1. Module 'mesh'

This module is contained in `mesh.py` and used to create a uniform and regular mesh on the *space-time* domain $[0,1] \times [0,T]$; an object of class `Mesh` creates data structures related to this kind of mesh.

1.1. Class: `Mesh`

Class constructor arguments: `N`, `K`, `T`. These arguments initialize the class data members of the same name below.

```
## Example: how to create a object of class Mesh
###################################
import mesh as msh

T = 2
N = 10
K = 20
# We create an object of class 'Mesh.
Th = msh.Mesh(N,K,T)

# Recovering the class data members 'T', 'N' and 'K'.
print(Th.T) # This prints '2'
print(Th.N) # This prints '10'
print(Th.K) # This prints '20'
```

Class data members:

1) `N`: integer number of subdivions of the interval $[0,1]$.
2) `K`: integer number of subdivions of the interval $[0,T]$.
3) `T`: (real number) length of time interval.
4) `DelT`: real number given by the expression `DelT=T/K`.
5) `DelX`: real number given by the expression `DelX=1/N`.
6) `NbPoints`: number of nodes in the mesh.
7) `Nelem`: number of triangles in the mesh. Is equal to `2NK`.
8) `points`: `(NbPoints,2)`-shaped Numpy array of real numbers. Each row contains the xy-coordinates of a point in the mesh.
9) `connect`: `(Nelem,4)`-shaped Numpy array of integers. This is the so-called **connectivity array**. Each row of this array contains 4 elements of integer type which completely determine a triangle of the uniform mesh:
   i) The first three columns on each row are the indices of the vertices of the triangle. For instance `connect[i,j]` would be the j-th ($j = 0,1,2$) point in the i-th triangle of the mesh; if we had an object `Th` of class `Mesh` and we wanted to return the coordinates of that point, we would only have to write `Th.points[self.connect[i,j],:]`. On each row of `connect`, the points are ordered counterclokwise.
   ii) The 4th item on each row contains information about the boundaries of the triangle. The values of this item can be: `0` (interior triangle), `1` (base boundary), `2` (right boundary), `3` (top boundary), `4` (left boundary).

10)-13) `base`: `(N+1,)`-shaped Numpy array of integers. It contains the row-indices of the nodes in `points` that belong to the base boundary. The Numpy arrays `right`, `top` and `left` are analogous and have shapes `(K+1,)`, `(N+1,)` and `(K+1,)` respectively.

2. Module QuadratureRules

This module contains a 2D gaussian quadrature rule on the reference triangle

$$\hat{T} = \{(x, y) \in \mathbb{R}^2 : x + y \leq 1, x, y \in (0, 1)\}$$

and a 1D gaussian quadrature rule on the interval $(-1, 1)$.

- `gaussRule2D`: `(19,3)`-shaped Numpy array of real numbers. It constains a 19 points gaussian quadrature rule over the reference triangle $\hat{T}$ used in [2]. Each row of `gaussRule2D` corresponds to a gaussian point of the quadrature rule. Each row stores a `(3,)`-shapes Numpy array containing (in this order) the x-coordinate, the y-coordiante and the weight of the gaussian point. The numerical evaluation of this rule was obtained using the MATLAB software developed by John Burkardt, available at `https://people.sc.fsu.edu/~jburkardt/m_src/triangle_dunavant_rule/triangle_dunavant_rule.html`
- `gaussRule1D`: `(6,1)`-shaped Numpy array of real numbers. It contains a 6 points gaussian quadrature rule on the interval $(-1, 1)$. Each row contains the x-coordinate of the gaussian point and its corresponding weight-

3. Module HCTMasterFunctions

3.1. Class: `MasterFunctions`

The class `MasterFunctions` has members which consist in arrays containing the evaluation of the master functions $Phi0$, $Phi1$, $Phi2$ and *beta* and their first and second derivatives on the reference triangle. References for both the master functions and the reference triangle are given in reference [2].

Class constructor arguments: `deg`, mask-type argument, the value by default is `deg=[True,True,True,True]`. `deg` must be a list of 4 elements of boolean type, i.e, the possible values for each item in `deg` are `True` or `False`. The values of `deg[i]` activate (if `deg[i]` stores a `True` value) or deactivate (if `deg[i]` stores a `False` value) the initialization of:

- `i=0`: the class data memebrs `gPhi01d`,...,`gPhi21d`, `gbeta1d`.
- `i=1`: the class data memebrs `gPhi02d`,...,`gPhi22d`, `gbeta2d`.
- `i=2`: the class data memebrs `gDPhi0`,...,`gDPhi2`, `gDbeta`.
- `i=3`: the class data memebrs `gD2Phi0`,...,`gD2Phi2`, `gD2beta`.

The initialization is carried out evaluating the Master Functions and the corresponding derivatives activated by `deg` at the specified gaussian points (see the specification fthe class data memebrs below) and storing them in the class data members 1)-16) which are activated by the `deg` mask.

```
## Example: how to create a object of class MasterFunctions
####################################
import HCTMasterFunctions as mf

eval_mask = [True,True,True,True]
# We create an object of class MaserFunctions
MasterFunctionsAtGaussianPoints = mf.MasterFuncions(eval_mask)
```

Class data members:

1)-3) : `gPhi01d`,...,`gPhi21d`: `(N,1,3)`-shaped Numpy arrays of real numbers, where `N` is the number of gaussian points in the 1D quadrature rule imported from the module `QuadratureRules`. These arrays are intended to contain the evaluation of the Master Functions $Phi0$, $Phi1$, $Phi2$ (defined in [2]) at the gaussian points of the 1D gaussian quadrature rule imported from `QuadratureRules`.

4) `gbeta1d`: `(N,1)`-shaped Numpy array of real numbers (`N` represents the same number as for class data memebrs 1)-3)). This array is inteded to store the evaluation of the functions *beta* (defined in [2]) at the gaussian points of the 1D gaussian quadrature rule imported from `QuadratureRules`.

5)-7) : `gPhi02d`,...,`gPhi22d`: `(N,1,3)`-shaped Numpy arrays, where `N` is the number of gaussian points in the 2D quadrature rule imported from `QuadratureRules`. Functions analogous to class data memebers 1)-3) but using a 2D gaussian quadrature rule.

8) `gbeta2d`: `(N,1)`-shaped Numpy array of real numbers with the same `N` as in 5)-7).

9)-11) : `gDPhi0`,...,`gDPhi2`: `(N,2,3)`-shaped Numpy arrays of reals numbers, where `N` represents the number of gaussian points in the 1D gaussian quadrature rule imported from `QuadratureRules`. These arrays are inteded to store the evaluation of the gradients of the Master Functions $Phi0$, $Phi1$, $Phi2$ at the gaussian points of the 1D quadrature rule.

12) : `gDbeta`: `(N,2,1)`-shaped Numpy array of real numbers. `N` is the same as in 9)-11). This array is intended to store the evaluation of the gradient of *beta* at the gaussian points of the 1D quadrature rule imported from `QuadratureRules`.

Observe that the class data members 9)-12) are only used in the numerical approximation of integrals of the form

$$\int_{\partial K \cap \{x=1\}} \partial_x \varphi_i \partial_x \varphi_j,$$

which are integrals on 1D segments, thus we only need the evaluation of the gradients of Master Functions at the gaussian points of a 1D quadrture rule, but not over a the points of a 2D Quadrature rule.

13)-15) : `gD2Phi0`,...,`gD2Phi2`: `(N,3,3)`-shaped arrays, where `N` is the number of gaussian points in the 2D quadrature rule imported from `QuadratureRules`. These arrays are intended to store the second derivatives of the Master Functions $Phi0$, $Phi1$, $Phi2$. For example, for fixed `i`,`k`, `gD2Phi0[i,:,k]` is a `(3,)`-shaped Numpy array containing the evaluation of the $(\partial_{xx}, \partial_{yy}, \partial_{xy})$-derivatives of the `k`-th component (`k=0,1,2`) of the Master Function $Phi0$ at the `i`-th gaussian point of the 2D gaussian quadrature rule.

16) : `gD2beta` is `(N,3,1)`-shaped array, with `N` the same as in 13)-15).

Class methods The class also contains method corresponding to the master functions used to initialize the class data members

4. Module rHCTelement

The file `rHCTelement.py` contains the definitions of the class `rHCT_FE`, which implements methods to compute the local interior and boundary contributions to build the global stiffness matrix. See the file `rHCTelement.py` for more explanations.

Each object of class `rHCT_FE` represents an element and contains all the necessary information to compute the local contributions to the global stiffness matrix. To initialize an object of class `rHCT_FE` we need the three points that define the element given in counterclock-wise order and an evaluation of the master functions (see reference [2]) at the gaussian points of a suitable gaussian quadrature rule. The `rHCT\_FE` implementation is based in the algorithm given in reference [2].

Remark 1. *For the nonlinear controllability this module should be modified to introduce the possibility of solving controllabillity problems associated to wave equations with potentials. New arguments should be added to the contructor of the class and to the method* `InteriorStifness()` *to be able to update the potential on each Newton or fixed point iteration.*

4.1. **Class:** `rHCT_FE`

Class constructor arguments:

- `points`: a `(3,2)`-shaped Numpy array of real numbers. Each row represents a vertex of a triangle element and contains the xy-coordinates of the vertex.
- `D`: an object of class `HCTMasterFunctions.MasterFunctions`.

Remark 2. *The best way to initialize the* `MasterFunctions` *object* `D` *used here is by using the mask* `eval_mask=[True,True,True,True]` *as the argument for constructor of the* `MasterFunctions` *class. However, a long explanation comes in: observe that, in order to be able to evaluate the necessary integrals to build the stiffness matrix, we need that the mask* `eval_mask` *used to initialize the object* `D` *satisfies at least* `eval_mask[2]=True` *and* `eval_mask[3]=True`*, but if we consider a wave equation with a potential, we may need to evaluate the Master Functions themselves (not only their derivatives) at the gaussian points of the 2D quadrature rule, thus in this case it will be necessary to use an* `eval_mask` *s.t.* `eval_mask[1]=True`*. In order to evaluate the integrals corresponding to the initial data (left hand side), we will need to evaluate the Master Functions and their first derivatives at the gaussian points of the 1D quadrature rule, thus, it will be necessary to initialize* `D` *using a mask* `\eval_mask` *s.t.* `\eval_mask[0]=True` *and* `\eval_mask[1]=True`*.*

```
## Example: how to create an object of class rHCT_FE
####################################
import HCTMasterFunctions as mf
import rHCTelement as fe
import numpy as np

# We create an evaluation of the master functions and
# its derivatives at gaussian points
# see HCTMasterFunctions.py for more info
D = mf.MasterFunctions([1,1,1,1])

points = np.array([[0,0],[1,0],[0,1]])
Element = fe.rHCT_FE(points, D)
```

Class methods:

The main methods in the class `rHCT\_FE` are:

1) `InteriorStiffness()`: This method computes the local contribution to the global stiffness matrix corresponding to the integral $\int_K (p_t t - p_x x)(q_t t - q_x x)dxdt$, where K is a triangle in the mesh. It returns an (9,9)-shaped Numpy array.
2) `BoundaryStiffness(k)`: This method computes the local contribution to the global stiffness matrix corresponding to the integral $\int_0^T p_x q_x dt$, where T is the final time. It returns an (9,9)-shaped Numpy array. The argument k is the index belonging to 0,1,2 that tells the method on which edge we want to integrate. It returns an (9,9)-shaped Numpy array.
3) `InitPositionMatrix(k)`: This method computes the local contribution to the matrix corresponding to the integrals

$$\int_0^1 p_t q dx,$$

where p is C^1 a function and q is a piecewise-linear and continuous function that interpolates the initial position. It returns a $(9,3)$-shaped Numpy array. The argument $k \in \{0,1,2\}$ that tells the method on which edge we want to integrate. It is used to compute the part of the right hand side of the linear system arising from the variational formulation.

4) `InitVelocityMatrix(k)`: similar to InitPositionMatrix(k) but corresponding to the integral $\int_0^1 pq dx$ instead. Here, q is supposed to interpolate the initial velocity.

```python
## Example: how to compute local contribution to the stiffness
# matrix and the RHS matrix
############################################
import HCTMasterFunctions as mf
import rHCTelement as fe
import numpy as np


D = mf.MasterFunctions([1,1,1,1])
points = np.array([[0,0],[1,0],[0,1])
Element = fe.rHCT_FE(points, D)

localInterior = Element.InteriorStiffness()

# We now set k=0, this means that the local boundary contribution
# to the stiffness matrix will be computed on the 1st edge
# (indices run in {0,1,2}) which is the edge opposite to the
# third point in array 'points', with coordinates (0,0).
# That is, the integration is carried out over the edge defined
# by the points of coordinates (1,0),(0,1).

k = 0
localBoundaryInterior = Element.BoundaryStiffness(k)

# To compute the local contribution to the stifdness matrix that will
# be used to compute the righ hand side, we will integrate over the
# 3rd edge, corresponding to k = 2, which is defined by the points
# of coordinates (0,0), (1,0).
k = 2
localPosition = Element.InitPositionMatrix(2)
```

5. HCTAssembly

This module contains the functions `StiffnessAssembly`, `InterpolationP1`, `PosVelAssembly`:

- `StiffnessAssembly(D,Th)`: this function returns a SCIPY `sparse.csr_matrix` matrix containing the global stiffness matrix associated to the finite element approximation of the positive definite quadratic form Q given by

$$Q(q,p) = \int_{Q_T} (q_{tt} - q_{xx})(p_{tt} - p_{xx})\,dx\,dt + \int_0^T q_x(1,t)p_x(1,t)\,dt.$$

The arguments for `StiffnessAssembly(D,Th)` are:

1) `D`: an `HCTMasterFunctions.MasterFunctions` type object.
2) `Th`: a `mesh.Mesh` type object

Remark 3. *For the non-linear controllability problem, new arguments should be added to the method* `StiffnessAssembly` *so that we can choose a potential in the considered wave operator.*

- `InterpolationP1`: this function returns a `(N+1,)`-shaped Numpy array with the P_1 interpolation of an initial data given by a function whose analytic expression is known.
  The arguments are
  1) `f`: an univariate Numpy-vectorized function defined in the interval $(0,1)$
  2) `N`: the number of spatial subdivisions at time $t=0$.
- `PosVelassembly`: this function computes the matrices corresponding to the integrals $\int_0^1 \partial_y\varphi_i\varphi_j dx$ and $\int_0^1 \varphi_i\varphi_j dx$, used to aproximate the integrals
$$\int_0^1 \partial_y\varphi_i p_0, \quad \int_0^1 \varphi_i p_1 dx$$
  for each φ_i in the space of finite elements considered (the rHCT based space Φ_h in the notation of [1]). This function returns two SCIPY `sparse.matrix_csr` matrices `Lp` and `Lv`, which correspond to $\int_0^1 \partial_y\varphi_i\varphi_j dx$ and $\int_0^1 \varphi_i\varphi_j dx$ respectively.

Remark 4. *The implementation of* `StiffnessAssembly` *and* `PosVelAssembly` *depends on the fact that the mesh is a regular and uniform mesh, so each time that we call a method which returns a local boundary contribution to the global stiffness matrix, we know on which edge of the subtriangle we have to integrate and we pass it as the argument* `k` *to the method* `BoundaryStiffness`*,* `InitPositionMatrix` *or* `InitVelocityMatrix`*. In order to work with unstructured meshes, we should first identify (using the information provided by the mesh generator) the subtriangle whose exterior boundary will support the 1D integration so that we can pass the correct* `k` *argument to* `BoundaryStiffness`*,* `InitPositionMatrix` *or* `InitVelocityMatrix`*.*

6. WaveTimeMarching

The module `WaveTimeMarching` contains some implicit and explicit time marching methods based on space and time finite differences to solve the following problems:

$$\begin{cases} Ly+\lambda f(y)=0, & \text{in } Q_T=(0,L)\times(0,T),\\ y(0,t)=0,\ y(L,t)=v(t), & \text{in } (0,T),\\ y(0)=y_0,\ y_t(0)=y_1, & \text{in } (0,L). \end{cases} \tag{6.1}$$

$$\begin{cases} Ly+\lambda f(y)=0, & \text{in } Q_T,\\ y(0,t)=0,\ y(L,t)=q_x(L,t), & t\in(0,T)\\ L_{\lambda f'(y)}q=by, & \text{in } Q_T,\\ q(0,t)=q(L,t)=0, & t\in(0,T),\\ y(0)=y_0,\ y_t(0)=y_1,\ q(0)=q_0,\ q_t(0)=q_1. \end{cases} \tag{6.2}$$

The functions in this module are:

i) `Implicit(u0, u1, boundaryData, f, L, T, N, K)`: it returns a `(K+1,N+1)` shaped Numpy array containing a numerical approximation of the solution to (6.1) computed using an implicit method. The arguments are:
   1) `u0`: a (`N+1`,)-shaped Numpy array containing a sample of y_0 in (6.1) in the interval $(0,L)$.
   2) `u1`: a (`N+1`,)-shaped Numpy array containing a sample of y_1 in (6.1) in the interval $(0,L)$.
   3) `boundaryData`: a `K+1` Numpy array containing a sample of v in (6.1) in the interval $(0,T)$.
   4) `f`: the nonlinear function in (6.1).
   5) `L`: a real number representing L in (6.1).
   6) `T`: a real number representing T in (6.1).

7) `K`: the number of time steps (an integer number), $\Delta t = T/K$.
8) `N`: the number of space steps (an integer number), $\Delta x = 1/N$.

ii) `Explicit(u0, u1, boundaryData, f, L, T, N, K)`: it returns a `(K+1,N+1)` shaped Numpy array containing a numerical approximation of the solution to (6.1) computed using an explicit method. The arguments are the same as for the function `Explicit`.

The file `WaveTimeMarching_test.py` contains some example of usage, where the boundary and initial conditions used to test the functions `Implicit`, `Explicit`, `ImplicitSystem` and `ExplicitSystem` are taken to check the Example 1 in the reference [1].

The module also contains the function `toGrid` which take as argument the solution of the linear system associated to the variational formulation and puts it into a $(K+1) \times (N+1)$ Numpy array, where K and N are the number of time and spatial steps respectively. This representation of the solution is more convenient for the graphical representation of the solution to the linear control system; but it is also necessary to interface the controllability problem and the direct problem at each Newton iteration.

References

[1] N. Cindea, A. Münch, A mixed formulation for the direct approximation of the control of minimal L2 norm for linear type wave equations.Calcolo

[2] A. Meyer. A simplified calculation of reduced HCT-basis in FE context. Computational Methods in Applied Mathematics.