RIS

March 13, 2020

RIS in Ternay Alloys

1. Fractional concentration

$$X_{Fe} = \Omega C_{Fe}$$
$$X_{Cr} = \Omega C_{Cr}$$
$$X_{Ni} = \Omega C_{Ni}$$

   Where Average atomic volume

$$\Omega = \frac{1}{N}$$

   Atomic density

$$N = 9.1 \times 10^{28} \#/m^3$$

2. Continuity equations

$$\frac{dX_{Fe}}{dt} = -\Omega \times \nabla J_{Fe}$$
$$\frac{dX_{Cr}}{dt} = -\Omega \times \nabla J_{Cr}$$
$$\frac{dX_{Ni}}{dt} = -\Omega \times \nabla J_{Ni}$$
$$\frac{dX_v}{dt} = -\Omega \times \nabla J_v + G_v - R \times X_v \times X_i$$
$$\frac{dX_i}{dt} = -\Omega \times \nabla J_i + G_i - R \times X_v \times X_i$$

3. Flux eqaution for atoms and defects

$$\Omega \times J_{Fe} = -\alpha D_{Fe}(1 - X_{Fe})\nabla X_{Fe} + [\alpha D_{Cr}\nabla X_{Cr} + \alpha D_{Ni}\nabla X_{Ni} + (d^v_{Fe} - D_v)\nabla X_v + (D_i - d^i_{Fe})\nabla X_i]X_{Fe}$$
$$\Omega \times J_{Cr} = -\alpha D_{Cr}(1 - X_{Cr})\nabla X_{Cr} + [\alpha D_{Fe}\nabla X_{Fe} + \alpha D_{Ni}\nabla X_{Ni} + (d^v_{Cr} - D_v)\nabla X_v + (D_i - d^i_{Cr})\nabla X_i]X_{Cr}$$
$$\Omega \times J_{Ni} = -\alpha D_{Ni}(1 - X_{Ni})\nabla X_{Ni} + [\alpha D_{Cr}\nabla X_{Cr} + \alpha D_{Fe}\nabla X_{Fe} + (d^v_{Ni} - D_v)\nabla X_v + (D_i - d^i_{Ni})\nabla X_i]X_{Ni}$$
$$\Omega \times J_v = -D_v\nabla X_v + \alpha X_v(d^v_{Fe}\nabla X_{Fe} + d^v_{Cr}\nabla X_{Cr} + d^v_{Ni}\nabla X_{Ni})$$
$$\Omega \times J_i = -D_i\nabla X_i - \alpha X_i(d^i_{Fe}\nabla X_{Fe} + d^i_{Cr}\nabla X_{Cr} + d^i_{Ni}\nabla X_{Ni})$$

4. Diffusion coefficients

$$D_{Fe} = d^v_{Fe} X_v + d^i_{Fe} X_i$$
$$D_{Cr} = d^v_{Cr} X_v + d^i_{Cr} X_i$$
$$D_{Ni} = d^v_{Ni} X_v + d^i_{Ni} X_i$$
$$D_v = d^v_{Fe} X_{Fe} + d^v_{Cr} X_{Cr} + d^v_{Ni} X_{Ni}$$
$$D_i = d^i_{Fe} X_{Fe} + d^i_{Cr} X_{Cr} + d^i_{Ni} X_{Ni}$$

5. Diffusivities
   Vacancies

$$\lambda^v_a = \lambda$$
$$d^v_{Fe} = \nu^v_{Fe} (\lambda^v_a)^2$$
$$d^v_{Cr} = \nu^v_{Cr} (\lambda^v_a)^2$$
$$d^v_{Ni} = \nu^v_{Ni} (\lambda^v_a)^2$$

   Interstitial

$$\lambda^i_a = \sqrt{\frac{2}{3}} \lambda$$
$$d^i_{Fe} = \nu^i_{Fe} (\lambda^i_a)^2$$
$$d^i_{Cr} = \nu^i_{Cr} (\lambda^i_a)^2$$
$$d^i_{Ni} = \nu^i_{Ni} (\lambda^i_a)^2$$

6. Jump frequency

$$\nu^v_{Fe} = \nu^{v,0}_{Fe} e^{-\frac{E^{v,m}_{Fe}}{k_b T}}$$
$$\nu^v_{Cr} = \nu^{v,0}_{Cr} e^{-\frac{E^{v,m}_{Cr}}{k_b T}}$$
$$\nu^v_{Ni} = \nu^{v,0}_{Ni} e^{-\frac{E^{v,m}_{Ni}}{k_b T}}$$
$$\nu^i_{Fe} = f^i \nu^{i,0} e^{-\frac{E^{i,m}}{k_b T}}$$
$$\nu^i_{Cr} = f^i \nu^{i,0} e^{-\frac{E^{i,m}}{k_b T}}$$
$$\nu^i_{Ni} = f^i \nu^{i,0} e^{-\frac{E^{i,m}}{k_b T}}$$

7. Migration energy
   Pair interaction energy
   Like atoms

$$E_{CrCr} = \frac{2E^{Cr}_{coh}}{Z}$$

$$E_{FeFe} = \frac{2E^{Fe}_{coh}}{Z}$$

$$E_{NiNi} = \frac{2E^{Ni}_{coh}}{Z}$$

   Unlike atoms

$$E_{NiCr} = \frac{E_{NiNi} + E_{CrCr}}{2} - E^{ord}_{NiCr}$$

$$E_{FeCr} = \frac{E_{FeFe} + E_{CrCr}}{2} - E^{ord}_{FeCr}$$

$$E_{FeNi} = \frac{E_{FeFe} + E_{NiNi}}{2} - E^{ord}_{FeNi}$$

   Atoms and vacancies

$$E_{Cr-v} = \frac{E^{Cr}_{coh} + E^{Cr-v}_{f}}{Z}$$

$$E_{Ni-v} = \frac{E^{Ni}_{coh} + E^{Ni-v}_{f}}{Z}$$

$$E_{Fe-v} = \frac{E^{Fe}_{coh} + E^{Fe-v}_{f}}{Z}$$

   Saddle point energy

$$ES^{Cr}_{pure} = E^{Cr-v}_{m} + Z(E_{CrCr} + E_{Cr-v})$$

   Vacancy migration energy

$$E^{v,m}_{Fe} = C_{Fe}(-\frac{3}{2}E^{coh}_{Fe} + \frac{1}{2}E^{v,m}_{Fe,pure} - \frac{1}{2}E^{v,f}_{Fe}) + C_{Cr}(-E^{coh}_{Fe} - \frac{1}{2}E^{coh}_{Cr} + \frac{1}{2}E^{v,m}_{Cr,pure} - \frac{1}{2}E^{v,f}_{Cr} - ZE^{ord}_{FeCr}) + C_{Ni}(-E^{coh}_{Fe} - \frac{1}{2}$$

$$E^{v,m}_{Cr} = C_{Cr}(-\frac{3}{2}E^{coh}_{Cr} + \frac{1}{2}E^{v,m}_{Cr,pure} - \frac{1}{2}E^{v,f}_{Cr}) + C_{Fe}(-E^{coh}_{Cr} - \frac{1}{2}E^{coh}_{Fe} + \frac{1}{2}E^{v,m}_{Fe,pure} - \frac{1}{2}E^{v,f}_{Fe} - ZE^{ord}_{FeCr}) + C_{Ni}(-E^{coh}_{Cr} - \frac{1}{2}$$

$$E^{v,m}_{Ni} = C_{Ni}(-\frac{3}{2}E^{coh}_{Ni} + \frac{1}{2}E^{v,m}_{Ni,pure} - \frac{1}{2}E^{v,f}_{Ni}) + C_{Fe}(-E^{coh}_{Ni} - \frac{1}{2}E^{coh}_{Fe} + \frac{1}{2}E^{v,m}_{Fe,pure} - \frac{1}{2}E^{v,f}_{Fe} - ZE^{ord}_{FeNi}) + C_{Cr}(-E^{coh}_{Ni} - \frac{1}{2}$$

   Interstitial migration energy

$$E^{i,m} = 0.9eV$$

7. Defect production rate

$$G_v = \eta\phi + \rho D_v(X_v^{th} - X_v)$$
$$X_v^{th} = e^{\frac{S^{v,f}}{k_b}} e^{-\frac{E^{v,f}}{k_b T}}$$

$$G_i = \eta\phi + \rho D_i(X_i^{th} - X_i)$$
$$X_v^{th} = small$$
$$R = [(\nu_{Fe}^v + \nu_{Fe}^i)X_{Fe} + (\nu_{Cr}^v + \nu_{Cr}^i)X_{Cr} + (\nu_{Ni}^v + \nu_{Ni}^i)X_{Ni}]Z$$

Algorithms

1) Input data
2) Set up all variables
3) Claculate energies, jump frequencies, diffusivities, diffusion coefficients
4) Discretize fractional concentrations
5) Apply PDE solver to continuity equations
6) Output fractional concentration tensor (LŒTŒ5)

Solving Partial Differential Equations

A method of lines for partial differential equations (PDEs) where one reduces a PDE to a system of ODE and then applies standard methods.

$$\frac{\partial u(x,t)}{\partial t} = \beta\frac{\partial^2 u(x,t)}{\partial x^2} + f(x,t), x \in [0,L], t \in [0,T]$$
$$u(0,t) = s(t), t \in [0,T]$$
$$\frac{\partial}{\partial x}u(L,t) = 0, t \in [0,T]$$
$$u(x,0) = I(x), x \in [0,L]$$

Discretizing the 2nd-order derivative in space with a finite difference on a mesh

$$x_i = i\Delta x, i = 1, ..., N-1$$

then ODE:

$$\frac{\partial u_i(t)}{\partial t} = \beta\frac{u_{i+1}(t) - 2u_i(t) + u_{i-1}(t)}{\Delta x^2} + f_i(t), i = 1, ..., N-1$$

The boundary condition on x=0, u(0,t)=s(t), gives rise to the ODE

$$u_0^{'} = s^{'}(t), u_0(0) = s(0)$$

At the other end, x=L, we use a centered difference approximation The boundary condition on x=0, u(0,t)=s(t), gives rise to the ODE

$$\frac{u_{N+1}(t) - u_{N-1}(t)}{2\Delta x} = 0$$

and combine it with the scheme for i=N to obtain the modified boundary ODE

$$\frac{\partial u_N(t)}{\partial t} = \beta \frac{2u_{N-1}(t) - 2u_N(t)}{\partial x^2} + f_N(t)$$

To summarize, the ODE system reads

$$\frac{du_0}{dt} = s^{'}(t)$$

$$\frac{du_i}{dt} = \frac{\beta}{\Delta x^2}(u_{i+1}(t) - 2u_i(t) + 2u_{i-1}(t)) + f_i(t), i = 1, ..., N-1$$

$$\frac{du_N}{dt} = \frac{2\beta}{\Delta x^2}(u_{N-1}(t) - u_N(t)) + f_i(t)$$

The initial conditions are

$$u_0(0) = s(0)$$

$$u_i(0) = I(x_i), i = 1, ..., N$$

Points

6/03/2020 Compléter des unités pour chaque paramètre Rlier une variable et son unité Rendre des variables en ordre et lisible Evaluer des algorithms en fonction de la complexité en temps

13/03/2020 Distinguer des paramètres physiques et ceux-ci numériques Faire une table de tous les parameters avec certaines sets

[0]:
```
"""
Python version:
This code need Python 2.7 compile environment beacause of a package odespy.

Install odespy:
This operation as followed is uniquely supported in Google Colab. If you use
another platform such as Anaconda, please remove this line and look over a
website https://github.com/hplgit/odespy to install odespy.

No other requirements:
Despite Odespys many dependencies on other software, you can run the basic
solvers implemented in pure Python without any additional software packages.
"""
!pip install git+https://github.com/hplgit/odespy.git
```

[0]:
```
# -*- coding: utf-8 -*-
"""
Created on Fri Feb 28 08:50:29 2020

@author: Chao PAN

Simulation for RIS with model MIK (Modified Inverse Kirkendall)
Ternary Alloys Ee-Cr-Ni

This program calculates the amount of radiation induced segregation for a
ternary concentrated alloy. The formulation is based on the perks model
and is solved numerically using the gear subroutines.
```

```
"""

#import odespy
import numpy as np
#import matplotlib.pyplot as plt


##################################File name##################################
ERROR_FILE, OUTPUT_FILE = "error.txt", "output.txt"
##################################File name##################################

#################################Input to MIK################################
# Distance to mesh groups (m)
R1, R2, RF = 4.0, 18.0, 2018.0

# No. of points in mesh groups
N1, N2, N3 = 16, 14, 20

# Input time setp to gear
HO = 1e-9

# Error control parameter
EPS = 1e-9

# Peak displacement rate (dpa/s)
DISPRT = 1.4e-6

# Vacancy/Interstitial production efficiency
ETAV, ETAI = 1.0, 1.0

# Dose
DOSE = 1.0

# Peak temperature (řC)
TEMPC = 360.

# Concentration of B, C (fractional concentration #%)
CONCB, CONCC = 0.21, 0.09

# Peak dislocation density (#/m2)
DISL = 1e14

# No. density (#/m3)
NAT = 9.1e28

# Jump distance (m)
LAMBDA = 3.5e-10
```

```
# Jump correlation factors for A, B, C, interstitial
FAV, FBV, FCV, FI = 0.785, 0.668, 0.872, 0.660

# Relative vacancy jump frequency ratio for A, B, C
# 1.6(1.4), 2.4(2.3) 1.0 or 1.86666666666, 3.3333333333
WAV, WBV, WCV = 1.8, 3.2, 1.0

# Relative interstitial jump frequency ratio for A, B, C
WAI, WBI, WCI = 1.0, 1.0, 1.0

# Cohesive energies: -4.28, -4.10, -4.44 (eV)
ECOHA, ECOHB, ECOHC = -4.28, -4.10, -4.44

# Interstitial migration energies for A, B, C (eV)
EMIA, EMIB, EMIC = 0.9, 0.9, 0.9

# Vacancy formation enthalpy (=kb)
SV = 1.0

# Pure element [vacancy] migratio energies for A, B, C (eV)
EMA, EMB, EMC = 1.28, 0.97, 1.04

# Pure element [vacancy] formation energies for A, B, C (eV)
EFA, EFB, EFC = 1.4, 1.6, 1.79

# Grain boundary formation energy (eV)
EFGB = 1.4

# Ordering energies (eV)
EORDAB, EORDAC, EORDBC = 0.003, -0.001, 0.005

# Debye frequencies (/s)
NUOV, NUOI = 1.5e13, 1.5e12

# Thermo factor
AL = 1.0

# Neighbor atoms
Z = 12.0

# Dislocation bias for vacancy/interstitial
BIASV, BIASI = 1.0, 1.0

# User-required output times
TOUTPT = [1e-0, 1e1, 5e2, 1e3, 5e3, 1.4e4, 5e4, 7.1e4, 1e5, 1.4e5, 3.6e5,
          4.3e5, 7.1e5, 2.1e6, 2.2e6, 2.3e6, 5e6, 7e7, 1e8, 0e0]
```

```
# Indicator whether profiles will be used
FRAC = "N"

# Fraction of max temperature
TFRAC = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
         1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]

# Fraction of peak atom A
CAFRAC = [1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
         1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000]

# Fraction of peak atom B
CBFRAC = [0.1231, 0.2991, 0.3912, 0.4573, 0.5100, 0.6988, 0.8144, 0.8969,
          0.9613, 1.0150, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000]

# Fraction of peak atom C
CCFRAC = [1.1670, 1.1334, 1.1160, 1.1033, 1.0933, 1.0574, 1.0354, 1.0196,
          1.0074, 0.9971, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000]

# Fraction of peak damage
DFRAC = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
         1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]

# Fraction of max dislocation density
SFRAC = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
         1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
#################################Input to MIK################################

###################################Constant##################################
# Boltzmann constant = 1.38064852e-23 m2.kg/s/K
BOLTZ = 8.617e-5

#
SCFAC = 1e-9
###################################Constant##################################

##############################Global variables###############################
# Time steps
NSTEP = N1+N2+N3
N = 5*NSTEP

# Concentration of A
CONCA = 1.0-(CONCB+CONCC)


# Fraction of max temperature, peak atoms, damage, max dislocation density
# Profiles will not be used
if FRAC == "N":
    TFRAC = np.ones(NSTEP)
    CAFRAC = np.ones(NSTEP)
    CBFRAC = np.ones(NSTEP)
    CCFRAC = np.ones(NSTEP)
    SFRAC = np.ones(NSTEP)
    DFRAC = np.ones(NSTEP)
else:
    _temp = np.zeros(NSTEP)

    _temp[:NSTEP-1] = TFRAC[:NSTEP-1]
    # TFRAC = np.zeros(NSTEP)
    TFRAC[:NSTEP-1] = _temp[:NSTEP-1]

    _temp[:NSTEP] = CAFRAC[:NSTEP]
    CAFRAC = np.zeros(NSTEP)
    CAFRAC[:NSTEP] = _temp[:NSTEP]

    _temp[:NSTEP] = CBFRAC[:NSTEP]
    CBFRAC = np.zeros(NSTEP)
    CBFRAC[:NSTEP] = _temp[:NSTEP]

    _temp[:NSTEP] = CCFRAC[:NSTEP]
    CCFRAC = np.zeros(NSTEP)
    CCFRAC[:NSTEP] = _temp[:NSTEP]

    _temp[:NSTEP] = SFRAC[:NSTEP]
    SFRAC = np.zeros(NSTEP)
    SFRAC[:NSTEP] = _temp[:NSTEP]

    _temp[:NSTEP] = DFRAC[:NSTEP]
    DFRAC = np.zeros(NSTEP)
    DFRAC[:NSTEP] = _temp[:NSTEP]
    del _temp
###############################Global variables#############################

class RIS:
    """
    Simulation of RIS is based on MIK
    """
    def __init__(self):
        """
        Initialise model including variables and files.
        Seu up all variables for this model, including concentrations, fluxes,
        diffusivities, geometry, defects, temperature, energy etc.
        Arguments:
            None
        Return:
            None
        """
        # Variables
        self.X_Fe_fraction = np.ones(NSTEP)
        self.X_Cr_fraction = np.ones(NSTEP)
        self.X_Ni_fraction = np.ones(NSTEP)
        self.T_fraction = np.ones(NSTEP)
        self.Damage_fraction = np.ones(NSTEP)
        self.Max_dislocation_density = np.ones(NSTEP)
        if FRAC == "N":
            SFRAC = np.ones(NSTEP)
            DFRAC = np.ones(NSTEP)
        # Dose
        self.DOSE = DOSE
        # Time
        self.TOUT = 0.0
        self.TSTOP = DOSE/DISPRT
        #
        self.NOUT = 0

        # Temperature
        self.TEMP = (TEMPC+273)*TFRAC
        self.TKT = BOLTZ*self.TEMP

        # Concentrations
        self.CA = CONCA*CAFRAC
        self.CB = CONCB*CBFRAC
        self.CC = CONCC*CCFRAC
        self.CI = np.zeros(NSTEP)
        self.CVTHER = self.set_CVTHER()
        self.CV = self.CVTHER.copy()
        self.CERR = np.zeros(NSTEP)
        self.NA = np.zeros(NSTEP)
        self.NB = np.zeros(NSTEP)
        self.NC = np.zeros(NSTEP)
        self.NV = np.zeros(NSTEP)
        self.NI = np.zeros(NSTEP)
        # Gradient
        self.GRADCA = np.zeros(NSTEP)
        self.GRADCB = np.zeros(NSTEP)
        self.GRADCC = np.zeros(NSTEP)
        self.GRADCV = np.zeros(NSTEP)
        self.GRADCI = np.zeros(NSTEP)

        #
        self.CADOT = np.zeros(NSTEP)
        self.CBDOT = np.zeros(NSTEP)
        self.CCDOT = np.zeros(NSTEP)
        self.CVDOT = np.zeros(NSTEP)
        self.CIDOT = np.zeros(NSTEP)
        self.RECOMB = np.zeros(NSTEP)
        self.INTSINK = np.zeros(NSTEP)
        self.VACSINK = np.zeros(NSTEP)
        self.VACSOUR = np.zeros(NSTEP)

        # Energy
        self.EAA = self.calculate_energy("A", "A")
        self.EBB = self.calculate_energy("B", "B")
        self.ECC = self.calculate_energy("C", "C")
        self.EAB = self.calculate_energy("A", "B")
        self.EAC = self.calculate_energy("A", "C")
        self.EBC = self.calculate_energy("B", "C")
        self.EAV = self.calculate_energy("A", "V")
        self.EBV = self.calculate_energy("B", "V")
        self.ECV = self.calculate_energy("C", "V")
        self.ESA = self.calculate_energy("S", "A")
        self.ESB = self.calculate_energy("S", "B")
        self.ESC = self.calculate_energy("S", "C")
        self.EA = self.calculate_energy("A", "VM")
        self.EB = self.calculate_energy("B", "VM")
        self.EC = self.calculate_energy("C", "VM")

        # Jump frequency between defects (vacancy or interstitial)
        # and atoms (Fe, Cr, Ni)
        self.NUIA = self.calculate_NU_dk("I", "A")
        self.NUIB = self.calculate_NU_dk("I", "B")
        self.NUIC = self.calculate_NU_dk("I", "C")
        self.NUVA = self.calculate_NU_dk("V", "A")
        self.NUVB = self.calculate_NU_dk("V", "B")
        self.NUVC = self.calculate_NU_dk("V", "C")

        # Diffusivities
        self.DAI = self.calculate_D_dk("I", "A")
        self.DBI = self.calculate_D_dk("I", "B")
        self.DCI = self.calculate_D_dk("I", "C")
        self.DAV = self.calculate_D_dk("V", "A")
        self.DBV = self.calculate_D_dk("V", "B")
        self.DCV = self.calculate_D_dk("V", "C")
        self.RECA = self.set_REC_k("A")
        self.RECB = self.set_REC_k("B")
        self.RECC = self.set_REC_k("C")
        self.DA = self.calculate_D_dk(None, "A")
        self.DB = self.calculate_D_dk(None, "B")
        self.DC = self.calculate_D_dk(None, "C")
        self.DV = self.calculate_D_dk("V", None)
        self.DI = self.calculate_D_dk("I", None)
        self.DIFV = self.calculate_D_dk("V", None)
        self.DIFI = self.calculate_D_dk("I", None)

        # Geometry
        self.MESHSP = np.zeros(NSTEP)
        self.XVALUE = np.zeros(NSTEP)
        self.MESHSI = np.zeros(NSTEP)

        # Fluxes
        self.JA = np.zeros(NSTEP)
        self.JB = np.zeros(NSTEP)
        self.JC = np.zeros(NSTEP)
        self.JV = np.zeros(NSTEP)
        self.JI = np.zeros(NSTEP)
        self.JO = np.zeros(NSTEP)
        self.JA0 = 0.0
        self.JB0 = 0.0
        self.JC0 = 0.0
        self.DIVJA = np.zeros(NSTEP)
        self.DIVJB = np.zeros(NSTEP)
        self.DIVJC = np.zeros(NSTEP)
        self.DIVJV = np.zeros(NSTEP)
        self.DIVJI = np.zeros(NSTEP)

        # Defects
        self.DISLOC = DISL*SFRAC
        self.DISPV = np.zeros(NSTEP)
        self.DISPI = np.zeros(NSTEP)

        # Displacement of vacancy/interstitial
        self.DISPV = self.calculate_displacement_d("V")
        self.DISPI = self.calculate_displacement_d("I")

        #
        # Y0[N], Y[N], YDOT[N], RWORK[N**2+20], IWORK[2*N+20]
        self.Y0 = np.zeros(N)
        self.Y0[:NSTEP] = self.CA
        self.Y0[NSTEP:2*NSTEP] = self.CB
        self.Y0[2*NSTEP:3*NSTEP] = self.CC
        self.Y0[3*NSTEP:4*NSTEP] = self.CV
        self.Y0[4*NSTEP:] = self.CI
        self.Y = np.zeros(N)
        self.YDOT = np.zeros(N)
        self.RWORK = np.zeros(N**2+20)
        self.IWORK = np.zeros(2*N+20)
        self.XOUT = np.zeros(NSTEP)

        #
        self.MF = 0
        self.IERR = 0
        self.ITOL = 0
        self.IOPT = 0
        self.ITASK = 0
        self.ISTEP = 0
        self.T0 = 0

        #
        self.EPSA = 0.0

        #
        self.PSTOP = "N"

        self.fn = ""
        self.flag = True

        #
        self.PD = np.zeros(NSTEP)

    def main(self):
        """

        """
        self.empty_files()

        self.write_input_data()

        # Mesh grid
        self.mesh()

        while self.V["PSTOP"] == "N":
            self.preprocess()
            if self.V["PSTOP"] == "Y":
                break
            self.fex()
            self.solve()
            if self.IERR < -1:
                with open(ERROR_FILE, 'a+') as f:
                    f.write("ERROR RETURN WITH IERR= {]\n".format(self.IERR))
                    print("Terminated with error")
                    return
            elif self.IERR == -1:
                self.IERR = 2
            else:
                pass
            self.output()
        print("Completed")

    def set_CVTHER(self):
        """
        """
        CVTHER = np.exp(SV)*np.exp(-EFGB/self.TKT)
        CVTHER[-1] = CVTHER[-2]
        for i in range(1, NSTEP-1):
            CVTHER[i] = 0.5*(CVTHER[i]+CVTHER[i-1])
        return CVTHER

    def set_REC_k(self, k):
        """
        """
        return Z*(eval("self.NUI"+k)+eval("self.NUV"+k))

    def set_GRAD_C(self, token):
        """

        """
        _res = np.zeros(NSTEP)
        _res[:-1] = eval("self.C"+token)[1:]-eval("self.C"+token)[:-1]
        _res = _res/self.MESHSP
        return _res

    def empty_files(self):
        """
        Clean up all contents of ancient files. This operation is usually
        done at the beginning of one simulation.
        """
        # Clean up ancient files
        for FILE in [ERROR_FILE, OUTPUT_FILE]:
            with open(FILE, "w") as f:
                f.write("")

    def set_DIVJ(self):
        """
        """
        self.DIVJA[0] = 2.0*(self.JA[0]-self.JA0)/self.MESHSP[0]
        self.DIVJB[0] = 2.0*(self.JB[0]-self.JB0)/self.MESHSP[0]
        self.DIVJC[0] = 2.0*(self.JC[0]-self.JC0)/self.MESHSP[0]
        self.MESHSI[1:-1] = 0.5*(self.MESHSP[1:-1]+self.MESHSP[:-2])
        self.DIVJA[1:-1] = (self.JA[1:-1]-self.JA[:-2])/self.MESHSI[1:-1]
        self.DIVJB[1:-1] = (self.JB[1:-1]-self.JB[:-2])/self.MESHSI[1:-1]
        self.DIVJC[1:-1] = (self.JC[1:-1]-self.JC[:-2])/self.MESHSI[1:-1]
        self.DIVJV[1:-1] = (self.JV[1:-1]-self.JV[:-2])/self.MESHSI[1:-1]
        self.DIVJI[1:-1] = (self.JI[1:-1]-self.JI[:-2])/self.MESHSI[1:-1]

        self.DIVJA[-1] = 2.0*(self.JA[-1]-self.JA[-2])/self.MESHSP(-2)
        self.DIVJB[-1] = 2.0*(self.JB[-1]-self.JB[-2])/self.MESHSP(-2)
        self.DIVJC[-1] = 2.0*(self.JC[-1]-self.JC[-2])/self.MESHSP(-2)
        self.DIVJV[-1] = 2.0*(self.JV[-1]-self.JV[-2])/self.MESHSP(-2)
        self.DIVJI[-1] = 2.0*(self.JI[-1]-self.JI[-2])/self.MESHSP(-2)

    def calculate_energy(self, i, j):
        """
        For like atoms, the paire interaction energy is the cohesive energy,
        E_coh divided by the numbre of nearest neighbor bond pairs.
        e.g. EAA = ECOHA/(Z/2)
             EBB = ECOHB/(Z/2)
             ECC = ECOHC/(Z/2)

        For unlike atoms, pair interaction energies are determined from the
        average value of the like-atom pair energies less any ordering energy.
        e.g. EAB = 0.5*(EAA+EBB)-EORDAB
             EAC = 0.5*(EAA+ECC)-EORDAC
             EBC = 0.5*(EBB+ECC)-EORDBC

        For atoms and vacancies, pair interaction energies is fitted to
        the formation energy of the pure metal.
        e.g. EAV = (ECOHA+EFA)/Z
             EBV = (ECOHB+EFB)/Z
             ECV = (ECOHC+EFC)/Z

        The saddle point energy in the pure metal
        e.g. ESA = EMA+Z*(EAA+EAV)
             ESB = EMB+Z*(EBB+EBV)
             ESC = EMC+Z*(ECC+ECV)

        The migration energy for Cr, Ni and Fe via vacancies can be expressed
        as:
        EA(I)=(ESA+ESA*NA(I)+ESB*NB(I)+ESC*NC(I))/2-
               ((Z*(NA(I)*EAA+NB(I)*EAB+NC(I)*EAC+NV(I)*EAV))
               +(Z*(NA(I)*EAV+NB(I)*EBV+NC(I)*ECV)))
        EB(I)=(ESB+ESA*NA(I)+ESB*NB(I)+ESC*NC(I))/2-
               ((Z*(NA(I)*EAB+NB(I)*EBB+NC(I)*EBC+NV(I)*EBV))
               +(Z*(NA(I)*EAV+NB(I)*EBV+NC(I)*ECV)))
         EC(I)=(ESC+ESA*NA(I)+ESB*NB(I)+ESC*NC(I))/2-
```

```
            ((Z*(NA(I)*EAC+NB(I)*EBC+NC(I)*ECC+NV(I)*ECV))
            +(Z*(NA(I)*EAV+NB(I)*EBV+NC(I)*ECV)))
    Arguments:
        i: atom (Fe, Cr, Ni) or defect (vacancy, interstitial) or
           saddle point
        j: atom (Fe, Cr, Ni) or defect (Vacancy, interstitial) or
           saddle point
    Return:
        Energy (Migration energy, saddle point energy) value in float
    """
    _atoms = ["A", "B", "C"]
    _res = 0
    if i in _atoms and j in _atoms:
        # Like atoms
        if i == j:
            _res = eval("ECOH{}".format(i))/(Z/2)
        # Unlike atoms
        else:
            _res = 0.5*(eval("self.E"+i+i)+
                        eval("self.E"+j+j))-eval("EORD"+i+j)
    # Interaction energy between atoms and vacancies
    elif i in _atoms and j == "V":
        _res = (eval("ECOH"+i)+eval("EF"+i))/Z
    # Migration energy for Fe Cr Ni via vacancies
    elif i in _atoms and j == "VM":
        _term1 = (eval("self.ES"+i)+self.ESA*self.NA+self.ESB*self.NB+
                  self.ESC*self.NC)/2
        if i == "A":
            _term2 = Z*(self.NA*self.EAA+self.NB*self.EAB+
                        self.NC*self.EAC+self.NV*eval("self.E"+i+"V"))
        elif i == "B":
            _term2 = Z*(self.NA*self.EAB+self.NB*self.EBB+
                        self.NC*self.EBC+self.NV*eval("self.E"+i+"V"))
        elif i == "C":
            _term2 = Z*(self.NA*self.EAC+self.NB*self.EBC+
                        self.NC*self.ECC+self.NV*eval("self.E"+i+"V"))
        else:
            assert False, "Wrong arguments: {}".format(i)
        _term3 = (Z*(self.NA*self.EAV+self.NB*self.EBV+self.NC*self.ECV))
        _res = _term1-(_term2+_term3)
    # Saddle point energy
    elif i == "S" and j in _atoms:
        _res = eval("EM"+j)+Z*(eval("self.E"+j+j)+eval("self.E"+j+"V"))
    else:
        assert False, "Wrong arguments: {} {}".format(i, j)
    return _res
```

```python
    def mesh(self):
        """
        In order to discretize PDE, it is necessary to have a well-mesh grid.
        Arguments:
            MESHSP:
            XVALUE:
        Return:
            None
        """
        # ??? Why ignore last elemet
        self.MESHSP[:N1-1] = R1*SCFAC/N1
        # ??? Why ignore last elemet
        self.MESHSP[N1-1:N1+N2-1] = (R2-R1)*SCFAC/N2
        self.MESHSP[N1+N2-1:N1+N2+N3-1] = (RF-R2)*SCFAC/N3

        for i in range(1, NSTEP):
            self.XVALUE[i] = self.XVALUE[i-1]+self.MESHSP[i-1]

    def calculate_displacement_d(self, d):
        """
        Defect displacement (vacancy or interstitial) =
        displacement ratio * defect production efficiency * defect fraction
        e.g. DISPV(I)=DISPRT*ETAV*DFRAC(I)
             DISPI(I)=DISPRT*ETAI*DFRAC(I)
        Arguments:
            d: token for defect type (vacancy or interstitial)
        Return:
            array of defect displacement
        """
        return DISPRT*eval("ETA"+d)*DFRAC

    def calculate_NU_dk(self, d, k):
        """
        Jump frequency includes vacancy jump frequency and interstitial jump
        frequency.

        Formula:
            NU = NU0*W*F*EXP(-EM/kb/T)
        where:
            NU: jump frequency
            NU0: debye frequency
            W: relative defect jump frequency
            EM: defect migration energies
            kb: boltzmann constant
            T: Temperature
            NU0*W*F: standard jump frequency
        For vacancy jump frequency:
        e.g. NUVA(I) = NUOV*WAV*FAV*EXP((-1*EA(I)/TKT(I)))
             NUVB(I) = NUOV*WBV*FBV*EXP((-1*EB(I)/TKT(I)))
             NUVC(I) = NUOV*WCV*FCV*EXP((-1*EC(I)/TKT(I)))

        For interstitial jump frequency:
        e.g. NUIA(I) = NUOI*WAI*FI*EXP((-1*EMIA)/TKT(I))
             NUIB(I) = NUOI*WBI*FI*EXP((-1*EMIB)/TKT(I))
             NUIC(I) = NUOI*WCI*FI*EXP((-1*EMIC)/TKT(I))

        Notion:
        A same standard interstitial jump frequency and a same interstitial
        correlation fator for Fe/Cr/Ni

        Arguments:
            k: token for one of three (Fe, Cr, Ni)
            d: token for one of two (vacancy, interstitial)
        return:
            array of jump frequency for k and d
        """
        if d == "V":
            _res = NUOV*eval("W"+k+"V")*eval("F"+k+"V")*\
                np.exp(-eval("self.E"+k)/self.TKT)
        elif d == "I":
            _res = NUOI*eval("W"+k+"I")*FI*np.exp(-eval("EMI"+k)/self.TKT)
        else:
            assert False, "Wrong defect type of d={}".format(d)
        return _res

    def calculate_D_dk(self, d, k):
        """
        This function is to calculate diffusivity between atom and defect.

        Formula:
            D(d, k) = NU(d, k)*lambda(d)^2
        where:
            D: diffusivity between defects and atoms
            lambda(d): unit cell size for defect
            NU(d, k): jump frequence between defects and atoms

        e.g. DAIO(I) = 0.66667*NUIA(I)*LAMBDA**2
             DBIO(I) = 0.66667*NUIB(I)*LAMBDA**2
             DCIO(I) = 0.66667*NUIC(I)*LAMBDA**2
             DAV(I) = NUVA(I)*LAMBDA**2
             DBV(I) = NUVB(I)*LAMBDA**2
             DCV(I) = NUVC(I)*LAMBDA**2
        Formula:
            DA(I)=DAV(I)*NV(I)+DAI(I)*NI(I)
            DB(I)=DBV(I)*NV(I)+DBI(I)*NI(I)
            DC(I)=DCV(I)*NV(I)+DCI(I)*NI(I)
            DIFV(I)=DAV(I)*NA(I)+DBV(I)*NB(I)+DCV(I)*NC(I)
            DIFI(I)=DAI(I)*NA(I)+DBI(I)*NB(I)+DCI(I)*NC(I)

        Parameters:
            k: token for one of three Fe, Cr, Ni
            d: token for one of two defects vacancy, interstitial
        return:
            array of diffusivity between atom and defect
        """
        if k is None:
            _res = eval("self.DA"+d)*self.NA+eval("self.DB"+d)*self.NB+\
                eval("self.DC"+d)*self.NC
        elif d is None:
            _res = eval("self.D"+k+"V")*self.NV+eval("self.D"+k+"I")*self.NI
        else:
            if d == "I":
                _res = 2/3*eval("self.NUI"+k)*LAMBDA**2
            elif d == "V":
                _res = eval("self.NUV"+k)*LAMBDA**2
            else:
                assert False, "Wrong defect type of d={}".format(d)
        return _res

    def calculate_J_dk(self, d, k):
        """
         JV(I)=NAT*(-1*DV(I)*GRADCV(I)+NV(I)*AL*(DAV(I)*GRADCA(I)+
                    DBV(I)*GRADCB(I)+DCV(I)*GRADCC(I)))
         JI(I)=NAT*(-1*DI(I)*GRADCI(I)-NI(I)*AL*(DAI(I)*GRADCA(I)+
                    DBI(I)*GRADCB(I)+DCI(I)*GRADCC(I)))
         JA(I)=NAT*(-1*DA(I)*AL*GRADCA(I)+NA(I)*(DAV(I)*GRADCV(I)-
                    DAI(I)*GRADCI(I)))-JO(I)*NA(I)
         JB(I)=NAT*(-1*DB(I)*AL*GRADCB(I)+NB(I)*(DBV(I)*GRADCV(I)-
                    DBI(I)*GRADCI(I)))-JO(I)*NB(I)
         JC(I)=NAT*(-1*DC(I)*AL*GRADCC(I)+NC(I)*(DCV(I)*GRADCV(I)-
                    DCI(I)*GRADCI(I)))-JO(I)*NC(I)
        """
        if k is None:
            _term1 = -1*eval("self.D"+d)*eval("self.GRADC"+d)
            _term2 = eval("self.DA"+d)*self.GRADCA+\
                eval("self.DB"+d)*self.GRADCB+eval("self.DC"+d)*self.GRADCC
            if d == "V":
                _term2 = self.NV*AL*_term2
            elif d == "I":
                _term2 = -self.NI*AL*_term2
            _res = _term1 + _term2
        elif d is None:
            _res = NAT*(-eval("self.D"+k)*AL*eval("self.GRADC"+k)+
                        eval("self.N"+k)*(eval("self.D"+k+"V")*self.GRADCV-
                                          eval("self.D"+k+"I")*self.GRADCI))-\
                        self.JO*eval("self.N"+k)
        return _res

    def write_input_data(self):
        """Export input data"""
        with open(OUTPUT_FILE, "a+") as f:
            f.write("EPS={}\n".format(EPS))
            f.write("DISPRT={}, ETAV={}, ETAI={}, DOSE={}\n".format(DISPRT,
                    ETAV, ETAI, DOSE))
            f.write("TEMP={} řC\n".format(TEMPC))
            f.write("CB={}, CC={}\n".format(CONCB, CONCC))
            f.write("DISL={}, NAT={}, LAMBDA={}\n".format(DISL, NAT, LAMBDA))
            f.write("FAV={}, FBV={}, FCV={}, FI={}\n".format(FAV, FBV, FCV,
                                                            FI))
            f.write("WAV={}, WBV={}, WCV={}\n".format(WAV, WBV, WCV))
            f.write("WAI={}, WBI={}, WCI={}\n".format(WAI, WBI, WCI))
            f.write("ECOHA={}, ECOHB={}, ECOHC={}\n".format(ECOHA, ECOHB,
                    ECOHC))
            f.write("EMIA={}, EMIB={}, EMIC={}, SV={}\n".format(EMIA, EMIB,
                    EMIC, SV))
            f.write("EMA={}, EMB={}, EMC={}\n".format(EMA, EMB, EMC))
            f.write("EFA={}, EFB={}, EFC={}, EFGB={}\n".format(EFA, EFB,
                    EFC, EFGB))
            f.write("EORDAB={}, EORDAC={}, EORDBC={}\n".format(EORDAB, EORDAC,
                    EORDBC))
            f.write("NUOV={}, NUOI={}\n".format(NUOV, NUOI))
            f.write("AL={}, Z={}, BIASV={}, BIASI={}\n".format(AL, Z, BIASV,
                    BIASI))
            f.write("TFRAC=\n{}\n".format(" ".join(TFRAC.astype(str))))
            f.write("CAFRAC=\n{}\n".format(" ".join(CAFRAC.astype(str))))
            f.write("CBFRAC=\n{}\n".format(" ".join(CBFRAC.astype(str))))
            f.write("CCFRAC=\n{}\n".format(" ".join(CCFRAC.astype(str))))
            f.write("DFRAC=\n{}\n".format(" ".join(DFRAC.astype(str))))
            f.write("SFRAC=\n{}\n".format(" ".join(SFRAC.astype(str))))

    def write_error_info(self, _error_info):
        with open(ERROR_FILE, 'a+') as f:
            f.write("ERROR RETURN WITH IERR= {]\n".format(self.IERR))

    def preprocess(self):
        """
        """
        if self.ISTEP == 0:
            self.T0 = 0
            self.MF = 222
            self.IERR = 1
            self.TOUT = TOUTPT[0]
            self.ISTEP += 1
            self.ITOL = 1
            self.ITASK = 1
            self.IOPT = 1
            self.Y = self.Y0.copy()
            self.IWORK[5] = 1000000
        elif self.ISTEP < self.NOUT:
            self.ISTEP += 1
            self.TOUT = TOUTPT[self.ISTEP]
        else:
            self.PSTOP = "Y"
            with open(ERROR_FILE, 'a+') as f:
                f.write("Stopping Time Reached\n")

    def fex(self, u, t):
        """
        Right-hand side of the ODE
        """
        self.CA = self.Y[:NSTEP]
        self.CB = self.Y[NSTEP:2*NSTEP]
        self.CC = self.Y[2*NSTEP:3*NSTEP]
        self.CV = self.Y[3*NSTEP:4*NSTEP]
        self.CI = self.Y[4:NSTEP]

        self.NA[:NSTEP] = 0.5*(self.CA[1:]+self.CA[:NSTEP])
        self.NB[:NSTEP] = 0.5*(self.CB[1:]+self.CB[:NSTEP])
        self.NC[:NSTEP] = 0.5*(self.CC[1:]+self.CC[:NSTEP])
        self.NV[:NSTEP] = 0.5*(self.CV[1:]+self.CV[:NSTEP])
        self.NI[:NSTEP] = 0.5*(self.CI[1:]+self.CI[:NSTEP])

        self.NUVA = self.calculate_NU_dk("V", "A")
        self.NUVB = self.calculate_NU_dk("V", "B")
        self.NUVC = self.calculate_NU_dk("V", "C")

        self.DAV = self.calculate_D_dk("V", "A")
        self.DBV = self.calculate_D_dk("V", "B")
        self.DCV = self.calculate_D_dk("V", "C")

        self.RECA = self.set_REC_k("A")
        self.RECB = self.set_REC_k("B")
        self.RECC = self.set_REC_k("C")
        self.RECA[-1] = self.RECA[-2]
        self.RECB[-1] = self.RECB[-2]
        self.RECC[-1] = self.RECC[-2]
        self.CVTHER[-1] = self.CVTHER[-2]
        self.DIFV = self.calculate_D_dk("V", None)
        self.DIFI = self.calculate_D_dk("I", None)
        self.DIFV[-1] = self.DIFV[-2]
        self.DIFI[-1] = self.DIFI[-2]
        self.RECA[1:-1] = 0.5*(self.RECA[1:-1]+self.RECA[:-2])
        self.RECB[1:-1] = 0.5*(self.RECB[1:-1]+self.RECB[:-2])
        self.RECC[1:-1] = 0.5*(self.RECC[1:-1]+self.RECC[:-2])
        self.CVTHER[1:-1] = 0.5*(self.CVTHER[1:-1]+self.CVTHER[:-2])
        self.DIFV[1:-1] = 0.5*(self.DIFV[1:-1]+self.DIFV[:-2])
        self.DIFI[1:-1] = 0.5*(self.DIFI[1:-1]+self.DIFI[:-2])

        self.JA0 = self.JB0 = self.JC0 = 0.0
        self.JA[-1] = self.JB[-1] = self.JC[-1] = 0.0
        self.GRADCA = self.set_GRAD_C("A")
        self.GRADCB = self.set_GRAD_C("B")
        self.GRADCC = self.set_GRAD_C("C")
        self.GRADCV = self.set_GRAD_C("V")
        self.GRADCI = self.set_GRAD_C("I")
        self.DA = self.calculate_D_dk(None, "A")
        self.DB = self.calculate_D_dk(None, "B")
        self.DC = self.calculate_D_dk(None, "C")
        self.DV = self.calculate_D_dk("V", None)
        self.DI = self.calculate_D_dk("I", None)

        self.JV = self.calculate_J_dk("V", None)
        self.JI = self.calculate_J_dk("I", None)
        self.JO = self.JI-self.JV
        self.JA = self.calculate_J_dk(None, "A")
        self.JB = self.calculate_J_dk(None, "B")
        self.JC = self.calculate_J_dk(None, "C")
        self.JV = self.JV-self.JO*self.NV
        self.JI = self.JI-self.JO*self.NI

        self.set_DIVJ()

        self.CADOT = -self.DIVJA/NAT
        self.CBDOT = -self.DIVJB/NAT
        self.CCDOT = -self.DIVJC/NAT
        self.RECOMB = self.RECA*self.CA+self.RECB*self.CB+self.RECC*self.CC
        self.INTSINK = self.DISLOC*self.DIFI
        self.VACSINK = self.DISLOC*self.DIFV
        self.VACSOUR = self.DISLOC*self.DIFV*self.CVTHER
        self.CVDOT[0] = 0.0
        self.CIDOT[0] = 0.0
        self.CVDOT[1:] = -self.DIVJV[1:]/NAT-self.RECOMB[1:]*self.CV[1:]*\
            self.CI[1:]-BIASV*self.VACSINK[1:]*self.CV[1:]+\
            self.VACSOUR[1:]+self.DISPV[1:]
        self.CIDOT[1:] = -self.DIVJI[1:]/NAT-self.RECOMB[1:]*self.CV[1:]*\
            self.CI[1:]-BIASI*self.INTSINK[1:]*self.CI[1:]+\
            self.DISPI[1:]

        self.Y[:NSTEP] = self.CA
        self.Y[NSTEP:2*NSTEP] = self.CB
        self.Y[2*NSTEP:3*NSTEP] = self.CC
        self.Y[3*NSTEP:4*NSTEP] = self.CV
        self.Y[4*NSTEP:] = self.CI
        self.YDOT[:NSTEP] = self.CADOT
        self.YDOT[NSTEP:2*NSTEP] = self.CBDOT
        self.YDOT[2*NSTEP:3*NSTEP] = self.CCDOT
        self.YDOT[3*NSTEP:4*NSTEP] = self.CVDOT
        self.YDOT[4*NSTEP:] = self.CIDOT

    def set_pde(self, u, t, L=None, beta=None, x=None):
        N = len(u) - 1
        dx = x[1] - x[0]
        rhs = np.zeros(N+1)
        rhs[0] = self.dsdt(t)
        for i in range(1, N):
            rhs[i] = (beta/dx**2)*(u[i+1] - 2*u[i] + u[i-1]) + self.f(x[i], t)
        rhs[N] = (beta/dx**2)*(2*u[i-1] - 2*u[i]) + self.f(x[N], t)
        return rhs

    def set_jacobin(self, u, t, L=None, beta=None, x=None):
        N = len(u) - 1
        dx = x[1] - x[0]
        K = np.zeros((N+1,N+1))
        K[0,0] = 0
        for i in range(1, N):
            K[i,i-1] = beta/dx**2
            K[i,i] = -2*beta/dx**2
            K[i,i+1] = beta/dx**2
        K[N,N-1] = (beta/dx**2)*2
        K[N,N] = (beta/dx**2)*(-2)
        return K

    def s(self, t):
        return 423
    def dsdt(self, t):
        return 0

    def f(self, x, t):
        return 0

    def solve(self, method='RKC'):
        N = 40
        L = 1
        x = np.linspace(0, L, N+1)
        f_kwargs = dict(L=L, beta=1, x=x)
        u = np.zeros(N+1)

        U_0 = np.zeros(N+1)
        U_0[0] = self.s(0)
        U_0[1:] = 283

        solvers = {
                "FE": odespy.ForwardEuler(self.set_pde, f_kwargs=f_kwargs),
                "BE": odespy.BackwardEuler(self.set_pde, f_is_linear=True,
                                           jac=self.set_jacobin,
                                           f_kwargs=f_kwargs,
                                           jac_kwargs=f_kwargs),
                "B2": odespy.Backward2Step(self.set_pde, f_is_linear=True,
                                           jac=self.set_jacobin,
                                           f_kwargs=f_kwargs,
                                           jac_kwargs=f_kwargs),
                "theta": odespy.ThetaRule(self.set_pde, f_is_linear=True,
                                          jac=self.set_jacobin, theta=0.5,
                                          f_kwargs=f_kwargs,
                                          jac_kwargs=f_kwargs),
                "RKF": odespy.RKFehlberg(self.set_pde, rtol=1E-6, atol=1E-8,
                                         f_kwargs=f_kwargs),
                "RKC": odespy.RKC(self.set_pde, rtol=1E-6, atol=1E-8,
                                  f_kwargs=f_kwargs, jac_constant=True)
                }
        dx = x[1] - x[0]
        beta = 1
        dt = dx**2/(2*beta) # Forward Euler limit
        print("Forward Euler stability limit:{}".format(dt))
        T = 1.2

        solver = solvers[method]
        solver.set_initial_condition(U_0)
        N_t = int(round(T/float(dt)))
        time_points = np.linspace(0, T, N_t+1)
        u, t = solver.solve(time_points)
        return u, t

    def output(self):
        """
        """
        self.CA = self.Y[:NSTEP]
        self.CB = self.Y[NSTEP:2*NSTEP]
        self.CC = self.Y[2*NSTEP:3*NSTEP]
        self.CV = self.Y[3*NSTEP:4*NSTEP]
        self.CI = self.Y[4:NSTEP]
        self.DOSE = DISPRT*TOUTPT[self.ISTEP]

        print("TOUT={}, DOSE={}".format(TOUTPT[self.ISTEP], self.DOSE))
        self.CERR = 1-(self.CA+self.CB+self.CC)
        self.XOUT = self.XVALUE*1e9

        TEMP1 = (self.CA*np.exp(-self.XOUT/0.8452)).sum()/\
            np.exp(-self.XOUT/0.8452).sum()
        TEMP2 = (self.CB*np.exp(-self.XOUT/0.7474)).sum()/\
            np.exp(-self.XOUT/0.7474).sum()
        TEMP3 = (self.CC*np.exp(-self.XOUT/0.9472)).sum()/\
            np.exp(-self.XOUT/0.9472).sum()
        CASURF = TEMP1/(TEMP1+TEMP2+TEMP3)
        CBSURF = TEMP2/(TEMP1+TEMP2+TEMP3)
        CCSURF = TEMP3/(TEMP1+TEMP2+TEMP3)

        with open(OUTPUT_FILE, "a+") as f:
            f.write("TIME={}, DOSE={}\n".format(TOUTPT[self.ISTEP],
                    self.DOSE))
            for i in range(NSTEP):
                f.write("{} {} {} {} {} {}\n".format(self.XOUT[i], self.CA[i],
                        self.CB[i], self.CC[i], self.CV[i], self.CI[i]))
            f.write("CASURF={}, CBSURF={}, CCSURF={}\n".format(CASURF,
                    CBSURF, CCSURF))

    def is_stop(self):
        """
        u[step_no] holds solution at t[step_no]
        """
        return


if __name__ == "__main__":
    ris = RIS()
```

Reference:
A Tutorial for the Odespy Interface to ODE Solvers
http://hplgit.github.io/odespy/doc/pub/tutorial/html/main_odespy.html

Example codes

https://github.com/hplgit/odespy/blob/master/doc/src/tutorial/src-odespy/pde_diffusion.py