SAND2020-8806PE

# Software@Sandia

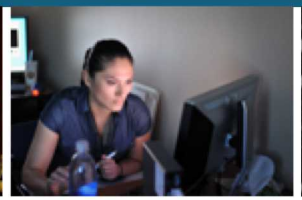
U.S. DEPARTMENT OF ENERGY

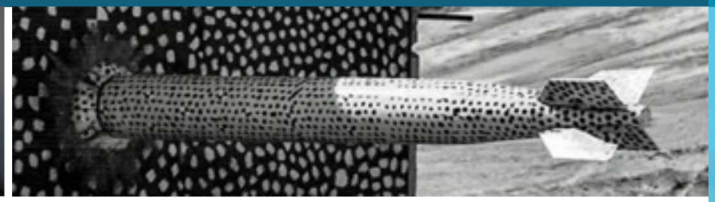
CCR

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

National Nuclear Security Administration

# Windowed least-squares model reduction in Pressio





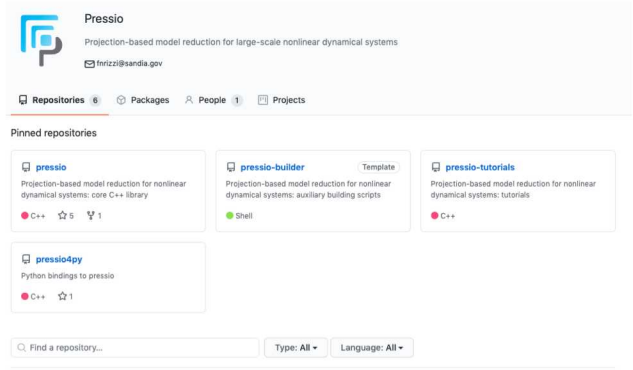
*Eric Parish and Francesco Rizzi*

Software@Sandia Series August 24, 2020

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

# Overview

- Outline windowed least-squares projection-based model reduction in Pressio
- Projection-based reduced-order models (ROMs) are data-informed surrogate models
- Pressio is a computational framework aimed at providing performant projection-based ROMs to **generic** application codes
  - Developed and maintained at Sandia
  - Roughly two years old
  - Has been coupled to several Sandia application codes

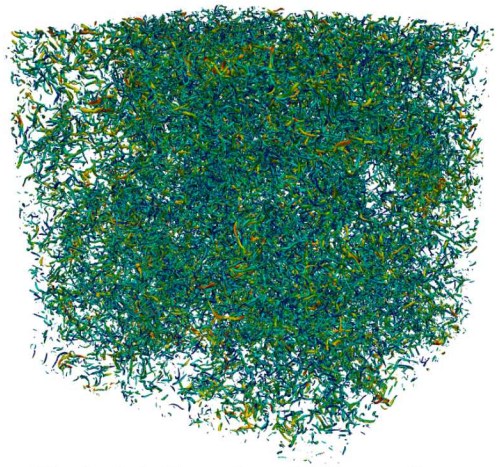



https://github.com/Pressio

- Tutorial overview
  - Outline projection-based reduced-order models and the windowed least-squares method
  - Provide an overview of Pressio
  - End-to-end example of building a ROM in Pressio for the shallow water equations

# Projection-based ROMs: Motivation

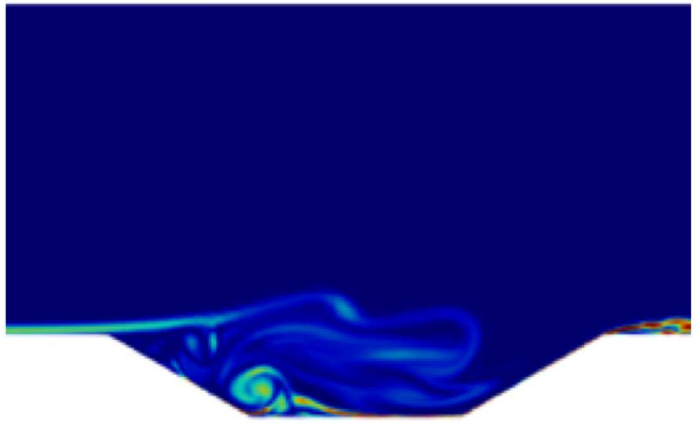
- High-fidelity full-order model simulations (FOMs) are ubiquitous in engineering and science



*High fidelity simulation of homogeneous isotropic turbulence*

- FOMs incur a prohibitively high computational cost
  - Can require millions of hours of CPU time

- Many engineering problems are time-critical or many-query by nature
  - Require many evaluations of the FOM
  - Require low wall-clock times



*High-fidelity simulation of cavity flow*

- **Computational cost of the FOM is a bottleneck**

- **Projection-based reduced-order models** provide a solution
  - Enable **accurate** approximate solutions at a **low** computational cost

# What are projection-based ROMs?

- Projection-based ROMs (pROMs) are data-informed numerical methods
  - Similar to spectral methods

Analytic basis functions

Spectral method

$$u(x,t) = \sum_{k=1}^{n} e^{ikx} a_k(t)$$

- Operate by projecting governing equations onto low-dimensional data-driven subspaces
  - pROMs use basis functions constructed from data
  - Involves solving the governing equations

Projection-based reduced-order model

$$u(x,t) = \sum_{k=1}^{n} u_k^{\text{data}}(x) a_k(t)$$

Data-driven basis functions

- Other types of ROMs:
  - Data-centric surrogate models: create a purely data-driven surrogate model
  - Reduced fidelity/physics models: use coarse meshes and/or reduced physics

# Brief description of projection-based ROMs

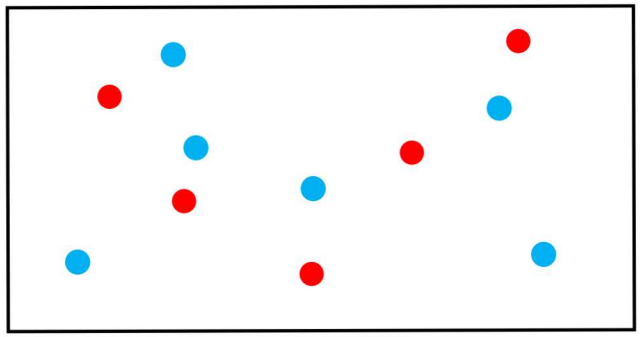
- We focus on the dynamical system

$$\frac{d}{dt}\boldsymbol{x}(t) = \boldsymbol{f}(\boldsymbol{x}, t, \boldsymbol{\mu})$$

State: $\boldsymbol{x}(t) \in \mathbb{R}^N$

Parameters: $\boldsymbol{\mu} \in \mathcal{D} \subset \mathbb{R}^{N_\mu}$

Parameter domain: $\mathcal{D}$



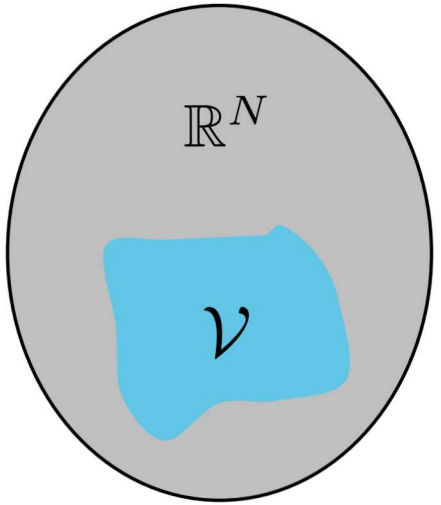
$\mathcal{D}$

$$\boldsymbol{x}_1(t, \boldsymbol{\mu}_1)\ , \boldsymbol{x}_2(t, \boldsymbol{\mu}_2)\ , \ldots , \boldsymbol{x}_s(t, \boldsymbol{\mu}_s)$$

- pROMs aim to expedite the solution process

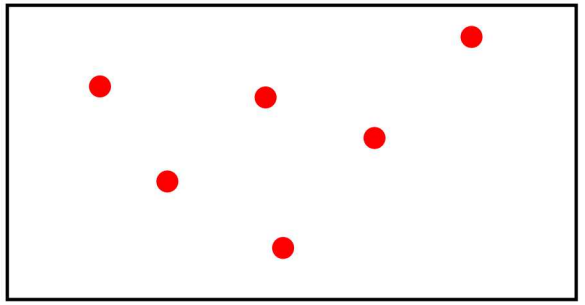
- pROMs involve the following steps:

1. Solve the FOM to obtain "training data"
2. Identify low-dimensional structure (subspace) within the training data
3. Project the FOM onto the low dimensional subspace
4. Solve the pROM for new predictive cases



# Trial subspaces: some details

- Step 2: Identify low-dimensional structure within the training data
  - Typically achieved with principal component analysis

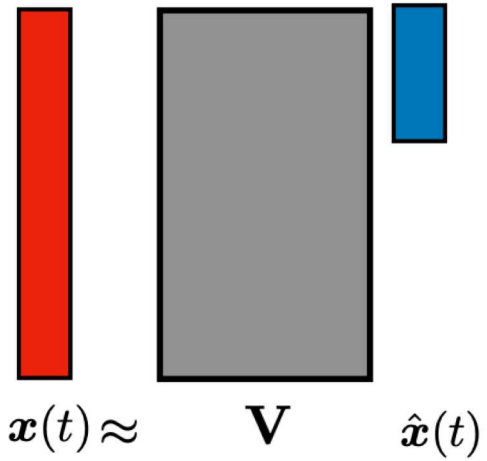


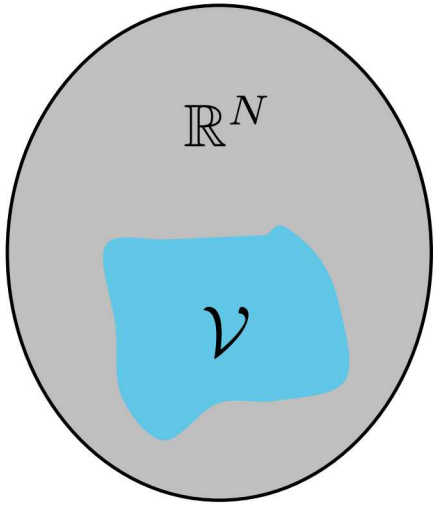
$\mathcal{D}$

$$\boldsymbol{x}_1(t, \boldsymbol{\mu}_1), \boldsymbol{x}_2(t, \boldsymbol{\mu}_2), \ldots, \boldsymbol{x}_s(t, \boldsymbol{\mu}_s)$$

- Results in the approximation: $\boldsymbol{x}(t) \approx \mathbf{V}\hat{\boldsymbol{x}}(t)$

Generalized coordinates: $\hat{\boldsymbol{x}}(t) \in \mathbb{R}^K$




$$\text{Range}(\mathbf{V}) = \mathcal{V}$$

# Building the pROM

- We approximate the state as

$$\boldsymbol{x}(t) \approx \mathbf{V}\hat{\boldsymbol{x}}(t)$$

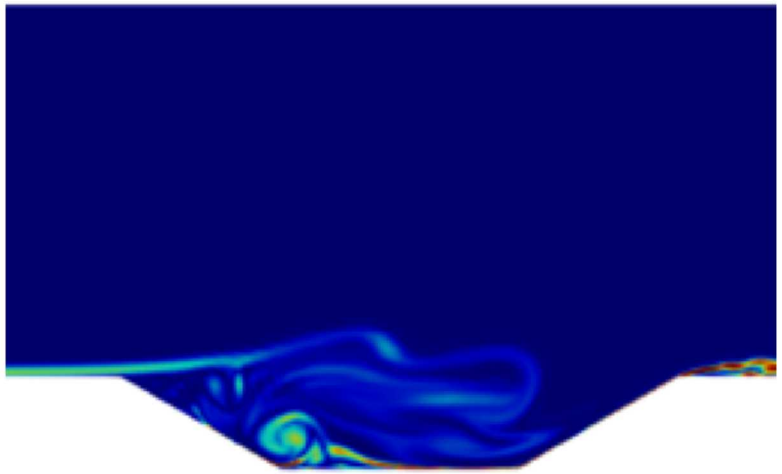
- Insert approximation into the governing equations:

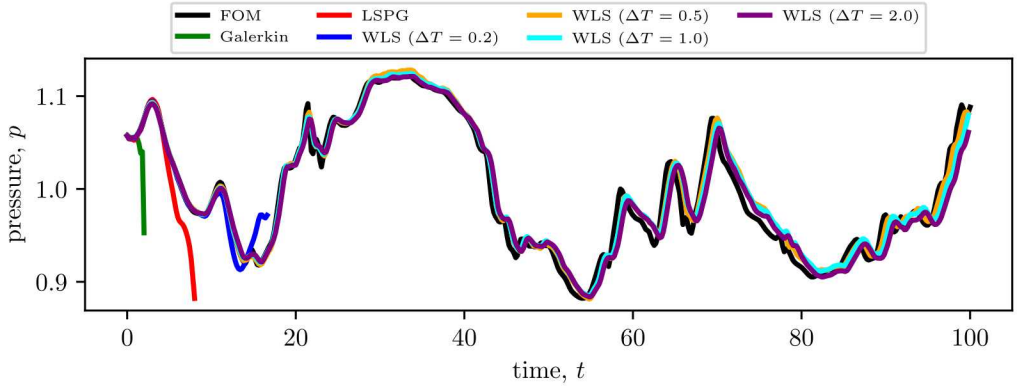
$$\mathbf{V}\frac{d\hat{\boldsymbol{x}}}{dt} = \boldsymbol{f}(\mathbf{V}\hat{\boldsymbol{x}}, t, \boldsymbol{\mu})$$

  - N equations for K unknowns (K << N)

- Various techniques exist to reduce the system dimension
  - Galerkin method
  - Least-squares Petrov—Galerkin method (LSPG)
  - **Windowed least-squares method (WLS)**

E. Parish and K. Carlberg, *"Windowed least-squares model reduction for dynamical systems"*, Journal of Computational Physics, 2020 (under revision).

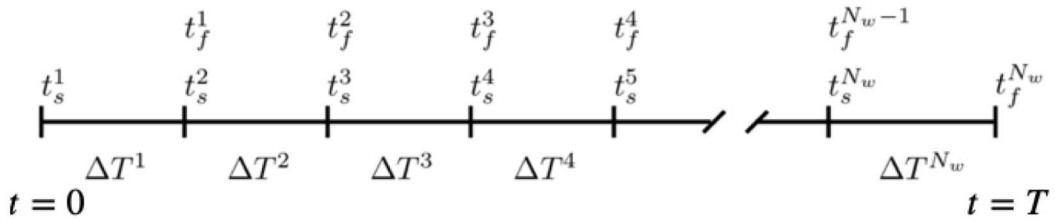





- **Galerkin and LSPG can be unstable**

# Windowed least squares ROM

- Decompose the time domain into windows




$$[t_s^n, t_f^n] \subseteq [0,T]$$
$$\Delta T^n = t_f^n - t_s^n$$
$$t_s^1 = 0,\ t_f^{N_w} = T$$

- Over each window, define an objective functional that measures the **residual**

$$\boldsymbol{J}^n(\boldsymbol{x}) = \frac{1}{2}\int_{t_s^n}^{t_f^n} \|\dot{\boldsymbol{x}}(t) - \boldsymbol{f}(\boldsymbol{x}, t, \boldsymbol{\mu})\|_{\mathbf{A}}^2$$

- **WLS:** sequentially solve a minimization problem over each window

$$\hat{\boldsymbol{x}}^n = \underset{\hat{\boldsymbol{y}}(t)\in\mathbb{R}^K}{\arg\min}\ \boldsymbol{J}^n(\mathbf{V}\hat{\boldsymbol{y}})$$

- **WLS formulation yields enhanced stability over Galerkin & LSPG**
- **In practice, WLS is solved via discrete least-squares problem**

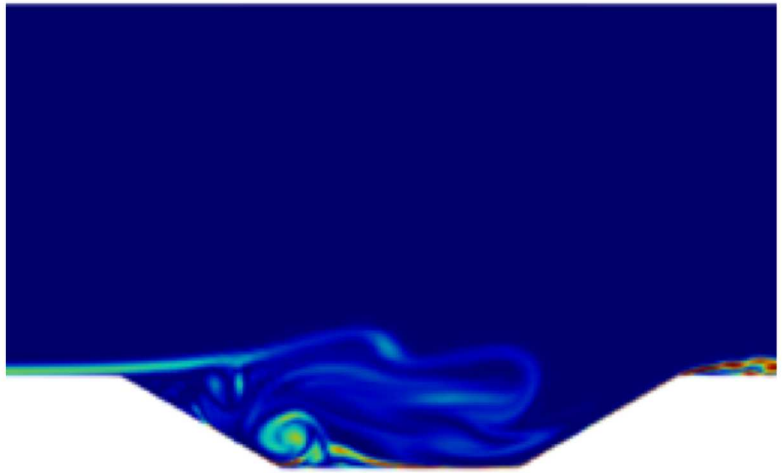
# What is the $\mathbf{A}$? Hyper-reduction

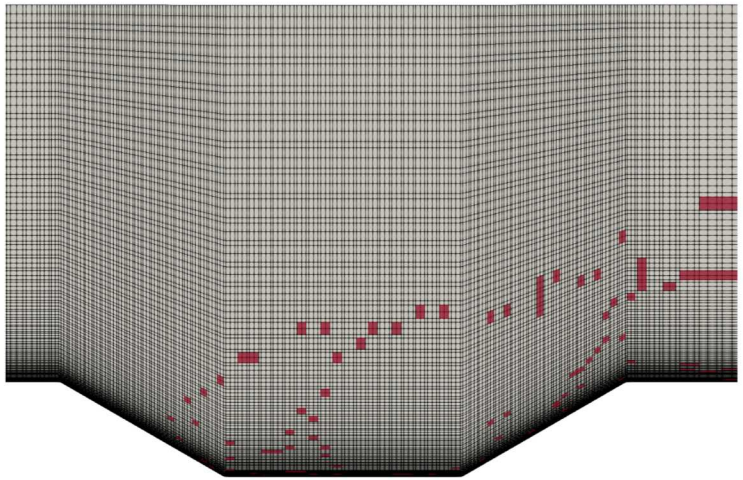
- For nonlinear problems, pROMs are still expensive

$$\boldsymbol{J}^n(\mathbf{V}\hat{\boldsymbol{x}}) = \frac{1}{2}\int_{t_s^n}^{t_f^n} \|\mathbf{V}\dot{\hat{\boldsymbol{x}}}(t) - \boldsymbol{f}(\mathbf{V}\hat{\boldsymbol{x}}, t, \boldsymbol{\mu})\|_{\mathbf{A}}^2$$

- $\mathbf{A} = \mathbf{P}^T\mathbf{P}$ is a "sampling" matrix that addresses this issue

$$\mathbf{P} = \begin{bmatrix} & \blacksquare & & & \\ \blacksquare & & \blacksquare & & \\ & & & \blacksquare & \\ & & & & \blacksquare \end{bmatrix}$$





- Corresponds to evaluating velocity at a subset of the mesh

# Key Challenges

- Projection-based ROMs are intrusive

$$\boldsymbol{J}^n(\boldsymbol{x}) = \frac{1}{2}\int_{t_s^n}^{t_f^n} \|\dot{\boldsymbol{x}}(t) - \boldsymbol{f}(\boldsymbol{x}, t, \boldsymbol{\mu})\|_{\mathbf{A}}^2$$

- Commonly, ROMs are implemented directly in the application code
  - ✗ **Highly intrusive**: changes to application code
  - ✗ **Not extensible**: individual ROM implementation for each application
  - ✗ **Access requirements**: developers need direct access to application

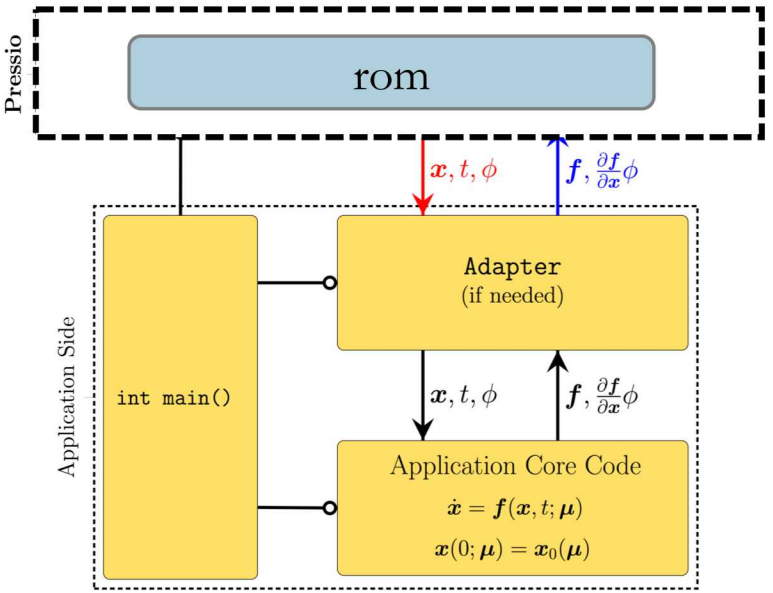
- Motivates ***Pressio***

# Pressio

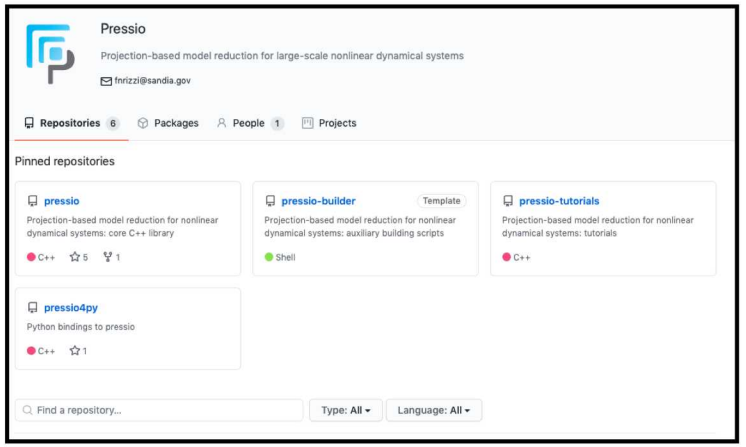
- A computational framework aimed at providing performant pROMs to **generic** application codes

- **Open source code developed at Sandia:**
  - Lead developer: Francesco Rizzi
  - Team includes: Patrick Blonigan, Eric Parish, Kenny Chowdhary, John Tencer, Victor Brunini, Flint Pierce, and more
  - Former developers: Kevin Carlberg and Mark Hoemmen

- **Main idea:**
  - Separate the "application" and the ROM
  - ROM methods are contained in the Pressio framework
  - Pressio "plugs in" to an application code




Rizzi et al, "*Pressio: Enabling projection-based model reduction for large-scale nonlinear dynamical systems*" arXiv:2003.07798



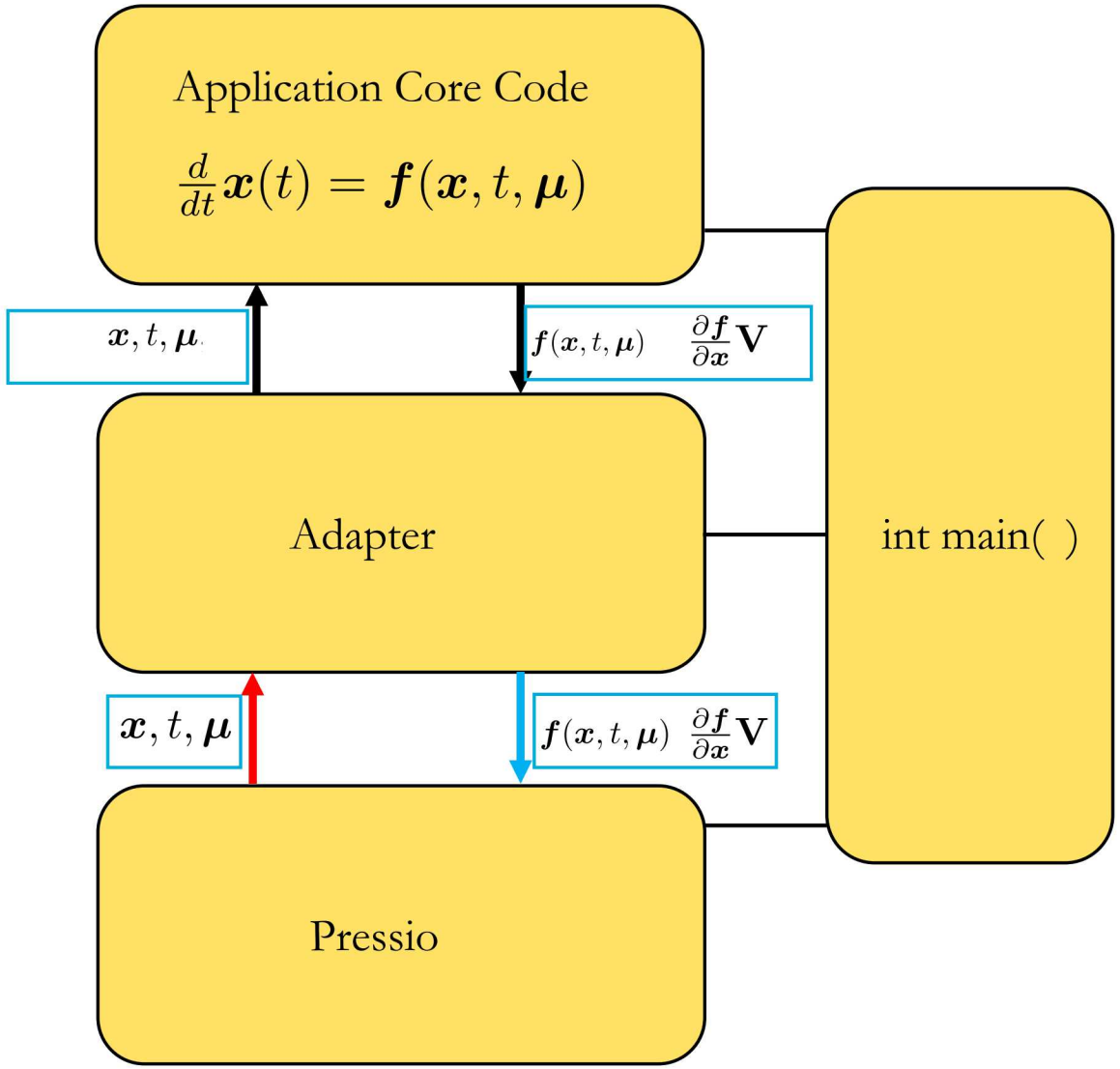

https://github.com/Pressio

# High-level features

Pressio

- Header-only C++11 library
  - Benefits portability
  - Leverages C++11 and metaprogramming for type detection and compile-time dispatching
- Supports HPC performance portability (Kokkos)
- Natively support data structures from Trillinos
  - tPetra
  - tPetraBlock
  - ePetra
- Supports a Python API
  - Enables Python users to use the C++ Pressio functionalities from Python
- Supports Galerkin, LSPG, and WLS ROMs (w/ hyperreduction)

# How does Pressio talk to an application?



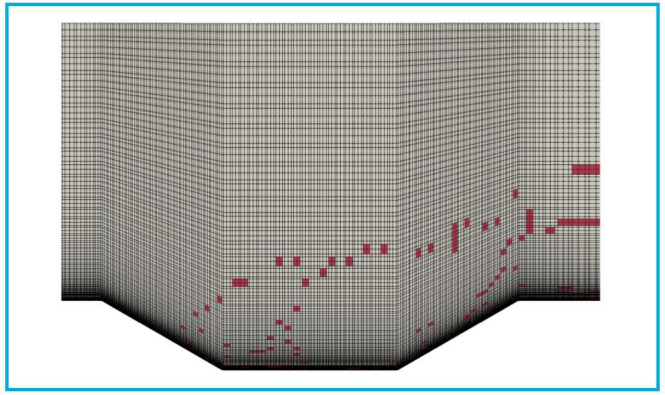

- Pressio's API requires the application to expose two main functions:

1. `velocity`: $\boldsymbol{f}(\boldsymbol{x}, t, \boldsymbol{\mu})$

2. `applyJacobian`: $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\mathbf{V}$

- Pressio uses these functions to construct the ROMs

- For hyperreduction:



# Main.cpp: How do we build and run a ROM?

LISTING 1
*Sample C++ main to create and run WLS using* **Pressio**.

```cpp
int main(int argc, char *argv[]){
  using adapterT        = /*adapter class type*/;
  using scalarT         = typename adapterT::scalar_t;
  using fomStateT       = typename adapterT::state_t;
  using decoderJacT     = typename adapterT::dmatrix_t;
  using romStateT       = /*ROM state type*/;
  using namespace pressio;

  // create app or adapter object
  adapterT fomObj(argc, argv, /*other args needed*/);

  const std::size_t romSize = /*set the Rom size */;
  const std::size_t numStepsInWindow = /*set the number of stpes in a window */;
  const std::size_t nWindows = /*set the number of windows */;

  // create the decoder, e.g. linear decoder
  decoderJacT phi = /*load/compute the decoder's Jacobian*/;
  using decoderT  = rom::LinearDecoder<decoderJacT>;
  decoderT decoderObj(phi);

  // define ROM state
  romStateT romState(romSize);

  // create the WLS problem
  using stepperTag = ode::implicitmethods::BDF2;
  using rom::wls::DefaultProblem;
  using wlsT = DefaultProblem<stepperTag, romStateT,
                              adapterT, decoderT>;
  wlsT wlsProb(fomObj, numStepsInWindow, decoderObj, romState);

  // advance in time
  advanceNWindows(wlsProb, romState, t0, dt, nWindows);
  // map the final ROM state to the corresponding fom
  auto xFomFinal = wlsProblem.fomReconstructor()(romState);

  return 0;
}
```

Type detection

Create the FOM object

Load/compute basis vectors

Create the ROM problem

Advance in time

# Example application of Pressio: Overview

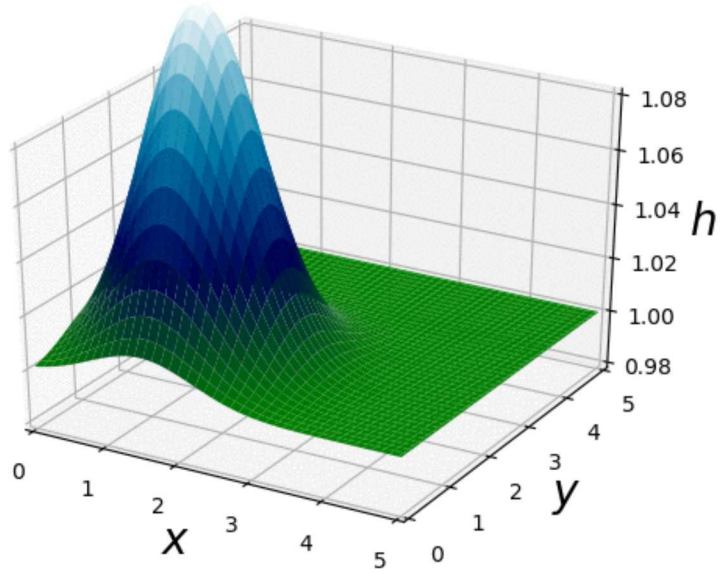
- Construct a ROM of the shallow water equations

$$\frac{\partial h}{\partial t} + \frac{\partial}{\partial x}(hu) + \frac{\partial}{\partial y}(hv) = 0$$
$$\frac{\partial hu}{\partial t} + \frac{\partial}{\partial x}(hu^2 + \frac{1}{2}\mu_1 h^2) + \frac{\partial}{\partial y}(huv) = 0$$
$$\frac{\partial hv}{\partial t} + \frac{\partial}{\partial x}(huv) + \frac{\partial}{\partial y}(hv^2 + \frac{1}{2}\mu_1 h^2) = 0$$

- Initial conditions

$$h(x, y, 0) = 1 + \mu_2 e^{(x-1.5)^2+(y-1.5)^2}$$
$$u(x, y, 0) = v(x, y, 0) = 0$$

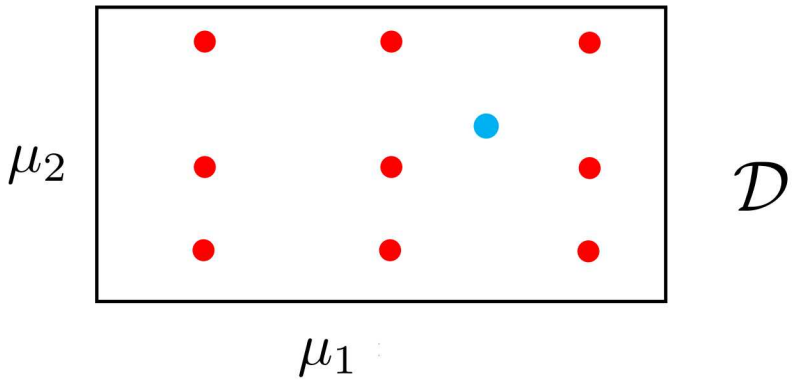
- System parameters:
  - $\mu_1$ : gravity parameter
  - $\mu_2$ : controls the magnitude of the pulse




*Surface plot of the water height*

https://github.com/eparish1/pressio_wls_tutorial

# Example application of Pressio




- Have access to an app that solves the shallow water equations
  - 1st order finite volume scheme
  - 128 x 128 mesh
  - tPetraBlock data structures
  - Capable of evaluating residual at a subset of the mesh
  - Takes 20 seconds to run one simulation

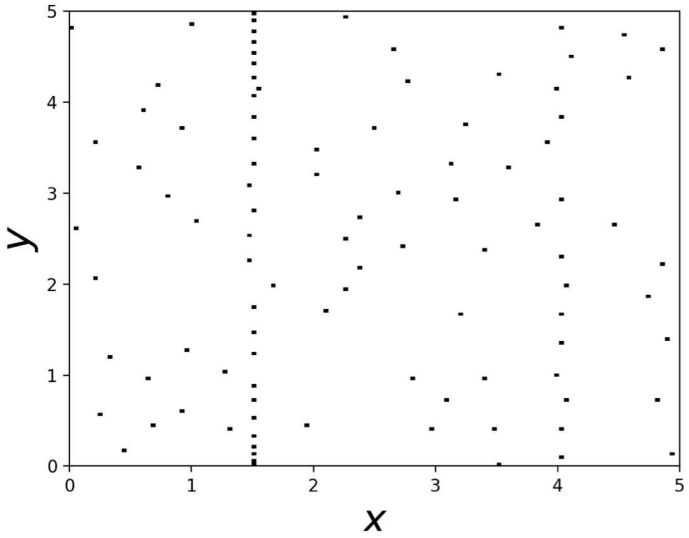
- Execute FOM for training parameter instances

$$\mu_1 = \{3, 6, 9\}$$
$$\mu_2 = \{0.05, 0.1, 0.2\}$$

- Post process FOM data for basis vectors and sample mesh

- Execute the ROM for a test parameter instance

$$\mu_1 = 7.5,\ \mu_2 = 0.125$$

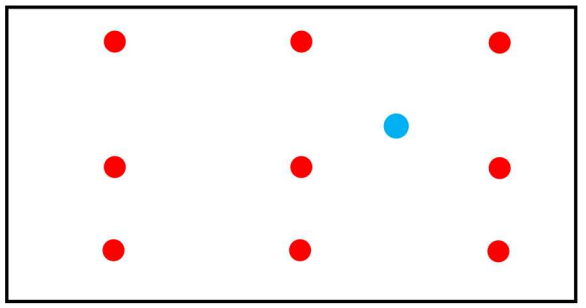



*Sample mesh*

https://github.com/eparish1/pressio_wls_tutorial

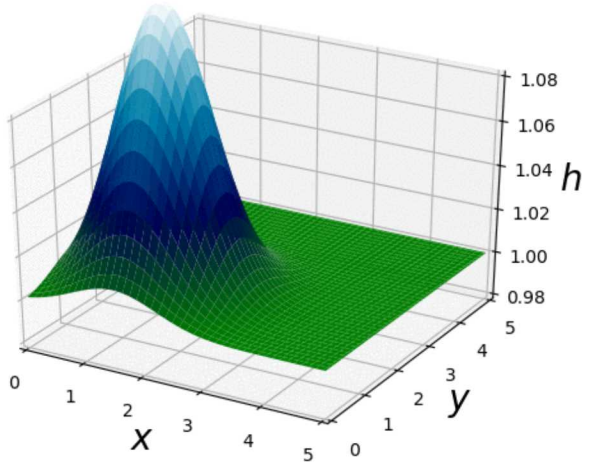
# Results

- Results at a novel parameter instance
  - 0.3% error
  - 3x speed up over FOM
  - 6x better than nearest neighbors solution

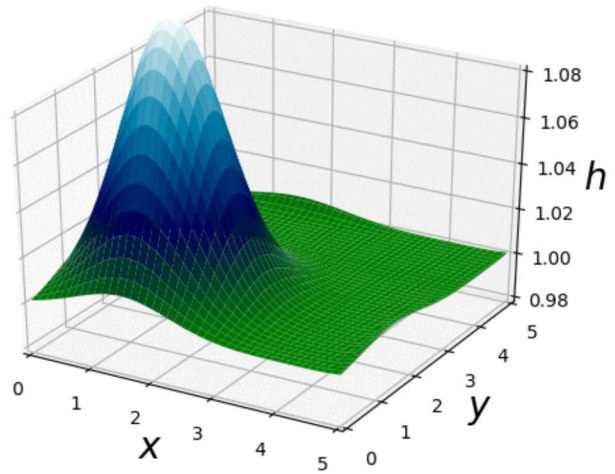


$\mathcal{D}$

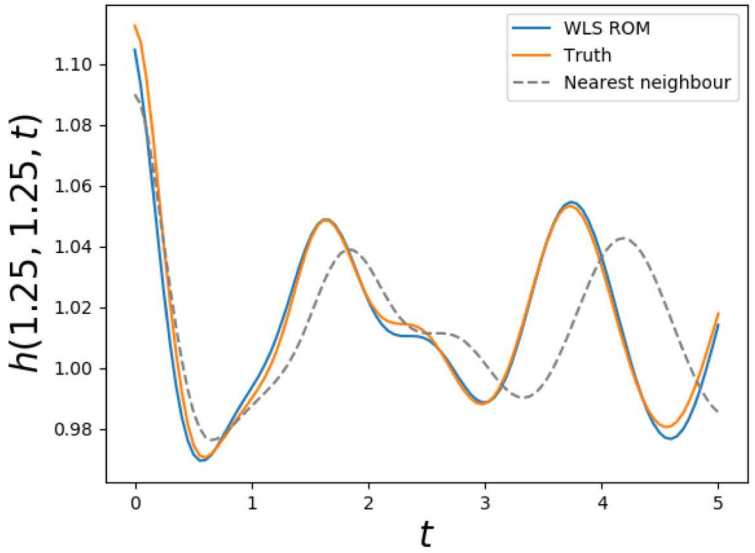
- Get more speed benefit as the FOM dimension grows




*FOM solution*




*ROM solution*



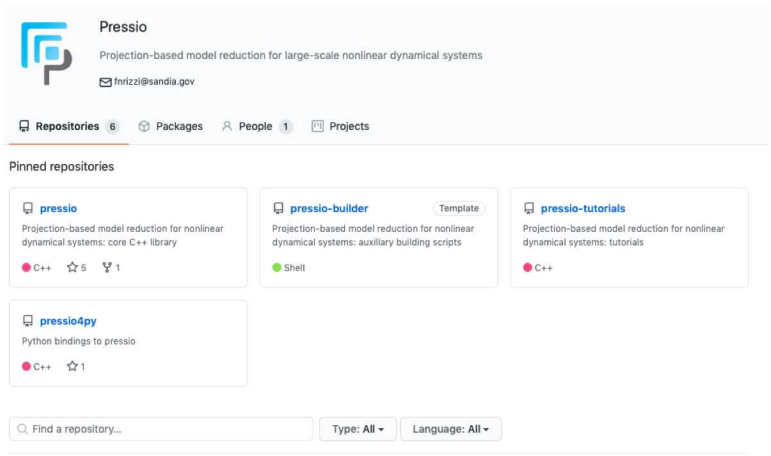

*Temporal evolution of water height*

https://github.com/eparish1/pressio_wls_tutorial

# Conclusions

- Projection-based ROMs enable fast and efficient approximate solutions
- The windowed-least squares formulation enables robust solutions
- Pressio is an open-source framework developed at Sandia aimed at providing performant pROMs
  - Freely available on github
  - Header only C++11 library
  - Supports arbitrary datatypes and HPC programming models
  - Supports a Python API (WLS is ongoing)
- We are growing the Pressio project
  - Coupling to new application codes
  - Adding new solvers and ROM techniques
  - Improving user interfaces and APIs
- **We are looking for collaborators!**

# Software Points of Contact




https://github.com/Pressio

https://github.com/eparish1/pressio_wls_tutorial

- Contact email for Pressio: fnrizzi@sandia.gov
- Contact email for WLS implementation: ejparis@sandia.gov
- Pressio paper: Rizzi et al., "*Pressio: Enabling projection-based model reduction for large-scale nonlinear dynamical systems*" arXiv:2003.07798

# Example application of Pressio

- Construct ROM basis from FOM data
  - External to Pressio
- For hyper-reduction
  - Identify FOM cells at which to evaluate the residual
- Write main file for ROM application using Pressio
  - Read in basis
  - Declare problem information
- Execute ROM app and post process data
  - Post processing is external to pressio

LISTING 1
*Sample C++ main to create and run WLS using* **Pressio**.

```
1  int main(int argc, char *argv[]){
2    using adapterT        = /*adapter class type*/;
3    using scalarT         = typename adapterT::scalar_t;
4    using fomStateT       = typename adapterT::state_t;
5    using decoderJacT     = typename adapterT::dmatrix_t;
6    using romStateT       = /*ROM state type*/;
7    using namespace pressio;
8
9    // create app or adapter object
10   adapterT fomObj(argc, argv, /*other args needed*/);
11
12   const std::size_t romSize = /*set the Rom size */;
13   const std::size_t numStepsInWindow = /*set the number of stpes in a window */;
14   const std::size_t nWindows = /*set the number of windows */;
15
16   // create the decoder, e.g. linear decoder
17   decoderJacT phi = /*load/compute the decoder's Jacobian*/;
18   using decoderT  = rom::LinearDecoder<decoderJacT>;
19   decoderT decoderObj(phi);
20
21   // define ROM state
22   romStateT romState(romSize);
23
24   // create the WLS problem
25   using stepperTag = ode::implicitmethods::BDF2;
26   using rom::wls::DefaultProblem;
27   using wlsT = DefaultProblem<stepperTag, romStateT,
28                               adapterT, decoderT>;
29   wlsT wlsProb(fomObj, numStepsInWindow, decoderObj, romState);
30
31   // advance in time
32   advanceNWindows(wlsProb, romState, t0, dt, nWindows);
33   // map the final ROM state to the corresponding fom
34   auto xFomFinal = wlsProblem.fomReconstructor()(romState);
35
36   return 0;
37 }
```

*Sample "main" file for WLS in pressio*

# How does Pressio talk to an application?

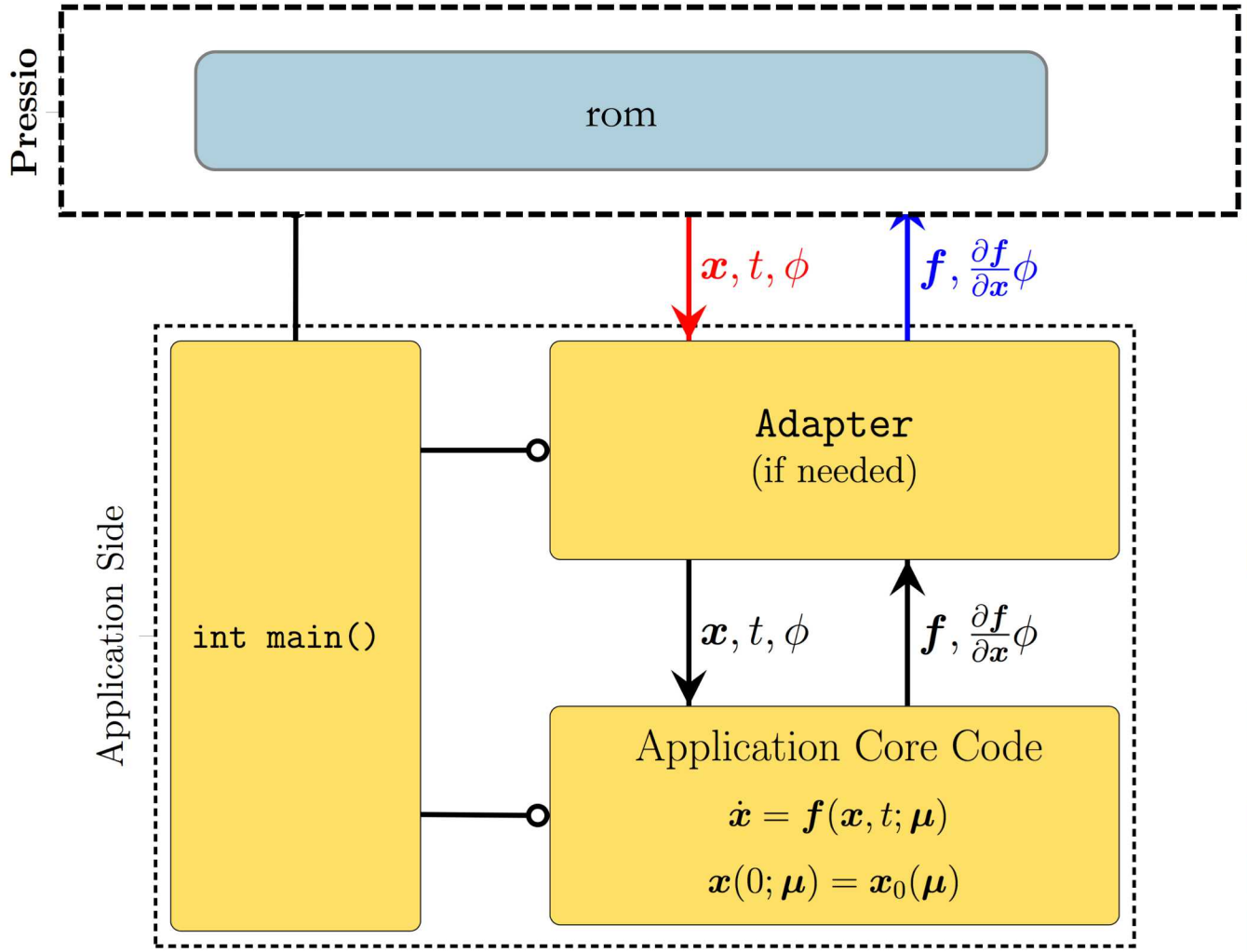
- Pressio's API requires the application to expose two main functions:

1. `velocity`: $\boldsymbol{f}(\boldsymbol{x}, t, \boldsymbol{\mu})$

2. `applyJacobian`: $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\boldsymbol{\Phi}$

- Pressio uses these functions to construct the ROMs

- For hyper-reduction, we require `velocity` and `applyJacobian` to return results at only a subset of the mesh



# Trial subspaces: some details

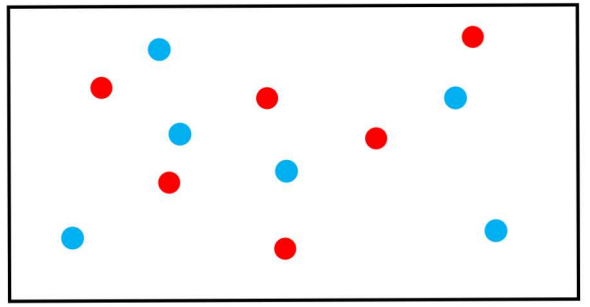
- Step 2: Identify low-dimensional structure within the training data
  - Results in identifying a low-dimensional subspace

Trial space: $\mathcal{V} \subset \mathbb{R}^N$

$$\dim(\mathcal{V}) = K, \ (K \ll N)$$

Trial basis: $\mathbf{V} \in \mathbb{R}^{N \times K}, \ \mathbf{V}^T\mathbf{V} = \mathbf{I}$

$$\text{Range}(\mathbf{V}) = \mathcal{V}$$



$\mathcal{D}$

$\boldsymbol{x}_1(t, \boldsymbol{\mu}_1), \boldsymbol{x}_2(t, \boldsymbol{\mu}_2), \ldots, \boldsymbol{x}_s(t, \boldsymbol{\mu}_s)$

- Results in the approximation: $\boldsymbol{x}(t) \approx \mathbf{V}\hat{\boldsymbol{x}}(t)$

Generalized coordinates: $\hat{\boldsymbol{x}}(t) \in \mathbb{R}^K$