A reduced order modelling methodology for external cylindrical concentrated solar power central receivers

University of Cape Town

Prepared by:

James M. Heydenrych

HYDJAM002

Department of Mechanical Engineering

University of Cape Town

Supervised by:

Pieter G. Rousseau

Colin F. du Sart

October 2023

Submitted to the Department of Mechanical Engineering at the University of Cape Town in partial fulfilment of the academic requirements for a Masters of Science degree in Mechanical Engineering

Key Words: Concentrated solar power, supercritical carbon dioxide, solar energy

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

Abstract

The use of supercritical carbon dioxide (sCO_2) power cycles for concentrated solar power (CSP) applications is becoming increasingly attractive since these cycles may offer lower capital costs and increased thermal efficiency. However, there are currently no utility-scale sCO_2-CSP tower plants in operation. Therefore, to aid in the design and analysis process, there is a need to develop sufficiently accurate and computationally inexpensive models for such plants. This dissertation presents a reduced order modelling methodology for external cylindrical concentrated solar power central receivers.

The methodology is built on a one-dimensional thermofluid network to model the heat transfer through the tube walls, coupled to a fluid flow network of the solar salt flowing inside the tubes. This is combined with a neural network surrogate model to determine the radiative heat flux impinging upon the tube surfaces.

The receiver geometry is discretized along the height and around the circumference and each increment is represented by an equivalent thermal resistance network that represents the heat transfer within the tube walls. The heat transfer network parameters are calibrated using a detailed computational fluid dynamics model, which enables the calculation of the maximum tube wall temperatures. The heat transfer network is connected to the fluid flow network that solves the mass, energy, and momentum balance equations to determine the mass flow rates, pressure drops and temperature distributions.

The radiative heat flux profile impinging on the receiver is typically calculated for a specific location and specific time of the day using a tool such as SolarPILOT. However, this can be computationally expensive since the central tower is surrounded by thousands of individual heliostats that are all sources of radiative flux, which depends on the position relative to the sun and relative to the receiver, as well as the direct normal irradiation (DNI) at that location and time. To reduce the associated computational expense, a multilayer perceptron (MLP) surrogate model is developed that allows the prediction of the flux profile for a range of plant configurations and atmospheric conditions at a specific location.

The application of the methodology is demonstrated via a case study. The methodology may be used in future studies where sCO_2-CSP tower plants are investigated, especially those with an interest in the detail design and analysis of the central receiver.

Declaration

I, James Michael Heydenrych, know the meaning of plagiarism and declare that all the work in the document, save for that which is properly acknowledged, is my own. This thesis/dissertation has been submitted to the Turnitin module (or equivalent similarity and originality checking software) and I confirm that my supervisor has seen my report and any concerns revealed by such have been resolved with my supervisor.

The work in this dissertation includes work published in HEFAT-ATE 2022 - The 16th International Conference on Heat Transfer, Fluid Mechanics and Thermodynamics and the Editorial Board and Applied Thermal Engineering (Heydenrych, Rousseau & du Sart, 2022). Work in this dissertation was also used for a paper currently under review for the 17th International Heat Transfer Conference.

Signed by candidate

24 October 2023

Signature

Date

Acknowledgements

I would like to thank my supervisor, Prof Pieter Rousseau, for his guidance, patience, and kindness over the past two years. The opportunity of working towards a master's degree would not have been possible without your belief in me.

I would like to thank my co-supervisor, Colin du Sart, for your guidance and specifically for helping with my writing. Your constant commitment to this project through extremely hard personal circumstances is something I truly appreciate.

I would like to thank Ryno Laubscher for his input and with consultations for the machine learning component of this project.

I would like to thank my colleagues in the ATProM research unit for your friendship and assistance over the last two years. I would especially like to thank Brad Rawlins for your help with the CFD component of my project, Kristina Laugksch for your help with the machine learning component, and Priyesh Gosai for your mentorship and professional development opportunities.

I would like to thank my girlfriend Marilyn for her constant love, kindness, and support over the past two years. Without your love my masters would have been infinitely more difficult.

I would like to thank my family for their consistent support over the past two years, to my father for believing in me and financially supporting me in this time, to my mother for her unwavering kindness, and to my sister for her support.

Table of Contents

1. Introduction 12
   1.1 Background and motivation 12
   1.2 Problem statement 13
   1.3 Project scope and objectives 14
2. Literature review 15
   2.1 Solar receiver 15
      2.1.1 Solar receiver geometry 15
      2.1.2 Solar receiver modelling 16
   2.2 Heat transfer fluids 17
   2.3 Thermal energy storage 20
   2.4 Machine learning modelling 23
   2.5 Summary 24
3. Solar receiver thermofluid model 26
   3.1 Solar receiver geometry 27
   3.2 Governing equations 28
   3.3 Half tube model development 29
   3.4 Solution method 35
   3.5 Model calibration 39
   3.6 Grid independence study 41
   3.7 Validation 43
   3.8 Summary 46
4. Data-driven surrogate modelling 47
   4.1 Data generation 48
      4.1.1 Flux surrogate model 48
      4.1.2 Thermofluid surrogate model 50
   4.2 Machine learning modelling 52

4.2.1 Flux surrogate model training 54

4.2.2 Thermofluid surrogate model 57

5. Monte Carlo analysis 59

5.1 Demonstration plant 60

5.2 Statistical methods 64

5.3 Results 67

6. Conclusions and Recommendations 70

6.1 Conclusions 70

6.2 Recommendations 71

7. List of References 72

Appendix A. View factor calculation 75

Appendix B. Validation Case 1C Jupyter Notebook 76

Appendix C. CoPylot script 96

Appendix D. Solar angle calculation 100

Appendix E. TSM training data script 101

Appendix F. FSM neural network training 107

Appendix G. TSM Pytorch training 117

Appendix H. PCRH cycle parameters 127

Appendix I. Solar salt heat exchanger and receiver design 130

Appendix J. FLiBe heat exchanger calculations 135

List of Figures

Figure 1: SPT CSP plant schematic 13

Figure 2: Solar Receivers (a) External Cylindrical Receiver (b) Cavity Receiver taken from Hashem, Wai & Basim, 2018 15

Figure 3: Two tank TES schematic 20

Figure 4: Packed Bed Single Vessel Thermocline TES System taken from Angelini, Lucchini & Manzolini, 2014 and Wu, Xu & He, 2014 21

Figure 5: Transient Salt Tank Model taken from (Zaversky et al., 2013) 22

Figure 6: External cylindrical solar receiver, adapted from Zhang et al., 2021 27

Figure 7: Discretised solar receiver flow diagram 28

Figure 8: Discretised control volume 28

Figure 9: Discretized half-tube geometry 30

Figure 10: Equivalent thermal resistance diagram 31

Figure 11: Schematic of discretised areas 32

Figure 12: Schematic of thermal resistance lengths 34

Figure 13: Nodal heat transfer solving procedure 36

Figure 14: Solar receiver flow network solving algorithm 38

Figure 15: CFD computational domain 39

Figure 16: Ansys Fluent thermal contour plot 41

Figure 17: Grid independence study 42

Figure 18: Grid independence study increment height vs temperature 42

Figure 19: Validation flux and maximum tubular temperature heat maps 44

Figure 20: DNI vs azimuthal angle 49

Figure 21: MLP architecture 52

Figure 22: Error vs size of dataset 55

Figure 23: Error vs model complexity 55

Figure 24: Actual flux from dataset vs DDSM predicted flux 56

Figure 25: Flux map comparison, left - FSM, right - SolarPilot 56
Figure 25: Flux map comparison, left - FSM, right - SolarPilot 56
Figure 26: Thermofluid surrogate model value from dataset vs DDSM predicted value (solar salt) 57
Figure 27: PCRH sCO2 Brayton cycle 60
Figure 28: Solar receiver, thermal energy storage and heat exchanger schematic 61
Figure 29: Annotated solar receiver architecture schematic 62
Figure 30: Discretised solar receiver flow diagram 63
Figure 31: DNI fitted distribution histogram 64
Figure 31: Wind and temperature fitted distribution histogram 64
Figure 33: Monte Carlo analysis methodology 66
Figure 34: Monte Carlo analysis plant energy output 67
Figure 35: Monte Carlo analysis maximum tubular temperature 68
Figure 36: Flux profiles for varying seasons 69

List of Tables

Table 1: Molten Salt Properties (Serrano-López, FradEera & Cuesta-López, 2013) 19

Table 2: Fluent calibration 40

Table 3: View factors 40

Table 4: Grid independence study 43

Table 5: Solar Two receiver geometry 44

Table 6: Validation results 45

Table 7: FSM independent variables 48

Table 8: Solar salt TSM independent variables for LHS 51

Table 9: FLiBe TSM independent variables for LHS 51

Table 10: FSM hyperparameters 55

Table 11: Solar Salt main heater and reheater parameters 61

Table 12: FLiBe main heater and reheater parameters 62

Table 13: Case study solar receiver geometry 63

List of Nomenclature

General symbols

A	Area	m^2
C_p	Specific heat	J/kg K
D	Hydraulic diameter	m
e_t	Emissivity	
f	Friction factor	
F_{view}	View factor	
g	Gravitational acceleration	m/s^2
H	Enthalpy	J/kg
h	Heat transfer coefficient	W/m^2
K	Sum of secondary loss factor	
k	Thermal conductivity	W/mK
L	Length	m
$\dot{m}$	Mass flow rate	kg/s
Nu	Nusselt number	
p	Pressure	Pa
Pr	Prandtl number	
$\dot{Q}$	Heat transfer	W
Q_{flux}	Heat flux	W/m^2
R	Thermal resistance	K/W
Ra	Rayleigh number	
Re	Reynolds number	
T	Temperature	K
u	Velocity	m/s
$\dot{V}$	Volume flow rate	m^2/s
v_c	Kinematic viscosity	m^2/s

Greek symbols

ρ	Density	kg/m^3
Δx	Length of increment	m
σ	Stefan Boltzmann constant	W/m^2K^4

σ_l	Activation function	
α	Thermal diffusivity	m^2/s
β	Thermal expansion coefficient	1/K

Acronyms and Abbreviations

a^l	Layer output
ACC	Air cooled condenser
ANN	Artificial neural networks
b	Back facing to radiation
b	Bias vector
bb	Most back facing to radiation
C	Compression
c	Central node
CFB	Coal fired boiler
CFD	Computational fluid dynamics
CSP	Concentrated solar power
CSR	Cavity solar receiver
CST	Cold salt tank
DDSM	Data-driven surrogate model
DoE	Department of energy
DOE	Design of experiments
ECSR	External cylindrical solar receiver
Er	Error
f	Front facing to radiation
ff	Most front facing to radiation
FSM	Flux surrogate model
G	Generator
HHFM	Homogenous heat flux model
HPC	High pressure compressor
HPT	High pressure turbine
HRX	High heat recuperator
HST	Hot salt tank
HTF	Heat transfer fluid
HTM	Homogenous temperature model
l	Neural network layer
LCOE	Levelized cost of electricity

LHS	Latin hypercube sampling
LPC	Low pressure compressor
LPT	Low pressure turbine
LRX	Low heat recuperator
LSTM	Long short term memory
MDN	Mixture density network
MH	Main heater
MHP	Main heater pump
MLP	Multi-layer perceptron
MS	Molten salt
MSE	Mean squared error
NREL	National renewable energy laboratories
PC	Cooling
RH	Reheater
RHP	Reheater pump
RNN	Recursive neural network
RVS	Random variate sampling
RX	Recuperation
sCO_2	Supercritical Carbon Dioxide
SGD	Stochastic gradient descent
SM	Surrogate model
SPT	Solar power tower
SR	Solar receiver
T	Expansion
TES	Thermal energy storage
TIT	Turbine inlet temperature
TSM	Thermofluid surrogate model
VAE	Variational auto encoder
w	Weight matrix
$\hat{y}$	Value predicted in forward propagation
Y	Target value from dataset

1. Introduction

1.1 Background and motivation

Concentrated solar thermal power (CSP) is one of the few renewable technologies with proven energy storage capabilities (Reyes-Belmonte et al., 2016). This means that CSP technologies can dispatch energy constantly, unlike most forms of renewable energy, which produce and dispatch electricity intermittently. This is achieved by heating and storing a heat transfer fluid (HTF), typically a molten salt solution, during periods of high solar irradiance and then using this solution when there is little or no solar irradiation. Various CSP technologies exist, however, the technology of interest includes a solar power tower (SPT) configuration as it has the greatest potential for scaling and achieving higher receiver efficiencies (Conroy, Collins & Grimes, 2020). Consequently, SPT is forecast to be the dominant CSP technology in the future (Wang, He & Zhu, 2017).

South Africa currently has the fifth largest installed electrical capacity of CSP plants in the world after Spain, the United States, China, and Morocco (NREL, 2022). If CSP technology receives further investment in South Africa the country could be primed to become a world leader in the technology. This could provide much needed support for the South African economy, since it would contribute to the security of supply that is currently a serious concern.

The supercritical Carbon Dioxide (sCO_2) Brayton cycle has been identified widely in literature as having the potential to increase the thermal efficiency of a wide variety of power plants, especially nuclear and CSP plants. Furthermore, it is attractive because lower mass flow rates are required compared to conventional steam power plants, allowing for the use of more compact turbomachinery.

The combination of the aforementioned technologies is widely discussed in literature, with the sCO_2 CSP plant seen as an ideal candidate for the third generation of CSP plants (Mehos et al., 2017). A schematic of such a plant, with a molten salt thermal energy storage (TES) system is provided in Figure 1. The central tower houses the solar receiver (SR), which captures solar irradiance concentrated towards it by a field of heliostats. Molten salt is pumped through the receiver from a cold salt tank (CST), heated in the receiver, and then stored in the hot salt tank (HST). The MS is then used to heat sCO_2, which serves as the working fluid in a Brayton cycle type power block. The power cycle includes single or many stages of compression (C), expansion (T), heating (H), cooling (PC), recuperation (RX), and a generator (G).

While the modelling of both solar receivers and main heat exchangers for sCO_2 CSP plants has been widely reviewed in literature, many of these models are computationally expensive to solve.

Generally, the solar receiver models are discretised in a 2-D or 3-D fashion leading to networks of over 1000 nodes, and nodal networks for the main heat exchanger are similarly large. Solar receiver models are implemented this way so that the maximum wall temperature of the receiver can be determined. However, using these model to run transient or quasi-steady state Monte-Carlo simulations of a full plant will require significant computational resources and time. Therefore, a fast 1-D steady state model that can calculate the solar receiver wall temperatures with sufficient accuracy would valuable for the modelling of an entire sCO_2 CSP plant.

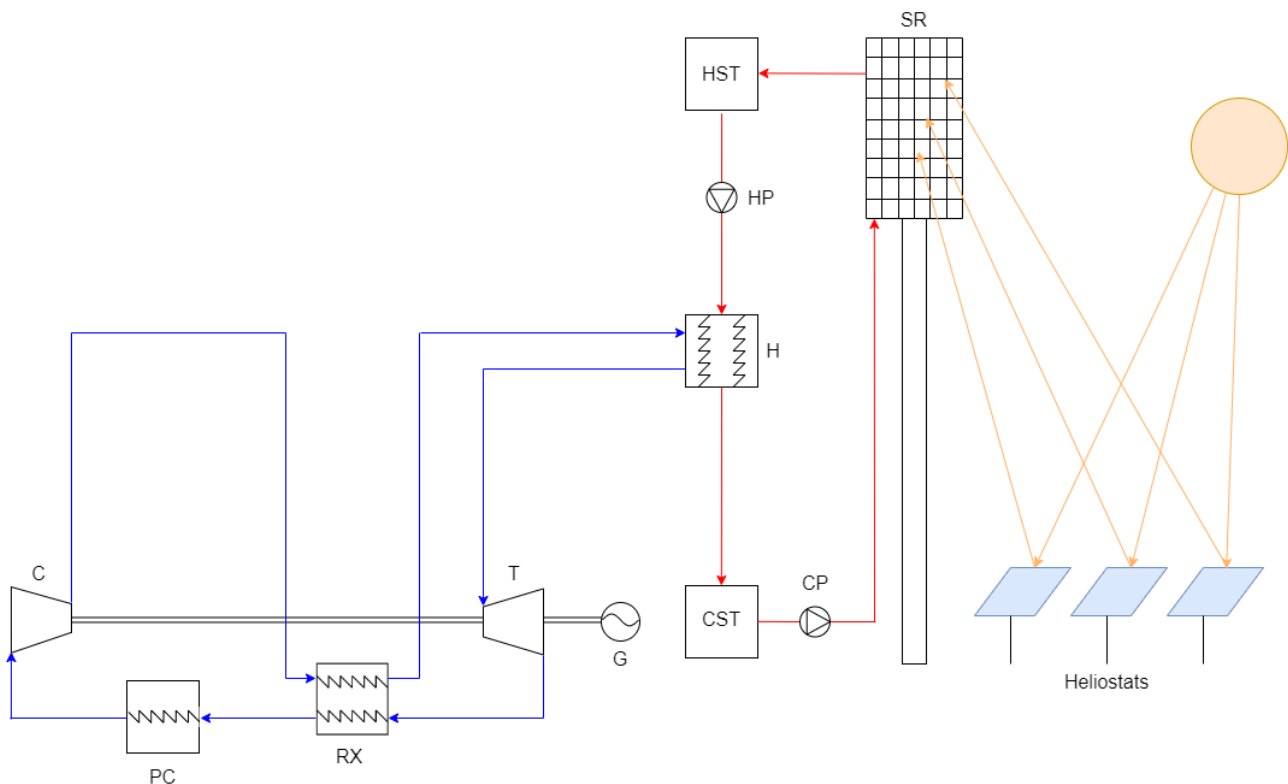


Figure 1: SPT CSP plant schematic

1.2 Problem statement

Modelling a complete integrated sCO_2-CSP plant using conventional 2-D and 3-D approaches will require significant computational resources and time. The computational expense comes not only from modelling the thermofluid phenomena in the receiver flow network, but also from the calculation of the radiative heat flux impinging upon the receiver. While calculating the heat flux can be implemented easily using off-the-shelf solutions such as SolarPILOT (Wagner & Wendelin, 2018), this remains a computationally expensive task as the central tower is surrounded by thousands of individual heliostats, which are all sources of radiative flux. A need therefore exists for a sufficiently accurate and computationally inexpensive model of the solar field and central receiver

of sCO_2-CSP tower plants. While 1-D thermofluid modelling greatly reduces computational expense compared to 2-D and 3-D modelling, it can potentially be reduced even further by applying artificial neural networks (ANNs). If a methodology can be developed to exploit ANNs for this application, it could provide a fast and sufficiently accurate reduced order model of the solar field and central receiver. This can be integrated with the rest of the plant to conduct quasi-steady state Monte-Carlo or dynamic simulations.

1.3 Project scope and objectives

This project aims to develop a reduced order modelling methodology for the solar field and central receiver of a sCO_2 CSP plant. This will be achieved by completing the following objectives:

- Develop a calibrated and validated discretized thermofluid process model of a solar receiver using a one-dimensional (1D) thermofluid network modelling approach.
- Use machine learning techniques to create a data-driven surrogate model (DDSM) which can accurately predict the impinging radiative heat flux onto the solar receiver.
- Create a DDSM of the solar receiver thermofluid process model.
- Integrate the two DDSMs to create an integrated solar field and central receiver model.
- Demonstrate the use of the integrated model by doing an Monte Carlo simulation over a full year of operation using statistical models and full-year meteorological and climatological data.

The scope of the project includes the development of a 1-D thermofluid process model of a solar receiver, the development of a DDSM of the process model, and the development of a DDSM to predict the radiative flux impinging upon the receiver. The data for the heat flux impinging upon the receiver will be generated using SolarPILOT rather than from first principles. The scope does not include the modelling of the rest of the sCO_2 CSP plant. Lastly, this project is limited to the use of proven CSP technologies, rather than the development of new state-of-the-art technologies.

2. Literature review

2.1 Solar receiver

2.1.1 Solar receiver geometry

The solar receiver is one of the most important components of a SPT CSP plant. The two most common solar receiver types are the external cylindrical solar receiver (ECSR) and the cavity solar receiver (CSR), as shown in Figure 2 (Hashem, Wai & Basim, 2018). Both receivers consist of headers which split into multiple tubes coated in solar absorbent material.

The type of solar receiver is selected based on a wide variety of factors. These factors include the plant's geographical location, size, HTF used as well as the inlet and outlet HTF temperatures (Zheng et al., 2020). Zheng, et al. (2020) mentioned that a consistent comparison between receiver types is not possible as it relies on this wide variety of factors. However, each receiver type has its own set of advantages and disadvantages.

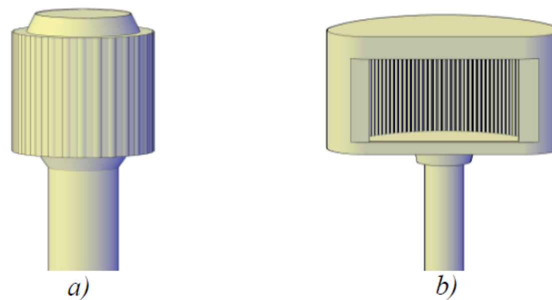


Figure 2: Solar Receivers (a) External Cylindrical Receiver (b) Cavity Receiver taken from Hashem, Wai & Basim, 2018

Cavity receivers were widely used in early first generation CSP SPT plants such as the Solar One pilot plant run by the United States Department of Energy (DoE). Cavity receivers are thought to be a leading candidate for the next generation of SPT CSP plants as their enclosed nature leads to lower radiative losses at higher temperatures (above 600°C), leading to more efficient receivers (Montes et al., 2020). This is because cavity receivers have a higher degree of reradiation to mitigate radiative losses. It was found that radiative losses were decreased by 41% through reradiation in cavity receivers (Wang, Qiu, et al., 2020). Radiative losses become an important design factor at higher temperatures since radiative losses increase exponentially with increasing temperature as they are proportional to the fourth power of the tubular wall temperature. Due to the nature of the enclosure of a cavity receiver, the heliostat field is required to be either fully north or south facing

depending on the hemisphere the plant is built in. This leads to taller towers needing to be constructed when compared to external cylindrical receivers of the same capacity, as a fully north/south facing field means the heliostats are further from the receiver on average.

External cylindrical receivers are currently the most used solar receiver technology in industry. The design has been used in many operational grid-connected plants such as Solar Two and Crescent Dunes in the US, GemaSolar in Spain, and Redstone in South Africa which is due to commence commercial operation in late 2023. All three of these plants have a receiver outlet temperature between 550 and 600°C. The National Renewable Energy Laboratory (NREL) suggest that external receivers are the most mature technology and research should be focused on perfecting these receivers rather than the design of new types of receivers (Mehos et al., 2017).

2.1.2 Solar receiver modelling

The steady state analysis and dynamic simulation of tubular receivers is a topic of research with increasing interest. Many approaches may be used to model these receivers. However, three-dimensional (3D) computational fluid dynamics (CFD) models, and 1D thermofluid network models are mostly used. The latter is gaining popularity as it is computationally less expensive when compared to CFD modelling (Conroy et al., 2020).

Rodriguez-Sanchez (2014) conducted a thorough review of a number of heat transfer models for external cylindrical solar receivers. In their work, they developed a standard simplified discretized network model, two more complex discretized network models, as well as a CFD simulation. These discretized models use heat transfer coefficients to calculate the heat transfer from radiation being absorbed by the heat transfer fluid (HITEC in this case). The models include thermal losses due to natural and forced convection as well as radiation. The two complex network models account for both axial and circumferential variation in the tubular wall temperature. The iterative solutions for the two network models, which account for circumferential variation of the tubular wall temperature, are calculated relatively similarly. The first model iterates for each increment of the tubular receiver until the heat flux converges and is referred to as the homogenous heat flux model (HHFM). The other model iterates until the wall temperature of the receiver converges and is referred to as the homogenous temperature model (HTM). The standard model is similar to the HHFM, but it only accounts for the axial variation in the tubular wall temperature (Rodríguez-Sánchez et al., 2014).

It was found that the simplified standard model was not able to accurately predict the maximum tubular wall temperature, underestimating it by 25% (200°C). The HHFM and HTM both were able to accurately estimate the maximum tubular wall temperature to within 4% (30°C) of the CFD model.

The underestimation of the tubular wall temperature put the longevity of the plant at risk as it will underestimate the thermal fatigue of the tubes as well as potential chemical decomposition of the HTF.

The physical similarities between solar receivers and the membrane walls of pulverized coal boiler furnaces mean that thermofluid network-based models of these technologies may be similarly developed. Rousseau and Laubscher (2020) created a model of a membrane wall using a network-based approach with equivalent thermal resistances. They found that the model accurately modelled the conductance (UA value) of the membrane wall to within 5% of an equivalent CFD simulation as well as predicting the maximum fin temperature to within 2% of the CFD simulation. This model was created to accurately predict the maximum fin temperature of the membrane wall without needing to discretize the model into a 2-D or 3-D geometry (Rousseau & Laubscher, 2020). In the context of this work, Rousseau and Laubscher's (2020) model was created to solve a problem similar to one that exists with solar receiver models, where the maximum tubular/wall temperature of the radiation surface is to be estimated.

2.2 Heat transfer fluids

Heat transfer fluids can be separated into five categories. First, and most commercially proven technology for SPTs is molten salts, which have the longest proven TES capacity with commercial plants having up to twelve hours of TES being operational (Achkari & Fadar, 2020). A second choice of HTF is directly heating sCO_2. This however, requires thicker piping than molten salts due to its higher pressure as well as requiring an additional sCO_2 to molten salt heat exchanger if significant TES is required (Ma & Turchi, 2011). A third choice would be using a thermal oil, which generally limits the TES capacity to around six hours, as well as having lower maximum operating temperatures when compared to molten salts. (NREL, 2022) Fourth are liquid metals, which function in a similar manner to molten salts. Though they have potential to allow for higher maximum operating temperatures there is significant risk due to high corrosion rates and flammability (Heller, 2013). Fifth is the particle-based receiver, which promises to have high efficiencies and operate at higher temperatures than other HTFs, but is not proven technology at this stage (Ho, 2016).

There are several types of molten salt that can be used in CSP plants. The most common and mature technology currently is a eutectic mixture of 60% $NaNO_3$ and 40% KNO_3. This mixture is commonly referred to in industry as "Solar Salt". This mixture is a stable liquid between 223°C and 600°C (Vignarooban et al., 2015). Once above 600°C the mixture begins to behave erratically, partially boiling and ionizing while also causing high rates of corrosion. One of Solar Salt's main advantages is in its high volumetric heat capacity compared to other thermal heat transfer media (Conroy et al., 2020). Solar Salt's volumetric heat capacity is 2.72 MJ/m^3K while other chloride-based salts typically

have volumetric heat capacities in the region of 1.9 MJ/m^3K. The drawback of this medium is the fact that due to its thermal stability limit of 600°C, the maximum temperature of Solar Salt is generally limited to 565°C. This means that the turbine inlet temperature (TIT) is generally limited to 550°C in a system that deploys this technology. Table 1 provides properties of various eutectic molten salt mixtures.

The US DoE currently has a goal of developing mature technology that allows for both TES and TITs of above 700°C (Mehos et al., 2017). This is called the SunShot Initiative. This would require a HTF to be thermally stable at 750°C, which is a technical hurdle that currently has not been overcome, although research is ongoing. Another avenue being researched with funding from the SunShot Initiative is using liquid metals such as sodium as a heat transfer medium (Turchi et al., 2021).

Recent research on molten salt technologies has been promising, with chloride, fluoride, and carbonate mixtures all having been shown to be thermally stable at up to 1000°C. At these temperatures, the thermal efficiency of the Brayton cycle may be increased. However, technical issues surrounding corrosion have not been resolved to this point. Another issue with these salt mixtures is their high melting points, and therefore high solidification point of between 300°C and 450°C compared to solar salt's 223°C (Serrano-López, Fradera & Cuesta-López, 2013). The higher solidification temperatures would lead to many technical difficulties as molten salt solidifying in a heat exchanger or piping would be catastrophic and likely result in equipment needing to be replaced.

Liquid metals such as sodium have been proven to work in previous test plants in Spain. The advantage with using liquid Sodium is its wide temperature range, remaining a stable liquid between 98°C and 883°C. However, issues remain such as its combustibility when exposed to water and the fact that it is close to four times the cost of liquid salt (Vignarooban et al., 2015). This means that it has never been used in a commercial plant, though a 1 MW_{th} test receiver has been proposed to test a liquid sodium receiver coupled with a molten salt TES system (Turchi et al., 2021).

Table 1: Molten Salt Properties (Serrano-López, FradEera & Cuesta-López, 2013)

	Melting Point	Maximum Operating Temperature	Heat Capacity (Cp)	Density	Volumetric Heat Capacity	Dynamic Viscosity	Thermal Conductivity
	°C	°C	J/kg.K	kg/m^3	kJ/m^3.K	Pa.s	W/m.k
Solar Salt	222	600	1396.044+0.172·T	2263.628-0.636·T	2724.69	$0.075439-2.77\cdot10^{-4}\cdot(T-273)+3.49\cdot10{-}7\cdot(T-273)^2+1.474\cdot10^{-10}\cdot(T-273)^3$	0.45
Hitec	142	500	1560	2279.799-0.7324·T	2699.58	exp(-4.343-2.0143·(ln(T-273)-5.011))	0.48
FLiBe	454-459	821	2385	2413.03-0.4884·T	4826.19	$1.16\cdot10^{-4}\cdot\exp(3755/T)$	1.1
CloKMag	426-435	756.85	1155	2007-0.4571·T	1922.12	$1.408\cdot10^{-4}\cdot\exp(2261.3/T)$	0.55

Solar Salt refers to a eutectic mixture of 60% $NaNO_3$ and 40% KNO_3

Hitec refers to a eutectic mixture of 7% $NaNO_3$, 49% $NaNO_2$ and 44% KNO_3

FLiBe refers to a eutectic mixture of 66% 2LiF and 34% BeF_2

CloKMag refers to a eutectic mixture of 68% KCl and 32% MgC

2.3 Thermal energy storage

TES is a key component of a CSP plant as it allows the plant to continue to produce electricity even during times with low solar irradiance. Currently the most mature TES technology being used in a variety of commercially operating CSP plants is a two-tank molten salt system. This involves molten salt being pumped from the cold salt tank (CST) to be heated in the solar receiver and then stored in the hot salt tank (HST). The molten salt in the HST is then pumped through the source heat exchangers to be used as the heat source for the power cycle (sCO_2 cycle in this case). The cooled salt is then again stored in the CST to complete the TES cycle. A schematic of a two-tank TES system

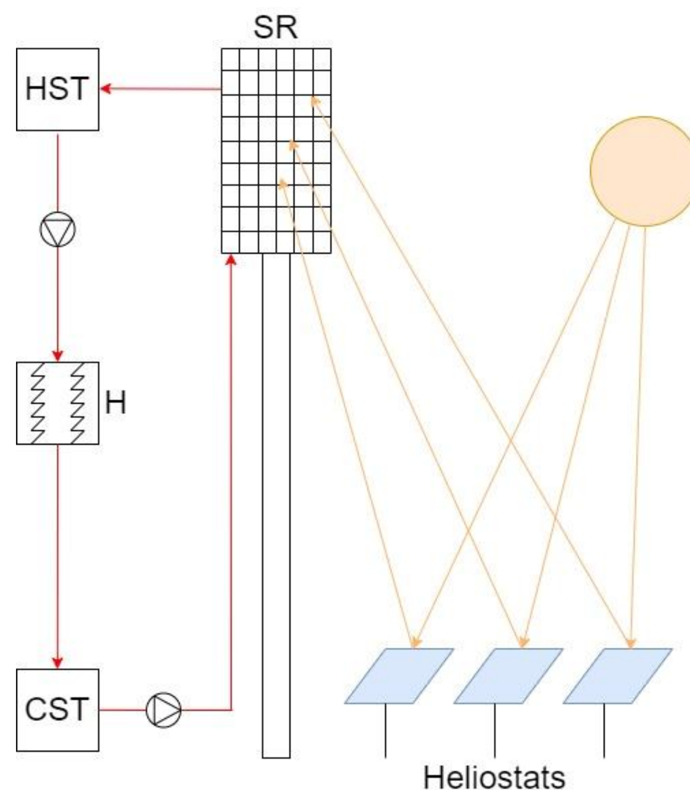


Figure 3: Two tank TES schematic

is shown in Figure 3.

One of the alternatives to two-tank TES systems being explored is a single tank thermocline TES system. This involves a single vessel being filled with molten salt where a temperature gradient exists between the top and the bottom of the tank, referred to as a thermocline. Cold salt is drawn from the bottom of the tank to be heated in the solar receiver and then pumped to the top of the tank. The hot salt at the top of the tank is then pumped through the source heater to drive the power cycle. The now cooler salt is then pumped back into the bottom of the tank to complete the cycle. Both the top and the bottom of the tank are filled with packed pebble beds to act as the

primary thermal energy store. These systems are less expensive, with studies finding that it reduces the cost of a TES system by 33% when compared to a two-tank system of the same capacity (Pacheco, Showalter & Kolb, 2002). This is not only due to fewer tanks being required, but also

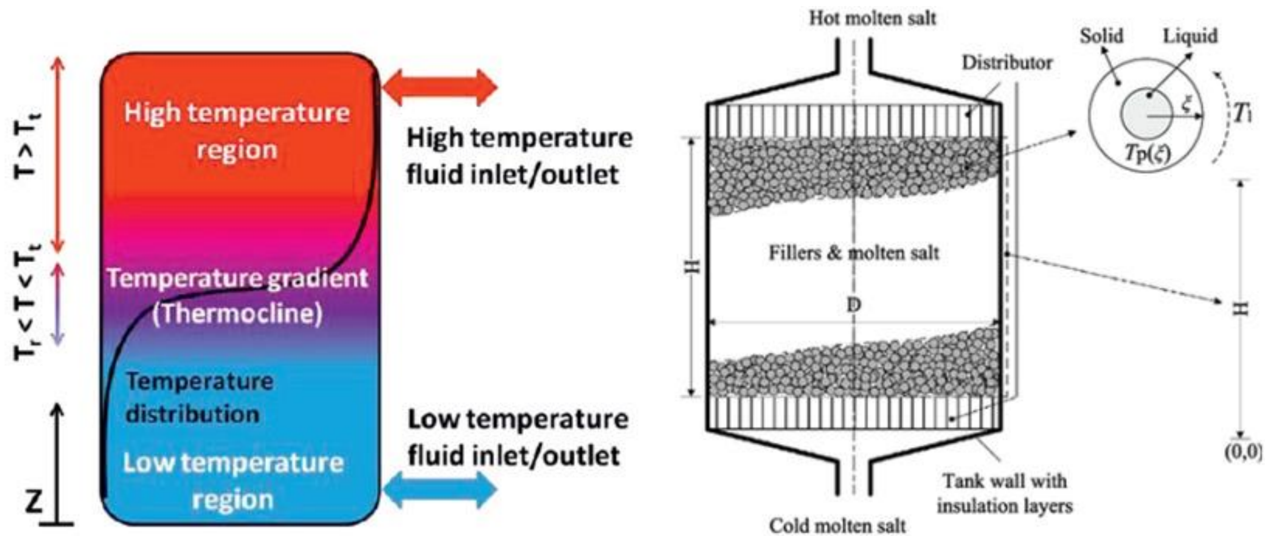


Figure 4: Packed Bed Single Vessel Thermocline TES System taken from Angelini, Lucchini & Manzolini, 2014 and Wu, Xu & He, 2014

because a packed pebble bed is a less expensive medium of TES than molten salt. Schematics of this TES technology are provided in Figure 4.

While a single vessel thermocline-based TES system is more cost effective than a two-tank TES system, the discharge efficiency is lower. This is due to the hot and cold molten salt being in direct contact with one another. It was found that a two-tank system was able to supply 100% of the heat store at above 545°C, while the single vessel thermocline could only supply 64% of the stored heat (Angelini, Lucchini & Manzolini, 2014).

The modelling of two-tank systems has been widely explored in literature, with validation data available from multiple test facilities as well as commercial plants. In the solar two test facility it was found that the heat losses were 1% or lower for the TES system (Pacheco, Bradshaw, et al., 2002). From this wide array of available data, many empirical and quasi-steady state models have been developed with varying degrees of accuracy. Zaversky et al., 2013 created a fully transient model of both the HST and CST to simulate the performance and heat transfer losses of a two-tank TES system. A schematic of the original transient model can be seen in Figure 5.

In an analysis of the detailed transient model, Zaversky et al., 2013 found that some of the heat transfer between the molten salt, the air above the molten salt inside the tank, and the tank itself was negligible and some could be assumed to be constant. The convective heat transfer between the molten salt and the wetted inner surface of the steel tank could be assumed to be constant. The

convective heat loss to the air above the molten salt surface and hence the convective and radiative losses of the air to the inner tank can be assumed to be zero. The radiative heat transfer between the molten salt surface and the non-wetted inner tank surface is an important factor that needs to be modelled. The ambient conditions including temperature, solar radiation, and wind speed all have a significant impact on the heat transfer to the environment and hence need to be modelled

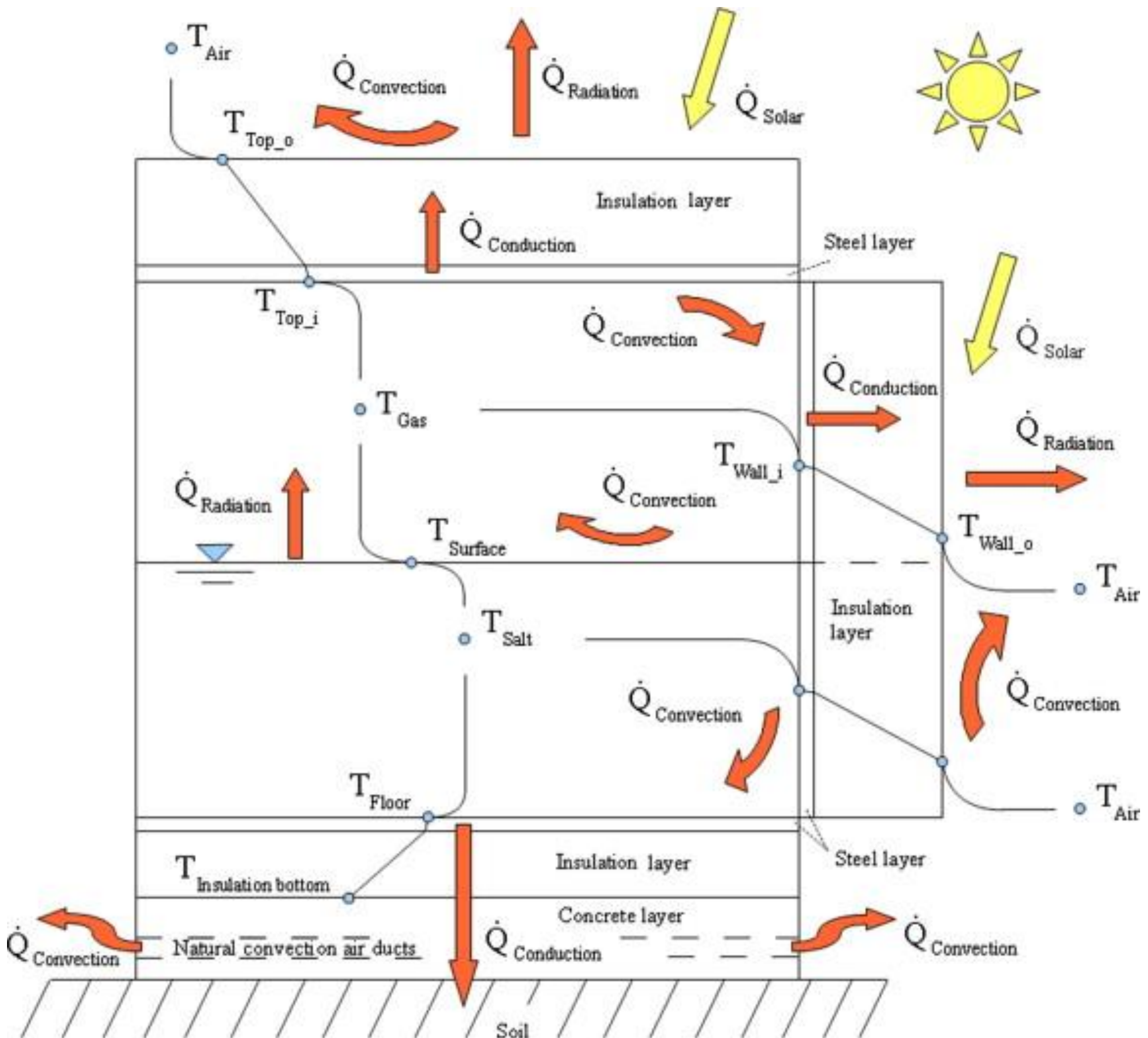


Figure 5: Transient Salt Tank Model taken from (Zaversky et al., 2013)

in a transient state. When taking these simplifications into account along with the important factors left in transient mode, the heat loss of the tanks was found to be within 0.1% of the original fully transient model.

2.4 Machine learning modelling

While data-driven surrogate modelling is a relatively new research field, the use of DDSMs to predict thermofluid phenomena has grown in recent years. While most DDSMs are developed using experimental data, several DDSMs have been developed using theoretical models, especially CFD models. The acquisition of experimental data is often prohibitively expensive, fuelling the need for computational models to develop training data. There have also been several studies which use process modelling to train a DDSM. Using a process model to create training data of ANNs to create DDSMs has the advantage of being less computationally expensive than using CFD models, as well as the other inherent advantages of using computational models to train DDSMs. While there have been many studies which have looked at modelling thermofluid phenomena in power plants, including coal fired boilers (CFBs) and air cooled condensors (ACCs) as well as the modelling of photovoltaic solar plants, very few studies have used machine learning techniques to model CSP plants.

Machine learning techniques have been used widely in literature to create DDSMs to model coal fired boilers (CFBs). Much of this work concentrates around the combustion process where the species of combustion and therefore the emissions can be modelled in a more computationally efficient manner. Tan et al. (Tan et al., 2019) used a long short-term memory (LSTM) machine learning algorithm to predict the NOx emissions from a CFB in China using real plant data. Laubscher (Laubscher, 2019) used time series site data to predict the reheater metal temperatures of a CFB in South Africa. Laubscher’s study made use of a recurrent neural network and was able to predict the tubular metal temperatures to within 1% of the site data. Liu et al. (Liu et al., 2013) used a fuzzy neural network to model the thermal efficiency of 1000 MW supercritical boiler using site data.

It was found that DDSM could accurately model of the combustion of a jet fuelled flame using a combination of a variational autoencoder (VAE) and a multilayer perceptron (MLP) neural network (Laubscher & Rousseau, 2021). Further explanations on MLPs can be found in Chapter 4.2. The dataset to train and test the DDSM was created using Ansys Fluent. The reason a VAE was used in conjunction with a MLP was due to the dimensionality of the output dataset being much larger than the dimensionality of the input dataset. The DDSM was found to be able to accurately predict temperature, velocity, and species mass fraction profiles for the flame with high level (and hence low dimensional) inputs. The VAE encodes the high dimensional output variables into a lower dimensional dataset, allowing for the MLP to be more easily trained before the outputs of the MLP are encoded with the VAE back into the original high dimensionality of the output dataset. It was found that the species mass fraction, temperature, and velocity profiles could be predicted to within 0.3%, 1.7% and 7.1% respectively.

Machine learning techniques have also been used in literature to develop DDSMs for ACCs. Raidoo and Laubscher (Raidoo & Laubscher, 2022) used an encoder-decoder mixture-density network (MDN) and a recursive neural network (RNN) to predict the backpressure of an ACC. This study used site data to train the neural network.

A MLP was used to develop a DDSM of an ACC (Haffejee & Laubscher, 2021). The MLP was trained using data produced using a process model. The process model was a fully discretized cell-based model developed in Flownex SE. The process model was validated against site data meaning that a range of data could be simulated to train the MLP. The DDSM was able to predict the backpressure of the ACC to within 6% of the validation dataset in 93.5% of the cases. The data was split into three seasons for training.

Wang, et al. (Wang, Qiu, et al., 2020) developed a DDSM to evaluate the thermal performance of a solar receiver. This model only finds the overall heat transfer and hence receiver efficiency of the solar receiver. Meteorological data for the DNI, solar altitude angle, solar azimuth angle, barometric pressure, relative humidity, ambient temperature, and wind speed were taken as independent variables. The model was validated against the thermal efficiency of the Solar Two plant and was found to be within 2% of the plant's efficiency. The training and the modelling was split into four separate models for each season. From here a case study was created for a plant based in Zhangbei, China. The same set of variables were used as inputs for a neural network, with the predicted receiver efficiency used as the output. While this study showed the ability of neural networks to predict the thermal efficiency of a solar receiver, it was not able to model the thermal profile of the tubular elements of the receiver, which are important in assessing the performance of a CSP sCO_2 plant.

It can be seen in the literature that various machine learning architectures can be used to model heat transfer as well as temperature and pressure profiles of thermodynamic phenomena with a sufficient degree of accuracy while significantly decreasing the computational times. From Haffejee and Laubscher's (Haffejee & Laubscher, 2021) work it can be seen that using a MLP, which is a relatively easily implemented neural network architecture, thermofluid phenomena can be accurately predicted with data produced using a process model. While Laubscher and Rousseau's (Laubscher & Rousseau, 2021) work shows that high dimensionality of output data can be overcome by using a variational autoencoder in conjunction with a MLP.

2.5 Summary

From the literature it is clear that the most viable technology to be used for the solar receiver is an external cylindrical receiver. The most viable steady state modelling technique for the receiver is

one based on Rodriguez-Sanchez et al.'s HHFM model, and Rousseau and Laubscher's coal furnace network-based model.

Solar salt is the most proven HTF technology according to literature, however it is only thermally stable to 600°C. Therefore, a study comparing Solar Salt to another molten salt chemistry could be valuable. A two-tank TES system is the most efficient and most proven technology. The TES system can be simplified to a steady state model as Zaversky, et al.'s transient modelling found the thermal losses to be negligible at 1%.

Machine learning techniques can model various thermofluid phenomena with sufficient accuracy while reducing solve times significantly. The literature shows that a MLP will likely be able to build a DDSM with sufficient accuracy to model the thermofluid phenomena of a SR and be trained using a process model. However, the DDSM built to predict the flux profile impinging upon the receiver may need a VAE in conjunction with a MLP due to its high dimensionality (over 100 cartesian flux points).

3. Solar receiver thermofluid model

Modelling an external cylindrical solar receiver using typical 3-D CFD models is computationally expensive. Therefore, in this work, a reduced order 1-D thermofluid modelling methodology was developed. While Rodríguez-Sánchez et al's (2014) model also employs a 1-D thermofluid model, it is finely discretized with 37 discrete circumferential nodes per element. This leads to longer solve times than necessary. The modelling methodology developed is an adaptation of the work of Rousseau & Laubscher (2020), which employs a coarser discretization, but which still provides accurate results. The model characteristics are as follows:

- It accounts for the layout of the tube network and the variations in heat flux and temperature along the height and around the circumference of the solar receiver.
- It accounts for the radiative, convective, and reflective losses to the surroundings.
- The model does not take wind direction into account as this would entail further study on the effects of pressure drops of the wind around the receiver, including the effects of a wind shadow.
- It predicts the position and magnitude of the maximum temperature of the tube material within 0.2% of CFD simulations.
- It predicts the heat transfer into the HTF (molten salt) within the tubes within 5% of plant data.
- Reasonable computational times are achieved compared to existing models, with the model taking 5 seconds to solve a single validation case. A similar study by Rodríguez-Sánchez et al., 2014 solved in 15 seconds. Rodríguez-Sánchez et al used an Intel Quad Core CPU, this study similarly used an Intel Quad Core CPU.

The proceeding text provides a description of the external cylindrical solar receiver modelled, and the methodology used to develop the 1-D thermofluid model.

3.1 Solar receiver geometry

The layout of an external cylindrical solar receiver is shown in Figure 6 (Zhang et al., 2021). The HTF, in this case molten salt, is pumped from the CST up the central tower to the inlet of the solar receiver. From here the flow of the salt is split into two flow paths, the eastern and western paths. The flow is then further split by a header into individual tubes. The HTF then flows through the tubes, where it is heated by the impinging heat flux before the flow is combined again in the next header. The combination of the lower and upper header with the tubes is referred to as a panel. The HTF continues to flow through these panels in a serpentine pattern as can be seen in a flattened cartesian representation in Figure 7. The fluid flows are then combined at the outlet before proceeding to the HST.

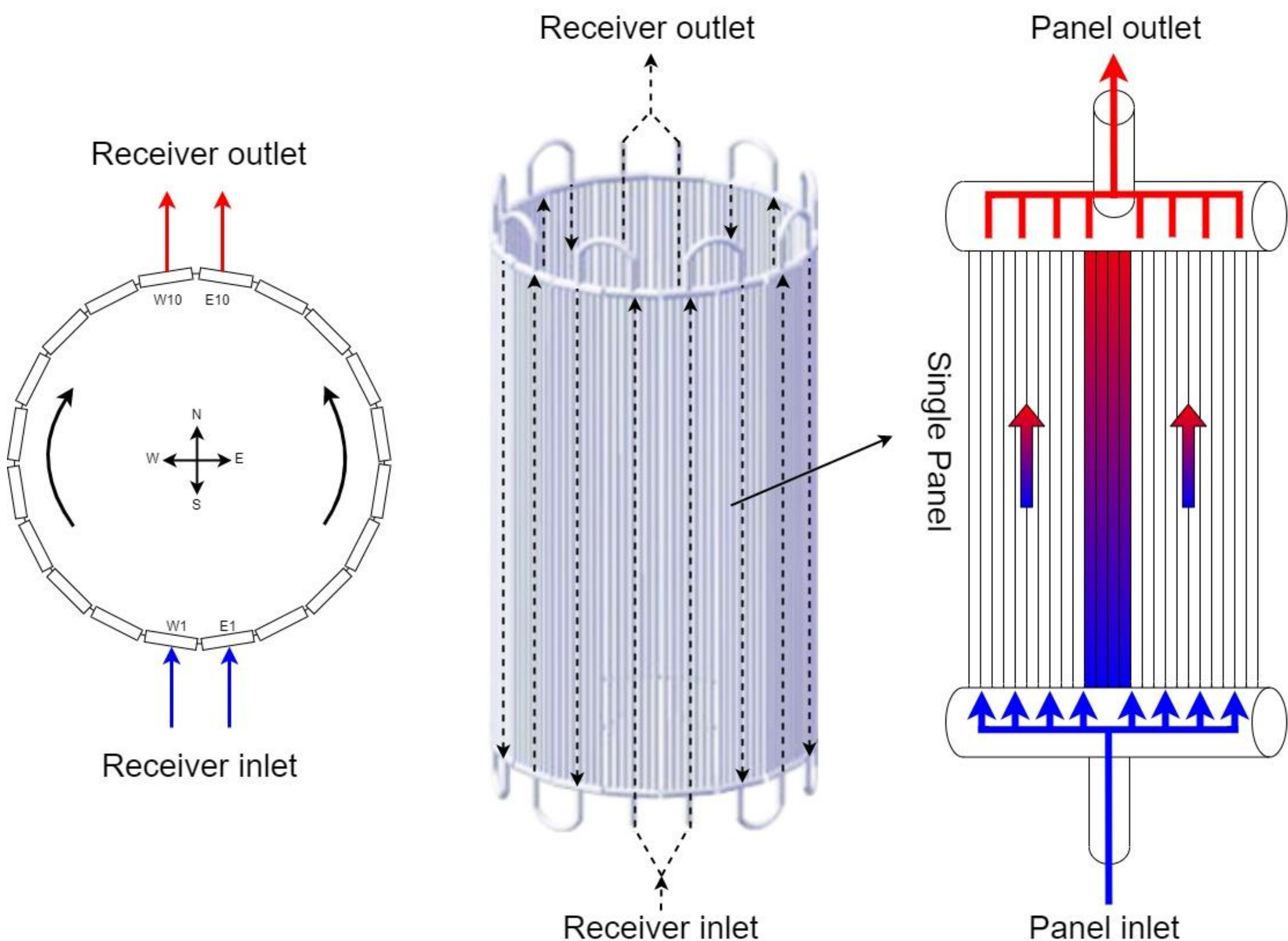


Figure 6: External cylindrical solar receiver, adapted from Zhang et al., 2021

Within a panel, it is assumed that the mass flow is uniformly distributed between the individual tubes, and that the temperature gradient across each tube is the same. Therefore, only a single representative tube is explicitly modelled per panel. Each tube is discretized vertically as shown in Figure 7. Based on the results of a detailed grid independence study (see Section 3.6), there are 40 increments for each representative tube.

Figure 7: Discretised solar receiver flow diagram

3.2 Governing equations

The modelling of the external cylindrical solar receiver follows the flow network in Figure 8 with each square represents a node with each circle representing an element. The exit node for each element is the inlet node for the subsequent element. Each element is a 1-D control volume. Each 1-D control volume solves for the steady state mass, momentum, and energy balance equations (see Figure 8 and Equations (3-1) to (3-3)). The methodology solves for the radiation impinging upon the tubes, the radiative and convective losses to the surrounding environment, the conductive resistance to heat transfer as well as the molten salt-side control volume.

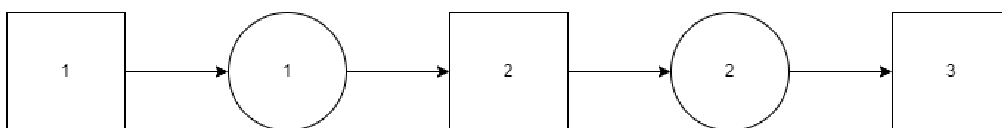


Figure 8: Discretised control volume

The 1-D modelling methodology was implemented using the Python programming language. The molten salt is assumed to be entirely in the liquid phase and the nature of the flow is turbulent. The specific heat capacity, density, thermal conductivity, and kinematic viscosity of the molten salt are assumed to only vary with temperature (See Table 1). The temperature, density, pressure, velocity, and volume flow rate are all defined at the centre of the control volume. The inlet and outlet molten salt temperatures are solved for each increment with the average of these values used to define the various fluid properties inside the control volume.

Conservation of mass is given by:

$$\rho_e \dot{V}_e - \rho_i \dot{V}_i = 0 \tag{3-1}$$

where ρ is density in kg/m^3, $\dot{V}$ is volume flow rate in m^3/s. Where the subscripts *i* and *e* represent the inlet and outlet nodes of the element.

Conservation of momentum is given by:

$$p_{0_e} - p_{0_i} = -\left(\frac{f \Delta x}{D} + \sum K\right)\frac{\rho_e |u| u}{2} \tag{3-2}$$

where p is pressure in Pa, $p_0 = p + \frac{1}{2}\rho_H V_h^2 + \rho_H g z$, f is the friction factor, Δx is the length of the increment in m, D is the hydraulic diameter of the increment, $\sum K$ is the sum of secondary loss factors, and u is the velocity of the fluid in m/s

Energy conservation is given by:

$$\rho_e \dot{V}_e h_e - \rho_i \dot{V}_i h_i = \dot{Q}_n \tag{3-3}$$

where h is enthalpy in J/kg and $\dot{Q}_n$ is heat transfer in W in element *n*.

The discretized 1-D control volumes (elements) are solved in a successive approximation algorithm where the outlet node is used as the inlet node for the next element.

3.3 Half tube model development

The overall model consists of a network of representative tube increments connected in series and parallel as required to emulate the actual HTF pipe layout around the solar receiver. Although variations in heat flux along the height and around the circumference of the solar receiver is accounted for, the heat flux is assumed to be uniform along the length and across the width of any single tube increment. Also, the heat conduction along the length of the tube increment is assumed to be negligible compared to the radial and circumferential conduction. This means that each

representative tube increment is effectively reduced to a two-dimensional (2D) geometry, and due to symmetry only half of the tube needs to be explicitly modelled. To determine the maximum tube metal temperature, each of these representative tube increments is discretized in the radial and circumferential directions.

A key question in the development of the thermal network for a single tube increment is what the required level of discretization should be. The need to predict the maximum tube metal temperature with sufficient accuracy implies that the temperature distribution around the circumference on the outer surface of each tube must be determined. This could require a large number of increments in both the radial and circumferential directions. However, to obtain reasonable computational times, the model employs a rather coarse discretization, but one that still captures all the relevant thermofluid phenomena, which is calibrated using results obtained from a more detailed CFD model.

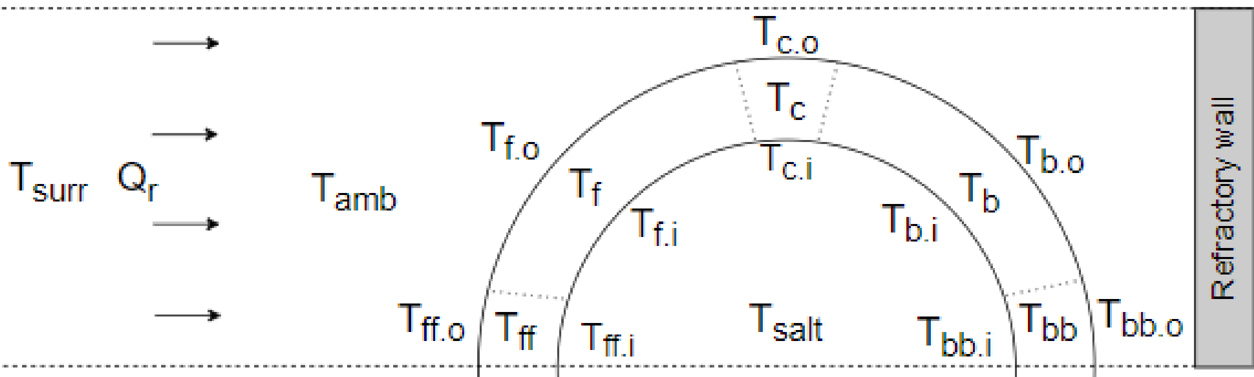


Figure 9: Discretized half-tube geometry

Figure 9 illustrates the proposed representative half tube geometry. The half tube is discretised into five sectors. These are: ff – the most front facing sector of the front wall, f – the rest of the front wall, c – the central sector facing perpendicular to the impinging radiation, bb – the most back facing sector of the back wall, and b – the rest of the back wall. The refractory wall behind the tubes is assumed to be a perfectly insulated and therefore also reflective surface. The subscripts *I* and *o* represent the inner and outer surfaces of the tube respectively.

The corresponding equivalent thermal resistance network is shown in Figure 10. Note that each discretised sector is represented by various thermal resistances accounting for radiative heat transfer (rad), convective heat transfer (cv), and conductive heat transfer (cd) respectively.

The net radiation into the outer nodes in W is given by:

$$\dot{Q}_{rad.in.net} = \alpha_{ab} F_{view} q_{flux} A_{flux} \tag{3-4}$$

where α_{ab} is the absorptivity is the solar absorbent coating on the tube, F_{view} is the view factor from the projected area of the tube to the tube increment, Q_{flux} is the heat flux impinging upon the increment in W/m^2, and A_{flux} is the projected area of the tube in m^2, assumed to be the product of the height of the increment and the pitch of the tube.

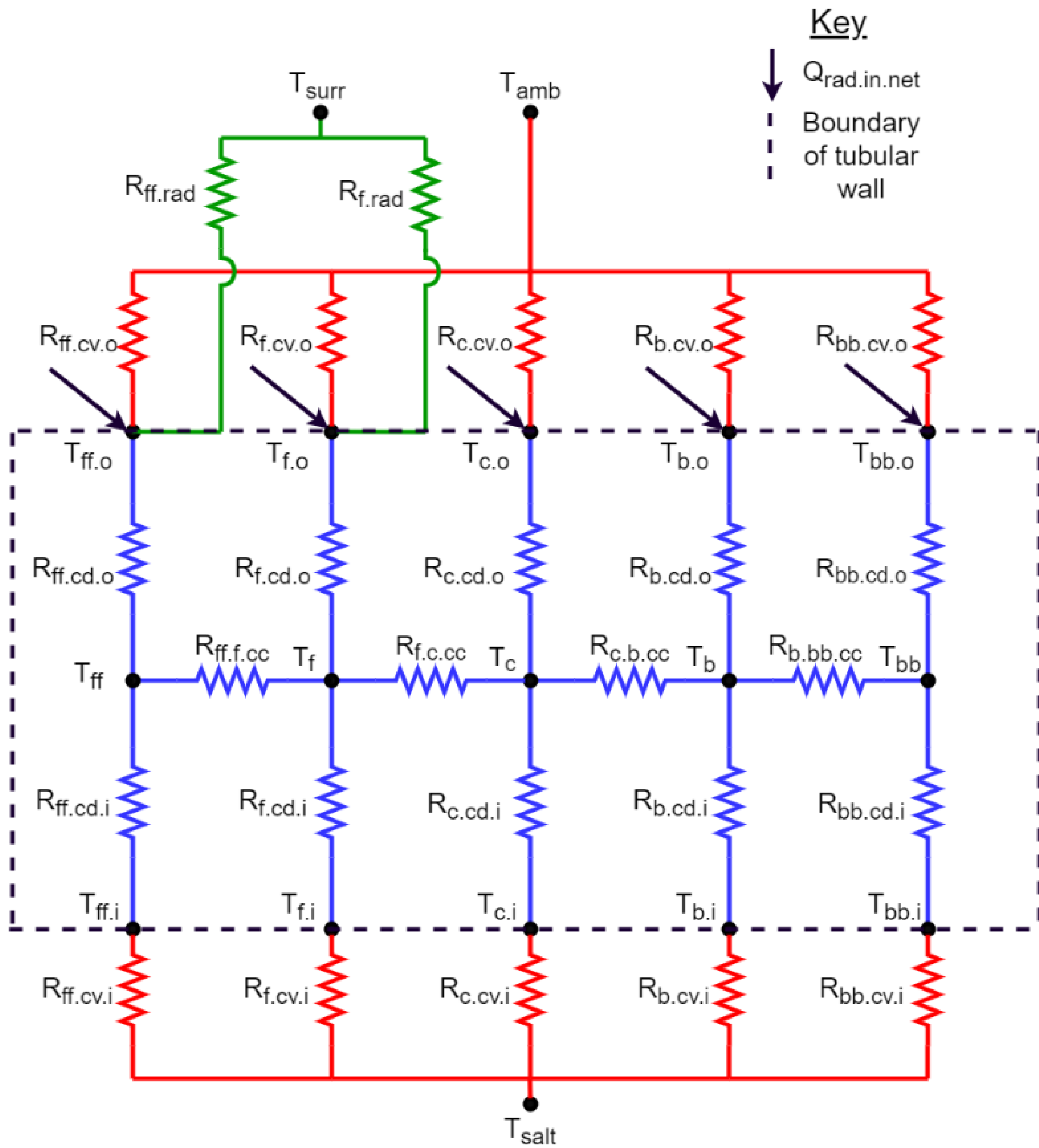


Figure 10: Equivalent thermal resistance diagram

The radiative thermal resistance in K/W is given by:

$$R_{rad} = \frac{1}{h_{rad} A_{s.o}} \tag{3-5}$$

where $A_{s.o}$ is the outer surface area in m^2 as shown in Figure 11, and h_{rad} is the effective radiative heat transfer coefficient in W/m^{2}K which is given by:

$$h_{rad} = F_{view} e_t \sigma \left(T_{t_o}^2 + T_{surr}^2\right)\left(T_{t_o} + T_{surr}\right) \quad (3\text{-}6)$$

F_{view} is the view factor, e_t the emissivity of the tube and T_{t_o} is the outer temperature of the tube. T_{surr} is the bulk temperature of the surrounding atmosphere and space which absorbs the radiative losses.

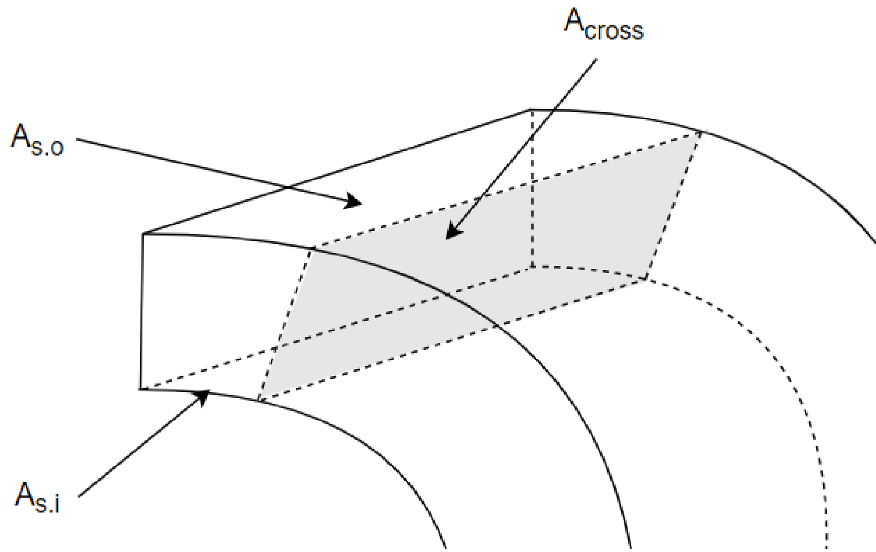


Figure 11: Schematic of discretised areas

The convective thermal resistance between the outer surface of the tube and the ambient air, T_{amb} is given by:

$$R_{cv.o} = \frac{1}{h_{cv.0} A_{s.o}} \quad (3\text{-}7)$$

where $h_{cv.o}$ is the combined heat transfer coefficient for natural and forced convection on the outside of the tube given by:

$$h_{cv} = \left(h_{nc}^{3.2} + h_{fc}^{3.2}\right)^{1/3.2} \quad (3\text{-}8)$$

where h_{nc} is the heat transfer coefficient for natural convection which is a function of the Rayleigh (*Ra*), Prandtl (*Pr*) and Nusselt (Nu_{nc}) numbers for a vertical cylinder as per (Çengel, Cimbala & Turner, 2012) and are found as follows:

$$h_{nc} = \frac{k Nu_{nc}}{D_o} \quad (3\text{-}9)$$

$$Nu_{nc} = \left\{ 0.825 + \frac{\left(0.387 Ra^{\frac{1}{6}}\right)}{\left(1 + \left(\frac{0.492}{Pr}\right)^{\frac{9}{16}}\right)^{\frac{8}{27}}} \right\}^2 \quad (3\text{-}10)$$

$$Ra = \frac{g\beta(T_{t.o} - T_{amb})D_o^3}{\alpha_{nc} v_c} \quad (3\text{-}11)$$

where k is thermal conductivity in W/mK, g is acceleration due to gravity in m/s, β is the thermal expansion coefficient of air in K, D_o is the outer diameter of the tube, α_{nc} is thermal diffusivity of air, and v_c is the kinematic viscosity of air in m^2/s.

h_{fc} is the heat transfer coefficient due to forced convection (wind) which is also a function of the Reynolds (Re_{fc}), Prandtl (*Pr*) and Nusselt (Nu_{fc}) numbers for a vertical cylinder as per (Çengel et al., 2012) and are calculated as follows:

$$h_{fc} = \frac{kNu_{fc}}{D_o} \quad (3\text{-}12)$$

$$Nu_{fc} = 0.3 + \frac{0.62 Re_d^{\frac{1}{2}} Pr^{\frac{1}{3}}}{\left(1 + \left(\frac{0.4}{Pr}\right)^{\frac{2}{3}}\right)^{\frac{1}{4}}} \quad (3\text{-}13)$$

$$Re_{fc} = \frac{u_{air} D_o}{v_c} \quad (3\text{-}14)$$

where u_{air} is the wind speed in m/s.

While the three front facing sections (*ff*, *f*, and *c*) of the tube take both natural and forced convection into account as per Equation (3-8), the two back facing segments (*b* and *bb*) assume forced convection to be negligible as they are not in direct contact with the wind.

The conductive thermal resistance between the outer surface of the tube and the midpoint of the tube wall is given by:

$$R_{cd.o} = \frac{L_{c.o}}{k_t\, A_{s.mean}} \quad (3\text{-}15)$$

where $L_{c.o}$ is the length between the outer and middle nodes, as shown in Figure 12. $A_{s.mean}$ is the mean surface area of the inner and outer surfaces $A_{s.i}$ and $A_{s.o}$ shown in Figure 11 and k_t is the thermal conductivity of the tube material.

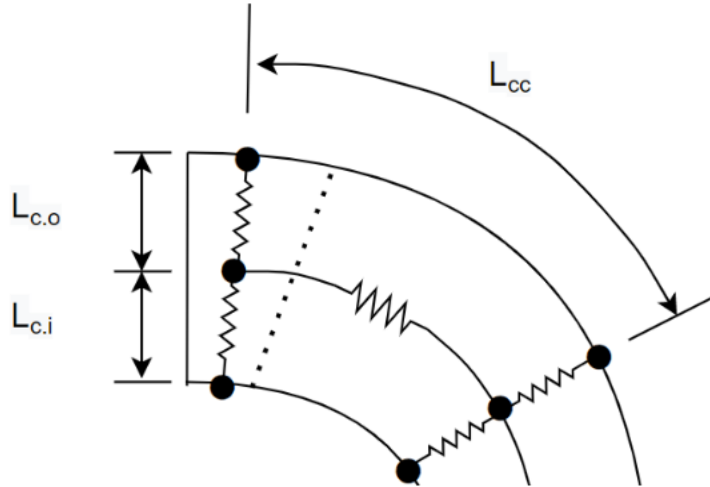


Figure 12: Schematic of thermal resistance lengths

The convective thermal resistance between the outer surface of the tube and the ambient air, T_{amb} is given by:

$$R_{cd.i} = \frac{L_{c.i}}{k_t A_{s.mean}} \quad \text{(3-16)}$$

where $L_{c.i}$ is the length between the inner and middle node as shown in Figure 12.

The conductive thermal resistance in the circumferential direction between the nodes situated at the midpoint of the tube wall is given by:

$$R_{cc} = \frac{L_{cc}}{k_t . A_{cross}} \quad \text{(3-17)}$$

where L_{cc} is the circumferential distance between nodes as shown in Figure 12. A_{cross} is the cross-sectional area as shown in Figure 11.

The convective thermal resistance between the inner surface of the tube and the average salt temperature is given by:

$$R_{cv.i} = \frac{1}{h_{cv.i} . A_{s.i}} \quad \text{(3-18)}$$

where $A_{s.i}$ is the inner surface area as shown in Figure 11. $h_{cv.i}$ is the inner convective heat transfer coefficient given by:

$$h_{cv.i} = \frac{Nu.\,k_{salt}}{D_i/2} \quad \textit{(3-19)}$$

where D_i is the inner diameter of the tube and Nu is the Nusselt number found using the Dittus Boelter correlation since the Reynolds number is found to be greater than 10 000 in all cases (Çengel et al., 2012) given by:

$$\mathrm{Nu} = 0.023\mathrm{Re}_{\mathrm{D}}^{0.8}\,\mathrm{Pr}_{\mathrm{salt}}^{0.4} \quad \textit{(3-20)}$$

3.4 Solution method

The equivalent thermal resistance network for a single tube increment consists of 15 unknown temperature nodes and therefore requires the simultaneous solution of 15 independent equations. The network is analogous to an electrical resistance network by treating temperatures as voltages, thermal resistances as resistances, and heat transfer as current. Kirchhoff's Current Law (KCL) is therefore employed and the net current into each node should add up to zero.

For temperature nodes $T_{ff.o}$, $T_{f.o}$, $T_{c.o}$, $T_{b.o}$, $T_{bb.o}$, T_{ff}, T_f, T_c, T_b, T_{bb}, $T_{ff.i}$, $T_{f.i}$, $T_{c.i}$, $T_{b.i}$, and $T_{bb.i}$ respectively when KCL is applied the sum of the heat transfer into the nodes are given by Equations (3-21) to (5-1).

$$Q_{rad.net.ff} - \frac{T_{ff.o} - T_{ff}}{R_{ff.cd.o}} - \frac{T_{ff.o} - T_{surr}}{R_{ff.rad}} - \frac{T_{ff.o} - T_{amb}}{R_{ff.cv.o}} = 0 \quad \textit{(3-21)}$$

$$Q_{rad.net.f} - \frac{T_{f.o} - T_f}{R_{f.cd.o}} - \frac{T_{f.o} - T_{surr}}{R_{f.rad}} - \frac{T_{f.o} - T_{amb}}{R_{f.cv.o}} = 0 \quad \textit{(3-22)}$$

$$Q_{rad.net.c} - \frac{T_{c.o} - T_c}{R_{c.cd.o}} - \frac{T_{c.o} - T_{amb}}{R_{c.cv.o}} = 0 \quad \textit{(3-23)}$$

$$Q_{rad.net.b} - \frac{T_{b.o} - T_b}{R_{b.cd.o}} - \frac{T_{b.o} - T_{amb}}{R_{b.cv.o}} = 0 \quad \textit{(3-24)}$$

$$Q_{rad.net.bb} - \frac{T_{bb.o} - T_{bb}}{R_{bb.cd.o}} - \frac{T_{bb.o} - T_{amb}}{R_{bbcv.o}} = 0 \quad \textit{(3-25)}$$

$$\frac{T_{ff}-T_{ff.o}}{R_{ff.cd.o}}+\frac{T_{ff}-T_{ff.i}}{R_{ff.cd.i}}+\frac{T_{ff}-T_{f}}{R_{ff.f.cc}}=0 \quad (3\text{-}26)$$

$$\frac{T_{f}-T_{f.o}}{R_{f.cd.o}}+\frac{T_{f}-T_{f.i}}{R_{f.cd.i}}+\frac{T_{f}-T_{ff}}{R_{ff.f.cc}}+\frac{T_{f}-T_{c}}{R_{f.c.cc}}=0 \quad (3\text{-}27)$$

$$\frac{T_{c}-T_{c.o}}{R_{c.cd.o}}+\frac{T_{c}-T_{c.i}}{R_{c.cd.i}}+\frac{T_{c}-T_{f}}{R_{f.c.cc}}+\frac{T_{c}-T_{b}}{R_{c.b.cc}}=0 \quad (3\text{-}28)$$

$$\frac{T_{b}-T_{b.o}}{R_{b.cd.o}}+\frac{T_{b}-T_{b.i}}{R_{b.cd.i}}+\frac{T_{b}-T_{c}}{R_{c.b.cc}}+\frac{T_{b}-T_{bb}}{R_{b.bb.cc}}=0 \quad (3\text{-}29)$$

$$\frac{T_{bb}-T_{bb.o}}{R_{bb.cd.o}}+\frac{T_{bb}-T_{bb.i}}{R_{bb.cd.i}}+\frac{T_{bb}-T_{b}}{R_{b.bb.cc}}=0 \quad (3\text{-}30)$$

$$\frac{T_{ff.i}-T_{salt}}{R_{ff.cv.i}}+\frac{T_{ff.i}-T_{ff}}{R_{ff.cd.i}}=0 \quad (3\text{-}31)$$

$$\frac{T_{f.i}-T_{salt}}{R_{f.cv.i}}+\frac{T_{f.i}-T_{f}}{R_{f.cd.i}}=0 \quad (3\text{-}32)$$

$$\frac{T_{c.i}-T_{salt}}{R_{c.cv.i}}+\frac{T_{c.i}-T_{c}}{R_{c.cd.i}}=0 \quad (3\text{-}33)$$

$$\frac{T_{b.i}-T_{salt}}{R_{b.cv.i}}+\frac{T_{b.i}-T_{b}}{R_{b.cd.i}}=0 \quad (3\text{-}34)$$

$$\frac{T_{bb.i}-T_{salt}}{R_{bb.cv.i}}+\frac{T_{bb.i}-T_{bb}}{R_{bb.cd.i}}=0 \quad (3\text{-}35)$$

These 15 equations are solved simultaneously using Powell’s dog leg method (Powell, 1970) within Python’s Scipy.Optimize library using the fsolve function. Once convergence has been reached the heat transfer absorbed by the molten salt, Q_{salt}, is found using the sum of the heat transfer between the inner nodes of the tube and the average temperature of the molten salt as follows:

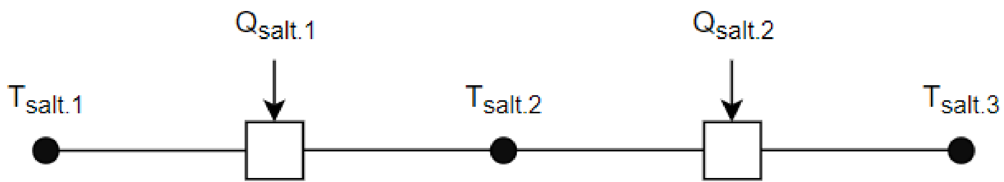


Figure 13: Nodal heat transfer solving procedure

$$Q_{salt} = \frac{T_{ff.i} - T_{salt}}{R_{ff.co.i}} + \frac{T_{f.i} - T_{salt}}{R_{f.co.i}} + \frac{T_{c.i} - T_{salt}}{R_{c.co.i}} + \frac{T_{b.i} - T_{salt}}{R_{b.co.i}} + \frac{T_{bb.i} - T_{salt}}{R_{bb.co.i}} \quad (3\text{-}36)$$

The nodal solving procedure to calculate the outlet salt temperature can be seen in Figure 13. With the salt temperature at the inlet known, the salt temperature at outlet of the discretised tube section is then calculated using an energy balance as follows:

$$T_{salt.i+1} = T_{salt.i} + \frac{Q_{salt.i}}{\dot{m}_{salt}\, Cp_{salt}} \quad (3\text{-}37)$$

where $\dot{m}_{salt}$ is the mass flow rate of the molten salt in kg/s and Cp_{salt} is the specific heat of the molten salt in J/kgK.

Similarly, the total heat losses due to radiation to the surrounding environment are found by summing the heat transfers between the outer tube surface nodes and the surrounding temperature as follows:

$$Q_{rad.loss} = \frac{T_{ff.i} - T_{surr}}{R_{ff.rad}} + \frac{T_{f.i} - T_{surr}}{R_{f.rad}} \quad (3\text{-}38)$$

The total heat lost due to convection is found by summing the heat transfers between the outer tube nodes and ambient air surrounding the receiver as follows:

$$Q_{cv.loss} = \frac{T_{ff.i} - T_{amb}}{R_{ff.cv.o}} + \frac{T_{f.i} - T_{amb}}{R_{f.cv.o}} + \frac{T_{c.i} - T_{amb}}{R_{c.cv.o}} + \frac{T_{b.i} - T_{amb}}{R_{b.cv.o}} + \frac{T_{bb.i} - T_{amb}}{R_{bb.cv.o}} \quad (3\text{-}39)$$

The solving procedure follows the algorithm outlined in Figure 14. This first begins with an initial estimation of the 15 unknown tubular nodal temperatures, as well as the bulk salt temperature (T_{salt}). Equations (3-21) through (3-34) are then solved, finding the 15 unknown tubular temperatures. From here the heat transfer into the molten salt as well as the losses due to radiative and convective heat transfer are found using equations (3-37), (5-1), and (3-39). The outlet salt temperature is then found using equation (3-37), the bulk salt temperature is then found by averaging the inlet and outlet salt temperature for the increment. The bulk salt temperatures are then compared for the two previous consecutive iterations. The solution is deemed to have reached convergence when the error in salt temperature between the two consecutive iterations, TOL, is less than 10^{-6}. The algorithm then moves to the next discretized element with the model using the

outlet conditions of the previous discretized section as the inlet conditions for next discretized element.

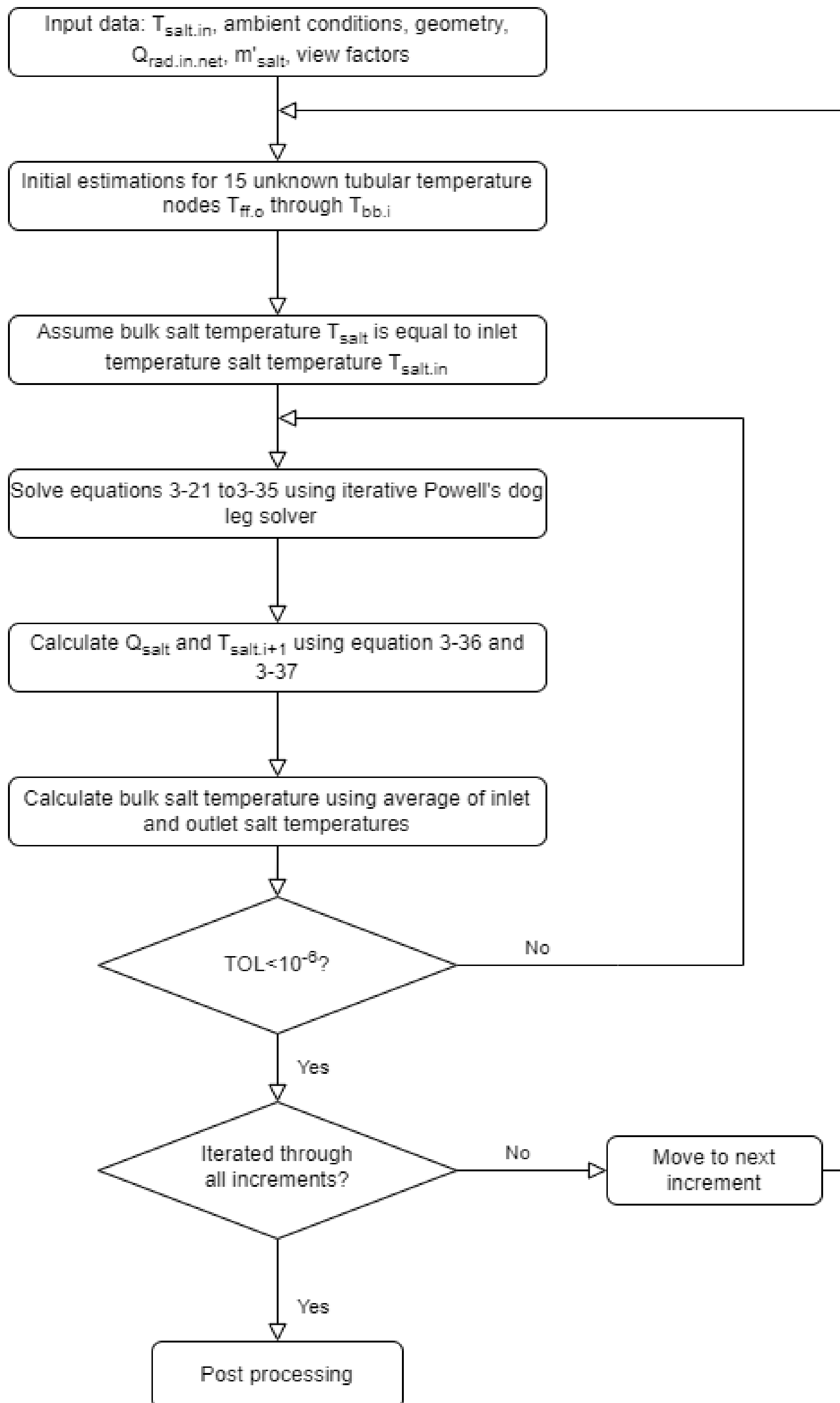


Figure 14: Solar receiver flow network solving algorithm

3.5 Model calibration

Due to the lack of real-plant data, the calculated circumferential temperature distribution around the tube cannot reasonably be verified using empirical data. Therefore, a CFD model was developed in Ansys Fluent and used to calibrate the network model. The calibration essentially entails the determination of an appropriate division of the front tube wall between the sectors ff , f and c, with the sectors b and bb linked to it to obtain an axially symmetric layout.

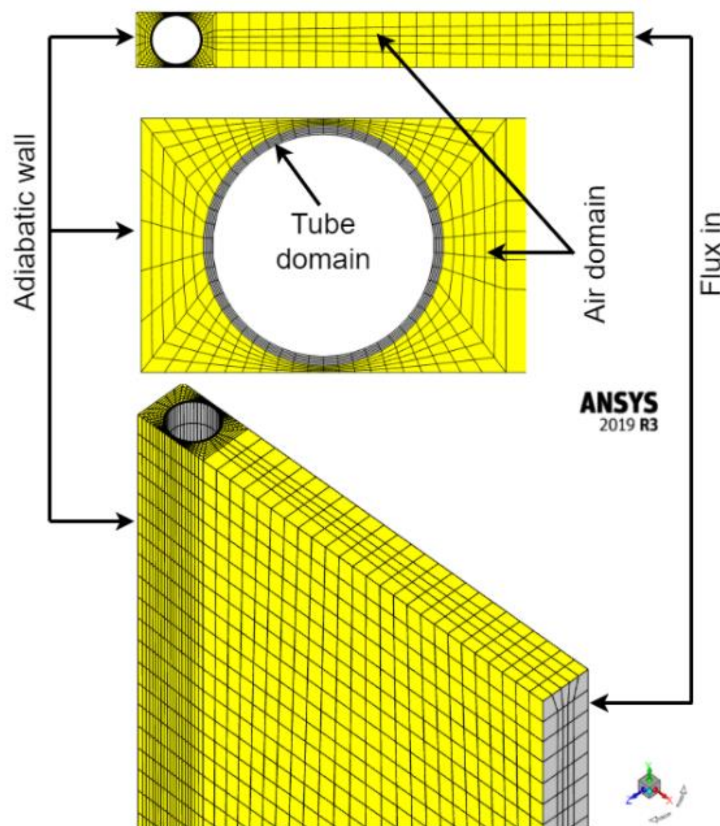


Figure 15: CFD computational domain

The computational domain for the CFD model is shown in Figure 15. Although the representative tube geometry is effectively reduced to a 2D problem, it was convenient to model a one-meter length of tube in 3D. A tube with an inner diameter of 38.9 mm, an outer diameter of 42.2 mm and a tubular pitch of 44.5 mm was used in the computational domain. The model contains an air domain and tube domain as shown. The top, bottom and sides of the air domain are symmetry boundaries, and the impinging radiation is specified as a wall flux boundary condition. The refractory wall is an adiabatic wall as it is assumed to be an adiabatic surface.

The salt flow inside the tube is not modelled explicitly. Instead, a wall boundary condition with a specified free stream temperature and convective heat transfer coefficient is specified. The appropriate value for the convective heat transfer coefficient was determined using the Dittus Boelter correlation (Çengel et al., 2012).

The CFD model employed the Surface-to-Surface radiation model where the view factors are pre-calculated from the geometry. The specification of a heat flux boundary effectively results in a given temperature at the flux inlet surface, which in turn results in thermal radiation being projected from the wall and impinging onto the tube surface.

Table 2: Fluent calibration

Salt temperature (°C)	**Flux (kW/m²)**	**CFD max. tube temperature (°C)**	**Network model max. tube temperature (°C)**	**Absolute error %**
499.71	196.42	539.00	538.27	0.1354
413.98	832.68	591.00	590.84	0.0271
467.35	150.89	498.00	497.84	0.0321
530.19	469.77	620.00	620.06	0.0097
492.81	353.43	563.00	562.62	0.0675
449.38	590.28	571.00	570.65	0.0613
432.63	389.23	514.00	513.86	0.0272
543.87	944.71	722.00	722.64	0.0886
464.67	724.50	612.00	611.43	0.0931
556.65	812.42	709.00	708.88	0.0169

For calibration the average salt temperature and radiation heat flux were used as independent variables. A data set containing 10 samples were generated to cover the two-dimensional input variable space using Latin hypercube sampling (LHS) available in the PyDOE design of experiments library. The range for the salt temperature was set to be between 400 and 565°C while the range for the impinging flux was set to be between 100 and 1000 kW/m^2. The CFD model was then used to simulate each of the 10 cases and the calculated maximum tube temperatures were recorded for each case.

A range of values were determined for the division of the tube wall amongst the different sectors as described above. The network model was then employed to determine the maximum tube metal temperature for each of the 10 cases. The tube wall division was then varied according to a simple exhaustive search optimisation routine to find the division that results in the least sum squared error value between the maximum temperature results obtained with the CFD and network models.

Table 3: View factors

Discretised Section	*ff*	*f*	*c*	*b*	*bb*
View factor	0.127	0.840	0.006	0.023	0.004

The minimum sum squared error was obtained with the ff sector spanning 2.13% of the tube circumference. This implies, using the crossed strings method, that the view factor from the flux

inlet surface to the ff sector is 0.127. The calculation for the view factor can be found in Appendix A. Once calibrated, the network model predicted the maximum metal temperatures in the receiver to within 0.2% of the Fluent model. The results of the calibration are provided in Table 3 showing the values obtained for the view factor in each section.

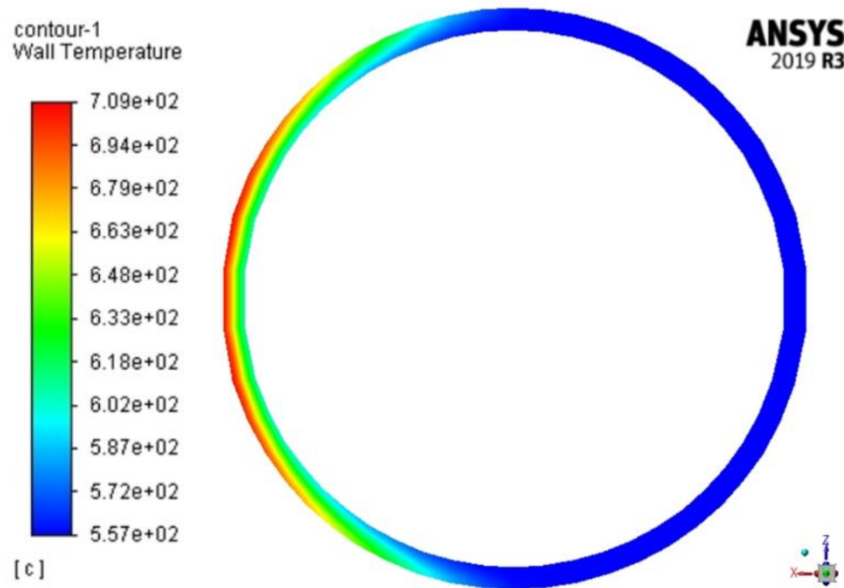


Figure 16: Ansys Fluent thermal contour plot

The temperature contours obtained with the calibrated model for the last case in Table 2 are presented in Figure 16. This shows how large the variation in temperature is, with the maximum temperature in the tube being 709°C while the minimum temperature is 557°C. This is clearly driven by heat flux.

3.6 Grid independence study

The half tube model was developed and the increments of the tubes calibrated using Ansys Fluent such that the maximum tubular temperature of each vertical increment can be solved. From here the increments can be used to build a flow network for a full receiver model. This model will need to have a grid independent solution.

The half tube model was developed and the increments of the tubes calibrated using Ansys Fluent such that the maximum tubular temperature of each vertical increment can be solved. From here the increments can be used to build a flow network for a full receiver model. This model will need to have a grid independent solution

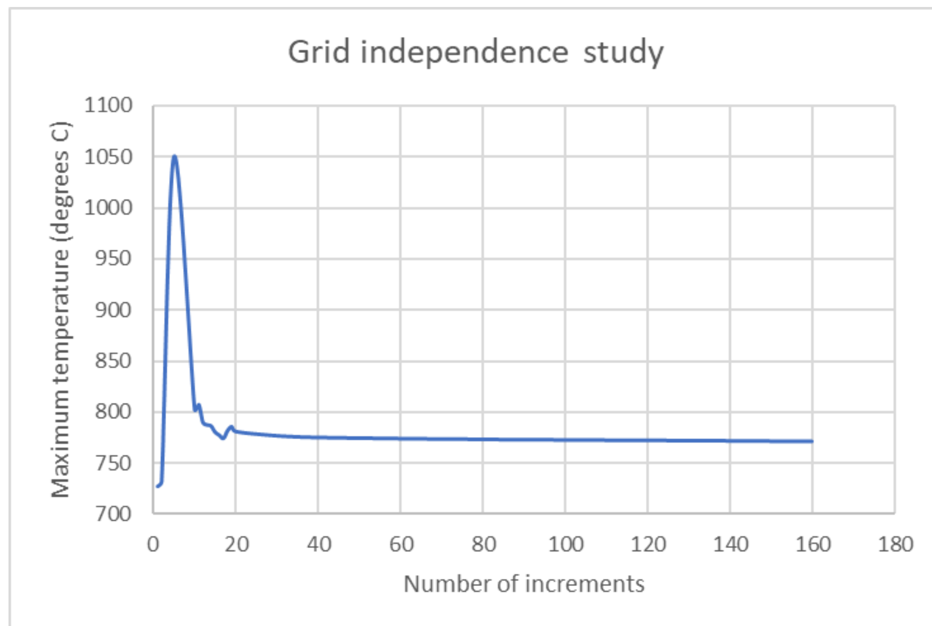


Figure 17: Grid independence study

A model receiver was built to match the specification of the Solar Two plant run by the DoE located east of Barstow, CA in 1990s (Pacheco, Bradshaw, et al., 2002). The full specification of the plant can be found in Chapter 3.7. The flux profile impinging upon the receiver as well as the heliostat field layout were recreated using SolarPILOT (Wagner & Wendelin, 2018). A single tube from each panel in the flow network was used to represent each panel. The number of vertical increments were then varied. The maximum and minimum tubular temperature were then recorded for each simulation.

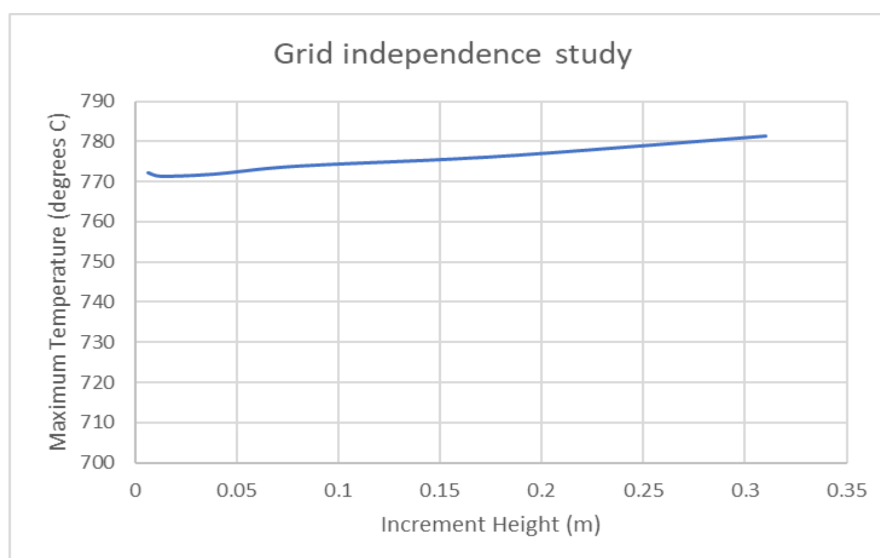


Figure 18: Grid independence study increment height vs temperature

Table 4: Grid independence study

Number of Increments	**Increment Height (m)**	**Maximum Temperature (degrees C)**	**Minimum Temperature (degrees C)**	**Expected Error**
1000	0.006	772.193	290.572	0.221%
500	0.012	771.304	290.572	0.036%
160	0.039	771.844	290.578	0.148%
80	0.077	773.779	290.590	0.548%
40	0.155	775.624	290.599	0.927%
30	0.207	777.296	290.638	1.267%
20	0.310	781.421	290.689	2.097%

To find a grid independent solution the tubular temperatures at an infinite number of discretised increments, or an increment height of zero, of discretised increments needs to be estimated. This is referred to as the expected value. The expected value is found by finding the y-intercept of the tubular temperature by performing a regression on the tubular temperature and increment height from where the relationship begins to become linear. It can be seen in Figure 17 that the relationship becomes linear from 20 increments onwards. The expected error is found as follows:

$$Er = \frac{T_{max.i} - T_{max,exp}}{T_{max.i} - T_{min,exp}} \tag{3-40}$$

where $T_{max.i}$ is the maximum tubular temperature for the given number of increments, $T_{max.exp}$ and $T_{min.exp}$ are the expected value for the maximum and minimum tubular temperature found using linear regression respectively. It can be seen in Figure 17 that the maximum temperature plateaus after 40 increments are reached. This is confirmed in Table 4 as the expected error is below 1% when 40 increments are reached. This means the flow network can be assumed to be grid independent at 40 vertical increments per panel.

3.7 Validation

The lack of accessible detail plant data for CSP tower plants means that only the heat transfer into the solar salt could be validated. This was done using plant data from the Solar Two plant. The flux profiles impinging upon the receiver as well as the heliostat field layout were reproduced using SolarPILOT.

The solar receiver geometry and plant location data were used as inputs to the model. Solar Two uses an external cylindrical solar receiver with a similar architecture to the architecture shown in Figure 6. However, as the plant is located in the northern hemisphere, the inlet is situated at the north side of the receiver and the outlet at the south end of the receiver. The solar receiver also

consists of 24 panels rather than 20. The detailed geometry and specifications of the solar receiver can be seen in Table 5

Table 5: Solar Two receiver geometry

Heat transfer fluid	Solar Salt
Tube material	316H Stainless Steel
Tube absorptivity	0.95 (Black pyromark coating)
Receiver height/diameter	6.2/5.1 m
Number of flow circuits	2
Number of panels	24
Number of tubes per panel	32
Tube outer/inner diameter	21/18.8 mm

Solar Two used two different heliostat sizes for the field as old heliostats from the original Solar One test plant were repurposed for Solar Two. Only 5% of the heliostats in the field were repurposed from Solar One, but the exact layout of these heliostats is not publicly available data. Therefore, the field was approximated to the closest possible layout of the Solar Two plant using a single heliostat size.

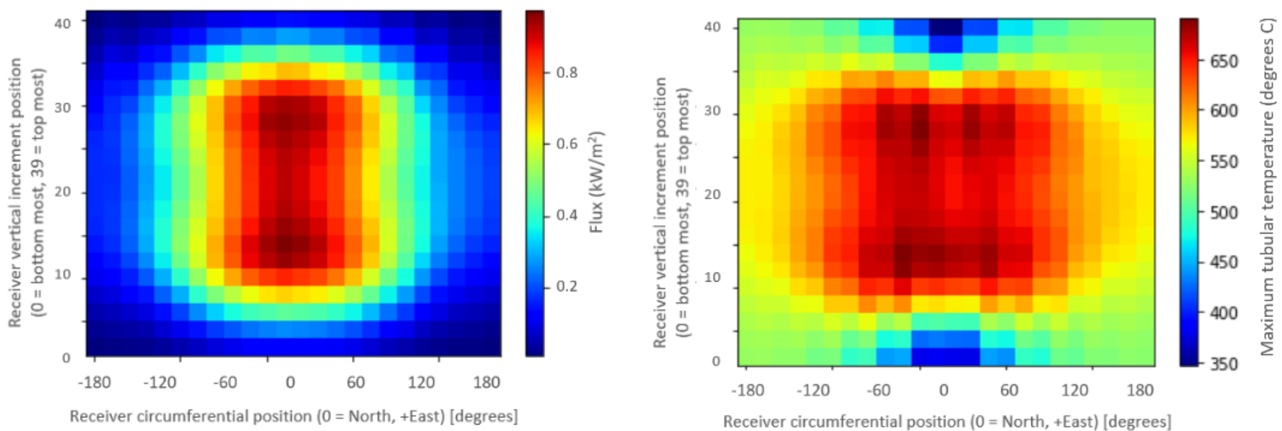


Figure 19: Validation flux and maximum tubular temperature heat maps

The number of heliostats in service, direct normal irradiation (DNI), wind speed, ambient temperature, inlet temperature and mass flow rate of the Solar Salt HTF of the plant were used as input data for the equivalent thermal resistance model. Several days including a variety of wind conditions and ambient temperatures were used from the plant data to give a robust dataset. The model results for the heat absorption rate are within 5% for all cases. It can be seen in Table 6 that the heat absorption rate is both over and under-estimated by the model, meaning that the model is biased towards neither over nor underestimation. The hours with higher wind speeds both underestimate the heat transfer into the molten salt. This is due to the model predicting high forced

convection losses. This could potentially be since the wind speed is assumed to be equal for all panels while in reality the wind speed would vary around the circumference of the external cylindrical solar receiver. However, further studies into the effects of wind are outside the scope of this project.

Figure 19 depicts the relationship between the flux impinging upon the different increments and the resulting maximum tubular temperature (the *ff* sector of each increment) for the validation case on the 29th of September 1997 at midday. One can see how closely the map of the maximum temperature correlates with that of the flux map. This makes sense as higher fluxes will inherently lead to higher tubular temperatures. The Python script used to assess 29th of September 1997 at midday can be found in Appendix B.

Table 6: Validation results

Solar Two Test Date	29/09/97 11:00	29/09/97 12:00	30/09/97 11:00	30/09/97 12:00	12/03/99 11:00	12/03/99 12:00	23/03/99 11:00	23/03/99 12:00
Plant heliostats	1767	1767	1764	1758	1685	1684	1626	1625
Model heliostats	1767	1767	1767	1767	1683	1683	1627	1627
DNI (W/m^2)	887	931	975	976	865	915	858	875
Wind speed (m/s)	0.6	0.6	1	0.4	2.0	2.2	9.0	6.9
Ambient temperature (C)	32	32	33	33	14	14	16	16
Inlet temperature (C)	294	296	300	301	302	303	300	302
Plant outlet temperature (C)	555	553	552	554	564	564	563	564
Model outlet temperature (C)	555.539	565.213	558.89	560.673	575.454	573.704	558.786	556.405
Mass flow rate (kg/s)	80	85	90	91	67	73	61	65
Plant heat absorption (MW)	31.655	33.118	34.113	34.916	26.638	28.915	24.341	25.843
Model heat absorption (MW)	30.995	31.979	35.344	35.851	27.821	30.006	23.946	25.083
Heat absorption error	**-2.086%**	**-3.440%**	**3.609%**	**2.678%**	**4.441%**	**3.773%**	**-1.623%**	**-2.941%**

3.8 Summary

A thermofluid network model was developed and calibrated using CFD. Following this, a grid independence study was performed and the final calibrated grid independent model was validated using plant data. The calibration was done by finding the most suitable size of the front most facing sector (*ff*) so that the maximum tubular temperature is captured as accurately as possible. The calibration resulted in the predicted maximum tubular temperature being within 0.2% of the results obtained with the CFD model. The validation using plant data was completed with limited access to plant data. This meant that only the heat transfer into the molten salt could be validated. This validation is deemed successful since the results of the network model are within 5% of the plant data, despite the broad range of uncertainties stemming from the atmospheric and operational conditions.

4. Data-driven surrogate modelling

Referring to Figure 6, each panel consist of several identical tubes in parallel. In the thermofluid model, these tubes are characterised by a single representative tube section which is discretised vertically, as shown in Figure 7. Based on the grid independence study presented in Section 3.6, there are 40 vertical increments for each representative tube section.

The impinging flux needs to be calculated for each of the heat transfer network increments, which can be generalised to a 20 by 40 grid for the geometry of receiver to be studied. This calculation, while easily implemented in software such as SolarPILOT (Wagner & Wendelin, 2018), is computationally expensive since the central tower is surrounded by thousands of individual heliostats, which are all sources of radiative flux. This computational expense is compounded when doing large simulations such as Monte Carlo analyses.

To reduce this computational expense a surrogate model can be employed. While data-driven surrogate modelling is a relatively new research field, its use to predict thermofluid phenomena has grown in recent years. While surrogate models are typically developed using experimental data, the acquisition of experimental data is often prohibitively expensive. Therefore, it can also be based on results generated with fundamental models, such as computational fluid dynamics (CFD), or thermofluid network/process models. Using a process model to generate training data has the advantage of being less computationally expensive than using CFD models. While there have been many studies which have looked at modelling thermofluid phenomena in power plants, including coal fired boilers, air-cooled condensers, and photovoltaic solar plants, very few studies have used machine learning techniques to model CSP plants.

To reduce the computational expense of the process, a DDSM was developed using a multilayer perceptron neural network that predicts the flux profile impinging on the receiver for a range of plant configurations and atmospheric conditions at a specific location. It accounts for different power requirements, central tower heights and receiver geometries, different azimuthal and elevation angles of the sun, as well as DNI. The model was trained on data for Upington in Southern Africa produced with SolarPILOT combined with the CoPylot Python API (Hamilton, Wagner & Zolan, 2021). This DDSM will be referred to as the FSM – flux surrogate model in the rest of the report.

The detailed discretised thermofluid network model of the solar receiver is not as computationally intensive as predicting the heat flux profile. However, it was found that when combined with the FSM, it formed a bottleneck in the analysis process. This bottleneck occurs specifically because when simulating the receiver only the inlet and target outlet temperatures of the molten salt are known. The required molten salt flow rate is not known since in reality it would be controlled by the plant

to provide the targeted outlet temperature depending on the heat flux profile. Therefore, a control loop is needed to calculate the mass flow rate required to achieve the target outlet temperature. This control loop uses a Newton-Raphson algorithm to iterate the mass flow rate until it converges on the target outlet temperature of the receiver. The control loop generally needs to simulate the full thermofluid model 10 times until it converges on a solution. This means that while the thermofluid model takes approximately 5 seconds to run, the control loop takes approximately 50 seconds to run. It was therefore decided to also create a fast data driven surrogate model with the aid of training data generated with the thermofluid network model. This will be referred to as the TSM - thermofluid surrogate model in the remainder of the report.

This chapter will provide details of the techniques used for the data driven surrogate modelling. This will be broken into two main sections. The first section is data generation, where the methods used to create the training datasets will be discussed. The second section is machine learning modelling, where the theory behind MLPs as well as the specific techniques used to create the FSM and TSM surrogate models will be discussed.

4.1 Data generation

4.1.1 Flux surrogate model

The motivation behind the development of the FSM was not only to reduce computational expense, but also to be able to easily predict the flux map for a variety of plant sizes and solar receiver geometries without needing to interface with software such as SolarPILOT. Therefore, a dataset to train the model was created such that a range of solar receiver geometries and plant sizes could be simulated, as well as taking into account the local DNI and position of the sun at the plant location.

Table 7: FSM independent variables

Variable	Minimum value	Maximum value	Rationale
Solar field design power (MW_{th})	300	700	300 MW_{th} lowest realistic power for 50 MW_e plant with 12 hours TES, 700 MW_{th} highest for 100 MW_e plant
Tower height (m)	140	263	SolarPaces database on similar plants (NREL, 2022)
Maximum allowable flux (MW/m^2)	0.88	1	Current thermal material limits (Liao et al., 2014)
Height to diameter ratio	1.2	1.5	Generally accepted design parameter
DNI (W/m^2)	0	1200	1103W/m^2 maximum DNI in weather data
Elevation angle (°)	0	90	Geometry
Azimuthal angle (°)	0	360	Geometry

The independent variables for the dataset can be broken up into two categories, namely receiver geometrical parameters, and meteorological and climatological parameters. The receiver

geometrical parameters are the solar field design power output, tower height, maximum allowable flux, and height to diameter ratio for the SR. The meteorological and climatological parameters are DNI, elevation angle, and azimuthal angle. These seven independent variables translate to an input feature layer dimensionality of seven for the MLP.

To adequately cover the seven-dimensional input space when generating the training data, Latin hypercube sampling (LHS) was employed via the PyDOE design of experiments library. LHS is a sampling method developed by Mckay, Beckman & Conover, 2000. This sampling methodology selects *n* samples from *k* input parameters. Each input parameter has its range divided into *n* non-overlapping intervals with an equal probability per interval. This method ensures the widest range on data combinations is sampled with no repeated samples. To investigate the impact of training sample size, LHS was used to generate different input data sets with different sample sizes. It is important to note that for each sample size a unique LHS must be performed. In other words, one cannot simply take an existing smaller sample size and add additional data to obtain a larger sample size.

The ranges used for each variable are provided in Table 7. The plant was assumed to have 12 hours of thermal energy storage with the design point being the summer solstice. As indicated before, the SR has a fixed number of 20 panels with 40 vertical increments each. This would result in an output layer dimensionality of 800 for the MLP. The methods used to deal with this large difference in dimensionality of the input and output variables are described in detail in Section 4.2.1. The datasets were created using SolarPILOT's Python API called CoPylot (Hamilton et al., 2021). The Python script used to create the dataset can be found in Appendix C.

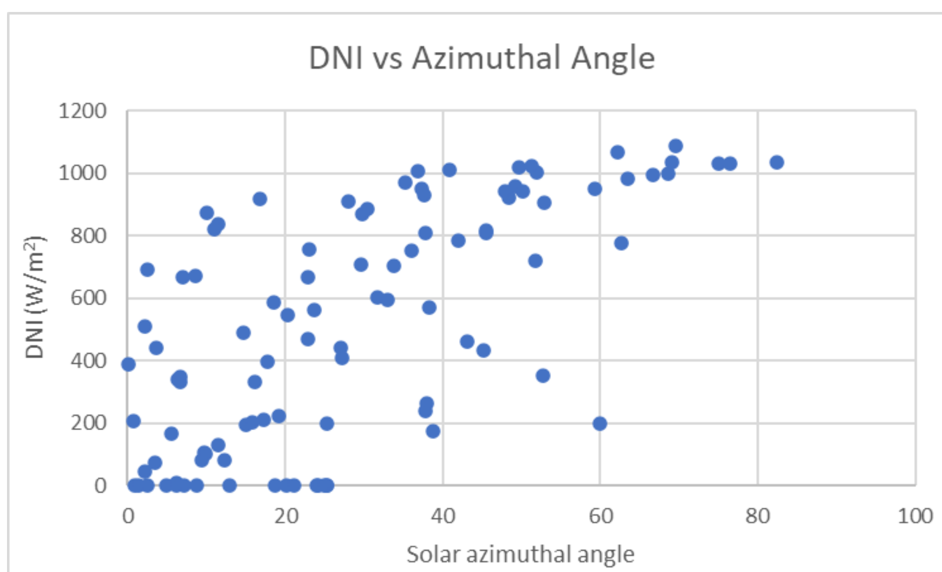


Figure 20: DNI vs azimuthal angle

The meteorological and climatological parameters (DNI, elevation angle and azimuthal angle) are not completely independent of one another. This is demonstrated by the data points shown in Figure 20, which shows a correlation between the DNI and the solar azimuth angle. Therefore, to reduce the time needed for training, the training data points must all contain combinations that represent realistic conditions found at the chosen location. To achieve this, the values of DNI, elevation angle and azimuthal angle were not allowed to vary independently within the LHS. Rather, the LHS used one discrete variable to sample unique hours from the climatic dataset which spanned 28 years. The corresponding values of DNI, elevation angle and azimuthal angle were retrieved from the weather data as the associated training variables.

The elevation and azimuthal angles are calculated from first principles using the latitude and longitude of the plant as well as the day, month and hour being simulated. This was completed using a method described in Geyer & Stine, 2001 which can be found in Appendix D.

4.1.2 Thermofluid surrogate model

As mentioned previously, the reason for creating a DDSM for the thermofluid network is because the computational expense of running the thermofluid model was compounded ten-fold by the overarching control loop needed to determine the required mass flow rate to obtain the specified outlet temperature. It was therefore decided to create a data driven surrogate model to represent a single vertical increment with the aid of training data generated with the thermofluid network model.

The independent variables for this surrogate model are the inlet molten salt temperature, impinging heat flux, wind velocity, ambient temperature, HTF mass flow rate and height of the receiver. The dependent variables are the maximum tubular temperature, minimum tubular temperature, heat absorption rate, and the change in HTF temperature over the discretised increment. The geometry of the tube was 42.2 mm outer diameter and 1.2 mm wall thickness. Two separate FSM models were developed, one using Solar Salt as the HTF and the other using FLiBe as the HTF. This was done to compare plant performance when different molten salt chemistries are used in the receiver. Two separate DDSMs needed to be developed as the two salt chemistries have different physical properties such as heat capacity and viscosity as well as having different operating temperatures.

A similar development methodology to that of the heat flux surrogate model was employed, with the PyDOE library used to create a six-dimensional input variable space using LHS. The ranges for LHS sampling for the TSM using Solar Salt and FLiBe can be found in Table 8 and Table 9 respectively. The Python script used to develop the TSM training sets can be found in Appendix E.

Table 8: Solar salt TSM independent variables for LHS

Variable	Minimum value	Maximum value	Rationale
Tubular inlet temperature (°C)	290	565	Typical inlet and outlet receiver temperatures in Solar Two, 565°C maximum temperature for Solar Salt before corrosion becomes an issue
Impinging heat flux (MW/m^2)	0.05	3	Minimum and maximum values found to be produced by FSM
Wind speed (m/s)	0.5	15	Minimum and maximum wind speeds found in weather dataset
Ambient temperature (°C)	-4	42	Minimum and maximum temperatures found in weather dataset
HTF mass flow rate (kg/s)	0.5	25	Maximum HTF flow rate through receiver of 1000 kg/s for 100 MW plant, divided by 2 flow paths then further divided by 20 tubes per panel
Height of the receiver (m)	11.4	18.2	Minimum and maximum receiver height based on FSM parameters

Table 9: FLiBe TSM independent variables for LHS

Variable	Minimum value	Maximum value	Rationale
Tubular inlet temperature (°C)	460	800	Freezing point of FLiBe is 458°C and outlet temperature for solar receiver is to be set as 740°C
Impinging heat flux (MW/m^2)	0.05	3	Minimum and maximum values found to be produced by FSM
Wind speed (m/s)	0.5	15	Minimum and maximum wind speeds found in weather dataset
Ambient temperature (°C)	-4	42	Minimum and maximum temperatures found in weather dataset
HTF mass flow rate (kg/s)	0.5	25	Maximum HTF flow rate through receiver of 1000 kg/s for 100 MW plant, divided by 2 flow paths then further divided by 20 tubes per panel
Height of the receiver (m)	11.4	18.2	Minimum and maximum receiver height based on FSM parameters

4.2 Machine learning modelling

A MLP network is a system of interconnected neurons, like a biological brain. MLP networks can be split into three sections, the input layer, the hidden layers, and the output layer, as shown in Figure 21. The number of neurons in the input layer is dictated by the number of independent variables in the dataset, while the number of neurons in the output layer is dictated by the number of dependent variables. The number of hidden layers and the number of neurons in each hidden layer are dependent on the level of linearity that a dataset represents. More linear problems typically require fewer hidden layers and neurons per hidden layer.

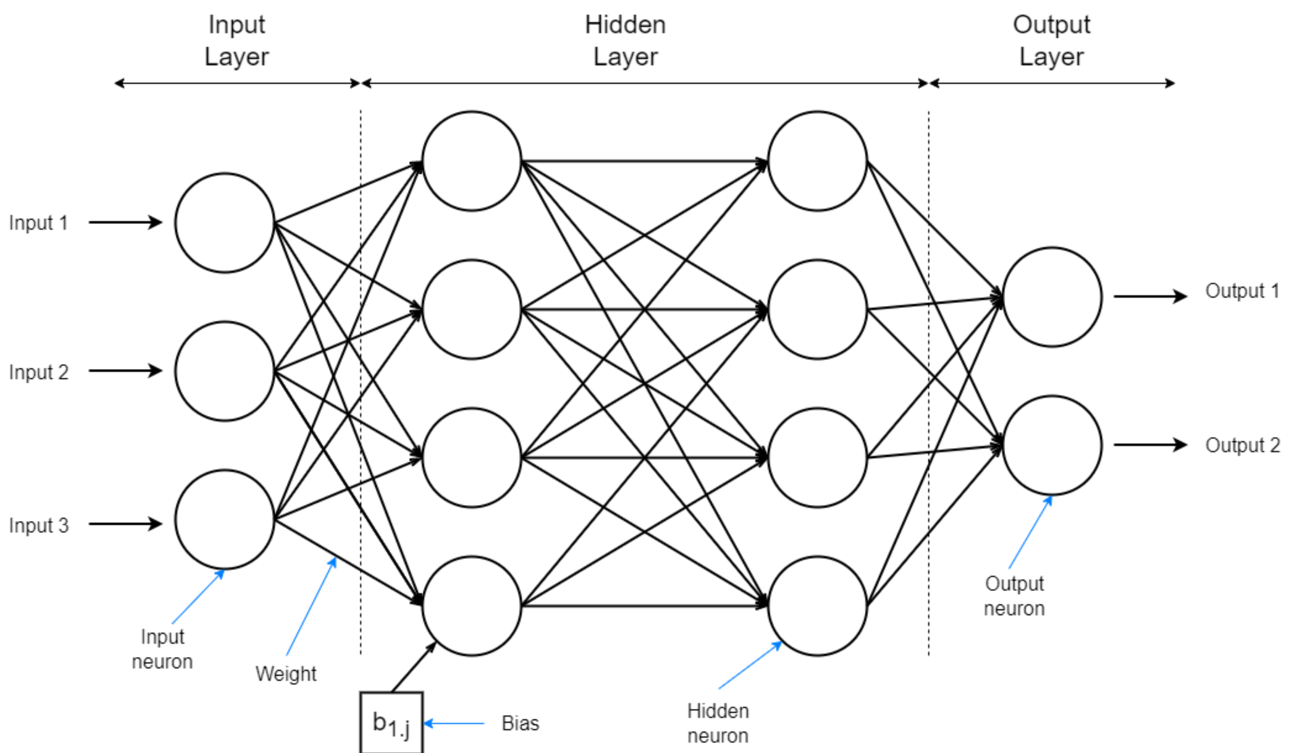


Figure 21: MLP architecture

The number of hidden layers and the number of neurons per layer are hyper parameters that are determined empirically to minimise the prediction error of the network.

As shown in Figure 21, each neuron is connected to all the neurons in the prior layer (l-1) and the subsequent layer (l+1). Each neuron sums all the signals it receives including a bias value (b), which is an additional signal each neuron receives. The connection between each neuron is scaled by an associated weight (w). These weight matrices and bias vectors are the parameters in the network that are trained through forward and backward propagation.

Forward propagation starts at the input layer, where the algorithm calculates the output for each layer (a^l), feeding this through all the layers until the output layer is reached. The output of a layer

is found by passing the weighted sum of the incoming signals through an activation function. In vector form the forward propagation step is given by:

$$a^l = \sigma_l(x^{l-1}\, w^l + b^l) \tag{4-1}$$

where σ_l is the activation function for the layer, x^{l-1} is the output vector from the previous layer, w^l is the weight matrix for the layer, and b^l is the bias vector for the layer.

The weights and biases are the trainable parameters optimised by minimising a loss function. The mean squared error (MSE) is typically selected as the loss function (Alpaydin, 2020), and was also used in this work. The MSE is calculated as

$$MSE(\hat{y}, Y) = \frac{1}{n}\sum_{i=1}^{N}(\hat{y_i} - Y_i)^2 \tag{4-2}$$

where $\hat{y}$ is the value predicted during forward propagation and Y is the target value from the dataset.

For backward propagation the gradient of the loss function is calculated with respect for the weight matrices and bias vectors. Once these gradients are calculated the trainable parameters are updated iteratively using the learning rate, similar to a relaxation factor in numerical iterative solvers, to smoothly iterate using a slow gradient descent. This is achieved alongside the mini-batch technique, which passes through batches of data and only updates the network parameters after all the data in the batch has been passed through, rather than updating after each sample of data, as in Stochastic gradient descent (SGD). Recent studies (Laubscher & Rousseau, 2021), (Haffejee & Laubscher, 2021) have found that the Adam optimization algorithm, which makes use of a variable learning rate, alongside the minibatch technique achieves a smoother gradient descent than SGD. The number of datasets in a minibatch, referred to as batch size, and the learning rate are tuneable hyperparameters.

The MLP was implemented in the PyTorch library in Python for both the FSM and TSM, with the scikit-learn library used to prepare the mini-batches as well as split the data into training and testing data. The training data is used to train the MLP while the testing data remains unseen and is used to check the out of sample error once training is completed. A ratio of 80% training data to 20% testing data was used.

4.2.1 Flux surrogate model training

A broad range of flux datasets which includes both variations in receiver geometry and solar conditions were developed as outlined in Section 4.1.1. The next step is to develop a DDSM that can predict the flux profile impinging onto the receiver with as small an error as possible. As previously mentioned, the input layer has seven variables while the output layer has 800 variables that represent the map of 800 flux values that are needed to obtain a grid independent solution of the SR model. This large disparity between the output and input dimensionality is expected to lead to a trained MLP with very low accuracy. This was confirmed in the current project by conducting various training experiments. As pointed out in the literature review, a VAE could be used to reduce the output dimensionality, as shown by Laubscher & Rousseau, 2021. However, this would create a further layer of complexity and therefore other methods were explored to overcome this problem.

Three options were considered in the structuring of the MLP. The first is to directly predict the 800 output values of heat flux based on the seven input variables. The second was to include the vertical position of each increment as an independent variable. This implies that the input dimensionality increases to eight, while the output dimensionality is reduced to 20, i.e. one flux value for each panel. The third is to include both the vertical position and the panel number as independent variables. This implies that the input dimensionality increases to nine, while the output dimensionality reduces to one, i.e. the flux value for each increment.

For a training data sample size of 16 000 it was found that both the first and last option could not be trained successfully, resulting in R^2 values of 0.2 and 0.01 respectively, despite extensive hyperparameter tuning. However, the second option with input layer dimensionality of eight and output layer dimensionality of 20 resulted in a final R^2 value of 0.99995. The final MLP input dimensionality was therefore fixed at eight, with an output dimensionality of 20.

To investigate the impact of training sample size, LHS was used to generate input data sets with the sample size varying between 100 and 12 000 samples. Each time a unique LHS was performed. Hyperparameter tuning while finding the minimum required dataset size is an interlinked and iterative empirical process. First one will train and test a DDSM using, for example a sample size of 2000 datapoints. The hyperparameters for this dataset will be tuned until a minimum in and out of sample error is reached. These hyperparameters will then be used to train DDSMs on other sample sizes, in this example 100, 500, 8000 and 12 000. From this it could be seen that the errors plateau at a sample size 8000. The hyperparameters will then be tuned again for this sample size until the error is once again minimised. These hyperparameters will then be tested once again on the various sample sizes. If it is found that the in and out of sample errors plateau again at 8000, the iterative process is complete.

It was found that both the in-sample and out-of-sample errors plateaued once 8000 datapoints were included in the dataset, as shown in Figure 22. It can also be noted that the generalisation error, i.e. the difference between the in-sample and out-of-sample error, is very low thereby minimizing overfitting.

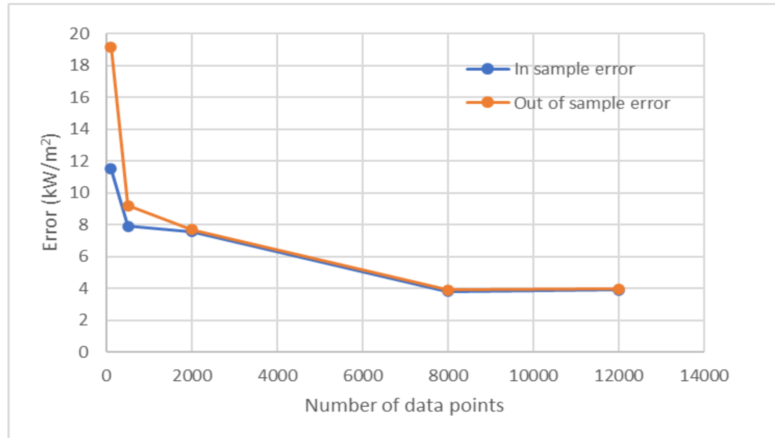


Figure 22: Error vs size of dataset

Table 10 provides a summary of the various network hyperparameters that were tested, while Figure 23 provides a graphic representation of the in and out of sample error versus the total number of neurons in the hidden layers. The final selected MLP layout has five hidden layers of 64 neurons each, as this architecture was found to minimise both the in and out of sample error. The Python script used to develop this model can be found in Appendix F.

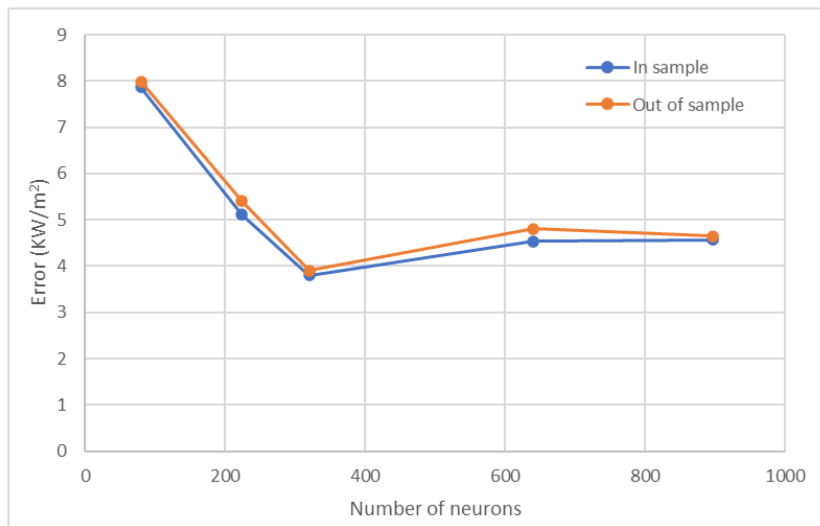


Figure 23: Error vs model complexity

Table 10: FSM hyperparameters

MLP Network	**1**	**2**	**3**	**4**	**5**
Hidden layers	5	7	5	10	7
Neurons per hidden layer	16	32	64	64	128
Total hidden neurons	80	224	448	640	896
Learning rate	8E-4	5E-4	5E-4	8E-4	4E-4

MLP Network	1	2	3	4	5
Batch size	256	256	256	256	256
Epochs	500	800	1000	500	1000
In sample error (kW/m^2)	7.8643	5.1114	3.7986	4.5323	4.5668
Out of sample error (kW/m^2)	7.9896	5.4083	3.8966	4.8109	4.6421

Figure 24 shows that the FSM accurately predicts the flux values over the entire range of the input dataset, the figure represents only out of the sample flux predictions from the dataset with 8000 samples. The surrogate model is more than 300 times faster than directly solving the flux using SolarPILOT. This means that it can produce hourly flux profiles for a full year of operation in one-minute real time, compared to more than six hours when using SolarPILOT.

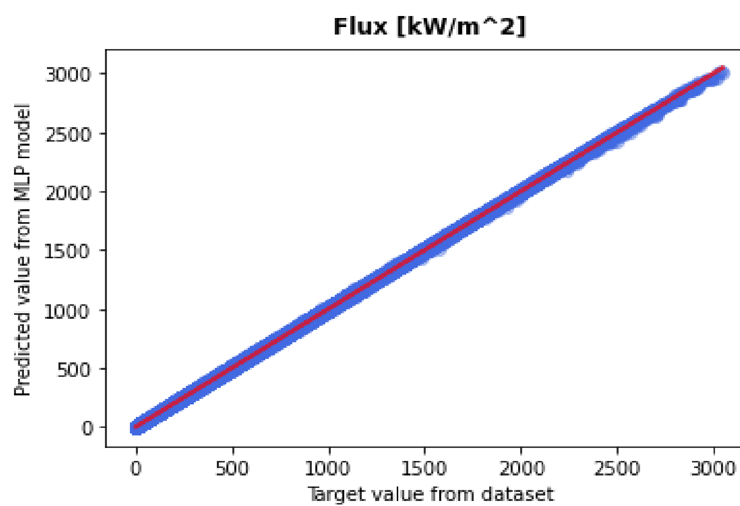


Figure 24: Actual flux from dataset vs DDSM predicted flux

As can be seen in Figure 26, the FSM can accurately predict the flux profile impinging onto the receiver. The left heat map is an example flux profile predicted using CoPylot. The right heat map

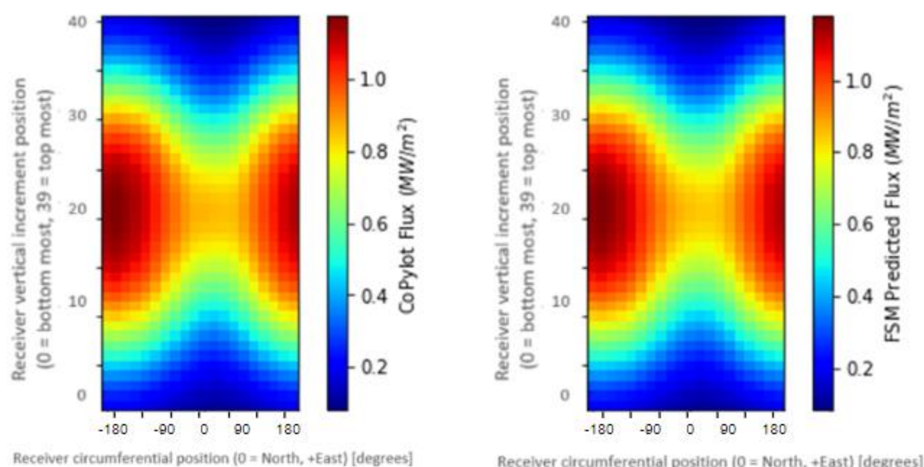


Figure 25: Flux map comparison, left - FSM, right - SolarPilot

used the FSM to predict the flux profile. This example is completely out of sample as it is from a separate LHS file to the data used to train the model. This shows that the model is robust in its prediction capabilities.

4.2.2 Thermofluid surrogate model

A broad range of datasets were developed for training the TSM in Section 4.1.2. The same iterative and empirical process of hyperparameter tuning while finding the minimum dataset size used for the FSM was used to train and tune the TSM. It was found that five hidden layers of 32 neurons each with a sample size of 16 000 datasets yield the most suitable surrogate model configuration.

The surrogate model can predict the maximum tubular temperature to within 0.86 K, the outlet molten salt temperature to within 0.014 K, the minimum tubular temperature to within 0.15 K and the heat absorption rate to within 21.4 W (with the mean absorption rate of over 10 kW) of the thermofluid half tube model. The accuracy of the model for a large range of output values is shown in Figure 27, which shows all the out of sample predictions used to test the trained model (3200 predictions from the 16 000 datasets as an 80:20 training to testing split was used). The surrogate model proved to be 35 times faster than the original thermofluid network model. The Python script

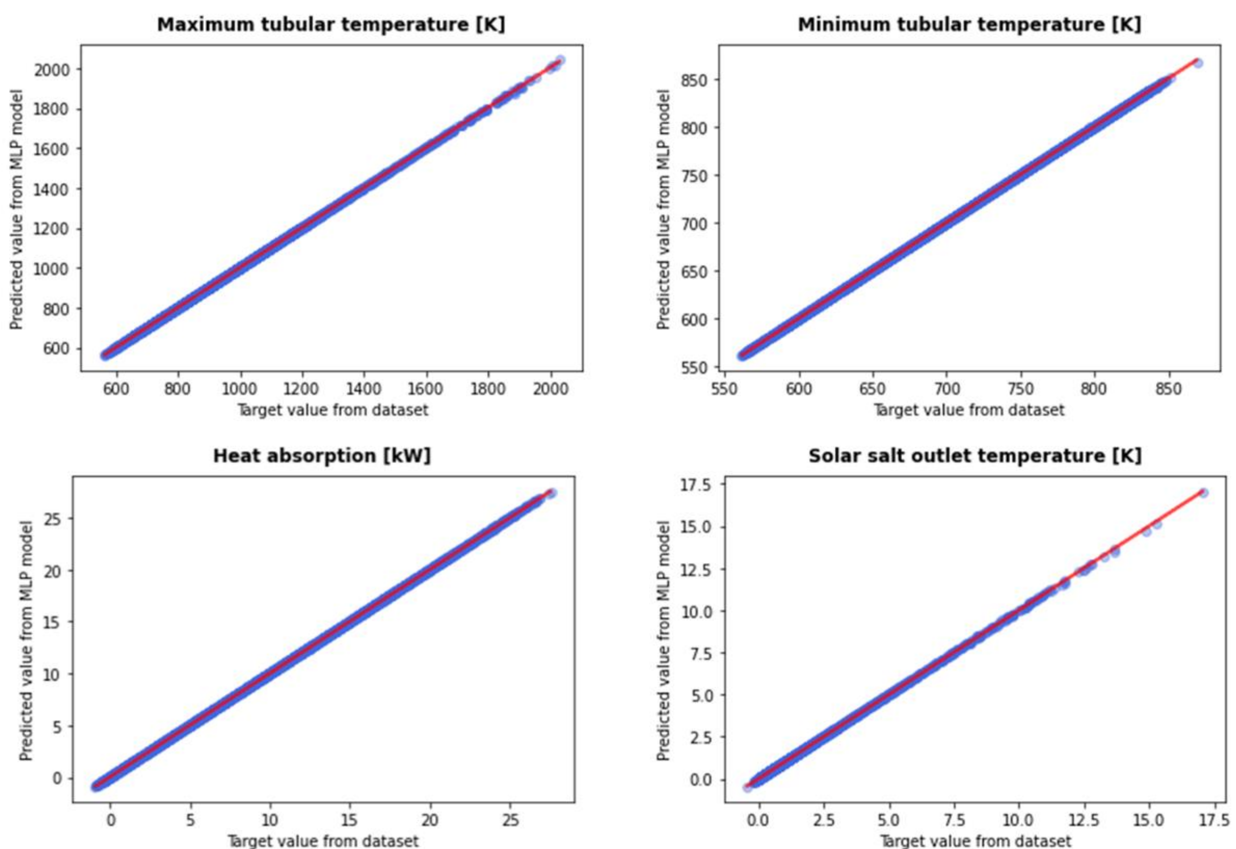


Figure 27: Thermofluid surrogate model value from dataset vs DDSM predicted value (solar salt)

used to train the TSM can be found in Appendix G. The training and hyperparameter tuning methodology for the TSM using Solar Salt and FLiBe as HTFs was identical due to the inherent similarities of these two DDSMs

.

5. Monte Carlo analysis

To demonstrate the use of the FSM and TSM, the FSM was used to predict the impinging flux profile for a receiver made up of increments consisting of the TSM rather than the original thermofluid tubular model. This will be referred to as the integrated model. The integrated model was applied in a Monte Carlo analysis for a proposed receiver design for a CSP sCO_2 plant in Upington, South Africa.

A Monte Carlo analysis is a statistical method used to model a system that has uncertain inputs. This is achieved by generating random samples from probability distributions that represent the uncertainty of the chosen inputs. These random samples are then used as inputs to a model that simulates the system. From here the results are used to analyse the range of possible outcomes as well as the probability of specific events occurring.

A Monte Carlo analysis will be used to compare the performance of two different salt chemistries, namely Solar Salt and FLiBe. The same receiver design and boundary conditions for a sCO_2 power block were used to compare the performance of the two salt chemistries. The fluid properties of these salts are summarized in Table 1. Solar salt is proven and currently used as the HTF in commercial CSP Rankine cycle plants with TITs of approximately 550°C. FLiBe has not been used commercially, but has attractive physical properties for CSP sCO_2 plants. The most attractive properties of FLiBe are its higher maximum operating temperature of 821°C (compared to Solar Salt's 600°C) and its higher volumetric heat capacity of 4826 kJ/m^2K (compared to Solar Salt's 2725 kJ/m^2K). However, FLiBe's melting point is much higher (459°C) than that of Solar Salt (222°C) meaning that solidification is of higher concern. Therefore, the minimum temperature to be considered for FLiBe will be 500°C.

This chapter will be split into three subsections. The first subsection will discuss the proposed solar receiver, TES, and sCO_2 power block design parameters. The second subsection will discuss the statistical methods used to find the probability distributions and random samples generated as the inputs for the Monte Carlo analysis. Finally, the third subsection will discuss the results of the Monte Carlo analyses, comparing the performance of the plant when using the differing salt chemistries.

5.1 Demonstration plant

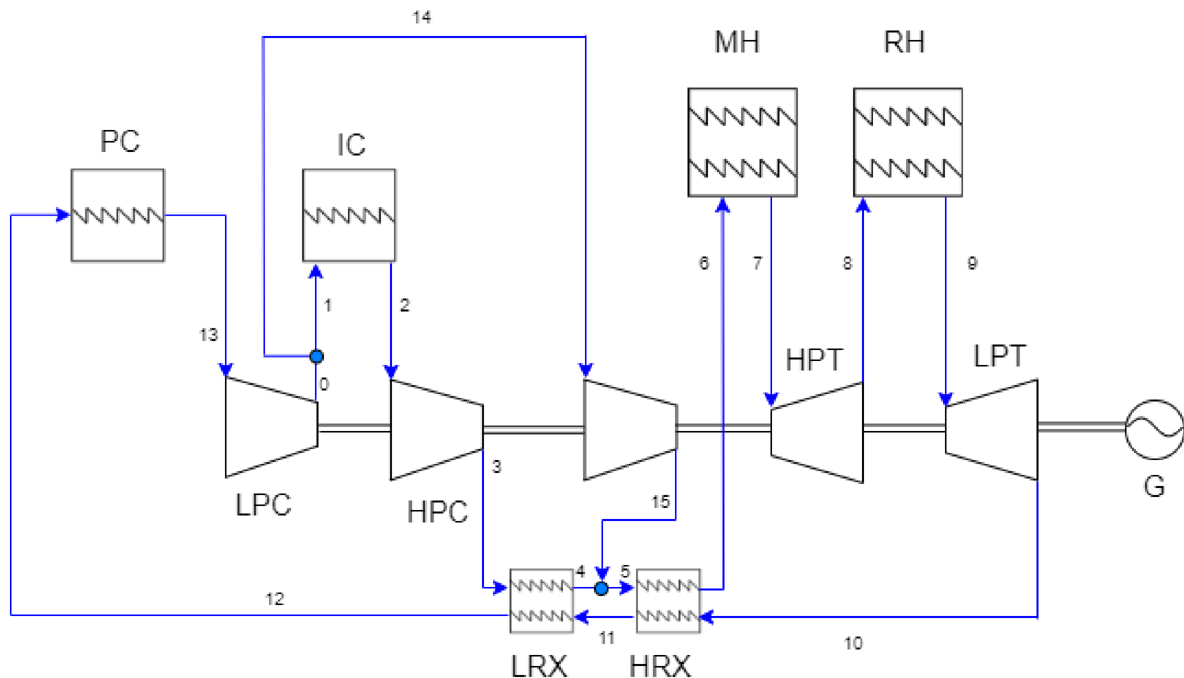


Figure 28: PCRH sCO_2 Brayton cycle

PC – precooler, IC- intercooler, MH – main heater, RH- reheater, HRX – higher heat recuperator, LRX- lower heat recuperator, LPC – low pressure compressor, HPC – high pressure compressor, HPT – high pressure turbine, LPT – low pressure turbine

The integrated model was applied in a Monte Carlo analysis for a proposed receiver design in Upington, South Africa. The receiver was specified to provide salt to a partial cooled with reheating (PCRH) sCO_2 Brayton cycle for a 50 MW_e CSP plant in Southern Africa as presented by du Sart, Rousseau & Laubscher, 2021, with the layout shown in Figure 28. The full analysis of the PCRH sCO_2 cycle including temperatures, pressures and enthalpies provided by du Sart is presented in Appendix H. The plant is designed to have 12 hours of thermal energy storage at summer solstice.

The schematic for the power block heating system with the TES and solar receiver is shown in Figure 29. The main heater (MH) is supplied with molten salt from the HST via the main heater pump (MHP) while the reheater (RH) is supplied with molten salt via the reheater pump (RHP).

To make the comparison between the two salt chemistries as equitable as possible all the parameters in the receiver design and sCO_2 power block were kept constant other than the HTF mass flow rate as well as inlet and outlet temperatures.

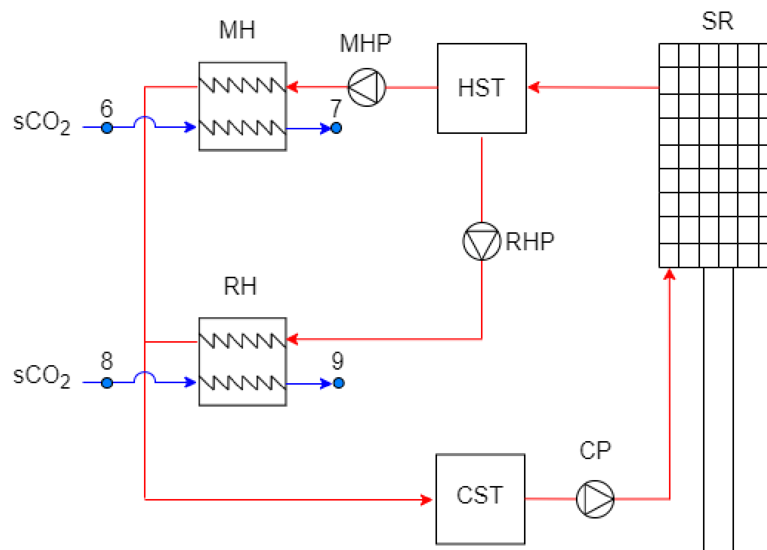


Figure 29: Solar receiver, thermal energy storage and heat exchanger schematic

CP – cold salt pump, MHP- main heater pump, RHP- reheater pump

The temperatures and heat rate required for each heater in the design case for the plant using Solar Salt as a HTF can be found in Table 11. The HST has a bulk temperature of 565°C as corrosion rates dramatically increase above this temperature when using Solar Salt. The CST has a bulk temperature of 460.5°C after mixing the streams from the MH and RH. The thermal losses in the HST and CST are assumed to be negligible, and both tanks are modelled as steady state constant temperature reservoirs as discussed in the literature review. The effectiveness of the main heater and reheater are 0.902 and 0.804 respectively. The detailed calculations for the heat exchangers using Solar Salt as a HTF as well as other design parameters for the solar receiver can be found in Appendix I.

Table 11: Solar Salt main heater and reheater parameters

Component	Fluid	Temperature in (°C)	Temperature out (°C)	Mass flow rate (kg/s)	Heat rate (MW)
Main heater	Solar salt	565	427.5	435.679	91.559
	sCO_2	412.5	550	532.6	
Reheater	Solar salt	565	494.5	423.264	45.779
	sCO_2	479.5	550	532.6	

The temperatures and heat rate required for each heater in the design case for the plant using FLiBe as a HTF can be found in Table 12. The effectiveness of the main heater and reheater are 0.733 and 0.921 respectively. The outlet temperatures of the FLiBe HTF were set at 500°C due to the risk of solidification at lower temperatures. The inlet temperature of the FLiBe salt was set at 740°C as FLiBe is thermally stable at higher temperatures and the goal of the SunShot programme is to reach

outlet temperatures in this range.The detailed calculations for the heat exchangers using FLiBe as HTF can be found in Appendix J. It can be noted that the mass flow rate of the FLiBe HTF through both the main heater and reheater is significantly lower than that of the Solar Salt due to the larger temperature difference between the inlets and outlets and higher heat capacity.

Table 12: FLiBe main heater and reheater parameters

Component	Fluid	Inlet temperature (°C)	Outlet temperature (°C)	Mass flow rate (kg/s)	Heat rate (MW)
Main heater	FLiBe	740	500	159.96	91.559
	sCO_2	412.5	550	532.6	
Reheater	FLiBe	565	494.5	79.98	45.779
	sCO_2	740	500	532.6	

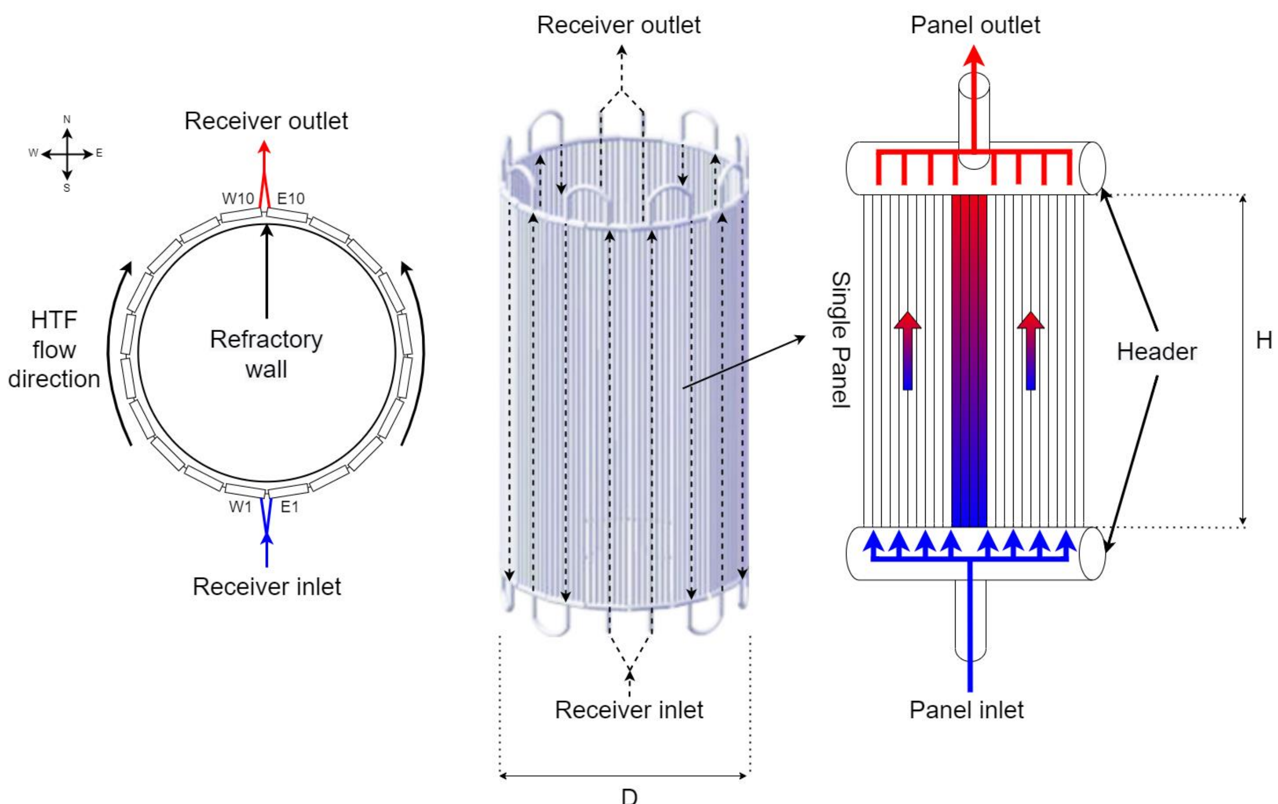


Figure 30: Annotated solar receiver architecture schematic

The solar receiver consists of 20 panels, each made up of 30 tubes in parallel, with every tube having an outer diameter of 42.2 mm and 1.2 mm wall thickness. The height and diameter (H and D in Figure 30) of the cylindrical receiver are 13.233 m and 8.828 m respectively. The receiver efficiency was assumed to be 85%. The full specifications of the receiver can be found in Table 13, with the detailed design calculation presented in Appendix I.

Table 13: Case study solar receiver geometry

Receiver diameter - D	8.828 m
Receiver height - H	13.233 m
Receiver area	367.02 m^2
Height to diameter ratio	1.5
Number of tubes per panel	30
Number of panels	20
Tube outer diameter	42.2 mm
Tube inner diameter	39.8 mm
Tubular pitch	46 mm
Receiver design power	323.3 MW
Receiver design flux	0.88 MW/m^2

The tubes in each of the 20 panels are represented by a single equivalent tube which is discretized into 40 increments along its length. This results in 20 panels x 40 increments = 800 tube increments in total. The flow network layout is shown schematically in Figure 31 (only 20 increments are shown per panel). The flow starts on the southern side of the receiver where it is split into two flow paths. From here the HTF is further split into 30 tubes by a header. The model simulates a representative tube in each panel, traversing the receiver in a serpentine manner until the flow is combined at the outlet.

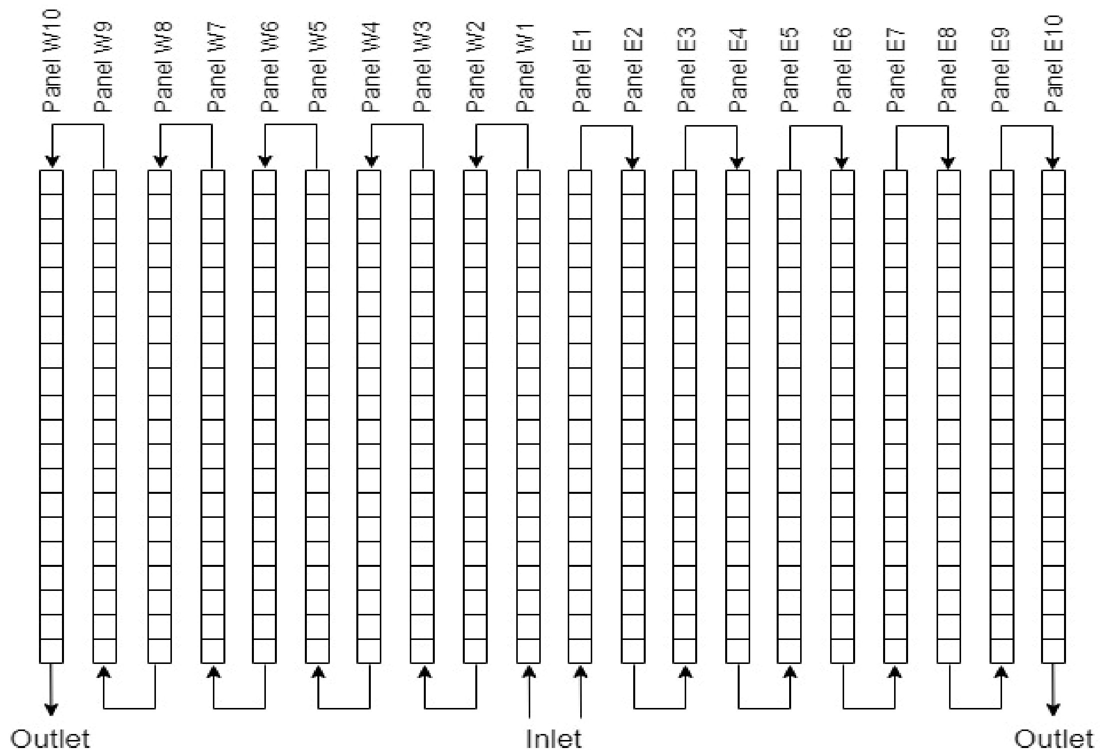


Figure 31: Discretised solar receiver flow diagram

The DNI obtained from the statistical sampling, the calculated azimuthal and elevation angles, as well as the specifications of the solar receiver are used as inputs to the FSM. The FSM outputs a flux profile of 800 fluxes for the receiver. The TSM is then used to model the thermofluid performance

of the solar receiver. This is achieved by using the TSM as a surrogate model for each discretised increment in the solar receiver.

5.2 Statistical methods

To test the design of the solar receiver in terms of both power output and thermal performance a Monte Carlo simulation was performed using the integrated model. The three independent variables chosen for the Monte Carlo analysis were DNI, ambient temperature and wind speed. Hourly weather data including windspeed, wind direction, dry bulb temperature, relative humidity and DNI was obtained from SolarGIS for a 28-year period, running from 1994 to 2022.

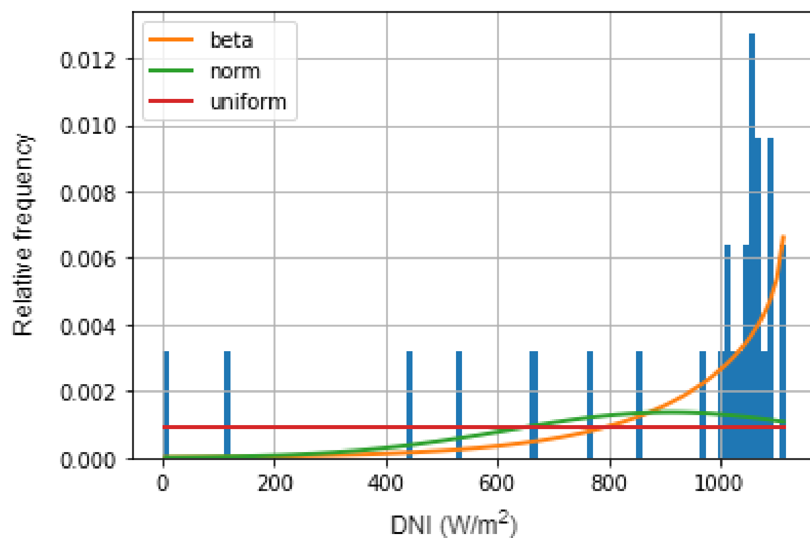


Figure 32: DNI fitted distribution histogram

The DNI, ambient temperature and wind speed respectively for the 28 year period were binned into individual hours of the year, resulting in 8760 bins for each variable, each containing 28 data points. The measured data for each hour of the year was analysed to determine appropriate probability

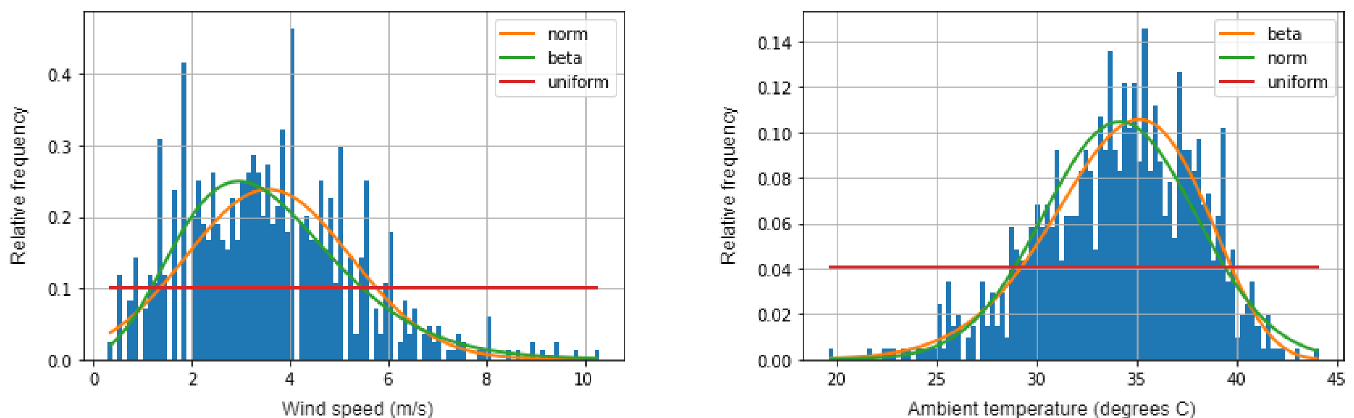


Figure 33: Wind and temperature fitted distribution histogram

distributions using the Fitter library in Python. It was found that the beta distribution best represents the DNI data. This can be seen in the example presented in Figure 32, which shows the distribution of DNI for the 12th hour of the 1st of January.

Both wind speed and ambient temperature were found to be best represented by normal distributions. The average Bayesian Information Criterion for windspeed was 684.57 for the normal distribution and 707.36 for the beta distribution. The average Bayesian Information Criterion for ambient temperature was 837.37 for the normal distribution and 851.26 for the beta distribution. Meanwhile The fact that both the normal and beta distributions provide good fits for wind speed and temperature can be seen in the examples in Figure 33, which shows the distribution of wind speed and temperature for the 8th hour on the 1st of January.

In order to do the statistical sampling of input data based on the appropriate distributions, the location and scale parameters need to be determined for the normal distributions, while the shape, location and scale parameters need to be determined for the beta distributions, for each hour of the year. The scipy.stats library was used to determine these parameters via the method of maximum likelihood estimation (Richards, 1961). The probability density functions are then used to obtain 10 random samples for DNI, ambient temperature, and wind speed for each hour of the year using random variate sampling (RVS) to simulate 10 consecutive years of operation.

The flux predicted for each increment by the FSM as well as the temperature and wind speed obtained from random variate sampling are used as inputs for the TSM. The outlet temperature predicted by the TSM from the preceding increment is used as the inlet temperature for the subsequent increment as is the case for the solar receiver thermofluid model as described in Chapter 3.

The mass flow rate through the receiver is calculated iteratively. This is done by guessing a mass flow rate value based off the average flux of the receiver given by:

$$\dot{m_{guess}} = \frac{\dot{q_{flux.ave}}\, A_{rec}}{Cp_{salt}(T_{rec.out} - T_{rec.in})} \tag{5-1}$$

where $\dot{q_{flux.ave}}$ is the average flux around the receiver in W/m^2, A_{rec} is the area of the cylindrical receiver, $T_{rec.out}$ is the target outlet temperature of the receiver and $T_{rec.in}$ is the inlet temperature of the receiver. The guessed mass flow rate is then used to simulate the receiver through all 800 increments, thereby calculating the outlet temperature of the receiver. The mass flow rate is then iteratively solved using the Newton Raphson method until convergence when the desired target outlet temperature is obtained. The Newton Raphson method is implemented using the

scipy.optimize.newton function. As the mass flow rate has been iterated for and converged upon all the inputs needed for the FSM are now known and the receiver can be simulated.

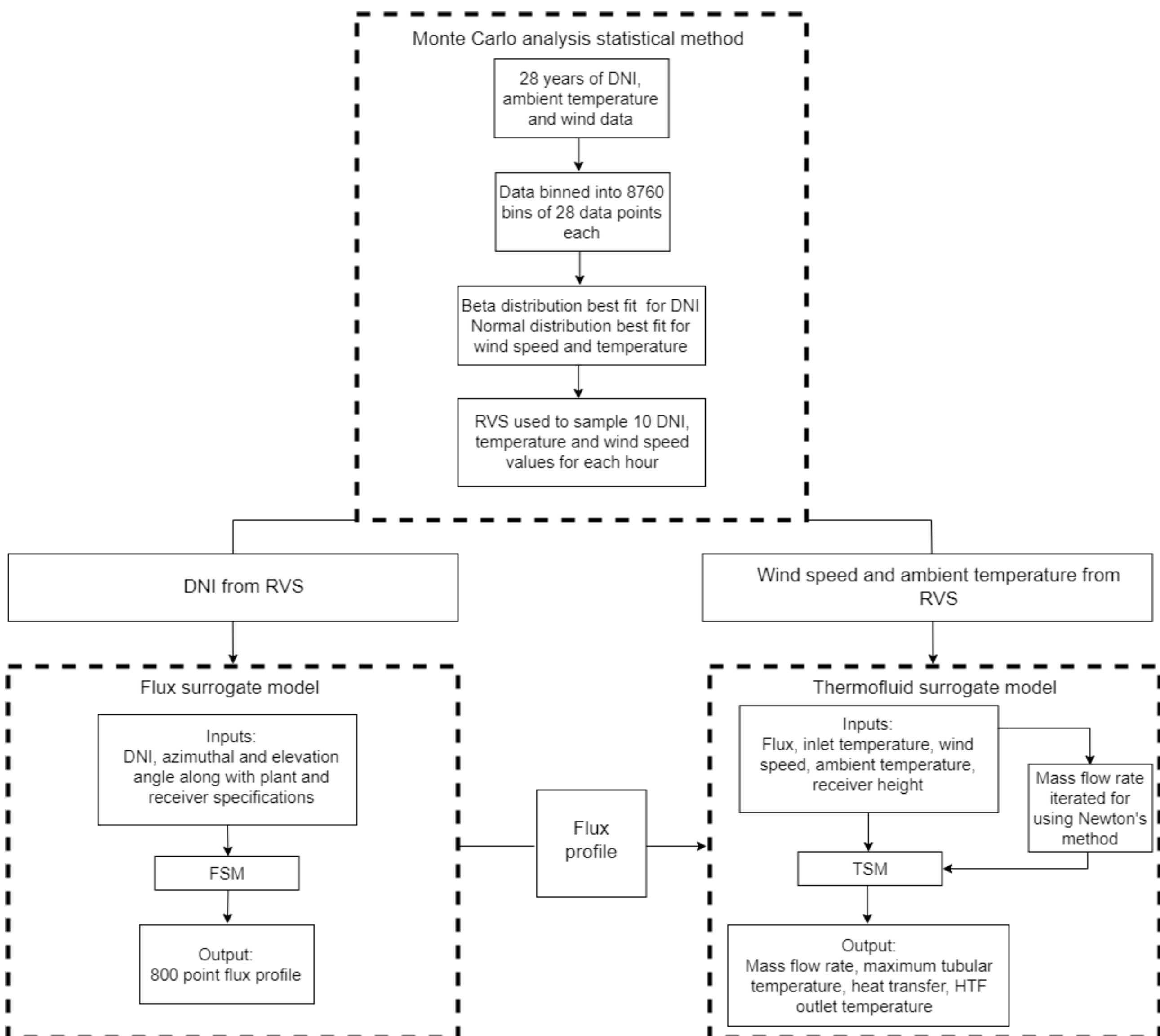


Figure 34: Monte Carlo analysis methodology

The methodology applied in the Monte Carlo analysis is shown in Figure 34. First the distributions fitting the DNI, wind speed, and ambient temperature are determined. The probability density functions for each input variable are then calculated and then used to complete RVS. The sampled DNI value as well as the calculated azimuthal and elevation angles are then used as inputs to the FSM which produces an 800-point flux profile. This flux profile is then used as the input to the TSM along with the randomly sampled wind speed and ambient temperature. The mass flow rate is then iterated until the desired outlet temperature is obtained. The receiver is then simulated using the TSM and the outputs are recorded. This process is repeated until each hour of the year has been

simulated 10 times. The receiver using the Solar Salt and FLiBe HTFs used the exact same set of input variables sampled using RVS to keep the comparison as equitable as possible.

5.3 Results

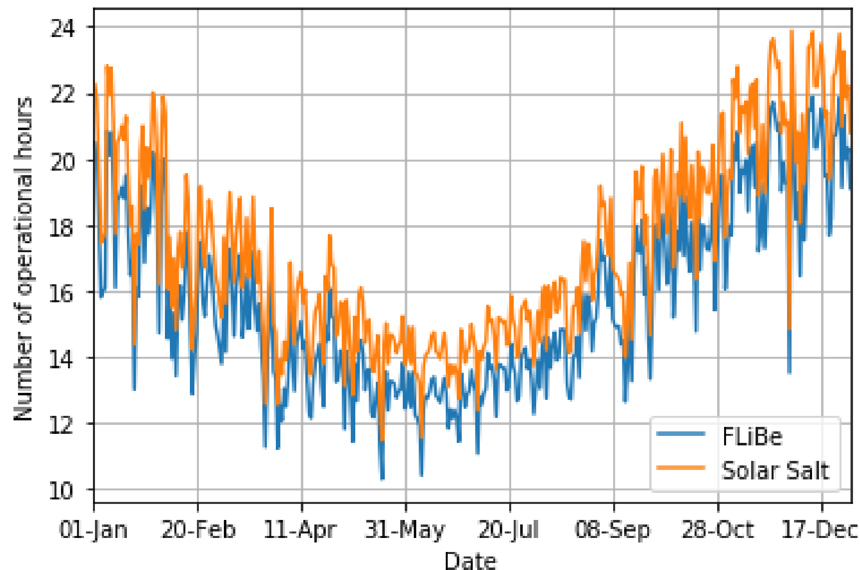


Figure 35: Monte Carlo analysis plant energy output

Figure 35 shows the predicted average number of hours per day for which the SR and storage system can provide hot HTF to the power plant for two cases using Solar Salt and FLiBe as HTFs. This was calculated as the cumulative mass flow provided by the receiver during a given day and dividing it by the required mass flow rate through the main heater and reheater.

As expected, the maximum is achieved around the summer solstice in December, while in the midwinter months of June and July the operational capacity is much lower, with an average of around 14 hours per day for the receiver which used Solar Salt. Interestingly, the FLiBe receiver consistently underperforms the Solar Salt receiver by 8-11%. This makes sense as the higher fluid temperatures lead to higher tube metal temperatures. These higher metal temperatures lead to higher convective and radiative losses. The higher tube metal temperatures can be seen in Figure 36. The maximum tube metal temperature of the FLiBe receiver is consistently 120°C higher than that of the Solar Salt receiver.

On average each year the plant using Solar Salt as a HTF will be able to produce power at full load for 6311 hours, giving it a load factor 0.720. The plant using FLiBe as a HTF will be able to produce power at full load for 5739 hours yielding a load factor of 0.655. However, due to the lower mass flow of the FLiBe based receiver the power consumed by the cold salt pump (CP) at full load is

significantly lower than that of the Solar Salt based system. The Solar Salt and FLiBe based CPs consume 2.179 MW and 0.481 MW respectively.

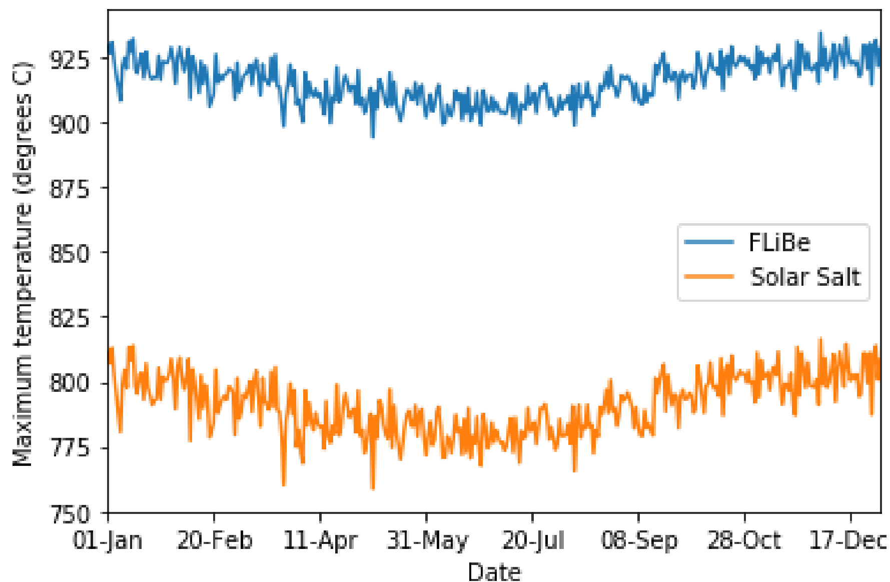


Figure 36: Monte Carlo analysis maximum tubular temperature

The mass flow rate of the FLiBe system is lower due to the fact that the FLiBe have a higher specific heat capacity than Solar Salt, and that the difference in temperature between in the inlet of outlet of the receiver is larger for the FLiBe system. This fact also means that the FLiBe based system will have significantly smaller tank sizes. For a tank that can provide 12 hours of TES the Solar Salt based system will require 40 902 tons of fluid which takes up a volume of 21 442 m^3. An equivalent FLiBe TES system will require 11 425 tons of fluid taking up 6203 m^3. The assumption was made that the diameter is double the height for the HST and CST. This means that the HST and CST would each need to have a diameter of 37.94 m and a height of 18.98 m for the Solar Salt based TES system. The FLiBe based system would require a diameter of 25.09 m and a height of 12.55 m in comparison. The calculation for the pump and tank sizing for the Solar Salt and FLiBe based system can be found in Appendix I and Appendix J respectively.

The maximum tube wall temperatures shown in Figure 36 follows the same trend as the number of operational hours with peaks during summer and troughs during winter. This is due to the lower DNI as well as the lower ambient temperatures, which results in more efficient cooling of the tube walls. While the location of the maximum tubular temperature was tracked, the results were remarkably consistent with only four of the 800 discretized sections accounting for the maximum temperature throughout the full 10 year period. These four sections all also neighbour one another, being situated in the middle south facing section of the receiver. This is an interesting finding in terms of condition monitoring, and highlights that the default aiming strategy simulated originally in SolarPILOT clearly causes high fluxes on this area.

To illustrate this phenomenon three separate flux profiles produced using both the FSM (1) and CoPylot (2) were considered in more detail as shown in Figure 37. Flux profile A was produced on a mid-winter's morning at 8am with a DNI of 500 W/m^2. Flux profile B was produced at noon in mid-summer with a DNI of 1100 W/m^2. Flux profile C was produced in the late afternoon in autumn at 6pm with a DNI of 300 W/m^2. While the intensity of the highest flux is different in all the cases, the patterns are near identical. It can be seen that flux profiles 1A (winter morning using the FSM) and 2A (winter morning using CoPylot) are near identical in both pattern and magnitude. The same can be seen for flux profiles 1B and 2B, and 1C and 2C. This shows that the FSM can accurately predict flux profiles for a range of conditions and seasons.

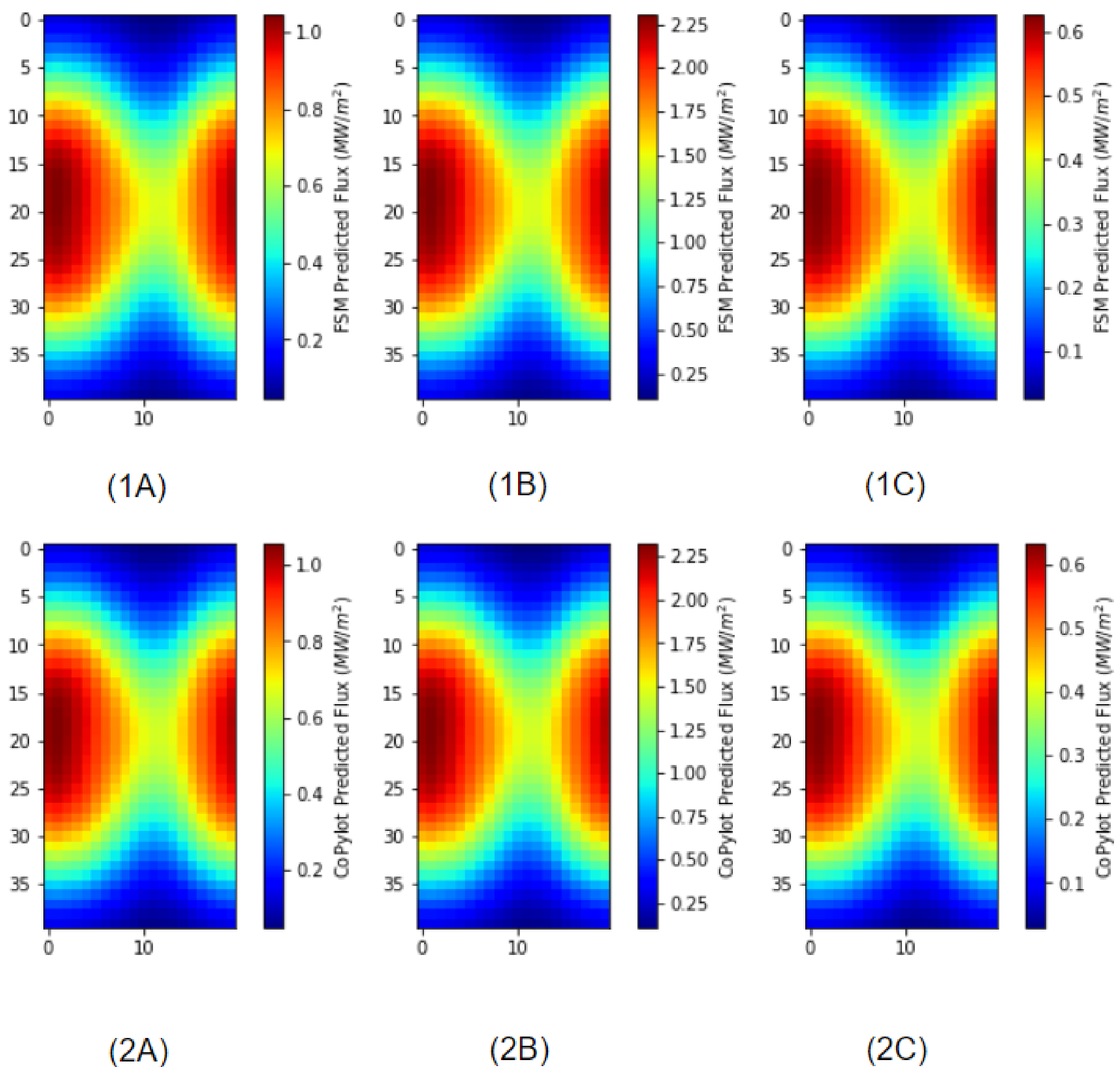


Figure 37: Flux profiles for varying seasons

Produced using the FSM: (1A) - winter morning, (1B) - summer midday, (1C) – autumn afternoon

Produced using CoPylot: (1A) - winter morning, (1B) - summer midday, (1C) – autumn afternoon

6. Conclusions and Recommendations

6.1 Conclusions

The original aim of the project was to develop a reduced order modelling methodology for external cylindrical solar receivers that can aid in the design and analysis process, based on models that are sufficiently accurate and computationally inexpensive.

A thermofluid network-based model of the solar receiver was first developed. The model makes use of a coarse discretization procedure which reduces the computational cost relative to CFD models, but still captures all the relevant thermofluid phenomena. The receiver geometry is discretized along the height and around the circumference and each increment is represented by an equivalent thermal resistance network. The resistance network was calibrated using a detailed computational fluid dynamics model. The calibrated model allows the maximum tube wall temperatures to be calculated within 0.2% of the CFD model.

The overall model also solves the mass, energy, and momentum balance equations for the heat transfer fluid inside the tubes to determine the mass flow, pressure drop and temperature distributions. The solar heat flux profiles incident on the receiver were produced using SolarPILOT. The model was made grid independent by conducting a grid independence study. The grid independence study showed that 40 vertical increments are sufficient for the expected error of maximum tubular temperature to be below 1%. The model was validated by comparing the results with plant data for the Solar Two plant. The model results for the overall heat transfer are within 5% of real-plant data for eight cases covering a range of different operational conditions.

The use of DDSMs was shown reduce computational cost whilst not significantly reducing the accuracy of the thermofluid model as well as the flux model. The FSM that than can predict fluxes of all magnitudes with accuracy while reducing computational cost. This was achieved by using the vertical increment number on the receiver as an independent variable in the training process. The FSM is more than 300 times faster than directly solving the flux using SolarPILOT. This means that it can produce hourly flux profiles for a full year of operation in one-minute real time, compared to more than six hours when using SolarPILOT. The TSM of the solar receiver thermofluid network also captures all the necessary information while reducing computational cost by a factor of 35.

The integrated model that combines the two surrogate models were successfully applied to conduct a Monte Carlo analysis. The Monte Carlo analysis proved useful in comparing the operation of a case study receiver using different HTFs. The receiver using Solar Salt as a HTF is able to produce 9% more heat over the span of an average year than the receiver using FLiBe as a HTF. However, due to the

FLiBe receiver's lower mass flow rate, the power consumption of the pump as well as the size of the TES tanks are significantly lower than that of the Solar Salt receiver.

This methodology can be used in the design and analysis of solar receivers since it can predict whether the plant has the operational capacity for which it was designed, while also tracking the maximum tube metal temperatures. This can also be valuable as part of a condition monitoring tool in future CSP plants.

6.2 Recommendations

The current work was limited to steady state modelling of thermofluid phenomena. However, there is a gap in literature in the transient modelling of solar receivers. While the effects of clouds causing transient events (Augsburger & Favrat, 2013) as well as transient modelling of the sCO_2 Brayton cycle is reported in literature , direct modelling of solar receivers in a transient manner has not been widely explored. The transient modelling of solar receivers, therefore, would be of value in the development of sCO_2-CSP plants.

In the thermofluid modelling section it was assumed that all of the tubes in the receiver are equally impacted by wind speed. This is not how a receiver would function as only the tubes facing the direction the wind is coming from would get the full impact of the wind. A study into this phenomenon was outside of the scope of the project. A study into this phenomenon could allow for more accurate modelling of solar receivers in future if the thermal effects could be better understood.

The receiver which used Solar Salt as the HTF was able to power the proposed plant for an average of 9.15% more hours than the receiver which used FLiBe as the HTF. However, the lower mass flow rate of the FLiBe system meant that the pumps consumed less power and the overall tank sizes were smaller in volume by a magnitude of three. A LCOE analysis on this could be valuable when deciding which HTF to use in the future. This, however, was outside of the scope of the project.

7. List of References

Achkari, O. & Fadar, A. El. 2020. Latest developments on TES and CSP technologies – Energy and environmental issues , applications and research trends. *Applied Thermal Engineering*. 167(October 2018):114806. DOI: 10.1016/j.applthermaleng.2019.114806.

Alpaydin, E. 2020. *Introduction to machine learning*. MIT press.

Angelini, G., Lucchini, A. & Manzolini, G. 2014. Comparison of thermocline molten salt storage performances to commercial two-tank configuration. *Energy Procedia*. 49:694–704. DOI: 10.1016/j.egypro.2014.03.075.

Augsburger, G. & Favrat, D. 2013. Modelling of the receiver transient flux distribution due to cloud passages on a solar tower thermal power plant. *Solar Energy*. 87(1):42–52. DOI: 10.1016/j.solener.2012.10.010.

Çengel, Y.A., Cimbala, J.M. & Turner, R.H. 2012. *Fundamentals of Thermal-fluid Sciences*. (Çengel series in engineering thermal-fluid sciences). McGraw-Hill. Available: https://books.google.co.za/books?id=oDpvpwAACAAJ.

Conroy, T., Collins, M.N. & Grimes, R. 2020. A review of steady-state thermal and mechanical modelling on tubular solar receivers. *Renewable and Sustainable Energy Reviews*. 119(November 2019):109591. DOI: 10.1016/j.rser.2019.109591.

Geyer, M. & Stine, W. 2001. *Power From The Sun*.

Haffejee, R.A. & Laubscher, R. 2021. Energy and AI Application of machine learning to develop a real-time air-cooled condenser monitoring platform using thermofluid simulation data. *Energy and AI*. 3:100048. DOI: 10.1016/j.egyai.2021.100048.

Hamilton, W.T., Wagner, M.J. & Zolan, A.J. 2021. Demonstrating SolarPILOT's Python API Through Heliostat Optimal Aimpoint Strategy Use Case: Preprint. In *5th International Conference on Energy Sustainability (ES2021)*. Available: https://www.osti.gov/biblio/1778189.

Hashem, S., Wai, L.C. & Basim, I.F. 2018. Solar Thermal Power : Appraisal of Solar Power. 04003:1–6.

Heller, L. 2013. *Literature review on heat transfer fluids and thermal energy storage systems in CSP plants*. Available: http://sterg.sun.ac.za/wp-content/uploads/2011/08/HTF_TESmed_Review_2013_05_311.pdf.

Heydenrych, J., Rousseau, P. & du Sart, C. 2022. Reduced-order modelling of central solar tower receivers using an equivalent thermal resistance network. In *16th INTERNATIONAL CONFERENCE ON HEAT TRANSFER, FLUID MECHANICS AND THERMODYNAMICS*. 911–916.

Ho, C.K. 2016. A review of high-temperature particle receivers for concentrating solar power. *Applied Thermal Engineering*. 109:958–969. DOI: 10.1016/j.applthermaleng.2016.04.103.

Laubscher, R. 2019. Time-series forecasting of coal- fi red power plant reheater metal temperatures using encoder-decoder recurrent neural networks. *Energy*. 189:116187. DOI: 10.1016/j.energy.2019.116187.

Laubscher, R. & Rousseau, P. 2021. An integrated approach to predict scalar fields of a simulated

turbulent jet diffusion flame using multiple fully connected variational autoencoders and MLP networks. *Applied Soft Computing*. 101:107074. DOI: 10.1016/j.asoc.2020.107074.

Liao, Z., Li, X., Xu, C., Chang, C. & Wang, Z. 2014. Allowable flux density on a solar central receiver. *Renewable Energy*. 62:747–753. DOI: 10.1016/j.renene.2013.08.044.

Liu, X.J., Kong, X.B., Hou, G.L. & Wang, J.H. 2013. Modeling of a 1000 MW power plant ultra super-critical boiler system using fuzzy-neural network methods. *Energy Conversion and Management*. 65:518–527. DOI: 10.1016/j.enconman.2012.07.028.

Ma, Z. & Turchi, C.S. 2011. Advanced Supercritical Carbon Dioxide Power Cycle Configurations for Use in Concentrating Solar Power Systems: Preprint. In *Supercritical CO2 Power Cycle Symposium, 24-25 May 2011, Boulder, Colorado*. Available: https://digital.library.unt.edu/ark:/67531/metadc833207/.

Mckay, M.D., Beckman, R.J. & Conover, W.J. 2000. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output From a Computer Code. *Technometrics*. 42(1):55–61. DOI: 10.1080/00401706.2000.10485979.

Mehos, M., Turchi, C., Vidal, J., Wagner, M., Ma, Z., Ho, C., Kolb, W., Andraka, C., et al. 2017. Concentrating Solar Power Gen3 Demonstration Roadmap Concentrating Solar Power Gen3 Demonstration Roadmap. (January).

Montes, M.J., Linares, J.I., Barbero, R. & Rovira, A. 2020. Proposal of a new design of source heat exchanger for the technical feasibility of solar thermal plants coupled to supercritical power cycles. *Solar Energy*. 211(March):1027–1041. DOI: 10.1016/j.solener.2020.10.042.

NREL. 2022. *Concentrating Solar Power Project Database*. Available: https://solarpaces.nrel.gov/ [2022, December 13].

Pacheco, J., Showalter, S. & Kolb, W. 2002. Development of a Molten-Salt Thermocline Thermal Storage System for Parabolic Trough Plants. *Journal of Solar Energy Engineering*. 124:153. DOI: 10.1115/1.1464123.

Pacheco, J., Bradshaw, R.W., Dawson, D.B., De la Rosa, W., Gilbert, R., Goods, S.H., Hale, M.J., Jacobs, P., et al. 2002. Final Test and Evaluation Results from the Solar Two Project. *Contract*. (January):294. DOI: 10.2172/793226.

Powell. 1970. A hybrid method for nonlinear equations. *Numerical Methods for Nonlinear Algebraic Equations*. Available: https://cir.nii.ac.jp/crid/1571135649454396800.

Raidoo, R. & Laubscher, R. 2022. Data-driven forecasting with model uncertainty of utility-scale air-cooled condenser performance using ensemble encoder-decoder mixture-density recurrent neural networks. *Energy*. 238:122030. DOI: 10.1016/j.energy.2021.122030.

Reyes-Belmonte, M.A., Sebastián, A., Romero, M. & González-Aguilar, J. 2016. Optimization of a recompression supercritical carbon dioxide cycle for an innovative central receiver solar power plant. *Energy*. 112:17–27. DOI: 10.1016/j.energy.2016.06.013.

Richards, F.S.G. 1961. A method of maximum-likelihood estimation. *Journal of the Royal Statistical Society: Series B (Methodological)*. 23(2):469–475.

Rodríguez-Sánchez, M.R., Marugan-Cruz, C., Acosta-Iborra, A. & Santana, D. 2014. Comparison of simplified heat transfer models and CFD simulations for molten salt external receiver. *Applied Thermal Engineering*. 73(1):993–1005. DOI: 10.1016/j.applthermaleng.2014.08.072.

Rousseau, P. & Laubscher, R. 2020. A thermofluid network-based model for heat transfer in membrane walls of pulverized coal boiler furnaces. *Thermal Science and Engineering Progress*. 18(December 2019):100547. DOI: 10.1016/j.tsep.2020.100547.

du Sart, C.F., Rousseau, P.G. & Laubscher, R. 2021. Cycle selection and system-level optimisation of a 50 MWe sCO2 CSP plant. In *Proceedings of the 15th international conference on heat transfer, fluid mechanics and thermodynamics*. 560–565. Available: https://www.researchgate.net/publication/353572355_Cycle_selection_and_system-level_optimisation_of_a_50_MWe_sCO2_CSP_plant.

Serrano-López, R., Fradera, J. & Cuesta-López, S. 2013. Molten salts database for energy applications. *Chemical Engineering and Processing - Process Intensification*. 73:87–102. DOI: 10.1016/j.cep.2013.07.008.

Tan, P., He, B., Zhang, C., Rao, D., Li, S., Fang, Q. & Chen, G. 2019. Dynamic modeling of NO X emission in a 660 MW coal- fi red boiler with long short-term memory. *Energy*. 176(X):429–436. DOI: 10.1016/j.energy.2019.04.020.

Turchi, C., Gage, S., Martinek, J., Jape, S., Armijo, K., Coventry, J., Pye, J., Asselineau, C., et al. 2021. *CSP Gen3 : Liquid-Phase Pathway to SunShot*.

Vignarooban, K., Xu, X., Arvay, A., Hsu, K. & Kannan, A.M. 2015. Heat transfer fluids for concentrating solar power systems – A review. *Applied Energy*. 146:383–396. DOI: 10.1016/j.apenergy.2015.01.125.

Wagner, M.J. & Wendelin, T. 2018. SolarPILOT : A power tower solar fi eld layout and characterization tool. 171(June):185–196. DOI: 10.1016/j.solener.2018.06.063.

Wang, J., Guo, L., Zhang, C., Song, L., Duan, J. & Duan, L. 2020. Thermal power forecasting of solar power tower system by combining mechanism modeling and deep learning method. *Energy*. 208:118403. DOI: 10.1016/j.energy.2020.118403.

Wang, K., He, Y.L. & Zhu, H.H. 2017. DOI: 10.1016/j.apenergy.2017.03.099.

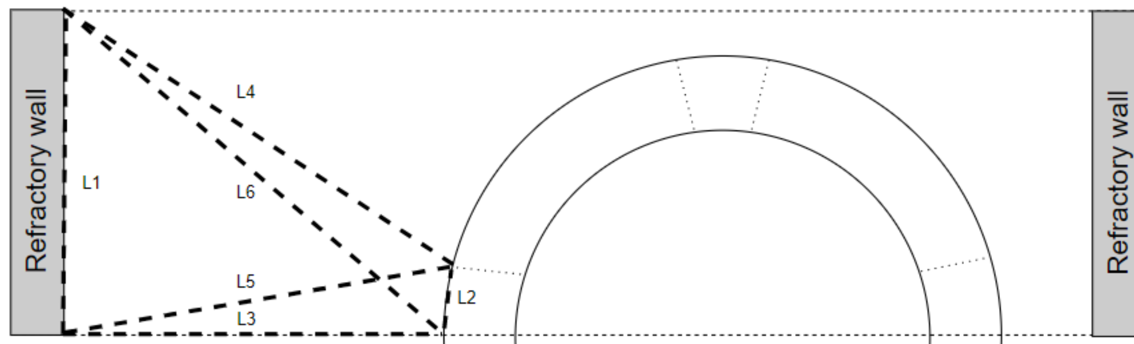
Wang, W.Q., Qiu, Y., Li, M.J., He, Y.L. & Cheng, Z.D. 2020. Coupled optical and thermal performance of a fin-like molten salt receiver for the next-generation solar power tower. *Applied Energy*. 272(April):115079. DOI: 10.1016/j.apenergy.2020.115079.

Zaversky, F., García-Barberena, J., Sánchez, M. & Astrain, D. 2013. Transient molten salt two-tank thermal storage modeling for CSP performance simulations. *Solar Energy*. 93:294–311. DOI: 10.1016/j.solener.2013.02.034.

Zhang, Q., Jiang, K., Kong, Y., Wu, J. & Du, X. 2021. Study on Outlet Temperature Control of External Receiver for Solar Power Tower. *Energies*. 14(2). DOI: 10.3390/en14020340.

Zheng, M., Zapata, J., Asselineau, C.A., Coventry, J. & Pye, J. 2020. Analysis of tubular receivers for concentrating solar tower systems with a range of working fluids, in exergy-optimised flow-path configurations. *Solar Energy*. 211(March):999–1016. DOI: 10.1016/j.solener.2020.09.037.

Appendix A. View factor calculation



View factor calculation for the *ff* sector

$$L_1 = 22.25\ mm\ (half\ of\ 44.5\ mm\ pitch)$$

$$L_2 = 2.82385\ mm\ (2.13\%\ of\ outer\ circumerence)$$

$$L_3 = 0\ mm\ (flux\ produced\ by\ SolarPILOT\ assumed\ to\ impinge\ directly\ upon\ the\ tube)$$

$$L_4 = \sqrt{L_3^2 + (L_1 - L_2)^2} = 19.426\ mm$$

$$L_5 = \sqrt{L_2^2 + L_3^2} = 2.824\ mm$$

$$L_6 = \sqrt{L_1^2 + L_3^2} = 22.25\ mm$$

$$F_{1-2} = \frac{(L_5 + L_6) - (L_3 + L_4)}{2L_1} = 0.127$$

This process was repeated for sectors *f, c, b, bb*.

Appendix B. Validation Case 1C Jupyter Notebook

```
import numpy as np
import scipy as sp
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import fsolve
import time as time

start_time = time.time()

kelv = 273.15

T_amb = 32+kelv #K
T_ss_i = 296+kelv #Solar Salt Initial Temperature (K)
T_ss_e = 553+kelv
T_surr = 0
m_dot_ss= 85/2/32/2
u_air = 0.6 #wind speed (m/s)

#Eastern Panels
n_panels = 12

df = pd.read_csv("flux_validation1C.csv") #import flux csv
Q = df.to_numpy() #convert flux df to numpy
Q_n = np.zeros((20,n_panels)) #North facing flux
Q_n_inv = np.zeros((n_panels,20)) #create empty array to flip

for i in range(n_panels):
    Q_n[:,i]=Q[:,i]

for i in range(n_panels):
    for j in range(20):
        Q_n_inv[n_panels-1-i,j] = Q_n[j,i]

Q_n_inv_copy = Q_n_inv

for i in range(n_panels):
    for j in range(20):
        if i%2==1:
            Q_n_inv[i,j]=Q_n_inv_copy[i,19-j]

Q_n_inv_copy = Q_n_inv

#for i in range(10):
    #Q_n_inv[i,:]=Q_n_inv_copy[9-i,:]

Q_n_re = Q_n_inv.reshape(-1)
Q_n_re = Q_n_re*1000 #correcting units
```

```
Q_h = Q_n_re
#print(Q_h)
kelv = 273.15
pi = np.pi

#Dynamic viscosity of solar salt (Pa.s)
def mu_ss(x):
    return 0.075439-(2.77*10**-4)*(x-273.15)+(3.49*10**-7)*(x-273.15)**2-(1.474
*10**-10)*(x-273.15)**3

#Density of solar salt (kg/m^3)
def rho_ss(x):
    return 2263.628-0.636*x

#Thermal conductivity of solar salt (W/m.K)
k_ss = 0.45

#Heat Capacity of solar salt (J/kg.K)
def Cp_ss(x):
    return (1396.044+0.172*x)

#Kinematic viscosity of solar salt()
def v_ss(x):
    return mu_ss(x)/rho_ss(x)

D_o = 21*10**-3 #tubular outer diameter (m)
D_i = D_o-2*1.1*10**-3 #tubular inner diameter (m)
D_mean = (D_o+D_i)/2 #mean diameter of tube (m)
H = 6.2 #m
n_inc_tube = 20
h = H/n_inc_tube #m
B = 22*10**-3 #tubular pitch length (m)
A_i = (np.pi/4*D_i**2)/2 #inner tubular area (m^2)
A_r = h*B*0.5 #area of radiation in front of tubes

T_ss_ave = (T_ss_i+T_ss_e)/2
sigma = 5.67*10**-8 #W/m^2*K^4 (Stefan-Boltzman constant)
T_surr = 0

alpha = 0.95 #Tube absorptivity
e_t = 0.87 #Tube emissivity
k_t = 11.5 #Tube thermal conductivity (W/mK)

#convective losses precalcs
g = 9.81 #m/s
beta = 0.002 #thermal expansion coefficient(1/K)
v_c = 4.765*10**-5 #kinematic viscosity (m^2/s)
alpha_nc = 3.352*10**-6 #thermal diffusity (m^2/s)
lambda_c = 0.04418  #W/mK

Pr_air = 0.704
epsilon = 0.002e-3
#secondary loss coeffs
```

```
k_i = 1
k_e = 0.2
k_bend = 0.16

def delta_P(T_ss,i):
    u_ss = m_dot_ss/rho_ss(T_ss)/A_i #velocity of solar salt
    Re = u_ss*D_i/v_ss(T_ss) #Reynolds number
    f_d = 0.25/(np.log(5.74/Re+epsilon/(3.7*D_i)))**2
    p_pipe = (f_d*h*rho_ss(T_ss)*u_ss**2)/(2*D_i)
    if i%19==0:
      p_sec = (2*k_bend+2*k_i+2*k_e)*(rho_ss(T_ss)*((u_ss**2)/2))
    else:
      p_sec =0
    p_tot = p_pipe+p_sec
    return p_tot


def R_rad(T_t,A_s,F_view_l):
    h_rad = F_view_l*e_t*sigma*(T_t**2+T_surr**2)*(T_t+T_surr)
    R_r = 1/(h_rad*A_s)
    return R_r

def R_conv_ext(T_t,A_s):
    Ra = (g*beta*(abs(T_t-T_amb)*D_o**3))/(alpha_nc*v_c) #Rayleigh number
    Nu_d_nc = (0.825+(0.387*Ra**(1/6))/((1+(0.492/Pr_air)**(9/16))**(8/27)))**2
#Nusselt number for natural convection (Cengel p540)
    h_nc = lambda_c*Nu_d_nc/D_o #natural convection co-efficient
    Re_d_fc = u_air*D_o/v_c #Reynolds number for forced convection
    Nu_d_fc = 0.3+(0.62*(Re_d_fc**0.5)*(Pr_air**(1/3)))/((1+(0.4/Pr_air)**(2/3)
)**0.25) #Nusselt number for forced convection
    h_fc = lambda_c*Nu_d_fc/D_o #forced convection hx co-efficient
    h_conv = (h_nc**3.2+h_fc**3.2)**(1/3.2) #combined hc co-efficient
    R_c = 1/(h_conv*A_s)
    return R_c

def R_conv_ext_back(T_t,A_s):
    Ra = (g*beta*(abs(T_t-T_amb)*D_o**3))/(alpha_nc*v_c) #Rayleigh number
    Nu_d_nc = (0.825+(0.387*Ra**(1/6))/((1+(0.492/Pr_air)**(9/16))**(8/27)))**2
#Nusselt number for natural convection
    h_nc = 2*lambda_c*Nu_d_nc/D_o #natural convection co-efficient
    R_c_back = 1/(h_nc*A_s)
    return R_c_back

def R_conv_in(T_ss,A_s):
    u_ss = m_dot_ss/rho_ss(T_ss)/A_i #velocity of solar salt
    Re = u_ss*D_i/v_ss(T_ss) #Reynolds number
    Pr = mu_ss(T_ss)*Cp_ss(T_ss)/k_ss #Prandtl number
    Nu = 0.023*(Re**0.8)*(Pr**0.4) #Nusselt number
    h_ss = Nu*k_ss/(D_i/2)
    R_c_in = 1/(h_ss*A_s)
    return R_c_in

def R_cond(A):
```

```
    L_c_c = (D_o-D_i)/4 #conductive characterstic length
    R_c = L_c_c/(k_t*A)
    return R_c

def f_tube(T): #iterative solver for tubular temperature profile
    f1 = -Q_ff+(T[0]-T[5])/R_cond(A_ff_o)+(T[0]-T_amb)/R_conv_ext(T[0],A_ff_o)+
(T[0]-T_surr)/R_rad(T[0],A_ff_o,0.99)
    f2 = -Q_f+(T[1]-T[6])/R_cond(A_f_o)+(T[1]-T_amb)/R_conv_ext(T[1],A_f_o)+(T[
1]-T_surr)/R_rad(T[1],A_f_o,0.99)
    f3 = -Q_c+(T[2]-T[7])/R_cond(A_c_o)+(T[2]-T_amb)/R_conv_ext(T[2],A_c_o)
    f4 = -Q_b+(T[3]-T[8])/R_cond(A_b_o)+(T[3]-T_amb)/R_conv_ext_back(T[3],A_b_o
)
    f5 = -Q_bb+(T[4]-T[9])/R_cond(A_bb_o)+(T[4]-T_amb)/R_conv_ext_back(T[4],A_b
b_o)
    f6 = (T[5]-T[0])/R_cond(A_ff_o)+(T[5]-T[10])/R_cond(A_ff)+(T[5]-T[6])/R_cc
    f7 = (T[6]-T[1])/R_cond(A_f_o)+(T[6]-T[11])/R_cond(A_f)+(T[6]-T[5])/R_cc+(T
[6]-T[7])/R_cc
    f8 = (T[7]-T[2])/R_cond(A_c_o)+(T[7]-T[12])/R_cond(A_c)+(T[7]-T[6])/R_cc+(T
[7]-T[8])/R_cc
    f9 = (T[8]-T[3])/R_cond(A_b_o)+(T[8]-T[13])/R_cond(A_b)+(T[8]-T[7])/R_cc+(T
[8]-T[9])/R_cc
    f10 = (T[9]-T[4])/R_cond(A_bb_o)+(T[9]-T[14])/R_cond(A_bb)+(T[9]-T[8])/R_cc
    f11 = (T[10]-T[5])/R_cond(A_ff)+(T[10]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A
_ff_i)
    f12 = (T[11]-T[6])/R_cond(A_f)+(T[11]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_
f_i)
    f13 = (T[12]-T[7])/R_cond(A_c)+(T[12]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_
c_i)
    f14 = (T[13]-T[8])/R_cond(A_b)+(T[13]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_
b_i)
    f15 = (T[14]-T[9])/R_cond(A_bb)+(T[14]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A
_bb_i)
    return [f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15]

L_cc = pi*D_mean/8 #cross conduction length
A_cc = h*(D_o-D_i)/2 #cross conduction area
R_cc = L_cc/(k_t*A_cc)

f_ff = 0.022 #fraction of area ff takes up

#Outer surface area calcs
A_ff_o = pi*D_o*h*f_ff
A_f_o = pi*D_o*h*(0.25-2*f_ff)
A_c_o = pi*D_o*h*2*f_ff
A_b_o = A_f_o
A_bb_o = A_ff_o

#mean inner area calcs
A_ff = pi*D_mean*h*f_ff
A_f = pi*D_mean*h*(0.25-2*f_ff)
A_c = pi*D_mean*h*2*f_ff
A_b = A_f
A_bb = A_ff
```

```
#Inner surface area calcs
A_ff_i = pi*D_i*h*f_ff
A_f_i = pi*D_i*h*(0.25-2*f_ff)
A_c_i = pi*D_i*h*2*f_ff
A_b_i = A_f_i
A_bb_i = A_ff_i

F_view_ff = 0.127
F_view_f = 0.84
F_view_c = 0.006
F_view_b = 0.023
F_view_bb = 0.004

n = 15 #number of nodes
A = np.zeros((n,n))
T = np.zeros(n)
T_guess = np.zeros(n)
Q = np.zeros(n)
R = np.zeros(30)

for i in range(n):
    T_guess[i] = 400+kelv #initial guess such that resistances can be estimated

Q_abs_ss = np.zeros(5)

n_sections = n_panels*n_inc_tube
T_tube = np.zeros((n_sections,n))
T_ss = np.zeros(n_sections+1)
T_ss_ave = np.zeros(n_sections)
T_ss_ave_new = np.zeros(n_sections)
T_ss_ave_err = np.zeros(n_sections)
Q_rad_tot = np.zeros(n_sections)
Q_abs_ss = np.zeros(5)
eta = np.zeros(n_sections)
eta_weighted = np.zeros(n_sections)
Q_abs_ss_tot = np.zeros(n_sections)
p_loss = np.zeros(n_sections)

for i in range(n_sections+1):
    T_ss[i] = T_ss_i

for i in range(n_sections):
    T_ss_ave[i] = T_ss_i
    T_ss_ave_err[i] = 1

for i in range(n_sections):
    while T_ss_ave_err[i]>10**-6:
        Q_ff = Q_h[i]*A_r*F_view_ff
        Q_f = Q_h[i]*A_r*F_view_f
        Q_c = Q_h[i]*A_r*F_view_c
        Q_b = Q_h[i]*A_r*F_view_b
        Q_bb = Q_h[i]*A_r*F_view_bb
```

```
        T_tube[i] = fsolve(f_tube,T_guess)
        Q_abs_ss[0] = (T_tube[i,10]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_ff_i)
        Q_abs_ss[1] = (T_tube[i,11]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_f_i)
        Q_abs_ss[2] = (T_tube[i,12]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_c_i)
        Q_abs_ss[3] = (T_tube[i,13]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_b_i)
        Q_abs_ss[4] = (T_tube[i,14]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_bb_i)
        Q_abs_ss_tot[i] = np.sum(Q_abs_ss)
        Q_rad_tot[i] = Q_h[i]*h*B/2
        eta[i] = Q_abs_ss_tot[i]/Q_rad_tot[i]
        T_ss[i+1] = T_ss[i]+Q_abs_ss_tot[i]/(m_dot_ss*Cp_ss(T_ss[i]))
        T_ss_ave_new[i] = (T_ss[i+1]+T_ss[i])/2
        T_ss_ave_err[i] = np.absolute(T_ss_ave[i]-T_ss_ave_new[i])
        T_ss_ave[i] = T_ss_ave_new[i]
        eta_weighted[i] = eta[i]*Q_abs_ss_tot[i]
        p_loss[i] = delta_P(T_ss[i],i)

T_ss_n = T_ss
T_tube_n = T_tube
eta_n = eta
Q_h_n = Q_h
m_dot_ss_n = m_dot_ss*2*32
Q_ss_tot_n = m_dot_ss_n*Cp_ss((T_ss_n[0]+T_ss_n[239])/2)*(T_ss_n[239]-T_ss_n[0]
)

end_time = time.time()
pred_time = end_time-start_time
w_pump = m_dot_ss*2*32*2*np.sum(p_loss)/(0.6*rho_ss(np.average(T_ss)))

i=10
print('m_dot_ss total =',m_dot_ss*2*32,'kg/s')
print('T_tube_max =',np.max(T_tube)-kelv)
print('T_ss_i =',T_ss[0]-kelv,'deg C')
print('T_ss_e =',T_ss[239]-kelv,'deg C')
print('Flux =',Q_h[i]/10**6,'MW/m^2')
print('Heat transfer absorbed =',Q_abs_ss_tot[i]/1000,'kW')
print('Receiver efficiency =',np.sum(eta_weighted)/np.sum(Q_abs_ss_tot))
print('Time for sim =',pred_time,'s')
print('Pump power =',w_pump/1e6,'MW')
#print(T[10])
m_dot_ss_west = m_dot_ss

m_dot_ss total = 42.5 kg/s
T_tube_max = 623.5062018805099
T_ss_i = 296.0 deg C
T_ss_e = 546.612627105924 deg C
Flux = 0.9186865940000001 MW/m^2
Heat transfer absorbed = 2.898135155517919 kW
Receiver efficiency = 0.9011615173020978
Time for sim = 1.9947304725646973 s
Pump power = 0.02649540646735149 MW

print(np.sum(p_loss)/1000)
```

```
334.3893250830936

u_air = 0.6 #wind speed (m/s)

#Western Panels
n_panels = 12
Q = df.to_numpy() #convert flux df to numpy
Q_n = np.zeros((20,n_panels)) #North facing flux
Q_n_inv = np.zeros((n_panels,20)) #create empty array to flip

for i in range(n_panels):
    Q_n[:,i]=Q[:,12+i]

for i in range(n_panels):
    for j in range(20):
        Q_n_inv[i,j] = Q_n[j,i]

Q_n_inv_copy = Q_n_inv

for i in range(n_panels):
    for j in range(20):
        if i%2==1:
            Q_n_inv[i,j]=Q_n_inv_copy[i,19-j]

Q_n_inv_copy = Q_n_inv

#for i in range(10):
    #Q_n_inv[i,:]=Q_n_inv_copy[9-i,:]

Q_n_re = Q_n_inv.reshape(-1)
Q_n_re = Q_n_re*1000 #correcting units
Q_h = Q_n_re
#print(Q_h)
kelv = 273.15
pi = np.pi

#Dynamic viscosity of solar salt (Pa.s)
def mu_ss(x):
    return 0.075439-(2.77*10**-4)*(x-273.15)+(3.49*10**-7)*(x-273.15)**2-(1.474
*10**-10)*(x-273.15)**3

#Density of solar salt (kg/m^3)
def rho_ss(x):
    return 2263.628-0.636*x

#Thermal conductivity of solar salt (W/m.K)
k_ss = 0.45

#Heat Capacity of solar salt (J/kg.K)
def Cp_ss(x):
    return (1396.044+0.172*x)

#Kinematic viscosity of solar salt()
```

```
def v_ss(x):
    return mu_ss(x)/rho_ss(x)

D_o = 21*10**-3 #tubular outer diameter (m)
D_i = D_o-2*1.1*10**-3 #tubular inner diameter (m)
D_mean = (D_o+D_i)/2 #mean diameter of tube (m)
H = 6.2 #m
n_inc_tube = 20
h = H/n_inc_tube #m
B = 22*10**-3 #tubular pitch length (m)
A_i = (np.pi/4*D_i**2)/2 #inner tubular area (m^2)
A_r = h*B*0.5 #area of radiation in front of tubes

x = 0.95 #angular coefficient
T_ss_ave = (T_ss_i+T_ss_e)/2
sigma = 5.67*10**-8 #W/m^2*K^4 (Stefan-Boltzman constant)
T_surr = 0

alpha = 0.93 #Tube absorptivity
e_t = 0.87 #Tube emissivity
k_t = 11.5 #Tube thermal conductivity (W/mK)

#convective losses precalcs
g = 9.81 #m/s
beta = 0.002 #thermal expansion coefficient(1/K)
v_c = 4.765*10**-5 #kinematic viscosity (m^2/s)
alpha_nc = 3.352*10**-6 #thermal diffusity (m^2/s)
lambda_c = 0.04418  #W/mK

Pr_air = 0.704

def R_rad(T_t,A_s,F_view_l):
    h_rad = F_view_l*e_t*sigma*(T_t**2+T_surr**2)*(T_t+T_surr)
    R_r = 1/(h_rad*A_s)
    return R_r

def R_conv_ext(T_t,A_s):
    Ra = (g*beta*(abs(T_t-T_amb)*D_o**3))/(alpha_nc*v_c) #Rayleigh number
    Nu_d_nc = (0.825+(0.387*Ra**(1/6))/((1+(0.492/Pr_air)**(9/16))**(8/27)))**2
#Nusselt number for natural convection (Cengel p540)
    h_nc = lambda_c*Nu_d_nc/D_o #natural convection co-efficient
    Re_d_fc = u_air*D_o/v_c #Reynolds number for forced convection
    Nu_d_fc = 0.3+(0.62*(Re_d_fc**0.5)*(Pr_air**(1/3)))/((1+(0.4/Pr_air)**(2/3)
)**0.25) #Nusselt number for forced convection
    h_fc = lambda_c*Nu_d_fc/D_o #forced convection hx co-efficient
    h_conv = (h_nc**3.2+h_fc**3.2)**(1/3.2) #combined hc co-efficient
    R_c = 1/(h_conv*A_s)
    return R_c

def R_conv_ext_back(T_t,A_s):
    Ra = (g*beta*(abs(T_t-T_amb)*D_o**3))/(alpha_nc*v_c) #Rayleigh number
    Nu_d_nc = (0.825+(0.387*Ra**(1/6))/((1+(0.492/Pr_air)**(9/16))**(8/27)))**2
#Nusselt number for natural convection
```

```
    h_nc = 2*lambda_c*Nu_d_nc/D_o #natural convection co-efficient
    R_c_back = 1/(h_nc*A_s)
    return R_c_back

def R_conv_in(T_ss,A_s):
    u_ss = m_dot_ss/rho_ss(T_ss)/A_i #velocity of solar salt
    Re = u_ss*D_i/v_ss(T_ss) #Reynolds number
    Pr = mu_ss(T_ss)*Cp_ss(T_ss)/k_ss #Prandtl number
    Nu = 0.0243*(Re**0.8)*(Pr**0.4) #Nusselt number
    h_ss = Nu*k_ss/(D_i/2)
    R_c_in = 1/(h_ss*A_s)
    return R_c_in

def R_cond(A):
    L_c_c = (D_o-D_i)/4 #conductive characterstic length
    R_c = L_c_c/(k_t*A)
    return R_c

def f_tube(T): #iterative solver for tubular temperature profile
    f1 = -Q_ff+(T[0]-T[5])/R_cond(A_ff_o)+(T[0]-T_amb)/R_conv_ext(T[0],A_ff_o)+
(T[0]-T_surr)/R_rad(T[0],A_ff_o,0.99)
    f2 = -Q_f+(T[1]-T[6])/R_cond(A_f_o)+(T[1]-T_amb)/R_conv_ext(T[1],A_f_o)+(T[
1]-T_surr)/R_rad(T[1],A_f_o,0.99)
    f3 = -Q_c+(T[2]-T[7])/R_cond(A_c_o)+(T[2]-T_amb)/R_conv_ext(T[2],A_c_o)
    f4 = -Q_b+(T[3]-T[8])/R_cond(A_b_o)+(T[3]-T_amb)/R_conv_ext_back(T[3],A_b_o
)
    f5 = -Q_bb+(T[4]-T[9])/R_cond(A_bb_o)+(T[4]-T_amb)/R_conv_ext_back(T[4],A_b
b_o)
    f6 = (T[5]-T[0])/R_cond(A_ff_o)+(T[5]-T[10])/R_cond(A_ff)+(T[5]-T[6])/R_cc
    f7 = (T[6]-T[1])/R_cond(A_f_o)+(T[6]-T[11])/R_cond(A_f)+(T[6]-T[5])/R_cc+(T
[6]-T[7])/R_cc
    f8 = (T[7]-T[2])/R_cond(A_c_o)+(T[7]-T[12])/R_cond(A_c)+(T[7]-T[6])/R_cc+(T
[7]-T[8])/R_cc
    f9 = (T[8]-T[3])/R_cond(A_b_o)+(T[8]-T[13])/R_cond(A_b)+(T[8]-T[7])/R_cc+(T
[8]-T[9])/R_cc
    f10 = (T[9]-T[4])/R_cond(A_bb_o)+(T[9]-T[14])/R_cond(A_bb)+(T[9]-T[8])/R_cc
    f11 = (T[10]-T[5])/R_cond(A_ff)+(T[10]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A
_ff_i)
    f12 = (T[11]-T[6])/R_cond(A_f)+(T[11]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_
f_i)
    f13 = (T[12]-T[7])/R_cond(A_c)+(T[12]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_
c_i)
    f14 = (T[13]-T[8])/R_cond(A_b)+(T[13]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_
b_i)
    f15 = (T[14]-T[9])/R_cond(A_bb)+(T[14]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A
_bb_i)
    return [f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15]

L_cc = pi*D_mean/8 #cross conduction length
A_cc = h*(D_o-D_i)/2 #cross conduction area
R_cc = L_cc/(k_t*A_cc)

f_ff = 0.022 #fraction of area ff takes up
```

```
#Outer surface area calcs
A_ff_o = pi*D_o*h*f_ff
A_f_o = pi*D_o*h*(0.25-2*f_ff)
A_c_o = pi*D_o*h*2*f_ff
A_b_o = A_f_o
A_bb_o = A_ff_o

#mean inner area calcs
A_ff = pi*D_mean*h*f_ff
A_f = pi*D_mean*h*(0.25-2*f_ff)
A_c = pi*D_mean*h*2*f_ff
A_b = A_f
A_bb = A_ff

#Inner surface area calcs
A_ff_i = pi*D_i*h*f_ff
A_f_i = pi*D_i*h*(0.25-2*f_ff)
A_c_i = pi*D_i*h*2*f_ff
A_b_i = A_f_i
A_bb_i = A_ff_i

F_view_ff = 0.127
F_view_f = 0.84
F_view_c = 0.006
F_view_b = 0.023
F_view_bb = 0.004

n = 15 #number of nodes
A = np.zeros((n,n))
T = np.zeros(n)
T_guess = np.zeros(n)
Q = np.zeros(n)
R = np.zeros(30)

for i in range(n):
    T_guess[i] = 400+kelv #initial guess such that resistances can be estimated

Q_abs_ss = np.zeros(5)

n_sections = n_panels*n_inc_tube
T_tube = np.zeros((n_sections,n))
T_ss = np.zeros(n_sections+1)
T_ss_ave = np.zeros(n_sections)
T_ss_ave_new = np.zeros(n_sections)
T_ss_ave_err = np.zeros(n_sections)
Q_rad_tot = np.zeros(n_sections)
Q_abs_ss = np.zeros(5)
eta = np.zeros(n_sections)
eta_weighted = np.zeros(n_sections)
Q_abs_ss_tot = np.zeros(n_sections)

for i in range(n_sections+1):
```

```
    T_ss[i] = T_ss_i

for i in range(n_sections):
    T_ss_ave[i] = T_ss_i
    T_ss_ave_err[i] = 1

for i in range(n_sections):
    while T_ss_ave_err[i]>10**-6:
        Q_ff = Q_h[i]*A_r*F_view_ff
        Q_f = Q_h[i]*A_r*F_view_f
        Q_c = Q_h[i]*A_r*F_view_c
        Q_b = Q_h[i]*A_r*F_view_b
        Q_bb = Q_h[i]*A_r*F_view_bb
        T_tube[i] = fsolve(f_tube,T_guess)
        Q_abs_ss[0] = (T_tube[i,10]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_ff_i)
        Q_abs_ss[1] = (T_tube[i,11]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_f_i)
        Q_abs_ss[2] = (T_tube[i,12]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_c_i)
        Q_abs_ss[3] = (T_tube[i,13]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_b_i)
        Q_abs_ss[4] = (T_tube[i,14]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_bb_i)
        Q_abs_ss_tot[i] = np.sum(Q_abs_ss)
        Q_rad_tot[i] = Q_h[i]*h*B/2
        eta[i] = Q_abs_ss_tot[i]/Q_rad_tot[i]
        T_ss[i+1] = T_ss[i]+Q_abs_ss_tot[i]/(m_dot_ss*Cp_ss(T_ss[i]))
        T_ss_ave_new[i] = (T_ss[i+1]+T_ss[i])/2
        T_ss_ave_err[i] = np.absolute(T_ss_ave[i]-T_ss_ave_new[i])
        T_ss_ave[i] = T_ss_ave_new[i]
        eta_weighted[i] = eta[i]*Q_abs_ss_tot[i]
        p_loss[i] = delta_P(T_ss[i],i)

T_ss_s = T_ss
T_tube_s = T_tube
eta_s = eta
Q_h_s = Q_h
m_dot_ss_s = m_dot_ss*2*32
Q_ss_tot_s = m_dot_ss_s*Cp_ss((T_ss_s[0]+T_ss_s[239])/2)*(T_ss_s[239]-T_ss_s[0]
)
Q_inc = np.zeros(240)
Q_inc = Q_h*h*D_o
Q_inc_tot = np.sum(Q_inc)*32

w_pump = m_dot_ss*2*32*2*np.sum(p_loss)/(0.6*rho_ss(np.average(T_ss)))

i=10
print('m_dot_ss total =',m_dot_ss*2*32,'kg/s')
print('T_tube_max =',np.max(T_tube)-kelv)
print('T_ss_i =',T_ss[0]-kelv,'deg C')
print('T_ss_e =',T_ss[239]-kelv,'deg C')
print('Flux =',Q_h[i]/10**6,'MW/m^2')
print('Heat transfer absorbed =',Q_abs_ss_tot[i]/1000,'kW')
print('Receiver efficiency =',np.sum(eta_weighted)/np.sum(Q_abs_ss_tot))
print('Time for sim =',pred_time,'s')
print('Pump power =',w_pump/1e6,'MW')
#print(T[10])
```

```
m_dot_ss total = 42.5 kg/s
T_tube_max = 614.6121826924664
T_ss_i = 296.0 deg C
T_ss_e = 541.950436515495 deg C
Flux = 0.89831796 MW/m^2
Heat transfer absorbed = 2.835663460146784 kW
Receiver efficiency = 0.9114695987984447
Time for sim = 1.9947304725646973 s
Pump power = 0.026470858324947992 MW
```

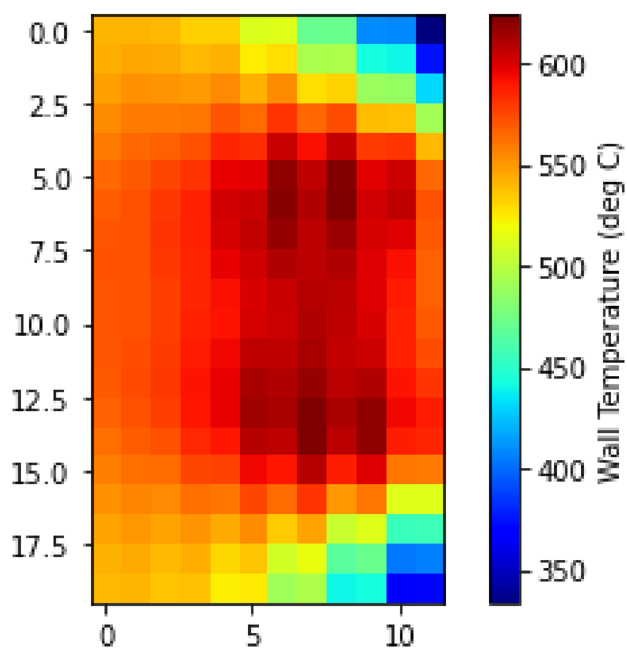
```
m_dot_total = m_dot_ss_s+m_dot_ss_n
m_dot_ex = 85
Q_ex = m_dot_ex*Cp_ss((T_ss_e+T_ss_i)/2)*(T_ss_e-T_ss_i)
T_ss_e_ave = ((T_ss_n[239])*m_dot_ss_n+(T_ss_s[239])*m_dot_ss_s)/(m_dot_ss_n+m_
dot_ss_s)
err_m_dot = (np.absolute(m_dot_total-m_dot_ex)/m_dot_ex)*100
err_T = np.absolute((T_ss_e_ave-T_ss_e))/T_ss_e*100
Q_abs_real = Q_ss_tot_s+Q_ss_tot_n
err_Q = (np.absolute(Q_abs_real-Q_ex)/Q_ex)*100
print('Mass flow rate:',round(m_dot_total,3),'kg/s')
print('Percentage error m_dot:',round(err_m_dot,3),'%')
print('Expected outlet temperature:',T_ss_e-kelv,'deg C')
print('Actual outlet temperature:',round(T_ss_e_ave-kelv,3),'deg C')
print('Percentage error T:',round(err_T,3),'%')
print('Expected heat absorption:',round(Q_ex/10**6,3),'MW')
print('Actual heat absorption:',round(Q_abs_real/10**6,3),'MW')
print('Percentage error Q:',round(err_Q,3),'%')
```

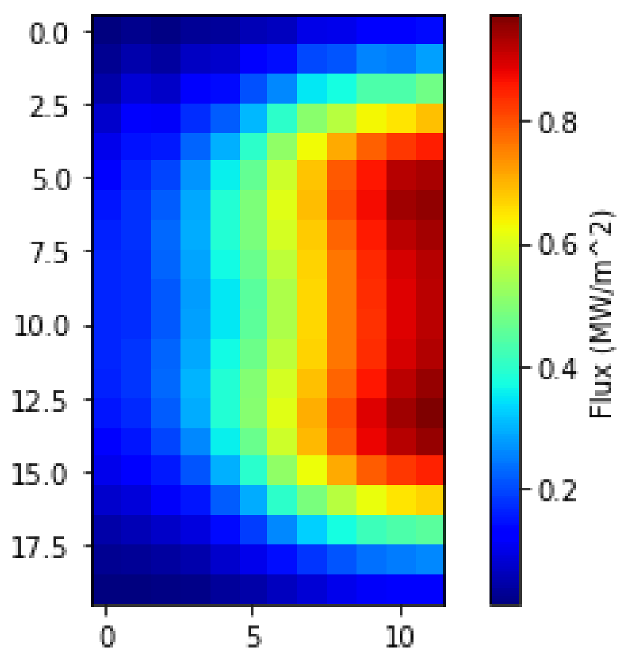
```
Mass flow rate: 85.0 kg/s
Percentage error m_dot: 0.0 %
Expected outlet temperature: 553.0 deg C
Actual outlet temperature: 544.282 deg C
Percentage error T: 1.055 %
Expected heat absorption: 33.118 MW
Actual heat absorption: 31.979 MW
Percentage error Q: 3.44 %
```

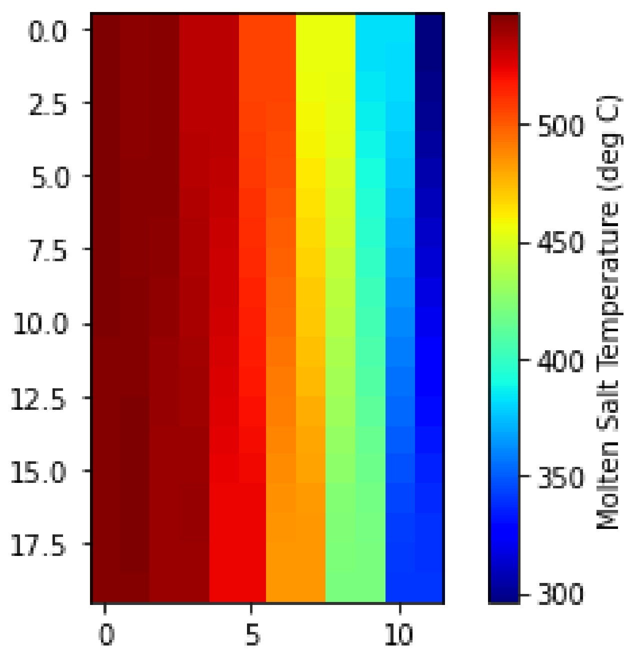
```
def heatmap2d_wall(arr: np.ndarray):
    plt.imshow(arr, cmap='jet')
    plt.colorbar(label = 'Wall Temperature (deg C)')
    plt.show()

def heatmap2d_ss(arr: np.ndarray):
    plt.imshow(arr, cmap='jet')
    plt.colorbar(label = 'Molten Salt Temperature (deg C)')
    plt.show()

def heatmap2d_eta(arr: np.ndarray):
    plt.imshow(arr, cmap='jet')
    plt.colorbar(label = 'Increment Efficiency')
    plt.show()

def heatmap2d_flux(arr: np.ndarray):
```

```
    plt.imshow(arr, cmap='jet')
    plt.colorbar(label = 'Flux (MW/m^2)')
    plt.show()

#print(T_tube[10]-kelv)
T_ss_map = np.zeros((12,20))
T_ss_map_flip = np.zeros((20,12))
T_ss_map_flip_copy = np.zeros((20,12))
T_ss_new = np.zeros(240)
T_tube_max_map = np.zeros((12,20))
T_tube_map_flip = np.zeros((20,12))
T_tube_map_flip_copy = np.zeros((20,12))
eta_map = np.zeros((12,20))
eta_map_flip = np.zeros((20,12))
eta_map_flip_copy = np.zeros((20,12))
Q_n_map = np.zeros((12,20))
Q_n_map_flip = np.zeros((20,12))

for i in range(240):
    T_ss_new[i]=T_ss_n[i+1]-kelv

T_tube_max_map = np.reshape(T_tube_n[:,0]-kelv,(12,20))
T_ss_map = np.reshape(T_ss_new,(12,20))
eta_map = np.reshape(eta_n,(12,20))
Q_n_map = np.reshape(Q_h_n/10**6,(12,20))

for i in range(12):
    for j in range(20):
        T_ss_map_flip[j,11-i] = T_ss_map[i,j]
        T_ss_map_flip_copy[j,11-i] = T_ss_map[i,j]
        T_tube_map_flip[j,11-i] = T_tube_max_map[i,j]
        T_tube_map_flip_copy[j,11-i] = T_tube_max_map[i,j]
        eta_map_flip[j,11-i] = eta_map[i,j]
        Q_n_map_flip[j,11-i] = Q_n_map[i,j]

#print(T_ss_map_flip)
# T_ss_map_flip_copy = T_ss_map_flip
#print(T_ss_map_flip_copy[:,0])

for j in range(12):
    if j%2 == 0:
        for i in range(20):
            T_ss_map_flip[19-i][j]=T_ss_map_flip_copy[i][j]
            T_tube_map_flip[19-i][j]=T_tube_map_flip_copy[i][j]

#print(T_ss_map_flip_copy[:,0])
#print(T_ss_map_flip[:,0])

T_tube_map_flip_n = T_tube_map_flip
Q_n_map_flip_n = Q_n_map_flip
T_ss_map_flip_n = T_ss_map_flip
eta_map_flip_n = eta_map_flip
```

```
heatmap2d_wall(T_tube_map_flip)
heatmap2d_flux(Q_n_map_flip)
heatmap2d_ss(T_ss_map_flip)
```

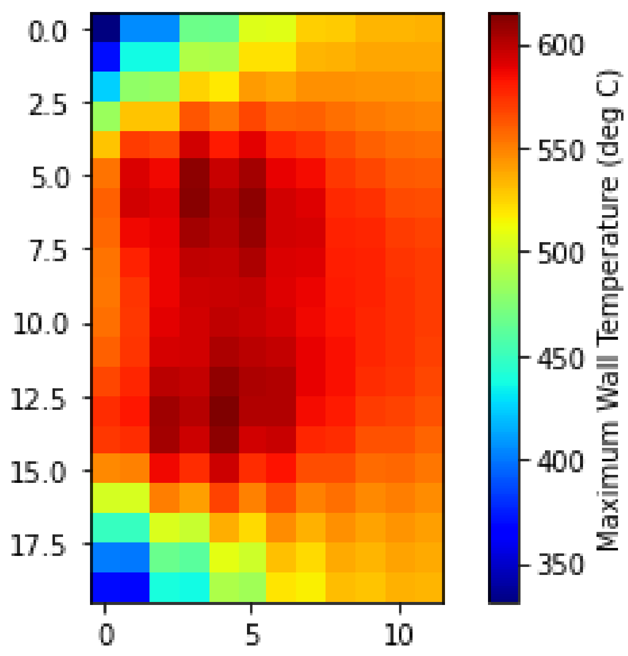




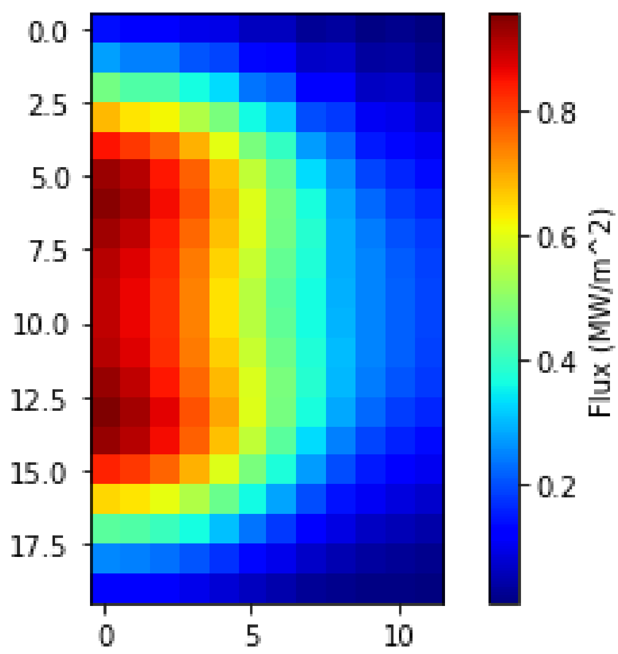



```
def heatmap2d_wall(arr: np.ndarray):
    plt.imshow(arr, cmap='jet')
    plt.colorbar(label = 'Maximum Wall Temperature (deg C)')
    plt.show()

def heatmap2d_ss(arr: np.ndarray):
    plt.imshow(arr, cmap='jet')
    plt.colorbar(label = 'Molten Salt Temperature (deg C)')
    plt.show()

def heatmap2d_eta(arr: np.ndarray):
    plt.imshow(arr, cmap='jet')
    plt.colorbar(label = 'Increment Efficiency')
    plt.show()

def heatmap2d_flux(arr: np.ndarray):
    plt.imshow(arr, cmap='jet')
    plt.colorbar(label = 'Flux (MW/m^2)')
    plt.show()

T_ss_map = np.zeros((12,20))
T_ss_map_flip = np.zeros((20,12))
T_ss_map_flip_copy = np.zeros((20,12))
T_ss_new = np.zeros(240)
T_tube_max_map = np.zeros((12,20))
T_tube_map_flip = np.zeros((20,12))
T_tube_map_flip_copy = np.zeros((20,12))
eta_map = np.zeros((12,20))
eta_map_flip = np.zeros((20,12))
eta_map_flip_copy = np.zeros((20,12))
Q_s_map = eta_map = np.zeros((10,22))
Q_s_map_flip = np.zeros((20,12))
```

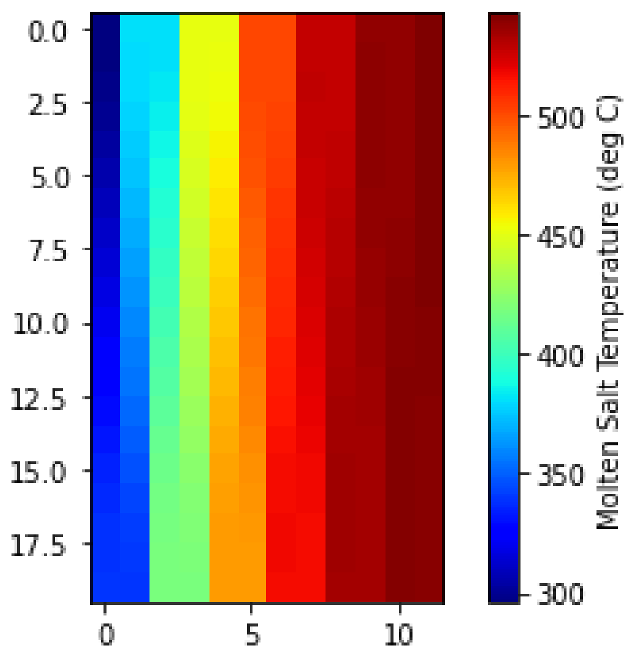
```
for i in range(240):
    T_ss_new[i]=T_ss[i+1]-kelv

T_tube_max_map = np.reshape(T_tube_s[:,0]-kelv,(12,20))
T_ss_map = np.reshape(T_ss_new,(12,20))
eta_map = np.reshape(eta,(12,20))
Q_s_map = np.reshape(Q_h/10**6,(12,20))

for i in range(12):
    for j in range(20):
        T_ss_map_flip[j,i] = T_ss_map[i,j]
        T_ss_map_flip_copy[j,i] = T_ss_map[i,j]
        T_tube_map_flip[j,i] = T_tube_max_map[i,j]
        T_tube_map_flip_copy[j,i] = T_tube_max_map[i,j]
        eta_map_flip[j,i] = eta_map[i,j]
        Q_s_map_flip[j,i] = Q_s_map[i,j]

#print(T_ss_map_flip)
# T_ss_map_flip_copy = T_ss_map_flip
#print(T_ss_map_flip_copy[:,0])

for j in range(12):
    if j%2 == 1:
        for i in range(20):
            T_ss_map_flip[19-i][j]=T_ss_map_flip_copy[i][j]
            T_tube_map_flip[19-i][j]=T_tube_map_flip_copy[i][j]



T_tube_map_flip_s = T_tube_map_flip
Q_n_map_flip_s = Q_s_map_flip
T_ss_map_flip_s = T_ss_map_flip
eta_map_flip_s = eta_map_flip

heatmap2d_wall(T_tube_map_flip)
heatmap2d_flux(Q_s_map_flip)
heatmap2d_ss(T_ss_map_flip)
#print(T_tube_map_flip_s)
```



0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
5
10
600
550
500
450
400
350
Maximum Wall Temperature (deg C)

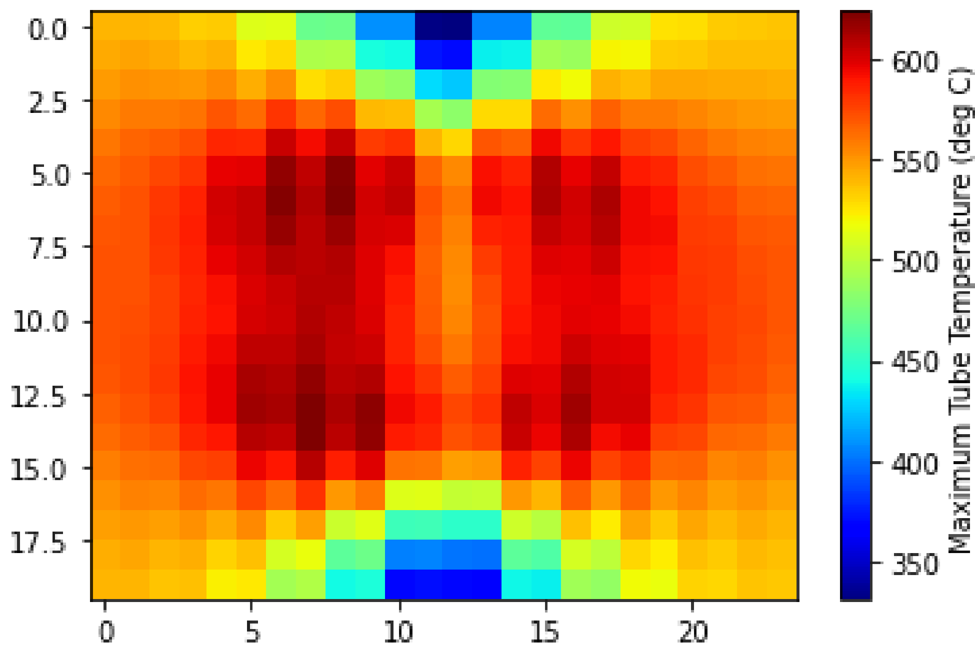


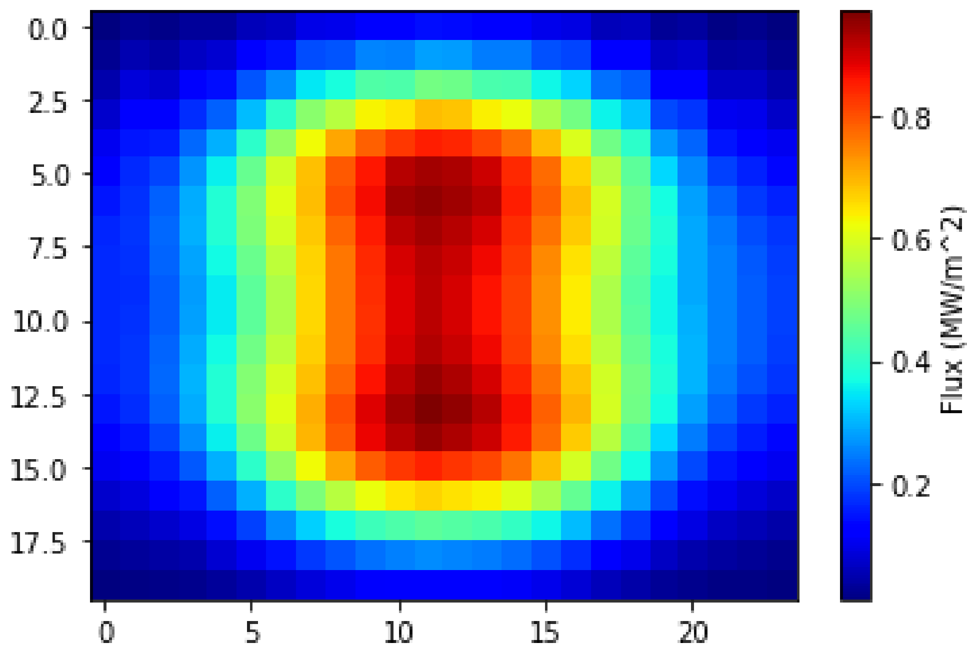
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
5
10
0.8
0.6
0.4
0.2
Flux (MW/m^2)

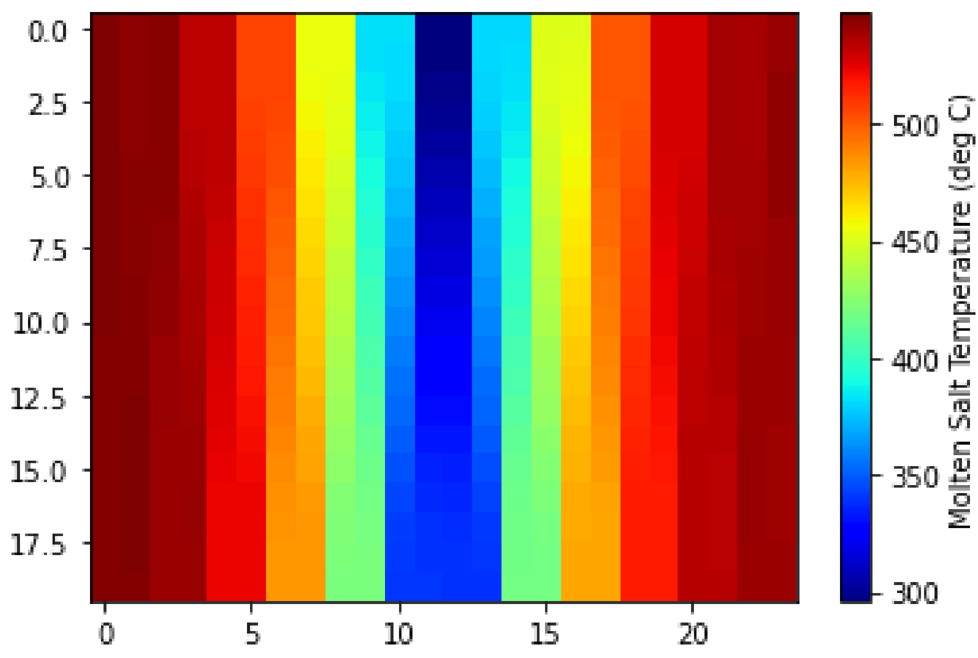



```
def heatmap2d_wall(arr: np.ndarray):
    plt.imshow(arr, cmap='jet')
    plt.colorbar(label = 'Maximum Tube Temperature (deg C)')
    #plt.xticks(x_lab)
    #plt.clim(450,950)
    plt.show()

def heatmap2d_ss(arr: np.ndarray):
    plt.imshow(arr, cmap='jet')
    plt.colorbar(label = 'Molten Salt Temperature (deg C)')
    plt.show()

def heatmap2d_eta(arr: np.ndarray):
    plt.imshow(arr, cmap='jet')
    plt.colorbar(label = 'Increment Efficiency')
    plt.show()

def heatmap2d_flux(arr: np.ndarray):
    plt.imshow(arr, cmap='jet')
    plt.colorbar(label = 'Flux (MW/m^2)')
    #plt.clim(0,2)
    plt.show()

T_tube_map_flip_comb = np.zeros((20,24))
Q_n_map_flip_comb = np.zeros((20,24))
T_ss_map_flip_comb = np.zeros((20,24))
eta_map_flip_comb = np.zeros((20,24))

for i in range(12):
    T_tube_map_flip_comb[:,i] = T_tube_map_flip_n[:,i]
    T_tube_map_flip_comb[:,12+i] = T_tube_map_flip_s[:,i]
    Q_n_map_flip_comb[:,i] = Q_n_map_flip_n[:,i]
    Q_n_map_flip_comb[:,12+i] = Q_n_map_flip_s[:,i]
```

```
    T_ss_map_flip_comb[:,i] = T_ss_map_flip_n[:,i]
    T_ss_map_flip_comb[:,12+i] = T_ss_map_flip_s[:,i]
    eta_map_flip_comb[:,i] = eta_map_flip_n[:,i]
    eta_map_flip_comb[:,12+i] = eta_map_flip_s[:,i]

heatmap2d_wall(T_tube_map_flip_comb)
heatmap2d_flux(Q_n_map_flip_comb)
heatmap2d_ss(T_ss_map_flip_comb)
```






0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
5
10
15
20
500
450
400
350
300
Molten Salt Temperature (deg C)

Appendix C. CoPylot script

```
import matplotlib.pyplot as plt
from copylot import CoPylot
import pandas as pd
import numpy as np

x_res = 20 #resolution of flux profile (fixed)
y_res = 40 #resolution of flux profile (fixed)
df = pd.read_csv("LHS_500_cases_v3.csv") #import csv
df_np = df.to_numpy() #convert df to numpy
n_cases = len(df_np) #number of cases to simulate
flux_combined = np.zeros((n_cases*y_res,x_res)) #empty array to store flux prof
iles

## Minimum working example -> Must update path to weather file
cp = CoPylot()
r = cp.data_create()
assert cp.data_set_string(
        r,
        "ambient.0.weather_file",
        "../climate_files/South Africa RSA Upington (INTL).csv",
    )
assert cp.generate_layout(r)
field = cp.get_layout_info(r)
assert cp.data_set_number(r, "fluxsim.0.x_res", x_res)
assert cp.data_set_number(r, "fluxsim.0.y_res", y_res)

for i in range(n_cases):
    q_des = df_np[i,9] #design power (MW)
    th = df_np[i,10] #tower height (m)
    max_flux = df_np[i,11] #maximum allowable flux (MW/m^2)
    h_d = df_np[i,12] #height to diameter ratio
    a_sr = q_des/max_flux #area of solar receiver (m^2)
    d_sr = (a_sr/(1.5*np.pi))**0.5 #diameter of solar receiver
    h_sr = h_d*d_sr #height of solar receiver

    #assert geometrical conditions to generate heliostat field layout
    assert cp.data_set_number(r,"receiver.0.rec_diameter",d_sr)
    assert cp.data_set_number(r,"receiver.0.rec_height", h_sr)
    assert cp.data_set_number(r,"solarfield.0.q_des", q_des)
    assert cp.data_set_number(r,"receiver.0.peak_flux",max_flux)

    assert cp.generate_layout(r) #generate heliostatfield layout
    field = cp.get_layout_info(r)
    assert cp.data_set_number(r, "fluxsim.0.x_res", x_res)
    assert cp.data_set_number(r, "fluxsim.0.y_res", y_res)
    day = int(df_np[i,1])
    month = int(df_np[i,2])
    hour = int(df_np[i,3])
    DNI = df_np[i,4]
```

```
        el = df_np[i,5]
        az = df_np[i,6]
        assert cp.data_set_number(r, "fluxsim.0.flux_dni", DNI)
        assert cp.data_set_number(r, "fluxsim.0.flux_solar_el_in", el)
        assert cp.data_set_number(r, "fluxsim.0.flux_solar_az_in", az)
        assert cp.simulate(r)
        flux = cp.get_fluxmap(r)
        flux_combined[i*y_res:(i+1)*y_res,0:x_res] = flux
#print(flux)
assert cp.data_free(r)

# Plotting (default) solar field and flux map
# Solar Field
#plt.scatter(field['x_location'], field['y_location'], s=1.5)
#plt.tight_layout()
#plt.show()

# flux
#im = plt.imshow(flux)
#plt.colorbar(im)
#plt.tight_layout()
#plt.show()

df1 = pd.DataFrame(flux_combined)
print(df1)
df1.to_csv("Combined_flux_500_cases_v4.csv")

                0           1           2           3           4           5  \
0      271.753226  276.652510  266.905237  253.568468  234.974711  211.586611
1      337.772034  343.856961  332.053465  316.961512  295.277546  266.703576
2      416.013578  423.450459  409.267884  392.266818  367.105727  332.480681
3      506.901508  515.825202  498.964053  479.893588  450.886158  409.401644
4      610.199320  620.697474  600.910101  579.605603  546.429679  497.418704
...           ...         ...         ...         ...         ...         ...
19995  716.951207  722.934593  707.215208  684.301659  656.018102  611.846351
19996  575.870264  580.805954  567.839049  548.337531  524.791929  489.577126
19997  445.370793  449.352683  439.069846  422.980818  403.859539  376.896231
19998  333.286362  336.436487  328.544818  315.609951  300.339312  280.248389
19999  243.327974  245.779613  239.857100  229.637064  217.519944  202.668646

                6           7           8           9          10          11  \
0      189.065373  167.735789  149.688094  140.251458  133.415721  125.451436
1      238.614160  211.146944  187.911601  176.197256  168.317959  159.238814
2      297.793631  262.977182  233.621570  219.152850  210.055100  199.802171
3      367.129477  323.833154  287.509002  269.745401  259.190125  247.716633
4      446.718128  394.034280  350.079180  328.425686  316.090636  303.360508
...           ...         ...         ...         ...         ...         ...
19995  542.147830  496.291157  451.023183  412.980239  389.231111  384.584571
19996  433.640563  396.215224  361.801582  333.127305  313.691556  308.850189
19997  334.475044  305.439514  280.713681  260.626107  245.528389  240.819158
19998  249.625717  228.064620  211.170142  198.124775  186.987718  182.697879
```

```
19999  181.323687  165.807572  154.684498  146.823633  138.999522  135.317340

               12          13          14          15          16          17
\
0      121.363781  125.184674  141.456751  169.420693  194.279634  220.285663
1      154.509443  159.029137  179.409362  215.104102  246.007122  277.686021
2      194.400669  199.754646  225.019447  269.854041  307.888766  346.140065
3      241.647331  248.023213  278.957150  334.219654  380.420263  426.069898
4      296.677715  304.332600  341.671156  408.371000  463.603723  517.312835
...           ...         ...         ...         ...         ...         ...
19995  398.271059  422.102857  467.686970  520.601735  578.579544  628.773621
19996  320.956093  338.927795  373.351773  414.856375  461.961111  502.710118
19997  250.819972  263.234803  287.537138  318.425792  354.731511  386.415672
19998  190.366948  198.118694  214.153193  236.115922  262.887067  286.752975
19999  140.732378  145.001847  154.929395  170.047811  189.224972  206.948049

               18          19
0      242.650905  257.360504
1      304.074150  320.482660
2      377.091421  395.354593
3      462.088061  482.407043
4      558.817153  581.444860
...           ...         ...
19995  662.856287  683.348422
19996  530.309734  548.259012
19997  408.222519  423.646708
19998  303.792533  316.834034
19999  220.322212  231.217992

[20000 rows x 20 columns]
```

```
import matplotlib.pyplot as plt
from copylot import CoPylot
import pandas as pd
import numpy as np

x_res = 20 #resolution of flux profile (fixed)
y_res = 40 #resolution of flux profile (fixed)
n_cases = 100 #number of cases to simulate

df = pd.read_csv("Combined_flux_2000_cases_v4.csv") #import csv
flux_combined = df.to_numpy() #convert flux df to numpy

flux_combined_edit = np.zeros((y_res*n_cases,x_res)) #create array to store flu
x data
flux_combined_edit = flux_combined[0:y_res*n_cases,1:x_res+1] #remove row indic
ators from flux data
flux_vector = np.zeros(n_cases*x_res*y_res)
flux_3D = np.zeros((n_cases,x_res,y_res))
#flux_vector = np.ndarray.flatten(flux_combined_edit)
for i in range(n_cases):
    for j in range(x_res):
        for k in range(y_res):
```

```
            flux_3D[i,j,k] = flux_combined_edit[20*i+k,j]
#print(flux_3D)
flux_vector = np.ndarray.flatten(flux_3D)

print(flux_vector)
df1 = pd.DataFrame(flux_vector,columns=["Flux"])
print(df1)
df1.to_csv("Flux_vector_2000_cases_v5.csv")

[ 184.98298374  228.30846148  279.66983474 ... 1435.58588837 1427.11727979
 1421.94281852]
              Flux
0       184.982984
1       228.308461
2       279.669835
3       339.186519
4       406.416922
...            ...
79995  1462.816825
79996  1447.616863
79997  1435.585888
79998  1427.117280
79999  1421.942819

[80000 rows x 1 columns]
```

Appendix D. Solar angle calculation

```
import numpy as np
N_day = 180 #day of the year Jan 1 = day 1
LCT = 9 #local clock time

def solar_angles(N_day,LCT): #from https://www.powerfromthesun.net/Book/chapter
03/chapter03.html
    long = 21.09699 #longitude
    long_merid = 30 #longitude of meridian for timezone
    LC = (long_merid-long)/15 #Longitudinal correction factor
    lat = np.radians(-28.482431) #latitude
    phi = lat #latitude in radians
    B = np.radians((360*(N_day-1)/365.242)) #angle for EOT
    EOT = 0.258*np.cos(B)-7.416*np.sin(B)-3.468*np.cos(2*B)-9.228*np.sin(2*B) #
Eqn of time
    t_s = LCT+EOT/60-LC #solar time
    omega = np.radians(15*(t_s-12)) #hour angle
    delta = 0.006918-0.399912*np.cos(B)+0.070257*np.sin(B)-0.006758*np.cos(2*B)
+0.000907*np.sin(2*B)-0.002697*np.cos(3*B)+0.00148*np.sin(3*B)#declination angl
e
    zenith = np.arccos(np.cos(phi)*np.cos(delta)*np.cos(omega)+np.sin(phi)*np.s
in(delta)) #elavtion angle
    el = 90-np.degrees(zenith) #elavtion angle in degrees
    alpha = np.radians(el) #elavation angle in radians
    az_star = np.arccos((np.sin(delta)*np.cos(phi)-np.cos(delta)*np.cos(omega)*
np.sin(phi))/np.cos(alpha)) #placeholder for azimuth angle
    if omega > 0: #test to see which quardant azimuth lies in
        az = 360-np.degrees(az_star)
    elif omega <= 0:
        az = np.degrees(az_star)
    return el, az

el,az = solar_angles(N_day,LCT)

print("Elevation angle =", el)
print("Azimuth angle =", az)

Elevation angle = 16.1673754623899
Azimuth angle = 51.26655786087524
```

Appendix E. TSM training data script

```
import numpy as np
import scipy as sp
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import fsolve
import time

df = pd.read_csv("LHS_tube_16000_cases_v2.csv") #import csv
df_np = df.to_numpy() #convert df to numpy
n_cases = len(df_np) #number of cases to simulate
print(n_cases)

16000

kelv = 273.15

T_amb = 32+kelv #K
T_ss_i = 296+kelv #Solar Salt Initial Temperature (K)
T_surr = 0
m_dot_ss= 85/2/32/2
u_air = 0.6 #wind speed (m/s)
pi = np.pi

#Dynamic viscosity of solar salt (Pa.s)
def mu_ss(x):
    return 0.075439-(2.77*10**-4)*(x-273.15)+(3.49*10**-7)*(x-273.15)**2-(1.474
*10**-10)*(x-273.15)**3

#Density of solar salt (kg/m^3)
def rho_ss(x):
    return 2263.628-0.636*x

#Thermal conductivity of solar salt (W/m.K)
k_ss = 0.45

#Heat Capacity of solar salt (J/kg.K)
def Cp_ss(x):
    return (1396.044+0.172*x)

#Kinematic viscosity of solar salt()
def v_ss(x):
    return mu_ss(x)/rho_ss(x)

# D_o = 42.2*10**-3 #tubular outer diameter (m)
# D_i = 38.91*10**-3 #tubular inner diameter (m)
# D_mean = (D_o+D_i)/2 #mean diameter of tube (m)
# H = 6.2 #m
# n_inc_tube = 40
# h = H/n_inc_tube #m
# B = 44.2*10**-3 #tubular pitch length (m)
```

```
# A_i = np.pi/4*D_i**2 #inner tubular area (m^2)
# A_r = h*B*0.5 #area of radiation in front of tubes

x = 0.95 #angular coefficient
T_ss_ave = T_ss_i
sigma = 5.67*10**-8 #W/m^2*K^4 (Stefan-Boltzman constant)
T_surr = 0

alpha = 0.93 #Tube absorptivity
e_t = 0.87 #Tube emissivity
k_t = 11.5 #Tube thermal conductivity (W/mK)

#convective losses precalcs
g = 9.81 #m/s
beta = 0.002 #thermal expansion coefficient(1/K)
v_c = 4.765*10**-5 #kinematic viscosity (m^2/s)
alpha_nc = 3.352*10**-6 #thermal diffusity (m^2/s)
lambda_c = 0.04418  #W/mK
Pr_air = 0.704

def R_rad(T_t,A_s,F_view_l):
    h_rad = F_view_l*e_t*sigma*(T_t**2+T_surr**2)*(T_t+T_surr)
    R_r = 1/(h_rad*A_s)
    return R_r

def R_conv_ext(T_t,A_s):
    Ra = (g*beta*(abs(T_t-T_amb)*D_o**3))/(alpha_nc*v_c) #Rayleigh number
    Nu_d_nc = (0.825+(0.387*Ra**(1/6))/((1+(0.492/Pr_air)**(9/16))**(8/27)))**2
#Nusselt number for natural convection (Cengel p540)
    h_nc = lambda_c*Nu_d_nc/D_o #natural convection co-efficient
    Re_d_fc = u_air*D_o/v_c #Reynolds number for forced convection
    Nu_d_fc = 0.3+(0.62*(Re_d_fc**0.5)*(Pr_air**(1/3)))/((1+(0.4/Pr_air)**(2/3)
)**0.25) #Nusselt number for forced convection
    h_fc = lambda_c*Nu_d_fc/D_o #forced convection hx co-efficient
    h_conv = (h_nc**3.2+h_fc**3.2)**(1/3.2) #combined hc co-efficient
    R_c = 1/(h_conv*A_s)
    return R_c

def R_conv_ext_back(T_t,A_s):
    Ra = (g*beta*(abs(T_t-T_amb)*D_o**3))/(alpha_nc*v_c) #Rayleigh number
    Nu_d_nc = (0.825+(0.387*Ra**(1/6))/((1+(0.492/Pr_air)**(9/16))**(8/27)))**2
#Nusselt number for natural convection
    h_nc = 2*lambda_c*Nu_d_nc/D_o #natural convection co-efficient
    R_c_back = 1/(h_nc*A_s)
    return R_c_back

def R_conv_in(T_ss,A_s):
    u_ss = m_dot_ss/rho_ss(T_ss)/A_i #velocity of solar salt
    Re = u_ss*D_i/v_ss(T_ss) #Reynolds number
    Pr = mu_ss(T_ss)*Cp_ss(T_ss)/k_ss #Prandtl number
    Nu = 0.0243*(Re**0.8)*(Pr**0.4) #Nusselt number
    h_ss = Nu*k_ss/(D_i/2)
    R_c_in = 1/(h_ss*A_s)
```

```
    return R_c_in

def R_cond(A):
    L_c_c = (D_o-D_i)/4 #conductive characterstic length
    R_c = L_c_c/(k_t*A)
    return R_c

def f_tube(T): #iterative solver for tubular temperature profile
    f1 = -Q_ff+(T[0]-T[5])/R_cond(A_ff_o)+(T[0]-T_amb)/R_conv_ext(T[0],A_ff_o)+
(T[0]-T_surr)/R_rad(T[0],A_ff_o,0.99)
    f2 = -Q_f+(T[1]-T[6])/R_cond(A_f_o)+(T[1]-T_amb)/R_conv_ext(T[1],A_f_o)+(T[
1]-T_surr)/R_rad(T[1],A_f_o,0.99)
    f3 = -Q_c+(T[2]-T[7])/R_cond(A_c_o)+(T[2]-T_amb)/R_conv_ext(T[2],A_c_o)
    f4 = -Q_b+(T[3]-T[8])/R_cond(A_b_o)+(T[3]-T_amb)/R_conv_ext_back(T[3],A_b_o
)
    f5 = -Q_bb+(T[4]-T[9])/R_cond(A_bb_o)+(T[4]-T_amb)/R_conv_ext_back(T[4],A_b
b_o)
    f6 = (T[5]-T[0])/R_cond(A_ff_o)+(T[5]-T[10])/R_cond(A_ff)+(T[5]-T[6])/R_cc
    f7 = (T[6]-T[1])/R_cond(A_f_o)+(T[6]-T[11])/R_cond(A_f)+(T[6]-T[5])/R_cc+(T
[6]-T[7])/R_cc
    f8 = (T[7]-T[2])/R_cond(A_c_o)+(T[7]-T[12])/R_cond(A_c)+(T[7]-T[6])/R_cc+(T
[7]-T[8])/R_cc
    f9 = (T[8]-T[3])/R_cond(A_b_o)+(T[8]-T[13])/R_cond(A_b)+(T[8]-T[7])/R_cc+(T
[8]-T[9])/R_cc
    f10 = (T[9]-T[4])/R_cond(A_bb_o)+(T[9]-T[14])/R_cond(A_bb)+(T[9]-T[8])/R_cc
    f11 = (T[10]-T[5])/R_cond(A_ff)+(T[10]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A
_ff_i)
    f12 = (T[11]-T[6])/R_cond(A_f)+(T[11]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_
f_i)
    f13 = (T[12]-T[7])/R_cond(A_c)+(T[12]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_
c_i)
    f14 = (T[13]-T[8])/R_cond(A_b)+(T[13]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_
b_i)
    f15 = (T[14]-T[9])/R_cond(A_bb)+(T[14]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A
_bb_i)
    return [f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15]

F_view_ff = 0.127
F_view_f = 0.84
F_view_c = 0.006
F_view_b = 0.0023
F_view_bb = 0.004

n = 15 #number of nodes
A = np.zeros((n,n))
T = np.zeros(n)
T_guess = np.zeros(n)
Q = np.zeros(n)
R = np.zeros(30)

for i in range(n):
    T_guess[i] = 400+kelv #initial guess such that resistances can be estimated
```

```
Q_abs_ss = np.zeros(5)

n_sections = n_cases
T_tube = np.zeros((n_sections,n))
T_ss = np.zeros(n_sections+1)
T_ss_i = np.zeros(n_sections)
T_ss_e = np.zeros(n_sections)
T_ss_ave = np.zeros(n_sections)
T_ss_ave_new = np.zeros(n_sections)
T_ss_ave_err = np.zeros(n_sections)
Q_rad_tot = np.zeros(n_sections)
Q_abs_ss = np.zeros(5)
eta = np.zeros(n_sections)
eta_weighted = np.zeros(n_sections)
Q_abs_ss_tot = np.zeros(n_sections)

for i in range(n_sections):
    T_ss_ave[i] = 500
    T_ss_ave_err[i] = 1

Q_h = df_np[:,1]
T_ss_i_ar = df_np[:,2]
u_air_ar = df_np[:,3]
T_amb_ar = df_np[:,4]
m_dot_ss_ar= df_np[:,5]
H_rec = df_np[:,6]
T_surr = 0
#print(Q_h)
pred_start = time.time()
for i in range(n_cases):
    D_o = 42.2*10**-3 #tubular outer diameter (m)
    D_i = 38.9*10**-3 #tubular inner diameter (m)
    D_mean = (D_o+D_i)/2 #mean diameter of tube (m)
    H = H_rec[i]#m
    n_inc_tube = 40
    h = H/n_inc_tube #m
    B = 44.2*10**-3 #tubular pitch length (m)
    A_i = np.pi/4*D_i**2 #inner tubular area (m^2)
    A_r = h*B*0.5 #area of radiation in front of tubes
    T_amb = T_amb_ar[i]
    u_air = u_air_ar[i]
    m_dot_ss = m_dot_ss_ar[i]
    T_ss[i] = T_ss_i_ar[i]
    L_cc = pi*D_mean/8 #cross conduction length
    A_cc = h*(D_o-D_i)/2 #cross conduction area
    R_cc = L_cc/(k_t*A_cc)

    f_ff = 0.022 #fraction of area ff takes up

    #Outer surface area calcs
    A_ff_o = pi*D_o*h*f_ff
    A_f_o = pi*D_o*h*(0.25-2*f_ff)
    A_c_o = pi*D_o*h*2*f_ff
```

```
    A_b_o = A_f_o
    A_bb_o = A_ff_o

    #mean inner area calcs
    A_ff = pi*D_mean*h*f_ff
    A_f = pi*D_mean*h*(0.25-2*f_ff)
    A_c = pi*D_mean*h*2*f_ff
    A_b = A_f
    A_bb = A_ff

    #Inner surface area calcs
    A_ff_i = pi*D_i*h*f_ff
    A_f_i = pi*D_i*h*(0.25-2*f_ff)
    A_c_i = pi*D_i*h*2*f_ff
    A_b_i = A_f_i
    A_bb_i = A_ff_i
    A_i = np.pi/4*D_i**2 #inner tubular area (m^2)
    A_r = h*B*0.5 #area of radiation in front of tubes
    A_ff_o = pi*D_o*h*f_ff
    A_f_o = pi*D_o*h*(0.25-2*f_ff)
    A_c_o = pi*D_o*h*2*f_ff
    A_b_o = A_f_o
    A_bb_o = A_ff_o
    #print(T_amb)
    while T_ss_ave_err[i]>10**-6:
        Q_ff = Q_h[i]*A_r*F_view_ff
        Q_f = Q_h[i]*A_r*F_view_f
        Q_c = Q_h[i]*A_r*F_view_c
        Q_b = Q_h[i]*A_r*F_view_b
        Q_bb = Q_h[i]*A_r*F_view_bb
        T_tube[i] = fsolve(f_tube,T_guess)
        Q_abs_ss[0] = (T_tube[i,10]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_ff_i)
        Q_abs_ss[1] = (T_tube[i,11]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_f_i)
        Q_abs_ss[2] = (T_tube[i,12]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_c_i)
        Q_abs_ss[3] = (T_tube[i,13]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_b_i)
        Q_abs_ss[4] = (T_tube[i,14]-T_ss_ave[i])/R_conv_in(T_ss_ave[i],A_bb_i)
        Q_abs_ss_tot[i] = np.sum(Q_abs_ss)
        Q_rad_tot[i] = Q_h[i]*h*B/2
        eta[i] = Q_abs_ss_tot[i]/Q_rad_tot[i]
        T_ss[i+1] = T_ss[i]+Q_abs_ss_tot[i]/(m_dot_ss*Cp_ss(T_ss[i]))
        T_ss_e[i] = T_ss[i]+Q_abs_ss_tot[i]/(m_dot_ss*Cp_ss(T_ss[i]))
        T_ss_ave_new[i] = (T_ss[i+1]+T_ss[i])/2
        T_ss_ave_err[i] = np.absolute(T_ss_ave[i]-T_ss_ave_new[i])
        T_ss_ave[i] = T_ss_ave_new[i]
        eta_weighted[i] = eta[i]*Q_abs_ss_tot[i]

pred_end = time.time()
pred_time = pred_end-pred_start
print('Average prediction time for one sample: {:.4e}',pred_time/n_sections,'s'
)
i=10
print('m_dot_ss total =',m_dot_ss*2*32,'kg/s')
print('T_tube_max =',np.max(T_tube)-kelv)
```

```
print('T_ss_i =',T_ss_i[i]-kelv,'deg C')
print('T_ss_e =',T_ss_e[i]-kelv,'deg C')
print('Flux =',Q_h[i]/10**6,'MW/m^2')
print('Heat transfer absorbed =',Q_abs_ss_tot[i]/1000,'kW')
print('Receiver efficiency =',np.sum(eta_weighted)/np.sum(Q_abs_ss_tot))
#print(T[10])
```

```
Average prediction time for one sample: {:.4e} 0.010939599514007569 s
m_dot_ss total = 67.55546084096389 kg/s
T_tube_max = 2055.190094451454
T_ss_i = -273.15 deg C
T_ss_e = 524.1074288215042 deg C
Flux = 1.450178964185312 MW/m^2
Heat transfer absorbed = 8.116304375413709 kW
Receiver efficiency = 0.8855666562252141
```

```
comb = np.zeros((n_sections,4))
comb[:,0] = T_ss_e-T_ss_i_ar
comb[:,1] = T_tube[:,1]
comb[:,2] = T_tube[:,14]
comb[:,3] = Q_abs_ss_tot
df1 = pd.DataFrame(comb,columns=["T_ss_delta","T_tube_max","T_tube_min","Q_abs"
])
print(df1)
df1.to_csv("Tube_results_16000_cases_v3.csv")
```

```
       T_ss_delta   T_tube_max  T_tube_min         Q_abs
0        0.776846   950.933493  706.066059   6952.380465
1        1.316864  1180.968641  791.983962  17775.543120
2        4.847706  1319.587612  770.341088   8058.132294
3        0.037510   657.862359  639.063992    790.981631
4        2.417361  1199.096693  684.382783  18033.055125
...           ...          ...         ...           ...
15995    0.313231   873.430936  693.652596   7085.540938
15996    0.385029   939.143309  695.119283  11263.153523
15997    0.199396   820.147707  716.780318   4503.523227
15998    3.273853  1321.108474  818.114601  18010.971431
15999    6.800064  1713.066183  700.377673  10843.862513

[16000 rows x 4 columns]
```

Appendix F. FSM neural network training

```
from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive

import numpy as np
import pandas as pd
import time
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
import torch
from torch import nn, optim
from torch.nn import functional as F
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# Specify GPU for PyTorch, else CPU
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Check which device is being used
print (device)

# Define datatype as 32-bit floating point
dtype = torch.float

cuda

class Net(nn.Module):
    def __init__(self, input_dim, output_dim, n_hidden=64):
        super(Net, self).__init__()

        self.input_dim = input_dim
        self.output_dim = output_dim
        self.n_hidden = n_hidden

        self.input_layer = nn.Linear(self.input_dim,self.n_hidden)
        self.h1 = nn.Linear(self.n_hidden,self.n_hidden)
        self.h2 = nn.Linear(self.n_hidden, self.n_hidden)
        self.h3 = nn.Linear(self.n_hidden, self.n_hidden)
        self.h4 = nn.Linear(self.n_hidden, self.n_hidden)
        self.h5 = nn.Linear(self.n_hidden, self.n_hidden)
        self.output_layer = nn.Linear(self.n_hidden, self.output_dim)

    def forward(self, x):
        x = torch.tanh(self.input_layer(x))
        x = torch.tanh(self.h1(x))
        x = torch.tanh(self.h2(x))
        x = torch.tanh(self.h3(x))
        x = torch.tanh(self.h4(x))
        x = torch.tanh(self.h5(x))
```

```
        x = self.output_layer(x)
        return x

  #Define function to calculate R^2
def R2(yhat, ytrue):
    y_mean = np.mean(ytrue)
    sum_squared_residuals = 0
    sum_squared_total = 0
    for i in range(len(ytrue)):
        sum_squared_residuals += (ytrue[i] - yhat[i])**2
        sum_squared_total += (ytrue[i] - y_mean)**2
    return (1 - sum_squared_residuals/sum_squared_total)

debug = 0

# Read all the data from excel
path1 = "/content/drive/MyDrive/Colab CSVs/LHS_8000_cases_v4.csv"
path2 = "/content/drive/MyDrive/Colab CSVs/Combined_flux_8000_cases_v4.csv"
df1 = pd.read_csv(path1) #import csv
input_data = df1.to_numpy() #convert flux df to numpy
df2 = pd.read_csv(path2) #import csv
y_flux = df2.to_numpy() #convert flux df to numpy

x = input_data[:,4:14]
y = y_flux[:,1:21]

# Number of samples being used
samples = len(x)
x_norm = np.zeros((samples,8))

#precalculations to normalize by hand
DNI_max = np.amax(x[:,0])
DNI_min = np.amin(x[:,0])
el_max = np.amax(x[:,1])
el_min = np.amin(x[:,1])
az_max = np.amax(x[:,2])
az_min = np.amin(x[:,2])
temp_max = np.amax(x[:,3])
temp_min = np.amin(x[:,3])
RH_max = np.amax(x[:,4])
RH_min = np.amin(x[:,4])
qd_max = np.amax(x[:,5])
qd_min = np.amin(x[:,5])
th_max = np.amax(x[:,6])
th_min = np.amin(x[:,6])
af_max = np.amax(x[:,7])
af_min = np.amin(x[:,7])
hd_max = np.amax(x[:,8])
hd_min = np.amin(x[:,8])
np_max = np.amax(x[:,9])
np_min = np.amin(x[:,9])
y_max = np.amax(y)
# Normalize the inputs and outputs
```

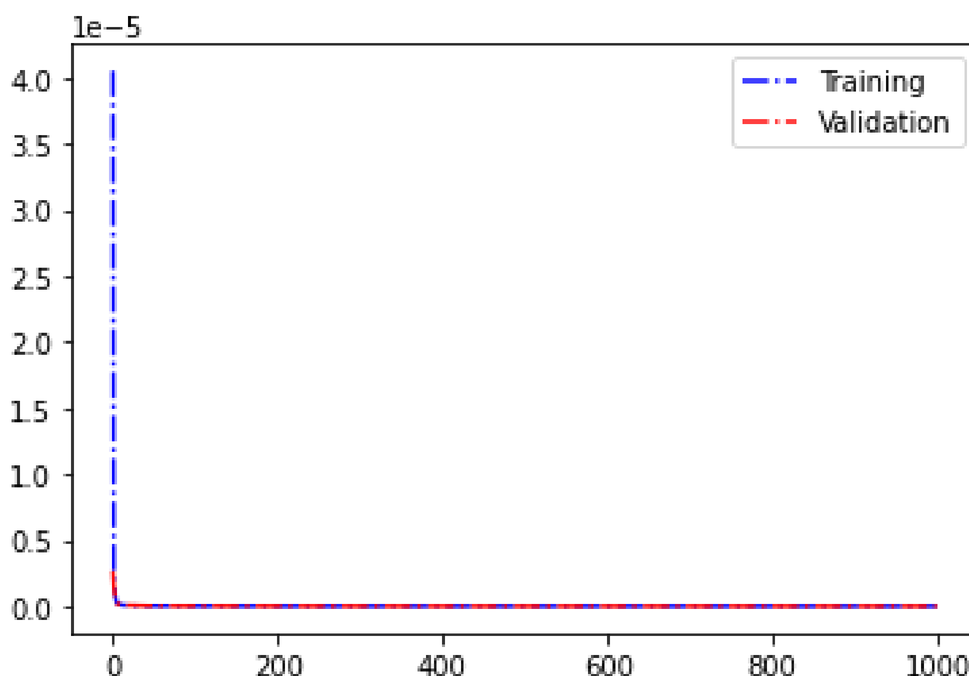
```
scaler = MinMaxScaler()
# x_norm = scaler.fit_transform(x)
x_norm[:,0] = (x[:,0]-DNI_min)/(DNI_max-DNI_min)
x_norm[:,1] = (x[:,1]-el_min)/(el_max-el_min)
x_norm[:,2] = (x[:,2]-az_min)/(az_max-az_min)
x_norm[:,3] = (x[:,5]-qd_min)/(qd_max-qd_min)
x_norm[:,4] = (x[:,6]-th_min)/(th_max-th_min)
x_norm[:,5] = (x[:,7]-af_min)/(af_max-af_min)
x_norm[:,6] = (x[:,8]-hd_min)/(hd_max-hd_min)
x_norm[:,7] = (x[:,9]-np_min)/(np_max-np_min)
y_norm = y/y_max
print(x_norm)

# Convert data in NumPy arrays into tensors
x_tensor = torch.from_numpy(x_norm).float().to(device)       # Specify datatype
as 32-bit floating point
y_tensor = torch.from_numpy(y_norm).float().to(device)
#print(x)

[[0.85222121 0.58645922 0.17896472 ... 0.37974811 0.97606469 0.         ]
 [0.85222121 0.58645922 0.17896472 ... 0.37974811 0.97606469 0.02564103]
 [0.85222121 0.58645922 0.17896472 ... 0.37974811 0.97606469 0.05128205]
 ...
 [0.         0.62795844 0.05874837 ... 0.76184045 0.53793465 0.94871795]
 [0.         0.62795844 0.05874837 ... 0.76184045 0.53793465 0.97435897]
 [0.         0.62795844 0.05874837 ... 0.76184045 0.53793465 1.         ]]

# Split data for training, validation (development) and testing
test_size = 0.2
x_tad, x_test, y_tad, y_test = train_test_split(x_tensor, y_tensor, test_size =
test_size)    # tad = training and development
valid_size = 0.1
x_train, x_valid, y_train, y_valid = train_test_split(x_tad, y_tad, test_size =
valid_size)

# Create datasets and dataloaders:
batch_size = 256
# Training
train_ds = torch.utils.data.TensorDataset(x_train, y_train)
train_dl = torch.utils.data.DataLoader(train_ds, batch_size = batch_size, shuff
le = False)
# Validation (Development)
valid_ds = torch.utils.data.TensorDataset(x_valid, y_valid)
valid_dl = torch.utils.data.DataLoader(valid_ds, batch_size = batch_size, shuff
le = False)
# Testing (Hold-out data)
test_ds = torch.utils.data.TensorDataset(x_test, y_test)

# Hyperparameter selection (parameter used to control the learning process)
input_data_dim = len(x_norm[0])
output_data_dim = len(y_norm[0])
epochs = 500
lr = 5e-4
```

```
n_hidden = 128

# Form model
model = Net(input_dim = input_data_dim, output_dim = output_data_dim, n_hidden
= n_hidden).to(device)

# Use torch.optim to initialize chosen optimiser
optimizer = torch.optim.Adam(model.parameters(), lr = lr)

# Print the model configuration
print(model)

# Define the loss function
loss_fn = torch.nn.MSELoss(reduction = 'mean')

# Set up graph to plot loss (errors)
history_train = []
history_valid = []
```

```
Net(
  (input_layer): Linear(in_features=8, out_features=128, bias=True)
  (h1): Linear(in_features=128, out_features=128, bias=True)
  (h2): Linear(in_features=128, out_features=128, bias=True)
  (h3): Linear(in_features=128, out_features=128, bias=True)
  (h4): Linear(in_features=128, out_features=128, bias=True)
  (h5): Linear(in_features=128, out_features=128, bias=True)
  (output_layer): Linear(in_features=128, out_features=20, bias=True)
)
```

```
# Train model:
start = time.time()
for epoch in range(epochs):
  # Training
  model.train()
  train_loss = 0
  for xb, yb in train_dl:
    xb = xb.to(device)
    yb = yb.to(device)
    optimizer.zero_grad()
    y_p = model(xb.float())
    loss = loss_fn(y_p, yb.float())
    loss.backward()
    optimizer.step()
    train_loss = train_loss + loss.item()

  # Validating
  model.eval()
  valid_loss = 0
  with torch.no_grad():
    for xb, yb in valid_dl:
      y_p = model(xb.to(device).float())
      loss = loss_fn(y_p,yb.float())
      valid_loss = valid_loss + loss.item()
```

```
  if epoch % 25 == 0:
    print('Epoch: {}  Average training model loss: {:.4e}'.format(epoch, train_
loss / len(train_dl.dataset)), '    ', \
          'Average validation model loss: {:.4e}'.format(valid_loss / len(valid
_dl.dataset)))

  history_train.append(train_loss / len(train_dl.dataset))      # Adding the av
erage error per training batch (Total error / Number of batches)
  history_valid.append(valid_loss / len(valid_dl.dataset))

end = time.time()

print('\n Total training time = {:.4f}'.format(end - start), ' [s]')

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-066adeffedd3> in <module>
      3 for epoch in range(epochs):
      4   # Training
----> 5   model.train()
      6   train_loss = 0
      7   for xb, yb in train_dl:

NameError: name 'model' is not defined

# Plot loss function
plt.figure(0)
plt.plot(history_train, 'b-.', markersize = 2, label = 'Training')
plt.plot(history_valid, 'r-.', markersize = 2, label = 'Validation')
plt.legend()
plt.show()
```



```
# Save the model
torch.save(model.state_dict(),'model.pth')

yhat_np_in =  y_p.cpu().detach().numpy()
y_in_np = yb.float().cpu().detach().numpy()

R2_nn_in = R2(np.ndarray.flatten(yhat_np_in),np.ndarray.flatten(y_in_np))
MSE_in = train_loss / len(train_dl.dataset)
print('R2 in sample =',R2_nn_in)
print('MSE in sample =',MSE_in)

R2 in sample = 0.9999553320391559
MSE in sample = 6.420509412525317e-09

def RMS(y, yhat):
  mse = np.mean(np.power((yhat - y),2.0))
  rmse = np.sqrt(mse)
  return rmse

yhat_np_in_rescaled =  y_p.cpu().detach().numpy()*y_max
y_in_np_rescaled = yb.float().cpu().detach().numpy()*y_max

R2_in_rescaled = R2(np.ndarray.flatten(yhat_np_in_rescaled),np.ndarray.flatten(
y_in_np_rescaled))
RMS_in_rescaled = RMS_out2 = RMS(np.ndarray.flatten(y_in_np_rescaled),np.ndarra
y.flatten(yhat_np_in_rescaled))
print('Rescaled RMS in sample=',RMS_in_rescaled)
print('Rescaled R^2 in sample=',R2_in_rescaled)

Rescaled RMS in sample= 3.7985559
Rescaled R^2 in sample= 0.9999553320312118

yhat = np.zeros((len(x_test), output_data_dim))
yhat_tensor = torch.from_numpy(yhat).float().to(device)

# Test hold-out set
with torch.no_grad():
  model.eval()
  test_loss = 0
  pred_time = np.zeros(len(x_test))
  for i in range(len(x_test)):
    pred_start = time.time()
    yhat_tensor[i] = (model(x_test[i,:]))
    pred_end = time.time()
    pred_time[i] = pred_end - pred_start
    loss = loss_fn(yhat_tensor[i], y_test[i].to(device).float())    # Total los
s for every output entry in sample i
    test_loss = test_loss + loss.item()

def R2(yhat, ytrue):
    y_mean = np.mean(ytrue)
    sum_squared_residuals = 0
    sum_squared_total = 0
    for i in range(len(ytrue)):
```

```
        sum_squared_residuals += (ytrue[i] - yhat[i])**2
        sum_squared_total += (ytrue[i] - y_mean)**2
    return (1 - sum_squared_residuals/sum_squared_total)
yhat_np_out =  yhat_tensor.cpu().detach().numpy()
y_test_np = y_test.cpu().detach().numpy()

R2_nn_out = R2(np.ndarray.flatten(yhat_np_out),np.ndarray.flatten(y_test_np))

print('R2 out sample =',R2_nn_out)
print('Average testing model loss:',test_loss / len(x_test))
print('Average prediction time for one sample: {:.4e}'.format(np.average(pred_t
ime*20)), ' [s]')

R2 out sample = 0.999956669020875
Average testing model loss: 1.7237890195015566e-06
Average prediction time for one sample: 5.7691e-03  [s]

y_test_rescaled = y_test_np*y_max
y_pred_rescaled = yhat_np_out*y_max
R2_out_rescaled = R2(np.ndarray.flatten(y_pred_rescaled),np.ndarray.flatten(y_t
est_rescaled))
RMS_out = RMS(np.ndarray.flatten(y_test_rescaled),np.ndarray.flatten(y_pred_res
caled))
print('Rescaled RMS out sample =',RMS_out)
print('Rescaled R^2 out sample =',R2_out_rescaled)

Rescaled RMS out sample = 3.8299232
Rescaled R^2 out sample = 0.9999566690207269

print(np.average(y))
print(np.amax(y))

662.1718980647154
2917.0779208821755

print(y_test_np)
print('')
print(yhat_np_out)

[[0.36327276 0.36717603 0.3618586  ... 0.32182714 0.33915913 0.35223058]
 [0.67138034 0.6810634  0.6654943  ... 0.5998916  0.6315795  0.6473655 ]
 [0.3638869  0.36681178 0.36149344 ... 0.33087024 0.344616   0.35253438]
 ...
 [0.16749717 0.170378   0.16567332 ... 0.14542937 0.15714684 0.16090192]
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]]

[[ 3.60213697e-01  3.63831520e-01  3.58495891e-01 ...  3.20654780e-01
   3.37154865e-01  3.50189537e-01]
 [ 6.71553373e-01  6.78750634e-01  6.66482449e-01 ...  6.01281881e-01
   6.31252825e-01  6.49020255e-01]
 [ 3.62376153e-01  3.65845084e-01  3.60294700e-01 ...  3.29431206e-01
   3.43064278e-01  3.51427555e-01]
 ...
```

```
 [ 1.67294979e-01  1.69746473e-01  1.66151211e-01 ...  1.46752909e-01
   1.57172024e-01  1.59965128e-01]
 [-7.40036368e-04 -3.99768353e-04 -2.18495727e-04 ... -2.85029411e-04
  -6.11782074e-04 -4.40031290e-04]
 [ 1.31875277e-04  7.61151314e-05  5.90980053e-05 ... -1.14023685e-04
  -1.38685107e-04 -1.72346830e-04]]

# Read all the data from excel
path1 = "/content/drive/MyDrive/Colab CSVs/LHS_2000_cases_v4.csv"
path2 = "/content/drive/MyDrive/Colab CSVs/Combined_flux_2000_cases_v4.csv"
df1 = pd.read_csv(path1) #import csv
input_data = df1.to_numpy() #convert flux df to numpy
df2 = pd.read_csv(path2) #import csv
y_flux2 = df2.to_numpy() #convert flux df to numpy

x2 = input_data[:,4:14]
y2 = y_flux2[:,1:21]

# Number of samples being used
samples2 = len(x2)
x_norm2 = np.zeros((samples2,8))

# Normalize the inputs and outputs
scaler = MinMaxScaler()
# x_norm = scaler.fit_transform(x)
x_norm2[:,0] = (x2[:,0]-DNI_min)/(DNI_max-DNI_min)
x_norm2[:,1] = (x2[:,1]-el_min)/(el_max-el_min)
x_norm2[:,2] = (x2[:,2]-az_min)/(az_max-az_min)
x_norm2[:,3] = (x2[:,5]-qd_min)/(qd_max-qd_min)
x_norm2[:,4] = (x2[:,6]-th_min)/(th_max-th_min)
x_norm2[:,5] = (x2[:,7]-af_min)/(af_max-af_min)
x_norm2[:,6] = (x2[:,8]-hd_min)/(hd_max-hd_min)
x_norm2[:,7] = (x2[:,9]-np_min)/(np_max-np_min)

# Normalize the inputs and outputs
y_norm2 = y2/y_max

# Convert data in NumPy arrays into tensors
x_tensor2 = torch.from_numpy(x_norm2).float().to(device)       # Specify datatY
pe as 32-bit floating point
y_tensor2 = torch.from_numpy(y_norm2).float().to(device)
print(x_norm2)

[[0.42973708 0.46897534 0.86714595 ... 0.73656726 0.05549877 0.        ]
 [0.42973708 0.46897534 0.86714595 ... 0.73656726 0.05549877 0.02564103]
 [0.42973708 0.46897534 0.86714595 ... 0.73656726 0.05549877 0.05128205]
 ...
 [0.52765186 0.22858522 0.18178842 ... 0.30577551 0.69843294 0.94871795]
 [0.52765186 0.22858522 0.18178842 ... 0.30577551 0.69843294 0.97435897]
 [0.52765186 0.22858522 0.18178842 ... 0.30577551 0.69843294 1.        ]]

yhat2 = np.zeros((len(x_tensor2), output_data_dim))
yhat_tensor2 = torch.from_numpy(yhat2).float().to(device)
```

```
# Test new hold-out set
with torch.no_grad():
  model.eval()
  test_loss = 0
  pred_time = np.zeros(len(x_tensor2))
  for i in range(len(x_tensor2)):
    pred_start = time.time()
    yhat_tensor2[i] = (model(x_tensor2[i,:]))
    pred_end = time.time()
    pred_time[i] = pred_end - pred_start
    loss = loss_fn(yhat_tensor2[i], y_tensor2[i].to(device).float())    # Total
loss for every output entry in sample i
    test_loss = test_loss + loss.item()

yhat_np_out2 =  yhat_tensor2.cpu().detach().numpy()
y_test_np2 = y_tensor2.cpu().detach().numpy()
print(np.shape(yhat_np_out2))
print(np.shape(y_test_np2))

R2_nn_out2 = R2(np.ndarray.flatten(yhat_np_out2),np.ndarray.flatten(y_test_np2)
)

print('R2 out sample =',R2_nn_out2)
print('Average testing model loss:',test_loss / len(x_tensor2))
print('Average prediction time for one sample: {:.4e}'.format(np.average(pred_t
ime*20)), ' [s]')
```

```
(80000, 20)
(80000, 20)
R2 out sample = 0.9999551020188919
Average testing model loss: 1.7843543908426683e-06
Average prediction time for one sample: 5.9576e-03  [s]
```

```
y_test_rescaled2 = y_test_np2*y_max
y_pred_rescaled2 = yhat_np_out2*y_max
R2_out_rescaled2 = R2(np.ndarray.flatten(y_pred_rescaled2),np.ndarray.flatten(y
_test_rescaled2))
RMS_out2 = RMS(np.ndarray.flatten(y_test_rescaled2),np.ndarray.flatten(y_pred_r
escaled2))
print('Rescaled RMS out sample =',RMS_out2)
print('Rescaled R^2 out sample=',R2_out_rescaled2)
```

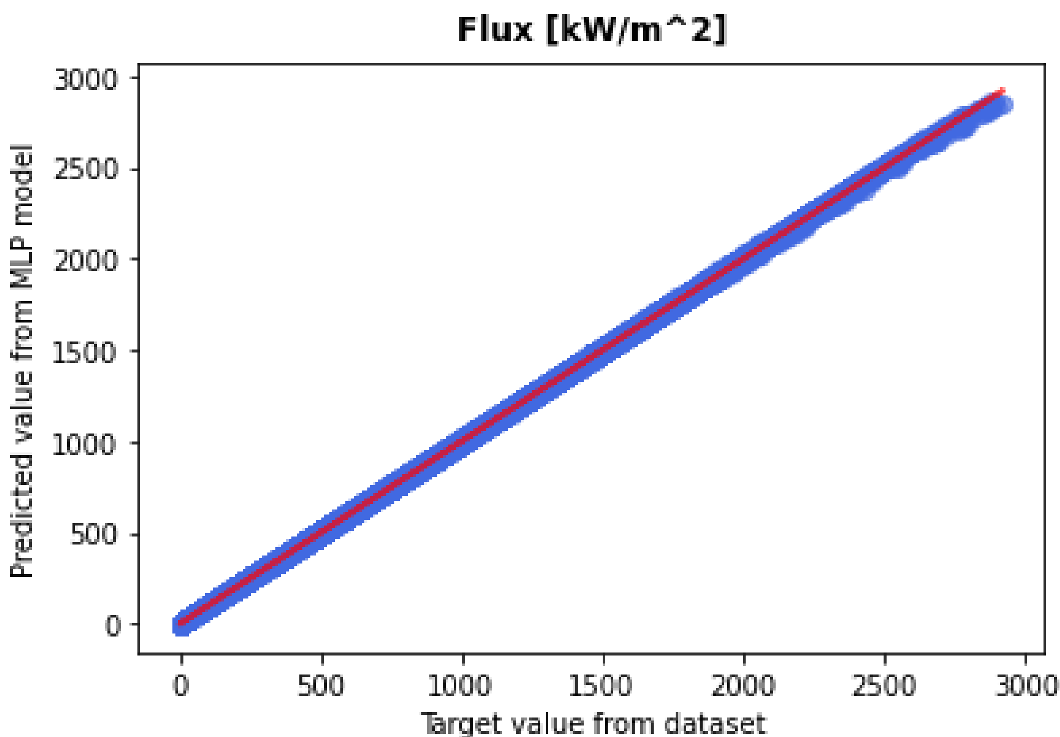
```
Rescaled RMS out sample = 3.8966243
Rescaled R^2 out sample= 0.9999551020201892
```

```
print(y_test_rescaled2)
print(y_pred_rescaled2)
```

```
[[184.98299 188.0346  183.53278 ... 152.19992 167.36081 176.99644]
 [228.30846 232.1117  226.84624 ... 190.25838 208.00328 218.74922]
 [279.66983 284.32513 278.20737 ... 235.62529 256.28827 268.24765]
 ...
 [297.61215 299.82062 290.19736 ... 247.4862  266.12497 283.03772]
```

```
 [230.93748 232.64548 225.13493 ... 190.24443 205.3582  219.36171]
 [175.88963 177.23695 171.45648 ... 143.21681 155.39702 166.82523]]
[[181.31682 183.57915 180.05109 ... 149.67392 164.27838 173.43614]
 [224.40816 227.26555 222.68152 ... 187.75385 204.99133 214.80441]
 [275.91177 279.52838 273.72617 ... 233.32837 253.66193 264.23477]
 ...
 [301.40076 304.74805 297.01172 ... 250.05931 270.91278 287.14346]
 [235.25229 237.88597 231.70737 ... 193.01485 210.01073 223.65668]
 [180.41786 182.48271 177.64949 ... 145.77768 159.54552 171.03873]]
```

```
y_test_flat = np.ndarray.flatten(y_test_rescaled2)
y_pred_flat = np.ndarray.flatten(y_pred_rescaled2)
plt.scatter(y_test_flat, y_pred_flat, marker = 'o', c = 'royalblue', alpha = 0.
4)
m, c = np.polyfit(y_test_flat, y_pred_flat, 1)
plt.plot(y_test_flat, m*y_test_flat+c, c = 'r', linewidth = 2, alpha = 0.7)
plt.title('Flux [kW/m^2]', fontweight = 'bold', pad = 10)
plt.xlabel("Target value from dataset")
plt.ylabel("Predicted value from MLP model")
plt.show()
```




```
torch.save(model.state_dict(),'model.pth')
```

Appendix G. TSM Pytorch training

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
import numpy as np
import pandas as pd
import time
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
import torch
from torch import nn, optim
from torch.nn import functional as F
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# Specify GPU for PyTorch, else CPU
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Check which device is being used
print (device)

# Define datatype as 32-bit floating point
dtype = torch.float
```

```
cuda
```

```
class Net(nn.Module):
    def __init__(self, input_dim, output_dim, n_hidden=64):
        super(Net, self).__init__()

        self.input_dim = input_dim
        self.output_dim = output_dim
        self.n_hidden = n_hidden

        self.input_layer = nn.Linear(self.input_dim,self.n_hidden)
        self.h1 = nn.Linear(self.n_hidden,self.n_hidden)
        self.h2 = nn.Linear(self.n_hidden, self.n_hidden)
        self.h3 = nn.Linear(self.n_hidden, self.n_hidden)
        self.h4 = nn.Linear(self.n_hidden, self.n_hidden)
        self.h5 = nn.Linear(self.n_hidden, self.n_hidden)
        self.output_layer = nn.Linear(self.n_hidden, self.output_dim)

    def forward(self, x):
        x = torch.tanh(self.input_layer(x))
        x = torch.tanh(self.h1(x))
        x = torch.tanh(self.h2(x))
        x = torch.tanh(self.h3(x))
        x = torch.tanh(self.h4(x))
        x = torch.tanh(self.h5(x))
```

```
        x = self.output_layer(x)
        return x

  #Define function to calculate R^2
def R2(yhat, ytrue):
    y_mean = np.mean(ytrue)
    sum_squared_residuals = 0
    sum_squared_total = 0
    for i in range(len(ytrue)):
        sum_squared_residuals += (ytrue[i] - yhat[i])**2
        sum_squared_total += (ytrue[i] - y_mean)**2
    return (1 - sum_squared_residuals/sum_squared_total)

debug = 0

# Read all the data from excel
path1 = "/content/drive/MyDrive/Colab CSVs/LHS_8000_cases_v4.csv"
path2 = "/content/drive/MyDrive/Colab CSVs/Combined_flux_8000_cases_v4.csv"
df1 = pd.read_csv(path1) #import csv
input_data = df1.to_numpy() #convert flux df to numpy
df2 = pd.read_csv(path2) #import csv
y_flux = df2.to_numpy() #convert flux df to numpy

x = input_data[:,4:14]
y = y_flux[:,1:21]

# Number of samples being used
samples = len(x)
x_norm = np.zeros((samples,8))

#precalculations to normalize by hand
DNI_max = np.amax(x[:,0])
DNI_min = np.amin(x[:,0])
el_max = np.amax(x[:,1])
el_min = np.amin(x[:,1])
az_max = np.amax(x[:,2])
az_min = np.amin(x[:,2])
temp_max = np.amax(x[:,3])
temp_min = np.amin(x[:,3])
RH_max = np.amax(x[:,4])
RH_min = np.amin(x[:,4])
qd_max = np.amax(x[:,5])
qd_min = np.amin(x[:,5])
th_max = np.amax(x[:,6])
th_min = np.amin(x[:,6])
af_max = np.amax(x[:,7])
af_min = np.amin(x[:,7])
hd_max = np.amax(x[:,8])
hd_min = np.amin(x[:,8])
np_max = np.amax(x[:,9])
np_min = np.amin(x[:,9])
y_max = np.amax(y)
# Normalize the inputs and outputs
```

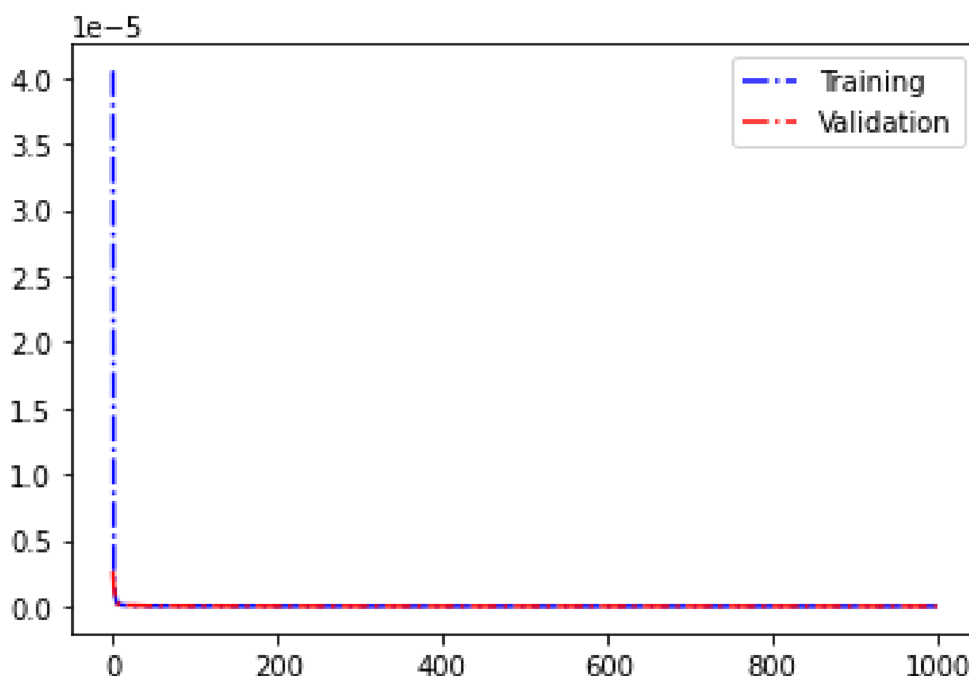
```
scaler = MinMaxScaler()
# x_norm = scaler.fit_transform(x)
x_norm[:,0] = (x[:,0]-DNI_min)/(DNI_max-DNI_min)
x_norm[:,1] = (x[:,1]-el_min)/(el_max-el_min)
x_norm[:,2] = (x[:,2]-az_min)/(az_max-az_min)
x_norm[:,3] = (x[:,5]-qd_min)/(qd_max-qd_min)
x_norm[:,4] = (x[:,6]-th_min)/(th_max-th_min)
x_norm[:,5] = (x[:,7]-af_min)/(af_max-af_min)
x_norm[:,6] = (x[:,8]-hd_min)/(hd_max-hd_min)
x_norm[:,7] = (x[:,9]-np_min)/(np_max-np_min)
y_norm = y/y_max
print(x_norm)

# Convert data in NumPy arrays into tensors
x_tensor = torch.from_numpy(x_norm).float().to(device)       # Specify datatype
as 32-bit floating point
y_tensor = torch.from_numpy(y_norm).float().to(device)
#print(x)

[[0.85222121 0.58645922 0.17896472 ... 0.37974811 0.97606469 0.        ]
 [0.85222121 0.58645922 0.17896472 ... 0.37974811 0.97606469 0.02564103]
 [0.85222121 0.58645922 0.17896472 ... 0.37974811 0.97606469 0.05128205]
 ...
 [0.         0.62795844 0.05874837 ... 0.76184045 0.53793465 0.94871795]
 [0.         0.62795844 0.05874837 ... 0.76184045 0.53793465 0.97435897]
 [0.         0.62795844 0.05874837 ... 0.76184045 0.53793465 1.        ]]

# Split data for training, validation (development) and testing
test_size = 0.2
x_tad, x_test, y_tad, y_test = train_test_split(x_tensor, y_tensor, test_size =
test_size)    # tad = training and development
valid_size = 0.1
x_train, x_valid, y_train, y_valid = train_test_split(x_tad, y_tad, test_size =
valid_size)

# Create datasets and dataloaders:
batch_size = 256
# Training
train_ds = torch.utils.data.TensorDataset(x_train, y_train)
train_dl = torch.utils.data.DataLoader(train_ds, batch_size = batch_size, shuff
le = False)
# Validation (Development)
valid_ds = torch.utils.data.TensorDataset(x_valid, y_valid)
valid_dl = torch.utils.data.DataLoader(valid_ds, batch_size = batch_size, shuff
le = False)
# Testing (Hold-out data)
test_ds = torch.utils.data.TensorDataset(x_test, y_test)

# Hyperparameter selection (parameter used to control the learning process)
input_data_dim = len(x_norm[0])
output_data_dim = len(y_norm[0])
epochs = 500
lr = 5e-4
```

```
n_hidden = 128

# Form model
model = Net(input_dim = input_data_dim, output_dim = output_data_dim, n_hidden
= n_hidden).to(device)

# Use torch.optim to initialize chosen optimiser
optimizer = torch.optim.Adam(model.parameters(), lr = lr)

# Print the model configuration
print(model)

# Define the loss function
loss_fn = torch.nn.MSELoss(reduction = 'mean')

# Set up graph to plot loss (errors)
history_train = []
history_valid = []
```

```
Net(
  (input_layer): Linear(in_features=8, out_features=128, bias=True)
  (h1): Linear(in_features=128, out_features=128, bias=True)
  (h2): Linear(in_features=128, out_features=128, bias=True)
  (h3): Linear(in_features=128, out_features=128, bias=True)
  (h4): Linear(in_features=128, out_features=128, bias=True)
  (h5): Linear(in_features=128, out_features=128, bias=True)
  (output_layer): Linear(in_features=128, out_features=20, bias=True)
)
```

```
# Train model:
start = time.time()
for epoch in range(epochs):
  # Training
  model.train()
  train_loss = 0
  for xb, yb in train_dl:
    xb = xb.to(device)
    yb = yb.to(device)
    optimizer.zero_grad()
    y_p = model(xb.float())
    loss = loss_fn(y_p, yb.float())
    loss.backward()
    optimizer.step()
    train_loss = train_loss + loss.item()

  # Validating
  model.eval()
  valid_loss = 0
  with torch.no_grad():
    for xb, yb in valid_dl:
      y_p = model(xb.to(device).float())
      loss = loss_fn(y_p,yb.float())
      valid_loss = valid_loss + loss.item()
```

```
  if epoch % 25 == 0:
    print('Epoch: {}  Average training model loss: {:.4e}'.format(epoch, train_
loss / len(train_dl.dataset)), '    ', \
          'Average validation model loss: {:.4e}'.format(valid_loss / len(valid
_dl.dataset)))

  history_train.append(train_loss / len(train_dl.dataset))      # Adding the av
erage error per training batch (Total error / Number of batches)
  history_valid.append(valid_loss / len(valid_dl.dataset))

end = time.time()

print('\n Total training time = {:.4f}'.format(end - start), ' [s]')

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-066adeffedd3> in <module>
      3 for epoch in range(epochs):
      4   # Training
----> 5   model.train()
      6   train_loss = 0
      7   for xb, yb in train_dl:

NameError: name 'model' is not defined

# Plot loss function
plt.figure(0)
plt.plot(history_train, 'b-.', markersize = 2, label = 'Training')
plt.plot(history_valid, 'r-.', markersize = 2, label = 'Validation')
plt.legend()
plt.show()
```



```
# Save the model
torch.save(model.state_dict(),'model.pth')

yhat_np_in =  y_p.cpu().detach().numpy()
y_in_np = yb.float().cpu().detach().numpy()

R2_nn_in = R2(np.ndarray.flatten(yhat_np_in),np.ndarray.flatten(y_in_np))
MSE_in = train_loss / len(train_dl.dataset)
print('R2 in sample =',R2_nn_in)
print('MSE in sample =',MSE_in)

R2 in sample = 0.9999553320391559
MSE in sample = 6.420509412525317e-09

def RMS(y, yhat):
  mse = np.mean(np.power((yhat - y),2.0))
  rmse = np.sqrt(mse)
  return rmse

yhat_np_in_rescaled =  y_p.cpu().detach().numpy()*y_max
y_in_np_rescaled = yb.float().cpu().detach().numpy()*y_max

R2_in_rescaled = R2(np.ndarray.flatten(yhat_np_in_rescaled),np.ndarray.flatten(
y_in_np_rescaled))
RMS_in_rescaled = RMS_out2 = RMS(np.ndarray.flatten(y_in_np_rescaled),np.ndarra
y.flatten(yhat_np_in_rescaled))
print('Rescaled RMS in sample=',RMS_in_rescaled)
print('Rescaled R^2 in sample=',R2_in_rescaled)

Rescaled RMS in sample= 3.7985559
Rescaled R^2 in sample= 0.9999553320312118

yhat = np.zeros((len(x_test), output_data_dim))
yhat_tensor = torch.from_numpy(yhat).float().to(device)

# Test hold-out set
with torch.no_grad():
  model.eval()
  test_loss = 0
  pred_time = np.zeros(len(x_test))
  for i in range(len(x_test)):
    pred_start = time.time()
    yhat_tensor[i] = (model(x_test[i,:]))
    pred_end = time.time()
    pred_time[i] = pred_end - pred_start
    loss = loss_fn(yhat_tensor[i], y_test[i].to(device).float())    # Total los
s for every output entry in sample i
    test_loss = test_loss + loss.item()

def R2(yhat, ytrue):
    y_mean = np.mean(ytrue)
    sum_squared_residuals = 0
    sum_squared_total = 0
    for i in range(len(ytrue)):
```

```
        sum_squared_residuals += (ytrue[i] - yhat[i])**2
        sum_squared_total += (ytrue[i] - y_mean)**2
    return (1 - sum_squared_residuals/sum_squared_total)
yhat_np_out =  yhat_tensor.cpu().detach().numpy()
y_test_np = y_test.cpu().detach().numpy()

R2_nn_out = R2(np.ndarray.flatten(yhat_np_out),np.ndarray.flatten(y_test_np))

print('R2 out sample =',R2_nn_out)
print('Average testing model loss:',test_loss / len(x_test))
print('Average prediction time for one sample: {:.4e}'.format(np.average(pred_t
ime*20)), ' [s]')
```

```
R2 out sample = 0.999956669020875
Average testing model loss: 1.7237890195015566e-06
Average prediction time for one sample: 5.7691e-03  [s]
```

```
y_test_rescaled = y_test_np*y_max
y_pred_rescaled = yhat_np_out*y_max
R2_out_rescaled = R2(np.ndarray.flatten(y_pred_rescaled),np.ndarray.flatten(y_t
est_rescaled))
RMS_out = RMS(np.ndarray.flatten(y_test_rescaled),np.ndarray.flatten(y_pred_res
caled))
print('Rescaled RMS out sample =',RMS_out)
print('Rescaled R^2 out sample =',R2_out_rescaled)
```

```
Rescaled RMS out sample = 3.8299232
Rescaled R^2 out sample = 0.9999566690207269
```

```
print(np.average(y))
print(np.amax(y))
```

```
662.1718980647154
2917.0779208821755
```

```
print(y_test_np)
print('')
print(yhat_np_out)
```

```
[[0.36327276 0.36717603 0.3618586  ... 0.32182714 0.33915913 0.35223058]
 [0.67138034 0.6810634  0.6654943  ... 0.5998916  0.6315795  0.6473655 ]
 [0.3638869  0.36681178 0.36149344 ... 0.33087024 0.344616   0.35253438]
 ...
 [0.16749717 0.170378   0.16567332 ... 0.14542937 0.15714684 0.16090192]
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]]

[[ 3.60213697e-01  3.63831520e-01  3.58495891e-01 ...  3.20654780e-01
   3.37154865e-01  3.50189537e-01]
 [ 6.71553373e-01  6.78750634e-01  6.66482449e-01 ...  6.01281881e-01
   6.31252825e-01  6.49020255e-01]
 [ 3.62376153e-01  3.65845084e-01  3.60294700e-01 ...  3.29431206e-01
   3.43064278e-01  3.51427555e-01]
 ...
```

```
 [ 1.67294979e-01  1.69746473e-01  1.66151211e-01 ...  1.46752909e-01
   1.57172024e-01  1.59965128e-01]
 [-7.40036368e-04 -3.99768353e-04 -2.18495727e-04 ... -2.85029411e-04
  -6.11782074e-04 -4.40031290e-04]
 [ 1.31875277e-04  7.61151314e-05  5.90980053e-05 ... -1.14023685e-04
  -1.38685107e-04 -1.72346830e-04]]

# Read all the data from excel
path1 = "/content/drive/MyDrive/Colab CSVs/LHS_2000_cases_v4.csv"
path2 = "/content/drive/MyDrive/Colab CSVs/Combined_flux_2000_cases_v4.csv"
df1 = pd.read_csv(path1) #import csv
input_data = df1.to_numpy() #convert flux df to numpy
df2 = pd.read_csv(path2) #import csv
y_flux2 = df2.to_numpy() #convert flux df to numpy

x2 = input_data[:,4:14]
y2 = y_flux2[:,1:21]

# Number of samples being used
samples2 = len(x2)
x_norm2 = np.zeros((samples2,8))

# Normalize the inputs and outputs
scaler = MinMaxScaler()
# x_norm = scaler.fit_transform(x)
x_norm2[:,0] = (x2[:,0]-DNI_min)/(DNI_max-DNI_min)
x_norm2[:,1] = (x2[:,1]-el_min)/(el_max-el_min)
x_norm2[:,2] = (x2[:,2]-az_min)/(az_max-az_min)
x_norm2[:,3] = (x2[:,5]-qd_min)/(qd_max-qd_min)
x_norm2[:,4] = (x2[:,6]-th_min)/(th_max-th_min)
x_norm2[:,5] = (x2[:,7]-af_min)/(af_max-af_min)
x_norm2[:,6] = (x2[:,8]-hd_min)/(hd_max-hd_min)
x_norm2[:,7] = (x2[:,9]-np_min)/(np_max-np_min)

# Normalize the inputs and outputs
y_norm2 = y2/y_max

# Convert data in NumPy arrays into tensors
x_tensor2 = torch.from_numpy(x_norm2).float().to(device)       # Specify dataty
pe as 32-bit floating point
y_tensor2 = torch.from_numpy(y_norm2).float().to(device)
print(x_norm2)

[[0.42973708 0.46897534 0.86714595 ... 0.73656726 0.05549877 0.        ]
 [0.42973708 0.46897534 0.86714595 ... 0.73656726 0.05549877 0.02564103]
 [0.42973708 0.46897534 0.86714595 ... 0.73656726 0.05549877 0.05128205]
 ...
 [0.52765186 0.22858522 0.18178842 ... 0.30577551 0.69843294 0.94871795]
 [0.52765186 0.22858522 0.18178842 ... 0.30577551 0.69843294 0.97435897]
 [0.52765186 0.22858522 0.18178842 ... 0.30577551 0.69843294 1.        ]]

yhat2 = np.zeros((len(x_tensor2), output_data_dim))
yhat_tensor2 = torch.from_numpy(yhat2).float().to(device)
```

```
# Test new hold-out set
with torch.no_grad():
  model.eval()
  test_loss = 0
  pred_time = np.zeros(len(x_tensor2))
  for i in range(len(x_tensor2)):
    pred_start = time.time()
    yhat_tensor2[i] = (model(x_tensor2[i,:]))
    pred_end = time.time()
    pred_time[i] = pred_end - pred_start
    loss = loss_fn(yhat_tensor2[i], y_tensor2[i].to(device).float())    # Total
loss for every output entry in sample i
    test_loss = test_loss + loss.item()

yhat_np_out2 =  yhat_tensor2.cpu().detach().numpy()
y_test_np2 = y_tensor2.cpu().detach().numpy()
print(np.shape(yhat_np_out2))
print(np.shape(y_test_np2))

R2_nn_out2 = R2(np.ndarray.flatten(yhat_np_out2),np.ndarray.flatten(y_test_np2)
)

print('R2 out sample =',R2_nn_out2)
print('Average testing model loss:',test_loss / len(x_tensor2))
print('Average prediction time for one sample: {:.4e}'.format(np.average(pred_t
ime*20)), ' [s]')
```

```
(80000, 20)
(80000, 20)
R2 out sample = 0.9999551020188919
Average testing model loss: 1.7843543908426683e-06
Average prediction time for one sample: 5.9576e-03  [s]
```

```
y_test_rescaled2 = y_test_np2*y_max
y_pred_rescaled2 = yhat_np_out2*y_max
R2_out_rescaled2 = R2(np.ndarray.flatten(y_pred_rescaled2),np.ndarray.flatten(y
_test_rescaled2))
RMS_out2 = RMS(np.ndarray.flatten(y_test_rescaled2),np.ndarray.flatten(y_pred_r
escaled2))
print('Rescaled RMS out sample =',RMS_out2)
print('Rescaled R^2 out sample=',R2_out_rescaled2)
```

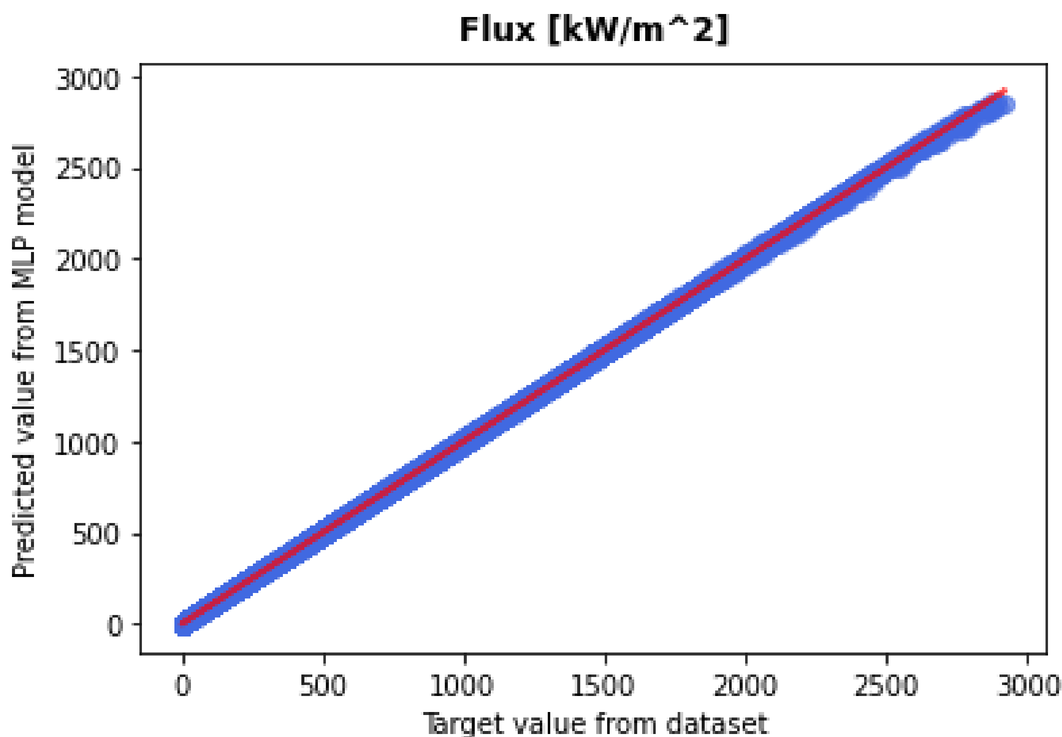
```
Rescaled RMS out sample = 3.8966243
Rescaled R^2 out sample= 0.9999551020201892
```

```
print(y_test_rescaled2)
print(y_pred_rescaled2)
```

```
[[184.98299 188.0346  183.53278 ... 152.19992 167.36081 176.99644]
 [228.30846 232.1117  226.84624 ... 190.25838 208.00328 218.74922]
 [279.66983 284.32513 278.20737 ... 235.62529 256.28827 268.24765]
 ...
 [297.61215 299.82062 290.19736 ... 247.4862  266.12497 283.03772]
```

```
 [230.93748 232.64548 225.13493 ... 190.24443 205.3582  219.36171]
 [175.88963 177.23695 171.45648 ... 143.21681 155.39702 166.82523]]
[[181.31682 183.57915 180.05109 ... 149.67392 164.27838 173.43614]
 [224.40816 227.26555 222.68152 ... 187.75385 204.99133 214.80441]
 [275.91177 279.52838 273.72617 ... 233.32837 253.66193 264.23477]
 ...
 [301.40076 304.74805 297.01172 ... 250.05931 270.91278 287.14346]
 [235.25229 237.88597 231.70737 ... 193.01485 210.01073 223.65668]
 [180.41786 182.48271 177.64949 ... 145.77768 159.54552 171.03873]]
```

```
y_test_flat = np.ndarray.flatten(y_test_rescaled2)
y_pred_flat = np.ndarray.flatten(y_pred_rescaled2)
plt.scatter(y_test_flat, y_pred_flat, marker = 'o', c = 'royalblue', alpha = 0.
4)
m, c = np.polyfit(y_test_flat, y_pred_flat, 1)
plt.plot(y_test_flat, m*y_test_flat+c, c = 'r', linewidth = 2, alpha = 0.7)
plt.title('Flux [kW/m^2]', fontweight = 'bold', pad = 10)
plt.xlabel("Target value from dataset")
plt.ylabel("Predicted value from MLP model")
plt.show()
```

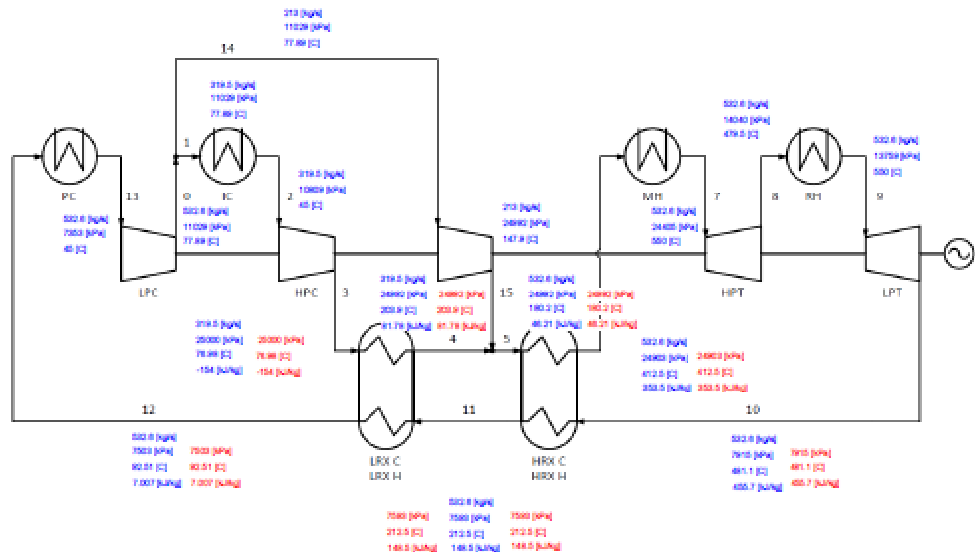



```
torch.save(model.state_dict(),'model.pth')
```

Appendix H. PCRH cycle parameters

File:PCRH Cycle Rev Ac (Intercooling Study).EES 2021-10-04 11:04:04 AM Page 1

EES Ver. 10.829: #5825: For use only by Colin du Sart, University of Cape Town, South Africa



SOLUTION

Unit Settings: SI C kPa kJ mass deg

$A_{ch,HRX}$ = 0.000001571 [m^2]

$A_{ch,LRX}$ = 0.000001571 [m^2]

Capacity$ = 'W_dot_G'

ΔE_{HRXC} = 0.000005214 [kW]

ΔE_{HRXH} = -3.252E-13 [kW]

ΔE_{LRXC} = 4.967E-13 [kW]

ΔE_{LRXH} = -3.178E-13 [kW]

ΔE_{Net} = -0.000005361 [kW]

$\Delta T_{RX,min}$ = 8.651 [C]

$D_{ch,HRX}$ = 0.002 [m]

$D_{ch,LRX}$ = 0.002 [m]

$D_{h,HRX}$ = 0.001222 [m]

$D_{h,LRX}$ = 0.001222 [m]

ε_{HRX} = 0.8936

ε_{LRX} = 0.9498

$\eta_{C,s}$ = 0.89

η_{G} = 0.891

$\eta_{th,Carnot}$ = 0.6135

$\eta_{th,cycle}$ = 0.40860

$\eta_{tot,cycle}$ = 0.3641

$\eta_{T,s}$ = 0.93

Fluid$ = 'CarbonDioxide'

$h_{HPC,out,s}$ = -156.7 [kJ/kg]

$h_{HPT,out,s}$ = 441.2 [kJ/kg]

$h_{LPC,out,s}$ = -54.08 [kJ/kg]

$h_{LPT,out,s}$ = 449.9 [kJ/kg]

$h_{RCC,out,s}$ = -12.09 [kJ/kg]

k_p = 0.0144 [kW/m-K]

L_{HRX} = 2.997 [m]

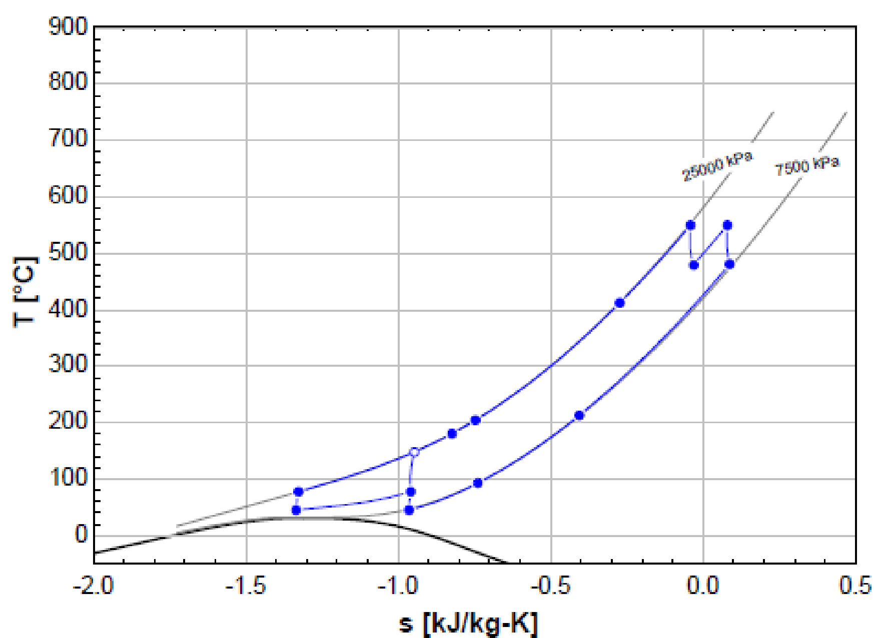
L_{LRX} = 1.333 [m]

$L_{p,HRX}$ = 0.005142 [m]

$L_{p,LRX}$ = 0.005142 [m]
$\dot{m}$ = 532.6 [kg/s]
ne_{HRX} = 40
ne_{LRX} = 40
$n_{ch,HRX}$ = 1000000
$n_{ch,HRXC}$ = 500000
$n_{ch,HRXH}$ = 500000
$n_{ch,LRX}$ = 1000000
$n_{ch,LRXC}$ = 500000
$n_{ch,LRXH}$ = 500000
n_r = 5
n_p = 14
$n_{unit,HRX}$ = 500000
$n_{unit,LRX}$ = 500000
PR = 3.4
Pressure$ = 'PR'
PROptimisation$ = 'off'
PR_c = 3.4
PR_{HPC} = 2.313
PR_{HPT} = 0.5753
PR_{HRXC} = 0.9964
PR_{HRXH} = 0.9592
PR_{HX} = 0.98
PR_{IC} = 0.98
PR_{LPC} = 1.5
PR_{LPT} = 0.5753
PR_{LRXC} = 0.9997
PR_{LRXH} = 0.9882
PR_{MH} = 0.98
PR_{PC} = 0.98
PR_{RH} = 0.98
PR_T = 0.3243
P_{max} = 25000 [kPa]
P_{min} = 7353 [kPa]
$\dot{Q}_C$ = -81221 [kW]
$\dot{Q}_H$ = 137338 [kW]
$\dot{Q}_{HRX}$ = 163633 [kW]
$\dot{Q}_{HRXC}$ = 163633 [kW]
$\dot{Q}_{HRXC,max,abs}$ = 209173 [kW]
$\dot{Q}_{HRXH}$ = -163633 [kW]
$\dot{Q}_{HRXH,max,abs}$ = 183114 [kW]
$\dot{Q}_{HRX,max,abs}$ = 183114 [kW]
$\dot{Q}_{IC}$ = -40324 [kW]
$\dot{Q}_{LRX}$ = 75348 [kW]
$\dot{Q}_{LRXC}$ = 75348 [kW]
$\dot{Q}_{LRXC,max,abs}$ = 79331 [kW]
$\dot{Q}_{LRXH}$ = -75348 [kW]
$\dot{Q}_{LRXH,max,abs}$ = 86904 [kW]
$\dot{Q}_{LRX,max,abs}$ = 79331 [kW]
$\dot{Q}_{MH}$ = 91559 [kW]
$\dot{Q}_{Net}$ = 56117 [kW]
$\dot{Q}_{PC}$ = -40898 [kW]
$\dot{Q}_{RH}$ = 45779 [kW]
$\dot{Q}_{RX}$ = 238981 [kW]
Ra = 0.00001 [m]
Recuperator$ = 'straight'
SR = 0.6
SROptimisation$ = 'off'
SR_{choose} = 0.6

File:PCRH Cycle Rev Ac (Intercooling Study).EES 2021-10-04 11:04:05 AM Page 3
EES Ver. 10.829: #5825: For use only by Colin du Sart, University of Cape Town, South Africa

SR_{max} = 1
$s_{HPC,out,s}$ = -1.338 [kJ/kg-K]
$s_{HPT,out,s}$ = -0.0406 [kJ/kg-K]
$s_{LPC,out,s}$ = -0.9642 [kJ/kg-K]
$s_{LPT,out,s}$ = 0.08031 [kJ/kg-K]
$s_{RCC,out,s}$ = -0.9587 [kJ/kg-K]
TmaxOptimisation$ = 'off'
TminOptimisation$ = 'off'
T_{max} = 550 [C]
T_{min} = 45 [C]
$t_{p,HRX}$ = 0.00163 [m]
$t_{p,LRX}$ = 0.00163 [m]
UA_{HRX} = 13379 [kW/K]
UA_{LRX} = 4869 [kW/K]
$\dot{W}_C$ = -26750 [kW]
$\dot{W}_G$ = 50000 [kW]
$\dot{W}_{HPC}$ = -7767 [kW]
$\dot{W}_{HPT}$ = 41708 [kW]
$\dot{W}_{LPC}$ = -9398 [kW]
$\dot{W}_{LPT}$ = 41159 [kW]
$\dot{W}_{Net}$ = 56117 [kW]
$\dot{W}_{RCC}$ = -9585 [kW]
$\dot{W}_T$ = 82867 [kW]
w_{HPC} = -24.31 [kJ/kg]
$w_{HPC,s}$ = -21.63 [kJ/kg]
w_{HPT} = 78.32 [kJ/kg]
$w_{HPT,s}$ = 84.21 [kJ/kg]
w_{LPC} = -17.65 [kJ/kg]
$w_{LPC,s}$ = -15.71 [kJ/kg]
w_{LPT} = 77.29 [kJ/kg]
$w_{LPT,s}$ = 83.1 [kJ/kg]
w_{RCC} = -45 [kJ/kg]
$w_{RCC,s}$ = -40.05 [kJ/kg]



Appendix I. Solar salt heat exchanger and receiver design

$$Cp_{ss}(T_{ss}) := \left(1396.044 + 0.172 \cdot \frac{T_{ss}}{K}\right)\frac{J}{kg \cdot K}$$

$$v_{ss}(T_{ss}) := \frac{\mu_{ss}(T_{ss})}{\rho_{ss}(T_{ss})}$$

$$Q_{mh} := 91.559MW$$

$$p_{mh_in} := 24.903MPa$$

$$T_{co2_mh_in} := 412.5\,°C$$

$$m_{co2} := 532.6\,\frac{kg}{s}$$

$$p_{co2_mh_out} := 24.429kPa$$

$$T_{co2_mh_out} := 550\,°C$$

$$Q_{rh} := 45.779MW$$

$$p_{co2_mh_in} := 13.958MPa$$

$$T_{co2_rh_in} := 479.5\,°C$$

$$p_{co2_rh_out} := 13.679MPa$$

$$T_{co2_rh_out} := 550\,°C$$

$$T_{ss_max} := 565\,°C$$

$$T_{ss_mh_in} := 565\,°C$$

$$T_{ss_rh_in} := T_{ss_mh_in}$$

$$cp_{co2_mh} := 1.252068\,\frac{kJ}{kg \cdot K}$$

$$cp_{co2_rh} := 1.218275\,\frac{kJ}{kg \cdot K}$$

$$C_{c_mh} := m_{co2} \cdot cp_{co2_mh} = 0.667 \cdot \frac{MW}{K}$$

$$\Delta T_{co2_mh} := T_{co2_mh_out} - T_{co2_mh_in} = 137.5\,K$$

$T_{ss_mh_out} := T_{ss_mh_in} - \Delta T_{co2_mh} = 427.5 \cdot °C$ Guess value $T_{co2_mh_in} = 412.5 \cdot °C$

$$m_{ss_mh} := \frac{Q_{mh}}{Cp_{ss}\left(\frac{T_{ss_mh_out} + T_{ss_mh_in}}{2}\right) \cdot \left(T_{ss_mh_in} - T_{ss_mh_out}\right)} = 435.679 \frac{kg}{s}$$

$$C_{h_mh} := m_{ss_mh} \cdot Cp_{ss}\left(\frac{T_{ss_mh_out} + T_{ss_mh_in}}{2}\right) = 0.666 \cdot \frac{MW}{K}$$

$$Q_{mh_max} := C_{h_mh} \cdot \left(T_{ss_mh_in} - T_{co2_mh_in}\right) = 101.547 \cdot MW$$

$$Q_{mh} = 91.559 \cdot MW$$

$$\varepsilon_{mh} := \frac{Q_{mh}}{Q_{mh_max}} = 0.902$$

$$C_{c_rh} := m_{co2} \cdot cp_{co2_rh} = 0.649 \cdot \frac{MW}{K}$$

$$\Delta T_{co2_rh} := T_{co2_rh_out} - T_{co2_rh_in} = 70.5 K$$

$$T_{ss_rh_out} := T_{ss_rh_in} - \Delta T_{co2_rh} = 494.5 \cdot °C$$

$$m_{ss_rh} := \frac{Q_{rh}}{Cp_{ss}\left(\frac{T_{ss_rh_out} + T_{ss_rh_in}}{2}\right) \cdot \left(T_{ss_rh_in} - T_{ss_rh_out}\right)} = 423.264 \frac{kg}{s}$$

$$C_{h_rh} := m_{ss_rh} \cdot Cp_{ss}\left(\frac{T_{ss_rh_out} + T_{ss_rh_in}}{2}\right) = 0.649 \cdot \frac{MW}{K}$$

$$Q_{rh_max} := C_{h_mh} \cdot \left(T_{ss_rh_in} - T_{co2_rh_in}\right) = 56.933 \cdot MW$$

$$Q_{rh} = 45.779 \cdot MW$$

$$\varepsilon_{rh} := \frac{Q_{rh}}{Q_{rh_max}} = 0.804$$

$$T_{ss_cst} := \frac{T_{ss_mh_out} \cdot m_{ss_mh} + T_{ss_rh_out} \cdot m_{ss_rh}}{m_{ss_mh} + m_{ss_rh}} = 460.516 \cdot °C$$

$$T_{ss_hst} := T_{ss_mh_in} = 565 \cdot °C$$

$$Q_{flux_heliostats} := 0.881 \frac{MW}{m^2}$$

$$A_{SR} := \frac{Q_{SR}}{Q_{flux_heliostats}} = 367.018\,m^2$$

$$R_{SR} := 1.499$$

$$D_{SR} := \sqrt{\frac{A_{SR}}{R_{SR} \cdot \pi}} = 8.828\,m$$

$$H_{SR} := D_{SR} \cdot R_{SR} = 13.233\,m$$

Tubular geometry:

$$D_o := 42.2mm \qquad t := 1.2mm$$

$$D_i := D_o - 2 \cdot t = 39.8 \cdot mm$$

$B_{SR} := 46mm$ tubular pitch

$$N_{tubes_per_panel} := 30$$

$$B_{panel} := B_{SR} \cdot N_{tubes_per_panel} = 1.38\,m$$

$$N_{panels} := \frac{\pi \cdot D_{SR}}{B_{panel}} = 20.097$$

$$A_i := \frac{\pi \cdot D_i^2}{4} = 1.244 \times 10^{-3}\,m^2$$

$$u_{ss} := \frac{m_{hst_in}}{\rho_{ss}(T_{ss_cst}) \cdot A_i \cdot N_{tubes_per_panel} \cdot 2} = 12.807 \frac{m}{s}$$

$$Re_{ss} := \frac{\rho_{ss}(T_{ss_hst}) \cdot u_{ss} \cdot D_i}{\mu_{ss}(T_{ss_hst})} = 2.347 \times 10^5$$

$$\varepsilon := 0.002mm$$

$$m_{dot_summer} := m_{hst_in}$$

$$f_D := \frac{0.25}{\left(\log\left(\frac{5.74}{Re_{ss}} + \frac{\varepsilon}{3.7 \cdot D_i}\right)\right)^2} = 0.014$$

$$K_i := 1.0$$

$$K_e := 0.2$$

$$K_{bend} := 0.16 \qquad \text{(Miller)}$$

$$\Delta p_{pipe} := f_D \cdot \frac{H_{SR} \cdot \rho_{ss}(T_{ss_hst}) \cdot u_{ss}^2}{2D_i} = 166.853 \cdot kPa$$

$$\Delta p_{secondary} := (2 \cdot K_{bend} + 2K_i + 2K_e) \cdot \left(\rho_{ss}(T_{ss_hst}) \cdot \frac{u_{ss}^2}{2}\right) = 96.502 \cdot kPa$$

$$\Delta p_{panel} := \Delta p_{pipe} + \Delta p_{secondary} = 0.263 \cdot MPa$$

$$\Delta p_{total} := \frac{round(N_{panels})}{2} \cdot \Delta p_{panel} = 1.317 \cdot MPa$$

$$W_{pump} := \frac{m_{dot_summer} \cdot \Delta p_{total}}{0.6\, \rho_{ss}(T_{ss_hst})} = 2.179 \cdot MW$$

$$F_{power} := \frac{W_{pump} \cdot 100}{50MW} = 4.357$$

$$m_{TES} := m_{hst_out} \cdot 12hr = 40902.746 \cdot ton$$

$$V_{hst} := \frac{m_{TES}}{\rho_{ss}(T_{ss_hst})} = 21441.758 \cdot m^3$$

$$D_{hst} := \left(\frac{4 \cdot V_{hst}}{\pi}\right)^{\frac{1}{3}} = 30.111\, m$$

Appendix J. FLiBe heat exchanger calculations

$kelv := 273.15 \qquad kJ := 1000 \cdot J$

$$\mu_{ss}(T_{ss}) := \left[1.16 \cdot 10^{-4} e^{\frac{3755}{\left(\frac{T_{ss}}{K}\right)}} \right] Pa \cdot s$$

$$\rho_{ss}(T_{ss}) := \left(2413.03 - 0.7324 \cdot \frac{T_{ss}}{K} \right) \frac{kg}{m^3}$$

$$k_{ss} := 1.1 \frac{W}{m \cdot K}$$

$$Cp_{ss}(T_{ss}) := (2385) \frac{J}{kg \cdot K}$$

$$v_{ss}(T_{ss}) := \frac{\mu_{ss}(T_{ss})}{\rho_{ss}(T_{ss})}$$

$Q_{mh} := 91.559MW$

$p_{mh_in} := 24.903MPa$

$T_{co2_mh_in} := 412.5\,°C$

$m_{co2} := 532.6 \frac{kg}{s}$

$p_{co2_mh_out} := 24.429kPa$

$T_{co2_mh_out} := 550\,°C$

$Q_{rh} := 45.779MW$

$p_{co2_mh_in} := 13.958MPa$

$T_{co2_rh_in} := 479.5\,°C$

$p_{co2_rh_out} := 13.679MPa$

$T_{co2_rh_out} := 550\,°C$

$$T_{ss_max} := 740\,°C$$

$$T_{ss_mh_in} := T_{ss_max}$$

$$T_{ss_rh_in} := T_{ss_mh_in}$$

$$cp_{co2_mh} := 1.252068\,\frac{kJ}{kg\cdot K}$$

$$cp_{co2_rh} := 1.218275\,\frac{kJ}{kg\cdot K}$$

$$C_{c_mh} := m_{co2}\cdot cp_{co2_mh} = 0.667\cdot\frac{MW}{K}$$

$$\Delta T_{co2_mh} := T_{co2_mh_out} - T_{co2_mh_in} = 137.5\,K$$

$$T_{ss_mh_out} := T_{ss_mh_in} - 240K = 500\cdot°C$$

$$m_{ss_mh} := \frac{Q_{mh}}{Cp_{ss}\left(\frac{T_{ss_mh_out} + T_{ss_mh_in}}{2}\right)\cdot\left(T_{ss_mh_in} - T_{ss_mh_out}\right)} = 159.956\,\frac{kg}{s}$$

$$C_{h_mh} := m_{ss_mh}\cdot Cp_{ss}\left(\frac{T_{ss_mh_out} + T_{ss_mh_in}}{2}\right) = 0.381\cdot\frac{MW}{K}$$

$$Q_{mh_max} := C_{h_mh}\cdot\left(T_{ss_mh_in} - T_{co2_mh_in}\right) = 124.94\cdot MW$$

$$Q_{mh} = 91.559\cdot MW$$

$$\varepsilon_{mh} := \frac{Q_{mh}}{Q_{mh_max}} = 0.733$$

$$C_{c_rh} := m_{co2}\cdot cp_{co2_rh} = 0.649\cdot\frac{MW}{K}$$

$$\Delta T_{co2_rh} := T_{co2_rh_out} - T_{co2_rh_in} = 70.5\,K$$

$$T_{ss_rh_out} := T_{ss_rh_in} - 240K = 500\cdot°C$$

$$m_{ss_rh} := \frac{Q_{rh}}{Cp_{ss}\left(\frac{T_{ss_rh_out} + T_{ss_rh_in}}{2}\right)\cdot\left(T_{ss_rh_in} - T_{ss_rh_out}\right)} = 79.977\,\frac{kg}{s}$$

$$C_{h_rh} := m_{ss_rh}\cdot Cp_{ss}\left(\frac{T_{ss_rh_out} + T_{ss_rh_in}}{2}\right) = 0.191\cdot\frac{MW}{K}$$

$$Q_{rh_max} := C_{h_rh}\cdot\left(T_{ss_rh_in} - T_{co2_rh_in}\right) = 49.689\cdot MW$$

$$Q_{rh} = 45.779\cdot MW$$

$$\varepsilon_{rh} := \frac{Q_{rh}}{Q_{rh_max}} = 0.921$$

$$T_{ss_cst} := \frac{T_{ss_mh_out}\cdot m_{ss_mh} + T_{ss_rh_out}\cdot m_{ss_rh}}{m_{ss_mh} + m_{ss_rh}} = 500\cdot°C$$

$$T_{ss_hst} := T_{ss_mh_in} = 740\cdot°C$$

$$m_{hst_out} := m_{ss_mh} + m_{ss_rh} = 239.934\,\frac{kg}{s}$$

$$Q_{hx} := Q_{mh} + Q_{rh} = 137.338\cdot MW$$

$\eta_{SR} := 0.85$ (Assumed to be lower than calculated such that other inefficiencies are accounted for)

$TES := 12hr$ Thermal energy storage

$T_{summer} := 12hr$ Number of hours of sunlight in summer

$$m_{hst_in} := m_{hst_out}\cdot\left(1 + \frac{TES}{T_{summer}}\right) = 479.867\,\frac{kg}{s}$$

$$Q_{SR} := m_{hst_in}\cdot Cp_{ss}\left(\frac{T_{ss_hst} + T_{ss_cst}}{2}\right)\cdot\left(T_{ss_hst} - T_{ss_cst}\right)\cdot\frac{1}{\eta_{SR}} = 323.148\cdot MW$$

$$Q_{flux_heliostats} := 0.881 \frac{MW}{m^2}$$

$$A_{SR} := \frac{Q_{SR}}{Q_{flux_heliostats}} = 366.797\,m^2$$

$$R_{SR} := 1.499$$

$$D_{SR} := \sqrt{\frac{A_{SR}}{R_{SR} \cdot \pi}} = 8.825\,m$$

$$H_{SR} := D_{SR} \cdot R_{SR} = 13.229\,m$$

Tubular geometry:

$$D_o := 42.2mm \qquad t := 2mm$$

$$D_i := D_o - 2 \cdot t = 38.2 \cdot mm$$

$B_{SR} := 46mm$ tubular pitch

$$N_{tubes_per_panel} := 30$$

$$B_{panel} := B_{SR} \cdot N_{tubes_per_panel} = 1.38\,m$$

$$N_{panels} := \frac{\pi \cdot D_{SR}}{B_{panel}} = 20.091$$

$$A_i := \frac{\pi \cdot D_i^2}{4} = 1.146 \times 10^{-3}\,m^2$$

$$u_{ss} := \frac{m_{hst_in}}{\rho_{ss}(T_{ss_cst}) \cdot A_i \cdot N_{tubes_per_panel} \cdot 2} = 3.779 \frac{m}{s}$$

$$Re_{ss} := \frac{\rho_{ss}(T_{ss_hst}) \cdot u_{ss} \cdot D_i}{\mu_{ss}(T_{ss_hst})} = 5.109 \times 10^4$$

$$\varepsilon := 0.002mm$$

$$m_{dot_summer} := m_{hst_in}$$

$$f_D := \frac{0.25}{\left(\log\left(\frac{5.74}{Re_{ss}} + \frac{\varepsilon}{3.7 \cdot D_i}\right)\right)^2} = 0.016$$

$$K_i := 1.0$$

$$K_e := 0.2$$

$$K_{bend} := 0.16 \qquad \text{(Miller)}$$

$$\Delta p_{pipe} := f_D \cdot \frac{H_{SR} \cdot \rho_{ss}(T_{ss_hst}) \cdot u_{ss}^2}{2D_i} = 67.98 \cdot kPa$$

$$\Delta p_{secondary} := (2 \cdot K_{bend} + 2K_i + 2K_e) \cdot \left(\rho_{ss}(T_{ss_hst}) \cdot \frac{u_{ss}^2}{2}\right) = 32.448 \cdot kPa$$

$$\Delta p_{panel} := \Delta p_{pipe} + \Delta p_{secondary} = 0.1 \cdot MPa$$

$$\Delta p_{total} := \frac{round(N_{panels})}{2} \cdot \Delta p_{panel} = 1.004 \cdot MPa$$

$$W_{pump} := \frac{m_{dot_summer} \cdot \Delta p_{total}}{0.6 \, \rho_{ss}(T_{ss_hst})} = 0.481 \cdot MW$$

$$F_{power} := \frac{W_{pump} \cdot 100}{50MW} = 0.961$$

$$m_{TES} := m_{hst_out} \cdot 12hr = 11425.602 \cdot ton$$

$$V_{hst} := \frac{m_{TES}}{\rho_{ss}(T_{ss_hst})} = 6.203 \times 10^3 \cdot m^3$$

$$D_{hst} := \left(\frac{4 \cdot V_{hst}}{\pi}\right)^{\frac{1}{3}} = 19.915 \, m$$