
soa

Release 0.0.1

Oct 01, 2020

Contents:

1	Getting Started	3
2	Free Software	5
3	Documentation	7
4	Index	31

This project formed part of the journal paper [An Artificial Intelligence Approach to Optimal Control of Sub-Nanosecond SOA-Based Optical Switches](#). All data (<https://doi.org/10.5522/04/12356696.v1>) used in the paper for PSO were generated with this code. The PSO algorithm is used to optimise the driving signal for SOA switches both in simulation and experiment.

This `soa` package can be used to quickly and easily play around with different SOA simulations and PSO hyperparameters.

CHAPTER 1

Getting Started

Follow the *instructions* to install this project, then have a look at the *tutorial*. This project is not actively maintained, therefore the best way to get started is to follow the simple tutorial example(s) to become familiar with using some basic parts of the code, then have a look inside the code to understand how the different methods and functions work. You can then edit the code yourself according to what you want to investigate with respect to PSO and SOA drive signal optimisation.

Unfortunately the Python code for this project is not the most clean, however there are not too many lines of complex interdependencies/file structures to deal with and the code is commented sufficiently such that you should be able to read through it and adapt it as you wish. Intermediate-strong Python programmers are encouraged to completely re-write parts they consider poorly implemented.

CHAPTER 2

Free Software

This is free software; you can redistribute it and/or modify it under the terms of the Apache License 2.0. Contributions and forks are welcome - see the [contribute guide](#). Contact cwfparsonson@gmail.com for questions.

3.1 Install

Open Git Bash. Change the current working directory to the location where you want to clone this [GitHub](#) project, and run:

```
$ git clone https://github.com/cwfparsonson/soa_driving.git
```

In `soa_driving/pso`, install the required packages with either conda:

```
$ conda install --file requirements/default.txt
```

or pip:

```
$ pip install -r requirements/default.txt
```

Still in `soa_driving/pso`, make the `soa` python module importable from anywhere on your machine:

```
$ python setup.py develop
```

You should then be able to import the `soa` module into your Python script from any directory on your machine:

```
>>> import soa
```

3.2 Tutorial

This tutorial can help you get started with using this Python package to optimise semiconductor optical amplifiers (SOAs) with particle swarm optimisation (PSO).

Pre-requisites (key concepts described in the following such as what rise time, settling time, overshoot, dynamic PSO, transfer functions etc. will not be re-explained in this tutorial):

- An Artificial Intelligence Approach to Optimal Control of Sub-Nanosecond SOA-Based Optical Switches (and any references in the paper not recognised by the reader)
- Chapter 16 of S. Kiranyaz's 2014 book 'Particle Swarm Optimisation'
- Basic Python

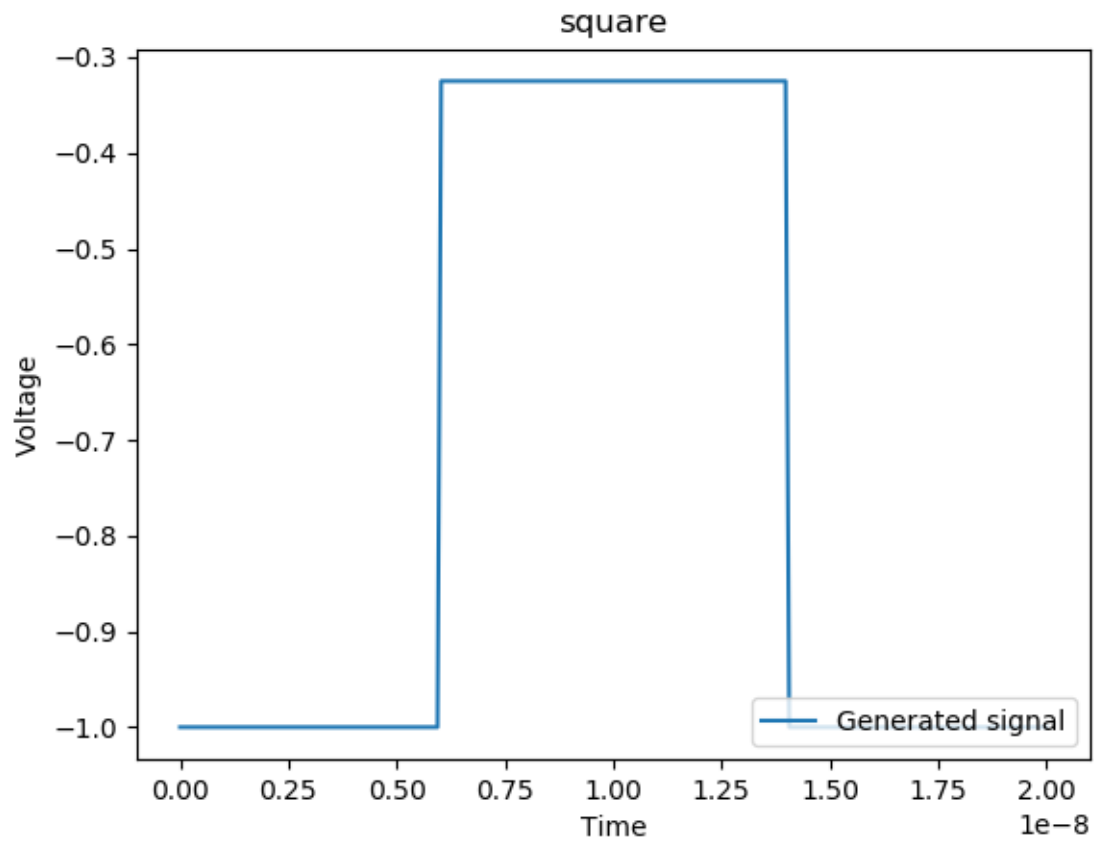
3.2.1 Introduction

SOA Switch Optimisation

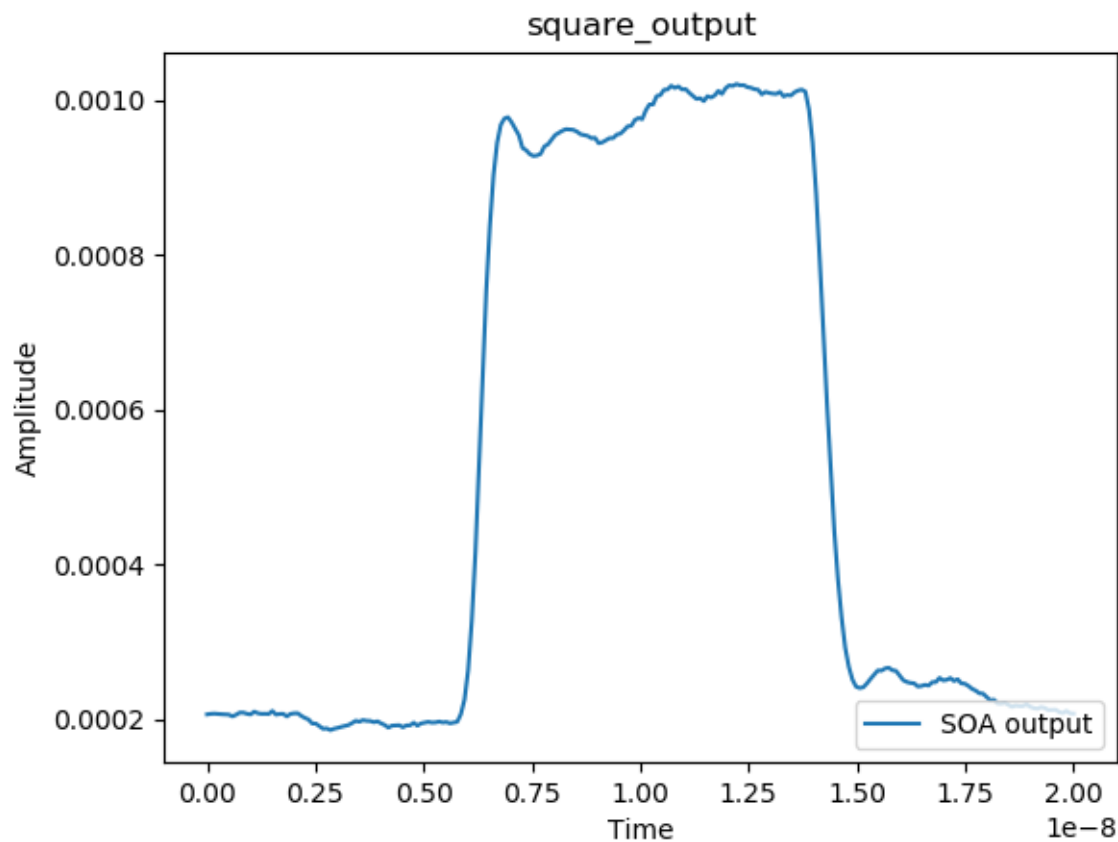
Particle Swarm Optimisation (PSO) is a metaheuristics used to solve a range of optimisation problems, and falls under the broad category of 'artificial intelligence' techniques. For an excellent in-depth summary, see chapter 16 of S. Kiranyaz's 2014 book 'Particle Swarm Optimisation'.

Semiconductor optical amplifiers (SOAs) are devices which, given (1) an electronic driving signal and (2) an optical input signal, will amplify the light signal by a combination of spontaneous and stimulated emission. This allows SOAs to act as a *switch* capable of switching light on and off. Light is a popular medium in which to communicate information owing to its associated high bandwidth, energy efficiency, speed and equipment costs. However, to communicate with light, you must be able to switch and change the properties of your light signal. For certain applications, such as next-generation optical circuit switched data centre networks, the *speed* with which you can switch light signals is extremely important. It turns out that SOAs are excellent for switching light quickly, with theoretical switching speeds limited only by their ~ 100 pico- (10^{-12}) second excitation times. However, due to instability issues, this switching speed is in practice much slower (on the nano- (10^{-9}) second scale), which is too slow for many desirable applications.

To help fix instability issues and improve SOA switching times, researchers need to *optimise* their SOA. Optimisation can come in two forms; optimising the physical properties of the SOA and its surrounding setup, *or* optimising the electronic drive signal passed into the SOA in order to turn the SOA 'on'. The most basic electronic drive signal to apply is the *step* drive signal:



However, this simple signal results in the following unstable optical output, resulting in nanosecond-scale switching times:



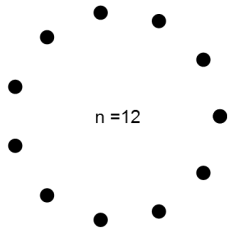
To remedy this, researchers previously attempted to apply a range of different electronic drive signals, from standard PID solutions from control theory to carefully tuned PISIC and MISIC formats. However, none of these solutions proved to work sufficiently well to achieve sub-nanosecond (hundred-picosecond scale) switching, and none were scalable to the thousands/millions of SOAs each with different properties that you would find in a real optical communication network such as a data centre.

PSO Applied to SOA Drive Signal Optimisation

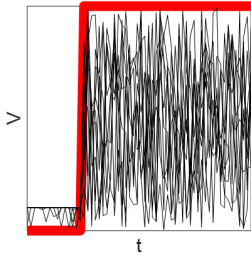
To help solve this short-coming in SOA drive signal optimisation, this package uses a particle swarm optimisation approach to optimising the electronic drive signal put into the SOA. The below figure is a step-by-step visualisation of how PSO was applied to the problem of SOA drive signal optimisation at the most basic level.

Initialise the algorithm:

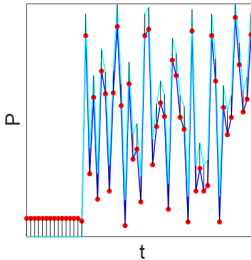
Initialise n random particles



Use a step input signal as an initial 'guess' to embed amongst the particles (red)

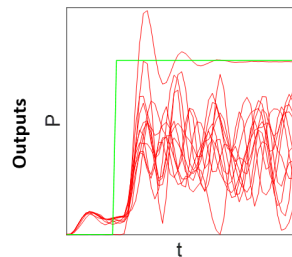
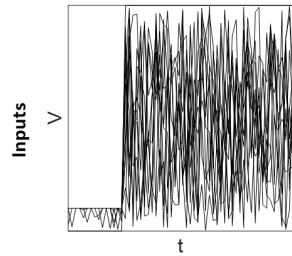
**For each particle:**

Find the distance between its p_{best} (cyan) and its current position, x (blue). Bias this with a personal confidence acceleration coefficient c_1 to mimic exploitation of its own knowledge, and by a random coefficient r_1 to mimic exploration.

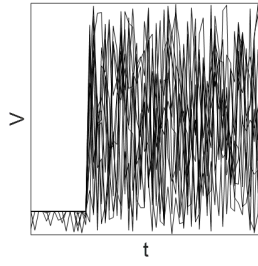


Repeat process until particles converge on position with best fitness. This is the optimum driving signal for the SOA.

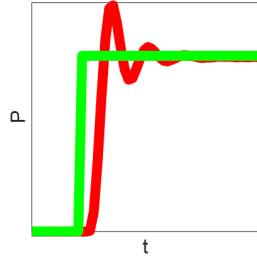
Iter = 0



where each particle position represents an input signal

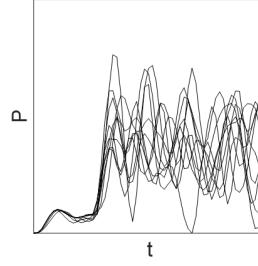


and use its corresponding output to define an ideal target SP (green).

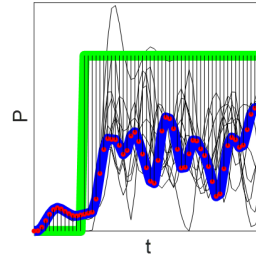


Use the fitness improvement of x relative to p_{best} to get an 'inertia weight' w . Multiply this by the current velocity v to get the particle's momentum.

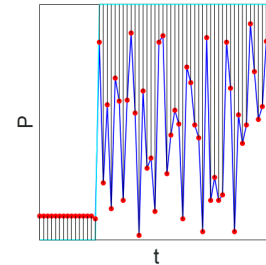
with n corresponding optical output signals



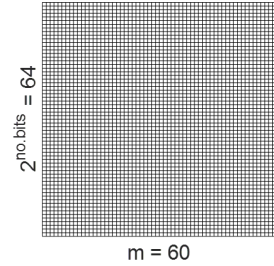
For each particle, find the MSE between its output and the SP , and store this as its 'fitness.'



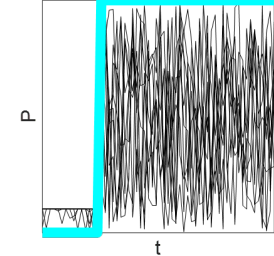
Find the distance between x and g_{best} . Bias with a social confidence acceleration coefficient c_2 to mimic exploitation of the population's knowledge, and by a random coefficient r_2 to mimic exploration.



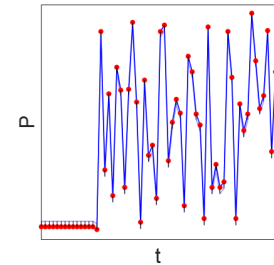
in a search space with m dimensions that can each take $2^{no. bits}$ values.



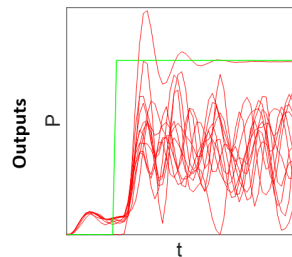
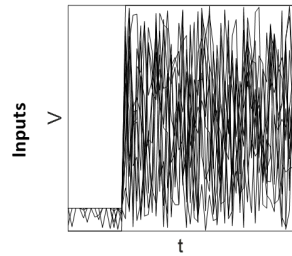
Find the position with the global best fitness and set this as g_{best} (cyan).



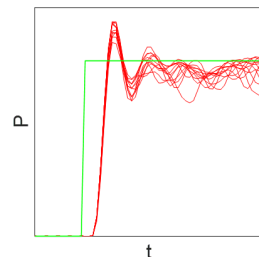
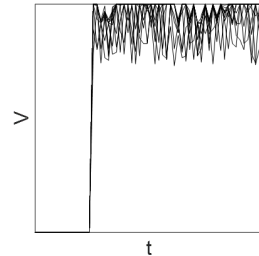
Update the particle's velocity by summing the personal and social distance terms with the momentum term. Add the new velocity to the current position to get the new particle position, $x(t+1)$.



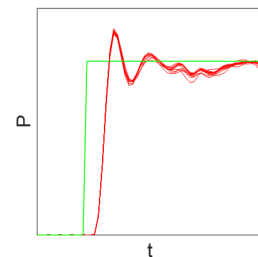
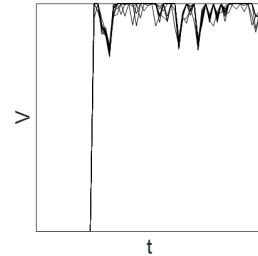
Iter = 0



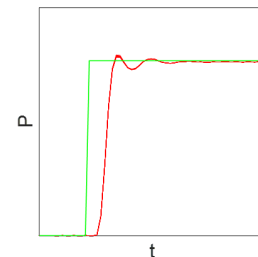
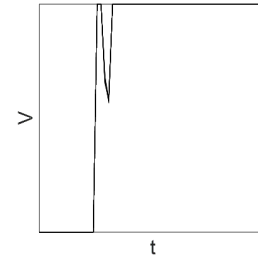
Iter = 28



Iter = 35



Iter = 150



The paper implements a couple of other key ideas such as the concept of a PISIC shell to reduce the PSO algorithm's search space and dynamic PSO, however the above is a succinct visualisation of the core idea.

3.2.2 Running Simulations

Note: As mentioned, this PSO code has not been cleanly implemented for the most part and is not actively maintained. There are definitely better ways of constructing it. This tutorial is intended to help you get started with the code and understand how PSO can be used to optimise SOAs. You are encouraged to fork the project and re-write parts/add functionality as you see fit. Contact cwfpersonson@gmail.com if you are interested in merging your code.

Note: In the below examples, to save time, only $n = 3$ particles have been used. Having such a low number of particles allows for short optimisation times, but will significantly reduce the efficacy of the final optimal solution. To find better solutions, you should increase the number of particles. If you change certain hyperparameters, you may also need to change e.g. the number of particles. For example, increasing the number of dimensions (i.e. number of points) in the particles/drive signals will increase the size of the search space and therefore also likely require more particles to find sufficiently good solutions.

A Walkthrough Example

In this example, you will see how to use PSO to optimise the driving signals for 10 different SOAs by simulating 10 different transfer functions.

In your Python script, start by importing the following core functionalities from the soa module:

```
from soa import devices, signalprocessing, analyse, distort_tf
from soa.optimisation import PSO, run_test
```

The purpose of each of the above is the following:

- `devices.py`: Module for interfacing with the SOA experimental setup (**not needed if only using transfer function simulations**)
- `signalprocessing.py`: Module for generating standard literature SOA driving signals (e.g. PISIC, MISIC, etc.) and for evaluating optical response cost in terms of e.g. mean squared error
- `analyse.py`: Module for analysing signal performance and retrieving optical output signal metrics (e.g. rise time, settling time, overshoot etc.)
- `distort_tf.py`: Module for distorting the original SOA transfer function to generate new transfer functions and therefore simulate different SOAs (useful for testing the generalisability of your optimisation method(s))
- `optimisation.py`: The main module that holds the `PSO` class (the key class with the PSO algorithm, send & receive functionality for drive & optical signal(s) respectively, plotting, etc.) and the `run_test` function, which is a function used for multiprocessing (running multiple PSO experiments in parallel - not required but recommended to speed up your tests).

Note: Use the above as a guide for when you want to add more functionality/find a specific function being used by this `soa` package/see how things work etc. Feel free to add to, rewrite or replace any of the above modules/functions/methods/classes.

Import any other useful packages you'd like to use


```
import numpy as np
import multiprocessing
import pickle
from scipy import signal
import os
import matplotlib.pyplot as plt
```

Create a folder on your local machine in which to store your PSO/SOA data and create a directory variable with the path to this folder:

```
directory = '../data/'
```

Initialise the basic parameters of the input and output signal you want to optimise:

```
num_points = 240 # number of points in the signal (corresponds to number of
↳dimensions)* for each particle)
time_start = 0 # time at which signal period should begin
time_stop = 20e-9 # time at which signal period should end
t = np.linspace(time_start, time_stop, num_points) # time axis of all signals
```

Configure the basic PSO hyperparameters:

```
n = 3 # number of particles (i.e. number of signals)
iter_max = 150 # maximum number of iterations to perform before stopping the PSO
↳algorithm
rep_max = 1 # number of times to repeat PSO optimisation
max_v_f = 0.05 # maximum velocity factory by which to multiply the maximum parameter
↳values by to get the maximum particle velocity for each iteration
init_v_f = max_v_f # factor by which to multiply initial positions by to get initial
↳velocity for first iteration
cost_f = 'mSE' # cost function to use to evaluate performance. Must be 1 of: 'mSe',
↳'st', 'mSE+st', 's_mse+st', 'mse+st+os', 'zlpC'
w_init = 0.9 # initial inertia weight value (0 <= w <= 1)
w_final = 0.5 # final inertia weighth value (0 <= w <= 1)
on_suppress_f = 2.0 # factor by which to multiply initial guess by when in 'off'
↳state and add to this to get space constraint
```

Define the initial drive signal (e.g. a step):

```
init_OP = np.zeros(num_points) # initial drive signal (e.g. a step)
init_OP[:int(0.25*num_points)], init_OP[int(0.25*num_points):] = -1, 0.5
```

Define the SOA(s) transfer function(s) you want to optimise for (N.B. init_OP must have a low point of -1 for the transfer function):

```
# initial transfer function numerator and denominator coefficients
num = [2.01199757841099e85]
den = [
    1.64898505756825e0,
    4.56217233166632e10,
    3.04864287973918e21,
    4.76302109455371e31,
    1.70110870487715e42,
    1.36694076792557e52,
    2.81558045148153e62,
    9.16930673102975e71,
    1.68628748250276e81,
```

(continues on next page)

(continued from previous page)

```

    2.40236028415562e90,
]
tf = signal.TransferFunction(num, den)
tfs, _ = distort_tf.gen_tfs(num_facs=[1.0, 1.2, 1.4],
                             a0_facs=[0.8],
                             a1_facs=[0.7, 0.8, 1.2],
                             a2_facs=[1.05, 1.1, 1.2],
                             all_combos=False)

```

Get the initial output of the initial transfer function and derive the target set point that you want to optimise towards as your target:

```

init_PV = distort_tf.getTransferFunctionOutput(tf, init_OP, t)
sp = analyse.ResponseMeasurements(init_PV, t).sp.sp

```

Run your experiments in parallel using the multiprocessing functionality provided by run_test:

```

pso_objs = multiprocessing.Manager().list()
jobs = []
test_nums = [test+1 for test in range(len(tfs))]
dirs = [directory + '/test_{}'.format(test_num) for test_num in test_nums]
for tf, direc in zip(tfs, dirs):
    if os.path.exists(direc) == False:
        os.mkdir(direc)
    p = multiprocessing.Process(target=run_test,
                               args=(direc,
                                      tf,
                                      t,
                                      init_OP,
                                      n,
                                      iter_max,
                                      rep_max,
                                      init_v_f,
                                      max_v_f,
                                      w_init,
                                      w_final,
                                      True,
                                      'pisc_shape',
                                      on_suppress_f,
                                      True,
                                      None,
                                      cost_f,
                                      None,
                                      True,
                                      True,
                                      sp,
                                      pso_objs,))

    jobs.append(p)
    p.start()
for job in jobs:
    job.join()

```

Pickle your PSO objects so you can re-import and analyse them later:

```
PIK = directory + '/pickle.dat'
```

(continues on next page)

(continued from previous page)

```
data = pso_objs
with open(PIK, 'wb') as f:
    pickle.dump(data, f)
```

Pulling all of the above together, your script should look something like:

```
from soa import devices, signalprocessing, analyse, distort_tf
from soa.optimisation import PSO, run_test

import numpy as np
import multiprocessing
import pickle
from scipy import signal
import os
import matplotlib.pyplot as plt

# set dir to save data
directory = '../..data/'

# init basic params
num_points = 240
time_start = 0
time_stop = 20e-9
t = np.linspace(time_start, time_stop, num_points)

# set PSO params
n = 3
iter_max = 150
rep_max = 1
max_v_f = 0.05
init_v_f = max_v_f
cost_f = 'mSE'
w_init = 0.9
w_final = 0.5
on_suppress_f = 2.0

# define initial drive signal
init_OP = np.zeros(num_points) # initial drive signal (e.g. a step)
init_OP[:int(0.25*num_points)], init_OP[int(0.25*num_points):] = -1, 0.5

# initial transfer function numerator and denominator coefficients
num = [2.01199757841099e85]
den = [
    1.64898505756825e0,
    4.56217233166632e10,
    3.04864287973918e21,
    4.76302109455371e31,
    1.70110870487715e42,
    1.36694076792557e52,
    2.81558045148153e62,
    9.16930673102975e71,
    1.68628748250276e81,
    2.40236028415562e90,
]
tf = signal.TransferFunction(num, den)
tfs, _ = distort_tf.gen_tfs(num_facs=[1.0, 1.2, 1.4],
```

(continues on next page)

(continued from previous page)

```

a0_facs=[0.8],
a1_facs=[0.7,0.8,1.2],
a2_facs=[1.05,1.1,1.2],
all_combos=False)

# get initial output of initial signal and use to generate a target set point
init_PV = distort_tf.getTransferFunctionOutput(tf,init_OP,t)
sp = analyse.ResponseMeasurements(init_PV, t).sp.sp

# run PSO tests in parallel with multiprocessing
pso_objs = multiprocessing.Manager().list()
jobs = []
test_nums = [test+1 for test in range(len(tfs))]
dirs = [directory + '/test_{}'.format(test_num) for test_num in test_nums]
for tf, direc in zip(tfs, dirs):
    if os.path.exists(direc) == False:
        os.mkdir(direc)
    p = multiprocessing.Process(target=run_test,
                               args=(direc,
                                     tf,
                                     t,
                                     init_OP,
                                     n,
                                     iter_max,
                                     rep_max,
                                     init_v_f,
                                     max_v_f,
                                     w_init,
                                     w_final,
                                     True,
                                     'pistic_shape',
                                     on_suppress_f,
                                     True,
                                     None,
                                     cost_f,
                                     None,
                                     True,
                                     True,
                                     sp,
                                     pso_objs,))

    jobs.append(p)
p.start()
for job in jobs:
    job.join()

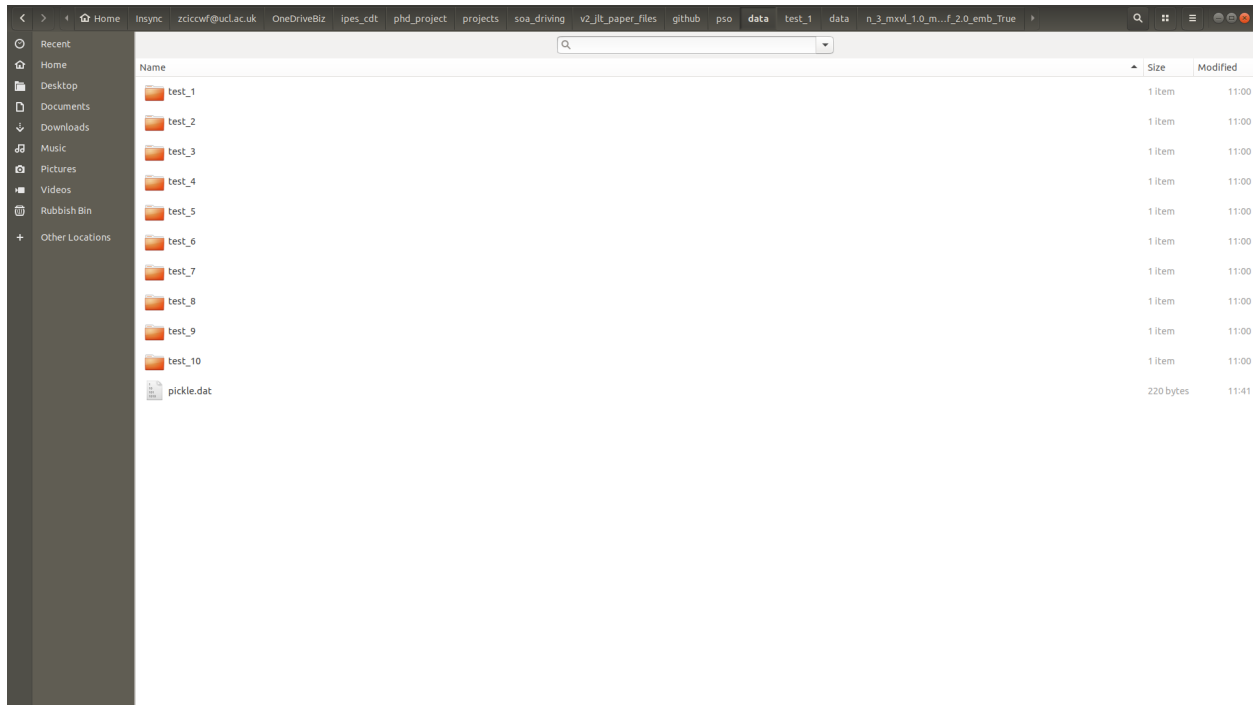
# pickle PSO objects so can re-load later if needed
PIK = directory + '/pickle.dat'
data = pso_objs
with open(PIK, 'wb') as f:
    pickle.dump(data, f)

```

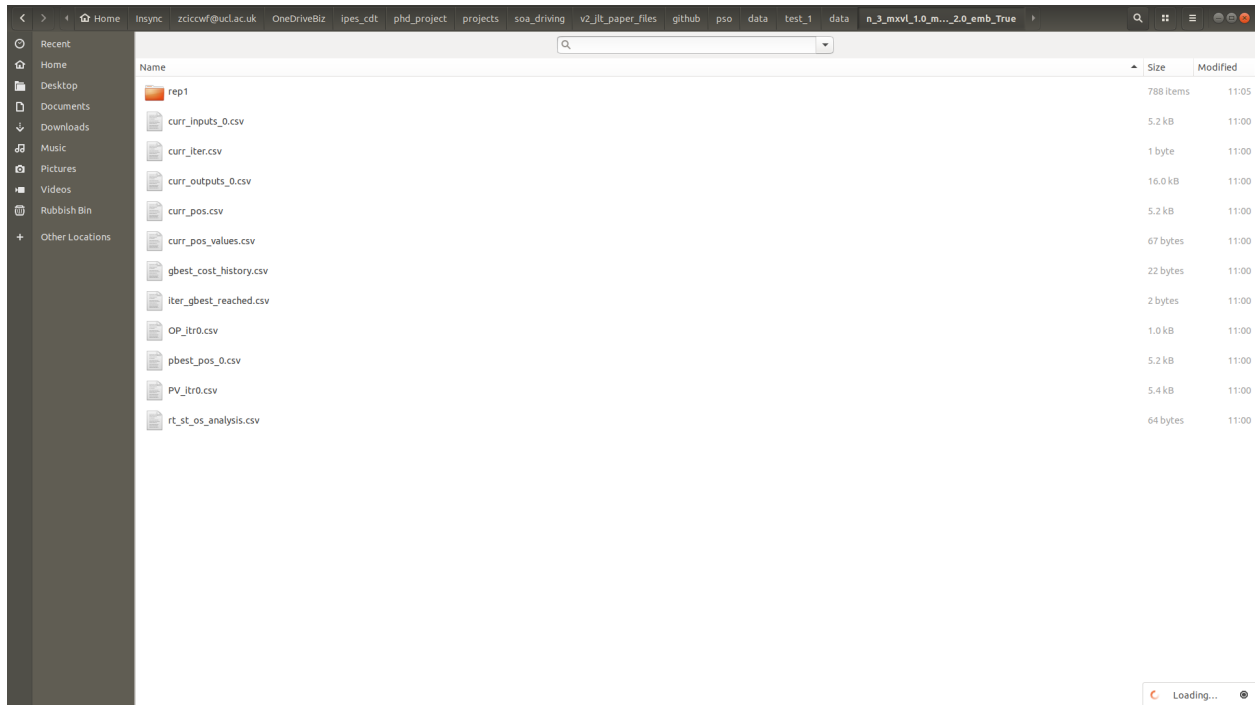
Run the above by executing your Python script. Some information will be printed out in your terminal telling you how many iterations have passed and how much the ‘cost’ (in the above example, the mean squared error) has been reduced by so far.

Note: Since the above example uses multiprocessing by running the 10 tests in parallel, the information will be printed out to your terminal in parallel, which is quite messy. Feel free to go into the PSO code in `optimisation.py` and organise this to improve the message printed out (if any message at all).

In your `directory` path, you should now have a folder for each of the tests you ran (in the above example, you used multiprocessing to run a PSO optimisation for 10 different transfer functions (SOAs) in parallel, so you have 10 test folders (one for each test you ran)). You should also have the PSO objects saved as a pickle so that you can re-load them later if needed.

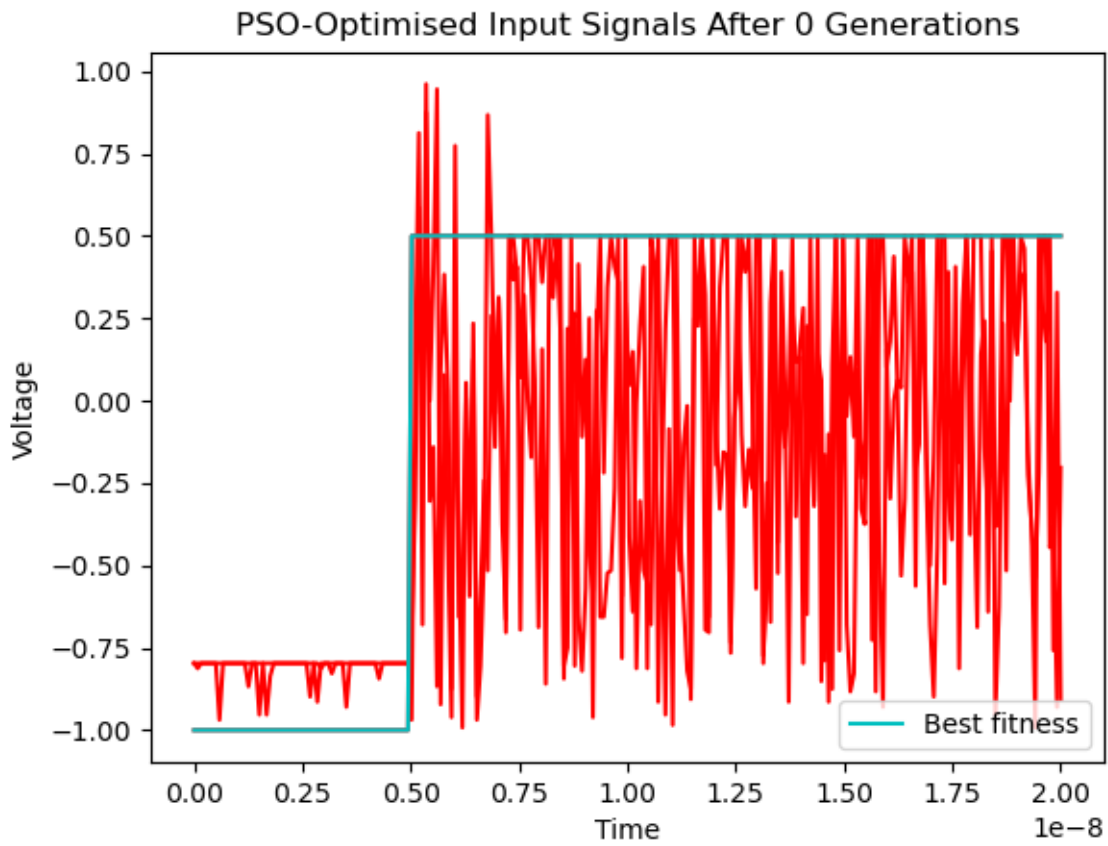


In each test file, there is a data folder which contains a folder whose name is automatically determined by the hyper-parameters you used. Inside this folder is some key data:

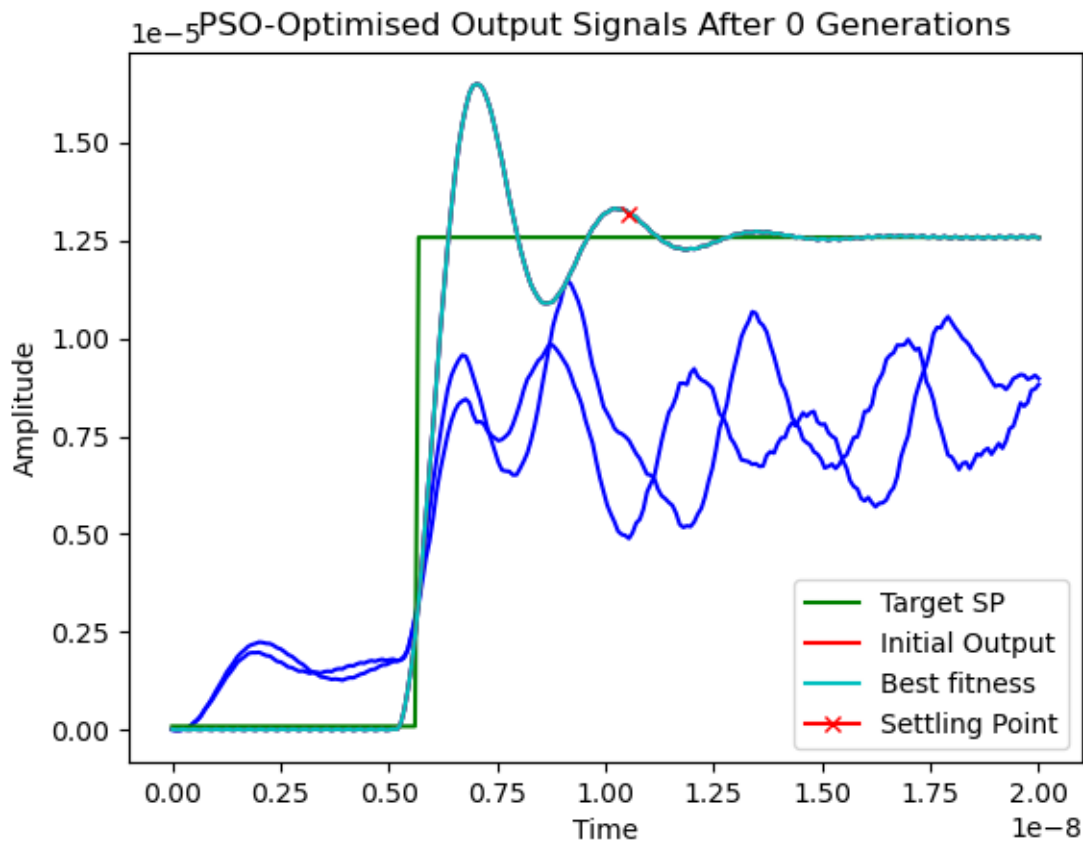


The data saved is either a `.png` file (for quick visualisation of what your PSO run has done) or a `.csv` file (for loading and analysing yourself). In the above folder, the key data are:

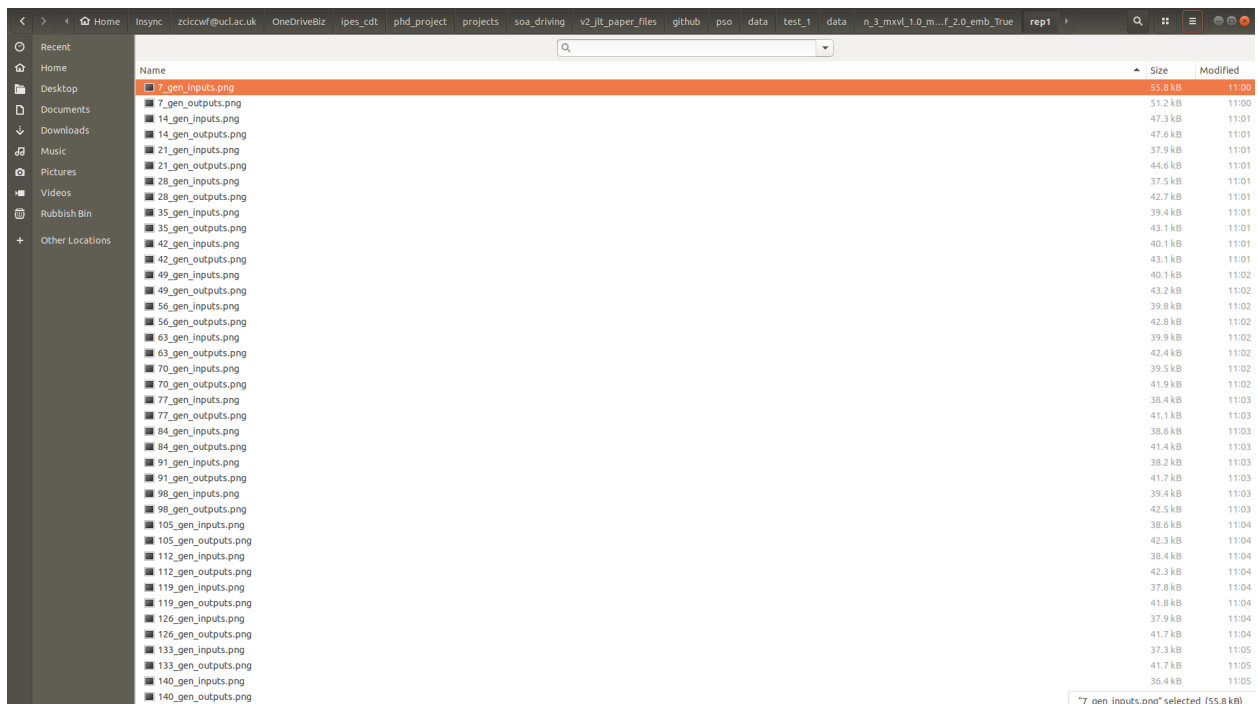
- `0_gen_inputs.png`: The input driving signals at the 0th generation of the PSO i.e. before any optimisation has begun (as described in the paper, we embed a particle whose position is equivalent to a step driving signal, which initially will be better than all other particles since other particle positions are randomly initialised (within the PISIC shell constrained area)). In the above example, we used $n = 3$ particles, therefore there are 3 driving signals (and a 4th driving signal, which is the embedded step signal)



- `0_gen_outputs.png`: The corresponding output driving signals of the 0th generation particles



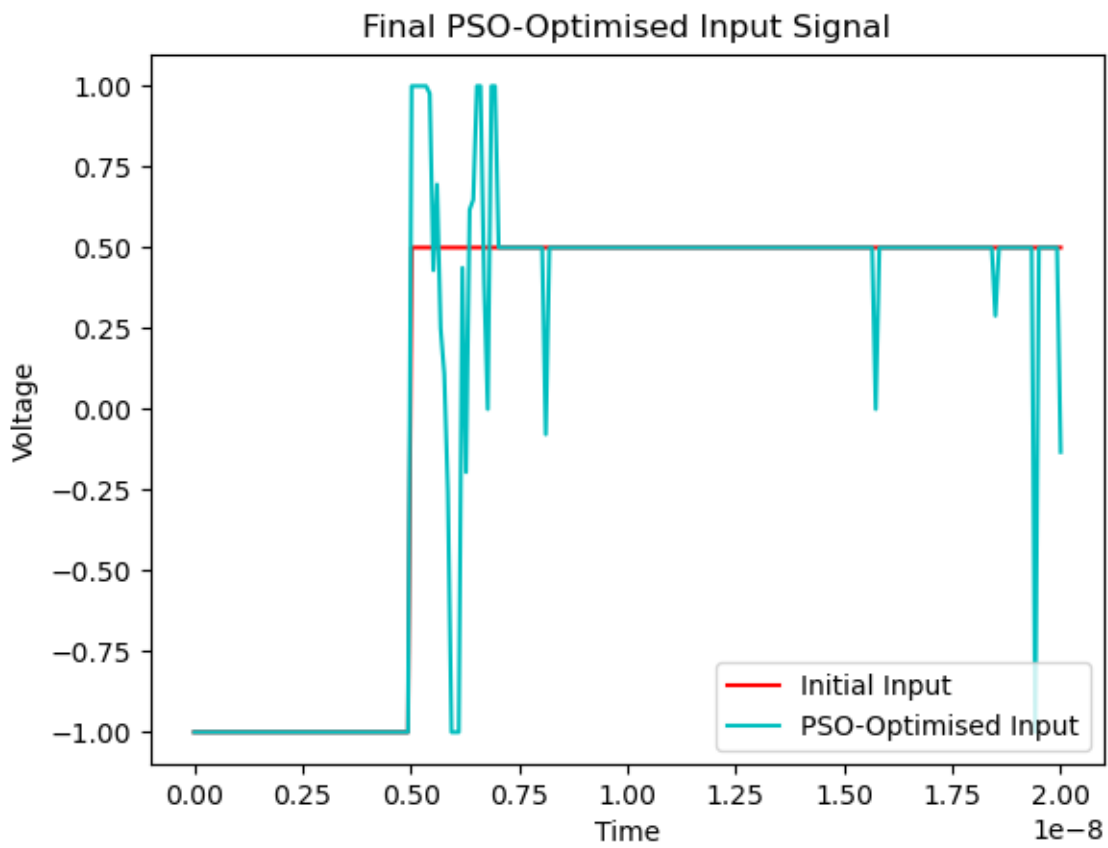
For each repetition of the PSO, there will be a `rep_X` folder. Since you set `rep_max = 1` in the above example, there is only one repetition folder, `rep_1`. Inside this `rep_1` folder are the key PSO data.

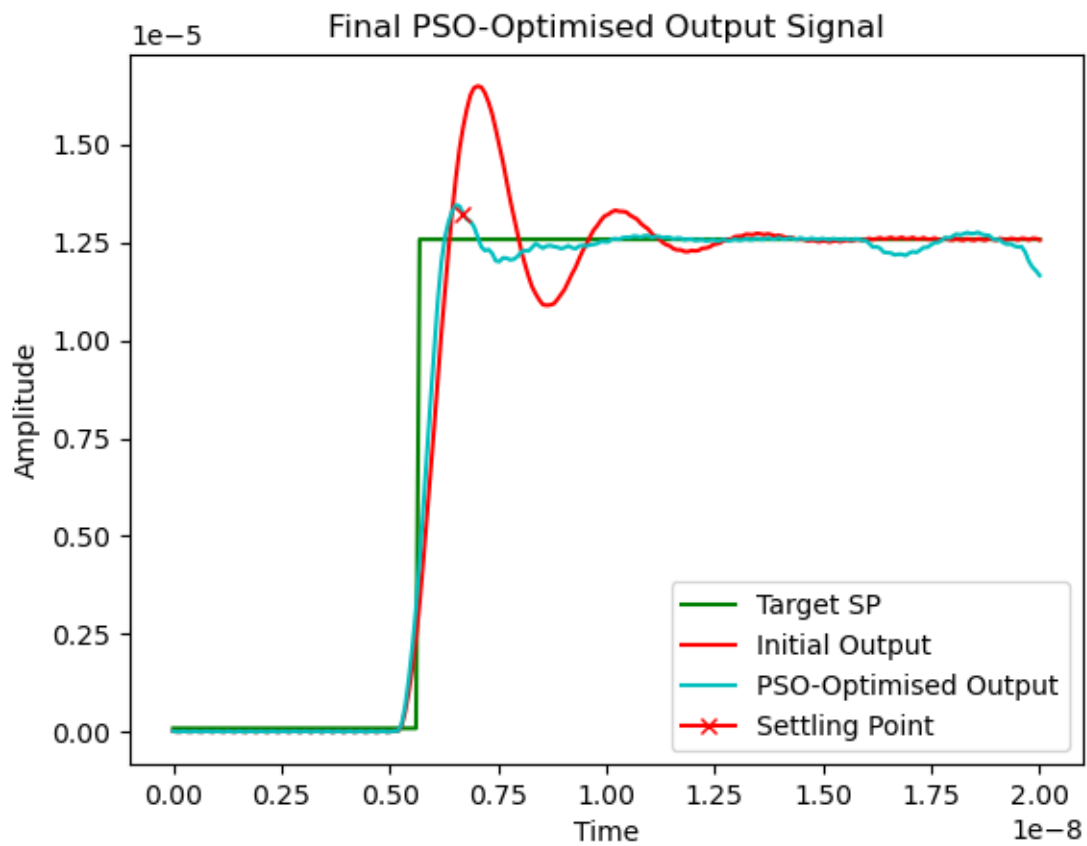


Again, the raw data is saved as a `.csv` file, and some `.png` images are saved for quick visualisation of the PSO optimisation process. In the above folder, the PSO class was configured to save data every 7 generations. Some of the key data saved includes:

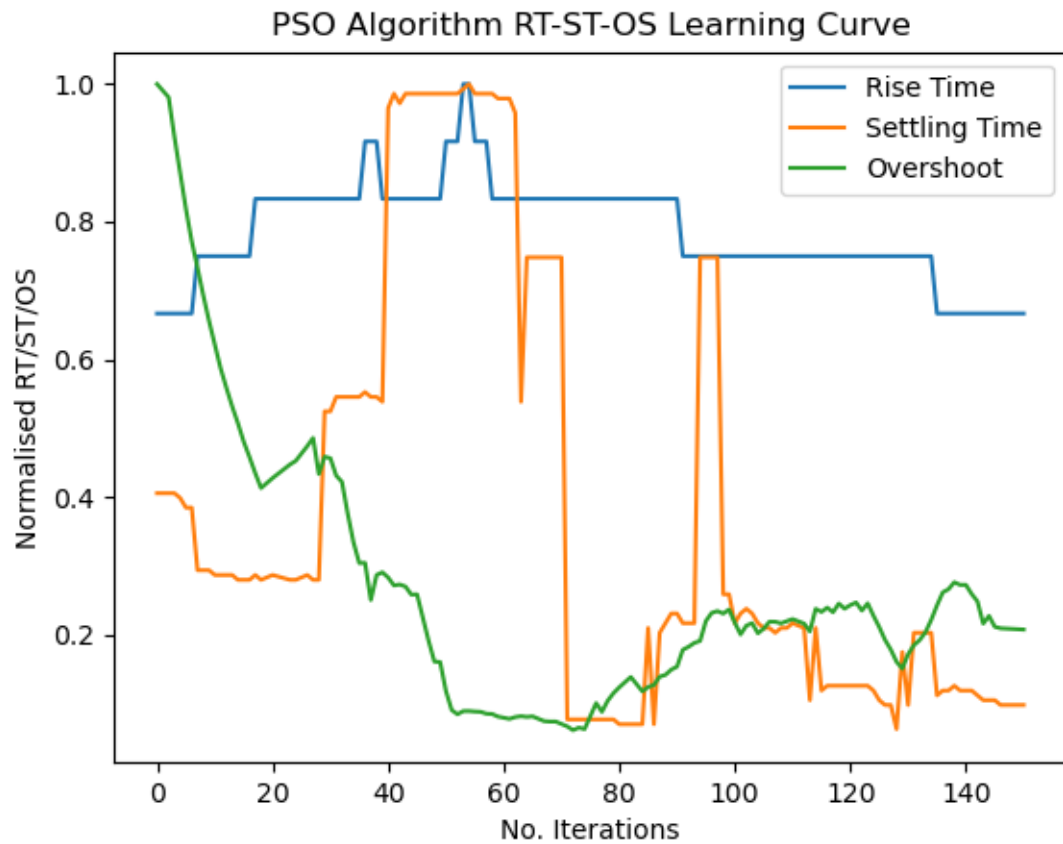
- `X_gen_inputs`: Particle (driving signal) positions at generation X
- `X_gen_outputs`: Corresponding SOA optical output in response to particle driving signals at generation X
- `curr_inputs_X`: Raw data of particle positions at generation X (useful to save incase PSO crashes and want to resume without having to start over)
- `curr_outputs_X`: Raw data of corresponding optical outputs of particle positions
- `final_input`: The final 'optimal' driving signal found by the PSO optimisation process after finishing all iterations
- `final_output`: The corresponding SOA optical output of the final optimal driving signal
- `final_learning_curve`: How the 'cost' (in the above example, the mean squared error) varied with the number of PSO iterations
- `g_best_cost_history`: The global best costs that were found across all iterations
- `iter_gbest_reached`: The corresponding iteration that each global best cost value was found at (use these data to plot cost curves)
- `initial_OP`: The initial driving signal (in the above example, a step driving signal)
- `initial_PV`: The corresponding optical output of the initial driving signal
- `optimised_OP`: The final optimised driving signal found by the PSO
- `optimised_PV`: The corresponding optical output of the final optimised driving signal
- `pbest_post_X`: The personal best position found by each particle at iteration X
- `rt_st_os_analysis`: The rise time, settling time, overshoot, and settling time index (i.e. the index in the output signal at which the signal was registered as having settled) of the best particle's corresponding optical output at each iteration of the PSO
- `rtstos_learning_curve`: How rise time, settling time and overshoot varied across iterations
- `SP`: The target set point that was used as the target SOA optical output signal that the PSO was trying to achieve (in the above example, a step signal with 0 rise time, settling and rise time generated from the initial PV)
- `time`: The time axis used by all signals

Looking at what the PSO did in the above examples, you can see that it converged on an interesting final driving signal with an improved corresponding optical output signal:





This led to a significant decrease in settling time, which is usually considered the key metric for switching time:



Additional Example Scripts

Running PSO with 120, 140, 160, 180, 200, 220 and 240 dimensions (number of points in the drive signal):

```
from soa import devices, signalprocessing, analyse, distort_tf
from soa.optimisation import PSO, run_test

import numpy as np
import multiprocessing
import pickle
from scipy import signal
import os
import matplotlib.pyplot as plt

# set dir to save data
directory = '../data/'

# init basic params
num_points_list = np.arange(120, 260, 20)
time_start = 0
time_stop = 20e-9

# set PSO params
```

(continues on next page)

(continued from previous page)

```

n = 3
iter_max = 150
rep_max = 1
max_v_f = 0.05
init_v_f = max_v_f
cost_f = 'mSE'
w_init = 0.9
w_final = 0.5
on_suppress_f = 2.0

# initial transfer function numerator and denominator coefficients
num = [2.01199757841099e85]
den = [
    1.64898505756825e0,
    4.56217233166632e10,
    3.04864287973918e21,
    4.76302109455371e31,
    1.70110870487715e42,
    1.36694076792557e52,
    2.81558045148153e62,
    9.16930673102975e71,
    1.68628748250276e81,
    2.40236028415562e90,
]
tf = signal.TransferFunction(num, den)

# run PSO tests in parallel with multiprocessing
pso_objs = multiprocessing.Manager().list()
jobs = []
for num_points in num_points_list:
    # make directory for this test
    direc = directory + '/num_points_{}'.format(num_points)
    if os.path.exists(direc) == False:
        os.mkdir(direc)

    # basic params
    t = np.linspace(time_start, time_stop, num_points)

    # define initial drive signal
    init_OP = np.zeros(num_points) # initial drive signal (e.g. a step)
    init_OP[int(0.25*num_points):] = -1, 0.5

    # get initial output of initial signal and use to generate a target set point
    init_PV = distort_tf.getTransferFunctionOutput(tf, init_OP, t)
    sp = analyse.ResponseMeasurements(init_PV, t).sp.sp

    p = multiprocessing.Process(target=run_test,
                               args=(direc,
                                      tf,
                                      t,
                                      init_OP,
                                      n,
                                      iter_max,
                                      rep_max,
                                      init_v_f,
                                      max_v_f,

```

(continues on next page)

(continued from previous page)

```

w_init,
w_final,
True,
'pisc_shape',
on_suppress_f,
True,
None,
cost_f,
None,
True,
True,
sp,
pso_objs,))

jobs.append(p)
p.start()
for job in jobs:
    job.join()

# pickle PSO objects so can re-load later if needed
PIK = directory + '/pickle.dat'
data = pso_objs
with open(PIK, 'wb') as f:
    pickle.dump(data, f)

```

3.2.3 Overview of Backend PSO and SOA Code

The following Python files make up the backend of this package:

- `devices.py`: Module for interfacing with the SOA experimental setup (**not needed if only using transfer function simulations**)
- `signalprocessing.py`: Module for generating standard literature SOA driving signals (e.g. PISIC, MISIC, etc.) and for evaluating optical response cost in terms of e.g. mean squared error
- `analyse.py`: Module for analysing signal performance and retrieving optical output signal metrics (e.g. rise time, settling time, overshoot etc.)
- `distort_tf.py`: Module for distorting the original SOA transfer function to generate new transfer functions and therefore simulate different SOAs (useful for testing the generalisability of your optimisation method(s))
- `optimisation.py`: The main module that holds the PSO class (the key class with the PSO algorithm, send & receive functionality for drive & optical signal(s) respectively, plotting, etc.) and the `run_test` function, which is a function used for multiprocessing (running multiple PSO experiments in parallel - not required but recommended to speed up your tests).

You are encouraged to change the code in these, add new functionality, re-write parts etc. Each file has reasonably good comments and you should be able to follow the code. `optimisation.py` is the key file which holds the PSO code, so this is likely the file you will want to become most familiar with.

3.3 Contribute

This guide will help you contribute to e.g. fix a bug or add a new feature for `soa_driving`.

3.3.1 Development Workflow

1. If you are a first-time contributor:

- Go to https://github.com/cwfpersonson/soa_driving and click the “fork” button to create your own copy of the project.

- Clone the project to your local computer:

```
git clone git@github.com:your-username/soa_driving.git
```

- Navigate to the folder `soa_driving` and add the upstream repository:

```
git remote add upstream git@github.com:cwfpersonson/soa_driving.git
```

- Now, you have remote repositories named:

- upstream, which refers to the `soa_driving` repository
- origin, which refers to your personal fork

- Next, you need to set up your build environment. Here are instructions for two popular environment managers:

- `venv` (pip based)

```
# Create a virtualenv named ``soa_driving-dev`` that lives in the
↳directory of
# the same name
python -m venv soa_driving-dev
# Activate it
source soa_driving-dev/bin/activate
# Install main development and runtime dependencies of soa_driving
pip install -r <(cat requirements/{default,docs}.txt)
#
# These packages require that you have your system properly configured
# and what that involves differs on various systems.
#
# In the soa_driving root directory folder, run
python setup.py develop
```

- `conda` (Anaconda or Miniconda)

```
# Create a conda environment named ``soa_driving-dev``
conda create --name soa_driving-dev
# Activate it
conda activate soa_driving-dev
# Install main development and runtime dependencies of soa_driving
conda install -c conda-forge `for i in requirements/{default,doc}.txt; do
↳echo -n " --file $i "; done`
#
# These packages require that you have your system properly configured
# and what that involves differs on various systems.
#
# In the soa_driving root directory folder, run
python setup.py develop
```

- Finally, it is recommended you use a pre-commit hook, which runs `black` when you type `git commit`:

```
pre-commit install
```

2. Develop your contribution:

- Pull the latest changes from upstream:

```
git checkout master
git pull upstream master
```

- Create a branch for the feature you want to work on. Since the branch name will appear in the merge message, use a sensible name such as ‘bugfix-for-issue-1480’:

```
git checkout -b bugfix-for-issue-1480
```

- Commit locally as you progress (`git add` and `git commit`)

3. Submit your contribution:

- Push your changes back to your fork on GitHub:

```
git push origin bugfix-for-issue-1480
```

- Go to GitHub. The new branch will show up with a green Pull Request button—click it.
- If you want, email cwfparsonson@gmail.com to explain your changes or to ask for review.

4. Review process:

- Your pull request will be reviewed.
- To update your pull request, make your changes on your local repository and commit. As soon as those changes are pushed up (to the same branch as before) the pull request will update automatically.

Note: If the PR closes an issue, make sure that GitHub knows to automatically close the issue when the PR is merged. For example, if the PR closes issue number 1480, you could use the phrase “Fixes #1480” in the PR description or commit message.

5. Document changes

If your change introduces any API modifications, please update `doc/release/release_dev.rst`.

If your change introduces a deprecation, add a reminder to `doc/developer/deprecations.rst` for the team to remove the deprecated functionality in the future.

Note: To reviewers: make sure the merge message has a brief description of the change(s) and if the PR closes an issue add, for example, “Closes #123” where 123 is the issue number.

3.3.2 Divergence from upstream master

If GitHub indicates that the branch of your Pull Request can no longer be merged automatically, merge the master branch into yours:

```
git fetch upstream master
git merge upstream/master
```

If any conflicts occur, they need to be fixed before continuing. See which files are in conflict using:


```
git status
```

Which displays a message like:

```
Unmerged paths:
  (use "git add <file>..." to mark resolution)

   both modified:   file_with_conflict.txt
```

Inside the conflicted file, you'll find sections like these:

```
<<<<<< HEAD
The way the text looks in your branch
=====
The way the text looks in the master branch
>>>>>> master
```

Choose one version of the text that should be kept, and delete the rest:

```
The way the text looks in your branch
```

Now, add the fixed file:

```
git add file_with_conflict.txt
```

Once you've fixed all merge conflicts, do:

```
git commit
```

c

3.4 License

soa_driving is distributed with the Apache License 2.0.

3.5 Citing

If you use this package or any derivatives of it, please cite it with:

```
@ARTICLE{parsonson2020,
  author={C. W. F. {Parsonson} and Z. {Shabka} and W. K. {Chlupka} and B. {Goh} and G.
  ↪ {Zervas}},
  journal={Journal of Lightwave Technology},
  title={{Optimal Control of SOAs With Artificial Intelligence for Sub-Nanosecond_
  ↪Optical Switching}},
  year={2020},
  volume={38},
  number={20},
  pages={5563–5573}, }
```


CHAPTER 4

Index

- `genindex`
- `modindex`
- `search`