FACULTY OF ENGINEERING, LUND UNIVERSITY

MASTER'S THESIS

Implementing spatial variance in the rate of photolytic breakdown of NO_2 in urban street canyons

---

Author:
Fabian FRIBERG

Supervisors:
Hesameddin Fatehi
Elna Heimdal Nilsson

A thesis submitted in fulfillment of the requirements for the degree of Master's degree in Computer Science

in the

Department of Energy Sciences

April 13, 2022

Declaration of Authorship

I, Fabian FRIBERG, declare that this thesis titled, "Implementing spatial variance in the rate of photolytic breakdown of NO_2 in urban street canyons" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at Lund University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

FACULTY OF ENGINEERING, LUND UNIVERSITY

Abstract

Faculty of Engineering

Department of Energy Sciences

Master's degree in Computer Science

Implementing spatial variance in the rate of photolytic breakdown of NO_2 in urban street canyons

by Fabian FRIBERG

In this study a module for calculating the solar exposure of individual fluid cells was developed to more accurately represent the effect of photolysis on the pollutant dispersion in an urban street canyon. The module was tested and validated on 12 different cases, each with a different geographical location, time of year and/or time of day. It was used for four parallel simulations performed on a street canyon model representing a real-life street canyon in Gothenburg, Sweden, each one using a different model for NO_2 photolysis. NO_x and O_3 levels were measured and compared at street level, mid-canyon level, and sky level, with results showing negligible difference between models using partial sunlight and no sunlight, while showing significant difference between the partial sunlight model and the uniform sunlight model. We conclude from our results that simulations without any modeling of photolytic reactions are sufficient for accurately predicting NO and O_3 levels, but fail to accurately predict NO_2 levels, while simulations utilizing a solar model with uniform photolysis throughout the simulated domain are sufficient for accurately predicting NO and NO_2 levels, but fail to accurately predict O_3 levels. For any study placing an emphasis on both NO_2 and O_3 levels within the domain, it is our recommendation to incorporate a non-uniform solar model. Such a model was developed for this thesis, and aims to be integrated into the OpenFOAM framework in the future.

Acknowledgements

I would like to thank my main supervisor Hesameddin Fatehi for giving me consistent guidance and support during the whole project period, and for being thoroughly engaged throughout the whole process, answering the many questions that popped up during these turbulent months.

I would also like to thank my supervisor Elna Heimdal Nilsson for giving me engaging and informative lectures on atmospheric chemistry, for providing me with useful references and for writing well-written templates for my Introduction and Previous Work chapters. These allowed me to gain a better understanding of both the project and thesis-writing in general.

Lastly I would like to thank my mother for very actively helping me stay motivated through the toughest times of the project, and for providing support in various ways throughout. I could not have done this without you.

Contents

Declaration of Authorship iii

Abstract v

Acknowledgements vii

1 Introduction 1

2 Previous Work 3

3 Chemistry 5

4 Basics of CFD 7
- 4.1 Governing equations 7
  - 4.1.1 Continuity equation 7
  - 4.1.2 Momentum equations 8
  - 4.1.3 Energy equation 8
  - 4.1.4 Equations of state 9
- 4.2 Transport equation 10
- 4.3 Turbulence 10
- 4.4 Radiation 10

5 OpenFOAM 13
- 5.1 rhoEqn.H 13
- 5.2 UEqn.H 14
- 5.3 pEqn 18
- 5.4 EEqn 21
- 5.5 YEqn 22

6 Radiation models in OpenFOAM 25
  - 6.0.1 FvDOM 25
  - 6.0.2 viewFactor 26
  - 6.0.3 P1 26
- 6.1 solarLoad 26
    - updateHitFaces() 28
    - updateDirectHitRadiation() 30
    - updateSkyDiffusiveRadiation() 31
    - updateReflectedRays() 34
  - 6.1.1 solarCalculator 36
    - sunDirConstant 37
    - sunDirTracking 37
    - sunLoadConstant 37
    - sunLoadFairWeatherConditions 37

sunLoadTheoreticalMaximum 38

7 Methodology 39

7.1 Implementing the Photolytic Solver 39

7.2 Simulation Setup 44

7.3 Resources 47

8 Results 49

8.1 Validation 49

8.1.1 Cyclic boundary algorithm 49

8.1.2 J field solver 49

8.1.3 Photolytic reaction rate 51

8.1.4 Thermal parameter calibration 52

8.2 Simulation results 54

9 Conclusion 61

A Validation of cyclic J Field solarLoad library 63

B Simulations using different A and Ta values vs. Experimental Data 67

Bibliography 75

List of Symbols

α_{eff}	Thermal conductivity	$\mathrm{W\,m^{-1}\,K^{-1}}$
Γ	Diffusive constant	$\mathrm{kg\,m\,mol^{-1}\,s^{-1}}$
k	Turbulent kinetic energy	$\mathrm{m^2\,s^{-2}}$
μ	Dynamic viscosity	$\mathrm{Pa\,s}$
ν	Kinematic viscosity	$\mathrm{m^2\,s^{-1}}$
p	Pressure	Pa
qr	Radiative heat flux	$\mathrm{m\,s^{-3}}$
ρ	Density	$\mathrm{kg\,m^{-3}}$
T	Temperature	K
τ	Viscous stress	$\mathrm{Pa\,s}$
$\mathbf{u}$	Velocity vector	$\mathrm{m\,s^{-1}}$
u	Velocity in x direction	$\mathrm{m\,s^{-1}}$
v	Velocity in y direction	$\mathrm{m\,s^{-1}}$
w	Velocity in z direction	$\mathrm{m\,s^{-1}}$
S_{Mx}	x-momentum source	$\mathrm{m\,s^{-1}}$
S_{My}	y-momentum source	$\mathrm{m\,s^{-1}}$
S_{Mz}	z-momentum source	$\mathrm{m\,s^{-1}}$
K	Kinetic energy $(u^2 + v^2 + w^2)$	J
i	Internal energy	J
S_i	internal energy source	$\mathrm{J\,s^{-1}}$
Φ	Dissipation equation	$\mathrm{J\,s^{-1}}$

Chapter 1

Introduction

One of the keys to reducing air pollution levels and increasing the air quality of a city lies in its geometrical configuration (Yassin MF, 2012). Certain urban layouts can cause air vortices to be formed between buildings in so called "street canyons", causing pollutant molecules from vehicular combustion and industry to become trapped within, in turn causing these molecules to spend a greater part of their lifecycle near ground level (Yazid et al., 2014). A consequence of this is that reactions with relatively slow reaction times can take place close to the ground, exposing city inhabitants to substances usually found only at higher altitudes (Kikumoto and Ooka, 2012).

In order to comply with air quality directives such as Directive 2008/50/EC of the European Parliament (*Directive 2008/50/EC*), accurate predictive models are of great importance for both industry and regulatory institutions. However, the complexity of factors lending into the production and dispersion of air pollutants makes analytical predictions impossible, while the large spacial scale and variety of possible street and building configurations makes experimental studies prohibitively expensive. Simulations based on CFD (Computational Fluid Dynamics) have a higher level of scalability and configurability compared to experimental models, and can therefore be applied on a wider scale. Experimental data is however still essential for validation of obtained CFD results.

The pollutant substances most relevant for analysis in street canyons are NO, NO_2 and O_3 (Pu and Yang, 2014). These chemicals have known adverse health effects for local residents and pedestrians (Cao et al., 2011). One of the most important reactions involving these chemicals include the reaction $NO_2 + hv- > NO + O$, where hv stands for incident solar irradiation intensity, meaning the reaction is photolytic (Jacob, 1999). Existing CFD simulations modelling NO_x gases have been limited in their calculation of the photolysis rate J, and have therefore had to approximate it.

In cooperation with the Division of Combustion Physics and the Department of Energy Sciences at Lund University of Sciences, this masters' thesis aims to implement a solar module capable of making real-time calculations of local solar intensity values within CFD simulations, improving the predictive accuracy of simulations involving photolytic processes, factoring in both solar intensity and temperature into the breakdown rate of NO_2 into NO and O. It will be tested by comparing NO_x and O_3 concentration levels in an urban street canyon model with the module turned on and off, after which the results will be compared to experimental data.

Chapter 2

Previous Work

CFD simulations using LES (Large Eddy Simulation) turbulence models have long been used to investigate non-reactive air pollutant dispersion in various setups. Baker et al (Baker, Walker, and Cai, 2004) was the first paper to perform LES in a street canyon that included chemical reactions. The authors simulated a street canyon of equal height and width and a constant dispersion of NO_x against a background concentration of O_3 , with a constant overhead wind perpendicular to the street orientation. Proceeding studies have extended this setup. Two examples of this would be Zhong et al. (Zhong, Cai, and Bloss, 2016), incorporating VOCs (Volatile Organic Compounds) and Han et al. (Ming et al., 2020), who studied the effects of roof-level turbulence. Spatially variant photolysis has been studied by Grawe et al (Grawe, Cai, and Harrison, 2007), who simulated the effects of partial sunlight by assigning a triangle-shaped shaded area to their domain (representing a sunlight direction vector of [1 0 -1]), decreasing the ozone breakdown rate in said area in relation to the rest of the domain. Their results clearly demonstrate that the effects of shading are significant for ground level NO_x concentrations. All the papers mentioned up to this point have modeled an isothermal system, with a constant temperature of 293K.

During the writing of this paper a paper by Liu et al. was published (Liu et al., 2021) on the topic of NO_x and O_3 dispersion in a thermally variant street canyon, with time-variant sunlight. Their J value was also calculated according to Shetter et al. (Shetter et al., 1988) and assumed global sunlight exposure.

The general observed trend in most papers is that NO and NO_2 concentrations are high at the proximity of the source points (the ground) and at the leeward side of the street. O_3 tends to enter the inter-building area through the windward wall, rapidly dissipating by reacting with NO. In each case chemical equilibrium is reached in the middle of the main vortex. It has been shown that buoyancy effects are significant at high wind speeds, and that models that include buoyancy should incorporate an LES-type turbulence model for more accurate results (Bohnenstengel et al., 2004) (Tominaga and Stathopoulos, 2011).

Chapter 3

Chemistry

NO_x is a common term for the two chemicals Nitric Oxide (NO) and Nitrogen Dioxide (NO_2). Although their two components, Nitrogen and Oxygen, are very common in the atmosphere, NO_x gases do not appear naturally in significant amounts in our atmosphere. This is because the main reaction they are created by, 3.1 , only occurs at very high temperatures. These temperatures are, however, reached by combustion engines used by automobiles, airplanes and in industry, and so significant concentrations of NO_x will be attained in inner city areas (Jacob, 1999).

$$N_2 + O_2 = 2NO \tag{3.1}$$

In the atmosphere, these Nitric Oxide molecules will participate in the following reactions:

$$NO + O_3 - > NO_2 + O_2 \tag{3.2}$$

$$NO_2 + hv - > NO + O \tag{3.3}$$

$$O + O_2 + M - > O_3 + M \tag{3.4}$$

Where $\dot{O}$ is an Oxygen radical, hv is an absorbed low wavelength photon, and M is an inert molecule of any type, which is required for absorbing excess energy created in the reaction.

In other words, NO depletes ozone in the atmosphere, while NO_2 synthesizes it when exposed to the sun. Total NO_x levels will be at their highest in the daytime when there is more traffic, while the ratio of NO to NO_2 will be higher during the day and lower during the night because of solar radiation levels.

The mixing ratio between NO and NO2 can be calculated as:

$$[NO]/[NO_2] = [O_3] * k1/J * c(air) \tag{3.5}$$

(Jacob, 1999)

k1 is the reaction rate for equation 3.2, which is dependent only on temperature (Lippmann, Jesser, and Schurath, 1980), and the tropospheric value of c(air) is a constant $2.7 * 1019$ (Bannon, 1996), while O_3 concentration will be calculated in real-time. J will be calculated according to 3.6 (Shetter et al., 1988), multiplied by a solar factor given by the simulation.

$$J = 8.14 * 10^{-3}(0.97694 + 8.3700 * 10^{-4} * (T - 273.15) + 4.5173 * 10^{-6} * (T - 273.15)^2)s^{-1} \tag{3.6}$$

In addition to NO_x, O_2 and O_3, chemicals known as VOCs (Volatile Organic Compounds) also affect urban pollution concentrations. Baker (Baker, Walker, and Cai, 2004) considered these negligible, as their reaction times are much longer than those of reactions 3.3, 3.2, and 3.4, which equilibriate on the scale of 1-2 minutes. This notion is, however, contested by Carpenter et al. (Carpenter et al., 1998), who determined that for conditions with lower concentrations of NO_x, peroxy contributes significantly to the creation of NO_2. In a review done by Zhong et al. (Zhong, Cai, and Bloss, 2016) it is also stated that the simplified NO_x/O_3 model often gives inaccurate predictions compared to experimental data. In a study performed by Zhong et al. (Zhong, Cai, and Bloss, 2017), a direct comparison is made between the simplified NO_x/O_3 model and one including VOC modelling. NO_2 and O_3 levels were found to be 30-40% higher in the models including VOCs, with NO levels being lower.

However, even the relatively limited VOC model used by Seinfeld and Pandis (Seinfeld and Pandis, 1998) utilizes 20 reactions and 23 species, which is a significant addition to the four reactions and four species existing in our current model. Considering our time of day is set to be 10 A.M. (when NO_x levels are relatively high because of traffic), limited computational capabilities and our emphasis on photolytic effects, we did not include a VOC chemistry model in our simulation. It is therefore important to keep in mind that NO_2 and O_x levels in our simulations could be significantly lower than real life levels.

Chapter 4

Basics of CFD

Computational Fluid Dynamics, abbreviated CFD, is the field of fluid simulation using computers. It acts as a middle ground between analytical and experimental approaches to flow solving. The high level of non-linearity involved in most fluid problems makes analytical solutions infeasible outside of very simple case setups. Experimental studies, while more accurate than CFD simulations, often involve long setup times, heavy costs, low malleability and low reproducibility. Changing a parameter can be done by changing a single line of a file in a CFD setup, while it may take rebuilding a whole architecture when using an experimental approach.

In the following chapter we will go through some basics of CFD. Information was taken from "An Introduction to CFD" (Versteeg and Malalasekera, 1995).

4.1 Governing equations

In CFD, governing equations tell us the rate of change for certain variables of interest, given the current value of other variables as input. They must be solved for each variable, in each cell, for each timestep, in order to properly determine the behaviour of a fluid under certain conditions. The most commonly used governing equations in CFD modelling are those that concern the 3-dimensional flow and heat transfer of a Newtonian compressible fluid. They can be derived from the Navier-Stokes equations, which in turn are derived from simple laws of physics, such as the conservation of mass, Newton's second law (F=ma) and the first law of thermodynamics (the rate of change of energy is equal to the sum of the rate of heat addition to and the rate of work done on a fluid particle).

There are five main governing equations utilized in CFD, conserving the mass, x-momentum, y-momentum, z-momentum, and energy of a fluid. They are the continuity equation, momentum equations (in the x, y and z directions) and the energy equation.

4.1.1 Continuity equation

$$\frac{d\rho}{dt} + \mathrm{div}(\rho \mathbf{u}) = 0 \tag{4.1}$$

The continuity equation, also known as the mass balance equation, is based on the law of conservation of mass (in this case density, since we are dealing with fixed volumes). It states that the change of mass in a volume is equal to the mass entering/leaving it via the flow.

4.1.2 Momentum equations

$$\frac{d\rho u}{dt} + \mathrm{div}(\rho u \mathbf{u}) = -\frac{dp}{dx} + \mathrm{div}(\mu \mathrm{grad}(u)) + S_{Mx} \tag{4.2}$$

$$\frac{d\rho u}{dt} + \mathrm{div}(\rho v \mathbf{u}) = -\frac{dp}{dy} + \mathrm{div}(\mu \mathrm{grad}(v)) + S_{My} \tag{4.3}$$

$$\frac{d\rho u}{dt} + \mathrm{div}n(\rho w \mathbf{u}) = -\frac{dp}{dz} + div(\mu \mathrm{grad}(w)) + S_{Mz} \tag{4.4}$$

The three momentum equations in the x, y and z directions tell us that the rate of change in momentum in a certain direction in a volume is given by the momentum in that direction from fluid flowing into/out of the volume + the momentum in that direction given by pressure (the minus sign in front of this term comes from the fact that pressure pushes in the opposite direction of its gradient) + the momentum in that direction applied by shear stresses + momentum in that direction applied by sources. Some solvers will implement gravity by adding a y-momentum source term in the y-momentum equation.

4.1.3 Energy equation

The energy equation looks different depending on whether one solves it using internal energy i or enthalpy $h_0 = i + \frac{p}{\rho}$ + K. They are as follows:

$$\begin{multline}\rho\frac{di}{dt} = -p\mathrm{div}(\mathbf{u} + \mathrm{div}(\alpha_{eff}\mathrm{grad}(T)) \\ +\tau_{xx}\frac{du}{dx} + \tau_{yx}\frac{du}{dy} + \tau_{zx}\frac{du}{dz} + \tau_{xy}\frac{dv}{dx} + \tau_{yy}\frac{dv}{dy} + \tau_{zy}\frac{dv}{dz} + \tau_{xz}\frac{dw}{dx} + \tau_{yz}\frac{dw}{dy} + \tau_{zz}\frac{dw}{dz} + S_i\end{multline} \tag{4.5}$$

$$\begin{multline}\frac{d(\rho h_0)}{dt} + \mathrm{div}(\rho h_0 \mathbf{u}) = \mathrm{div}(\alpha_{eff}\mathrm{grad}(T)) + \frac{dp}{dt} + \frac{d(u\tau_{xx})}{dx} + \frac{d(u\tau_{yx})}{dy} + \frac{d(u\tau_{zx})}{dz} \\ + \frac{d(v\tau_{xy})}{dx} + \frac{d(v\tau_{yy})}{dy} + \frac{d(v\tau_{zy})}{dz} + \frac{d(w\tau_{xz})}{dx} + \frac{d(w\tau_{yz})}{dy} + \frac{d(w\tau_{zz})}{dz} + S_h\end{multline} \tag{4.6}$$

What these equations tell us is that the change rate of internal energy in a volume is equal to the internal energy from fluid flowing into/out of the volume + the potential energy from pressure it receives/loses due to advection + the energy it receives/loses due to thermal conductivity + the thermal energy gained from viscous friction + energy from sources. Some solvers will implement gravity into this equation by adding a potential energy source term.

These equations can be simplified by defining the dissipation Φ as:

$$\Phi = \mu\{2[\frac{du}{dx}^2 + \frac{dv}{dy}^2 + \frac{dw}{dz}^2] \\ +[(\frac{du}{dy} + \frac{dv}{dx})^2 + (\frac{du}{dz} + \frac{dw}{dx})^2 + (\frac{dv}{dz} + \frac{dw}{dy})^2]\} \\ +\lambda(\mathrm{div}(\mathbf{u}))^2 \tag{4.7}$$

Where $\lambda = 2/3\mu$. This term represents kinetic energy being converted into thermal energy by viscous stresses. It is assumed to be negligible for most CFD configurations.

Applying this to the internal energy equation gives:

$$\frac{d(\rho i)}{dt} + \mathrm{div}(\rho i\mathbf{u}) = -p\mathrm{div}(\mathbf{u}) + \mathrm{div}(\alpha_{eff}\mathrm{grad}(T)) + \Phi + S_i \tag{4.8}$$

The continuity equation, momentum equations and energy equation give us the following system of equations:

$$\frac{d\rho}{dt} + div(\rho\mathbf{u}) = 0 \tag{4.1}$$

$$\frac{d\rho u}{dt} + \mathrm{div}(\rho u\mathbf{u}) = -\frac{dp}{dx} + \mathrm{div}(\mu\mathrm{grad}(u)) + S_{Mx} \tag{4.2}$$

$$\frac{d\rho u}{dt} + \mathrm{div}(\rho v\mathbf{u}) = -\frac{dp}{dy} + \mathrm{div}(\mu\mathrm{grad}(v)) + S_{My} \tag{4.3}$$

$$\frac{d\rho u}{dt} + \mathrm{div}(\rho w\mathbf{u}) = -\frac{dp}{dz} + \mathrm{div}(\mu\mathrm{grad}(w)) + S_{Mz} \tag{4.4}$$

$$\frac{d(\rho i)}{dt} + \mathrm{div}(\rho i\mathbf{u}) = -p\mathrm{div}(\mathbf{u}) + \mathrm{div}(\alpha_{eff}\mathrm{grad}(T)) + \Phi + S_i \tag{4.8}$$

We now have five equations and seven unknowns. This discrepancy is due to the pressure p and temperature T not having any equations of their own. This is solved by adding equations of state to the system.

4.1.4 Equations of state

The variables p and i can be expressed as functions of density ρ and temperature T. These functions are called the equations of state. In most configurations the ideal gas law suffices as an approximation for these equations:

$p = \rho RT$

$i = C_v T$

R = ideal gas constant

C_v = heat capacity

With these equations provided, we now have 7 equations and 7 unknowns, and can therefore close the system.

4.2 Transport equation

Since we already have enough information to solve the flow of our model, we can now calculate the rate of change for arbitrary fields carried by the flow, commonly generalized by the letter ϕ. The generalized transport equation for such a field is given by figure 4.9. This equation is most often used to represent the rate of change for the concentration of certain compounds within a fluid.

$$\frac{d\phi}{dt} + \mathrm{div}(\rho\phi\mathbf{u}) = \mathrm{div}(\Gamma \mathrm{grad}(\phi)) + S_\phi \tag{4.9}$$

What this equation says is that the ϕ gained in a unit volume ϕ leaving the unit volume through advection equals the ϕ entering the unit volume through diffusion plus the ϕ generated within the volume.

4.3 Turbulence

Turbulence is a phenomenon that occurs in fluid flows with a high velocity in relation to their flow field and viscosity. It can be measured through its Reynolds number:

$Re = \frac{VL}{v}$

where V is the velocity, L is the characteristic length of the surrounding vessel and v is the kinetic viscosity of the fluid. Turbulent properties are usually found in flows with Re > 2300. Unlike laminar flows, where a constant velocity input always yields a constant velocity output, a turbulent flow is characterized by a velocity output that varies with time, even with a constant input.

Characteristic of turbulence is that a turbulent fluid will contain many eddies scattered throughout the flow, varying greatly in shape and angular momentum. The smallest of these eddies occur on a microscopic level with high velocities and they are therefore very expensive to compute. Only DNS (Detailed Numerical Simulations) simulations attempt to compute turbulent flow contributions all the way to the lowest level. They are generally costly and only applicable for very small and simple cases. Instead RANS and LES models are commonly used to approximate turbulent phenomena with slightly lower accuracy but at a greatly reduced cost.

RANS stands for Reynold's Averaged Navier-Stokes and is a time-averaging turbulence model. It averages the time-variant velocities to calculate a turbulent viscosity contribution in the flow. Common models include the k-epsilon, k-SST omega and SSG.

LES stands for Large Eddy Simulation. In LES simulations, instead of averaging over time like in RANS, one averages over space. Most Eddies are allowed to resolve, but unlike DNS which requires very fine grids, a model is still required for eddies whose resolution is smaller than the grid cells. Common models include Smagorinsky, WALE and kEqn.

4.4 Radiation

Radiation is the transmission of energy between bodies through space in the form of electromagnetic waves. No medium is required for this, which is why the radiative rays from the sun are able to reach the earth despite there being no air between the two bodies.

For a given body, incident radiative energy is either absorbed, reflected, or transmitted, according to its **absorptivity** (a), **reflexivity** (r) and, **transmissivity** (τ) coefficients. These range from 0 to 1, and and have a sum equal to 1 due to the law of conservation of energy.

In most CFD cases, including this one, all boundary surfaces are opaque, meaning $\tau = 0$.

The equation for how much energy a radiative body emits is given by the Stefan-Boltzmann law:

$$j_r = A\epsilon\sigma_{SB}T^4 \tag{4.10}$$

- A is the area of the body.
- ϵ is the **emissivity** of the body. It is defined as the fraction of energy radiated from the body in relation to the amount of energy that would be radiated by an ideal blackbody of equal surface area. Emissivity for a surface depend on wavelength, temperature and material.
- σ is **Stefan-Boltzmann's constant**, defined as approximately $5.67 * 10^{-8} Wm^{-2}K^{-4}$.
- T is the temperature of the body.

We can now set up an equation for the radiative enthalpy contribution h_r:

$$h_r = aG - (\epsilon\sigma_{SB}T^4) \tag{4.11}$$

Where G is equal to the incident radiative energy.

This means that the radiative energy contribution to the energy equation is equal to the absorbed radiative energy (aG) minus the emitted radiative energy $(\epsilon\sigma_{SB}T^4)$.

Chapter 5

OpenFOAM

OpenFOAM was chosen as the simulation tool of choice to be used in this thesis. It is a C++-based open source program capable of handling a variety of flow problems in different environments. It performs calculations based on the governing equations discussed in the previous chapter. Flow problems are solved by dividing the volume of a fluid into a three-dimensional (3D) mesh, where the governing equations are calculated for each cell of that mesh. This is done iteratively for each time step until the flow is defined for the entire time segment specified by the user.

The main loop in OpenFOAM is performed by so-called solvers, excecutable files with various functionality based on what type of flow you want to solve. Different solvers make different assumptions about the flow, including compressibility, existence of radiation, existence of combustion etc. Different input parameters are required by different solvers.

For this case we will modify the rhoReactingBuoyantFoam solver. "rho" means that it handles compressible fluids, "Reacting" means that it handles chemical reactions, and "Buoyant" means that it includes the effects of buoyancy for the flow.

Solvers in OpenFOAM have one header file for each Navier-Stokes equation. In each of these files a matrix is created that perfectly represents its corresponding physical equation in C++ code form. We will now go through each Eqn file of the reactingFoam solver and see how the OpenFOAM solvers set up each matrix and what lines of code correspond to what terms of the Navier-Stokes equations, as described in the previous chapter.

5.1 rhoEqn.H

The continuity equation in reactingFoam is represented by:

```
fvScalarMatrix rhoEqn
(
    fvm::ddt(rho)
  + fvc::div(phi)
  ==
    fvOptions(rho)
);
```

This can be directly converted to the following equation:

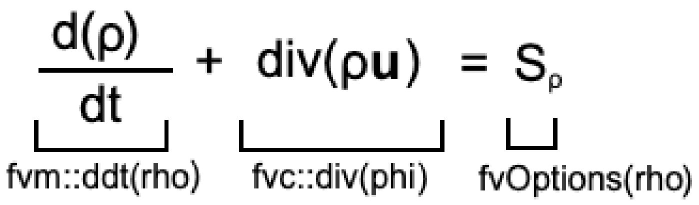


FIGURE 5.1: Continuity equation in mathematical form

Note that phi in OpenFOAM is equal to flux, that is $\rho\mathbf{u}$.

This is already very similar to equation 4.1, except the mass source term has been added. It is often omitted from governing equations as it is uncommon to include a mass source term in fluid calculations.

The two namespaces fvm:: and fvc:: have different meanings. A function in the fvm:: namespace expresses the to-be-solved term ($\frac{d\rho}{dt}$) in an implicit way, that is, it is expressed as a matrix of coefficients to be solved in the current calculation. This is unlike functions in the fvc:: namespace, which return to-be-solved values explicitly (by evaluating source terms and/or terms solved in previous calculations).

5.2 UEqn.H

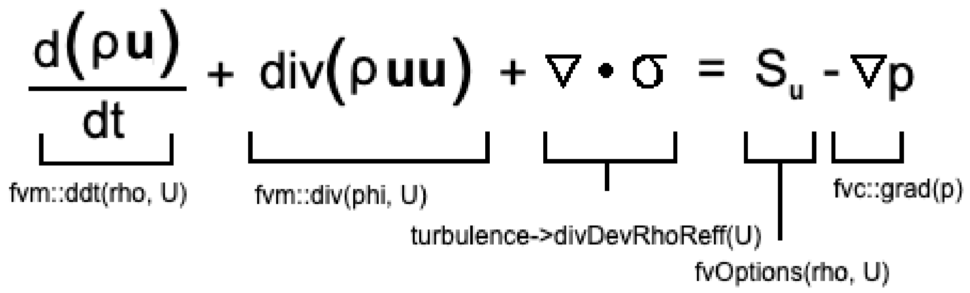
The velocity equation in reactingFoam is represented by:

```
tmp<fvVectorMatrix> tUEqn
(
    fvm::ddt(rho, U) + fvm::div(phi, U)
  + MRF.DDt(rho, U)
  + turbulence->divDevRhoReff(U)
 ==
    fvOptions(rho, U)
);

...

if (pimple.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));
}
```

This can be directly converted to the following equation:



$\frac{d(\rho u)}{dt} + div(\rho \mathbf{uu}) + \nabla \cdot \sigma = S_u - \nabla p$
fvm::ddt(rho, U)
fvm::div(phi, U)
turbulence->divDevRhoReff(U)
fvOptions(rho, U)
fvc::grad(p)

FIGURE 5.2: Momentum equation in mathematical form

MRF stands for multiple reference field, and has to do with rotational phenomena like fans and golf balls. For most purposes it is equal to zero.

The turbulence->divDevRhoReff function returns the density times the effective viscous stress contribution to the momentum equation. Its implementation depends on the turbulence model chosen. The implementation in linearViscousStress.C is as follows:

```
Foam::linearViscousStress<BasicTurbulenceModel>::divDevRhoReff
(
    volVectorField& U
) const
{
    return
    (
      - fvc::div((this->alpha_*this->rho_*this->nuEff())*dev2(T(fvc::grad(U))))
      - fvm::laplacian(this->alpha_*this->rho_*this->nuEff(), U)
    );
}
```

Alpha_ is phase fraction, which for single phase solvers like reactionFoam is equal to 1. For the viscosity, $\mu_{eff} = \mu_t + \mu$.

μ_t is calculated based on the turbulence model, and μ is typically read from the **constant/transportProperties** file.

dev stands for **deviatory**, and is defined by

$\text{dev}(A) = A - \frac{1}{3} * tr(A)I$

dev2 is a slight modification, where $\text{dev2}(A) = A - \frac{2}{3} * tr(A)I$.

From this, divDevRhoEff can be directly converted into the following equation:

$$\underbrace{-\nabla\left(\rho\mu_{eff}(\nabla \mathbf{u})\right)}_{\text{A}} \quad \underbrace{-(\nabla \mathbf{u})^{T} - \tfrac{2}{3}\,\mathrm{tr}\left((\nabla \mathbf{u})^{T}\right)}_{\text{B}}$$

```
A: - fvm::laplacian(this->alpha_*this->rho_*this->nuEff(), U)
B: - fvc::div((this->alpha_*this->rho_*this->nuEff())*dev2(T(fvc::grad(U))))
```

FIGURE 5.3: divDevRhoEff equation in mathematical form

In their book An Introduction to CFD, Versteeg and Malalasekera give us the following equation for the viscous stress in the x-direction:

$$\begin{aligned} \tau_x = \frac{d}{dx}(\mu\frac{du}{dx}) + \frac{d}{dy}(\mu\frac{du}{dy}) + \frac{d}{dz}(\mu\frac{du}{dz}) + \\ [\frac{d}{dx}(\mu\frac{du}{dx}) + \frac{d}{dy}(\mu\frac{dv}{dx}) + \frac{d}{dz}(\mu\frac{dw}{dx}) + \frac{d}{dx}(\lambda \mathrm{div}(\mathbf{u}))] \\ = \mathrm{div}(\mu \mathrm{grad} u) + \\ [\beta_x] \end{aligned} \tag{5.1}$$

Where λ is a variable that for gases is usually approximated to $-\frac{2}{3}\mu$, and β_x is a new variable. Equivalent equations exist for the y and z directions.

The $(\nabla \mathbf{u})^T$ term in our divDevRhoEff equation can be expanded as:

$$(\mathrm{grad}(\mathbf{u}))^T = \begin{bmatrix} \frac{du}{dx} & \frac{du}{dy} & \frac{du}{dz} \\ \frac{dv}{dx} & \frac{dv}{dy} & \frac{dv}{dz} \\ \frac{dw}{dx} & \frac{dw}{dy} & \frac{dw}{dz} \end{bmatrix}^T = \begin{bmatrix} \frac{du}{dx} & \frac{dv}{dx} & \frac{dw}{dx} \\ \frac{du}{dy} & \frac{dv}{dy} & \frac{dw}{dy} \\ \frac{du}{dz} & \frac{dv}{dz} & \frac{dw}{dz} \end{bmatrix} \tag{5.2}$$

We can see that the first row of $grad(\mathbf{u})^T$ (corresponding to the shear stress in the x-direction) matches the inner gradients of the β_x term in equation 5.1.

If we write out the β terms of each direction:

$$\beta_x = \frac{d}{dx}(\mu\frac{du}{dx}) + \frac{d}{dy}(\mu\frac{dv}{dx}) + \frac{d}{dz}(\mu\frac{dw}{dx}) + \frac{d}{dx}(\lambda \mathrm{div}(\mathbf{u})) \tag{5.3}$$

$$\beta_y = \frac{d}{dx}(\mu\frac{du}{dy}) + \frac{d}{dy}(\mu\frac{dv}{dy}) + \frac{d}{dz}(\mu\frac{dw}{dy}) + \frac{d}{dy}(\lambda\mathrm{div}(\mathbf{u})) \quad (5.4)$$

$$\beta_z = \frac{d}{dx}(\mu\frac{du}{dz}) + \frac{d}{dy}(\mu\frac{dv}{dz}) + \frac{d}{dz}(\mu\frac{dw}{dz}) + \frac{d}{dz}(\lambda\mathrm{div}(\mathbf{u})) \quad (5.5)$$

Since $div(\mathbf{u}) = \frac{du}{dx} + \frac{dv}{dy} + \frac{dw}{dz} = tr(grad(\mathbf{u})) = tr(grad(\mathbf{u})^T)$, we can move λ out of the parenthesis, expand it into $-2/3\mu$ and express the last term of each equation as $-\frac{2}{3}\mu\frac{d}{dx}(tr(\mathrm{grad}(\mathbf{u}))^T)$ (except $\frac{d}{dx}$ is replaced with $\frac{d}{dy}$ and $\frac{d}{dz}$ for β_y and β_z respectively).

If we take a look at the first three terms of each β equation (nine terms in total) we can see that they make up the terms of $\mu\mathrm{div}(\mathrm{grad}(\mathbf{u})^T)$ perfectly:

Since $\frac{d}{dx}(A) - \frac{d}{dx}(B) = \frac{d}{dx}(A - B)$, we can put together the first and fourth terms of equation 5.3, the second and fourth terms of equation 5.4 and the third and fourth terms of equation 5.5. Finally, in matrix form, with, we will have our three β equations represented by:

$$\beta =$$
$$\mu\begin{bmatrix} \frac{d}{dx}((\frac{du}{dx}) - \frac{2}{3}(tr(\mathrm{grad}(\mathbf{u}))^T)) & \frac{d}{dy}(\frac{du}{dy}) & \frac{d}{dz}(\frac{du}{dz}) \\ \frac{d}{dx}(\frac{dv}{dx}) & \frac{d}{dy}((\frac{dv}{dy}) - \frac{2}{3}(tr(\mathrm{grad}(\mathbf{u}))^T)) & \frac{d}{dz}(\frac{dv}{dz}) \\ \frac{d}{dx}(\frac{dw}{dx}) & \frac{d}{dy}(\frac{dw}{dy}) & \frac{d}{dz}((\frac{dw}{dz}) - \frac{2}{3}(tr(\mathrm{grad}(\mathbf{u}))^T)) \end{bmatrix}$$
$$= \mu\begin{bmatrix} \frac{d}{dx}(\frac{du}{dx}) & \frac{d}{dy}(\frac{du}{dy}) & \frac{d}{dz}(\frac{du}{dz}) \\ \frac{d}{dx}(\frac{dv}{dx}) & \frac{d}{dy}(\frac{dv}{dy}) & \frac{d}{dz}(\frac{dv}{dz}) \\ \frac{d}{dx}(\frac{dw}{dx}) & \frac{d}{dy}(\frac{dw}{dy}) & \frac{d}{dz}(\frac{dw}{dz}) \end{bmatrix} - \mu\begin{bmatrix} \frac{d}{dx}\frac{2}{3}(tr(\mathrm{grad}(\mathbf{u}))^T) & 0 & 0 \\ 0 & \frac{d}{dy}\frac{2}{3}(tr(\mathrm{grad}(\mathbf{u}))^T) & 0 \\ 0 & 0 & \frac{d}{dz}\frac{2}{3}(tr(\mathrm{grad}(\mathbf{u}))^T) \end{bmatrix}$$
$$= \mathrm{div}(\mu\mathrm{grad}(\mathbf{u})^T) - \mathrm{div}(\mu\frac{2}{3}\mathrm{tr}(\mathrm{grad}(\mathbf{u}))^T)))\mathbb{I}$$
$$= \mathrm{div}(\mu\mathrm{dev2}(\mathrm{grad}(\mathbf{u})^T)) \quad (5.6)$$

If we once again take a look at the divDevRhoEff function:

```
- fvc::div((this->alpha_*this->rho_*this->nuEff())*dev2(T(fvc::grad(U))))
- fvm::laplacian(this->alpha_*this->rho_*this->nuEff(), U)
```

β corresponds to the first row. The minus signs come from the equation being moved to the other side of the = sign. this->alpha_ is equal to 1 in mono-phase solvers, and the factor ρ is multiplied with the equation to convert the unit from kinematic to dynamic viscosity. The second row corresponds to the $\mathrm{div}(\mu\mathrm{grad}u)$ term in equation 5.1, except the x-direction component u has been replaced by the complete flow vector **u** so that it includes the y and z directions.

Looking once again at figure 5.2, we can see that divDevRhoEff is represented by $\nabla\sigma$, where σ is the Newtonian stress tensor, and is equal to $\mu\rho(\mathrm{grad}\mathbf{u} + \mathrm{dev2}(\mathrm{grad}(\mathbf{u})^T)$.

After splitting the equation in figure 5.2 into its three components, expanding $\nabla\sigma$ and moving it to the RHS we get the equations:

$$\frac{d\rho u}{dt} + \mathrm{div}(\rho u \mathbf{u}) = -\frac{dp}{dx} + \mu\rho(\mathrm{grad}u + \mathrm{dev2}(\mathrm{grad}(\mathbf{u})^T) + S_{Mx} \tag{5.7}$$

$$\frac{d\rho u}{dt} + \mathrm{div}(\rho v \mathbf{u}) = -\frac{dp}{dy} + \mu\rho(\mathrm{grad}v + \mathrm{dev2}(\mathrm{grad}(\mathbf{u})^T) + S_{My} \tag{5.8}$$

$$\frac{d\rho u}{dt} + \mathrm{div}n(\rho w \mathbf{u}) = -\frac{dp}{dz} + \mu\rho(\mathrm{grad}w + \mathrm{dev2}(\mathrm{grad}(\mathbf{u})^T) + S_{Mz} \tag{5.9}$$

These are almost identical to the momentum equations 4.2, 4.3 and 4.4, except with the addition of the $\mathrm{dev2}(\mathrm{grad}(\mathbf{u})^T)$ term (corresponding to turbulence) and the added ρ factor (to convert to dynamic viscosity).

5.3 pEqn

The pressure equation in OpenFOAM is represented by:

```
    rho = thermo.rho();

volScalarField rAU(1.0/UEqn.A());
surfaceScalarField rhorAUf("rhorAUf", fvc::interpolate(rho*rAU));
volVectorField HbyA(constrainHbyA(rAU*UEqn.H(), U, p));

...

{
    surfaceScalarField phiHbyA
    (
        "phiHbyA",
        (
            fvc::flux(rho*HbyA)
          + MRF.zeroFilter(rhorAUf*fvc::ddtCorr(rho, U, phi))
        )
    );

...

    while (pimple.correctNonOrthogonal())
    {
        fvScalarMatrix pEqn
        (
            fvm::ddt(psi, p)
          + fvc::div(phiHbyA)
          - fvm::laplacian(rhorAUf, p)
         ==
            fvOptions(psi, p, rho.name())
        );

        pEqn.solve();
```

```
         if (pimple.finalNonOrthogonalIter())
         {
             phi = phiHbyA + pEqn.flux();
         }
     }
 }

...

 U = HbyA - rAU*fvc::grad(p);
```

This can be directly converted to the following equation:

$$\underbrace{\nabla\left(\rho A^{-1}{}_{f}\,(\nabla p)\right)}_{\text{fvm::laplacian(rhorAUf, p)}} = -\underbrace{\frac{d(\psi p)}{dt}}_{\text{fvm::ddt(psi, p)}} - \underbrace{\mathrm{div}(\rho \mathbf{u} A^{-1} H)_{f}}_{\text{fvc::div(phiHbyA)}} + \underbrace{S_{\rho_f}}_{\text{fvOptions(psi, p, rho.name())}}$$

FIGURE 5.4: Pressure equation in mathematical form

The PEqn.H file first solves the pressure p iteratively and predicts the velocity U in accordance with the PISO loop. Defining the matrix M as the coefficient matrix for the momentum equation excluding the pressure term, we have the equation **MU** = $-\nabla p$. From this the following matrices are defined:

A = diag(M)
H = AU - MU
rAU = A^{-1}
rhorAUf = ρA^{-1} (evaluated at the faces)
HbyA = $A^{-1}H$ phiHbyA = flux(rho*HbyA)

A is the diagonal of M. It represents the pressure coefficients of each cell from its own velocity value. The H matrix represents the pressure contributions that each cell recieves from its neighbours.

Looking at the at the pressure correction loop:

```
    while (pimple.correctNonOrthogonal())
     {
         fvScalarMatrix pEqn
         (
             fvm::ddt(psi, p)
           + fvc::div(phiHbyA)
           - fvm::laplacian(rhorAUf, p)
```

```
          ==
            fvOptions(psi, p, rho.name())
        );

        pEqn.solve();

        if (pimple.finalNonOrthogonalIter())
        {
            phi = phiHbyA + pEqn.flux();
        }
    }
```

The loop will iterate a certain amount of times based on how orthogonal the matrix is, in order to iteratively calculate the fluxes. For each iteration, the term fvc::div(phiHbyA) will change, since it is calculated explicitly and depends on p. psi = $\psi = \frac{rho}{p}$, meaning

the ddt term is the same as $\frac{d\rho}{dt}$ Note that since we are solving for ∇p, which depends

on the flux on cell faces, we are first solving for cell faces rather than cell centers. Because of this, our terms are multiplied by ρ in order to preserve consistency with the continuity equation when calculating the divergence of phiHbyA.

The equation in 5.4 looks similar to the discretized cell-wise equation in "A low-Mach number solver for variable density flows" (A. Hay and Nilsson, 2018) (Ch. 1.4.1):

$$\underbrace{\sum_f \left(\left(\frac{\rho}{a_p} \right)_f (\nabla p'_d)_f \right) \cdot \mathbf{S}_f}_{fvm::laplacian(rhorAUf,pd)} = \underbrace{\sum_f \left(\left(\frac{\rho}{a_p} H(U) \right)_f - \left(\frac{\rho}{a_p} \right)_f (\mathbf{g} \cdot \mathbf{h})_f \left(\nabla \rho \right)_f \right) \cdot \mathbf{S}_f}_{fvc::div(phiHbyA)} + \underbrace{\frac{\partial \psi}{\partial t} p^0 \cdot V}_{fvc::ddt(psi)*p} \tag{1.17}$$

Since our equation came from the standard reactingFoam solver which does not include buoyancy, and since reactingFoam doesn't use a low-mach approximation, the gh term is missing and the pressure terms are slightly different.

In the paper, the above equation is derived from the following equation:

$$a_p U_p + \sum_N a_N U_N - \frac{U^0}{\Delta t} = -\nabla p'_d - \mathbf{g} \cdot \mathbf{h} \nabla \rho$$

The right hand side of that equation is equivalent to the $-\nabla p$ in compressible buoyant solvers.

Going from cell-wise notation to matrix notation, we get:

$$a_p U_p = AU$$

and

$$\sum_{n=1} a_N U_N - \frac{U^0}{dt} = -H$$

(as per Ch. 1.4.1 in the paper) together with the fact that

AU - H = MU

we get that

MU = -∇p Which is how M was defined. In this way we have derived the pressure

matrix equation from the OpenFOAM source code in the pEqn.H file.

After the pressure has been calculated, the density is calculated using the continuity equation. Then the new pressure is used to calculate a new U vector, by setting

```
    U = HbyA - RAU*fvc::grad(p);
```

5.4 EEqn

The energy equation in reactingFoam is represented by:

```
{
     fvScalarMatrix EEqn
     (
         fvm::ddt(rho, he) + mvConvection->fvmDiv(phi, he)
       + fvc::ddt(rho, K) + fvc::div(phi, K)
       + (
             he.name() == "e"
           ? fvc::div
             (
                 fvc::absolute(phi/fvc::interpolate(rho), U),
                 p,
                 "div(phiv,p)"
             )
           : -dpdt
         )
       - fvm::laplacian(turbulence->alphaEff(), he)
      ==
         reaction->Qdot()
       + fvOptions(rho, he)
     );
 }
```

This can be directly converted to the following equation:

$$\underbrace{\frac{d(\rho h)}{dt}}_{A} + \underbrace{\mathrm{div}(\rho \mathbf{u} h)}_{B} + \underbrace{\frac{d(\rho K)}{dt}}_{C} + \underbrace{\mathrm{div}(\rho \mathbf{u} K)}_{D} - \underbrace{\frac{dp}{dt}}_{E} - \underbrace{\nabla(k(\nabla T))}_{F} = \underbrace{RT}_{G} + \underbrace{S_h}_{H}$$

A: fvm::ddt(rho, he)
B: mvConvection->fvmDiv(phi, he)
C: fvc::ddt(rho, K)
D: fvc::div(phi, K)
E: dpdt
F: fvm::laplacian(turbulence->alphaEff(), he)
G: reaction->Qdot()
H: fvOptions(rho, he)

FIGURE 5.5: Energy equation in mathematical form

This is assuming we are using the enthalpy energy equation, where he.name() == "h". RT is the temperature gained from chemical reactions.

Since $h_0 = h + K$ we have $\frac{d\rho h}{dt} + \frac{d\rho K}{dt} = \frac{d\rho h_0}{dt}$ and $\mathrm{div}(h) + \mathrm{div}(K) = \mathrm{div}(h_0)$ Moving the -dpdt term and the diffusive term to the RHS we get:

$$\frac{d(\rho h_0)}{dt} + \mathrm{div}(\rho \mathbf{u} h_0) = \frac{dp}{dt} + \nabla(k(\nabla T)) + RT + S_h \tag{5.10}$$

This is the same as equation 4.6 except that the dissipation term has been ignored and we have an extra reaction term RT. Since there is no radiation term in this solver's energy equation file, we added it ourselves, by adding the term radiation->Sh() to the right-hand side.

5.5 YEqn

YEqn.H solves the scalar transport equation for the concentration of chemical compounds. It is represented by:

```
{
    reaction->correct();
    volScalarField Yt(0.0*Y[0]);

    forAll(Y, i)
    {
        if (i != inertIndex && composition.active(i))
        {
            volScalarField& Yi = Y[i];

            fvScalarMatrix YiEqn
            (
                fvm::ddt(rho, Yi)
              + mvConvection->fvmDiv(phi, Yi)
              - fvm::laplacian(turbulence->muEff(), Yi)
             ==
        reaction->R(Yi)
      + fvOptions(rho, Yi)
      );

      YiEqn.relax();

      fvOptions.constrain(YiEqn);

      YiEqn.solve("Yi");

      fvOptions.correct(Yi);

      Yi.max(0.0);
      Yt += Yi;
    }
  }

  Y[inertIndex] = scalar(1) - Yt;
  Y[inertIndex].max(0.0);
}
```

This can be directly converted to the following equation:

$$\underbrace{\frac{d(\rho\varphi)}{dt}}_{A} + \underbrace{div(\rho \mathbf{u} \varphi)}_{B} - \underbrace{\nabla\left(\Gamma(\nabla\varphi)\right)}_{C} = \underbrace{RR}_{D} + \underbrace{S_{\varphi}}_{E}$$

A: fvm::ddt(rho, Yi)
B: mvConvection->fvmDiv(phi, Yi)
C: fvm::laplacian(turbulence->muEff(), Yi)
D: reaction->R(Yi)
E: fvOptions(rho, Yi)

FIGURE 5.6: Compund concentration equation in mathematical form

inertIndex represents the index for inert mass, i.e. the fluid molecules that are not explicit components in any reaction, but might act as energy receptors for the reactions of other molecules.

Y_i represents the mass fraction of a certain species, that is $\frac{\rho_i}{\rho}$.

We can easily see how it resembles the scalar transport equation defined in equation 4.9. The only difference is the RR term, which represents the reaction source.

Chapter 6

Radiation models in OpenFOAM

Radiation in OF is added into the energy equation through the radiation->Sh(thermo) term in the EEqn.H file:

```
Foam::tmp<Foam::fvScalarMatrix> Foam::radiation::radiationModel::Sh
 (
     const basicThermo& thermo,
     const volScalarField& he
 ) const
 {
     const volScalarField Cpv(thermo.Cpv());
     const volScalarField T3(pow3(T_));

     return
     (
         Ru()
       - fvm::Sp(4.0*Rp()*T3/Cpv, he)
       - Rp()*T3*(T_ - 4.0*he/Cpv)
     );
 }
```

The most widely used models are **S2S**, **FvDOM** and **P1**. They each implement the **Ru()** and **Rp()** functions, which are the constant and temperature-varying contributions to **Sh()**, respectively. The equation looks complicated at first glance, but the fvm::Sp() row and parts of the third row are simply an addition and subtraction of the same term in order to increase diagonal dominance. It's made more clear with the shifting of some terms:

$$Sh() = Ru() - (4Rp * \frac{T^3 h}{C_p}) - Rp()T^4 + (4Rp * \frac{T^3 h}{C_p}) = Ru() - Rp()T^4$$

The radiation models will in different ways calculate Ru() and Rp() such that they make Sh() represent the radiative enthalpy contribution h_r from equation 4.11 as accurately as possible.

A brief summary of the different available radiation models available in OpenFOAM will be discussed below.

6.0.1 FvDOM

The FvDOM (Finite volume Discrete Ordinates Model) model divides emitted rays into a discrete number of solid angles, and solves them through ray-tracing. It is a computationally expensive but accurate radiation model.

6.0.2 viewFactor

The viewFactor model calculates, for each boundary face of the mesh, a view factor coefficient for each other face in the mesh, determined by how "visible" the faces are to each other. The incident radiation for a face is then calculated by adding together the emitted radiative energy of all faces visible to that face, weighted by their view factor. The model requires a lot of memory space to store the view factors, especially for finer meshes.

6.0.3 P1

The P1 model assumes directional equilibrium between rays, and solves for radiative energy contribution as if it were a diffusive attribute. It works best for cases where the optical thickness (a * L) is high, i.e. where a substantial fraction of a beam's energy is absorbed by the fluid before reaching a boundary.

It sets incident radiation G as a variable and uses the following transport equation to solve ∇G:

$\nabla * (\Gamma \nabla G) - aG + 4\epsilon\sigma T^4 + E = 0$

Which can be found in the P1.C file as:

```
// Solve G transport equation
solve
(
    fvm::laplacian(gamma, G_)
    - fvm::Sp(a_, G_)
    ==
    - 4.0*(e_*physicoChemical::sigma*pow4(T_) ) - E_
);
```

Here, **e_** is the emmissivity, **sigma** is the Stefan-Boltzmann constant and **E_** is the emission contribution.

Gamma is the "diffusive" coefficient for the P1 radiation, similar to the thermal conductivity coefficient k in the general thermal governing equation, and in the code is calculated to equal the following function:

```
1.0/(3.0*a_ + sigmaEff + a0)
```

Here sigmaEff represents the scattering coefficient, which tells us how much ray intensity fades for each unit length travelled through the fluid.

6.1 solarLoad

The solarLoad library, implemented for the OpenFOAM Foundation's ditribution of OpenFOAM, is used to calculate the face-wise solar radiation contribution in a CFD mesh. The direction of the sunbeams is either set to a constant direction or is calculated dynamically with the included solarCalculator class. The solar intensity is either set directly by the user or calculated using the Fair Weather Conditions method from the ASHRAE Handbook (Heating and Engineers, 2017).

The solarLoad radiation model can either act as a standalone model or be added onto the viewFactor and FvDOM radiation models by setting **useSolarLoad** to **true** in **constant/radiationProperties**. The parameters for your solar model are then set in the **solarLoadCoeffs** dictionary in the same file. If combining solarLoad with FvDOM, you can only use one frequency band in the domain.

Looking at solarLoad.C's Rp() function in solarLoad.C:

```
Foam::tmp<Foam::volScalarField> Foam::radiation::solarLoad::Rp() const
{
    return tmp<volScalarField>::New
    (
        IOobject
        (
            "Rp",
            mesh_.time().timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::NO_WRITE,
            false
        ),
        mesh_,
        dimensionedScalar
        (
            dimMass/pow3(dimTime)/dimLength/pow4(dimTemperature),
            Zero
        )
    );
}
```

We can see that its Rp() function returns zero. This is because the Rp() coefficient represents the part of the radiation contribution that scales with T^4. Since the radiative energy given by the solarLoad radiation model depends only on the sun and not on the temperatures within the mesh, the solarLoad model only contributes to the Ru() part of Sh().

The Ru value is calculated with the calculate() function:

```
void Foam::faceShading::calculate()
{

...

bool facesChanged = updateHitFaces();

...

updateDirectHitRadiation(hitFacesId, includeMappedPatchBasePatches);

...

updateSkyDiffusiveRadiation
        (
            includePatches,
            includeMappedPatchBasePatches
        );

...

if (useReflectedRays_)
        {
            updateReflectedRays(includeMappedPatchBasePatches);
        }

...

}
```

It calls on 4 sub-functions, described below.

updateHitFaces()

This sub-function recalculates the sun's position and subsequently determines the boundary faces that are directly hit by sunlight based on the new position by calling hitFaces_->correct(). The hit faces are stored in the hitFaces_ object, which is an instance of the class faceShading, defined in faceShading.C.

```
bool Foam::radiation::solarLoad::updateHitFaces()
{

    ...

        if (updateIndex > updateTimeIndex_)
        {
            Info << "Updating Sun position..." << endl;
            updateTimeIndex_ = updateIndex;
            solarCalc_.correctSunDirection();
            hitFaces_->direction() = solarCalc_.direction();
            hitFaces_->correct();
            return true;
        }

    ...

}

void Foam::faceShading::correct()
{
    calculate();
}

void Foam::faceShading::calculate()
{

    ...

        if (((direction_ & nf) > 0) && (t[faceI] == 0.0))
        {
            dynFacesI.append(faceI + pp.start());
            dynCf.append(cf[faceI]);
            nFaces++;
        }

    ...
```

First, all the faces whose normal vectors have a component facing away from the sun, determined by **(direction_ & nf > 0)** (& is a dot product operator in OpenFOAM), and whose transparency equal 0, are collected.

```
    ...

    scalar maxBounding = 5.0*mag(mesh_.bounds().max() - mesh_.bounds().min());

    ...

    do
        {
            for (; i < Cfs.size(); i++)
            {
                const point& fc = Cfs[i];

                const label myFaceId = hitFacesIds[i];

                const vector d(direction_*maxBounding);

                start.append(fc - 0.001*d);

                startIndex.append(myFaceId);

                end.append(fc - d);

            }

        }while (returnReduce(i < Cfs.size(), orOp<bool>()));

    ...
```

Here we, for each face that was stored earlier, store its center (or rather, the point just before the sun hits the center) as a starting point. Then we project a vector from that point towards the sun with a magnitude of approximately 5 times the diagonal of the mesh (**direction_*maxBounding**), and store the point at the other end of that vector as the corresponding end point. The **startIndex** list helps the program remember which face the points correspond to.

```
    ...

        List<pointIndexHit> hitInfo(startIndex.size());
            surfacesMesh.findLine(start, end, hitInfo);
            // Collect the rays which has 'only one not wall' obstacle between
            // start and end.
            // If the ray hit itself get stored in dRayIs
            forAll(hitInfo, rayI)
            {
                if (!hitInfo[rayI].hit())
                {
                    rayStartFace.append(startIndex[rayI]);
                }
            }

    ...

}
```

The surfacesMesh.findLine(start, end, hitInfo) function determines, for every start and end point pair, whether or not the line going from the start point to the end point ever intersects the mesh, and stores that information in hitInfo. In the forAll loop that follows, every face that has an unblocked path towards the sun is stored in rayStartFace. These faces are transferred to the public list rayStartFaces_, which can be accessed by solarLoad.C.

updateDirectHitRadiation()

This sub-function calculates the heat flux contribution from direct solar exposure.

```
bool Foam::radiation::solarLoad::updateDirectHitRadiation()
{

    ...

        const vector qPrim =
            solarCalc_.directSolarRad()*solarCalc_.direction();

        const vectorField& n = pp.faceNormals();

        {
            qprimaryBf[patchID][localFaceI] +=
                (qPrim & n[localFaceI])
                * spectralDistribution_[bandI]
                * absorptivity_[patchID][bandI]()[localFaceI];
        }

        if (includeMappedPatchBasePatches[patchID])
        {
            qrBf[patchID][localFaceI] += qprimaryBf[patchID][localFaceI];
        }
        else
        {
            const vectorField& sf = mesh_.Sf().boundaryField()[patchID];
            const label cellI = pp.faceCells()[localFaceI];
            Ru_[cellI] +=
                (qPrim & sf[localFaceI])
              * spectralDistribution_[bandI]
              * absorptivity_[patchID][bandI]()[localFaceI]
              / V[cellI];
        }
```

The solarCaculator class is first used to calculate the current solar intensity and sunlight direction vector. The vector qPrim is defined to have the same direction as the sun and the same magnitude as the solar intensity (directSolarRad()). Then, for each frequency band, each face of each patch gets added to it a heat flux value (qprimaryBf), equal to the dot product between the sun and the face normal, multiplied by the normalized spectral distribution of that band, multiplied by the absorptivity of that face for that band. The dot product factor is used in order to account for the angular visibility of the face.

The calculated flux is then added either to the qr_ variable (which qrBf references) or to the energy equation of the cells next to the walls, depending on settings set in the case files.

updateSkyDiffusiveRadiation()

This sub-function calculates the sky diffusive radiation contribution, which represents the radiation absorbed by the atmosphere and ground and then re-emitted in a diffuse manner throughout the mesh.

It is calculated differently depending on the solar model used.

```
void Foam::radiation::solarLoad::updateSkyDiffusiveRadiation
(
    const labelHashSet& includePatches,
    const labelHashSet& includeMappedPatchBasePatches
)
{
    const polyBoundaryMesh& patches = mesh_.boundaryMesh();
    const scalarField& V = mesh_.V();
    volScalarField::Boundary& qrBf = qr_.boundaryFieldRef();

    switch(solarCalc_.sunLoadModel())
    {
        case solarCalculator::mSunLoadFairWeatherConditions:
        case solarCalculator::mSunLoadTheoreticalMaximum:
        {
            for (const label patchID : includePatches)
            {
                const polyPatch& pp = patches[patchID];
                const scalarField& sf = mesh_.magSf().boundaryField()[patchID];

                const vectorField n = pp.faceNormals();
                const labelList& cellIds = pp.faceCells();

                forAll(n, faceI)
                {
                    const scalar cosEpsilon(verticalDir_ & -n[faceI]);

                    scalar Ed(0.0);
                    scalar Er(0.0);
                    const scalar cosTheta(solarCalc_.direction() & -n[faceI]);

                    {
                        // Above the horizon
                        if (cosEpsilon == 0.0)
                        {
                            // Vertical walls
                            scalar Y(0);

                            if (cosTheta > -0.2)
                            {
                                Y = 0.55+0.437*cosTheta + 0.313*sqr(cosTheta);
                            }
                            else
                            {
                                Y = 0.45;
                            }
                            Ed = solarCalc_.C()*Y*solarCalc_.directSolarRad();
                        }
                        else
                        {
                            //Other than vertical walls
                            Ed =
                                solarCalc_.C()
                              * solarCalc_.directSolarRad()
                              * (1.0 + cosEpsilon)/2.0;
                        }

                        // Ground reflected
                        Er =
                            solarCalc_.directSolarRad()
                          * (solarCalc_.C() + Foam::sin(solarCalc_.beta()))
                          * solarCalc_.groundReflectivity()
                          * (1.0 - cosEpsilon)/2.0;
                    }

                    const label cellI = cellIds[faceI];
                    if (includeMappedPatchBasePatches[patchID])
                    {
                        for (label bandI = 0; bandI < nBands_; bandI++)
                        {
                            qrBf[patchID][faceI] +=
                                (Ed + Er)
                              * spectralDistribution_[bandI]
                              * absorptivity_[patchID][bandI]()[faceI];
                        }
                    }
                    else
                    {
                        for (label bandI = 0; bandI < nBands_; bandI++)
                        {
                            Ru_[cellI] +=
                                (Ed + Er)
                              * spectralDistribution_[bandI]
                              * absorptivity_[patchID][bandI]()[faceI]
                              * sf[faceI]/V[cellI];
                        }
                    }
                }
            }
        }
        break;
    }
```

For the mSunLoadFairWeatherConditions and mSunLoadTheoreticalMaximum models, the model creates a cosEpsilon value for each face, being the dot product between the reversed face normal and the vector pointing towards the earth's core (in almost every case, the normalized g vector). Since both the reversed normal vector and the downwards pointing vector are normalized, the dot product will be equal to the cosine of the angle between them. This means that a face on the floor, whose reversed normal vector points downwards, will get the cosEpsilon value of cos(0) = 1. A vertical wall face, whose normal (and reverse normal) will always be at a 90 degree angle from the g vector, giving the face a cosEpsilon value of cos(90) = 0. Non-orthogonal faces will have a value inbetween 0 and 1. An exception to this are faces with a normal vector with a downwards facing component, which have a cosEpsilon value of [-1, 0).

Each face then is then assigned an Ed and an Er value, which each represents the diffusive radiance contribution from the sky and ground respectively. Their equations both feature a view factor. For sky diffusivity it equals (1.0 - cosEpsilon)/2.0 and for ground diffusivity it equals (1.0 - cosEpsilon)/2.0. They both range from 0 to 1, and add up to 1 for any one face. They can be seen as angle-dependent weights for a face. A face on the floor will have a ground diffusivity wieght of 0 and a sky diffusivity of 1, since it can only see the sky and not the ground.

Vertical faces get special treatment. Faces where cosTheta > -0.2, i.e. where the normal vector points somewhat away from the sun, i.e. significantly shaded faces, are assigned a sky diffusive weight of 0.55+0.437*cosTheta + 0.313*sqr(cosTheta). Vertical faces somewhat pointing towards the sun are assigned a sky diffusive wieght of 0.45, slightly lower than the 0.5 they would have been assigned by the (1.0 + cosEpsilon)/2.0 function.

The diffusive radiation intensities for sky and ground diffusion are determined by the diffusive constant C, the solar intensity directSolarRad and the groundReflectivity, all defined at run-time in the solarLoadCoeffs dictionary.

The mSunLoadConstant model has a much simpler calculation:

```
        case solarCalculator::mSunLoadConstant:
        {
            ...

                {
                    for (label bandI = 0; bandI < nBands_; bandI++)
                    {
                        qrBf[patchID][faceI] +=
                            solarCalc_.diffuseSolarRad()
                          * spectralDistribution_[bandI]
                          * absorptivity_[patchID][bandI]()[faceI];
                    }
                }
                else
                {
                    for (label bandI = 0; bandI < nBands_; bandI++)
                    {
                        Ru_[cellI] +=
                            (
                                spectralDistribution_[bandI]
                              * absorptivity_[patchID][bandI]()[faceI]
                              * solarCalc_.diffuseSolarRad()
                            )*sf[faceI]/V[cellI];
                    }
                }

            ...

        }
    }
}
```

Here, the user-defined diffusive solar irradiation intensity (diffuseSolarRad) is simply added on to walls directly in a uniform manner. The calculation is the same as in the updateDirectHitRadiation function, except there is no reduction of absorption based on the angle of exposure.

updateReflectedRays()

This sub-function calculates specular reflection with a depth of 1 bounce. It does so in a manner similar to that of FvDOM, where only a discrete number of angles are available. Each time a bounce direction is calculated it selects the discrete angle that is as close to it as possible, and this becomes its actual direction.

```
void Foam::radiation::solarLoad::updateReflectedRays
(
    const labelHashSet& includePatches
)
{

    ...

        reflectedFaces_->correct();
    ...
```

reflectedFaces is an instance of the faceReflecting class, defined in faceReflecting.C.

```
void Foam::faceShading::correct()
{
    calculate();
}

void Foam::faceReflecting::correct()
{
    calculate();
}

void Foam::faceReflecting::calculate()
{


    ...

        vector refDir =
            sunDir + 2.0*(-sunDir & n[faceI]) * n[faceI];

          // Look for the discrete direction
                    scalar dev(-GREAT);
                    label rayIndex = -1;
                    forAll(refDiscAngles_, iDisc)

            forAll(refDiscAngles_, iDisc)
                {
                    scalar dotProd = refDir & refDiscAngles_[iDisc];
                    if (dev < dotProd)
                    {
                        dev = dotProd;
                        rayIndex = iDisc;
                    }
                }

                if (rayIndex >= 0)
                {
                    if (refDisDirsIndex[rayIndex] == -1)
                    {
                        refDisDirsIndex[rayIndex] = 1;
                    }
                }

                refFacesDirIndex.insert
                (
                    globalNumbering.toGlobal(globalID),
                    rayIndex
                );

    ...
```

Here refDir is calculated as the exact reflection angle off the face. In the following forAll loop, each available discrete angle is read and compared to refDir. The angle whose dot product with refDir is the largest is stored as rayIndex.

Since cos(x) gets larger as x approaches 0, the discrete angle with the largest dot product will also have the smallest angle in reference to refDir. rayIndex, i.e. the chosen angle, is then stored in refFacesDirIndex together its global ID.

```
...

scalar maxBounding = 5.0*mag(mesh_.bounds().max() - mesh_.bounds().min());

...

    forAll(refDisDirsIndex, dirIndex)
                {
                    if (refDisDirsIndex[dirIndex] > -1)
                    {
                        if ((nf & refDiscAngles_[dirIndex]) > 0)
                        {
                            const vector direction = -refDiscAngles_[dirIndex];

                            start.append(fc + 0.001*direction);

                            startIndex.append(myFaceId);
                            dirStartIndex.append(dirIndex);

                            end.append(fc + maxBounding*direction);
                        }
                    }
                }
            }

        }while (returnReduce(i < Cfs_->size(), orOp<bool>()));

        List<pointIndexHit> hitInfo(startIndex.size());

        surfacesMesh_->findLine(start, end, hitInfo);

...
```

Here the faces hit by the reflected rays are determined in a similar way they were in updateDirectHits(), except here hitInfo is not used to find the lines without intersections, but is instead used to find the faces it intersects with.

6.1.1 solarCalculator

The solarCalculator class keeps track of the sun direction and solar intensity. There are two models for direction and three for intensity. Direction is calculated in the calculateSundirection function, which uses the beta and theta angle variables, which are the sun angle above the horizon and the sun's cardinal angle in relation to the southern direction, respectively. These are calculated in the calculateBetaTheta function. The models for sun direction are:

sunDirConstant

Sun direction is set in the dictionary with the sunDirection entry.

sunDirTracking

Beta and theta are calculated from the following parameters:

- localStandardMeridian : GMT (Local Zone Meridian) in hours
- startDay : day from 1 to 365)
- startTime: in hours
- longitude: in degrees
- latitude: in degrees
- gridUp: grid orientation upwards
- gridEast grid orientation eastwards

The parameters are specified in the solarLoadCoeffs dictionary in the **constant/radiationProperties** file.

As for intensity, the available models are:

sunLoadConstant

Solar intensity is set in the dictionary entries directSolarRad and diffuseSolarRad.

sunLoadFairWeatherConditions

The solar intensity follows the Fair Weather Conditions Method from the ASHRAE Handbook(Heating and Engineers, 2017). The entries are:

- skyCloudCoverFraction: Fraction of sky covered by clouds (0-1)
- A: Apparent solar irradiation at air mass m = 0
- B: Atmospheric extinction coefficient
- beta: Solar altitude (in degrees) above the horizontal plane. Is either entered or calculated
- groundReflectivity : Ground reflectivity

The direct solar flux is calculated as:

$$directSolarRad = (1 - 0.75 * skyCloudCoverFraction^3) * \frac{A}{exp(B/sin(beta)})$$

sunLoadTheoreticalMaximum

The solar intensity is the same as for sunlight falling at a 0 degree angle with no cloud cover.

The entries are (as taken from the source code comments):

- Setrn
- SunPrime
- roundReflectivity

In this model the flux is calculated as: directSolarRad = Setrn*SunPrime

Sky diffusivity is calculated in the same way as in sunLoadFairWeatherConditions.

Chapter 7

Methodology

The built-in OF solver that comes the closest to including all of the functionality required by our project is the rhoReactingBuoyantFoam solver. It includes compressibility, chemical reactions and buoyancy. Radiation is not included in this solver, so this needed to be added by modifying rhoReactingBuoyantFoam and saving it as a new solver. This was done by adding the lines

```
+ radiation->Sh(thermo, he)
```

and

```
radiation->correct();
```

to the **EEqn.H** file,

```
#include "createRadiationModel.H"
```

to the **createFields.H** file and

```
#include "radiationModel.H"
```

to the **rhoReactingBuoyantFoam.C** file in the rhoReactingBuoyantFoam folder.

7.1 Implementing the Photolytic Solver

J field calculation couldn't be implemented only by changing solver scripts. It had to be implemented in the OF source code library. Most of the implementation was done in the **src/thermophysicalModels/radiation/radiationModels/solarLoad/faceShading/faceShading.C** file.

In tandem with this, a solution had to be implemented to account for the fact that the standard solar radiation model did not account for cyclic boundary conditions. Cyclic boundaries are used to "transport" field values from one side of the mesh to another, commonly used for setups with long repeating patterns, like pipe flows. It is used in our simulation in order to, at least partially, account for the surroundings of the street canyon in question, in practice simulating what is equivalent to an infinite number of parallel infinitely long street canyons. However, the solar rays of the solar radiation model we were working with was not detected by the cyclic boundary conditions, which means that rays that were supposed to be blocked by the surrounding simulated environment were not blocked.

Below is a 2D example of how the unmodified solver calculates which boundary faces are hit by the sun:

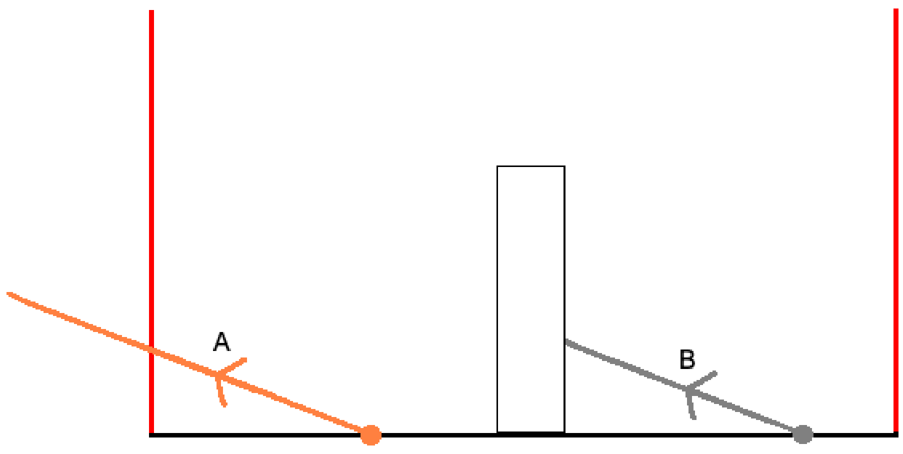


FIGURE 7.1: Ray A travels right through a cyclic boundary, as it is has an opacity of 0, and is considered unblocked. Ray B is blocked as normal.

In this case, ray B is blocked, as it should be. However, ray A goes through the cyclic boundary and therefore successfully "escapes" the mesh without being blocked. A more accurate model would look like this:

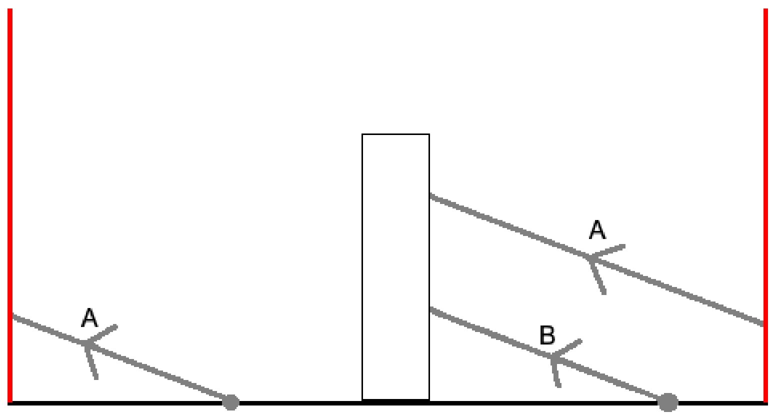


FIGURE 7.2: Ray A hits a cyclic boundary. A new identical ray is then projected from the same face on the opposite side of the mesh, which is then blocked. The face A was projected from is now considered to be shaded. Ray B is blocked as normal

Here, ray A is transferred through the cyclic boundary to the opposite side, where it continues its path until it is blocked. The boundary from which ray A was projected is now considered to be in the shade.

As explained in 5, the original face shading model calls on the faceShading::calculate() function to define a start array and an end array and fills them with start- and end points for tracing a ray from each face to a point outside the mesh in the sun's direction. Each face with a ray that intersects the boundary mesh gets categorized as being in the shadow. The searchableSurface::findline() function is the function used to find which rays intersect the boundary mesh and which ones do not. Here it is shown once again for reference:

```
forAll(n, faceI)
    {
        const vector nf(n[faceI]);
        if (((direction_ & nf) > 0) && (t[faceI] == 0.0))
        {
            dynFacesI.append(faceI + pp.start());
            dynCf.append(cf[faceI]);
            nFaces++;
        }
    }

...

for (; i < Cfs.size(); i++)
    {
        const point& fc = Cfs[i];

        const label myFaceId = hitFacesIds[i];

        const vector d(direction_*maxBounding);

        start.append(fc - 0.001*d);

        startIndex.append(myFaceId);

        end.append(fc - d);

    }

...

forAll(hitInfo, rayI)
    {
        if (!hitInfo[rayI].hit())
        {
            rayStartFace.append(startIndex[rayI]);
        }
    }
```

It was changed so that when a ray projected from a face facing the sun intersects the mesh, the face intersecting the ray is evaluated. If it belongs to an opaque boundary, the ray continues travelling but is considered to be shaded, and its last mesh intersection is stored. If it belongs to a cyclic boundary the ray stops and a new corresponding ray is projected starting from the corresponding face of the opposite cyclic patch. This is done iteratively for each ray, stopping when a ray segment escapes the mesh without intersecting with any boundary face, signifying the ray has "escaped" into the atmosphere. The ray path, with all its cyclic segments, is stored, where its end point is set as its last opaque face intersection. All the ray paths that never intersected an opaque face are considered unblocked and discarded.

It is demonstrated in the following figure:

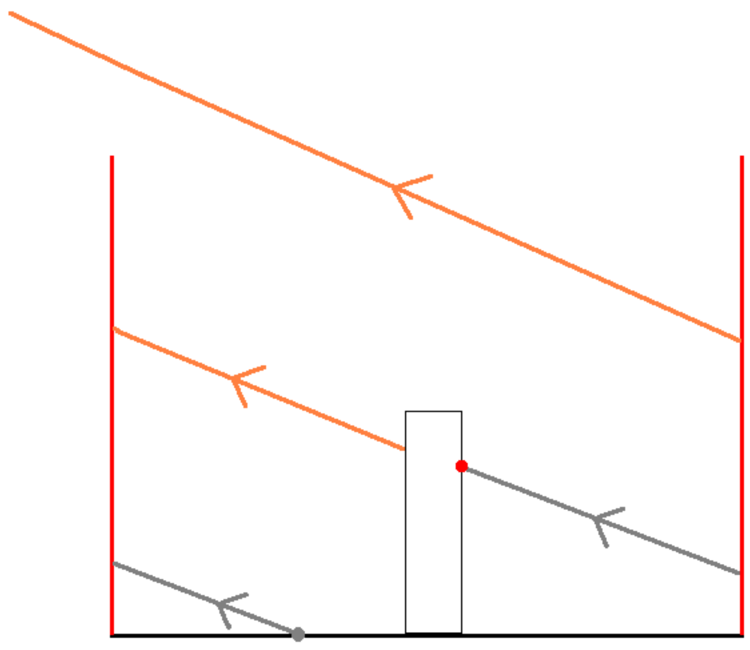


FIGURE 7.3: A ray is projected from a face in the bottom boundary of the mesh, hits a cyclic boundary, gets transported to the other side of the mesh, hits an opaque face, hits a cyclic boundary again, gets transported to the other side of the mesh, and finally escapes the mesh.

Here, the ray hits a cyclic boundary twice during its lifetime, creating 2 new segments of the same ray for a total of 3 segments. In the second segment, an opaque face is hit (belonging to the block in the center). The point of intersection with the opaque face is stored as a "shadow end point", marked by a red dot.

In the case of multiple opaque intersections in multiple segments, the last intersection of the last of those segments is used. In 7.4, the second intersection of the second segment is chosen as the shadow end point, marked by a red dot, and the other intersections are discarded.

For each segment up until the one with the shadow end point, we store their start and end points. For the segment with the shadow end point, we store the shadow end point instead of its previous end point. These start and end points are then used to project "shadow rays", one for each segment of each ray (except those that never hit an opaque boundary face). Unlike the previous algorithm where we only stored the rays' intersection with the boundary mesh, here we store each intersection with the internal field, that is, the fluid area. Each internal field face that is intersected by these shadow rays is set as a shaded face. For each shaded face, both of its corresponding cells are set as shaded cells. The complement of these cells are stored in the list "sunCells". In this way, we know which parts of the fluid are exposed to sunlight and which ones are not.

When the J field is updated during runtime, only the J field indices in sunCells are updated, meaning only the cells exposed to the sun have their J value updated with a non-zero value. This is done in the **updateJField()** function of **solarLoad.C**:

```
void Foam::radiation::solarLoad::updateJField(const labelHashSet& litCellsId)
```

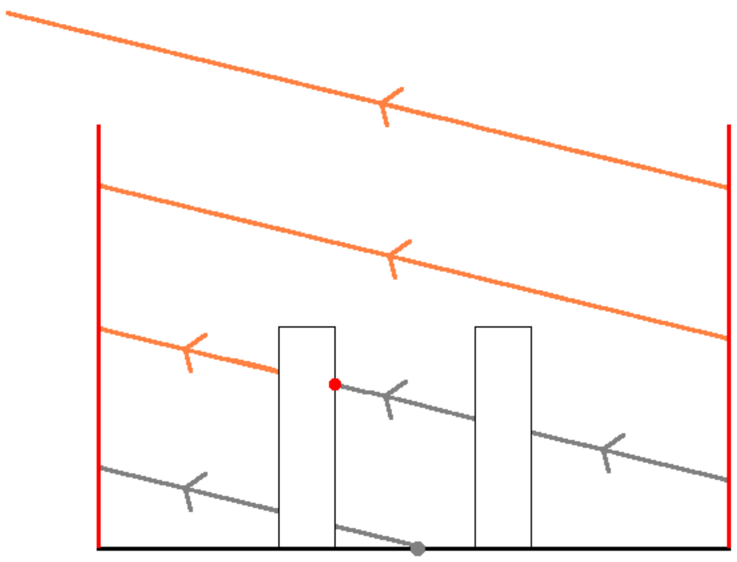


FIGURE 7.4: Identical to 7.3, except that the ray travels through multiple opaque faces. The last opaque face hit is set as a shadow end point, and a shadow ray is projected from the starting point to the shadow end point.

```
{

    for (label bandI = 0; bandI < nBands_; bandI++)
    {
        for (const label cellI : litCellsId)
        {
J_[cellI] +=
(solarCalc_.directSolarRad()/solarCalc_.A())
  * spectralDistribution_[bandI]
  * photoFactors_[bandI];
        }
    }
}
```

The J value acts as a weight between 0 and 1, where 0 is no solar exposure and 1 is solar exposure at air mass = 0, that is, the solar exposure at the top of the atmosphere. photoFactors is a weight vector describing the photolytic effect of each frequency solar band. In general high frequency bands (approaching or in the UV spectral range) should be assigned higher values.

Since J is a new field not previously existing in OpenFOAM, there are no available reactions that use J when calculating their reaction rates. Because of this, a new photolysisRate class had to be implemented, and the existing chemistry models needed to be modified in order to be able to utilize this reaction rate. The reaction rate is set as the J weight value multiplied by equation 3.6.

The module currently does not account for reflective or diffusive radiation when calculating the J field.

7.2 Simulation Setup

The simulations employed a mesh with a width of 14.2m and a height of 75m, with the canyon width being half the mesh width (7.2m) and the canyon height being a fifth of the mesh height (15m). Cyclic boundaries were set at the x-wise and y-wise boundaries, effectively making the mesh equivalent to that of an infinite number of infinitely long parallel canyons.

A non-uniform grading was used, with the highest mesh resolution at the ground and along the vertical walls, and with resolution rapidly decreasing along the z direction. The mesh employed a total of 1 048 576 cells.

The air above the canyon was set to have a constant wind with a direction perpendicular to that of the street's orientation with a velocity of 2.71 m/s, a temperature of 293K and an O_3 background concentration of $4.5 * 10^{-8}$. NO_2, NO and were also emitted from the bottom boundary of the mesh, both with a rate of $6.2 * 10^{-6} s^{-1}$ respectively.

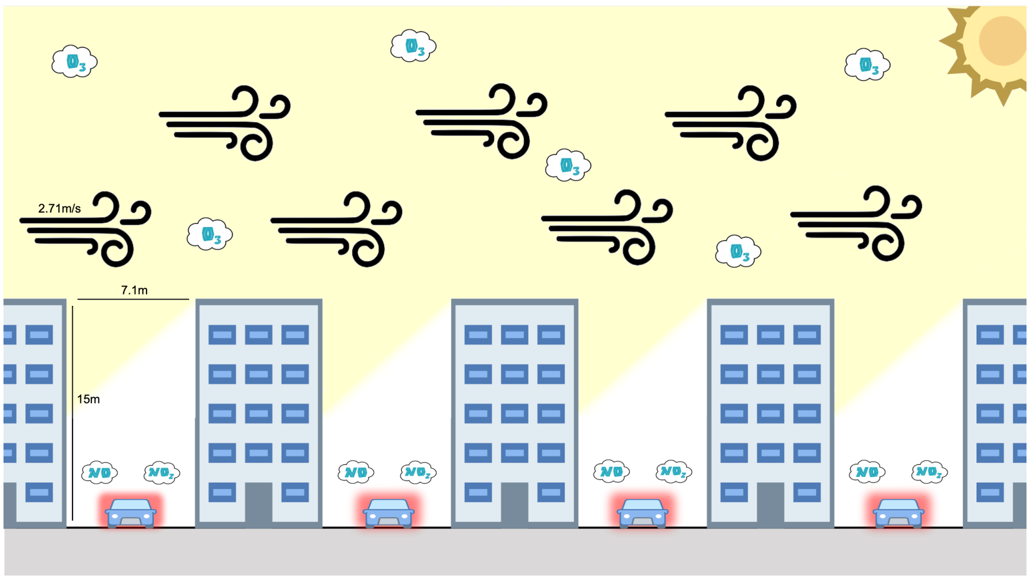


FIGURE 7.5: Simulation setup

The timestep was set to a base of $3.5e^{-3}$ seconds with a courant number max limit of 3.

Tominaga et al. (Tominaga and Stathopoulos, 2011) showed that for street canyon simulations that include buoyant effects, LES turbulence models show a greater correspondence to experimental data compared to RANS models. For this simulation we used the kEqn Les model.

The solarLoad model was utilized to calculate solar radiation. The standalone model was used, i.e. it was not combined with the viewFactor or FvDOM radiation models. The geographic location and polar orientation of the canyon was set according to the experimental studies performed by Offerle (Offerle et al., 2007), with the coordinates set to those of the Swedish city Gothenburg, and with the positive y direction pointing North, with a 10° tilt toward the east. The time of day in the simulation is set to 10 A.M., which gives a solar vector of approximately [-0.65 0.38 -0.65] and runs for about 20 minutes. The A parameter, which represents the apparent solar irradiation at air mass = 0 was set to 45. How this value was arrived at is further discussed in 8.

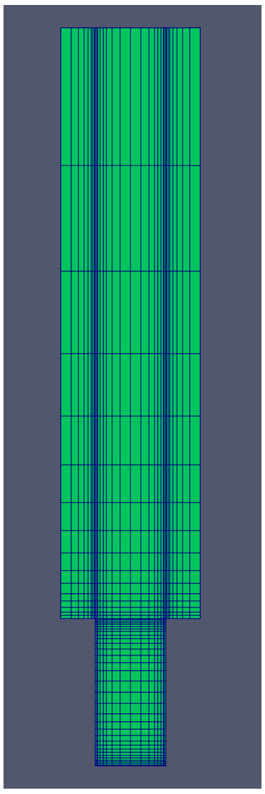


FIGURE 7.6: Mesh grid. Resolution has been lowered globally for better viewing.

The absorptivity and emissivity of all boundaries were set to 0.7. Specular reflection was turned off, as the surfaces of urban houses are mostly diffusive.

The building roof and wall boundary condition, as well as the ground's boundary condition, were set as the **externalWallHeatFluxTemperature** type. This boundary condition specifies an ambient temperature and thermal conductivity for a patch. It has the following parameters:

Property	Description	Required	Default
q	heat flux [W/m2]	yes*	
Ta	ambient temperature [K]	yes*	
h	heat transfer coefficient [W/m2/K]	yes*	
thicknessLayers	list of thicknesses per layer [m]	yes	
kappaLayers	list of thermal conductivities per layer [W/m/K]	yes	
Qr	name of the radiative field	no	no
relaxation	relaxation factor for radiative field	no	1
kappaMethod	inherited from temperatureCoupledBase	inherited	
kappa	inherited from temperatureCoupledBase	inherited	

FIGURE 7.7: Coefficients for the externalWallHeatFluxTemperature boundary condition. Source: Boundary Conditions - OpenFOAM 4.1, NEXTFOAM

The heat flux q is calculated depending on the mode of the boundary conditions. If mode is set to flux, it is supplied by the user using the q parameter. If mode is set to coefficient, it is calculated with the following equation:

$$q = (Ta\_ - Tp) * (1.0/h\_ + totalSolidRes) \tag{7.1}$$

Where Tp is the temperature of the outside cell, and totalSolidRes can be interpreted as the wall's thermal resistance, determined by the thicknessLayers and kappaLayers coefficients. This boundary condition therefore regulates the temperature of the wall to converge towards the specified ambient temperature Ta, with a convergence speed depending on the thermal conductivity h and the thickness/conductivity layers of the wall (NEXTFOAM, 2017). We set Ta_ to 283K and modelled thickness layers and thermal conductivity according to typical building walls in Gothenburg, Sweden. How Ta_ was decided will be further discussed in Chapter 8.

The simulations were run without reactions for an initial 1 000 seconds in order to minimize the effect of transient flow.

Four simulations were performed in total:

- No-sun: A simulation without photolytic reaction. Solar thermal effects are still included.
- Global-sun: A simulation where the whole domain is considered exposed to sunlight. Similar to the setup of Liu et al. (Liu et al., 2021).
- Partial-sun: A simulation utilizing the implemented J field solver. Photolytic breakdown occurs only in areas exposed to the sun, i.e. with J > 0.
- Strong-sun: Same as Partial sun but with the photolytic increased 1000-fold. Thermal effects are unchanged.

The simulations were run for a total of 500 seconds each.

7.3 Resources

Computers at the Department of Energy Sciences were supplied for the project. For cases which require high computational power, resources provided by the Swedish National Infrastructure for Computing (SNIC) at PDC (Beskow) were used. OpenFOAM and external libraries used are open source and freely available on the web. For the duration of the project an office desk and computer were supplied for the student at the K faculty building at LTH.

Chapter 8

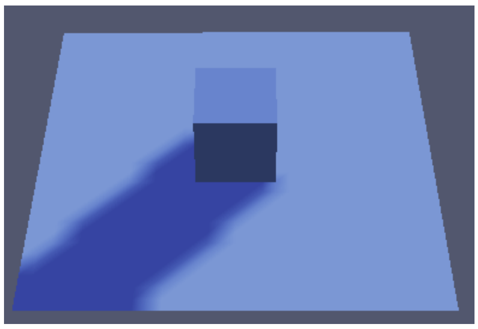
Results

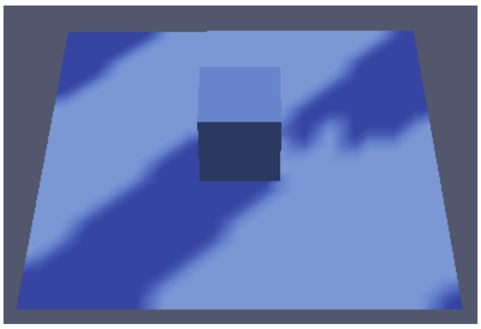
8.1 Validation

8.1.1 Cyclic boundary algorithm

The cyclic boundary algorithm was validated using the mesh from the ExternalSolarLoad OpenFOAM tutorial case mesh.

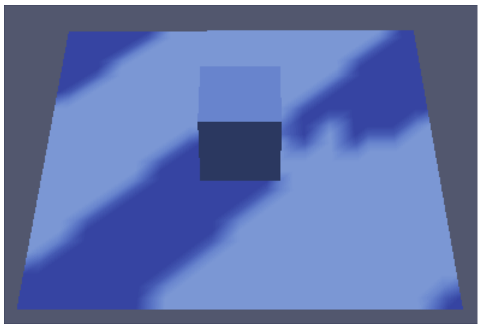
Geographical coordinates were set to that of Tokyo, Japan. Time of year was set to the 1st of January and the time of day was set to 8 A.M.. Below is a comparison of the solar heat flux field qr with the standard solarLoad library and the implemented cyclic boundary solarLoad library.

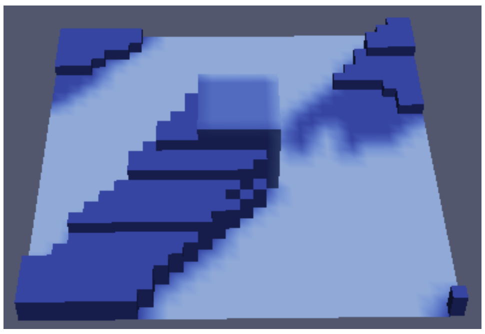

qr field in case without cyclic boundaries, calculated using the standard solarLoad library


qr field in case with cyclic boundaries, calculated using the modified solarLoad library

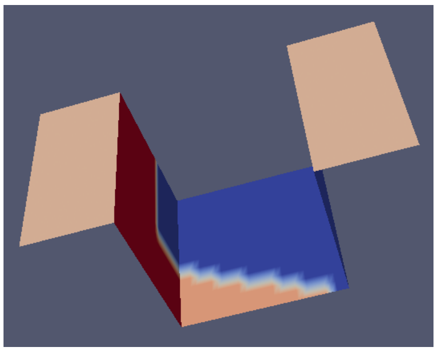
8.1.2 J field solver

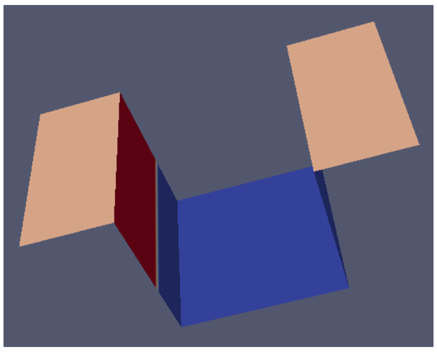
The J field solver was validated using the same mesh as the cyclic boundary algorithm. Configurations with different geographical locations, times of year and times of day were tested and validated. Below are the results for the settings previously used. The results for all configurations tested can be found in Appendix A.

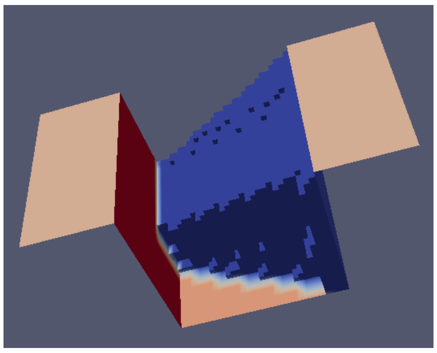

qr field for Tokyo, Japan, January 1st, 8 A.M.

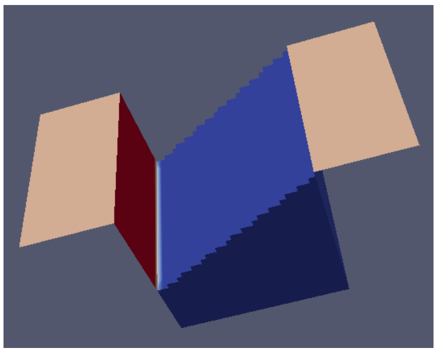

J field for Tokyo, Japan, January 1st, 8 A.M.

Applied on a 1:1 street canyon:


Solar heat flux field, unmodified solver


Solar heat flux field, cyclic boundary algorithm active


Inverse J field (J = 0), cyclic boundary algorithm inactive


Inverse J field (J = 0), cyclic boundary algorithm active

The module does however perform worse when used on finer meshes, and meshes with a non-round width-to-height ratio (like a 15x16 mesh). This mesh is very fine, with a width/height ratio of $14.2/15 = 0.946667$ and over a million cells, and so errors can be found at the sides of mesh (figure 8.5). The total volume of these areas is however quite small, and so their effect on the simulation is possibly negligible. Further research, or an improved module, is required.

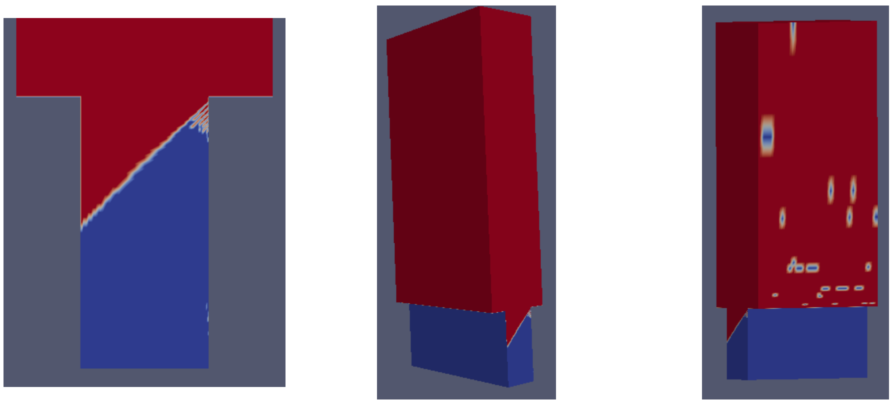


FIGURE 8.5: J field error areas in main mesh

8.1.3 Photolytic reaction rate

The J dependent reaction rate and the corresponding chemistry model were validated using a custom testing case. In the center of the urban street canyon mesh used in the main simulation of this thesis, in the initial timestep of 0, a column of air was set to have an initial NO_2 concentration of $4 * 10-7$. Wind velocity was set globally to 0, and all reactions except reaction 3.3 were disabled. The same solar radiation parameters were set as in the main simulation, i.e. the location was set to Gothenburg, Sweden, and the time of day was set to 10 AM. The results can be seen in figure 8.6.

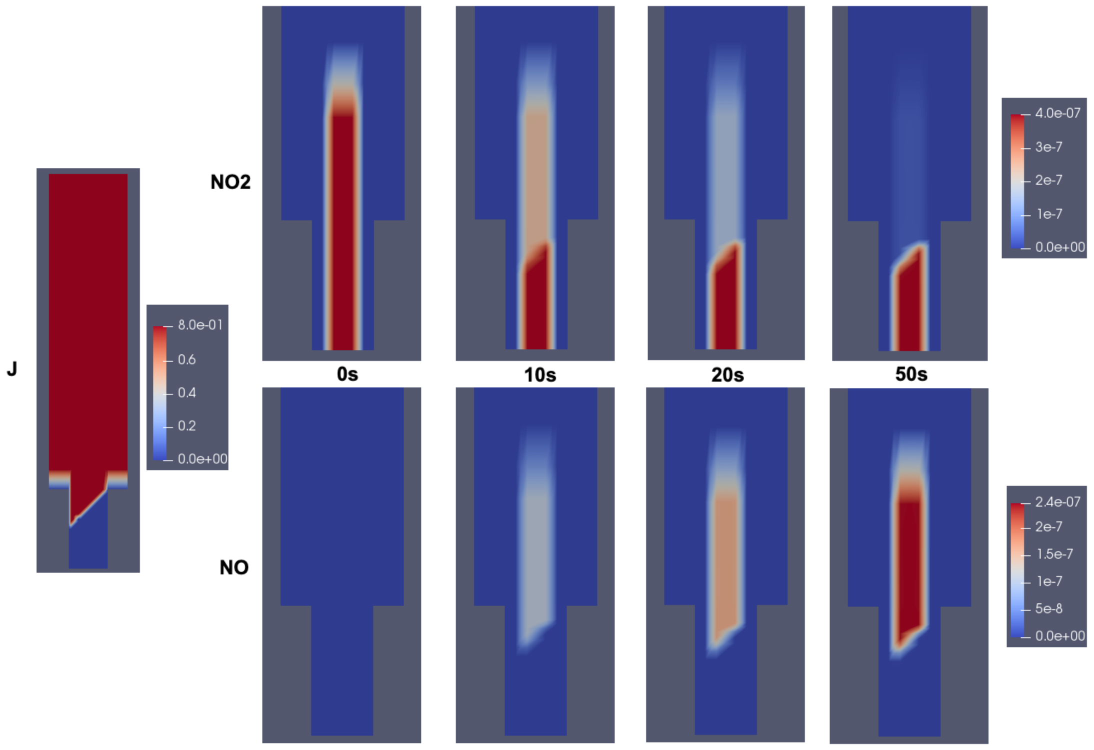


FIGURE 8.6: Influence of J field on NO_2 -> NO + O reaction rate

The impact of photolytic breakdown in relation to the other reactions included in our chemistry model was briefly tested using the chemFoam solver in OpenFOAM.

This is a specialized solver that solves the chemical interactions of a single cell.

For initial conditions we set the temperature to 293K, pressure to 1 bar, J value to 0.8 (representing a time of day of around 10 AM), NO concentration to $4 * 10^{-7}$, NO_2 concentration to $1 * 10^{-7}$ and O_3 concentration to $4 * 10^{-8}$. We then let chemFoam run for 1 000 seconds once without photolytic effect, once with photolytic effect and once with x1000 photolytic effect. The resulting mass fractions of the different compounds can be seen in the following table:

Chemical mass fractions after 1000s			
Solar multiplier	NO	NO_2	O_3
0	$3.73 * 10^{-7}$	$2.23 * 10^{-7}$	$7.31 * 10^{-70}$
1	$3.80 * 10^{-7}$	$2.13 * 10^{-7}$	9.56^{-9}
1000	$5.14 * 10^{-7}$	$6.52 * 10^{-9}$	$2.25 * 10^{-7}$

We see from these results that in the absence of sunlight, O_3 becomes the limiting factor for NO_2 production, and also that the effect of normal sunlight is not a dominant factor for NO_2 concentration levels, but a dominant factor for O_3 concentration levels.

8.1.4 Thermal parameter calibration

When performing our first test simulations using an A value of 1360 W/m^2 and setting our wall Ta_ coefficients to that of room temperature (294K), we were getting temperature values much higher than what was physically possible. We therefore performed parallel experimental simulations with different boundary conditions, A, and Ta_ values in order to calibrate our case as to better match the experimental data available. After multiple rounds of simulations, the final values were set as A = 45 and Ta_ = 283. This is a potential cause for errors and will be further discussed in chapter 9. The temperature plot for our final temperature simulation can be seen in Figure 8.7. The plots for the remaining simulations from the last round of testing can be found in Appendix B.

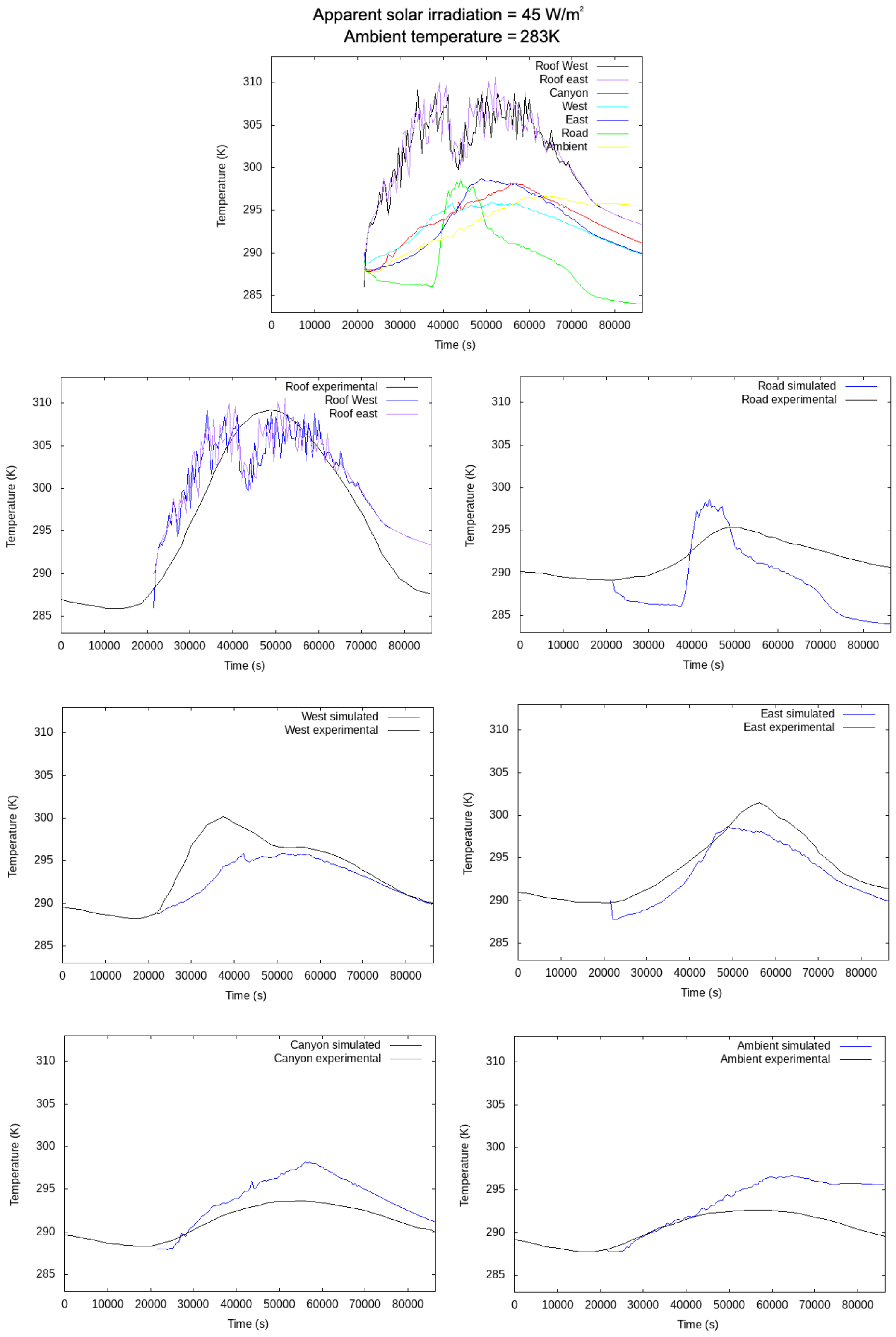


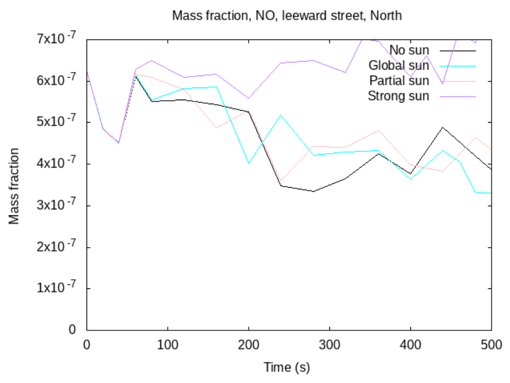
FIGURE 8.7: Temperature graphs over time for the final configuration used in our simulations. A = 45, Ta_ = 283. Experimental data taken from Offerle et al. (Offerle et al., 2007)

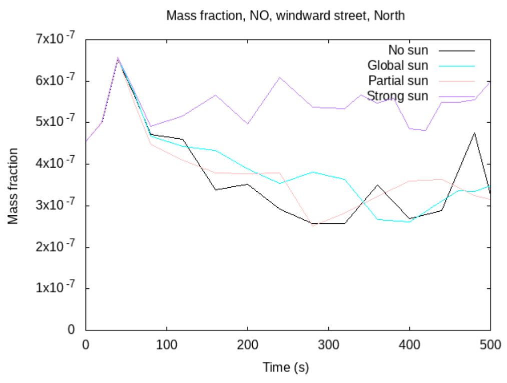
More preferable would be a model including radiative heat loss from the boundaries. This could be done by combining the solarLoad model with the viewFactor or

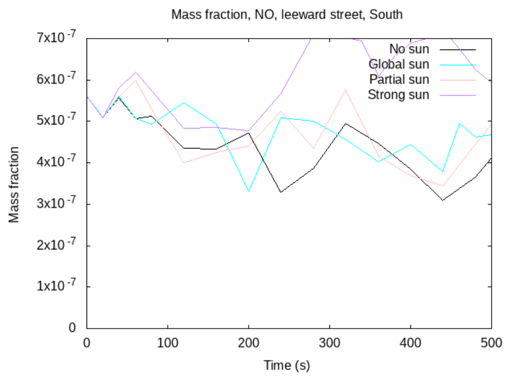
FvDOM models.

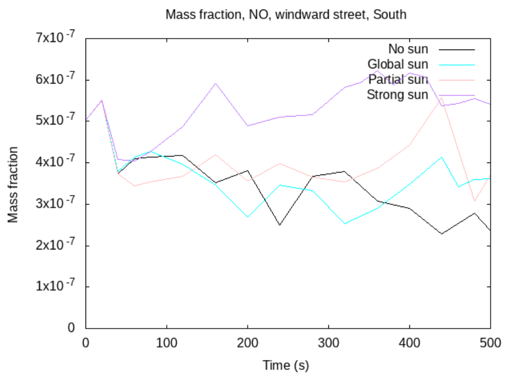
8.2 Simulation results

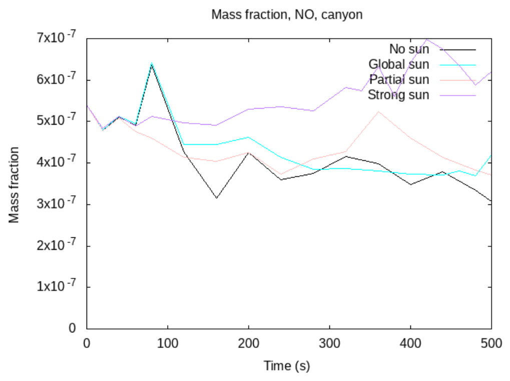
Levels of NO, NO_2 and O_3 were measured at six points: four on the sidewalk, half a meter from the wall, at a height of 1.6m — meant to emulate pedestrian exposure, one in the middle of the canyon at 10 meters height, and one in the sky at 20 meters height. A spatial average of an area around each point was measured, working as a low-pass filter. The results are shown in the figures below:

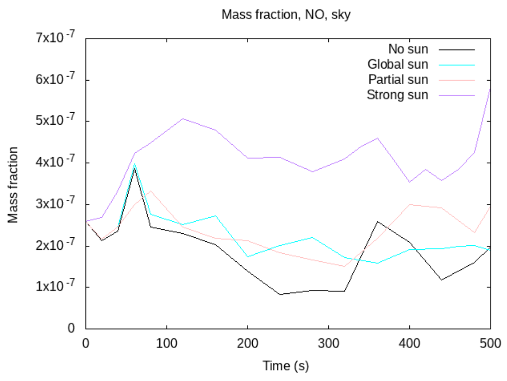


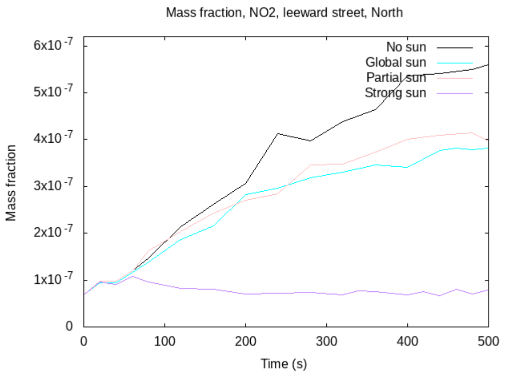


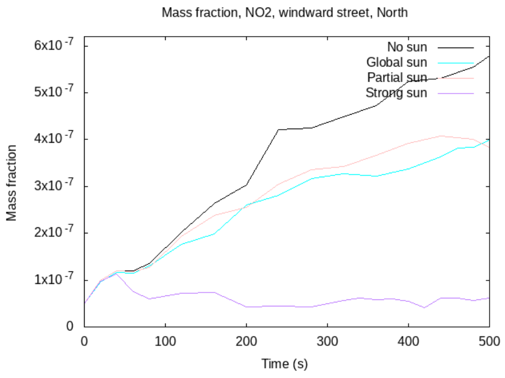


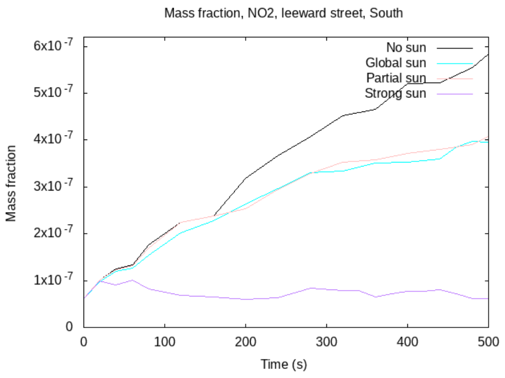


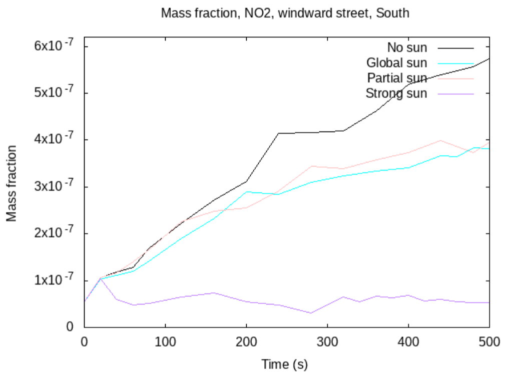


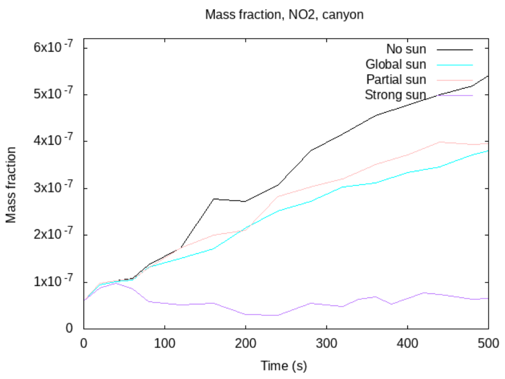


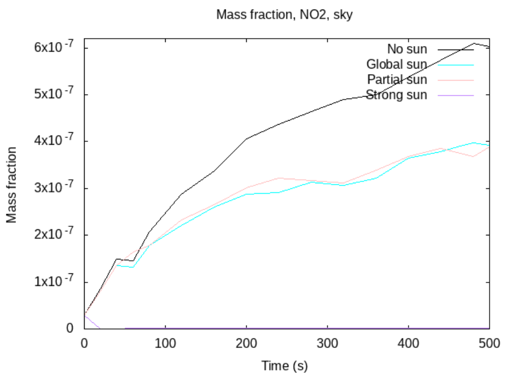

Mass fraction, NO2, leeward street, North
Mass fraction
Time (s)
No sun
Global sun
Partial sun
Strong sun

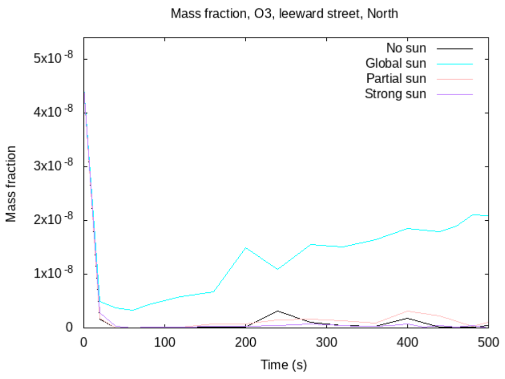

Mass fraction, NO2, windward street, North
Mass fraction
Time (s)
No sun
Global sun
Partial sun
Strong sun

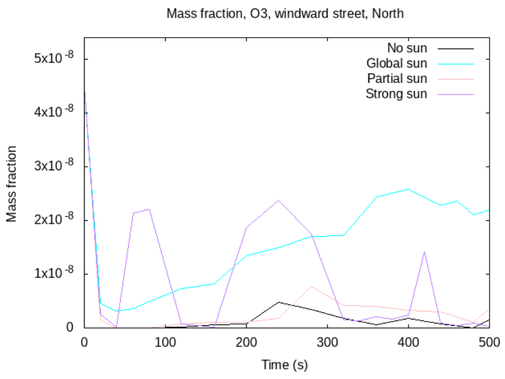

Mass fraction, NO2, leeward street, South
Mass fraction
Time (s)
No sun
Global sun
Partial sun
Strong sun

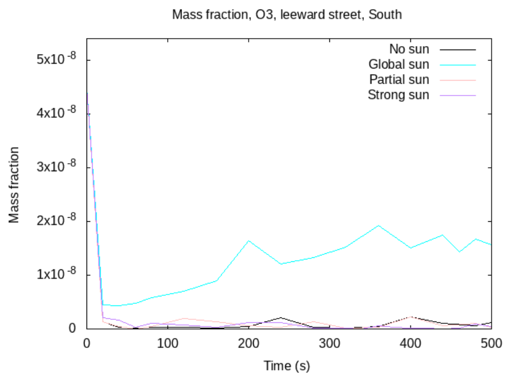

Mass fraction, NO2, windward street, South
Mass fraction
Time (s)
No sun
Global sun
Partial sun
Strong sun

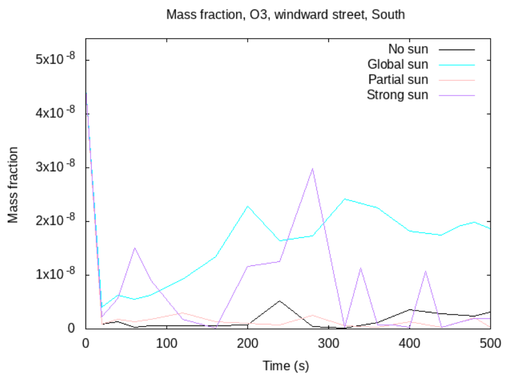

Mass fraction, NO2, canyon
Mass fraction
Time (s)
No sun
Global sun
Partial sun
Strong sun

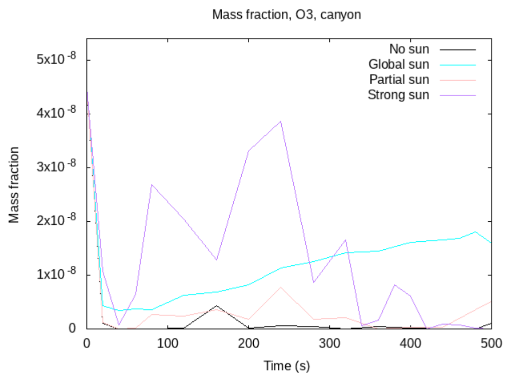

Mass fraction, NO2, sky
Mass fraction
Time (s)
No sun
Global sun
Partial sun
Strong sun

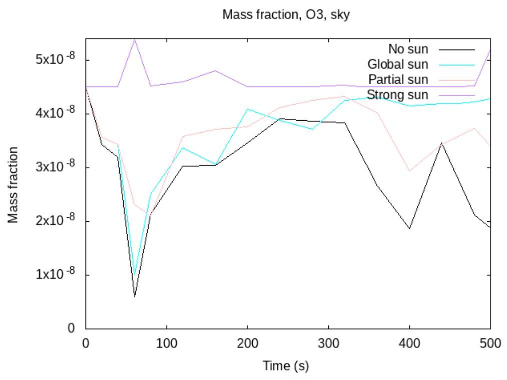

Mass fraction, O3, leeward street, North
Mass fraction
Time (s)
No sun
Global sun
Partial sun
Strong sun


Mass fraction, O3, windward street, North
Mass fraction
Time (s)
No sun
Global sun
Partial sun
Strong sun







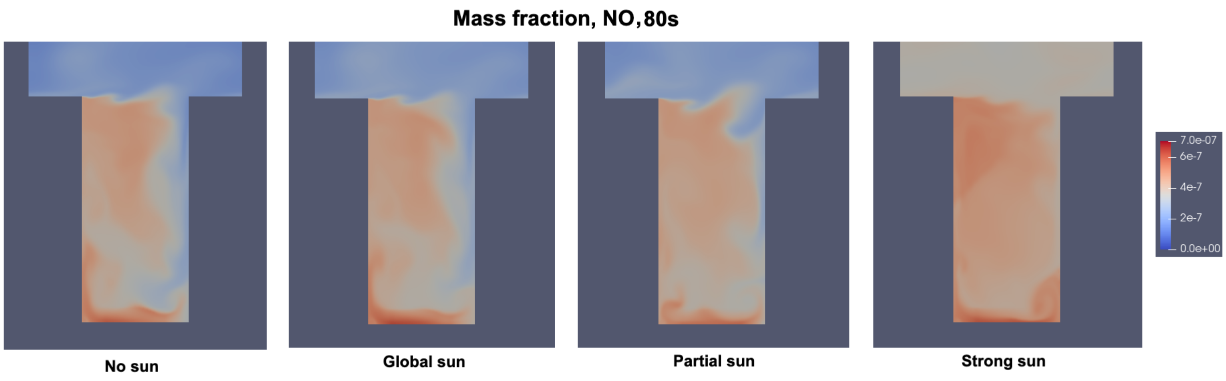


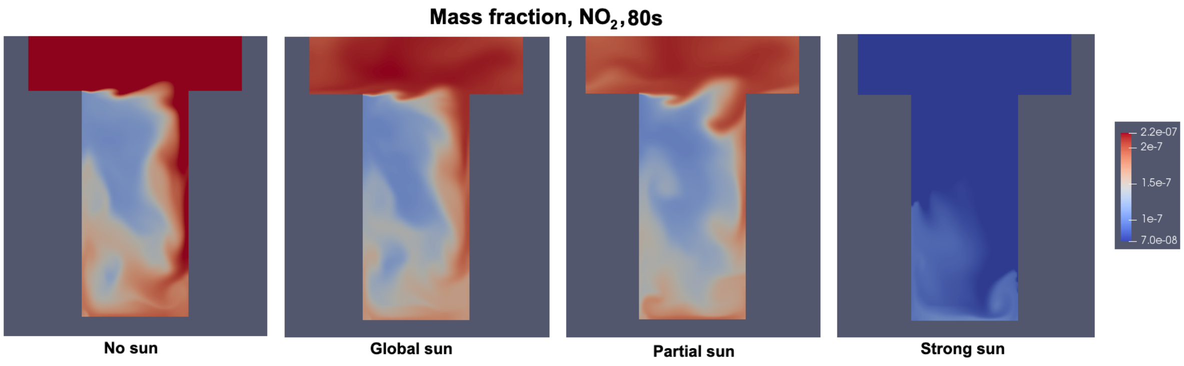
No statistically significant difference between NO concentration levels of the first three sun models can be observed after 500s. The no-sun and no-sun models have similar levels of NO_2 throughout the whole simulation time, while the no-sun model has significantly higher values. However, when it comes to O_3 levels, the no-sun and no-sun models have a much stronger correlation than the no-sun model, which has O_3 levels significantly higher than the other models (with the exception of sky values, where levels are heavily regulated by the source ozone). O_3 levels in the no-sun model are generally unstable, possibly due to the highly contrasting O_3 levels in the shaded area compared to the sunlit area, making convection play a bigger role than in the other models.

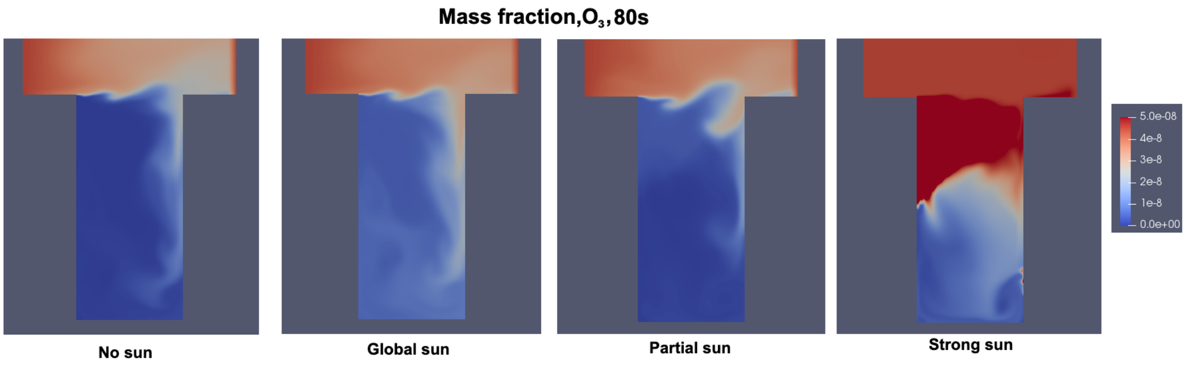
The significantly higher no-sun NO_2 values can be explained by NO_2 having no proper outlet. Since there is no sink that consumes NO_2, values will keep increasing linearly. The presence of sun, whether it be partial or global, is enough to make the trend logarithmic rather than linear. This might be the reason as to why the difference between no-sun and no-sun is bigger than the difference between no-sun and no-sun.

When it comes to O_3 levels, the no-sun model stands out due to there being no O_3 source at ground level. For the no-sun and no-sun models, O_3 can only reach the bottom of the canyon through convection, which as can be seen in the graphs representing street O_3 levels, has a very low impact. Even the no-sun model, which has very high amounts of O_3 levels in the sky, still has low O_3 levels at ground level (on the leeward side). This means that O_3 generated through local photolysis is dominant, hence why the no-sun model has the most ground level ozone.

The distribution of gases can be see more clearly in the following simulation snapshots, taken after 80s:







Here we can see the general mixing process of a typical urban street canyon: NO is produced by vehicles in the bottom of the canyon, which is then transported by the main vortex of the canyon into the sky, where it mixes with ozone through reaction 3.3, becoming NO_2, after which it is transported back into the canyon by the main vortex. Photolytic effects seem to somewhat mitigate sky NO_2 levels, but are not a dominating factor. In the no-sun simulation, photolysis (reaction 3.3) very clearly dominates reaction 3.2, confirming that the two reactions are in contention with each other.

NO_x levels are higher at ground level than in the middle of the canyon. This is likely due to being closer to the source, where turbulent forces help move NO and NO_2 toward the windward side of the street.

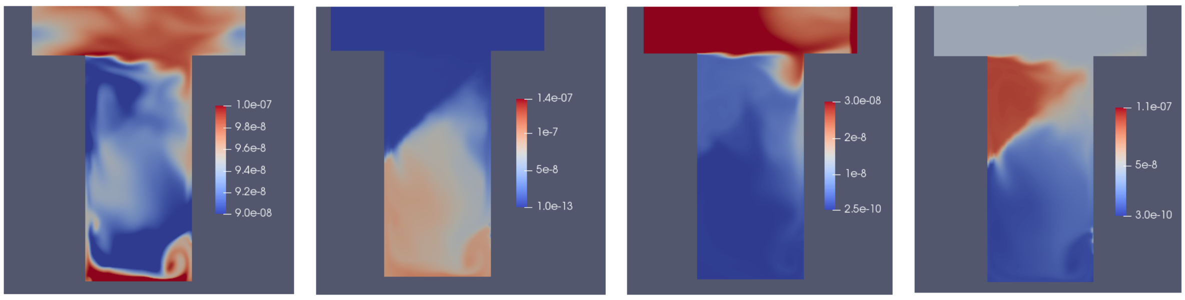
The *NO* figure resembles an inverted version of the NO_2 and O_3 figures. This is due to the relatively high reaction rate of reaction 3.3, leading to a rapid conversion of

NO and O_3 to NO_2 in areas where they coincide. The reason NO and NO_2 tend to not exist in the same areas is that the main source of NO_2, reaction 3.3, consumes NO. The bottom of the canyon is somewhat of an exception to these rules, as a higher ratio of NO and NO_2 emanate from external sources and are not created by chemical reactions.

Despite the depth of the canyon (aspect ratio > 2), NO_2 produced from interacting with above-canyon ozone is still able to travel along the leeward wall and reach the bottom of the canyon. NO and NO_2 are also, to a certain degree, transported by the vortex along the leeward wall into the sky.

Ozone levels are largely determined by sunlight exposure, confirming the results we obtained in the chemFoam simulations. The source of O_3, in most urban settings, comes from an outside source, but the no-sun simulation shows that with strong sunlight, ozone production from reactions 3.3 and 3.4 becomes dominant.

The effect of partial sunlight can be seen more clearly when tuning the scale:

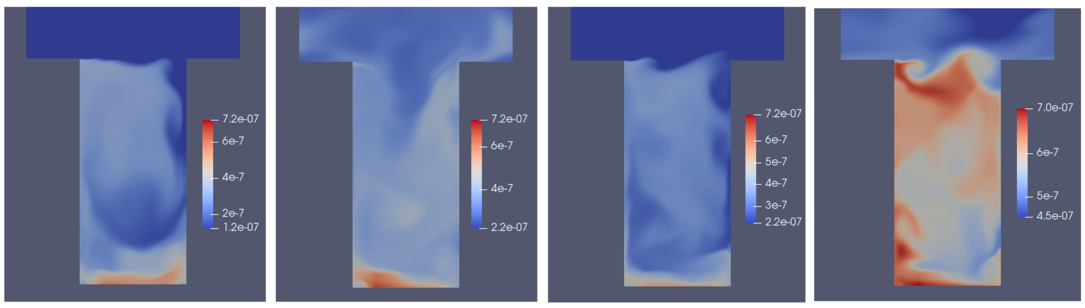


NO_2, partial sun NO_2, strong sun O_3, partial sun O_3, strong sun

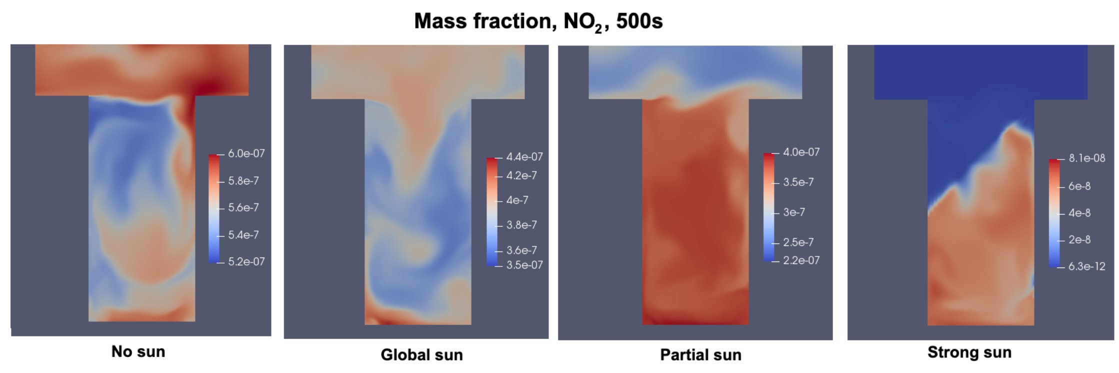
This makes it clear that moderate sunlight also has an effect on the chemistry, and that local sun variation plays a role, albeit seemingly a minor one.

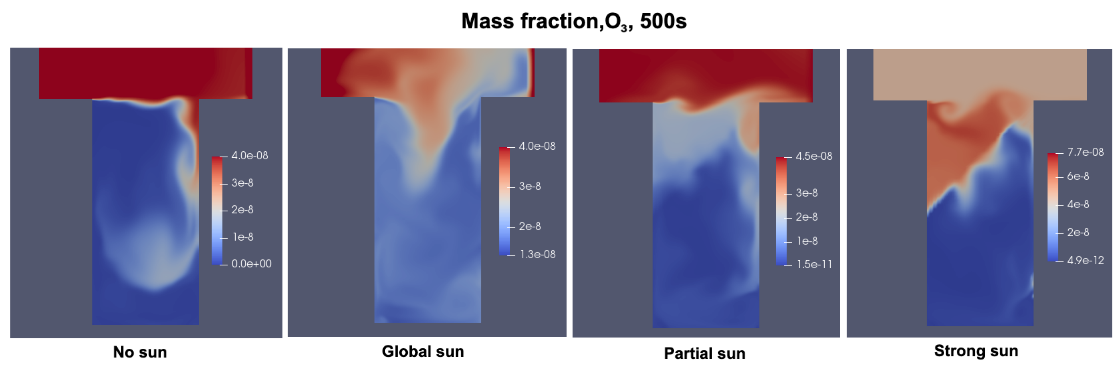
The following are canyon snapshots from the last timestep of the simulation (500s):

Mass fraction, NO, 500s



No sun   Global sun   Partial sun   Strong sun

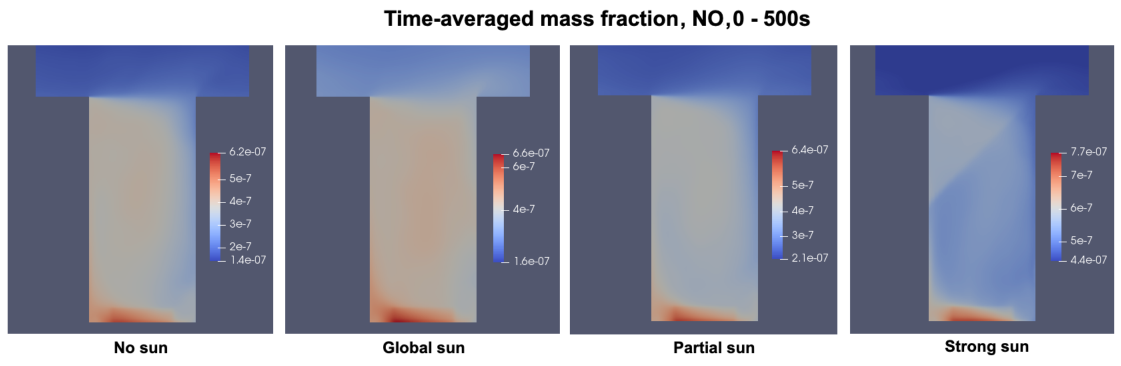


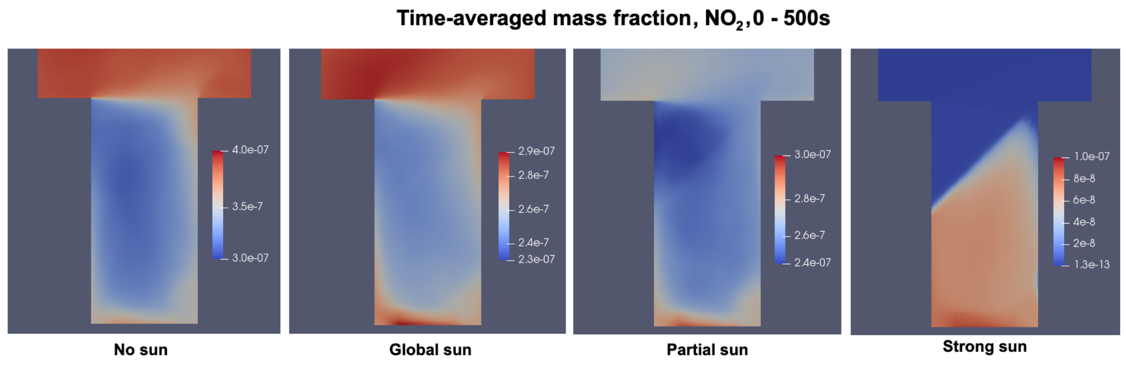


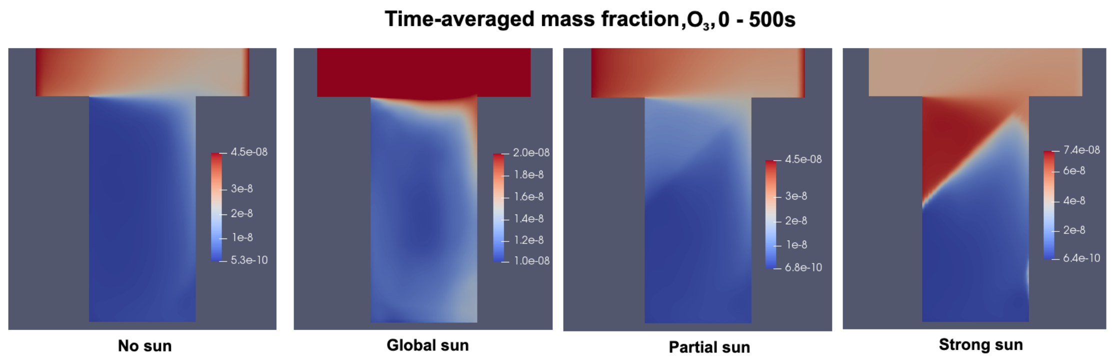
Because of the higher concentration variance between different models compared to the snapshots at 80s, the color range was individually set for each model, meaning that the images represent only the relative concentration distribution within each model. Therefore one must be wary to not make direct visual comparisons between snapshots of different models when looking at this set of images.

Here we can see that relative distributions roughly remain the same after a longer period of time, although overall pollutant concentration levels are higher due to the vehicular NO_x source having been active for a longer duration. An exception to this is the no-sun model, where NO_2 levels are now higher in the canyon than in the sky. The reason for this is unknown. A gradual shift of concentrations where in-canyon NO_2 levels start surpassing sky NO_2 levels can be observed at around the 300s mark of the simulation. It is caused by the no-sun model having lower sky concentration levels. The cause of this cannot be determined at the moment. More research is required in order to determine if this trend is reproducible or not.

We can also observe the concentration levels averaged over the full 500 seconds of the simulation:







These images support the conclusions drawn previously. NO_2 levels are at their highest at the bottom of the canyon close to the source, NO_2 and O_3 enter the canyon through the leeward wall, and NO_2 and O_3 levels are locally affected by sunlight

Chapter 9

Conclusion

Four simulations were set up, measuring NO_x and O_3 dispersion in an urban street canyon measuring 7.2x15m. Each of the four simulations utilized different models representing the photolysis-triggered reaction NO_2 -> NO + O. The no-sun simulation was without photolysis, the global-sun simulation applied photolytic reaction to the entire domain, the partial-sun simulation applied photolytic reaction only to areas of the domain exposed to the sun, and the strong-sun simulation applied photolytic reaction to areas exposed to the sun with a 1000-fold increased intensity rate. In order to perform the partial-sun and strong-sun simulations, a new CFD library was developed to calculate parts of the internal field of the domain exposed to the sun.

For all simulations, NO_x levels were the highest at the bottom of the canyon, near the source. For all simulations except the strong-sun simulation, photolytic effects were dominated by the reaction NO + O_3 -> NO_2 + O, making an area's prevalence of ozone a bigger factor for NO_2 levels than solar exposure. Ozone-rich air reaching the bottom of the canyon through the windward wall is therefore a major contributor to pedestrian and resident NO_2 exposure.

Since photolytic effects were subtle in the simulation, it was beneficial to exclude VOCs from our chemistry model, as including them would have made post-processing and module validation more complex.

The simulations showed significant discrepancies in NO_2 levels between the partial-sun and no-sun models, and also showed significant discrepancies in O_3 levels between the partial-sun and global-sun models, while showing no other statistically significant discrepancies between the models (except for the strong-sun model, which is already assumed to be an outlier).

Assuming that the partial-sun model is the most accurate one (given it has the highest complexity of the models and reflects real-world conditions), this tells us that for future studies regarding only NO and/or NO_2 levels, a global-sun model might be sufficient, while for studies regarding only NO and/or O_3 levels, the no-sun model might be sufficient. For research analyzing the levels of both pollutants simultaneously, however, a model using partial sunlight, as developed in this thesis, might be required for accurate results, as shown in this paper.

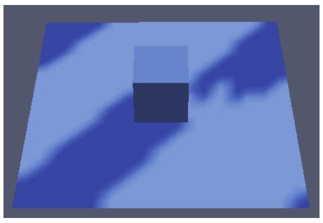
For such studies, until the solar module implemented in this thesis is ready for commercial use, our recommendation for the near future for research including the analysis of both of these pollutants is to only apply photolytic breakdown of NO_2 above the canyon, using a simple horizontal cut-off method. Since convection has been shown in this study to be of minor significance, this should give fairly accurate results as long as you don't make measurements within an area that has sun/shadow wrongly applied to it. When the solar module has been properly integrated into the OpenFOAM framework, our recommendation is to apply it every such simulation instead of using fixed cut-off methods for solar modeling.

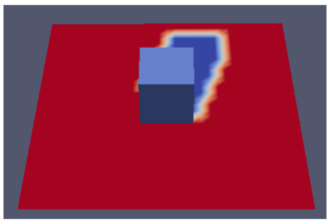
For future work, the following areas should be looked into further:

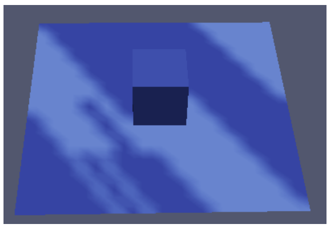
- Several studies having shown them to be significant, an LES simulation utilizing a chemistry model including VOC reactants would be beneficial for the accuracy of the simulation results.
- The J field module developed in tandem with this paper was not fully utilized. Only one time of day was simulated, with a limited runtime of 500 seconds, which is not enough to fully explore the effects of the sun moving over time, which is included in the implemented module. The potential for further research exploring the impact of photolytic effects at different times of day with longer run times should therefore be looked into.
- The module is also not ready for commercial use. There are still errors to be addressed and modifications to be made in the code to make it better adapted to the existing OpenFOAM framework. This may be developed in the future.
- The strong temperature errors given by test simulations using standard A values for the OpenFOAM solarLoad radation model should be further investigated. After calibrating the A value to match the temperature of experimental data, the W/m^2 values reaching the surface of our mesh were much lower than those observed in experimental studies. Most likely there is an error in the conversion from radiative energy to temperature, either in the solarLoad library or in our boundary conditions. A model including radiative heat loss into the atmosphere should be tested on this case, as that would give more accurate temperature values for the simulations.
- The J field solver should be further developed so as to also work with reflective rays and with the FvDOM/view factor radiation models. A more realistic model of radiative heat loss in the mesh would aid greatly in getting simulation temperature levels to coincide better with experimental data.
- More complex geometries should be explored, including obstacles and/or more complex building structures. These simulations would be greatly augmented by the J field module, as shaded areas for each obstacle and building could be calculated in real-time.

Appendix A

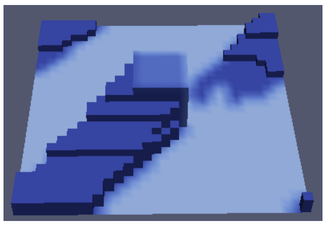
Validation of cyclic J Field solarLoad library

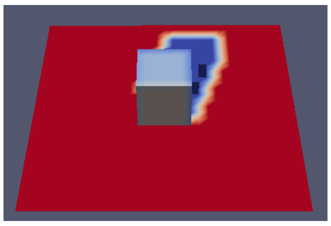

8 AM

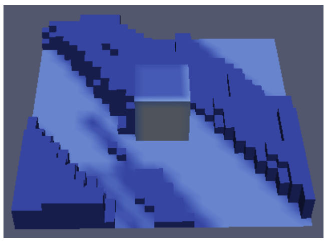

1 PM


4 PM

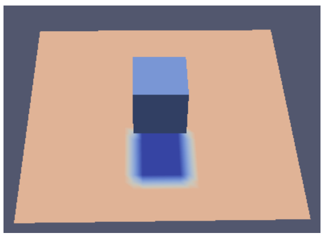
qr field for Tokyo, Japan, January 1st. Scale is 0-80.

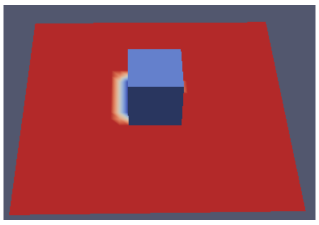

8 AM

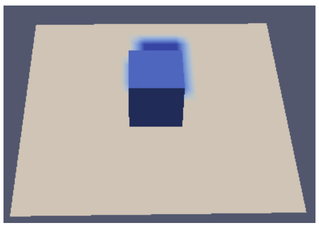

1 PM


4 PM

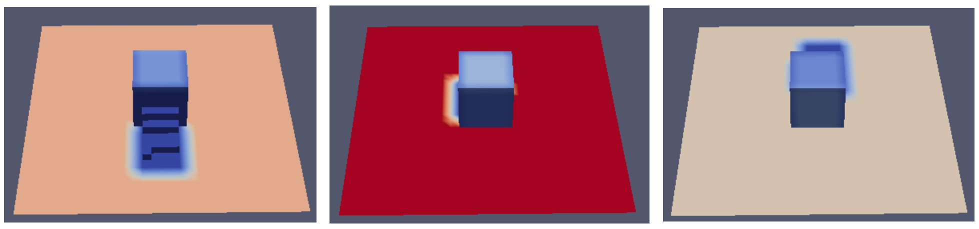
Inverse J field (J=0) for Tokyo, Japan, January 1st


8 AM

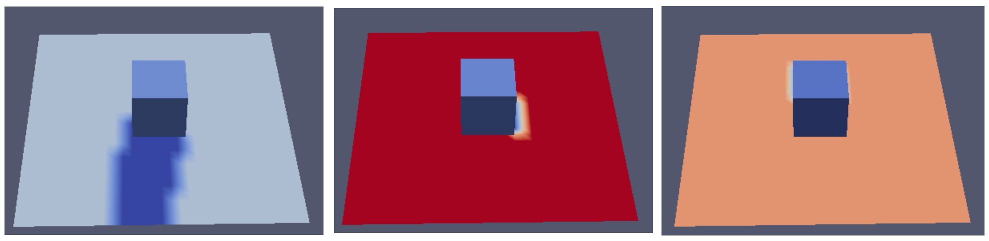

1 PM


4 PM

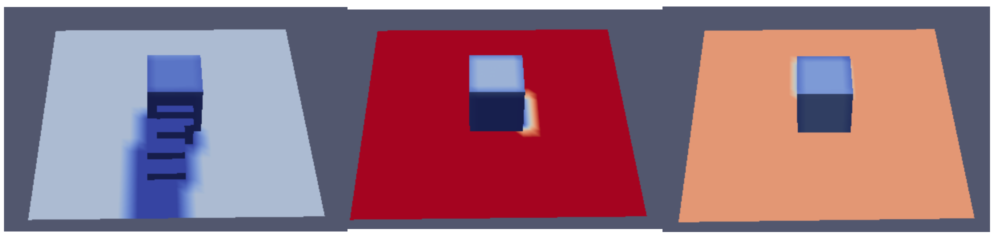
qr field for Tokyo, Japan, July 1st. Scale is 0-160.


8 AM 1 PM 4 PM

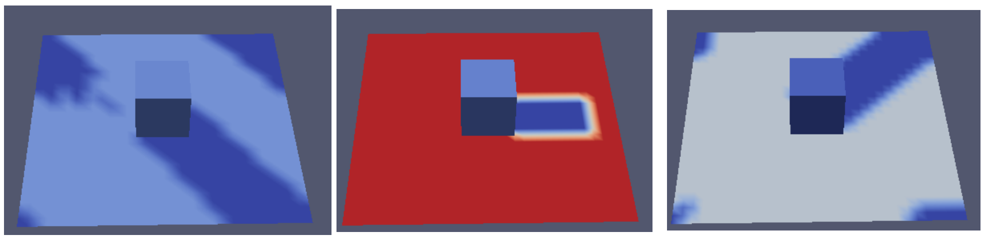
Inverse J field (J=0) for Tokyo, Japan, July 1st


8 AM 1 PM 4 PM

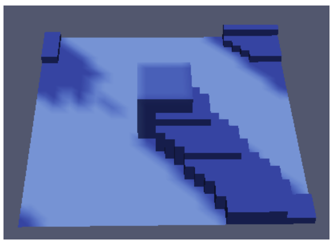
qr field for Cape Town, South Africa, January 1st. Scale is 0-160.


8 AM 1 PM 4 PM

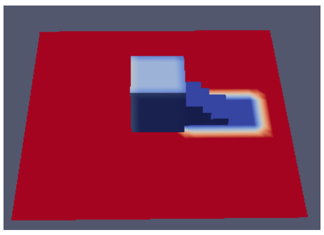
Inverse J field (J=0) for Cape Town, South Africa, January 1st


9 AM 1 PM 4 PM

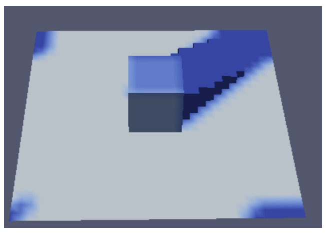
qr field for Cape Town, South Africa, July 1st. Scale is 0-80.



9 AM



1 PM

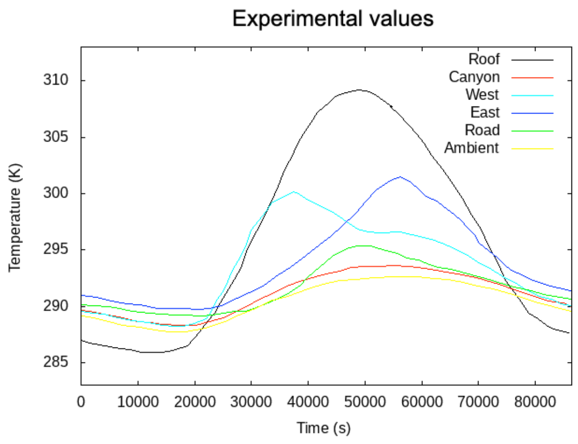


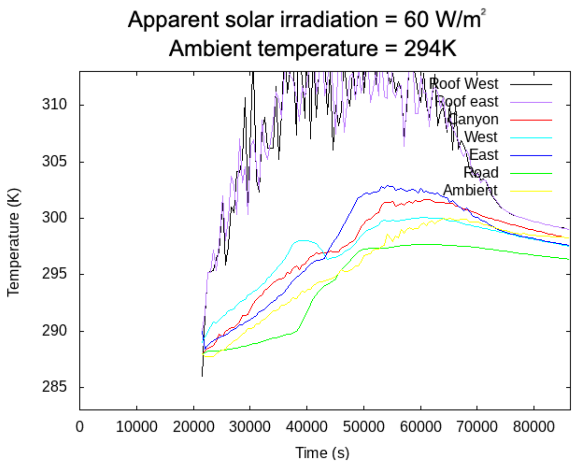
4 PM

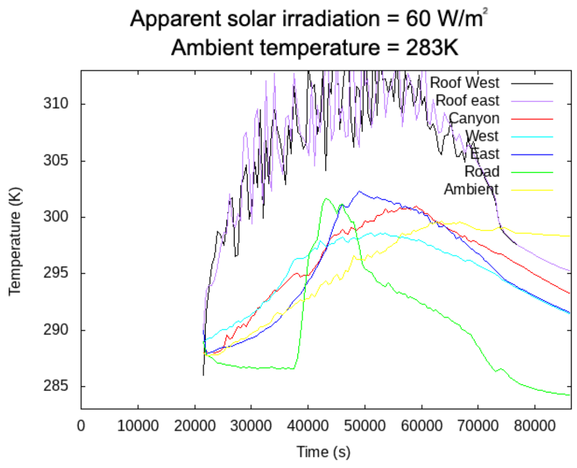
Inverse J field (J=0) for Cape Town, South Africa, July 1st

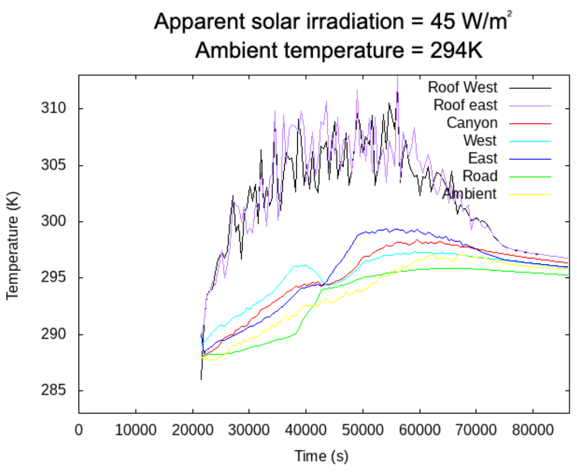
Appendix B

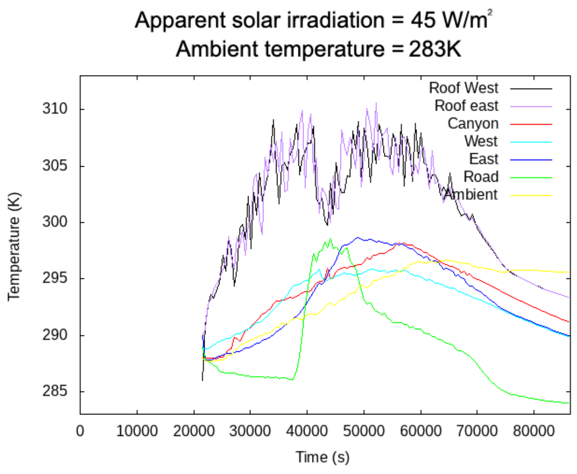
Simulations using different A and Ta values vs. Experimental Data

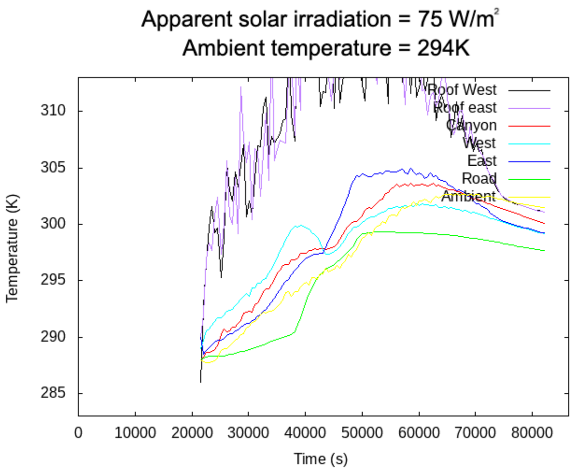

Experimental values
Roof
Canyon
West
East
Road
Ambient
Temperature (K)
Time (s)

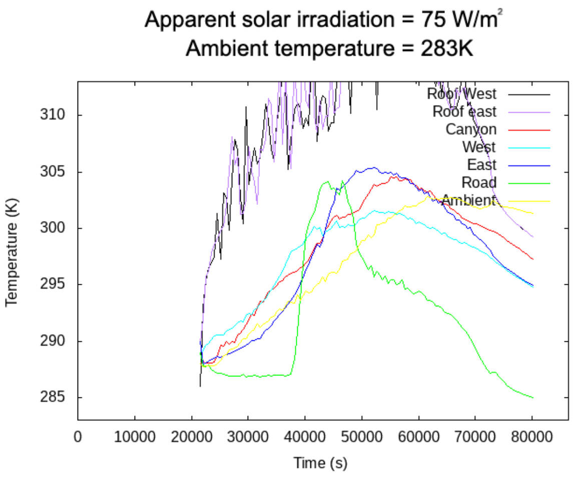

Apparent solar irradiation = 60 W/m²
Ambient temperature = 294K
Roof West
Roof east
Canyon
West
East
Road
Ambient
Temperature (K)
Time (s)

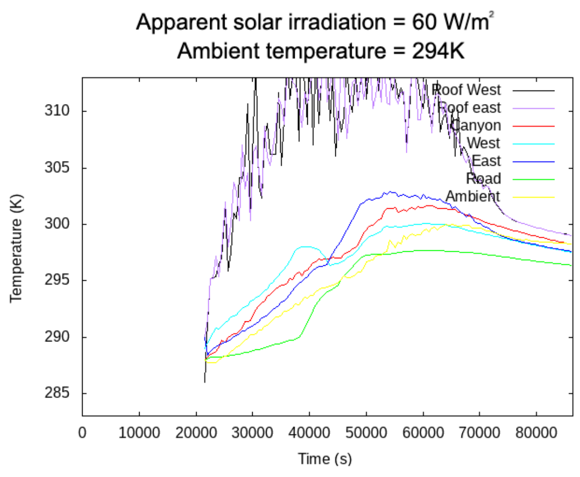

Apparent solar irradiation = 60 W/m²
Ambient temperature = 283K
Roof West
Roof east
Canyon
West
East
Road
Ambient
Temperature (K)
Time (s)

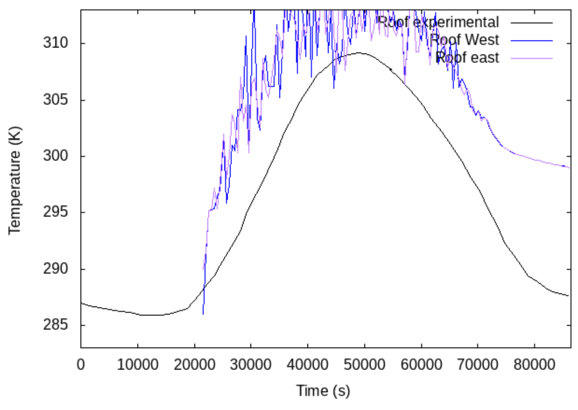

Apparent solar irradiation = 45 W/m²
Ambient temperature = 294K
Roof West
Roof east
Canyon
West
East
Road
Ambient
Temperature (K)
Time (s)

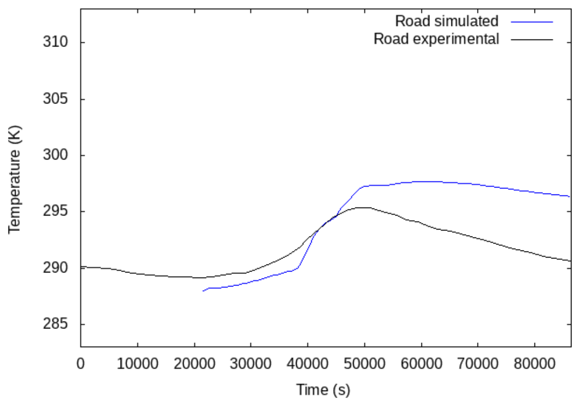

Apparent solar irradiation = 45 W/m²
Ambient temperature = 283K
Roof West
Roof east
Canyon
West
East
Road
Ambient
Temperature (K)
Time (s)

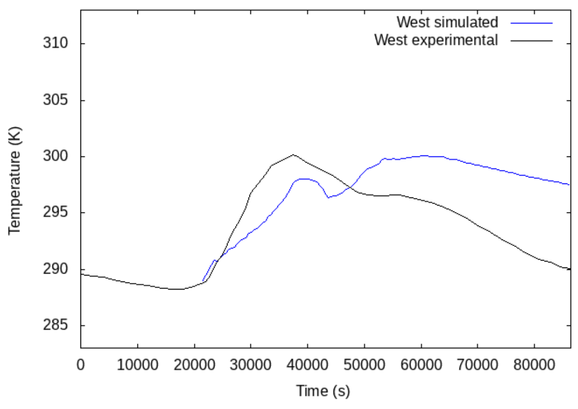

Apparent solar irradiation = 75 W/m²
Ambient temperature = 294K
Roof West
Roof east
Canyon
West
East
Road
Ambient
Temperature (K)
Time (s)

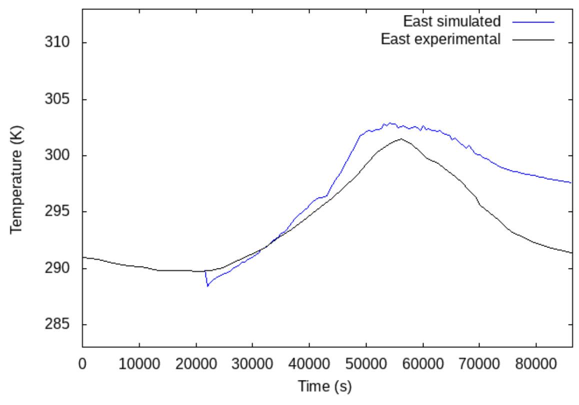

Apparent solar irradiation = 75 W/m²
Ambient temperature = 283K
Roof West
Roof east
Canyon
West
East
Road
Ambient
Temperature (K)
Time (s)

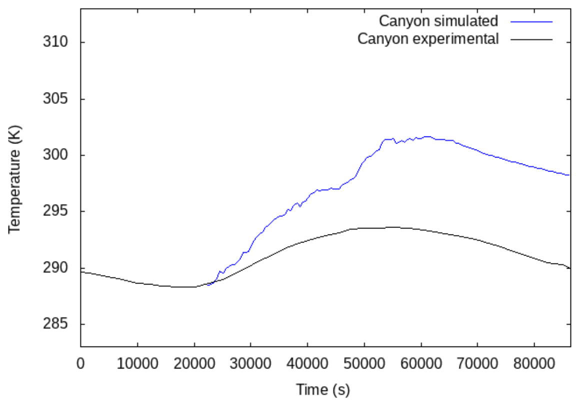

Apparent solar irradiation = 60 W/m²
Ambient temperature = 294K
Roof West
Roof east
Canyon
West
East
Road
Ambient
Temperature (K)
Time (s)

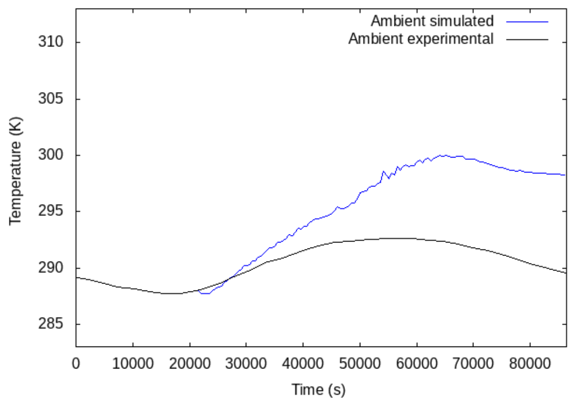

Roof experimental
Roof West
Roof east
Temperature (K)
Time (s)

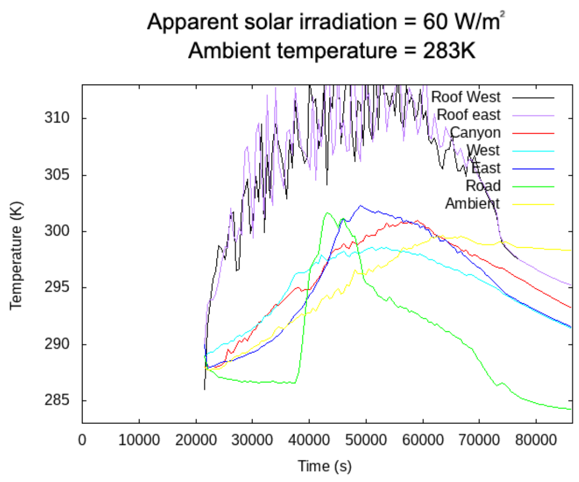

Road simulated
Road experimental
Temperature (K)
Time (s)

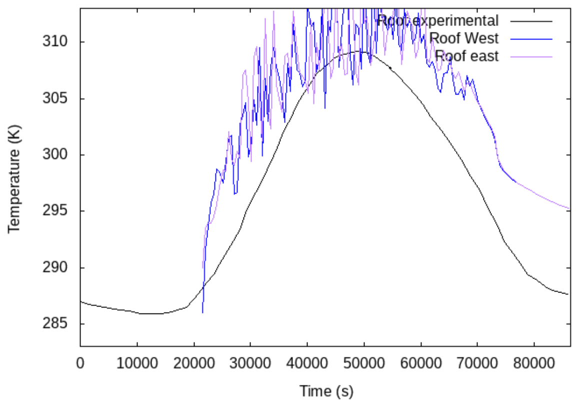

West simulated
West experimental
Temperature (K)
Time (s)

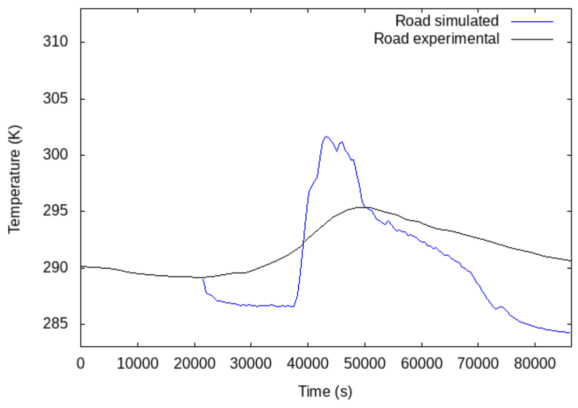

East simulated
East experimental
Temperature (K)
Time (s)

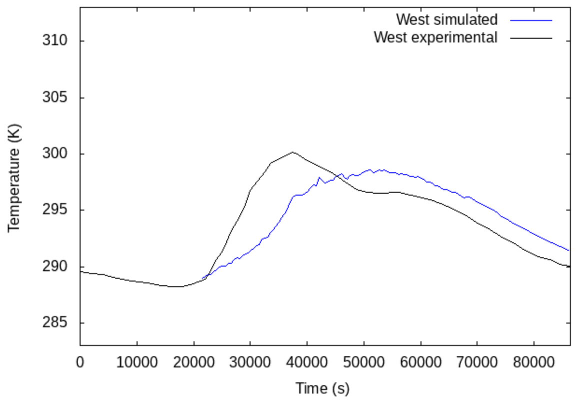

Canyon simulated
Canyon experimental
Temperature (K)
Time (s)

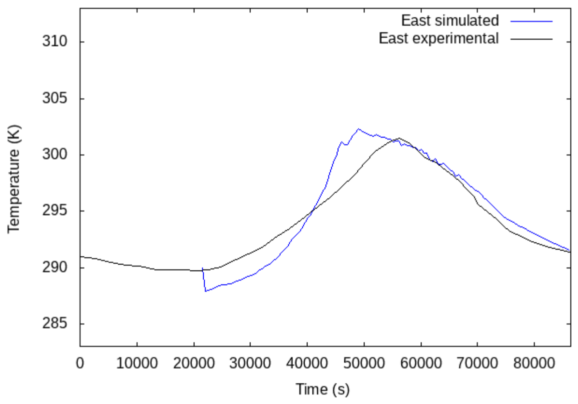

Ambient simulated
Ambient experimental
Temperature (K)
Time (s)

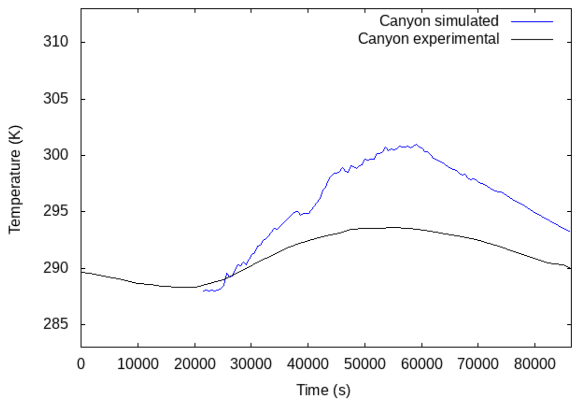

Apparent solar irradiation = 60 W/m²
Ambient temperature = 283K
Roof West
Roof east
Canyon
West
East
Road
Ambient
Temperature (K)
Time (s)

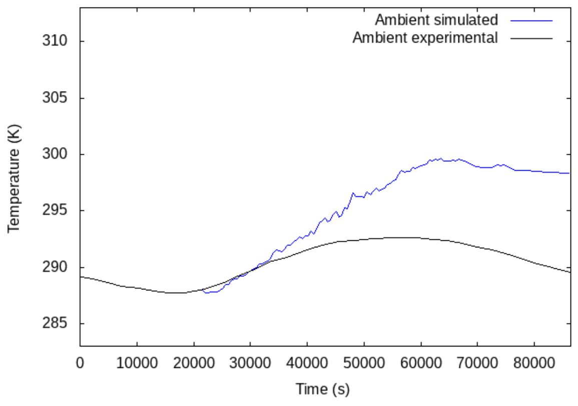

Roof experimental
Roof West
Roof east
Temperature (K)
Time (s)


Road simulated
Road experimental
Temperature (K)
Time (s)

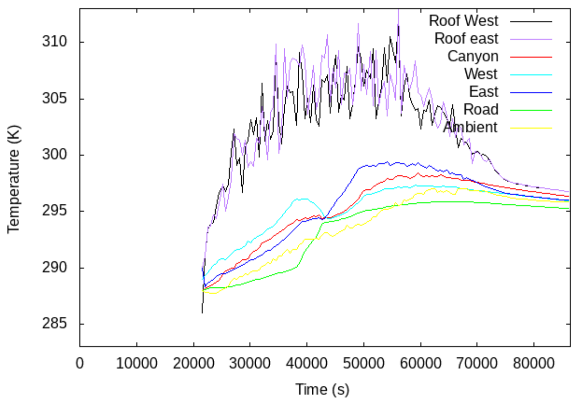

West simulated
West experimental
Temperature (K)
Time (s)

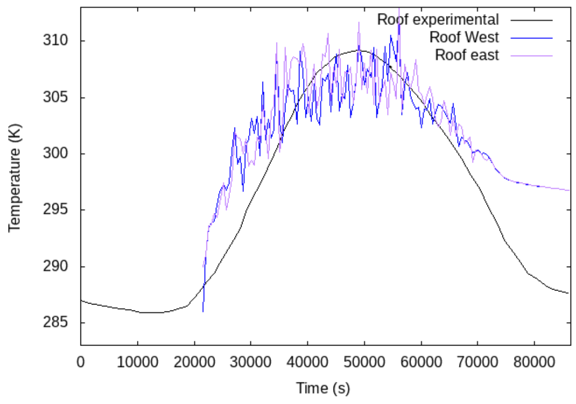

East simulated
East experimental
Temperature (K)
Time (s)

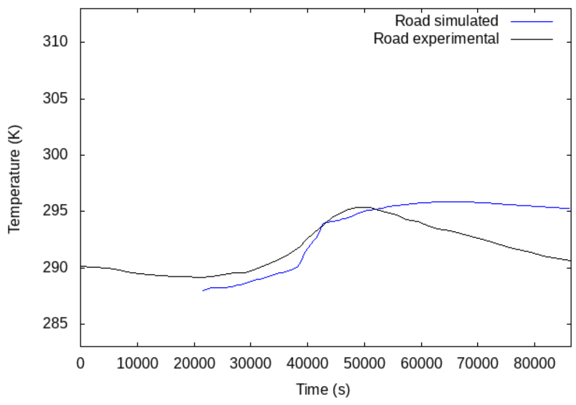

Canyon simulated
Canyon experimental
Temperature (K)
Time (s)

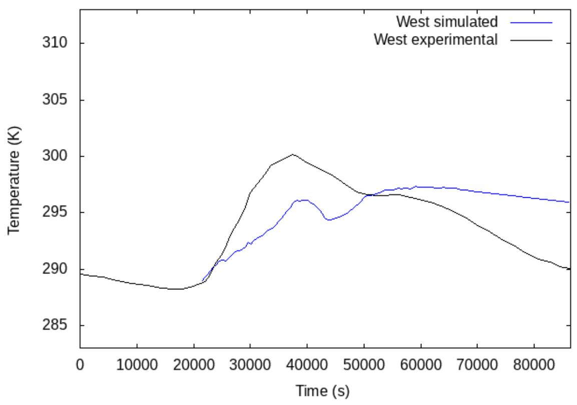

Ambient simulated
Ambient experimental
Temperature (K)
Time (s)

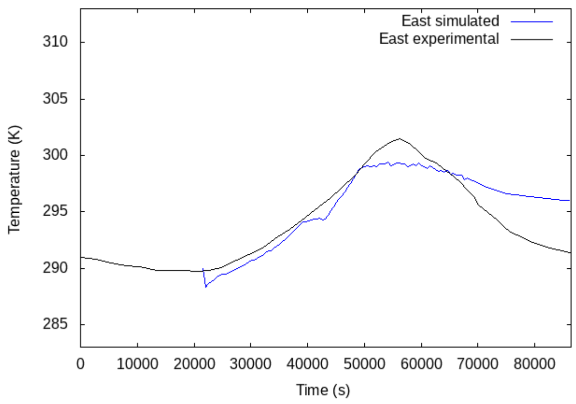
Apparent solar irradiation = 45 W/m^2
Ambient temperature = 294K

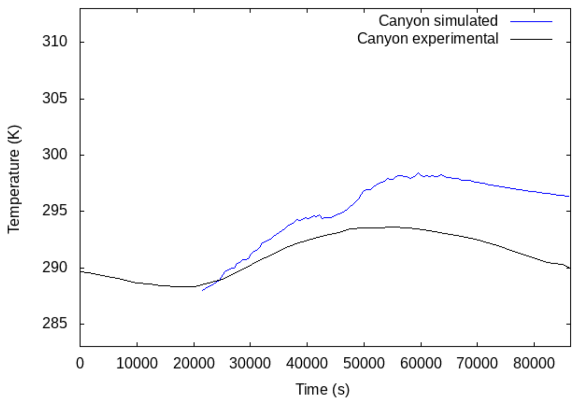


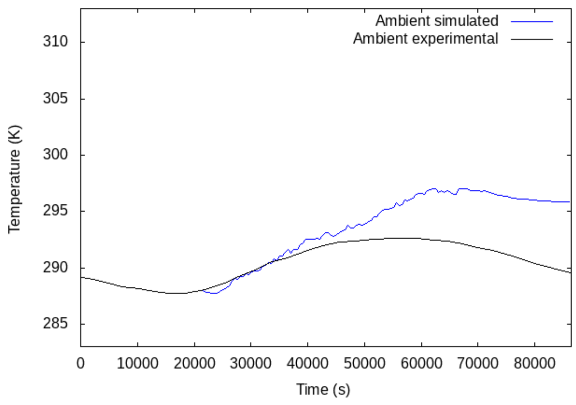




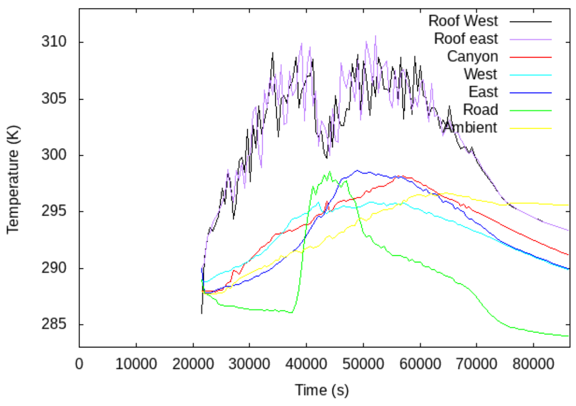


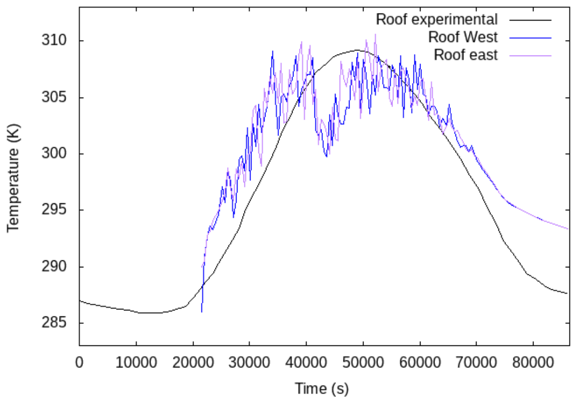


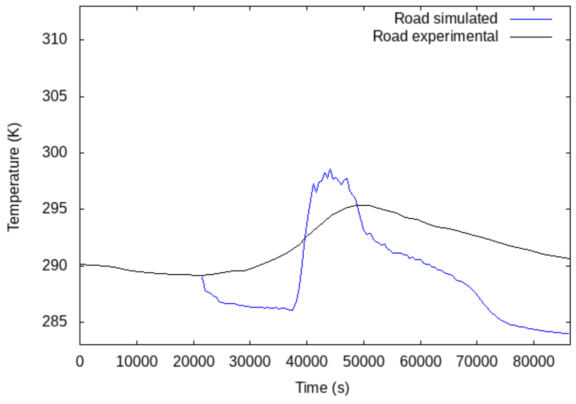


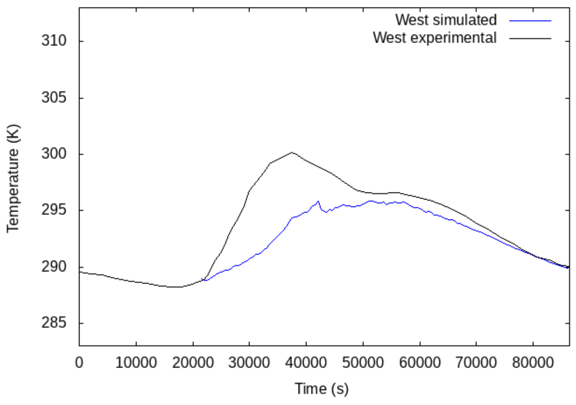


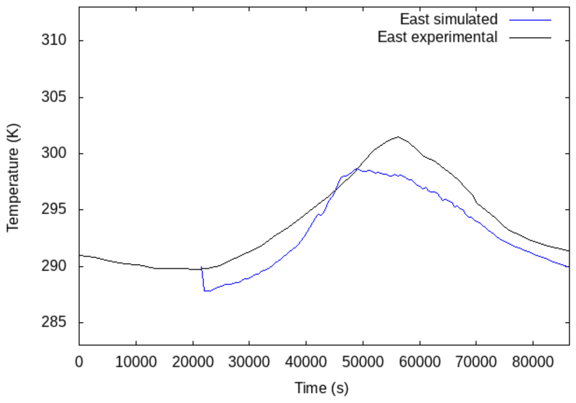
Apparent solar irradiation = 45 W/m^2
Ambient temperature = 283K

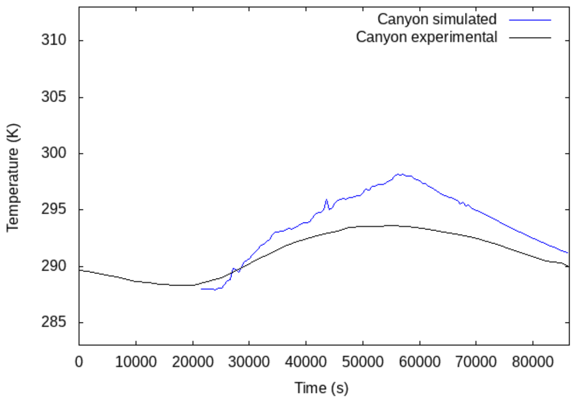


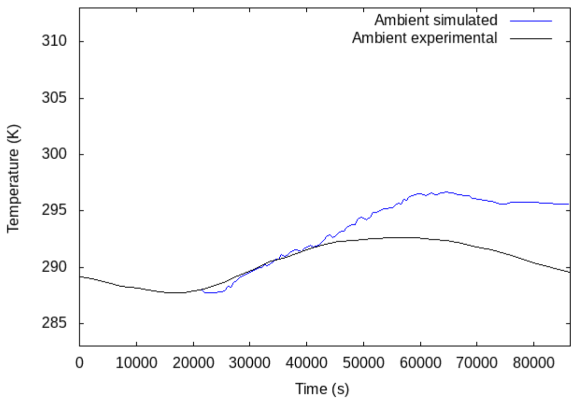




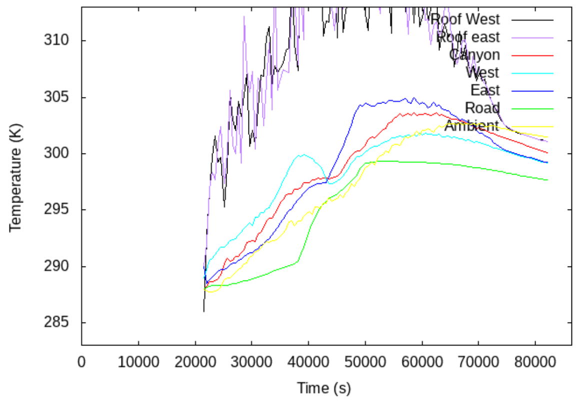


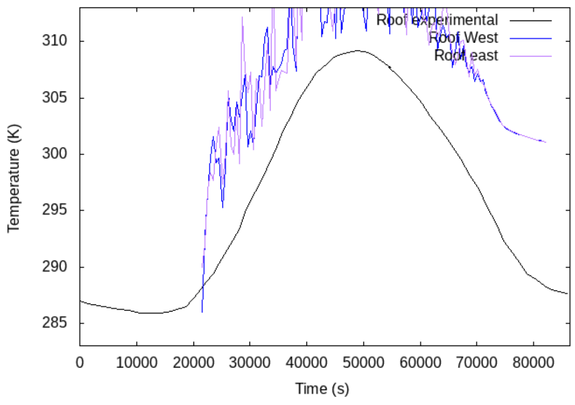


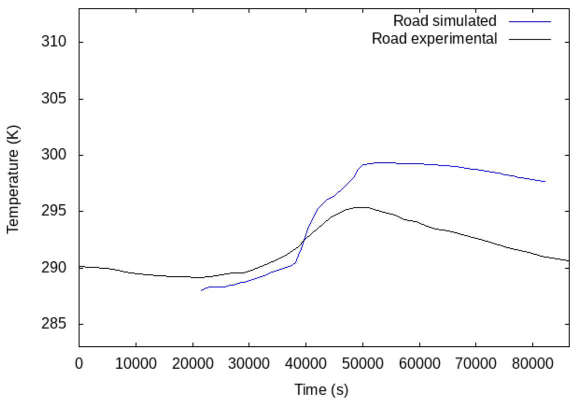


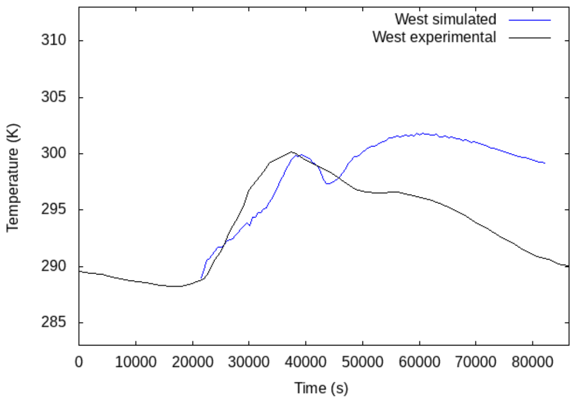


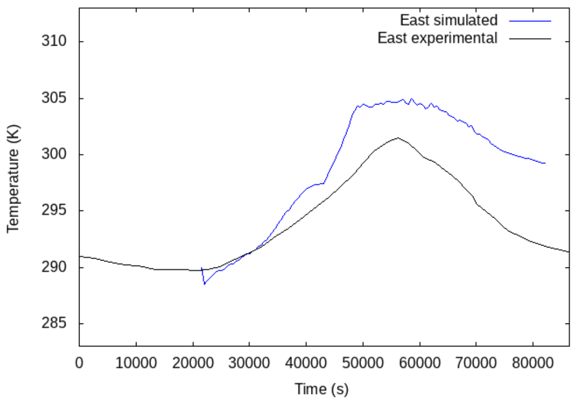
Apparent solar irradiation = 75 W/m²
Ambient temperature = 294K

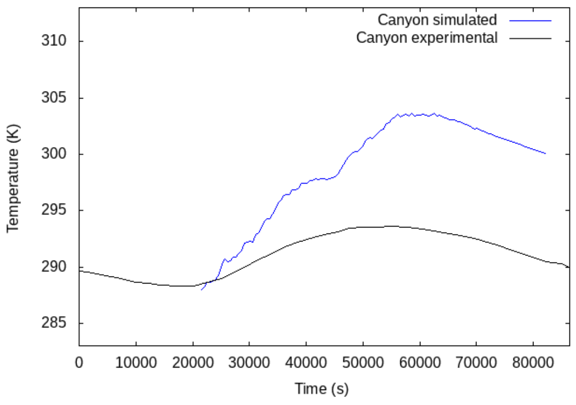


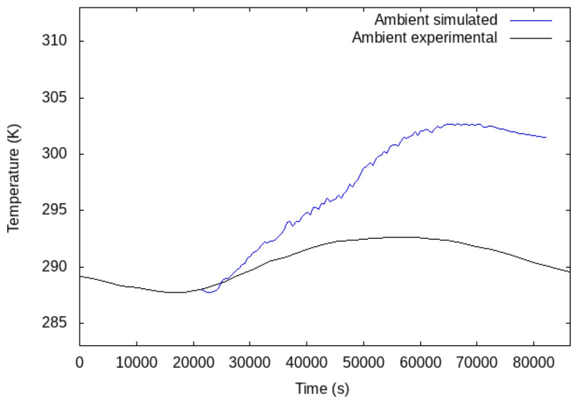






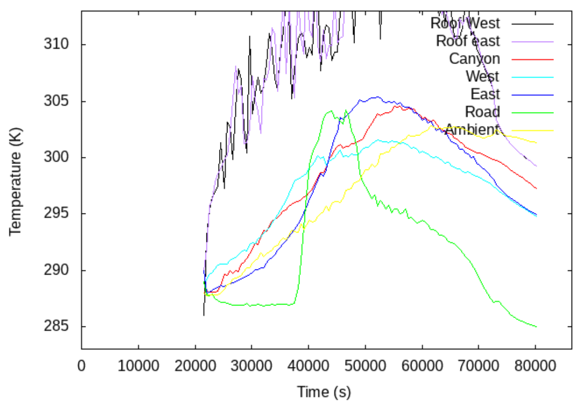


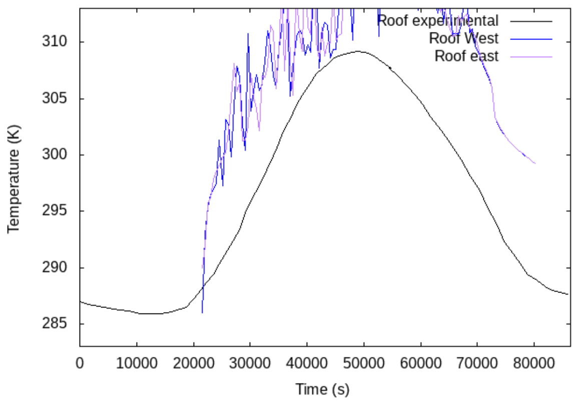


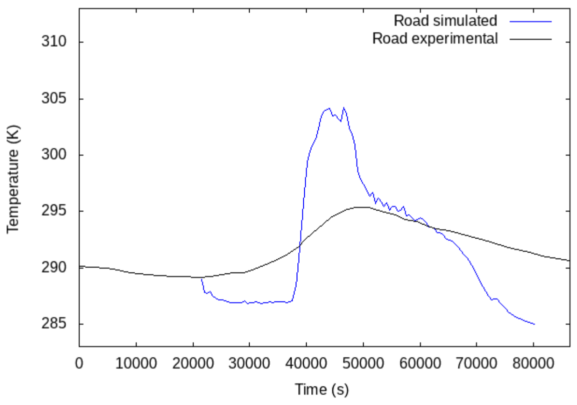


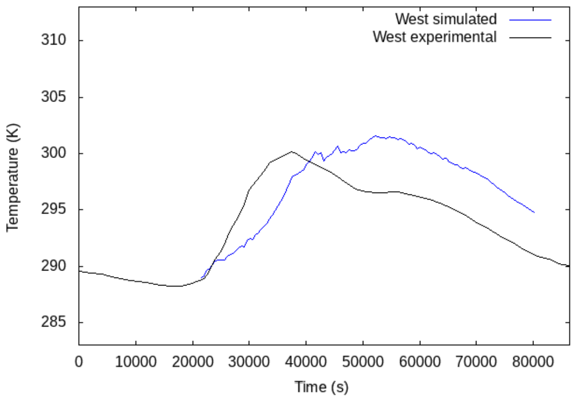
Apparent solar irradiation = 75 W/m^2

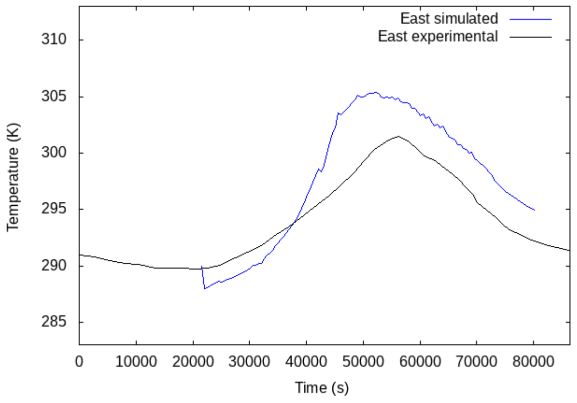
Ambient temperature = 283K

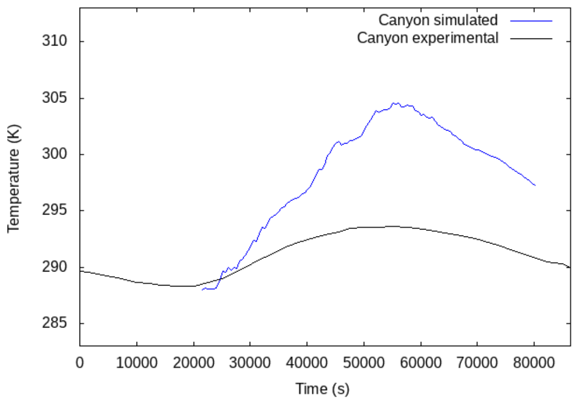


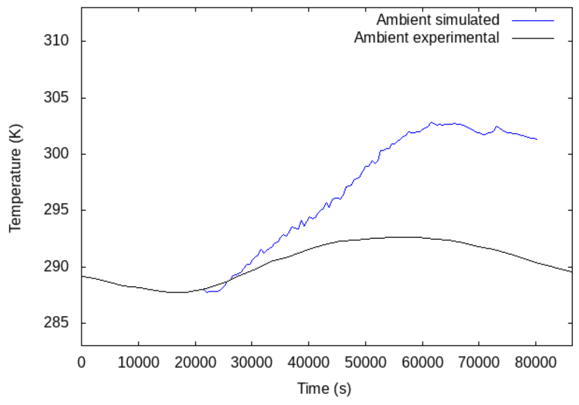












Bibliography

A. Hay, William and Håkan Nilsson (2018). *A low-Mach number solver for variable density flows*. http://dx.doi.org/10.17196/OS_CFD. Accessed: 2021-06-02.

Baker, Jacob, Helen Walker, and Xiaoming Cai (Dec. 2004). "A study of the dispersion and transport of reactive pollutants in and above street canyons - A large eddy simulation". In: *Atmospheric Environment* 38, pp. 6883–6892. DOI: 10.1016/j.atmosenv.2004.08.051.

Bannon, Peter R. (1996). *The Molecular Dynamics of Air*. https://personal.ems.psu.edu/~bannon/moledyn.html. Accessed: 2021-06-10.

Bohnenstengel, S. et al. (Nov. 2004). "Influence of thermal effects on street canyon circulations". In: *Meteorologische Zeitschrift* 13, pp. 381–386. DOI: 10.1127/0941-2948/2004/0013-0381.

Cao, Jie et al. (Feb. 2011). "Association between long-term exposure to outdoor air pollution and mortality in China: A cohort study". In: *Journal of hazardous materials* 186, pp. 1594–600. DOI: 10.1016/j.jhazmat.2010.12.036.

Carpenter, L.J. et al. (1998). "Investigation and evaluation of the NOx/O3 photochemical steady state". In: *Atmospheric Environment* 32.19, pp. 3353–3365. ISSN: 1352-2310. DOI: https://doi.org/10.1016/S1352-2310(97)00416-0. URL: https://www.sciencedirect.com/science/article/pii/S1352231097004160.

Directive 2008/50/EC. https://eur-lex.europa.eu/legal-content/en/ALL/?uri=CELEX:32008L0050. Accessed: 2021-05-30.

Grawe, David, Xiaoming Cai, and Roy Harrison (Nov. 2007). "Large eddy simulation of shading effects on NO2 and O3 concentrations within an idealised street canyon". In: *Atmospheric Environment* 41, pp. 7304–7314. DOI: 10.1016/j.atmosenv.2007.05.015.

Heating, Refrigerating American Society of and Air-Conditioning Engineers (2017). *2017 ASHRAE handbook. Fundamentals*. Atlanta, GA : ASHRAE.

Jacob, Daniel J. (1999). *Introduction to Atmospheric Chemistry*. Accessed: 2021-05-30.

Kikumoto, Hideki and Ryozo Ooka (July 2012). "A study on air pollutant dispersion with bimolecular reactions in urban street canyons using large-eddy simulations". In: *Journal of Wind Engineering and Industrial Aerodynamics* s 104–106, 516–522. DOI: 10.1016/j.jweia.2012.03.001.

Lippmann, H. H., Barbara Jesser, and Ulrich Schurath (1980). "The rate constant of NO + O3 $\rightarrow$ NO2 + O2 in the temperature range of 283–443 K". In: *International Journal of Chemical Kinetics* 12.8, pp. 547–554. DOI: https://doi.org/10.1002/kin.550120805. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/kin.550120805. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/kin.550120805.

Liu, Jiarui et al. (2021). "The influence of solar natural heating and NOx-O3 photochemistry on flow and reactive pollutant exposure in 2D street canyons". In: *Science of The Total Environment* 759, p. 143527. ISSN: 0048-9697. DOI: https://doi.org/10.1016/j.scitotenv.2020.143527. URL: https://www.sciencedirect.com/science/article/pii/S0048969720370583.

Ming, Tingzhen et al. (Oct. 2020). “Field synergy analysis of pollutant dispersion in street canyons and its optimization by adding wind catchers”. In: *Building Simulation*, pp. 1–15. DOI: 10.1007/s12273-020-0720-4.

NEXTFOAM (2017). *Boundary Conditions - OpenFOAM-4.1*. https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwi0q-CWl5XxAhVysYsKHRAfDxsQFjAAegQIAhAD&url=http%3A%2F%2Fwww.nextfoam.co.kr%2Flib%2Fdownload.php%3Fidx%3D135228%26sid%3D235c1d3fc28364657dbb43ccfe025b25&usg=AOvVaw3HM8SNAUzUX7dRmNotaJpm. Accessed: 2021-06-13.

Offerle, B. et al. (Feb. 2007). “Surface heating in relation to air temperature, wind and turbulence in an urban street canyon”. In: *Boundary-layer meteorology* 122, pp. 273–292. DOI: 10.1007/s10546-006-9099-8.

Pu, Yichao and Chao Yang (2014). “Estimating urban roadside emissions with an atmospheric dispersion model based on in-field measurements”. In: *Environmental Pollution* 192, pp. 300–307. ISSN: 0269-7491. DOI: https://doi.org/10.1016/j.envpol.2014.05.019. URL: https://www.sciencedirect.com/science/article/pii/S0269749114002218.

Seinfeld, J and S Pandis (1998). *Atmospheric Chemistry and Physics: From Air Pollution to Climate Change*, p. 1326.

Shetter, R.E. et al. (1988). “Temperature dependence of the atmospheric photolysis rate coefficient for NO2”. In: *Journal of Geophysical Research* 93.D6, pp. 7113–7118.

Tominaga, Yoshihide and Ted Stathopoulos (Apr. 2011). “CFD modeling of pollution dispersion in a street canyon: Comparison between LES and RANS”. In: *Journal of Wind Engineering and Industrial Aerodynamics* 99, pp. 340–348. DOI: 10.1016/j.jweia.2010.12.005.

Versteeg, Henk Kaarle and Weeratunge Malalasekera (1995). *An introduction to computational fluid dynamics - the finite volume method.* Addison-Wesley-Longman, pp. I–X, 1–257. ISBN: 978-0-582-21884-0.

Yassin MF, Ohba M. (July 2012). “Experimental simulation of air quality in street canyon under changes of building orientation and aspect ratio”. In: *J Expo Sci Environ Epidemiol.* 22(5), pp. 502–515. DOI: 10.1038/jes.2012.59.

Yazid, Muhammad et al. (Apr. 2014). “A review on the flow structure and pollutant dispersion in urban street canyons for urban planning strategies”. In: *Simulation* 90. DOI: 10.1177/0037549714528046.

Zhong, Jian, Xiao-Ming Cai, and William James Bloss (2017). “Large eddy simulation of reactive pollutants in a deep urban street canyon: Coupling dynamics with O3-NOx-VOC chemistry”. In: *Environmental Pollution* 224, pp. 171–184. ISSN: 0269-7491. DOI: https://doi.org/10.1016/j.envpol.2017.01.076. URL: https://www.sciencedirect.com/science/article/pii/S0269749116312398.

Zhong, Jian, Xiaoming Cai, and William Bloss (July 2016). “Coupling dynamics and chemistry in the air pollution modelling of street canyons: A review”. In: *Environmental Pollution* 214, pp. 690–704. DOI: 10.1016/j.envpol.2016.04.052.