Tinkrete

Release alpha

Gang Li

Apr 16, 2021

CONTENTS:

Version: 0.2.2 alpha

CHAPTER

ONE

INTRODUCTION

Tinkrete is a practical life cycle deterioration modelling framework. It utilizes the field survey data and provides probabilistic predictions of the RC structure deterioration through different stages of the service life cycle. It covers various deterioration mechanisms such as membrane deterioration, concrete carbonation and chloride penetration, corrosion and cracking.

CHAPTER

TWO

MODULES

2.1 membrane module

2.1.1 Summary

A statistical model is used to predict the probability of failure for the membrane.

- The resistance is the service life of the membrane, and the load is the age in service.
- The limit-state is when the ago is greater than the service life.

The initial estimation of the service life is assumed to follow the normal distribution with a mean and standard deviation such that the manufacture-labelled service life is 95% guaranteed. The estimated distribution of service life is then calibrated to the field survey results, where the failure rate of the membrane is reported through the half-cell test. So that the model matches the field observation at a given time by the updated standard deviation. This process captures the varied uncertainty of the field service life by adjusting the service life distribution. Then, the calibrated model is used to project future deterioration. The accuracy of the calibrated model is improved when more historical data is available.

class membrane.**Membrane_Model**(*pars*)

Bases: object

calibrate(*membrane_age_field*, *membrane_failure_ratio_field*)

calibrate membrane model to field condition

Parameters

- **membrane_age_field** (*float, int*) – membrane age when membrane failure rate is surveyed
- **membrane_failure_ratio_field** (*float*) – falure rate e.g. 0.1 for 10%

Returns calibrated model

Return type memebrane model object instance

copy()

create a deepcopy

membrane_failure_with_year(*year_lis*, *plot=True*, *amplify=80*)

solve pf, beta at a list of time steps with plot option

postproc(*plot=False*)

solve pf, beta, attach R distribution with plot option

run(*t*)

attach the resistance: membrane age

membrane.**Pf_RS_special**(*R_info*, *S*, *R_distrib_type='normal'*, *plot=False*)

special case of helper_fuc.Pf_RS, here the "load" S is a number and it calculates the probability of failure Pf = P(R-S<0), given the R(resistance) and S(load) with three three methods and use method 3 if it is checked "OK" with the other two

1. crude monte carlo
2. numerical integral of g kernel fit
3. R S integral: $F_R(S)$, reliability index(beta factor) is calculated with simple 1st order g.mean()/g.std()

Parameters

- **R_info** (*tuple*) – distribution of Resistance, for this specicial case, the membrane service life. R_distrib_type='normal' -> tuple(m,s) for normal m: mean s: standard deviation other distribution form will be ignored.
- **S** (*numpy array*) – distribution of load, for this special case, the membrane age.
- **R_distrib_type** (*str, optional*) – by default 'normal'
- **plot** (*bool, optional*) – plot distribution, by default False

Returns (probability of failure, beta factor)

Return type tuple

Note: R_info only supports two-parameter normal distribution.

---

membrane.**RS_plot_special**(*model*, *ax=None*, *t_offset=0*, *amplify=1*)

plot R S distribution vertically at a time to an axis special case: S is a number.

Parameters

- **model** (*model object instance*) –
  - model.R_distrib : scipy.stats._continuous_distns, normal or beta [calculated in Pf_RS() through model.postproc()]
  - model.S : single number for this special case
- **ax** (*axes*) – subplot axis
- **t_offset** (*int, float*) – time offset to move the plot along the t-axis. default is zero
- **amplify** (*int*) – scale the height of the pdf plot

membrane.**calibrate_f**(*model_raw*, *t*, *membrane_failure_ratio_field*, *tol=1e-06*, *max_count=100*, *print_out=True*)

calibrate membrane model to field condition by finding the corresponding membrane service life std that matches the failure ratio in the field

Parameters

- **model_raw** (*model instance*) – model to be calibrated
- **t** (*int, float*) – membrane age when membrane failure rate is surveyed [year]
- **membrane_failure_ratio_field** (*float*) – falure rate e.g. 0.1 for 10%
- **tol** (*float, optional*) – optimization tolerance, by default 1e-6
- **max_count** (*int, optional*) – optimization max iteration number, by default 100

- **print_out** (*bool, optional*) – if True print out model vs field compare, by default True

Returns calibrated model

Return type memebrane model object instance

membrane.**membrane_age**(*t*)
return the membrane age as the "resistance"

Parameters **t** (*int, float*) – membrane age

Returns membrane age

Return type int, float

Note: This function is a placeholder for more complex age input

membrane.**membrane_failure_year**(*model*, *year_lis*, *plot=True*, *amplify=30*)
membrane_failure_year: run model over a list of time steps

Parameters

- **model** (*class instance*) – Membrane_model class instance
- **year_lis** (*list, array-like*) – a list of time steps
- **plot** (*bool, optional*) – if True, plot the Pf, beta, R S distribtuion, by default True
- **amplify** (*int, optional*) – the arbitray comparable size of the distribution curve, by default 80

Returns (pf list , beta list)

Return type tuple

membrane.**membrane_life**(*pars*)
calculate the mean value of the service life from the manufacture's service life label(eg. 30 years with 95% confidence) with the given standard deviation

Parameters **pars** (*pamameter object instance*) – raw pamameters pars.life_product_label_life pars.life_confidence pars.life_std

Returns service life mean value

Return type float

2.2 carbonation module

2.2.1 Summary

carbonation.**C_S**(*C_S_emi=0*)
Calculate CO2 density[kg/m^3] in the environment, it is about 350-380 ppm in the atm plus other source or sink

Parameters **C_S_emi** (*additional emission, positive or negative(sink), default is 0*) –

carbonation.**Carb_depth**(*t*, *pars*)
Master model function, calculate carbonation depth and the k constant of sqrt of time from all the parameters.

The derived parameters is also calculated within this funcion. Caution: The pars instance is mutable, so a deepcopy of the original instance should be used if the calculation is not intended for "inplace".

Parameters

- **t** (*time [year]*) –
- **pars** (*object/instance of wrapper class(empty class)*) – a wrapper of all material and environmental parameters deep-copied from the raw data

Returns out

Return type carbionation depth at the time t [mm]

Note: intermediate parameters calculated and attached to

pars k_e : environmental function [-]

k_c : execution transfer parameter [-]

account for curing measures

k_t : regression parameter [-]

R_ACC_0_inv: inverse effective carbonation resistance of concrete(accelerated) [(mm^2/year)/(kg/m^3)] eps_t : error term [-]

C_S : CO2 concentration [kg/m^3]

W_t : weather function [-]

k : constant before the sqrt of time(time[year], carbonation depth[mm]) [mm/year^0.5] typical value of k =3~4 for unit mm,year [https://www.researchgate.net/publication/272174090_Carbonation_Coefficient_of_Concrete_in_Dhaka_City]

---

class carbonation.**Carbonation_Model**(*pars*)

Bases: object

calibrate(*t*, *carb_depth_field*, *print_out=False*)

return a new model instance with calibrated param

carb_with_year(*year_lis*, *plot=True*, *amplify=80*)

copy()

postproc(*plot=False*)

run(*t*)

t[year]

carbonation.**R_ACC_0_inv**(*pars*)

Calculate R_ACC_0_inv[(mm^2/year)/(kg/m^3)], the inverse effective carbonation resistance of concrete(accelerated)

From ACC test or from existion empirical data interpolation for orientation purpose test condition: duration time = 56 days CO2 = 2.0 vol%, T =25 degC RH_ref =65

Parameters

- **pars.x_c** (*float*) – measured carbonation depth in the accelerated test[m]
- **pars.option.choose** (*bool*) – if true -> choose to use interpolation method
- **pars.option.df_R_ACC** (*pd.dataframe*) – data table for interpolate, loaded by function load_df_R_ACC, interpolated by function interp_extrap_f

Returns **out** – parameter value with sample number = N_SAMPLE(defined globally)

Return type numpy arrays

Notes

Pay special attention to the units in the source code

carbonation.**W_t**(*t*, *pars*)

Calculate weather function W_t, a parameter considering the meso-climatic conditions due to wetting events of concrete surface

Parameters

- **pars.ToW** (*time of wetness [-]*) – ToW = (days with rainfall h_Nd >= 2.5 mm per day)/365
- **pars.p_SR** (*probability of driving rain [-]*) – Vertical -> weather station Horizontal 1.0 Interior 0.0
- **exponent of regression [-] ND(0.446** (*pars.b_w;*) –
- **0.163)** –
- **param t_0** (*built-in*) –

Returns **out**

Return type numpy array

carbonation.**calibrate_f**(*model_raw*, *t*, *carb_depth_field*, *tol=1e-06*, *max_count=50*, *print_out=True*)

carb_depth_field[mm]-> find corresponding x_c(accelerated test carb depth[m]) Calibrate the carbonation model with field carbonation test data and return the new calibrated model object/instance Optimization method: searching for the best accelerated test carbonation depth x_c[m] so the model matches field data on the mean value of the carbonation depth)

Parameters

- **model_raw** (*object/instance of Carbonation_Model class, mutable, so a deepcopy will be used in this function*) –
- **t** (*float or int*) – survey time, age of the concrete[year]
- **carb_depth_field** (*numpy array*) – at time t, field carbonation depths[mm]
- **tol** (*float*) – accelerated carbonation depth x_c optimization tolerance, default is 1e-5 [mm]
- **max_count** (*int*) – maximun number of searching iteration, default is 50

Returns **out** – new calibrated model

Return type object/instance of Carbonation_Model class

carbonation.**carb_year**(*model*, *year_lis*, *plot=True*, *amplify=80*)

run model over time

carbonation.**eps_t**()

Calculate error term, eps_t[(mm^2/years)/(kg/m^3)], considering inaccuracies which occur conditionally when using the ACC test method k_t[-]

Notes

for R_ACC_0_inv[(mm^2/years)/(kg/m^3)]

carbonation.**k_c**(*pars*)
calculate k_c: execution transfer parameter [-], effect of period of curing for the accelerated carbonation test

Parameters

- **pars.t_c** (*period of curing [d]*) – constant
- **b_c** (*exponent of regression [-]*) –

  normal distribution, m: -0.567 s: 0.024

carbonation.**k_e**(*pars*)
Calculate k_e[-], environmental factor, effect of relative humidity

Parameters

- **pars.RH_ref** (*65 [%]*) –
- **g_e** (*2.5 [-]*) –
- **f_e** (*5.0 [-]*) –

carbonation.**k_t**()
Calculate test method regression parameter k_t[-]

Notes

for R_ACC_0_inv[(mm^2/years)/(kg/m^3)]

carbonation.**load_df_R_ACC**()
load the data table of the accelerated carbonation test for R_ACC interpolation.

Returns

Return type Pandas Dataframe

Notes

w/c 0.45 cemI is comparable to ACC of 3 mm.

2.3 chloride module

2.3.1 Summary

TODO: make t input vectorized

chloride.**A_t**(*t*, *pars*)
calculate A_t considering the ageing effect

Parameters

- **t** (*int, float*) – time [year]
- **pars** (*instance of param object*) – a wrapper of all material and environmental parameters deep-copied from the raw data

– **pars.concrete_type** [string] Option:

'Portland cement concrete',

'Portland fly ash cement concrete',

'Blast furnace slag cement concrete'

Returns **out** – subfunction considering the 'ageing'[-]

Return type numpy array

Note: built-in parameters

- pars.k_t : transfer parameter, k_t =1 was set for experiment [-]
- pars.t_0 : reference point of time, 0.0767 [year]

---

chloride.**C_S_0**(*pars*)

Return (surface) chloride saturation concentration C_S_0 [wt.-%/cement] caused by C_eqv [g/l]

Parameters

- **pars.C_eqv** (*float*) – calculated with by C_eqv(pars) [g/L]
- **pars.C_eqv_to_C_S_0** (*global function*) – This function is based experiment with the info of
  - binder-specific chloride-adsorption-isotherms
  - the concrete composition(cement/concrete ratio)
  - potential chloride impact C_eqv [g/L]

Returns chloride saturation concentration C_S_0 [wt.-%/cement]

Return type float

Note: The conversion function C_eqv_to_C_S_0(pars.C_eqv) is derived from experiment data of 300kg cement w/c=0.5 OPC. TODO: update to a conversion function dependent on the proportioning and cementitious material

---

chloride.**C_S_dx**(*pars*)

return the substitute chloride surface concentration, i.e. chloride content just below the advection zone.

Fick's 2nd law applies below the advection zone(depth=dx). No advection effect when dx = 0 condition considered: continuous/intermittent expsure - 'submerged','leakage', 'spray', 'splash' where C_S_dx = C_S_0. The advection depth dx is calculated in the dx() function externally.

if exposure_condition_geom_sensitive is True: the observed/empirical highest chloride content in concrete C_max is used, C_max is calculated by C_max()

Parameters

- **pars** (*object/instance of param class*) – contains material and environment parameters
- **pars.C_S_0** (*float or numpy array*) – chloride saturation concentration C_S_0 [wt.-%/cement] built-in calculation with C_S_0(pars)
- **pars.C_max** (*float*) – maximum content of chlorides within the chloride profile, [wt.-%/cement] built-in calculation with C_max(pars)

- **pars.exposure_condition** (*string*) – continuous/intermittent expsure - 'submerged','leakage', 'spray', 'splash'
- **pars.exposure_condition_geom_sensitive** (*bool*) – if True, the C_max is used instead of C_S_0

Returns C_S_dx, the substitute chloride surface concentration [wt.-%/cement]

Return type float or numpy arrays

chloride.**C_crit_param**()

return the beta distribution parameters for the critical chloride content(total chloride), C_crit [wt.-%/cement]

Returns parameters of general beta distribution (mean, std, lower_bound, upper_bound)

Return type tuple

chloride.**C_eqv**(*pars*)

Evaluate the Potential chloride impact -> equivalent chloride solution concentration, C_eqv[g/L] from the source of

1. marine or coastal and/or
2. de icing salt

It is later used to estimate the boundary condition C_S_dx of contineous exposure or NON-geometry-sensitive intermittent exposure

Parameters **pars** (*instance of the param object*) – a wrapper of all material and environmental parameters deep-copied from the raw data See Note for details

Returns C_eqv, potential chloride impact [g/L]

Return type float

Note:

1. marine or coastal

- pars.C_0_M : natural chloride content of sea water [g/l]

2. de-icing salt (hard to quantify)

- pars.C_0_R : average chloride content of the chloride contaminated water [g/l]
- pars.n : average number of salting events per year [-]
- pars.C_R_i : average amount of chloride spread within one spreading event [g/m2]
- pars.h_S_i : amount of water from rain and melted snow per spreading period [l/m2]

C_eqv is used for contineous exposure or NON-geometry-sensitive intermittent exposure. For geometry-sensitive condition(road side splash) the tested C_max() should be used.

---

chloride.**C_eqv_to_C_S_0**(*C_eqv*)

Convert solution chloride content to saturated chloride content in concrete interpolate function for 300kg cement w/c=0.5 OPC. Other empirical function should be used if available

Parameters **C_eqv** (*float*) – chloride content of the solution at the surface[g/L]

Returns saturated chloride content in concrete[wt-%/cement]

Return type float

chloride.**C_max**(*pars*)

C_max: maximum content of chlorides within the chloride profile [wt.-%/cement] calculate from empirical equations or from test data [wt.-%/concrete]

Parameters

- **pars.cement_concrete_ratio** (*float*) – cement/concrete weight ratio, used to convert [wt.-%/concrete] -> [wt.-%/cement]
- **pars.C_max_option** (*string*) – "empirical" - use empirical equation "user_input" - use user input, from test
- **pars.x_a** – "empirical" option: horizontal distance from the roadside [cm]
- **pars.x_h** – "empirical" option: height above road surface [cm]
- **pars.C_max_user_input** – "user_input" option: Experiment-tested maximum chloride content [wt.-%/concrete]

Returns **C_max** – maximum content of chlorides within the chloride profile, [wt.-%/cement]

Return type float

Note: The empirical expression should be determined for structures of different exposure or concrete mixe. A typical C_max used by default in this function is from

- location: urban and rural areas in Germany
- time of exposure of the considered structure: 5-40 years
- concrete: CEM I, w/c = 0.45 up to w/c = 0.60,

---

class chloride.**Chloride_Model**(*pars_raw*)

Bases: object

calibrate(*t*, *chloride_content_field*, *print_proc=False*, *plot=True*)

return a calibrated model with calibrate_chloride_f_group() function

Parameters

- **t** (*int, float*) – time [year]
- **chloride_content_field** (*pandas dataframe*) – containts field chloride contents at various depths [wt.-%/cement]
- **print_proc** (*bool, optional*) – if true, print the optimization process, by default False
- **plot** (*bool, optional*) – if true, plot the field vs model comparison, by default True

Returns a new calibrated model with the averaged calibrated D_RCM_0

Return type instance of Chloride_Model object

chloride_with_year(*depth*, *year_lis*, *plot=True*, *amplify=1*)

chloride_with_year runs the model for a list of time steps

Parameters

- **depth** (*float*) – depth at which the chloride concrete is calculated, x[mm]
- **year_lis** (*list*) – a list of time steps [year]
- **plot** (*bool, optional*) – if true, plot the R S curve, pf, beta with time axis, by default True

- **amplify** (*int, optional*) – a scale parameter adjusting the hight of the distribution curve, by default 80

Returns (pf list, beta list)

Return type tuple

copy()
create a deepcopy of the instance, to preserve the mutable object

postproc(*plot=False*)
postproc the solved model and attach the Pf and beta to the model object

Parameters **plot** (*bool, optional*) – if true, plot the R S curve, by default False

run(*x*, *t*)
solve the chloride content at depth x and time t :param x: depth x[mm] :type x: int, float :param t: time[year] :type t: float

chloride.**Chloride_content**(*x*, *t*, *pars*)
Chloride_content is the master model function, calculate chloride content at depth x and time t with Fick's 2nd law below the convection zone (x > dx) The derived parameters is also calculated within this funcion.

- Caution: The pars instance is mutable, so a deepcopy of the original instance should be used if the calculation is not intended for "inplace".

Parameters

- **x** (*float, int*) – depth at which chloride content C_x_t is reported [mm]
- **t** (*float, int*) – time [year]
- **pars** (*instance of param object*) – a wrapper of all material and environmental parameters deep-copied from the raw data

Returns sample of the distribution of the chloride content in concrete at a depth x (suface x=0) at time t [wt-.%/c]

Return type numpy array

Note: intermediate parameters were calculated and attached to pars

- C_0 : initial chloride content of the concrete [wt-.%/cement]
- C_S_dx : chloride content at a depth dx and a certain point of time t [wt-.%/cement]
- dx : depth of the convection zone (concrete layer, up to which the process of chloride penetration differs from Fick's 2nd law of diffusion) [mm]
- D_app : apparent coefficient of chloride diffusion through concrete [mm^2/year]
- erf : imported error function

---

chloride.**D_RCM_0**(*pars*)
Return the chloride migration coefficient from Rapid chloride migration test [m^2/s] see NT Build 492 if the test data is not available from pars, use interpolation of existion empirical data for orientation purpose Pay attention to the units output [mm^2/year], used for the model

Parameters

- **pars** (*instance of param object*) – a wrapper of all material and environmental parameters deep-copied from the raw data

- **pars.D_RCM_test** (*int or float*) – RCM test results[m^2/s], the mean value from the test is used, and standard deviation is estimated based on mean
- **pars.option.choose** (*bool*) – if true interpolation from existing data table is used
- **pars.option.df_D_RCM_0** (*pandas dataframe*) – experimental data table(cement type, and w/c eqv) for interpolation
- **pars.option.cement_type** (*string*) – select cement type for data interpolation of the df_D_RCM_0, Options: 'CEM_I_42.5_R'

  'CEM_I_42.5_R+FA'

  'CEM_I_42.5_R+SF'

  'CEM_III/B_42.5'
- **pars.option.wc_eqv** (*float*) – equivalent water cement ratio considering supplementary cementitious materials

Returns D_RCM_0_final [mm^2/year]

Return type numpy array

chloride.**D_app**(*t*, *pars*)

Calculate the apparent coefficient of chloride diffusion through concrete D_app[mm^2/year]

Parameters

- **t** (*float, int*) – time [year]
- **pars** (*instance of param object*) – a wrapper of all material and environmental parameters deep-copied from the raw data

Returns sample of the distribution of the apparent coefficient of chloride diffusion through concrete [mm^2/year]

Return type numpy array

Note: intermediate parameters calculated and attached to pars

- k_e : environmental transfer variable [-]
- D_RCM_0 : chloride migration coefficient [mm^2/year]
- k_t : transfer parameter, k_t =1 was set in A_t()[-]
- A_t : subfunction considering the 'ageing' [-]

---

chloride.**b_e**()

provide the large sample array of b_e : regression variable [K]

chloride.**calibrate_chloride_f**(*model_raw*, *x*, *t*, *chloride_content*, *tol=1e-15*, *max_count=50*, *print_out=True*, *print_proc=False*)

calibrate chloride model to field data at one depth at one time. Calibrate the chloride model with field chloride test data and return the new calibrated model object/instance Optimization method: Field chloride content at depth x and time t -> find corresponding D_RCM_0(repaid chloride migration diffusivity[m^2/s])

Parameters

- **model_raw** (*object/instance of Chloride_model class (to be calibrated)*) –
- **x** (*float*) – depth [mm]

- **t** (*: int or float*) – time [year]
- **chloride_content** (*float or int*) – field chloride_content[wt.-%/cement] at time t, depth x,
- **tol** (*float*) – D_RCM_0 optimization absolute tolerance 1e-15 [m^2/s]
- **max_count** (*int*) – maximun number of searching iteration, default is 50
- **print_out** (*bool*) – if true, print model and field chloride content
- **print_proc** (*bool*) – if turn, print optimization process. (debug message in the logger)

Returns new calibrated model

Return type instance of Chloride_Model object

Note: calibrate model to field data at three depths in calibrate_chloride_f_group() chloride_content_field[wt.-%/cement] at time t

- optimizing corresponding D_RCM_0,
- fixed C_S_dx (exposure type dependent)
- fixed dx (determined by the original model)

---

chloride.**calibrate_chloride_f_group**(*model_raw*, *t*, *chloride_content_field*, *plot=True*, *print_proc=False*)

use calibrate_chloride_f() to calibrate model to field chloride content at three or more depths, and return the new calibrated model with the averaged D_RCM_0

Parameters

- **model_raw** (*object/instance of Chloride_Model class*) – model object to be calibrated), model_raw.copy() will be used
- **chloride_content_field** (*pandas dataframe*) – containts field chloride contents at various depths [wt.-%/cement]
- **t** (*int or float*) – time [year]

Returns a new calibrated model with the averaged calibrated D_RCM_0

Return type object/instance of Chloride_model class

chloride.**chloride_year**(*model*, *depth*, *year_lis*, *plot=True*, *amplify=80*)

run model over a list of time steps

chloride.**dx**(*pars*)

return dx : advection depth [mm] dependent on the exposure conditions

chloride.**k_e**(*pars*)

Calculate k_e: environmental transfer variable [-]

Parameters **pars** (*instance of param object*) – a wrapper of all material and environmental parameters deep-copied from the raw data

- pars.T_ref : standard test temperatrue 293 [K]
- pars.T_real : temperature of the structural element [K]
- pars.b_e : regression variable [K]

Returns large sample of the distribution of k_e

Return type numpy array

chloride.**load_df_D_RCM**()
load the data table of the Rapid Chloride Migration(RCM) test for D_RCM interpolation.

Returns Data table from experiment

Return type Pandas Dataframe

2.4 corrosion module

2.4.1 Summary

corrosion.**C_f**(*T*)
C_f returns BET model parameter C sampled from a normal distribution

Parameters **T** (*float*) – temperature [K]

Note: C varies from 10 to 50. This function is not applicable for elevated temperatures

class corrosion.**Corrosion_Model**(*pars*)
Bases: object

calibrate(*field_data*)

copy()

run()
solve for icorr and the corresponding section loss rate

corrosion.**Cs_g_f**()
atmospheric O2 concentration in gas phase on the boundary [mol/m^3], converted from 20.95% by volume

corrosion.**De_O2_f**(*pars*)
calculate the O2 effective diffusivity of concrete :param pars: :type pars: instance of Param object

Returns O2 effective diffusivity of concrete

Return type float, numpy array

Notes

important intermediate Parameters

- epsilon_p : porosity of hardened cement paste,
- RH : relative humidity [-%]

Gas diffusion along the aggregate-paste interface makes up for the lack of diffusion through the aggregate particles themselves. Therefore, the value of effective diffusivity is considered herein as a function of the porosity of hardened cement paste. [TODO: add temperature dependence]

corrosion.**RH_to_WaterbyMassHCP**(*pars*)
return water content(g/g hardended cement paste) from RH in pores/environment based on w_c, cement_type, Temperature by using modified BET model

Note: Refernce: Xi, Y., Bazant, Z. P., & Jennings, H. M. (1993). Moisture Diffusion in Cementitious Materials Adsorption Isotherms.

class corrosion.**Section_loss_Model**(*pars*)

Bases: object

copy()

copy return a deep copy

postproc(*plot=False*)

calculate the Pf and beta from accumulated section loss and section loss limit

Parameters **plot** (*bool, optional*) – if true plot the R S curve, by default False

run(*t_end*)

run model to solve the accumulated section loss at t_end by using x_loss_t_fun()

Parameters **t_end** (*int, float*) – year

section_loss_with_year(*year_lis*, *plot=True*, *amplify=1*)

use x_loss_year() to report the accumulated section loss at each time step and the corresponding Pf and beta.

Parameters

- **year_lis** (*list*) – a list of time step [year]
- **plot** (*bool, optional*) – if true plot the RS pf beta with time, by default True
- **amplify** (*int, optional*) – scale factor to adjust the height of the distribution curve, by default 1

Returns (pf_list, beta_list)

Return type tuple

corrosion.**V_m_f**(*t*, *w_c*, *cement_type*)

Calculate V_m, a BET model parameter

Parameters

- **t** (*curing time/concrete age [day]*) –
- **w_c** (*water-cement ratio*) –

Returns V_m : BET model parameter

Return type numpy array

Note: ASTM C150 cement type:

Cement Type Description

Type I : Normal

Type II : Moderate Sulfate Resistance

Type II (MH) : Moderate Heat of Hydration (and Moderate Sulfate Resistance)

Type III : High Early Strength

Type IV : Low Heat Hydration

Type V : High Sulfate Resistance

---

corrosion.**WaterbyMassHCP_to_RH**(*pars*)

return RH in pores/environment from water content(g/g hardened cement paste) based on w_c, cement_type, Temperature a reverse function of RH_to_WaterbyMassHCP()

corrosion.**WaterbyMassHCP_to_theta_water**(*pars*)
convert water content from g/g hardened cement paste(HCP) to volumetric in HCP to volumetric in concrete

corrosion.**calibrate_f**(*raw_model*, *field_data*)
A placeholder function for future use. field_data: temperature, theta_water, icorr_list

corrosion.**epsilon_p_f**(*pars*)
calculate the porosity of the hardened cement paste from the concrete porosity

pars : instance of Param object

Returns

Return type float, numpy array

Note: [TODO: when the concrete porosity is not known, the calculated porosity is time dependent at young age, a function of concrete mix and t]

---

corrosion.**iL_f**(*pars*)
calculate O2 limiting current density :param pars: parameter object that contains the material properties :type pars: instance of Param object

Returns O2 limiting current density [A/m^2]

Return type float, numpy array

Note: intermidiate parameters

- z : number of charge, 4 for oxygen
- delta : thickness of diffusion layer [m]
- pars.De_O2 : diffusivity [m^2/s]
- pars.Cs_g : bulk concentration [mol/m^3]
- pars.epsilon_g : gas phase fraction

Returns iL : current density over the steel concrete interface [A/m^2]

Return type float, numpy array

corrosion.**icorr_base**(*rho*, *T*, *iL*, *d*)
calculate averaged corrosion current density over the rebar-concrete surface from resistivity, temperature, limiting current and cover thickness :param rho: resistivity [ohm.m] :type rho: float, numpy array :param T: temperature [K] :type T: float, numpy array :param iL: limiting current, oxygen diffusion [A/m^2] :type iL: float, numpy array :param d: concrete cover depth [m] :type d: float, numpy array

Returns icorr : corrosion current density, treated as uniform corrosion [A/m^2]

Return type float array

Note: reference: Pour-Ghaz, M., Isgor, O. B., & Ghods, P. (2009)The effect of temperature on the corrosion of steel in concrete. Part 1: Simulated polarization resistance tests and model development. Corrosion Science, 51(2), 415–425. https://doi.org/10.1016/j.corsci.2008.10.034 parameters from ref SI units

---

corrosion.**icorr_f**(*pars*)
A wrapper of the icorr_base() with modified parameters (resistivity rho -> volumetric water content, theta_water) by theta2rho_fun().

Parameters **pars** (*instance of Param class*) –

- pars.theta_water,
- pars.T,
- pars.iL,
- pars.d,
- pars.a,
- pars.b

Returns icorr : corrosion current density [A/m^2]

Return type float, numpy array

corrosion.**icorr_to_mmpy**(*icorr*)
icorr_to_mmpy converts icorr [A/m^2] to corrosion rate[mm/year] using Faraday's laws

Parameters **icorr** (*float*) – corrosion current density [A/m^2]

Returns corrosion rate, section loss [mm/year]

Return type float

corrosion.**k_f**(*C_mean*, *w_c*, *t*, *cement_type*)
returns BET model parameter k

corrosion.**mmpy_to_icorr**(*rate*)
mmpy_to_icorr converts corrosion rate[mm/year] to icorr [A/m^2] using Faraday's laws

Parameters **rate** (*float*) – corrosion rate, section loss [mm/year]

Returns corrosion current density [A/m^2]

Return type float

corrosion.**theta2rho_fun**(*theta_water*, *a*, *b*)
volumetric water content to resistivity, index regression function used

corrosion.**theta_water_to_WaterbyMassHCP**(*pars*)
convert water content from volumetric by concrete to volumetric in HCP to g/g inHCP a reverse function of WaterbyMassHCP_to_theta_water()

corrosion.**x_loss_t_fun**(*t_end*, *n_step*, *x_loss_rate*, *p_active_t_curve*)
x_loss_t_fun returns x_loss samples at a SINGLE given time t_tend. the samples represents distribution of all possible x_loss with different corrosion history

Parameters

- **t_end** (*int, float*) – year in which the x_loss is reported
- **n_step** (*int*) – number of time steps
- **r_corr_mean** (*float*) – averaged corrosion rate i.e. x-loss rate
- **p_active_t_curve** (*tuple*) – (t_lis_curve, pf_lis_curve)

Returns section loss at t_end year, a large sample from the distribution

Return type numpy array

corrosion.**x_loss_year**(*model*, *year_lis*, *plot=True*, *amplify=80*)
run x_loss_t_fun() function over time

2.5 cracking module

2.5.1 Summary

Input + raw section loss

Output internal stress strain through the concrete cover and the location of the crack tip critical section loss to cause crack on the cover surface

[update]

- fully vectorized func with direct masking method
- numpy func instead of "numpy.vectorize" to run faster
- random variable
- object
- open crack width
- u_p as a function of w/c
- random r_v
- P of f is not consistent between P_RS and crack_condition count. Because the R and S here are dependent: S.max() = R. Currently use crack_condition count. for example if 100% fully cracked, then R==S, Pf = 50%, wrong answer

[TODO]

- fix negative w_open (surface crack width)

class cracking.**Cracking_Model**(*pars*)
Bases: object

copy()
create a deepcopy

postproc()
calculate the crack length and surface crack rate

run(*stochastic=True*)
Solve stress and strain and crack tip location in concrete cover

cracking.**bilinear_stress_strain**(*epsilon_theta*, *f_t*, *E_0*)
returns the stress in concrete from strain using the bilinear stress-strain curve

Parameters

- **epsilon_theta** (*numpy array*) – strain[-]
- **f_t** (*numpy array*) – cracking tensile strength[MPa]
- **E_0** (*numpy array*) – modulus of elasticity[MPa]

Returns stress[MPa]

Return type numpy array

Note: TODO: modulus of elasticity reduction due to creep

cracking.**crack_width_open**(*a*, *b*, *u_st*, *f_t*, *E_0*)
calculate crack opening on the concret cover surface

Parameters

- **a** (*numpy array*) – inner radius boundary of the rust (center of rebar to rust-concrete) [m]
- **b** (*numpy array*) – outer radius boundary of the concrete (center of rebar to cover surface) [m]
- **u_st** (*numpy array*) – rust expansion(to original rebar surface) beyond the porous zone [m]
- **f_t** (*numpy array*) – ultimate tensile strength [MPa]
- **E_0** (*numpy array*) – modulus of elasticity [MPa]

Returns sample of crack opening on the concret cover surface

Return type numpy array

cracking.**solve_stress_strain_crack_deterministic**(*pars*, *number_of_points=100*)
solve the stress and strain along the polar axis using strain_stress_crack_f(). One deterministic solution is returned by the means of all input variables.

Parameters

- **pars** (*Param object instance*) – an object instance containing material properties
- **number_of_points** (*int, optional*) – number of points where the stress and strain is reported along the polar axis, by default 100

Returns

(epsilon_theta, sigma_theta, rust_thickness, crack_condition, R_c, w_open)

(strain, stress, rust thickness, crack condition code, crack front cooridnate, open crack width)

Return type tuple

cracking.**solve_stress_strain_crack_stochastic**(*pars*, *number_of_points=100*)
solve the stress and strain along the polar axis using strain_stress_crack_f(). the stochastic solution matrix is returned, where each row represents a deterministic solution

Parameters

- **pars** (*Param object instance*) – an object instance containing material properties
- **number_of_points** (*int, optional*) – number of points where the stress and strain is reported along the polar axis, by default 100

Returns

(epsilon_theta, sigma_theta, rust_thickness, crack_condition, R_c, w_open)

(strain, stress, rust thickness, crack condition code, crack front cooridnate, open crack width)

Return type tuple

cracking.**strain_f**(*r*, *a*, *b*, *u_st*, *f_t*, *E_0*, *crack_condition*)
strain_f returns the strain along the polar axis r, a<=r<=b, fully vectorized with numpy funcions

Parameters

- **r** (*2D numpy array*) – coordinate along the polar axis, a matrix with rows representing each r grid, column number is repeated values [m]
- **a** (*numpy array*) – inner radius boundary of the rust (center of rebar to rust-concrete interface) [m]
- **b** (*numpy array*) – outer radius boundary of the concrete (center of rebar to cover surface) [m]
- **u_st** (*numpy array*) – rust expansion(to original rebar surface) beyond the porous zone [m]
- **f_t** (*array*) – ultimate tensile strength [MPa]
- **E_0** (*array*) – modulus of elasticity [MPa]
- **crack_condition** (*array*) – crack_condition array [int]. Each element corresponds to the condition of each row of the matrix
  - 0 'sound cover'
  - 1 'partially cracked'
  - 2 'fully cracked'

Returns strain, epsilon_theta matrix, row is the strain along the polar axis

Return type 2D numpy array

cracking.**strain_stress_crack_f**(*r*, *r0_bar*, *x_loss*, *cover*, *f_t*, *E_0*, *w_c*, *r_v*, *plot=False*, *ax=None*)

calculate the stress, strain, crack_condition for the whole concrete cover (fully vectorized with numpy matrix funcions)

Parameters

- **r** (*2D numpy array*) – coordinate along the polar axis, a matrix with rows representing each r grid, column number is repeated values [m]
- **r0_bar** (*numpy array*) – original rebar radius [m]
- **x_loss** (*numpy array*) – section loss of the steel due to corrosion
- **cover** (*numpy array*) – concrete cover depth [m]
- **f_t** (*array*) – ultimate tensile strength [MPa]
- **E_0** (*array*) – modulus of elasticity [MPa]
- **w_c** (*float, array*) – water cement ratio
- **r_v** (*numpy array*) – expansion rate r_v ranges from 2 to 6.5 times
- **plot** (*bool, optional*) – if true, plot the stress and strain along r, by default False
- **ax** (*axis instance*) – subplot axis, by default None

Returns

(epsilon_theta, sigma_theta, rust_thickness, crack_condition, R_c, w_open)

(strain, stress, rust thickness, crack condition code, crack front cooridnate, open crack width)

Return type tuple

Note: Vectorization: r is a matrix. Other material property parameters(such as E) are 1-D arrays (to be converted to column vector in the calculation)

2.6 helper_func module

2.6.1 Summary

The helper module is designed to handle the repeated math operations that are not directly related to the mechanistic model calculation. These operations include the following

- distribution sampling from a distribution (uniform, beta)
- distribution curve fitting to data with an analytical or a numerical method
- interpolation function for data tables
- numerical integration for probability density functions
- reliability probability calculation
- statistical calculation to find mean and standard distribution ignoring not-a-number (nan).
- figure sub-plotting

helper_func.**Beta_custom**(*m*, *s*, *a*, *b*, *n_sample=100000*, *plot=False*)

Beta_custom draws samples from a general beta distribution described by mean, std and lower and upper bounds X~General Beta(a,b, loc = c, scale = d) Z~std Beta(alpha, beta)

X = c + d*Z

E(X) = c + d * E(Z)

var(X) = d^2 * var(Z)

Parameters

- **m** (*mean*) –
- **s** (*standard deviation*) –
- **a** (*lower bound, not shape param a(alpha)*) –
- **b** (*upper bound, not shape param b(beta)*) –
- **n_sample** (*int*) – sample number
- **plot** (*bool*) – default is False

Returns sample array from the distribution

Return type numpy array

helper_func.**Fit_distrib**(*s*, *fit_type='kernel'*, *plot=False*, *xlabel=''*, *title=''*, *axn=None*)

fit data to a probability distribution function(parametric or numerical) and return a continuous random variable or a random variable represented by Gaussian kernels parametric : normal numerical : Gaussian kernels

Parameters

- **s** (*array-like*) – sample data
- **fit_type** (*string*) – fit type keywords, 'kernel', 'normal'

- **plot** (*bool*) – when True, create a plot with histogram and fitted pdf curve

Returns

when parametric normal is used continuous random variable : stats.norm(loc = mu, scale = sigma)

when kernel is used Gaussian kernel random variable : (stats.gaussian_kde)

Return type instance of random variable

helper_func.**Get_mean**(*x*)
get mean ignoring nans

helper_func.**Get_std**(*x*)
get standard deviation ignoring nans

helper_func.**Hist_custom**(*S*)
plot histogram with N_SAMPLE//100 bins ignoring nans

helper_func.**Normal_custom**(*m*, *s*, *n_sample=100000*, *non_negative=False*, *plot=False*)
Sampling from a normal distribution

Parameters

- **m** (*int or float*) – mean
- **s** (*int or float*) – standard deviation
- **n_sample** (*int*) – sample number, default is a Global var N_SAMPLE
- **non_negative** (*bool*) – if true, return truncated distribution with no negatives, default is False
- **plot** (*bool*) – default is False

Returns sample array from the distribution

Return type numpy array

helper_func.**Pf_RS**(*R_info*, *S*, *R_distrib_type='normal'*, *plot=False*)
Pf_RS calculates the probability of failure Pf = P(R-S<0), given the R(resistance) and S(load) with three three methods and use method 3 if it is checked "OK" with the other two

1. crude monte carlo
2. numerical integral of g kernel fit
3. R S integral: $\int_{-\infty}^{\infty} F_R(x) f_S(x) dx$, reliability index(beta factor) is calculated with simple 1st order g.mean()/g.std()

Parameters

- **R_info** (*tuple, numpy array*) – distribution of Resistance, e.g. cover thickness, critical chloride content, tensile strength can be array or distribution parameters

  R_distrib_type='normal' -> tuple(m,s) for normal m: mean s: standard deviation

  R_distrib_type='beta' -> tuple(m,s,a,b) for (General) beta distribution m: mean, s: standard deviation a,b : lower, upper bound

  R_distrib_type='array' -> array: for not-determined distribution, will be treated numerically(R S integral is not applied)

- **S** (*numpy array*) – distribution of load, e.g. carbonation depth, chloride content, tensile stress the distribution type is calculated S is usually not determined, can vary a lot in different cases, therefore fitted with kernel
- **R_distrib_type** (*str, optional*) – 'normal', 'beta', 'array', by default 'normal'
- **plot** (*bool, optional*) – plot distribution, by default False

Returns (probability of failure, reliability index)

Return type tuple

Note: For R as arrays R S integral is not applied R S integration method: $P_f = P(R - S <= 0) = \int_{-\infty}^{\infty} f_S(y) \int_{-\infty}^{y} f_R(x)dxdy$ the dual numerical integration seems too computationally expensive, so consider fit R to analytical distribution in the future versions[TODO]

---

helper_func.**RS_plot**(*model*, *ax=None*, *t_offset=0*, *amplify=1*)

plot R S distribution vertically at a time to an axis

Parameters

- **model.R_distrib** (*scipy.stats._continuous_distns, normal or beta*) – calculated in Pf_RS() through model.postproc()
- **model.S_kde_fit** (*stats.gaussian_kde*) – calculated in Pf_RS() through model.postproc() distribution of load, e.g. carbonation depth, chloride content, tensile stress. The distrubtion type is calculated S is usually not determined, can vary a lot in different cases, therefore fitted with kernel
- **model.S** (*numpy array*) – load, e.g. carbonation depth, chloride content, tensile stress
- **ax** (*axis*) –
- **t_offset** (*time offset to move the plot along the t-axis. default is zero*) –
- **amplify** (*scale the height of the pdf plot*) –

helper_func.**dropna**(*x*)

removes nans

helper_func.**f_solve_poly2**(*a*, *b*, *c*)

find the two roots of $ax^2 + bx + c = 0$

helper_func.**find_mean**(*val*, *s*, *confidence_one_tailed=0.95*)

return the mean value of a unknown normal distribution based on the given value at a known one-tailed confidence level(default 95%)

Parameters

- **val** (*float*) – cut-off value
- **s** (*standard deviation*) –
- **confidence_one_tailed** (*confidence level*) –

Returns mean value of the unknown normal distribution

Return type float

helper_func.**find_similar_group**(*item_list*, *similar_group_size=2*)

find_similar_group finds most alike values in a list

Parameters

- **item_list** (*list*) – a list to choose from
- **similar_group_size** (*int, optional*) – number of the alike values, by default 2

Returns a sublist with alike values

Return type list

helper_func.**interp_extrap_f**(*x*, *y*, *x_find*, *plot=False*)

interpolate or extrapolate value from an array with fitted2-deg or 3-deg polynomial

Parameters

- **x** (*array-like*) – variable
- **y** (*array-like*) – function value
- **x_find** (*int or float or array-like*) – look-up x
- **plot** (*bool*) – plot curve fit and data points, default if false

Returns inter/extrapolated value(s), raise warning when extrapolation is used

Return type int or float or array-like

helper_func.**sample_integral**(*Y*, *x*)

integrate Y over x, where every Y data point is a bunch of distribution samples,

Parameters

- **Y** (*numpy array*) – 2D

  column: y data point

  row: samples for each y data point
- **x** (*numpy array*) – 1D

Returns int_y_x : integral of y over x for all sampled data

Return type numpy array

Examples

[y0_sample1, y0_sample2

y1_sample1, y1_sample2]

2.7 test_helper_func module

class test_helper_func.**TestHelperFunc**(*methodName='runTest'*)

Bases: unittest.case.TestCase

setUp()

Hook method for setting up the test fixture before exercising it.

tearDown()

Hook method for deconstructing the test fixture after testing it.

test_Beta_custom()

test_Fit_distrib()

test_Get_mean()

test_Get_std()

test_Normal_custom()

test_Pf_RS()

test_RS_plot()

test_dropna()

test_f_solve_poly2()

test_find_mean()

test_find_similar_group()

test_interp_extrap_f()

test_sample_integral()

CHAPTER

THREE

EXAMPLES

3.1 membrane module example

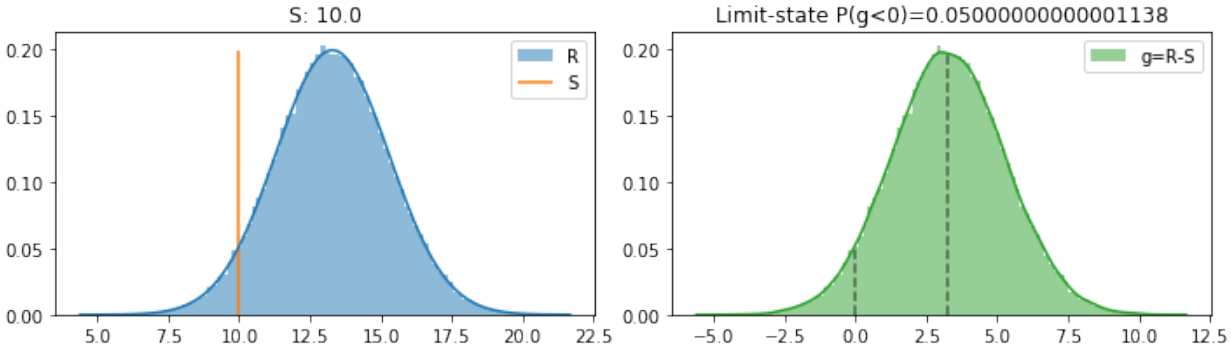
- Raw parameter data
- initialize model
- run model
- calibrate model

[18]:
```
%matplotlib inline
import numpy as np
from membrane import Membrane_Model
```

[7]:
```
# Case study Raw parameter data
class Param: pass

raw_pars = Param()

# product information
raw_pars.life_product_label_life = 10  # year, defined as 95% confident non-failure
raw_pars.life_std = 0.2 * raw_pars.life_product_label_life # assume, calibrate later
raw_pars.life_confidence = 0.95

# calibration data (if available)
# field survey result
raw_pars.membrane_failure_ratio_field = 0.01
raw_pars.membrane_age_field = 5  # [year]
```

[8]:
```
# initialize model
mem_model = Membrane_Model(raw_pars)

# run and postproc (uncalibrated)
mem_model.run(10)  # 10 years
mem_model.postproc(plot=True)
```

```
Pf(g = R-S < 0) from various methods
    sample count: 0.05058
    g integral: 0.051280490624611694
    R S integral: 0.05000000000001138
    beta_factor: 1.6378434656157241
```

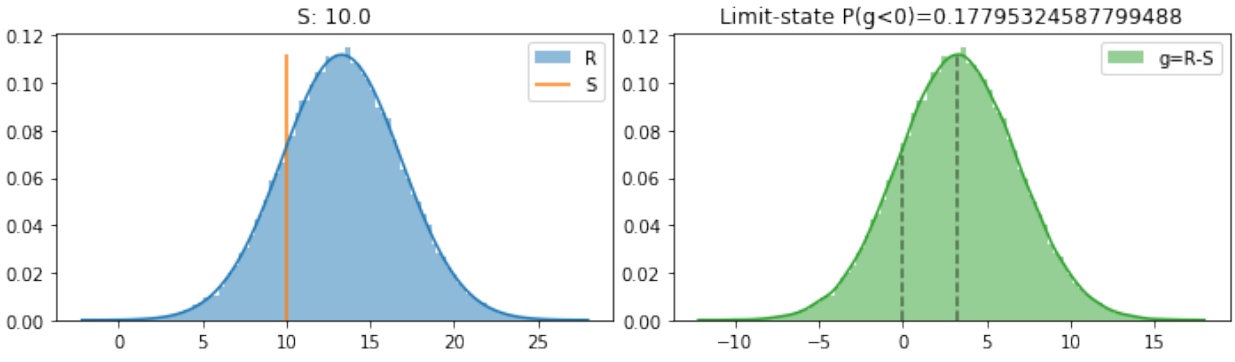


[12]:
```
# calibration to field data
mem_model_cal = mem_model.calibrate(raw_pars.membrane_age_field, raw_pars.membrane_
↪failure_ratio_field)
```

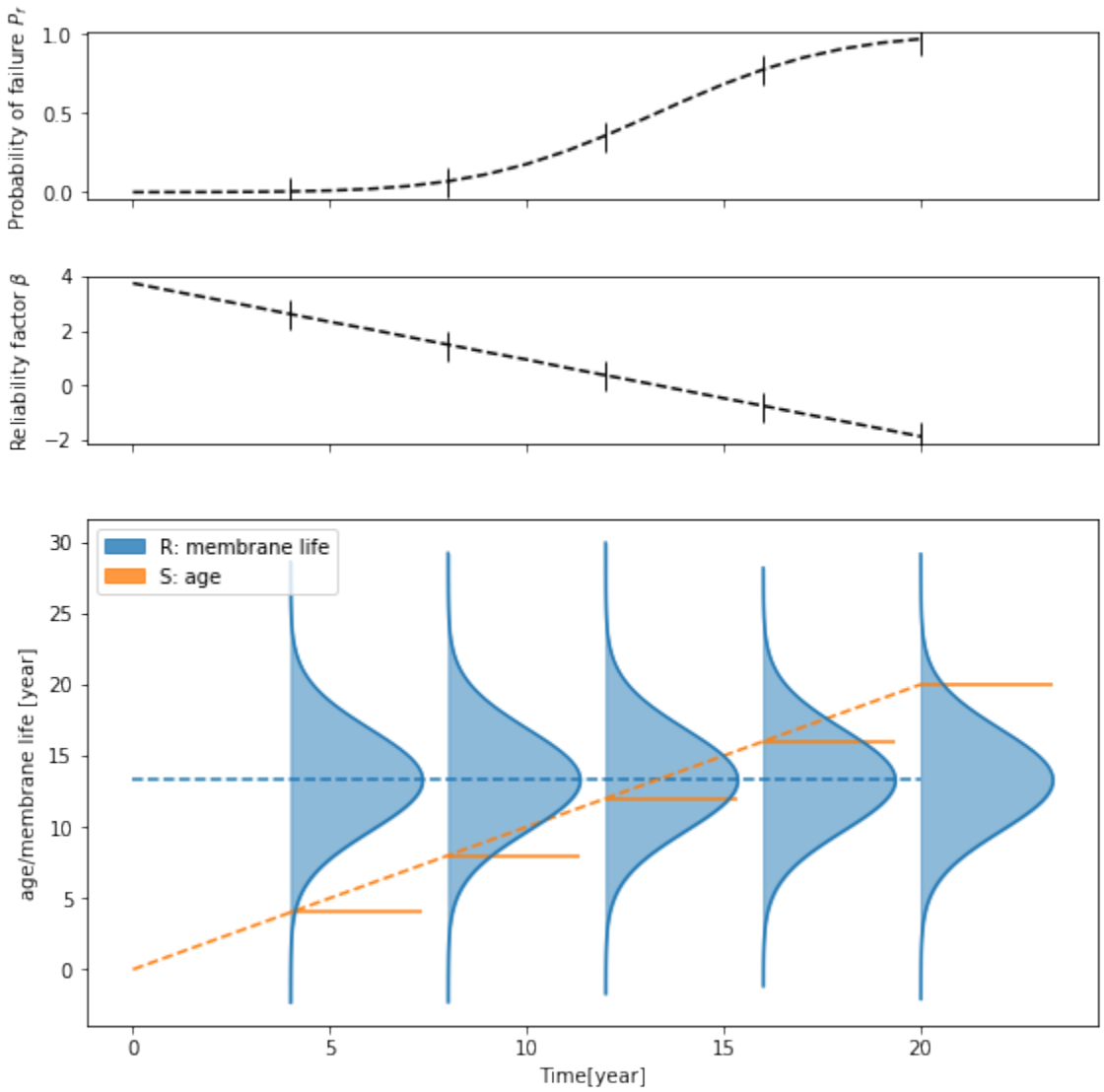
```
probability of failure:
model: 0.010000011916189768
field: 0.01
```

[13]:
```
# run and postproc (calibrated)
mem_model_cal.run(10)  # 10 years
mem_model_cal.postproc(plot=True)
```

```
Pf(g = R-S < 0) from various methods
    sample count: 0.17761
    g integral: 0.1791927385519138
    R S integral: 0.17795324587799488
    beta_factor: 0.9208084394524149
```



[22]:
```
# model with a list of time steps
t_lis = np.arange(0,21,1)
pf_lis, beta_lis = mem_model_cal.membrane_failure_with_year(year_lis=t_lis, plot=True,
↪ amplify=30)
```



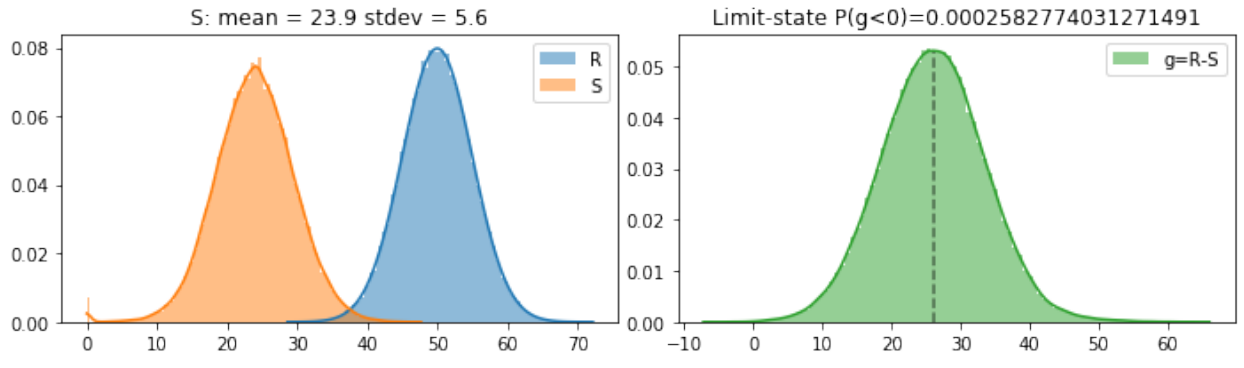
3.2 carbonation module example

- Raw parameter data
- initialize model
- run model
- calibrate model

[10]:
```
%matplotlib inline
import helper_func as hf
import numpy as np
from carbonation import Carbonation_Model, load_df_R_ACC
```

[4]:
```
# Case study

# global - Raw parameters
class Param: pass

pars = Param()

pars.cover_mean = 50  # mm
pars.cover_std = 5
pars.RH_real = 60
pars.t_c = 28
pars.x_c = 0.008  # m
pars.ToW = 2 / 52.
pars.p_SR = 0.0
pars.C_S_emi = 0.

pars.option = Param()
pars.option.choose = False
pars.option.cement_type = 'CEM_I_42.5_R+SF'
pars.option.wc_eqv = 0.6
pars.option.df_R_ACC = load_df_R_ACC()
pars.option.plot = True

# initialize model
carb_model = Carbonation_Model(pars)

# run and postproc model
carb_model.run(50)
carb_model.postproc(plot=True)
```

```
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/carbonation.py:
↪281: RuntimeWarning: divide by zero encountered in power
  W = (t_0 / t) ** ((p_SR * ToW) ** b_w / 2.0)
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/carbonation.py:76:
↪ RuntimeWarning: invalid value encountered in sqrt
  ) ** 0.5 * pars.W_t
Pf(g = R-S < 0) from various methods
    sample count: 0.0002001220744654239
    g integral: 0.00021125729406379793
    R S integral: 0.0002582774031271491
    beta_factor: 3.4595900423913237
```



[8]:
```
# calibration to field data
# field data: field carbonation after 20 years, mean=30, std=5
```

(continues on next page)

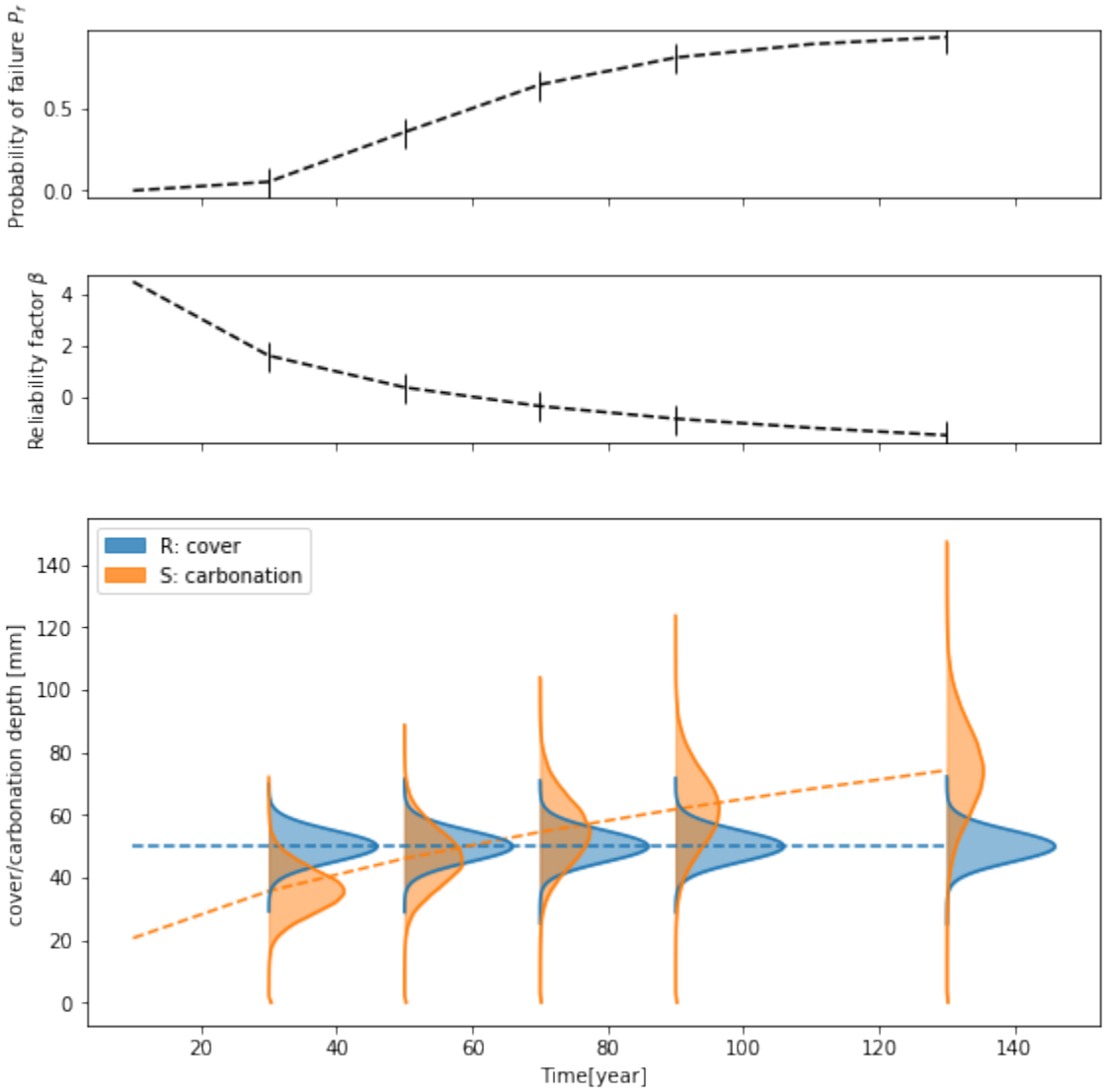
(continued from previous page)

```
carb_depth_field = hf.Normal_custom(30, 5, n_sample=12)  # mm

carb_model_cal = carb_model.calibrate(20, carb_depth_field, print_out=True)
```

```
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/carbonation.py:
↪281: RuntimeWarning: divide by zero encountered in power
  W = (t_0 / t) ** ((p_SR * ToW) ** b_w / 2.0)
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/carbonation.py:76:
↪ RuntimeWarning: invalid value encountered in sqrt
  ) ** 0.5 * pars.W_t
carb_depth:
model:
mean:29.108260004178188
std:6.106447400167045
field:
mean:29.079887643042
std:4.42009388341032
```

[15]:
```
# carbonation for a list of time steps

year_lis = np.arange(10,150,20)

pf_lis, beta_lis = carb_model_cal.carb_with_year(year_lis=year_lis, plot=True,␣
↪amplify=200)
```

```
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/carbonation.py:
↪281: RuntimeWarning: divide by zero encountered in power
  W = (t_0 / t) ** ((p_SR * ToW) ** b_w / 2.0)
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/carbonation.py:76:
↪ RuntimeWarning: invalid value encountered in sqrt
  ) ** 0.5 * pars.W_t
warning: very small Pf
```



[179]:
```
# fig.savefig('RS_time_carbonation.pdf',dpi=1200)
```

[ ]:
```
```

3.3 chloride module example

[8]:
```
%matplotlib inline
from chloride import Chloride_Model, load_df_D_RCM, C_crit_param, C_eqv_to_C_S_0
import pandas as pd
```

[4]:
```
# raw data
class Param: pass
```

(continues on next page)

(continued from previous page)

```
pars_raw = Param()

pars_raw.marine = False

# 1)marine or coastal
pars_raw.C_0_M = 18.980 # natural chloirde content of sea water [g/l]

# 2) de-icing salt (hard to quantify)
pars_raw.C_0_R = 0  # average chloride content of the chloride contaminated water [g/
↪l]
pars_raw.n = 0      # average number of salting events per year [-]
pars_raw.C_R_i = 0  # average amount of chloride spread within one spreading event [g/
↪m2]
pars_raw.h_S_i = 1  # amount of water from rain and melted snow per spreading period␣
↪[l/m2]

pars_raw.C_eqv_to_C_S_0 = C_eqv_to_C_S_0 # imported correlation function for chloride␣
↪content from soluiton to concrete

pars_raw.exposure_condition = 'splash'
pars_raw.exposure_condition_geom_sensitive = True
pars_raw.T_real = 273 + 25  # averaged ambient temperature[K]

pars_raw.x_a = 10.
pars_raw.x_h = 10.
pars_raw.D_RCM_test = 'N/A'
pars_raw.concrete_type = 'Portland cement concrete'
pars_raw.cement_concrete_ratio = 300./2400.
pars_raw.C_max_user_input = None
pars_raw.C_max_option = 'empirical'
pars_raw.C_0 = 0

pars_raw.C_crit_distrib_param = C_crit_param()  # critical chloride content import␣
↪from Chloride module 0.6 wt.% cement (mean value)

# more options
pars_raw.option = Param()
pars_raw.option.choose = True
pars_raw.option.cement_type = 'CEM_I_42.5_R+SF'
pars_raw.option.wc_eqv = 0.4    # equivalent water/binder ratio
pars_raw.option.df_D_RCM_0 = load_df_D_RCM()
```

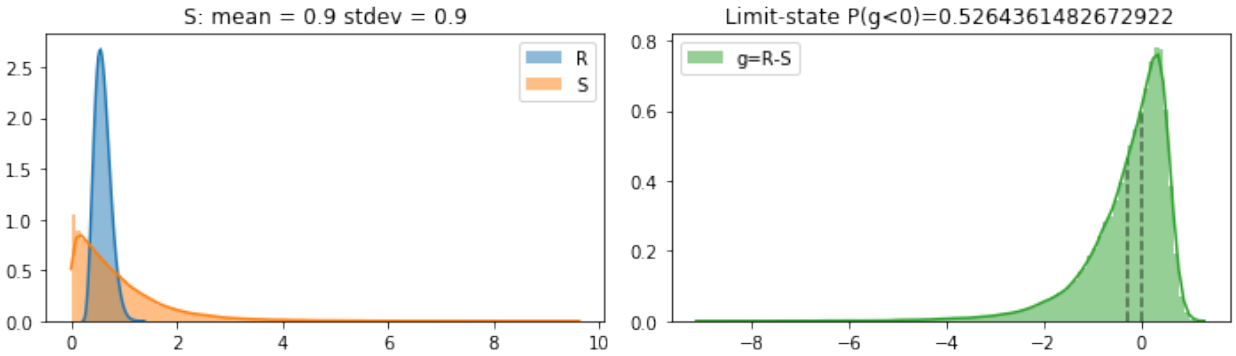
[6]:
```
# initialize model
model_cl = Chloride_Model(pars_raw)

# run for 40 mm and 10 year
model_cl.run(x = 40, t = 10)

# postproc
model_cl.postproc(plot=True)
```

```
/Users/gangli/anaconda3/lib/python3.7/site-packages/scipy/optimize/minpack.py:808:␣
↪OptimizeWarning: Covariance of the parameters could not be estimated
  category=OptimizeWarning)
Pf(g = R-S < 0) from various methods
    sample count: 0.5239
    g integral: 0.5264259274316304
```

(continues on next page)

(continued from previous page)

```
    R S integral: 0.5264361482672922
    beta_factor: -0.31087440419437906
```

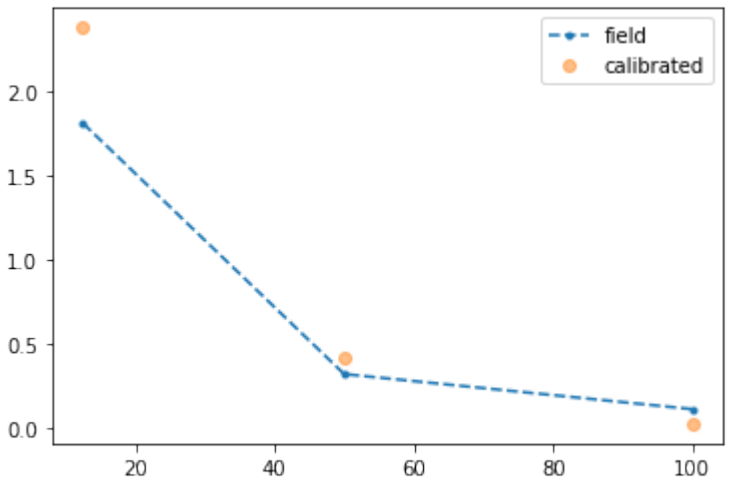


[9]:
```
# Calibration
# field data at three depth
chloride_content_field = pd.DataFrame()
chloride_content_field['depth'] = [12.5, 50, 100]  # [mm]
chloride_content_field['cl'] = np.array([0.226, 0.04, 0.014]) / pars_raw.cement_
↪concrete_ratio  # chloride_content[wt.-%/cement]
print(chloride_content_field)
```

```
   depth     cl
0   12.5  1.808
1   50.0  0.320
2  100.0  0.112
```

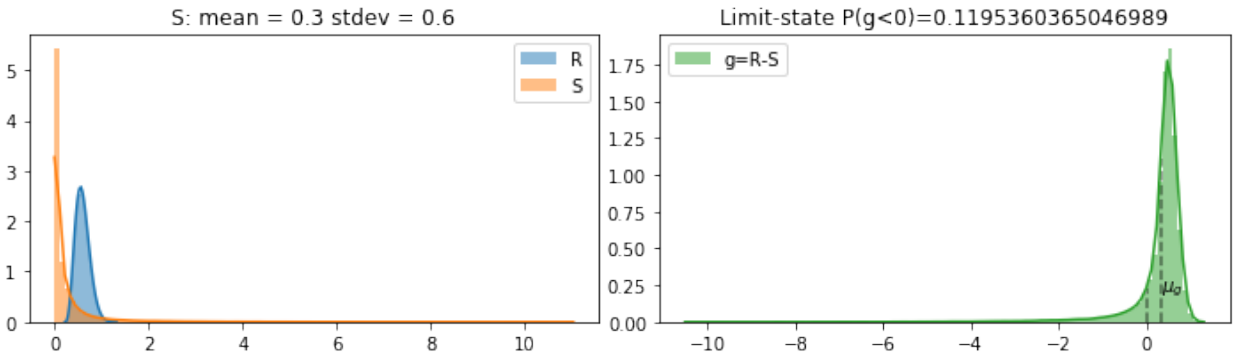
[11]:
```
#calibrate model to the field chloride content
model_cl_cal = M.calibrate(40, chloride_content_field,print_proc=False, plot=True)
```

```
2.9516601562500007e-13
1.0129394531250003e-12
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/chloride.py:66:␣
↪RuntimeWarning: invalid value encountered in sqrt
  1 - erf((x - pars.dx) / (2 * (pars.D_app * t) ** 0.5))
2.574462890625e-12
```



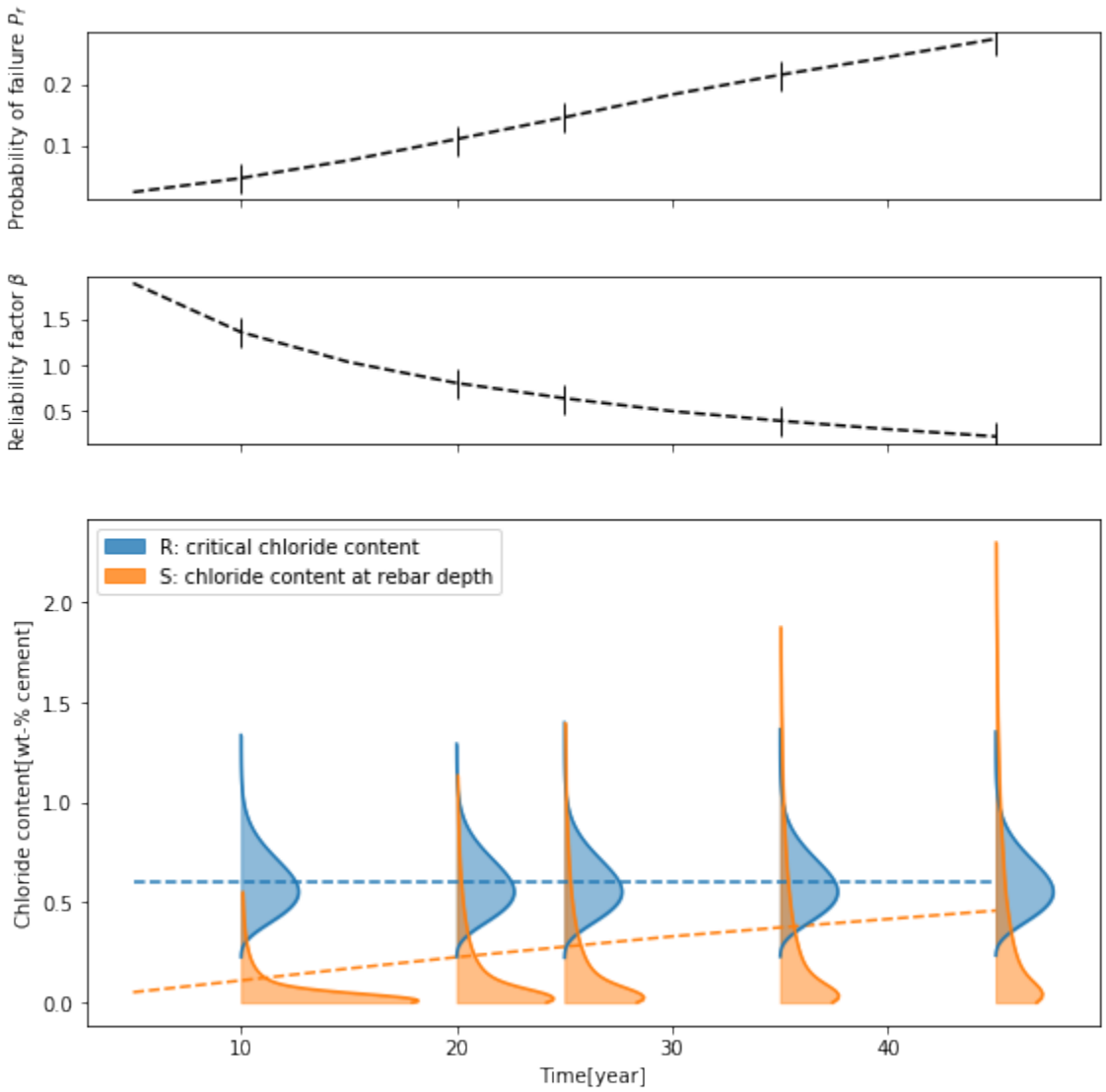
[13]:
```
# run the calibrated model for 40 mm and 10 year
model_cl_cal.run(x = 40, t = 10)
model_cl_cal.postproc(plot=True)
# plt.savefig('chloride_at_rebar_40year.pdf',dpi=1200)
```

```
Pf(g = R-S < 0) from various methods
    sample count: 0.11808
    g integral: 0.11947147378471051
    R S integral: 0.1195360365046989
    beta_factor: 0.5079907001618054
```



[19]:
```
# run model for a list of time steps
t_lis = np.arange(5,50,5)
cover = 50
pf_lis, beta_lis = model_cl_cal.chloride_with_year(depth=cover, year_lis=t_lis,
↪amplify=1)
# fig.savefig('RS_time_chloride.pdf',dpi=1200)
```

```
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/chloride.py:66:␣
↪RuntimeWarning: invalid value encountered in sqrt
  1 - erf((x - pars.dx) / (2 * (pars.D_app * t) ** 0.5))
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/helper_func.py:
↪440: IntegrationWarning: The maximum number of subdivisions (50) has been achieved.
  If increasing the limit yields no improvement it is advised to analyze
  the integrand in order to determine the difficulties.  If the position of a
  local difficulty can be determined (singularity, discontinuity) one will
  probably gain from splitting up the interval and calling the integrator
  on the subranges.  Perhaps a special-purpose integrator should be used.
  lambda x: R_distrib.cdf(x) * S_kde_fit(x)[0], 0, S_dropna.max()
```



[ ]:

3.4 corrosion module example

- Input Raw data
- moisture
- temperature
- corrosion state determined by chloride and carbonation from other modules
- Output
- icorr and corrosion rate
- accumulated sectionloss with time

[1]:
```
%matplotlib inline
import numpy as np
from corrosion import Corrosion_Model, Section_loss_Model
import helper_func as hf
import matplotlib.pyplot as plt
```

[2]:
```
class Param: pass
raw_pars = Param()

# geometry and age
raw_pars.d = 0.04  # cover depth [m]
raw_pars.t = 3650  # age[day]

# concrete composition
raw_pars.cement_type = 'Type I'
raw_pars.concrete_density = 2400 #kg/m^3
raw_pars.a_c = 2         # aggregate(fine and coarse)/cement ratio
raw_pars.w_c = 0.5       # water/cement ratio
raw_pars.rho_c= 3.1e3    # density of cement particle [kg/m^3]
raw_pars.rho_a= 2600.    # density of aggregate particle(fine and coarse) range 2400-
↪2900 [kg/m^3]


# concrete condition
raw_pars.epsilon = 0.25      # porosity of concrete
raw_pars.theta_water = 0.12 # volumetric water content
raw_pars.T = 273.15+25       # temperature [K]
```

[3]:
```
# initialize and run model
model_corr = Corrosion_Model(raw_pars)
model_corr.run()

# result
model_corr.icorr

# icorr
print(f"icorr [A/m^2]: {model_corr.icorr.mean()}")
# section loss
model_corr.x_loss_rate
print(f"section loss rate [mm/year]: {model_corr.x_loss_rate.mean()}")
```

```
icorr [A/m^2]: 0.006407370834095256
section loss rate [mm/year]: 0.007420513570849189
```

- Accumulated section loss with the increasing probability of active corrosion

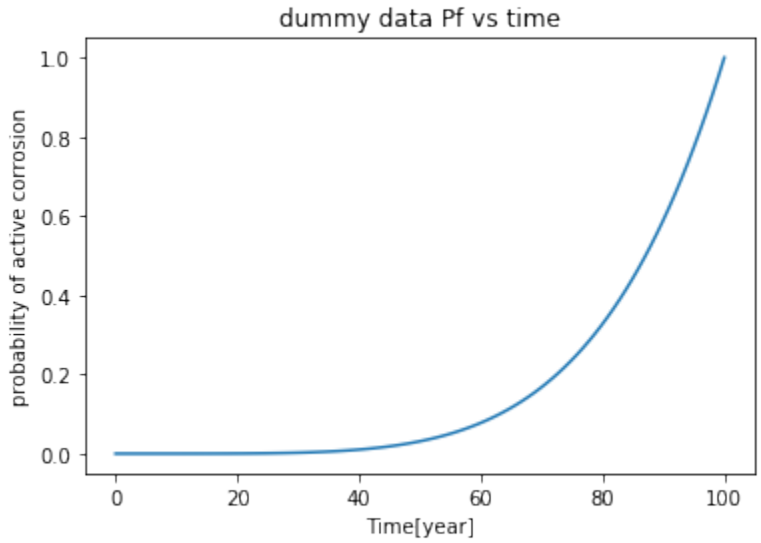
[10]:
```
# time steps
t_lis = np.linspace(0, 365*100 , 100)

# Given probability of active corrosion with time, and the section loss  (determined␣
↪by membrane, carbonation, chloride module)
# dummy data used for this example
pf_lis = np.linspace(0,1,len(t_lis))**5
plt.plot(t_lis / 365, pf_lis)
```

(continues on next page)

(continued from previous page)

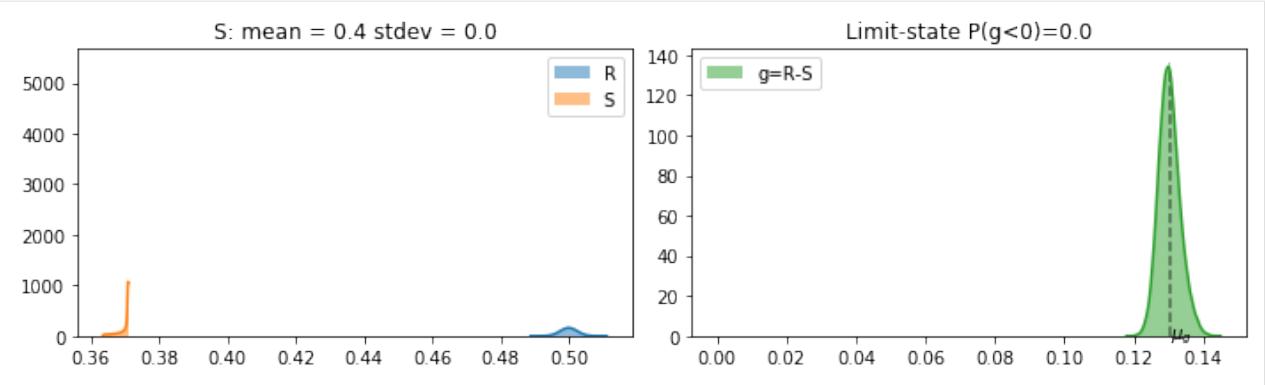
```
plt.title('dummy data Pf vs time')
plt.xlabel('Time[year]')
plt.ylabel('probability of active corrosion')
```

[10]:
```
Text(0, 0.5, 'probability of active corrosion')
```



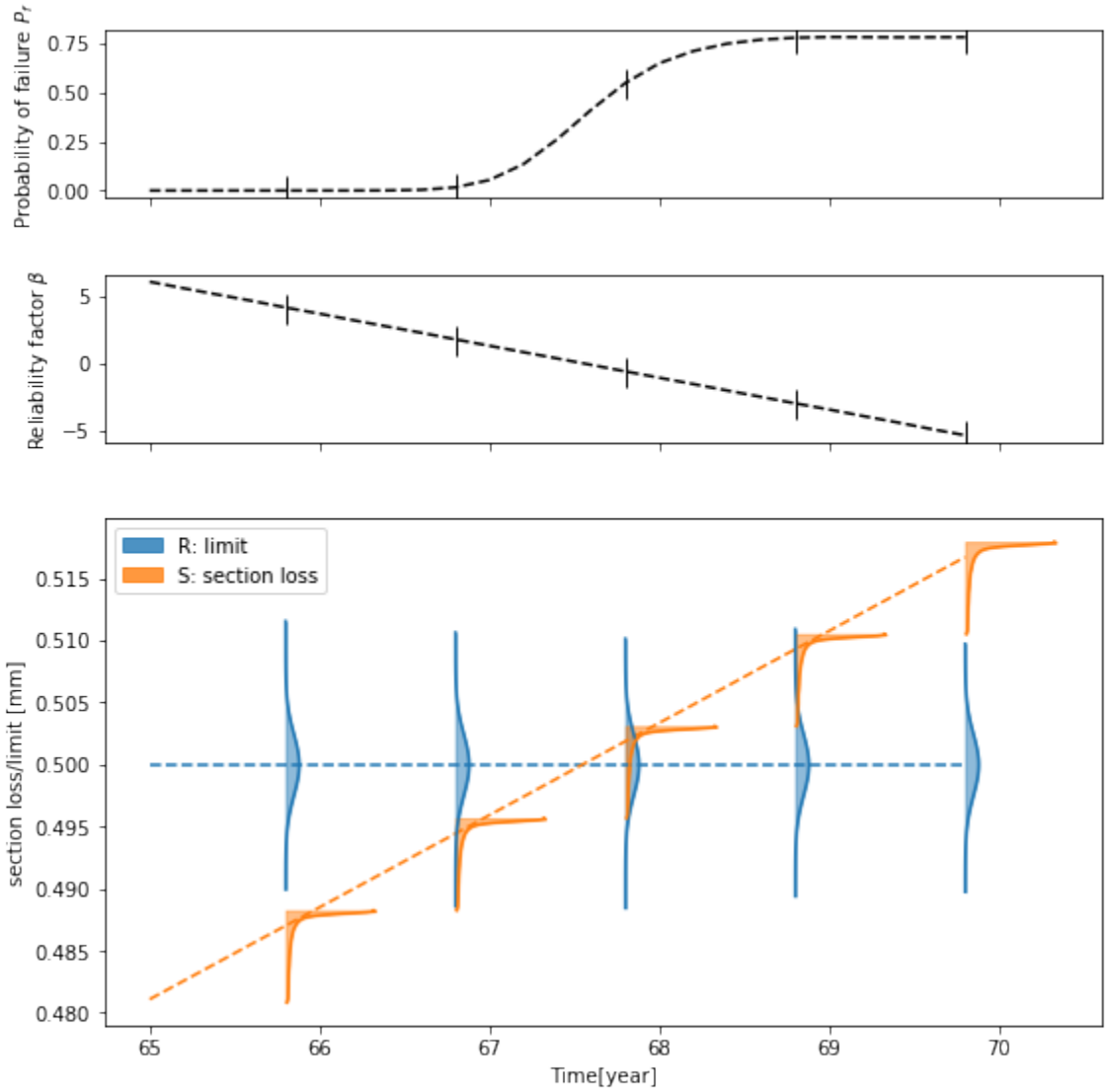
[16]:
```
# prepare Param object for section loss object
pars_sl = Param()
pars_sl.x_loss_rate = model_corr.x_loss_rate.mean()      # mm/year mean section loss␣
↪rate from the corrosion model
pars_sl.p_active_t_curve = (pf_lis, t_lis)               # use dummy data for this␣
↪example

# critical section loss from the external structural analysis
pars_sl.x_loss_limit_mean = 0.5          # mm
pars_sl.x_loss_limit_std = 0.5 * 0.005   # mm

# initialize section loss model object
model_sl = Section_loss_Model(pars_sl)

# run model for one time step, 80 year
model_sl.run(t_end = 50)
model_sl.postproc(plot=True)
```

```
warning: very small Pf
Pf(g = R-S < 0) from various methods
    sample count: 0.0
    g integral: -5.000000269139826e-06
    R S integral: 0.0
    beta_factor: 41.75751987706608
```



[27]:
```
# run the model through a list of year steps
pf_sl, beta_sl = model_sl.section_loss_with_year(year_lis=np.arange(65,70,0.2),␣
↪amplify=5e-4)
```

```
warning: very small Pf
warning: very small Pf
warning: very small Pf
warning: very small Pf
warning: very small Pf
warning: very small Pf
```



[ ]:

3.5 cracking model example

[39]:
```
%matplotlib inline
# %load_ext autoreload
# %autoreload 2

import helper_func as hf
from cracking import Cracking_Model
```

[40]:
```
# raw data
class Param: pass
```

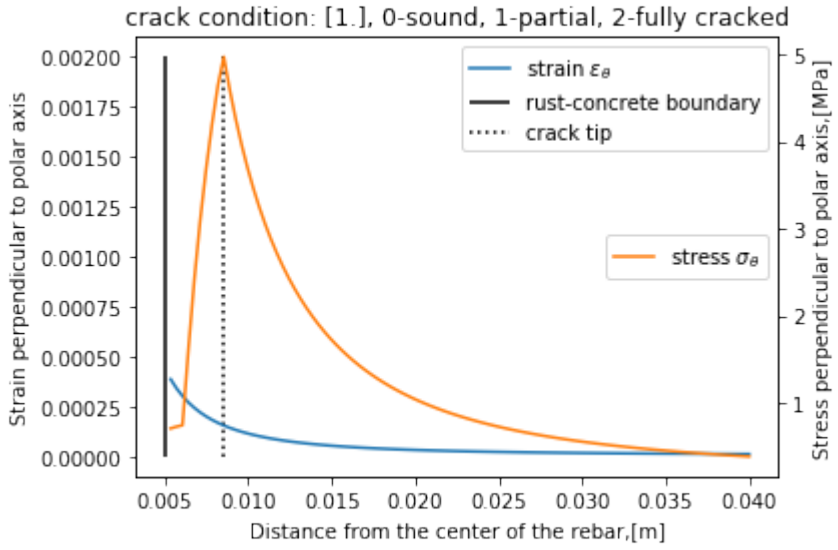
(continues on next page)

(continued from previous page)

```
raw_pars = Param()

# material properties
r0_bar_mean = 5e-3          # rebar diameter [m]
f_t_mean=5.                 # concrete ultimate tensile strength[MPa]
E_0_mean=32e3               # concrete modulus of elesticity [Mpa]

x_loss_mean = 12.5e-6*0.6   # rebar section loss, mean [m]
cover_mean = 4e-2           # cover thickness, mean [m]

raw_pars.r0_bar = Normal_custom(r0_bar_mean, 0.1*r0_bar_mean, non_negative=True)
raw_pars.x_loss = Normal_custom(x_loss_mean, 0.1*x_loss_mean, non_negative=True)  #␣
↪or from the corrosion model solution
raw_pars.cover = Normal_custom(cover_mean, 0.1*cover_mean, non_negative=True)
raw_pars.f_t = Normal_custom(f_t_mean, 0.1*f_t_mean, non_negative=True)
raw_pars.E_0 = Normal_custom(E_0_mean, 0.1*E_0_mean, non_negative=True)
raw_pars.w_c = Normal_custom(0.5, 0.1*0.6, non_negative=True)
raw_pars.r_v = Beta_custom(2.96, 2.96*0.05, 3.3, 2.6)  # rust volumetric expansion␣
↪rate  2.96 lower 2.6  upper: 3.3
```

[41]:
```
# initialize model
model_crack = Cracking_Model(raw_pars)
# run model in deterministic mode to check the stress and strain diagram
model_crack.run(stochastic=False)
```

```
deterministic
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:256:␣
↪RuntimeWarning: invalid value encountered in greater_equal
  sol = solve_stress_strain_crack_stochastic(self.pars)  # no plot
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:257:␣
↪RuntimeWarning: invalid value encountered in less_equal
  else:
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:273:␣
↪RuntimeWarning: invalid value encountered in less
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:70:␣
↪RuntimeWarning: invalid value encountered in less_equal
  return sigma_theta
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:73:␣
↪RuntimeWarning: invalid value encountered in greater
  def crack_width_open(a, b, u_st, f_t, E_0):
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:73:␣
↪RuntimeWarning: invalid value encountered in less_equal
  def crack_width_open(a, b, u_st, f_t, E_0):
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:78:␣
↪RuntimeWarning: invalid value encountered in greater
  inner radius boundary of the rust (center of rebar to rust-concrete) [m]
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:78:␣
↪RuntimeWarning: invalid value encountered in less_equal
  inner radius boundary of the rust (center of rebar to rust-concrete) [m]
```



[42]:
```
# run model in stochastic mode
model_crack.run(stochastic=True)
model_crack.postproc()

print(model_crack.crack_visible_rate_count)
print(model_crack.R_c - model_crack.pars.r0_bar) #/ M.pars.cover
print(model_crack.pars.cover)
```

```
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:166:␣
↪RuntimeWarning: divide by zero encountered in true_divide

/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:168:␣
↪RuntimeWarning: divide by zero encountered in true_divide

/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:256:␣
↪RuntimeWarning: invalid value encountered in greater_equal
  sol = solve_stress_strain_crack_stochastic(self.pars)  # no plot
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:257:␣
↪RuntimeWarning: invalid value encountered in less_equal
  else:
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:267:␣
↪RuntimeWarning: divide by zero encountered in true_divide
  crack_length_over_cover[np.isnan(crack_length_over_cover)] = 0.0  # crack length=0␣
↪for no crack
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:273:␣
↪RuntimeWarning: invalid value encountered in less
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:70:␣
↪RuntimeWarning: invalid value encountered in less_equal
  return sigma_theta
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:73:␣
↪RuntimeWarning: invalid value encountered in greater
  def crack_width_open(a, b, u_st, f_t, E_0):
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:73:␣
↪RuntimeWarning: invalid value encountered in less_equal
  def crack_width_open(a, b, u_st, f_t, E_0):
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:78:␣
↪RuntimeWarning: invalid value encountered in greater
```

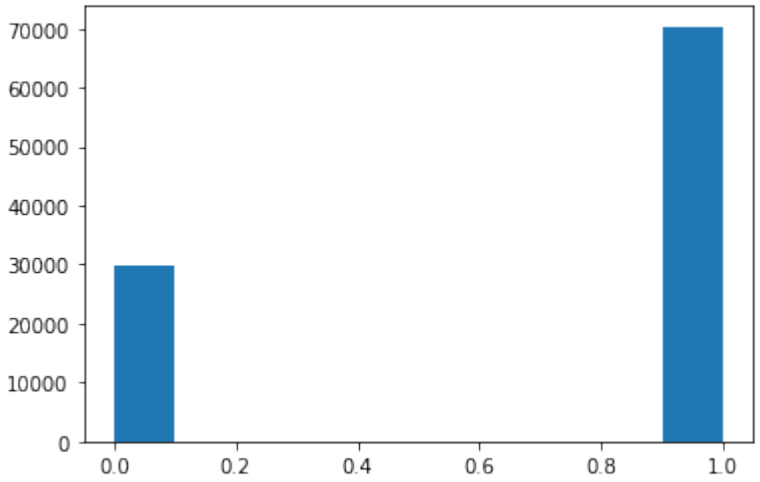
(continues on next page)

(continued from previous page)

```
  inner radius boundary of the rust (center of rebar to rust-concrete) [m]
/Users/gangli/Local Documents/Mitacs project local/Tinkrete/modules/cracking.py:78:␣
↪RuntimeWarning: invalid value encountered in less_equal
  inner radius boundary of the rust (center of rebar to rust-concrete) [m]
0.0
[0.01133316 0.00386504        nan ... 0.00987846 0.00390584 0.00404105]
[0.04600693 0.03706798 0.04092084 ... 0.04372599 0.03084513 0.03971957]
```

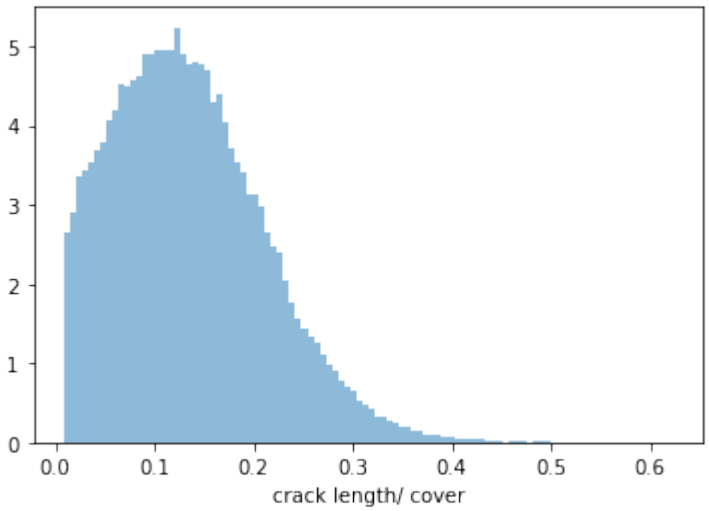
[43]:
```
plt.figure()
hf.Hist_custom(model_crack.crack_condition)
```

[43]:
```
(array([29667.,     0.,     0.,     0.,     0.,     0.,     0.,     0.,
            0., 70333.]),
 array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ]),
 <a list of 10 Patch objects>)
```



[52]:
```
# histgram of the relative crack length though the cover
hf.Hist_custom(model_crack.crack_length_over_cover[model_crack.crack_length_over_
↪cover != 0]) # eliminate the uncracked case
plt.xlabel('crack length/ cover')
```

[52]:
```
Text(0.5, 0, 'crack length/ cover')
```



[62]:

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

V

W

X