Annex F. Source Code

F.1. *Pre-processor functions*

```
import re
import pandas as pd
import numpy as np
from collections import defaultdict

def nodes(data='nodes_RU_v4_detail_cc.csv', env='99999', inact='99998' ):
    """
    Reads Esatan node data csv export file and returns dict of node numbers grouped by node
          labels
    Note: labels need to be defined in Esatan model

    Parameters
    ----------

    env :   str
        environment (deep space) node number in Esatan model
    inact :   str
        inactive node number in Esatan model
    data : str
        Filepath for the Esatan nodal output csv file (must include node labels)

    Returns
    -------
    nn   :   int
        number of thermal nodes in the model excluding deep space and inactive nodes

    labels : dict
        dictionary of node numbers lists grouped by node labels

    """

    # read node data

    df = pd.read_csv(data, header=1)

    # extract node labels

    nodes = df.filter(regex = 'L(?!{})(?!{})(\d+)'.format(env, inact) )

    nodes.columns = nodes.columns.str.extract(r'L(\d+)', expand=False) # keep just node number

    node_dict = nodes.to_dict('records')[0]

    nn = len(node_dict) # number of nodes

    labels = {}

    # group nodes by labels
    for key, value in sorted(node_dict.items()):
        labels.setdefault(value, []).append(key)

    # change node numbers from strings to int
    for key in labels:
        labels[key] = list(map(int, (labels[key])))

    # collect node temperatures from analysis case

    temp = df.filter(regex = 'T(?!{})(?!{})(\d+)'.format(env, inact))
    temp.columns = temp.columns.str.extract(r'T(\d+)', expand=False) # keep just node number
    temp.index = ['T_ref']
    #temp.index = df['TIME']
    output = nodes.append(temp)

    # collect node areas
```

```
    area_df = df.filter(regex = 'A(?!{})(?!{})(\d+)'.format(env, inact))
    area = area_df.to_numpy()[0]

    area = np.insert(area,0,0)

    return nn, labels, output.T, area


def inits(data='nodal_detail_cc.csv', env='99999', inact='99998'):
    """
    Reads Esatan node data csv export file and returns initial heat load boundary conditions

    Parameters
    ----------

    env :   str
        environment (deep space) node number in Esatan model
    inact :   str
        inactive node number in Esatan model
    data : str
        Filepath for the Esatan nodal output csv file (must include at least QI, QS entities)

    Returns
    -------

    QI_init :   n x 1 numpy array
        Internal heat source vector
    QS_init :   n x 1 numpy array
        Solar flux heat source vector

    """

    # read node data

    df = pd.read_csv(data, header=1)

    # prepare initial boundary conditions

    QI = df.filter(regex = 'QI(?!{})(?!{})'.format(env, inact)) # get rid of environment and
        inactive nodes
    QI_init = QI.to_numpy()[0]

    QI_init = np.insert(QI_init, 0, 0) # insert zero for deep space node
    QI_init = QI_init[np.newaxis,:].T # make array 2D


    QS = df.filter(regex = 'QS(?!{})(?!{})(?!I)'.format(env, inact))
    QS_init = QS.to_numpy()[0]
    QS_init = np.insert(QS_init, 0, 0)
    QS_init = QS_init[np.newaxis,:].T

    return QI_init, QS_init

def conductors(nn, data='Cond_detail_cc.csv', env='99999', inact='99998'):
    """
    Parse Esatan conductor data csv export file at given filepath and returns GL and GR
        conductors
    Note: At least GL and GR entities need to be selected in output calls, print conductor CSV
        options

    Parameters
    ----------

    env :   str
        environment (deep space) node number in Esatan model
    inact :   str
        inactive node number in Esatan model
    data  : str
        Filepath for the Esatan conductor output csv file (must include at least GR, GL entities)
    nn    :   int
        number of thermal nodes in the model excluding deep space and inactive nodes
```

```
    Returns
    -------
    GL_init : n+1 x n+1 numpy array
        Linear conductor matrix (symmetric)
    GR_init : n+1 x n+1 numpy array
        Radiative exchange factor (REF) matrix (non-symmetric)

    """

    # prepare linear conductors

    df = pd.read_csv(data, header=1)

    GLs = df.filter(regex ='GL')
    GLs.columns = GLs.columns.str.extract(r'(\d+;\d+)', expand=False) # leave only indices in the
          column names

    GL_init = np.zeros((nn+1,nn+1))

    for Name, Data in GLs.iteritems():
        i = int(Name.split(';')[0])
        j = int(Name.split(';')[1])
        idx = (i,j)
        GL_init[idx] = Data.values

    #define GL diagonal elements as negative sum of respective row (all node conductor couplings)
    diag = np.negative(np.sum(GL_init, 1))
    di = np.diag_indices(nn+1)
    GL_init[di] = diag

    GL_init[0,0] = 1.0 # deep space node temperature = 0 K (this coef is needed to avoid
         singularity in heat equations)

    # prepare radiative conductors

    GRs = df.filter(regex ='GR')
    GRs.columns = GRs.columns.str.extract(r'(\d+;\d+)', expand=False) # leave only indices in the
          column names

    GRs.columns = GRs.columns.str.replace(env, '0') #replace default esatan deep space node
         number to 0

    GRs = GRs.drop(GRs.filter(regex=inact).columns, axis=1) # drop inactive nodes

    GR_init = np.zeros((nn+1,nn+1))

    for Name, Data in GRs.iteritems():
        i = int(Name.split(';')[0])
        j = int(Name.split(';')[1])
        idx = (i,j)
        GR_init[idx] = Data.values

    #esatan does not include stefan-boltzman const in GR output
    sigma = 5.670374e-8
    GR_init = GR_init*sigma

    GR_init[0,:] = 0. # REFs from deep space = 0
    GR_init = GR_init.T # needs to be transposed to result in proper equilibrium equations

    # REF entries on the main diagonal must be formed by the negative sum of the respective
         column
    diag = np.negative(np.sum(GR_init, 0))
    di = np.diag_indices(nn+1)
    GR_init[di] = diag

    return GL_init, GR_init


# set up regular expressions
# use https://regexper.com to visualise these if required
```

```
def parse_vf(filepath='ViewFactors.txt'):
    """
    This script parses the Esatan view factor report file (produced by REPORT_VF procedure) to
          collect data required to calculate radiative exchange factors to deep space.
    Note:
    only surfaces exposed to deep space are collected (if VF to environment = 0, the node is
          droped), thus unwanted surfaces have to be either conductive or inactive
    For proper results each emitting face must have unique node number (different nodes for
          surface 1 and 2 if both sides are emitting)
    "Report against thermal nodes" option in the report menu has to bee unticked

    Parameters
    ----------
    filepath : str
        Filepath for view factor report file to be parsed

    Returns
    -------
    data : list of dict of node areas, emissivities and view factors grouped by node numbers as
          keys

    """
    rx_dict = {
    'node number': re.compile(r'Emitting Node = (?P<node>\d*)\n'),
    #'area': re.compile(r'Area = (?P<area>\d.\d*)'),
    #'emissivity': re.compile(r'Emissivity = (?P<eps>\d.\d*)'),
    'view factor': re.compile(r'VF to environment = (?P<vf>\d.\d*)')
    }

    def _parse_line(line):
        """
        Do a regex search against all defined regexes and
        return the key and match result of the first matching regex

        """

        for key, rx in rx_dict.items():
            match = rx.search(line)
            if match:
                return key, match
        # if there are no matches
        return None, None

    data = {}  # create an empty dict to collect the data
    # open the file and read through it line by line
    with open(filepath, 'r') as file_object:

        for line in file_object:

            # at each line check for a match with a regex
            key, match = _parse_line(line)

            # extract node number
            if key == 'node number':
                node = match.group('node')

            # extract area and emissivity
            #if key == 'area':
                #area = match.group('area')

                #key, match = _parse_line(line.split(',')[1])
                #if key == 'emissivity':
                    #eps = match.group('eps')

            #extract view factor
            if key == 'view factor':
                vf = float(match.group('vf'))

                #if vf <= 0.02:
                    #continue # filter out internal surfacess
```

```
                # create a dictionary containing this row of data
                entry = {int(node): {'vf': vf}}

                # append the dictionary to the data list
                data.update(entry)

    return data

def opticals(node_labels, geom, viewFactors, areas):
    """
    Returns a dictionary of external face optical properties
    """

    # new dict of user selected node labels
    node_groups = {new_key: node_labels[new_key] for new_key in geom}

    faces = []

    for grp in node_groups:

        ar =[]
        VFs = []
        emissivities = []

        for node in node_groups[grp]:
            a = areas[node]
            vf = viewFactors[node]['vf']
            eps = geom[grp]
            ar.append(a)
            VFs.append(vf)
            emissivities.append(eps)

        entry = {'name':grp, 'nodes':node_groups[grp], 'areas':ar, 'VFs': VFs, 'eps':emissivities
              }
        faces.append(entry)

    return faces

def parse_hf(filepath='Radiative_results.txt', start=-5.0, step=5.0):
    """
    Parse Esatan radiative restults report file at given filepath

    Parameters
    ----------
    filepath : str
        Filepath for report file to be parsed

    start : float
        starting point of training variable vector

    step : float
        angle separation in degrees between consequtive heat flux training variables (angles)

    Returns
    -------
    train_data : numpy array
        Parsed training data for surrogate model

    """

    rx = re.compile(r'Node (?P<node>\d+)')

    data = []  # create an empty list to collect the data
    # open the file and read through it line by line
    with open(filepath, 'r') as f:
        node = ''
        for line in f:  # This keeps reading the file line by line

            # at each line check for a match with a regex of a node number
            match = rx.search(line)
```

```
            if match:
                node = match.group('node')
                next(f)
                next(f)
                line = f.readline()

                while line.strip(): # This keeps reading until end of table

                    data.append(line.split()[2])

                    line = f.readline()

        q_s = list(map(float, data))

        nn = int(node) # number of nodes in the model

        npts = len(q_s)//nn # number of training points

        xt = np.arange(0,npts+1)*step + start # training variables

        yt = np.array(q_s).reshape((nn,npts))

        yt = np.concatenate((yt[:,1][:,np.newaxis],yt), 1) # add extra point to opposite side to
             improove deriv prediction at 0 angle

        train_data = np.vstack([xt,yt]).T

    return train_data

def parse_cond(filepath='conductors.txt'):
    """
    Parses the Esatan conductor report file (produced by REPORT_CONDUCTORS procedure) to prepare
         user-defined conductor data for openmdao

    Parameters
    ----------
    filepath : str
        Filepath for report file to be parsed

    Returns
    -------
    data : list
        Parsed data

    """
    # set up regular expressions
    # use https://regexper.com to visualise these if required

    rx_dict = {
    'cond_name': re.compile(r'USER DEFINED CONDUCTOR = (?P<name>.*)\n'),
    'cond_type': re.compile(r'TYPE\s*= (?P<cond_type>.*)\n'),
    'nodes': re.compile(r'\[(?P<nodes>\d.*)\]'),
    'shape factor': re.compile(r'Shape Factor: (?P<sf>\d.\d*)'),
    'conductivity': re.compile(r'Conductivity: (?P<cond>\d.\d*)'),
    }

    def _parse_line(line):
        """
        Do a regex search against all defined regexes and
        return the key and match result of the first matching regex

        """

        for key, rx in rx_dict.items():
            match = rx.search(line)
            if match:
                return key, match
        # if there are no matches
        return None, None

    data = []  # create an empty list to collect the data
```

```
# open the file and read through it line by line
with open(filepath, 'r') as file_object:

    # Skips text before the beginning of the user links block:

    for line in file_object:
        if line.strip() == 'USER-DEFINED LINKS':
            break

    for line in file_object:  # This keeps reading the file

        # at each line check for a match with a regex
        key, match = _parse_line(line)

        # extract conductor name
        if key == 'cond_name':
            cond = match.group('name')

        # extract cond_type
        if key == 'cond_type':
            cond_type = match.group('cond_type')

        # extract nodes
        if key == 'nodes':

            nodes = []
            shape_factors = []
            values = []

            # read each line of the table until a blank line
            while line.strip():

                key, match = _parse_line(line.split('=')[0])

                node_pair = tuple(map(int, match.group('nodes').split(',')))

                key, match = _parse_line(line.split('=')[1])

                #extract shape factor

                SF = float(match.group('sf'))

                if SF == 0.0:
                    SF = 1.0 # override by user

                    k = line.split('=')[1].split(',')[2].split(':')[1].strip() # conductivity
                          equals user overriden value

                else:
                    key, match = _parse_line(line.split('=')[1].split(',')[1])
                    k = match.group('cond') # else use normal conductivity

                nodes.append(node_pair)
                shape_factors.append(SF)
                values.append(float(k))

                line = file_object.readline()

            # create a dictionary containing this conductor data
            entry = {
                'cond_name': cond,
                'cond_type': cond_type,
                'nodes': nodes,
                'SF': shape_factors,
                'values': values
            }
            # append the dictionary to the data list
            data.append(entry)

            line = file_object.readline() # This keeps reading the file
```

```
            # keeps reading until 'contact zone' block

            if line.strip() == 'CONTACT-ZONE LINKS':
                break

    return data

def idx_dict(src, ref):
    """
    Group node value indexes in src by corresponding node labels in ref

    """
    indices = defaultdict(list)

    for idx, val in enumerate(src):
        for name in ref:
            if val not in ref[name]:
                continue

            indices[name].append(idx)

    return indices

def parse_ar(filepath):
    """
    Parses Esatan radiative model report file and returns true (one-sided) areas of nodes

    """
    area = []
    nodes =[]
    with open(filepath, 'r') as f:
        #skip header lines
        for i in range(11):
            f.readline()

        line = f.readline()
        #read until end of table
        while line.strip():
            row = line.split('|')
            n = row[2].strip() # node number
            a = row[-1].strip() # node area
            area.append(a)
            nodes.append(n)
            f.readline() #skip every second line with empty data
            line = f.readline()
    area = list(map(float, area))
    nodes = list(map(int, nodes))
    area_n = np.array(nodes, dtype=int)
    area_val = np.array(area)
    data_raw = np.vstack((area_n, area_val))
    # remove duplicate entries (for two-sided nodes)
    data = np.unique(data_raw, axis=1)

    return data[1,:]
```

F.2. *GLmtxComp*

```
"""
Component for generating thermal model linear conductor matrix based on input parameters and
      esatan conductors.
This component takes thermal conductivities k as input variables and generates linear conductors
      GL = k * SF,
where SF is shape factor given by SF = A/L (A-crossectional area of conductor, L - length of
      conductor).
Initial GLs are to be provided from esatan model as option parameter GL_init
"""
import openmdao.api as om
import numpy as np

class GLmtxComp(om.ExplicitComponent):
    def initialize(self):
        self.options.declare('n', types=int, desc='number of diffusion nodes in thermal model')
        self.options.declare('GL_init', desc='initial conductor matrix from thermal model as n+1
             x n+1 array')
        self.options.declare('user_links', types=list, desc='list of user conductor data
             dictionaries')

    def setup(self):
        n = self.options['n'] + 1
        conductors = self.options['user_links']
        self.add_output('GL', shape=(n,n), units='W/K')

        for invar in conductors:
            name = invar['cond_name']
            nodes = invar['nodes']
            shape_factors = invar['SF']
            self.add_input(name) # adds input conductivity (scalar) with the same name as user
                 conductor name

            partials = np.zeros((n,n))

            for idx, SF in zip(nodes, shape_factors):
                partials[idx] = SF  # derivative of dGL/dk = SF
            # do the same as in compute function
            i_lower = np.tril_indices(n, -1)
            partials[i_lower] = partials.T[i_lower]
            di = np.diag_indices(n)
            diag = np.negative(np.sum(partials, 1))
            partials[di] = diag

            # note: we define sparsity pattern of constant partial derivatives, openmdao expects
                 shape (n*n, 1)
            flat_partials = partials.flatten()
            rows = np.nonzero(flat_partials)[0]
            values = flat_partials[rows]
            self.declare_partials('GL', name, rows=rows, cols=[0]*len(rows), val=values)

    def compute(self, inputs, outputs):
        n = self.options['n'] + 1
        GL = np.copy(self.options['GL_init'])
        conductors = self.options['user_links']

        for invar in conductors:
            name = invar['cond_name']
            nodes = invar['nodes']
            shape_factors = invar['SF']
            for idx, SF in zip(nodes, shape_factors):
                GL[idx] = SF * inputs[name]  # updates GL values based on input

        # mirror values from upper triangle to lower triangle
        i_lower = np.tril_indices(n, -1)
        GL[i_lower] = GL.T[i_lower]

        #define diagonal elements as negative of all node conductor couplings (sinks)
```

```
        di = np.diag_indices(n)
        GL[di] = np.zeros(n) # delete old result
        diag = np.negative(np.sum(GL, 1))
        GL[di] = diag

        GL[0,0] = 1.0 # deep space node temperature = 0 K (this coef is needed to avoid
              singularity in heat equations)

        outputs['GL'] = GL

    def compute_partials(self, inputs, partials):
        pass
```

F.3. *GRmtxComp*

```
"""
Component for assembling thermal model radiative exchange factor (REF) matrix based on input
      parameters and esatan model data.
REFs to deep space for each external surface node are calculated by GR = A * vf * eps * sigma,
      where A - area, eps - IR emissivity, vf - view factor to deep space.
Emissivity is taken as input parameter. Remaining REFs are imported from the model as option
      parameter GR_init
"""
import openmdao.api as om
import numpy as np

class GRmtxComp(om.ExplicitComponent):
    def initialize(self):
        self.options.declare('n', types=int, desc='number of diffusion nodes in thermal model')
        self.options.declare('GR_init', desc='initial REF matrix from thermal model as n+1 x n+1
             array')
        self.options.declare('faces', types=list, desc='names and optical properties of input
             faces')

    def setup(self):
        n = self.options['n'] + 1
        faces = self.options['faces']
        #VF = self.options['VF']
        #area = self.options['A']
        sigma = 5.670374e-8
        self.add_output('GR', shape=(n,n))
        for face in faces:
            self.add_input(face['name']) # adds input variable as face node group label

            rows = []
            derivs = []
            for i,node in enumerate(face['nodes']):
                deriv = face['areas'][i] * face['VFs'][i] * sigma # derivative of REF with
                     respect to this node emissivity
                rows.extend([node, node * n + node])
                derivs.extend([deriv, -1 * deriv ])

            self.declare_partials('GR', face['name'], rows=rows, cols=[0]*len(face['nodes'])*2,
                 val=derivs)

        # note: we define sparsity pattern of constant partial derivatives, openmdao expects
             shape (n*n, 1)

    def compute(self, inputs, outputs):
        n = self.options['n'] + 1
        GR = np.copy(self.options['GR_init'])
        faces = self.options['faces']
        sigma = 5.670374e-8

        for face in faces:
            GR[0, face['nodes']] = np.array(face['areas']) * np.array(face['VFs']) * sigma *
                 inputs[face['name']] # updates REFs to deep space based on input emissivity,
                 view factor and area

        #need to update diagonals

        di = np.diag_indices(n)
        GR[di] = np.zeros(n)
        diag = np.negative(np.sum(GR, 0))
        GR[di] = diag

        outputs['GR'] = GR

    def compute_partials(self, inputs, partials):
        pass
```

F.4. *Solar group*

```
import numpy as np
import os
from smt.surrogate_models import RMTB, RMTC
import openmdao.api as om
import smt
from Pre_process import parse_hf

class HeatFluxComp(om.ExplicitComponent):
    """
    Vectorized surrogate model to predict solar heat flux based on yaw angle
    Regularized minimal-energy tensor-product splines (RMTS) from SMT toolbox

    """
    def __init__(self, nodes, npts, model=None, method='RMTB'):
        super(HeatFluxComp, self).__init__()

        self.npts = npts # number of points (phi angles) at which to evaluate results

        self.ny = len(nodes) # number of outputs

        fpath = os.path.dirname(os.path.realpath(__file__))
        model_dir = fpath + '/Esatan_models/' + model
        train_data = parse_hf(filepath=model_dir+'/sdf.txt')

        xt = train_data[:,0]
        yt = train_data[:,nodes]

        xlimits = np.array([[xt[0], xt[-1]]])

        if method == 'RMTB':

            self.sm = RMTB(
                print_training=False,
                print_prediction=False,
                print_solver=False,
                xlimits=xlimits,
                energy_weight=1e-5,
                regularization_weight=1e-14,
                min_energy=True,
                num_ctrl_pts=50
            )
        else:
            self.sm = RMTC(
                print_training=False,
                print_prediction=False,
                print_solver=False,
                xlimits=xlimits,
                num_elements=15,
                energy_weight=1e-5,
                regularization_weight=1e-14,
                min_energy=True
            )

        self.sm.set_training_values(xt, yt)
        self.sm.train()

    def setup(self):

        m = self.npts
        ny = self.ny
        self.add_input('phi', val=np.zeros(m), units='deg' )
        self.add_output('q_s', val=np.zeros((m,ny)))
        self.declare_partials(of='*', wrt='*')

    def compute(self, inputs, outputs):

        y = self.sm.predict_values(inputs['phi'])
```

```
        outputs['q_s'] = np.absolute(y)  # sometimes surrogate predicts negative values that are
            close to zero, so need to take absolute here

    def compute_partials(self, inputs, partials):

        dy_dx = self.sm.predict_derivatives(inputs['phi'], 0)

        #need to fit into proper shape
        n = np.shape(dy_dx)[1]
        m = np.shape(dy_dx)[0]
        jac = np.zeros((m,n*m))
        idx = np.array([i for i in range(n)])

        for i in range(m):
            jac[i,[idx+n*i]] = dy_dx[i,:]

        partials['q_s', 'phi'] = jac.T


import openmdao.api as om
import numpy as np
from HeatFluxComp import HeatFluxComp

class Incident_Solar(om.ExplicitComponent):
    def initialize(self):
        self.options.declare('npts', default=1, types=int, desc='number of points')
        self.options.declare('areas', desc='1D array of areas of surface nodes')
    def setup(self):
        n = len(self.options['areas'])
        m = self.options['npts']
        self.add_input('dist', val=np.ones(m), desc='distane in AU')
        self.add_input('q_s', val=np.zeros((m,n)))
        self.add_output('QIS', val=np.ones((n, m)), units='W')

        self.A = self.options['areas']

        rows = np.arange(n*m)
        cols1 = np.arange(m)[np.newaxis].repeat(n,axis=0).flatten()
        cols2 = np.arange(n*m).reshape((m,n)).flatten(order='F')
        self.declare_partials(of='QIS', wrt='dist', rows=rows, cols=cols1)
        self.declare_partials(of='QIS', wrt='q_s', rows=rows, cols=cols2)

    def compute(self, inputs, outputs):

        n = len(self.options['areas'])
        m = self.options['npts']
        d = inputs['dist']
        q_s = inputs['q_s']

        QIS = np.zeros((n,m))
        for i in range(m):
            QIS[:,i] = q_s[i,:] * self.A * d[i]**(-2) # incident solar power at distance d at
                each point

        outputs['QIS'] = QIS

    def compute_partials(self, inputs, partials):

        n = len(self.options['areas'])
        partials['QIS', 'dist'] = (-2 * inputs['q_s'] * (inputs['dist'][:,np.newaxis])**(-3) *
            self.A).flatten('F')
        partials['QIS', 'q_s'] = (((inputs['dist'][:,np.newaxis]).repeat(n,axis=1))**(-2) * self.
            A).flatten('F')

class SolarPower(om.ExplicitComponent):

    def initialize(self):
        self.options.declare('n_in', types=int, desc='number of input nodes')
        self.options.declare('npts', default=1, types=int, desc='number of points')
        self.options.declare('alp_sc', default=.91, lower=.0, upper=1., desc='absorbtivity of the
```

```
                solar cell' )

    def setup(self):
        n = self.options['n_in']
        m = self.options['npts']

        self.add_input('QIS', shape=(n,m), desc='incident solar power', units='W')
        self.add_input('alp_r', shape=(n,1), desc='solar absorbtivity of the input node radiating
              surface')
        self.add_input('cr', shape=(n,1), desc='solar cell or radiator installation decision for
             input nodes')
        self.add_output('QS_c', shape=(n,m), desc='solar cell absorbed power over time', units='W
             ')
        self.add_output('QS_r', shape=(n,m), desc='radiator absorbed power over time', units='W')

        rows = np.arange(n*m)
        cols1 = rows
        cols2 = np.arange(n).repeat(m)

        self.declare_partials(of='QS*', wrt='QIS', rows=rows, cols=cols1)
        self.declare_partials(of='QS*', wrt='cr', rows=rows, cols=cols2)
        self.declare_partials(of='QS_r', wrt='alp_r', rows=rows, cols=cols2)
        self.declare_partials(of='QS_c', wrt='alp_r', dependent=False)

    def compute(self, inputs, outputs):

        alp_sc = self.options['alp_sc']

        QIS = inputs['QIS']
        alp_r = inputs['alp_r']
        cr = inputs['cr']

        outputs['QS_c'] = QIS * alp_sc * cr
        outputs['QS_r'] = QIS * alp_r * (1 - cr)

    def compute_partials(self, inputs, partials):

        alp_sc = self.options['alp_sc']
        m = self.options['npts']

        partials['QS_c', 'QIS'] = alp_sc * inputs['cr'].repeat(m)
        partials['QS_r', 'QIS'] = (inputs['alp_r'] * (1-inputs['cr'])).repeat(m)
        partials['QS_c', 'cr'] = alp_sc * inputs['QIS'].flatten()
        partials['QS_r', 'cr'] = (-inputs['alp_r'] * inputs['QIS']).flatten()
        partials['QS_r', 'alp_r'] = ((1 - inputs['cr']) * inputs['QIS']).flatten()

class Solar(om.Group):
    def __init__(self, npts, areas, nodes, model):
            super(Solar, self).__init__()

            self.npts = npts # number of points
            self.areas = areas # areas input nodes
            self.nodes = nodes # input node numbers
            self.n = len(nodes) # number of input external surface nodes
            self.model = model #Esatan radiative model name


    def setup(self):

        self.add_subsystem('hf', HeatFluxComp(nodes=self.nodes, npts=self.npts, model=self.model)
              , promotes=['*'])
        self.add_subsystem('is', Incident_Solar(npts=self.npts, areas=self.areas), promotes
             =['*'])
        self.add_subsystem('sol', SolarPower(n_in=self.n, npts=self.npts), promotes=['*'])
```

F.5. *Thermal Group*

```
import openmdao.api as om
import numpy as np

class TempsComp(om.ImplicitComponent):
    """Computes steady state node temperatures over multiple design points."""
    def initialize(self):
        self.options.declare('n', default=1, types=int, desc='number of diffusion nodes')
        self.options.declare('npts', default=1, types=int, desc='number of points')
    def setup(self):
        n = self.options['n'] + 1
        m = self.options['npts']
        self.add_output('T', val=np.zeros((n,m)), units='K')
        self.add_input('GL', val=np.zeros((n,n)), units='W/K')
        self.add_input('GR', val=np.zeros((n,n)))
        self.add_input('QS', val=np.zeros((n,m)), units='W')
        self.add_input('QI', val=np.zeros((n,m)), units='W')
        seq = np.arange(n*m)
        nn = np.arange(n*n).reshape((n,n))
        rows = seq.repeat(n)
        cols = np.repeat(nn,m,axis=0).flatten()
        self.declare_partials(of='T', wrt='G*', rows=rows, cols=cols)
        self.declare_partials(of='T', wrt='Q*', rows=seq, cols=seq, val=1.)
        self.declare_partials(of='T', wrt='T')

    def apply_nonlinear(self, inputs, outputs, residuals):
        GL = inputs['GL']
        GR = inputs['GR']
        QS = inputs['QS']
        QI = inputs['QI']
        T = outputs['T']

        residuals['T'] = GL.dot(T) + GR.dot(T**4) + QS + QI

    def linearize(self, inputs, outputs, partials):
        n = self.options['n'] + 1
        m = self.options['npts']
        GL = inputs['GL']
        GR = inputs['GR']
        T = outputs['T']

        partials['T', 'GL'] = np.tile(T.T,(n,1)).flatten()
        partials['T', 'GR'] = np.tile((T**4).T,(n,1)).flatten()
        partials['T', 'T'] = np.einsum('ik, jl', GL, np.eye(m, m)) + 4.0 * T**3 * np.einsum('ik,
             jl', GR, np.eye(m, m))

    def guess_nonlinear(self, inputs, outputs, residuals):
        n = self.options['n'] + 1
        m = self.options['npts']
        #gues values
        outputs['T'] = -np.ones((n,m))*50 + 273


import openmdao.api as om
import numpy as np
from TempsComp import TempsComp

class SolarCell(om.ExplicitComponent):
    def initialize(self):
        self.options.declare('nodes', desc='list of input external surface node numbers')
        self.options.declare('npts', default=1, types=int, desc='number of points')
        self.options.declare('alp_sc', default=.91, lower=.0, upper=1., desc='absorbtivity of the
              solar cell' )

    def deta_dT(self): # derivative value of efficiency wrt temperature
        alp_sc = self.options['alp_sc']
        T0 = 28. #reference temperature
        eff0 = .285 #efficiency at ref temp
        T1 = -150.
```

```
        eff1 = 0.335
        slope = (eff1 - eff0) / (T1 - T0) /alp_sc
        return slope

    def setup(self):
        nodes = self.options['nodes']
        n = len(nodes)
        m = self.options['npts']

        idx_list = [[(i,j) for j in range(m)] for i in nodes]

        self.add_input('T', val=np.ones((n,m))*28., src_indices=idx_list, units='degC')
        self.add_output('eta', val=np.ones((n,m))*0.3/0.91, desc='solar cell efficiency with
            respect to absorbed power for input surface nodes over time ')
        rows = np.arange(n*m)
        cols = rows
        self.declare_partials(of='eta', wrt='T', rows=rows, cols=cols, val=self.deta_dT())

    def compute(self, inputs, outputs):

        alp_sc = self.options['alp_sc']

        T0 = 28. #reference temperature
        eff0 = .285 #efficiency at ref temp

        delta_T = inputs['T'] - T0

        outputs['eta'] = eff0/alp_sc + self.deta_dT() * delta_T

    def compute_partials(self, inputs, partials):
        pass

class ElectricPower(om.ExplicitComponent):
    def initialize(self):
        self.options.declare('nodes', desc='list of input external surface node numbers')
        self.options.declare('npts', default=1, types=int, desc='number of points')
        self.options.declare('ar', default=.8, lower=.0, upper=1., desc='solar cell to node
            surface area ratio')
        self.options.declare('eta_con', default=.95, lower=.0, upper=1., desc='MPPT converter
            efficiency')

    def eta_in(self): # input path efficiency (mppt losses * area losses)
        ar = self.options['ar']
        eta_con = self.options['eta_con']
        eta_in = eta_con * ar
        return eta_in

    def setup(self):
        n = len(self.options['nodes'])
        m = self.options['npts']

        self.add_input('eta', val=np.ones((n,m))*0.3/0.91, desc='solar cell efficiency with
            respect to absorbed power for input surface nodes over time ')
        self.add_input('QS_c', shape=(n,m), desc='solar cell absorbed power over time', units='W
            ')
        self.add_output('P_el', shape=(n,m), desc='Electrical power output over time', units='W')
        rows = np.arange(n*m)
        cols = rows
        self.declare_partials('P_el', 'eta', rows=rows, cols=cols)
        self.declare_partials('P_el', 'QS_c', rows=rows, cols=cols)

    def compute(self, inputs, outputs):

        eta = inputs['eta'] * self.eta_in() # total efficiency = cell eff * input path eff
        QS = inputs['QS_c']

        outputs['P_el'] = np.multiply(QS, eta)

    def compute_partials(self, inputs, partials):

        n = len(self.options['nodes'])
```

```
        m = self.options['npts']
        partials['P_el','eta'] = (inputs['QS_c'] * self.eta_in()).reshape(n*m,)
        partials['P_el', 'QS_c'] = (inputs['eta'] * self.eta_in()).reshape(n*m,)

class QSmtxComp(om.ExplicitComponent):
    def initialize(self):
        self.options.declare('nn', types=int, desc='number of diffusion nodes in thermal model')
        self.options.declare('nodes', desc='list of input external surface node numbers')
        self.options.declare('npts', default=1, types=int, desc='number of points')

    def setup(self):
        nn = self.options['nn'] + 1
        nodes = self.options['nodes']
        n = len(nodes)
        m = self.options['npts']
        self.add_input('P_el', shape=(n,m), desc='solar cell electric power over time', units='W
            ')
        self.add_input('QS_c', shape=(n,m), desc='solar cell absorbed heat over time', units='W')
        self.add_input('QS_r', shape=(n,m), desc='radiator absorbed heat over time', units='W')
        self.add_output('QS', val=np.zeros((nn,m)), desc='solar absorbed heat over time', units='
            W')

        y = np.arange(nn*m).reshape((nn,m))
        rows = y[nodes,:].flatten()
        cols = np.arange(n*m)
        self.declare_partials('QS', 'P_el', rows=rows, cols=cols, val=-1.)
        self.declare_partials('QS', 'QS_c', rows=rows, cols=cols, val=1.)
        self.declare_partials('QS', 'QS_r', rows=rows, cols=cols, val=1.)

    def compute(self, inputs, outputs):
        nn = self.options['nn'] + 1
        m = self.options['npts']
        QS = np.zeros((nn,m))
        P_el = inputs['P_el']
        QS_c = inputs['QS_c']
        QS_r = inputs['QS_r']
        nodes = self.options['nodes']
        for i,node in enumerate(nodes):
            QS[node,:] = QS_c[i,:] + QS_r[i,:] - P_el[i,:] # energy balance
        outputs['QS'] = QS

    def compute_partials(self, inputs, partials):
        pass

class Thermal_Cycle(om.Group):
    def __init__(self, nn, npts, nodes):
            super(Thermal_Cycle, self).__init__()

            self.nn = nn
            self.npts = npts
            self.nodes = nodes

    def setup(self):

        nodes = self.nodes

        self.add_subsystem('sc', SolarCell(nodes=nodes, npts=self.npts), promotes=['*'])
        self.add_subsystem('el', ElectricPower(nodes=nodes, npts=self.npts), promotes=['*'])
        self.add_subsystem('QS', QSmtxComp(nn=self.nn, nodes=nodes, npts=self.npts), promotes
            =['*'])
        self.add_subsystem('temps', TempsComp(n=self.nn, npts=self.npts), promotes=['*'])

        self.nonlinear_solver = om.NewtonSolver(solve_subsystems=False)
        self.nonlinear_solver.options['iprint'] = 2
        self.nonlinear_solver.options['maxiter'] = 15
        self.nonlinear_solver.linesearch = om.ArmijoGoldsteinLS()
        self.nonlinear_solver.linesearch.options['maxiter'] = 10
        self.nonlinear_solver.linesearch.options['iprint'] = 2
        self.linear_solver = om.DirectSolver(assemble_jac=True)
        self.options['assembled_jac_type'] = 'csc'
```