Arby

Aarón Villanueva

Sep 12, 2021

CONTENTS

1 Installation **3**

2 Tutorial **5**
- 2.1 Simple pendulum: build a surrogate model . . . 5
  - 2.1.1 Solving the pendulum equations . . . 6
  - 2.1.2 Building the surrogate . . . 7
  - 2.1.3 Benchmarks . . . 9
- 2.2 Build a reduced basis . . . 11
- 2.3 Further reading . . . 12

3 Module API **15**
- 3.1 Module `arby.rom` . . . 15
- 3.2 Module `arby.basis` . . . 17
- 3.3 Module `arby.integrals` . . . 19

4 Quick Usage **21**

Bibliography **23**

Python Module Index **25**

Index **27**

Arby is a fully data-driven Python module to construct surrogate models, reduced bases and empirical interpolants from training data.

This module implements a type of Reduced Order Modeling technique for reducing the computational complexity of mathematical models in numerical simulations.

CHAPTER

ONE

INSTALLATION

To install the latest stable version of Arby from PyPI:

```
$ pip install arby
```

To install the developer version (may be unstable):

```
$ git clone https://github.com/aaronuv/arby.git
$ cd arby
$ pip install .
```

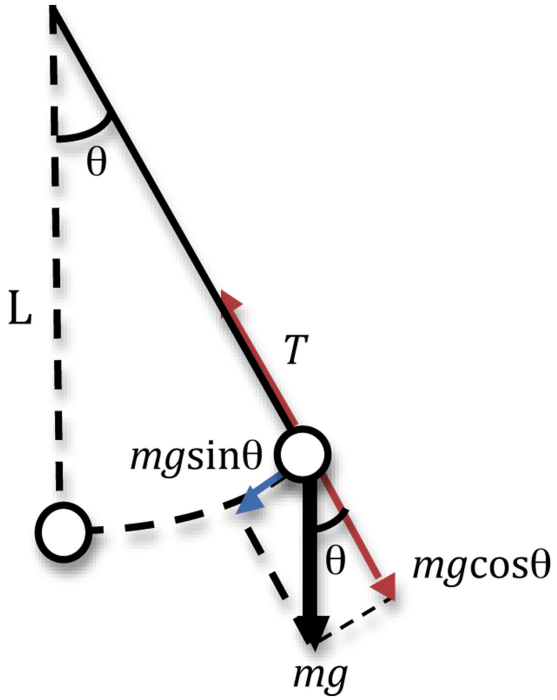
CHAPTER

TWO

TUTORIAL

2.1 Simple pendulum: build a surrogate model

In this tutorial we build a surrogate model for the damped simple pendulum using Arby. The physical setting is shown below.



This system can be described by the ordinary differential equation (ODE)

$$\ddot{\theta} = -b\dot{\theta} - \frac{g}{l}\sin\theta \tag{2.1}$$

where g, l denote the gravity strenght and the pendulum longitude, respectively. The parameter b is the damping factor.

We can decompose the 2nd order ODE above in two 1st order ODEs leading to the desired pendulum equations

$$\dot{\theta} = \omega \tag{2.2}$$

$$\dot{\omega} = -b\omega - \lambda\sin\theta \tag{2.3}$$

2.1.1 Solving the pendulum equations

We import first some Python packages for handling and visualizing data.

[2]:
```
import numpy as np
```

[3]:
```
import matplotlib.pyplot as plt
```

Also we need an ODE solver. We use a module from the SciPy Python package.

[4]:
```
from scipy.integrate import odeint
```

Let's define the ODE system and fix some initial conditions

[5]:
```
def pend(y, t, b, ):
    ,  = y
    dydt = [, -b* - *np.sin()]

    return dydt
```

[6]:
```
# set friction strength
b = 0.2
# set initial conditions
y0 = [np.pi/2, 0.]
```

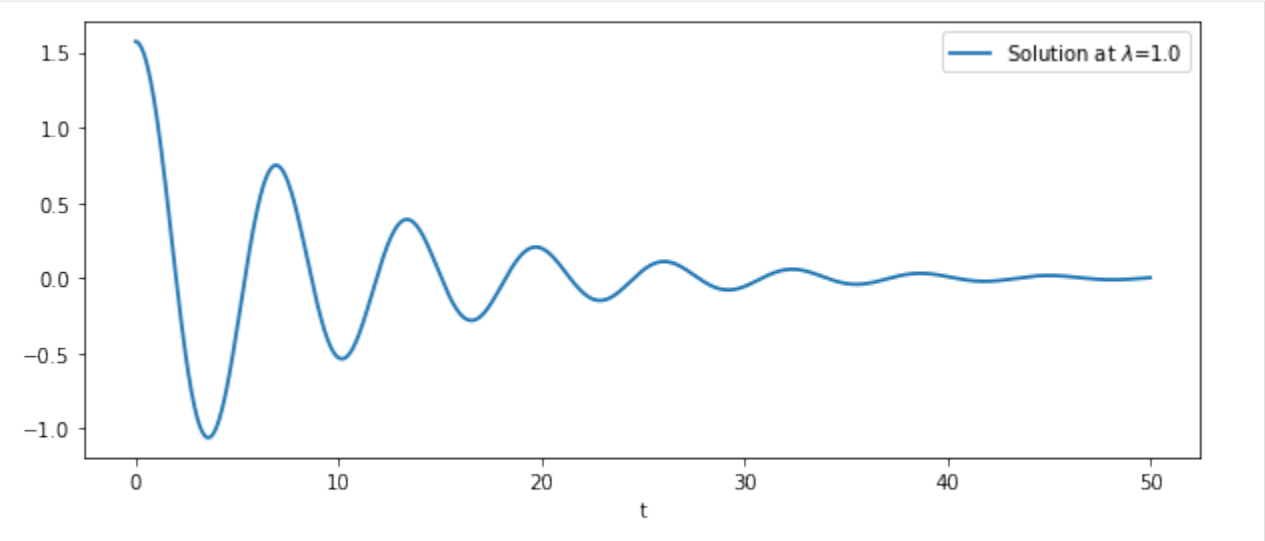
[7]:
```
# set a time discretization
times = np.linspace(0,50,1001)
```

[8]:
```
# plot a simple solution
 = 1.
sol = odeint(pend,y0, times, (b,))
```

Lets see how a solution looks

[9]:
```
plt.figure(figsize=(10,4))
plt.plot(times, sol[:,0], label=f'Solution at $\lambda$={}', lw=1.8)
plt.xlabel('t')
plt.legend()
```

[9]:
```
<matplotlib.legend.Legend at 0x7f3ea40e6a60>
```



2.1.2 Building the surrogate

Import the Arby main class for building surrogates

[11]:
```
from arby import ReducedOrderModel as ROM
```

and define a discretization of the parametric domain. We chose the domain for λ as $[1, 5]$ and a discretization of 101 points.

[15]:
```
param = np.linspace(1,5,101)
```

Lets build a training set.

[16]:
```
training = []
for  in param:
    sol = odeint(pend,y0, times, (b,))
    training.append(sol[:,0])
```

So far we have the main ingredients for building a surrogate for the pendulum system. Finally, create a pendulum model as a ROM object

[17]:
```
pendulum = ROM(training, times, param, greedy_tol=1e-14, poly_deg=5)
```

and simply call/build the surrogate for some parameter value

[18]:
```
pendulum.surrogate(1.14)
```

[18]:
```
array([ 1.57079633,  1.56937605,  1.56513412, ..., -0.00282289,
       -0.00331644, -0.00379569])
```

That's all! We built a surrogate model for pendulum solutions. Now, for the sake of brevity, name the surrogate function.

[19]:
```
surr = pendulum.surrogate
```

We want the surrogate to produce predictions. So lets look for an out-of-sample parameter to build a prediction.

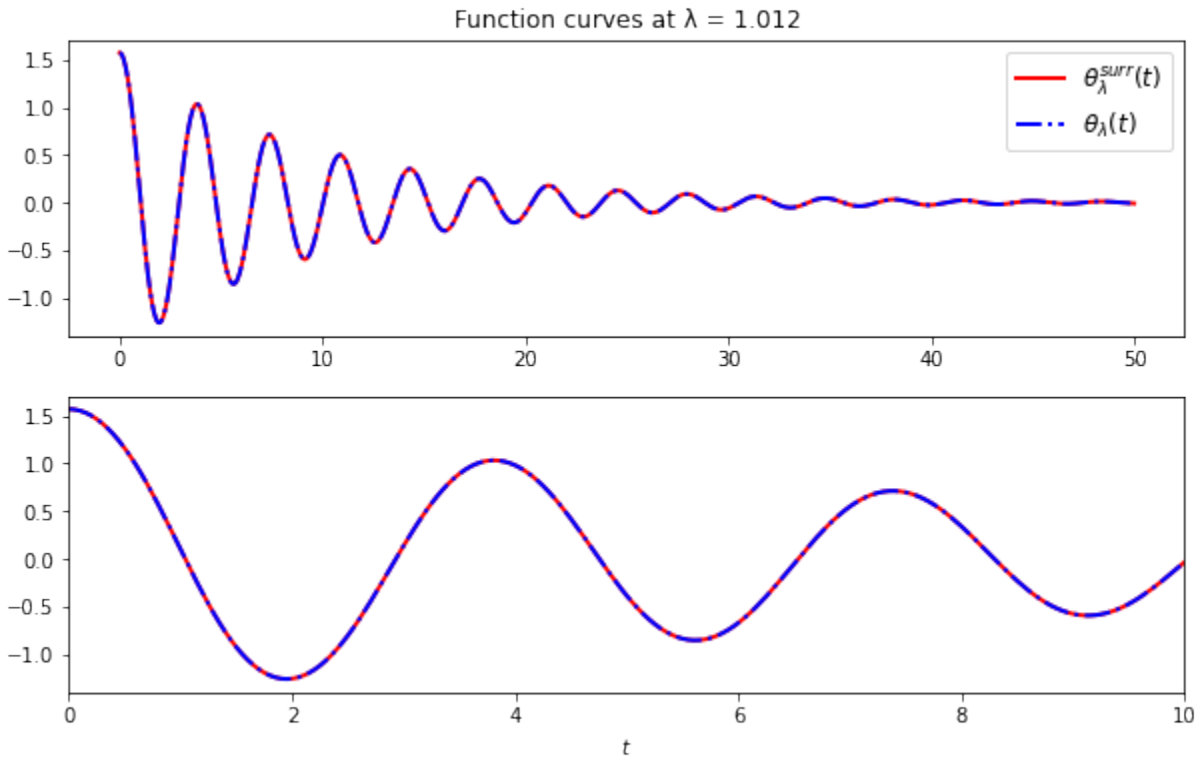
[20]:
```
param
```

[20]:
```
array([1.  , 1.04, 1.08, 1.12, 1.16, 1.2 , 1.24, 1.28, 1.32, 1.36, 1.4 ,
       1.44, 1.48, 1.52, 1.56, 1.6 , 1.64, 1.68, 1.72, 1.76, 1.8 , 1.84,
       1.88, 1.92, 1.96, 2.  , 2.04, 2.08, 2.12, 2.16, 2.2 , 2.24, 2.28,
       2.32, 2.36, 2.4 , 2.44, 2.48, 2.52, 2.56, 2.6 , 2.64, 2.68, 2.72,
       2.76, 2.8 , 2.84, 2.88, 2.92, 2.96, 3.  , 3.04, 3.08, 3.12, 3.16,
       3.2 , 3.24, 3.28, 3.32, 3.36, 3.4 , 3.44, 3.48, 3.52, 3.56, 3.6 ,
       3.64, 3.68, 3.72, 3.76, 3.8 , 3.84, 3.88, 3.92, 3.96, 4.  , 4.04,
       4.08, 4.12, 4.16, 4.2 , 4.24, 4.28, 4.32, 4.36, 4.4 , 4.44, 4.48,
       4.52, 4.56, 4.6 , 4.64, 4.68, 4.72, 4.76, 4.8 , 4.84, 4.88, 4.92,
       4.96, 5.  ])
```

We choose `par = 3.42`. Now compare prediction and true solution.

[21]:
```
par = 3.42
sol = odeint(pend,y0, times, (b,par))[:,0]
fig, ax = plt.subplots(2,1, figsize=(10,6))
ax[0].plot(times, surr(par), 'r', lw=2, label='$_^{surr}(t)$')
ax[0].plot(times, sol, 'b-.', lw=2, label='$_(t)$')
ax[1].plot(times, surr(par), 'r', lw=2, label='$_^{surr}(t)$')
ax[1].plot(times, sol, 'b-.', lw=2, label='$_(t)$')
ax[1].set(xlim=(0,10))
ax[1].set(xlabel='$t$')
ax[0].set_title('Function curves at  = 1.012')
ax[0].legend(fontsize = 'large')
```

[21]:
```
<matplotlib.legend.Legend at 0x7f23db2fe670>
```



Nice! At least they match at eyeball resolution. How fast is it with respect to solving the whole ODE system for some

parameter?

To respond that we perform a simple profiling for evaluating computation times for both, surrogate and ODE solver

[22]:
```
 = 2.17
%timeit -t sol = odeint(pend,y0, times, (b,))
```

```
5.84 ms ± 389 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[23]:
```
%timeit -t pendulum.surrogate()
```

```
1.41 ms ± 95.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

The surrogate is ~ 4 times faster than solving the ODE system.

2.1.3 Benchmarks

We'd like to quantify the accuracy of the surrogate. To this, we perform a benchmark measuring the L_2 relative error between the surrogate and a dense validation or test set of true solutions

$$e(\lambda)^2 := \frac{\|\theta_\lambda^{surr} - \theta_\lambda^{true}\|^2}{\|\theta_\lambda^{true}\|^2}$$

where

$$\|\theta\|^2 := \int \theta^2(t)dt$$

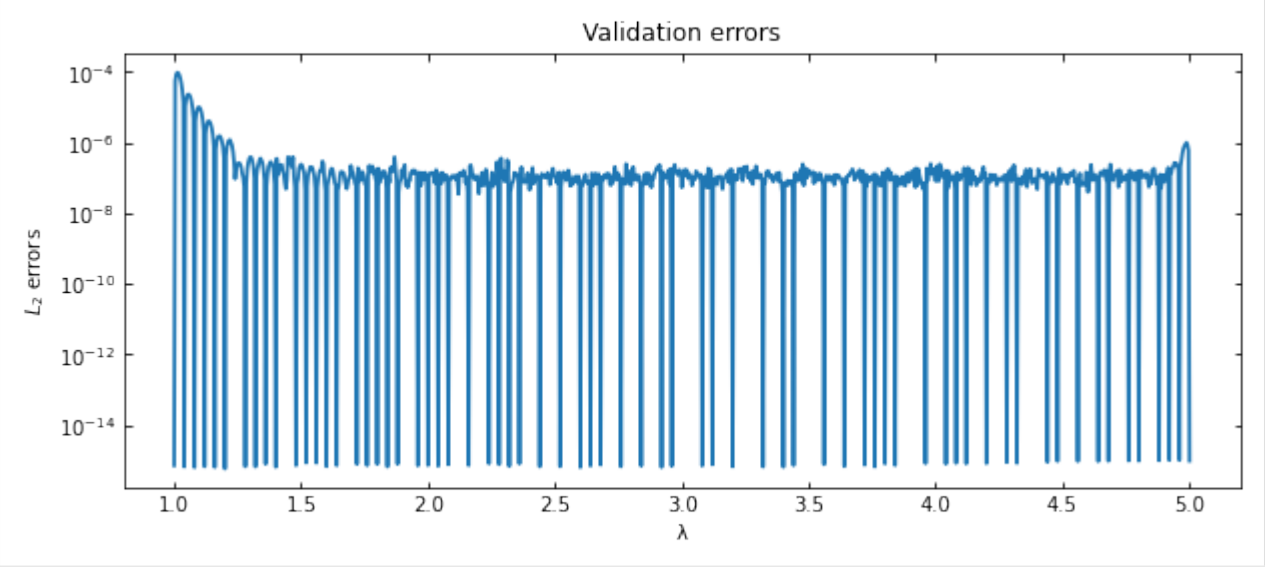
[25]:
```
# name the norm tool from the integration object
norm = pendulum.basis_.integration.norm
# create a dense validation set (10x the size of the training set)
param_val = np.linspace(1,5,1001)
# compute errors
errors = []
for  in param_val:
    sol = odeint(pend,y0, times, (b,))[:,0]
    errors.append(norm(sol - surr())/norm(sol))
```

Plot the errors

[26]:
```
fig, ax = plt.subplots(figsize=(10,4))

ax.plot(param_val, errors, lw=1.8)
ax.set_yscale('log')
ax.yaxis.set_ticks_position('both')
ax.tick_params(direction='in', which='both')
ax.xaxis.set_ticks_position('both')
ax.set_xlabel('')
ax.set_ylabel('$L_2$ errors')
ax.set_title('Validation errors')
```

[26]:
```
Text(0.5, 1.0, 'Validation errors')
```

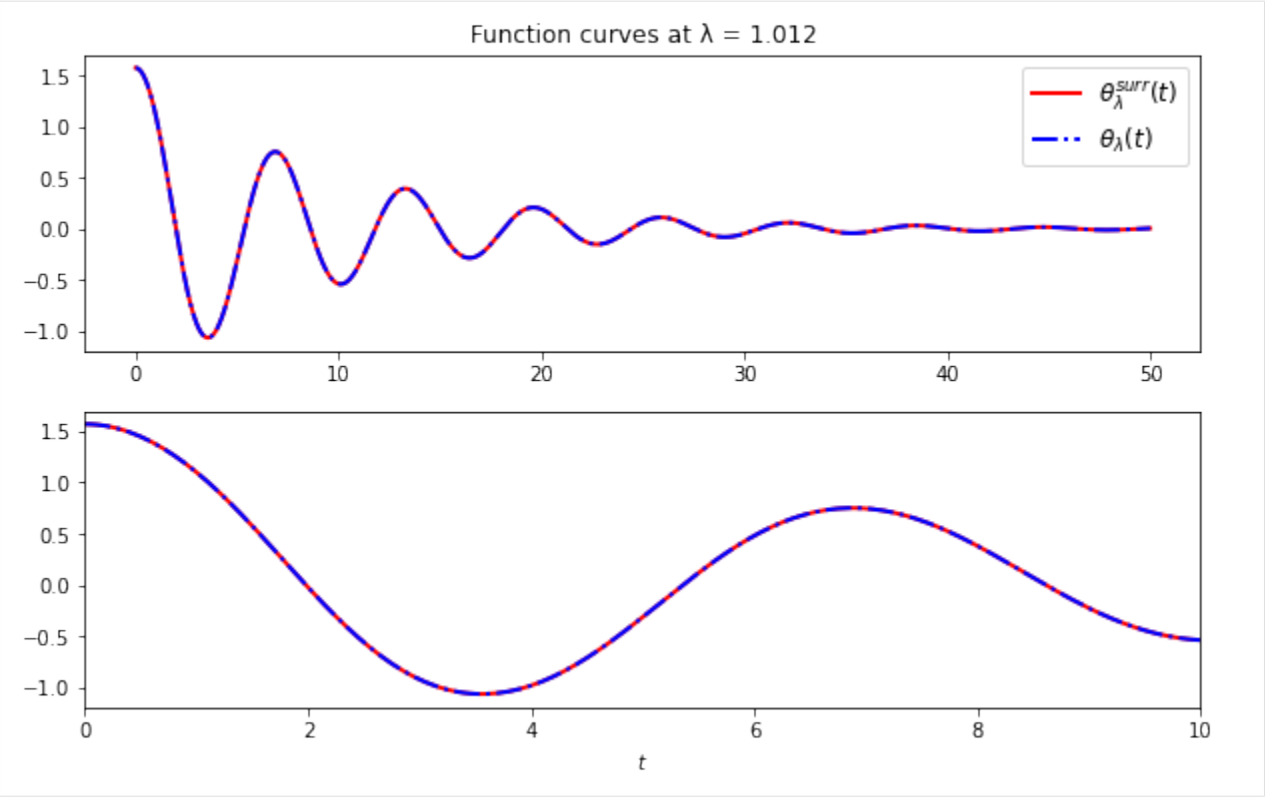


Which is the parameter corresponding to the largest error?

[27]:
```
worst_lambda = param_val[np.argmax(errors)]
```

[28]:
```
# plot the overlap between the surrogate and the true solution model
# at the worst parameter
par = worst_lambda
sol = odeint(pend,y0, times, (b,par))[:,0]
fig, ax = plt.subplots(2,1, figsize=(10,6))
ax[0].plot(times, surr(par), 'r', lw=2, label='$_^{surr}(t)$')
ax[0].plot(times, sol, 'b-.', lw=2, label='$_(t)$')
ax[1].plot(times, surr(par), 'r', lw=2, label='$_^{surr}(t)$')
ax[1].plot(times, sol, 'b-.', lw=2, label='$_(t)$')
ax[1].set(xlim=(0,10))
ax[1].set(xlabel='$t$')
ax[0].set_title(f'Function curves at  = {worst_lambda}')
ax[0].legend(fontsize = 'large')
```

[28]:
```
<matplotlib.legend.Legend at 0x7f23d9cbb4c0>
```



The curves are indistinguishable to eyeball resolution even for the worst case scenario.

2.2 Build a reduced basis

Lets go deeper. The Reduced Basis Method is a reduced order modeling technique for building a near-optimal basis of functions that spans the training set at an user-specified tolerance. The basis is built by iteratively choosing those training functions which best represent the entire set. In this way, as opposed to other dimensional reduction techniques such as Proper Orthogonal Decomposition, the reduced basis is directly interpretable since it is built out from training functions. Another kindness of this approach is that whenever we want more accuracy we just add more basis elements to the computed one: the construction is hierarchical.

Suppose we have a training set $\{f_{\lambda_i}\}_{i=1}^N$ of parameterized real functions. This set may represent a non-linear model, perhaps solutions to PDEs. We'd like, if possible, to reduce the dimensionality/complexity of it by finding a compact representation in terms of linear combinations of basis elements $\{e_i\}_{i=1}^n$, that is,

$$f \approx \sum_{i=1}^{n} c_i e_i \, .$$

f is an arbitrary training function and the c_i's are the projection coefficients $< e_i, f >$ computed in some inner product $< \cdot, \cdot >$ on the space of functions. The RB method chooses a set of optimal functions belonging to the training set itself which defines a finite dimensional subspace capable to represent the entire training set up to a user-specified tolerance.

To build a reduced basis with Arby, you just provide the training set of functions and the discretization of the physical variable x to the `reduced_basis` function. The later is to define the integration scheme that is used for computing inner products. For the pendulum example,

```
[ ]: from arby import reduced_basis

rb_data = reduced_basis(training_set=training,
                        physical_points=times, greedy_tol=1e-12)
```

The `greedy_tol` parameter is the accuracy in the L_2-norm that our reduced basis is expected to achieve. The output `rb_data` contains all the relevant information related to greedy calculations. It contains a `basis` object which comprises the reduced basis and several utilities for interacting with it. The other outputs are the greedy `errors` and `indices`, and the `projection_matrix`, which stores projection coefficients built in the greedy algorithm. For example, to call the reduced basis array do

```
[ ]: rb_data.basis.data
```

The reduced basis is an orthonormalized version of the set of functions selected by the greedy algorithm and indexed by the greedy indices. You can obtain those functions by filtering the training set

```
[ ]: training_set[rb_data.indices]
```

For conditioning purposes, the greedy algorithm orthonormalizes the basis.

The number of basis elements `rb_data.basis.Nbasis_` represents the dimension of the reduced space. It is not a fixed quantity since we can change it by modifying the greedy tolerance. The lower the tolerance, the bigger the number of basis elements needed to reach that accuracy. With Arby, we can tune the reduced basis accuracy through the `greedy_tol` parameter.

To quantify the representation effectiveness of the reduced basis to approximate a solution `f`, we can compute the norm of the difference between a training function `f` and its projected version using the tools coming inside the `rb_data.basis` class object.

```
[ ]: projected_f = rb_data.basis.project(f)
norm = rb_data.basis.integration.norm
L2_error = norm(f - projected_f)
```

Or take a shortcut by doing

```
[ ]: projection_error = rb_data.basis.projection_error
squared_L2_error = projection_error(f)
```

The output is the square version of the error computed in the previous code block.

2.3 Further reading

- This package was built upon previous work developed in the field of Gravitational Wave (GW) science. Important papers are
  - Reduced basis catalogs for gravitational wave templates
  - Towards beating the curse of dimensionality for gravitational waves using Reduced Basis
  - Fast prediction and evaluation of gravitational waveforms using surrogate models
- For some theoretical aspects on the Reduced Basis greedy algorithm and the Empirical Interpolation Method check
  - Two-step greedy algorithm for reduced order quadratures
  - A Note on QR-Based Model Reduction: Algorithm, Software, and Gravitational Wave Applications

  - On the stability and accuracy of the Empirical Interpolation Method and Gravitational Wave Surrogates
- For a thorough review on Reduced Order Methods as implemented in this package, check
  - Reduced Order and Surrogate Models for Gravitational Waves
- For nice visuals on the application of surrogate modeling in GW Science check this.

CHAPTER

THREE

MODULE API

3.1 Module arby.rom

Reduced Order Modeling module.

class arby.rom.**ReducedOrderModel**(*training_set*, *physical_points*, *parameter_points*, *integration_rule: str = 'riemann'*, *greedy_tol: float = 1e-12*, *poly_deg: int = 3*)

Build reduced order models from training data.

This class comprises a set of tools to build and handle reduced bases, empirical interpolants and predictive models from pre-computed training set of functions. The underlying or ground truth model describing the training set is a real function g(v,x) parameterized by a *training* parameter v. The *physical* variable x belongs to a domain for which an inner product can defined. The surrogate model is built bringing together the Reduced Basis (RB) greedy algorithm and the Empirical Interpolation Method (EIM) to work in synergy towards a predictive model for the ground truth model.

Parameters

- **training_set** (*array_like*) – Array of training functions.
- **physical_points** (*array_like*) – Array of physical points.
- **parameter_points** (*array_like*) – Array of parameter points.
- **integration_rule** (*str, optional*) – The quadrature rule to define an integration scheme. Default = "riemann".
- **greedy_tol** (*float, optional*) – The greedy tolerance as a stopping condition for the reduced basis greedy algorithm. Default = 1e-12.
- **poly_deg** (*int, optional*) – Degree <= 5 of polynomials used for splines. Default = 3.

Examples

Build a surrogate model

```
>>> from arby import ReducedOrderModel as ROM
```

Input the three most important parameters (the others are optional).

```
>>> model = ROM(training_set, physical_points, parameter_points)
```

Build/evaluate the surrogate model. The building stage is done once and for all at the first call. It could take some time for large training sets. For this reason it is called the *offline* stage. Subsequent calls will invoke the already built surrogate model and just evaluates it. That corresponds to the *online* stage.

```
>>> model.surrogate(parameter)
```

For attempting to improve the model's accuracy without the addition of more training functions, tune the class parameters `greedy_tol` and `poly_deg` to control the precision of the reduced basis (see the `arby.reduced_basis` method) or the internal splines model.

property Ntrain_
Return the number of training functions or parameter points.

property Nsamples_
Return the number of samples or physical points.

property basis_
Return a reduced basis object.

The reduced basis is computed at the first call and stored as a class object of `arby.Basis`, which comprises several tools for handling bases. See also the `arby.reduced_basis` documentation.

property greedy_indices_
Greedy indices from the RB algorithm.

See the `arby.reduced_basis` documentation.

property greedy_errors_
Errors computed in the RB algorithm.

See the `arby.reduced_basis` documentation.

property projection_matrix_
Projection coefficients from the RB algorithm.

See the `arby.reduced_basis` documentation.

property eim_
Return EIM data.

See `arby.Basis.eim_` documentation.

surrogate(*param*)
Evaluate the surrogate model at parameter/s.

Build a surrogate model valid for the entire parameter domain. The building stage is performed only once for the first function call. For subsequent calls, the method invokes the already fitted model and just evaluates it. The output is an array storing surrogate evaluations at the parameter/s.

Parameters **param** (*float or array_like(float)*) – Point or set of parameters inside the parameter domain.

Returns **h_surrogate** – The evaluated surrogate function for the given parameters.

Return type numpy.ndarray

3.2 Module arby.basis

Basis analysis module.

class arby.basis.**EIM**(*interpolant: numpy.ndarray*, *nodes: list*)
Container for EIM information.

Parameters

- **interpolant** (*numpy.ndarray*) – Interpolant matrix.
- **nodes** (*list*) – EIM nodes.

class arby.basis.**RB**(*basis: numpy.ndarray*, *indices: numpy.ndarray*, *errors: numpy.ndarray*, *projection_matrix: numpy.ndarray*)
Container for RB information.

Parameters

- **basis** (*np.ndarray*) – Reduced basis object.
- **indices** (*np.ndarray*) – Greedy indices.
- **errors** (*np.ndarray*) – Greedy errors.
- **projection_matrix** (*np.ndarray*) – Projection coefficients.

class arby.basis.**Basis**(*data*, *integration: numpy.ndarray*)
Basis object and utilities.

Create a basis object introducing an orthonormalized set of functions `data` and an `integration` class instance to enable integration utilities for the basis.

Parameters

- **data** (*numpy.ndarray*) – Orthonormalized basis.
- **integration** (arby.integrals.Integration) – Instance of the `Integration` class.

property Nbasis_: int
Return the number of basis elements.

property eim_: arby.basis.EIM
Implement EIM algorithm.

The Empirical Interpolation Method (EIM) [TiglioAndVillanueva2021] introspects the basis and selects a set of interpolation `nodes` from the physical domain for building an `interpolant` matrix using the basis and the selected nodes. The `interpolant` matrix can be used to approximate a field of functions for which the span of the basis is a good approximant.

Returns Container for EIM data. Contains (`interpolant`, `nodes`).

Return type *arby.basis.EIM*

projection_error(*h*, *s=(None)*)
Compute the squared projection error of a function h onto the basis.

The error is computed in the L2 norm (continuous case) or the 2-norm (discrete case), that is, ||h - P h||^2, where P denotes the projector operator associated to the basis.

Parameters

- **h** (*np.ndarray*) – Function to be projected.
- **s** (*tuple, optional*) – Slice the basis. If the slice is not provided, the whole basis is considered. Default = (None,)

Returns **error** – Square of the projection error.

Return type float

`project`(*h*, *s=(None)*)
Project a function h onto the basis.

This method represents the action of projecting the function h onto the span of the basis.

Parameters

- **h** (*np.ndarray*) – Function or set of functions to be projected.
- **s** (*tuple, optional*) – Slice the basis. If the slice is not provided, the whole basis is considered. Default = (None,)

Returns **projected_function** – Projection of h onto the basis.

Return type np.ndarray

`interpolate`(*h*)
Interpolate a function h at EIM nodes.

This method uses the basis and associated EIM nodes (see the `arby.Basis.eim_` method) for interpolation.

Parameters **h** (*np.ndarray*) – Function or set of functions to be interpolated.

Returns **h_interpolated** – Interpolated function at EIM nodes.

Return type np.ndarray

arby.basis.**gram_schmidt**(*functions*, *integration*, *max_iter=3*) → numpy.ndarray
Orthonormalize a set of functions.

This algorithm implements the Iterated, Modified Gram-Schmidt (GS) algorithm [Hoffmann1989] to build an orthonormal basis from a set of functions.

Parameters

- **functions** (*array_like, shape=(m, L)*) – Functions to be orthonormalized, where *m* is the number of functions and *L* is the length of the sample.
- **integration** (`arby.integrals.Integration`) – Instance of the *Integration* class for defining inner products.
- **max_iter** (*int, optional*) – Maximum number of iterations. Default = 3.

Returns **basis** – Orthonormalized array.

Return type numpy.ndarray

Raises **ValueError** – If functions are not linearly independent for a given tolerance.

References

arby.basis.**reduced_basis**(*training_set*, *physical_points*, *integration_rule='riemann'*, *greedy_tol=1e-12*, *normalize=False*) → *arby.basis.RB*
Build a reduced basis from training data.

This function implements the Reduced Basis (RB) greedy algorithm for building an orthonormalized reduced basis out from training data. The basis is built for reproducing the training functions within a user specified tolerance [TiglioAndVillanueva2021] by linear combinations of its elements. Tuning the `greedy_tol` parameter allows to control the representation accuracy of the basis.

The `integration_rule` parameter specifies the rule for defining inner products. If the training functions (rows of the `training_set`) does not correspond to continuous data (e.g. time), choose `"euclidean"`. Otherwise choose any of the quadratures defined in the `arby.Integration` class.

Set the boolean `normalize` to True if you want to normalize the training set before running the greedy algorithm. This condition not only emphasizes on structure over scale but may leads to noticeable speedups for large datasets.

The output is a container which comprises RB data: a `basis` object storing the reduced basis and handling tools (see `arby.Basis`); the greedy `errors` corresponding to the maxima over the `training set` of the squared projection errors for each greedy swept; the greedy `indices` locating the most relevant training functions used for building the basis; and the `projection_matrix` storing the projection coefficients generated by the greedy algorithm. For example, we can recover the training set (more precisely, a compressed version of it) by multiplying the projection matrix with the reduced basis.

Parameters

- **`training_set`** (*`numpy.ndarray`*) – The training set of functions.
- **`physical_points`** (*`numpy.ndarray`*) – Physical points for quadrature rules.
- **`integration_rule`** (*`str, optional`*) – The quadrature rule to define an integration scheme. Default = "riemann".
- **`greedy_tol`** (*`float, optional`*) – The greedy tolerance as a stopping condition for the reduced basis greedy algorithm. Default = 1e-12.
- **`normalize`** (*`bool, optional`*) – True if you want to normalize the training set. Default = False.

Returns Container for RB data. Contains (`basis`, `errors`, `indices`, `projection_matrix`).

Return type *arby.basis.RB*

Raises **`ValueError`** – If `training_set.shape[1]` doesn't coincide with quadrature rule weights.

Notes

If `normalize` is True, the projection coefficients are with respect to the original basis but the greedy errors are relative to the normalized training set.

References

3.3 Module `arby.integrals`

Integration schemes module.

class `arby.integrals.`**`Integration`**(*interval*, *rule='riemann'*)

Integration scheme.

This class fixes a frame for performing integrals, inner products and derived operations. An integral is defined by a quadrature rule composed by nodes and weights which are used to construct a discrete approximation to the true integral (or inner product).

For completeness, an "euclidean" rule is available for which inner products reduce to simple discrete dot products.

Parameters

- **`interval`** (*`numpy.ndarray`*) – Equispaced set of points as domain for integrals or inner products.

- **rule** (*str, optional*) – Quadrature rule. Default = "riemann". Available = ("riemann", "trapezoidal", "euclidean")

integral(*f*)
Integrate a function.

Parameters **f** (*np.ndarray*) – Real or complex numbers array.

dot(*f*, *g*)
Return the dot product between functions.

Parameters

- **f** (*np.ndarray*) – Real or complex numbers array.
- **g** (*np.ndarray*) – Real or complex numbers array.

norm(*f*)
Return the norm of a function.

Parameters **f** (*np.ndarray*) – Real or complex numbers array.

normalize(*f*)
Normalize a function.

Parameters **f** (*np.ndarray*) – Real or complex numbers array.

CHAPTER

FOUR

QUICK USAGE

Suppose we have a set of real functions parametrized by a real number λ. This set, the *training set*, represents an underlying parametrized model $f_\lambda(x)$ with continuous dependency in λ. Without complete knowledge about f_λ, we'd like to produce an accurate approximation only through the access to the training set.

With Arby we can build an accurate *surrogate model* to represent the training set. For simplicity, suppose a discretization of the parameter domain [`par_min`, `par_max`] with `Ntrain` samples indexing the training set

```
params = np.linspace(par_min, par_max, Ntrain)
```

and a discretization of the x domain $[a, b]$ in `Nsamples` points

```
x_samples = np.linspace(a, b, Nsamples)
```

Next, we build a training set

```
training_set = [f(par, x_samples) for par in params]
```

that has shape (`Ntrain`, `Nsamples`).

Then we build the surrogate model with Arby by doing:

```
from arby import ReducedOrderModel as ROM
f_model = ROM(training_space=training_set,
              physical_interval=x_samples,
              parameter_interval=params)
```

With `f_model` we can get function samples for any parameter `par` in the interval [`par_min`, `par_max`] simply by calling it:

```
f_model_at_par = f_model.surrogate(par)
plt.plot(x_samples, model_at_par)
plt.show()
```

BIBLIOGRAPHY

[Hoffmann1989] Hoffmann, W. Iterative algorithms for Gram-Schmidt orthogonalization. Computing 41, 335-348 (1989). https://doi.org/10.1007/BF02241222

[TiglioAndVillanueva2021] Reduced Order and Surrogate Models for Gravitational Waves. Tiglio, M. and Villanueva A. arXiv:2101.11608 (2021)

PYTHON MODULE INDEX

a

arby.basis, 17
arby.integrals, 19
arby.rom, 15

INDEX

A

arby.basis
    module, 17
arby.integrals
    module, 19
arby.rom
    module, 15

B

Basis (*class in arby.basis*), 17
basis_ (*arby.rom.ReducedOrderModel property*), 16

D

dot() (*arby.integrals.Integration method*), 20

E

EIM (*class in arby.basis*), 17
eim_ (*arby.basis.Basis property*), 17
eim_ (*arby.rom.ReducedOrderModel property*), 16

G

gram_schmidt() (*in module arby.basis*), 18
greedy_errors_ (*arby.rom.ReducedOrderModel property*), 16
greedy_indices_ (*arby.rom.ReducedOrderModel property*), 16

I

integral() (*arby.integrals.Integration method*), 20
Integration (*class in arby.integrals*), 19
interpolate() (*arby.basis.Basis method*), 18

M

module
    arby.basis, 17
    arby.integrals, 19
    arby.rom, 15

N

Nbasis_ (*arby.basis.Basis property*), 17
norm() (*arby.integrals.Integration method*), 20
normalize() (*arby.integrals.Integration method*), 20
Nsamples_ (*arby.rom.ReducedOrderModel property*), 16
Ntrain_ (*arby.rom.ReducedOrderModel property*), 16

P

project() (*arby.basis.Basis method*), 18
projection_error() (*arby.basis.Basis method*), 17
projection_matrix_ (*arby.rom.ReducedOrderModel property*), 16

R

RB (*class in arby.basis*), 17
reduced_basis() (*in module arby.basis*), 18
ReducedOrderModel (*class in arby.rom*), 15

S

surrogate() (*arby.rom.ReducedOrderModel method*), 16