compute the uncertainty heatmap for the reconstructed state as shown in Figure 4b. Using the TPGR optimizer enables us to calculate the one point energy landscape as shown in Figure 4c and two point energy landscapes. Both energy landscape computations require prior and noise as an input. Two point landscape also requires the selected sensors whose two-point interactions need to be computed. If the selected sensors are an array of multiple sensors, the two point energy landscape will be the sum of two point interactions of all the selected sensors as shown in Figure 4d. If the selected sensors is just a single sensor, the two point energy landscape will be the two point interaction of that particular sensor.

```

sensors = model.get_selected_sensors()
data_test = data_test[:, sensors]
predicted_state = model.predict(data_test, prior, noise)
sigma = model.std(prior)
one_pt_landscape = model.one_pt_energy_landscape(prior, noise)
two_pt_landscape = model.two_pt_energy_landscape(prior, noise,
    selected_sensors)

```

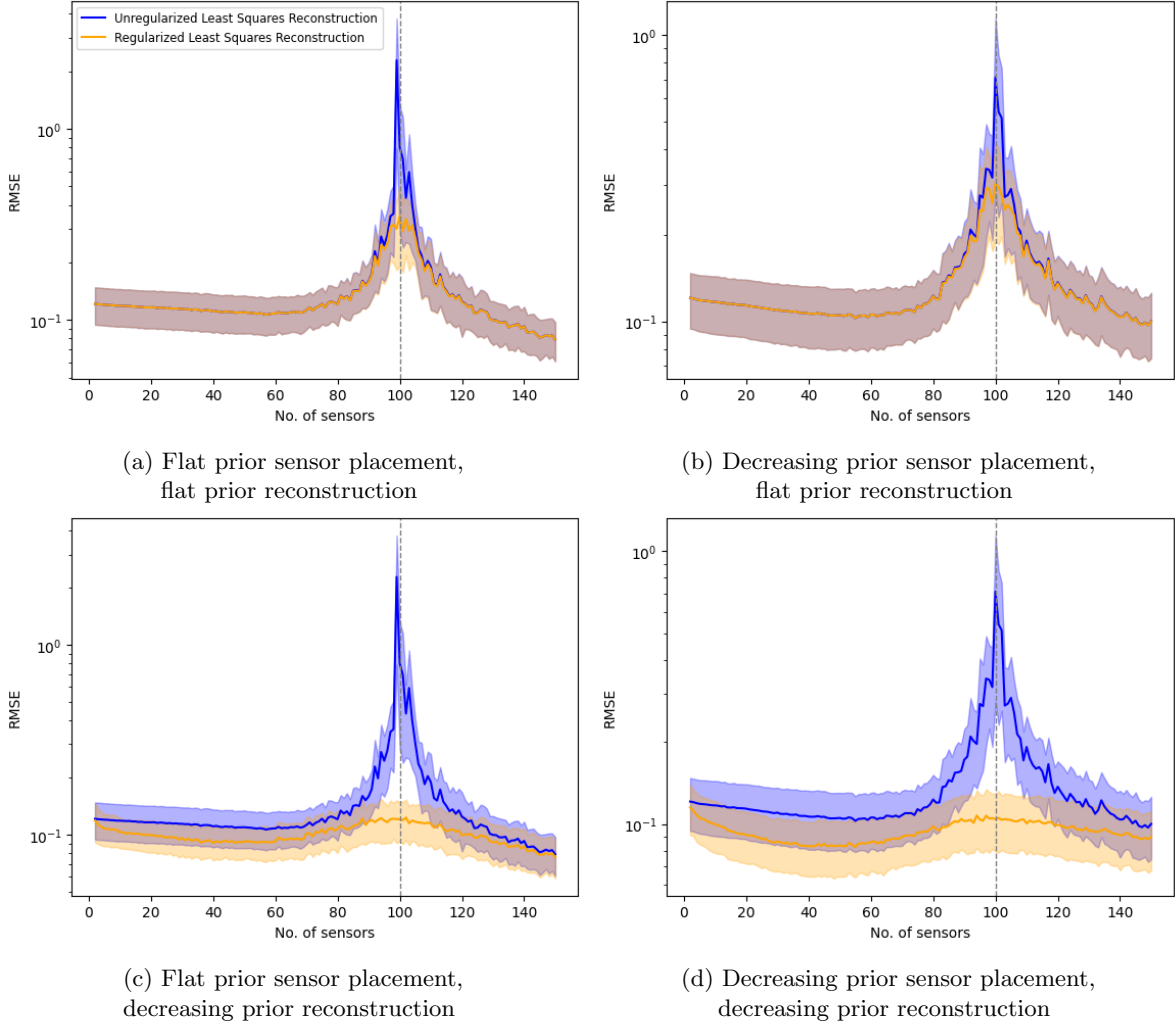


Figure 5: Reconstruction error curves for different choices of the sensor placement prior and reconstruction prior on the Olivetti dataset.

4.3 Reconstruction Comparison Example

In another notebook, we compare the reconstruction methods on the Olivetti dataset: unregularized reconstruction and regularized reconstruction. We use the TPGR optimizer for sensor placement using either a flat prior or a decreasing prior. We then reconstruct using both unregularized and regularized reconstruction, and compare their RMSE values with the true states along a range of number of placed sensors, from 1 to 150 sensors. Since regularized reconstruction also requires a prior, there are four possible combinations that need to be considered: Figure 5a shows sensor placement using a flat prior TPGR optimizer and a flat prior for the regularized reconstruction. In Figure 5b, sensors are placed using the decreasing prior and regularized reconstruction is done using a flat prior. Figure 5c uses flat prior for sensor placement and decreasing prior for regularized reconstruction. Finally, Figure 5d uses a decreasing prior for both sensor placement and regularized reconstruction. The flat prior selected for this particular dataset is the scaled identity $4I$.

We then compare the reconstruction methods for the QR and TPGR optimizers, using only the decreasing prior for sensor placement and regularized reconstruction. Figure 6 shows a comparison of four plots, QR with unregularized reconstruction, QR with decreasing prior regularized reconstruction, decreasing prior TPGR with unregularized reconstruction and decreasing prior TPGR with decreasing prior regularized reconstruction. Empirically, using a regularized reconstruction with a decreasing prior reduces the RMSE significantly.

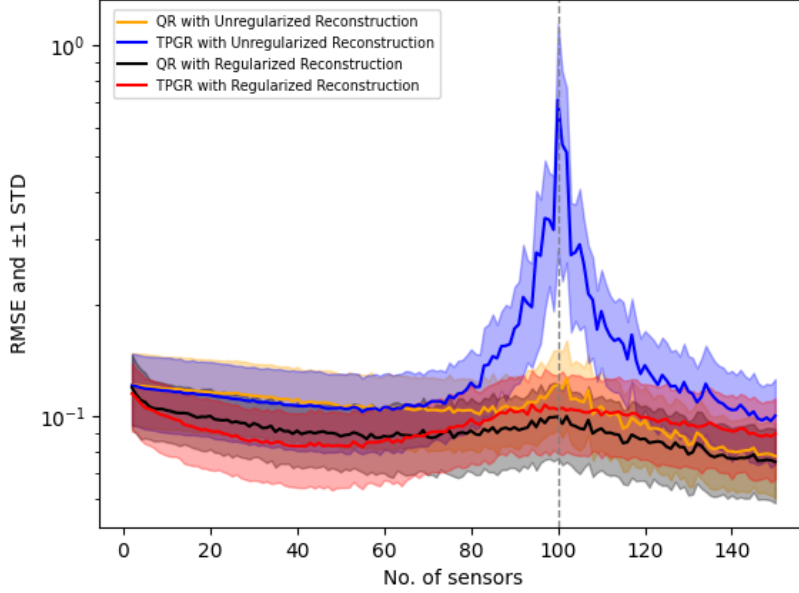


Figure 6: Comparison of sensor placement using QR and TPGR optimizers for unregularized and regularized reconstruction on the Olivetti Dataset

5 Practical Tips

In this section, we delineate key practical considerations for optimizing sensor placements using **PySensors** 2.0. Our discussion focuses primarily on the reconstruction methodology, as this domain encompasses the most significant enhancements implemented in the latest version.

The selection of an appropriate basis is a critical determinant in both classification and reconstruction processes, particularly when dealing with high-dimensional systems. For static data or image analysis, the Singular Value Decomposition (SVD) basis often proves optimal as it efficiently captures the maximum variance within the dataset. Conversely, for time-dependent phenomena, Dynamic Mode Decomposition (DMD) or Fourier-based approaches may yield superior results by inherently accounting for temporal evolution patterns. The enhanced custom basis functionality in **PySensors** 2.0 now facilitates seamless integration with specialized basis types, such as DMD modes derivable from complementary packages like **PyDMD** [25], as demonstrated in the basis example notebook available in the **PySensors** repository.

Generally, reconstruction error is expected to diminish as both sensor quantity p and basis mode count r increase. However, the relative proportion between sensors and modes represents a crucial design parameter. When employing an SVD basis, two counterintuitive phenomena emerge. First, as mode count increases beyond a certain threshold, additive measurement noise begins to dominate the error profile, paradoxically increasing overall reconstruction error (see [26]). In the previous version of **PySensors**, this effect could be mitigated by oversampling i.e. deploying more sensors than modes which would lead to randomized sensor selection once sensors exceeded modes $p > r$ [13]. Second, the reconstruction tends to be particularly unstable when the number of sensors matches the number of modes $p \approx r$ since in that regime the reconstruction requires inverting a particularly ill-conditioned matrix. This phenomenon is known as *double descent* across machine learning literature. At fixed mode number r , the reconstruction error first decreases with number of sensors p , then spikes at $p \approx r$, then decreases again. In practical terms, two techniques reduce the reconstruction error: using a regularized reconstruction (Figure 7) and deliberately *under*-sampling ($p < r$) the number of sensors for reconstruction. Before deploying a set of sensors optimized with **PySensors**, we recommend constructing the RMSE curves and deciding on the values of p, r that lead to the best compromise between reconstruction error and deployment cost (not necessarily $p = r$). A detailed theory of double descent in sensors, including its origins and mitigation, is subject of a forthcoming publication [27].

When implementing spatial constraints, it is optimal to maintain $p = r$ for sensor placement and then evaluate the best number of sensors for reconstruction. Exceeding this threshold causes **GQR** to

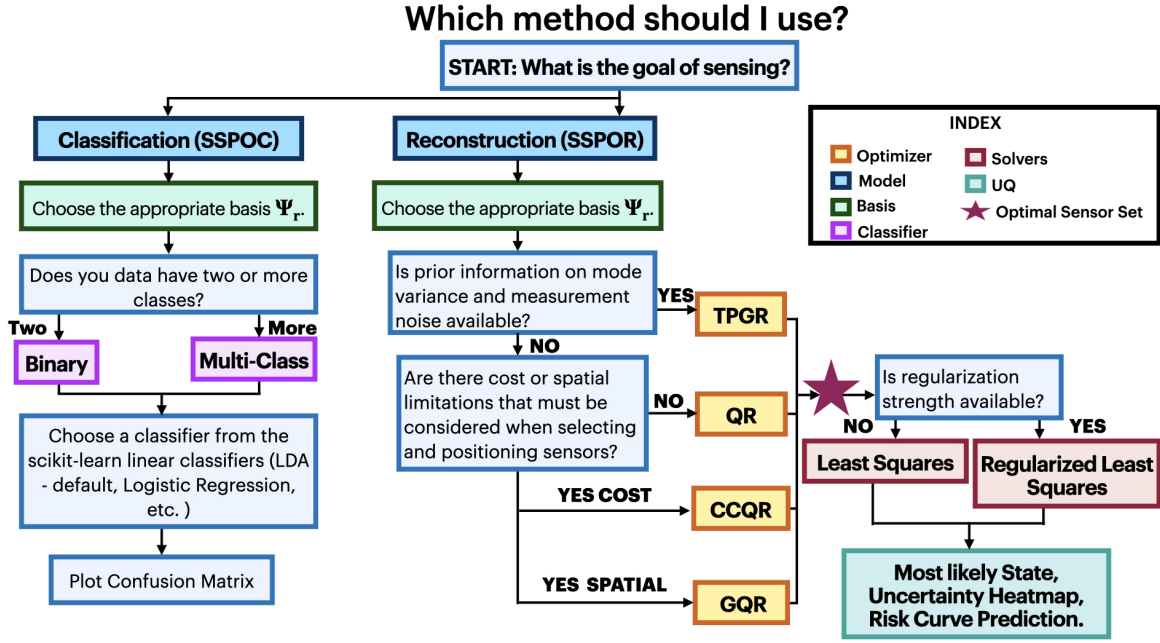


Figure 7: When selecting a sensing method in PySensors, consider your primary objective: For field reconstruction in standard settings, use QR with Identity or SVD basis. For classification tasks, leverage SVD basis with SSPOC optimizer. When facing spatial constraints, choose GQR optimizer. For under-sampling ($p < r$) and over-sampling cases ($p > r$) scenarios, select TPGR optimizer. In noisy environments enable uncertainty quantification for robust results.

place sensors randomly, resulting in loss of control over sensor placement within the constrained region. This random allocation risks potential selection of sensors within the constrained region when $p > r$.

In conclusion, Figure 7 presents a comprehensive flowchart for near-optimal sensor placement using PySensors 2.0. Users can select a basis for their data, and then determine whether to place sensors for reconstruction or classification purposes. For reconstruction applications, different optimizers are available based on specific requirements such as cost limitations, spatial constraints, or prior knowledge of the data. The current default method for reconstruction is regularized least squares, where users can either provide their own prior or allow PySensors to calculate it automatically. Alternatively, users can opt for the standard least squares solver if regularization is not desired. After obtaining the optimal sensor configuration, we recommend analyzing the placement effectiveness through uncertainty heatmaps and reconstruction error metrics to validate the solution for your specific application.

6 Future Functionality

Future work to enhance PySensors functionality includes integrating the two-point greedy (TPGR) method for sensor placement with additional cost landscapes and spatial constraints. This will prove beneficial in both under- and over-sampling scenarios where spatial constraints limit sensor placement options, a common challenge in most engineering applications. Another critical development involves incorporating all uncertainty sources beyond measurement noise into the UQ heatmap and risk curve predictions. This comprehensive uncertainty quantification would help explain the double descent phenomenon observed in sensor-based reconstruction, where reconstruction error initially decreases with additional sensors but then increases due to overfitting or noise amplification.

7 Acknowledgments

The authors acknowledge support from the Boeing Company, NSF AI Institute in Dynamic Systems under grant 2112085 and through the Idaho National Laboratory (INL) Laboratory Directed Research

References

- [1] N. B. Erichson, L. Mathelin, Z. Yao, S. L. Brunton, M. W. Mahoney, and J. N. Kutz, “Shallow neural networks for fluid flow reconstruction with limited sensors,” *Proceedings of the Royal Society A*, vol. 476, no. 2238, p. 20 200 097, 2020.
- [2] K. Manohar, T. Hogan, J. Buttrick, A. G. Banerjee, J. N. Kutz, and S. L. Brunton, “Predicting shim gaps in aircraft assembly with machine learning and sparse sensing,” *Journal of manufacturing systems*, vol. 48, pp. 87–95, 2018.
- [3] M. T. Alonso, P. López-Dekker, and J. J. Mallorquí, “A novel strategy for radar imaging based on compressive sensing,” *IEEE Transactions on Geoscience and Remote Sensing*, vol. 48, no. 12, pp. 4285–4295, 2010.
- [4] N. Karnik, M. G. Abdo, C. E. Estrada-Perez, J. S. Yoo, J. J. Cogliati, R. S. Skifton, P. Calderoni, S. L. Brunton, and K. Manohar, “Constrained optimization of sensor placement for nuclear digital twins,” *IEEE Sensors Journal*, 2024.
- [5] A. Krause, A. Singh, and C. Guestrin, “Near-optimal sensor placements in gaussian processes: Theory, efficient algorithms and empirical studies,” *Journal of Machine Learning Research*, vol. 9, no. 2, 2008.
- [6] A. Alexanderian, “Optimal experimental design for infinite-dimensional bayesian inverse problems governed by pdes: A review,” *Inverse Problems*, vol. 37, no. 4, p. 043 001, 2021.
- [7] D. L. Donoho, “Compressed sensing,” *IEEE Transactions on information theory*, vol. 52, no. 4, pp. 1289–1306, 2006.
- [8] T. H. Summers, F. L. Cortesi, and J. Lygeros, “On submodularity and controllability in complex dynamical networks,” *IEEE Transactions on Control of Network Systems*, vol. 3, no. 1, pp. 91–101, 2015.
- [9] S. Joshi and S. Boyd, “Sensor selection via convex optimization,” *IEEE Transactions on Signal Processing*, vol. 57, no. 2, pp. 451–462, 2008.
- [10] B. M. de Silva, K. Manohar, E. Clark, B. W. Brunton, S. L. Brunton, and J. N. Kutz, “Pysensors: A python package for sparse sensor placement,” *arXiv preprint arXiv:2102.13476*, 2021.
- [11] K. Manohar, B. W. Brunton, J. N. Kutz, and S. L. Brunton, “Data-driven sparse sensor placement for reconstruction: Demonstrating the benefits of exploiting known patterns,” *IEEE Control Systems Magazine*, vol. 38, no. 3, pp. 63–86, 2018.
- [12] B. W. Brunton, S. L. Brunton, J. L. Proctor, and J. N. Kutz, “Sparse sensor placement optimization for classification,” *SIAM Journal on Applied Mathematics*, vol. 76, no. 5, pp. 2099–2122, 2016.
- [13] E. Clark, S. L. Brunton, and J. N. Kutz, “Multi-fidelity sensor selection: Greedy algorithms to place cheap and expensive sensors with cost constraints,” *IEEE Sensors Journal*, vol. 21, no. 1, pp. 600–611, 2020.
- [14] A. A. Klishin, J. N. Kutz, and K. Manohar, “Data-induced interactions of sparse sensors,” *arXiv preprint arXiv:2307.11838*, 2023.
- [15] K. A. Klise, B. L. Nicholson, and C. D. Laird, “Sensor placement optimization using chama,” Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2017.
- [16] S. D. Narayanan, Z. B. Patel, A. Agnihotri, and N. Batra, “A toolkit for spatial interpolation and sensor placement,” in *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, 2020, pp. 653–654.
- [17] T.-H. Yi, H.-N. Li, and M. Gu, “Optimal sensor placement for structural health monitoring based on multiple optimization strategies,” *The Structural Design of Tall and Special Buildings*, vol. 20, no. 7, pp. 881–900, 2011.

- [18] R. G. Baraniuk, V. Cevher, M. F. Duarte, and C. Hegde, “Model-based compressive sensing,” *IEEE Transactions on information theory*, vol. 56, no. 4, pp. 1982–2001, 2010.
- [19] E. J. Candes, J. K. Romberg, and T. Tao, “Stable signal recovery from incomplete and inaccurate measurements,” *Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences*, vol. 59, no. 8, pp. 1207–1223, 2006.
- [20] P. J. Schmid, “Dynamic mode decomposition of numerical and experimental data,” *Journal of fluid mechanics*, vol. 656, pp. 5–28, 2010.
- [21] M. Barrault, Y. Maday, N. C. Nguyen, and A. T. Patera, “An ‘empirical interpolation’ method: Application to efficient reduced-basis discretization of partial differential equations,” *Comptes Rendus Mathematique*, vol. 339, no. 9, pp. 667–672, 2004.
- [22] S. Chaturantabut and D. C. Sorensen, “Nonlinear model reduction via discrete empirical interpolation,” *SIAM Journal on Scientific Computing*, vol. 32, no. 5, pp. 2737–2764, 2010.
- [23] Z. Drmac and S. Gugercin, “A new selection operator for the discrete empirical interpolation method—improved a priori error bound and extensions,” *SIAM Journal on Scientific Computing*, vol. 38, no. 2, A631–A648, 2016.
- [24] K. Manohar, J. N. Kutz, and S. L. Brunton, “Optimal sensor and actuator selection using balanced model reduction,” *IEEE Transactions on Automatic Control*, vol. 67, no. 4, pp. 2108–2115, 2021.
- [25] S. M. Ichinaga, F. Andreuzzi, N. Demo, M. Tezzele, K. Lapo, G. Rozza, S. L. Brunton, and J. N. Kutz, “Pydmd: A python package for robust dynamic mode decomposition,” *Journal of Machine Learning Research*, vol. 25, no. 417, pp. 1–9, 2024.
- [26] B. Peherstorfer, Z. Drmac, and S. Gugercin, “Stability of discrete empirical interpolation and gappy proper orthogonal decomposition with randomized and deterministic sampling points,” *SIAM Journal on Scientific Computing*, vol. 42, no. 5, A2837–A2864, 2020.
- [27] A. A. Klishin, S. E. Otto, J. N. Kutz, and K. Manohar, “Origins and mitigation of double descent in reduced order modeling,” *In preparation*, 2025.