Lumped Electrochemistry: a MATLAB code to utilize the stabilising effects of lumped integration schemes for the simulation of metal-electrolyte reactions

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 lumped electrochemistry. 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 volume and interface reactions, which are integrated using a lumped integration scheme.

If using this module for research or industrial purposes, please cite: T. Hageman & E. Martínez-Pañeda, Stabilising effects of lumped integration schemes for the simulation of metal-electrolyte reactions. Journal of computational physics (2023).

Keywords: MATLAB, electrochemistry, finite element method, lumped integration, oscillations, hydrogen absorption

Contents

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

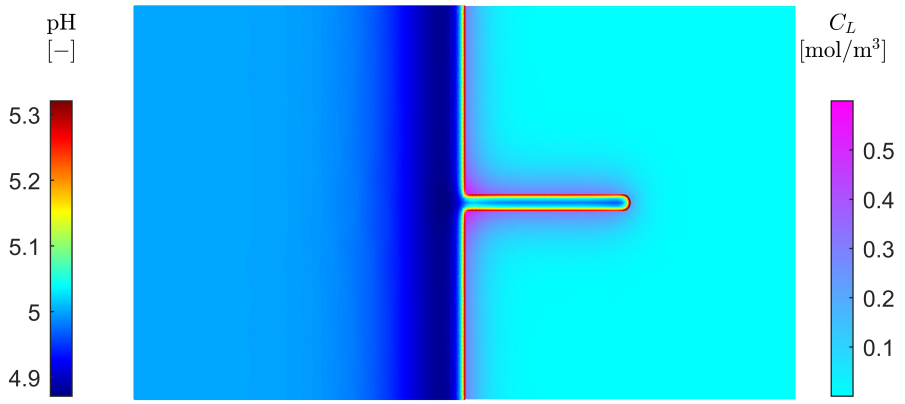

Figure 1: Example of surface data plotted during the post-processing procedure.

1. Introduction

Electro-chemical systems often include reactions with rates varying by several orders of magnitude. Some reaction rates might enforce a near-instant equilibrium, while others occur over many hours to years. While it is possible to enforce equilibrium reactions a priori, this requires assumptions of the dominant reaction mechanics which limit the applicability of the resulting model to a narrow range of circumstances. In contrast, when chemical reactions are modelled as-is, the numerical models are commonly hindered by poor convergence rates or spurious oscillations, both of which require small time increments and which make simulating realistic time-scales costly or even infeasible. Here, the numerical model, implemented within MATLAB, is presented that accompanies T. Hageman & E. Martínez-Pañeda, Stabilising effects of lumped integration schemes for the simulation of metal-electrolyte reactions. Journal of computational physics (2023). This model uses a lumped integration scheme, which has been shown to greatly improve the stability. The remainder of this documentation first discusses basic usage of the provided finite element method code, then provides a more in-depth description of the implementation aspects and components of the code, and finally discusses the lumped integration scheme in detail. This code has been verified to work with matlab versions 2021b and 2022a, older versions might not be compatible.

DISCLAIMER: While this code has been cross-verified with comparison to simulation results from the commercial finite element package COMSOL, it can not be guaranteed to be error-free. Before using this code for relevant or critical applications, especially when simulating cases not directly included, please perform your own verification. The authors are not responsible for any issues arising from mistakes within this matlab code.

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. A separate file is present to perform post-processing based on saved output files, “PostProcessing.m”, which based on a path to the output files visualises the results.

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 linear-elastic description of the solid domain:

main.m

```
59          % Linear elastic material description for metal domain
60          physics_in{1}.type = "LinearElastic";
61          physics_in{1}.Egroup = "Metal";
62          physics_in{1}.young = 200e9;    % Youngs modulus [Pa]
63          physics_in{1}.poisson = 0.3;    % Poisson ratio [-]
```

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

```
164     for tstep = startstep:n_max
165         disp("Step: "+string(tstep));
166         disp("Time: "+string(physics.time));
167         physics.dt = dt*1.05^(tstep-1);
168         disp("dTime: "+string(physics.dt));
169
170         % solve current time increment
171         solver.Solve();

191         if (physics.time>tmax)
192             break
193         end
194     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

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

, and committing history dependent or path dependent variables:

```
13          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{3}.type = "Constrainer";
physics_in{3}.Egroup = "M_Bottom";
physics_in{3}.dofs = {"dx"};
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

```
% Linear elastic material description for metal domain
physics_in{1}.type = "LinearElastic";
physics_in{1}.Egroup = "Metal";
physics_in{1}.young = 200e9;    % Youngs modulus [Pa]
physics_in{1}.poisson = 0.3;    % Poisson ratio [-]
```

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

```
% add contribution to stiffness matrix and force vector
physics.fint = physics.fint + obj.myK*physics.StateVec;
physics.K = physics.K + obj.myK;
```

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

2.2.4. HydrogenDiffusion

This model implements the hydrogen mass conservation, through the diffusion equation:

$$\dot{C}_L + \boldsymbol{\nabla} \cdot \left(-\frac{D_L}{1 - C_L/N_L} \boldsymbol{\nabla} C_L\right) + \boldsymbol{\nabla} \cdot \left(\frac{D_L C_L \overline{V}_H}{RT} \boldsymbol{\nabla} \sigma_H\right) = 0 \tag{4}$$

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

main.m

```
% Hydrogen diffusion within the metal domain
physics_in{2}.type = "HydrogenDiffusion";
physics_in{2}.Egroup = "Metal";
physics_in{2}.DL = 1e-9;     % Lattice difusivity
physics_in{2}.NL = 1e6;      % Amount of interstitial lattice sites [mol/m^3]
```

This model presumes the linear-elastic model is provided within input 1, from which the Young's modulus and Poisson ratio are taken.

2.2.5. Electrolyte

The electrolyte model implements the Nernst-Planck mass balance:

$$\dot{C}_\pi + \boldsymbol{\nabla}\cdot(-D_\pi \boldsymbol{\nabla} C_\pi) + \frac{z_\pi F}{RT}\boldsymbol{\nabla}\cdot(-D_\pi C_\pi \boldsymbol{\nabla}\varphi) + R_\pi = 0 \tag{5}$$

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:

$$\sum z_\pi C_\pi = 0 \tag{6}$$

and bulk reactions:

$$H_2O \underset{k'_w}{\overset{k_w}{\rightleftharpoons}} H^+ + OH^- \tag{7}$$

$$Fe^{2+} + H_2O \underset{k'_{fe}}{\overset{k_{fe}}{\rightleftharpoons}} FeOH^+ + H^+ \tag{8}$$

$$FeOH^+ + H_2O \xrightarrow{k_{feoh}} Fe(OH)_2 + H^+ \tag{9}$$

with reaction rates:

$$R_{H^+,w} = R_{OH^-} = k_w C_{H_2O} - k'_w C_{H^+} C_{OH^-} = k_{eq}\left(K_w - C_{H^+} C_{OH^-}\right) \tag{10}$$

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

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

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

For this model, the input properties required are:

main.m

```
% Nernst-planck, electroneutrality, and volume reactions for electrolyte
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;  % Diffusion coefficients [m/s]
    for ions: H OH Na Cl Fe FeOH
physics_in{7}.z = [1; -1; 1; -1; 2; 1]; % ionic charges
physics_in{7}.pH0 = 5;  % Initial condition pH
physics_in{7}.NaCl = 0.6e3; % Initial concentration of NaCl
physics_in{7}.Lumped = [true; true]; % Flag for using lumped integration for water
    auto-ionisation and metal-ion reactions
physics_in{7}.k = [1e6; 1e-1; 1e-3; 1e-3]; % Reaction constants k_eq, k_fe, k_fe',
    k_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 Section 3

2.2.6. ElectrolyteInterface

Finally, the electrolyteInterface model implements the metal-electrolyte coupling through the surface reactions:

$$\text{Volmer (acid):} \quad \mathrm{H^+ + M + e^-} \underset{k'_{Va}}{\overset{k_{Va}}{\rightleftharpoons}} \mathrm{MH}_{ads} \tag{14}$$

$$\text{Heyrovsky (acid):} \quad \mathrm{H^+ + e^- + MH}_{ads} \underset{k'_{Ha}}{\overset{k_{Ha}}{\rightleftharpoons}} \mathrm{M + H_2} \tag{15}$$

$$\text{Volmer (base):} \quad \mathrm{H_2O + M + e^-} \underset{k'_{Vb}}{\overset{k_{Vb}}{\rightleftharpoons}} \mathrm{MH}_{ads} + \mathrm{OH^-} \tag{16}$$

$$\text{Heyrovsky (base):} \quad \mathrm{H_2O + e^- + MH}_{ads} \underset{k'_{Hb}}{\overset{k_{Hb}}{\rightleftharpoons}} \mathrm{M + H_2 + OH^-} \tag{17}$$

$$\text{Tafel:} \quad 2\mathrm{MH}_{ads} \underset{k'_{T}}{\overset{k_{T}}{\rightleftharpoons}} 2\mathrm{M + H_2} \tag{18}$$

$$\text{Absorption:} \quad \mathrm{MH}_{ads} \underset{k'_{A}}{\overset{k_{A}}{\rightleftharpoons}} \mathrm{MH}_{abs} \tag{19}$$

$$\text{Corrosion:} \quad \mathrm{Fe^{2+} + 2e^-} \underset{k'_{c}}{\overset{k_{c}}{\rightleftharpoons}} \mathrm{Fe} \tag{20}$$

with reaction rates:

	Forward	Backward	
Volmer(acid) :	$\nu_{Va} = k_{Va} C_{\mathrm{H^+}} (1-\theta_{ads}) e^{-\alpha_{Va}\frac{\eta F}{RT}}$	$\nu'_{Va} = k'_{Va}\theta_{ads} e^{(1-\alpha_{Va})\frac{\eta F}{RT}}$	(21)
Heyrovsky(acid) :	$\nu_{Ha} = k_{Ha} C_{\mathrm{H^+}} \theta_{ads} e^{-\alpha_{Ha}\frac{\eta F}{RT}}$	$\nu'_{Ha} = k'_{Ha}(1-\theta_{ads}) p_{\mathrm{H_2}} e^{(1-\alpha_{Ha})\frac{\eta F}{RT}}$	(22)
Volmer(base) :	$\nu_{Vb} = k_{Vb}(1-\theta_{ads}) e^{-\alpha_{Vb}\frac{\eta F}{RT}}$	$\nu'_{Vb} = k'_{Vb} C_{\mathrm{OH^-}} \theta_{ads} e^{(1-\alpha_{Vb})\frac{\eta F}{RT}}$	(23)
Heyrovsky(base) :	$\nu_{Hb} = k_{Hb}\theta_{ads} e^{-\alpha_{Hb}\frac{\eta F}{RT}}$	$\nu'_{Hb} = k'_{Hb}(1-\theta_{ads}) p_{\mathrm{H_2}} C_{\mathrm{OH^-}} e^{(1-\alpha_{Hb})\frac{\eta F}{RT}}$	(24)
Tafel :	$\nu_{T} = k_{T} \lvert\theta_{ads}\rvert \theta_{ads}$	$\nu'_{T} = k'_{T}(1-\theta_{ads}) p_{\mathrm{H_2}}$	(25)
Absorption :	$\nu_{A} = k_{A}(N_L - C_L)\theta_{ads}$	$\nu'_{A} = k'_{A} C_L (1-\theta_{ads})$	(26)
Corrosion :	$\nu_{c} = k_{c} C_{\mathrm{Fe^{2+}}} e^{-\alpha_{c}\frac{\eta F}{RT}}$	$\nu'_{c} = k'_{c} e^{(1-\alpha_{c})\frac{\eta F}{RT}}$	(27)

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

Models/@ElectrolyteInterface/ElectrolyteInterface.m

```
364         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. These reaction rates are integrated through a lumped scheme, with details about this scheme discussed in Section 3. In addition to the reaction rates, the electrolyte interface model also resolves the surface mass balance:

$$N_{ads}\dot{\theta}_{ads} - (\nu_{Va} - \nu'_{Va}) + \nu_{Ha} + 2\nu_T + (\nu_A - \nu'_A) - (\nu_{Vb} - \nu'_{Vb}) + \nu_{Hb} = 0 \tag{28}$$

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

main.m

```
113         % Metal-electrolyte interface
114         physics_in{9}.type = "ElectrolyteInterface";
115         physics_in{9}.Egroup = "Interface";
```

```
physics_in{9}.NAds = 1e-3; % Concentration of surface sites [mol/m^2]
physics_in{9}.k = k; %Reaction constants
physics_in{9}.NL = physics_in{2}.NL; % Concentration of interstitial lattice sites
    [mol/m^3]
physics_in{9}.Em = Em; % Metal Potential [V_SHE]
physics_in{9}.Lumped = [1 1 1 1 1 1 1]; %Flags to enable lumped integration on a
    per-reaction basis
```

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:

$$k = \begin{bmatrix} k_{Va} & k'_{Va} & \alpha_{Va} & E_{eq,Va} \\ k_{Ha} & k'_{Ha} & \alpha_{Ha} & E_{eq,Ha} \\ k_T & k'_T & - & - \\ k_A & k'_A & - & - \\ k_{Vb} & k'_{Vb} & \alpha_{Vb} & E_{eq,Vb} \\ k_{Hb} & k'_{Hb} & \alpha_{Hb} & E_{eq,Hb} \\ k_c & k'_c & \alpha_c & E_{eq,c} \end{bmatrix} \tag{29}$$

with the empty entries not used within the model.

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 "Metal" for the metal domain composed of surface elements, and defining an element group "Interface" composed of line elements which coincide with the left boundary of the metal and the right boundary of the electrolyte). The geometry of the problem is defined through procedures within the mesh class, specifically within "@Mesh/Geometry_Generator.m":

@Mesh/Geometry_Generator.m

```
R1 = [3,4,0,Lx,Lx,0,0,0,Ly,Ly]';
C1 = [1,5e-3-0.2e-3,5e-3,0.2e-3]';
C1 = [C1;zeros(length(R1) - length(C1),1)];
R2 = [3,4,0,5e-3-0.2e-3,5e-3-0.2e-3,0,Ly/2-0.2e-3,Ly/2-0.2e-3,Ly/2+0.2e-3,Ly/2+0.2e
    -3]';
R3 =  [3,4,-Lx,0,0,-Lx,0,0,Ly,Ly]';
gm = [R1,C1,R2,R3];
sf = '(R1-C1-R2)+(R3+C1+R2)';
```

Defining rectangle R1 to represent the metal domain and rectangle R3 for the electrolyte domain, and adding/substracting circle C1 and rectangle R2 to create the crack geometry. The mesh uses the standard matlab mesh generator "GenerateMesh" to convert this geometric description, allowing for element sizes to be defined:

```
generateMesh(geo,'Hmax',1e-3,'Hgrad',1.2,'Hedge',{[2,3,7,8,11,12], 0.1e-3});
```

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. 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

```
48          function Assemble(obj)
49              %Assemble stiffness matrix and internal force vector
50              dofcount = obj.dofSpace.NDofs;
51
52              obj.condofs = [];
53              obj.convals = [];
54
55              nonz = round(nnz(obj.K)*1.2);
56              obj.K = spalloc(dofcount, dofcount, nonz);
57              obj.fint = zeros(dofcount, 1);
58
59              disp("    Assembling:")
60              for m=1:length(obj.models)
61                  obj.models{m}.getKf(obj);
62              end
63          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

```
25          PlotNodal(obj, dofName, dispscale, plotloc) %exterior defined, plots nodal
                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 ("Metal" for the metal domain, "Electrolyte" for the electrolyte, and "Interface" 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

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

after which they can be added to nodes through:

```
50          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:

```
82          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 direct solver is used in conjunction with a preconditioner:

@Solver/Solve.m

```
21          %matrix preconditioning
22          recalc_pre = true;
23          if (recalc_pre)
24              [P,R,C] = equilibrate(obj.physics.K);
25              recalc_pre = false;
26          end
27
28          %solve linear system
29          if true
30              d = -R*P*obj.physics.fint;
31              B = R*P*obj.physics.K*C;
32              if true %if true, using direct solver, else using iterative gmres
33                  dy = B\d;
34              else
35                  [L,U] = ilu(B,struct('type','nofill'));
36                  dy = gmres(B,d,[],1e-4,500,L,U);
37              end
38              dx = C*dy;
39          else
40              dx = -obj.physics.K\obj.physics.fint;
41          end
42          tsolve = toc(tsolve);
43          fprintf("        (Solver time:"+string(tsolve)+")\n");
```

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

3. Specifics: Lumped integration

This section will go into the implementation specifics regarding the lumped integration scheme. Within this scheme, the volume reactions (inside the electrolyte) and surface reactions (at the metal-electrolyte interface) are not integrated using the standard Gauss integration scheme, but instead are implemented using a lumped scheme. In our paper, it has been shown that this enhanced the stability of the scheme, while also suppressing non-physical oscillations.

The lumped integration scheme is performed on a per-element basis. As a first step, the lumped weight vector is constructed within the standard finite element integration loop:

Models/@ElectrolyteInterface/ElectrolyteInterface.m

```
91          %Assembly, loop over all elements
92          parfor n_el=1:size(obj.mesh.Elementgroups{obj.myGroupIndex}.Elems, 1)
93
94              % get nodes and shape functions for element
95              Elem_Nodes = obj.mesh.getNodes(obj.myGroupIndex, n_el);
96              [N, G, w] = obj.mesh.getVals(obj.myGroupIndex, n_el);

136             % initialize lumped weight vector
137             C_Lumped = zeros(length(dofsE), 1);
138
139             %Gauss integration loop
140             for ip=1:length(w)
```

```
            % lumped integration weight
            C_Lumped = C_Lumped + w(ip)*N(ip,:)';
        end
```

Here, the lumped weight vector is calculated as:

$$\mathbf{W}_{Lumped} = \int_{\Omega_{el}} \mathbf{N}^T \, \mathrm{d}\Omega_{el} \tag{30}$$

and provides weights relating to the relative influence of each node within the current element. This lumped weights vector is then used to perform the integration of the reaction terms on a node-by-node basis:

$$\mathbf{R} = \sum_{el} \sum_{nd} \mathbf{W}_{lumped}(nd) R \tag{31}$$

where $\mathbf{R}$ is the reaction rate term allocated to the force vector, and R the local reaction point based on the nodal values. Implementation wise, this corresponds to a loop over all the nodes contained within the element:

```
    % lumped integrations loop
    for i=1:length(dofsT)
        % get reaction rates based on nodal values
        [react, dreact, products] = obj.reactions(C(i,1), C(i,2),C(i,5), T(i),
            E(i), CL(i));

        %Add to force vector and tangential matrix
        for r=1:7
            q_C(i,1) = q_C(i,1) - C_Lumped(i)*(react(r,1)-react(r,2))*products(
                r,1)*obj.Lumped(r);
            q_C(i,2) = q_C(i,2) - C_Lumped(i)*(react(r,1)-react(r,2))*products(
                r,2)*obj.Lumped(r);
            q_C(i,5) = q_C(i,5) - C_Lumped(i)*(react(r,1)-react(r,2))*products(
                r,3)*obj.Lumped(r);
            q_T(i)   = q_T(i)   - C_Lumped(i)*(react(r,1)-react(r,2))*products(
                r,4)*obj.Lumped(r);
            q_CL(i)  = q_CL(i)  - C_Lumped(i)*(react(r,1)-react(r,2))*products(
                r,5)*obj.Lumped(r);

            for n=1:obj.n_species
                dqC_dC(i,i,1,n) = dqC_dC(i,i,1,n)-C_Lumped(i)*(dreact(r,1,3+n)-
                    dreact(r,2,3+n))*products(r,1)*obj.Lumped(r);
                dqC_dC(i,i,2,n) = dqC_dC(i,i,2,n)-C_Lumped(i)*(dreact(r,1,3+n)-
                    dreact(r,2,3+n))*products(r,2)*obj.Lumped(r);
                dqC_dC(i,i,5,n) = dqC_dC(i,i,5,n)-C_Lumped(i)*(dreact(r,1,3+n)-
                    dreact(r,2,3+n))*products(r,3)*obj.Lumped(r);
                dqT_dC(i,i,n)   = dqT_dC(i,i,n)  -C_Lumped(i)*(dreact(r,1,3+n)-
                    dreact(r,2,3+n))*products(r,4)*obj.Lumped(r);
            end
            dqC_dE(i,i,1)   = dqC_dE(i,i,1) - C_Lumped(i)*(dreact(r,1,1)-dreact
                (r,2,1))*products(r,1)*obj.Lumped(r);
            dqC_dE(i,i,2)   = dqC_dE(i,i,2) - C_Lumped(i)*(dreact(r,1,1)-dreact
                (r,2,1))*products(r,2)*obj.Lumped(r);
            dqC_dE(i,i,5)   = dqC_dE(i,i,5) - C_Lumped(i)*(dreact(r,1,1)-dreact
                (r,2,1))*products(r,3)*obj.Lumped(r);
            dqT_dE(i,i)     = dqT_dE(i,i)   - C_Lumped(i)*(dreact(r,1,1)-dreact
                (r,2,1))*products(r,4)*obj.Lumped(r);

            dqC_dT(i,i,1)   = dqC_dT(i,i,1) - C_Lumped(i)*(dreact(r,1,2)-dreact
                (r,2,2))*products(r,1)*obj.Lumped(r);
            dqC_dT(i,i,2)   = dqC_dT(i,i,2) - C_Lumped(i)*(dreact(r,1,2)-dreact
                (r,2,2))*products(r,2)*obj.Lumped(r);
            dqC_dT(i,i,5)   = dqC_dT(i,i,5) - C_Lumped(i)*(dreact(r,1,2)-dreact
                (r,2,2))*products(r,3)*obj.Lumped(r);
```

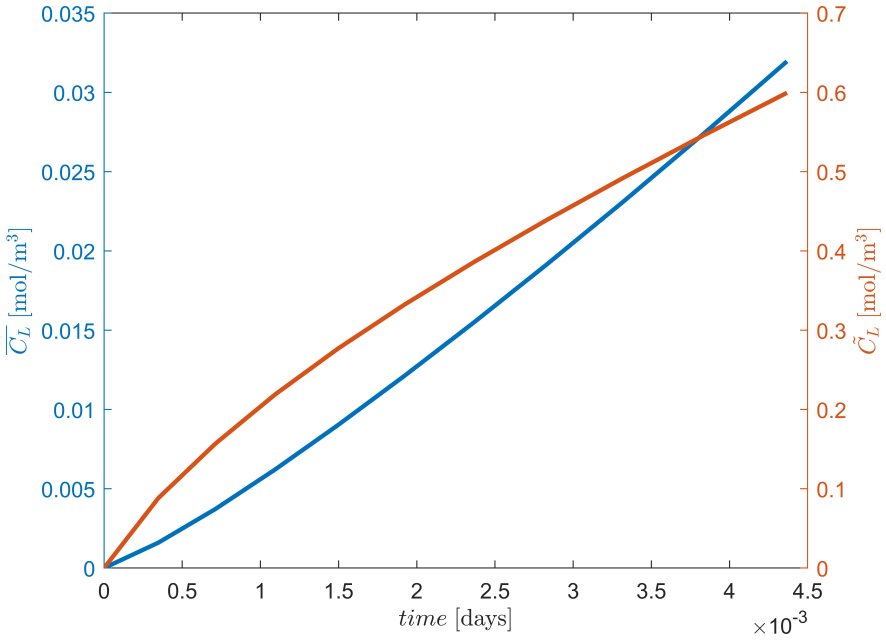

Figure 2: Example of temporal data plotted during the post-processing procedure.

```
                dqT_dT(i,i)     = dqT_dT(i,i)   - C_Lumped(i)*(dreact(r,1,2)-dreact
                    (r,2,2))*products(r,4)*obj.Lumped(r);
                dqCL_dT(i,i)    = dqCL_dT(i,i)  - C_Lumped(i)*(dreact(r,1,2)-dreact
                    (r,2,2))*products(r,5)*obj.Lumped(r);

                dqT_dCL(i,i)    = dqT_dCL(i,i)  - C_Lumped(i)*(dreact(r,1,3)-dreact
                    (r,2,3))*products(r,4)*obj.Lumped(r);
                dqCL_dCL(i,i)   = dqCL_dCL(i,i) - C_Lumped(i)*(dreact(r,1,3)-dreact
                    (r,2,3))*products(r,5)*obj.Lumped(r);
            end
        end
```

where first the nodal reaction rates are determined, after which these reaction rates are added to their respective locations within the force vector, and tangential matrix.

4. Post-Processing and sample results

Within the simulation code, data files are written at regular intervals. These files can be used for post-processing of the data, and additionally allow for simulations to be resumed from the point when the file was saved. For post-processing, an example is provided within PostProcessing.m, specifically plotting temporal data for the interstitial lattice hydrogen concentration:

PostProcessing.m

```
f1 = figure;
    yyaxis left

    % Average concentrations are saved within CL_vec
    plot(tvec/3600/24,CL_vec,'LineWidth',2, 'DisplayName', fileTitle);
    hold on
    xlabel('$time \;[\mathrm{days}]$','Interpreter','latex')
    ylabel('$\overline{C_L} \;[\mathrm{mol}/\mathrm{m}^3]$','Interpreter','latex')
```

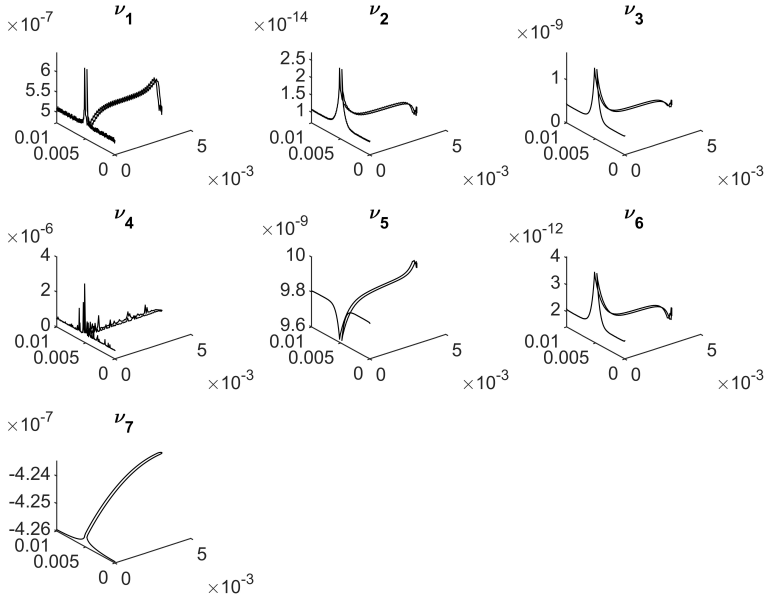

Figure 3: Example of reaction rates plotted during the post-processing procedure.

```
    yyaxis right

    %maximum concentrations saved within Cmax_vec
    plot(tvec/3600/24,Cmax_vec,'LineWidth',2, 'DisplayName', fileTitle);
    ylabel('$\tilde{C_L} \;[\mathrm{mol}/\mathrm{m}^3]$','Interpreter','latex')

    savefig(f1, "Figures/HydrogenOverTime")
```

where the vector tvec contains the time (in seconds) at the end of each time step, the vector CL_vec the volume-averaged hydrogen concentration (in mol/m^3), and the vector Cmax_vec the maximum hydrogen concentration (also in mol/m^3). This results in the figure shown in Fig. 2.

Next, the nodal values of the interstitial lattice hydrogen concentration and pH are plotted as a surface, through the functions:

```
    physics.PlotNodal("CL",-1, "Metal");


    physics.models{7}.plotpH(physics);
```

with more formatting performed by surrounding functions. This results in Fig. 1. This figure shows the pH at the metal surface increasing due to the hydrogen being adsorbed onto the metal surface. Further into the electrolyte, the metal ions react to produce hydrogen as by-product, reducing the pH.

Reaction rates can be visualised through:

```
f3 = figure;
    physics.models{9}.plotReactions(physics);
    savefig(f3, "Figures/ReactionRates")
```

, resulting in Fig. 3. One thing to note is that this plots the reaction rates based on the *integration point* values. As such, small oscillations within these rates are not detrimental for the numerical scheme. The presence of non-physical oscillations can be more directly judges through, for instance, the surface hydrogen coverage, produced by:

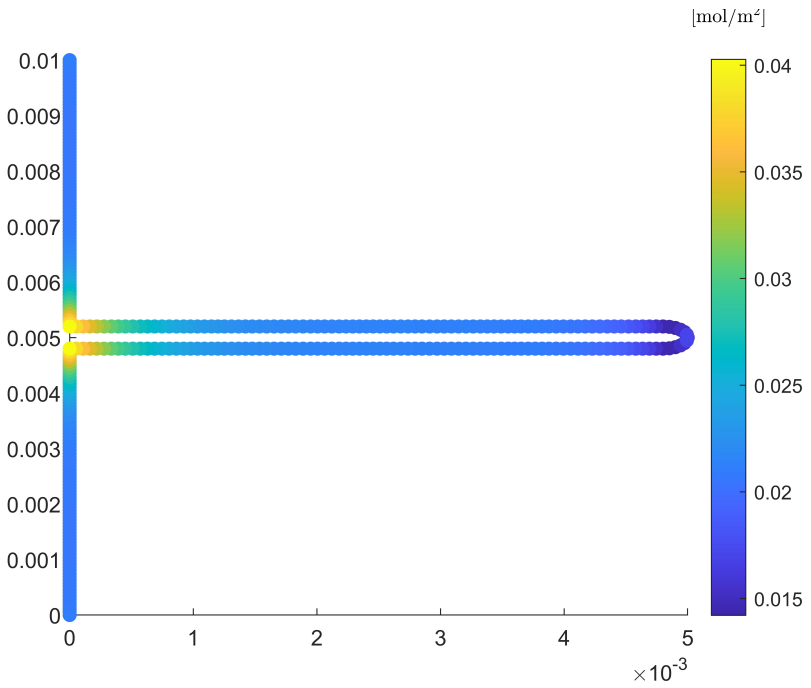

Figure 4: Example of surface coverage plotted during the post-processing procedure.

```
f4 = figure;
    physics.PlotNodal("Theta",-1, "Interface");
    cb = colorbar;
    cb.Title.String = {'$\theta$', '[$\mathrm{mol}/\mathrm{m}^2$]', ' '};
    cb.Title.Interpreter='latex';
    savefig(f4, "Figures/SurfaceCoverage")
```

and shown in Fig. 4, indicating the state of the surface is oscillation-free.