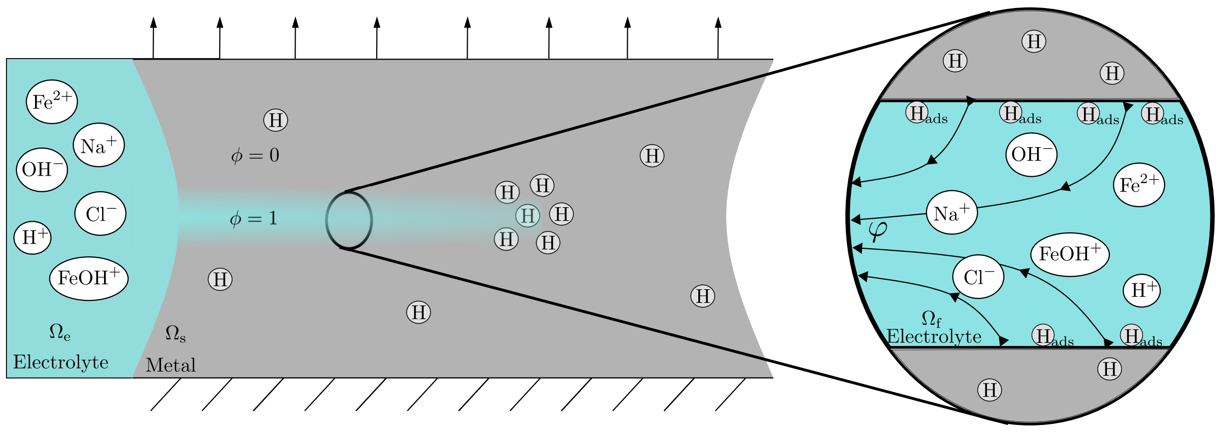

Figure 1: Overview of the system simulated using the presented MATLAB code

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

Contents

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

1. Introduction

Development of cracks due to interactions with the environment is a common failure mechanism, with causes of this environment-induced failure including stress-corrosion-cracking and hydrogen embrittlement. Accurately predicting these phenomena does not only require capturing the metal and neighbouring electrolyte, but also requires the effects of newly created and propagating cracks to be captured. One framework enabling crack propagation in arbitrary directions and geometries is the phase field method, representing cracks in a smeared manner. While the mechanical side of phase field fracture models is well-developed, models to represent fluids and species contained within the cracks are usually of a more empirical nature (for instance, requiring fitting parameters that are problem-dependent). Here, we present a MATLAB implementation of a physics-based model for ionic diffusion and reactions within cracks compatible with the phase field framework. This physics-based model is described in 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], where its ability to accurately predict hydrogen uptake and expected failure times is shown. In the remainder of this documentation, the physical models and mathematical objects included within this code will be detailed. Special attention is given to the manner in which crack opening heights are reconstructed, as this requires procedures that are non-standard in finite element methods.

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=l;              %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

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

and committing history dependent or path dependent variables:

```matlab
        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:

```matlab
        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

```matlab
        %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 $\mathbf{u} = [\text{"dx" "dy"}]$. The properties used to initialize this model given as input by:

main.m

```matlab
        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

```matlab
                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:

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

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:

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

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

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

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

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]:

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

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

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

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

$$\text{Tafel:} \qquad 2MH_{ads} \xrightleftharpoons[k'_{T}]{k_{T}} 2M + H_2 \tag{19}$$

$$\text{Absorption:} \qquad MH_{ads} \xrightleftharpoons[k'_{A}]{k_{A}} MH_{abs} \tag{20}$$

$$\text{Corrosion:} \qquad Fe^{2+} + 2e^- \xrightleftharpoons[k'_{c}]{k_{c}} Fe \tag{21}$$

with reaction rates:

$$\begin{array}{rll} & \text{Forward} & \text{Backward} \end{array}$$

$$\text{Volmer(acid)}: \quad \nu_{Va} = k_{Va} C_{H^+}(1-\theta_{ads}) e^{-\alpha_{Va}\frac{\eta F}{RT}} \qquad \nu'_{Va} = k'_{Va}\theta_{ads} e^{(1-\alpha_{Va})\frac{\eta F}{RT}} \tag{22}$$

$$\text{Heyrovsky(acid)}: \quad \nu_{Ha} = k_{Ha} C_{H^+}\theta_{ads} e^{-\alpha_{Ha}\frac{\eta F}{RT}} \qquad \nu'_{Ha} = k'_{Ha}(1-\theta_{ads}) p_{H_2} e^{(1-\alpha_{Ha})\frac{\eta F}{RT}} \tag{23}$$

$$\text{Volmer(base)}: \quad \nu_{Vb} = k_{Vb}(1-\theta_{ads}) e^{-\alpha_{Vb}\frac{\eta F}{RT}} \qquad \nu'_{Vb} = k'_{Vb} C_{OH^-}\theta_{ads} e^{(1-\alpha_{Vb})\frac{\eta F}{RT}} \tag{24}$$

$$\text{Heyrovsky(base)}: \quad \nu_{Hb} = k_{Hb}\theta_{ads} e^{-\alpha_{Hb}\frac{\eta F}{RT}} \qquad \nu'_{Hb} = k'_{Hb}(1-\theta_{ads}) p_{H_2} C_{OH^-} e^{(1-\alpha_{Hb})\frac{\eta F}{RT}} \tag{25}$$

$$\text{Tafel}: \quad \nu_T = k_T\,|\theta_{ads}|\,\theta_{ads} \qquad \nu'_T = k'_T(1-\theta_{ads}) p_{H_2} \tag{26}$$

$$\text{Absorption}: \quad \nu_A = k_A(N_L - C_L)\theta_{ads} \qquad \nu'_A = k'_A C_L(1-\theta_{ads}) \tag{27}$$

$$\text{Corrosion}: \quad \nu_c = k_c C_{Fe^{2+}} e^{-\alpha_c\frac{\eta F}{RT}} \qquad \nu'_c = k'_c e^{(1-\alpha_c)\frac{\eta F}{RT}} \tag{28}$$

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. PhasefieldDamage

This model implements the momentum conservation for a damaged linear-elastic material:

$$0 = \boldsymbol{\nabla} \cdot \left(d(\phi) \frac{\partial \psi_0}{\partial \boldsymbol{\varepsilon}} \right) \tag{31}$$

and the phase-field evolution equation [12, 13, 14, 15]:

$$\frac{\phi}{\ell} - \ell \boldsymbol{\nabla}^2 \phi = -\frac{\partial d(\phi)}{\partial \phi} \mathcal{H} \qquad \text{where} \qquad \mathcal{H} = \frac{\psi_0}{G_c(C_L)}, \;\; \dot{\mathcal{H}} >= 0 \tag{32}$$

using the linear-elastic energy function for an undamaged material:

$$\psi_0 = \varepsilon : \mathcal{D} : \varepsilon \tag{33}$$

and the damage function:

$$d(\phi) = k_0 + (1 - k_0)(1 - \phi)^2 \tag{34}$$

Within this model, the displacement and phase-field damage variable can not be updated at the same time, instead requiring Eqs. (31) and (32) to be set to different staggered solver steps. The history field required within this model is calculated each time the tangent matrix is assembled:

Models/@PhaseFieldDamage/PhaseFieldDamage.m

```
                                if (H>=Energy_el)
                                    f_el = f_el + d_dam_fun*w(ip)*N(ip,:)'*H;

                                    K_el = K_el + d_dam_dphi*w(ip)*H*N(ip,:)'*N(ip,:);

                                    HNew(n_el, ip) = obj.HistOld(n_el, ip);
                                else
                                    f_el = f_el + d_dam_fun*w(ip)*N(ip,:)'*Energy_el;

                                    K_el = K_el + w(ip)*d_dam_dphi*Energy_el*N(ip,:)'*N(ip,:);

                                    HNew(n_el, ip) = Energy_el;
                                end
```

and changes are committed to the new time increments once the phsics object calls its “Commit” function.
For the initialization of the phase-field parameter, an estimated history field is set via:

$$\phi^{\text{init}} = \exp\left(-\left|\mathrm{d}x\right|/\ell\right) \qquad \mathcal{H}^{\text{init}} = \frac{1/\ell \;\; \mathbf{N}_\phi \boldsymbol{\phi}^{\text{init}} + \ell \left(\boldsymbol{\nabla}\mathbf{N}_\phi \boldsymbol{\phi}^{\text{init}}\right)^T \left(\boldsymbol{\nabla}\mathbf{N}_\phi \boldsymbol{\phi}^{\text{init}}\right)}{k_0 - 2(1-k_0)\left(1 - \mathbf{N}_\phi \boldsymbol{\phi}^{\text{init}}\right)} \tag{35}$$

the first time the “OncePerStep” function is called:

Models/@PhaseFieldDamage/PhaseFieldDamage.m

```
        function OncePerStep(obj, physics, stp)
            if (stp == obj.phi_step && obj.doInit)
                Hdom = max(obj.mesh.Nodes(:,2));
                Lfrac = 5e-3;

                %% set Values
                nodecons = [];

                allNodes = obj.mesh.GetAllNodesForGroup(obj.myGroupIndex);
                PhiDofs = obj.dofSpace.getDofIndices(obj.dofTypeIndices(3), allNodes);
                initvals = 0*allNodes;
                for i=1:length(allNodes)
                    xy = [obj.mesh.Nodes(allNodes(i),1), obj.mesh.Nodes(allNodes(i),2)];

                    if (xy(1)<Lfrac)
                        dst = Hdom/2-xy(2);
                    else
                        dst=sqrt((xy(2)-Hdom/2)^2+(xy(1)-Lfrac)^2);
                    end

                    pf = exp(-abs(dst)/(obj.l));

                    initvals(i) = pf;
                end

                physics.StateVec{obj.phi_step}(PhiDofs) = initvals;
                physics.StateVec_Old{obj.phi_step}(PhiDofs) = initvals;

                %% set history field
                for n_el=1:size(obj.mesh.Elementgroups{obj.myGroupIndex}.Elems, 1)
                    Elem_Nodes = obj.mesh.getNodes(obj.myGroupIndex, n_el);
                    [N, G, w] = obj.mesh.getVals(obj.myGroupIndex, n_el);
                    G2 = obj.mesh.getG2(obj.myGroupIndex, n_el);

                    dofsPhi = obj.dofSpace.getDofIndices(obj.dofTypeIndices(3), Elem_Nodes)
                        ;
                    PHI = physics.StateVec{obj.phi_step}(dofsPhi);

                    for ip=1:length(w)
```

```
                        NPhi = N(ip,:)*PHI;
                        GPhi = squeeze(G(ip,:,:))'*PHI;

                        d_dam_fun = -2*(1-obj.kmin)*(1-NPhi);
                        H = abs((NPhi/(obj.l)+obj.l*(GPhi'*GPhi))/(d_dam_fun+obj.kmin));

                        obj.Hist(n_el, ip) = H;
                        obj.HistOld(n_el, ip) = H;
                    end

                end

                obj.doInit = false;
            end
```

As can be seen, this initial field is set based on the fracture length "Lfrac", and inserted in the centre of the domain.

Input parameters that need to be set for this model are:

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]
```

2.2.8. HydrogenDiffusion

Similar to the model presented in Section 2.2.4 for hydrogen diffusion in undamaged materials, this model simulates the diffusion in damaged materials. While the governing equations displayed in Eq. (4) are still being simulated, the hydrostatic stress gradient is now interacts with the phase-field damage as:

$$\nabla\sigma_h = \frac{E}{3(1-2\nu)}\left(d(\phi)\boldsymbol{B}^*\mathbf{u} + \frac{\partial d(\phi)}{\partial\phi}\nabla\phi\overline{\boldsymbol{B}}\mathbf{u}\right) \tag{36}$$

where

$$\overline{\boldsymbol{B}} = \begin{bmatrix} \frac{\partial N_{u1}}{\partial x} & \frac{\partial N_{u1}}{\partial x} & \dots\frac{\partial N_{u1}}{\partial y} & \frac{\partial N_{u1}}{\partial y} & \dots \end{bmatrix} \tag{37}$$

Input parameters required for this model are:

main.m

```
        %Interstitial lattice hydrogen diffusion model
        physics_in{2}.type = "HydrogenDiffusion";
        physics_in{2}.Egroup = "Internal";
        physics_in{2}.DL = 1e-9;                    %Diffusivity [m/s]
        physics_in{2}.NL = physics_in{1}.NL;        %Concentration of interstitial lattice
            sites [mol/m^3]
        physics_in{2}.gb = physics_in{1}.gb;        %Grain boundary binding energy [J/mol]
        physics_in{2}.NT = 1e2;                     %concentration of trapping sites
        physics_in{2}.kmin = physics_in{1}.kmin;%residual stiffness factor [-]
```

2.2.9. *PhasefieldElectrolyte*

This model simulates the Nernst-Planck and electroneutrelity equations, surface adsorbed hydrogen concentration, and bulk and volume reactions for a crack-contained electrolyte. This system is described by:

$$0 = \beta_c \dot{C}_\pi - \boldsymbol{\nabla} \cdot \left(\boldsymbol{R}^T \boldsymbol{\beta}_\mathbf{d} \boldsymbol{R} D_\pi \boldsymbol{\nabla} C_\pi \right) - \frac{z_\pi F}{RT} \boldsymbol{\nabla} \cdot \left(\boldsymbol{R}^T \boldsymbol{\beta}_\mathbf{d} \boldsymbol{R} D_\pi C_\pi \boldsymbol{\nabla} \varphi \right) + \beta_c R_\pi + \beta_s \nu_\pi \tag{38}$$

$$0 = \beta_c \sum_\pi z_\pi C_\pi \tag{39}$$

$$0 = \beta_s \left(N_{ads} \dot{\theta}_{ads} - (\nu_{Va} - \nu'_{Va}) + \nu_{Ha} + 2\nu_T + (\nu_A - \nu'_A) - (\nu_{Vb} - \nu'_{Vb}) + \nu_{Hb} \right) \tag{40}$$

The capacity, diffusion, and surface distributors are accordingly defined as:

Distributed diffusion	**Physics−based**	
$\beta_c = \phi^m$	$\beta_c = h \left(\frac{1}{2\ell}\phi^2 + \frac{\ell}{2} \lvert \boldsymbol{\nabla}\phi \rvert^2 \right)$	(41)
$\boldsymbol{\beta}_d = \begin{bmatrix} \phi^m D_{\pi,2}/D_\pi & 0 \\ 0 & 0 \end{bmatrix}$	$\boldsymbol{\beta}_d = \left(\frac{1}{2\ell}\phi^2 + \frac{\ell}{2} \lvert \boldsymbol{\nabla}\phi \rvert^2 \right) \begin{bmatrix} h & 0 \\ 0 & D_\infty \end{bmatrix}$	(42)
$\beta_s = \frac{1}{\ell}\phi^2 + \ell \lvert \boldsymbol{\nabla}\phi \rvert^2$	$\beta_s = \frac{1}{\ell}\phi^2 + \ell \lvert \boldsymbol{\nabla}\phi \rvert^2$	(43)

where the distribution method is set within the code to either "WuLorenzis" for the distributed diffusion model described in [15] or "Subgrid" for the physics-based model from this work [1]. The manner in which the crack opening height is obtained is described in detail in Section 3.

Input parameters for this model are:

main.m

```
        %Crack-contained electrolyte diffusion, electro-migration, and reactions
        physics_in{6}.type = "PhaseFieldElectrolyte";
        physics_in{6}.Egroup = "Internal";
        physics_in{6}.D = [9.3; 5.3; 1.3; 2; 1.4; 1]*1e-9;  %Diffusion coefficients for [H
            OH Na CL Fe FeOH] species respectively [m/s]
        physics_in{6}.z = [1; -1; 1; -1; 2; 1];             %ionic charges for [H OH Na CL
            Fe FeOH] [-]
        physics_in{6}.pH0 = 5;                              %Initial and boudnary pH [-]
        physics_in{6}.NaCl = 0.6e3;                         %Initial and boundary Cl-
            concentration [mol/m^3]
        physics_in{6}.Lumped = [true; true];                %Flag indicationg whtehr to ude
             lumped integration for the water autoionisation and metal ion reactions
        physics_in{6}.k = [1e6; 1e-1; 1e-3; 1e-3];          %Dummy constant for the water
            auto-ionisation reaction, and reaction rates for Fe, Fe', FeOH reactions
        physics_in{6}.NAds = 1e-3;                          %Concentration of surface
            adsorption sites
        physics_in{6}.ksurf = [ 1e-4,   1e-10,  0.5,     0;  %Reaction constants for surface
             reactions, [k k' alpha E_eq]
                                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];
        physics_in{6}.NL = physics_in{1}.NL;                %Concentration of interstitial
            lattice sites [mol/m^3]
        physics_in{6}.Em = 0;                               %Metal electric potential
        physics_in{6}.Lumpedsurf = [1 1 1 1 1 1 1];         %Flags to indicate the use of
            lumped integration for surface reactions
        physics_in{6}.h0 = 1e-12;                           %small offset used to prevent
            ill-conditioned systems
        physics_in{6}.Flowtype = model;                     % Model to use for electrolyte
            diffusion, either Subgrid  WuLorenzis
        physics_in{6}.l = physics_in{1}.l;                  %Phase-field length scale
```

where the surface reaction rates k_{surf} follows the format from Eq. (30). This model also implements post-processing functions specifically for the electro-chemical system contained within the crevasse:

Models/@PhaseFieldElectrolyte/PhaseFieldElectrolyte.m

```
        function plotHeightData(obj, x_eval, h_est)
```

which plots the crack opening height and the crack-normal vectors,

```
        function plotFields(obj, physics)
```

plotting electrolyte concentrations and the electrolyte potential, and

```
        function plotFieldspH(obj, physics)
```

plotting the pH of the electrolyte. This last function solely show results where $\phi > 0.1$, filtering out the non-physical state of the electrolyte.

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)
```

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
```

```matlab
            %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:

```matlab
        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:

```matlab
        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:

```matlab
        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. Input parameters controlling the solver behaviour are:

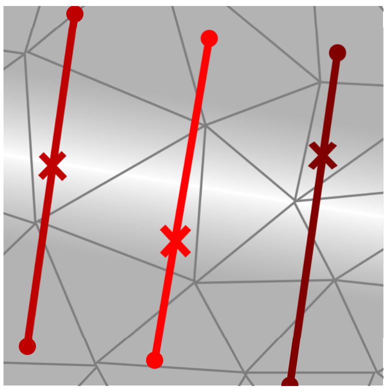

Figure 2: Schematic overview of integrations performed to calculate the crack opening height

main.m

```
        %% solver inputs
        solver_in.maxIt = 100;      %maximum amount of iterations within the nonlinear
            solver
        solver_in.Conv = 1e-6;      %Relative Energy-based convergence criterion
        solver_in.tiny = 1e-4;      %Absolute Energy-based convergence criterion
        solver_in.linesearch = false; %flag to indicate the use of a linear line-search
        solver_in.linesearchLims = [0.1 1]; %limits within which the line-search is
            performed
        solver_in.OuterLoops = 10;  %Maximum number of staggeres dolution loops to perform
```

3. Specifics: Fracture opening height

Within the model for the crack-contained electrolyte, Section 2.2.9, the estimated opening height of the crevasse is needed. These opening heights are calculated through an integral normal to the fracture surface [16, 17]:

$$h = \int \mathbf{u} \cdot \boldsymbol{\nabla}\phi \, \mathrm{d}n \tag{44}$$

with the line along which this integral is performed shown in Fig. 2, and the normal vector based on the phase-field as:

$$\boldsymbol{n} = \frac{\boldsymbol{\nabla}\phi}{|\boldsymbol{\nabla}\phi|} \tag{45}$$

These integrals are implemented in:

Models/@PhaseFieldElectrolyte/PhaseFieldElectrolyte.m

```
        function Construct_H_map(obj, physics)
            %Function which, for all integration points, calculates the
            %crack opening heights

            fprintf("        PhaseFieldElectrolyte Mapping Heights")

            obj.Heights = zeros(length(obj.ActiveElems), obj.mesh.ipcount1D^2);
            obj.Normals = zeros(length(obj.ActiveElems), obj.mesh.ipcount1D^2,2);
```

which fills the “Height” and “normals” matrices with the opening heights and normal vectors for all elements and all integration points. This function is only called once per staggered iteration step, as the phase-field and displacements are assumed to stay constant during the electro-chemical solution step.

First, the displacements and gradients within all integration points are determined:

Models/@PhaseFieldElectrolyte/PhaseFieldElectrolyte.m

```
            %% get integration-point data
            xy = zeros(length(obj.mesh.Elementgroups{obj.myGroupIndex}.Elems), obj.mesh.
                ipcount1D^2, 2);
            u = zeros(length(obj.mesh.Elementgroups{obj.myGroupIndex}.Elems), obj.mesh.
                ipcount1D^2, 2);
            dphi = zeros(length(obj.mesh.Elementgroups{obj.myGroupIndex}.Elems), obj.mesh.
                ipcount1D^2, 2);

            Svec = physics.StateVec;
            xmax = 0; ymax = 0;
            for el=1:length(obj.mesh.Elementgroups{obj.myGroupIndex}.Elems)
                [N, G, ~] = obj.mesh.getVals(obj.myGroupIndex, el);
                coords = obj.mesh.getIPCoords(obj.myGroupIndex, el);

                Elem_Nodes = obj.mesh.getNodes(obj.myGroupIndex, el);
                dofsX = obj.dofSpace.getDofIndices(obj.dofTypeIndices(1), Elem_Nodes);
                dofsY = obj.dofSpace.getDofIndices(obj.dofTypeIndices(2), Elem_Nodes);
                dofsPhi = obj.dofSpace.getDofIndices(obj.dofTypeIndices(3), Elem_Nodes);

                X = Svec{obj.dx_step}(dofsX);
                Y = Svec{obj.dx_step}(dofsY);
                PHI= Svec{obj.phi_step}(dofsPhi);

                for ip=1:obj.mesh.ipcount1D^2
                    xy(el,ip,1:2) = coords(:,ip);
                    u(el,ip,1:2)  = [N(ip,:)*X N(ip,:)*Y];
                    dphi(el,ip,:) = squeeze(G(ip,:,:))'*PHI;
                end

                xmax = max(max(coords(1,:)),xmax);
                ymax = max(max(coords(2,:)),ymax);
            end
```

and these opening heights are used to construct interpolation functions:

Models/@PhaseFieldElectrolyte/PhaseFieldElectrolyte.m

```
            %% construct interpolation functions
            x = xy(:,:,1); y = xy(:,:,2);
            ux = u(:,:,1); uy = u(:,:,2);
            dphix = squeeze(dphi(:,:,1)); dphiy = squeeze(dphi(:,:,2));

            F_ux = scatteredInterpolant(x(:),y(:),ux(:));
            warning('off','last')
            F_uy = scatteredInterpolant(x(:),y(:),uy(:));
            F_fx = scatteredInterpolant(x(:),y(:),dphix(:));
            F_fy = scatteredInterpolant(x(:),y(:),dphiy(:));
```

These interpolation functions are then used to construct an interpolation function for the crack opening height based on Eq. (44), for instance when the crack path is solely horizontal:

Models/@PhaseFieldElectrolyte/PhaseFieldElectrolyte.m

```
            %% calculate opening heights
            if obj.propDir == "Hor"
                x_eval = linspace(0,xmax,1000);  ny = 1000;
                h_est = zeros(size(x_eval));
                for iy=0:ny
                    y_eval = 0*x_eval+iy*ymax./ny;
                    h_est = h_est + abs(F_ux(x_eval,y_eval).*F_fx(x_eval,y_eval)*0 + F_uy(
                        x_eval,y_eval).*F_fy(x_eval,y_eval))*ymax./ny;
                end
                h_est = h_est;
                Fh = griddedInterpolant(x_eval, h_est);
```

and finally the crack opening heights and normal vectors within integration points are produced as:

Models/@PhaseFieldElectrolyte/PhaseFieldElectrolyte.m

```
269                 for n_el=1:length(obj.ActiveElems)
270                     el = obj.ActiveElems(n_el);
271                     coords = obj.mesh.getIPCoords(obj.myGroupIndex, el);
272 
273                     for ip=1:obj.mesh.ipcount1D^2
274                         coords_ip = coords(:,ip);
275                         if obj.propDir == "Hor"
276                             obj.Heights(n_el,ip) = Fh(coords_ip(1));
277                         else
278                             obj.Heights(n_el,ip) = Fh(coords_ip(2));
279                         end
280                         obj.Normals(n_el,ip,1) = F_fx(coords_ip(1),coords_ip(2));
281                         obj.Normals(n_el,ip,2) = F_fy(coords_ip(1),coords_ip(2));
282                         obj.Normals(n_el,ip,:) = obj.Normals(n_el,ip,:)/(sqrt(obj.Normals(n_el,
                            ip,1)^2+obj.Normals(n_el,ip,2)^2));
283 
284                         if obj.Heights(n_el,ip)<obj.h0
285                             obj.Heights(n_el,ip) = obj.h0;
286                         end
287                     end
```

It should be noted that this does not produce opening heights which exactly match to the expected opening based on the displacement and phase-field. However, as long as enough intermediate points are used (e.g. the 1000 points declared on line 248), these results will be virtually indistinguishable from the opening heights that could be calculated by evaluating the phase-field gradient and displacements exactly. The implementation presented here furthermore does not take into account branching cracks, instead assuming the phase-field tends towards zero away from the crack interface.

4. Sample results

4.1. Static cracks

For verification cases using a static crack, a single simulatioin can be run via "main_Static(model, l, u)", where model is either "Subgrid" or "WuLorenzis" for the physics-based or distributed diffusion models, l is the phase-field length scale, and u is the displacement (for instance, running "main_Static("Subgrid", 5e-4, 1e-5)"). Once running, a folder is created in "./Results_NoProp/" where output data is saved, and results are plotted after each converged time increment. Results after the first time increment are shown in Figs. 3 and 4.

In addition to running a single case, the file "DoSweep.m" performs a parametric sweep over the two models, and several external displacements and phase field length scales. It also runs simulations for a discrete representation of the crack, and creates a figure showing the comparison between the models. please note, as this function runs quite a few simulations, it takes a while to run fully.

4.2. Propagating cracks

The function "main.m" performs a simulation of a propagating crack in a square plate, using the parameters set in this main file. Please note that this function takes a while to run. Results after the first time increment are shown in Figs. 5 and 6.

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.

[3] T. Hageman, C. Andrade, E. Martínez-Pañeda, Corrosion rates under charge-conservation conditions, Electrochimica Acta 461 (2023) 142624.

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

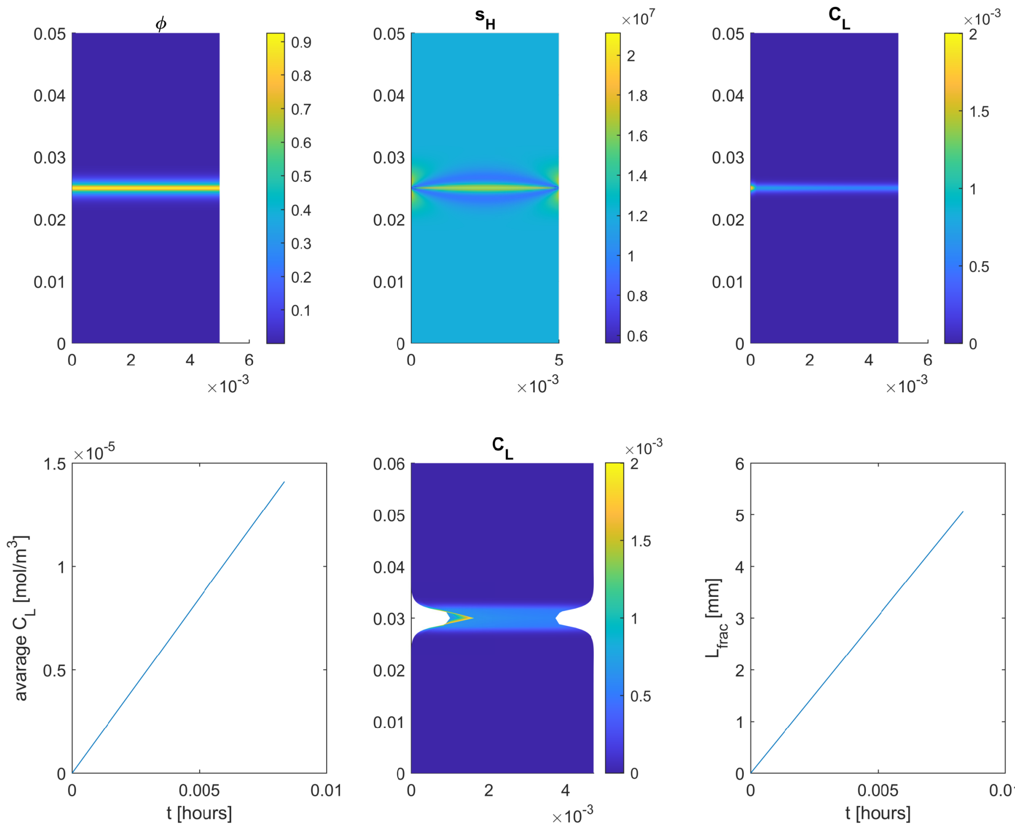

Figure 3: Results produced by running “main_Static(”Subgrid“, 5e-4, 1e-5)”. The first row shows the phase field, hydrostatic stress, and interstitial lattice hydrogen concentration. The second row shows the avarage hydrogen uptake over time, interstitial lattice concentration in the deformed configuration, and fracture length over time.

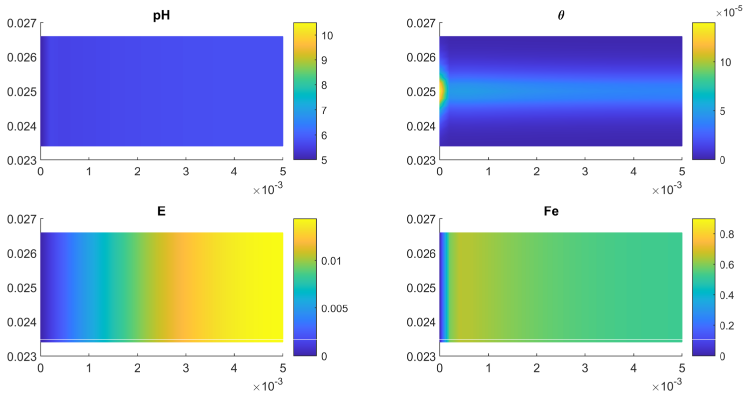

Figure 4: Results produced by running “main_Static(”Subgrid“, 5e-4, 1e-5)”, showing the electrolyte pH, surface adsorbed hydrogen concentration, electrolyte potential and Fe^{2+} concentration.

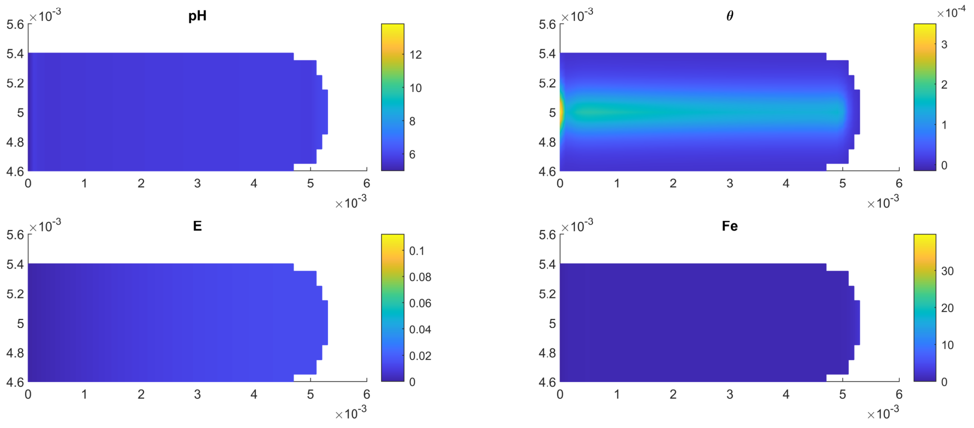

Figure 5: Results produced by running "main(1)". The first row shows the phase field, hydrostatic stress, and interstitial lattice hydrogen concentration. The second row shows the avarage hydrogen uptake over time, interstitial lattice concentration in the deformed configuration, and fracture length over time.

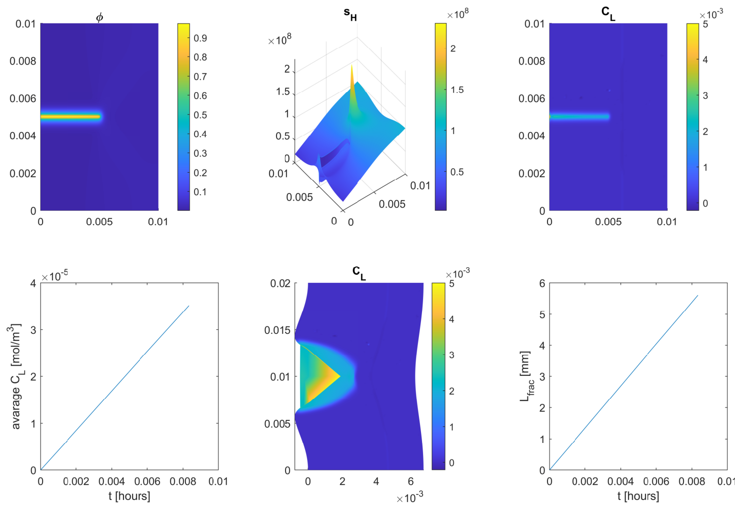

Figure 6: Results produced by running "main(1)", showing the electrolyte pH, surface adsorbed hydrogen concentration, electrolyte potential and Fe^{2+} concentration.

[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.

[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.

[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.

[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.

[12] C. Miehe, M. Hofacker, F. Welschinger, A phase field model for rate-independent crack propagation: Robust algorithmic

implementation based on operator splits, Computer Methods in Applied Mechanics and Engineering 199 (45-48) (2010) 2765–2778.

[13] C. Miehe, F. Welschinger, M. Hofacker, Thermodynamically consistent phase-field models of fracture: Variational principles and multi-field FE implementations, International Journal for Numerical Methods in Engineering 83 (10) (2010) 1273–1311.

[14] M. J. Borden, C. V. Verhoosel, M. A. Scott, T. J. Hughes, C. M. Landis, A phase-field description of dynamic brittle fracture, Computer Methods in Applied Mechanics and Engineering 217-220 (2012) 77–95.

[15] T. Wu, L. De Lorenzis, A phase-field approach to fracture coupled with diffusion, Computer Methods in Applied Mechanics and Engineering 312 (2016) 196–223.

[16] K. Yoshioka, D. Naumov, O. Kolditz, On crack opening computation in variational phase-field models for fracture, Computer Methods in Applied Mechanics and Engineering 369 (2020) 113210.

[17] C. Chukwudozie, B. Bourdin, K. Yoshioka, A variational phase-field model for hydraulic fracturing in porous media, Computer Methods in Applied Mechanics and Engineering 347 (2019) 957–982.