A Java Package for Simulation of Combustion Engines

Master's thesis
performed in **Vehicular Systems**

by
Johan Gill

Reg nr: LiTH-ISY-EX-3342-2003

5th February 2003

A Java Package for Simulation of Combustion Engines

Master's thesis

performed in **Vehicular Systems**,
Dept. of Electrical Engineering
at **Linköpings universitet**

by **Johan Gill**

Reg nr: LiTH-ISY-EX-3342-2003

Supervisor: **Gunnar Cedersund**
Linköpings Universitet

Examiner: **Assistant professor Lars Eriksson**
Linköpings Universitet

Linköping, 5th February 2003

Avdelning, Institution Division, Department Vehicular Systems, Dept. of Electrical Engineering 581 83 Linköping	**Datum** Date 5th February 2003

Språk Language ☐ Svenska/Swedish ☒ Engelska/English ☐ ______	**Rapporttyp** Report category ☐ Licentiatavhandling ☒ Examensarbete ☐ C-uppsats ☐ D-uppsats ☐ Övrig rapport ☐ ______	**ISBN** — **ISRN** LITH-ISY-EX-3342-2003 **Serietitel och serienummer** **ISSN** Title of series, numbering ——

URL för elektronisk version

http://www.vehicular.isy.liu.se
http://www.ep.liu.se/exjobb/isy/2003/3342/

Titel	Ett javapaket för simulering av förbränningsmotorer
Title	A Java Package for Simulation of Combustion Engines
Författare Author	Johan Gill

Sammanfattning
Abstract

A design and implementation in Java of a one-cylinder combustion engine simulation system using a zero dimensional multi-zone in-cylinder model is described. Simulation results and speed are compared between this system and a system implemented in Matlab and Simulink using the same mathematical model, and the functionality of the two systems is also briefly discussed. Furthermore, the extendibility and maintainability of the Java system are commented and some demonstrations of the multi-zone functionality are done. Suggestions for future work are also given.

Nyckelord
Keywords engine, java, model, simulation, zone

Abstract

A design and implementation in Java of a one-cylinder combustion engine simulation system using a zero dimensional multi-zone in-cylinder model is described. Simulation results and speed are compared between this system and a system implemented in Matlab and Simulink using the same mathematical model, and the functionality of the two systems is also briefly discussed. Furthermore, the extendibility and maintainability of the Java system are commented and some demonstrations of the multi-zone functionality are done. Suggestions for future work are also given.

Keywords: engine, java, model, simulation, zone

Contents

Chapter 1

Introduction

Pressure simulation of a combustion engine is a powerful tool for engine researchers worldwide because of its usefulness for analysing new designs before putting a prototype together. This report covers a design of a system for such simulation, aimed at implementation on the Java platform, and an implementation of the design. The implementation relies on a multi-zone model of a cylinder.

There already exists a Matlab implementation using the same model, but that implementation only supports two zones. This becomes a problem when more detailed spatial information is wanted, since the zones are considered homogeneous. Increasing the number of zones increases the spatial resolution of the model, making analyses possible that otherwise would not be accessible.

1.1 Objectives

The main objective is to design and implement in Java a simulation of the compression and combustion strokes of a spark-ignited combustion engine, using a specified multi-zone model. To handle the differential equation system in the model, the Janet package is to be used.

1.2 Thesis Outline

In chapter 2 the underlying combustion model is covered. Chapter 3 lays out the design of an object-oriented simulation system which uses that model, and also presents some Janet information useful in understanding the design. Chapter 4 presents an implementation of the design. The Matlab system used as reference for comparison is introduced in chapter 5, which also contains the comparison itself. Some suggestions for future work are given in chapter 6. Chapter 7 concludes the thesis.

The interfaces presented in chapter 3 are described in more detail in appendix A. Appendix B lists parameters used in the simulations presented in this work.

Chapter 2

Theory

In this chapter, the operating cycle of a four-stroke combustion engine is described, and the mathematical model used for the simulation system in this thesis is given.

2.1 Purpose of the Model

The model is supposed to facilitate simulation of the operating cycle of spark-ignited combustion engines. Although the model is usable for parts of the operating cycle for two-stroke engines, that case will not be covered here.

2.2 The Four-stroke Combustion Engine

Most cars have a four-stroke combustion engine. It is a *reciprocating* engine, meaning it has one or more cylinders with a piston moving back and forth inside each one of them. In a four-stroke combustion engine, the piston is connected to a crank with a connecting rod. The basic geometry of one such cylinder is given in figure 2.1.

The operating cycle of a four-stroke combustion engine has four strokes, as the name implies. They are as follows.

Intake

The piston moves downwards, making fuel and air flow into the combustion chamber.

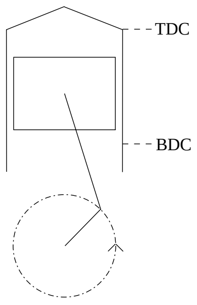

Figure 2.1: Basic geometry of a four-stroke combustion engine.

Compression

The mixture is compressed as the piston moves upwards. Pressure and temperature increases.

Combustion

Also called the Power Stroke. A spark plug ignites the mixture. The combustion makes the pressure rise, and the piston is forced downwards. Hence, chemical energy is transformed into mechanical energy.

Exhaust

The piston goes up again, forcing the burned gases out of the chamber.

2.3 The Model

A zone is a part of the combustion chamber. Two zones can not overlap. A zone contains either unburned or burned fuel. Consider a cylinder with N zones. Let t denote the time passed since the crank angle was 0. The combustion chamber has a certain pressure $p(t)$, volume $V(t)$ and temperature $T(t)$. It is assumed that the mass flow rate $\frac{dm_{ij}}{dt}$ from zone j to zone i is known, for every pair of zones (i, j). Each zone i has its own volume $V_i(t)$, as expected, but also a temperature $T_i(t)$. This is because the zone is assumed to be homogeneous. Using this assumption, a homogeneous combustion chamber can be modelled by using a model with one zone.

Thus, the cylinder state can be described by the combination of the vector

$$x(t) = (p, V_1, T_1, V_2, T_2, \dots).$$

and the mass flow information.

In this work, only the compression stroke and the combustion stroke are simulated. To simulate them, the system of ordinary differential equations $A\dot{x}(t) = B$, with A and B defined below, is solved for the initial time t_{ivc} and with an initial state believed to be a good approximation of $x(t_{ivc})$. No mass flow exists at this point.

$$A = \begin{pmatrix} 0 & 1 & 0 & \dots & 1 & 0 \\ a_1 & p & b_1 & \dots & 0 & 0 \\ c_1 & p & d_1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_N & 0 & 0 & \dots & p & b_N \\ c_N & 0 & 0 & \dots & p & d_N \end{pmatrix}$$

$$B = \begin{pmatrix} \frac{dV}{dt} \\ R_1T_1\sum_{i\neq 1}\frac{dm_{1i}}{dt} \\ \frac{\partial Q_1}{\partial t} + \sum_{i\neq 1}(h_1i - h_1 + R_1T_1)\frac{dm_{1i}}{dt} \\ \vdots \\ R_NT_N\sum_{i\neq 1}\frac{dm_{Ni}}{dt} \\ \frac{\partial Q_N}{\partial t} + \sum_{i\neq N}(h_Ni - h_N + R_NT_N)\frac{dm_{Ni}}{dt} \end{pmatrix}$$

with

$$\begin{aligned} a_i &= V_i(1 - \tfrac{p}{R_i}(\tfrac{\partial R_i}{\partial p})_{T_i}) \\ b_i &= -m_i(R_i + T_i(\tfrac{\partial R_i}{\partial T_i})_p) \\ c_i &= -m_iT_i(\tfrac{T_i}{p}(\tfrac{\partial R_i}{\partial T_i})_p + (\tfrac{\partial R_i}{\partial p})_{T_i}) \\ d_i &= m_i(c_p - R_i - T_i(\tfrac{\partial R_i}{\partial T_i})_p) \end{aligned}$$

$\frac{\partial Q_i}{\partial t}$ is the heat transfer rate from zone i to the cylinder walls. It is calculated as $A_ih(T-T_w)$, where A_i is the cylinder contact area for zone i and h can be determined in many ways. Chapter 12 of Heywood's classic [5] gives more information on heat transfer, and [6] is a book fully devoted to heat transfer. A full derivation of the overall model is given in [3], available from the Vehicular Systems division on Linköping University.

Chapter 3

Design

In this chapter two Java terms are explained and some Janet information is given. A design of an object-oriented simulation system using the theory given in chapter 2 is then presented.

3.1 Java Essentials

Java is an object-oriented programming language developed by Sun Microsystems. Some concepts related to Java will be explained here since they are frequently used in this chapter. An in-depth coverage of Java is beyond the scope of this report, but there are numerous sources of information available. Sun Microsystems has a web site with developer information at [8]. [4] is a book on object-oriented analysis and design.

To implement an interface

A class C implements an interface I if both conditions below hold.

- C has defined all methods and fields contained in I.
- C claims to implement I.

Interface

An interface in Java is a description of methods and fields that have to be present in every class implementing the interface. It is often used to ensure certain functionality in a class, or to signal that a class has a certain property.

3.2 Janet Essentials

Janet is a set of Java classes and interfaces developed at the Department of Physics at DTU, Denmark. The purpose of Janet is to help doing analyses of discrete and time continuous dynamical systems. The interfaces used in the design will be described here. The official web site of Janet is [7].

NonAutonomous

This interface must be implemented by any class representing a time dependent dynamical system.

TimeConscious

This interface signals that the implementing class is conscious about time, and declares methods to get and set time.

TimeContinuous

This interface must be implemented by any class representing a time continuous dynamical system.

VectorType

This interface gives access to various vector operations. The simulation system uses it to store state vectors.

3.3 Building Blocks

It was decided to have five modules in the design. Five Java interfaces corresponding to the modules were created. The interfaces are presented below, with their methods listed. For a description of the methods, see appendix A. Figure 3.1 shows the relations between the modules.

Controller

This is the interface to the simulation control. It handles things like running the simulation and obtaining simulation data. Note that simulation details like the amount of zones and cylinders to use depend on the implementation. Simulation parameters and geometry parameters make it easy to change the simulation situation without changing the actual code. Parameters are explained in section 3.4. Controller inherits the interface TimeConscious from Janet, since it is aware of the

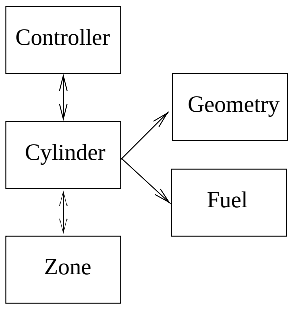

Figure 3.1: The relations implied by the interface definitions. A class implementing an interface at the tail of an arrow has access to the class implementing the interface at the head.

current simulation time. The methods in this interface are getCylinder, numberOfCylinders, setFuel, setGeometry, getGeometryParameter, setGeometryParameter, getInitialTimestep, plot, run, getSimulationParameter, setSimulationParameter, getState, getStates and getTimeVector.

Cylinder

Every Cylinder implementation has means to manage zones and the mass flows between them. Cylinder also contains the equation system to be solved, so changes to the combustion theory will result in changes to the implementation of this interface. Simulation- and geometry parameters local to the cylinder are manipulated here as well. Cylinder extends the Janet interfaces NonAutonomous and TimeContinuous, since it contains the system to be solved. The methods in this interface are getController, getFuel, setFuel, getGeometry, setGeometry, getGeometryParameter, setGeometryParameter, getId, getMassflow, setMassflow, getSimulationParameter, setSimulationParameter, getVibe, getDVibe, addZone, getZone and getZones.

Fuel

This interface declares methods for retrieval of thermochemical properties of the fuel being used. The methods in this interface are getBurnedProperties and getUnburnedProperties.

Geometry

This interface introduces methods related to the geometry of the cylinder. The point in having this separate interface is to make it easy for an existing system to simulate engines of different geometries. A change in geometry that goes beyond changing parameters is reflected by exchanging the class implementing this interface. The methods in this interface are getArea, getVolume and getDVolume.

Zone

A cylinder can have several zones of different types. This interface contains methods to retrieve the thermochemical properties of a zone, to manage neighbours of a zone and to divide a zone. Zone should typically retrieve the thermochemical properties from Fuel. The implementation covered in chapter 4 can be studied for an example of how this can be done. The methods in this interface are getCylinder, divide, getId, setId, addNeighbour, addNeighbours, getNeighbours, getBurnedNeighbours, getUnburnedNeighbours, removeNeighbour, removeNeighbours, getcp, getcv, getDHdp, geth, getMass, getM, getdQ, getDRdp, getDRdT, getR, getPhiBurned, setPhiBurned, getPhiUnburned, setPhiUnburned, getResidualFraction and setResidualFraction.

3.4 Parameters

A parameter is a property whose value is not determined when an implementation of the system is compiled. An implementation can store a parameter for example on disk or in a database. There are two primary kinds of parameters supported by the design:

- Geometry parameters
  Parameters like crank radius and bore should go here.

- Simulation parameters
  Parameters not related to the geometry, angular velocity for instance, should go here.

It is quite common that different cylinders have somewhat different parameters as long as the geometry is not involved. The design supports this, by having parameter methods in Cylinder. A Cylinder implementation not supporting this could just call the corresponding Controller methods. It is less common to have different geometries among cylinders in one engine, but there are such parameter methods in Cylinder as well.

Chapter 4

Implementation

This chapter describes an implementation of the system described in chapter 3. The source code can be obtained from the web page [1].

4.1 Janet Revisited

To understand the implementation better, some more Janet interfaces and classes are covered.

MatrixType

This interface makes various matrix operations available.

RungeKuttaPairIntegrator

This is a pair of explicit Runge Kutta integrators of order four and five. The maximum absolute and relative differences between the results of the two integrators are given by the user and if any difference would exceed the specified value, the time step is reduced and a new integration is attempted. The principles of Runge Kutta integrators are covered in the Numerics book [2], chapter 10.

StandardRungeKutta4Integrator

StandardRungeKutta4Integrator implements the standard fourth order explicit Runge Kutta integrator. This integrator has a fixed time step, which introduces the risk that errors get larger than acceptable.

4.2 Implementation Classes

An implementation of the design from chapter 3 is presented below. A class diagram of the implementation can be seen in figure 4.1. Information needed to run the program is found in the description of SimulatorNZoneState.

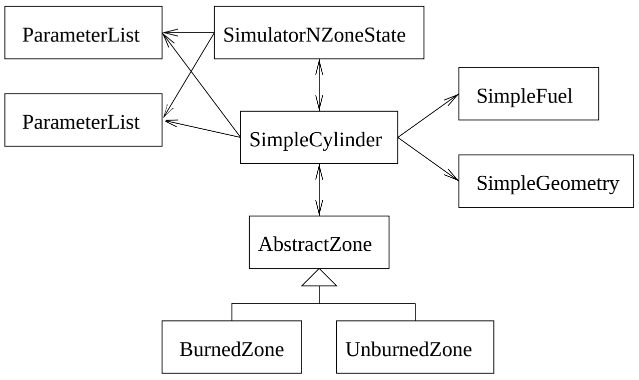

Figure 4.1: An implementation of the design. The class at the tail of an arrow has access to the class at the head. BurnedZone and UnburnedZone extends the abstract class AbstractZone. There can be an arbitrary amount of BurnedZone instances.

SimpleCylinder

SimpleCylinder implements Cylinder. It has a MatrixType which contains the mass flow between every zone. The matrix has to be reallocated for every zone added to the cylinder. An alternative approach could be allocating a larger matrix than needed at first and then also allocate more than needed when space runs out, or calculating the needed size directly.

SimpleFuel

This is an implementation of Fuel. On creation it reads the files `burned` and `unburned`, which contain the tables found in the global variables CHEMPROPB and CHEMPROPU used in the PS implementation (section 5.2). A sequence of Matlab commands to generate the files from the initialised PS implementation is

```
fdunburned = fopen('unburned', 'w');
fwrite(fdunburned, CHEMPROPU.cpu, 'float64');
fwrite(fdunburned, CHEMPROPU.hu,  'float64');
fwrite(fdunburned, CHEMPROPU.Mu,  'float64');
fwrite(fdunburned, CHEMPROPU.T,   'float64');
fwrite(fdunburned, CHEMPROPU.phi, 'float64');
fclose(fdunburned);
fdburned = fopen('burned', 'w');
fwrite(fdburned, CHEMPROPB.cpb,   'float64');
fwrite(fdburned, CHEMPROPB.cvb,   'float64');
fwrite(fdburned, CHEMPROPB.hb,    'float64');
fwrite(fdburned, CHEMPROPB.Mb,    'float64');
fwrite(fdburned, CHEMPROPB.dhbdp, 'float64');
fwrite(fdburned, CHEMPROPB.dRbdp, 'float64');
fwrite(fdburned, CHEMPROPB.dRbdT, 'float64');
fwrite(fdburned, CHEMPROPB.T,     'float64');
fwrite(fdburned, CHEMPROPB.p,     'float64');
fwrite(fdburned, CHEMPROPB.phi,   'float64');
fclose(fdburned);
```

When the properties are requested, they are looked up in the tables. If the properties for a certain set of variables are not present in the tables, they are guessed by interpolation.

SimpleGeometry

This is an implementation of Geometry. It contains functions that depend on the engine geometry. In this implementation there is only one cylinder, and even with multiple cylinders the geometry would probably be shared, but it is possible to have different cylinders access different Geometry implementations.

AbstractZone

This is the base class for all zones in this implementation. It implements Zone methods for neighbour handling and heat transfer. The heat transfer is calculated as in the PS implementation, except for the case of more than two zones. In that case, the cylinder contact area for a burned zone is calculated as the contact area to burned fuel in the 2-zone case divided by the number of burned zones.

A method devised by Woschni to calculate h is used. This method is described in Woschni's article [9].

BurnedZone

This class implements Zone methods that return thermochemical properties for a burned zone. For each time step, the thermochemical properties are retrieved from SimpleFuel if they are not already present. The desired property is then returned.

UnburnedZone

This class implements Zone methods that return thermochemical properties for a burned zone. For each time step, the thermochemical properties are retrieved from SimpleFuel if they are not already present. The desired property is then returned.

ParameterList

This class is used to handle parameters. Every instance reads a file when constructed and puts every name-value pair in a HashMap. If any value is an expression, an instance of Janet.ExpressionParser parses it so a numerical value can be put into the HashMap. This implementation holds two instances of ParameterList.

SimulatorNZoneState

This is an implementation of Controller, and also has a main method so it can be run. The syntax is `java SimulatorNZoneState` *`numZones integrator`*.

`numZones` should be replaced by the number of zones the user wants to allocate space for. The simulation parameter volumeLimit gives the volume limit at which a burned zone is divided.

`integrator` should be 1 to run the simulation with a StandardRungeKutta4Integrator, and 2 if a RungeKuttaPairIntegrator should be used.

The files `simulation.txt`, `engine.txt`, `burned` and `unburned` must be present in the current working directory.

Two instances of ParameterList are created. One instance holds geometry parameters and reads the file `engine.txt`. The other instance holds simulation parameters and reads `simulation.txt`.

The simulation starts at the time given by the simulation parameter `tStart` with a single zone, to handle compression. The initial value of the pressure is 59255.7 Pa, the initial value of the volume is the volume returned by getVolume(`w`*`tStart`), where `w` is the simulation parameter determining the angular velocity of the crank, and the initial value of the temperature is 350. getVolume is contained in SimpleGeometry, of which one instance is created and passed to the constructor of SimpleCylinder.

When the getVibe method in SimpleCylinder returns a value greater than zero, combustion starts. An instance of BurnedZone is created and a mass flow is established between the two zones. The mass flow is then changed every time step since it is calculated as getDVibe*angular velocity*(mass of unburned zone+mass of first burned zone). This is not really right if there is more than one burned zone. Appendix section A.2 has more information on getVibe and getDVibe.

Let the state vector before the start of combustion be (p_0, V_0, T_0). The initial state vector (p, V_1, T_1, V_2, T_2) of the new system is

$$(p_0, (1 - initFrac) \cdot V_0, T_0, initFrac \cdot V_0, T_0 + deltaT_{ad} \cdot (1 - 0.109) \cdot 0.995)$$

where $initFrac$ is the value of the simulation parameter named `initBurnedVolFrac` and $deltaT_{ad}$ is the value of the simulation parameter `deltaT_ad`. Appendix B contains the values of these parameters.

A BurnedZone is divided if its volume after an integration exceeds the value of the volumeLimit simulation parameter. The division strategy is to keep the original temperature, but give both zones resulting from the division half the original volume and to let the new zone get all burned neighbours of the zone getting divided, making the zone getting divided lose all burned neighbours. Research is needed to determine whether the strategy is good enough.

When the time reaches the value of the simulation parameter `tStop`, the simulation ends and three files are written to the current working directory. `states` contains the saved state vectors in a format that Matlab can read, `time` contains the saved times in the same format and `states.info` is a text file containing some information about the two other files. `states.info` can for example look like this:

```
Number of rows 280
Number of columns 5
Start time 0.051923545246831244
Stop time 0.07356562797156091
Initial timestep 2.0E-5
maxStep 8.0E-5
volumeLimit 0.0020
[U1: Unburned neighbours: Burned neighbours: 2
, C2: Unburned neighbours: 1 Burned neighbours:
]
```

The following Matlab code opens, reads and closes the data files.

```
fid1 = fopen('time');
fid2 = fopen('states');
jtime = fread(fid1, 280, 'float64');
jstates = fread(fid2, [280,5], 'float64');
fclose(fid1);
fclose(fid2);
```

The last three lines in `states.info` might look a bit cryptic. It is simply a printout of the zones in the cylinder. U1 means that zone 1 is unburned, C2 means that zone 2 is burned (combusted). The number 2 at the end of one line means that zone 1 has the burned zone with id 2 as neighbour.

After the files are written, the user is asked what he or she wants to plot. Strings accepted are documented in appendix section A.1, under the method plot. Pressing Ctrl-C aborts the program.

4.3 Format of Parameter Files

A special file format has been defined for storage of the parameters. It is as follows.

- A valid file consists of arbitrarily many triplet blocks.
- A triplet block consists of a comment block, a name, a type and a value, in that order.
- A comment block consists of arbitrarily many comments.
- A comment is a line of text that begins with one of the characters !”#$%&’()*+,-./:;<=>?@[\]ˆ_‘{—}˜.
- A name is a line of text that begins with a letter and contains nothing but letters, numbers and the character _.
- A type is a line containing only one word. The possible words are `double`, `string` and `expression`.
- If the type is `double`, the value must be a string in a format that matches a Java floating-point literal, followed by a line terminator and preceded by an optional sign character.
- If the type is `string`, the value can be any line of text.
- If the type is `expression`, the value must be an expression that can be parsed by a Janet ExpressionParser. Appendix B has plenty of examples.

Chapter 5

Comparison and Results

In this chapter another implementation based on the theory in chapter 2 is introduced. Simulation speed and results from the two systems are compared, and differences in functionality are briefly discussed. Some words on the extendibility and maintainability of the Java implementation are given. In addition, some results from using more than two zones in the Java implementation are presented.

5.1 Matlab and Simulink

Matlab is a program particularly useful for operations on matrices. Simulink is an add-on package that makes it possible to define and simulate dynamic systems easily.

5.2 The PS Implementation

The model presented in chapter 2 has also been implemented as a Simulink model. From here on it is referred to as the PS implementation. The Vehicular Systems division of Linköping University has access to this implementation.

5.3 Simulation Setup

If not stated otherwise, the simulations were run with the Janet integrator RungeKuttaPairIntegrator, which is an integrator with variable time step. The maximum relative error was set to 10^{-9}. Appendix B lists the parameters used during all simulations. No simulation has additional parameters, but parameters may have values different from those listed in appendix B. When that is the case, it is noted in its proper context.

The PS implementation was run using the Dormand-Prince integrator, which uses the same principles as the RungeKuttaPairIntegrator, and the maximum relative error was set to 10^{-9}.

5.4 Speed Comparison

The PS implementation was hard to time because it simulates the whole operating cycle. It should however be noted that it needs at least 10 minutes to simulate one full cycle. An older version of the implementation which only handles compression and combustion finishes the simulation in 15–20 seconds.

The Java implementation took 14–15 seconds on the same machine under approximately the same workload, so speed is not a problem so far.

5.5 Result Comparison

Pressure, volume and temperature graphs from both implementations are compared below.

Pressure

Figure 5.1 shows pressure obtained from both simulation systems. According to [3], the measured pressure in a real engine using the same parameters is lower than the pressure given by the PS implementation. If the PS pressure is used as reference, the end pressure from the Java implementation has a relative difference of 4–5%.

Volume

The figures show good agreement between the two implementations. Figure 5.2 shows no visible difference for unburned volume, and figure 5.3 confirms that the burned volumes agree well.

Temperature

The temperature of the unburned zone is compared in figure 5.4. Obviously there is no significant difference there. However, as can be seen in figure 5.5, the temperature of the burned zone is significantly higher in the Java implementation. The relative difference at the end of simulation is 5%. This is logical since the pressure differs, and the volume does not. To elaborate, since $pV_{Java} > pV_{PS}$ and $pV = mRT$ for an

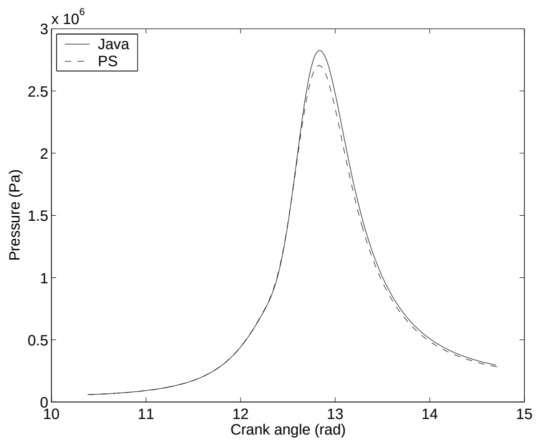

Figure 5.1: Cylinder pressure using two zones. The Java pressure is slightly higher.

ideal gas it follows that $mRT_{Java} > mRT_{PS}$. The agreement between relative differences for pressure and temperature tells that the mass does not differ much between implementations.

5.6 Functionality

Two differences are notable regarding functionality offered.

- At the time of writing, the PS implementation can simulate the whole operating cycle of a four-stroke combustion engine, while the Java system can only simulate compression and combustion.
- The Java system can have arbitrarily many burned zones in the simulation.

5.7 Extendibility and Maintainability

Some things can be noted regarding the Java implementation:

- The Java design consists of interfaces representing concrete concepts like Cylinder and Fuel. If properly done, such a design eases maintenance of the code since the components make it easy to know where to change a certain detail.
- Some effort has gone into making it possible in the future to simulate an engine with more than one cylinder, but the support in place might be insufficient. See also the following point.

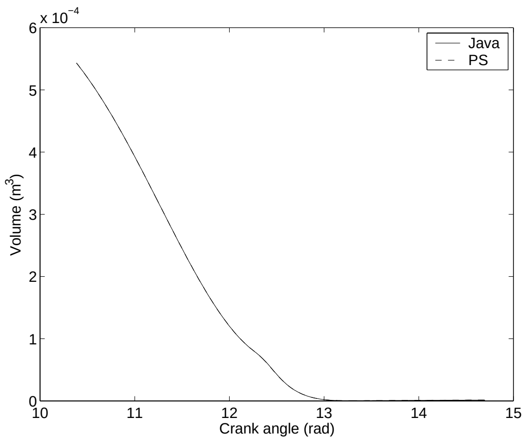

Figure 5.2: The volume of the unburned zone is practically the same in both implementations.

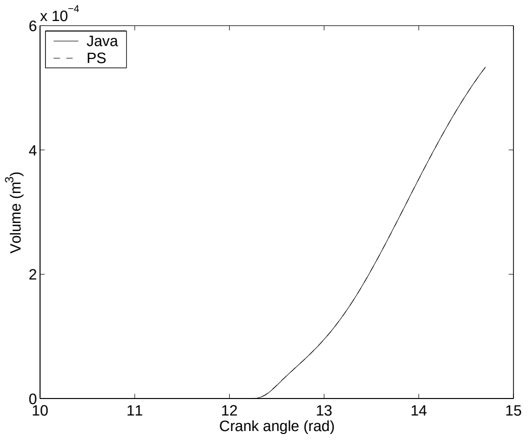

Figure 5.3: The volume of the burned zone is practically the same in both implementations.

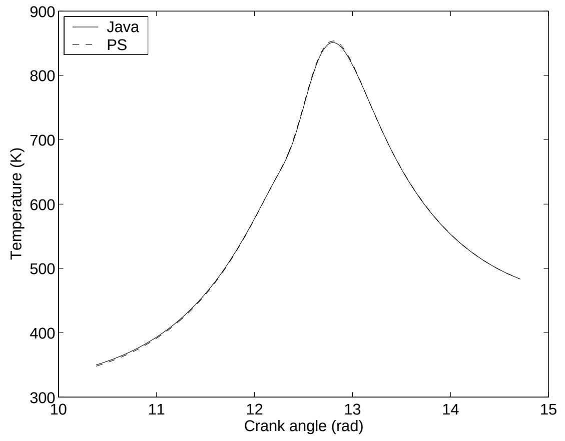

Figure 5.4: The temperature of the unburned zone is practically the same in both implementations.

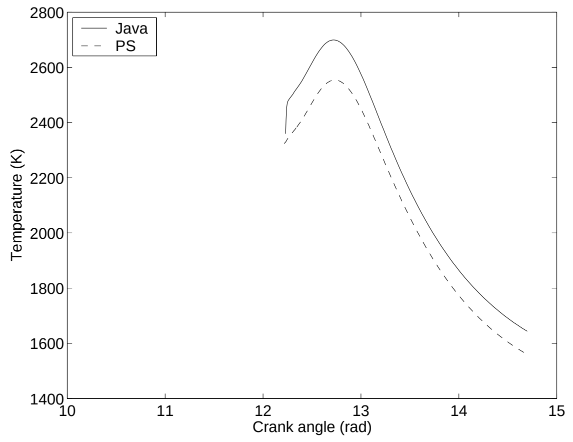

Figure 5.5: The temperature of the burned zone differs significantly.

- The design is supposed to encourage implementations that do not rely on internal details in other components. However, it comes at the cost of having to modify at least one interface if a need for more functionality in the design arises.

It should be noted that the PS implementation is also written with maintainability and extendibility in mind, but without the direct connection between objects in the program and physical objects in the engine that is made possible with the Java design.

5.8 Multi-zone simulation

To demonstrate the multi-zone capabilities of the Java implementation, and to show some remaining issues with it, figure 5.6 shows the pressure graphs for 2 zones, 9 zones and 35 zones. The pressure gets lower with many zones. This could be due to the way the contact area between the zone and the cylinder walls is calculated. At the moment it is just the value valid when using two zones, divided by the number of burned zones.

Figure 5.7 shows the temperature of 34 burned zones at the end of the 35 zone simulation previously used, as a function of the *unburned zone distance* of every zone.

The unburned zone distance of a burned zone is 1 if it is a neighbour of the unburned zone and more generally n if there are n-1 neighbours between the burned zone and the unburned zone.

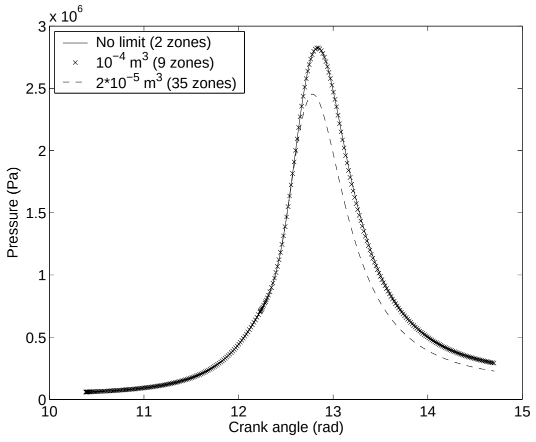

Figure 5.6: Pressures obtained when using different volume limits, resulting in different amounts of zones.

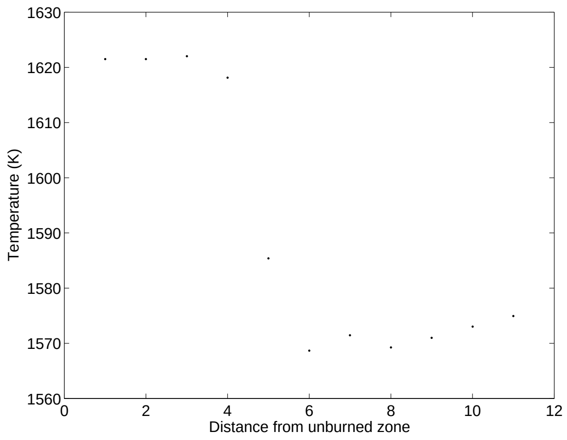

Figure 5.7: Temperature as a function of the unburned zone distance.

Chapter 6

Future Work

In this chapter, some suggestions for future work will be given. This includes possible development of the implementation and the model, as well as usage of the data generated from the current implementation and model.

6.1 Work Already Planned

At the time of writing there are two Master students, just about to start on their projects, who both will do a more or less direct continuation of this work. These projects are described below.

- Analysis of the chemical kinetics behind knock detection: Engine knock is an unwanted state in the running engine. It means that the flame is not spreading smoothly because the fuel ignites spontaneously due to high pressure. This can be highly damaging to the engine, but unfortunately you usually get the most effect out of the cycle right before the onset of knock. Therefore it is desirable to understand it as fully as possible. Given the data from the model and implementation done in this thesis it is possible to do a post-processing of the data (actually another simulation). This will give the chemical composition in each zone and by comparing different cycles, with and without knock, the critical reactions can finally be identified.

- Zonal models for Ionisation Currents: In the gap between the two ends of the place for the ignition there can sometimes flow a current. There are some preliminary models for describing this phenomenon where the current is modelled as a function of pressure and temperature. These models are, however, not spatially extended and since the current is intrinsically a spatial phenomenon (it's a function of the distance between the two ends), this is a fundamental lack in these models. The zonal model, and its implementation developed in this thesis could, however, lay a basis for the development of a *zonal* ionisation current model. Once these are developed some of the time delays which create a current mismatch between the experiments and models of ionisation current might disappear. At the time of writing there is a Master's thesis just starting up, devoted to this question.

6.2 Other Suggestions

- Setting the mass flow more correctly between the unburned zone and the first burned zone.
- Extension of the theory to divide in different ways. This requires that the mass flows between zones currently said to have no mass flow are studied in more detail.
- Implementation of the above given extension.
- Extension of the implementation to include the full cycle.
- Improved handling of the mass flow matrix in SimpleCylinder.
- Improved user interface.
- Extension of the model to include the full vehicle.

Chapter 7

Conclusions

This chapter consists of a summary of the work together with a restatement of all the conclusions drawn.

7.1 Summary

The purpose of the thesis was to implement a given model for a combustion engine in a Java environment, based on the simulation package Janet. The theory behind the model can handle an arbitrary amount of zones and the only existing implementation of it can handle only two. Therefore, one of the main initial goals of the program was to use the full capacity of the theory, i.e. to be able to handle an arbitrary amount of zones. Some other goals were to see whether there were any fundamental differences, be it problems or possibilities, between doing an implementation in Java and an implementation in Matlab.

The first part of the thesis rephrased the model in an object oriented formulation. This meant dividing the various functions needed into interfaces, or building blocks. The building blocks introduced were defined in such a way that they correspond to physical objects, this to make the modelling more intuitive. A more detailed description is given in section 3.3, but the introduced objects are the following:

- Controller: Handling the actual simulation.
- Cylinder: Handles the various zones and keeps track of the mass flows between them.
- Fuel: Handles the retrieval of the thermochemical properties of the fuel to be used.

- Geometry: Contains the functions that changes when the geometry changes.
- Zone: Contains all information specific for each zone, such as its specific thermochemical properties and its neighbours. A zone also has the ability to divide.

After this reformulation a specific example implementation was presented. This implementation can, as was one of the main goals, handle an arbitrary amount of zones. For the special case of two zones it was also shown that there is a reasonable agreement with the existing implementation in Matlab, but there is a relative difference reaching 5% when studying pressure and burned temperature. The performance of the Java implementation was found to be sufficient, and some points where made regarding future maintenance. Some preliminary plots over simulations with more than two zones were also given, this mainly to show that a more deep analysis is possible, and finally some suggestions for future work was given.

7.2 Conclusions

The most important conclusions drawn from this thesis are:

- It is possible to do an implementation of the given combustion model in Java.
- The performance of the here presented Java package, in terms of speed, and the accuracy of the simulations, in terms of simulation output, are for co-solvable problems in reasonable agreement with the existing Matlab implementation. Some further investigation into the differences that exist are however necessary.
- The Java package can handle simulations using more than two zones, which the Matlab implementation cannot do currently.
- The object-oriented design of the Java package has some conceptual advantages. One of these is the possibility to create objects which are more close to physical objects, and therefore make the modelling task more direct.
- The object-oriented design of the Java package also has some maintenance advantages. For instance should the geometry, or the chemical properties of the fuel, or the requirements for division of a zone etc., be subject to change, the location for this change can easily be found by looking at in which physical aspect the change has been made. It can also be carried out simply,

either as a new implementation of the corresponding interface, or as a class that inherits from a previous implementation. It should, however, be noted that the Matlab implementation also has easily exchangeable modules, since it is also written, as far as possible, in an object-oriented manner.

- The output of the simulations done with the Java package can be easily sent to Matlab, should a post-processing of the data find this advisable. This enables the study of the properties of multi-zone models, like temperature gradients comparisons with different number of zones etc. A preliminary plot showing the possibilities is already given.

- The model, and implementation done here, can be a basis for future work. Two examples of this, which are already decided to take place, are the analysis of the chemical background to engine knock and the analysis of ionisation currents.

References

[1] Gunnar Cedersund. Gunnar Cedersund - Information about my Master Students. Available on the Internet. http://www.fs.isy.liu.se/˜gunnar/MasterStudents.

[2] Lars Eldén and Linde Wittmeyer-Koch. *Numerisk analys - en introduktion*. Studentlitteratur, Lund, Sweden, 3rd edition, 1996. In Swedish.

[3] Lars Eriksson. Thermodynamics of Unsteady Flows and Zero Dimensional In-Cylinder Models. 2002.

[4] Johan Fagerström. *Objektorienterad analys och design -en andra generationens metod*. Studentlitteratur, Lund, Sweden, 2nd edition, 1999. In Swedish.

[5] John B. Heywood. *Internal Combustion Engine Fundamentals*. McGraw-Hill, 1988.

[6] J. P. Holman. *Heat Transfer*. McGraw-Hill, 2nd edition, 1997.

[7] Carsten Knudsen. Janet Home Page. Available on the Internet. http://www.fysik.dtu.dk/˜janet.

[8] Sun Microsystems. Developer Services. Available on the Internet. http://developer.java.sun.com/.

[9] G. Woschni. A universally applicable equation for the instantaneous heat transfer coefficient in the internal combustion engine. *SAE Technical Paper 670931*, 1967.

Notation

Abbreviations and Acronyms

BDC	Bottom Dead Centre, position for piston.
TDC	Top Dead Centre, position for piston.

Variables and Parameters

c_p	Mass-specific heat capacity under constant pressure.
R_i	Mass-specific gas constant in zone i. Caution: R can also be the gas constant fulfilling the equation $pV = nRT$.
t_{ivc}	Time at intake valve close.
T_w	Temperature of the cylinder walls.

Appendix A

Package zonal

A.1 Controller

```
public interface class Controller
   implements TimeConscious
```

```
zonal.Controller
```

This interface holds the cylinders together and controls simulation.

Methods

`public Object getGeometryParameter(String name)`
Looks up a geometry parameter. This method must never call getGeometryParameter(String) in Cylinder.

Parameters

`java.lang.String name` : The name of the parameter.

return `java.lang.Object` :
The value of the parameter, or null if the parameter is not set.

`public Object setGeometryParameter(String name, Object value)`

Sets a geometry parameter.

Parameters

`java.lang.String name` : The name of the parameter.

`java.lang.Object value` : The value of the parameter.

return `java.lang.Object` :
The old value of the parameter, or null if the parameter was not previously set.

`public Object getSimulationParameter(String name)`
Looks up a simulation parameter. This method must never call getSimulationParameter(String) in Cylinder.

Parameters

`java.lang.String name` : The name of the parameter.

return `java.lang.Object` :
The value of the parameter, or null if the parameter is not set.

`public Object setSimulationParameter(String name, Object value)`

Sets a simulation parameter.

Parameters

`java.lang.String name` : The name of the parameter.

`java.lang.Object value` : The value of the parameter.

return `java.lang.Object` :
The old value of the parameter, or null if the parameter was not previously set.

`public void setFuel(Fuel fuel)`
Sets a new fuel for all cylinders.

Parameters

`zonal.Fuel fuel` : A new fuel.

`public void setGeometry(Geometry geometry)`
Sets a new geometry for all cylinders.

Parameters

`zonal.Geometry geometry` : A new geometry.

`public Cylinder getCylinder(int i)`
Returns the specified cylinder.

Parameters

`int i` : 1 for the first cylinder, 2 for the second,...

return `zonal.Cylinder` :
Cylinder i, or null if there is no cylinder i.

`public int numberOfCylinders()`
Returns the number of cylinders.

return `int` :
The number of cylinders present in this simulation.

`public VectorType getState(int cylinder)`
Returns the current state of the integration.

Parameters

`int cylinder` : The number of the cylinder whose state is wanted.

return `Janet.VectorType` :
A Janet VectorType representing the current state, with contents p,V1,T1,V2,T2,..., or null if a non-existent state is requested.

`public VectorType [] getStates(int cylinder)`
Returns the matrix containing all the old states of all zones. This is not a copy, so don't alter it unless you really know what you are doing!

Parameters

`int cylinder` : The number of the cylinder whose states are wanted.

return `Janet.VectorType` :
An array of VectorType, where each VectorType represents a property.

`public VectorType getTimeVector(int cylinder)`
Returns the time vector corresponding to the saved states.

Parameters

`int cylinder` :

return `Janet.VectorType` :

`public double getInitialTimeStep()`
Returns the initial timestep of the integration.

return `double` :
The initial timestep of the integration.

`public void plot(String string, VectorType reference, int length)`

Plots a property given by the first parameter.

Parameters

- `java.lang.String string` : ”ip” to plot the pressure in the ith cylinder, ”iVj” to plot the volume of the jth zone in the ith cylinder, ”iTj” to plot the temperature of the jth zone in the ith cylinder.
- `Janet.VectorType reference` : A VectorType to plot against.
- `int length` : The number of relevant entries in reference.

`public int run()`
Runs the simulation.

return `int` :
the number of data rows on success, -1 on failure.

`public int run(int cylinder)`
Runs the simulation.

Parameters

- `int cylinder` : The number of the cylinder to run.

return `int` :
the number of data rows on success, -1 on failure.

A.2 `Cylinder`

```
public interface class Cylinder
   implements NonAutonomous, TimeContinuous
```

`zonal.Cylinder`

This interface represents a cylinder with a geometry.

Methods

`public Controller getController()`
Returns the controller of this cylinder.

return `zonal.Controller` :
The controller of this cylinder.

`public Fuel getFuel()`
Returns the fuel.

return `zonal.Fuel` :

`public Fuel setFuel(Fuel fuel)`
Sets a new fuel for this cylinder.

Parameters

`zonal.Fuel fuel` : A new fuel.

return `zonal.Fuel` :
The old fuel.

`public Geometry getGeometry()`
Returns the geometry of this cylinder.

return `zonal.Geometry` :
The geometry of this cylinder.

`public Geometry setGeometry(Geometry geometry)`
Sets a new geometry for this cylinder.

Parameters

`zonal.Geometry geometry` : A new geometry.

return `zonal.Geometry` :
The old geometry.

`public Object getGeometryParameter(String name)`
Looks up a geometry parameter for this cylinder.

Parameters

`java.lang.String name` : The name of the parameter.

return `java.lang.Object` :
The value of the parameter, or null if the parameter is not set.

`public Object setGeometryParameter(String name, Object value)`

Sets a geometry parameter for this cylinder. This method must never call setGeometryParameter(String, Object) in Controller.

Parameters

`java.lang.String name` : The name of the parameter.
`java.lang.Object value` : The value of the parameter.

return `java.lang.Object` :
The old value of the parameter, or null if the parameter was not previously set.

`public double getMassflow(int to)`
Returns the massflow to the zone with id to.

Parameters

`int to` : The id of the destination zone.

return `double` :
The massflow to the specified destination zone.

`public double getMassflow(int to, int from)`
Returns the massflow between two specified zones.

Parameters

`int to` : The id of the destination zone.
`int from` : The id of the source zone.

return `double` :
The massflow from source to destination.

`public void setMassflow(int to, int from, double flow)`
Sets the massflow between two specified zones.

Parameters

`int to` : The id of the destination zone.

`int from` : The id of the source zone.

`double flow` : The massflow from source to destination.

`public Object getSimulationParameter(String name)`
Looks up a simulation parameter of this cylinder.

Parameters

`java.lang.String name` : The name of the parameter.

return `java.lang.Object` :
The value of the parameter, or null if the parameter is not set.

`public Object setSimulationParameter(String name, Object value)`

Sets a simulation parameter of this cylinder. This method must never call setSimulationParameter(String, Object) in Controller.

Parameters

`java.lang.String name` : The name of the parameter.

`java.lang.Object value` : The value of the parameter.

return `java.lang.Object` :
The old value of the parameter, or null if the parameter was not previously set.

`public double getDVibe(double angle)`
Returns the derivative, with respect to angle, of the burned mass fraction.

Parameters

`double angle` : The current crank angle.

return `double` :
The derivative, with respect to angle, of the burned mass fraction. If the combustion has not started, 0 is returned.

`public double getVibe(double angle)`
Returns the burned mass fraction.

Parameters

`double angle` : The current crank angle.

return `double` :
The burned mass fraction. If the combustion has not started, 0 is returned.

`public List getZones()`
Returns all the zones of the cylinder.

return `java.util.List` :
A list of all zones of the cylinder. The zones are sorted with respect to ID, starting with the zone with the smallest ID.

`public Zone getZone(int id)`
Returns the zone with the given ID.

Parameters

`int id` : The ID of the zone to retrieve.

return `zonal.Zone` :
The zone with the given ID, or null if there is no such zone.

`public void addZone(Zone zone)`
Adds a zone to this cylinder. The zone gets an id greater than zero at this point.

Parameters

`zonal.Zone zone` : Zone to add.

`public int getId()`
Returns the id of this cylinder.

return `int` :
An integer id with the property that getCylinder(id) in the Controller implementation returns this cylinder.

A.3 `Fuel`

`public interface class Fuel`

`zonal.Fuel`

Methods

`public VectorType getBurnedProperties(double T, double p, double phi)`

Returns thermochemical properties of burned mixture.

Parameters

`double T` : The temperature of the mixture.

`double p` : The pressure in the cylinder.

`double phi` : The fuel/air ratio of the mixture.

return `Janet.VectorType` :
A VectorType (cp, cv, h, R, dhdp, dRdp, dRdT).

`public VectorType getUnburnedProperties(double T, double phi)`

Returns thermochemical properties of unburned mixture.

Parameters

`double T` : The temperature of the mixture.

`double phi` : The fuel/air ratio of the mixture.

return `Janet.VectorType` :
A VectorType (cp,h,R).

A.4 Geometry

`public interface class Geometry`

`zonal.Geometry`

Methods

`public double getArea(Cylinder cylinder, double theta)`
Calculates the cylinder and piston area exposed to heat transfer for the crank angle theta. The cylinder liner area above the position when the piston is at TDC is neglected.

Parameters

`zonal.Cylinder cylinder` : The cylinder.

`double theta` : The crank angle.

return `double` :
The calculated area.

`public double getVolume(Cylinder cylinder, double theta)`
Calculates the cylinder volume for the crank angle theta, and adds the clearance volume.

Parameters

`zonal.Cylinder cylinder` :

`double theta` : The crank angle.

return `double` :
The total cylinder volume. handled.

`public double getDVolume(Cylinder cylinder, double theta)`
Calculates the crank angle derivative of the cylinder volume for the crank angle theta.

Parameters

`zonal.Cylinder cylinder` :

`double theta` : The crank angle.

return `double` :
dV/dtheta.

A.5 Zone

`public interface class Zone`

`zonal.Zone`

This is the interface of a zone, ie all features common to burned and unburned zones. All methods taking the system state as a parameter expect the order p,V1,T1,V2,T2,...

Methods

`public List getNeighbours()`
Returns all the neighbours of this zone.

return `java.util.List` :
A list containing the neighbours.

`public List getBurnedNeighbours()`
Returns all burned neighbours of this zone.

return `java.util.List` :
A list containing the burned neighbours.

`public List getUnburnedNeighbours()`
Returns all unburned neighbours of this zone.

return `java.util.List` :
A list containing the unburned neighbours.

`public void addNeighbour(Zone neighbour)`
Adds a neighbour to this zone. This method does not add this zone as a neighbour of the other though.

Parameters

`zonal.Zone neighbour` : The zone that should be added.

`public void addNeighbours(List neighbours)`
Adds neighbours to this zone. This zone is also added as a neighbour to all zones in the list.

Parameters

`java.util.List neighbours` : A list containing the zones that should be added.

`public Zone divide(List movedNeighbours)`

Makes the zone divide itself so that the copy gets movedNeighbours as neighbours and the original gets the remaining neighbours. The massflow of the original zone is preserved and the new zone is given no massflow.

Parameters

- `java.util.List movedNeighbours` : The neighbours to move to the copy.

return `zonal.Zone` :
The new zone.

`public Zone divide(List neighboursForCopy, List neighboursForOriginal, VectorType massflowToCopyNeighbours, VectorType massflowToOriginalNeighbours)`

Makes the zone divide itself.

Parameters

- `java.util.List neighboursForCopy` : Neighbours that the copy will get.
- `java.util.List neighboursForOriginal` : Neighbours that the original will get.
- `Janet.VectorType massflowToCopyNeighbours` : Massflow to neighboursForCopy.
- `Janet.VectorType massflowToOriginalNeighbours` : Massflow to neighboursForOriginal.

return `zonal.Zone` :
The new zone.

`public boolean removeNeighbour(Zone neighbour)`

Removes a neighbour from this zone, but this zone can still be a neighbour to the specified zone after calling this function.

Parameters

- `zonal.Zone neighbour` : The neighbour to remove.

return `boolean` :
false if the zone was not a neighbour.

public boolean removeNeighbours(List neighbours)
Removes neighbours from this zone. This zone is also removed as neighbour from all zones in the list.

Parameters

`java.util.List neighbours` : The neighbours to remove.

return `boolean` :
false if any zone was not a neighbour.

public double getdQ(VectorType state, double time)
Returns the heat transfer from this zone.

Parameters

`Janet.VectorType state` : A Janet.VectorType holding the current state of the whole system.

`double time` : The current simulated time.

return `double` :
The heat transfer from this zone.

public Cylinder getCylinder()
Returns the cylinder this zone is in.

return `zonal.Cylinder` :
The cylinder this zone is in.

public double getDHdp(VectorType state)
Returns the partial derivative of the enthalpy with respect to pressure.

Parameters

`Janet.VectorType state` : A Janet.VectorType holding the current state of the whole system.

return `double` :
The partial derivative of the enthalpy with respect to pressure.

public double getDRdT(VectorType state)
Returns the partial derivative of R with respect to temperature.

Parameters

`Janet.VectorType state` : A Janet.VectorType holding the current state of the whole system.

return `double` :
The partial derivative of R with respect to temperature.

`public double getDRdp(VectorType state)`
Returns the partial derivative of R with respect to pressure.

Parameters

`Janet.VectorType state` : A Janet.VectorType holding the current state of the whole system.

return `double` :
The partial derivative of R with respect to pressure.

`public double getR(VectorType state)`
Returns R, the mass-specific gas constant.

Parameters

`Janet.VectorType state` : A Janet.VectorType holding the current state of the whole system.

return `double` :
R, the mass-specific gas constant.

`public double getM(VectorType state)`
Returns the molecular mass of this zone.

Parameters

`Janet.VectorType state` : A Janet.VectorType holding the current state of the whole system.

return `double` :
The molecular mass of this zone.

`public double getcp(VectorType state)`
Returns the mass-specific heat capacity under constant pressure.

Parameters

`Janet.VectorType state` : A Janet.VectorType holding the current state of the whole system.

return `double` :
The mass-specific heat capacity under constant pressure.

`public double getcv(VectorType state)`
Returns the mass-specific heat capacity with constant volume.

Parameters

`Janet.VectorType state` : A Janet.VectorType holding the current state of the whole system.

return `double` :
The mass-specific heat capacity with constant volume.

`public double geth(VectorType state)`
Returns the enthalphy of this zone.

Parameters

`Janet.VectorType state` : A Janet.VectorType holding the current state of the whole system.

return `double` :
The enthalphy of this zone.

`public double getu(VectorType state)`
Returns the internal energy of this zone.

Parameters

`Janet.VectorType state` : A Janet.VectorType holding the current state of the whole system.

return `double` :
The internal energy of this zone.

`public int getId()`
Returns the id of this zone.

return `int` :
An integer identifying this zone.

`public void setId(int id)`
Sets the id of this zone.

Parameters

`int id` : The new id, an integer greater than 0.

`public double getMass()`
Returns the mass of this zone.

return `double` :
The mass of this zone.

`public double getPhiBurned()`
Returns fuel/air ratio of the burned fuel.

return `double` :
Fuel/air ratio of the burned fuel.

`public void setPhiBurned(double phiBurned)`
Sets fuel/air ratio of the burned fuel.

Parameters

`double phiBurned` : The new value.

`public double getPhiUnburned()`
Returns fuel/air ratio of the unburned fuel.

return `double` :
Fuel/air ratio of the unburned fuel.

`public void setPhiUnburned(double phiUnburned)`
Sets fuel/air ratio of the unburned fuel.

Parameters

`double phiUnburned` : The new value.

`public double getResidualFraction()`
Returns the residual gas mass fraction.

return `double` :
The residual gas mass fraction.

`public void setResidualFraction(double resFraction)`
Sets the residual gas mass fraction.

Parameters

`double resFraction` : The new value.

Appendix B

Contents of Parameter Files

Below are the parameters used by the implementation presented in chapter 4. If a difference exists in any simulation presented in chapter 5, it is noted there.

B.1 Geometry Parameters

```
// crank radius
a
double
4.5e-2
// connecting rod
l
double
14.7e-2
// bore
B
expression
2*a
; compression ratio
r_c
double
10.1
. Displacement volume
V_d
expression
3.14159265358979*B^2/4*2*a
. Clearance volume
```

```
V_c
expression
V_d/(r_c-1)
. stroke
S
expression
2*a
. String identifying this engine
name
string
SAAB 2.3l naturally aspirated
```

B.2 Simulation Parameters

```
. The temperature of the cylinder wall
Twall
double
470
. Pi
pi
double
3.14159265358979
.
. Some heat-transfer stuff
WoschniC1
double
1
WoschniC2
expression
1/2.28
.
. The angular velocity
w
double
200
.
. Start of simulation, Intake Valve Close
tStart
expression
(540+55)/180*pi/w
.
. End of simulation, Exhaust Valve Open
tStop
expression
```

```
(4*pi+(180-57)/180*pi)/w
.
. ThetaS for the Vibe function
thetaS
expression
4*pi-20/180*pi
.
. ThetaE for the Vibe function
thetaE
expression
4*pi+40/180*pi
.
. The eta parameter for the Vibe function
eta
double
0.99
.
. The m parameter for the Vibe function
m
double
2
.
. The a parameter for the Vibe function
a
double
6.9
.
. Pressure at intake valve close
pivc
double
50000
.
. Temperature at intake valve close
Tivc
double
350
.
. fuel/air ratio
phi
double
1
. fuel/air ratio of fuel remaining from last cycle
phi_res
double
```

```
1
. Residual gass mass fraction
x_res
double
0.1
kappa
double
1.4
Lst
double
14.7
. The timestep of the simulation
h
double
2e-5
. After each integration it is checked if the timestep has exceeded
. maxStep. If it has, it is set to maxStep.
maxStep
double
8e-5
: Initial volume of the first burned zone.
initBurnedVolFrac
double
1e-4
deltaT_ad
double
1900
. The volume a zone is allowed to reach before dividing. This value
. does not cause any divisions since it is greater than the total
. cylinder volume.
volumeLimit
double
2e-3
```