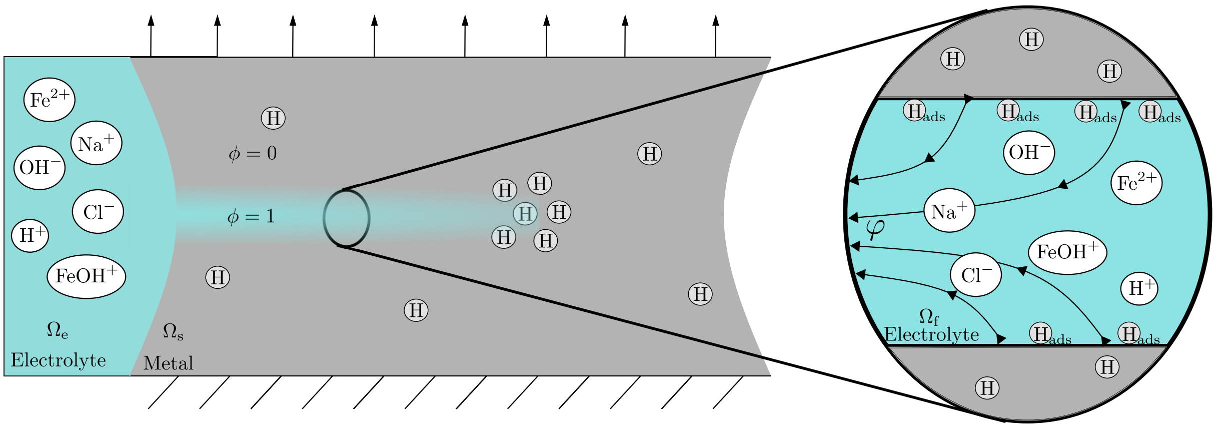

Figure 1: Overview of the system simulated using the presented MATLAB code

Physbased Diffusion Embrittlement: a MATLAB code to accurately predict hydrogen embrittlement within the phase-field framework

Tim Hageman*, Emilio Martínez-Pañeda

Department of Civil and Environmental Engineering, Imperial College London, London SW7 2AZ, UK

Abstract

Documentation that accompanies the *MATLAB* code Physbased Diffusion Embrittlement, available from here. This documentation explains the usage of the implemented finite element framework, and highlight the main files. Special attention is paid to the parts of the code that implement the physics-based diffusion model for electrolytes within cracks, and how the crack opening height is obtained.

If using this module for research or industrial purposes, please cite: T. Hageman & E. Martínez-Pañeda, *A phase field-based framework for electro-chemo-mechanical fracture: crack-contained electrolytes, chemical reactions and stabilisation.* Computer Methods in Applied Mechanics and Engineering XX (2023) YY [1].

update citeMe with correct reference once main paper is published

Keywords: MATLAB, electrochemistry, phasefield, hydrogen embrittlement, finite element method

Todo list

update citeMe with correct reference once main paper is published 1
Intro 2
HydrogenDiffusion 7
PhasefieldDamage 7
PhasefieldElectrolyte 7

*Corresponding author
Email address: `t.hageman@imperial.ac.uk` (Tim Hageman)

Opening heights 10
Section 10

Contents

1 Introduction **2**
1.1 Basic usage 2

2 Summary of included files **3**
2.1 main.m 3
2.2 Models 3
2.2.1 BaseModel 4
2.2.2 Constrainer 4
2.2.3 LinearElastic 4
2.2.4 HydrogenDiffusionUnDamaged 5
2.2.5 Electrolyte 5
2.2.6 ElectrolyteInterface 6
2.2.7 HydrogenDiffusion 7
2.2.8 PhasefieldDamage 7
2.2.9 PhasefieldElectrolyte 7
2.3 Mesh 7
2.4 Shapes 8
2.5 Physics 8
2.6 Dofspace 9
2.7 Solver 9

3 Specifics: Fracture opening height **10**

4 Post-Processing and sample results **10**

1. Introduction

Intro

Note: as this code is developed along other codes, parts of this code (and the documentation) correspond to the codes accompanying [2] (hydrogen absorption stabilised with lumped integration) and [3] (corrosion under charge conservation conditions).

1.1. Basic usage

For simulating the model as provided, running the function “main_ Static.m” performs all required actions for simulating a static fracture: It automatically generates the geometry and mesh, initialises all simulation components, and prints outputs to the screen and saves them to a folder within results. Simple changes, e.g. editing parameters, can be done within main.m without requiring altering other files. Similarly, “main.m” performs the simulations for a dynamic fracture. Files are also provided for the post-processing: “Animations.m” to generate time-series for the output files, and “compare.m” to plot multiple simulations at a set time.

2. Summary of included files

The code is set up in a object-oriented manner, defining matlab classes for each sub-component and providing their accompanying methods. As a result, a clear distinction is made between different components, and each can be used and altered with limited/no impact on other components. Here, the different classes are described. The commenting style employed within the code is compatible with the matlab help function, as such information about all usable methods within a class can be accessed by including the relevant folders, and typing, for instance, “help Solver” to print all variables contained within and all function available from the solver.

2.1. main.m

This is the main file, from which all classes are constructed and the actual simulation is performed. Within it, all properties used within other classes are defined as inputs, for instance for the momentum balance and phase-field evolution equations within the solid domain:

main.m

```
        %Momentum balance and phase-field evolution
        physics_in{1}.type = "PhaseFieldDamage";
        physics_in{1}.Egroup = "Internal";
        physics_in{1}.young = 200e9;    %Youngs Modulus [Pa]
        physics_in{1}.poisson = 0.3;    %Poisson ratio [-]
        physics_in{1}.kmin = 1e-10;     %residual stiffness factor [-]
        physics_in{1}.l=1;              %Phase-field length scale [m]
        physics_in{1}.Gc=2.0e3;         %Fracture release energy [J/m^2]
        physics_in{1}.GDegrade = 0.9;   %Maximum hydrogen degradation factor
        physics_in{1}.NL = 1e6;         %Concentration of interstitial lattice sites [mol/m
            ^3]
        physics_in{1}.gb = 30e3;        %Grain boundary binding energy [J/mol]
```

where “physics_in” is the array of options (in this case, physical models) passed to the physics object at construction.

The actual time-dependent simulations are also performed within this file:

main.m

```
    for tstep = startstep:n_max
        disp("Step: "+string(tstep));
        disp("Time: "+string(physics.time));
        physics.dt = dt*1.05^(min(100,tstep-1));
        disp("dTime: "+string(physics.dt));

        %solve for current time increment
        solver.Solve();
```

Notably, while this performs the time-stepping scheme and controls the time increment size and termination of the simulations, it does not by itself solve anything, instead calling the “solver.Solve()” function which performs a Newton-Raphson procedure using the parameters used to initialize the class, and once the current timestep is converged returns to the main code.

2.2. Models

The files included within the Models folder form the main implementation of all the physical phenomena involved. They implement the assembly of tangential matrices and force vectors, when requested by the solving procedures, and store model-specific parameters.

2.2.1. BaseModel

This is an empty model, inherited by all other models to provide consistency within the available functions. While empty within here, the potential functions that can be defined within other models include assembling the system matrix and force vector:

Models/@BaseModel/BaseModel.m

```
        function getKf(obj, physics, stp)
```

and committing history dependent or path dependent variables:

```
        function Commit(obj, physics, commit_type)
```

where the keyword "commit_type" indicates the type of history or path dependence to commit at the current point. It also provides a function for performing once per staggered solution step actions:

```
        function OncePerStep(obj, physics, stp)
```

2.2.2. Constrainer

This model is used to apply fixed boundary constraints to a degree of freedom at a set location. Within the main file, the inputs required are:

main.m

```
        %displacement constrain at bottom boundary
        physics_in{3}.type = "Constrainer";
        physics_in{3}.Ngroup = "Bottom";
        physics_in{3}.dofs = {"dy"};
        physics_in{3}.conVal = [0];
```

and multiple definitions of this model are allowed, allowing for constraints to be applied to several element groups. These constraints are integrated within the tangential matrix and force vector through allocation matrices $\boldsymbol{C}_{con}$ and $\boldsymbol{C}_{uncon}$, reordering the system into a constrained and unconstrained part. This allows the constrained system to be solved as:

$$\boldsymbol{C}_{uncon}^T \boldsymbol{K} \boldsymbol{C}_{uncon} \mathbf{y} = -\left(\boldsymbol{C}_{uncon}^T \boldsymbol{f} + \boldsymbol{C}_{uncon}^T \boldsymbol{K} \boldsymbol{C}_{con} \mathbf{c}\right) \tag{1}$$

with the values of the boundary constraints contained in the vector $\mathbf{c}$. After solving, the state vector is then incremented through:

$$\mathbf{x}^{new} = \mathbf{x}^{old} + \boldsymbol{C}_{uncon} \mathbf{y} + \boldsymbol{C}_{con} \mathbf{c} \tag{2}$$

2.2.3. LinearElastic

The linear-elastic model implements the momentum balance for the metal domain:

$$\boldsymbol{\nabla} \cdot \boldsymbol{\sigma} = \mathbf{0} \tag{3}$$

where the stresses $\boldsymbol{\sigma}$ are based on the displacement $\mathbf{u} = [$"dx" "dy"$]$. The properties used to initialize this model given as input by:

main.m

```
        physics_in{1}.type = "LinearElastic";
        physics_in{1}.Egroup = "Metal";
        physics_in{1}.young = 200e9;
        physics_in{1}.poisson = 0.3;
```

Notably, since the tangential matrix for linear-elasticity is constant, it is assembled once and saved locally within the model, after which during the global matrix assembly process, it is copied over to the global matrix:

Models/@LinearElastic/LinearElastic.m

```
                physics.fint{stp} = physics.fint{stp} + obj.myK*physics.StateVec{obj.
                    dx_Step};
                physics.K{stp} = physics.K{stp} + obj.myK;
```

with the force vector also being updated based on this locally saved stiffness matrix.

2.2.4. HydrogenDiffusionUnDamaged

This model implements the hydrogen mass conservation, through the diffusion equation [4, 5, 6, 7, 8]:

$$\left(1+\frac{N_\mathrm{T}/N_\mathrm{L}\exp\left(\Delta g_\mathrm{b}/RT\right)}{\left(C_\mathrm{L}/N_\mathrm{L}+\exp\left(\Delta g_\mathrm{b}/RT\right)\right)^2}\right)\dot{C}_\mathrm{L}-\boldsymbol{\nabla}\cdot\left(\frac{D_\mathrm{L}}{1-C_\mathrm{L}/N_\mathrm{L}}\boldsymbol{\nabla}C_\mathrm{L}\right)+\boldsymbol{\nabla}\cdot\left(\frac{D_\mathrm{L}C_\mathrm{L}\overline{V}_\mathrm{H}}{RT}\boldsymbol{\nabla}\sigma_\mathrm{H}\right)=0 \tag{4}$$

with the interstitial lattice hydrogen concentration C_L indicated within the code by "CL". Input properties for this model constitute:

mainDiscreteFrac.m

```
        physics_in{2}.type = "HydrogenDiffusionUnDamaged";
        physics_in{2}.Egroup = "Metal";
        physics_in{2}.DL = 1e-9;
        physics_in{2}.NL = 1e6;
        physics_in{2}.gb = 30e3;
        physics_in{2}.NT = 1e2;
```

This model presumes the first model provides the Young's modulus and Poisson ratio. The hydrostatic stress gradients required in this model are obtained as $\boldsymbol{\nabla}\sigma_h=\boldsymbol{B}^*_\mathrm{u}\mathbf{u}$, with the $\boldsymbol{B}^*$ matrix defined as:

$$\boldsymbol{B}^*_u=E/(3(1-2\nu))\begin{bmatrix}\frac{\partial^2N_{u1}}{\partial x^2} & \frac{\partial^2N_{u2}}{\partial x^2} & \dots & \frac{\partial^2N_{u1}}{\partial x\partial y} & \frac{\partial^2N_{u2}}{\partial x\partial y} & \dots \\ \frac{\partial^2N_{u1}}{\partial x\partial y} & \frac{\partial^2N_{u2}}{\partial x\partial y} & \dots & \frac{\partial^2N_{u1}}{\partial y^2} & \frac{\partial^2N_{u2}}{\partial y^2} & \dots\end{bmatrix} \tag{5}$$

2.2.5. Electrolyte

The electrolyte model implements the Nernst-Planck mass balance for non-crack contained electrolytes [**?**]:

$$\dot{C}_\pi+\boldsymbol{\nabla}\cdot\left(-D_\pi\boldsymbol{\nabla}C_\pi\right)+\frac{z_\pi F}{RT}\boldsymbol{\nabla}\cdot\left(-D_\pi C_\pi\boldsymbol{\nabla}\varphi\right)+R_\pi=0 \tag{6}$$

for the ionic species and their name within the model file: H^+ ("H"), OH^- ("OH"), Na^+ ("Na"), Cl^- ("Cl", using lower case l; upper case L provides the lattice hydrogen concentration), Fe^{2+} ("Fe"), and FeOH^+ ("FeOH"). Additionally, it implements the electro-neutrality condition [9, 10]:

$$\sum z_\pi C_\pi=0 \tag{7}$$

and bulk reactions:

$$\mathrm{H_2O}\underset{k'_w}{\overset{k_w}{\rightleftharpoons}}\mathrm{H}^++\mathrm{OH}^- \tag{8}$$

$$\mathrm{Fe}^{2+}+\mathrm{H_2O}\underset{k'_{fe}}{\overset{k_{fe}}{\rightleftharpoons}}\mathrm{FeOH}^++\mathrm{H}^+ \tag{9}$$

$$\mathrm{FeOH}^++\mathrm{H_2O}\xrightarrow{k_{feoh}}\mathrm{Fe(OH)_2}+\mathrm{H}^+ \tag{10}$$

with reaction rates:

$$R_{\mathrm{H}^+,w}=R_{\mathrm{OH}^-}=k_wC_{\mathrm{H_2O}}-k'_wC_{\mathrm{H}^+}C_{\mathrm{OH}^-}=k_{eq}\left(K_w-C_{\mathrm{H}^+}C_{\mathrm{OH}^-}\right) \tag{11}$$

$$R_{\mathrm{Fe}^{2+}}=-k_{fe}C_{\mathrm{Fe}^{2+}}+k'_{fe}C_{\mathrm{FeOH}^+}C_{\mathrm{H}^+} \tag{12}$$

$$R_{\mathrm{FeOH}^+}=k_{fe}C_{Fe^{2+}}-C_{\mathrm{FeOH}^+}(k_{feoh}+k'_{fe}C_{\mathrm{H}^+}) \tag{13}$$

$$R_{\mathrm{H}^+,fe}=k_{fe}C_{\mathrm{Fe}^{2+}}-C_{\mathrm{FeOH}^+}(k'_{fe}C_{\mathrm{H}^+}-k_{feoh}) \tag{14}$$

For this model, the input properties required are:

mainDiscreteFrac.m

```
        physics_in{7}.type = "Electrolyte";
        physics_in{7}.Egroup = "Electrolyte";
        physics_in{7}.D = [9.3; 5.3; 1.3; 2; 1.4; 1]*1e-9;  %H OH Na CL Fe FeOH
        physics_in{7}.z = [1; -1; 1; -1; 2; 1];
        physics_in{7}.pH0 = 5;
        physics_in{7}.NaCl = 0.6e3;
        physics_in{7}.Lumped = [true; true]; %water, metal
        physics_in{7}.k = [1e6; 1e-1; 1e-3; 1e-3]; %water, Fe, Fe', FeOH
```

This model employs a lumped integration scheme when the vector "Lumped" contains true. Details for the implementation of this lumped scheme are given in [2].

2.2.6. ElectrolyteInterface

The electrolyteInterface model implements the metal-electrolyte coupling through the surface reactions for explicitly represented electrolytes [11, 8]:

$$\text{Volmer (acid):} \qquad H^+ + M + e^- \underset{k'_{Va}}{\overset{k_{Va}}{\rightleftharpoons}} MH_{ads} \tag{15}$$

$$\text{Heyrovsky (acid):} \qquad H^+ + e^- + MH_{ads} \underset{k'_{Ha}}{\overset{k_{Ha}}{\rightleftharpoons}} M + H_2 \tag{16}$$

$$\text{Volmer (base):} \qquad H_2O + M + e^- \underset{k'_{Vb}}{\overset{k_{Vb}}{\rightleftharpoons}} MH_{ads} + OH^- \tag{17}$$

$$\text{Heyrovsky (base):} \qquad H_2O + e^- + MH_{ads} \underset{k'_{Hb}}{\overset{k_{Hb}}{\rightleftharpoons}} M + H_2 + OH^- \tag{18}$$

$$\text{Tafel:} \qquad 2MH_{ads} \underset{k'_{T}}{\overset{k_{T}}{\rightleftharpoons}} 2M + H_2 \tag{19}$$

$$\text{Absorption:} \qquad MH_{ads} \underset{k'_{A}}{\overset{k_{A}}{\rightleftharpoons}} MH_{abs} \tag{20}$$

$$\text{Corrosion:} \qquad Fe^{2+} + 2e^- \underset{k'_{c}}{\overset{k_{c}}{\rightleftharpoons}} Fe \tag{21}$$

with reaction rates:

$$\begin{array}{rll} & \text{Forward} & \text{Backward} \\ \text{Volmer(acid)}: & \nu_{Va} = k_{Va}C_{H^+}(1-\theta_{ads})e^{-\alpha_{Va}\frac{\eta F}{RT}} & \nu'_{Va} = k'_{Va}\theta_{ads}e^{(1-\alpha_{Va})\frac{\eta F}{RT}} \qquad (22) \\ \text{Heyrovsky(acid)}: & \nu_{Ha} = k_{Ha}C_{H^+}\theta_{ads}e^{-\alpha_{Ha}\frac{\eta F}{RT}} & \nu'_{Ha} = k'_{Ha}(1-\theta_{ads})p_{H_2}e^{(1-\alpha_{Ha})\frac{\eta F}{RT}} \qquad (23) \\ \text{Volmer(base)}: & \nu_{Vb} = k_{Vb}(1-\theta_{ads})e^{-\alpha_{Vb}\frac{\eta F}{RT}} & \nu'_{Vb} = k'_{Vb}C_{OH^-}\theta_{ads}e^{(1-\alpha_{Vb})\frac{\eta F}{RT}} \qquad (24) \\ \text{Heyrovsky(base)}: & \nu_{Hb} = k_{Hb}\theta_{ads}e^{-\alpha_{Hb}\frac{\eta F}{RT}} & \nu'_{Hb} = k'_{Hb}(1-\theta_{ads})p_{H_2}C_{OH^-}e^{(1-\alpha_{Hb})\frac{\eta F}{RT}} \qquad (25) \\ \text{Tafel}: & \nu_T = k_T\,|\theta_{ads}|\,\theta_{ads} & \nu'_T = k'_T(1-\theta_{ads})p_{H_2} \qquad (26) \\ \text{Absorption}: & \nu_A = k_A(N_L - C_L)\theta_{ads} & \nu'_A = k'_A C_L(1-\theta_{ads}) \qquad (27) \\ \text{Corrosion}: & \nu_c = k_c C_{Fe^{2+}}e^{-\alpha_c\frac{\eta F}{RT}} & \nu'_c = k'_c e^{(1-\alpha_c)\frac{\eta F}{RT}} \qquad (28) \end{array}$$

These reaction rates are implemented in a separate function from the matrix assembly:

Models/@ElectrolyteInterface/ElectrolyteInterface.m

```
        function [react, dreact, products] = reactions(obj, CH, COH, CFE, theta, phil, CLat
            )
```

which takes the local hydrogen, hydroxide, and iron concentrations, the surface coverage, electrolyte potential, and interstitial lattice hydrogen concentration. It functions for both the integration-point variables as well as for the nodal values. In addition to the reaction rates, the electrolyte interface model also resolves the surface mass balance:

$$N_{ads}\dot{\theta}_{ads} - (\nu_{Va} - \nu'_{Va}) + \nu_{Ha} + 2\nu_T + (\nu_A - \nu'_A) - (\nu_{Vb} - \nu'_{Vb}) + \nu_{Hb} = 0 \tag{29}$$

For this model, the input variables to define are given as:

mainDiscreteFrac.m

```
        physics_in{8}.type = "ElectrolyteInterface";
        physics_in{8}.Egroup = "Interface";
        physics_in{8}.NAds = 1e-3;
        physics_in{8}.k = k;
        physics_in{8}.NL = 1e6;
        physics_in{8}.Em = Em;
        physics_in{8}.Lumped = [1 1 1 1 1 1 1];
```

```
    k = [1e-4,  1e-10,  0.5,    0;
        1e-10,  0,      0.3,    0;
        1e-6,   0,      0,      0;
        1e1,    7e5,    0,      0;
        1e-8,   1e-13,  0.5,    0;
        1e-10,  1e-14,  0.3,    0;
        3e-5/(2*96485.3329),3e-5/(2*96485.3329), 0.5, -0.4];
```

with the vector "Lumped" allowing for individual interface reactions to be either integrated using a standard Gauss integration scheme (0) or a lumped integration scheme (1). the reaction constants matrix k is defined as:

$$k = \begin{bmatrix} k_{Va} & k'_{Va} & \alpha_{Va} & E_{eq,Va} \\ k_{Ha} & k'_{Ha} & \alpha_{Ha} & E_{eq,Ha} \\ k_T & k'_T & - & - \\ k_A & k'_A & - & - \\ k_{Vb} & k'_{Vb} & \alpha_{Vb} & E_{eq,Vb} \\ k_{Hb} & k'_{Hb} & \alpha_{Hb} & E_{eq,Hb} \\ k_c & k'_c & \alpha_c & E_{eq,c} \end{bmatrix} \tag{30}$$

with the empty entries not used within the model.

2.2.7. HydrogenDiffusion

HydrogenDiffusion

2.2.8. PhasefieldDamage

PhasefieldDamage

2.2.9. PhasefieldElectrolyte

PhasefieldElectrolyte

2.3. Mesh

This class contains the nodes and elements that describe the geometry, and provides support for evaluating shape functions. Within its implementation, it uses a multi-mesh approach, defining element groups for each entity within the domain (for instance, defining an element group "Interior" for the metal domain composed of surface elements, and defining an element group "Interface" composed of line elements which coincide with the electrolyte-metal interface). The geometry of the problem is defined through procedures within the mesh class.

The mesh class also provides a direct interfaces from which to get the element shape functions, second gradients, and surface-normal vectors, providing an element group number and the index of the element itself:

@Mesh/mesh.m

```
        [N, G, w] = getVals(obj, group, elem);
        G2 = getG2(obj, group, elem);
        [n, t] = getNormals(obj, group, elem);
```

which returns a matrix containing the shape functions N within all integration points of the element, gradients of the shape function G, and the integration weights for all integration points w. Additionally, for the construction of the hydrogen diffusion model, the second-order gradients G2 are provided through a separate function.

2.4. Shapes

The classes within this folder provide basic shape functions, and are used by the mesh to provide shape functions and integration weights. The included shape functions are square Lagrangian and triangular Bernstein surface elements (Q9 and T6), quadratic Lagrangian and Bernstein line elements (L3 and L3B), and interface elements (LI6, unused).

2.5. Physics

This class provides all the support required for constructing and managing state and force vectors, tangential matrices, and boundary constraints. Most notably, during its initialization it generates an array of all the physical models, from which it then is able to construct the tangential matrix when required:

@Physics/Physics.m

```
        function Assemble(obj, stp)
            %Assemble stiffness matrix and internal force vector for the
            %current step

            dofcount = obj.dofSpace.NDofs(stp);

            obj.condofs{stp} = [];
            obj.convals{stp} = [];

            if isempty(obj.K{stp})

            else
                obj.nonz(stp) = round(nnz(obj.K{stp}));
            end
            obj.K{stp} = spalloc(dofcount, dofcount, obj.nonz(stp));
            obj.fint{stp} = zeros(dofcount, 1);

            disp("    Assembling:")
            for m=1:length(obj.models)
                obj.models{m}.getKf(obj, stp);
            end
```

This calls each of the models, and passes a handle to the physics object itself through which the individual models can add their contributions. Notably, the force vector and tangent matrix being assembled by this function are those associated with the staggered solver step “stp”, with this step index being passed onto the individual models, which then either add their contributions to the tangent matrix or do not undertake nay action. In a similar manner, actions which should be performed once at the start of each staggered solution step are performed via:

@Physics/Physics.m

```
        function OncePerStep(obj, stp)
            %procedures that should be performed once per step

            for m=1:length(obj.models)
                obj.models{m}.OncePerStep(obj, stp);
            end
        end
```

The physics class also provides the ability for post-processing the results through the function;

@Physics/Physics.m

```
        PlotNodal(obj, dofName, dispscale, plotloc) %exterior defined, plots nodal
            quantities
        PlotIP(obj, varName, plotloc)               %exterior defined, plots integration
            point quantities
```

This function requires the name of a degree of freedom (for instance "dx" for the horizontal displacements, or "H" for the hydrogen ion concentration), a scale to indicate whether the mesh is plotted in deformed (scale>0) or undeformed (scale=0) configuration, and the name of an element group on which to plot the results. Similarly, the "PlotIP" function plots integration-point specific variables, for instance stresses or history fields.

2.6. Dofspace

This class converts the node numbering, degree of freedom type, and solution step to an index for the degree of freedom, corresponding to its location within the unconstrained state vector and tangential matrix. Specific types of degree of freedom are registered on initialization of the dofspace through:

@DofSpace/DofSpace.m

```
        function obj = DofSpace(mesh, dofs_in)
            %DOFSPACE Construct an instance of this class

            obj.NSteps = max(dofs_in.Step);

            obj.mesh = mesh;
            obj.DofTypes = dofs_in.dofs;
            obj.DofSteps = dofs_in.Step;
            obj.NDofs = zeros(obj.NSteps,1);
            obj.DofNumbering = sparse(length(obj.mesh.Nodes), length(obj.DofTypes));
        end
```

For the numbering scheme saved within "DofNumbering", each solution step retains its own numbering.

After initializing the dofspace, degrees of freedom can be added to nodes through:

```
        function addDofs(obj, dofIndices, nodeIndex)
            % Adds degrees of freedom for type "dofIndices" to the nodes
            % "nodeIndex"
```

These functions automatically check for duplicates, such that each model can safely add all the degrees of freedom relevant to itself, without taking into account potential interactions with other models. During the finite element assembly, the managed degrees of freedom indices are requestable by providing the degree of freedom type index and the node number:

```
        function DofIndices = getDofIndices(obj, dofType, NodeIndices)
           % gets the indices for a combination of degree of freedom
           % "doftype" and nodes "NodeIndices"
```

which assumes the model already knows which staggered solver step the degree of freedom is associated with. These solver step indices can also be requested through:

```
        function [DofTypeIndex, DofStepIndex] = getDofType(obj, dofnames)
            % returns the dof type index for pre-existing degrees of
            % freedom
```

which takes the name of the degree of freedom, and returns its index and the staggered step (and thus state vector) associated with this degree of freedom.

2.7. Solver

The solver class implements a iteratively staggered Newton-Raphson type nonlinear solver, including the ability to perform linear line-searches to improve the convergence rate and stability. During its creation, it gets linked to the physics object, such that it can automatically request updated tangential matrices. An example of the staggered solution scheme solved through this solver is shown in Algorith 1.

Algorithm 1 Overview of solution method

```
1: Start of time increment
2: while not converged do
3:     Step 1: Update $\boldsymbol{\phi}$
4:     Step 2: Update $\mathbf{u}$
5:     Perform OncePerStep 3: Update $h$
6:     while $\mathbf{C}_\mathrm{L}$, $\mathbf{C}_\pi$, $\boldsymbol{\varphi}$, $\boldsymbol{\theta}$ are not converged do
7:         Step 3: update $\mathbf{C}_\mathrm{L}$, $\mathbf{C}_\pi$, $\boldsymbol{\varphi}$, $\boldsymbol{\theta}$
8:         Calculate energy based residual for $\mathbf{C}_\mathrm{L}$, $\mathbf{C}_\pi$, $\boldsymbol{\varphi}$, $\boldsymbol{\theta}$
9:     end while
10:    Calculate energy based residual for $\mathbf{u}$ and $\boldsymbol{\phi}$
11: end while
12: Go to next time increment
```

3. Specifics: Fracture opening height

Opening heights

4. Post-Processing and sample results

Section

References

[1] T. Hageman, E. Martínez-Pañeda, Stabilising Effects of Lumped Integration Schemes for the Simulation of Metal-Electrolyte Reactions, Journal of The Electrochemical Society 170 (2) (2023) 021511.

[2] T. Hageman, E. Martínez-Pañeda, Stabilising Effects of Lumped Integration Schemes for the Simulation of Metal-Electrolyte Reactions, Journal of The Electrochemical Society 170 (2) (2023) 021511.
URL `https://iopscience.iop.org/article/10.1149/1945-7111/acb971https://iopscience.iop.org/article/10.1149/1945-7111/acb971/meta`

[3] T. Hageman, C. Andrade, E. Martínez-Pañeda, Corrosion rates under charge-conservation conditions, Electrochimica Acta 461 (2023) 142624.
URL `https://linkinghub.elsevier.com/retrieve/pii/S0013468623008022`

[4] R. A. Oriani, P. H. Josephic, Equilibrium aspects of hydrogen-induced cracking of steels, Acta Metallurgica 22 (9) (1974) 1065–1074.

[5] E. Martínez-Pañeda, A. Díaz, L. Wright, A. Turnbull, Generalised boundary conditions for hydrogen transport at crack tips, Corrosion Science 173 (2020) 108698.

[6] P. K. Kristensen, C. F. Niordson, E. Martínez-Pañeda, A phase field model for elastic-gradient-plastic solids undergoing hydrogen embrittlement, Journal of the Mechanics and Physics of Solids 143 (2020) 104093.

[7] A. Golahmar, P. K. Kristensen, C. F. Niordson, E. Martínez-Pañeda, A phase field model for hydrogen-assisted fatigue, International Journal of Fatigue 154 (July 2021) (2022) 106521.
URL `https://doi.org/10.1016/j.ijfatigue.2021.106521https://linkinghub.elsevier.com/retrieve/pii/S0142112321003765`

[8] T. Hageman, E. Martínez-Pañeda, An electro-chemo-mechanical framework for predicting hydrogen uptake in metals due to aqueous electrolytes, Corrosion Science 208 (2022) 110681.
URL `https://linkinghub.elsevier.com/retrieve/pii/S0010938X22005996`

[9] S. Sarkar, W. Aquino, Electroneutrality and ionic interactions in the modeling of mass transport in dilute electrochemical systems, Electrochimica Acta 56 (24) (2011) 8969–8978.
URL `http://dx.doi.org/10.1016/j.electacta.2011.07.128`

[10] S. W. Feldberg, On the dilemma of the use of the electroneutrality constraint in electrochemical calculations, Electrochemistry Communications 2 (7) (2000) 453–456.

[11] Q. Liu, A. D. Atrens, Z. Shi, K. Verbeken, A. Atrens, Determination of the hydrogen fugacity during electrolytic charging of steel, Corrosion Science 87 (2014) 239–258.