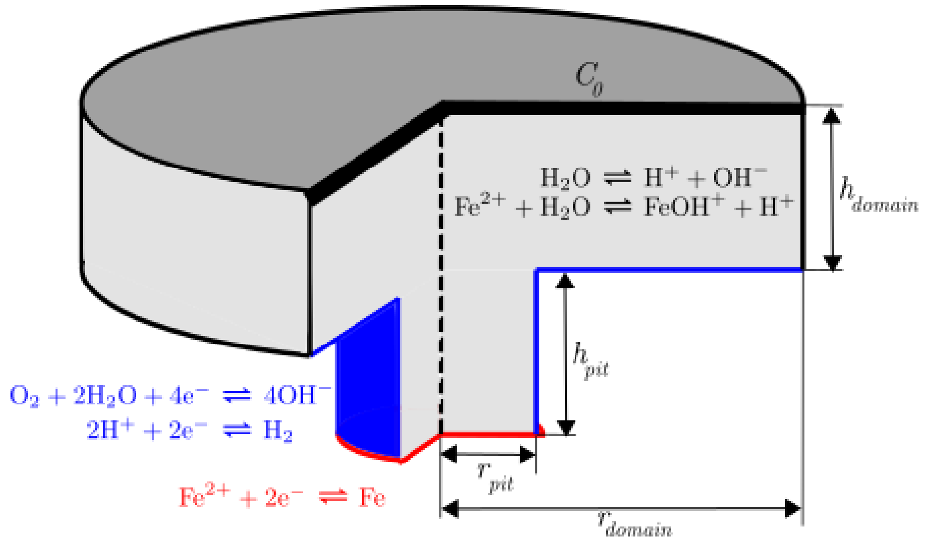
Corr_Sup_Surf_Sim: a MATLAB code to analyse corrosion rates under charge conservation conditions.

Tim Hageman[a,*], Emilio Martínez-Pañeda[a], Carmen Andrade[b]

[a] *Department of Civil and Environmental Engineering, Imperial College London, London SW7 2AZ, UK*
[b] *International Center of Numerical Methods in Engineering (CIMNE), Madrid 28010, Spain*

Abstract


C_0
$H_2O \rightleftharpoons H^+ + OH^-$
$Fe^{2+} + H_2O \rightleftharpoons FeOH^+ + H^+$
h_{domain}
h_{pit}
$O_2 + 2H_2O + 4e^- \rightleftharpoons 4OH^-$
$2H^+ + 2e^- \rightleftharpoons H_2$
$Fe^{2+} + 2e^- \rightleftharpoons Fe$
r_{pit}
r_{domain}

Documentation that accompanies the *MATLAB* code Corr_Sup_Surf_Sim. This documentation explains the usage of the implemented finite element framework, and highlight the main files. If using this module for research or industrial purposes, please cite: T. Hageman & C. Andrade & E. Martínez-Pañeda, Determination of corrosion rates under charge-conservation conditions with a supporting surface through numerical simulation. Journal of XXXXX (2023).

Keywords: MATLAB, electrochemistry, finite element method, charge conservation, corrosion

Contents

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

1. Introduction

Intro

1.1. Basic usage

For simulating the model as provided, running the function "main.m" performs all required actions: 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.

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 description of the electrolyte:

main.m

```
    %Electrolyte domain
    physics_in{1}.type = "Electrolyte";
    physics_in{1}.Egroup = "Electrolyte";
    physics_in{1}.D = [9.3; 5.3; 1.3; 2; 1.4; 1; 1]*1e-9;  %Diffusivity of H OH Na CL Fe
        FeOH O2 [m/s]
    physics_in{1}.z = [1; -1; 1; -1; 2; 1; 0];              %charge of each ionic species
        [-]
    physics_in{1}.pH0 = initpH;                             %initial pH [-]
    physics_in{1}.NaCl = initNaCL;                          %Initial Cl- concentration [mol
        /m^3]
    physics_in{1}.O2 = initO2;                              %Initial oxygen concentration [
        mol/m^3]
    physics_in{1}.Lumped = [true; true];        %should lumped integration be used to
        stabilide the water, metal volume reactions
    physics_in{1}.k = [1e6; 1e-1; 1e-3; 1e-3];  %reaction rates for the volume reactions:
        k_eq, k_f, k_f', k_feoh
```

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

```
%% Time stepping loop
for tstep = startstep:n_max
    %print information for commencing step
    disp("Step: "+string(tstep));
    disp("Time: "+string(physics.time));
    physics.dt = min(3600,dt*1.05^(tstep-1));
    disp("dTime: "+string(physics.dt));

    %stop simulations once maximum time reached
    if (physics.time>tmax)
        break
    end
```

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)
```

, 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.

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

```
    physics_in{4}.type = "Constrainer";
    physics_in{4}.Egroup = "E_Right";
    physics_in{4}.dofs = {"H";"OH";"Na";"Cl";"Fe";"FeOH"};
    physics_in{4}.conVal = [initH; initOH; initNa; initCl; 0; 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. OxygenLimiter

Similar to the "constrainer" model detailed in the previous section, but also accepts a time up to which the constraints are applied. After this time is reached, no further constraints are applied by this model. Required input properties to be defined:

main.m

```
        physics_in{3}.type = "OxygenLimiter";
        physics_in{3}.Egroup = "E_Top";
        physics_in{3}.dofs = {"O2"}; %Name of degree of freedom
        physics_in{3}.conVal = [initO2];    %value of oxygen constraint
        physics_in{3}.tmax = 48*3600;   %time after which constraint is removed
```

2.2.4. Electrolyte

The electrolyte model implements the Nernst-Planck mass balance:

$$\dot{C}_\pi + \boldsymbol{\nabla}\cdot\left(-D_\pi \boldsymbol{\nabla} C_\pi\right) + \frac{z_\pi F}{RT}\boldsymbol{\nabla}\cdot\left(-D_\pi C_\pi \boldsymbol{\nabla}\varphi\right) + R_\pi = 0 \tag{3}$$

for the ionic species and their name within the model file: H^+ ("H"), OH^- ("OH"), Na^+ ("Na"), Cl^- ("Cl"), Fe^{2+} ("Fe"), and $FeOH^+$ ("FeOH"). Additionally, it implements the electro-neutrality condition:

$$\sum z_\pi C_\pi = 0 \tag{4}$$

and bulk reactions:

$$\mathrm{H_2O} \underset{k'_w}{\overset{k_w}{\rightleftharpoons}} \mathrm{H^+} + \mathrm{OH^-} \tag{5}$$

$$\mathrm{Fe^{2+}} + \mathrm{H_2O} \underset{k'_{fe}}{\overset{k_{fe}}{\rightleftharpoons}} \mathrm{FeOH^+} + \mathrm{H^+} \tag{6}$$

$$\mathrm{FeOH^+} + \mathrm{H_2O} \xrightarrow{k_{feoh}} \mathrm{Fe(OH)_2} + \mathrm{H^+} \tag{7}$$

with reaction rates:

$$R_{\mathrm{H^+},w} = R_{\mathrm{OH^-}} = k_w C_{\mathrm{H_2O}} - k'_w C_{\mathrm{H^+}} C_{\mathrm{OH^-}} = k_{eq}\left(K_w - C_{\mathrm{H^+}} C_{\mathrm{OH^-}}\right) \tag{8}$$

$$R_{\mathrm{Fe^{2+}}} = -k_{fe} C_{\mathrm{Fe^{2+}}} + k'_{fe} C_{\mathrm{FeOH^+}} C_{\mathrm{H^+}} \tag{9}$$

$$R_{\mathrm{FeOH^+}} = k_{fe} C_{Fe^{2+}} - C_{\mathrm{FeOH^+}}(k_{feoh} + k'_{fe} C_{\mathrm{H^+}}) \tag{10}$$

$$R_{\mathrm{H^+},fe} = k_{fe} C_{\mathrm{Fe^{2+}}} - C_{\mathrm{FeOH^+}}(k'_{fe} C_{\mathrm{H^+}} - k_{feoh}) \tag{11}$$

For this model, the input properties required are:

main.m

```
    physics_in{1}.type = "Electrolyte";
    physics_in{1}.Egroup = "Electrolyte";
    physics_in{1}.D = [9.3; 5.3; 1.3; 2; 1.4; 1; 1]*1e-9;  %Diffusivity of H OH Na CL Fe
        FeOH O2 [m/s]
    physics_in{1}.z = [1; -1; 1; -1; 2; 1; 0];              %charge of each ionic species
        [-]
    physics_in{1}.pH0 = initpH;                              %initial pH [-]
```

```
        physics_in{1}.NaCl = initNaCL;                            %Initial Cl- concentration [mol
            /m^3]
        physics_in{1}.O2 = initO2;                                %Initial oxygen concentration [
            mol/m^3]
        physics_in{1}.Lumped = [true; true];        %should lumped integration be used to
            stabilide the water, metal volume reactions
        physics_in{1}.k = [1e6; 1e-1; 1e-3; 1e-3];  %reaction rates for the volume reactions:
            k_eq, k_f, k_f', k_feoh
```

This model employs a lumped integration scheme when the vector "Lumped" contains true.

2.2.5. *ElectrolyteInterface*

Finally, the electrolyteInterface model implements the metal-electrolyte coupling through the corrosion reaction at the anodic surface:

$$\mathrm{Fe}^{2+} + 2\mathrm{e}^- \underset{k_c'}{\overset{k_c}{\rightleftharpoons}} \mathrm{Fe} \tag{12}$$

and the cathodic surface includes oxygen and hydrogen-related reactions:

$$\mathrm{O}_2 + 2\mathrm{H}_2\mathrm{O} + 4\mathrm{e}^- \underset{k_o'}{\overset{k_o}{\rightleftharpoons}} 4\mathrm{OH}^- \tag{13}$$

$$2\mathrm{H}^+ + 2\mathrm{e}^- \underset{k_h'}{\overset{k_h}{\rightleftharpoons}} H_2 \tag{14}$$

It also implements the charge conservation at the surface:

$$\int_\Gamma i_c + i_o + i_h \,\mathrm{d}\Gamma = \int_\Gamma 2F\nu_c + 4F\nu_o + 2F\nu_h \,\mathrm{d}\Gamma = 0 \tag{15}$$

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

main.m

```
        F_const = 96485.3329; %Faraday constant
        physics_in{2}.type = "ElectrolyteInterface";
        physics_in{2}.Anode = "Anode";
        physics_in{2}.Cathode = "Cathode";
        physics_in{2}.k = [ 1e-1/F_const,   1e-1/F_const,   0.5,    -0.4;  % Fe <-> Fe2+ (anode
            )
                            1e-4/F_const,   1e-6/F_const,   0.5,    0;     % H+ <->H2    (
                                cathode)
                            1e-6/F_const,   1e-6/F_const,   0.5,    0.4;   % O2 <->OH-   (
                                cathode)
                            ]; %reaction constants k, k', alpha, E_eq
        physics_in{2}.ChargeConserve = true; %Enforce charge-conservation conditions
        physics_in{2}.Em = 0;   %Metal potential (unused if chargeconserve=true
        physics_in{2}.Lumped = [1 1 1]; %Use lumped integration to stabilise surface reactions?
```

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:

$$k = \begin{bmatrix} k_c & k_c' & \alpha_c & E_{eq,c} \\ k_h & k_h' & \alpha_h & E_{eq,h} \\ k_o & k_o' & \alpha_o & E_{eq,o} \end{bmatrix} \tag{16}$$

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 "Electrolyte" for the electrolyte domain composed of surface elements, and defining an element group "Anode" composed of line elements which coincide with the anodic part of the metal-electrolyte boundary). The geometry of the problem is defined through procedures within the mesh class, specifically within "@Mesh/CorrosionPit_Generator.m":

@Mesh/CorrosionPit_Generator.m

```
        R1 = [3,4,0,Lx,Lx,0,0,0,Ly,Ly]';
        R2 = [3,4,0,Lfrac,Lfrac,0,0,0,-Hfrac,-Hfrac]';
        gm = [R1,R2];
        sf = '(R1+R2)';
```

Defining rectangle R1 to represent the outer part of the domain and rectangle R2 for the pit. The mesh uses the standard matlab mesh generator "GenerateMesh" to convert this geometric description, allowing for element sizes to be defined:

```
        geo = createpde(1);
        geometryFromEdges(geo,shp);
        generateMesh(geo,'Hmax',dxmax,'Hgrad',1.3,'Hedge',{[3,4], dxmin});
```

which allows for defining minimum element sizes through Hedge, and maximum sizes through Hmax.

The mesh class also provides a direct interface from which to get the element shape functions, 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);
```

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.

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) and quadratic Lagrangian and Bernstein line elements (L3 and L3B).

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)
            %Assemble stiffness matrix and internal force vector
            dofcount = obj.dofSpace.NDofs;

            obj.condofs = [];
            obj.convals = [];

            nonz = round(nnz(obj.K)*1.2);
            obj.K = spalloc(dofcount, dofcount, nonz);
            obj.fint = zeros(dofcount, 1);

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

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
```

This function requires the name of a degree of freedom (for instance “H” for the hydrogen ion concentration), a scale to indicate whether the mesh is plotted in deformed (scale>0) or undeformed (scale=0) configuration (for the simulations performed, this is always 0, since no solid deformations are simulated), and the name of an element group on which to plot the results (“Electrolyte”, and “Anode”/“Cathode” for the metal-electrolyte interface.

2.6. Dofspace

This class converts the node numbering and degree of freedom type 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 through a string indicating their name:

@DofSpace/DofSpace.m

```
        function dofIndex = addDofType(obj, dofnames)
```

after which they can be added to nodes through:

```
        function addDofs(obj, dofIndices, 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)
```

2.7. Solver

The solver class implements a 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. To obtain a solution for the linearised system, a sparse iterative solver is used (with as back-up option a direct solver) in conjunction with a preconditioner:

@Solver/Solve.m

```
        recalc_pre = true;
        if (recalc_pre)
            [P,R,C] = equilibrate(obj.physics.K);
            recalc_pre = false;
        end

        %solve linear system
        if true
            d = -R*P*obj.physics.fint;
            B = R*P*obj.physics.K*C;

            if false
                dy = B\d;
            else
                try
                    [L,U] = ilu(B,struct('type','nofill'));
                    [dy,~] = gmres(B,d,500,1e-4,5000,L,U);
                catch
                    dy = B\d;
                end
            end
            dx = C*dy;
        else
```

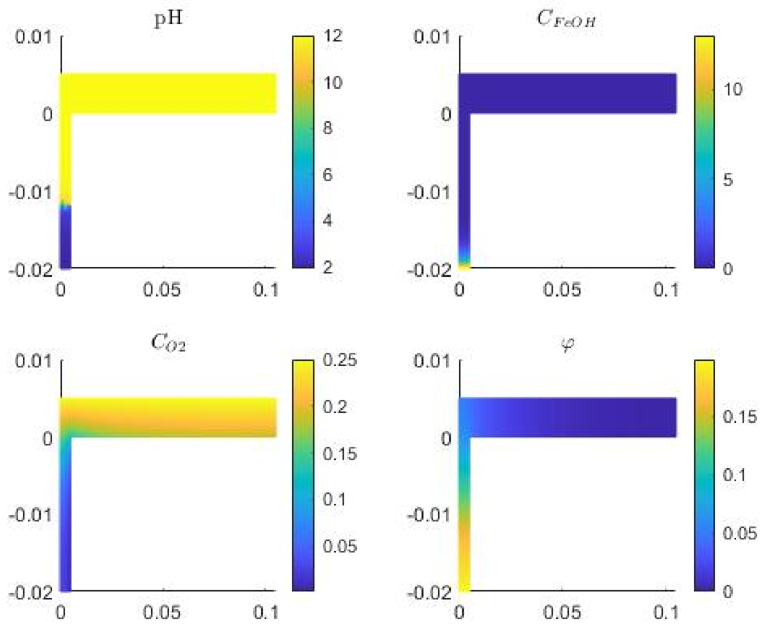
in which the equilibriate preconditioner greatly decreases the conditioning number of the matrix, thereby reducing errors during the solving process.

3. Sample results

Simulating the pencil electrode case as set up within "main.m" (equivalent to the case described in Sec. 4 from T. Hageman & C. Andrade & E. Martínez-Pañeda, Determination of corrosion rates under charge-conservation conditions with a supporting surface through numerical simulation. Journal of XXXXX (2023), except for a smaller domain radius) automatically opens figures showing the results as the simulation proceeds. These results can also be obtained after it finishes. Results shown during the simulation are:

main.m

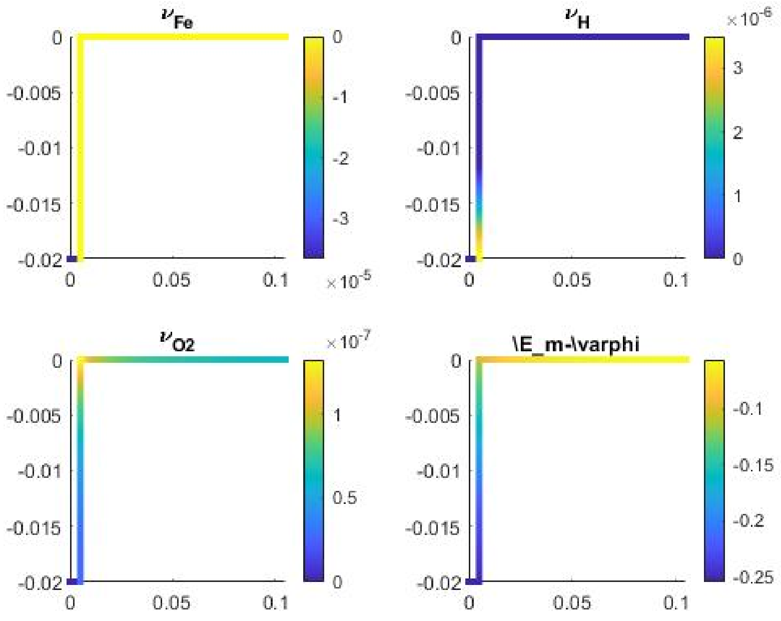
```
figure(43)
    clf
    physics.models{1}.plotFields(physics);
```



which displays the pH, concentrations of $FeOH^+$ ions and oxygen, and the electrolyte potential φ at the current time (or, if done during post-processing, at the time the datafile was saved). Additionally, reaction rates can be obtained through

main.m

```
figure(44)
    clf
    physics.models{2}.plotReactions(physics);
```



Showing the local reaction rates for the corrosion, hydrogen, and oxygen reactions, using the convention of possitive signs being electron-consuming, while negative signs indicate the electron-producing direction of the reactions. Additionally, the electric overpotential is shown at the interface.