Polarization Curve Modeling with Galvanic Couple Demonstration

Steven A. Policastro[[1]](#footnote-1), Ph.D.

Center for Corrosion Science and Engineering, U.S. Naval Research Laboratory 4555 Overlook Avenue SW Washington, DC 20375

This document serves as a summary and README for the set of physics-based and response surface models that are used to calculate simplified polarization curves for several alloy systems.

Polarization curves can be obtained across a range of temperatures,  concentrations, pH values, and electrolyte flow velocities. Note, though, that the simplified nature of the polarization curves arises becuase these are intended to serve as boundary conditions for FEM of complex parts and components exposed to corrosive environments. The BCs on the electrode surfaces in those calculations are restricted to single-valued functions. Thus, observed polarization behavior that results in active-passive transitions, for example, will not be captured by these models.

All functions and classes are written in MATLAB® R2024a.

Record of Code Revisions:

* Created - October 2021
* Last revision: 27-Jul-24

# Main Function

Unlike the classes that are described in the following section, the main function is not unique. It serves as the entry point to the code for modeling a polarization curve. Whatever this function is named, however, it must accomplish the following actions:

* Clear the command window, all figures, and all variables from the workspace.
* Instantiate an instance of the Constants class.
* Define the potential region of interest
* Define the various environmental conditions for generating the polarization curves
* Iterate through the environmental conditions to construct instances of the PolarizationCurveModel class and call the plotting function for the defined polarization curves.

Two examples of the polarization curves for the same material but with different environmental conditions that can be generated are shown in Figure 1.

function polCurveMain  
 clc;  
 clear all;  
 C = Constants;  
 vRange = -1.5:0.005:0.5; %VSCE  
 alloy = {'ss316'};  
 T = [25.0,5.0]+ C.convertCtoK; %C  
 pH = [7.0,3.0];  
 cCl = [0.6,0.02]; %M  
 velocity = [5.0,5.0]; %m/s  
 envConds = [T;pH;cCl;velocity];  
  
 for i = 1:size(envConds,2)  
 aPolCurve = PolarizationCurveModel(i,alloy,vRange, envConds(:,i));  
 PolarizationCurveModel.Plot\_Polarization\_Data\_and\_Model(aPolCurve,600+i)  
 end  
end

|  |  |
| --- | --- |
| a. | b. |
|  |  |

Figure 1. a. Modeled polarization curve for a 316 stainless steel in 0.6 M NaCl solution at 25oC, pH = 7, and a flow velocity of 5 m/s. b. Modeled polarization curve for a 316 stainless steel in 0.02 M NaCl solution at 5oC, pH = 3, and a flow velocity of 5 m/s.

# Principal Sub-Functions and Classes

The principal supporting classes and sub-functions that are used in the model include the following:

* Constants - a class that defines a number of conversion values for converting units as well as a number of physical parameters for electrochemical calculations.
* PolarizationCurveModel - a class that creates an object to calculate and plot a polarization curve.

# Constants and Conversion Factors

This class contains variables storing physical constant values for use in electrochemical calculations.

*Record of Revisions*:

* Created - October 2022
* Last revision: 12-Oct-2022

classdef Constants  
 % Defines physical constants and conversion factors for use in  
 % electrochemical simulations.  
 properties (Constant = true)  
 R = 8.314; %Ideal gas constant, J/(mol K)  
 F = 96485.3; %Faraday constant, coul/mol  
 kb = 1.38e-23;%Boltzmann's constant, m2 kg/(s2 K)  
 planck\_h = 6.626e-34; %Planck's constant, m2 kg / s  
 eps0 = 8.85418782e-12; %Permittivity of free space m-3 kg-1 s4 A2  
 e = 1.6e-19; % Charge on electron, C  
 E\_SHE\_to\_SCE = 0.244; %Convert standard hydrogen eelctrode potential to saturated calomel potential  
 convertCtoK = 273.15; %Convert celsius to Kelvin  
 convertGtoKg = 1.0 / 1000.0; %Convert grams to kilograms  
 convertKgtoMg = 1000.0 \* 1000.0; %Convert kilograms to milligrams  
 convertLtoCm3 = 1000; %Convert liters to cm3 (or mL)  
 convertMtoCm = 100; %Convert meters to cm  
 convertCm2toM2 = 1.0e-4; %Convert cm2 to m2  
 cH2O = 55.55; %Concentration of water, mol/L  
 cO2 = 0.209476; %Concentration of oxygen in air, %  
 M\_H2 = 2.016; %Molar mass of H2, g/mol  
 M\_OH = 17.008; %Molar mass of OH-, g/mol  
 M\_O2 = 32.0; %Molar mass of O2, g/mol  
 M\_H2O = 18.01528; %Molar mass of H2O, g/mol  
 M\_Cl = 35.5; %Molar mass of Cl-, g/mol  
 M\_NaCl = 58.4; %Molar mass of NaCl, g/mol  
 M\_Cr = 51.9961; %Molar mass of Cr, g/mol  
 MCr2O3 = 151.99; %g/mol  
 densityCr2O3 = 5.22; %g/cm3  
 rhoCr2O3\_0 = 5000.0e9; %Ohm/cm  
 iPassCr2O3 = 1.0e-6; %A/cm2  
 tCr2O3film = 2.5e-7; %cm  
 M\_Fe = 55.845; %Molar mass of Fe, g/mol  
 M\_Ni = 58.6934; %Molar mass of Ni, g/mol  
 M\_Cu = 63.546;%Molar mass of Cu, g/mol  
 D\_H = 9.311e-5; %Diffusivity of H3O-, cm2/sec  
 D\_H2O = 2.299e-5; %Diffusivity of H2O, cm2/sec  
 D\_Fe = 2.5e-12; %Diffusivity of Fe in oxide, cm2/sec  
  
 epsH2O = 80.1; %Dielectric constant of water  
 epsPolyurethane = 6.19; %Dielectric constant of polyurethane  
 epsEpoxy = 3.6; %Dielectric constant of epoxy  
 e0\_orr\_acid = 1.223; %Thermodynamic electrode potential of ORR in acidic solution, V\_SHE  
 e0\_orr\_2e\_alk = -0.065; %Thermodynamic electrode potential of ORR in acidic solution, V\_SHE  
 e0\_orr\_alk = 0.401; %Thermodynamic electrode potential of ORR in neutral and alkaline solutions, V\_SHE  
 e0\_her\_alk = -0.83; %Thermodynamic electrode potential of HER in neutral and alkaline solutions, V\_SHE  
 e0\_her\_acid = 0.0; %Thermodynamic electrode potential of HER in acidic solution , V\_SHE  
 e0\_me\_ox = 0.0; %Thermodynamic electrode potential of generic metal, V\_SHE  
 e0\_Cr\_ox = -0.74; %Thermodynamic electrode potential of Cr oxidation, V\_SHE  
 e0\_Fe\_ox = -0.501; %-0.41; %Thermodynamic electrode potential of Fe oxidation, V\_SHE  
 e0\_Ni\_ox = -0.23; %Thermodynamic electrode potential of Ni oxidation, V\_SHE  
 e0\_Cu\_ox = 0.52; %Thermodynamic electrode potential of Ni oxidation, V\_SHE  
 z\_orr = 4; %Number of electrons exchanged in the ORR reaction  
 z\_her = 2; %Number of electrons exchanged in the HER reaction  
 z\_Cr\_ox = 3; %Number of electrons exchanged in the Cr oxidation reaction  
 z\_Fe\_ox = 2; %Number of electrons exchanged in the Fe oxidation reaction  
 z\_Cu\_ox = 1; %Number of electrons exchanged in the Fe oxidation reaction  
 z\_Fe\_red = 1; %Number of electrons exchanged in an Fe reduction reaction  
 z\_Ni\_ox = 2; %Number of electrons exchanged in the Ni oxidation reaction  
 VO2 = 22.414; %Molar volume of 1 mol of O2, L/mol  
 VNaCl = 16.6; %Molar volume of 1 mol NaCl, L/mol  
 % ==========================  
 end  
  
 methods  
 function obj = Constants()  
 %Constants Construct an instance of this class  
 % Constructor of the constants class that takes no arguments  
 end  
 end  
  
 methods (Static = true)  
 function [cH,cOH] = calculatecHandcOH(pH)  
 %calculatecHandcOH calculate cH and cOH  
 % Function that returns the concentration of H+ and OH-  
 % ions in solution, based on the input pH value.  
 cH = 10.0^-(pH); %mol/L  
 cOH = 10.0^-(14.0-pH); %mol/L  
 end  
  
 function y = LinearLinear(b,x)  
 %LinearLinear returns the solution of a linear-linear rational  
 % Function which, for a set of 3 parameters, b(1), b(2), and,  
 % b(3), and a domain, x, returns a range, y, for a  
 % linear-linear rational function.  
 num = b(1) + b(2).\*x;  
 denom = 1.0 + b(3).\*x;  
 y = num./denom;  
 end  
  
 end  
  
end

# Polarization Curve Class

This class serves as the template to create an object that can calculate and plot a polarization curve.

Alloys specifically included in this version of the code:

* HY 80
* HY 100
* SS 316
* Ti
* CuNi (Wrought)
* I625

**Note** This version of the code contains estimates for reaction properties for the alloys specified above. If additional alloys are to be considered, their model reaction properties must be entered below, or this version of the code must be changed so the values can be obtained elsewhere. *Record of Revisions*:

* Created - October 2021
* Last revision: 30-Jul-2024

## Class definition

classdef PolarizationCurveModel  
% PolarizationCurveModel - Class for instantiating, defining, and plotting  
% a model polarization curve for a specified alloy in specified exposure  
% conditions.

## Internal class properties

These properties are not accessible outside the class.

properties (SetAccess = private)  
 tick\_label\_size = 16;  
 axis\_label\_size = 18;  
 title\_label\_size = 20;  
 axis\_line\_width = 3;  
 font\_weight = 'bold';  
 plot\_line\_width = 3;  
 plot\_line\_width\_2 = 2;  
 outOfRangeVal = -1000;  
 min2Deriv = 0.001; %0.004;  
 offsetPeak = 900;  
 cutOffPeak1Left = 0.02;  
 cutOffPeak1Right = 0.5;  
 minFilteredCurrentForORRPeak = 1.0e-6;  
 minFileredCurrentForFeRedPeak = 1.0e-8;  
 minActivationHERCurrent = 5.0e-5;  
 lineTypes = {'--','-.',':','-'};  
 end

## Public class properties

These properties are accessible by outside code, once an instance of this class has been initialized. These properties cover the solution properties, environmental conditions, and currents and potential values for the polarization curve calculations.

properties  
 ItemNo int16  
 Name char  
 UNSname char  
 solution naclSolutionChemistry  
 Temp double  
 cCl double  
 pH double  
 flowvelocity double  
 exposedArea double  
 allCatRxnsModel ElectrochemicalReductionReaction  
 inclCatRxnsModel reactionNames  
 allAnRxnsModel ElectrochemicalOxidationReaction  
 inclAnRxnsModel reactionNames  
 eAppModel double  
 iTotModel double  
 numCatReactions int32  
 numAnReactions int32  
 catReactsDict  
 anReactsDict  
 plotSymbol string  
 end  
 methods (Access = public)

## Object constructor method

function obj = PolarizationCurveModel(no, fName, pdata, envcon)  
 %PolarizationCurveModel - Constructor method for this class.  
 %  
 %The purpose of this method is to construct an instance of the  
 %PolarizationCurveModel class. The constructor  
 %requires:  
 % =============================================  
 % Name = char array  
 % Applied potentials = numeric array (V\_{SCE})  
 % Temperature = numeric value (^oC)  
 % Chloride concentration = numeric value ([M])  
 % pH = numeric value  
 % =============================================  
 % The constructor returns an instance of the class  
 % =============================================  
 plotCurrentContributions = false;  
 C = Constants;  
 obj.ItemNo = no;  
 obj.Name = fName;  
 obj.Temp = envcon(1);  
 obj.pH = envcon(2);  
 obj.cCl = envcon(3);  
 obj.flowvelocity = envcon(4);  
 % =============================================  
 % Setup the applied potential vector  
 % =============================================  
 obj.eAppModel = pdata;  
 % =============================================  
 % Calculate solution properties affected by chloride  
 % concentration and temperature  
 % =============================================  
 [cOH, cH] = Constants.calculatecHandcOH(obj.pH);  
 obj.solution = naclSolutionChemistry(obj.cCl,obj.Temp);  
 % =============================================  
 % Create the electrochemical reactions  
 % =============================================  
 % Create instances of corroding metal and define reaction  
 % properties  
 % =============================================  
 switch obj.Name  
 case 'ss316'  
 metal = SS316(obj.Name,obj.cCl,obj.Temp,obj.pH);  
 unsmetal = 'UNS S31600';  
 obj.UNSname = unsmetal;  
 acolor = 'b';  
 % =====  
 % ORR  
 % =====  
 cReact = [obj.solution.cO2,((obj.solution.aW/(1000))\*C.M\_H2O)^2]; %g/cm3  
 cProd = [1.0, ((cOH/1000)\*C.M\_OH)^4]; %g/cm3  
 Dcoeff = obj.solution.dO2;  
 obj.allCatRxnsModel(1) = ElectrochemicalReductionReaction(reactionNames.ORR, cReact, cProd, obj.Temp, C.z\_orr, C.e0\_orr\_alk, Dcoeff, ...  
 obj.eAppModel, metal);  
 % =====  
 % HER  
 % =====  
 cReact = [((obj.solution.aW/(1000))\*C.M\_H2O)^2, 1.0]; %g/cm3 55.55;  
 cProd = [1.0, ((cOH/1000)\*C.M\_OH)^2]; %g/cm3 10.0^-(14.0-obj.pH); %mol/L  
 Dcoeff = C.D\_H2O;  
 obj.allCatRxnsModel(2) = ElectrochemicalReductionReaction(reactionNames.HER, cReact, cProd, obj.Temp, C.z\_her, C.e0\_her\_alk, Dcoeff, ...  
 obj.eAppModel,metal);  
 % =====  
 % Passivation  
 % =====  
 cReact = 1.0;  
 cProd = 1.0e-6;  
 obj.allAnRxnsModel(1) = ElectrochemicalOxidationReaction(reactionNames.Passivation, cReact, cProd, obj.Temp, obj.eAppModel, metal);  
 % =====  
 % Pitting  
 % =====  
 cReact = 1.0;  
 cProd = 1.0e-6;  
 obj.allAnRxnsModel(2) = ElectrochemicalOxidationReaction(reactionNames.Pitting, cReact, cProd, obj.Temp, obj.eAppModel, metal);  
 case 'hy80'  
 metal = HY80(obj.Name,obj.cCl,obj.Temp,obj.pH);  
 unsmetal = 'UNS K31820';  
 obj.UNSname = unsmetal;  
 acolor = 'r';  
 % =====  
 % ORR  
 % =====  
 cReact = [obj.solution.cO2,((obj.solution.aW/(1000))\*C.M\_H2O)^2]; %g/cm3  
 cProd = [1.0, ((cOH/1000)\*C.M\_OH)^4]; %g/cm3  
 Dcoeff = obj.solution.dO2;  
 obj.allCatRxnsModel(1) = ElectrochemicalReductionReaction(reactionNames.ORR, cReact, cProd, obj.Temp, C.z\_orr, C.e0\_orr\_alk, Dcoeff, ...  
 obj.eAppModel, metal);  
 % =====  
 % HER  
 % =====  
 cReact = [((obj.solution.aW/(1000))\*C.M\_H2O)^2, 1.0]; %g/cm3 55.55;  
 cProd = [1.0, ((cOH/1000)\*C.M\_OH)^2]; %g/cm3 10.0^-(14.0-obj.pH); %mol/L  
 Dcoeff = C.D\_H2O;  
 obj.allCatRxnsModel(2) = ElectrochemicalReductionReaction(reactionNames.HER, cReact, cProd, obj.Temp, C.z\_her, C.e0\_her\_alk, Dcoeff, ...  
 obj.eAppModel,metal);  
 % =====  
 % Fe oxidation  
 % =====  
 cReact = 1.0;  
 cProd = 1.0e-6;  
 obj.allAnRxnsModel(1) = ElectrochemicalOxidationReaction(reactionNames.Fe\_Ox, cReact, cProd, obj.Temp, obj.eAppModel, metal);  
 % =====  
 % Pitting  
 % =====  
 cReact = 1.0;  
 cProd = 1.0e-6;  
 obj.allAnRxnsModel(2) = ElectrochemicalOxidationReaction(reactionNames.Pitting, cReact, cProd, obj.Temp, obj.eAppModel, metal);  
 case 'hy100'  
 metal = HY100(obj.Name,obj.cCl,obj.Temp,obj.pH);  
 unsmetal = 'UNS K32045';  
 obj.UNSname = unsmetal;  
 acolor = 'm';  
 % =====  
 % ORR  
 % =====  
 cReact = [obj.solution.cO2,((obj.solution.aW/(1000))\*C.M\_H2O)^2]; %g/cm3  
 cProd = [1.0, ((cOH/1000)\*C.M\_OH)^4]; %g/cm3  
 Dcoeff = obj.solution.dO2;  
 obj.allCatRxnsModel(1) = ElectrochemicalReductionReaction(reactionNames.ORR, cReact, cProd, obj.Temp, C.z\_orr, C.e0\_orr\_alk, Dcoeff, ...  
 obj.eAppModel, metal);  
 % =====  
 % HER  
 % =====  
 cReact = [((obj.solution.aW/(1000))\*C.M\_H2O)^2, 1.0]; %g/cm3 55.55;  
 cProd = [1.0, ((cOH/1000)\*C.M\_OH)^2]; %g/cm3 10.0^-(14.0-obj.pH); %mol/L  
 Dcoeff = C.D\_H2O;  
 obj.allCatRxnsModel(2) = ElectrochemicalReductionReaction(reactionNames.HER, cReact, cProd, obj.Temp, C.z\_her, C.e0\_her\_alk, Dcoeff, ...  
 obj.eAppModel,metal);  
 % =====  
 % Fe oxidation  
 % =====  
 cReact = 1.0;  
 cProd = 1.0e-6;  
 obj.allAnRxnsModel(1) = ElectrochemicalOxidationReaction(reactionNames.Fe\_Ox, cReact, cProd, obj.Temp, obj.eAppModel, metal);  
 % =====  
 % Pitting  
 % =====  
 cReact = 1.0;  
 cProd = 1.0e-6;  
 obj.allAnRxnsModel(2) = ElectrochemicalOxidationReaction(reactionNames.Pitting, cReact, cProd, obj.Temp, obj.eAppModel, metal);  
 case 'i625'  
 metal = I625(obj.Name,obj.cCl,obj.Temp,obj.pH);  
 unsmetal = 'UNS N06625';  
 obj.UNSname = unsmetal;  
 acolor = 'k';  
 % =====  
 % ORR  
 % =====  
 cReact = [obj.solution.cO2,((obj.solution.aW/(1000))\*C.M\_H2O)^2]; %g/cm3  
 cProd = [1.0, ((cOH/1000)\*C.M\_OH)^4]; %g/cm3  
 Dcoeff = obj.solution.dO2;  
 obj.allCatRxnsModel(1) = ElectrochemicalReductionReaction(reactionNames.ORR, cReact, cProd, obj.Temp, C.z\_orr, C.e0\_orr\_alk, Dcoeff, ...  
 obj.eAppModel, metal);  
 % =====  
 % HER  
 % =====  
 cReact = [((obj.solution.aW/(1000))\*C.M\_H2O)^2, 1.0]; %g/cm3 55.55;  
 cProd = [1.0, ((cOH/1000)\*C.M\_OH)^2]; %g/cm3 10.0^-(14.0-obj.pH); %mol/L  
 Dcoeff = C.D\_H2O;  
 obj.allCatRxnsModel(2) = ElectrochemicalReductionReaction(reactionNames.HER, cReact, cProd, obj.Temp, C.z\_her, C.e0\_her\_alk, Dcoeff, ...  
 obj.eAppModel,metal);  
 % =====  
 % Passivation  
 % =====  
 cReact = 1.0;  
 cProd = 1.0e-6;  
 obj.allAnRxnsModel(1) = ElectrochemicalOxidationReaction(reactionNames.Passivation, cReact, cProd, obj.Temp, obj.eAppModel, metal);  
 case 'ti'  
 metal = Ti(obj.Name,obj.cCl,obj.Temp,obj.pH);  
 unsmetal = 'UNS R50700';  
 obj.UNSname = unsmetal;  
 acolor = 'c';  
 % =====  
 % ORR  
 % =====  
 cReact = [obj.solution.cO2,((obj.solution.aW/(1000))\*C.M\_H2O)^2]; %g/cm3  
 cProd = [1.0, ((cOH/1000)\*C.M\_OH)^4]; %g/cm3  
 Dcoeff = obj.solution.dO2;  
 obj.allCatRxnsModel(1) = ElectrochemicalReductionReaction(reactionNames.ORR, cReact, cProd, obj.Temp, C.z\_orr, C.e0\_orr\_alk, Dcoeff, ...  
 obj.eAppModel, metal);  
 % =====  
 % HER  
 % =====  
 cReact = [((obj.solution.aW/(1000))\*C.M\_H2O)^2, 1.0]; %g/cm3 55.55;  
 cProd = [1.0, ((cOH/1000)\*C.M\_OH)^2]; %g/cm3 10.0^-(14.0-obj.pH); %mol/L  
 Dcoeff = C.D\_H2O;  
 obj.allCatRxnsModel(2) = ElectrochemicalReductionReaction(reactionNames.HER, cReact, cProd, obj.Temp, C.z\_her, C.e0\_her\_alk, Dcoeff, ...  
 obj.eAppModel,metal);  
 % =====  
 % Passivation  
 % =====  
 cReact = 1.0;  
 cProd = 1.0e-6;  
 obj.allAnRxnsModel(1) = ElectrochemicalOxidationReaction(reactionNames.Passivation, cReact, cProd, obj.Temp, obj.eAppModel, metal);  
 case 'cuni'  
 metal = CuNi(obj.Name,obj.cCl,obj.Temp,obj.pH,obj.flowvelocity);  
 unsmetal = 'UNS C71500';  
 obj.UNSname = unsmetal;  
 acolor = 'g';  
 % =====  
 % ORR  
 % =====  
 cReact = [obj.solution.cO2,((obj.solution.aW/(1000))\*C.M\_H2O)^2]; %g/cm3  
 cProd = [1.0, ((cOH/1000)\*C.M\_OH)^4]; %g/cm3  
 Dcoeff = obj.solution.dO2;  
 obj.allCatRxnsModel(1) = ElectrochemicalReductionReaction(reactionNames.ORR, cReact, cProd, obj.Temp, C.z\_orr, C.e0\_orr\_alk, Dcoeff, ...  
 obj.eAppModel, metal);  
 % =====  
 % HER  
 % =====  
 cReact = [((obj.solution.aW/(1000))\*C.M\_H2O)^2, 1.0]; %g/cm3 55.55;  
 cProd = [1.0, ((cOH/1000)\*C.M\_OH)^2]; %g/cm3 10.0^-(14.0-obj.pH); %mol/L  
 Dcoeff = C.D\_H2O;  
 obj.allCatRxnsModel(2) = ElectrochemicalReductionReaction(reactionNames.HER, cReact, cProd, obj.Temp, C.z\_her, C.e0\_her\_alk, Dcoeff, ...  
 obj.eAppModel,metal);  
 % =====  
 % Cu oxidation  
 % =====  
 cReact = 1.0;  
 cProd = 1.0e-1;  
 obj.allAnRxnsModel(1) = ElectrochemicalOxidationReaction(reactionNames.Cu\_Ox, cReact, cProd, obj.Temp, obj.eAppModel, metal);  
 end  
 obj.plotSymbol = strcat(char(obj.lineTypes{obj.ItemNo}),acolor);  
 if plotCurrentContributions == true  
 obj.PlotReactionCurrents();  
 end  
 obj.iTotModel = zeros(size(obj.eAppModel));  
 for j = 1:numel(obj.allCatRxnsModel)  
 obj.iTotModel = obj.iTotModel + obj.allCatRxnsModel(j).i;  
 end  
 for j = 1:numel(obj.allAnRxnsModel)  
 obj.iTotModel = obj.iTotModel + obj.allAnRxnsModel(j).i;  
 end  
 outXML = true;  
 if outXML  
 dNode = 'PolarizationCurve';  
 instNode.name = 'NRL';  
 instNode.city = 'Washington';  
 instNode.state = 'DC';  
 instNode.country = 'USA';  
  
 metalNode.code = unsmetal;  
 metalNode.name = obj.Name;  
 metalNode.surf = '600 grit SiC';  
 metalNode.area = 1.0/(100\*100);  
  
 electNode.clconc = obj.cCl;  
 electNode.temp = obj.Temp;  
 electNode.pH = obj.pH;  
 electNode.o2 = obj.solution.cO2;  
 electNode.sigma = 1.0/obj.solution.rhoNaCl;  
 electNode.flow = 0.25;  
 electNode.s2 = 0.08;  
  
 metalNode.N = numel(obj.eAppModel);  
 dataNode(metalNode.N).ianode = 0.0;  
 dataNode(metalNode.N).iorr = 0.0;  
  
 dataNode(metalNode.N).iher = 0.0;  
 dataNode(metalNode.N).itot = 0.0;  
 dataNode(metalNode.N).v = 0.0;  
 for j = 1:metalNode.N  
 if numel(obj.allAnRxnsModel) > 1  
 dataNode(j).ianode = abs(obj.allAnRxnsModel(1).i(j)) + abs(obj.allAnRxnsModel(2).i(j));  
 else  
 dataNode(j).ianode = abs(obj.allAnRxnsModel(1).i(j));  
 end  
 dataNode(j).iorr = obj.allCatRxnsModel(1).i(j); % + obj.allCatRxnsModel(2).i(j)  
 dataNode(j).iher = obj.allCatRxnsModel(2).i(j);  
 dataNode(j).itot = dataNode(j).iorr + dataNode(j).iher;  
 dataNode(j).v = obj.eAppModel(j);  
 end  
 fn = strcat(obj.Name,'.xml');  
 outputxml(dNode,instNode,metalNode,electNode,dataNode,fn)  
 end  
 end

## Plotting method during debugging when adding new materials

Internal plotting method for checking the currents arising from individual reactions. This helps in debugging because currents that are non-physical can be difficult to detect if they are much smaller than other contributions.

function PlotReactionCurrents(obj)  
 figure(500)  
 hold on  
 plot(abs(obj.allAnRxnsModel(1).i),obj.eAppModel,'-k','LineWidth', obj.plot\_line\_width-2)  
 plot(abs(obj.allAnRxnsModel(2).i),obj.eAppModel,'-.b','LineWidth', obj.plot\_line\_width-2)  
 plot(abs(obj.allCatRxnsModel(1).i),obj.eAppModel,'--r','LineWidth', obj.plot\_line\_width-2)  
 plot(abs(obj.allCatRxnsModel(2).i),obj.eAppModel,'-.g','LineWidth', obj.plot\_line\_width-2)  
 axis square  
 box on  
 ylim([-1.4,0.0])  
 xlim([1.0e-13,0.01])  
 xlabel('Current density (A/cm^2)', 'FontSize', obj.axis\_label\_size,'FontWeight',obj.font\_weight)  
 ylabel('Potential (V\_{SCE})', 'FontSize', obj.axis\_label\_size,'FontWeight',obj.font\_weight)  
 ax = gca;  
 ax.XScale = 'log';  
 ax.FontSize = obj.tick\_label\_size;  
 ax.FontWeight = obj.font\_weight;  
 ax.LineWidth = obj.axis\_line\_width;  
 ax.XTick = [1.0e-13,1.0e-12,1.0e-11,1.0e-10,1.0e-9,1.0e-8,1.0e-7,1.0e-6,1.0e-5,1.0e-4,1.0e-3,0.01,0.1];  
 ax.YTick = -1.4:0.2:0.1;  
 ax.XMinorTick = 'on';  
 ax.YMinorTick = 'on';  
 legend('passive','pit','ORR','HER')  
 legend boxoff  
 hold off  
 end  
 end

## Public static methods

These methods, for plotting figures and file outputs, do not require an instantiated class object to exist.

methods (Static)  
  
 function Plot\_Polarization\_Data\_and\_Model(obj,fig\_num) %,aD  
 sName = strcat(char(obj.UNSname), ', T =',num2str(obj.Temp),'K, [Cl^-] = ',num2str(obj.cCl),'M, pH = ',num2str(obj.pH));  
 Plot\_name = strcat(sName,'\_',num2str(fig\_num));  
 h1 = figure(fig\_num);  
 set(h1,'Position', [10 20 400 400])  
 hold on  
 % Plot the average polarization data as a thin black dashed  
 % line  
 plot(abs(obj.iTotModel(1:length(obj.eAppModel))),obj.eAppModel,obj.plotSymbol,'LineWidth', obj.plot\_line\_width-1)  
 axis square  
 box on  
 ylim([-1.5,0.5])  
 xlim([1.0e-11,1.0e5])  
 xlabel('Current density (A/cm^2)', 'FontSize', obj.axis\_label\_size,'FontWeight',obj.font\_weight)  
 ylabel('Potential (V\_{SCE})', 'FontSize', obj.axis\_label\_size,'FontWeight',obj.font\_weight)  
 ax = gca;  
 ax.XScale = 'log';  
 ax.FontSize = obj.tick\_label\_size;  
 ax.FontWeight = obj.font\_weight;  
 ax.LineWidth = obj.axis\_line\_width;  
 ax.XTick = [1.0e-11,1.0e-10,1.0e-9,1.0e-8,1.0e-7,1.0e-6,1.0e-5,1.0e-4,1.0e-3];  
 ax.YTick = -1.3:0.2:0.1;  
 ax.XMinorTick = 'on';  
 ax.YMinorTick = 'on';  
 legendString = {sName};  
 legend(legendString,'Location','best')  
 legend boxoff  
 exportgraphics(ax,strcat(char(Plot\_name),'.png'),'Resolution',300)  
 hold off  
 end  
  
 function OutputFitValues(fileName,TC,cCl,fVals)  
 if isfile(fileName)  
 delete(char(fileName));  
 end  
 writecell({'T','Cl-','dG\_Cathodic','dG\_Anodic','alpha','Diffusion\_Length'},char(fileName),'Range','A1:F1')  
 writematrix([TC,cCl],char(fileName),'Range','A2:B2')  
 writematrix(fVals,char(fileName),'Range','C2:F5')  
 end  
 end  
  
end  
%------------- END OF CODE --------------

[*Published with MATLAB® R2024a*](https://www.mathworks.com/products/matlab)

# Pipe Spool Model Demo Application

This document serves as a summary and README for the set of physics-based and response surface models that are used to calculate the galvanic interactions between 2 pipe spools composed of different materials and exposed to a user-specified electrolyte, as shown in Figure 2.

|  |
| --- |
|  |

Figure 2. Galvanic couple between pipe spools of different materials.

The pipe spools are simplified to 2-dimensional interactions by assuming axial and radial symmetry.

The polarization curves that are used for the electrode boundary conditions in this work are modeled using the functions and procedures developed in the "Polarization Curve Modeling" project.

*Record of Revisions*:

* Created - January 2023
* Last revision: 30-Jul-2024

## Main Function

This example function serves as the entry point to the code for modeling the galvanic interaction between two pipe spools. It must accomplish the following actions:

* Clear the command window, all figures, and all variables from the workspace.
* Ensure the code can find supporting classes in the respective sub-folders
* Provide paths to the raw data and any other model results (if needed)
* Instantiate a PipeLoopModel object
* Define the properties of the pipe spools and the computational cell properties
* Define the various environmental conditions for generating the polarization curves
* Instantiate a GalvanicCorrosion object
* Iterate through the environmental conditions to define and plot the defined polarization curves.

Examples of the polarization curve calculations for the different materials and the potential distribution are shown in Figure 3a and Figure 3b, respectively.

function pipeLoopAnalysis  
 clc;  
 clear all;  
 addpath('plotFunctions')  
 addpath('supportingClasses')  
 addpath('materials')  
 %==============================  
 % Some definitions  
 %==============================  
 fn = 'Pipe Loop Data Only';  
 fnP1 = 'PipeLoop4ElsycaModel';  
 ext = '.csv';  
 filename = fullfile(strcat(fn,ext));  
 PLE = PipeLoopExperiments(filename);  
 fn2 = fullfile(strcat(fnP1,ext));  
 PLM = PipeLoopModel(fn2);  
 PLM.hasMyModel = true;  
 PLM.hasElsyca = true;  
 %=============================  
 % Plots of potential along the pipes at the last time-stamp in the files  
 %=============================  
 endTimePlots = true;  
 cathodeWidth = 3.0; %m  
 anodeWidth = 3.0; %m  
 widthTotal = cathodeWidth + anodeWidth;  
 deltaDistance = 3.0e-2;%m  
 aNodes = round(anodeWidth/deltaDistance);  
 cNodes = round(cathodeWidth/deltaDistance);  
 totNodes = aNodes + cNodes;  
 electrolyteNodes = 50;  
 electrolyteHeight = electrolyteNodes \* deltaDistance;  
 edgesBC = {'neumann','neumann','neumann'};  
 vApp = 0.0;  
 aReactName = 'cuni';  
 cReactName = 'i625';  
 env = [0.6, 25.0, 9.0, 5.0];  
 gC = galvanicCorrosion(edgesBC,widthTotal,electrolyteHeight,deltaDistance,electrolyteNodes,totNodes,aNodes,cNodes,aReactName,cReactName,vApp,12,env);  
 gC.aSim.phi = galvanicCorrosion.JacobiSolver(gC.aSim);  
 [PLM.pipeLoopPotentialSimpleModel,PLM.xSimple] = PipeLoopModel.GetData(PLM,gC.aSim);  
 if endTimePlots == true  
 numPipes = size(PLE.pipeLoopsAnodic,2);  
 endTimeIndex = numel(PLE.time);  
 for i = 2:2 %1:numPipes  
 cPotsExp = PLE.pipeLoopsCathodic(1,i);  
 aPotsExp = PLE.pipeLoopsAnodic(1,i);  
 corrosionPotentialsEndTime(i,endTimeIndex,cPotsExp,aPotsExp,PLM)  
 end  
 end  
end

|  |  |
| --- | --- |
| a. | b. |
|  |  |

Figure 3. a. Individual polarization curves and combined galvanic corrosion response for the modeled I625 and wrought CuNi pipe spools. b. Comparison of the measured and modeled potential distributions for a pipe spool galvanic couple between I625 and wrought CuNi in flowing seawater.

## Principal Sub-Functions and Classes

The principal supporting classes and sub-functions that are used in the model include the following:

* PipeLoopModel - a class that contains model predictions and experimental results for comparisons.
* galvanicCorrosion - a class that contains functions for performing the potential distribution calculation for the galvanic couple between the pipe spools.

# Pipe Loop Model

This class serves as a holding center for gathering experimental data and model outcomes from files.

## Class definition

classdef PipeLoopModel < handle  
 % PipeLoopModel - Class for handling experimental data and model  
 % results.

## Public properties

These properties are accessible by outside code, once an instance of this class has been initialized. They provide hooks for accessing values extracted from model output files.

properties  
 convertInToCm = 2.54;  
 convertCmToM = 0.01;  
 nPos  
 xElsyca  
 xSimple  
 x2D  
 y2D  
 hasElsyca  
 hasMyModel  
 pipeLoopPotentialElsyca  
 pipeLoopCurrentElsyca  
 pipeLoopPotentialSimpleModel  
 pipeLoopPotentialSimpleModel2  
 pipeLoopPotentialSimpleModel3  
 end  
  
 methods

## 

## Class Constructor

This method receives a filename that contains other model outcomes that can be loaded for comparison with the simple pipe spool model.

function obj = PipeLoopModel(fn2)  
 %PipeLoopModel - Class constructor.  
 aCalc = readtable(fn2,'NumHeaderLines',1);  
 obj.nPos = size(aCalc,1);  
 halfWay = round(obj.nPos/2);  
 obj.xElsyca = aCalc.Var1- aCalc.Var1(halfWay); % (obj.nPos:-1:1)  
 obj.pipeLoopCurrentElsyca = aCalc.Var2./(100\*100); %cm2  
 obj.pipeLoopPotentialElsyca = aCalc.Var3;  
 obj.hasElsyca = false;  
 obj.hasMyModel = false;  
 end  
 end

## Public static methods

This method, for determining the potential distribution from another model file and statistics on the dimensions of the pipes, do not require an instantiated class object to exist

methods (Static)  
 function [pot,dist] = GetData(~,aSim)  
 pot = aSim.phi(:,10);  
 totdist = aSim.L;  
 npp = numel(pot);  
 dx = totdist/(npp-1);  
 halfDist = totdist/2;  
 dist = -halfDist:dx:halfDist;  
 end  
 end  
end

# Galvanic Corrosion Simulation

This class defines the computational cell and contains the static functions that perform the Symmetric Over-Relaxation (SOR) calculation to obtain the potential distribution by solving the Laplace equation in 2 dimensions.

## Class definition

classdef galvanicCorrosion  
 %galvanicCorrosion Calculates the galvanic interaction between 2 metals  
 % This class contains properties and functions to define material  
 % properties, electrochemical reactions, and a solver for the laplace  
 % equation

## Public properties

Contains only a single property of type galvCorrSim that calculates the polarization curves for the different materials in the galvanic couple.

properties  
 aSim galvCorrSim  
 end  
  
 methods

## Class Constructor

This method receives multiple parameters that are used to define the computational cell and calculates the polarization curves using the methods developed in the Polarization Curve Model project.

function obj = galvanicCorrosion(topBC,length,height,dL,eNodes,numNodes,aNodes,cNodes,aReactName,cReactName,Vapp,nF,env)  
 %galvanicCorrosion Construct an instance of this class  
 obj.aSim = galvCorrSim(topBC,length,height,dL,eNodes,numNodes,aNodes,cNodes,aReactName,cReactName,Vapp,nF,env); %  
 galvCorrSim.aTafelPolCurve(obj.aSim);  
 end  
 end

## Public static methods

These methods perform the following tasks:

methods (Static)

* JacobiSolver - Uses the Jacbi method to perform the SOR algorithm to obtain the potential distribution in 2D.

function aPhi = JacobiSolver(aSim)  
 %JacobiSolver - Uses the Jacbi method to perform the SOR  
 %algorithm to obtain the potential distribution in 2D  
 maxIter = aSim.NY\*aSim.NX;  
 maxIter2 = 15;  
 tol = 1.0e-6;  
 converged2 = false;  
  
 N = aSim.NX\*aSim.NY;  
 E = zeros(1,N);  
  
 L = 1;  
 aPhi = zeros(aSim.NY,aSim.NX);  
  
 idx = 1:1:N;  
  
 topPts = zeros(1,N);  
 leftPts = zeros(1,N);  
 rightPts = zeros(1,N);  
 botPts = zeros(1,N);  
 rowAboveBotPts = zeros(1,N);  
  
 topPts(1:aSim.NX) = idx(1:aSim.NX);  
 botPts((N-aSim.NX)+1:N) = idx((N-aSim.NX)+1:N);  
 rowAboveBotPts(1:aSim.NX) = idx((N-aSim.NX)+1:N); %idx((N-(2\*aSim.NX))+1:(N-aSim.NX));  
 leftPts(aSim.NX+1:aSim.NX:(N-(2\*aSim.NX))+1) = idx(aSim.NX+1:aSim.NX:(N-(2\*aSim.NX))+1);  
 rightPts((2\*aSim.NX):aSim.NX:N-aSim.NX) = idx((2\*aSim.NX):aSim.NX:N-aSim.NX);  
  
 intPts1 = ~(idx&topPts);  
 intPts2 = ~(idx&botPts);  
 intPts3 = ~(idx&leftPts);  
 intPts4 = ~(idx&rightPts);  
 intPtsL = intPts1&intPts2&intPts3&intPts4;  
 intPts = idx(intPtsL);  
  
 topPts = nonzeros(topPts);  
 botPts = nonzeros(botPts);  
 leftPts = nonzeros(leftPts);  
 rightPts = nonzeros(rightPts);  
 rowAboveBotPts = nonzeros(rowAboveBotPts);  
  
 % Bottom electrode  
 NB = numel(botPts);  
 NT = numel(topPts);  
 % halfway = floor(NB/2);  
 NBC = 1:aSim.NXc;  
 NBA = (aSim.NXc+1):(aSim.NXa+aSim.NXc);  
 EBL = zeros(size(NBC));  
 EBR = zeros(size(NBA));  
  
 Elp = zeros(size(leftPts));  
 Erp = zeros(size(rightPts));  
 Etop = zeros(size(topPts));  
 Eint = zeros(size(intPts));  
  
 % ===============  
 % Base  
 % ===============  
 EbotsAboveC(NBC) = aSim.eCorr;  
 EbotsAboveA(NBA-101) = aSim.eCorr;  
 EBL(NBC) = galvanicCorrosion.newtonRaphson2(EbotsAboveC,'cathodic',aSim, aSim.dy, aSim.xpos);  
 EBR(NBA) = galvanicCorrosion.newtonRaphson2(EbotsAboveA,'anodic',aSim, aSim.dy, aSim.xpos);  
  
 while L <= maxIter2 && converged2 == false  
 converged3 = false;  
 converged1 = false;  
 k = 1;  
  
 while k <= maxIter && converged1 == false  
  
 if converged3 == false  
 Ebot = E(botPts);  
 for ii = 1:10  
 Ebot(1) = (EBL(1) + E(botPts(1)-aSim.NX) + 2\*Ebot(2))/4.0;  
 for i = 2:numel(Ebot)-1  
 Eleft = Ebot(i-1);  
 Eright = Ebot(i+1);  
 if (i <= NBC(end))  
 Ebottom = EBL(i);  
 else  
 Ebottom = EBR(i);  
 end  
 Etopper = E(botPts(i)-aSim.NX);  
 Ebot(i) = (Eleft + Eright + Ebottom + Etopper)./4.0;  
 end  
  
 Ebot(end) = (EBR(end) + E(botPts(end)-aSim.NX) + 2\*Ebot(numel(Ebot)-1))/4.0;  
 end  
  
 E(botPts) = Ebot;  
 converged3 = true;  
 % figure(101)  
 % hold on  
 % plot(Ebot,'-b')  
 % hold off  
 % disp('Here!')  
 end  
  
 % ===============  
 % Sides  
 % ===============  
 % Left  
 switch aSim.bcL  
 case 'dirichlet'  
 if numel(aSim.Eapp) > 0  
 Elp = aSim.Eapp(2);  
 else  
 Elp = 0.0;  
 end  
 case 'neumann'  
 for i = 1:numel(leftPts)  
 ptNum = leftPts(i);  
 Eright = E(ptNum + 1);  
 Etopper = E(ptNum-aSim.NX);  
 Ebottom = E(ptNum+aSim.NX);  
 Elp(i) = ((2.0\*Eright) + Etopper + Ebottom)./4.0;  
 end  
 end  
 E(leftPts) = Elp;  
  
 % Right  
 switch aSim.bcR  
 case 'dirichlet'  
 if numel(aSim.Eapp) > 0  
 Erp = aSim.Eapp(3);  
 else  
 Erp = 0.0;  
 end  
 case 'neumann'  
 for i = 1:numel(rightPts)  
 ptNum = rightPts(i);  
 Eleft = E(ptNum - 1);  
 Etopper = E(ptNum-aSim.NX);  
 Ebottom = E(ptNum+aSim.NX);  
 Erp(i) = ((2.0\*Eleft) + Etopper + Ebottom)./4.0;  
 end  
 end  
 E(rightPts) = Erp;  
  
 % ===============  
 % Top  
 % ===============  
 switch aSim.bcT  
 case 'dirichlet'  
 if numel(aSim.Eapp) > 0  
 Etop = aSim.Eapp(1);  
 else  
 Etop = aSim.Eapp;  
 end  
 case 'neumann'  
 Etop(1) = (2\*E(2) + 2\*E(1+aSim.NX))/4.0;  
 Etop(NT) = (2\*E(NT-1) + 2\*E(NT+aSim.NX))/4.0;  
 for i = 2:NT-1  
 ptNum = topPts(i);  
 Eleft = E(ptNum - 1);  
 Eright = E(ptNum + 1);  
 Ebottom = E(ptNum+aSim.NX);  
 Etop(i) = (Eleft + Eright + (2.0\*Ebottom))./4.0;  
 end  
 end  
 E(topPts) = Etop;  
  
 % ===============  
 % Interior  
 % ===============  
 for i = 1:numel(Eint)  
 ptNum = intPts(i);  
 Eleft = E(ptNum-1);  
 Eright = E(ptNum+1);  
 Ebottom = E(ptNum+aSim.NX);  
 Etopper = E(ptNum-aSim.NX);  
 Eint(i) = (Eleft + Eright + Ebottom + Etopper)./4.0;  
 end  
 % EintPlot = reshape(Eint, [aSim.NX-2,aSim.NY-2]);  
 % figure(98)  
 % surf(EintPlot)  
  
 E(intPts) = Eint;  
  
 % ===============  
 % Convergence check!  
 % ===============  
 eps = abs(E(intPts) - ((E(intPts-1) + E(intPts+1) + E(intPts-aSim.NX) + E(intPts+aSim.NX))/4.0));  
 convCheck = eps(eps<=tol);  
  
 if numel(convCheck) == numel(E(intPts))  
 converged1 = true;  
 % aPhi = reshape(E, [aSim.NX,aSim.NY]);  
  
 % figure(99)  
 % surf(aPhi)  
 break;  
 end  
 k = k + 1;  
  
 end  
 if converged1 == true  
 aPhi = reshape(E, [aSim.NX,aSim.NY]);  
 % figure(98)  
 % surf(aPhi)  
 converged2 = true;  
 % if numel(epsL) == aSim.NXc && numel(epsR) == aSim.NXa  
 % converged2 = true;  
 % end  
 end  
  
 L = L + 1;  
  
 end  
  
 % figure(99)  
 % surf(aPhi)  
  
 if converged2 == true  
 aPhi = reshape(E, [aSim.NX,aSim.NY]);  
 % figure(99)  
 % surf(aPhi)  
 end  
 end

* newtonRaphson2 - Uses Newton's method to determine the electrode potential at the electrode boundary condition for a given potential at the node just outside the electrode and the Nernst potential inside the electrode.

function EBOut = newtonRaphson2(E1,rtype,galvSim,h,xpos)  
 %newtonRaphson2 - Uses Newton's method to obtain the electrode  
 %boundary condition potential  
 numIters = 30;  
 tolNR = 1.0e-3;  
 aDelta = 1.0e-2;  
 aDelta1 = 1.0e-4;  
 EBOut = E1;  
  
 switch rtype  
 case 'anodic'  
 R = galvSim.bVreactions(1).C.R;  
 T = galvSim.bVreactions(1).T;  
 RT = R\*T;  
 F = galvSim.bVreactions(1).C.F;  
 a = galvSim.bVreactions(1).aMetal.BetaMetalOxidation;  
 z = galvSim.bVreactions(1).aMetal.OxidationLevelZ;  
  
 Ec = galvSim.bVreactions(1).E0;  
 beta = RT/(a\*z\*F);  
 EBold = (1.0-aDelta)\*Ec;  
 invkappa = 1.0/galvSim.bVreactions(1).kappa;  
 basedistx = galvSim.NXc \* h;  
 for i = 1:numel(E1)  
 Rpass = 0.0;  
 R2 = invkappa/h;  
 R1 = R2 + Rpass;  
 for jdx = 1:numIters  
 if EBold > E1(i)  
 deltaE = EBold-E1(i);  
 iC = (deltaE/R1)\*aDelta1;  
 i0 = galvSim.bVreactions(1).i0;  
 logterm = log(iC/i0);  
 correction = Ec - beta\*logterm;  
 fE = correction - EBold;  
 dFdE = -beta/deltaE - 1;  
 adj = fE/dFdE;  
 EBnew = EBold - adj;  
 else  
 deltaE = E1(i)-EBold;  
 iC = (deltaE/R1)\*aDelta1;  
 i0 = galvSim.bVreactions(1).i0;  
 logterm = log(iC/i0);  
 correction = Ec - beta\*logterm;  
 fE = correction - EBold;  
 dFdE = beta/deltaE - 1;  
 adj = fE/dFdE;  
 EBnew = EBold - adj;  
 end  
 tolCheck = abs((EBnew - EBold)/EBold);  
 if tolCheck <= tolNR  
 EBOut(i) = EBnew;  
 break;  
 else  
 EBold = EBnew;  
 end  
 end  
 end  
 case 'cathodic'  
 R = galvSim.bVreactions(2).C.R;  
 T = galvSim.bVreactions(2).T;  
 RT = R\*T;  
 F = galvSim.bVreactions(2).C.F;  
 a = galvSim.bVreactions(2).aMetal.BetaORR;  
 z = galvSim.bVreactions(2).C.z\_orr;  
 Ec = galvSim.bVreactions(2).E0;  
 beta = RT/(a\*z\*F);  
 EBold = (1.0-aDelta)\*Ec;  
 invkappa = 1.0/galvSim.bVreactions(2).kappa;  
 for i = 1:numel(E1)  
 Rpass = 0.0;  
 R2 = invkappa/h;  
 R1 = R2 + Rpass;  
 for jdx = 1:numIters  
 if EBold > E1(i)  
 deltaE = EBold-E1(i);  
 iC = (deltaE/R1)\*aDelta1;  
 i0 = galvSim.bVreactions(2).i0;  
 logterm = log(iC/i0);  
 correction = Ec - beta\*logterm;  
 fE = correction - EBold;  
 dFdE = -beta/deltaE - 1;  
 adj = fE/dFdE;  
 EBnew = EBold - adj;  
 else  
 deltaE = E1(i)-EBold;  
 iC = (deltaE/R1)\*aDelta1;  
 i0 = galvSim.bVreactions(2).i0;  
 logterm = log(iC/i0);  
 correction = Ec - beta\*logterm;  
 fE = correction - EBold;  
 dFdE = beta/deltaE - 1;  
 adj = fE/dFdE;  
 EBnew = EBold - adj;  
 end  
 tolCheck = abs((EBnew - EBold)/EBold);  
 if tolCheck <= tolNR  
 EBOut(i) = EBnew;  
 break;  
 else  
 EBold = EBnew;  
 end  
 end  
 end  
 end  
  
 end

* plotPhis - Outputs contour plots of the equipotential lines throughout the depth of the electrolyte as well as cross-sections of the potential field at 2 different depths.

function plotPhis(aSim)  
 %======================================================================  
 tick\_label\_size = 16;  
 axis\_label\_size = 18;  
 title\_label\_size = 20;  
 axis\_line\_width = 3;  
 font\_weight = 'bold';  
 plot\_line\_width = 4;  
 plot\_line\_width\_2 = 2;  
 marker\_size = 8;  
 colorVec = {[0 0 1],[0 0 0.95],[0 0 0.9]};  
 colorVec1 = {[1 0 0],[0.95 0 0],[0.9 0 0]};  
 markerVec = {'bo','bo','bo','rs','rs','rs','k^','k^','k^','g<','g<','g<','c>','c>','c>'};  
 markerVec1 = {'bo','rs','k^','g<','c>','o','s','^','<','>','o','s','^','<','>'};  
 linVec = {'-b','--b','-.b','-r','--r','-.r','-k','--k', ...  
 '-.k','none','none','none','none','none'};  
 someColors = {'#006400','#00BFFF','#00FF7F','#B0C4DE','#FF7F50','#6495ED','#4682B4','#0000FF','#FFD700'};  
 %======================================================================  
  
 lengthConvert = 1.0e3; %m -> mm  
 [X,Y] = meshgrid(aSim.xpos,aSim.ypos);  
 fo = double(aSim.nFig);  
  
 %=====================================================  
 % Plot the 2D representation of the field  
 %=====================================================  
 figure(fo);  
 hold on  
 c = colorbar;  
 c.Label.String = 'Potential (V\_{SCE})';  
 contour(X.\*lengthConvert,Y.\*lengthConvert,aSim.phi + aSim.potentialOffset,20,'LineWidth', plot\_line\_width); %[-0.188 -0.146, -0.104, -0.064, 0.0, 0.064, 0.104, 0.146, 0.188]  
 ylim([0.0 aSim.H].\*lengthConvert)  
 xlabel('x (mm)', 'FontSize', axis\_label\_size,'FontWeight',font\_weight)  
 ylabel('y (mm)', 'FontSize', axis\_label\_size,'FontWeight',font\_weight)  
 ax = gca;  
 ax.ZGrid = 'off';  
 ax.XGrid = 'off';  
 ax.YGrid = 'off';  
 ax.FontName = 'Times New Roman';  
 ax.FontSize = tick\_label\_size;  
 ax.FontWeight = font\_weight;  
 ax.XAxis.LineWidth = axis\_line\_width;  
 ax.YAxis.LineWidth = axis\_line\_width;  
 box on  
 hold off  
 %=====================================================  
 xPot = aSim.phi(1,:) + aSim.potentialOffset;  
 mid = floor(aSim.NY);  
 xPot2 = aSim.phi(mid,:) + aSim.potentialOffset;  
 ylocLabels = (aSim.bV(2).E0 + aSim.potentialOffset)-0.01;  
 %=====================================================  
 % Plot cross-sections of the potential field  
 %=====================================================  
 figure(fo+1)  
 hold on  
 plot(aSim.xpos.\*lengthConvert,xPot,'-.g')  
 plot(aSim.xpos.\*lengthConvert,xPot2,'-r')  
 xline(0.0,'--k');  
 text(-5.0,ylocLabels,aSim.bV(2).reaction)  
 text(10.0,ylocLabels,aSim.bV(1).reaction)  
 xlabel('Position (mm)')  
 ylabel('Potential (V\_{SCE})')  
 ylim([aSim.bV(1).E0 + aSim.potentialOffset, aSim.bV(2).E0 + aSim.potentialOffset])  
 box on  
 % legend('depth = 2 mm')  
 % legend boxoff  
 axis square  
 hold off  
 end  
  
 end  
end

[*Published with MATLAB® R2024a*](https://www.mathworks.com/products/matlab)

1. Contact info: [steven.a.policastro.civ@us.navy.mil](mailto:steven.a.policastro.civ@us.navy.mil) [↑](#footnote-ref-1)