Annex 1: EGSnrc Data Generation and Surrogate Model Training

```
#!/bin/sh

edit_sweep_main  () {
  # Ncase is dealt with in ./sweep_main.sh
  #       > necessary to have only a single variable input for
  #       parallelization.
  ncase=$1
  sigma=$2
  cp ./misc/sweep_main.sh.backup sweep_main.sh
  sed -i "s/ncase_main_placeholder/$ncase/" sweep_main.sh
  sed -i "s/sigma_main_placeholder/$sigma/" sweep_main.sh

}

date >> surrogate_optimization_prep.txt

ncase=$1
ncase_bk=$(cat .default_ncase)
ncase=${ncase:=$ncase_bk}

sigma=$2
sigma_bk=$(cat .default_sigma)
sigma=${sigma:=$sigma_bk}

N_sample=$3
N_sample_bk=$(cat .default_Nsample)
N_sample=${N_sample:=$N_sample_bk}

edit_sweep_main $ncase $sigma

dirname="run_-_ncase_$ncase""_-_sigma_$sigma""_-_Nsample_$N_sample"
#$(date -I)_-_$(ls -1 | grep $(date -I) | wc -l))
echo $dirname
```

```
mkdir $dirname

# Write destination dir for sweep_main.sh
echo "$dirname" > most_recent_run.txt

mkdir $dirname/og_results/
mkdir $dirname/og_results/csv_dir
mkdir $dirname/og_results/input_test_dir
mkdir $dirname/og_results/egsinp_dir
mkdir $dirname/og_results/egsdat_dir
mkdir $dirname/og_results/egslog_dir
mkdir $dirname/og_results/include_dir
mkdir $dirname/og_results/ptracks_dir
mkdir $dirname/og_results/mederr_dir
mkdir $dirname/og_results/input_log_dir


# Collect random numbers
N_Mat=$(./misc/Get_N_Mat.sh)
touch random_numbers
rm random_numbers
for n in $(seq $N_sample)
do
        rn=$(python ./rand.py $N_Mat)
        echo $rn >> random_numbers
done

# Keep only distinct random numbers
sort -un random_numbers -o random_numbers

# For each random number, run simulation
#for rn in $(cat random_numbers)
#do
#  ./sweep_main.sh $rn
#  sbatch --partition=lpmoonracer \
#               ./sweep_main.sh $rn #$xs $ys $zs $calc_n
#done
```

```
# In parallel, pass each number to sweep_main
cat random_numbers | parallel ./sweep_main.sh
mv random_numbers $dirname

cp include* $dirname/og_results/

cd $dirname/og_results/csv_dir
cp ../../../egs_data_collect.py ./

# Collect data
python ./egs_data_collect.py
rm egs_data_collect.py
mv *.hdf5 ../..
cd ../..

## Copy template input-file for record.
input_file_name=$(echo scatter-learn-placeholder_-_"$dirname".egsinp \
                  | sed 's|\/\.egsinp|.egsinp|')
cp ../scatter-learn-placeholder.egsinp ./"$input_file_name"

# Return to main, and train new surrogate model
cd ..
python MakeMod.py $dirname $ncase $sigma $N_sample

## Copy-in other simulation setup. Both for visulization and #
#  for record.
date >> surrogate_optimization_prep.txt
```

Annex 2: Scripting Single EGSnrc Simulation

```
#!/bin/sh

GenModId () {
  rand_num=$1
  ith_run=$(ls -1 | \
            grep "$rand_num" | \
            grep "detector_results.csv" | \
            wc -l)
  echo $ith_run
}

## Recieve vars
rand_num=$1

# Set variables to defaults if null
ncase=$2
ncase_bk=$(cat .default_ncase)
ncase=${ncase:=$ncase_bk}

sigma=$3
sigma_bk=$(cat .default_sigma)
sigma=${sigma:=$sigma_bk}

id_mod=$4
#it_turn=$2
id_mod_bk=$(GenModId $rand_num)
id_mod=${id_mod:=$id_mod_bk}

## Define id and file names
if test -n $id_mod
then
        id="num_$id_mod-$rand_num"
else
        id="num_$rand_num"
```

```
fi

input_file=$(echo "scatter-learn_-_"$id".egsinp")

## Make geometric mask in include-scatter-media_-_$$.dat file
geo_mask_file=scatter-learn_-_"$id"_-_include_scatter_media.dat
./generate_scatter_masks.py $rand_num > $geo_mask_file

# Create random seeds
rnd_seeds=$(./make_rnd_seed.sh 2)

## Make and edit custom input file to include new
#  media definitions
## Fails off
cp scatter-learn-placeholder.egsinp $input_file
sed -i "s/placeholder_id/$id/" $input_file
sed -i "s/placeholder_ncase/$ncase/" $input_file
sed -i "s/placeholder_rndseed/$rnd_seeds/" $input_file
sed -i "s/placeholder_sigma/$sigma/" $input_file

## Fails on
#cp scatter-learn.egsinp $input_file
#sed -i "s/ncase\ =\ .*/ncase = $ncase/" $input_file
#sed -i "s/scatter-media\scatter-media_-_$id/" $input_file

## Run scatter-learn,
scatter-learn -i $input_file > scatter-learn_-_$id.egslog

## parses outputs (of detector) into .csv
./output_parse_-_detector.sh scatter-learn_-_$id.egslog

## parses ~outputs~ (scatter input) into .csv
output_file=scatter-learn_-_"$id"_-_include_scatter_media.dat
./output_parse_-_scatter_media.sh $output_file
#
#./snapshot_EGSview.sh $rand_num $it_turn
echo $rand_num $id
```

```
## CLEAN-UP MOVED. ##
#
## leave as csv?
## dirname=$(ls -rt1 | grep $(date -I) | tail -n 1)
#dirname=$(cat most_recent_run.txt)
#cd $dirname
#cd og_results
#mv ../../scatter-learn_-_"$id"_-_detector_results.csv csv_dir
#mv ../../scatter-learn_-_"$id"_-_scatter_input.csv csv_dir
#mv ../../input_log_test_-_"$rand_num".csv csv_dir
#mv ../../$input_file egsinp_dir
#mv ../../scatter-learn_-_$id.egsdat egsdat_dir
#mv ../../scatter-learn_-_$id.egslog egslog_dir
#mv ../../scatter-learn_-_"$id"_-_include_scatter_media.dat include_dir
##mv ../../scatter-learn_-_$id.ptracks ptracks_dir
#rm ../../scatter-learn_-_$id.ptracks
#mv ../../scatter-learn_-_$id.mederr mederr_dir
#cd ../..
```

Annex 3: Data Processing and Surrogate Model Training

```
#from sklearnex import patch_sklearn
#patch_sklearn()
import numpy as np
import matplotlib as mpl
#mpl.use('Agg')
import matplotlib.pyplot as plt
from tqdm import tqdm
import random
import sklearn as sk
import sys
import os
import time
sys.path.append(os.getcwd())

#from scope_pattern import unscope_single
from LoadRecentData import LoadRecentData

random_seed = 12344
np.random.seed(random_seed)
random.seed(random_seed)

# Padding zeros for scope context
#try:
#    npad = sys.argv[2]
#    npad = int(npad)
#except:
#    npad = 2
    ## npad of 2 gives a input (scope) of 3x3.
    #
    #   - improvement in performance
```

```
    #     shows that deviation depends primarily on
    #     nearest neighbours. (6 nn)
    #
    #       > why would MORE infomation
    #         reduce accuracy???
npad = 2
try:
    run_directory = sys.argv[1]
    ncase= sys.argv[2]
    ncase= int(ncase)
    sigma=sys.argv[3]
    sigma = float(sigma)

    N_sample=sys.argv[4]
    N_sample = int(N_sample)
    print(f"ncase:{ncase}",
          f"sigma:{sigma}",
          f"N_sample:{N_sample}")

    dc, sid, mats, \
    ncase_bk, sigma_bk = LoadRecentData(
                             data_path=run_directory,
                             history_index=0
    )
    if ncase != ncase_bk:
        raise ValueError

except IndexError:
    run_directory=None
    run_directory="_-_".join((
                    "run",
                    "ncase_10000",
                    "sigma_0.1",
                    "Nsample_1000"))
```

```
    dc, sid, mats, \
    ncase, sigma = LoadRecentData(data_path=run_directory,
                                  history_index=0)
    N_sample = dc.shape[0]


N = dc.shape[0]
xdim = round(sid.shape[1]**(1/3))

# Frame, so that the no scope doesn't contain at least
# one pixel from ROI
f = npad+1

from ProcessInput import ProcessInput
ssc, sid = ProcessInput(sid, npad)

from TrainMod import SplitTestTrain, TrainModel

# For scoped models
input_data_train, \
input_data_test, \
tri, tei = SplitTestTrain(ssc, N)

# For un-scoped net.
#input_data_train, \
#input_data_test, \
#tri, tei = SplitTestTrain(sid,N)

# Choose the amount of data to use.
#each_nth = min((20, tri.__len__(), tei.__len__()))//2
#tri = tri[::each_nth]
#tei = tei[::each_nth]
```

```
#Reshape scope objects to N*m*m*f*f.
# Easier to select relevant data only..
from ProcessLabelData import ProcessLabelData
dr = dc[tri]
de = dc[tei]

# Test/train split is done at the per-mask level.
#   > Overlap between scoped views possible
#lv, lv_min, lv_minmax = ProcessLabelData(dc)
#labr = lv[tri].flatten()
#le = lv[tei].flatten()

#labr_min = 0
#labr_minmax = 1

# Normalize training data.
labr, labr_min, labr_minmax = ProcessLabelData(dr)

# Normalize test data using the same values
labe = ProcessLabelData(de,
                        lv_min=labr_min,
                        lv_minmax=labr_minmax)

labr = labr.flatten()
labe = labe.flatten()

## HYPER-PARAMETER OPTIMIZATION

# Cost-complexity was optimized for.
# Stacked Bootrap
#ccp_alpha = 0.022543969013557597

# Random Forest Regressor w/ [-1,1] normalization w/ mean.
#ccp_alpha = 0.00021876
```

```
# RFR w/ 0-1 normalization using -min
ccp_alpha = 0.0001329

#from TrainMod import HyperParameterOptimizer
#Optimizer = HyperParameterOptimizer()
#ccp_optif = lambda x: 1 - Optimizer.TestVal(input_data_train,
#                                            labr,
#                                            input_data_test,
#                                            le,
#                                            ccp_alpha=x)[1]
#from scipy import optimize as optim
#bnds = [[0,1]]
#ccp_opt = optim.minimize(ccp_optif,
#                         ccp_alpha,
#                         bounds=bnds)
#ccp_alpha = Optimizer.OptimalValue()[0]



## FOR SCOPED RFR SURROGATE
#mod = TrainModel(input_data_train[::10], labr[::10],
mod = TrainModel(input_data_train, labr,
                     ccp_alpha=ccp_alpha)

## FOR NON-SCOPED CNN SURROGATE
#from TrainNet import TrainNet
#mod = TrainNet(input_data_train, labr)



r2 = mod.score(input_data_test, labe)
oe = mod.predict(input_data_test)
```

```
#from ResidualModel import ResidualModel
#base_rfr = sk.ensemble.RandomForestRegressor(
#                                  ccp_alpha=ccp_alpha
#)
#res_rfr = sk.ensemble.RandomForestRegressor()
#res_mod = ResidualModel(base_mod=base_rfr,
#                        residual_mod = res_rfr)
#res_mod.fit(input_data_train, labr)

print("DONE TRAINING")
print("r2: ",r2)

test_label_image = labe.reshape((-1, xdim, xdim))
test_output_image = oe.reshape((-1, xdim, xdim))

img_ind = np.random.randint(0,len(sid[tei]))

try:
#if True:
    fig, (ax1, ax2, ax3,ax4) = plt.subplots(1, 4,figsize=(20,5))
    #fig, (ax3,ax4) = plt.subplots(1, 2,figsize=(20,5))
    fig.suptitle("".join((
                        "Voxelized Model Stages Testing: ",
                        f"{round(r2,3)}"))
    )

    from ColorbarFuncs import MakeMaterialsColorbar
    mats_norm, mats_cmap, \
    mats_mappable, mats_ticks = MakeMaterialsColorbar(
                                            input_data_train,
                                            input_data_test,
                                            fig)
                                            #cbar1_ax)
    #cb1.set_ticklabels(mats)
```

```
# Input image
input_data_image_width = input_data_test.shape[-1]
input_data_image_width = np.sqrt(input_data_image_width)
input_data_image_width = int(input_data_image_width)
inp_date_plot = input_data_test.reshape(
                                  (-1,
                                   input_data_image_width,
                                   input_data_image_width)
)
#inp_date_plot = input_data_test.squeeze()
assert inp_date_plot.shape[-1] == (npad + 1)
inp_image = np.vstack(
               [np.hstack(
                 [inp_date_plot[j*xdim + i] \
                            for i in range(xdim)])
                                  for j in range(xdim)])
                 #[inp_date_plot[img_ind][j][i] \
                 #           for i in range(npad+1)]) \
                 #                  for j in range(npad+1)])
ix = f//2 + 1
insh = inp_image.shape[0]
lines = np.dstack((
            np.vstack((np.arange(0,insh,f)+0.5,
                       np.zeros(insh//f))).T-0.5,
            np.vstack((np.arange(0,insh,f)+0.5,
                       np.ones(insh//f)*insh)).T-0.5
))

diff = (test_label_image - test_output_image)


for line_i in lines:
```

```
            ax2.plot(line_i[0,:]-0.5, #in y
                     line_i[1,:],
                     color='w')
            ax2.plot([-0.5,insh], # in x
                      np.array([1,1])*(ix)+0.5,
                      color='w')
            ix += f
        #
        #inp_image = np.hstack(
        #                [np.vstack(xs_i) for xs_i \
        #                        in inp_date_plot[0]]
        #)

        ax2.imshow(inp_image,
                   norm=mats_norm,
                   cmap=mats_cmap)

        ax2.set_title("input")

        mats_images = (ax1, ax2)
    #else:
    except Exception as e:
        print(f"Exception: {e}")
        # Close failed plot attempt
        plt.close()


        fig, (ax1,ax3,ax4) = plt.subplots(1,
                                          3,
                                          figsize=(20,5))

        fig.suptitle("".join(("CNN Testing: ",
                              str(round(r2,3))
                        ))
```

```
    )

    #cbar1_ax = fig.add_axes([0.15,0.045,0.25,0.05])

    from ColorbarFuncs import MakeMaterialsColorbar
    mats_norm, mats_cmap, \
    mats_mappable, mats_ticks = MakeMaterialsColorbar(
                                            input_data_train,
                                            input_data_test,
                                            fig)
    #cb1.set_ticklabels(mats)

    mats_images = [ax1]


# Object image
ax1.set_title("object")
im1 = ax1.imshow(sid[tei][img_ind],
                 norm=mats_norm,
                 cmap=mats_cmap)

# Untested
from ProcessCSV import GetSharedCBPos
cbar1_ax = GetSharedCBPos(mats_images)
cb1 = fig.colorbar(mats_mappable,
                   cax=cbar1_ax,
                   orientation="horizontal")
cb1.set_ticks(mats_ticks)
cb1.set_ticklabels(mats)
cb1.set_label("Material")

# Plot Output images.
from ProcessCSV import MakeCommonColorbar
target_norm, target_cmap, \
```

```
target_mappable = MakeCommonColorbar(
                        test_label_image[img_ind],
                        test_output_image[img_ind])

ax3.set_title("Prediction")
im3 = ax3.imshow(test_output_image[img_ind],
                 norm=target_norm,
                 cmap=target_cmap)

ax4.set_title("Label")
im4 = ax4.imshow(test_label_image[img_ind],
                 norm=target_norm,
                 cmap=target_cmap)

# Set position of color bar
cbar2_ax = GetSharedCBPos((ax3, ax4))
#cbar2_ax = fig.add_axes([0.475,0.045,0.38,0.05])

# Write Colorbar
cb2 = fig.colorbar(target_mappable,
                   cax=cbar2_ax,
                   orientation="horizontal")
cb2.set_label(r"Normalized␣Absorbed␣Dose␣[$Gy\cdot{cm}^2$]",
              loc='center')

# Plot difference plot
#ax5.set_title("difference")
#im5 = ax5.imshow(diff[img_ind],
#                 norm=target_norm,
#                 cmap=target_cmap)
##cbar3_ax = fig.add_axes([0.774,0.045,0.12,0.05])
##cb3 = fig.colorbar(im5,
#                   cax=cbar3_ax,
#                   orientation="horizontal")
```

```
##fig.savefig("rfr_modes_-_diff.png")


image_name= "_-_".join((f"work_flow",
                        f"ncase_{ncase}",
                        f"sigma_{sigma}",
                        f"Nsample_{N_sample}.png"))

fig_size = fig.get_size_inches()
fig.set_size_inches(fig_size[0],
                    h=fig_size[1]*1.25,
                    forward=True)
#fig.savefig(image_name)
plt.show()
plt.close()


## PREDICTED VS. EXPECTED ##
plt.figure()
plt.scatter(labe, oe)
plt.title(
    f"Predicted␣vs.␣Labelled␣Normalized␣Dose:␣{round(r2,␣3)}"
)
plt.xlabel("Label")
plt.ylabel("Prediction")

range_min = min(np.sign(np.min(labr)),0)
range_max = 1

plt.xlim([range_min, range_max])
plt.ylim([range_min, range_max])

line = np.linspace(range_min, range_max, 100)
plt.plot(line, line, c='k')
```

```
evp_name= "_-".join((
        "evp",
        f"ncase_{ncase}",
        f"sigma_{sigma}",
        f"Nsample_{N_sample}.png"
))
#plt.savefig(evp_name)
plt.show()
#plt.close()

## Save Mod
from pickle import dump as pickle_dump

mod.labr_min = labr_min
mod.labr_minmax = labr_minmax
#mod_name = "_-_".join((
#                    "mod",
#                    f"ncase_{ncase}",
#                    f"sigma_{sigma}",
#                    f"Nsample_{N_sample}.pkl"
#))
mod_name = "trained_model.pkl"
with open(mod_name,'wb') as fil:
    pickle_dump(mod, fil)

## Worth noting:
#  maximum values originating from where central voxel is 0.
#   > The max of this is undoubtably 1
#   > The mean is very close to the np.max(oe)
#       * since predictions end up being based on /average/
#         values under that type of pixel.
#zero_centrals= labr[np.dstack(
#                      np.where(input_data_train[:,4] == 0)
```

```
#)]



#with open("variety_records.txt",'a') as fil:
def GetModelType(mod):
    model_type = mod.__class__
    model_type = str(model_type)
    model_type = model_type.split("\'")
    model_type = model_type[1]

    model_type = model_type.split(".")
    model_type = model_type[-1]
    #model_type = mod.__class__()
    #model_type = str(model_type)
    #model_type = model_type.replace("()",'')
    return model_type

model_type = GetModelType(mod)
#with open(f"{model_type}_r2.csv",'a') as fil:
#    fil.write(f"{ncase},{sigma},{N_sample},{r2}")
#    fil.write("\n")
#
def PlotAllObjs():
    for i in range(sid.shape[0]):
        fig, axes = plt.subplot_mosaic("AB")
        axes["A"].imshow(sid[i])
        axes["B"].imshow(lv[i])
        fig.show()
#si, single_output_image = FullProcessSingle(csv_path,
#                                             mod=mod)
#PlotSinglePair(csv_path, mod=mod)
```

Annex 4: Simulated Annealing Script

```
import numpy as np
import matplotlib.pyplot as plt

from ProcessInput import ProcessInput
from ProcessLabelData import ProcessLabelData
from TrainMod import SplitTestTrain
from LoadModel import LoadModel, InferNPad
from PyEGSnrc import EGSnrcCall
from SimulatedAnnealer import *


# Load model
mod = LoadModel()
npad = InferNPad(mod)
f = npad+1

# Get Data
try:
    import sys
    rn = sys.argv[1]

except IndexError:
    # Toy goal
    from CollectSingleResults import CollectSingleResults
    rand_num = 432594882729282400242400315310082#0
    sid, dc = CollectSingleResults(rand_num=rand_num)

    # Most recent
#   from LoadRecentData import LoadRecentData
#   dc, sid, mats = LoadRecentData()
```

```
# Process Labels for goal
lv, lv_min, lv_minmax = ProcessLabelData(dc)
ssc, sid = ProcessInput(sid, npad)

# Get parameters
N = lv.shape[0]
n = 8
sq = 8


# DEV: THE PRE-TRAINED MODEL WILL NOT HAVE
#      BEEN TRAINED ON THIS TEST/TRAIN
#      SPLIT
#input_data_train, \
#input_data_test, \
#tri, tei = SplitTestTrain(ssc, N)

# Init Scoper
from ProcessInput import VoxScoper
S = VoxScoper(npad)
scope=lambda s: np.array(S.ScopeObj(s,sq)).reshape((-1,f*f))


# Define goal
ideal_solution = sid[0]
scoped_solution = scope(ideal_solution)#, sq, npad)

# Find prediction on ideal design
goal = mod.predict(scoped_solution)


# Guess Input
material_numbers = np.unique(ideal_solution)
design = np.random.choice(material_numbers,
```

```
                            ideal_solution.shape)
# Clear padding
design[:2,:] = 0
design[-2:,:] = 0
design[:,:2] = 0
design[:,-2:] = 0


# For optimization using a surrogate model
Evaluator = SurrogateCall(mod, scope)

# For Bruteforce optimization
#Evaluator = EGSnrcCall(lv_min, lv_max, N_mat, npad)


# Calculate initial score
iter_turn = 0
score_initial = score_func(design,
                           goal,
                           Evaluator,
                           iter_turn)

# Give non-sensical itteration value for ideal.
iter_turn = "inf"
score_ideal = score_func(ideal_solution,
                         goal,
                         Evaluator,
                         iter_turn)
del iter_turn # clear value

print("initial:", score_initial)
print("ideal:", score_ideal)
```

```
## OPTIMIZATION ##

# Init optimizer
Opti = SimulatedAnnealer(design,
                         goal,
                         Evaluator,
                         npad,
                         sq,
                         material_numbers)
prob_l = Opti.main()

# Execute operation and return list of probabilities
prob_l = list(prob_l)

print("OPTIMIZATION␣DONE")
print(f"time␣taken:␣{Opti.time_taken}")
##  END OF OPTIMIZATION  ##

##  PLOTTING  ##

# Figure layout
fig, ax = plt.subplot_mosaic([list("abcc"),list("decc")])


# Ideal solution and performance
ideal_solution_scoped = scope(ideal_solution)
ideal_performance = mod.predict(ideal_solution_scoped)
ideal_performance = ideal_performance.reshape((n,n))
ideal_score = score_func(ideal_solution,
                         goal,
                         Evaluator,
                         Opti.iter_turn)
ideal_score = round(ideal_score, 1)
```

```
ax['a'].imshow(ideal_solution)
ax['a'].set_title("Ideal Design")#+str(si))
ax['b'].imshow(ideal_performance)
ax['b'].set_title(
          f"Predicted Ideal Performance: MSE = {ideal_score}"
)


# Goal performance
goal_image = goal.reshape((n, n))
ax['c'].imshow(goal_image)
ax['c'].set_title("Simulation Result")


# Best performance
best_solution = Opti.best_solution
ax['d'].imshow(best_solution)
ax['d'].set_title("Best Design")

best_solution_scoped = scope(best_solution)
best_performance = mod.predict(best_solution_scoped)
best_performance = best_performance.reshape((n, n))
best_score = score_func(best_solution,
                        goal,
                        Evaluator,
                        Opti.iter_turn)
best_score = round(best_score, 1)

ax['e'].imshow(best_performance)
ax['e'].set_title(f"Best Performance: MSE = {best_score}")

# Final design and performance
#final_solution = Opti.design
#final_solution_scoped = scope(final_solution)
```

```
#final_performance = mod.predict(final_solution_scoped)
#final_performance = final_performance.reshape((n, n))
#final_score = score_func(final_solution, goal, mod)
#final_score = round(final_score, 1)
#
#ax['f'].imshow(Opti.design)
#ax['f'].set_title("Final Design")#+str(s))
#ax['g'].imshow(final_performance)
#ax['g'].set_title(f"Final Performance: MSE = {final_score}")
```

Annex 5: Simulated Annealing Optimizer

```
import numpy as np
import matplotlib.pyplot as plt
from time import time

def ComparePerformance(score_new, score_old):
    return score_new < score_old


class SurrogateCall():
    def __init__(self, mod, scope):
        self.mod = mod
        self.scope = scope

    def GetOutput(self, design, iter_turn):
        scoped_design = self.scope(design)
        prediction = self.mod.predict(scoped_design)
        return prediction


def loss_func(x, y):
    x -= y
    x **= 2
    return x.sum()


def score_func(design, goal, Evaluator, iter_turn):

    # For optimization with surrogate model
    prediction = Evaluator.GetOutput(design, iter_turn)
    loss = loss_func(prediction, goal)

    # Using built-in scoring
```

```
        #score = mod.score(scoped_design, goal)

        # To make Lead wall.
        #loss = np.sum(prediction)

        # Intersting non-determined goal.
        #loss = 1/np.var(prediction)
        return loss


class SimulatedAnnealer():
    def __init__(self,
                 design,
                 goal,
                 Evaluator,
                 npad,
                 sq,
                 material_numbers):

        self.npad = npad
        self.sq = sq
        self.material_numbers = material_numbers

        self.goal = goal
        self.Evaluator = Evaluator


        self.temp = 100
        self.cooling_fact = 0.90

        self.iter_turn = 0
        self.design = design

        score_initial = score_func(design,
```

```
                                    goal,
                                    self.Evaluator,
                                    self.iter_turn)

        self.score = score_initial
        self.score_avg = score_initial
        self.score_best = score_initial

        # Init local score history
        #self.n_optim = 4
        #self.score_hist = np.zeros(self.n_optim)

        # Init. figures
        fig, ax = plt.subplots()
        self.fig = fig
        self.ax = ax
        #self.SaveFig()

    def SaveFig(self):
        self.ax.imshow(self.design)
        self.ax.set_title(
            f"Optimization Iteration: {self.iter_turn}"
        )
        #self.fig.savefig("".join((
        #   "./optimization_plots/",
        #   f"surrogate/{self.iter_turn}_fig.png"))
        #)
        plt.close()

    def main(self):
        self.time_start = time()

        # Optimization Loop
        for turn in range(1000):
```

```
self.iter_turn += 1

# Select index to mutate
i, j = np.random.randint(self.npad,
                         self.npad + self.sq,
                         2)

# Record old value
old_val = self.design[i][j]

# Choose new value
new_val = np.random.choice(self.material_numbers)

# Mutate design
self.design[i][j] = new_val

# Score new design
self.score_new = score_func(self.design,
                            self.goal,
                            self.Evaluator,
                            self.iter_turn)

# Update History and running average
#hist_ind = self.iter_turn % n_optim
#self.score_hist[hist_ind] = self.score_new
self.score_avg *= 0.9
self.score_avg += 0.1*self.score_new

## Optimize
IsScoreImproved = ComparePerformance(
                        self.score_new,
                        self.score
)
if IsScoreImproved:
```

```
            self.score = self.score_new

            # Keep track of best observed design
            IsScoreBest = ComparePerformance(
                                self.score_new,
                                self.score_best
            )
            if IsScoreBest:
                self.score_best = self.score_new
                self.best_solution = self.design.copy()

        else:

            # If score increased above the time-weighted
            # average, => cool the optimization.
            if self.score >= self.score_avg:
                #self.SaveFig()
                self.temp *= self.cooling_fact
                self.score_avg = self.score

            else:
                pass


            # Accept / reject sampling method
            try:
                score_ratio = self.score_new / self.score
                score_ratio /= 8
                score_ratio **= 2
                prob = self.temp*np.exp(-(score_ratio))
                yield prob

            except ZeroDivisionError:
                break
```

```
            ## lim sp >> s: exp(s/sp) -> 0, pr -> 1
            ## lim sp << s: exp(s/sp) -> -\infty, pr -> 0

            u = np.random.uniform()
            if u < prob:
                # acceptance of `bad' design modification
                self.score = self.score_new
                #self.SaveFig()

            else:
                # return to original state.
                self.design[i][j] = old_val
                #self.SaveFig()

        print(f"itt: {self.iter_turn} | {self.score}")
        try:
            assert abs(self.score) > 0

        except AssertionError:
            break

self.time_taken = time() - self.time_start
```

Annex 6: PickAI Network Topology

```
import torch
import torch.nn as nn
from Hamiltonian import Hamiltonian_torch
nn.requires_grad=False

with torch.no_grad():
    class CNA(nn.Module):
        @torch.no_grad()
        def __init__(self):
            super(CNA, self).__init__()
            self.c1 = nn.Conv2d(1, 16, 3,  stride=1)
            self.c2 = nn.Conv2d(16, 32, 3, stride=1)
            self.d1 = nn.Conv2d(32, 16, 3, stride=1)
            self.d2 = nn.Conv2d(16, 1, 3,  stride=1)
            self.encoder = nn.Sequential(
                    self.c1,
                    nn.Tanh(),
                    self.c2,
                    nn.Tanh())
            self.decoder = nn.Sequential(
                    self.d1,
                    nn.Tanh(),
                    self.d2)

        @torch.no_grad()
        def one2one(self,x):
            x = torch.sign(2*torch.sign(x) - 1)
            return x

        @torch.no_grad()
        def MakeConfig(self, x):
            x = self.encoder(x)
```

```
        x = self.decoder(x)
        x = self.one2one(x)
        x.squeeze_()
        return x


@torch.no_grad()
def forward(self,x):
        x = self.encoder(x)
        x = self.decoder(x)
        x = self.one2one(x)
        x.squeeze_()
        x = Hamiltonian_torch(x)
        return x
```

Annex 7: NeuroEvolution Training

```
import torch
import numpy as np
import functools as ft
from itertools import cycle
import time
import os
import sys

sys.path.append(os.getcwd())
from generation_network_topology import CNA
from input_func import *
from smi_func import smi_func #smi_func is the Hamiltonian
from datafuncs import *

import GA_Tools as GA
from MomentsComparer import MomentsComparer


try:
    fn = int(sys.argv[1])
except:
    fn = 6

import Train_Checkpointing as checkpoint
input_width = checkpoint.DetermineInputSize(fn)


with torch.no_grad():

    # Optimization parameters
    opt_flag = "max"
    epoch_mult = 1024
```

```
    # Evolution parameters
    N_runners_up = 4
    N_best_candidates = 10

    max_mutation_rate = 1.0
    min_mutation_rate = 0.0001

    max_gen_size = 250
    min_gen_size = 25

    diff_mut_rate = (max_mutation_rate-min_mutation_rate)
    diff_mut_rate /= (epoch_mult+1)

    # Efficiency parameters
    batch_mult = 16
    max_batch_size = 512
    min_batch_size = 32


    # Below: round over corrects...
    #diff_gen_size = int(diff_gen_size)


##  Generation Initialization
    epoch, gen_size, \
    batch_size, cp_dict, \
    gen = checkpoint.ReturnToCheckpoint(epoch_mult,
                                        max_gen_size,
                                        min_gen_size
    )
    gen = list(gen)

    gen_size = GA.UpdateGenSize(epoch,
```

```
                                    epoch_mult,
                                    max_gen_size,
                                    min_gen_size)

    opt_func = GA.Init_OptFunc(opt_flag)

##  Train
    while epoch < epoch_mult:
        t_epoch = time.time()

        # contines for N mins. Then saves.
        while time.time() - t_epoch < 5*60:
            epoch += 1

            # Adjust generation size
            gen_size = GA.UpdateGenSize(epoch,
                                        epoch_mult,
                                        max_gen_size,
                                        min_gen_size)

            # Different reward levels (depending on gen_size)
            rewards = GA.ReDistribute_N_Offspring(
                                          gen_size,
                                          N_best_candidates,
                                          N_runners_up)

            # Learning rate moves from maximum to minimum
            # allowed over epoch_mult
            mutation_rate = GA.UpdateLearningRate(
                                        epoch,
                                        max_mutation_rate,
                                        diff_mut_rate)

            # Batch size increases as generation size decreases
```

```
#   (more testing per candidate near end)
batch_size = GA.UpdateBatchSize(epoch,
                                epoch_mult,
                                max_batch_size,
                                min_batch_size
)

# Pre-generate all batches.
batch_shape = (batch_mult,
               batch_size,
               1,
               input_width,
               input_width)

batch_all = input_func_all(batch_shape)

# Load all-batches into partial function for lazy
# evaluation.
EvaluateGeneration=ft.partial(
                     GA.EvaluateGeneration_Base,
                     batch_all=batch_all
)

# Lazy evaluation of candidate score
# (does all candidates in gen)
gen_score = map(EvaluateGeneration, gen)

##  Score
best_inds = opt_func(gen_score,
                     N_best_candidates)

##  Sort
best_candidates = (gen[i] for i in best_inds)
best_candidates = cycle(best_candidates)
```

```
        new_gen = GA.ReGenerate_Gen(mutation_rate,
                                    N_best_candidates,
                                    N_runners_up,
                                    gen_size,
                                    rewards,
                                    best_candidates)
        # Regenerate.
        gen = list(new_gen)
        median_ind = N_best_candidates//2
        median_best_score = EvaluateGeneration(
                                gen[median_ind]
        )

        print(median_best_score,
              f"{gen_size}/{len(gen)}",
              batch_size)


# Report progress
candidate_sample = map(gen[N_best_candidates//2],
                       batch_all)

candidate_sample = map(
                    MomentsComparer.EstimateMoments,
                    candidate_sample
)

moments = next(candidate_sample)
moments = [round(mmnt_i.item(), 3) \
               for mmnt_i in moments]
print(f"median best moments: {moments}")

# Compare against known range.
```

```
        ideal_moments = MomentsComparer.CalculateMoments(72,
                                                         -72)
        print(f"ideal: {ideal_moments}")

        # Save optimization
        checkpoint.MakeCheckpoint(epoch, epoch_mult, input_width,
                          best_candidates, N_best_candidates,
                          max_mutation_rate, diff_mut_rate,
                          gen_size, fn,
                          batch_mult, batch_size)


# Final optimization. Shrinks gen to only best
for i, gen_i in enumerate(gen[:N_best_candidates]):
    print(f"\ngen[{i}]")
    input_batch = input_func_all((50000,
                                  1,
                                  input_width,
                                  input_width))
    outs = (gen_i(input_batch)).detach()
    smi = smi_func(outs)

    sl, nl = np.unique(smi, return_counts=True)
    print("sl: ", sl)
    print("nl: ", nl)
    print("score: ", GA.score_metric_func(outs))
    print("test: ", smi_func(outs)[0])

for i in range(N_best_candidates):
    torch.save(gen_i.state_dict(),
               f"candidate_networks/inet{fn}_{i}.pt")
print("FIN")
```