SCRIMP

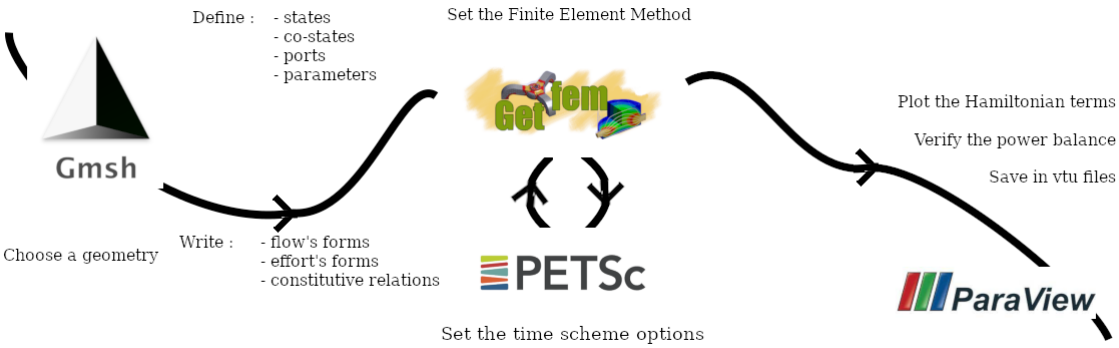
Release 1.1

Giuseppe Ferraro, Michel Fournié, Ghislain Haine

Aug 20, 2024

CONTENTS

Simulation and ContRol of Interactions in Multi-Physics



CHAPTER

ONE

WHAT IS SCRIMP?

SCRIMP (Simulation and ContRol of Interactions in Multi-Physics) is a python collection, *namely* a package, of *methods* and *classes* for the structure-preserving discretization and simulation of multi-physics models, using the formalism of port-Hamiltonian systems (van der Schaft and Maschke (2002)).

SCRIMP aims at speeding the coding process of the **Partitioned Finite Element Method** on a wide range of (multi-)physical systems (Brugnoli *et al.* (2021)), and scrimp and save time!

The documentation is available in pdf.

1.1 Port-Hamiltonian systems

1.1.1 What are they?

Let us sketch a rough portrait of port-Hamiltonian systems as they are considered in **SCRIMP**.

Port-Hamiltonian systems constitute a strongly structured class of control systems with collocated observation. It relies on a functional form $\mathcal{H}$ (the **Hamiltonian**), whose variables α_i are the **states** of the system. The **co-states** $M_i e_i := \delta_{\alpha_i}\mathcal{H}$ are defined as the variational derivative of the Hamiltonian with respect to the states, on the metric induced by the M_i matrices.

The dynamics is provided *via* trajectories belonging in a **Dirac** structure, which can be represented by two matrices (of operators) M symmetric and J skew-symmetric as

$$M \begin{pmatrix} \frac{\mathrm{d}}{\mathrm{d}t}\alpha_1(t) \\ \vdots \\ \frac{\mathrm{d}}{\mathrm{d}t}\alpha_k(t) \\ f_R(t) \\ -y_{exp}(t) \\ u_{imp}(t) \end{pmatrix} = J \begin{pmatrix} e_1(t) \\ \vdots \\ e_k(t) \\ e_R(t) \\ u_{exp}(t) \\ -y_{imp}(t) \end{pmatrix}$$

together with **constitutive relations**

$$M_i e_i(t) = \delta_{\alpha_i(t)}\mathcal{H}(\alpha_1(t), \cdots, \alpha_k(t)) \qquad \mathcal{N}(t, f_R(t), e_R(t)) = 0$$

This structure allows to describe the evolution of the Hamiltonian along the trajectories

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathcal{H}(\alpha_1(t), \cdots, \alpha_k(t)) = -e_R(t)^\top M_R f_R(t) + u_{exp}(t)^\top M_{exp} y_{exp}(t) + u_{imp}(t)^\top M_{imp} y_{imp}(t)$$

The first term of the right-hand side stands for a loss of *energy*, hence the name of *resistive (or dissipative) port* for the couple (f_R, e_R). The other two terms stands for exchanges with the environment through the *control ports*. One is *explicit*, u_{exp}, as a usual forcing term in the equations (its collocated output y_{exp} plays no role in the dynamics). The

other is *implicit*: u_{imp} does not appear directly in the dynamics, and its collocated output y_{imp} plays the role of the Lagrange multiplier imposing the value of u_{imp}.

Each indexed matrix M_ℓ is the appropriate sub-matrix of M.

A very important and useful fact is that the matrices M and J can depend on time and states!

1.1.2 The Partitioned Finite Element Method

The main objective of a **structure-preserving discretization** in the port-Hamiltonian formalism is to obtain a discrete version of the power balance satisfied by the Hamiltonian functional.

A recent scheme, known as the **Partitioned Finite Element Method** (PFEM) (Cardoso-Ribeiro *et al.* (2021)), achieves this goal.

The strategy follows three steps, inspired by the Mixed Finite Element Method for steady-state problem with homogeneous boundary condition

- write the weak form of the system;
- integrate by parts a **partition** of the state (such that *the control appears*); and
- project on finite element spaces.

1.2 Coding philosophy

SCRIMP assumes that the final user is not familiar with numerical simulations. The aim is to facilitate the first step from modelisation to simulation by sticking as much as possible to the port-Hamiltonian framework, getting rid of coding issues.

As such, these simplifications naturally imply a lack of optimization of the code. Nevertheless, the syntax of **SCRIMP** try to let confirmed users to reach finer tuning in order to perform more sophisticated simulations.

A basic usage of **SCRIMP** consists in a script with the following steps:

- Define a *domain*
- Define at least one *state*. And of course, its *co-state*, in order to get a *dynamical port*
- Define a Finite Element Method on this port: give at least an order, at first glance, default values are sufficient
- Define *algebraic ports* (not mandatory) and its FEM
- Define *control ports* (not mandatory) and its FEM
- Define *parameters*
- Write down the forms on the *flow side* of the Dirac structure, *i.e.* the **brick** defining the matrix M
- Write down the forms on the *effort side* of the Dirac structure, *i.e.* th **brick** defining the matrix J
- Write down all the forms defining the *constitutive relations*, always with **bricks**
- Set up time scheme options: again, at first glance, default values are sufficient
- Solve
- Plot
- Export

We try to eliminate as much as possible the *computer-side* of the simulations, by following the port-Hamiltonian vocabulary, always by keeping the possibility of fine tuning available.

CHAPTER

TWO

USER'S GUIDE

2.1 How to install

2.1.1 Anaconda

The easiest way to install SCRIMP is to use a conda environment.

1. Install Anaconda
2. Clone the git repository: `git clone https://github.com/g-haine/scrimp`
3. Enter the folder: `cd scrimp`
4. Create the conda environment: `conda env create --file /path/to/scrimp/scrimp.yml`
5. Activate the environment: `conda activate scrimp`
6. Add scrimp to the PATH: `conda develop /path/to/scrimp/`
7. Finish with pip: `pip install -e .`

2.1.2 Tests

You may test your installation by running avalaible examples in the `examples` folder.

2.1.3 Code structure

SCRIMP is developped as a *package*: the `__init__.py` file of the */path/to/scrimp/* folder is the root file. Each subdirectory is a sub-package of **SCRIMP**. Files are called *module* in this framework and may be called *via* the command **import**. For instance the module *linalg* gathering linear algebra functions of the *subpackage* utils can be imported with `import scrimp.utils.linalg`.

2.1.4 Documentation

You can find this documentation here.

It is automatically built upon the code comments using sphinx.

See Sphinx for further informations.

2.2 Getting started

In order to start using **SCRIMP**, you have to work in the conda environment *scrimp* from the installation by running `conda activate scrimp`.

To understand the coding philosophy of **SCRIMP**, let us consider the 1D wave equation with Neumann boundary control as a first example

$$\begin{cases} \rho(x)\partial_{tt}^2 w(t,x) - \partial_x\left(T(x)\partial_x w(t,x)\right) &= 0, & t\geq 0, x\in(0,1),\\ \partial_t w(0,x) &= v_0(x), & x\in(0,1),\\ \partial_x w(0,x) &= s_0(x), & x\in(0,1),\\ -T(0)\partial_x\left(w(t,0)\right) &= u_L(t), & t\geq 0,\\ T(1)\partial_x\left(w(t,1)\right) &= u_R(t), & t\geq 0, \end{cases}$$

where w denotes the deflection from the equilibrium position of a string, ρ is its mass density and T the Young's modulus. **Note** the minus sign on the control at the left end side, standing for the *outward normal* to the domain $(0,1)$.

The physics giving this equation has to be restated in the port-Hamiltonian formalism first.

2.2.1 Port-Hamiltonian framework

Let $\alpha_q := \partial_x w$ denotes the *strain* and $\alpha_p := \rho\partial_t w$ the *linear momentum*. One can express the total mechanical energy lying in the system $\mathcal{H}$, the **Hamiltonian**, as

$$\mathcal{H}(t) = \mathcal{H}(\alpha_q(t,x),\alpha_p(t,x)) := \underbrace{\frac{1}{2}\int_0^1 \alpha_q(t,x)T(x)\alpha_q(t,x)\mathrm{d}x}_{\text{Potential energy}} + \underbrace{\frac{1}{2}\int_0^1 \frac{\alpha_p(t,x)^2}{\rho(x)}\mathrm{d}x}_{\text{Kinetic energy}}.$$

The variables α_q and α_p are known as the **state variables**, or in the present case since $\mathcal{H}$ represents an energy, the **energy variables**.

Computing the variational derivative of $\mathcal{H}$ with respect to these variables leads to the **co-state variables**, or in our case the **co-energy variables**, *i.e.*

$$e_q := \delta_{\alpha_q}\mathcal{H} = T\alpha_q, \qquad e_p := \delta_{\alpha_p}\mathcal{H} = \frac{\alpha_p}{\rho},$$

that is the *stress* and the *velocity* respectively.

Newton's second law and Schwarz's lemma give the following dynamics

$$\begin{pmatrix}\partial_t\alpha_q\\ \partial_t\alpha_p\end{pmatrix} = \begin{bmatrix}0 & \partial_x\\ \partial_x & 0\end{bmatrix}\begin{pmatrix}e_q\\ e_p\end{pmatrix}.$$

Of course, trivial substitutions in this system would lead again to the initial string equation in second-order form. However, by keeping the system as is, an important structure appears. Indeed, the matrix of operators above is *formally* skew-symmetric. In other words, for all test functions f_q and f_p (compactly supported C^∞ functions), one has thanks to integration by parts

$$\begin{pmatrix}f_q & f_p\end{pmatrix}\begin{bmatrix}0 & \partial_x\\ \partial_x & 0\end{bmatrix}\begin{pmatrix}f_q\\ f_p\end{pmatrix} = 0.$$

Together with the boundary Neumann condition, and defining *collocated* Dirichlet observations, this defines a (Stokes-) **Dirac structure**, where solutions along time, *i.e. trajectories*, will belong.

The port-Hamiltonian system representing a (linear) vibrating string with Neumann boundary control and Dirichlet boundary observation then writes

$$\begin{pmatrix}\partial_t\alpha_q\\ \partial_t\alpha_p\end{pmatrix} = \begin{bmatrix}0 & \partial_x\\ \partial_x & 0\end{bmatrix}\begin{pmatrix}e_q\\ e_p\end{pmatrix},$$

$$\left\{\begin{array}{rcl} -e_q(t,0) &=& u_L(t),\\ e_q(t,1) &=& u_R(t),\\ y_L(t) &=& e_p(t,0),\\ y_R(t) &=& e_p(t,1),\end{array}\right.$$

$$\left\{\begin{array}{rcl} e_q &=& T\alpha_q,\\ e_p &=& \frac{\alpha_p}{\rho}.\end{array}\right.$$

The two first blocks, giving in particular the dynamics, define the **Dirac structure** of the system. The third block is known as the **constitutive relations**, and is needed to ensure uniqueness of solutions.

The importance of the **Dirac structure** relies, in particular, in the fact that it encloses the **power balance** satisfied by the **Hamiltonian**. Indeed, along the trajectories, one has

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathcal{H}(t) = \frac{\mathrm{d}}{\mathrm{d}t}\mathcal{H}(\alpha_q(t),\alpha_p(t)) = \underbrace{y_R(t)u_R(t)}_{\text{power flowing through the right}} + \underbrace{y_L(t)u_L(t)}_{\text{power flowing through the left}}.$$

In other words, the **Dirac structure** encodes the way the system communicates with its environment. In the present example, it says that the variation of the total mechanical energy is given by the power supplied to the system at the boundaries.

Each couple $(\partial_t\alpha_q, e_q)$, $(\partial_t\alpha_p, e_p)$, (u_L, y_L) and (u_R, y_R) is a **port** of the port-Hamiltonian system, and is associated to a physically meaningful term in the **power balance**.

2.2.2 Structure-preserving discretization

The objective of a structure-preserving discretization method is to obtain a **finite-dimensional Dirac structure** that encloses a *discrete version* of the power balance. There is several ways to achieve this goal, but **SCRIMP** focuses on a particular application of the Mixed Finite Element Mehod, called the **Partitioned Finite Element Method**.

Remark: The 1D case does simplify the difficulties coming from the boundary terms. Indeed, here the functional spaces for the controls u_L, u_R and the observations y_L, y_R are nothing but $\mathbb{R}$.

Let φ_q and φ_p be smooth test functions, and δ_{mx} denote the Kronecker symbol. One can write the weak formulation of the **Dirac Structure** as follows

$$\left\{\begin{array}{rcl} \int_0^1 \partial_t\alpha_q(t,x)\varphi_q(x)\mathrm{d}x &=& \int_0^1 \partial_x e_p(t,x)\varphi_q(x)\mathrm{d}x,\\ \int_0^1 \partial_t\alpha_p(t,x)\varphi_p(x)\mathrm{d}x &=& \int_0^1 \partial_x e_q(t,x)\varphi_p(x)\mathrm{d}x,\\ y_L(t) &=& \delta_{0x}e_p(t,x),\\ y_R(t) &=& \delta_{1x}e_p(t,x).\end{array}\right.$$

Integrating by parts the second line make the controls appear

$$\left\{\begin{array}{rcl} \int_0^1 \partial_t\alpha_q(t,x)\varphi_q(x)\mathrm{d}x &=& \int_0^1 \partial_x e_p(t,x)\varphi_q(x)\mathrm{d}x,\\ \int_0^1 \partial_t\alpha_p(t,x)\varphi_p(x)\mathrm{d}x &=& -\int_0^1 e_q(t,x)\partial_x\varphi_p(x)\mathrm{d}x + u_R(t)\varphi_p(1) + u_L(t)\varphi_p(0),\\ y_L(t) &=& \delta_{0x}e_p(t,x),\\ y_R(t) &=& \delta_{1x}e_p(t,x).\end{array}\right.$$

Now, let $(\varphi_q^i)_{1\leq i\leq N_q}$ and $(\varphi_p^k)_{1\leq k\leq N_p}$ be two finite families of approximations for the q-type port and the p-type port respectively, typically finite element families, and write the discrete weak formulation with those families, one has for all $1\leq i\leq N_q$ and all $1\leq k\leq N_p$

$$\left\{\begin{array}{rcl} \sum_{j=1}^{N_q}\int_0^1 \varphi_q^j(x)\varphi_q^i(x)\mathrm{d}x\,\frac{\mathrm{d}}{\mathrm{d}t}\alpha_q^j(t) &=& \sum_{\ell=1}^{N_p}\int_0^1 \partial_x\varphi_p^\ell(x)\varphi_q^i(x)\mathrm{d}x\,e_p^\ell(t),\\ \sum_{\ell=1}^{N_p}\int_0^1 \varphi_p^\ell(x)\varphi_p^k(x)\mathrm{d}x\,\frac{\mathrm{d}}{\mathrm{d}t}\alpha_p^\ell(t) &=& -\sum_{j=1}^{N_q}\int_0^1 \varphi_q^j(x)\partial_x\varphi_p^k(x)\mathrm{d}x\,e_q^j(t)\\ && \quad + u_R(t)\varphi_p^k(1) + u_L(t)\varphi_p^k(0),\\ y_L(t) &=& \sum_{\ell=1}^{N_p}\varphi_p^\ell(0)\,e_p^\ell(t),\\ y_R(t) &=& \sum_{\ell=1}^{N_p}\varphi_p^\ell(1)\,e_p^\ell(t),\end{array}\right.$$

which rewrites in matrix form

$$\underbrace{\begin{bmatrix} M_q & 0 & 0 & 0 \\ 0 & M_p & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{=M} \begin{pmatrix} \frac{\mathrm{d}}{\mathrm{d}t}\underline{\alpha_q}(t) \\ \frac{\mathrm{d}}{\mathrm{d}t}\underline{\alpha_p}(t) \\ -y_L(t) \\ -y_R(t) \end{pmatrix} = \underbrace{\begin{bmatrix} 0 & D & 0 & 0 \\ -D^\top & 0 & B_L & B_R \\ 0 & -B_L^\top & 0 & 0 \\ 0 & -B_R^\top & 0 & 0 \end{bmatrix}}_{=J} \begin{pmatrix} \underline{e_q}(t) \\ \underline{e_p}(t) \\ u_L(t) \\ u_R(t) \end{pmatrix},$$

where $\underline{\alpha_\star}(t) := \begin{pmatrix} \alpha_\star^1(t) & \cdots & \alpha_\star^{N_\star} \end{pmatrix}^\top$, $\underline{e_\star}(t) := \begin{pmatrix} e_\star^1(t) & \cdots & e_\star^{N_\star} \end{pmatrix}^\top$, and

$$(M_q)_{ij} := \int_0^1 \varphi_q^j(x)\varphi_q^i(x)\mathrm{d}x, \qquad (M_p)_{k\ell} := \int_0^1 \varphi_p^\ell(x)\varphi_p^k(x)\mathrm{d}x,$$

$$(D)_{i\ell} := \int_0^1 \partial_x\varphi_p^\ell(x)\varphi_q^i(x)\mathrm{d}x, \qquad (B_L)_k := \varphi_p^k(0), \qquad (B_R)_k := \varphi_p^k(1).$$

Abusing the language, the left-hand side will be called the **flow** of the **Dirac structure** in **SCRIMP**, while the right-hand side will be called the **effort**.

Now one can approximate the **constitutive relations** in those families by projection of their weak formulations

$$\left\{ \begin{array}{rcl} \int_0^1 e_q(t,x)\varphi_q(x)\mathrm{d}x & = & \int_0^1 T(x)\alpha_q(t,x)\varphi_q(x)\mathrm{d}x, \\ \int_0^1 e_p(t,x)\varphi_p(x)\mathrm{d}x & = & \int_0^1 \frac{\alpha_p(t,x)}{\rho(x)}\varphi_p(x)\mathrm{d}x, \end{array} \right.$$

from which one can deduce the matrix form of the discrete weak formulation of the constitutive relation

$$\left\{ \begin{array}{rcl} M_q\underline{e_q}(t) & = & M_T\underline{\alpha_q}(t), \\ M_p\underline{e_p}(t) & = & M_\rho\underline{\alpha_p}(t), \end{array} \right.$$

where

$$(M_T)_{ij} := \int_0^1 T(x)\varphi_q^j(x)\varphi_q^i(x)\mathrm{d}x, \qquad (M_\rho)_{k\ell} := \int_0^1 \frac{\varphi_p^\ell(x)}{\rho(x)}\varphi_p^k(x)\mathrm{d}x.$$

Finally, the **discrete Hamiltonian** $\mathcal{H}^d$ is defined as the evaluation of $\mathcal{H}$ on the approximation of the **state variables**

$$\mathcal{H}^d(t) := \mathcal{H}(\alpha_q^d(t,x), \alpha_p^d(t)) = \frac{1}{2}\underline{\alpha_q}(t)^\top M_T\underline{\alpha_q}(t) + \frac{1}{2}\underline{\alpha_p}(t)^\top M_\rho\underline{\alpha_p}(t).$$

The **discrete power balance** is then easily deduced from the above matrix formulations, thanks to the symmetry of M and the skew-symmetry of J

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathcal{H}^d(t) = y_R(t)u_R(t) + y_L(t)u_L(t).$$

Remark: The discrete system that has to be solved numerically is a Differential Algebraic Equation (DAE). There exists some case (as in this example), where one can write the **co-state** formulation of the system by substituting the **constitutive relations** at the continuous level to get a more classical Ordinary Differential Equation (ODE)

$$\begin{bmatrix} \widetilde{M}_q & 0 & 0 & 0 \\ 0 & \widetilde{M}_p & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} \frac{\mathrm{d}}{\mathrm{d}t}\underline{e_q}(t) \\ \frac{\mathrm{d}}{\mathrm{d}t}\underline{e_p}(t) \\ -y_L(t) \\ -y_R(t) \end{pmatrix} = \begin{bmatrix} 0 & D & 0 & 0 \\ -D^\top & 0 & B_L & B_R \\ 0 & -B_L^\top & 0 & 0 \\ 0 & -B_R^\top & 0 & 0 \end{bmatrix} \begin{pmatrix} \underline{e_q}(t) \\ \underline{e_p}(t) \\ u_L(t) \\ u_R(t) \end{pmatrix},$$

where this time the mass matrices on the left-hand side are both *weighted* with respect to the physical parameters

$$(\widetilde{M}_q)_{ij} := \int_0^1 T^{-1}(x)\varphi_q^j(x)\varphi_q^i(x)\mathrm{d}x, \qquad (\widetilde{M}_p)_{k\ell} := \int_0^1 \rho(x)\varphi_p^\ell(x)\varphi_p^k(x)\mathrm{d}x.$$

2.2.3 Coding within SCRIMP

The following code is available in the file `wave_1D.py` of the *sandbox* folder of scrimp.

To start, import **SCRIMP** and create a *distributed port-Hamiltonian system* (DPHS) called, *e.g.*, `wave`

```
import scrimp as S

wave = S.DPHS("real")
```

Then, define the domain $\Omega = (0, 1)$, with a mesh-size parameter h, and add it to the *DPHS*

```
domain = S.Domain("Interval", {"L": 1., "h": 0.01})
wave.set_domain(domain)
```

This creates a mesh of the interval $\Omega = (0, 1)$.

Important to keep in mind: the domain is composed of `regions`, denoted by integers. The *built-in* geometry of an interval available in the code returns 1 for the domain Ω, 10 for the left-end and 11 for the right-end. Informations about available geometries and the indices of their regions can be found in the documentation or *via* the function `built_in_geometries()` available in `scrimp.utils.mesh`.

On this domain, we define two **states** and add them to the *DPHS*

```
alpha_q = S.State("q", "Strain", "scalar-field")
alpha_p = S.State("p", "Linear momentum", "scalar-field")
wave.add_state(alpha_q)
wave.add_state(alpha_p)
```

and the two associated **co-states**

```
e_q = S.CoState("e_q", "Stress", alpha_q)
e_p = S.CoState("e_p", "Velocity", alpha_p)
wave.add_costate(e_q)
wave.add_costate(e_p)
```

These latter calls create automatically two *non-algebraic* **ports**, named after their respective **state**. Note that we simplify the notations and do not write `alpha_q` and `alpha_p` but `q` and `p` for the sake of readability.

Finally, we create and add the two control-observation **ports** with

```
left_end = S.Control_Port("Boundary control (left)", "U_L", "Normal force", "Y_L",
↪"Velocity", "scalar-field", region=10)
right_end = S.Control_Port("Boundary control (right)", "U_R", "Normal force", "Y_R",
↪"Velocity", "scalar-field", region=11)
wave.add_control_port(left_end)
wave.add_control_port(right_end)
```

Note the *crucial* keyword `region` to restrict each port to its end. By default, it would apply everywhere.

Syntaxic note: although y is the observation in the theory of port-Hamiltonian systems, it is also the second space variable for N-D problems. This name is thus reserved for this latter aim and forbidden in all definitions of a *DPHS*. Nevertheless, the code being case-sensitive, it is possible to name the observation `Y`. To avoid mistakes, we take the habit to always use this syntax, this is why we denoted our controls and observations with capital letters even if the problem does not occur in this 1D example.

To be able to write the discrete weak formulation of the system, one need to set four finite element families, associated to each **port**. Only two arguments are mandatory: the *name* of the port and the *degree* of the approximations.

```
V_q = S.FEM("q", 2)
V_p = S.FEM("p", 1)
V_L = S.FEM("Boundary control (left)", 1)
V_R = S.FEM("Boundary control (right)", 1)
```

This will construct a family of Lagrange finite elements (default choice) for each port, with the prescribed order. Remember that the boundary is only 2 disconnected points in this 1D case, so the only possibility for the finite element is 1 degree of freedom on each of them: Lagrange elements of order 1 is the easy way to do that.

Of course, this *FEM* must be added to the *DPHS*

```
wave.add_FEM(V_q)
wave.add_FEM(V_p)
wave.add_FEM(V_L)
wave.add_FEM(V_R)
```

Finally, the physical parameters of the experiment have to be defined. In **SCRIMP**, a *parameter* is associated to a *port*.

```
T = S.Parameter("T", "Young's modulus", "scalar-field", "1", "q")
rho = S.Parameter("rho", "Mass density", "scalar-field", "1 + x*(1-x)", "p")
wave.add_parameter(T)
wave.add_parameter(rho)
```

The first argument will be **the string that can be used in forms**, the second argument is a human-readable description, the third one set the kind of the parameter, the fourth one is the mathematical expression defining the parameter, and finally the fifth argument is the *name* of the associated port.

It is now possible to write the weak forms defining the system. *Only the non-zero blocks* are mandatory. Furthermore, the place of the block is automatically determined by GetFEM. The syntax follow a simple rule: the unknown trial function `q` is automatically associated to the test function `Test_q` (note the capital T on `Test`), and so on.

Like we did for each call, the first step is to create the object, then to add it to the *DPHS*. As there is a lot of *bricks*, let us make a loop using a python *list*

```
bricks = [
    # M matrix, on the flow side
    S.Brick("M_q", "q * Test_q", [1], dt=True, position="flow"),
    S.Brick("M_p", "p * Test_p", [1], dt=True, position="flow"),
    S.Brick("M_Y_L", "Y_L * Test_Y_L", [10], position="flow"),
    S.Brick("M_Y_R", "Y_R * Test_Y_R", [11], position="flow"),

    # J matrix, on the effort side
    S.Brick("D", "Grad(e_p) * Test_q", [1], position="effort"),

    S.Brick("-D^T", "-e_q * Grad(Test_p)", [1], position="effort"),
    S.Brick("B_L", "-U_L * Test_p", [10], position="effort"),
    S.Brick("B_R", "U_R * Test_p", [11], position="effort"),

    S.Brick("-B_L^T", "e_p * Test_Y_L", [10], position="effort"),
    S.Brick("-B_R^T", "-e_p * Test_Y_R", [11], position="effort"),

    # Constitutive relations
    S.Brick("-M_e_q", "-e_q * Test_e_q", [1]),
    S.Brick("CR_q", "q*T * Test_e_q", [1]),
```

(continues on next page)

(continued from previous page)

```
    S.Brick("-M_e_p", "-e_p * Test_e_p", [1]),
    S.Brick("CR_p", "p/rho * Test_e_p", [1]),
    ]

for brick in bricks:
    wave.add_brick(brick)
```

The first argument of a *brick* is a human-readable name, the second one is the form, the third is a list (hence the [and]) of integers, listing all the regions where the form applies. The optional parameter `dt=True` is to inform **SCRIMP** that this block matrix will apply on the time-derivative of the unknown trial function, and finally the option parameter `position="flow"` informs **SCRIMP** that this block is a part of the *flow side* of the Dirac structure, `position="effort"` do the same for the *effort side*, and without this keyword, **SCRIMP** places the *brick* as part of the *constitutive relations*.

Syntaxic note: the constitutive relations have to be written under an implicit formulation $F = 0$. Keep in mind that a minus sign will often appear because of that.

The port-Hamiltonian system is now fully stated. It remains to set the controls and the initial values of the states before solving

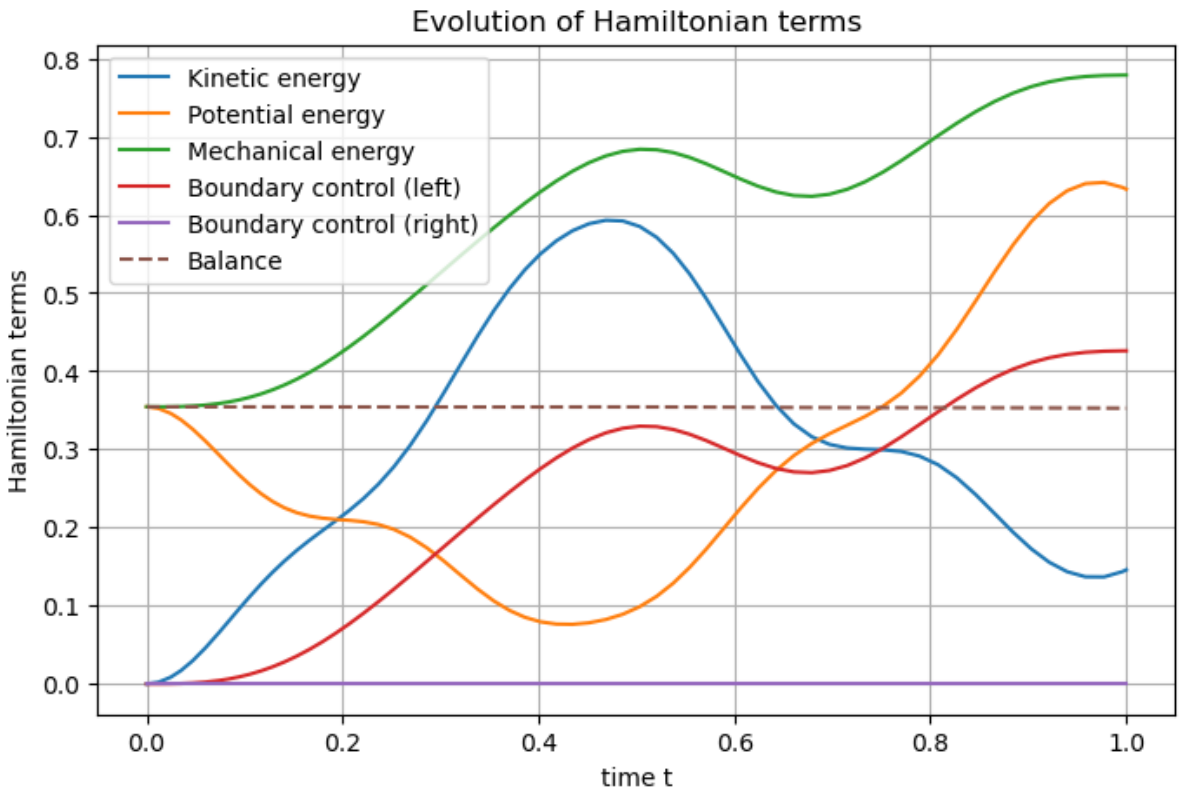
```
expression_left = "-sin(2*pi*t)"
expression_right = "0."
wave.set_control("Boundary control (left)", expression_left)
wave.set_control("Boundary control (right)", expression_right)

q_init = "2.*np.exp(-50.*(x-0.5)*(x-0.5))"
p_init = "0."
wave.set_initial_value("q", q_init)
wave.set_initial_value("p", p_init)
```

We can now solve the system (with default experiment parameters)

```
wave.solve()
```

To end, one can also add the Hamiltonian terms and plot the contribution of each port to the balance equation

```
wave.hamiltonian.set_name("Mechanical energy")
terms = [
    S.Term("Kinetic energy", "0.5*p*p/rho", [1]),
    S.Term("Potential energy", "0.5*q*T*q", [1]),
]

for term in terms:
    wave.hamiltonian.add_term(term)

wave.plot_Hamiltonian()
```



One can appreciate the *structure-preserving* property by looking at the dashed line, showing the evolution of

$$\mathcal{H}^d(t) - \int_0^t u_R(s)y_R(s)\mathrm{d}s - \int_0^t u_L(s)y_L(s)\mathrm{d}s.$$

And now? It is time to see more examples.

2.3 Examples

We provide some examples coming from our publications.

2.3.1 The wave equation

- file: examples/wave.py
- authors: Giuseppe Ferraro, Ghislain Haine
- date: 22 nov. 2022
- brief: 2D wave equations

`examples.wave.`**`wave_eq`**`()`

A structure-preserving discretization of the wave equation with mixed boundary control

Formulation DAE (energy/co-energy), Grad-Grad, Mixed boundary condition on the Rectangle Undamped case.

Setting

Let us consider the 2D wave equation with *mixed* boundary controls on a bounded rectangle $\Omega := (0, L) \times (0, \ell)$, with boundaries $\Gamma_N := ((0, L) \times \{0, \ell\}) \cup (\{L\} \times (0, \ell))$ and $\Gamma_D := \{0\} \times (0, \ell)$.

The deflection of the membrane from the equilibrium w satisfies classicaly

$$\begin{cases} \rho(x)\partial_{tt}^2 w(t,x) - \operatorname{div}\left(T(x) \cdot \operatorname{grad}\left(w(t,x)\right)\right) &= 0, & t \geq 0, x \in \Omega, \\ \partial_t w(0,x) &= v_0(x), & x \in \Omega, \\ \partial_x w(0,x) &= s_0(x), & x \in \Omega, \\ T(s) \cdot \operatorname{grad}\left(w(t,s)\right) &= u_N(t,s), & t \geq 0, s \in \Gamma_N, \\ \partial_t w(t,s) &= u_D(t,s), & t \geq 0, s \in \Gamma_D, \end{cases}$$

where ρ is the mass density and T the Young's modulus. The subscript N stands for **Neumann**, while the subscript D stands for **Dirichlet** (to be fair, this is not really a Dirichlet boundary condition, as it imposes $\partial_t w$ and not w at the boundary Γ_D).

Let us state the physics in the port-Hamiltonian formalism.

Port-Hamiltonian framework

Let $\alpha_q := \operatorname{grad} w$ denotes the *strain* and $\alpha_p := \rho \partial_t w$ the *linear momentum*. One can express the total mechanical energy lying in the system $\mathcal{H}$, the **Hamiltonian**, as

$$\mathcal{H}(t) = \mathcal{H}(\alpha_q(t,x), \alpha_p(t,x)) := \underbrace{\frac{1}{2}\int_\Omega \alpha_q(t,x) \cdot T(x) \cdot \alpha_q(t,x) \mathrm{d}x}_{\text{Potential energy}} + \underbrace{\frac{1}{2}\int_\Omega \frac{\alpha_p(t,x)^2}{\rho(x)} \mathrm{d}x}_{\text{Kinetic energy}}.$$

The **co-energy variables** are, as in the 1D case

$$e_q := \delta_{\alpha_q} \mathcal{H} = T \cdot \alpha_q, \qquad e_p := \delta_{\alpha_p} \mathcal{H} = \frac{\alpha_p}{\rho},$$

that is the *stress* and the *velocity* respectively.

Newton's second law and Schwarz's lemma give the following dynamics

$$\begin{pmatrix} \partial_t \alpha_q \\ \partial_t \alpha_p \end{pmatrix} = \begin{bmatrix} 0 & \operatorname{grad} \\ \operatorname{div} & 0 \end{bmatrix} \begin{pmatrix} e_q \\ e_p \end{pmatrix}.$$

Of course, this system allows to recover the initial wave equation in second-order form.

The port-Hamiltonian system representing a (linear) vibrating membrane with mixed boundary controls then writes

$$\begin{pmatrix} \partial_t \alpha_q \\ \partial_t \alpha_p \end{pmatrix} = \begin{bmatrix} 0 & \operatorname{grad} \\ \operatorname{div} & 0 \end{bmatrix} \begin{pmatrix} e_q \\ e_p \end{pmatrix},$$

$$\begin{cases} e_q(t,s) &= u_N(t,s), & t \geq 0, s \in \Gamma_N, \\ e_p(t,s) &= u_D(t,s), & t \geq 0, s \in \Gamma_D, \\ y_N(t,s) &= e_p(t,s), & t \geq 0, s \in \Gamma_N, \\ y_D(t,s) &= e_q(t,s), & t \geq 0, s \in \Gamma_D, \end{cases}$$

$$\begin{cases} e_q &= T \cdot \alpha_q, \\ e_p &= \frac{\alpha_p}{\rho}. \end{cases}$$

The **power balance** satisfied by the **Hamiltonian** is

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathcal{H}(t) = \underbrace{\langle y_N(t,\cdot), u_N(t,\cdot)\rangle_{\Gamma_N}}_{\text{power flowing through } \Gamma_N} + \underbrace{\langle u_D(t,\cdot), y_D(t,\cdot)\rangle_{\Gamma_D}}_{\text{power flowing through } \Gamma_D},$$

where $\langle \cdot, \cdot \rangle_\Gamma$ is a boundary duality bracket $H^{\frac{1}{2}}, H^{-\frac{1}{2}}$ at the boundary Γ.

Structure-preserving discretization

Let φ_q and φ_p be smooth test functions on Ω, and ψ_N and ψ_D be smooth test functions on Γ_N and Γ_D respectively. One can write the weak formulation of the **Dirac Structure** as follows

$$\left\{\begin{array}{rcl} \int_\Omega \partial_t \alpha_q(t,x)\varphi_q(x)\mathrm{d}x &=& \int_\Omega \mathrm{grad}\left(e_p(t,x)\right)\cdot\varphi_q(x)\mathrm{d}x, \\ \int_\Omega \partial_t \alpha_p(t,x)\varphi_p(x)\mathrm{d}x &=& \int_\Omega \mathrm{div}\left(e_q(t,x)\right)\varphi_p(x)\mathrm{d}x, \\ \langle y_N,\psi_N\rangle_{\Gamma_N} &=& \langle e_p,\psi_N\rangle_{\Gamma_N}, \\ \langle u_D,\psi_D\rangle_{\Gamma_D} &=& \langle e_p,\psi_D\rangle_{\Gamma_D}. \end{array}\right. \tag{2.1}$$

Integrating by parts the second line make the control u_N and the observation y_D appear

$$\int_\Omega \partial_t \alpha_p(t,x)\varphi_p(x)\mathrm{d}x = -\int_\Omega e_q(t,x)\cdot\mathrm{grad}\left(\varphi_p(x)\right)\mathrm{d}x + \langle\varphi_p,u_N\rangle_{\Gamma_N} + \langle\varphi_p,y_D\rangle_{\Gamma_D}.$$

Now, let $(\varphi_q^i)_{1\le i\le N_q}\subset L^2(\Omega)$ and $(\varphi_p^k)_{1\le k\le N_p}\subset H^1(\Omega)$ be two finite families of approximations for the q-type port and the p-type port respectively, typically discontinuous and continuous Galerkin finite elements respectively. Denote also $(\psi_N^m)_{1\le m_N\le N_N}\subset H^{\frac12}(\Gamma_N)$ and $(\psi_D^m)_{1\le m_D\le N_D}\subset H^{\frac12}(\Gamma_D)$. In particular, the latter choices imply that the duality brackets at the boundary reduce to simple L^2 scalar products.

Writing the discrete weak formulation with those families, one has for all $1\le i\le N_q$, all $1\le k\le N_p$, all $1\le m_N\le N_N$ and all $1\le m_D\le N_D$

$$\left\{\begin{array}{rcl} \sum_{j=1}^{N_q}\int_\Omega \varphi_q^j(x)\varphi_q^i(x)\mathrm{d}x\,\frac{\mathrm{d}}{\mathrm{d}t}\alpha_q^j(t) &=& \sum_{\ell=1}^{N_p}\int_\Omega \mathrm{grad}\left(\varphi_p^\ell(x)\right)\cdot\varphi_q^i(x)\mathrm{d}x\,e_p^\ell(t), \\ \sum_{\ell=1}^{N_p}\int_\Omega \varphi_p^\ell(x)\varphi_p^k(x)\mathrm{d}x\,\frac{\mathrm{d}}{\mathrm{d}t}\alpha_p^\ell(t) &=& -\sum_{j=1}^{N_q}\int_\Omega \varphi_q^j(x)\cdot\mathrm{grad}\left(\varphi_p^k(x)\right)\mathrm{d}x\,e_q^j(t) \\ && +\sum_{n_N=1}^{N_N}\int_{\Gamma_N}\varphi_p^k(s)\psi_N^{n_N}(s)\mathrm{d}s\,u_N^{n_N}(t) \\ && +\sum_{n_D=1}^{N_D}\int_{\Gamma_D}\varphi_p^k(s)\psi_D^{n_D}(s)\mathrm{d}s\,y_D^{n_D}(t), \\ \sum_{n_N=1}^{N_N}\langle\psi_N^{n_N},\psi_N^{m_N}\rangle_{\Gamma_N}\,y_N^{n_N}(t) &=& \sum_{\ell=1}^{N_p}\int_{\Gamma_N}\varphi_p^\ell(s)\psi_N^{m_N}(s)\mathrm{d}s\,e_p^\ell(t), \\ \sum_{n_D=1}^{N_D}\langle\psi_D^{n_D},\psi_D^{m_D}\rangle_{\Gamma_D}\,u_D^{n_D}(t) &=& \sum_{\ell=1}^{N_p}\int_{\Gamma_D}\varphi_p^\ell(s)\psi_D^{m_D}(s)\mathrm{d}s\,e_p^\ell(t), \end{array}\right. \tag{2.2}$$

which rewrites in matrix form

$$\underbrace{\begin{bmatrix} M_q & 0 & 0 & 0 \\ 0 & M_p & 0 & 0 \\ 0 & 0 & M_N & 0 \\ 0 & 0 & 0 & M_D \end{bmatrix}}_{=M}\begin{pmatrix} \frac{\mathrm{d}}{\mathrm{d}t}\underline{\alpha_q}(t) \\ \frac{\mathrm{d}}{\mathrm{d}t}\underline{\alpha_p}(t) \\ -\underline{y_N}(t) \\ \underline{u_D}(t) \end{pmatrix} = \underbrace{\begin{bmatrix} 0 & D & 0 & 0 \\ -D^\top & 0 & B_N & -B_D^T \\ 0 & -B_N^\top & 0 & 0 \\ 0 & B_D & 0 & 0 \end{bmatrix}}_{=J}\begin{pmatrix} \underline{e_q}(t) \\ \underline{e_p}(t) \\ \underline{u_N}(t) \\ -\underline{y_D}(t) \end{pmatrix},$$

where $\underline{\star}(t) := \begin{pmatrix}\star^1(t) & \cdots & \star^{N_\star}\end{pmatrix}^\top$ and

$$(M_q)_{ij} := \int_\Omega \varphi_q^j(x)\cdot\varphi_q^i(x)\mathrm{d}x, \qquad (M_p)_{k\ell} := \int_\Omega \varphi_p^\ell(x)\varphi_p^k(x)\mathrm{d}x, \tag{2.3}$$

$$(M_N)_{m_N n_N} := \int_{\Gamma_N}\psi_N^{n_N}(s)\psi_N^{m_N}(s)\mathrm{d}s, \qquad (M_D)_{m_D n_D} := \int_{\Gamma_D}\psi_D^{n_D}(s)\psi_D^{m_D}(s)\mathrm{d}s, \tag{2.4}$$

$$(D)_{i\ell} := \int_\Omega \mathrm{grad}\left(\varphi_p^\ell(x)\right)\cdot\varphi_q^i(x)\mathrm{d}x,$$

$$(B_N)_{n_N k} := \int_{\Gamma_N}\varphi_p^k(s)\psi_N^{n_N}(s)\mathrm{d}s, \qquad (B_D)_{m_D\ell} := \int_{\Gamma_D}\varphi_p^\ell(s)\psi_D^{m_D}(s)\mathrm{d}s,$$

Now one can approximate the **constitutive relations** in those families by projection of their weak formulations

$$\left\{\begin{array}{rcl} \int_\Omega e_q(t,x)\cdot\varphi_q(x)\mathrm{d}x &=& \int_\Omega \alpha_q(t,x)\cdot T(x)\cdot\varphi_q(x)\mathrm{d}x, \\ \int_\Omega e_p(t,x)\varphi_p(x)\mathrm{d}x &=& \int_\Omega \frac{\alpha_p(t,x)}{\rho(x)}\varphi_p(x)\mathrm{d}x, \end{array}\right.$$

from which one can deduce the matrix form of the discrete weak formulation of the constitutive relation

$$\left\{ \begin{array}{rcl} M_q \underline{e_q}(t) & = & M_T \underline{\alpha_q}(t), \\ M_p \underline{e_p}(t) & = & M_\rho \underline{\alpha_p}(t), \end{array} \right.$$

where

$$(M_T)_{ij} := \int_\Omega \varphi_q^j(x) \cdot T(x) \cdot \varphi_q^i(x) \mathrm{d}x, \qquad (M_\rho)_{k\ell} := \int_\Omega \frac{\varphi_p^\ell(x)}{\rho(x)} \varphi_p^k(x) \mathrm{d}x. \tag{2.5}$$

Finally, the **discrete Hamiltonian** $\mathcal{H}^d$ is defined as the evaluation of $\mathcal{H}$ on the approximation of the **state variables**

$$\mathcal{H}^d(t) := \mathcal{H}(\alpha_q^d(t,x), \alpha_p^d(t)) = \frac{1}{2} \underline{\alpha_q}(t)^\top M_T \underline{\alpha_q}(t) + \frac{1}{2} \underline{\alpha_p}(t)^\top M_\rho \underline{\alpha_p}(t).$$

The **discrete power balance** is then easily deduced from the above matrix formulations, thanks to the symmetry of M and the skew-symmetry of J

$$\frac{\mathrm{d}}{\mathrm{d}t} \mathcal{H}^d(t) = \underline{y_N}(t)^\top M_N \underline{u_N}(t) + \underline{u_D}(t)^\top M_D \underline{y_D}(t).$$

Simulation

Let us start by importing the scrimp package

```
# Import scrimp
import scrimp as S
```

Now define a real Distributed Port-Hamiltonian System

```
# Init the distributed port-Hamiltonian system
wave = S.DPHS("real")
```

The domain is 2-dimensional, and is a rectangle of length 2 and width 1. We use the built-in geometry `Rectangle` and choose a mesh size parameter of 0.1 with the following command.

```
# Set the domain (using the built-in geometry `Rectangle`)
# Labels: Omega = 1, Gamma_Bottom = 10, Gamma_Right = 11, Gamma_Top = 12, Gamma_Left = 13
rectangle = S.Domain("Rectangle", {"L": 2.0, "l": 1.0, "h": 0.1})

# And add it to the dphs
wave.set_domain(rectangle)
```

Defining the states and co-states, care must be taken: the Strain is a **vector-field**.

```
# Define the variables and their discretizations
states = [
    S.State("q", "Strain", "vector-field"),
    S.State("p", "Linear momentum", "scalar-field"),
]
costates = [
    S.CoState("e_q", "Stress", states[0]),
    S.CoState("e_p", "Velocity", states[1]),
]

# Add them to the dphs
```

(continues on next page)

(continued from previous page)

```
for state in states:
    wave.add_state(state)
for costate in costates:
    wave.add_costate(costate)
```

As the domain is the built-in geometry `Rectangle`, the boundary is composed of four parts, with indices 10, 11, 12 and 13, respectively for the lower, right, upper and left edge. Each of them will have its own control port, allowing *e.g.* **mixed** boundary conditions.

Indeed in the above example, we choose Neumann boundary condition on Γ_N, *i.e.* on 10, 11 and 12, while we choose Dirichlet boundary condition on Γ_D, *i.e.* on 13.

The choice to integrate by part the second line of (2.1) has a consequence for the port at boundary 13, as it is then in the *flow* part of the Dirac structure, as can be seen in (2.2). We indicate this using the keyword `position="flow"`.

```
# Define the control ports
control_ports = [
    S.Control_Port(
        "Boundary control (bottom)",
        "U_B",
        "Normal force",
        "Y_B",
        "Velocity trace",
        "scalar-field",
        region=10,
    ),
    S.Control_Port(
        "Boundary control (right)",
        "U_R",
        "Normal force",
        "Y_R",
        "Velocity trace",
        "scalar-field",
        region=11,
    ),
    S.Control_Port(
        "Boundary control (top)",
        "U_T",
        "Normal force",
        "Y_T",
        "Velocity trace",
        "scalar-field",
        region=12,
    ),
    S.Control_Port(
        "Boundary control (left)",
        "U_L",
        "Velocity trace",
        "Y_L",
        "Normal force",
        "scalar-field",
        region=13,
        position="flow",
```

(continues on next page)

(continued from previous page)

```
    ),
]

# Add them to the dphs
for ctrl_port in control_ports:
    wave.add_control_port(ctrl_port)
```

The choice for the finite element families is often the first difficulty of a simulation. Indeed, it can result in a failing time scheme, or a very instable solution. A key-point to take a first decision is to remember which field needs regularity (in the L^2-sense) in the Dirac structure. In our case, the p-type variables should be at least $H^1(\Omega)$, as can be inferred from (2.2). Hence, a first choice for the p-type variables is to take continuous Galerkin finite elements of order k. Since the time derivative of q will be, more or less, a gradient of a p-type variable, it will be a discontinuous Galerkin of order $k-1$ approximation. Finally, at least one trace of these variables, either the control, or the observation, is at most a discontinuous Galerkin of order $k-1$ approximation. Hence the following choices, with $k=2$.

```
# Define the Finite Elements Method of each port
FEMs = [
    S.FEM(states[0].get_name(), 1, "DG"),
    S.FEM(states[1].get_name(), 2, "CG"),
    S.FEM(control_ports[0].get_name(), 1, "DG"),
    S.FEM(control_ports[1].get_name(), 1, "DG"),
    S.FEM(control_ports[2].get_name(), 1, "DG"),
    S.FEM(control_ports[3].get_name(), 1, "DG"),
]

# Add them to the dphs
for FEM in FEMs:
    wave.add_FEM(FEM)
```

We can assume anisotropy and heterogeneity in our model by defining the parameters as follows. It has to be kept in mind that a parameter is always linked to a port (*i.e.*, to a pair *flow-effort*). In particular, a parameter linked to a port that is a vector-field, should be a **tensor-field**.

```
# Define physical parameters
parameters = [
    S.Parameter("T", "Young's modulus", "tensor-field", "[[5+x,x*y],[x*y,2+y]]", "q"),
    S.Parameter("rho", "Mass density", "scalar-field", "3-x", "p"),
]

# Add them to the dphs
for parameter in parameters:
    wave.add_parameter(parameter)
```

It is time to define the bricks of our model, *i.e.* the block matrices of our discretization, providing the weak forms given in (2.3), (2.4), and (2.5).

This is probably the most difficult part of the process, and care must be taken. Remember that the syntax is the Generic Weak-Form Language (GWFL), for which an on-line documentation exists on the GetFEM site.

For the block matrices appearing against time derivative of a variable, it is crucial not to forget the keyword `dt=True`.

```
# Define the pHs via `Brick` == non-zero block matrices == variational terms
bricks = [
    ## Define the Dirac structure
```

(continues on next page)

(continued from previous page)

```
    # Define the mass matrices from the left-hand side: the `flow` part of the Dirac␣
↪structure
    S.Brick("M_q", "q.Test_q", [1], dt=True, position="flow"),
    S.Brick("M_p", "p*Test_p", [1], dt=True, position="flow"),
    S.Brick("M_Y_B", "Y_B*Test_Y_B", [10], position="flow"),
    S.Brick("M_Y_R", "Y_R*Test_Y_R", [11], position="flow"),
    S.Brick("M_Y_T", "Y_T*Test_Y_T", [12], position="flow"),
    # The Dirichlet term is applied via Lagrange multiplier == the colocated output
    S.Brick("M_Y_L", "U_L*Test_Y_L", [13], position="flow"),
    # Define the matrices from the right-hand side: the `effort` part of the Dirac␣
↪structure
    S.Brick("D", "Grad(e_p).Test_q", [1], position="effort"),
    S.Brick("-D^T", "-e_q.Grad(Test_p)", [1], position="effort"),
    S.Brick("B_B", "U_B*Test_p", [10], position="effort"),
    S.Brick("B_R", "U_R*Test_p", [11], position="effort"),
    S.Brick("B_T", "U_T*Test_p", [12], position="effort"),
    # The Dirichlet term is applied via Lagrange multiplier == the colocated output
    S.Brick("B_L", "Y_L*Test_p", [13], position="effort"),
    S.Brick("C_B", "-e_p*Test_Y_B", [10], position="effort"),
    S.Brick("C_R", "-e_p*Test_Y_R", [11], position="effort"),
    S.Brick("C_T", "-e_p*Test_Y_T", [12], position="effort"),
    S.Brick("C_L", "-e_p*Test_Y_L", [13], position="effort"),
    ## Define the constitutive relations
    # Hooke's law under implicit form `- M_e_q e_q + CR_q q = 0`
    S.Brick("-M_e_q", "-e_q.Test_e_q", [1]),
    S.Brick("CR_q", "q.T.Test_e_q", [1]),
    # Linear momentum definition under implicit form `- M_e_p e_p + CR_p p = 0`
    S.Brick("-M_e_p", "-e_p*Test_e_p", [1]),
    S.Brick("CR_p", "p/rho*Test_e_p", [1]),
]

# Add all these `Bricks` to the dphs
for brick in bricks:
    wave.add_brick(brick)
```

The last step is to initialize the dphs, by providing the control functions and the initial values for q and p (*i.e.*, the variables that are derivated in time in the model).

```
## Initialize the problem
# The controls expression, ordered as the control_ports
t_f = 5.0
expressions = ["0.", "0.", "0.", f"0.1*sin(4.*t)*sin(4*pi*y)*exp(-10.*pow((0.5*{t_f}-t),
↪2))"]

# Add each expression to its control_port
for control_port, expression in zip(control_ports, expressions):
    # Set the control functions: it automatically constructs the related `Brick`s such␣
↪that `- M_u u + f(t) = 0`
    wave.set_control(control_port.get_name(), expression)

# Set the initial data
q_0 = "[0., 0.]"
```

(continues on next page)

(continued from previous page)

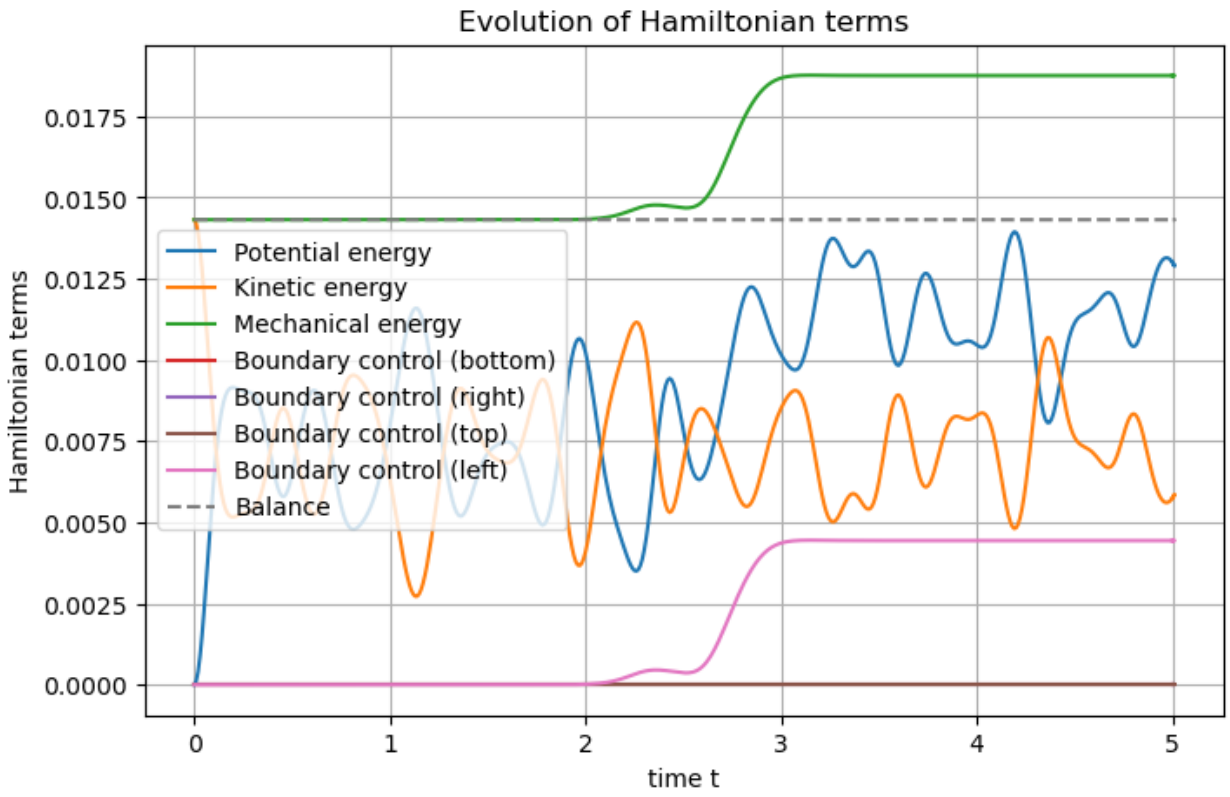
```
wave.set_initial_value("q", q_0)
p_0 = "3**(-20*((x-0.5)*(x-0.5)+(y-0.5)*(y-0.5)))"
wave.set_initial_value("p", p_0)
```

It remains to solve!

```
## Solve in time
# Define the time scheme ("cn" is Crank-Nicolson)
wave.set_time_scheme(ts_type="cn",
                     t_f=t_f,
                     dt_save=0.01,
                     )

# Solve
wave.solve()
```

Now we can set the Hamiltonian and plot it.

```
## Post-processing
# Set Hamiltonian's name
wave.hamiltonian.set_name("Mechanical energy")
# Define each Hamiltonian Term
terms = [
    S.Term("Potential energy", "0.5*q.T.q", [1]),
    S.Term("Kinetic energy", "0.5*p*p/rho", [1]),
]
# Add them to the Hamiltonian
for term in terms:
    wave.hamiltonian.add_term(term)

# Plot the Hamiltonian and save the output
wave.plot_Hamiltonian(save_figure=True, filename="Hamiltonian_Wave_2D_Conservative.png")
```



Adding Damping to the dphs

- file: examples/wave_dissipative.py
- authors: Ghislain Haine
- date: 06 aug. 2024
- brief: 2D dissipative wave equations

`examples.wave_dissipative.`**`wave_eq`**`()`

A structure-preserving discretization of the wave equation with mixed boundary control

Formulation DAE (energy/co-energy), Grad-Grad, Mixed boundary condition on the Rectangle Damped case.

The remining part of the notebook is focused on the way to deal with *dissipativity*, hence using an **algebraic port**.

Let us come back to the continuous system. Adding a (fluid) damping consists in an additive term in Newton second law, which is proportional to the velocity (in the linear case). More precisely, denoting $\nu \geq 0$ the viscous parameter, one has:

$$\rho(x)\partial_{tt}^2 w(t,x) - \operatorname{div}\left(T(x)\cdot\operatorname{grad}\left(w(t,x)\right)\right) + \nu(x)\partial_t w(t,x) = 0.$$

Using the framework of port-Hamiltonian system, this rewrites:

$$\begin{pmatrix}\partial_t \alpha_q\\ \partial_t \alpha_p\end{pmatrix} = \begin{bmatrix}0 & \operatorname{grad}\\ \operatorname{div} & 0\end{bmatrix}\begin{pmatrix}e_q\\ e_p\end{pmatrix} + \begin{pmatrix}0\\ -\nu e_p\end{pmatrix}.$$

One could include $-\nu$ inside the matrix of operators, this is the so-called $J-R$ framework. However, it does not exhibit the underlying Dirac structure, as it hides the resistive port. Let us introduce this hidden port, by denoting f_r

the flow and e_r the effort, as follows:

$$\begin{pmatrix} \partial_t \alpha_q \\ \partial_t \alpha_p \\ f_r \end{pmatrix} = \begin{bmatrix} 0 & \text{grad} & 0 \\ \text{div} & 0 & -I \\ 0 & I^\top & 0 \end{bmatrix} \begin{pmatrix} e_q \\ e_p \\ e_r \end{pmatrix}, \tag{2.6}$$

and supplemented by the resistive constitutive relation $e_r = \nu f_r$.

Of course, at the discrete level, this will increase the number of degrees of freedom, as two extra variables have to be discretized. Nevertheless, in more complicated situations (*e.g.* dealing with non-linearities), this is the price to pay to recover a correct discrete power balance.

The **power balance** satisfied by the **Hamiltonian** is then

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathcal{H}(t) = \underbrace{-\int_\Omega \nu(x) f_r^2(t,x)}_{\text{dissipated power}} \underbrace{+ \langle y_N(t,\cdot), u_N(t,\cdot)\rangle_{\Gamma_N}}_{\text{power flowing through } \Gamma_N} \underbrace{+ \langle u_D(t,\cdot), y_D(t,\cdot)\rangle_{\Gamma_D}}_{\text{power flowing through } \Gamma_D},$$

Another simulation

Let us start a new simulation with damping.

```
# Define a new dphs
wave_diss = S.DPHS("real")

# Set the domain (using the built-in geometry `Rectangle`)
# Labels: Omega = 1, Gamma_Bottom = 10, Gamma_Right = 11, Gamma_Top = 12, Gamma_Left = 13
rectangle = S.Domain("Rectangle", {"L": 2.0, "l": 1.0, "h": 0.1})

# On the rectangle domain
wave_diss.set_domain(rectangle)

# Define the variables
states = [
    S.State("q", "Strain", "vector-field"),
    S.State("p", "Linear momentum", "scalar-field"),
]
costates = [
    S.CoState("e_q", "Stress", states[0]),
    S.CoState("e_p", "Velocity", states[1]),
]

# Add them to the dphs
for (state,costate) in zip(states,costates):
    wave_diss.add_state(state)
    wave_diss.add_costate(costate)

# Define the control ports
control_ports = [
    S.Control_Port(
        "Boundary control (bottom)",
        "U_B",
        "Normal force",
        "Y_B",
```

(continues on next page)

(continued from previous page)

```
        "Velocity trace",
        "scalar-field",
        region=10,
    ),
    S.Control_Port(
        "Boundary control (right)",
        "U_R",
        "Normal force",
        "Y_R",
        "Velocity trace",
        "scalar-field",
        region=11,
    ),
    S.Control_Port(
        "Boundary control (top)",
        "U_T",
        "Normal force",
        "Y_T",
        "Velocity trace",
        "scalar-field",
        region=12,
    ),
    S.Control_Port(
        "Boundary control (left)",
        "U_L",
        "Velocity trace",
        "Y_L",
        "Normal force",
        "scalar-field",
        region=13,
        position="flow",
    ),
]

# Add them to the dphs
for ctrl_port in control_ports:
    wave_diss.add_control_port(ctrl_port)
```

The additional port is defined, added to the system `wave_diss` and a `FEM` is attached to it. Remark that we use the previously defined objects, *i.e.* we only append the `FEM` of the resistive port to the list of previously defined `FEM` objects. We choose continuous Galerkin of order 2, as the resistive effort is of p-type.

```
# Define a dissipative port
port_diss = S.Port("Damping", "f_r", "e_r", "scalar-field")

# Add it to the dphs
wave_diss.add_port(port_diss)

# Define the Finite Elements Method of each port
FEMs = [
    S.FEM(states[0].get_name(), 1, "DG"),
    S.FEM(states[1].get_name(), 2, "CG"),
```

(continues on next page)

(continued from previous page)

```
    S.FEM(control_ports[0].get_name(), 1, "DG"),
    S.FEM(control_ports[1].get_name(), 1, "DG"),
    S.FEM(control_ports[2].get_name(), 1, "DG"),
    S.FEM(control_ports[3].get_name(), 1, "DG"),
    S.FEM("Damping", 2, "CG"),
]

# Add all of them to the dphs
for FEM in FEMs:
    wave_diss.add_FEM(FEM)
```

The parameter ν is obviously linked to the `Damping` port. It can be heterogeneous, as for the other parameters.

```
# Define physical parameters
parameters = [
    S.Parameter("T", "Young's modulus", "tensor-field", "[[5+x,x*y],[x*y,2+y]]", "q"),
    S.Parameter("rho", "Mass density", "scalar-field", "3-x", "p"),
    S.Parameter("nu", "viscosity", "scalar-field", "0.5*(2.0-x)", "Damping"),
]

# Add them to the dphs
for parameter in parameters:
    wave_diss.add_parameter(parameter)
```

Looking at (2.6), only 3 non-zero block matrices have to be added to the list of the already constructed bricks, for the Dirac structure part. And finally, 2 bricks are needed to discretize the resistive constitutive relation.

```
# Define the pHs via `Brick` == non-zero block matrices == variational terms
bricks = [
    ## Define the Dirac structure
    # Define the mass matrices from the left-hand side: the `flow` part of the Dirac␣
↪structure
    S.Brick("M_q", "q.Test_q", [1], dt=True, position="flow"),
    S.Brick("M_p", "p*Test_p", [1], dt=True, position="flow"),
    S.Brick("M_Y_B", "Y_B*Test_Y_B", [10], position="flow"),
    S.Brick("M_Y_R", "Y_R*Test_Y_R", [11], position="flow"),
    S.Brick("M_Y_T", "Y_T*Test_Y_T", [12], position="flow"),
    # Mass matrix
    S.Brick("M_r", "f_r*Test_f_r", [1], position="flow"),
    # The Dirichlet term is applied via Lagrange multiplier == the colocated output
    S.Brick("M_Y_L", "U_L*Test_Y_L", [13], position="flow"),
    # Define the matrices from the right-hand side: the `effort` part of the Dirac␣
↪structure
    S.Brick("D", "Grad(e_p).Test_q", [1], position="effort"),
    S.Brick("-D^T", "-e_q.Grad(Test_p)", [1], position="effort"),
    S.Brick("B_B", "U_B*Test_p", [10], position="effort"),
    S.Brick("B_R", "U_R*Test_p", [11], position="effort"),
    S.Brick("B_T", "U_T*Test_p", [12], position="effort"),
    # The "Identity" operator
    S.Brick("I_r", "e_r*Test_p", [1], position="effort"),
    # Minus its transpose
    S.Brick("-I_r^T", "-e_p*Test_f_r", [1], position="effort"),
```

(continues on next page)

(continued from previous page)

```
    # The Dirichlet term is applied via Lagrange multiplier == the colocated output
    S.Brick("B_L", "Y_L*Test_p", [13], position="effort"),
    S.Brick("C_B", "-e_p*Test_Y_B", [10], position="effort"),
    S.Brick("C_R", "-e_p*Test_Y_R", [11], position="effort"),
    S.Brick("C_T", "-e_p*Test_Y_T", [12], position="effort"),
    S.Brick("C_L", "-e_p*Test_Y_L", [13], position="effort"),
    ## Define the constitutive relations
    # Hooke's law under implicit form `- M_e_q e_q + CR_q q = 0`
    S.Brick("-M_e_q", "-e_q.Test_e_q", [1]),
    S.Brick("CR_q", "q.T.Test_e_q", [1]),
    # Linear momentum definition under implicit form `- M_e_p e_p + CR_p p = 0`
    S.Brick("-M_e_p", "-e_p*Test_e_p", [1]),
    S.Brick("CR_p", "p/rho*Test_e_p", [1]),
    # Constitutive relation: linear viscous fluid damping `- M_e_r e_r + CR_r f_r = 0`
    S.Brick("-M_e_r", "-e_r*Test_e_r", [1]),
    S.Brick("CR_r", "nu*f_r*Test_e_r", [1]),
]
```

Again, we use the previsouly defined `Brick` objects, thus, the whole system is constructed by adding all the bricks.

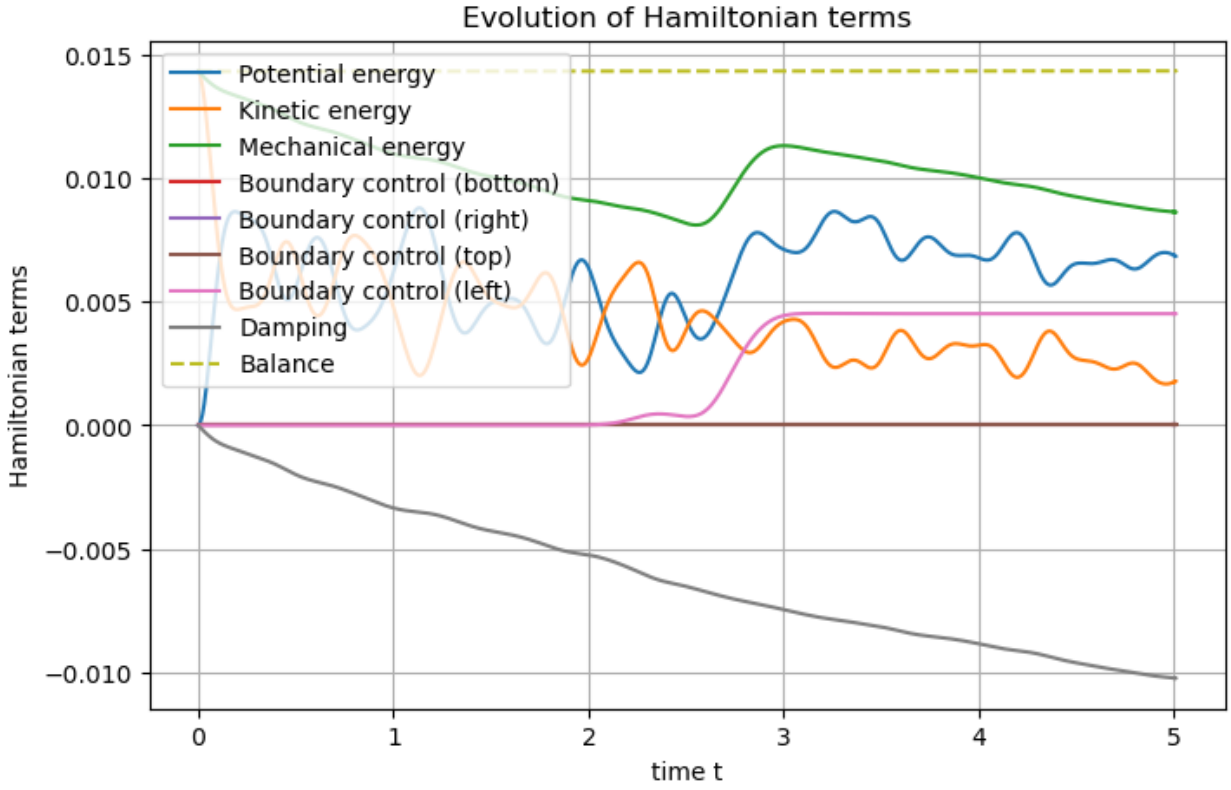
```
# Add all these `Bricks` to the dphs
for brick in bricks:
    wave_diss.add_brick(brick)
```

The initialization and solve steps are identical to the previous conservative case.

```
## Initialize the problem
# The controls expression, ordered as the control_ports
t_f = 5.
expressions = ["0.", "0.", "0.", f"0.1*sin(4.*t)*sin(4*pi*y)*exp(-10.*pow((0.5*{t_f}-t),
↪2))"]

# Add each expression to its control_port
for control_port, expression in zip(control_ports, expressions):
    # Set the control functions: it automatically constructs the related `Brick`s such␣
↪that `- M_u u + f(t) = 0`
    wave_diss.set_control(control_port.get_name(), expression)

# Set the initial data
q_0 = "[0., 0.]"
wave_diss.set_initial_value("q", q_0)
p_0 = "3**(-20*((x-0.5)*(x-0.5)+(y-0.5)*(y-0.5)))"
wave_diss.set_initial_value("p", p_0)

## Solve in time
# Define the time scheme
wave_diss.set_time_scheme(ts_type="cn",
                          t_f=t_f,
                          dt_save=0.01,
                          )

# Solve
wave_diss.solve()
```

Now one can define and plot the Hamiltonian.

```
## Post-processing
# Set Hamiltonian's name
wave_diss.hamiltonian.set_name("Mechanical energy")

# Define each Hamiltonian Term (needed to overwrite the previously computed solution)
terms = [
    S.Term("Potential energy", "0.5*q.T.q", [1]),
    S.Term("Kinetic energy", "0.5*p*p/rho", [1]),
]

# Add them to the Hamiltonian
for term in terms:
    wave_diss.hamiltonian.add_term(term)

# Plot the Hamiltonian and save the output
wave_diss.plot_Hamiltonian(save_figure=True, filename="Hamiltonian_Wave_2D_Dissipative.
↪png")
```



2.3.2 The heat equation

- file: examples/heat.py
- authors: Giuseppe Ferraro, Ghislain Haine
- date: 22 nov. 2022
- brief: 2D heat equation with Lyapunov Hamiltonian

`examples.heat.heat_eq()`

A structure-preserving discretization of the heat equation with mixed boundary control

Formulation with substitution of the co-state, Lyapunov L^2 functional, Div-Div, Mixed boundary condition on the Rectangle (including impedance-like absorbing boundary condition).

Setting

This example is the first simple case of intrinsically port-Hamiltonian Differential Algebraic Equation (known as pH-DAE).

The so-called *heat equation* is driven by the first law of thermodynamics.

Let $\Omega = (0,2) \times (0,1)$ be a bounded open connected set, with mass density $\rho(x)$, for all $x \in \Omega$, and n be the outward unit normal at the boundary $\partial\Omega$. We assume that:

- The domain Ω does not change over time: *i.e.* we work at constant volume in a solid
- No chemical reaction is to be found in the domain
- Dulong-Petit's model: internal energy is proportional to temperature

Let us denotes:

- u the internal energy density
- $\mathbf{J}_Q$ the heat flux
- T the local temperature
- $C_V := \left(\frac{du}{dT}\right)_V$ the isochoric heat capacity

The first law of thermodynamics, stating that in an isolated system, the energy is preserved, reads:

$$\rho(x)\partial_t u(t,x) = -\mathrm{div}\left(J_Q(t,x)\right), \qquad \forall t \geq 0, x \in \Omega.$$

Under Dulong-Petit's model, one has $u = C_V T$, which leads to

$$\rho(x)C_V(x)\partial_t T(t,x) = -\mathrm{div}\left(J_Q(t,x)\right), \qquad \forall t \geq 0, x \in \Omega.$$

As constitutive relation, the classical Fourier's law is considered:

$$J_Q(t,x) = -\lambda(x) \cdot \mathrm{grad}\left(T(t,x)\right), \qquad \forall t \geq 0, x \in \Omega,$$

where λ is the **tensor-valued** heat conductivity of the medium.

We assume furthermore that one wants to control the temperature $T = u_D$ at the lower, right and upper part of the boundary, denoted Γ_D (a **D**irichlet boundary condition), while the inward heat flux $-J_Q \cdot n = u_N$ will be prescribed at the left edge, denoted Γ_N (a **N**eumann boundary condition). Thus, the observations are $y_D = -J_Q \cdot n$ and $y_N = T$ respectively.

Port-Hamiltonian framework

Let us choose as Hamiltonian the usual quadratic form for parabolic equation

$$\mathcal{H}(T(t,x)) := \frac{1}{2}\int_\Omega \rho(x)Cv(x)T^2(t,x)\mathrm{d}x.$$

Computing the variational derivative with respect to the weigthed L^2-inner product $(\phi,\psi)_\Omega := \int_\Omega \rho(x)C_V(x)\phi(x)\psi(x)\mathrm{d}x$ leads to a co-state variable $e_T = T$. Hence, the first law of thermodynamics may be written as

$$\begin{pmatrix}\rho C_V T\\ \star\end{pmatrix} = \begin{bmatrix}0 & -\mathrm{div}\\ \star & 0\end{bmatrix}\begin{pmatrix}T\\ J_Q\end{pmatrix}.$$

As we want a *formally* skew-symmetric J operator, it has to be completed with $-\mathrm{grad}$, then

$$\begin{pmatrix}\rho C_V T\\ f_Q\end{pmatrix} = \begin{bmatrix}0 & -\mathrm{div}\\ -\mathrm{grad} & 0\end{bmatrix}\begin{pmatrix}T\\ J_Q\end{pmatrix},$$

and Fourier's law provides the constitutive relation $J_Q = \lambda f_Q$ to close the system.

Remark: ρC_V appears against the state variable as the weight of the L^2-inner product, it should not be ommited in the mass matrix at the discrete level.

The **power balance** satisfied by the **Hamiltonian** is

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathcal{H}(t) = \underbrace{-\int_\Omega \lambda\|f_Q(t,x)\|^2\mathrm{d}x}_{\text{dissipated power}} \underbrace{+\langle u_D(t,\cdot), y_D(t,\cdot)\rangle_{\Gamma_D}}_{\text{power flowing through }\Gamma_D} \underbrace{+\langle y_N(t,\cdot), u_N(t,\cdot)\rangle_{\Gamma_N}}_{\text{power flowing through }\Gamma_N},$$

where $\langle\cdot,\cdot\rangle_\Gamma$ is a boundary duality bracket $H^{\frac{1}{2}}, H^{-\frac{1}{2}}$ at the boundary Γ.

Structure-preserving discretization

Let φ_T and φ_Q be smooth test functions on Ω, and ψ_N and ψ_D be smooth test functions on Γ_N and Γ_D respectively. One can write the weak formulation of the **Dirac Structure** as follows

$$\left\{\begin{array}{rcl}\int_\Omega \rho(x)C_V(x)\partial_t T(t,x)\varphi_T(x)\mathrm{d}x &=& -\int_\Omega \mathrm{div}\left(J_Q(t,x)\right)\varphi_T(x)\mathrm{d}x,\\ \int_\Omega f_Q(t,x)\cdot\varphi_Q(x)\mathrm{d}x &=& -\int_\Omega \mathrm{grad}\left(T(t,x)\right)\cdot\varphi_Q(x)\mathrm{d}x,\\ \langle y_D,\psi_D\rangle_{\Gamma_D} &=& \langle -J_Q\cdot n,\psi_D\rangle_{\Gamma_D},\\ \langle u_N,\psi_N\rangle_{\Gamma_N} &=& \langle -J_Q\cdot n,\psi_N\rangle_{\Gamma_N}.\end{array}\right.$$

Integrating by parts the second line make the control u_N and the observation y_D appear

$$\int_\Omega f_Q(t,x)\cdot\varphi_Q(x)\mathrm{d}x = \int_\Omega T(t,x)\mathrm{div}\left(\varphi_Q(x)\right)\mathrm{d}x - \langle u_D,\varphi_Q\cdot n\rangle_{\Gamma_D} - \langle y_N,\varphi_Q\cdot n\rangle_{\Gamma_N}.$$

Now, let $(\varphi_T^i)_{1\le i\le N_T}\subset L^2(\Omega)$ and $(\varphi_Q^k)_{1\le k\le N_Q}\subset H_{\mathrm{div}}(\Omega)$ be two finite families of approximations for the T-type port and the Q-type port respectively, typically discontinuous and continuous Galerkin finite elements respectively. Denote also $(\psi_N^m)_{1\le m_N\le N_N}\subset H^{\frac{1}{2}}(\Gamma_N)$ and $(\psi_D^m)_{1\le m_N\le N_D}\subset H^{\frac{1}{2}}(\Gamma_D)$. In particular, the latter choices imply that the duality brackets at the boundary reduce to simple L^2 scalar products.

Writing the discrete weak formulation with those families, one has for all $1\le i\le N_T$, all $1\le k\le N_Q$, all $1\le m_N\le N_N$ and all $1\le m_D\le N_D$

$$\left\{\begin{array}{rcl}\sum_{j=1}^{N_T}\int_\Omega \varphi_T^j(x)\rho(x)C_V(x)\varphi_T^i(x)\mathrm{d}x\,\frac{\mathrm{d}}{\mathrm{d}t}T^j(t) &=& -\sum_{\ell=1}^{N_Q}\int_\Omega \mathrm{div}\left(\varphi_Q^\ell(x)\right)\varphi_T^i(x)\mathrm{d}x\,J_Q^\ell(t),\\ \sum_{\ell=1}^{N_Q}\int_\Omega \varphi_Q^\ell(x)\varphi_Q^k(x)\mathrm{d}x\,f_Q^\ell(t) &=& \sum_{j=1}^{N_Q}\int_\Omega \varphi_T^j(x)\mathrm{div}\left(\varphi_Q^k(x)\right)\mathrm{d}x\,T^j(t)\\ &&\quad -\sum_{n_D=1}^{N_D}\int_{\Gamma_D}\varphi_Q^k(s)\cdot n(s)\psi_D^{n_D}(s)\mathrm{d}s\,u_D^{n_D}(t)\\ &&\quad -\sum_{n_N=1}^{N_N}\int_{\Gamma_N}\varphi_Q^k(s)\cdot n(s)\psi_N^{n_N}(s)\mathrm{d}s\,y_N^{n_N}(t),\\ \sum_{n_D=1}^{N_D}\langle\psi_D^{n_D},\psi_D^{m_D}\rangle_{\Gamma_D}\,y_D^{n_D}(t) &=& -\sum_{\ell=1}^{N_p}\int_{\Gamma_D}\varphi_Q^\ell(s)\cdot n(s)\psi_D^{m_D}(s)\mathrm{d}s\,J_Q^\ell(t),\\ \sum_{n_N=1}^{N_N}\langle\psi_N^{n_N},\psi_N^{m_N}\rangle_{\Gamma_N}\,u_N^{n_N}(t) &=& -\sum_{\ell=1}^{N_p}\int_{\Gamma_N}\varphi_Q^\ell(s)\cdot n(s)\psi_N^{m_N}(s)\mathrm{d}s\,J_Q^\ell(t),\end{array}\right.$$

which rewrites in matrix form

$$\underbrace{\begin{bmatrix} M_T & 0 & 0 & 0 \\ 0 & M_Q & 0 & 0 \\ 0 & 0 & M_D & 0 \\ 0 & 0 & 0 & M_N \end{bmatrix}}_{=M} \begin{pmatrix} \frac{\mathrm{d}}{\mathrm{d}t}\underline{T}(t) \\ \underline{f_Q}(t) \\ -\underline{y_D}(t) \\ \underline{u_N}(t) \end{pmatrix} = \underbrace{\begin{bmatrix} 0 & D & 0 & 0 \\ -D^\top & 0 & B_D & -B_N^T \\ 0 & -B_D^\top & 0 & 0 \\ 0 & B_N & 0 & 0 \end{bmatrix}}_{=J} \begin{pmatrix} \underline{T}(t) \\ \underline{J_Q}(t) \\ \underline{u_D}(t) \\ -\underline{y_N}(t) \end{pmatrix},$$

where $\underline{\star}(t) := \begin{pmatrix} \star^1(t) & \cdots & \star^{N_\star} \end{pmatrix}^\top$ and

$$(M_T)_{ij} := \int_\Omega \varphi_T^j(x)\varphi_T^i(x)\mathrm{d}x, \qquad (M_Q)_{k\ell} := \int_\Omega \varphi_Q^\ell(x)\cdot\varphi_Q^k(x)\mathrm{d}x,$$

$$(M_D)_{m_D n_D} := \int_{\Gamma_D} \psi_D^{n_D}(s)\psi_D^{m_D}(s)\mathrm{d}s, \qquad (M_N)_{m_N n_N} := \int_{\Gamma_N} \psi_N^{n_N}(s)\psi_N^{m_N}(s)\mathrm{d}s,$$

$$(D)_{i\ell} := -\int_\Omega \mathrm{div}\left(\varphi_Q^\ell(x)\right)\cdot\varphi_T^i(x)\mathrm{d}x$$

$$(B_D)_{n_D k} := -\int_{\Gamma_D} \varphi_Q^k(s)\cdot n(s)\psi_D^{n_D}(s)\mathrm{d}s, \qquad (B_N)_{m_N \ell} := -\int_{\Gamma_N} \varphi_Q^\ell(s)\cdot n(s)\psi_N^{m_N}(s)\mathrm{d}s,,$$

Now one can approximate the **constitutive relation**

$$\int_\Omega J_Q(t,x)\cdot\varphi_Q(x)\mathrm{d}x = \int_\Omega f_Q(t,x)\cdot\lambda(x)\cdot\varphi_Q(x)\mathrm{d}x,$$

from which one can deduce the matrix form of the discrete weak formulation of the constitutive relation

$$M_Q\underline{J_Q}(t) = \Lambda\underline{f_Q}(t),$$

where

$$(\Lambda)_{k\ell} := \int_\Omega \varphi_Q^\ell(x)\cdot\lambda(x)\cdot\varphi_Q^k(x)\mathrm{d}x.$$

Finally, the **discrete Hamiltonian** $\mathcal{H}^d$ is defined as the evaluation of $\mathcal{H}$ on the approximation of the **state variable**

$$\mathcal{H}(t) := \mathcal{H}(T^d(t)) = \frac{1}{2}\underline{T}(t)^\top M_T\underline{T}(t).$$

The **discrete power balance** is then easily deduced from the above matrix formulations, thanks to the symmetry of M and the skew-symmetry of J

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathcal{H}^d(t) = -\underline{f_Q}(t)^\top\Lambda\underline{f_Q}(t)^\top + \underline{u_D}(t)^\top M_D\underline{y_D}(t) + \underline{y_N}(t)^\top M_N\underline{u_N}(t).$$

Simulation

As usual, we start by importing the **SCRIMP** package. Then we define the Distributed Port-Hamiltonian System and attach a (built-in) domain to it.

```
# Import scrimp
import scrimp as S

# Init the distributed port-Hamiltonian system
heat = S.DPHS("real")

# Set the domain (using the built-in geometry `Rectangle`)
# Omega = 1, Gamma_Bottom = 10, Gamma_Right = 11, Gamma_Top = 12, Gamma_Left = 13
heat.set_domain(S.Domain("Rectangle", {"L": 2.0, "l": 1.0, "h": 0.1}))
```

The next step is to define the state and its co-state. Care must be taken here: both are the temperature T, since the parameter ρC_V have been taken into account as a weight in the L^2-inner product. Hence, one may save some computational burden by using `substituted=True` which says to **SCRIMP** that the co-state is substituted into the state! Only **one** variable is approximated and will be computed in the sequel.

However, note that one could define a state e (namely the *internal energy*), and add Dulong-Petit's law as a constitutive relation $e = C_V T$ as usual.

```
# Define the variables and their discretizations and add them to the dphs
states = [
    S.State("T", "Temperature", "scalar-field"),
]
costates = [
    # Substituted=True indicates that only one variable has to be discretized on this␣
↪port
    S.CoState("T", "Temperature", states[0], substituted=True)
]
```

Let us define the algebraic port.

```
ports = [
    S.Port("Heat flux", "f_Q", "J_Q", "vector-field"),
]
```

And finally the control ports on each of the four boundary part.

```
control_ports = [
    S.Control_Port(
        "Boundary control (bottom)",
        "U_B",
        "Temperature",
        "Y_B",
        "- Normal heat flux",
        "scalar-field",
        region=10,
        position="effort",
    ),
    S.Control_Port(
        "Boundary control (right)",
        "U_R",
        "Temperature",
        "Y_R",
        "- Normal heat flux",
        "scalar-field",
        region=11,
        position="effort",
    ),
    S.Control_Port(
        "Boundary control (top)",
        "U_T",
        "Temperature",
        "Y_T",
        "- Normal heat flux",
        "scalar-field",
```

(continues on next page)

(continued from previous page)

```
        region=12,
        position="effort",
    ),
    S.Control_Port(
        "Boundary control (left)",
        "U_L",
        "- Normal heat flux",
        "Y_L",
        "Temperature",
        "scalar-field",
        region=13,
        position="flow",
    ),
]
```

Add all these objects to the DPHS.

```
for state in states:
    heat.add_state(state)
for costate in costates:
    heat.add_costate(costate)
for port in ports:
    heat.add_port(port)
for ctrl_port in control_ports:
    heat.add_control_port(ctrl_port)
```

Now, we must define the finite element families on each port. As stated in the beginning, **only the :math:`varphi_Q` family needs a stronger regularity**. Let us choose continuous Galerkin approximation of order 2. Then, the divergence of φ_Q is easily approximated by discontinuous Galerkin of order 1. At the boundary, this latter regularity will then occur, hence the choice of discontinuous Galerkin of order 1 as well.

```
FEMs = [
    S.FEM(states[0].get_name(), 1, FEM="DG"),
    S.FEM(ports[0].get_name(), 2, FEM="CG"),
    S.FEM(control_ports[0].get_name(), 1, FEM="DG"),
    S.FEM(control_ports[1].get_name(), 1, FEM="DG"),
    S.FEM(control_ports[2].get_name(), 1, FEM="DG"),
    S.FEM(control_ports[3].get_name(), 1, FEM="DG"),
]
for FEM in FEMs:
    heat.add_FEM(FEM)
```

It is now time to define the parameters, namely rho, C_V and λ. For the sake of simplicity, we assume that ρ will take C_V into account.

```
# Define the physical parameters
parameters = [
    S.Parameter("rho", "Mass density times heat capacity", "scalar-field", "3.", "T"),
    S.Parameter(
        "Lambda",
        "Heat conductivity",
        "tensor-field",
        "[[1e-2,0.],[0.,1e-2]]",
```

(continues on next page)

(continued from previous page)

```
        "Heat flux",
    ),
]
# Add them to the dphs
for parameter in parameters:
    heat.add_parameter(parameter)
```

Now the non-zero block matrices of the Dirac structure can be defined using the `Brick` object, as well as the constitutive relation, *i.e.* Fourier's law.

```
# Define the Dirac structure and the constitutive relations block matrices as `Brick`
bricks = [
    # Add the mass matrices from the left-hand side: the `flow` part of the Dirac␣
↪structure
    S.Brick("M_T", "T*rho*Test_T", [1], dt=True, position="flow"),
    S.Brick("M_Q", "f_Q.Test_f_Q", [1], position="flow"),
    S.Brick("M_Y_B", "Y_B*Test_Y_B", [10], position="flow"),
    S.Brick("M_Y_R", "Y_R*Test_Y_R", [11], position="flow"),
    S.Brick("M_Y_T", "Y_T*Test_Y_T", [12], position="flow"),
    # Normal trace is imposed by Lagrange multiplier on the left side == the collocated␣
↪output
    S.Brick("M_Y_L", "U_L*Test_Y_L", [13], position="flow"),
    # Add the matrices from the right-hand side: the `effort` part of the Dirac structure
    S.Brick("D", "-Div(J_Q)*Test_T", [1], position="effort"),
    S.Brick("-D^T", "T*Div(Test_f_Q)", [1], position="effort"),
    S.Brick("B_B", "-U_B*Test_f_Q.Normal", [10], position="effort"),
    S.Brick("B_R", "-U_R*Test_f_Q.Normal", [11], position="effort"),
    S.Brick("B_T", "-U_T*Test_f_Q.Normal", [12], position="effort"),
    # Normal trace is imposed by Lagrange multiplier on the left side == the collocated␣
↪output
    S.Brick("B_L", "-Y_L*Test_f_Q.Normal", [13], position="effort"),
    S.Brick("C_B", "J_Q.Normal*Test_Y_B", [10], position="effort"),
    S.Brick("C_R", "J_Q.Normal*Test_Y_R", [11], position="effort"),
    S.Brick("C_T", "J_Q.Normal*Test_Y_T", [12], position="effort"),
    S.Brick("C_L", "J_Q.Normal*Test_Y_L", [13], position="effort"),
    ## Define the constitutive relations as getfem `brick`
    # Fourier's law under implicit form - M_e_Q e_Q + CR_Q Q = 0
    S.Brick("-M_J_Q", "-J_Q.Test_J_Q", [1]),
    S.Brick("CR_Q", "f_Q.Lambda.Test_J_Q", [1]),
]
for brick in bricks:
    heat.add_brick(brick)
```

As controls, we assume that the temperature is prescribed, while the inward heat flux is proportional to the temperature (*i.e.* we consider an impedance-like absorbing boundary condition). This is easily achieved in **SCRIMP** by calling the variable in the expression of the control to apply.

The initial temperature profile is compatible with these controls, and has a positive bump centered in the domain.

```
# Initialize the problem
expressions = ["1.", "1.", "1.", "0.2*T"]

for control_port, expression in zip(control_ports, expressions):
```

(continues on next page)

(continued from previous page)

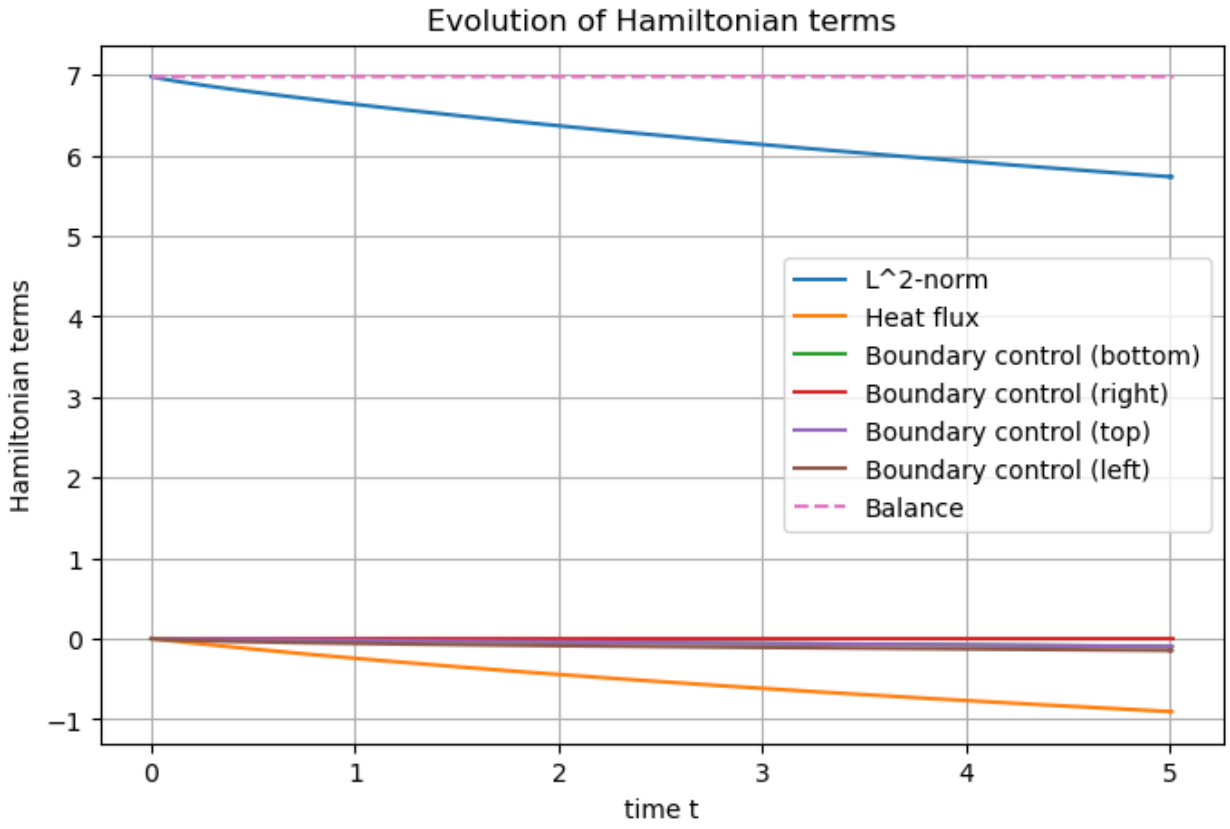
```
    # Set the control functions (automatic construction of bricks such that -M_u u +␣
↪f(t) = 0)
    heat.set_control(control_port.get_name(), expression)

# Set the initial data
heat.set_initial_value("T", "1. + 2.*np.exp(-50*((x-1)*(x-1)+(y-0.5)*(y-0.5))**2)")
```

We can now solve our Differential Algebraic Equation (DAE) using, *e.g.*, a Backward Differentiation Formula (BDF) of order 4.

```
## Solve in time
# Define the time scheme ("bdf" is backward differentiation formula)
heat.set_time_scheme(t_f=5.,
                     ts_type="bdf",
                     ts_bdf_order=4,
                     dt=0.01,
                     )

# Solve
heat.solve()
```

The Hamiltonian may be defined, computed and plot.

```
## Post-processing
# Set Hamiltonian name
heat.hamiltonian.set_name("Lyapunov formulation")
# Define the term
terms = [
    S.Term("L^2-norm", "0.5*T*rho*T", [1]),
]
# Add them to the Hamiltonian
for term in terms:
    heat.hamiltonian.add_term(term)

# Plot the Hamiltonian
heat.plot_Hamiltonian(save_figure=True, filename="Hamiltonian_Heat_2D.png")
```



2.3.3 Another wave equation

- file: examples/wave_coenergy.py
- authors: Giuseppe Ferraro, Ghislain Haine
- date: 22 nov. 2022
- brief: wave equations in co-energy formulation, two sub-domains

`examples.wave_coenergy.`**`wave_coenergy_eq`**`()`

A structure-preserving discretization of the wave equation with boundary control

Formulation co-energy, Grad-Grad, output feedback law at the boundary, damping on a subdomain

Setting

The objective of this example is to show how sub-domains may be used, and how substitutions reduce the computational burden: it assumes that this 2D wave example has already been studied.

Substitutions

The damped wave equation as a port-Hamiltonian system writes

$$\begin{pmatrix} \partial_t \alpha_q \\ \partial_t \alpha_p \\ f_r \end{pmatrix} = \begin{bmatrix} 0 & \text{grad} & 0 \\ \text{div} & 0 & -I \\ 0 & I^\top & 0 \end{bmatrix} \begin{pmatrix} e_q \\ e_p \\ e_r \end{pmatrix},$$

where α_q denotes the strain, α_p is the linear momentum, e_q is the stress, e_p is the velocity and (f_r, e_r) is the dissipative port.

This system must be close with **constitutive relations**, which are

$$e_q = T \cdot \alpha_q, \qquad e_p = \frac{\alpha_p}{\rho}, \qquad e_r = \nu f_r,$$

where T is the Young's modulus, ρ the mass density and ν the viscosity. Inverting these relations and substituting the results in the port-Hamiltonian system leads to the **co-energy formulation** (or more generally **co-state formulation**)

$$\begin{pmatrix} T^{-1} \cdot \partial_t e_q \\ \rho \partial_t e_p \\ \nu^{-1} e_r \end{pmatrix} = \begin{bmatrix} 0 & \text{grad} & 0 \\ \text{div} & 0 & -I \\ 0 & I^\top & 0 \end{bmatrix} \begin{pmatrix} e_q \\ e_p \\ e_r \end{pmatrix}.$$

At the discrete level, this allows to reduce the number of degrees of freedom by two.

Remark: In the example, ν only acts on a sub-domain, *i.e.* it is theoretically null on the complementary, and thus is not invertible! To be able to invert it, it is then mandatory to restrict the dissipative port to the sub-domain where $\nu > 0$.

Simulation

Let us start quickly until the definition of the dissipative port.

```
# Import scrimp
import scrimp as S

# Init the distributed port-Hamiltonian system
wave = S.DPHS("real")

# Set the domain (using the built-in geometry `Concentric`)
# Labels: Disk = 1, Annulus = 2, Interface = 10, Boundary = 20
omega = S.Domain("Concentric", {"R": 1.0, "r": 0.6, "h": 0.1})

# And add it to the dphs
wave.set_domain(omega)

## Define the variables
states = [
    S.State("q", "Stress", "vector-field"),
    S.State("p", "Velocity", "scalar-field"),
]
# Use of the `substituted=True` keyword to get the co-energy formulation
costates = [
    S.CoState("e_q", "Stress", states[0], substituted=True),
    S.CoState("e_p", "Velocity", states[1], substituted=True),
]
```

(continues on next page)

(continued from previous page)

```
# Add them to the dphs
for state in states:
    wave.add_state(state)
for costate in costates:
    wave.add_costate(costate)
```

In order to restrict the dissipative port to the internal disk, we use the `region` keyword.

```
# Define the dissipative port, only on the subdomain labelled 1 = the internal disk
ports = [
    S.Port("Damping", "e_r", "e_r", "scalar-field", substituted=True, region=1),
]

# Add it to the dphs
for port in ports:
    wave.add_port(port)
```

The control port is only at the external boundary, labelled by 20 in **SCRIMP**.

```
# Define the control port
control_ports = [
    S.Control_Port(
        "Boundary control",
        "U",
        "Normal force",
        "Y",
        "Velocity trace",
        "scalar-field",
        region=20,
    ),
]

# Add it to the dphs
for ctrl_port in control_ports:
    wave.add_control_port(ctrl_port)
```

The sequel is as for the already seen examples.

```
# Define the Finite Elements Method of each port
FEMs = [
    S.FEM(states[0].get_name(), 1, "DG"),
    S.FEM(states[1].get_name(), 2, "CG"),
    S.FEM(ports[0].get_name(), 1, "DG"),
    S.FEM(control_ports[0].get_name(), 1, "DG"),
]

# Add them to the dphs
for FEM in FEMs:
    wave.add_FEM(FEM)

# Define physical parameters: care must be taken,
# in the co-energy formulation, some parameters are
# inverted in comparison to the classical formulation
```

(continues on next page)

(continued from previous page)

```
parameters = [
    S.Parameter(
        "Tinv",
        "Young's modulus inverse",
        "tensor-field",
        "[[5+x,x*y],[x*y,2+y]]",
        "q",
    ),
    S.Parameter("rho", "Mass density", "scalar-field", "3-x", "p"),
    S.Parameter(
        "nu",
        "Viscosity",
        "scalar-field",
        "10*(0.36-(x*x+y*y))",
        ports[0].get_name(),
    ),
]

# Add them to the dphs
for parameter in parameters:
    wave.add_parameter(parameter)
```

Regarding the `Brick` objects, there is a major difference with the previous examples: here, we need to list **all** the subdomain labels for the wave equation, hence the `[1,2]`. On the other hand, the dissipation only occurs on the internal disk, labelled 1, and thus the block matrices corresponding to the identity operators which implement the dissipation **must be restrict to** `[1]`.

```
# Define the pHs via `Brick` == non-zero block matrices == variational terms
# Since we use co-energy formulation, constitutive relations are already taken into
# account in the mass matrices M_q and M_p
bricks = [
    ## Define the Dirac structure
    # Define the mass matrices from the left-hand side: the `flow` part of the Dirac␣
↪structure
    S.Brick("M_q", "q.Tinv.Test_q", [1, 2], dt=True, position="flow"),
    S.Brick("M_p", "p*rho*Test_p", [1, 2], dt=True, position="flow"),
    S.Brick("M_r", "e_r/nu*Test_e_r", [1], position="flow"),
    S.Brick("M_Y", "Y*Test_Y", [20], position="flow"),
    # Define the matrices from the right-hand side: the `effort` part of the Dirac␣
↪structure
    S.Brick("D", "Grad(p).Test_q", [1, 2], position="effort"),
    S.Brick("-D^T", "-q.Grad(Test_p)", [1, 2], position="effort"),
    S.Brick("I_r", "e_r*Test_p", [1], position="effort"),
    S.Brick("B", "U*Test_p", [20], position="effort"),
    S.Brick("-I_r^T", "-p*Test_e_r", [1], position="effort"),
    S.Brick("-B^T", "-p*Test_Y", [20], position="effort"),
    ## Define the constitutive relations
    # Already taken into account in the Dirac Structure!
]

# Add all these `Bricks` to the dphs
for brick in bricks:
```

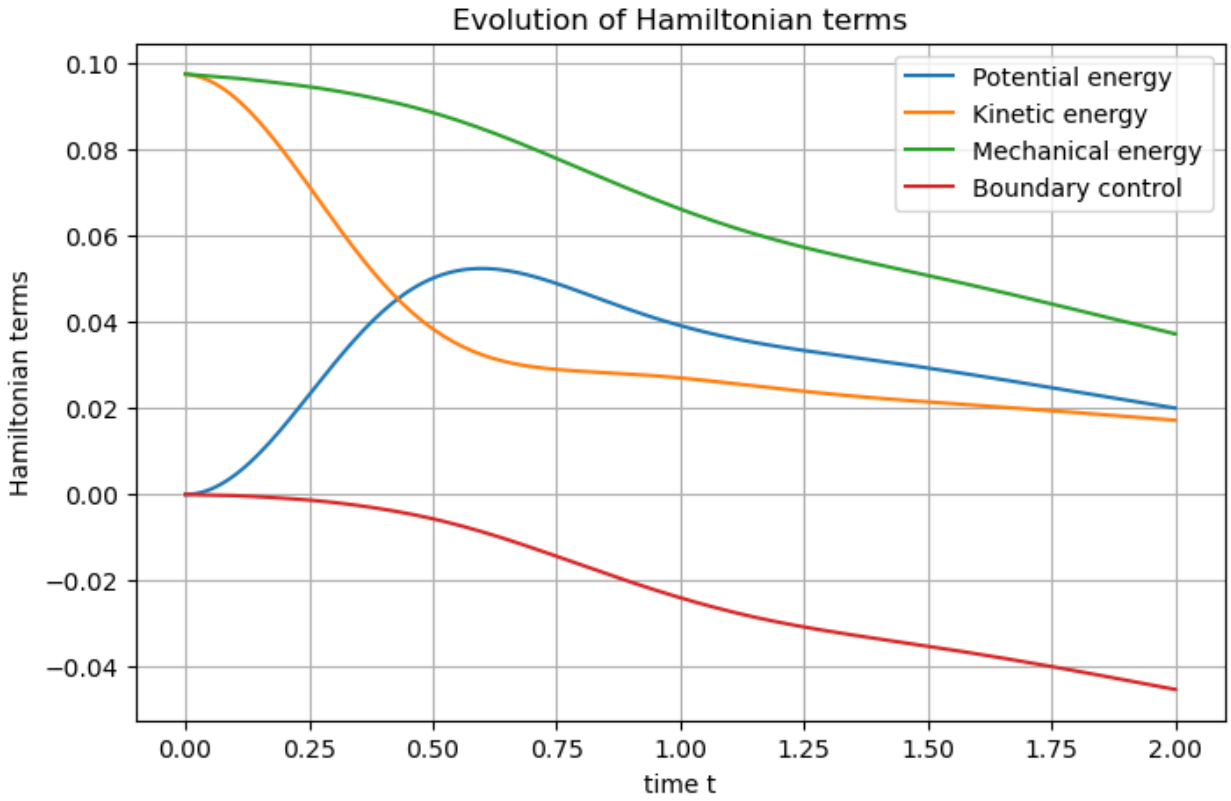
(continues on next page)

(continued from previous page)

```
    wave.add_brick(brick)
```

The remaining part of the code have already been explain in previous examples.

```
## Initialize the problem
# The controls expression
expressions = ["0.5*Y"]

# Add each expression to its control_port
for control_port, expression in zip(control_ports, expressions):
    # Set the control functions (automatic construction of bricks such that -M_u u +␣
↪f(t) = 0)
    wave.set_control(control_port.get_name(), expression)

# Set the initial data
wave.set_initial_value("q", "[0., 0.]")
wave.set_initial_value("p", "2.72**(-20*((x-0.5)*(x-0.5)+(y-0.5)*(y-0.5)))")

## Solve in time
# Define the time scheme ("cn" is Crank-Nicolson)
wave.set_time_scheme(ts_type="cn",
                     t_f=2.0,
                     dt_save=0.01,
                     )

# Solve
wave.solve()

## Post-processing
## Set Hamiltonian's name
wave.hamiltonian.set_name("Mechanical energy")
# Define each Hamiltonian Term
terms = [
    S.Term("Potential energy", "0.5*q.Tinv.q", [1, 2]),
    S.Term("Kinetic energy", "0.5*p*p*rho", [1, 2]),
]
# Add them to the Hamiltonian
for term in terms:
    wave.hamiltonian.add_term(term)

# Plot the Hamiltonian and save the output
wave.plot_Hamiltonian(save_figure=True)
```



2.3.4 Heat wave coupling

- file: sandbox/heat_hw.py
- authors: Giuseppe Ferraro, Ghislain Haine
- date: 15 dec. 2022
- brief: a 2D coupled heat-wave system

`examples.heat_wave.`**`heat_wave_eq`**(*heat_region=1*, *wave_region=2*)

A structure-preserving discretization of a coupled heat-wave equation

Co-energy formulations, heat: div-div, wave: grad-grad, gyrator interconnection On the *Concentric* built-in geometry: 1: internal disk, 2: exterior annulus

Args:

heat_region (int): the label of the region where the heat equation lies wave_region (int): the label of the region where the wave equation lies

Setting

It is assumed that the 2D wave equation, the 2D wave equation in co-energy formulation and the 2D heat equation have already been studied.

The objective of this example is to deal with interconnection in the sense of port-Hamiltonian systems.

We are interested in the coupled heat-wave system which can be formulated as follows: let $\Omega := \Omega_W \cup \Omega_H$ be a bounded domain in $\mathbb{R}^2$ such that $\Omega_W \cap \Omega_H = \emptyset$, we denote $\Gamma_I := \partial\Omega_W \cap \partial\Omega_H$ the interface between the two domains, and $\Gamma_W := \partial\Omega_W \setminus \Gamma_I$ and $\Gamma_H := \partial\Omega_H \setminus \Gamma_I$. The system of equations reads

$$\left\{\begin{array}{rcll} \partial_t T(t,x) & = & \operatorname{div}\left(\operatorname{grad}\left(T(t,x)\right)\right), & \forall t \geq 0, x \in \Omega_H, \\ \partial_{tt}^2 w(t,x) & = & \operatorname{div}\left(\operatorname{grad}\left(T(t,x)\right)\right), & \forall t \geq 0, x \in \Omega_H, \\ T(t,s) & = & 0, & \forall t \geq 0, s \in \Gamma_H, \\ w(t,s) & = & 0, & \forall t \geq 0, s \in \Gamma_W, \end{array}\right.$$

together with the transmission conditions across the interface

$$\left\{\begin{array}{rcll} T(t,s) & = & \partial_t w(t,s), & \forall t \geq 0, s \in \Gamma_I, \\ \partial_{n_H} T(t,s) & = & -\partial_{n_W} w(t,s), & \forall t \geq 0, s \in \Gamma_I, \end{array}\right.$$

where n_H is the outward normal to Ω_H and n_W is the outward normal to Ω_W. Hence, $n_H = -n_W$ on Γ_I.

Port-Hamiltonian framework

- The heat equation

The heat equation reads

$$\begin{pmatrix} \partial_t T \\ e_q \end{pmatrix} = \begin{bmatrix} 0 & -\operatorname{div} \\ -\operatorname{grad} & 0 \end{bmatrix} \begin{pmatrix} T \\ e_Q \end{pmatrix},$$

together with the boundary ports

$$\left\{\begin{array}{rcll} u_H^I & = & T, & \Gamma_I, \\ y_H^I & = & e_Q \cdot n_H, & \Gamma_I, \end{array}\right.$$

and

$$\left\{\begin{array}{rcll} u_H & = & T, & \Gamma_H, \\ y_H & = & e_Q \cdot n_H, & \Gamma_H. \end{array}\right.$$

- The wave equation

The Dirichlet boundary condition has to be relaxed by $\partial_t w = 0$ to fit the port-Hamiltonian framework. Providing this adaptation and the notation $p := \partial_t w$ and $q := \operatorname{grad}(w)$, the wave equation reads

$$\begin{pmatrix} \partial_t q \\ \partial_t p \end{pmatrix} = \begin{bmatrix} 0 & \operatorname{grad} \\ \operatorname{div} & 0 \end{bmatrix} \begin{pmatrix} q \\ p \end{pmatrix},$$

together with the boundary ports

$$\left\{\begin{array}{rcll} u_W^I & = & q \cdot n_W, & \Gamma_I, \\ y_W^I & = & p, & \Gamma_I, \end{array}\right.$$

and

$$\left\{\begin{array}{rcll} u_W & = & q \cdot n_W, & \Gamma_H, \\ y_W & = & p, & \Gamma_H. \end{array}\right.$$

- The interconnection

The transmission condition at the interface may be recast as a power-preserving interconnection. It can be either a **gyrator** or a **tranformer** interconnection, depending on the chosen causality for each system. We the above choices, we have a **gyrator interconnection**, indeed, one has

$$u_H^I = y_w^I, \qquad u_W^I = y_H^I.$$

Structure-preserving discretization

- The heat equation

We use the div-div formulation already presented in the 2D heat equation example, *i.e.* we obtain the following system

$$\underbrace{\begin{bmatrix} M_T & 0 & 0 & 0 \\ 0 & M_Q & 0 & 0 \\ 0 & 0 & M_H^I & 0 \\ 0 & 0 & 0 & M_H \end{bmatrix}}_{=M} \begin{pmatrix} \frac{\mathrm{d}}{\mathrm{d}t}\underline{T}(t) \\ \underline{e_Q}(t) \\ -\underline{y_H^I}(t) \\ -\underline{y_H}(t) \end{pmatrix} = \underbrace{\begin{bmatrix} 0 & D & 0 & 0 \\ -D^\top & 0 & B_H^I & B_H \\ 0 & -(B_H^I)^\top & 0 & 0 \\ 0 & -(B_H)^\top & 0 & 0 \end{bmatrix}}_{=J} \begin{pmatrix} \underline{T}(t) \\ \underline{e_Q}(t) \\ \underline{u_H^I}(t) \\ \underline{u_H}(t) \end{pmatrix},$$

- The wave equation

We use the grad-grad formulation already presented in the 2D wave equation example, *i.e.* we obtain the following system

$$\underbrace{\begin{bmatrix} M_q & 0 & 0 & 0 \\ 0 & M_p & 0 & 0 \\ 0 & 0 & M_W^I & 0 \\ 0 & 0 & 0 & M_W \end{bmatrix}}_{=M} \begin{pmatrix} \frac{\mathrm{d}}{\mathrm{d}t}\underline{q}(t) \\ \frac{\mathrm{d}}{\mathrm{d}t}\underline{p}(t) \\ -\underline{y_W^I}(t) \\ \underline{u_W}(t) \end{pmatrix} = \underbrace{\begin{bmatrix} 0 & D & 0 & 0 \\ -D^\top & 0 & B_W^I & -B_W^\top \\ 0 & -(B_W^I)^\top & 0 & 0 \\ 0 & B_W & 0 & 0 \end{bmatrix}}_{=J} \begin{pmatrix} \underline{q}(t) \\ \underline{p}(t) \\ \underline{u_W^I}(t) \\ -\underline{y_W}(t) \end{pmatrix},$$

- The transformer interconnection

This condition is easy to implement, and leads to

$$M_H^I \underline{u_H^I}(t) = M_W^I \underline{y_W^I}(t), \qquad M_W^I \underline{u_W^I}(t) = M_H^I \underline{y_H^I}(t).$$

Simulation

Let us start as usual, but using now the `Concentric` built-in geometry.

```
# Import scrimp
import scrimp as S

# Init the distributed port-Hamiltonian system
hw = S.DPHS("real")

# Set the domain (using the built-in geometry `Concentric`)
# Labels: Disk = 1, Annulus = 2, Interface = 10, Boundary = 20
omega = S.Domain("Concentric", {"R": 1.0, "r": 0.6, "h": 0.1})

# And add it to the dphs
hw.set_domain(omega)
```

It is important to remember here one of the objective of this example: to understand the `region` keyword.

For our study case, the heat equation will lie on a region `heat_region`, while the wave equation will lie on another region `wave_region`. And this has to be stated when defining the states and co-states, and everytime an integral (either the *weak forms* or the *Hamiltonian terms*) occurs.

```
# Define the states and costates, needs the heat and wave region's labels
heat_region = 1
wave_region = 2
states = [
    S.State("T", "Temperature", "scalar-field", region=heat_region),
    S.State("p", "Velocity", "scalar-field", region=wave_region),
    S.State("q", "Stress", "vector-field", region=wave_region),
]
# Use of the `substituted=True` keyword to get the co-energy formulation
costates = [
    S.CoState("T", "Temperature", states[0], substituted=True),
    S.CoState("p", "Velocity", states[1], substituted=True),
    S.CoState("q", "Stress", states[2], substituted=True),
]

# Add them to the dphs
for state in states:
    hw.add_state(state)
for costate in costates:
    hw.add_costate(costate)
```

The same is true for the resistive port for the heat equation.

```
# Define the algebraic port
ports = [
    S.Port("Heat flux", "e_Q", "e_Q", "vector-field", substituted=True, region=heat_
↪region),
]

# Add it to the dphs
for port in ports:
    hw.add_port(port)

# Define the control ports
control_ports = [
    S.Control_Port(
        "Interface Heat",
        "U_T",
        "Heat flux",
        "Y_T",
        "Temperature",
        "scalar-field",
        region=10,
        position="effort"
    ),
    S.Control_Port(
        "Interface Wave",
        "U_w",
```

(continues on next page)

(continued from previous page)

```
        "Velocity",
        "Y_w",
        "Velocity",
        "scalar-field",
        region=10,
        position="effort"
    ),
    # This port will be either for the wave or the heat equation
    # It corresponds to the exterior circle of radius R
    S.Control_Port(
        "Boundary",
        "U_bnd",
        "0",
        "Y_bnd",
        ".",
        "scalar-field",
        region=20,
        position="flow"
    ),
]

# Add it to the dphs
for ctrl_port in control_ports:
    hw.add_control_port(ctrl_port)
```

For the FEM choices, see the previous examples.

```
# Define the Finite Elements Method of each port
k = 1
FEMs = [
    S.FEM("T", k, "DG"),
    S.FEM("Heat flux", k+1, "CG"),
    S.FEM("Interface Heat", k, "DG"),
    S.FEM("p", k+1, "CG"),
    S.FEM("q", k, "DG"),
    S.FEM("Interface Wave", k, "DG"),
    S.FEM("Boundary", k, "DG"),
]

# Add them to the dphs
for FEM in FEMs:
    hw.add_FEM(FEM)
```

The `Brick` object does not have an *optional* `region` keyword, it is mandatory: more precisely, it requires a list of regions as third argument.

```
# Define the pHs via `Brick` == non-zero block matrices == variational terms
# Since we use co-energy formulation, constitutive relations are already taken into
# account in the mass matrices M_q and M_p
bricks = [
    # === Heat: div-div
    S.Brick("M_T", "T*Test_T", [heat_region], dt=True, position="flow"),
```

(continues on next page)

(continued from previous page)

```
    S.Brick("M_Q", "e_Q.Test_e_Q", [heat_region], position="flow"),
    S.Brick("M_Y_T", "Y_T*Test_Y_T", [10], position="flow"),

    S.Brick("D_T", "-Div(e_Q)*Test_T", [heat_region], position="effort"),
    S.Brick("D_T^T", "T*Div(Test_e_Q)", [heat_region], position="effort"),
    S.Brick("B_T", "U_T*Test_e_Q.Normal", [10], position="effort"),
    S.Brick("B_T^T", "e_Q.Normal*Test_Y_T", [10], position="effort"),

    # === Wave: grad-grad
    S.Brick("M_p", "p*Test_p", [wave_region], dt=True, position="flow"),
    S.Brick("M_q", "q.Test_q", [wave_region], dt=True, position="flow"),
    S.Brick("M_Y_w", "Y_w*Test_Y_w", [10], position="flow"),

    S.Brick("D_w", "-q.Grad(Test_p)", [wave_region], position="effort"),
    S.Brick("-D_w^T", "Grad(p).Test_q", [wave_region], position="effort"),
    S.Brick("B_w", "U_w*Test_p", [10], position="effort"),
    S.Brick("B_w^T", "p*Test_Y_w", [10], position="effort"),
]
# === Boundary depends on where is the heat equation / wave equation
if wave_region==1:
    bricks.append(S.Brick("M_Y_bnd", "Y_bnd*Test_Y_bnd", [20], position="flow"))
    bricks.append(S.Brick("B_bnd", "U_bnd*Test_e_Q.Normal", [20], position="effort"))
    bricks.append(S.Brick("B_bnd^T", "e_Q.Normal*Test_Y_bnd", [20], position="effort"))
else:
    bricks.append(S.Brick("M_Y_bnd", "U_bnd*Test_Y_bnd", [20], position="flow"))
    bricks.append(S.Brick("B_bnd", "Y_bnd*Test_p", [20], position="effort"))
    bricks.append(S.Brick("B_bnd^T", "p*Test_Y_bnd", [20], position="effort"))
for brick in bricks:
    hw.add_brick(brick)
```

Finally, the **gyrator** interconnection for a system is just an output feedback from the other. The subtility is that, while the normal along Γ_I depends from which side it is computed *on paper*, this is not the case *numerically*: a minus sign is necessary.

```
# Set the controls
# === Gyrator interconnection
hw.set_control("Interface Heat", "Y_w")
# CAREFUL: the numerical normal is the same for both sub-domains! Hence the minus sign.
hw.set_control("Interface Wave", "-Y_T")
# === Dirichlet boundary condition
hw.set_control("Boundary", "0.")

# Set the initial data
hw.set_initial_value("T", "5.*np.exp(-25*((x-0.6)*(x-0.6)+y*y))")
hw.set_initial_value("p", "5.*np.exp(-25*((x-0.6)*(x-0.6)+y*y))")
hw.set_initial_value("q", "[0.,0.]")

## Solve in time
# Define the time scheme ("bdf" is backward differentiation formula)
hw.set_time_scheme(ts_type="bdf",
                   t_f=15.,
                   dt=0.001,
```

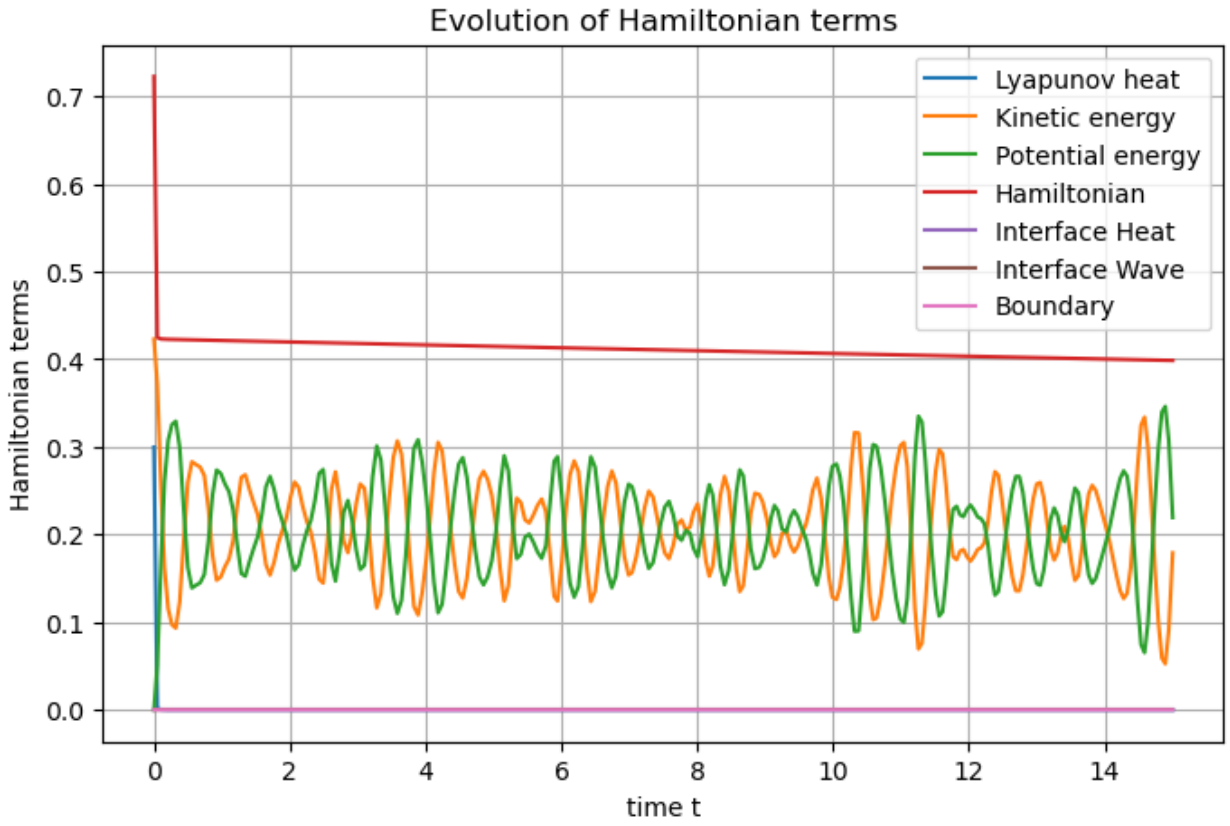
(continues on next page)

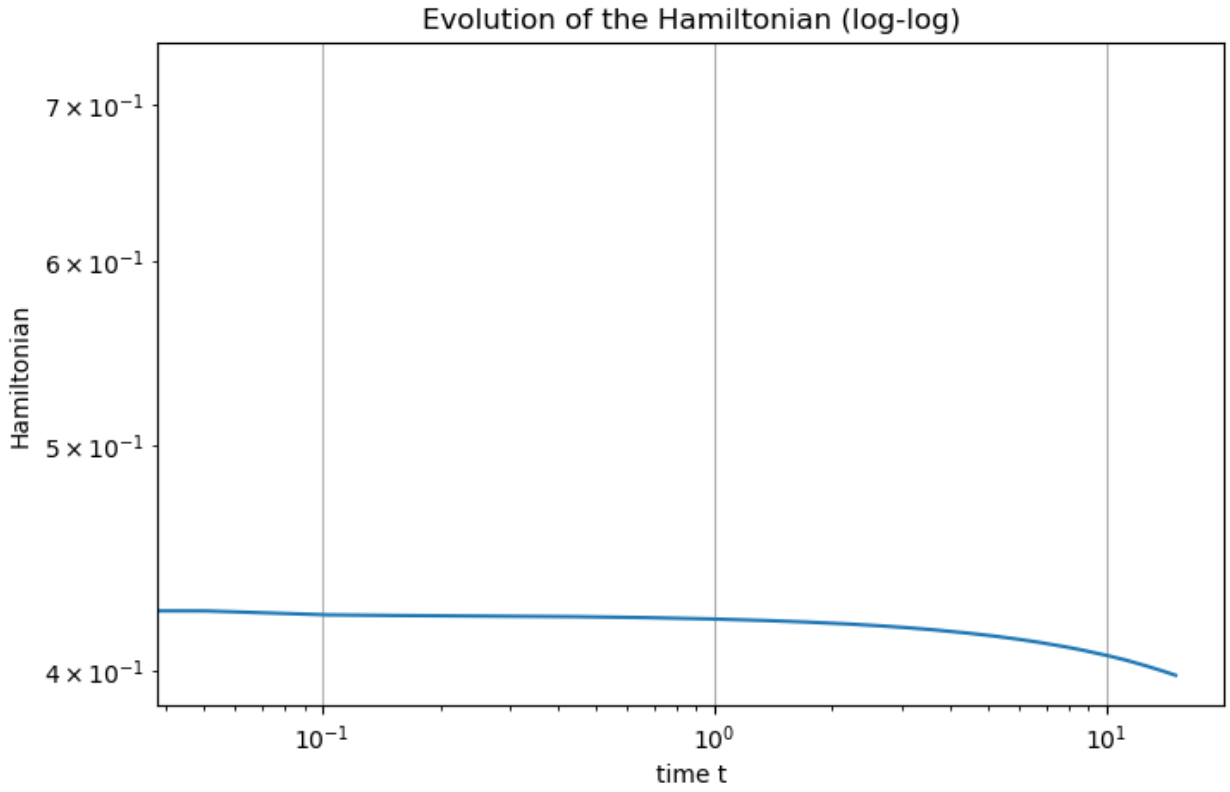
(continued from previous page)

```
                    dt_save=0.05,
                    ksp_type="preonly",
                    pc_type="lu",
                    pc_factor_mat_solver_type="mumps",
                    )

# Solve
hw.solve()
```

We end as usual with the Hamiltonian plot. Since our study case is known to be strongly stable, but never exponential nor uniformly in the initial state, we may also invocate the `get_Hamiltonian` method to make a log-log view of its evolution.

```
## Post-processing
## Set Hamiltonian's name
hw.hamiltonian.set_name("Hamiltonian")
# Define each Hamiltonian Term
terms = [
    S.Term("Lyapunov heat", "0.5*T*T", [heat_region]),
    S.Term("Kinetic energy", "0.5*p*p", [wave_region]),
    S.Term("Potential energy", "0.5*q.q", [wave_region]),
]
# Add them to the Hamiltonian
for term in terms:
    hw.hamiltonian.add_term(term)

# Plot the Hamiltonian and save the output
hw.plot_Hamiltonian(save_figure=True, filename="Hamiltonian_Heat"+str(heat_region)+"_Wave
↪"+str(wave_region)+"_2D.png")

# Plot the Hamiltonian in log-log scale
t = hw.solution["t"]
Hamiltonian = hw.get_Hamiltonian()
import matplotlib.pyplot as plt
fig = plt.figure(figsize=[8, 5])
ax = fig.add_subplot(111)
ax.loglog(t, Hamiltonian)
ax.grid(axis="both")
ax.set_xlabel("time t")
ax.set_ylabel("Hamiltonian")
ax.set_title("Evolution of the Hamiltonian (log-log)")
plt.show()
```


Evolution of Hamiltonian terms
Lyapunov heat
Kinetic energy
Potential energy
Hamiltonian
Interface Heat
Interface Wave
Boundary
Hamiltonian terms
time t
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
2
4
6
8
10
12
14


Evolution of the Hamiltonian (log-log)
Hamiltonian
time t
7×10^{-1}
6×10^{-1}
5×10^{-1}
4×10^{-1}
10^{-1}
10^{0}
10^{1}

2.3.5 The shallow water equation

- file: examples/shallow_water.py
- authors: Ghislain Haine
- date: 22 nov. 2022
- brief: inviscid shallow water equations

`examples.shallow_water.`**`shallow_water_eq`**`()`

A structure-preserving discretization of the inviscid shallow-water equation

Formulation Grad-Grad, homogeneous boundary condition, on a tank

Setting

The objective of this example is to show how to deal with **non-linearity**.

Let us consider a bounded domain $\Omega \subset \mathbb{R}^2$. The shallow water equations are constituted of two conservation laws

$$\begin{pmatrix} \partial_t h \\ \partial_t p \end{pmatrix} = \begin{bmatrix} 0 & -\mathrm{div} \\ -\mathrm{grad} & \frac{1}{h} G(\omega) \end{bmatrix} \begin{pmatrix} e_h \\ e_p \end{pmatrix},$$

where h is the height of the fluid, v is the velocity, ρ is the fluid density (supposed constant), $p := \rho v$ is the linear momentum, $\omega := \mathrm{curl}_{2D}(v) := \partial_x v_y - \partial_y v_x$ is the vorticity, $G(\omega) := \rho \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \omega$, $e_h = \frac{1}{2}\rho \|v\|^2 + \rho g h$ is the total pressure and $e_p = hv$ is the volumetric flow of the fluid. Thus, the first line of the matrix equation represents the conservation of the mass (or volume, since the fluid is assumed to be incompressible) and the second represents the conservation of linear momentum.

Port-Hamiltonian framework

One can define the system Hamiltonian (or total energy) as a functional of h and p, which are thus called energy variables

$$\mathcal{H}(h,p) := \frac{1}{2} \int_\Omega \frac{h(t,x)\|p(t,x)\|^2}{\rho} + \rho g h^2(t,x) \mathrm{d}x.$$

The co-energy variables can be computed from the variational derivative of the Hamiltonian such that

$$\begin{aligned} e_h &= \delta_h \mathcal{H} = \frac{1}{2}\rho \|v\|^2 + \rho g h, \\ e_p &= \delta_p \mathcal{H} = hv. \end{aligned}$$

The time-derivative of the Hamiltonian can then be obtained computed and depends only on the boundary variables

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathcal{H} = -\int_{\partial\Omega} e_h(t,s) e_p(t,s) \cdot n(s) \mathrm{d}s,$$

which enables to define collocated control and observation distributed ports along the boundary $\partial\Omega$. For example, one may define

$$\begin{aligned} u_\partial &= -e_p \cdot n, \\ y_\partial &= e_h, \end{aligned}$$

and the power-balance is given by a product between input and output boundary ports. The system is lossless, and conservative in the absence of control.

Structure-preserving discretization

First, let us multiply the linear momentum conservation equation by h.

Let us consider sufficiently regular test functions φ and Φ on Ω, and ψ test functions at the boundary $\partial\Omega$. The weak form of the previous equations reads

$$\left\{\begin{array}{rcl} (\partial_t h, \varphi)_{L^2} & = & -\left(\mathrm{div}\left(he_p\right), \varphi\right)_{L^2}, \\ (h\partial_t p, \Phi)_{(L^2)^2} & = & -\left(h\mathrm{grad}\left(e_h\right), \Phi\right)_{(L^2)^2} + \left(\mathrm{curl}_{2D}\left(p\right)\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} e_p, \Phi\right)_{(L^2)^2}, \\ (y_\partial, \psi)_{\partial\Omega} & = & (e_h, \psi)_{\partial\Omega}. \end{array}\right.$$

Applying integration by parts on the first line leads to

$$\left\{\begin{array}{rcl} (\partial_t h, \varphi)_{L^2} & = & (he_p, \mathrm{grad}\left(\varphi\right))_{L^2} + (hu_\partial, \varphi)_{\partial\Omega}, \\ (h\partial_t p, \Phi)_{(L^2)^2} & = & -\left(h\mathrm{grad}\left(e_h\right), \Phi\right)_{(L^2)^2} + \left(\mathrm{curl}_{2D}\left(p\right)\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} e_p, \Phi\right)_{(L^2)^2}, \\ (y_\partial, \psi)_{\partial\Omega} & = & (e_h, \psi)_{\partial\Omega}. \end{array}\right.$$

Furthermore, the weak form of the constitutive relations write

$$\left\{\begin{array}{rcl} (e_h, \varphi)_{L^2} & = & (\rho g h, \varphi)_{L^2} + \left(\frac{\|p\|^2}{2\rho}, \varphi\right)_{L^2}, \\ (e_p, \Phi)_{(L^2)^2} & = & \left(\frac{p}{\rho}, \Phi\right)_{(L^2)^2}. \end{array}\right.$$

Now, let us choose three finite families $(\varphi^i)_{1\leq i\leq N_h} \subset H^1(\Omega)$, $(\Phi^k)_{1\leq k\leq N_p} \subset (L^2(\Omega))^2$ and $(\psi^m)_{1\leq m\leq N_\partial}$ and project the weak formulations on them: for all $1 \leq i \leq N_h$, all $1 \leq k \leq N_p$ and all $1 \leq m \leq N_\partial$

$$\left\{\begin{array}{rcl} \sum_{j=1}^{N_h} \frac{\mathrm{d}}{\mathrm{d}t} h^j \left(\varphi^j, \varphi^i\right)_{L^2} & = & \sum_{\ell=1}^{N_p} e_p^\ell \left(h^d\Phi^\ell, \mathrm{grad}\left(\varphi^i\right)\right)_{L^2} + \sum_{n=1}^{N_\partial} u_\partial^n \left(h^d\psi^n, \varphi^i\right)_{\partial\Omega}, \\ \sum_{\ell=1}^{N_p} \frac{\mathrm{d}}{\mathrm{d}t} p^\ell \left(h^d\Phi^\ell, \Phi^k\right)_{(L^2)^2} & = & -\sum_{j=1}^{N_h} e_h^j \left(h^d\mathrm{grad}\left(\varphi^j\right), \Phi^k\right)_{(L^2)^2} \\ & & \quad + \sum_{\ell=1}^{N_p} e_p^\ell \left(\mathrm{curl}_{2D}\left(p^d\right)\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}\Phi^\ell, \Phi^k\right)_{(L^2)^2}, \\ \sum_{n=1}^{N_\partial} y_\partial^n \left(\psi^n, \psi^m\right)_{\partial\Omega} & = & \sum_{j=1}^{N_h} e_h^j \left(\varphi^j, \psi^m\right)_{\partial\Omega}, \end{array}\right.$$

where $h^d := \sum_{i=1}^{N_h} h^i\varphi^i$ is the approximation of h and $p^d := \sum_{k=1}^{N_p} p^k\Phi^k$ is the approximation of p. The constitutive relations read for all $1 \leq i \leq N_h$ and all $1 \leq k \leq N_p$

$$\left\{\begin{array}{rcl} \sum_{j=1}^{N_h} e_h^j \left(\varphi^j, \varphi^i\right)_{L^2} & = & \sum_{j=1}^{N_h} h^j \left(\rho g\varphi^j, \varphi^i\right)_{L^2} + \sum_{\ell=1}^{N_p} p^\ell \left(\frac{\Phi^\ell\cdot p^d}{2\rho}, \varphi^i\right)_{L^2}, \\ \sum_{\ell=1}^{N_p} e_p^\ell \left(\Phi^\ell, \Phi^k\right)_{(L^2)^2} & = & \sum_{\ell=1}^{N_p} p^\ell \left(\frac{\Phi^\ell}{\rho}, \Phi^k\right)_{(L^2)^2}. \end{array}\right.$$

Defining $\underline{\star}$ the vector gathering the coefficient of the approximation of the variable $\star$ in its appropriate finite family, one may write the discrete weak formulations in matrix notation

$$\begin{bmatrix} M_h & 0 & 0 \\ 0 & M_p[h^d] & 0 \\ 0 & 0 & M_\partial \end{bmatrix} \begin{pmatrix} \underline{h} \\ \underline{p} \\ -\underline{y_\partial} \end{pmatrix} = \begin{bmatrix} 0 & D[h^d] & B[h^d] \\ -D[h^d]^\top & G[p^d] & 0 \\ -B^\top & 0 & 0 \end{bmatrix} \begin{pmatrix} \underline{e_h} \\ \underline{e_p} \\ \underline{u_\partial} \end{pmatrix}.$$

where the matrices are given by

$$(M_h)_{ij} := \left(\varphi^j, \varphi^i\right)_{L^2} \qquad (M_p[h^d])_{k\ell} := \left(h^d\Phi^\ell, \Phi^k\right)_{(L^2)^2},$$

$$(D[h^d])_{i\ell} := \left(h^d\Phi^\ell, \mathrm{grad}\left(\varphi^i\right)\right)_{L^2}, \qquad (B[h^d])_{in} := \left(h^d\psi^n, \varphi^i\right)_{\partial\Omega},$$

and

$$(M_\partial)_{mn} := (\psi^n, \psi^m)_{\partial\Omega}, \qquad (B)_{in} := \left(\psi^n, \varphi^i\right)_{\partial\Omega}.$$

The constitutive relations read

$$\begin{bmatrix} M_h & 0 \\ 0 & M_p \end{bmatrix} \begin{pmatrix} \underline{e_h} \\ \underline{e_p} \end{pmatrix} = \begin{bmatrix} Q_h & P_h[h^d] \\ 0 & Q_p \end{bmatrix} \begin{pmatrix} \underline{h} \\ \underline{p} \end{pmatrix},$$

where the matrices are given by

$$(M_p)_{k\ell} := \left(\Phi^\ell, \Phi^k\right)_{(L^2)^2}, \qquad (Q_h)_{ij} := \left(\rho g \varphi^j, \varphi^i\right)_{L^2},$$

$$(P_h[h^d])_{i\ell} := \left(\frac{\Phi^\ell \cdot p^d}{2\rho}, \varphi^i\right)_{L^2}, \qquad (Q_d)_{k\ell} := \left(\frac{\Phi^\ell}{\rho}, \Phi^k\right)_{(L^2)^2}.$$

With these definition, one may prove the **discrete power balance**

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathcal{H}^d(t) = \underline{u_\partial}^\top(t) M_\partial \underline{y_\partial}(t).$$

Simulation

The beggining is classical: first import, then create the dphs and set the domain.

```
# Import scrimp
import scrimp as S

# Init the distributed port-Hamiltonian system
swe = S.DPHS("real")

# Set the domain (using the built-in geometry `Rectangle`)
# Labels: Omega = 1, Gamma_Bottom = 10, Gamma_Right = 11, Gamma_Top = 12, Gamma_Left = 13
swe.set_domain(S.Domain("Rectangle", {"L": 2.0, "l": 0.5, "h": 0.1}))
```

Then the states and co-states.

```
# Define the states and costates
states = [
    S.State("h", "Fluid height", "scalar-field"),
    S.State("p", "Linear momentum", "vector-field"),
]
costates = [
    S.CoState("e_h", "Pressure", states[0]),
    S.CoState("e_p", "Velocity", states[1]),
]

# Add them to the dphs
for state in states:
    swe.add_state(state)
for costate in costates:
    swe.add_costate(costate)
```

And the control ports.

```
# Define the control ports
control_ports = [
    S.Control_Port(
        "Boundary control 0",
```

(continues on next page)

(continued from previous page)

```
        "U_0",
        "Normal velocity",
        "Y_0",
        "Fluid height",
        "scalar-field",
        region=10,
        position="effort",
    ),
    S.Control_Port(
        "Boundary control 1",
        "U_1",
        "Normal velocity",
        "Y_1",
        "Fluid height",
        "scalar-field",
        region=11,
        position="effort",
    ),
    S.Control_Port(
        "Boundary control 2",
        "U_2",
        "Normal velocity",
        "Y_2",
        "Fluid height",
        "scalar-field",
        region=12,
        position="effort",
    ),
    S.Control_Port(
        "Boundary control 3",
        "U_3",
        "Normal velocity",
        "Y_3",
        "Fluid height",
        "scalar-field",
        region=13,
        position="effort",
    ),
]

# Add them to the dphs
for ctrl_port in control_ports:
    swe.add_control_port(ctrl_port)
```

Regarding the FEM, this is more challenging, as non-linearity are present. Nevertheless, let us stick to the way we choose until now: since the h-type variables will be derivated, thus we choose continuous Galerkin approximations of order $k+1$. The other energy variable is taken as continuous Galerkin approximation of order k, while boundary terms are given by discontinuous Galerkin approximations of order k.

```
# Define the Finite Elements Method of each port
k = 1
FEMs = [
```

(continues on next page)

(continued from previous page)

```
    S.FEM(states[0].get_name(), k+1, FEM="CG"),
    S.FEM(states[1].get_name(), k, FEM="CG"),
    S.FEM(control_ports[0].get_name(), k, "DG"),
    S.FEM(control_ports[1].get_name(), k, "DG"),
    S.FEM(control_ports[2].get_name(), k, "DG"),
    S.FEM(control_ports[3].get_name(), k, "DG"),
]

# Add them to the dphs
for FEM in FEMs:
    swe.add_FEM(FEM)
```

The parameters are *physically meaningful*!

```
# Define physical parameters
rho = 1000.
g = 10.
parameters = [
    S.Parameter("rho", "Mass density", "scalar-field", f"{rho}", "h"),
    S.Parameter("g", "Gravity", "scalar-field", f"{g}", "h"),
]

# Add them to the dphs
for parameter in parameters:
    swe.add_parameter(parameter)
```

Here are the difficult part. We need to define the weak form of each block matrices, and take non-linearities into account. To do so, the GFWL of GetFEM is transparent, but it is mandatory to say to **SCRIMP** that the `Brick` is non-linear, using the keyword `linear=False`. It is also possible to ask for this block to be considered *explicitly* in the time scheme (*i.e.* it will be computed with the previous time step values and be considered as a right-hand side), as done for the gyroscopic term below, using the keyword `explicit=True`.

```
# Define the pHs via `Brick` == non-zero block matrices == variational terms
# Some macros for the sake of readability
swe.gf_model.add_macro('div(v)', 'Trace(Grad(v))')
swe.gf_model.add_macro('Rot', '[[0,1],[-1,0]]')
swe.gf_model.add_macro('Curl2D(v)', 'div(Rot*v)')
swe.gf_model.add_macro('Gyro(v)', 'Curl2D(v)*Rot')
bricks = [
    # Define the mass matrices of the left-hand side of the "Dirac structure" (position=
↪"flow")
    S.Brick("M_h", "h * Test_h", [1], dt=True, position="flow"),
    S.Brick("M_p", "h * p . Test_p", [1], dt=True, linear=False, position="flow"),
    S.Brick("M_Y_0", "Y_0 * Test_Y_0", [10], position="flow"),
    S.Brick("M_Y_1", "Y_1 * Test_Y_1", [11], position="flow"),
    S.Brick("M_Y_2", "Y_2 * Test_Y_2", [12], position="flow"),
    S.Brick("M_Y_3", "Y_3 * Test_Y_3", [13], position="flow"),

    # Define the first line of the right-hand side of the "Dirac structure" (position=
↪"effort")
    S.Brick("-D^T", "h * e_p . Grad(Test_h)", [1], linear=False, position="effort"),
    # with the boundary control
```

(continues on next page)

(continued from previous page)

```
    S.Brick("B_0", "- U_0 * Test_h", [10], position="effort"),
    S.Brick("B_1", "- U_1 * Test_h", [11], position="effort"),
    S.Brick("B_2", "- U_2 * Test_h", [12], position="effort"),
    S.Brick("B_3", "- U_3 * Test_h", [13], position="effort"),
    # Define the second line of the right-hand side of the "Dirac structure" (position=
↪"effort")
    S.Brick("D", "- Grad(e_h) . Test_p * h", [1], linear=False, position="effort"),
    # with the gyroscopic term (beware that "Curl" is not available in the GWFL of␣
↪getfem)
    S.Brick("G", "(Gyro(p) * e_p) . Test_p", [1], linear=False, explicit=True, position=
↪"effort"),
    # Define the third line of the right-hand side of the "Dirac structure" (position=
↪"effort")
    S.Brick("C_0", "- e_h * Test_Y_0", [10], position="effort"),
    S.Brick("C_1", "- e_h * Test_Y_1", [11], position="effort"),
    S.Brick("C_2", "- e_h * Test_Y_2", [12], position="effort"),
    S.Brick("C_3", "- e_h * Test_Y_3", [13], position="effort"),

    ## Define the constitutive relations (position="constitutive", the default value)
    # For e_h: first the mass matrix WITH A MINUS because we want an implicit␣
↪formulation 0 = - M e_h + F(h)
    S.Brick("-M_e_h", "- e_h * Test_e_h", [1]),
    # second the linear part as a linear brick
    S.Brick("Q_h", "rho * g * h * Test_e_h", [1]),
    # third the non-linear part as a non-linear brick (linear=False)
    S.Brick("P_h", "0.5 * (p . p) / rho * Test_e_h", [1], linear=False),
    # For e_p: first the mass matrix WITH A MINUS because we want an implicit␣
↪formulation 0 = - M e_p + F(p)
    S.Brick("-M_e_p", "- e_p . Test_e_p", [1]),
    # second the LINEAR brick
    S.Brick("Q_p", "p / rho . Test_e_p", [1]),
]
for brick in bricks:
    swe.add_brick(brick)
```

As we just look at how it works, let us consider a step in the height, no initial velocity, and homogeneous Neumann boundary condition. This should look like a dam break experiment in a rectangular tank.

Remark: note the use of `np`, *i.e.* numpy, in the definition of the initial height h_0.

```
## Initialize the problem
swe.set_control("Boundary control 0", "0.")
swe.set_control("Boundary control 1", "0.")
swe.set_control("Boundary control 2", "0.")
swe.set_control("Boundary control 3", "0.")

# Set the initial data
swe.set_initial_value("h", "3. - (np.sign(x-0.5)+1)/3.")
swe.set_initial_value("p", f"[ 0., 0.]")

## Solve in time
# Define the time scheme
swe.set_time_scheme(
```

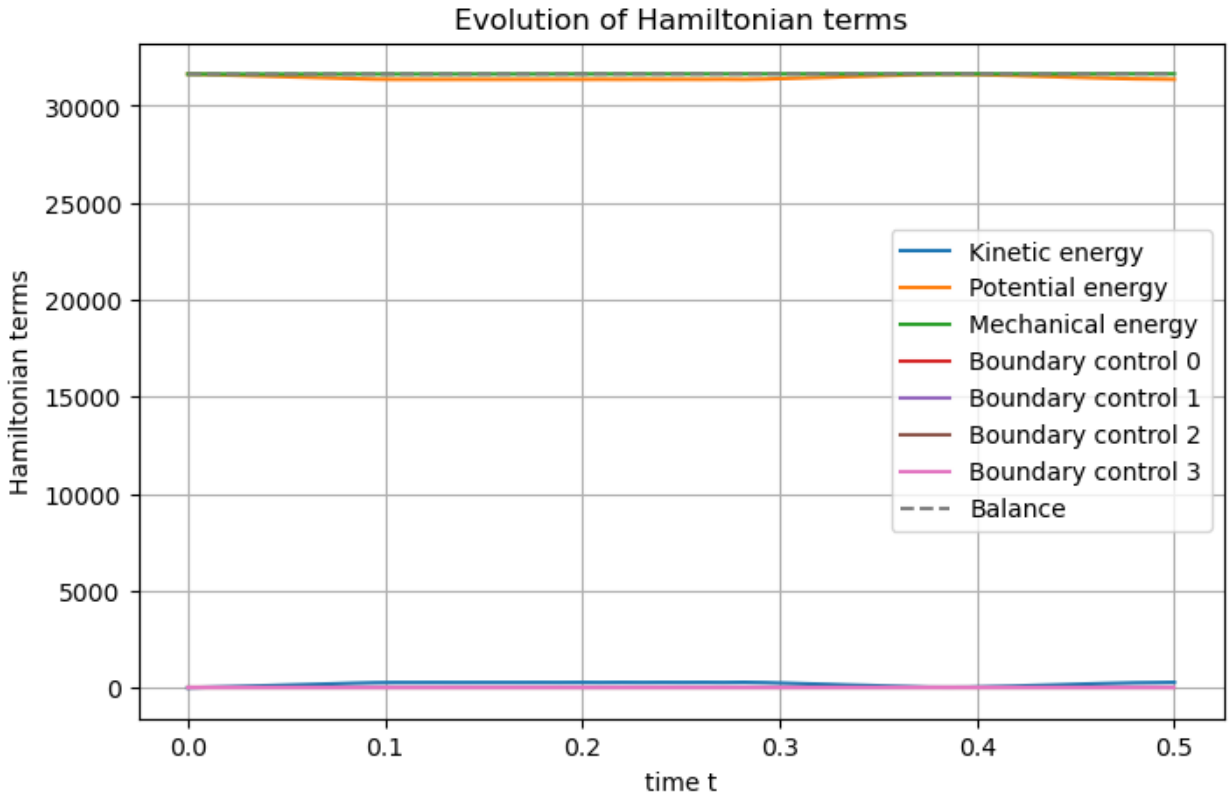
(continues on next page)

(continued from previous page)

```
    ts_type="bdf",
    ts_bdf_order=4,
    t_f=0.5,
    dt=0.0001,
    dt_save=0.01,
)

# Solve the system in time
swe.solve()
```

The Hamiltonian are then defined, computed, and shown.

```
## Post-processing
# Set Hamiltonian's name
swe.hamiltonian.set_name("Mechanical energy")
# Define Hamiltonian terms
terms = [
    S.Term("Kinetic energy", "0.5*h*p.p/rho", [1]),
    S.Term("Potential energy", "0.5*rho*g*h*h", [1]),
]
# Add them to the Hamiltonian
for term in terms:
    swe.hamiltonian.add_term(term)
# Plot the Hamiltonian
swe.plot_Hamiltonian(save_figure=True, filename="Hamiltonian_Inviscid_Shallow_Water_2D.
↪png")
```



2.4 Notebooks

Some examples coming from our publications are available in jupyter notebook format inside the `notebooks` folder in `examples`.

2.4.1 Install jupyter

To begin with, you'll need to install jupyter.

It can be done in the scrimp environment with:

1. *conda activate scrimp*
2. *conda install jupyter ipykernel*
3. *python -m ipykernel install –user –name scrimp –display-name "Python (scrimp)"*

2.4.2 Run jupyter

Then, run in the `notebooks` folder:

jupyter notebook &

And choose a notebook to launch.

If you aim at learning **SCRIMP**, the preferred order to study the notebooks is:

- Wave_1D
- Wave_2D
- Heat_2D
- Wave_2D_CoEnergy
- Heat_Wave_2D
- Shallow_water_2D

2.5 Graphical User Interface

As to increase the facility to perform simulations from scratch, a Graphical User Interface (GUI) is available to help beginners and confirmed users to scrimp and save time by sketching a first launchable script.

#TO COMPLETE#

2.6 Bibliography

Port-Hamiltonian systems is an ever-growing research area as it proposes a powerful framework for the control of multi-physics systems.

The following list of publications presents the main results of ours behind **SCRIMP**.

2.6.1 Articles

- Haine, Ghislain and Matignon, Denis and Serhani, Anass. **Numerical Analysis of a Structure-Preserving Space-Discretization for an Anisotropic and Heterogeneous Boundary Controlled N-Dimensional Wave Equation As a Port-Hamiltonian System.** (2023) *International Journal of Numerical Analysis and Modeling*, 20 (1). 92-133. DOI:10.4208/ijnam2023-1005
- Haine, Ghislain and Matignon, Denis and Monteghetti, Florian. **Long-time behavior of a coupled heat-wave system using a structure-preserving finite element method.** (2022) *Mathematical Reports*, 22 (1-2). 187-215. PDF
- Mora, Luis A. and Le Gorrec, Yann and Matignon, Denis and Ramirez, Hector and Yuz, Juan I.. **On port-Hamiltonian formulations of 3-dimensional compressible Newtonian fluids.** (2021) *Physics of Fluids*, 33 (11). 117117. DOI:10.1063/5.0067784
- Cardoso-Ribeiro, Flávio Luiz and Matignon, Denis and Lefèvre, Laurent. **A Partitioned Finite Element Method for power-preserving discretization of open systems of conservation laws.** (2021) *IMA Journal of Mathematical Control and Information*, 38 (2). 493-533. DOI:10.1093/imamci/dnaa038
- Brugnoli, Andrea and Haine, Ghislain and Serhani, Anass and Vasseur, Xavier. **Numerical Approximation of Port-Hamiltonian Systems for Hyperbolic or Parabolic PDEs with Boundary Control.** (2021) *Journal of Applied Mathematics and Physics*, 09 (06). 1278-1321. DOI:10.4236/jamp.2021.96088
- Brugnoli, Andrea and Alazard, Daniel and Pommier-Budinger, Valérie and Matignon, Denis. **Port-Hamiltonian flexible multibody dynamics.** (2021) *Multibody System Dynamics*, 51 (3). 343-375. DOI:10.1007/s11044-020-09758-6
- Brugnoli, Andrea and Alazard, Daniel and Pommier-Budinger, Valérie and Matignon, Denis. **A port-Hamiltonian formulation of linear thermoelasticity and its mixed finite element discretization.** (2021) *Journal of Thermal Stresses*, 44 (6). 643-661. DOI:10.1080/01495739.2021.1917322
- Cardoso-Ribeiro, Flávio Luiz and Matignon, Denis and Pommier-Budinger, Valérie. **Port-Hamiltonian model of two-dimensional shallow water equations in moving containers.** (2020) *IMA Journal of Mathematical Control and Information*, 37 (4). 1348-1366. DOI:10.1093/imamci/dnaa016
- Brugnoli, Andrea and Alazard, Daniel and Pommier-Budinger, Valérie and Matignon, Denis. **Port-Hamiltonian formulation and symplectic discretization of plate models Part I: Mindlin model for thick plates.** (2019) *Applied Mathematical Modelling*, 75. 940-960. DOI:10.1016/j.apm.2019.04.035
- Brugnoli, Andrea and Alazard, Daniel and Pommier-Budinger, Valérie and Matignon, Denis. **Port-Hamiltonian formulation and symplectic discretization of plate models. Part II : Kirchhoff model for thin plates.** (2019) *Applied Mathematical Modelling*, 75. 961-981. DOI:10.1016/j.apm.2019.04.036
- Aoues, Saïd and Cardoso-Ribeiro, Flávio Luiz and Matignon, Denis and Alazard, Daniel. **Modeling and Control of a Rotating Flexible Spacecraft: A Port-Hamiltonian Approach.** (2019) *IEEE Transactions on Control Systems Technology*, 27 (1). 355-362. DOI:10.1109/TCST.2017.2771244
- Cardoso-Ribeiro, Flávio Luiz and Matignon, Denis and Pommier-Budinger, Valérie. **A port-Hamiltonian model of liquid sloshing in moving containers and application to a fluid-structure system.** (2017) *Journal of Fluids and Structures*, 69. 402-427. DOI:10.1016/j.jfluidstructs.2016.12.007

2.6.2 Book Chapters

- Haine, Ghislain and Matignon, Denis. **Structure-Preserving Discretization of a Coupled Heat-Wave System, as Interconnected Port-Hamiltonian Systems.** (2021) In: *Geometric Science of Information.* Springer International Publishing AG, 191-199. DOI:10.1007/978-3-030-80209-7_22
- Serhani, Anass and Matignon, Denis and Haine, Ghislain. **A Partitioned Finite Element Method for the Structure-Preserving Discretization of Damped Infinite-Dimensional Port-Hamiltonian Systems with Boundary Control.** (2019) In: *Geometric Science of Information.* Springer International Publishing AG, Cham, Suisse, 549-558. DOI:10.1007/978-3-030-26980-7_57

2.6.3 Proceedings

- Brugnoli, Andrea and Haine, Ghislain and Matignon, Denis. **Explicit structure-preserving discretization of port-Hamiltonian systems with mixed boundary control.** (2022) In: 25th International Symposium on Mathematical Theory of Networks and Systems (MTNS 2022), 12 September 2022 - 16 September 2022 (Bayreuth, Germany).
- Haine, Ghislain and Matignon, Denis and Monteghetti, Florian. **Structure-preserving discretization of Maxwell's equations as a port-Hamiltonian system.** (2022) In: 25th International Symposium on Mathematical Theory of Networks and Systems (MTNS 2022), 12 September 2022 - 16 September 2022 (Bayreuth, Germany).
- Haine, Ghislain and Lefèvre, Laurent and Matignon, Denis. **PFEM: a mixed structure-preserving discretization method for port-Hamiltonian systems.** (2022) In: International Workshop on Operator Theory and its Applications, 6 September 2022 - 10 September 2022 (Cracovie, Poland).
- Brugnoli, Andrea and Matignon, Denis. **A port-Hamiltonian formulation for the full von-Kármán plate model.** (2022) In: 10th European Nonlinear Dynamics Conference (ENOC), 17 July 2022 - 22 July 2022 (Lyon, France).
- Cardoso-Ribeiro, Flávio Luiz and Matignon, Denis and Lefèvre, Laurent. **A Partitioned Finite Element Method (PFEM) for power-preserving discretization of port-Hamiltonian systems (pHs) with polynomial nonlinearity.** (2022) In: European Nonlinear Dynamics Conference (ENOC 2022), 17 July 2022 - 22 July 2022 (Lyon, France).
- Hélie, Thomas and Matignon, Denis. **Nonlinear damping laws preserving the eigenstructure of the momentum space for conservative linear PDE problems: a port-Hamiltonian modelling.** (2022) In: 10th European Nonlinear Dynamics Conference (ENOC), 17 July 2022 - 22 July 2022 (Lyon, France).
- Bendimerad-Hohl, Antoine and Haine, Ghislain and Matignon, Denis and Maschke, Bernhard. **Structure-preserving discretization of a coupled Allen-Cahn and heat equation system.** (2022) In: 4th IFAC Workshop on Thermodynamic Foundations of Mathematical Systems Theory - TFMST 2022, 25 July 2022 - 27 July 2022 (Montreal, Canada).
- Cardoso-Ribeiro, Flávio Luiz and Matignon, Denis and Lefèvre, Laurent. **Dissipative Shallow Water Equations: a port-Hamiltonian formulation.** (2021) In: Lagrangian and Hamiltonian Methodes for Nonlinear Control (7th LHMNC 2021), 11 October 2021 - 13 October 2021 (Berlin, Germany).
- Haine, Ghislain and Matignon, Denis. **Incompressible Navier-Stokes Equation as port-Hamiltonian systems: velocity formulation versus vorticity formulation.** (2021) In: Lagrangian and Hamiltonian Methodes for Nonlinear Control (7th LHMNC 2021), 11 October 2021 - 13 October 2021 (Berlin, Germany).
- Brugnoli, Andrea and Rashad, Ramy and Califano, Federico and Stramigioli, Stefano and Matignon, Denis. **Mixed finite elements for port-Hamiltonian models of von Kármán beams.** (2021) In: Lagrangian and Hamiltonian Methodes for Nonlinear Control (LHMNLC 2021), 11 October 2021 - 13 October 2021 (Berlin, Germany).

- Brugnoli, Andrea and Alazard, Daniel and Pommier-Budinger, Valérie and Matignon, Denis. **Structure-preserving discretization of port-Hamiltonian plate models.** (2021) In: Mathematical Theory of Networks and Systems, August 2021 - August 2021 (Cambridge, United Kingdom).
- Brugnoli, Andrea and Matignon, Denis and Haine, Ghislain and Serhani, Anass. **Numerics for Physics-Based PDEs with Boundary Control: the Partitioned Finite Element Method for Port-Hamiltonian Systems.** (2021) In: SIAM Conference on Computational Science and Engineering (CSE21), 1 March 2021 - 5 March 2021 (Virtual conference).
- Brugnoli, Andrea and Cardoso-Ribeiro, Flávio Luiz and Haine, Ghislain and Kotyczka, Paul. **Partitioned finite element method for structured discretization with mixed boundary conditions.** (2020) In: 21th IFAC World Congress, 11 July 2020 - 17 July 2020 (Berlin, Germany).
- Mora, Luis A. and Gorrec, Yann Le and Matignon, Denis and Ramirez, Hector and Yuz, Juan I.. **About Dissipative and Pseudo Port-Hamiltonian Formulations of Irreversible Newtonian Compressible Flows.** (2020) In: The 21st World Congress of The International Federation of Automatic Control (IFAC 2020), 11 July 2020 - 17 July 2020 (Vitual event, Germany).
- Payen, Gabriel and Matignon, Denis and Haine, Ghislain. **Modelling and structure-preserving discretization of Maxwell's equations as port-Hamiltonian system.** (2020) In: The 21st World Congress of The International Federation of Automatic Control (IFAC 2020), 11 July 2020 - 17 July 2020 (Virtual event, Germany).
- Treton, Anne-Sophie and Haine, Ghislain and Matignon, Denis. **Modelling the 1D piston problem as interconnected port-Hamiltonian systems.** (2020) In: The 21st World Congress of The International Federation of Automatic Control (IFAC 2020), 11 July 2020 - 17 July 2020 (Virtual event, Germany).
- Brugnoli, Andrea and Alazard, Daniel and Pommier-Budinger, Valérie and Matignon, Denis. **Interconnection of the Kirchhoff plate within the port-Hamiltonian framework.** (2020) In: 2019 IEEE 58th Conference on Decision and Control (CDC), 11 December 2019 - 13 December 2019 (Nice, France).
- Cardoso-Ribeiro, Flávio Luiz and Brugnoli, Andrea and Matignon, Denis and Lefèvre, Laurent. **Port-Hamiltonian modeling, discretization and feedback control of a circular water tank.** (2020) In: 2019 IEEE 58th Conference on Decision and Control (CDC), 11 December 2019 - 13 December 2019 (Nice, France).
- Serhani, Anass and Haine, Ghislain and Matignon, Denis. **Anisotropic heterogeneous n-D heat equation with boundary control and observation : I. Modeling as port-Hamiltonian system.** (2019) In: 3rd IFAC Workshop on Thermodynamic Foundations for a Mathematical Systems Theory (TFMST 2019), 3 July 2019 - 5 July 2019 (Louvain-la-Neuve, Belgium).
- Serhani, Anass and Haine, Ghislain and Matignon, Denis. **Anisotropic heterogeneous n-D heat equation with boundary control and observation : II. Structure-preserving discretization.** (2019) In: 3rd IFAC Workshop on Thermodynamic Foundations for a Mathematical Systems Theory (TFMST 2019), 3 July 2019 - 5 July 2019 (Louvain-la-Neuve, Belgium).
- Brugnoli, Andrea and Alazard, Daniel and Pommier-Budinger, Valérie and Matignon, Denis. **Partitioned finite element method for the Mindlin plate as a port-Hamiltonian system.** (2019) In: 3nd IFAC Workshop on Control of Systems Governed by Partial Differential Equations CPDE 2019, 20 May 2019 - 24 May 2019 (Oaxaca, Mexico). (Unpublished)
- Serhani, Anass and Matignon, Denis and Haine, Ghislain. **Partitioned Finite Element Method for port-Hamiltonian systems with Boundary Damping: Anisotropic Heterogeneous 2D wave equations.** (2019) In: 3rd IFAC/IEEE CSS Workshop on Control of Systems Governed by Partial Differential Equations CPDE and XI Workshop Control of Distributed Parameter Systems (CDPS 2019), 20 May 2019 - 24 May 2019 (Oaxaca, Mexico).
- Cardoso-Ribeiro, Flávio Luiz and Matignon, Denis and Lefèvre, Laurent. **A structure-preserving Partitioned Finite Element Method for the 2D wave equation.** (2018) In: 6th IFAC Workshop on Lagrangian and Hamiltonian Methods for Nonlinear Control, 1 May 2018 - 4 May 2018 (Valparaíso, Chile).

- Alazard, Daniel and Aoues, Saïd and Cardoso-Ribeiro, Flávio Luiz and Matignon, Denis. **Disturbance rejection for a rotating flexible spacecraft: a port-Hamiltonian approach.** (2018) In: 6th IFAC Workshop on Lagrangian and Hamiltonian Methods for Nonlinear Control, 1 May 2018 - 4 May 2018 (Valparaíso, Chile).
- Serhani, Anass and Matignon, Denis and Haine, Ghislain. **Structure-Preserving Finite Volume Method for 2D Linear and Non-Linear Port-Hamiltonian Systems.** (2018) In: 6th IFAC Workshop on Lagrangian and Hamiltonian Methods for Nonlinear Control, 1 May 2018 - 4 May 2018 (Valparaíso, Chile).

2.7 Code documentation

This part of the documentation is generated automatically from the python source using SPHINX.

2.7.1 Folders

scrimp.examples

We provide some examples coming from our publications.

The equations are explained here.

Wave

- file: examples/wave.py
- authors: Giuseppe Ferraro, Ghislain Haine
- date: 22 nov. 2022
- brief: 2D wave equations

examples.wave.**wave_eq**()

A structure-preserving discretization of the wave equation with mixed boundary control

Formulation DAE (energy/co-energy), Grad-Grad, Mixed boundary condition on the Rectangle Undamped case.

- file: examples/wave_coenergy.py
- authors: Giuseppe Ferraro, Ghislain Haine
- date: 22 nov. 2022
- brief: wave equations in co-energy formulation, two sub-domains

examples.wave_coenergy.**wave_coenergy_eq**()

A structure-preserving discretization of the wave equation with boundary control

Formulation co-energy, Grad-Grad, output feedback law at the boundary, damping on a subdomain

Heat

- file: examples/heat.py
- authors: Giuseppe Ferraro, Ghislain Haine
- date: 22 nov. 2022
- brief: 2D heat equation with Lyapunov Hamiltonian

examples.heat.**heat_eq**()

A structure-preserving discretization of the heat equation with mixed boundary control

Formulation with substitution of the co-state, Lyapunov L^2 functional, Div-Div, Mixed boundary condition on the Rectangle (including impedance-like absorbing boundary condition).

Heat-Wave coupling

- file: sandbox/heat_hw.py
- authors: Giuseppe Ferraro, Ghislain Haine
- date: 15 dec. 2022
- brief: a 2D coupled heat-wave system

examples.heat_wave.**heat_wave_eq**(*heat_region=1*, *wave_region=2*)

A structure-preserving discretization of a coupled heat-wave equation

Co-energy formulations, heat: div-div, wave: grad-grad, gyrator interconnection On the *Concentric* built-in geometry: 1: internal disk, 2: exterior annulus

Args:

heat_region (int): the label of the region where the heat equation lies wave_region (int): the label of the region where the wave equation lies

Shallow water

- file: examples/shallow_water.py
- authors: Ghislain Haine
- date: 22 nov. 2022
- brief: inviscid shallow water equations

examples.shallow_water.**shallow_water_eq**()

A structure-preserving discretization of the inviscid shallow-water equation

Formulation Grad-Grad, homogeneous boundary condition, on a tank

scrimp.utils

This folder contains useful functions that are beyond the port-Hamiltonian framework used in **SCRIMP**.

Config

- file: utils/config.py
- authors: Ghislain Haine
- date: 23 jun. 2023
- brief: functions to configure SCRIMP

scrimp.utils.config.**set_paths**(*path=None*)

Set the default path of scrimp

Args:
path (str): the path

scrimp.utils.config.**set_verbose**(*verbose=1*)

Set the verbosity level of scrimp (0: quiet, 1: info, 2: debug)

In *quiet* mode, debug are saved in a log file.

Args:
verbose (int): the level of verbosity, defaults to 1

scrimp.utils.config.**set_verbose_gf**(*verbose*)

Set the verbosity level of getfem

Args:
verbose (int): the level of verbosity

Linear algebra

- file: utils/linalg.py
- authors: Ghislain Haine, Florian Monteghetti
- date: 29 nov. 2022
- brief: linear algebra functions

scrimp.utils.linalg.**convert_PETSc_to_scipy**(*A*)

Convert from PETSc.Mat to scipy sparse (csr).

Args:
A (PETSc Mat): The matrix to convert

Returns:
scipy.sparse.csr.csr_matrix: the matrix A in scipy.sparse.csr.csr_matrix format

scrimp.utils.linalg.**convert_gmm_to_petsc**(*M*, *B*, *comm=petsc4py.PETSc.COMM_WORLD*)

Convert a GetFEM matrix M to a PETSc one B

Args:
M (SPMat GetFEM): matrix to transfer B (PETSc.Mat): matrix to fill M with comm (MPI_Comm): MPI communicator

Returns:
None

scrimp.utils.linalg.**extract_gmm_to_scipy**(*I*, *J*, *M*)

Extract a sub-matrix A from M, on interval I, J

Args:
I (Numpy array): line interval [begin, lenght] J (Numpy array): column interval [begin, lenght] M (SPMat GetFEM): matrix from which to extract the submatrix

Returns:
PETSc.Mat: matrix with value M(I,J) in CSR format

Returns:
scipy.sparse.csr.csr_matrix: the matrix A in scipy.sparse.csr.csr_matrix format

Mesh

- file: utils/mesh.py
- authors: Ghislain Haine
- date: 22 nov. 2022
- brief: built-in geometries for direct use in SCRIMP

scrimp.utils.mesh.**Ball**(*parameters={'R': 1.0, 'h': 0.1}*, *refine=0*, *terminal=1*)

The geometry of a Ball of radius R centered in (0,0,0) with mesh size h

- Domain *Omega*: 1,
- Boundary *Gamma*: 10

Args:

- parameters (dict): The dictionary of parameters for the geometry
- refine (int): Ask for iterative refinements by splitting elements
- terminal (int): An option to print meshing infos in the prompt, value *0* (quiet) or *1* (verbose, default)

Returns:
list[gf.Mesh, int, dict, dict]: The mesh to use with getfem, the dimension, a dict of regions with getfem indices for dim n and a dict of regions with getfem indices for dim n-1

scrimp.utils.mesh.**Concentric**(*parameters={'R': 1.0, 'h': 0.1, 'r': 0.6}*, *refine=0*, *terminal=1*)

The geometry of a Disk of radius r surrounded by an annulus of radii r and R with mesh size h

- Domain *Omega_Disk*: 1,
- Domain *Omega_Annulus*: 2,
- Interface *Interface*: 10,
- Boundary *Gamma*: 20

Args:

- parameters (dict): The dictionary of parameters for the geometry
- refine (int): Ask for iterative refinements by splitting elements
- terminal (int): An option to print meshing infos in the prompt, value *0* (quiet) or *1* (verbose, default)

Returns:
list[gf.Mesh, int, dict, dict]: The mesh to use with getfem, the dimension, a dict of regions with getfem indices for dim n and a dict of regions with getfem indices for dim n-1

scrimp.utils.mesh.**Disk**(*parameters={'R': 1.0, 'h': 0.1}*, *refine=0*, *terminal=1*)

The geometry of a Disk center in (0,0) with radius R and mesh size h

- Domain *Omega*: 1,
- Boundary *Gamma*: 10

Args:

- parameters (dict): The dictionary of parameters for the geometry
- refine (int): Ask for iterative refinements by splitting elements
- terminal (int): An option to print meshing infos in the prompt, value *0* (quiet) or *1* (verbose, default)

Returns:
list[gf.Mesh, int, dict, dict]: The mesh to use with getfem, the dimension, a dict of regions with getfem indices for dim n and a dict of regions with getfem indices for dim n-1

scrimp.utils.mesh.**Interval**(*parameters={'L': 1.0, 'h': 0.05}*, *refine=0*, *terminal=1*)

The geometry of a segment (0,L) with mesh size h

- Domain *Omega*: 1,
- Left boundary *Gamma_Left*: 10,
- Right boundary *Gamma_Right*: 11

Args:

- parameters (dict): The dictionary of parameters for the geometry
- refine (int): Ask for iterative refinements by splitting elements
- terminal (int): An option to print meshing infos in the prompt, value *0* (quiet) or *1* (verbose, default)

Returns:
list[gf.Mesh, int, dict, dict]: The mesh to use with getfem, the dimension, a dict of regions with getfem indices for dim n and a dict of regions with getfem indices for dim n-1

scrimp.utils.mesh.**Rectangle**(*parameters={'L': 2.0, 'h': 0.1, 'l': 1}*, *refine=0*, *terminal=1*)

The geometry of a Rectangle (0,L)x(0,l) with mesh size h

- Domain *Omega*: 1,
- Bottom boundary *Gamma_Bottom*: 10,
- Right boundary *Gamma_Right*: 11,
- Top boundary *Gamma_Top*: 12,
- Left boundary *Gamma_Left*: 13

Args:

- parameters (dict): The dictionary of parameters for the geometry
- refine (int): Ask for iterative refinements by splitting elements
- terminal (int): An option to print meshing infos in the prompt, value *0* (quiet) or *1* (verbose, default)

Returns:
list[gf.Mesh, int, dict, dict]: The mesh to use with getfem, the dimension, a dict of regions with getfem indices for dim n and a dict of regions with getfem indices for dim n-1

scrimp.utils.mesh.**built_in_geometries**()
A function to get all the infos about available built_in geometries

scrimp.sandbox

The `sandbox` folder is the **recommanded** folder to work in.

It already contains the example of the 1D wave equation presented here.

2.7.2 Distributed port-Hamiltonian system

- file: dphs.py
- authors: Giuseppe Ferraro, Ghislain Haine, Florian Monteghetti
- date: 22 nov. 2022
- brief: class for distributed port-Hamiltonian system

class scrimp.dphs.**DPHS**(*basis_field='real'*)
Bases: `object`

A generic class handling distributed pHs using the GetFEM tools

This is a wrapper in order to simplify the coding process

Access to fine tunings is preserved as much as possible

F
A PETSc Vec for residual computation

IFunction(*TS*, *t*, *z*, *zd*, *F*)
IFunction for the time-resolution of the dphs with PETSc TS fully implicit integrator

Args:
TS (PETSc TS): the PETSc TS object handling time-resolution t (float): time parameter z (PETSc Vec): the state zd (PETSc Vec): the time-derivative of the state F (PETSc Vec): the rhs vector

IJacobian(*TS*, *t*, *z*, *zd*, *sig*, *A*, *P*)
IJacobian for the time-resolution of the dphs with PETSc TS fully implicit integrator

Args:
TS (PETSc TS): the PETSc TS object handling time-resolution t (float): time parameter z (PETSc Vec): the state zd (PETSc Vec): the time-derivative of the state sig (float): a shift-parameter, depends on dt A (PETSc Mat): the jacobian matrix P:(PETSc Mat) the jacobian matrix to use for pre-conditionning

J
A PETSc Mat for Jacobian computation

add_FEM(*fem:* FEM)
This function adds a FEM (Finite Element Method) for the variables associated to a port of the dphs

TODO: handle *tensor-field*

Args:
fem (FEM): the FEM to use

add_brick(*brick:* Brick)

This function adds a *brick* in the getfem *Model* thanks to a form in GWFL getfem language

The form may be non-linear

Args:
brick (Brick): the brick

add_control_port(*control_port:* Control_Port)

This function adds a control *port* to the dphs

Args:
control_port (Control_Port): the Control Port.

add_costate(*costate:* CoState)

This function adds a costate to the costate dict of the dphs, then defines and adds the dynamical port gathering the couple (State, CoState).

Args:
costate (CoState): the costate

add_parameter(*parameter:* Parameter)

This function adds a time-independent (possibly space-varying parameter: x, y and z are the space variables) associated to a port of the dphs

Args:
parameter (Parameter): the parameter

add_port(*port:* Port)

This function adds a port to the port dict of the dphs

Args:
port (Port): the port

add_state(*state:* State)

This functions adds a state to state dict of the dphs.

Args:
state (State): the state

allocate_memory()

Pre-allocate memory for matrices and vectors

assemble_mass()

This function performs the assembly of the bricks dt=True and linear=True and set the PETSc.Mat attribute *mass*

assemble_nl_mass()

This function performs the assembly of the bricks dt=True and linear=False and set the PETSc.Mat attribute *nl_mass*

assemble_nl_stiffness()

This function performs the assembly of the bricks dt=False and linear=False and set the PETSc.Mat attribute *nl_stiffness*

assemble_rhs()

This function performs the assembly of the rhs and set the PETSc.Vec attribute *rhs*

assemble_stiffness()

This function performs the assembly of the bricks dt=False and linear=True and set the PETSc.Mat attribute *stiffness*

bricks

The dict of *bricks*, associating getfem *bricks* to petsc matrices obtained by the PFEM, store many infos for display()

buffer

A PETSc Vec buffering computation

compute_Hamiltonian()

Compute each *term* constituting the Hamiltonian

Wrapper to the function *compute* of Hamiltonian class

compute_powers()

Compute each power associated to each algebraic port if it is not substituted (because of the parameter-dependency if it is)

Wrapper to the function *compute* of Port class (loop on self.ports)

controls

The dict of *controls*, collecting information about control ports. Also appear in *ports* entries

costates

The dict of *costates*, store many infos for display()

disable_all_bricks()

This function disables all bricks in the *Model*

display(*verbose=2*)

A method giving infos about the dphs

Args:

- verbose (int): the level of verbosity (defaults to 2)

domain

The *domain* of a dphs is an object that handle mesh(es) and dict of regions with getfem indices (for each mesh), useful to define *bricks* (i.e. forms) in the getfem syntax

enable_all_bricks()

This function enables all bricks in the *Model*

event(*TS*, *t*, *z*, *fvalue*)

Check if the time step is not too small

exclude_algebraic_var_from_lte(*TS*)

Exclude the algebraic variable from the local error troncature in the time-resolution

Args:

TS (PETSc TS): the PETSc TS object handling time-resolution

export_matrices(*t=None*, *state=None*, *path=None*, *to='matlab'*)

TODO:

export_to_pv(*name_variable*, *path=None*, *t='All'*)

Export the solution to .vtu file(s) (for ParaView), with associated .pvd if t='All'

name_variable (str): the variable to export path (str): the path for the output file (default: in the *outputs/pv* folder next to your .py file) t (Numpy array): the time values of extraction (default: *All* the stored times), *All*, *Init*, *Final*

get_Hamiltonian()

This function returns the Hamiltonian in function of time

Returns:
numpy array: time array of the values of the Hamiltonian.

get_cleared_TS_options()

To ensure a safe database for the PETSc TS environment

get_quantity(*expression*, *region=-1*, *order=0*, *mesh_id=0*) → list

This functions computes the integral over region of the expression at each time step

Args:

- expression (str): the GFWL expression to compute
- region (int, optional): the id of the region (defaults to -1)
- order (int, optional): the order of the quantity to be computed (0: scalar, 1: vector, 2: tensor) (defaults to 0)
- mesh_id (int, optional): the id of the mesh (defaults to 0)

Returns:
list(float): a list of float at each time step (according to self.solution["t"])

get_solution(*name_variable*) → list

This functions is useful (especially in 1D) to handle post-processing by extracting the variable of interest from the PETSc Vec self.solution["z"]

Args:
name_variable (str): the name of the variable to extract

Returns:
list(numpy array): a list of numpy array (dofs of name_variable) at each time step (according to self.solution["t"])

gf_model

A getfem *Model* object that is use as core for the dphs

hamiltonian

The *Hamiltonian* of a dphs is a list of dict containing several useful information for each term

init_parameter(*name*, *name_port*)

This function initializes the parameter name in the FEM of the port name_port of the dphs and adds it to the getfem *Model*

Args:
name (str): the name of the parameter as defined with add_parameter() name_port (str): the name of the *port* where the parameter belongs

init_step()

Perform a first initial step with a pseudo bdf

It needs set_time_scheme(init_step=True)

initial_value_set

To check if the initial values have been set before time-resolution

linear_mass

To speed-up IFunction calls for linear systems

mass

Linear mass matrix of the system in PETSc CSR format

monitor(*TS*, *i*, *t*, *z*, *dt_save=1.0*, *t_0=0.0*, *initial_step=False*)

Monitor to use during iterations of time-integration at each successful time step

Args:

TS (PETSc TS): the PETSc TS object handling time-resolution i (int): the iteration in the time-resolution t (float): time parameter z (PETSc Vec): the state dt_save (float): save the solution each dt_save s (different from the time-step *dt* used for resolution) t_0 (float): the initial time initial_step (bool): *True* if this is the initial consistency step (default=`False`)

nl_mass

Non-linear mass matrix of the system in PETSc CSR format

nl_stiffness

Non-linear stiffness matrix of the system in PETSc CSR format

plot_Hamiltonian(*with_powers=True*, *save_figure=False*, *filename='Hamiltonian.png'*)

Plot each term constituting the Hamiltonian and the Hamiltonian

May include the *power terms*, i.e. the sum over [t_0, t_f] of the flow/effort product of algebraic ports

Args:

- with_powers (bool): if *True* (default), the plot will also contains the power of each algebraic ports
- save_figure (bool): if 'True' (defaults: False), save the plot
- filename (str): the name of the file where the plot is saved (defaults: *Hamiltonian.png*)

plot_powers(*ax=None*, *HamTot=None*)

Plot each power associated to each algebraic port

The time integration for visual comparison with the Hamiltonian is done using a midpoint method

If HamTot is provided, a *Balance* showing structure-preserving property is shown: must be constant on the plot

ax (Matplotlib axis): the gca of matplotlib when this function is called from plot_Hamiltonian() HamTot (Numpy array): the values of the Hamiltonian over time

ports

The dict of *ports*, store many infos for display()

postevent(*TS*, *event*, *t*, *z*, *forward*)

If the time step is too small, ask for the end of the simulation

powers_computed

To check if the powers have been computed

rhs

rhs of the system in PETSc Vec

set_control(*name*, *expression*)

This function applies a source term *expression* to the control port *name*

Args:
name (str): the name of the port expression (str): the expression of the source term

set_domain(*domain:* Domain)

This function sets a domain for the dphs.

TODO: If not built_in, given from a script 'name.py' or a .geo file with args in the dict 'parameters' should be able to handle several meshes e.g. for interconnections, hence the list type

Args:
name (str): id of the domain, either for built in, or user-defined auxiliary script parameters (dict): parameters for the construction, either for built in, or user-defined auxiliary script

set_from_vector(*name_variable*, *x*)

This function sets the value of the variable *name_variable* in the getfem *Model* from a numpy vector

Args:
name_variable (str): the name of the variable to set x (numpy array): the vector of values

set_initial_value(*name_variable*, *expression*)

This function sets the initial value of the variable *name_variable* of the dphs from an expression

Args:
name_variable (str): the name of the variable to set expression (str): the expression of the function to use

set_linear_flags()

This function set the "linear flags" to speed-up IFunction calls for linear systems.

set_time_scheme(***kwargs*)

Allows an easy setting of the PETSc TS environment

Args:
**kwargs: PETSc TS options and more (see examples)

solution

Will contain both time t and solution z

solve()

Perform the time-resolution of the dphs thanks to PETSc TS

The options database is set in the *time_scheme* attribute

solve_done

To check if the system has been solved

spy_Dirac(*t=None*, *state=None*)

!TO DO: improve a lot with position of bricks and no constitutive relations!!!

states

The dict of *states*, store many infos for display()

stiffness

Linear stiffness matrix of the system in PETSc CSR format

stop_TS

To stop TS integration by keeping already computed timesteps if one step fails

tangent_mass

Tangent (non-linear + linear) mass matrix of the system in PETSc CSR format

tangent_stiffness

Tangent (non-linear + linear) stiffness matrix of the system in PETSc CSR format

ts_start

For monitoring time in TS resolution

2.7.3 Domain

- file: domain.py
- authors: Giuseppe Ferraro, Ghislain Haine
- date: 31 may 2023
- brief: class for domain object

class scrimp.domain.**Domain**(*name: str*, *parameters: dict*, *refine=0*, *terminal=1*)

Bases: object

A class handling meshes and indices for regions

Lists are used to handle interconnection of pHs, allowing for several meshes in the dpHs.

display()

A method giving infos about the domain

get_boundaries() → list

This function gets the list of the bounderies for the domain.

Returns:
list: list of the boundaries for the domain

get_dim() → list

This function gets the list of dimensions for the domain.

Returns:
list: list of dimensions for the domain

get_isSet() → bool

This function gets the boolean vale indicating wether if a Mesh has been set for the domain or not.

Returns:
bool: boolean indicating if a Mesh has been set

get_mesh() → list

This function gets the list of mesh for the domain.

Returns:
list: list of mesh for the domain

get_name() → str

This function get the name of the domain.

Returns:
str: name of the domain.

get_subdomains() → list

This function gets the list of subdomains for the domain.

Returns:
list: list of the subdomains for the domain

set_mim_auto()

Define the integration method to a default choice

2.7.4 Hamiltonian / Term

- file: hamiltonian.py
- authors: Giuseppe Ferraro, Ghislain Haine
- date: 31 may 2023
- brief: class for hamiltonian and term objects

class scrimp.hamiltonian.**Hamiltonian**(*name: str*)

Bases: object

This class defines the Hamiltoninan.

add_term(*term:* Term)

This function adds a term to the term list of the Hamiltonian

Args:
term (Term): term for the Hamiltonian

compute(*solution: dict*, *gf_model: getfem.Model*, *domain:* Domain)

Compute each *term* constituting the Hamiltonian

Args:

- solutions (dict): The solution of the dphs
- gf_model (GetFEM Model): The model getfem of the dphs
- domain (Domain): The domain of the dphs

get_is_computed() → bool

This function returns True if the Hamiltonian is computed, False otherwise.

Returns:
bool: flag that indicates if the hamiltonian terms have been computed

get_name() → str

This function returns the name of the Hamiltonion.

Args:
name (str): name of the Hamiltonian

get_terms() → list

This function returns a copy of the list of terms of the Hamiltonian.

Returns:
list: list of terms of the Hamiltonian.

set_is_computed()

This function sets the Hamiltonian as computed.

set_name(*name: str*)

This function set the name for the Hamiltonion. For plotting purposes.

Args:
name (str): name of the Hamiltonian

class scrimp.hamiltonian.**Term**(*description: str*, *expression: str*, *regions: str*, *mesh_id: int = 0*)

Bases: object

This class defines a term for the Hamiltoninan.

get_description() → str

This function gets the description of the term.

Returns:
str: description of the term

get_expression() → str

This function gets the matematical expression of the term.

Returns:
str: matematical expression of the term.

get_mesh_id() → int

This function gets the mesh id of the mesh where the regions belong to..

Returns:
int: the mesh id of the mesh where the regions belong to.

get_regions() → str

This function gets the regions of the term.

Returns:
str: regions of the termthe region IDs of the mesh where the expression has to be evaluated

get_values() → list

This function gets the valeus of the term.

Returns:
list: list of values of the term

set_value(*value*)

This function sets a value for the term.

Args:
value : a value for the term

2.7.5 Port / Parameter

- file: port.py
- authors: Giuseppe Ferraro, Ghislain Haine
- date: 31 may 2023
- brief: class for port and parameter objects

class scrimp.port.**Parameter**(*name: str*, *description: str*, *kind: str*, *expression: str*, *name_port: str*)

Bases: object

This class describes the Parameter for a Port.

get_description() → str

This function gets the description of the parameter.

Returns:
str: description of the parameter

get_expression() → str

This function gets the matematical expression of the parameter.

Returns:
str: matematical expression of the parameter.

get_kind() → str

This function gets the kind of the parameter.

Returns:
str: kind of the parameter

get_name() → str

This function gets the name of the parameter.

Returns:
str: name of the parameter

get_name_port() → str

This function gets the name of the port whose the parameter is bounded.

Returns:
str: name of the port whose the parameter is bounded.

class scrimp.port.**Port**(*name: str*, *flow: str*, *effort: str*, *kind: str*, *mesh_id: int = 0*, *algebraic: bool = True*, *substituted: bool = False*, *dissipative: bool = True*, *region: int = None*)

Bases: object

A class to handle a port of a dpHs

It is mainly constituted of a flow variable, an effort variable, and a fem.

add_parameter(*parameter:* Parameter) → bool

This function adds a Parameter object that is acting on the variables of the port.

Args:
parameter (Parameter): parameter for the port.:

Returns:
bool: True if the insertion has been complete correctly, False otherwise

compute(*solution: dict*, *gf_model: getfem.Model*, *domain:* Domain)

Compute the power flowing through the algebraic port if it is not substituted (because of the parameter-dependency if it is)

Args:

- solutions (dict): The solution of the dphs
- gf_model (GetFEM Model): The model getfem of the dphs
- domain (Domain): The domain of the dphs

get_algebraic() → bool

This function gets boolean value of the algebraic parameter of the port.

If *True*, the equation associated to this port is algebraic, otherwise dynamic and the flow is derivated in time at resoltuion

Returns:

bool: value of the algebraic parameter

get_dissipative() → bool

This function gets boolean value for the dissipativeness flag of the port. If *True*, the power associated to the port gets a negative sign

Returns:

bool: value of the dissipativeness flag

get_effort() → str

This function gets the name of the effort variable.

Returns:

str: name of the effort variable

get_fem()

This function returns the fetfem Meshfem object to discretize the port.

Returns:

type: the getfem Meshfem object to discretize the port

get_flow() → str

This function gets the name of the flow variable.

Returns:

str: name of the flow variable

get_isSet() → bool

This funcion gets the boolean value that indicates wether the port is set or not.

Returns:

bool: value that indicates if the port is set.

get_is_computed() → bool

This function returns True if the power of the Port is computed, False otherwise.

Returns:

bool: flag that indicates if the power flowing through the Port has been computed

get_kind() → str

This function gets the type of the variables (e.g. *scalar-field*)

Returns:

str: type of the variables (e.g. *scalar-field*)

get_mesh_id() → int

This function gets he id of the mesh where the variables belong

Returns:

int: The id of the mesh where the variables belong

get_name() → str

This function gets the name of the port.

Returns:
str: name of the port

get_parameter(*name*) → *Parameter*

This function return the parameter with a specific name.

Args:
name (str): the name of the parameter of interest

Returns:
Parameter: the desired parameter, None otherwise

get_parameters() → list

This function returns the list of all the parameters inserted for the port.

Returns:
list(Parameter): list of all the parameters inserted for the port

get_power()

Gives access to the computed power of the port.

get_region() → int

This function gets the region of the mesh. If any, the int of the region of mesh_id where the flow/effort variables belong

Returns:
int: region of the mesh

get_substituted() → bool

This function gets boolean value of the subtituted parameter. If *True, the getfem `Model* will only have an unknown variable for the effort: the constitutive relation is substituted into the mass matrix on the flow side

Returns:
bool: value of the subtituted parameter

init_parameter(*name: str*, *expression: str*)

This function sets the chosen parameter object for the current port by initialization in the FE basis.

Args:
name (str): the name of the parameter object expression (str):

Returns:
out (numpy array): the evaluation of the parameter in the fem of the port.

set_fem(*fem:* FEM)

This function sets the Meshfem getfem object defining the finite element method to use to discretize the port.

Args:
fem (FEM): the FEM object to use

set_isSet() → bool

This funcion sets the boolean value that indicates the port is set.

Returns:
bool: value that indicates if the port is set.

set_is_computed()

This function sets the power of the Port as computed.

set_power(*power*)

This function sets the power along time of the Port.

2.7.6 State

- file: state.py
- authors: Giuseppe Ferraro, Ghislain Haine
- date: 31 may 2023
- brief: class for state object

class scrimp.state.**State**(*name: str*, *description: str*, *kind: str*, *region: int = None*, *mesh_id: int = 0*)

Bases: object

This class defines a State.

get_costate() → object

This function gets the Co-state of the state

Returns:
object: Costate

get_description() → str

This function gets the description of the State.

Returns:
str: description of the state

get_kind() → str

This function gets the kind of the State.

Returns:
str: kind of the state

get_mesh_id() → int

This funtion gets the integer number of the mesh of the state.

Returns:
int: id of the mesh

get_name() → str

This function gets the name of the State.

Returns:
str: name of the state

get_port() → object

This function gets the port of the state

Returns:
object: Port

get_region() → int

This function gets the integer number of the region of the state.

Returns:
int: region of the State.

set_costate(*costate*)

This function sets a Co-state to the State.

Args:
costate (Costate): Co-state

set_port(*port*)

This function sets a port to the state.

Args:
port (Port): Port

2.7.7 Co-state

- file: costate.py
- authors: Giuseppe Ferraro, Ghislain Haine
- date: 31 may 2023
- brief: class for co-state object

class scrimp.costate.**CoState**(*name: str*, *description: str*, *state:* State, *substituted=False*)

Bases: *State*

This class defines a Co-State.

get_state() → object

This function gets the State of the Costate.

Returns:
object: State

get_substituted() → bool

This function gets the boolean indicating wether to substitute the variable or not.

Returns:
bool: boolean indicating wether to substitute the variable

2.7.8 Control

- file: control.py
- authors: Giuseppe Ferraro, Ghislain Haine
- date: 31 may 2023
- brief: class for control port object

class scrimp.control.**Control_Port**(*name: str*, *name_control: str*, *description_control: str*, *name_observation: str*, *description_observation: str*, *kind: str*, *region: int = None*, *position: str = 'effort'*, *mesh_id: int = 0*)

Bases: *Port*

This class defines a Control Port.

get_description_control() → str

This function gets the description of the control.

Returns:
str: the description of the control

get_description_observation() → str

This function gets the description of the observation.

Returns:
str: the description of the observation

get_name_control() → str

This function gets the name of the control.

Returns:
str: the name of the control

get_name_obervation() → str

This function gets the name of the obervation.

Returns:
str: the name of the observation

get_position() → str

This function gets the position of the control in the Dirac structure.

Returns:
str: the position of the control

2.7.9 FEM

- file: fem.py
- authors: Giuseppe Ferraro, Ghislain Haine
- date: 31 may 2023
- brief: class for fem object

class scrimp.fem.**FEM**(*name*, *order*, *FEM='CG'*)

Bases: object

This class defines what is a FEM object in SCRIMP.

An negative order allows to access to GetFEM syntax for the FEM, e.g., by setting FEM="FEM_HERMITE(2)" for Hermite FE in dimension 2.

get_dim() → int

This function gets the dimension of the FEM.

Returns:
int: the dimension of FEM

get_fem()

This function returns the the FEM of getfem.

Returns:
gf.MeshFem: the FEM

get_isSet() → bool

This function gets the flag to know if the FEM are set in getfem.

Returns:
bool: the flag to assert setting in getfem

get_mesh()

This function gets the mesh of the FEM

Returns:
mesh: the mesh of FEM

get_name() → str

This function gets the name of the FEM.

Returns:
str: name of the FEM

get_order() → int

This function gets the order for the FEM.

Returns:
int: dim of the flow FEM

get_type() → str

This function gets the tyoe of the FEM.

Returns:
str: type of the FEM

set_dim(*dim: int*)

This function sets the dimension for the FEM.

Args:
dim (int): the dimension fro the FEM

set_fem()

This function sets the Meshfem getfem object defining the finite element method to use to discretize the port.

set_mesh(*mesh*)

This function sets the Meshfem getfem object FEM object of scrimp.

Args:
mesh (Mesh): the mesh where the FE are define

2.7.10 Brick

- file: brick.py
- authors: Giuseppe Ferraro, Ghislain Haine
- date: 31 may 2023
- brief: class for brick object

class scrimp.brick.**Brick**(*name: str*, *form: str*, *regions: list*, *linear: bool = True*, *dt: bool = False*, *position: str = 'constitutive'*, *explicit: bool = False*, *mesh_id: int = 0*)

Bases: object

This class defines a Brick.

add_id_brick_to_list(*id_brick: int*)

This function adds a brick ID to the brick ID list.

Args:
id_brick (int): the id of the brick

disable_id_bricks(*gf_model*)

This function disable the brick in the getfem model.

enable_id_bricks(*gf_model*)

This function enable the brick in the getfem model.

get_dt() → bool

This function returns the boolean that defines wether the matrices applied to time-derivative of a variable or not.

Returns:

bool: parameter to help easy identification of matrices applied to time-derivative of a variable (e.g. mass matrices).

get_explicit() → bool

This function returns the boolean that defines wether the brick is explicit or not.

Returns:

bool: parameter to help easy identification of explicit bricks.

get_form() → str

This function returns the form of the brick.

Returns:

str: the form in GWFL getfem language.

get_id_bricks() → list

This function returns the list of integers related to the ids of the bricks.

Returns:

list: the list of integers related to the ids of the bricks.

get_linear() → bool

This function returns the boolean that defines wether the brick is linear or not.

Returns:

bool: parameter to help easy identification of linear bricks.

get_mesh_id() → int

This function returns the ID of the brick.

Returns:

int: the id of the mesh where the form applies.

get_name() → str

This function returns the name of the brick.

Returns:

str: the name of the brick, will be used mainly for plotting purpose

get_position() → int

This function returns the id of the position where the form of the brick applies.

Returns:

int: the id of the mesh where the form applies. Defaults to 0.

get_regions() → list

This function returns the regions of the brick.

Returns:

list: the regions of mesh where the form applies.

CHAPTER

THREE

CREDITS

3.1 Development

Please report bug at: ghislain.haine@isae.fr, Giuseppe.Ferrarro@isae-supaero.fr

Current developers: Antoine Bendhimerad-Hohl, Giuseppe Ferraro, Michel Fournié, Ghislain Haine

Past: Andrea Brugnoli, Melvin Chopin, Florian Monteghetti, Anass Serhani, Xavier Vasseur

Please read the LICENSE

3.2 Funding

- ANR Project IMPACTS – IMplicit Port-hAmiltonian ConTrol Systems
- AID School Project FAMAS – Fast & Accurate MAxwell Solver
- ANR-DFG Project INFIDHEM – INterconnected inFinite-Dimensional systems for HEterogeneous Media

3.3 Third-party

The two **main** libraries used as core for SCRIMP are:

- GetFEM – An open-source finite element library
- PETSc – The Portable, Extensible Toolkit for Scientific Computation

Meshing is facilitated using (although not mandatory) GMSH – A three-dimensional finite element mesh generator

Post-processing visualization is encouraged via ParaView – Post-processing visualization engine

and finally, SCRIMP also needs for some routines

- matplotlib – Visualization with Python
- numpy – A well-known package for scientific computing

One of our choice for IDE is Spyder – A scientific Python development environment

3.4 How to cite SCRIMP?

Brugnoli, Andrea and Haine, Ghislain and Serhani, Anass and Vasseur, Xavier. *Numerical Approximation of Port-Hamiltonian Systems for Hyperbolic or Parabolic PDEs with Boundary Control.* (2021) **Journal of Applied Mathematics and Physics**, 09 (06). 1278-1321.

```
@article{Brugnoli2021,
author = {Brugnoli, Andrea and Haine, Ghislain and Serhani, Anass and Vasseur, Xavier},
title = { {Numerical Approximation of Port-Hamiltonian Systems for Hyperbolic or␣
↪Parabolic PDEs with Boundary Control} },
journal = {Journal of Applied Mathematics and Physics},
volume = {09},
issue = {06},
pages = {1278--1321},
year = {2021}
}
```

PYTHON MODULE INDEX

e

s

INDEX

M

N

P

R

S

T

W