LUND UNIVERSITY

TuFT: Tunnel Fire Tools - Teknisk dokumentation

Fridolf, Karl; Frantzich, Håkan

2014

Link to publication

Citation for published version (APA):
Fridolf, K., & Frantzich, H. (2014). *TuFT: Tunnel Fire Tools - Teknisk dokumentation*. Lund University.

Total number of authors:
2

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

Download date: 20. Dec. 2025

TuFT: Tunnel Fire Tools

Teknisk dokumentation

Karl Fridolf och Håkan Frantzich

Teknisk rapport
Rapportnummer: 3184

Organisation

Avdelningen för brandteknik, LTH, Lunds universitet

Lund, Sverige

Författare

Karl Fridolf och Håkan Frantzich

Dokumenttyp

TuFT: Tunnel Fire Tools - Teknisk dokumentation

Publiceringsdatum

2014-11-24

Finansiär

MSB (Myndigheten för samhällsskydd och beredskap)

Titel

TuFT: Tunnel Fire Tools - Teknisk dokumentation

Sammanfattning

Inom ramen för det svenska forskningsprojektet *Taktik och metodik vid brand i undermarksanläggningar* har ett enkelt planeringsverktyg utvecklats som bl a erbjuder stöd till beslutsfattare involverade i planeringen av räddningsinsatser i väg- och järnvägstunnlar. Verktyget, som har utvecklats vid Lunds universitet, heter TuFT och är en förkortning av Tunnel Fire Tools. TuFT kan beskrivas som ett mycket enkelt datorprogram, utvecklats i Java, som kan göra beräkningar av de branddynamiska konsekvenser som en brand i en tunnel kan orsaka. Informationen om de branddynamiska konsekvenserna kan dessutom användas för att bedöma påverkan dels på utrymmande människor och dels på den räddningsinsats som ev företas i tunneln. I denna rapport, som är skriven som en teknisk dokumentation, presenteras de beräkningstekniska förutsättningarna för TuFT. Det innebär att en redogörelse av de principer och ekvationer som ligger till grund för en simulering ges. Dessutom ingår som bilagor även en användarmanual, ett antal övningsuppgifter som syftar till att öka användbarheten samt den programmeringskod som utgör programmet.

Rapportens syfte är att tydliggöra de begränsningar som TuFT medför genom en beskrivning och presentation av de beräkningstekniska förutsättningar som ligger till grund för modellen. Syftet är vidare att med tydliga instruktioner förklara hur modellen kan och bör användas för att fungera. Det övergripande målet är att TuFT, tillsammans med rapporten, ska kunna användas av såväl ingenjörer aktiva inom byggnads- och brandskyddsteknisk projektering av undermarksanläggningar för väg- och järnvägstrafik samt av anställda inom landet kommunala räddningstjänster. Rapporten är skriven för version 14.10.22.1 av TuFT.

Summary

Within the Swedish research project *Rescue tactics and methodology during fires in underground facilities,* a simple decision support tool has been developed. The tool can, for example, aid decision makers involved in the early phase of road and rail tunnel fire risk assessments. The tool has been developed at Lund University and has been termed TuFT, which is an abbreviation for Tunnel Fire Tools. TuFT can be described as a very simple computer program, developed in Java, which has the capability to perform calculations of tunnel fire dynamic properties for a specified tunnel fire. In addition, this information can be used to perform assessments of evacuation possibilities as well as rescue operation possibilities for the same tunnel fire scenario. In this report, which has been written in the form of a technical documentation, the basic calculation principles of TuFT are presented. This means that the underlying models and equations building up TuFT are presented and described. In addition, a user manual, a number of simulation examples and the programming code are included in appendices to the report.

The purpose of this report is to highlight the limitations of TuFT by going through the technical principles of the calculations done in TuFT. The purpose is also to explain how the model can and should be used in order to perform well. The goal is that TuFT, together with this report, should be able to use by both fire protection engineers involved with fire safety design of road and rail tunnels as well as decision makers employed within the fire rescue services. The report is written for TuFT version 14.10.22.1.

Nyckelord

Tunnel; branddynamik; utrymning; räddningsinsats; modellering; beräkning

Klassifikation

Öppen teknisk rapport

Övrigt

ISRN: LUTVDG/TVBB--3184--SE, Report 3184.

Illustrationer framställda av Karl Fridolf och Håkan Frantzich om inte annat anges.

Språk

Svenska

ISSN

-

ISSN

1402-3504

Mottagares anteckningar

-

Antal sidor

44

Pris

-

Signatur _____Datum _

TuFT: Tunnel Fire Tools

Teknisk dokumentation

Karl Fridolf och Håkan Frantzich

LUND
UNIVERSITY

Copyright © Karl Fridolf och Håkan Frantzich

Lunds tekniska högskola, Institutionen för bygg- och miljöteknologi, Avdelningen för brandteknik

ISSN **1402-3504**

Lund 2014

Innehållsförteckning

1 Inledning 1

1.1 Bakgrund 1

1.2 Syfte och mål 2

1.3 Avgränsningar och begränsningar 2

1.3.1 Rapporten 2

1.3.2 TuFT 2

1.4 Ansvar 3

1.5 Kopiering 3

2 Allmänt om TuFT 5

2.1 Beskrivning av TuFT 5

3 Indata 7

3.1 Simuleringens omfattning 7

3.2 Tunnel 7

3.3 Brand 8

3.4 Tunnelbrandscenario 10

3.5 Branddynamik 10

3.6 Utrymning 11

3.6.1 Individ 11

3.6.2 Grupp 12

3.6.3 Tåg med individer 12

3.7 Räddningsinsats 13

4 Ekvationer för beräkning av branddynamiska effekter 15

4.1 Nomenklatur 15

4.2 Brandgastemperatur 16

4.3 Gaskoncentrationer 17

4.4 Siktlängd 17

4.5 Värmestrålning från brandkällan 18

4.6 Värmeöverföring från omgivande brandgaser 18

4.7 Backlayeringsträcka 18

5 Beräkningstekniska principer för de tre simuleringslägena 21

5.1 Nomenklatur 21

5.2 Branddynamik 22

5.2.1 Utdata 22

5.2.2 Exempel på utdata 23

5.3 Utrymning 28

5.3.1 Metod för bedömning av värme- och toxisk påverkan 30

5.3.2 Utdata 31

5.3.3 Exempel på utdata 33
5.4 Räddningsinsats 37
5.4.1 Utdata 39
5.4.2 Exempel på utdata 40
6 Referenser 43
Bilaga A: Användarmanual i
A.1 Användning i
A.2 Grundförutsättningar ii
A.3 Programmets indata-fil ii
A.3.1 Allmänna förutsättningar ii
A.3.2 Tunnelns utformning iii
A.3.3 Brandförloppet iii
A.3.4 Brandplatsen iv
A.3.5 Personer som utrymmer tunneln iv
A.3.6 Räddningsinsats vi
A.3.7 Aktuell brandmiljö vii
A.4 Analys av resultat viii
A.5 Felhantering viii
A.6 Vanliga frågor ix
A.6.1 Utrymning ix
A.6.2 Brandförloppet ix
A.6.3 Räddningsinsatsen ix
A.7 Exempelfil 1: Utrymning och räddningsinsats i vägtunnel x
A.8 Exempelfil 2: Utrymning och räddningsinsats i järnvägstunnel xi
Bilaga B: Övningsuppgifter xiii
B.1 Uppgift 1A: Enkel tunnelutrymning och rökdykningsinsats i vägtunnel xiii
B.1.1 Utrymning xiii
B.1.2 Räddningsinsats xiii
B.1.3 Lösningsförslag xiii
B.2 Uppgift 1B: Enkel tunnelutrymning och rökdykningsinsats i vägtunnel xv
B.2.1 Räddningsinsats xv
B.2.2 Lösningsförslag xv
B.3 Uppgift 2: Utrymning och räddningsinsats i en tågtunnel xv
B.3.1 Lösningsförslag xv
B.4 Uppgift 3: Brandförsök i Tistbrottets dolomitgruva xvii
B.4.1 Lösningsförslag xvii
Bilaga C: Programmeringskod xix
C.1 Tunnel.java xix
C.2 Fire.java xx
C.3 FireEvent.java xxiii
C.4 Measures.java xxxi
C.5 Person.java xxxvi
C.6 Group.java xliii
C.7 Train.java xlv
C.8 EvacSim.java xlvii

C.8 FireFighterPair.java lxi
C.9 RescueOperation.java lxiii
C.10 RescueSim.java lxix
C.11 Sim.java lxxii
C.12 App.java lxxvii

1 Inledning

Inom ramen för det svenska forskningsprojektet *Taktik och metodik vid brand i undermarksanläggningar*[1] har ett enkelt planeringsverktyg utvecklats för att erbjuda stöd bl a vid planeringen av räddningsinsatser i väg- och järnvägstunnlar. Planeringsverktyget har utformats som en beräkningsmodell, och fokus för modellen har varit utvecklingen av beräkningsmetodiken och inte presentationsformen. Den färdiga modellen kan användas för att simulera de branddynamiska konsekvenserna av en brand i en tunnel samt den påverkan som branden har dels på utrymmande människor och dels på rökdykare som ska göra en räddningsinsats i tunneln. Avsikten med denna rapport är att beskriva de beräkningstekniska principer som modellen bygger på. Dessutom innehåller rapporten en manual riktad till användare av modellen, se Bilaga A: Användarmanual. Till manualen redovisas även ett antal övningsuppgifter som syftar till att öka användarvänligheten av TuFT, se Bilaga B: Övningsuppgifter. Slutligen redovisas den fullständiga programmeringskoden i Bilaga C: Programmeringskod.

1.1 Bakgrund

Förekomsten av undermarksanläggningar för väg- och järnvägstrafik har under de senaste decennierna ökat, både i Sverige och utomlands. Dessutom går trenden mot att bygga längre tunnlar än tidigare; några exempel är Hallandsåstunneln som med sina två tunnelrör på vardera 8,7 kilometer blir den längsta järnvägstunneln i Sverige när den tas i drift, och Kanaltunneln mellan Frankrike och Storbritannien som med sina cirka 50 kilometer är den näst längsta järnvägstunneln i världen. I dagsläget finns det inget som tyder på att utvecklingen av byggandet under mark är på väg att avta.

Det finns flera skäl till denna utveckling. Ökade krav från samhället vad gäller trafiksäkerhet i tätbebyggda områden är ett. En annan anledning är att hänsyn lättare kan tas till buller och miljö när trafiken leds ner under marken; miljöpåverkan minskar och det störande bullret elimineras. Genom att anlägga väg- och järnvägsanläggningar under mark minskas naturligtvis också intrång i stads- och naturmiljö, på det befintliga kulturarvet och i andras annars berörda fastigheter. Undermarksanläggningar för väg- och järnvägstrafik byggs också inte minst för att öka den befintliga kapaciteten över vissa landområden och för att minska körsträckor där de geografiska förutsättningarna är besvärliga. Ett exempel är Hallandsåsen där branta stigningar och tvära kurvor gör att tåg idag varken kan köras i full hastighet eller med maximal godsvikt.

Den ökade exploateringen under mark ställer nya krav, inte minst vad gäller brandsäkerheten i dessa byggnadsverk. Även om det finns vissa likheter med traditionella byggnader ovan mark så finns det också uppenbara skillnader, bl a på grund av den geografiska placeringen. Dessa skillnader kommer att påverka inte bara brandens utveckling och brandgasernas spridning, utan också möjligheterna till en säker utrymning och en lyckad räddningsinsats, något som inte minst konstaterats av tidigare inträffade olyckor i den här typen av byggnadsverk (se t ex sammanställningen av Carvel and Marlair (2011)). För att nämna några exempel kan sägas att brandlasten ofta är mycket hög i väg- och järnvägstunnlar och möjligheterna att ventilera ut producerade brandgaser är begränsade. Till detta kommer att

[1] Ett forskningsprojekt lett av SP Sveriges Tekniska Forskningsinstitut i samarbete mellan SP, Mälardalens högskola, Lunds tekniska högskola, Storstockholms brandförsvar och Södra Älvsborgs räddningstjänstförbund. Det inleddes den första januari 2012, och avslutades den sista december 2014. Projektet finansierades av Myndigheten för samhällsskydd och beredskap (MSB), och det övergripande målet var att utveckla taktik och metodval för att underlätta räddningsinsatser i undermarksanläggningar.

utrymningen av människor kompliceras av det faktum att antalet utrymningsvägar ofta är begränsade, och att avståndet till närmaste utrymningsväg kan vara väldigt långt (ibland > 500 m). Av samma skäl kan räddningsinsatser i undermarksanläggningar för väg- och järnvägstrafik kraftigt försvåras, och avsaknaden av en översikt på brandplatsen bidrar ytterligare till att försvåra räddningsinsatser i de här byggnadsverken.

Sammantaget innebär ovanstående att projekterande ingenjörer i större utsträckning än tidigare kommer att behöva verifiera olika typer av brandskyddstekniska lösningar som en del av dimensioneringen av undermarksanläggningar. Det innebär också att räddningstjänsten i större utsträckning än tidigare kommer att behöva planera, träna och förbereda sig för att genomföra räddningsinsatser i undermarksanläggningar för väg- och järnvägstrafik. Behovet av ett enkelt planeringsverktyg för att underlätta i både den brandskyddstekniska projekteringen och insatsplaneringen av byggnadsverk under mark framstår därför som stort. Ett enkelt planeringsverktyg skulle kunna fungera som ett komplement till t ex de avancerade CFD-modeller som idag existerar och som används av ingenjörer, men även som ett komplement till de tränings- och planeringsaktiviteter som räddningstjänsten bedriver inom ramen för sitt uppdrag.

Med anledning av detta har ett enkelt planeringsverktyg i form av en beräkningsmodell namngiven TuFT (Tunnel Fire Tools) utvecklats. TuFT är ett datorprogram, utvecklat i Java, som kan göra beräkningar av de branddynamiska konsekvenser som en brand i en tunnel orsakar. Modellen kan också användas för att bedöma den påverkan som branden har dels på utrymmande människor och dels på den räddningsinsats som ev företas i tunneln. I denna rapport presenteras de beräkningstekniska förutsättningarna för modellen, d v s de principer och ekvationer som ligger till grund för en simulering. Presentationen görs i en s k teknisk dokumentation av modellen. Rapporten innehåller också en manual till modellen, d v s en beskrivning av hur indata ska anges för att modellen ska fungera (se Bilaga A: Användarmanual). Manualen har dessutom kompletterats med ett antal övningsuppgifter (se Bilaga B: Övningsuppgifter).

1.2 Syfte och mål

Syftet med den här rapporten är att ge en beskrivning av de beräkningstekniska förutsättningarna för TuFT, och därmed tydliggöra de begränsningar som dessa medför. Beskrivningen ges i en s k teknisk dokumentation som utgör rapportens huvuddel. Syftet med rapporten är också att med tydliga instruktioner förklara hur modellen kan och bör användas för att fungera. Detta görs i en användarmanual med tillhörande övningsuppgifter som bilagts rapporten (se Bilaga A: Användarmanual och Bilaga B: Övningsuppgifter). Det övergripande målet med rapporten är att modellen (tillsammans med rapporten) ska kunna användas av såväl ingenjörer aktiva inom byggnads- och brandteknisk projektering av undermarksanläggningar för väg- och järnvägstrafik samt av anställda inom landets kommunala räddningstjänster.

1.3 Avgränsningar och begränsningar

1.3.1 Rapporten

Rapporten bör i det närmaste betraktas som en dokumentation och en manual till TuFT. Presentationen av bakgrundsinformation, litteraturöversikter och relaterad forskning hålls därför mycket kort där den ingår.

1.3.2 TuFT

Utgångspunkten för TuFT är förenklade former av mass- och energiekvationer för ett endimensionellt flöde av brandgaser i en tunnel. Det innebär att samtliga beräkningar i TuFT begränsas till en dimension, vilket medför att t

ex temperaturer, koncentrationer och siktlängder alltid beräknas som ett medelvärde i tunneltvärsnittet. Noggrannheten i den utdata som genereras av TuFT är därmed lägre än de resultat som genereras i mer avancerade datormodeller, t ex CFD-modeller, och det är nödvändigt att användaren av TuFT har detta i åtanke när simuleringar utförs. I Fridolf och Wahlqvist (2014) görs ett försök till kvantifiering av TuFT:s prediktiva förmåga i förhållande till dels ett genomfört eldningsförsök i fullskala och dels FDS-simuleringar (Fire Dynamics Simulator) för motsvarande fall.

En annan begränsning med TuFT är att fokus vid programmeringen av beräkningsmodellen framför allt har legat på genomförandet av beräkningarna och inte själva presentationsformen för modellen. Det innebär bl a att all hantering av in- och utdata sker i textform. TuFT förutsätter vidare en s k ”perfekt” användare, d v s en användare som inte begår fel, t ex när indata till en simulering specificeras. Som ett exempel kan nämnas att TuFT utgår från punktkommatering (.) och inte komma-kommatering (,). Görs misstaget att använda fel typ av kommatering, eller liknande typ av fel, kommer modellen inte att fungera, alternativt leverera direkt felaktiga och missvisande resultat.

Slutligen ska sägas att TuFT delvis utvecklats inom ramen för utbildningen av en forskarstudent vid Avdelningen för brandteknik, LTH. Kvalitetsgranskning av beräkningsmodellen har genomförts av en senior forskare vid Avdelningen, men denne har saknat djuplodande programmeringstekniska kunskaper. Den genomförda kvalitetsgranskningen kan inte ersättas av den kvalitetsgranskning av resultatet som är nödvändig i andra sammanhang. TuFT bör därför användas med viss försiktighet och den som åberopar de resultat som beräkningsmodellen genererar bär själv ansvaret.

1.4 Ansvar

Den som använder TuFT bär själv ansvaret och varken Avdelningen för brandteknik vid Lunds universitet eller MSB tar något ansvar för de resultat som modellen genererar. För att möjliggöra identifiering av ev programmeringstekniska fel som kan ha uppstått i utvecklingen av TuFT samt för att underlätta möjligheten till egen kvalitetsgranskning redovisas som ett komplement till denna tekniska dokumentation även samtliga programmerade klasser som ingår i TuFT (d v s programmeringskoden), se Bilaga C: Programmeringskod.

1.5 Kopiering

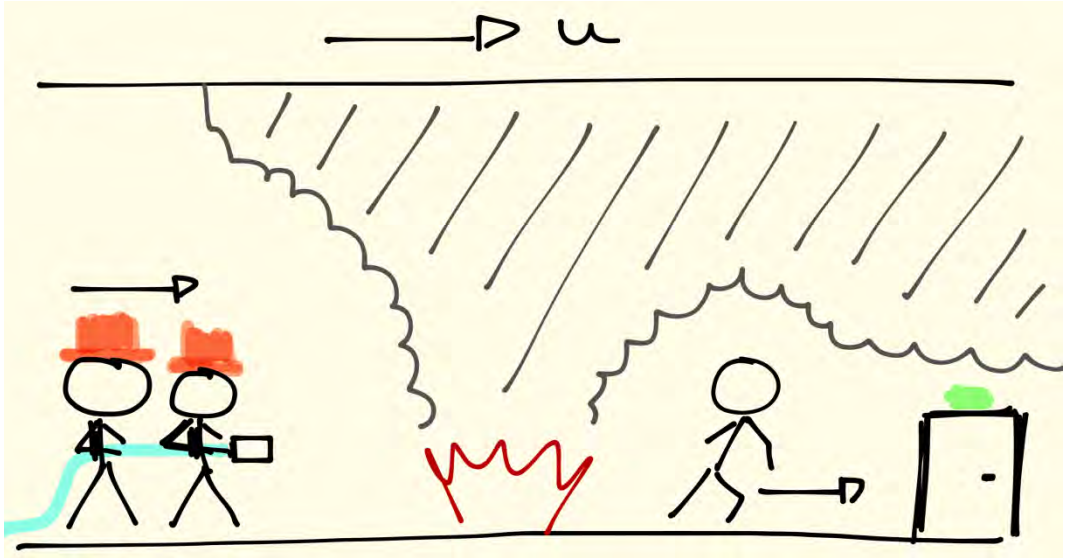
Användare av TuFT är fria att nyttja, modifiera och sprida det utan begränsningar.

2 Allmänt om TuFT

Detta kapitel inleds med en kort och allmän beskrivning av TuFT. I efterföljande kapitel presenteras typen av indata som krävs för simulering (kap 3), de ekvationer som ligger till grund för de branddynamiska beräkningarna (kap 4), de beräkningstekniska principerna för de tre simuleringslägena tillsammans med exempel på typen av utdata som genereras vid en simulering (kap 5). För praktiska råd och tips om hur t ex indata ska anges hänvisas till användarmanualen (Bilaga A: Användarmanual).

2.1 Beskrivning av TuFT

TuFT är en textbaserad beräkningsmodell för simulering av branddynamiska effekter, utrymnings- och insatsmöjligheter i väg- och järnvägstunnlar. Modellen har utvecklats i Java, och hanteringen av såväl in- som utdata är textbaserad. Beräkningen av olika branddynamiska egenskaper sker i tidssteg om 1 sekund, och involverar bl a temperatur-, koncentrations- och siktberäkningar. Informationen kan sedan utnyttjas i beräkningar av utrymnings- och insatsmöjligheterna, se illustration i Figur 1. De ingående ekvationerna för beräkningen av de olika branddynamiska effekterna baseras på den forskning som bl a finns sammanställd i Ingason (2012). Utgångspunkten för simuleringarna är alltså handberäkningsekvationer som empiriskt härletts med hjälp av brandförsök. De baseras på en i grunden förenklad form av energiekvationen för ett endimensionellt flöde av brandgaser i en tunnel. Det innebär att samtliga beräkningar i TuFT begränsas till en dimension, vilket innebär att t ex temperaturer, koncentrationer och siktlängder alltid beräknas som ett medelvärde i tunneltvärsnittet.

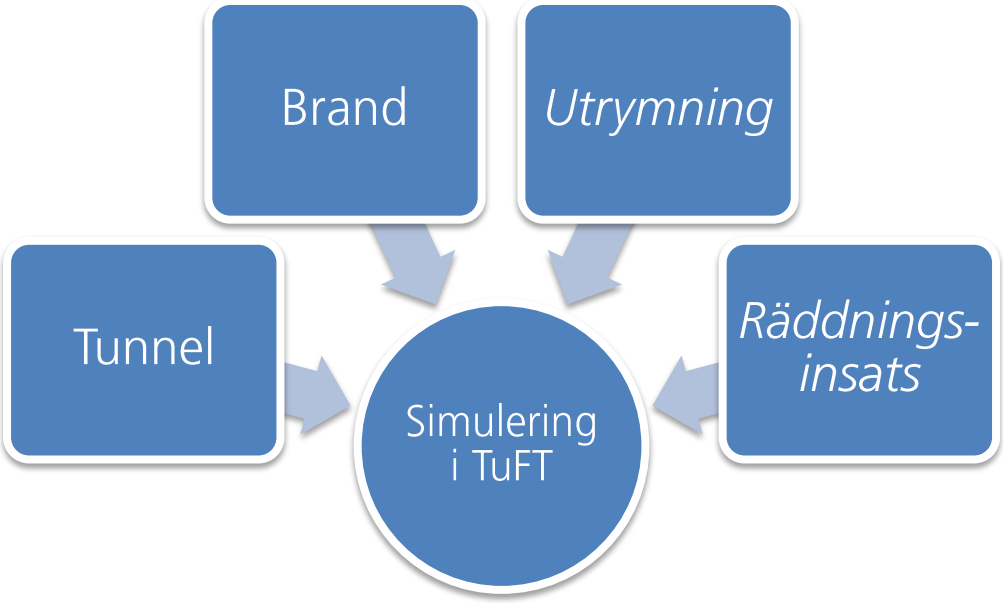


Figur 1. En schematisk illustration av de beräkningar som TuFT kan genomföra. Baserat på beräkningar av branddynamiska effekter kan TuFT räkna på möjligheterna för en säker utrymning (till höger i bilden) och en lyckad räddningsinsats (till vänster i bilden) för ett givet tunnelbrandscenario. Beräkningar görs i tidssteg om 1 s och brandens påverkan på utrymmande personer och rökdykare baseras på deras position *x* i tunneln, och tiden in i brandförloppet.

Inför en simulering med TuFT måste användaren av modellen bestämma sig för omfattningen av simuleringen. Om det t ex endast är intressant att studera de branddynamiska effekterna av en brand i en tunnel (t ex temperaturen på en viss position i tunneln under hela brandförloppet) räcker det med att specificera den tunnel och den brand som utgör grunden för scenariot, liksom positionerna där beräkningarna ska genomföras. Är användaren däremot också

intresserad av att veta vilka konsekvenser brandscenariot får för utrymnings- och insatsmöjligheterna i tunneln måste även ett utrymnings- och/eller insatsscenario specificeras, bl a genom att för utrymningsscenariot ange hur många personer som befinner sig i tunneln (och var) och för insatsscenariot ange hur många brandmän som finns tillgängliga och vilken utrustning de har att tillgå, se Figur 2. Simuleringens omfattning kan alltså kategoriseras enligt:

1. Branddynamik
2. Branddynamik och påverkan på utrymmande människor
3. Branddynamik och påverkan på räddningstjänstens insatsförmåga
4. Branddynamik och påverkan både på utrymmande människor och räddningstjänstens insatsförmåga



Figur 2. Indata kopplat till tunneln och branden är utgångspunkten för en simulering i TuFT. Information om utrymningsscenariot och/eller räddningsinsatsen är endast nödvändigt om avsikten är att göra en simulering av utrymnings- och/eller insatsmöjligheterna för det givna tunnelbrandscenariot.

3 Indata

Indata till TuFT beskrivs i textform och läses sedan in av programmet när simuleringen påbörjas. I detta avsnitt redogörs för vilken typ av indata som krävs för simulering och de ev inledande beräkningar som utförs av TuFT. Observera att beskrivningen är teknisk och sannolikt inte räcker för alla användare för att kunna starta en simulering. Därför bör den läsas tillsammans med den manual som tagits fram, se nedan.

3.1 Simuleringens omfattning

Simuleringens omfattning måste definieras innan några beräkningar kan genomföras. Möjligheterna finns beskrivna i Tabell 1.

Tabell 1. Sammanställning av de parametrar som användaren måste ange för att definiera simuleringens omfattning.

Parameter	Enhet	Datatyp
Branddynamik	true/false	Boolesk
Utrymning	true/false	Boolesk
Räddningsinsats	true/false	Boolesk

Simuleringens omfattning definieras med andra ord med s k ”sant/falskt”-villkor (booleska variabler), där *true* betyder att typen av simulering ska genomföras och *false* att den inte ska genomföras. Beroende på simuleringens omfattning behöver användaren även definiera ett antal andra parametrar kopplat till tunneln, branden, utrymningen och räddningsinsatsenen, se nedan.

3.2 Tunnel

Tillsammans med branden utgör information om tunneln den grundläggande information som behövs för att skapa ett tunnelbrandscenario. Användaren måste därför definiera ett antal parametrar, d v s egenskaper, kopplade till tunneln. Dessa finns beskrivna i Tabell 2.

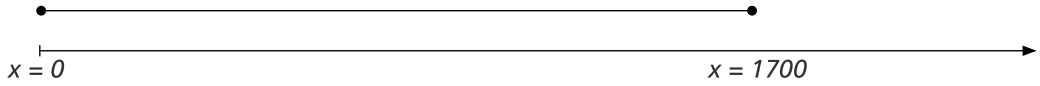
Tabell 2. Sammanställning av de parametrar som användaren måste ange för att definiera en tunnel.

Parameter	Enhet	Datatyp
Typ av tunnel	Väg- eller järnväg	Sträng
Längd	m	Reellt tal
Bredd	m	Reellt tal
Höjd	m	Reellt tal
Vindhastighet	m/s	Reellt tal
Omgivningstemperatur	°C	Reellt tal
Avstånd mellan utrymningsvägar[2]	m	Reellt tal

[2] I det fall det inte ska finnas några utrymningsvägar tillgängliga i tunneln anges värdet ”0.0”.

Baserat på den information som användaren anger beräknar TuFT bl a tunnelns tvärsnittsarea och omkrets, egenskaper som behövs i de branddynamiska beräkningar som senare ska genomföras.

Eftersom att TuFT bygger på enkla handberäkningsekvationer i en dimension innebär det att tunneln i teorin kan betraktas som en linje mellan två punkter där respektive punkt utgör tunnelns portaler. Den ena portalen ligger alltid i nollpunkten på axeln ($x = 0$), och den andra alltid på en position motsvarande tunnelns totala längd, se Figur 3.



Figur 3. Schematisk illustration av hur en tunnel i TuFT representeras. I illustrationen används en tunnel med längden 1700 m.

Utrymningsvägarnas position i tunneln bestäms av det av användaren definierade avståndet mellan dem. Den första utrymningsvägen placeras alltid på det angivna avståndet mätt från tunnelportalen vid positionen $x = 0$ m, och därefter med motsvarande avstånd till dess att den andra tunnelportalen nåtts. Om avståndet mellan utrymningsvägarna hade angetts till 500 m i ovan exempel i Figur 3 hade det inneburit att tre utrymningsvägar placerats ut; en vid $x = 500$ m, en vid $x = 1000$ m samt en vid $x = 1500$ m. I sammanhanget bör det noteras att utrymningsvägarna i tunneln (i TuFT) är att betrakta som positioner för en säker plats, snarare än en fysisk utrymningsväg. Praktiskt innebär det att de kan utgöra startpunkten för en räddningsinsats (om denna företas genom en utrymningsväg) eller slutpunkten för en utrymmande individ (om denne utrymmer via en utrymningsväg).

3.3 Brand

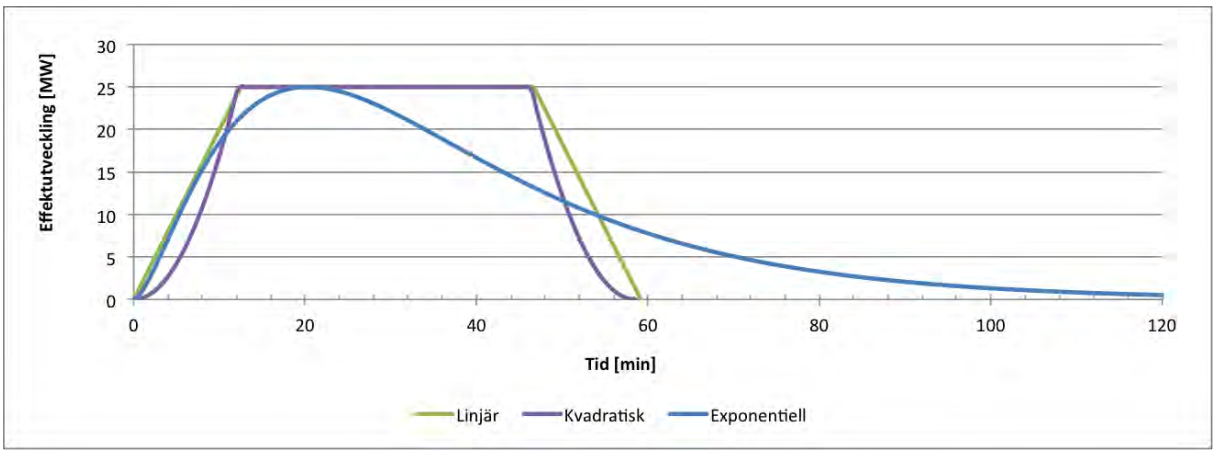
Tillsammans med tunneln utgör information om branden den grundläggande information som behövs för att skapa ett tunnelbrandscenario. Användaren måste därför definiera ett antal parametrar kopplade till branden. Beroende på om branden beskrivs som linjärt eller kvadratiskt tillväxande, eller om den beskrivs som exponentiellt tillväxande enligt det koncept som presenterats av Ingason (2005, 2006, 2009), behöver användaren definiera lite olika variabler. Dessa finns beskrivna i Tabell 3 för linjärt och kvadratiskt tillväxande bränder samt i Tabell 4 för exponentiellt tillväxande bränder. En grafisk illustration av de olika brandförloppstyperna framgår av Figur 4.

Tabell 3. Sammanställning av de parametrar som användaren måste ange för att definiera en linjärt eller kvadratiskt tillväxande brand.

Parameter	Enhet	Datatyp
Typ av tillväxt	Linjär eller kvadratisk	Sträng
Tillväxthastighet	kW eller kW/s^2	Reellt tal
Avsvalningshastighet	kW eller kW/s^2	Reellt tal
Maximal effektutveckling	kW	Reellt tal
Varaktighet för maximal effektutveckling	min	Heltal
Rökpotential	m^2/kg	Reellt tal
Förbränningsvärme	kJ/kg	Reellt tal
Förbränningseffektivitet	-	Reellt tal
CO-produktion	kg/kg	Reellt tal
CO_2-produktion	kg/kg	Reellt tal
HCN-produktion	kg/kg	Reellt tal

Tabell 4. Sammanställning av de parametrar som användaren måste ange för att definiera en exponentiellt tillväxande brand.

Parameter	Enhet	Datatyp
Typ av tillväxt	Exponentiell	Sträng
Maximal effektutveckling	kW	Reellt tal
Energiinnehåll	GJ	Reellt tal
Tid till maximal effektutveckling, t_{max}	min	Reellt tal
Rökpotential	m^2/kg	Reellt tal
Förbränningsvärme	kJ/kg	Reellt tal
Förbränningseffektivitet	-	Reellt tal
CO-produktion	kg/kg	Reellt tal
CO_2-produktion	kg/kg	Reellt tal
HCN-produktion	kg/kg	Reellt tal



Figur 4. Exempel på de tre olika brandförloppstyper som kan specificeras i TuFT.

Baserat på den information som användaren anger beräknar TuFT bl a den effektutvecklingskurva som ska ligga till grund för simuleringen. Tekniskt sett innebär det att effektutvecklingen för varje sekund av brandförloppet lagras i en vektor med längden $n + 1$, där n motsvarar det totala antalet sekunder för hela brandförloppet.

För den linjärt och kvadratiskt tillväxande branden beror utseendet på effektutvecklingskurvan på typen av tillväxt och avsvalning, maximal effektutveckling och den tid som branden antas brinna med maximal effektutveckling. Avsvalning antas alltid ske enligt samma princip som tillväxten, d v s om branden tillväxer linjärt så kommer avsvalningen också att ske linjärt. Konstruktionen av effektutvecklingskurvan beskrivs i Tabell 5, där t_0 motsvarar $t = 0$ s, t_1 motsvarar tiden då effektutvecklingen nått sitt maximala värde, t_2 motsvarar tiden t_1 plus varaktigheten för maximal effektutveckling och slutligen t_3 motsvarar tiden för hela brandförloppet.

Tabell 5. Beskrivning av hur effektutvecklingskurvan beräknas för linjärt och kvadratiskt tillväxande bränder.

	Effektutveckling	
Villkor	**Linjär tillväxt**	**Kvadratisk tillväxt**
$t \leq t_1$	$\dot{Q}(t) = \alpha_{growth} \cdot t$	$\dot{Q}(t) = \alpha_{growth} \cdot t^2$
$t_1 < t \leq t_2$	$\dot{Q}(t) = \dot{Q}_{max}$	$\dot{Q}(t) = \dot{Q}_{max}$
$t_2 < t \leq t_3$	$\dot{Q}(t) = \dot{Q}_{max} - \alpha_{decay} \cdot (t - t_2)$	$\dot{Q}(t) = \alpha_{decay} \cdot (t_3 - t)^2$

För den exponentiellt tillväxande branden sker konstruktionen av effektutvecklingskurvan enligt en annan princip (Ingason, 2005, 2006, 2009). I ett första steg beräknas tre variabler; n, r och k, se Ekvation 1-3. Därefter konstrueras effektutvecklingskurvan med hjälp av Ekvation 4 (observera att det är endast variabeln t_{max} som anges i minuter, t i t ex Ekvation 4 anges i sekunder).

$$n = \frac{e^{\left((2{,}9\cdot\dot{Q}_{max})/(E_{tot}\cdot 1000)\right)\cdot t_{max}\cdot 60}}{1{,}346}$$ Ekvation 1

$$r = \left(1 - \frac{1}{n}\right)^{1-n}$$ Ekvation 2

$$k = \frac{\dot{Q}_{max}}{E_{tot}} \cdot r$$ Ekvation 3

$$\dot{Q}(t) = \dot{Q}_{max} \cdot n \cdot r \cdot (1 - e^{-k\cdot t})^{n-1} \cdot e^{-k\cdot t}$$ Ekvation 4

Produktionstermer för kolmonoxid, koldioxid samt vätecyanid anges för att möjliggöra beräkning av respektive ämnes gaskoncentration i tunnelns tvärsnitt nedströms branden. Värdena bör betraktas som en beskrivning av hur stor andel av ämnet som bildas per avbrunnen massenhet bränsle och uttrycks i TuFT i kg/kg (Karlsson & Quintiere, 2000, p. 260). Information om produktionen för respektive ämne kan hämtas i litteraturen (se t ex Karlsson and Quintiere (2000, pp. 231-232), Ingason (2012, p. 291) och Tewarson (2008, pp. 142-146)) och härstammar mer eller mindre uteslutande från genomförda eldningsförsök. Saknas information om ett visst ämnes produktionsterm för det aktuella brandscenariot matas värdet *0.0* in (konsekvensen är att beräkningar av gaskoncentrationen för det berörda ämnet inte genomförs i simuleringen). I sammanhanget bör nämnas att ett ämnes produktion under ett verkligt brandförlopp kommer att variera, inte bara som en funktion av tiden t s in i brandförloppet utan också som en funktion av om branden är bränsle- eller ventilationskontrollerad. I TuFT finns dock ingen möjlighet att ange värden på produktionstermerna som varierar över tiden.

3.4 Tunnelbrandscenario

Som beskrivits ovan utgör information om tunneln och branden basen för det tunnelbrandscenario som ska simuleras. Slutligen måste branden placeras i tunneln och vindriktningen anges. Det görs med de två parametrar som beskrivs i Tabell 6.

Tabell 6. Sammanställning av de parametrar som användaren måste ange för att skapa ett tunnelbrandscenario.

Parameter	Enhet	Datatyp
Position	m	Reellt tal
Vindriktning från nollpunkt	true/false	Boolesk

När vindriktningen anges görs detta genom ett ”sant/falskt”-villkor, där *true* betyder att vinden blåser <u>från</u> nollpunkten (tunnelportalen vid positionen $x = 0$ m) och *false* betyder att vinden blåser <u>mot</u> nollpunkten. Vindriktningen får bl a konsekvenser för hur brandgaserna transporteras i tunneln, se nedan.

3.5 Branddynamik

Om endast en branddynamiksimulering ska genomföras måste en eller flera positioner i tunneln anges vid vilka de branddynamiska beräkningarna ska göras. Detsamma gäller om resultatet av branddynamiksimuleringen ska sparas vid en utrymnings- och/eller räddningsinsatssimulering. Positionerna definieras i enlighet med innehållet i Tabell 7, och anges alltid <u>relativt brandens position</u>. Skälet till varför positionerna ska anges relativt branden är att beräkning av de branddynamiska effekterna endast kan ske nedströms branden.

Tabell 7. Sammanställning av den parameter som användaren måste ange för att definiera en mätpunkt i tunneln.

Parameter	Enhet	Datatyp
Position	m	Reellt tal

3.6 Utrymning

Information kopplad till utrymningsscenariot är nödvändig om en utrymningssimulering ska kunna göras i TuFT. Beroende på typen av tunnel, d v s om det är en väg- eller järnvägstunnel som avses, behöver användaren definiera lite olika variabler. I en vägtunnel specificeras antingen individer eller grupper av individer, men i en järnvägstunnel specificeras istället ett tåg som innehåller ett antal individer. Beräkningsprinciperna för utrymningen av tunneln är dock mer eller mindre oberoende av typen av tunnel.

3.6.1 Individ

I en vägtunnel måste användaren definiera en eller flera individer som ska utrymma tunneln. Alternativt kan användaren definiera en eller flera grupper av individer, t ex för att representera en grupp av människor som färdats i samma fordon (se nedan). Individens egenskaper som måste anges beskrivs i Tabell 8.

Tabell 8. Sammanställning av de parametrar som användaren måste ange för att definiera en individ.

Parameter	Enhet	Datatyp
Startposition	m	Reellt tal
Varseblivningstid	s	Heltal
Förberedelsetid	s	Heltal
Tid innan individ lämnar fordon	s	Heltal
Utrymning via nödutgång	true/false	Boolesk
Deterministisk beräkning av gånghastighet	true/false	Boolesk

Individens startposition anges alltid relativt tunnelportalen vid positionen $x = 0$ m (detta eftersom att utrymning kan simuleras både upp- och nedströms branden). Varseblivnings- respektive förberedelsetid uppskattas av användaren, liksom tiden innan individen bestämmer sig för att lämna sitt fordon. Under den tid som individen sitter kvar i sitt fordon antas denne opåverkade av ev brandgaser, d v s denne påverkas varken av brandgasernas temperatur eller toxiska innehåll. Inte heller påverkas individen av strålning från branden så länge denne befinner sig i sitt fordon.

Om individen ska utrymma till närmaste nödutgång anges detta i ett ”sant/falskt”-villkor med *true*; vid *false* sker utrymning till någon av tunnelportalerna. Utrymning sker i TuFT alltid bort från branden (d v s, en individ kan aldrig utrymma förbi branden, även om det skulle innebära en kortare utrymningsväg).

Individens gånghastighet kommer att påverkas av om individen utrymmer i rök eller inte, och kan i TuFT antingen beräknas deterministiskt eller probabilistiskt. Om *true* anges beräknas gånghastigheten deterministiskt enligt beskrivningen i Tabell 9; vid *false* probabilistiskt. Det senare innebär att TuFT bestämmer individens gånghastighet från en normalfördelning (för olika siktintervall) med ett medelvärde och en standardavvikelse enligt Tabell 9 i början av simuleringen. Sambandet mellan gånghastighet och siktnedsättning bygger på de resultat som presenterats av Fridolf, Andrée, Nilsson, and Frantzich (2013).

Tabell 9. Individers gånghastighet i TuFT varierar beroende på siktnedsättning och om beräkningen sker deterministiskt eller probabilistiskt.

	Gånghastighet	
Sikt [m]	**Deterministiskt**	**Probabilistiskt**
s > 5,00	1,30	N(1,30;0,102)
5,00 < s ≤ 2,00	1,00	N(1,02;0,112)
2,00 < s ≤ 1,39	0,80	N(1,00;0,262)
1,39 < s ≤ 1,11	0,70	N(0,83;0,182)
1,11 < s ≤ 0,83	0,55	N(0,78;0,272)
0,83 < s ≤ 0,55	0,30	N(0,42;0,172)
s < 0,55	0,20	N(0,20;0,102)

3.6.2 Grupp

I en vägtunnel kan användaren definiera en eller flera grupper av individer, t ex för att representera en grupp av människor som färdats i samma fordon. I förhållande till när en individ definieras ligger den stora skillnaden för gruppen att flera av variablerna anges som från/till-värden, se Tabell 10. Beräkningstekniskt används dessa från/till-värden för att skapa ett intervall, inom vilket TuFT slumpmässigt bestämmer ett värde för att representera variabeln. Principen för hur TuFT beräknar värdet ges i ett exempel nedan. Det innebär att de individer som ingår i gruppen kan få varierande startpositioner, varseblivnings- och förberedelsetider samt olika tider varefter de lämnar fordonet. Ju större intervall, desto större spridning. Önskas ingen spridning mellan individerna i gruppen anges samma från/till-värden.

Tabell 10. Sammanställning av de parametrar som användaren måste ange för att definiera en grupp av individer.

Parameter	**Enhet**	**Datatyp**
Personantal	st	Heltal
Startposition (från)	m	Reellt tal
Startposition (till)	m	Reellt tal
Varseblivningstid (från)	s	Heltal
Varseblivningstid (till)	s	Heltal
Förberedelsetid (från)	s	Heltal
Förberedelsetid (till)	s	Heltal
Tid innan grupp lämnar fordon (från)	s	Heltal
Tid innan grupp lämnar fordon (till)	s	Heltal
Utrymning via nödutgång	true/false	Boolesk
Deterministisk beräkning av gånghastighet	true/false	Boolesk

Beräkningstekniskt bestämmer TuFT värdet på startpositionen, varseblivnings- och förberedelsetiden samt tiden innan individen lämnar fordonet enligt följande princip (exempel utgår från beräkning av individens startposition):

startposition = startposition(från) * (1 - t) + startposition(till) * t

t anger i detta sammanhang ett slumpmässigt, reellt tal mellan 0,0-1,0 som dragits från en uniform sannolikhetsfördelning. För varje variabel slumpas ett nytt reellt tal. Det betyder att det för varje individ genereras fyra slumptal.

3.6.3 Tåg med individer

Till skillnad från i vägtunneln definieras i en järnvägstunnel inga individer eller grupper av individer. Istället definieras ett tåg med ett antal passagerare, se Tabell 11.

Tabell 11. Sammanställning av de parametrar som användaren måste ange för att definiera ett tåg med passagerare.

Variabel	Enhet	Datatyp
Startposition	m	Reellt tal
Längd	m	Reellt tal
Antal tillgängliga dörrar	st	Heltal
Dörrbredd	m	Reellt tal
Antal passagerare	s	Heltal
Varseblivningstid	s	Heltal
Förberedelsetid	s	Heltal
Utrymning via nödutgång	true/false	Boolesk
Deterministisk beräkning av gånghastighet	true/false	Boolesk

Startpositionen motsvarar den del av tåget som befinner sig närmast tunnelportalen vid positionen $x = 0$ m. Tillsammans med information om tågets längd beräknar TuFT positionerna för de tillgängliga dörrarna, d v s vägarna ut ur tåget vid en utrymning. Placeringen av dörrarna sker alltid med tågets mittpunkt som centrum, och fördelningen sker uniformt över tågets längd enligt beräkningsprincipen nedan, där d står för avståndet mellan dörrarna, n för antalet dörrar, $pos(i)$ för positionen av dörr i i tåget.

$$d = \frac{l}{n+1} \qquad \text{Ekvation 5}$$

$$pos(i) = d * i \qquad \text{Ekvation 6}$$

En skillnad i förhållande till när individer eller grupper av individer definieras för en vägtunnel är att ingen tid för att lämna fordonet anges för tåget. Istället beräknar TuFT den tid det tar innan respektive individ kan lämna det, baserat på flödesrestriktioner i tågdörrarna. Beräkningsprincipen beskrivs nedan och bygger på resultat från Fridolf, Nilsson, and Frantzich (2014):

1. Antalet passagerare fördelas jämnt på de tillgängliga dörrarna (*antal passagerare / antal tillgängliga dörrar*)

2. Tiden det tar för passagerare j vid utgång i att lämna tåget beräknas (j * (*1 / (0,2 * dörrbredd)*))

En annan skillnad är att inte heller någon startposition för individen anges. Istället baseras individens startposition på var individen utrymmer tåget, d v s startpositionen blir den position x m i tunneln där individen lämnar tåget.

3.7 Räddningsinsats

För att kunna göra en simulering av räddningstjänstens insatsförmåga i TuFT behöver användaren ange information kopplat till räddningsinsatsen. De parametrar som krävs anges i Tabell 12.

Tabell 12. Sammanställning av de parametrar som användaren måste ange för att definiera en räddningsinsats.

Parameter	Enhet	Datatyp
Framkörnings- och förberedelsetid	min	Heltal
Insats uppströms	true/false	Boolesk
Insats via tunnelportal	true/false	Boolesk
Antal brandmän	st	Heltal
IR-kamera	true/false	Boolesk
Stora luftpaket	true/false	Boolesk
Längd på slangtyp 1	m	Reellt tal
Längd på slangtyp 2	m	Reellt tal
Tid för kopplingstyp 1	s	Heltal
Tid för kopplingstyp 2	s	Heltal

Användaren har alltså möjlighet att ange om räddningsinsatsen i TuFT ska genomföras i rökfri eller rökfylld miljö genom ett ”sant/falskt”-villkor. Anges *true* genomförs insatsen uppströms branden, d v s i rökfri miljö, och anges *false* så genomförs den nedströms i rök. Användaren har också möjlighet att ange huruvida branden ska angripas från någon av tunnelportalerna eller från närmaste utrymningsväg enligt samma princip. Baserat på hur dessa två variabler definieras beräknar TuFT startpositionen för räddningsinsatsen. När insats görs via en utrymningsväg väljs alltid den utrymningsväg som ligger närmast branden. Det bör i sammanhanget också noteras att TuFT alltid utgår från att räddningsinsatsen bedrivs i rökdykarpar om två personer (för mer information se nedan). Det antal som avses i Tabell 12 är aktiva brandmän som tillsammans kan bilda rökdykarpar (exkl ev rökdykarledare och annan personal som behövs för räddningsinsatsen).

Viss utrustning kan också definieras i TuFT, närmare bestämt om det för räddningsinsatsen finns tillgång till IR-kameror eller inte och om det finns luftpaket av modell större eller mindre. Variablerna definieras i ”sant/falskt”-villkor enligt ovan princip. Tillgång till IR-kamera påverkar brandmännens gånghastighet i rök enligt beskrivningen i Tabell 13. Tillgång till stora luftpaket ökar aktionstiden, d v s den tid som en brandman kan delta i insatsen, från 25 till 40 min. Aktionstiden gäller för det första rökdykarpar som inleder insatsen i tunneln; för de efterföljande antas aktionstiderna uppgå till 90 % av de för det första rökdykarparet. Skälet är ett försök att beakta den ansträngning det innebär för efterföljande rökdykarpar att innan deras insats påbörjas röra sig till det föregående rökdykarparets position i tunneln.

Tabell 13. Brandmännens gånghastighet i TuFT varierar beroende på siktnedsättning och om de har tillgång till IR-kamera eller inte.

	Gånghastighet	
Sikt [m]	**Med IR**	**Utan IR**
$s \geq 4{,}0$	1,00	1,00
$4{,}0 < s < 1{,}5$	1,00	$1{,}55 - 1{,}1 * EC$[3]
$s \leq 1{,}5$	1,00	0,1

Två typer av slanglängder kan specificeras och de läggs ut växelvis, först slangtyp 1 därefter slangtyp 2 och sedan igen slangtyp 1, o s v. På det viset kan olika taktiska upplägg angående förberedda slangpaket analyseras. På samma sätt fungerar TuFT när det gäller tiderna för sammankoppling av slanglängderna. Första gången två slangar ska kopplas samman används tiden för kopplingstyp 1, andra gången används tiden för kopplingstyp 2. Tredje gången slangar ska kopplas ihop används åter tiden för kopplingstyp 1. Motivet till att ha två olika kopplingstyper kan vara att en längre tid kan behövas för de tillfällen då slangar även ska vattenfyllas och inte bara kopplas samman.

[3] EC står för extinction coefficient och beräknas av TuFT. EC är direkt kopplat till siktnedsättningen (Jin, 2008).

4 Ekvationer för beräkning av branddynamiska effekter

TuFT kan göra uppskattningar av brandgastemperatur, gaskoncentrationer av CO, CO_2, HCN och O_2 samt siktbarhet (i form av en dämpningskoefficient) nedströms branden för ett specificerat tunnelbrandscenario. Nedan presenteras de ingående ekvationerna för de respektive branddynamiska beräkningarna.

Som tidigare nämnts baseras de underliggande ekvationerna för beräkning av olika branddynamiska parametrar på den forskning som bl a finns sammanställd av Ingason (2012). Det betyder att utgångspunkten för simuleringar med TuFT är enklare handberäkningsekvationer som empiriskt härletts bl a i eldningsförsök. De baseras på i grunden förenklade former av mass- och energiekvationer för ett endimensionellt flöde av brandgaser i en tunnel. Brandgaserna antas alltid strömma i vindens riktning och vara fullständigt omblandade, och beräkningar av t ex temperatur, gaskoncentrationer och siktbarhet görs alltid som medelvärden i tunneltvärsnittet[4] baserat på positionen x m i tunneln i förhållande till branden samt tiden t s in i brandförloppet.

Fördelen med att TuFT utgår från dessa förhållandevis enkla handberäkningsekvationer är att beräkningarna går mycket snabbare att genomföra än med t ex CFD-modeller. Nackdelen är dock att noggrannheten är lägre. Som ett exempel kan nämnas att antagandet om fullständigt omblandade brandgaser medför att TuFT bortser från skiktningen av brandgaser, även direkt vid brandkällan. Så är naturligtvis inte fallet i verkligheten. Vid vindhastigheter motsvarande 1 m/s är det snarare sannolikt att brandgaserna kommer att stiga upp mot taket vid brandkällan (d v s skiktas), för att sedan bli mer homogena över tvärsnittet på ett avstånd motsvarande 10-20 gånger den hydrauliska diametern[5] (Ingason, 2012). Antagandet om en fullständig omblandning innebär dock att tiden till kritiska förhållanden kommer att överskattas i såväl utrymnings- som räddningsinsatssimuleringar, vilket innebär att resultaten i detta avseende är att betrakta som konservativa.

4.1 Nomenklatur

A	Tvärsnittsarea [m^2]
c_p	Värmekapacitet [kJ/kg K]
D_{mass}	Rökpotential [m^2/kg]
ε	Emissivitet [-]
g	Gravitationskonstanten [m/s^2]
H	Tunnelhöjd [m]
H_{ec}	Effektiv förbränningsentalpi [kJ/kg]
H_T	Förbränningsentalpi (netto) [kJ/kg]
h	Lumpad värmeförlustkoefficient för konvektion och strålning [kW/m^2 K]
h_c	Konvektiv värmeförlustkoefficient [kW/m^2 K]

[4] Även om det inte explicit framgår i texten nedan avses alltid värden medelvärden beräknade i tunnelns tvärsnitt.

[5] Den hydrauliska diametern kan beräknas med följande ekvation: $D_h = \frac{4A}{P}$.

L	Backlayeringsträcka [m]
M_a	Molvikt för luft [kg/kmol]
M_i	Molvikt för ämne *i* [kg/kmol]
M_{O_2}	Molvikt för syre [kg/kmol]
$\dot{m}_a$	Massflöde för luft [kg/s]
P	Omkrets [m]
$\dot{Q}$	Effektutveckling [kW]
$\dot{q}_{conv,smoke}$	Konvektiv värmeöverföring från brandgaser [kW/m^2]
$\dot{q}_{rad,fire}$	Strålande värmeöverföring från brandkällan [kW/m^2]
$\dot{q}_{rad,smoke}$	Strålande värmeöverföring från brandgaser [kW/m^2]
R_0	Avstånd från brandkällan [m]
ρ_a	Luftens densitet vid omgivningstemperaturen [kg/m^3]
σ	Stefan–Boltzmanns konstant (5.67 * 10^{-11}) [kW/m^2 K^4]
τ	Tid då de brandgaser vid position *x* m nedströms branden lämnade brandkällan [s]
t	Tid [s]
T_a	Omgivningstemperatur [°C]
T_{avg}	Genomsnittlig temperatur i tunnelns tvärsnitt [°C]
$T_{avg,x=0}$	Genomsnittlig temperatur i tunnelns tvärsnitt vid brandkällan [°C]
T_m	Yttemperatur (antas alltid vara lika med omgivningstemperaturen) [°C]
u	Vindhastighet [m/s]
V	Siktlängd [m]
$X_{i,avg}$	Genomsnittlig gaskoncentration av ämne *i* i tunnelns tvärsnitt [-]
X_{O_2}	Genomsnittlig gaskoncentration av syre i tunnelns tvärsnitt [-]
χ	Förbränningseffektivitet [-]
χ_r	Strålningsfraktion [-]
x	Position i tunneln, antingen absolut eller relativt brandkällan (nedströms) [m]
Y_i	Massproduktion av ämne *i* [kg/kg]

4.2 Brandgastemperatur

Den genomsnittliga brandgastemperaturen [°C] i tunnelns tvärsnitt på en position *x* m nedströms branden vid tidpunkten *t* s in i brandförloppet beräknas enligt Ekvation 7.

$$T_{avg}(x,t) = T_a + \left[T_{avg,x=0}(\tau) - T_a\right]e^{-(hPx/\dot{m}_a c_p)} \qquad \text{Ekvation 7}$$

Beräkningen i Ekvation 7 bygger på den genomsnittliga brandgastemperaturen [°C] i tunnelns tvärsnitt vid brandkällan vid tiden τ s. τ betyder i detta sammanhang en tidsförskjutning [s] motsvarande den tid det tar att förflytta brandgaserna från brandkällan till positionen *x* m. Med andra ord genomförs beräkningen av brandgastemperaturen i en trestegsberäkning, där steg 1 består i att transporttiden beräknas med Ekvation 8, steg 2 i att brandgastemperaturen vid brandkällan beräknas med Ekvation 9, och steg 3 i att brandgastemperaturen vid den efterfrågade positionen beräknas med Ekvation 7.

$$\tau = t - (x/u) \qquad \text{Ekvation 8}$$

$$T_{avg,x=0}(\tau) = T_a + \frac{2}{3}\frac{\dot{Q}(\tau)}{\dot{m}_a c_p} \qquad \text{Ekvation 9}$$

I Ekvation 7 används en s k ”lumpad” värmeöverföringskoefficient *h* för både konvektion och strålning. För utsprängda tunnlar antar Bergqvist, Frantzich, Hasselrot, and Ingason (2001, p. 84) ett värde på *h* motsvarande 0,03 kW/m^2 K. I TuFT används dock värdet 0,02 kW/m^2 K då det har påvisats stämma bäst överens med fullskaleförsök

och FDS-simuleringar för vägtunnlar (Fridolf & Wahlqvist, 2014). I Ekvation 7 antas tunnelväggtemperaturen motsvara omgivningstemperaturen, och c_p antas alltid vara 1 kJ/kg K.

Som kan ses i Ekvation 9 görs antagandet att en tredjedel av den totala effektutvecklingen förloras i form av strålning. Vidare görs antagandet i Ekvation 8 att den tid det tar att transportera brandgaserna från brandkällan till den sökta positionen i tunneln endast beror på ventilationsförhållandena i tunneln, d v s vindhastigheten. Ingen annan påverkan, t ex på grund av den högre brandgastemperaturen i förhållande till omgivningstemperaturen, beaktas i beräkningarna. Det bör dock noteras att denna effekt sannolikt har stor påverkan, i synnerhet nära brandkällan.

Beräkningstekniskt returnerar TuFT en temperatur motsvarande omgivningstemperaturen innan brandgaserna beräknas ha nått positionen x m. Detta gäller såväl när branddynamiksimuleringar genomförs som när utrymnings- och räddningsinsatssimuleringar görs.

4.3 Gaskoncentrationer

Den genomsnittliga gaskoncentrationen för koldioxid (CO_2), kolmonoxid (CO) och vätecyanid (HCN) i tunnelns tvärsnitt på en position x m nedströms branden vid tidpunkten t s in i brandförloppet beräknas med Ekvation 10. På liknande sätt beräknas gaskoncentrationen för syre (O_2) med Ekvation 11. Resultaten presenteras som molfraktioner (uttryckt som volymprocent).

$$X_{i,avg} = Y_i \frac{M_i}{M_a} \frac{\dot{Q}(\tau)}{\dot{m}_a \chi H_T} \qquad \text{Ekvation 10}$$

$$X_{O_2} = 0.2095 - \frac{M_a}{M_{O_2}} \frac{\dot{Q}(\tau)}{\dot{m}_a 13100} \qquad \text{Ekvation 11}$$

Som kan ses i Ekvation 10 baseras beräkningen på information om respektive gas produktionsterm, d v s hur stor andel av gasen som bildas per avbrunnen massenhet bränsle [kg/kg]. Som även kan ses bygger principen på att tidsförskjutningen τ s motsvarande den tid det tar att förflytta brandgaserna från brandkällan till positionen x m beräknas i ett första steg. Detta sker vid en simulering implicit för varje tidssteg och avstånd.

Beräkningstekniskt returnerar TuFT *0.0* innan brandgaserna beräknas ha nått positionen x m (för syre *0.2095*). Detta gäller såväl när branddynamiksimuleringar genomförs som när utrymnings- och räddningsinsatssimuleringar görs.

4.4 Siktlängd

Den genomsnittliga siktlängden i tunnelns tvärsnitt på en position x m nedströms branden vid tidpunkten t s in i brandförloppet beräknas med Ekvation 12. Det beräknade värdet på siktlängden motsvarar siktbarheten för reflekterande föremål, t ex väggar, golv, dörrar och reflekterande skyltar (till skillnad från ljusemitterande skyltar, vilka i genomsnitt skulle kunna ses vid ungefär det dubbla avståndet (Jin, 1978, 2008)). Denna information har använts för att ta fram siktlängd för tunnlar (Ingason, 2012).

$$V = 0.87 \frac{uAH_{ec}}{\dot{Q}(\tau) D_{mass}} \qquad \text{Ekvation 12}$$

Precis som beskrivits i föregående avsnitt bygger principen på att tidsförskjutningen τ s motsvarande den tid det tar att förflytta brandgaserna från brandkällan till positionen x m beräknas i ett första steg. Detta sker vid en simulering implicit för varje tidssteg och avstånd.

Beräkningstekniskt returnerar TuFT *positive infinity* (d v s ett oändligt, positivt tal) innan brandgaserna beräknas ha nått positionen x m. Detta gäller såväl när branddynamiksimuleringar genomförs som när utrymnings- och

räddningsinsatssimuleringar görs. Skälet är att tydliggöra att sikten då ännu är opåverkad av de brandgaser som genererats i branden.

4.5 Värmestrålning från brandkällan

En enkel strålningsmodell används för att beräkna värmestrålning från brandkällan till en avlägsen punkt inom 50 m, se Ekvation 13 (Karlsson & Quintiere, 2000, p. 156). I TuFT antas värmestrålningen från brandkällan vara försumbar när avståndet från brandkällan är > 50 m, och strålningsfraktionen antas alltid vara 1/3 av den totala effektutvecklingen.

$$\dot{q}_{rad,fire} = \frac{\chi_r \dot{Q}(t)}{4\pi R_0^2} \qquad \text{Ekvation 13}$$

Beräkningstekniskt returnerar TuFT den beräknade strålningen i kW/m^2 så länge positionen som specificerats är mindre än 50 m, i annat fall returneras 0,0. Det gäller såväl när branddynamiksimuleringar genomförs som när utrymnings- och räddningsinsatssimuleringar görs.

4.6 Värmeöverföring från omgivande brandgaser

Nedströms branden antas alltid en fullständig omblandning av brandgaser. Information om den genomsnittliga brandgastemperaturen (som beräknas med Ekvation 7-9) kan därmed utnyttjas för att beräkna värmeöverföringen till ett hypotetiskt objekt (t ex en person eller ett fordon) någonstans nedströms branden (så länge brandgaserna har nått dit). I TuFT uppskattas denna värmeöverföring med Ekvation 14 och 15.

Ekvation 14 beskriver i detta sammanhang den strålande komponenten från de omgivande brandgaserna. Infallande strålning från endast varm luft är i de flesta sammanhang försumbar, men i detta fall där en fullständig omblandning av brandgaserna alltid antas så antas emissiviteten alltid vara 0,5 (Purser, 2008). Den konvektiva delen beskrivs av Ekvation 15. Den konvektiva värmeförlustskoefficienten antas i detta sammanhang vara 0,008 kW/m^2 K.

$$\dot{q}_{rad,smoke} = \varepsilon\sigma(T_{avg}^4 - T_m^4) \qquad \text{Ekvation 14}$$

$$\dot{q}_{conv,smoke} = h_c(T_{avg} - T_m) \qquad \text{Ekvation 15}$$

Beräkningstekniskt returnerar TuFT *0,0* innan brandgaserna beräknas ha nått positionen *x* m. Detta gäller såväl när branddynamiksimuleringar genomförs som när utrymnings- och räddningsinsatssimuleringar görs. Därefter returneras de beräknade värdena i kW/m^2. Informationen kan t ex användas för att göra grova bedömningar av t ex när den specificerade branden ev kan sprida sig till närliggande fordon. Informationen utnyttjas även i de s k fraktionsdosberäkningarna som genomförs vid en ev utrymningssimulering, se mer information nedan.

4.7 Backlayeringsträcka

Uppströms branden kan en effekt kallad backlayering inträffa vid tunnelbränder. Det innebär i korthet att en del av brandgaserna trycker sig uppströms (mot vindriktningen). Denna s k backlayeringsträcka beräknas med Ekvation 16.

$$L_b = 1{,}4H\left(\frac{g\dot{Q}}{\rho_a c_p T_a u^3 H}\right)^{0{,}3} \qquad \text{Ekvation 16}$$

Beräkningstekniskt returnerar TuFT alltid den beräknade sträckan för hela brandförloppets tid. Informationen utnyttjas dock endast när branddynamiksimuleringar görs, men kan t ex användas tillsammans med information om räddningstjänstens insats för att bedöma när insatsen kommer att möta rök när den genomförs uppströms branden.

5 Beräkningstekniska principer för de tre simuleringslägena

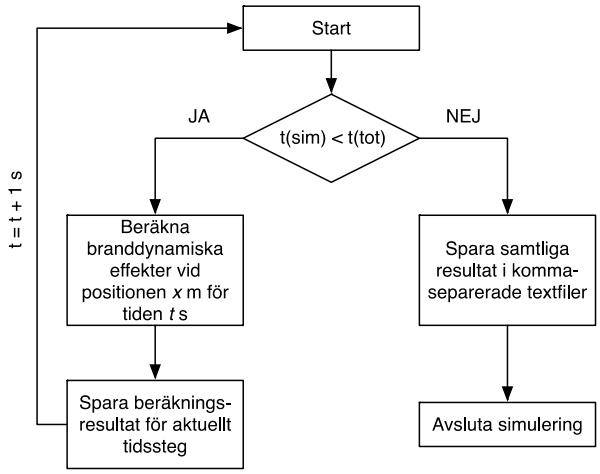
I detta kapitel presenteras de beräkningstekniska principerna för de tre simuleringslägena (branddynamik, utrymning och räddningsinsats). Presentationen görs generellt, och ingående ekvationer utelämnas. Utgångspunkten är de ekvationer som presenterats ovan. Den stora skillnaden mellan simuleringsläget branddynamik och de övriga två är att positionen x m i tunneln varierar i de senare. Skälet är naturligtvis att de utrymmande individerna och/eller räddningstjänstens personal förväntas röra sig i tunneln under brandförloppet. För varje tidssteg utvärderas därför först deras respektive position i tunneln i förhållande till branden. Hur snabbt förflyttningen sker under brandförloppet beror i huvudsak på siktlängden (d v s röktätheten), enligt de samband som presenterats ovan (se Tabell 9 och Tabell 13).

5.1 Nomenklatur

A_{body}	Kroppsyta (antas vara 1,85 för samtliga) [m^2]
c_p	Värmekapacitet [kJ/kg K]
D	Exponeringsdos (procent COHb) för medvetslöshet [%]
f	Faktor för areaökning p g a skyddskläder (antas vara 1,3 för samtliga) [-]
f_{eff}	Del av kroppsyta som är mottaglig för strålning (antas vara 0,71) [-]
FID	Fraktionsdos för medvetslöshet (avseende värme eller toxiska gaser) [-]
FLD	Fraktionsdos för död (avseende värme eller toxiska gaser) [-]
M	Värmeproduktion i kroppen (antas vara 300 för samtliga) [W/m^2 K]
m	Kroppsmassa (antas vara 75 för samtliga) [kg]
p_a	Vattenångas partialtryck i omgivningen (antas vara 700) [Pa]
p_s	Vattenångas partialtryck vid huden (antas vara 5940) [Pa]
$\dot{Q}$	Effektutveckling [kW]
$\dot{q}_{conv,smoke}$	Konvektiv värmeöverföring från brandgaser [kW/m^2]
$\dot{q}_{rad,fire}$	Strålande värmeöverföring från brandkällan [kW/m^2]
$\dot{q}_{rad,smoke}$	Strålande värmeöverföring från brandgaser [kW/m^2]
R_a/R_c	Andel av strålning som ej absorberas i kläder (antas vara .15 för samtliga) [-]
R_c	Värmemotstånd i skyddskläder (antas vara 0,465 för samtliga) [K m^2/W]
R_e	Skyddskläders ångmotstånd (antas vara 75 för samtliga) [Pa m^2/W]
r	Ändpunkt för värmeexponeringsdos [(kW/m^2)$^{1.33}$ min]
t	Tid [s]
Δt	Tid till värmeutmattning [°C]
ΔT	Ökning av kroppstemperatur [°C]
T_{body}	Kroppsmedeltemperatur (antas vara 37 för samtliga) [°C]
T_{avg}	Genomsnittlig temperatur i tunnelns tvärsnitt [°C]
V	Andningsfrekvens [l/min]
V_{CO_2}	Förhöjd andningsfrekvens orsakad av inandning av CO_2 [-]

5.2 Branddynamik

Sett ur ett beräkningstekniskt perspektiv är simuleringsläget ”branddynamik” det minst komplicerade. Baserat på den information som användaren angivit kopplat till branden, tunneln, tunnelbrandscenariot och mätpunkterna genomförs en simulering med utgångspunkt i de ekvationer som beskrivits ovan. För varje tidssteg (d v s varje sekund av brandförloppet) beräknas brandgastemperaturen, gaskoncentrationerna och siktlängden för de positioner nedströms branden som användaren definierat enligt beskrivningen i Figur 5. Det är alltså inte möjligt att genomföra beräkningar uppströms branden. Beräkningar görs för hela brandförloppet, och den totala tiden bestäms av brinntiden (d v s tiden det tar för branden att tillväxa till maximal effektutveckling, tiden branden brinner med maximal effektutveckling, och tiden det tar för branden att avsvalna till 0 kW). Under pågående simulering sparas resultatet temporärt i vektorer. Innehållet i dessa vektorer exporteras vid beräkningens slut till kommaseparerade textfiler för bearbetning och visualisering i t ex Microsoft Excel.



Figur 5. Den beräkningstekniska principen för simuleringsläget ”branddynamik”. Indata till simuleringen är information om tunneln, branden, tunnelbrandscenariot och de mätpunkter vid vilka beräkningarna ska ske.

5.2.1 Utdata

De kommaseparerade textfiler som genereras efter utförd simulering är sammanställda i Tabell 14 tillsammans med en kort beskrivning.

Tabell 14. Beskrivning av de kommaseparerade textfiler som genereras vid en simulering i simuleringsläge ”utrymning”.

Namn	Beskrivning
HRR.txt	Innehåller information om brandens effektutveckling för varje sekund av brandförloppet. Kolumn 1 redovisar tiden i s och kolumn 2 den motsvarande effektutvecklingen i kW.
tunnelBacklayeringDistance.txt	Innehåller information om den s k backlayeringsträckan för varje sekund av brandförloppet (oberoende av positioner som angivits). Kolumn 1 redovisar tiden i s och kolumn 2 den motsvarande backlayeringsträckan uppströms branden [m].
tunnelExtinction.txt	Innehåller information om den genomsnittliga dämpningskoefficienten i tunnelns tvärsnitt för de positioner nedströms branden som specificerats av användaren för varje sekund av brandförloppet. Kolumn 1 redovisar tiden i s och kolumn 2 till *n-1* (där *n* anger

Namn	Beskrivning
	antalet specificerade positioner) den motsvarande dämningskoefficienten [m^{-1}].
tunnelFireRadiation.txt	Innehåller information om den infallande värmestrålningen från brandkällan för de positioner nedströms branden som specificerats av användaren för varje sekund av brandförloppet. Kolumn 1 redovisar tiden i s och kolumn 2 till *n-1* (där *n* anger antalet specificerade positioner) den motsvarande strålningsintensiteten [kW/m^2].
tunnelGasMoleFraction.txt	Innehåller information om den genomsnittliga gaskoncentrationen för CO_2, CO, HCN och O_2 i tunnelns tvärsnitt för de positioner nedströms branden som specificerats av användaren för varje sekund av brandförloppet. Kolumn 1 redovisar tiden i s och kolumn 2 till *5+4*(n-1)* (där *n* anger antalet specificerade positioner) den motsvarande gaskoncentrationen av respektive ämne [vol-%].
tunnelHeatTransfer.txt	Innehåller information om den totala värmeöverföringen till ett hypotetiskt objekt för de positioner nedströms branden som specificerats av användaren för varje sekund av brandförloppet. Kolumn 1 redovisar tiden i s och kolumn 2 till *n-1* (där *n* anger antalet specificerade positioner) den motsvarande värmeöverföringen [kW/m^2]. Observera att summan inkluderar den infallande strålningen från brandkällan (om positionen som specificerats är < 50 m från brandkällan).
tunnelTemp.txt	Innehåller information om den genomsnittliga brandgastemperaturen i tunnelns tvärsnitt för de positioner nedströms branden som specificerats av användaren för varje sekund av brandförloppet. Kolumn 1 redovisar tiden i s och kolumn 2 till *n-1* (där *n* anger antalet specificerade positioner) den motsvarande brandgastemperaturen [°C].
tunnelVisibility.txt	Innehåller information om den genomsnittliga siktlängden i tunnelns tvärsnitt för de positioner nedströms branden som specificerats av användaren för varje sekund av brandförloppet. Kolumn 1 redovisar tiden i s och kolumn 2 till *n-1* (där *n* anger antalet specificerade positioner) den motsvarande siktlängden [m].

5.2.2 Exempel på utdata

Exempel på utdata för de respektive parametrarna presenteras nedan. Innehållet består av bearbetad data baserad på utvalda delar av de kommaseparerade textfiler som TuFT genererar vid en simulering. Exemplet bygger på ett scenario med de storheter som presenteras i Tabell 15-Tabell 19.

Tabell 15. Simulerings omfattning; endast simuleringsläge "branddynamik" aktiverat.

Parameter	Enhet	Indata
Branddynamik	true/false	*true*
Utrymning	true/false	*false*
Räddningsinsats	true/false	*false*

Tabell 16. Typen av tunnel; i detta fall en vägtunnel med en total längd på 1700 m.

Parameter	Enhet	Indata
Typ av tunnel	Väg- eller järnväg	*"road"*
Längd	m	*1700.0*
Bredd	m	*9.0*
Höjd	m	*6.0*
Vindhastighet	m/s	*2.0*
Omgivningstemperatur	°C	*20.0*
Avstånd mellan utrymningsvägar	m	*150.0*

Tabell 17. Typen av brand; i detta fall en kvadratiskt tillväxande brand med maximal effektutveckling motsvarande 60 MW. Information om produktionstermer baseras på ett simulerat bränsle likt fast polyuretanplast (Ingason, 2012).

Parameter	Enhet	Indata
Typ av tillväxt	Linjär eller kvadratisk	*"squared"*
Tillväxthastighet	kW eller kW/s^2	*0.19*
Avsvalningshastighet	kW eller kW/s^2	*0.19*
Maximal effektutveckling	kW	*60000.0*
Varaktighet för maximal effektutveckling	min	*60*
Rökpotential	m^2/kg	*304.0*
Förbränningsvärme	kJ/kg	*15500.0*
Förbränningseffektivitet	-	*1.0*
CO-produktion	kg/kg	*0.027*
CO_2-produktion	kg/kg	*1.50*
HCN-produktion	kg/kg	*0.01*

Tabell 18. Tunnelbrandscenariot; branden placeras på positionen *700* m i tunneln (mätt från *x* = *0* m). Vindriktningen är från *x* = *0* m till *x* = *1700 m*.

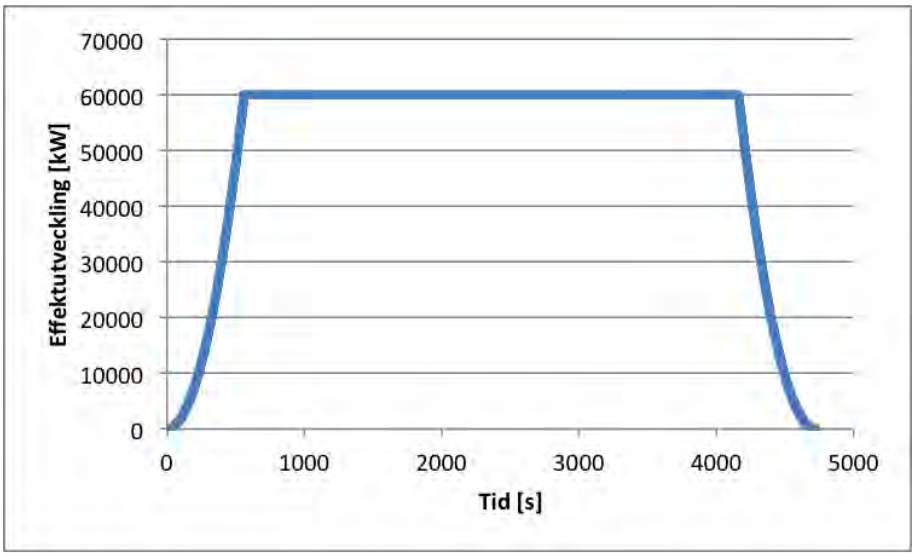
Parameter	Enhet	Indata
Position	m	*700.0*
Vindriktning från nollpunkt	true/false	*true*

Tabell 19. Mätpunkter; tre mätpunkter definieras nedströms branden. Det bör i sammanhanget poängteras att dessa punkter definieras relativt brandens position. De faktiska positionerna i tunneln är i detta fall alltså *800 m*, *1000 m* samt *1500 m*.

Parameter	Enhet	Indata
Position	m	*100.0*
Position	m	*300.0*
Position	m	*800.0*

5.2.2.1 Effektutveckling

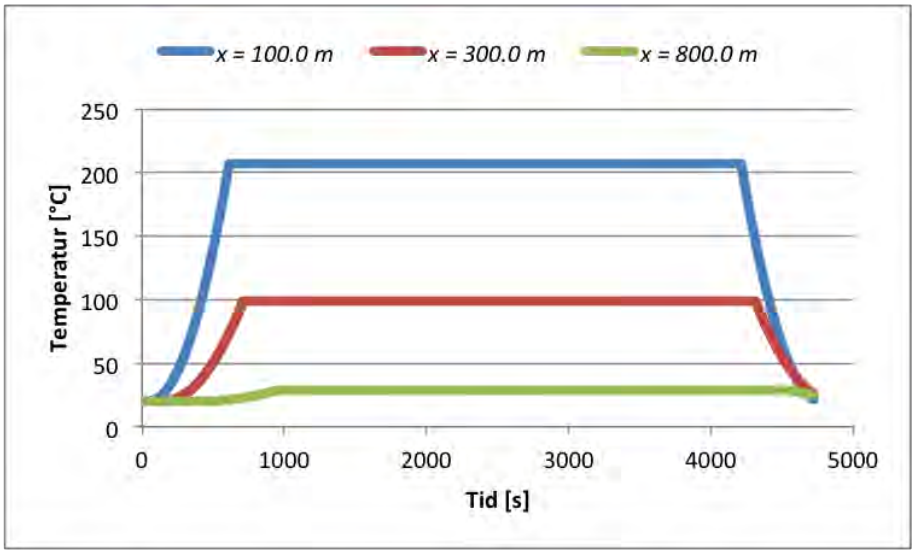
Oavsett simuleringsläge skapas alltid en textfil innehållandes information om brandens effektutveckling för varje sekund av brandförloppet. För den brand som beskrivs av Tabell 15-Tabell 19 kan därför effektutvecklingen som en funktion av tiden lätt illustreras i ett diagram, se t ex Figur 6.



Figur 6. Effektutvecklingskurvans utseende. Branden tillväxer kvadratiskt med 0,19 kW/s^2 för att sedan brinna i 60 min med sin maximala effektutveckling motsvarande 60 MW innan den avsvalnar med samma hastighet som den tillväxte.

5.2.2.2 Brandgastemperatur

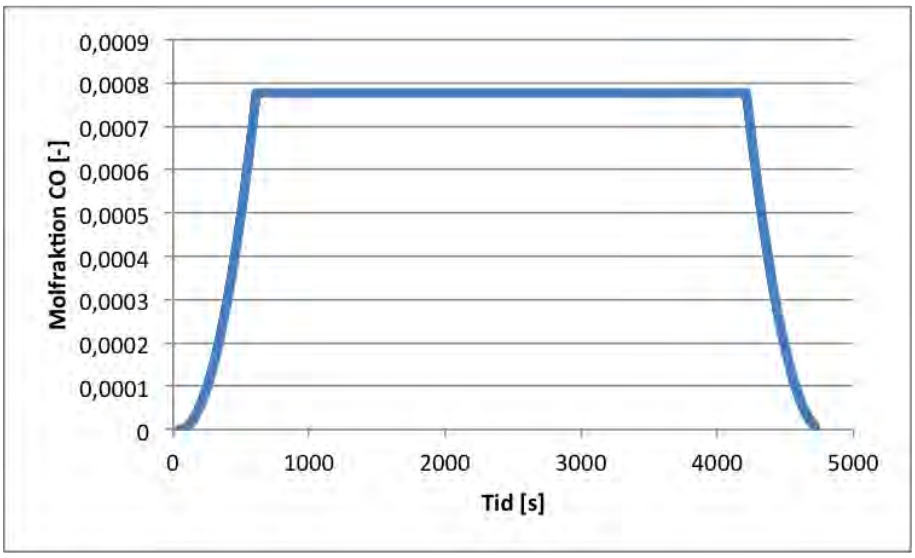
I simuleringsläget ”branddynamik” beräknas brandgastemperaturen för de positioner nedströms branden som specificerats av användaren, i detta fall 100 m, 300 m samt 800 m. Beräkningen sker m h a Ekvation 7-9 och resultatet sparas i en kommaseparerad textfil. Det gör att brandgastemperaturen lätt kan presenteras i ett diagram som en funktion av tiden för respektive position, se t ex Figur 7.



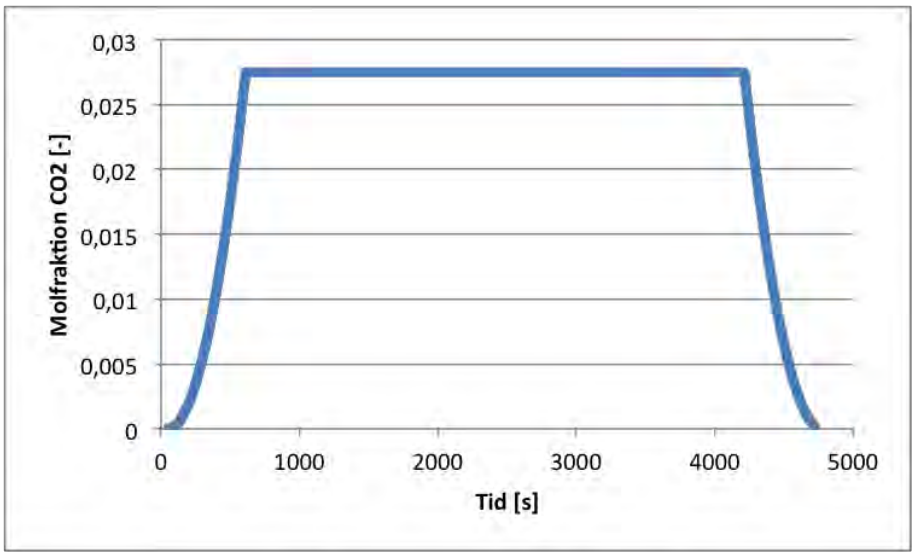
Figur 7. Brandgastemperaturen presenterad som en funktion av tiden i brandförloppet på tre olika positioner nedströms branden i tunneln. Ju längre bort från branden desto lägre temperatur p g a värmeförluster från brandgaserna till omgivningen.

5.2.2.3 Gaskoncentrationer

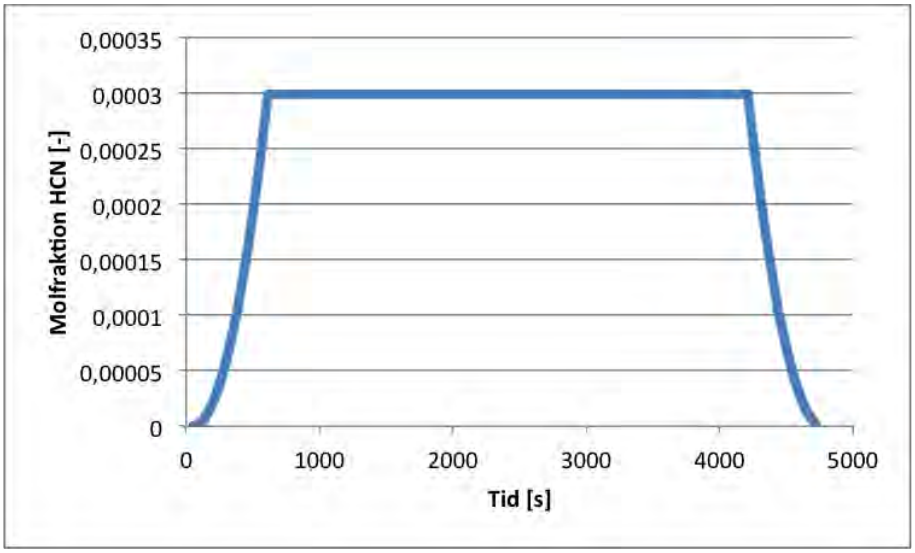
I simuleringsläget ”branddynamik” beräknas gaskoncentrationer för koldioxid, kolmonoxid, vätecyanid och syre för de positioner nedströms branden som specificerats av användaren, i detta fall 100 m, 300 m samt 800 m. Beräkningarna sker m h a Ekvation 10 och Ekvation 11 och resultatet sparas i en kommaseparerad textfil. Det gör att gaskoncentrationen för respektive ämne lätt kan presenteras i diagram som en funktion av tiden för respektive position, se t ex Figur 8-Figur 11, där positionen $x = 100$ m nedströms branden valts som exempel.



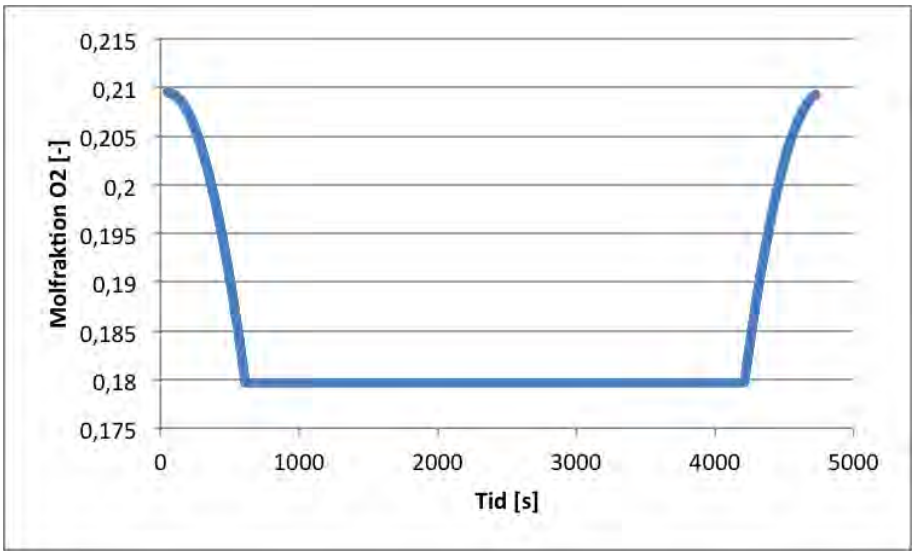
Figur 8. Koncentrationen av kolmonoxid presenterad som en funktion av tiden på positionen *x = 100 m* nedströms branden.



Figur 9. Koncentrationen av koldioxid presenterad som en funktion av tiden på positionen *x = 100 m* nedströms branden.



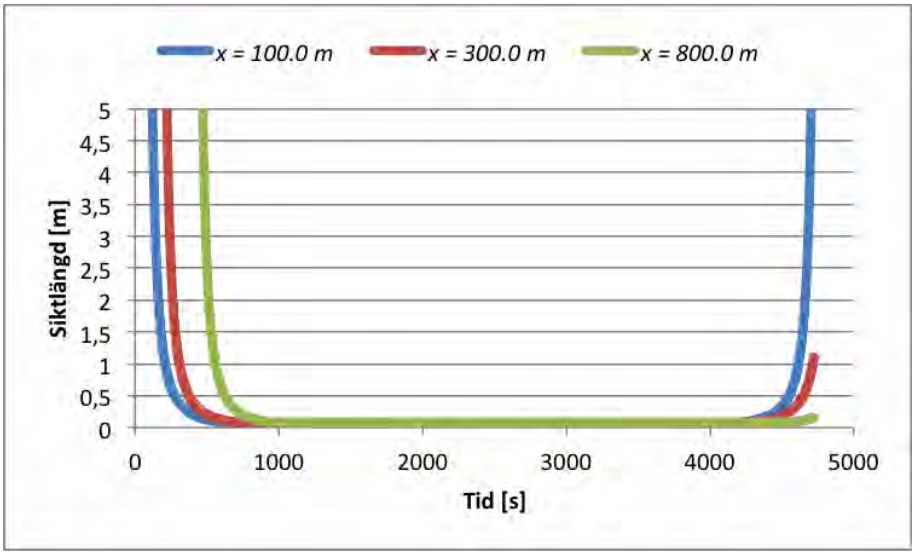
Figur 10. Koncentrationen av vätecyanid presenterad som en funktion av tiden på positionen *x = 100 m* nedströms branden.



Figur 11. Koncentrationen av syre presenterad som en funktion av tiden på positionen *x = 100 m* nedströms branden.

5.2.2.4 Siktlängd

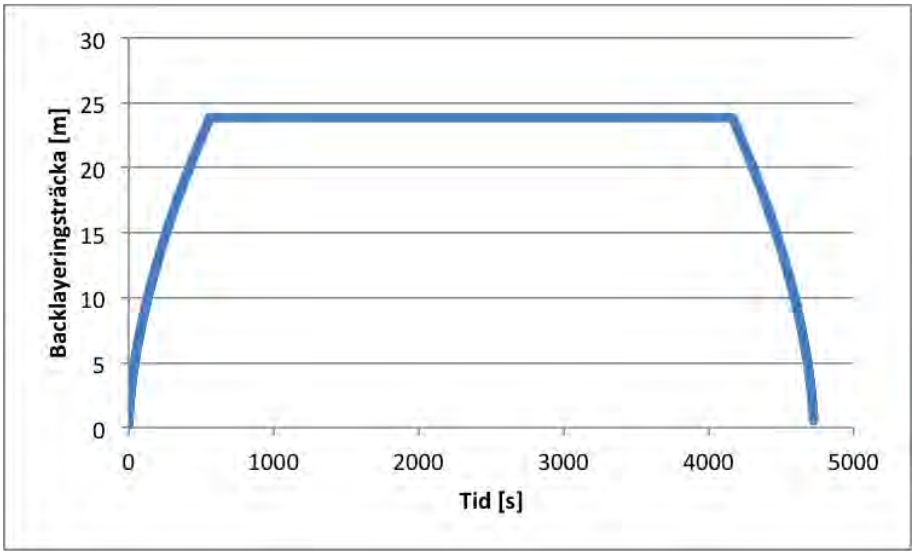
I simuleringsläget ”branddynamik” beräknas såväl dämpningskoefficienten (engelska *extinction coefficient*) som siktlängden för de positioner nedströms branden som av användaren, i detta fall 100 m, 300 m samt 800 m. Beräkningarna sker bl a med hjälp av Ekvation 12 och resultatet sparas i en kommaseparerad textfil. Det gör att siktlängden lätt kan presenteras i diagram som en funktion av tiden för respektive position, se t ex Figur 12.



Figur 12. Siktlängden presenterad som en funktion av tiden i brandförloppet på tre olika positioner nedströms branden i tunneln.

5.2.2.5 Backlayering

I simuleringsläget ”branddynamik” beräknas alltid backlayeringsträckan, oavsett vilka positioner som angivits. Beräkningarna sker m h a Ekvation 16 och resultatet sparas i en kommaseparerad textfil. Det gör att backlayeringsträckan lätt kan presenteras i diagram som en funktion av tiden i brandförloppet, se t ex Figur 13.



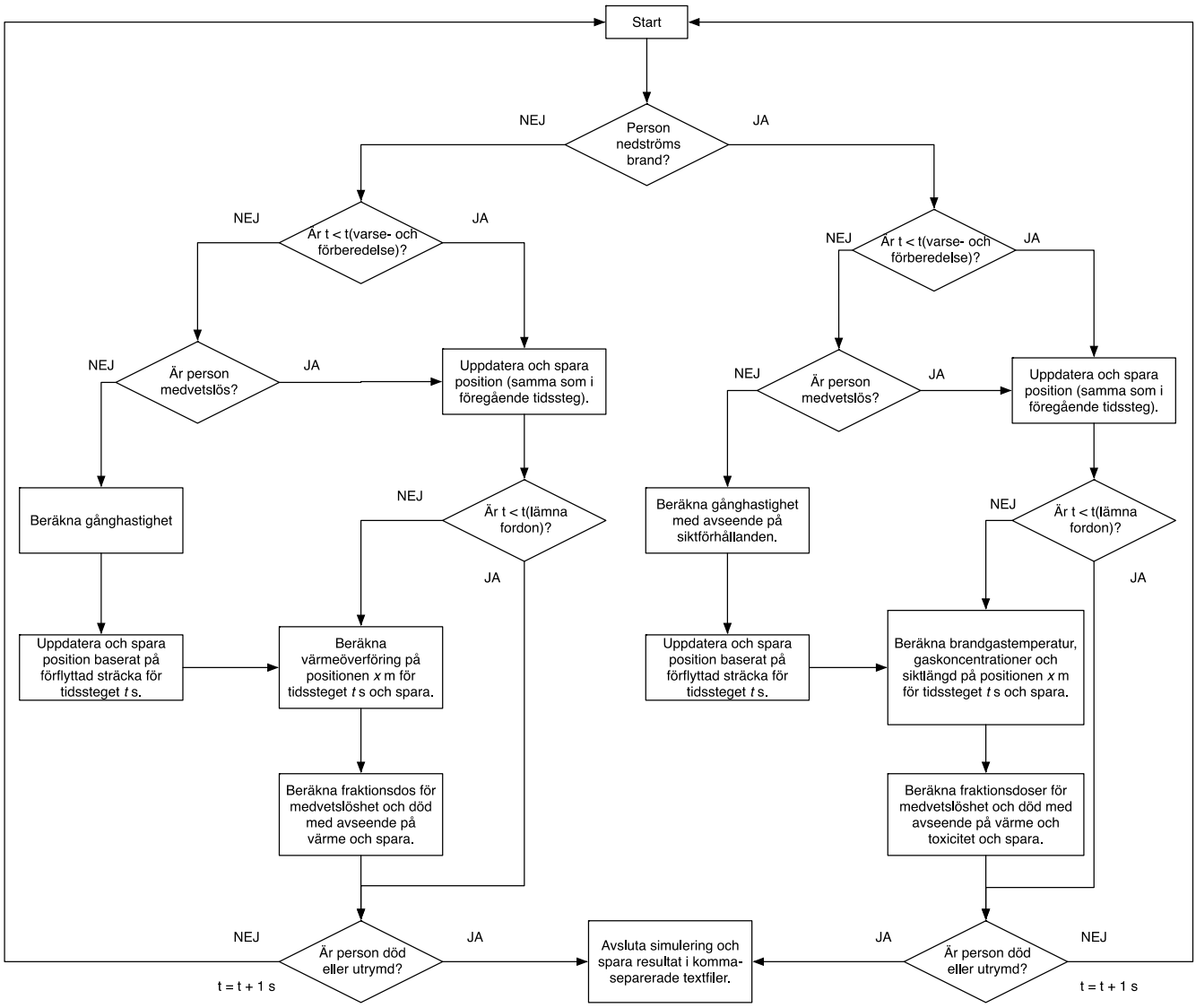
Figur 13. Backlayeringsträckan uppströms branden presenterad som en funktion av tiden i brandförloppet.

5.3 Utrymning

Simuleringsläget ”utrymning” möjliggör bedömning av en tunnelbrands påverkan på utrymmande personer. I sammanhanget ska åter nämnas att beräkningar sker endimensionellt, både för väg- och järnvägstunnlar. Beräkningstekniskt innebär det att TuFT simulerar utrymning från tunneln baserat på en start- och slutposition. Samtliga utrymmande individer antas alltid röra sig från brandkällan till en säker plats (antingen en utrymningsväg eller en av tunnelportalerna, beroende på vad användaren angivit i indata till simuleringen).

För varje tidssteg av simuleringen (d v s varje sekund av brandförloppet) beräknas bl a brandgastemperatur, gaskoncentrationer och siktlängd för den position i tunneln som den utrymmande personen befinner sig i (om denna befinner sig nedströms branden). Beräkningarna sker enligt de principer som beskrivits ovan, men den stora skillnaden är att positionen x m i tunneln varierar eftersom att de utrymmande personerna förväntas röra sig under utrymningen. Simuleringen pågår så länge det finns något att räkna på, d v s den görs inte för hela brandförloppet utan avslutas istället när den sista personen antingen utrymt tunneln eller avlidit p g a påverkan av värme och/eller toxiska produkter. Påverkan av värme och/eller toxiska produkter görs med s k fraktionsdosberäkningar enligt det FED-koncept som presenterats av Purser (2008). En närmare beskrivning av hur detta hanteras i TuFT (d v s hur TuFT bedömer att en individ blivit medvetslös eller avlidit) redogörs för i ett eget avsnitt nedan.

Principiellt kan beräkningsgången i varje tidssteg för alla individer som simuleras beskrivas enligt Figur 14. Beräkning sker för en individ i taget och när simuleringen av en individs totala utrymningsförlopp avslutats påbörjas den för nästa till dess att samtliga individer simulerats. Individerna antas med andra ord flytta sig helt oberoende av varandra och det finns ingen samverkan mellan dem (inte ens om de specificerats som en grupp). Under pågående simulering sparas resultatet temporärt i vektorer som vid simulerings slut exporteras till kommaseparerade textfiler för bearbetning och visualisering i t ex Microsoft Excel.



Figur 14. Den beräkningstekniska principen för simuleringsläget "utrymning". Indata till simuleringen är information om tunneln, branden, tunnelbrandscenariot och de personer som simuleras utrymma tunneln.

I utrymningssimuleringsläget börjar TuFT med den individ som angivits överst i indata-filen. Som kan ses i Figur 14 kontrolleras i ett första steg individens position i förhållande till branden. Därefter kontrolleras huruvida det aktuella tidssteget är större än summan av de specificerade s k varse- och förberedelsetiderna (d v s huruvida individen börjat förflytta sig). Den stora principiella skillnaden mellan att befinna sig uppströms istället för nedströms branden består i de fraktionsdosberäkningar som görs i varje tidssteg. Befinner sig en individ uppströms branden görs nämligen inga fraktionsdosberäkningar avseende toxiska gaser. Inte heller görs några fraktionsdosberäkningar av temperaturpåverkan från brandgaser. Skälet är att TuFT endast räknar på brandgasspridning nedströms branden, och i och med detta antagande befinner sig alla individer uppströms branden därmed i rökfri miljö. Dock sker med hjälp av fraktionsdosberäkningar en uppskattning av värmepåverkan kopplad till strålning från branden även för individer uppströms branden. Beräkningen sker om individen befinner sig mindre än 50 m från brandkällan, därefter antas strålningspåverkan vara försumbar (samma princip gäller för individer nedströms branden). Utöver principen för fraktionsdosberäkningarna är de beräkningstekniska skillnaderna mellan de två fallen (upp- eller nedströms branden) små, vilket också kan ses i Figur 14.

Enligt den beräkningstekniska principen sker fraktionsdosberäkningar i varje tidssteg av simuleringen. Undantaget är, som kan ses i Figur 14, om tiden för då individen lämnar fordonet ännu inte passerats. Antagandet som görs är att en individ som ännu inte lämnat sitt fordon inte heller påverkas av branden. Fraktionsdosberäkningen i varje tidssteg

sparas i en vektor, och används sedan för att kunna beräkna den ackumulerade fraktionsdosen vid kontroll av medvetslöshet eller dödsfall.

5.3.1 Metod för bedömning av värme- och toxisk påverkan

Som nämnts ovan och som kan ses i Figur 14 görs fraktionsdosberäkningar enligt FED-konceptet för att bedöma påverkan av värme- och giftiga gaser på utrymmande individer i TuFT. FED står i detta sammanhang för *fractional effective*[6] *dose* och principen för bedömningspåverkan bygger på den forskning och de modeller som utvecklats av Purser (2008). Den innebär att kännedom om olika branddynamiska parametrar används för att beräkna den för tidssteget aktuella upptagna fraktionsdosen för medvetslöshet/död av en viss miljöparameter. Beräkningen sker i tidssteg om 1 s (som kan ses i Figur 14), och för varje tidssteg adderas den beräknade fraktionsdosen till den dittills erhållna dosen avseende såväl toxicitet som värme. Den ackumulerade fraktionsdosen, d v s summan av de tidigare tidsstegens fraktionsdoser, används sedan för att bedöma om individen är medvetslös eller död.

Gränsvärden för medvetslöshet är i TuFT 1,0 för värme och toxiska produkter (det räcker att ett av värdena överskrids), d v s medvetslöshet bedöms inträda då den ackumulerade fraktionsdosen för medvetslöshet FID (*fractional incapacitating dose*) är lika med 1,0. På motsvarande sätt bedöms död inträffa då den ackumulerade fraktionsdosen för medvetslöshet FID är lika med 2,0 för toxiska produkter, och 1,67 för värme (i enlighet med Purser (2008)). Skillnaden i ändpunkt för dödsfall beror på olika beräkningsprinciper vid påverkan av toxiska produkter och vid påverkan av värme.

5.3.1.1 Toxisk påverkan

Bedömningen av toxisk påverkan görs enligt Ekvation 17-21. Indata till dessa beräkningar är bl a koncentrationen av de olika gaser som beräknas med Ekvation 10-11, baserat dels på individens position x m i tunneln och tiden t s in i brandförloppet. Innan eventuell medvetslöshet uppstår görs fraktionsdosberäkningarna för en individ med en aktivitetsnivå som motsvarar den för en lätt arbetande person ($V = 25$ l/min och $D = 30$ % COHb). Efter ev medvetslöshet görs istället fraktionsdosberäkningarna för en individ med en aktivitetsnivå som motsvarar den för en sovande eller vilande person ($V = 8{,}5$ l/min och $D = 40$ % COHb). Det innebär således att beräkningen av kolmonoxidupptaget förändras efter medvetslöshet (det sker långsammare p g a antagandet om en lägre andningsfrekvens), vilket i sin tur påverkar den ackumulerade dosen av toxiska produkter. Efter medvetslöshet kan dock individen inte förflytta sig, men beräkning fortsätter till dess att dödsfall inträffar för att illustrera den extratid som finns för att undsätta individen, t ex vid en räddningsinsats.

Som kan ses i Ekvation 17 bestäms för tidssteget aktuell fraktionsdos som ett samband av fraktionsdoserna för respektive ämne. Koldioxid antas i sammanhanget inte förgifta individerna, men påverkar istället upptaget av giftiga gaser p g a den hyperventilerande effekt som koldioxid har vid höga koncentrationer (Purser, 2008). Koncentrationerna av de respektive gaserna är markerade inom s k hakparanteser, t ex *[CO]* för kolmonoxid. Beroende på vilken ekvation som avses antingen koncentrationen i ppm eller i volymprocent, se dokumentation av Purser (2008) för mer information. Tidssteget, t, är alltid 1 s i TuFT men redovisas ändå nedan för tydlighetens skull.

$$FID_{asphyxiants} = [FID_{CO} + FID_{HCN}]V_{CO_2} + FID_{O_2} \qquad \text{Ekvation 17}$$

$$FID_{CO} = \left[\frac{3.317 \cdot 10^{-5} \cdot [CO]^{1.036} \cdot V}{D}\right]\frac{t}{60} \qquad \text{Ekvation 18}$$

$$FID_{HCN} = \left[\frac{e^{[HCN]/43}}{20} - .0045\right]\frac{t}{60} \qquad \text{Ekvation 19}$$

$$V_{CO_2} = e^{\frac{[CO_2]}{5}} \qquad \text{Ekvation 20}$$

[6] I detta sammanhang är effekten som söks antingen medvetslöshet eller dödsfall.

$$FID_{O_2} = \left[\frac{1}{e^{8.13-.54\cdot(20.9-[O_2])}}\right]\frac{t}{60}$$ Ekvation 21

5.3.1.2 Värmepåverkan

Bedömningen av värmepåverkan görs enligt Ekvation 22 (som utnyttjar Ekvation 13-15). Indata till dessa beräkningar är bl a information om den genomsnittliga brandgastemperaturen i tunnelns tvärsnitt för individens position *x* m vid tiden *t* s in i brandförloppet, och beräknas med Ekvation 7. Som kan ses bidrar tre komponenter till bedömningen av värmepåverkan på utrymmande individer i TuFT. För personer nedströms branden antas bidrag ske från alla tre komponenterna, men för utrymmande personer uppströms branden antas bara ett bidrag ske från brandkällan.

Det första bidraget utgörs av värmestrålning från själva brandkällan och beräknas med Ekvation 13 (Karlsson & Quintiere, 2000, p. 156). I enlighet med vad som skrivits ovan inkluderas dock denna term endast om den utrymmande individen befinner sig < 50 m från brandkällan, i annat fall antas påverkan i sammanhanget vara försumbar. I TuFT antas strålningsfraktionen uppgå till en tredjedel av brandens totala effektutveckling. Det andra och tredje bidraget består av infallande strålning och konvektion från de brandgaser som antas omsluta de utrymmande individerna när de befinner sig nedströms branden och beräknas med Ekvation 14-15 (Purser, 2008). I sammanhanget är det därför viktigt att återigen påpeka antagandet som görs om att brandgaserna antas vara helt omblandade (en förutsättning för de ekvationer för beräkning av branddynamiska effekter som beskrivits ovan) och att temperaturen därför antas vara lika över hela tunneltvärsnittet vid en given plats i tunneln. Detta är ett antagande som, i synnerhet nära brandkällan, inte är korrekt p g a den skiktning som kan förväntas uppstå.

Skillnaden i beräkningen av fraktionsdosen för medvetslöshet och död ligger i värdet på ändpunkten *r* i ekvationerna nedan. Vid bedömning av medvetslöshet används värdet 10 $(kW/m^2)^{1.33}$ min och vid bedömning av dödsfall används värdet 16.7 $(kW/m^2)^{1.33}$ min (Purser, 2008). Det innebär alltså *FID = 1,67* detsamma som *FLD = 1,0*, d v s en person bedöms som avliden när den ackumulerade fraktionsdosen för medvetslöshet är 167 % av vad som krävs för medvetslöshet.

$$FID_{heat}/FLD_{heat} = 1/\left(\left[\frac{r}{(\dot{q}_{rad,fire}+\dot{q}_{rad,smoke}+\dot{q}_{conv,smoke})^{1,33}}\right]\frac{t}{60}\right)$$ Ekvation 22

5.3.2 Utdata

De kommaseparerade textfiler som genereras efter utförd simulering är sammanställda i Tabell 20 tillsammans med en kort beskrivning.

Tabell 20. Beskrivning av de kommaseparerade textfiler som genereras vid en simulering i simuleringsläge "branddynamik".

Namn	Beskrivning
evacConcentrationCo.txt	Innehåller information om den gaskoncentration (volymprocent) kolmonoxid som omgivit individen för varje sekund av utrymningen. Beräknas för varje tidssteg *t* s i brandförloppet baserat på individens position *x* m i tunneln för det givna tidssteget. Kolumn 1 redovisar tiden i s och kolumn 2 till *n+1* (där *n* står för det totala antalet individer) den motsvarande gaskoncentrationen för individ *i* [vol-%].
evacConcentrationCo2.txt	Innehåller information om den gaskoncentration (volymprocent) koldioxid som omgivit individen för varje sekund av utrymningen. Beräknas för varje tidssteg *t* s i brandförloppet baserat på individens position *x* m i tunneln för det givna tidssteget. Kolumn 1 redovisar tiden i s och kolumn 2 till *n+1* (där *n* står för det totala

Namn	Beskrivning
	antalet individer) den motsvarande gaskoncentrationen för individ *i* [vol-%].
evacConcentrationHcn.txt	Innehåller information om den gaskoncentration (volymprocent) vätecyanid som omgivit individen för varje sekund av utrymningen. Beräknas för varje tidssteg *t* s i brandförloppet baserat på individens position *x* m i tunneln för det givna tidssteget. Kolumn 1 redovisar tiden i s och kolumn 2 till *n+1* (där *n* står för det totala antalet individer) den motsvarande gaskoncentrationen för individ *i* [vol-%].
evacConcentrationO2.txt	Innehåller information om den gaskoncentration (volymprocent) syre som omgivit individen för varje sekund av utrymningen. Beräknas för varje tidssteg *t* s i brandförloppet baserat på individens position *x* m i tunneln för det givna tidssteget. Kolumn 1 redovisar tiden i s och kolumn 2 till *n+1* (där *n* står för det totala antalet individer) den motsvarande gaskoncentrationen för individ *i* [vol-%].
evacEndPosition.txt	Innehåller en sammanfattning av samtliga simulerade individers slutposition i tunneln (d v s den position i tunneln där de antingen avled eller utrymde). Kolumn 1 anger individ *i*, kolumn 2 tidpunkten *t* s i brandförloppet då individen nådde slutpositionen, kolumn 3 positionen *x* m i tunneln och kolumn 4 information om personens hälsotillstånd när denne nådde slutpositionen (*dead* symboliserar att individen inte lyckades ta sig ut ur tunneln och *alive* att personen utrymde tunneln på den position *x* m som anges).
evacEvacuatedPerTime.txt	Innehåller en sammanfattning av antalet utrymda individer. Kolumn 1 anger tidpunkten *t* s i brandförloppet då individ *i* utrymde tunneln, kolumn 2 det ackumulerade antalet utrymda individer och kolumn 3 andelen i % av samtliga individer i tunneln.
evacFidAsphyxia.txt	Innehåller information om den ackumulerade fraktionsdosen för medvetslöshet avseende toxicitet. Kolumn 1 redovisar tiden *t* s i brandförloppet och kolumn 2 till *n+1* (där *n* står för det totala antalet individer) den motsvarande, ackumulerade fraktionsdosen för individ *i* (FID = 1,0 representerar medvetslöshet och 2,0 död).
evacFidHeat.txt	Innehåller information om den ackumulerade fraktionsdosen för medvetslöshet avseende värme. Kolumn 1 redovisar tiden *t* s i brandförloppet och kolumn 2 till *n+1* (där *n* står för det totala antalet individer) den motsvarande, ackumulerade fraktionsdosen för individ *i* (FID = 1,0 representerar medvetslöshet och 1,67 död).
evacPosition.txt	Innehåller information om den positionen *x* m i tunneln under utrymningsförloppet. Kolumn 1 redovisar tiden *t* s i brandförloppet och kolumn 2 till *n+1* (där *n* står för

Namn	Beskrivning
	det totala antalet individer) den motsvarande positionen *x* m i tunneln för individ *i*.
evacSummary.txt	Innehåller sammanfattad information om simuleringen, bl a tunnelns längd, brandens position i tunneln, antalet individer som ingick i utrymningssimuleringen och deras varse- och förberedelsetider, tid för att lämna fordonet, start- och slutposition i tunneln, sträckan som de avverkade innan de tog sig till en säker plats/bedömdes avlidna, ackumulerade fraktionsdoser, om de lyckades utrymma eller inte samt hur lång tid de vistades i tunneln innan de tog sig till en säker plats/bedömdes avlidna.
evacTemperature.txt	Innehåller information om den brandgastemperatur (i °C) som omgivit individen för varje sekund av utrymningen. Beräknas för varje tidssteg *t* s i brandförloppet baserat på individens position *x* m i tunneln för det givna tidssteget. Kolumn 1 redovisar tiden i s och kolumn 2 till *n+1* (där *n* står för det totala antalet individer) den motsvarande temperaturen för individ *i*.
evacVisibility.txt	Innehåller information om den siktlängd (i m) som omgivit individen för varje sekund av utrymningen. Beräknas för varje tidssteg *t* s i brandförloppet baserat på individens position *x* m i tunneln för det givna tidssteget. Kolumn 1 redovisar tiden i s och kolumn 2 till *n+1* (där *n* står för det totala antalet individer) den motsvarande siktlängden för individ *i*.

5.3.3 Exempel på utdata

Exempel på utdata vid en utrymningssimulering presenteras nedan. Innehållet består av bearbetad data baserad på ett urval av de kommaseparerade textfiler som TuFT genererar vid en simulering. Exemplet bygger på samma tunnelbrandscenario som beskrivits i Tabell 16-Tabell 19, med följande tillägg som är specificerade i Tabell 21. Eftersom att simulering av utrymning nu ska ske aktiveras denna modul i Tabell 15 genom att *false* ändras till *true*. I exemplet ingår endast en individ som utrymmer en vägtunnel, men principen är densamma vid simulering av flera individer eller utrymning av ett tåg i en järnvägstunnel.

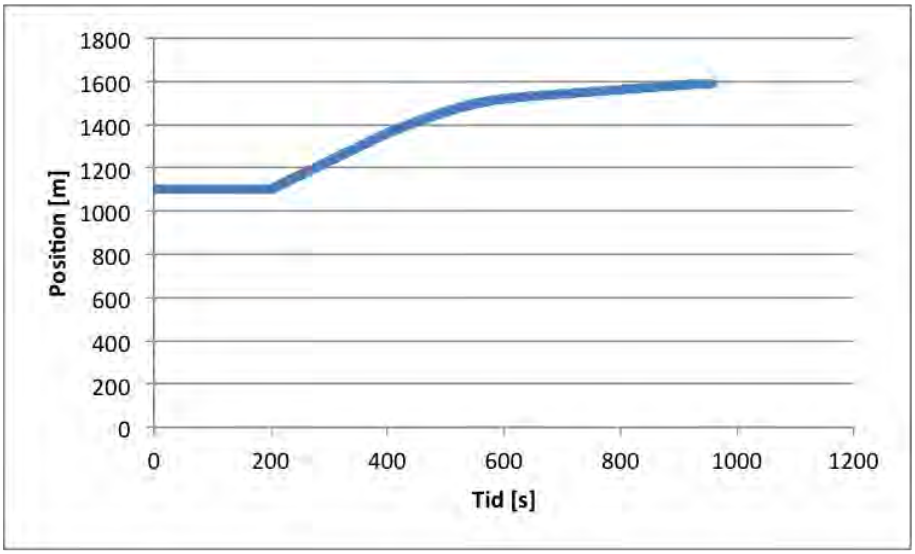
Scenariot som simuleras skulle kunna vara en individ som 400 m nedströms brandkällan sitter fast i en bilkö. Inledningsvis är individen ovetandes om den brand som inträffat 400 m bakom denne. Efter 2,5 min väljer individen att öppna dörren till sitt fordon och titta sig omkring för att de runtomkring har börjat göra likadant. Efter ytterligare 20 s blir individen varse den rök som närmar sig bilen. Ytterligare 30 s ägnas åt förberedande aktiviteter innan individen bestämmer sig för att utrymma tunneln. Utrymning påbörjas således i samband med att röken når bilen (p g a de rådande vindförhållandena). Med anledning av rådande omständigheter ser inte individen de utrymningsvägar som finns installerade med 150 m mellanrum utan tar istället sikte på tunnelportalen. I simuleringen beräknas gånghastigheten deterministiskt enligt Tabell 9.

Tabell 21. Typen av utrymningsscenario; i detta fall en ensam individ med startpositionen *x = 1100* m, d v s 400 m nedströms branden för det givna tunnelbrandscenariot.

Parameter	Enhet	Indata
Startposition	m	1100
Varseblivningstid	s	170
Förberedelsetid	s	30
Tid innan individ lämnar fordon	s	150
Utrymning via nödutgång	true/false	false
Deterministisk beräkning av gånghastighet	true/false	true

5.3.3.1 Individens position i tunneln

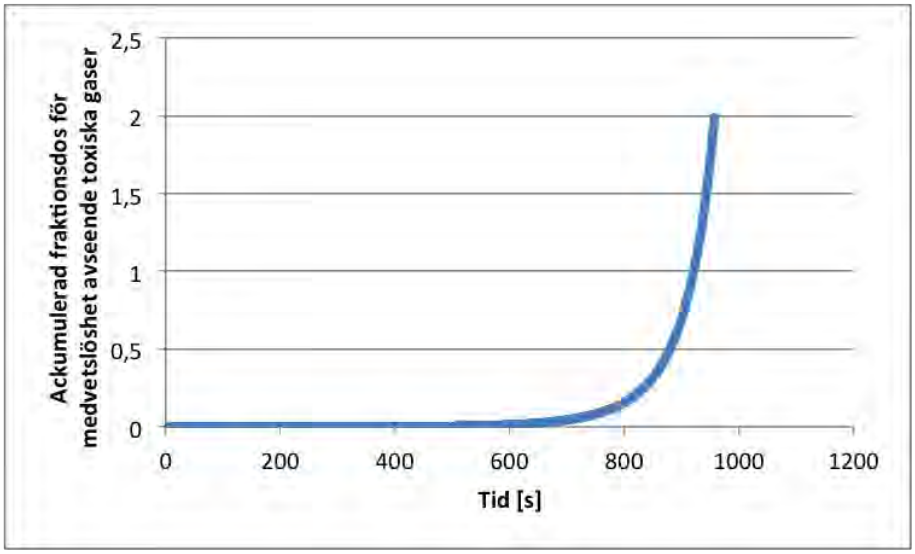
I simuleringsläget ”utrymning” lagras samtliga simulerades individer position *x* m i tunneln som en funktion av tiden *t* s i brandförloppet. Positionen beror naturligtvis på med vilken hastighet individen förflyttar sig, något som påverkas av de rådande siktförhållandena där individen befinner sig. Genom att positionen för varje individ sparas är det också lätt att illustrera denna i ett diagram som en funktion av tiden för respektive individ, se t ex Figur 15.



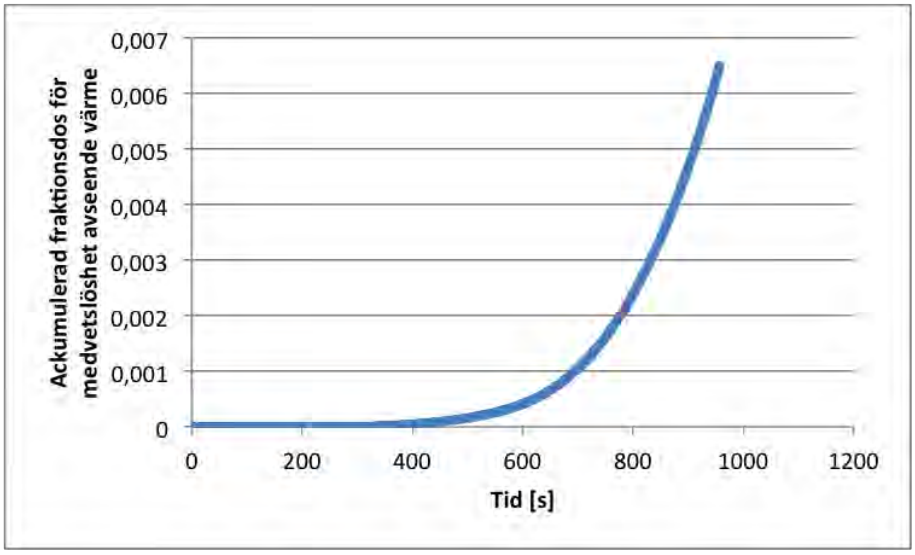
Figur 15. Individens position i tunneln presenterad som en funktion av tiden i brandförloppet. Som kan ses avtar i detta exempel individens gånghastighet ju längre in i brandförloppet beräkningen sker. Förklaringen är att branden under utrymningsförloppet hela tiden tillväxer och att siktförhållandena blir sämre och sämre (se Figur 19).

5.3.3.2 Fraktionsdoser

I simuleringsläget ”utrymning” lagras också samtliga individers ackumulerade fraktionsdoser avseende både toxicitet och värme som en funktion av tiden *t* s i brandförloppet. De ackumulerande fraktionsdoserna beror bl a på brandscenariot, d v s hur stor branden är och hur mycket toxiska gaser som genereras, men även på personens position i tunneln. Genom att de ackumulerade fraktionsdoserna för medvetslöshet sparas är det också lätt att illustrera dem i ett diagram som en funktion av tiden för respektive individ, se t ex Figur 16 och Figur 17.



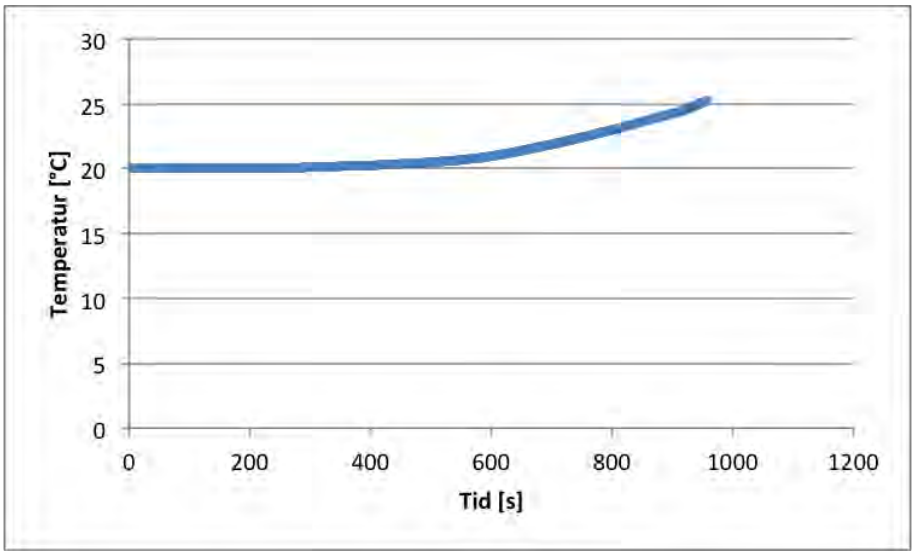
Figur 16. Individens ackumulerade fraktionsdos för medvetslöshet avseende toxiska gaser presenterad som en funktion av tiden i brandförloppet. Dosen byggs upp mycket snabbt på slutet i detta exempel, vilket bl a kan förklaras av den kvadratiskt tillväxande branden. Medvetslöshet uppstår efter drygt 15 min och död efter knappt 16 min.



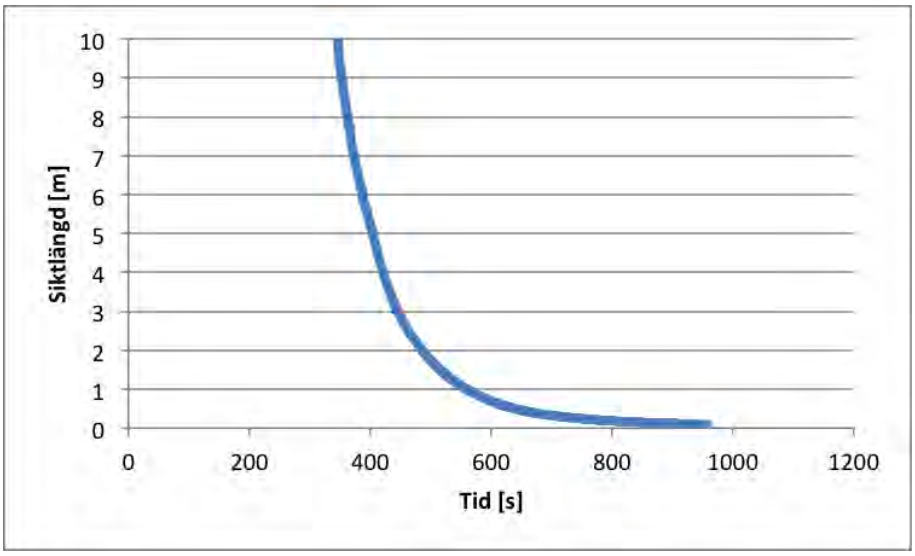
Figur 17. Individens ackumulerade fraktionsdos för medvetslöshet avseende värme presenterad som en funktion av tiden i brandförloppet. Precis som för toxiska gaser byggs dosen upp mycket snabbt på slutet i detta exempel. I förhållande till toxiska gaser är temperaturpåverkan dock inte av sådan storlek att individen påverkas nämnvärt (se även Figur 18).

5.3.3.3 Omslutande temperatur och siktlängd

I simuleringsläget ”utrymning” lagras en hel del textfiler med information om bl a vilken koncentration av olika brandgaser som omgivit individerna under utrymningsförloppet, liksom motsvarande information om omgivande temperatur och siktlängd. Två exempel illustreras nedan, se Figur 18 och Figur 19, där temperaturen och siktlängden i den position x m där individen befunnit sig vid tiden t s i brandförloppet redovisas.



Figur 18. Den omgivande brandgastemperaturen för individen presenterad som en funktion av tiden i brandförloppet. Som kan ses är temperaturpåverkan mycket liten, vilket också förklarar den låga ackumulerade fraktionsdosen som presenterades i Figur 17.



Figur 19. Den omgivande siktlängden för individen presenterad som en funktion av tiden i brandförloppet. Som kan ses är blir sikten väldigt dålig efter ca 6,5 min, vilket får som konsekvens att individens gånghastighet kraftigt reduceras (som vidare illustreras i Figur 15).

5.4 Räddningsinsats

Simuleringsläget ”räddningsinsats” möjliggör bedömning av en tunnelbrands påverkan på räddningstjänstens insatsförmåga. TuFT har begränsats till att endast simulera rörelse mot brandkällan, vilket bl a innebär att varken livräddande insats eller släckförmåga simuleras. Precis som i utrymningssimuleringsläget sker beräkningar endimensionellt, både för väg- och järnvägstunnlar. Beräkningstekniskt innebär det att TuFT simulerar räddningsinsatsen i tunneln baserat på en start- och en slutposition, där startpositionen definieras av användaren och slutpositionen alltid är brandkällan. Startpositionen kan antingen vara en av tunnelportalerna eller den utrymningsväg som ligger närmast brandkällan, och bestäms baserat på om insatsen genomförs upp- eller nedströms branden och om den sker via en tunnelportal eller en utrymningsväg.

För varje tidssteg (d v s varje sekund av brandförloppet) beräknas bl a brandgastemperatur och siktförhållanden för att kunna bedöma dels påverkan på aktionstiden (d v s den tid som en rökdykare kan vistas i tunnelmiljön) och dels påverkan på gånghastigheten (d v s med vilken hastighet som räddningsinsatsen rör sig mot branden). Som i utrymningssimuleringsläget varierar inte bara tiden t s i brandförloppet, utan också positionen x m p g a att insatsen rör sig mot brandkällan.

Vid bedömningen av räddningstjänstens insatsförmåga utgår TuFT alltid från att brandmännen jobbar i par om två personer. Hur länge rökdykarparet kan bedriva en insats i tunneln innan de behöver ersättas av nästa par beror dels på deras aktionstid som bl a styrs av storleken på deras luftpaket, men också på temperaturen i tunneln och ev infallande strålning från brandkällan (om avståndet till brandkällan är < 50 m). För bedömningen används ett koncept som påminner om det s k fraktionsdoskoncept som används i TuFT:s utrymningssimuleringsläge. Principen bygger på att det finns en koppling mellan rökdykarens aktionstid och den påverkan som rökdykarna upplever från branden avseende värme och strålning.

För detta ändamål används en uppvärmningsmodell som beskrivs i Ekvation 23. Modellen bygger på en enkel värmebalans för kroppen som finns närmare beskriven av Ingason, Bergqvist, Lönnermark, Frantzich, and Hasselrot (2005, p. 77). Antaganden om värden på ingående variabler presenteras ovan i nomenklaturen. I TuFT görs för varje tidssteg en beräkning av de konsekvenser som den omgivande värmen samt den infallande strålningen får för insatsen m h a denna modell. Beräkning av strålning sker enligt Ekvation 13.

$$\Delta T = \frac{f A_{body}\left(\dot{q}_{rad,fire} \cdot 10^3 \cdot f_{eff} \frac{R_a}{R_c} + \frac{T_{avg} - T_{body}}{R_c} + \frac{M}{f} - \frac{p_s - p_a}{R_e}\right)}{m c_p \Delta t} \qquad \text{Ekvation 23}$$

Ett rökdykarpar anses konsumerat när något av följande kriterier inträffar:

- Aktionstiden är slut p g a att all luft förbrukats
- Kroppstemperaturen har ökat med 2,5 °C
- Den infallande strålningen från brandkällan är > 5 kW/m^2

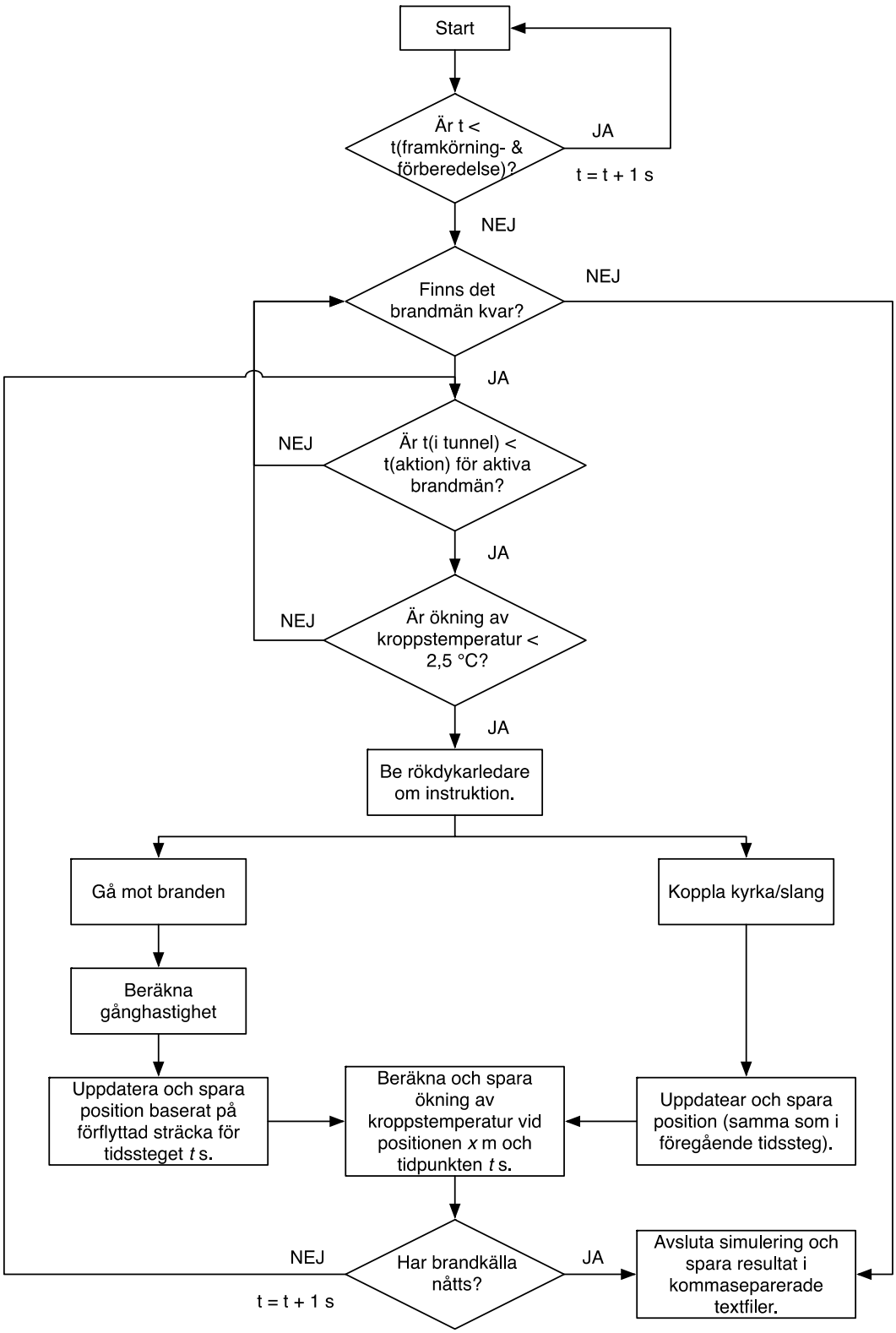
Då tar nästa rökdykarpar vid och fortsätter räddningsinsatsen till dess att också det är konsumerat. Simuleringen pågår till det att samtliga rökdykare konsumerats (eller till dess att brandförloppet är över, beroende på vilket som inträffar först).

Principiellt kan beräkningsprincipen i varje tidssteg beskrivas enligt Figur 20. Det bör i sammanhanget noteras att detta är en grov förenkling av hur räddningsinsatser bedrivs i verkligheten, och många moment har medvetet utelämnats för att underlätta beräkningen. Principen bygger på en förenklad version av den taktik och metodik som använts vid försök i andra delar av det aktuella forskningsprojektet. Det innebär att räddningsinsatsen för att nå branden i huvudsak inriktas på två moment: (1) att röra sig mot brandkällan samtidigt som slang- och annan släckutrustning transporteras, och; (2) att koppla slang. TuFT förutsätter att räddningsinsatsen alltid inleds med att det första rökdykarparet transporterar sig en sträcka som motsvarar längden för slangtyp 1, varpå tid måste ägnas åt

att koppla kopplingstyp 1 (t ex en kyrka). När kopplingen är slutförd och en ny slang ansluten fortsätter rökdykarparet sin insats mot brandkällan, nu en sträcka som motsvarar längden på slangtyp 2. När den slangen är utdragen måste ytterligare tid ägnas åt att koppla ihop slangen med nästa. Metodiken upprepas till dess att brandkällan nås (avståndet från brandkällan ≤ 5 m), eller till dess att samtliga brandmän konsumerats i enlighet med ovan villkor. I praktiken innebär det dock att det senare alltid kommer att inträffa p g a strålningen nära brandkällan sannolikt i de flesta fall kommer att överstiga 5 kW/m^2, vilket innebär att ett rökdykarpar som går in i insatsen nära branden konsumeras omgående. Detta måste beaktas i tolkningen av de simuleringsresultat som levereras av TuFT. Det antas att det alltid finns tillräckligt med luft så att ett rökdykarpar kan återvända till utgångspunkten.

Hur TuFT vet vilken aktivitet som det aktuella rökdykarparet ska ägna sig åt syns inte i den principiella beskrivningen i Figur 20. I momentet ”ber rökdykarledare om instruktion” görs dock en kontroll dels av hur lång sträcka rökdykarparet tillryggalagt och dels hur lång tid de ägnat sig åt aktiviteten. När, t ex, ett rökdykarpar förflyttat sig den sträcka som motsvaras av aktuell slanglängd ”meddelar” rökdykarledaren att det är dags att koppla slang. Aktiviteten bedrivs till dess att rökdykarledaren ånyo meddelar om nytt moment, då p g a att tiden motsvarande den tid det tar att koppla slang passerats.

Under pågående simulering sparas resultatet, precis som i övriga simuleringslägen, temporärt i vektorer som vid simuleringens slut exporteras till kommaseparerade textfiler för bearbetning och visualisering i t ex Microsoft Excel.



Figur 20. Den beräkningstekniska principen för simuleringsläget "räddningsinsats". Indata till simuleringen är information om tunneln, branden, tunnelbrandscenariot och de insatspersoner som finns tillgängliga.

5.4.1 Utdata

De kommaseparerade textfiler som genereras efter utförd simulering är sammanställda i Tabell 22 tillsammans med en kort beskrivning.

Tabell 22. Beskrivning av de kommaseparerade textfiler som genereras vid en simulering i simuleringsläge "räddningsinsats".

Namn	Beskrivning
operationSummary.txt	Innehåller sammanfattad information om simuleringen, bl a tunnelns längd, brandens position i tunneln, antal rökdykarpar som engagerades i insatsen, deras respektive aktionstider, start- och slutpositioner, tillryggalagd sträcka samt tiden de spenderade i tunneln innan de var tvungna att avbryta insatsen.
operationPosition.txt	Innehåller information om räddningsinsatsens position x m i tunneln under insatsen. Kolumn 1 redovisar tiden t s i brandförloppet, kolumn 2 den motsvarande positionen x m i tunneln för hela räddningsinsatsen och kolumn 3 den motsvarande tillryggalagda sträckan för hela räddningsinsatsen.

5.4.2 Exempel på utdata

Exempel på utdata vid en räddningsinsatssimulering presenteras nedan. Innehållet består av bearbetad data baserad på de kommaseparerade textfiler som TuFT genererar vid en simulering. Exemplet bygger på samma tunnelbrandscenario som beskrivits i Tabell 16-Tabell 19, med följande tillägg som är specificerade i Tabell 23. Eftersom att simulering av räddningsinsats nu sker aktiveras denna modul i Tabell 15 genom att *false* ändras till *true*.

Scenariot som simuleras skulle kunna illustreras av en räddningsinsats i en storstadsregion. Framkörnings- och förberedelsetid antas uppgå till totalt 15 min. Insatsen genomförs nedströms branden p g a svårigheter att tränga in via den rökfria tunnelporten. Totalt finns 6 disponibla brandmän som kan rökdyka (rökdykarledare som leder insatsen och skyddsgrupper anges inte). Rökdykarna saknar IR-utrustning och stora luftpaket vilket gör att varje rökdykarpar antas få en aktionstid motsvarande maximalt 25 min för det första rökdykarparet och 22,5 min för de efterföljande. I brandbilarna finns färdigkopplade slangpaket om 50 m, och kortare slangar om halva längden. Antagandet görs att det tar dem 5 min (300 s) att koppla en kyrka (kopplingstyp 1) och 1 min (60 s) att koppla ihop två slangar (kopplingstyp 2) (valda värden ligger i linje med vad som var fallet vid genomförda försök i andra delar av projektet).

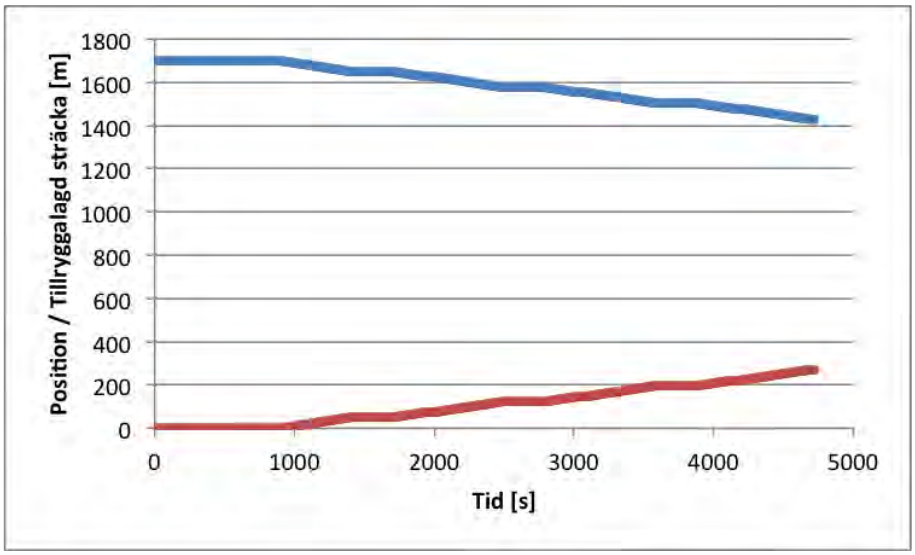
Tabell 23. Typen av insatsscenario; i detta fall en räddningsinsats som påbörjas nedströms branden via en av tunnelportalerna. I detta fall den tunnelportal som i TuFT lokaliseras vid $x = 1700$ m.

Variabel	Enhet	Indata
Framkörnings- och förberedelsetid	min	15
Insats uppströms	true/false	false
Insats via tunnelportal	true/false	true
Antal brandmän	st	6
IR-kamera	true/false	false
Stora luftpaket	true/false	false
Längd på slangtyp 1	m	50,0
Längd på slangtyp 2	m	25,0
Tid för kopplingstyp 1	s	300
Tid för kopplingstyp 2	s	60

5.4.2.1 Räddningsinsatsens position i tunneln

I simuleringsläget "räddningsinsats" lagras hela insatsens position x m i tunneln som en funktion av tiden t s i brandförloppet (blå linje), liksom den tillryggalagda sträckan (röd linje) som avverkats när rökdykarna närmat sig brandkällan. Information om varje rökdykarpar sparas således inte mer än i en sammanfattande text (se nedan).

Positionen i tunneln beror precis som i utrymningssimuleringsläget på med vilken hastighet som rökdykarna förflyttar sig, något som påverkas av de rådande siktförhållanden där det aktuella rökdykarparet befinner sig och huruvida de har IR-kameror med sig. Genom att positionen för räddningsinsatsen sparas är det också lätt att illustrera denna i ett diagram som en funktion av tiden, se t ex Figur 21.



Figur 21. Räddningsinsatsens position i tunneln presenterad som en funktion av tiden i brandförloppet. I exemplet ägnas 15 min åt framkörning och förberedelser. Därefter påbörjar det första rökdykarparet sin insats. Stoppen illustreras av horisontella avbrott i tunneln och beror på koppling av kyrkor och slangar innan insatsen kan röra sig framåt igen.

Som kan ses i exemplet rör sig räddningsinsatsen väldigt långsamt framåt p g a de dåliga siktförhållandena och avsaknaden av IR-utrustning. I detta fall nås inte branden innan brandförloppet är över.

5.4.2.2 Sammanfattning

I simuleringsläget ”räddningsinsats” sparas också en sammanfattning av insatsen. Denna kan användas som ett komplement till t ex de resultat som redovisades ovan för att utläsa när och varför ett visst rökdykarpar fick bytas ut. Sammanfattningen redovisas i en kommaseparerad textfil och ser för exemplet ut som följer:

TUNNEL LENGTH:	1700.0 m		
POSITION OF FIRE:	700		
FIRE FIGHTER PAIR [no]	1	2	3
ACTION TIME [s]	1500	1275	1275
START POS [m]	1700	1586	1502,5
END POS [m]	1586	1502,5	1428,5
DISTANCE MOVED [m]	114	83,5	74
TIME IN TUNNEL [s]	1500	1275	1049

Här kan ses att temperaturen inte dimensionerat något av rökdykarparens aktionstid p g a de hela tiden befinner sig väldigt långt ifrån brandkällan. Simuleringen avbryts innan sista rökdykarparet avslutat sin räddningsinsats p g a att brandförloppet är över (så som det specificerats i detta exempel).

6 Referenser

Bergqvist, A., Frantzich, H., Hasselrot, K., & Ingason, H. (2001). Räddningsinsatser vid tunnelbränder: Probleminventering och miljöbeskrivning vid brand i spårtunnel. Karlstad, Sweden: Räddningsverket.

Carvel, R., & Marlair, G. (2011). A history of fire incidents in tunnels. In A. Beard & R. Carvel (Eds.), *Handbook of Tunnel Fire Safety* (Second ed., pp. 3-23). London, UK: ICE Publishing.

Fridolf, K., Andrée, K., Nilsson, D., & Frantzich, H. (2013). The Impact of Smoke on Walking Speed. *Fire and Materials, 38*(7), 744-759. doi: 10.1002/fam.2217

Fridolf, K., Nilsson, D., & Frantzich, H. (2014). The Flow Rate of People during Train Evacuation in Rail Tunnels: Effects of Different Train Exit Configurations. *Safety Science, 62*(C), 515-529. doi: 10.1016/j.ssci.2013.10.008

Fridolf, K., & Wahlqvist, J. (2014). Predictive Capabilities of Computer Models for Simulation of Tunnel Fires. Lund: Lund University.

Ingason, H. (2005). Fire Development in Large Tunnel Fires. *Fire Safety Science 8*, 1497-1508. doi: 10.3801/IAFSS.FSS.8-1497

Ingason, H. (2006). *Modelling of Real World Fire Data.* Paper presented at the 2nd International Symposium on Tunnel Safety & Security (ISTSS), Madrid, Spain.

Ingason, H. (2009). Design fire curves for tunnels. *Fire Safety Journal, 44*(2), 259–265. doi: 10.1016/j.firesaf.2008.06.009

Ingason, H. (2012). Fire dynamics in tunnels. In A. Beard & R. Carvel (Eds.), *Handbook of Tunnel Fire Safety* (Second ed., pp. 273-307). London, UK: ICE Publishing.

Ingason, H., Bergqvist, A., Lönnermark, A., Frantzich, H., & Hasselrot, K. (2005). Räddningsinsatser i vägtunnlar. Karlstad, Sweden: Räddningsverket.

Jin, T. (1978). Visibility through Fire Smoke. *Journal of Fire & Flammability, 9*, 135-157.

Jin, T. (2008). Visibility and Human Behavior in Fire Smoke. In P. J. DiNenno (Ed.), *The SFPE Handbook of Fire Protection Engineering* (Fourth ed.). Quincy, Massachusetts: National Fire Protection Association.

Karlsson, B., & Quintiere, J. G. (2000). *Enclosure Fire Dynamics.* Boca Raton, USA: CRC Press LLC.

Palm, A., Kumm, M., & Ingason, H. (2014) Report from the tunnel rescue operations at Tistbrottet 2014. To be published.

Purser, D. (2008). Assessment of Hazards to Occupants from Smoke, Toxic Gases, and Heat. In P. J. DiNenno (Ed.), *The SFPE Hanbook of Fire Protection Engineering* (Fourth ed., pp. 2-96 - 92-193). Quincy, USA: National Fire Protection Association.

Tewarson, A. (2008). Generation of Heat and Gaseous, Liquid, and Solid Products in Fires. In P. DiNenno (Ed.), *The SFPE Handbook of Fire Protection Engineering* (pp. 3-109 - 103-194). Quincy, Massachusetts: National Fire Protection Agency.

Bilaga A: Användarmanual

Programmet består i princip av två filer; en körbar beräkningsfil: TuFT.jar samt en indatafil: settings.JSON. Som tillägg till dessa finns ytterligare ett antal exempelfiler med indata för några olika beräkningsfall i en separat mapp. En Excelfil finns också för att enklare kunna hantera resultaten och skapa diagram av dessa. Excelfilen innehåller en procedur som importerar data och skapar diagrammen.

Vid en simulering ska beräkningsfilen TuFT.jar och den aktuella indatafilen settings.JSON ligga i en gemensam mapp som t ex kan döpas till TuFT. Namnet är valfritt men de två filerna måste ligga i samma mapp. TuFT-mappen kan placeras på valfri plats på datorns hårddisk. Mappen med exempelfilerna samt Excelfilen för resultathanteringen kan läggas på valfri plats, men bör inte ligga i samma mapp som TuFT.jar och settings.JSON.

A.1 Användning

En typisk simulering innebär att användaren skriver en indatafil i valfri texteditor, t ex Anteckningar på en PC eller Textredigerare på en Mac. Vanligast är dock att utgå från en befintlig indatafil som ändras till de aktuella förutsättningarna.

Indatafilen settings.JSON är en ren textfil men som måste vara utformad på ett sätt som programmet kan läsa. Det är viktigt att filen är utformad så som exempelfilerna (se nedan), dvs med blocken i rätt ordning och med rätt formatering. Decimalavskiljare ska vara punkt (.) och variabler som i exempelfilen är angivna med decimal ska vara det, och de som saknar decimal ska också skrivas utan decimalavskiljare. Detta eftersom decimalavskiljaren används för att definiera att variabeln är ett reellt tal och inte ett heltal som datorn hanterar olika.

När indatafilen är skapad dubbelklickar man på den körbara filen TuFT.jar som då läser in informationen i settings.JSON, utför beräkningen samt skapar en rad resultatfiler. Dessa resultatfiler kan öppnas i Excel och användas för att t ex rita diagram som illustrerar de olika beräkningsresultaten. Ett medföljande makroskript som körs i Excel underlättar denna hantering avsevärt då det automatiskt importerar och ritar diagram baserat på av användaren specificerade resultatfiler.

Notera att programmet läser in den settings-fil som ligger i samma mapp som den körbara filen. I samband med en simulering kommer även nya resultatfiler att skapas vilka har samma namn som resultatfiler från en tidigare simulering vilket innebär att tidigare resultat skrivs över. Detta är ett i sammanhanget litet problem eftersom att en simulering kan göras mycket snabbt för att återskapa resultat med andra förutsättningar. För att undvika att tidigare resultat skrivs över bör dessa filer flyttas innan en ny simulering startas.

En simulering kan sammanfattas i följande steg:

1. Förberedelser av indatafilen settings.JSON med de korrekta beräkningsförutsättningarna.

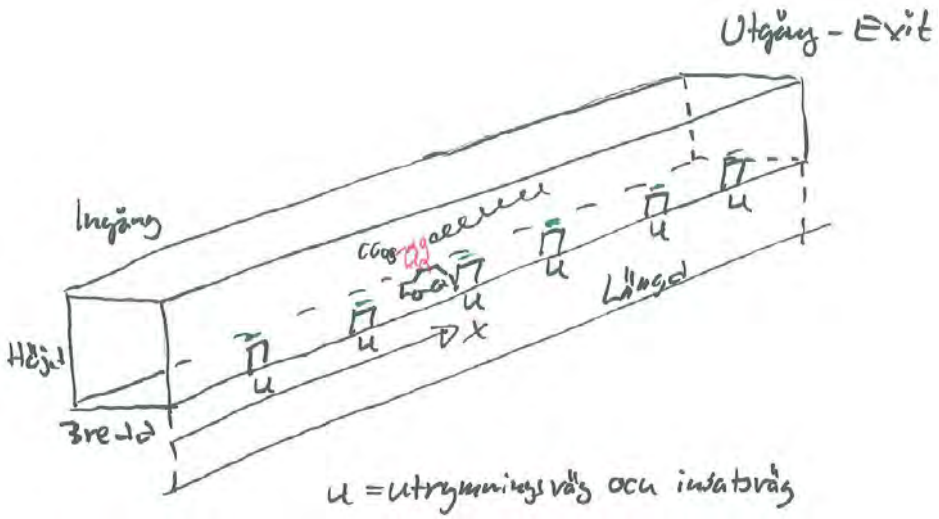
2. Körning av TuFT.jar.

3. Analys av resultaten t ex med hjälp av Excel eller annat analysprogram.

A.2 Grundförutsättningar

Programmet kan användas för att simulera en utrymning och en räddningsinsats från antingen en väg- eller en järnvägstunnel. I tunneln antas en brand uppstå på en plats som anges av användaren. I samtliga fall förutsätts att tunneln är horisontell och kan beskrivas med längd, bredd och höjd enligt Figur 22.

Positioner i tunneln för t ex branden, bilister som vistas i tunneln eller ett stillastående tåg anges med utgångspunkt från x-koordinaten x = 0. Undantaget är att beräkning av temperatur, koncentration av giftiga gaser och sikt som sker på platser som utgår från brandens placering. Detta beskrivs mer i detalj under blocket ”measurementPositions”, se nedan.



Figur 22. Beskrivning av tunneln och dess förutsättningar.

Både utrymning och räddningsinsats kan ske genom tunnelns mynningar eller genom utrymningsvägar i tunneln. Vilka som ska användas anges som en beräkningsförutsättning i indatafilen. Utrymning sker alltid i riktning från branden, och en räddningsinsats sker alltid till branden. Räddningsinsatsen kan ske från valfritt håll dvs både uppströms och nedströms branden, dock inte samtidigt. Räddningsinsatsen kan ske antingen genom en mynning eller genom närmaste utrymningsväg. Vilket som ska vara aktuellt anges i indatafilen.

A.3 Programmets indata-fil

Programmets indatafil (settings.JSON) är uppdelad i block och beroende på vad som ska simuleras så ska olika block inkluderas i filen. Vissa block är gemensamma för alla typer av simuleringar men beroende på om det är en vägtunnel eller en järnvägstunnel som ska simuleras så ser indatafilen lite olika ut. nedan redovisas indatafiler som är typiska för vägtunnel respektive järnvägstunnel.

I följande avsnitt beskrivs de olika blocken och vad variablerna betyder.

A.3.1 Allmänna förutsättningar

Första blocket anger vad som ska ingå i beräkningen.

```
"simulation":{
      "measure":"true",
      "evacuation":"true",
      "operation":"true",
      },
```

I ovanstående kommer beräkning att ske av brandmiljön (measure), utrymningsförloppet (evacuation) samt räddningsinsatsen (operation). Ändra till ”false” om något ej ska beräknas.

A.3.2 Tunnelns utformning

Blocket anger hur den fysiska tunneln ser ut, aktuell vindhastighet, tunneltemperatur och hur långt det är mellan utrymningsvägarna.

```
"tunnel":{
      "tunnelType":"road",
      "length":700.0,
      "width":9.0,
      "height":6.0,
      "windSpeed":2.0,
      "ambientTemperature":20.0,
      "distanceBetweenEmergencyExits":150.0,
      },
```

Första raden definierar vilken typ av tunnel som ska analyseras. Alternativen är ”road” och ”rail”. Valet av tunnel styr sedan vilka av nedanstående block som blir aktuella, d v s främst hur förutsättningarna för utrymningsförloppet ska definieras. I vägtunneln anges personerna på ett sätt som ska symbolisera bilister i bilar. I tågtunneln sker utrymningen till tunneln genom ett antal fördefinierade utgångar i tåget.

Övriga variabler används för att beskriva tunnelns dimensioner, aktuell vindhastighet (riktningen anges senare), tunnelns grundtemperatur i °C samt avståndet mellan ev utrymningsvägar. I fallet ovan kommer första utrymningsvägen 150 m in i tunneln från x = 0 och sedan var 150:e meter. Utrymningsvägarna kan även användas för räddningsinsats, vilket anges i förekommande fall senare.

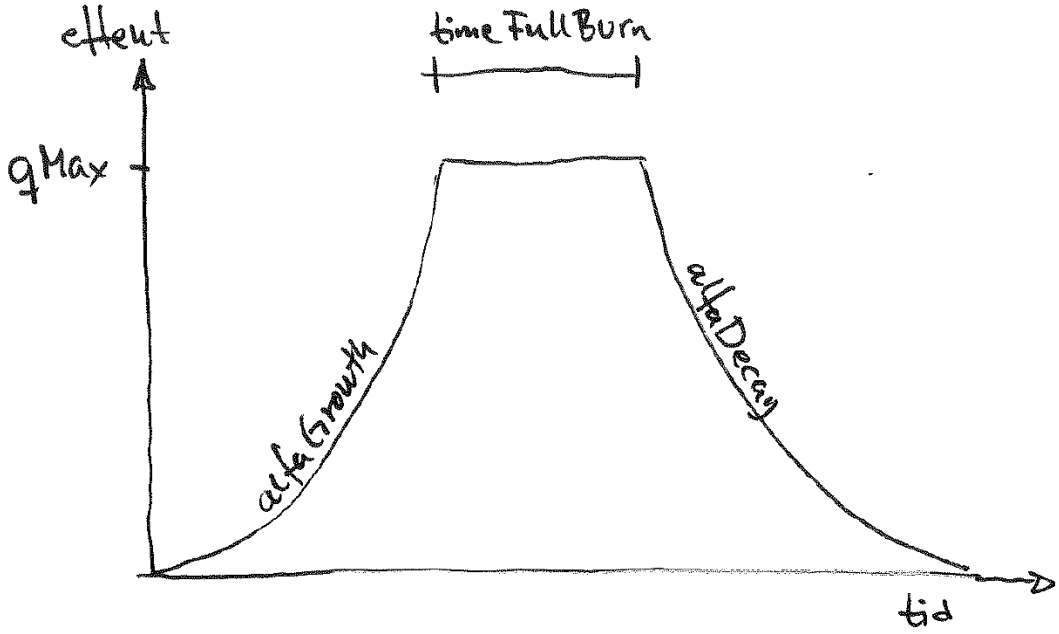
A.3.3 Brandförloppet

I detta block beskrivs det aktuella brandförloppet. Olika brandförlopp kan simuleras.

```
"fire":{
      "fireType":"squared",
      "alfaGrowth":0.19,
      "alfaDecay":0.19,
      "qMax":60000.0,
      "timeFullBurn":90,
      "massOpticalDensity":304.0,
      "heatOfCombustion":15500.0,
      "chi":1.0,
      "yieldCo2":1.50,
      "yieldCo":0.08,
      "yieldHcn":0.0,
      },
```

Första variabeln anger vilken typ av brandförlopp och styr hur av efterföljande variabler ska definieras. Tre typer av bränder kan anges ”squared”, ”linear” och ”exponential”. De två ”alfavärdena” anger hur branden tillväxer respektive avtar och hur siffrorna ska anges beror på vilken typ av brandförlopp som definierats, se avsnitt 3.3 i den tekniska dokumentationen. Variabeln qMax är maximal HRR och anges i kW. De övriga variablerna beskriver bl a hur stor rökproduktionen är, förbränningseffektivitet, m m, se vidare avsnitt 3.3 i den tekniska dokumentationen för vägledning. Notera att variabeln timeFullBurn ska anges i minuter.

I de exempelfiler som presenteras nedan finns brandförlopp med några typiska effektutvecklingar redovisade. Dessa kan klippas in i den aktuella indatafilen istället för det befintliga blocket. Figur 23 visar schematiskt brandförloppet enligt ”squared”.



Figur 23. Brandförlopp med αt^2-utveckling.

A.3.4 Brandplatsen

I detta block anges förutom platsen för branden även åt vilket håll luftströmmen går i den aktuella tunneln.

```
"fireEvent":{
      "firePos":300.0,
      "windDirExit":true,
      },
```

Avståndet 300 utgår från x = 0 och när windDirExit anges som ”true” är vindriktningen mot tunnelns utgång, d v s bort från ingångsmynningen (åt höger i Figur 22). Omvänd vindriktning får man genom att ange ”false” istället.

A.3.5 Personer som utrymmer tunneln

Beroende på vilken typ av tunnel som ska simuleras så definieras personerna lite olika. För en vägtunnel används blocken ”individuals” och ”groups” medan för en tågtunnel ska personerna definieras med blocket ”train”. Det är viktigt att rätt block används och programmet ger fel resultat om t ex blocket ”train” finns i en simulering för vägtunnel.

A.3.5.1 Individer

För en vägtunnel kan personerna i tunneln antingen anges som en eller flera individer som placeras ut eller i form av en eller flera grupper som slumpmässigt låter placera ut individer inom givna intervall. Blocket nedan anger egenskaperna för två enskilt utplacerad individer. Om flera enskilt utplacerade individer önskas kan fler block anges efter varandra.

```
"individuals":[
      {
      "personPosition":350.0,
      "recognitionTime":120,
      "responseTime":120,
      "timeToLeaveVehicle":90,
      "chooseEmergencyExit":false,
      "deterministic":true,
      },
{
      "personPosition":320.0,
      "recognitionTime":120,
      "responseTime":120,
      "timeToLeaveVehicle":90,
      "chooseEmergencyExit":false,
      "deterministic":true,
      },

      ],
```

Observera att blocket eller blocken måste omslutas av hakparenteser, [och]. I exemplet ovan omsluts de två individ-blocken av hakparenteserna.

Även om simuleringen för en vägtunnel inte inkluderar några enstaka individer (inlagda med blocket ”individuals”) så måste blockets rubrik och hakparenteserna finnas med i textfilen. Låt i så fall utrymmet vara tomt mellan dem.

Personen placeras ut på ett avstånd personPosition från x = 0 och har en fördröjningstid innan personen börjar utrymma som är summan av recognitionTime och responseTime. Dessutom anges när personen förväntas lämna sin bil. Under tiden personen vistas i bilen antas att personen inte utsätts för brandens påverkan utan det sker först när personen kommit ut ur bilen.

Det går att välja om personen förväntas gå till närmaste utrymningsväg eller utrymma genom någon av tunnelns mynningar. Det styrs genom variabeln chooseEmergencyExit. Om denna är ”true” går personen mot en utrymningsväg. Alternativet är ”false”. Alla personer utrymmer alltid bort från branden.

Personens gånghastighet i röken kan beskrivas på olika sätt. Antingen väljs hastigheten som ett bestämt värde beroende på röktätheten, vilket innebär ett värde på säkra sidan. Detta sätt att beskriva hastigheten erhålls om deterministic är ”true”. Alternativet är att gånghastigheten väljs slumpmässigt inom ett intervall, baserat på den rådande röktätheten. I sådant fall ska deterministic vara ”false”. Se den tekniska dokumentationen för mer detaljer kring valet av gånghastighet i rök.

A.3.5.2 Grupper

I vägtunnlar kan personer även anges som en grupp av individer. Då används blocket ”groups”. I det fallet placeras ett antal individer slumpmässigt ut inom givna intervall. Om flera grupper ska läggas in kan fler ”groups”-block läggas efter varandra.

```
"groups":[
      {
      "count":3,
      "personPositionIntervalFrom":350.0,
      "personPositionIntervalTo":360.0,
      "recognitionTimeIntervalFrom":90,
      "recognitionTimeIntervalTo":180,
      "responseTimeIntervalFrom":90,
      "responseTimeIntervalTo":180,
      "timeToLeaveVehicleFrom":90,
      "timeToLeaveVehicleTo":270,
      "chooseEmergencyExit":true,
      "deterministic":true,
      },

      ],
```

Observera att blocket eller blocken måste omslutas av hakparenteser, [och]. I exemplet ovan omsluts ett grupp-block av hakparenteserna.

Även om simuleringen för en vägtunnel inte inkluderar några grupper (inlagda med blocket ”groups”) så måste blockets rubrik och hakparenteserna finnas med i textfilen. Låt i så fall utrymmet vara tomt mellan dem.

Variabeln ”count” anger hur många individer som ingår i gruppen. I jämförelse med egenskaperna för en individ anges istället intervallens gränser för variablerna som är relevanta för individen. De sista två variablerna är identiska som för en enstaka individ.

A.3.5.3 Tåg

För järnvägstunnlar anges inte individerna som enstaka individer eller grupper. Dessa block ska således inte finnas i indatafilen. Istället finns ett tåg som utrymningen sker från. För att programmet ska förstå att blocket ”train” ska användas ska tunnelType ovan vara definierad som ”rail”. Tåget anges med variabeln ”position” som är den del av tåget som är närmast ingången till tunneln dvs vid x = 0. Sedan anges längden på tåget. I exemplet nedan står ett tåg mellan 300 m och 450 m från tunnelns ingångsmynning.

```
"train":{
      "position":300.0,
      "trainLength":150.0,
      "numberOfExits":18,
      "exitWidth":1.7,
      "numberOfPassengers":135,
      "recognitionTime":300,
      "responseTime":300,
      "deterministic":false,
      "chooseEmergencyExit":true,

      },
```

Antalet utgångar anges med variabeln "numberOfExits" och placeras ut på tågets ena sida. Varje utgångs bredd anges med "exitWidth" vilket påverkar flödet ut genom öppningen. Antalet passagerare på tåget fördelas lika på de respektive utgångarna i tåget och anges med variabeln "numberOfPassengers".

De sista fyra variablerna är lika som för individer eller grupper i vägtunnlar dvs reglerar fördröjningstiden innan utrymningen inleds, hur gånghastigheten för personerna som utrymmer genom rök ska beräknas samt om de kommer att utrymma genom utrymningsvägarna eller förflyttar sig mot tunnelns mynning.

A.3.6 Räddningsinsats

Förutsättningarna för räddningstjänstens inträngning i tunneln beskrivs i blocket ”operations”. Denna sker fristående från utrymningsförloppet och det finns ingen interaktion mellan utrymningen och räddningsinsatsen.

```
"operation":{
      "preparationTime":15,
      "upstream":false,
      "portal":true,
      "numberOfFireFighters":10,
      "thermalImaging":false,
      "muchAir":true,
      "hoseLengthUneven":50.0,
      "hoseLengthEven":25.0,
      "connectTimeUneven":300,
      "connectTimeEven":60,
      },
```

Grundförutsättningen för räddningsinsatsen är att den beskriver ett antal rökdykarpars gemensamma inträngning i en tunnel. Det betyder att de kan antas lägga ut slang (förflytta sig) eller koppla ihop utlagda slangar (stå stilla). Under insatsen kan de beroende på tidpunkten och vald inträngningsplats bli påverkade av brandgaser och värme. Det gör att exempelvis gånghastigheten för rökdykarna kan påverkas av sikten om de inte utrustats med IR-hjälpmedel. Rökdykare som använder IR-hjälpmedel för att kunna se genom röken går fortare än om de saknar IR-hjälpmedel.

Eftersom insatser i tunnlar vanligen är utdragna förlopp finns en möjlighet att låta rökdykarna ha extra luft, d v s större luftflaskor än normalt.

Insatsen pågår så länge det finns rökdykare kvar att sätta in. Programmet bryter inte simuleringen även om rökdykarpar kommer väldigt nära branden. I en verklig situation kan det förväntas att räddningspersonalen avbryter ett avancemang en sträcka innan branden men detta resonemang måste programmets användare själv föra eftersom förutsättningarna för en släckinsats kan variera. Ett tecken på att räddningspersonalen är mycket nära branden är när deras aktionstid är mycket kort.

I analysen förutsätts att det finns support-grupper som kan assistera de beräknade rökdykarnas insats. Eftersom insatssträckan kan vara lång är det inte rimligt att varje rökdykarpar bär med all sin egen utrustning utan det krävs fler grupper för att genomföra insatsen. Dessa stödjande grupper, så kallade skyddsgrupper och liknande, finns inte med i beräkningen utan antas fungera parallellt. Den grupp rökdykare som ingår i beräkningen är den eller de grupper som är längst fram och som definierar avancemanget in i tunneln.

För insatsen ska ett antal variabler anges. ”preparationTime” anger insatstiden i minuter. Det är den tid som förflyter från brandens start tills första rökdykare påbörjar sin insats. ”upstream” anges från vilket håll insatsen sker och om variabeln anges som ”false” betyder det att insatsen sker från nedströmssidan, d v s genom röken mot vindriktningen. Variabeln ”portal” anger varifrån insatsen sker, d v s från mynning eller från närmaste utrymningsväg. Vilken sida insatsen sker genom från branden räknat beror på hur variabeln ”upstream” definierats och påverkas även av hur "windDirExit" i blocket "fireEvent" angetts.

Variabeln ”numberOfFireFighters” anger det totala antalet rökdykare. I programmet antas att de arbetar i par vilket betyder att i exemplet ovan finns fem rökdykargrupper som kan användas för avancemang in i tunneln.

Gånghastigheten påverkas av siktförhållandena och om rökdykarna är försedda med IR-hjälpmedel kan en högre gånghastighet hållas även om sikten är dålig. Detta anges med variabeln ”thermalImaging” som kan vara ”true” (har IR-hjälpmedel) eller ”false”. Extra luftmängd regleras genom variabeln "muchAir" som gör det möjligt att välja 25 eller 40 minuters aktionstid.

En taktisk möjlighet är att förbereda rökdykarna med extra långa slangar redan från början. Det finns därför en möjlighet att låta dem arbeta med två olika slanglängder. Slangarna läggs ut växelvis; slang 1, slang 2, slang 3, o s v. Längderna för slang 1, slang 3, o s v, anges med variabeln ”hoseLengthUneven” medan slang 2, slang 4, o s v, anges med variabeln ”hoseLengthEven”. På samma sätt finns en möjlighet att variera hur lång tid det tar att koppla ihop slangar, första inkopplingen, andra inkopplingen, tredje inkopplingen, o s v. Dessa tider anges med variablerna "connectTimeUneven" respektive "connectTimeEven" för ihopkoppling av slangar första, andra, tredje, o s v gången, d v s för udda respektive jämna inkopplingsnummer. På det viset går det att simulera en kort tid som ska spegla tiden att koppla ihop två enskilda slangar och en lång tid som inkluderar ihopkoppling med en grenkoppling och efterföljande vattenfyllnad av slangsystemet fram till grenkopplingen.

A.3.7 Aktuell brandmiljö

Det finns en möjlighet att låta programmet beräkna och redovisavissa egenskaper i den aktuella miljön för angivna platser i tunneln som en funktion av tiden. Det görs genom att placera ut ett antal s k mätpunkter nedströms branden som kan registrera en rad miljövariabler.

```
"measurementPositions":[

      {
      "pos":75.0,
      },

      {
      "pos":150.0,
      },

      {
      "pos":300.0,
      },

],
```

Programmet kan beräkna sikt (i form av extinction coefficient), temperatur och molfraktionen av syre, koldioxid, kolmonoxid och vätecyanid för ett antal platser i tunneln. Dessa platser anges i det sista blocket "measurementPositions". Det bör noteras att de platser som ska anges utgår från brandens position och inte från tunnelns mynning. Det är alltid avstånd nedströms branden som förutsätts. Skälet är att dessa egenskaper inte kan beräknas uppströms branden.

A.4 Analys av resultat

Efter en genomförd simulering ska de genererade resultaten analyseras och lämpligen presenteras viss information som diagram. Utdatafilerna är kommaseparerade textfiler som enkelt kan läsas in i Excel. I den tekniska dokumentationen redovisas vilka utdatafiler som programmet genererar. Vissa av filerna är generella medan andra beror på vilka scenarier som undersöks, t ex om räddningsinsats simuleras eller ej eller om utrymning simuleras eller ej.

Lämpligen används den färdiga Excelfilen (makroskriptfilen) för att läsa in de simulerade resultaten och skapa diagram som presenterar resultaten på ett tydligt sätt.

Nedanstående text visar inledningen från den resultatfil (evacPosition.txt) som redovisar var personer befinner sig i varje tidssteg:

Output generated with TuFT beta 1.2. Anyone who uses the results produced by TuFT does so on his/her own responsibility.

```
TIME,PERSON 1,PERSON 2,PERSON 3,PERSON 4,PERSON 5
0,320.0,400.0,357.81695352369746,356.21517219778593,352.37351968082055
1,320.0,400.0,357.81695352369746,356.21517219778593,352.37351968082055
2,320.0,400.0,357.81695352369746,356.21517219778593,352.37351968082055
3,320.0,400.0,357.81695352369746,356.21517219778593,352.37351968082055
```

Simuleringen visar positionen för fem personer som inledningsvis befinner sig på följande avstånd från tunnelns ingångsmynning; 320 m, 400 m, 357,8 m, 356,2 m och 352,3 m. Exemplet visar de tre första sekunderna.

A.5 Felhantering

De fel som kan uppstå har med största säkerhet sitt ursprung i en felaktig indatafil. I flera fall visas inga felmeddelanden utan programmet räknar men presenterar fel svar. Dessa fall är svåra att identifiera. Mer allvarliga fel gör att programmet inte startar. Detta kan t ex bero på att en indatavariabel lagts in med decimal när det ska vara heltal eller tvärt om. Då visas detta med ett meddelande som Figur 24.

Figur 24. Felmeddelande orsakat av större fel i indatafilen.

A.6 Vanliga frågor

I detta kapitel redovisas svar på de vanligaste frågor som har dykt upp under utvecklingen av TuFT.

A.6.1 Utrymning

F: Kan jag ange att första dörrparet för utrymning av ett tåg ska användas av 30 % medan resten av passagerarna går till de övriga utgångarna?

S: Nej, alla passagerare fördelas lika på de tillgängliga utgångarna.

F: Kan jag simulera att dörrarna på tågets båda sidor används vid utrymningen.

S: Ja, det går men måste ske genom att man enbart betraktar utrymning från ena sidan tåget. Om tåget har 240 passagerare och alla dörrarna längs båda sidorna används ska antal passagerare för variabeln "numberOfPassengers" anges till 120. När konsekvenserna ska bedömas för utrymningen måste samtidigt alla värden dubbleras. Man kan se det som att utrymningssimuleringen speglas längs tågets mittlinje sedd i längdriktningen.

A.6.2 Brandförloppet

F: Kan jag ange min effektutvecklingskurva med datapunkter från ett verkligt försök?

S: Nej, enbart de specificerade effektutvecklingsmodellerna kan användas i nuläget.

A.6.3 Räddningsinsatsen

F: Varför utnyttjas inte alla rökdykarparen?

S: Det mest troliga är att brandförloppet är kortare än insatsens varaktighet. När branden slocknat kommer inte insatsen att fortsätta.

F: Hur ser jag när de olika rökdykarparen avlöser varandra?

S: Detta syns bara i utdatafilen operationSummary.txt. Resultatfilen operationPosition.txt anger den totala insatsens avancemang in i tunneln oavsett vilket rökdykarpar som är aktivt.

F: Kan jag simulera att rökdykarna använder en transportvagn för att snabbare lägga ut slangen?

S: Ja, enklast sker det genom att ange långa (redan ihopkopplade) slanglängder samt att eventuellt också ange korta tider för ihopkoppling av slanglängderna. Ytterligare aktionstid med hjälp av extra luftbehållare som kan tänkas finnas på vagnen kan inte simuleras utöver att använda de extra stora lufttuberna som ger 40 minuters aktionstid jämfört med normala 25 minuter.

A.7 Exempelfil 1: Utrymning och räddningsinsats i vägtunnel


```
{

	"simulation":{
		"measure":true,
		"evacuation":true,
		"operation":true,
		},

	"tunnel":{
		"tunnelType":"road",
		"length":1700.0,
		"width":9.0,
		"height":6.0,
		"windSpeed":2.0,
		"ambientTemperature":20.0,
		"distanceBetweenEmergencyExits":150.0,
		},

	"fire":{
		"fireType":"squared",
		"alfaGrowth":0.19,
		"alfaDecay":0.19,
		"qMax":60000.0,
		"timeFullBurn":60,
		"massOpticalDensity":304.0,
		"heatOfCombustion":15500.0,
		"chi":1.0,
		"yieldCo2":1.50,
		"yieldCo":0.027,
		"yieldHcn":0.01,
		},

	"fireEvent":{
		"firePos":700.0,
		"windDirExit":true,
		},

	"individuals":[
		{
		"personPosition":660.0,
		"recognitionTime":10,
		"responseTime":10,
		"timeToLeaveVehicle":10,
		"chooseEmergencyExit":false,
		"deterministic":true,
		},

		{
		"personPosition":740.0,
		"recognitionTime":40,
		"responseTime":10,
		"timeToLeaveVehicle":10,
		"chooseEmergencyExit":false,
		"deterministic":true,
		},

	],

	"groups":[
		{
		"count":3,
		"personPositionIntervalFrom":850.0,
		"personPositionIntervalTo":1050.0,
		"recognitionTimeIntervalFrom":90,
		"recognitionTimeIntervalTo":180,
		"responseTimeIntervalFrom":90,
		"responseTimeIntervalTo":180,
		"timeToLeaveVehicleFrom":90,
		"timeToLeaveVehicleTo":270,
		"chooseEmergencyExit":true,
		"deterministic":true,
		},

	],
```

```
	"operation":{
		"preparationTime":15,
		"upstream":false,
		"portal":true,
		"numberOfFireFighters":10,
		"thermalImaging":false,
		"muchAir":true,
		"hoseLengthUneven":50.0,
		"hoseLengthEven":25.0,
		"connectTimeUneven":300,
		"connectTimeEven":60,
		},

	"measurementPositions":[

		{
		"pos":25.0,
		},

		{
		"pos":300.0,
		},

		{
		"pos":800.0,
		},

	],

}
```


A.8 Exempelfil 2: Utrymning och räddningsinsats i järnvägstunnel


```
{

	"simulation":{
		"measure":true,
		"evacuation":true,
		"operation":true,
		},

	"tunnel":{
		"tunnelType":"rail",
		"length":1700.0,
		"width":9.0,
		"height":6.0,
		"windSpeed":2.0,
		"ambientTemperature":20.0,
		"distanceBetweenEmergencyExits":150.0,
		},

	"fire":{
		"fireType":"squared",
		"alfaGrowth":0.19,
		"alfaDecay":0.19,
		"qMax":60000.0,
		"timeFullBurn":60,
		"massOpticalDensity":304.0,
		"heatOfCombustion":15500.0,
		"chi":1.0,
		"yieldCo2":1.50,
		"yieldCo":0.027,
		"yieldHcn":0.01,
		},

	"fireEvent":{
		"firePos":700.0,
		"windDirExit":true,
		},

	"train":{
		"position":1300.0,
		"trainLength":150.0,
		"numberOfExits":18,
		"exitWidth":1.7,
```

```
        "numberOfPassengers":135,
        "recognitionTime":300,
        "responseTime":300,
        "deterministic":false,
        "chooseEmergencyExit":true,
        },

    "operation":{
        "preparationTime":15,
        "upstream":false,
        "portal":true,
        "numberOfFireFighters":10,
        "thermalImaging":false,
        "muchAir":true,
        "hoseLengthUneven":50.0,
        "hoseLengthEven":25.0,
        "connectTimeUneven":300,
        "connectTimeEven":60,
        },

    "measurementPositions":[

        {
        "pos":25.0,
        },

        {
        "pos":300.0,
        },

        {
        "pos":800.0,
        },

    ],

}
```

Bilaga B: Övningsuppgifter

I denna bilaga presenteras tre övningsuppgifter tillsammans med förslag på lösningar med syftet att öka förståelsen och användarens förmåga att använda TuFT.

B.1 Uppgift 1A: Enkel tunnelutrymning och rökdykningsinsats i vägtunnel

B.1.1 Utrymning

Beräkna total utrymningstid för en brand i en vägtunnel med tio personer som befinner sig nedströms branden. Tunneln är 500 m och branden uppstår 100 m från inkörsportalen. Branden uppstår i en buss som krockar, max 30 MW och fullt utvecklad brand under 90 minuter. Personerna i bussen hinner utrymma men de tio bilisterna sitter i en kö och måste ev utrymma genom röken. De befinner sig mellan 200 och 300 m nedströms branden och kan antas vara slumpmässigt placerade. Deras förberedelsetid är totalt mellan 180 och 240 sekunder. De sitter därefter kvar i sina bilar mellan 30 och 90 sekunder. Placera även två individer på 455 m resp 470 m in i tunneln som kontrollindivider (se så de går ut genom mynningen). Tunneln är 6 m bred och 5 m hög och vindhastigheten är 1,5 m/s. Avståndet är 150 m mellan utrymningsvägarna och dessa kan användas av dem som utrymmer. Temperaturen är 6 °C i tunneln.

B.1.2 Räddningsinsats

I tunneln ovan genomförs en rökdykningsinsats från nedströmshållet. Tre rökdykarpar finns till förfogande för själva avancemanget in i tunneln, utöver biträdande kollegor som utgör skyddsgrupper. Insatstiden är 15 minuter. De har enbart 25 m slangar (obegränsat antal) med sig och kan koppla ihop två sådana slangar innan de måste ansluta en grenkoppling och vattenfylla utlagd slang. De har traditionella luftpaket för andningsluft (20 min aktionstid). Hur långt in kommer de?

B.1.3 Lösningsförslag

Alla som utrymmer hinner ut och de tar utrymningsvägen som är vid x = 450 m, d v s 50 m innan tunnelns bortre mynning. Rökdykningsinsatsen sker genom en rökfylld tunnel men når inte hela vägen fram till branden. Första rökdykarparet kommer 98 m in i tunneln (x = 402 m), det andra slutar vid x = 330 m och den sista gruppen kommer till X = 268 m, d v s har ca 170 m kvar fram till branden.

B.1.3.1 Exempel på indata-fil

```
{

      "simulation":{
            "measure":true,
            "evacuation":true,
            "operation":true,
            },

      "tunnel":{
            "tunnelType":"road",
            "length":500.0,
            "width":6.0,
```

```
      "height":5.0,
      "windSpeed":1.5,
      "ambientTemperature":6.0,
      "distanceBetweenEmergencyExits":150.0,
      },

"fire":{
      "fireType":"squared",
      "alfaGrowth":0.1,
      "alfaDecay":0.007,
      "qMax":30000.0,
      "timeFullBurn":90,
      "massOpticalDensity":304.0,
      "heatOfCombustion":15500.0,
      "chi":1.0,
      "yieldCo2":1.50,
      "yieldCo":0.027,
      "yieldHcn":0.01,
      },

"fireEvent":{
      "firePos":100.0,
      "windDirExit":true,
      },

"individuals":[
      {
      "personPosition":455.0,
      "recognitionTime":10,
      "responseTime":10,
      "timeToLeaveVehicle":10,
      "chooseEmergencyExit":true,
      "deterministic":true,
      },

      {
      "personPosition":470.0,
      "recognitionTime":40,
      "responseTime":10,
      "timeToLeaveVehicle":10,
      "chooseEmergencyExit":true,
      "deterministic":true,
      },

],

"groups":[

      {
      "count":10,
      "personPositionIntervalFrom":300.0,
      "personPositionIntervalTo":400.0,
      "recognitionTimeIntervalFrom":30,
      "recognitionTimeIntervalTo":60,
      "responseTimeIntervalFrom":150,
      "responseTimeIntervalTo":180,
      "timeToLeaveVehicleFrom":30,
      "timeToLeaveVehicleTo":90,
      "chooseEmergencyExit":true,
      "deterministic":true,
      },

],


"operation":{
      "preparationTime":15,
      "upstream":false,
      "portal":true,
      "numberOfFireFighters":6,
      "thermalImaging":false,
      "muchAir":false,
      "hoseLengthUneven":25.0,
      "hoseLengthEven":25.0,
      "connectTimeUneven":60,
      "connectTimeEven":300,
      },

"measurementPositions":[
```

```
            {
            "pos":25.0,
            },

            {
            "pos":300.0,
            },

            {
            "pos":400.0,
            },

      ],

}
```


B.2 Uppgift 1B: Enkel tunnelutrymning och rökdykningsinsats i vägtunnel

B.2.1 Räddningsinsats

Hur kan räddningsinsatsen i uppgift 1A förbättras med enkla medel?

B.2.2 Lösningsförslag

Alternativ 1: Prova att utrusta rökdykarna med IR-hjälpmedel. De kan då gå fortare i tunneln och kommer till x = 155 m, d v s 55 m från branden. Alla tre rökdykarparen används. På grund av brandmiljön kan de inte komma längre än så. (Ändra "thermalImaging":false, från false till true)

Alternativ 2: Prova att istället låta insatsen ske genom en utrymningsväg. De inleder insatsen från utgången vid x = 150 m. De kommer knappt 40 m in i tunneln innan branden blir för besvärlig (x = 112 m). (Ändra "portal":true, från true till false)

B.3 Uppgift 2: Utrymning och räddningsinsats i en tågtunnel

De två parallella tunnelrören i Hallandsåstunneln är ca 8,7 km långa och har en diameter av ca 10,5 m. Nedre delen av tunneln är uppfylld med betong för att fästa spåren och innehåller även utrymmen för kablage mm. I tunneln kommer främst moderna motorvagnståg att färdas och ett vanligt tåg kan ha upp mot 240 passagerare. Avståndet mellan utrymningsvägarna är 500 m. Antag att ett tåg stannar i tunneln på en för en utrymning ogynnsam plats. Tiden från det att branden upptäcks tills att utrymningen startar kan vara mellan 5 och 10 minuter. Brandförloppet följer en ”medium” tillväxthastighet upp till 15 MW och är sedan konstant under en timma. När tåget står stilla kan det dröja mellan 2-4 minuter innan utrymningen inleds. Gör en bedömning av möjligheten att utrymma tågets passagerare till en säker plats och möjligheterna att komma till för att göra en räddningsinsats. Den lokala räddningstjänsten kan antas ha två rökdykarpar som kan avancera in i tunneln utöver nödvändiga skyddsgrupper. Insatstiden kan antas vara 30 minuter.

B.3.1 Lösningsförslag

Alla personer kan utrymma från tåget till närmaste utrymningsväg. Rökdykarna kommer inte hela vägen fram till tåget innan branden slocknat (sker efter ca 1,5 timmar). Inleder insatsen vid x = 3 000 m och kommer till x = 3 400 m. Första gruppen kommer till x = 3 300 m.

B.3.1.1 Exempel på indata-fil

Gjorda antaganden:

- Ändra cirkulära tvärsnittet till motsvarande rektangulärt. Arean är ca 90 m^2 vilket ger sidan ca 9 m. Lite fel i area och perimeter men får duga.
- Prova 1,5 m/s vindhastighet
- Tåglängden är 110 m, 8 dörrar per sida (Regina från Bombardier), platsen är 3,5 km in i tunneln och branden är placerad 3,5 km in i tunneln.
- Testa varseblivningstid 5 minuter och förberedelsetid 3 minuter.
- Insats uppströms och via utrymningsväg, ingen IR men stora luftpaket.


```
{
    "simulation":{
          "measure":true,
          "evacuation":true,
          "operation":true,
          },

    "tunnel":{
          "tunnelType":"rail",
          "length":8700.0,
          "width":9.0,
          "height":9.0,
          "windSpeed":1.5,
          "ambientTemperature":20.0,
          "distanceBetweenEmergencyExits":500.0,
          },

    "fire":{
          "fireType":"squared",
          "alfaGrowth":0.012,
          "alfaDecay":0.19,
          "qMax":15000.0,
          "timeFullBurn":60,
          "massOpticalDensity":304.0,
          "heatOfCombustion":15500.0,
          "chi":1.0,
          "yieldCo2":1.50,
          "yieldCo":0.027,
          "yieldHcn":0.01,
          },

    "fireEvent":{
          "firePos":3500.0,
          "windDirExit":true,
          },

    "train":{
          "position":3500.0,
          "trainLength":110.0,
          "numberOfExits":8,
          "exitWidth":1.7,
          "numberOfPassengers":130,
          "recognitionTime":300,
          "responseTime":180,
          "deterministic":false,
          "chooseEmergencyExit":true,
          },

    "operation":{
          "preparationTime":30,
          "upstream":true,
          "portal":false,
          "numberOfFireFighters":4,
          "thermalImaging":false,
          "muchAir":true,
          "hoseLengthUneven":25.0,
          "hoseLengthEven":25.0,
          "connectTimeUneven":60,
          "connectTimeEven":300,
          },
```

```
        "measurementPositions":[

                {
                "pos":25.0,
                },

                {
                "pos":300.0,
                },

                {
                "pos":800.0,
                },

        ],

}
```


B.4 Uppgift 3: Brandförsök i Tistbrottets dolomitgruva

Vid en insatsövning i en nedlagd gruva provas olika strategier att närma sig en brand. Brandscenariot är brand i en tågvagn och simuleras med träpallar som eldas i en fraktcontainer. Branden kan approximeras med en linjär tillväxt upp till 18 MW för att sedan avta igen linjärt under ytterligare 20 minuter. Vid försöken gjordes insatsen uppströms branden men med konström i insatsvägen vilket innebar att röktätheten var hög redan från början. Antag i beräkningen att insatsen sker nedströms branden. Utgå från en brand som genererar en tät rök när insatsen inleds utan att temperaturen blir påtagligt besvärande (effekt kring 10-20 MW kan vara lagom). Hur långt kommer tre rökdykarpar i den miljön? Rökdykarna har stora luftpaket och en av rökdykarna i ett par har IR-hjälpmedel. De har slangar med sig; 50 m + 25 m. Efter 75 m kopplas en grenkoppling, detta tar ca 5 minuter. Första kopplingen innebär bara ihopkoppling av två 63 mm slangar och detta kan göras på en minut. Tunneln är ca 8 m bred och 6 m hög och temperaturen är 10 °C. Lufthastigheten under försöken var ca 2 m/s.

B.4.1 Lösningsförslag

Rökdykarna når fram till branden och endast ett rökdykarpar behöver vara aktiva. I simuleringen inleds insatsen efter 15 minuter även om branden då inte nått maximal effekt 10 MW (sker vid 20 minuter) men röktätheten är hög och temperaturen ligger kring 45 °C. Detta för att simulera de förhållandena som rådde uppströms branden i försöken. Det tar dem drygt 3 minuter att nå ca 75 m in i tunneln. I de genomförda försöken tog det mellan 4 och 6 minuter att nå samma plats, enligt Palm, Kumm & Ingason (2014).

B.4.1.1 Exempel på indata-fil


```
{

        "simulation":{
                "measure":true,
                "evacuation":false,
                "operation":true,
                },

        "tunnel":{
                "tunnelType":"road",
                "length":300.0,
                "width":8.0,
                "height":6.0,
                "windSpeed":2.0,
                "ambientTemperature":10.0,
                "distanceBetweenEmergencyExits":150.0,
                },

        "fire":{
                "fireType":"linear",
                "alfaGrowth":8.3,
                "alfaDecay":8.3,
                "qMax":10000.0,
                "timeFullBurn":20,
                "massOpticalDensity":304.0,
```

```
        "heatOfCombustion":12000.0,
        "chi":1.0,
        "yieldCo2":1.50,
        "yieldCo":0.027,
        "yieldHcn":0.01,
        },

    "fireEvent":{
        "firePos":150.0,
        "windDirExit":true,
        },



    "operation":{
        "preparationTime":15,
        "upstream":false,
        "portal":true,
        "numberOfFireFighters":6,
        "thermalImaging":true,
        "muchAir":true,
        "hoseLengthUneven":50.0,
        "hoseLengthEven":25.0,
        "connectTimeUneven":60,
        "connectTimeEven":300,
        },

    "measurementPositions":[

        {
        "pos":15.0,
        },

        {
        "pos":50.0,
        },

        {
        "pos":800.0,
        },

    ],

}
```

Bilaga C: Programmeringskod

TuFT har programmerats i Java för att möjliggöra användning av planeringsverktyget oberoende av vilket operativsystem som programmet körs på. Tillsammans bygger flera klasser upp modellen, och nedan redovisas programmeringskoden öppet och fullständigt för respektive klass. Motivet är att öppna möjligheten för tillägg i programmet av andra samt för att möjliggöra för kvalitetsgranskning av erhållna resultat som producerats av TuFT.

C.1 Tunnel.java

```
import java.util.ArrayList;

/** This Java class is a part of TuFT, a decision support
 * tool for tunnel fires. The tool was developed by Karl Fridolf
 * and Håkan Frantzich at Lund University during 2013-2014.
 * Among other things, the tool offers the possibility to
 * simulate fires in tunnels, and furthermore, the effects
 * on evacuation and rescue operation possibilities.
 *
 * The tool's sub models, as well as the conclusions and
 * the results produced by the tool, have not been quality
 * controlled, verified or validated to the extent necessary
 * for quality assurance. TuFT must hence be used with caution.
 * Anyone who uses the results produced by TuFT does so on
 * his/her own responsibility */

public class Tunnel {

      private double length;
      private double width;
      private double height;
      private double crossSectionArea;
      private double perimeter;
      private double windSpeed;
      private double temperature;
      private double distanceBetweenEmergenecyExits;
      private ArrayList<Double> emergencyExits;

      /** Creates a tunnel with a length, width, height,
       * wind speed, ambient temperature and emergency
       * exits. */
      public Tunnel(double length, double width,
                  double height, double windSpeed,
                  double temperature, double
                  distanceBetweenEmergenecyExits) {
            this.length = length;
            this.width = width;
            this.height = height;
            this.windSpeed = windSpeed;
            this.temperature = temperature + 273;
            this.distanceBetweenEmergenecyExits =
                        distanceBetweenEmergenecyExits;
            crossSectionArea = width * height;
            perimeter = (2 * width) + (2 * height);
            emergencyExits = new ArrayList<Double>();
            for (int i = 0; i <= (int) Math.round((length /
                        distanceBetweenEmergenecyExits)); i++) {
                  emergencyExits.add(distanceBetweenEmergenecyExits * i);
            }
      }

      /** Returns the wind speed in m/s. */
      public double getWindSpeed() {
            return windSpeed;
```

```
	}

	/** Returns the height in m. */
	public double getHeight() {
		return height;
	}

	/** Returns the width in m. */
	public double getWidth() {
		return width;
	}

	/** Returns the length in m. */
	public double getLength() {
		return length;
	}

	/** Returns the perimeter in m. */
	public double getPerimeter() {
		return perimeter;
	}

	/** Returns the cross section area in m^2. */
	public double getCrossSectionArea() {
		return crossSectionArea;
	}

	/** Returns the ambient temperature in K. */
	public double getTemp() {
		return temperature;
	}

	/** Returns the distance between the emergency
	 * exits in m. */
	public double getDistanceBetweenEmergencyExits() {
		return distanceBetweenEmergenecyExits;
	}

	/** Returns the array of emergency exits (their
	 * positions measured from the entrance in m). */
	public ArrayList<Double> getEmergencyExits() {
		return emergencyExits;
	}

}
```

C.2 Fire.java

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;

/** This Java class is a part of TuFT, a decision support
 * tool for tunnel fires. The tool was developed by Karl Fridolf
 * and Håkan Frantzich at Lund University during 2013-2014.
 * Among other things, the tool offers the possibility to
 * simulate fires in tunnels, and furthermore, the effects
 * on evacuation and rescue operation possibilities.
 *
 * The tool's sub models, as well as the conclusions and
 * the results produced by the tool, have not been quality
 * controlled, verified or validated to the extent necessary
 * for quality assurance. TuFT must hence be used with caution.
 * Anyone who uses the results produced by TuFT does so on
 * his/her own responsibility */

public class Fire {
	private ArrayList<Double> heatReleaseRate;
	double alfaGrowth;
	double alfaDecay;
	boolean linearGrowth;
	double qMax;
	int devTime;
	double massOpticalDensity;
```

```
double heatOfCombustion;
double chi;
double yieldCO2;
double yieldCO;
double yieldHCN;
int totalTime;

/** Creates a linear or squared fire defined by a growth rate,
 * decay rate, maximum heat release rate, burning time
 * with maximum heat release rate, mass optical density,
 * heat of combustion, burning efficiency and yields for CO2,
 * CO and HCN. */
public Fire(boolean linearGrowth, double alfaGrowth, double alfaDecay,
            double qMax, int devTime, double massOpticalDensity,
            double heatOfCombustion, double chi,
            double yieldCO2, double yieldCO,
            double yieldHCN) {
      this.alfaGrowth = alfaGrowth;
      this.alfaDecay = alfaDecay;
      this.linearGrowth = linearGrowth;
      this.qMax = qMax;
      this.devTime = devTime;
      this.massOpticalDensity = massOpticalDensity;
      this.heatOfCombustion = heatOfCombustion;
      this.chi = chi;
      this.yieldCO2 = yieldCO2;
      this.yieldCO = yieldCO;
      this.yieldHCN = yieldHCN;
      int t1 = 0;
      if (linearGrowth) {
            t1 = (int) Math.round(qMax / alfaGrowth);
      } else if (!linearGrowth) {
            t1 = (int) Math.round(Math.sqrt(qMax / alfaGrowth));
      }
      int t2 = devTime * 60;
      int t3 = 0;
      if (linearGrowth) {
            t3 = (int) Math.round(qMax / alfaDecay);;
      } else if (!linearGrowth) {
            t3 = (int) Math.round(Math.sqrt(qMax / alfaDecay));
      }
      totalTime = t1 + t2 + t3;
      heatReleaseRate = new ArrayList<Double>();
      for (int i = 0; i < totalTime; i++) {
            double q = 0;
            if (i <= t1) {
                  if (linearGrowth) {
                        q = alfaGrowth * i;
                  } else if (!linearGrowth) {
                        q = alfaGrowth * Math.pow(i, 2);
                  }
            } else if (i > t1 && i <= (t1 + t2)) {
                  q = qMax;
            } else if (i > (t1 + t2) && i <= totalTime) {
                  if (linearGrowth) {
                        q = qMax - (alfaDecay * (i - (t1 + t2)));
                  } else if (!linearGrowth) {
                        q = alfaDecay * Math.pow(totalTime - i, 2);
                  }
            }
            heatReleaseRate.add(i, q);
      }
}

/** Creates an exponential fire, defined by research
 * done by Haukur Ingason. */
public Fire(double qMaxExp, double eTot, double tMax,
            double massOpticalDensity, double heatOfCombustion,
            double chi, double yieldCO2, double yieldCO,
            double yieldHCN) {
      this.qMax = qMaxExp;
      this.massOpticalDensity = massOpticalDensity;
      this.heatOfCombustion = heatOfCombustion;
      this.chi = chi;
      this.yieldCO2 = yieldCO2;
      this.yieldCO = yieldCO;
      this.yieldHCN = yieldHCN;
      double n = Math.exp(((2.9 * qMax) / (eTot * 1000.0)) * tMax * 60.0) / 1.346;
      double r = Math.pow((1.0 - (1.0 / n)), (1.0 - n));
```

```
            double k = (qMax / (eTot * 1000) * r);
            totalTime = 120 * 60;
            heatReleaseRate = new ArrayList<Double>();
            double q = 0.0;
            for (int i = 0; i < totalTime; i++) {
                  q = 1000 * (qMax / 1000.0) * n * r * Math.pow((1.0 - Math.exp(-k * i)), (n - 1)) *
Math.exp(-k * i);
                  heatReleaseRate.add(i, q);
            }
      }

      /** Returns the growth rate of the fire in
       * kW/s (linear) or kW/s2 (squared). */
      public double getAlfaGrowth() {
            return alfaGrowth;
      }

      /** Returns the growth rate of the fire in
       * kW/s (linear) or kW/s2 (squared) */
      public double getAlfaDecay() {
            return alfaDecay;
      }

      /** Returns true if the fire is growing
       * with a linear correlation, or false
       * if the fire is growing squared. */
      public boolean getGrowthRate() {
            return linearGrowth;
      }

      /** Returns the maximum HRR of the fire
       * in kW. */
      public double getQMax() {
            return qMax;
      }

      /** Returns the time with which the fire
       * is burning with its qMax (in min). */
      public int getTimeFullBurn() {
            return devTime;
      }

      /** Returns the heat release rate in
       * kW at the time t (in s) into the fire
       * development. */
      public double getQ(int t) {
            return heatReleaseRate.get(t);
      }

      /** Returns the mass optical density
       * in m^2/kg. */
      public double getDmass() {
            return massOpticalDensity;
      }

      /** Returns the heat of combustion
       * in kJ/kg. */
      public double getHeatOfCombustion() {
            return heatOfCombustion;
      }

      /** Returns the burning efficiency. */
      public double getChi() {
            return chi;
      }

      /** Returns the yield for CO2 in kg/kg. */
      public double getYieldCO2() {
            return yieldCO2;
      }

      /** Returns the yield for CO in kg/kg. */
      public double getYieldCO() {
            return yieldCO;
      }

      /** Returns the yield for HCN in kg/kg. */
      public double getYieldHCN() {
            return yieldHCN;
```

```
	}

	/** Returns the total burning time in s. */
	public int getTotalTime() {
		return totalTime;
	}

	/** Prinst an output file related to the
	 * heat release rate of the fire as a function
	 * of time in s. */
	public void printHrr() {
		String fileName = "HRR.txt";

		try {
			FileWriter fileWriter = new FileWriter(fileName);
			BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

			bufferedWriter.write("Output generated with TuFT (v 14.10.22.1). Anyone who uses the
results produced by TuFT does so on his/her own responsibility");
			bufferedWriter.newLine();
			bufferedWriter.newLine();

			bufferedWriter.write("TIME [s]");
			bufferedWriter.write(",");
			bufferedWriter.write("HRR [kW]");
			for (int i = 0; i < this.getTotalTime(); i++) {
				bufferedWriter.newLine();
				bufferedWriter.write(Integer.toString(i));
				bufferedWriter.write(",");
				bufferedWriter.write(Double.toString(this.getQ(i)));
			}

			bufferedWriter.close();
		}

		catch (IOException ex) {
		System.out.println(
		    "Error writing to file '"
		    + fileName + "'");
		}
	}

}
```

C.3 FireEvent.java

```
/** This Java class is a part of TuFT, a decision support
 * tool for tunnel fires. The tool was developed by Karl Fridolf
 * and Håkan Frantzich at Lund University during 2013-2014.
 * Among other things, the tool offers the possibility to
 * simulate fires in tunnels, and furthermore, the effects
 * on evacuation and rescue operation possibilities.
 *
 * The tool's sub models, as well as the conclusions and
 * the results produced by the tool, have not been quality
 * controlled, verified or validated to the extent necessary
 * for quality assurance. TuFT must hence be used with caution.
 * Anyone who uses the results produced by TuFT does so on
 * his/her own responsibility */

public class FireEvent {
	private Tunnel tunnel;
	private Fire fire;
	private double pos;
	private double windSpeed;
	private double ambientTemperatureK;
	private double ambientTemperatureC;
	private double massFlowAir;
	private double airDensity;
	private double airCp;
	private double perimeter;
	private double height;
	private double crossSectionArea;
	private boolean windTowardsExit;
```

```java
/** Creates a fire Fire in a tunnel Tunnel,
 * at the position pos m from the tunnel
 * entrance (x = 0). windDirection describes if
 * the wind is blowing towards the tunnel
 * exit or the tunnel entrance. */
public FireEvent(Tunnel tunnel, Fire fire,
            double pos, boolean windTowardsExit) {
      this.fire = fire;
      this.tunnel = tunnel;
      this.pos = pos;
      this.windTowardsExit = windTowardsExit;
      windSpeed = tunnel.getWindSpeed();
      ambientTemperatureK = tunnel.getTemp();
      ambientTemperatureC = ambientTemperatureK - 273;
      crossSectionArea = tunnel.getCrossSectionArea();
      airDensity = this.getAirDens(ambientTemperatureK);
      airCp = this.getAirCp(ambientTemperatureK);
      massFlowAir = airDensity * windSpeed * crossSectionArea;
      perimeter = tunnel.getPerimeter();
      height = tunnel.getHeight();
}

/** Returns the backlayering distance
 * upstream the fire in m at the time t in s. */
public double getBackLayeringLength(int t) {
      double q = fire.getQ(t);
      double lB = (1.4 * height *
                  Math.pow(((9.81 * q) / (airDensity
                              * airCp * ambientTemperatureK
                              * Math.pow(windSpeed, 3) *
                              height)), 0.3));
      return lB;
}

/** Returns the vertical averaged gas temperature
 * in °C at the position of the fire at
 * the time tau (in s). */
private double getTavgX0(int tau) {
      double qTau = fire.getQ(tau);
      double tAvgX0 = ambientTemperatureC +
                  ((2.0 / 3.0) * (qTau / (massFlowAir * airCp)));
      return tAvgX0;
}

/** Returns the vertical averaged gas temperature
 * in °C at the position x (relative to the fire)
 * at the time t (in s). */
public double getTavgX(int t, double x) {
      int tau = this.getTau(t, x);
      if (tau < 0) {
            return -1.0;
      } else {
            double tAvgX0 = this.getTavgX0(tau);
            double tAvgX = ambientTemperatureC +
                        ((tAvgX0 - ambientTemperatureC) *
                                    Math.exp(-((0.02 * perimeter * x) /
                                                (massFlowAir * airCp))));
            return tAvgX;
      }
}

/** Returns the Froude number. Only used internally
 * in this class. */
private double getFroude(int t, double x) {
      double tAvgX = this.getTavgX(t, x);
      if (tAvgX == -1) {
            return -1;
      } else {
            double numerator = Math.pow((windSpeed *
                        ((tAvgX + 273) /
                                    ambientTemperatureK)), 2);
            double denominator = (1.5 * ((tAvgX -
                        ambientTemperatureC) / (tAvgX + 273)) *
                        9.81 * height);
            return (numerator / denominator);
      }
}

/** Returns the region number at a position x m
```

```
 * downstream the fire at the time t in s. */
public int getRegion(int t, double x) {
      int region;
      if (this.getFroude(t, x) == -1) {
            region = -1;
      } else if (this.getFroude(t, x) <= 0.9 &&
                  this.getFroude(t, x) >= 0) {
            region = 1;
      } else if (this.getFroude(t, x) > 0.9 &&
                  this.getFroude(t, x) <= 10) {
            region = 2;
      } else {
            region = 3;
      }
      return region;
}

/** Returns the vertical, average extinction
 * coefficient in 1/m at a position x m
 * downstream the fire at the time t in s. */
public double getExtinctionCoefficient(int t, double x) {
      double vis = this.getVisibility(t, x);
      double extCoeff = 2 / vis;
      return extCoeff;
}

/** Returns the vertical, average visibility
 * in m at a position x m downstream the fire
 * at the time t in s. */
public double getVisibility(int t, double x) {
      double effectiveHeatOfCombustion = fire.getChi() *
                  fire.getHeatOfCombustion();
      int tau = this.getTau(t, x);
      if (tau < 0) {
            return -1.0;
      } else {
            double vis = (0.87 * ((windSpeed * crossSectionArea *
                        effectiveHeatOfCombustion) / (fire.getQ(tau) *
                                    fire.getDmass())));
            return vis;
      }
}

/** Returns the vertical, average mole fraction carbon
 * dioxide at a position x m downstream the fire at the
 * time t in s. */
public double getMoleFractionCO2(int t, double x) {
      double yCO2 = fire.getYieldCO2();
      int tau = this.getTau(t, x);
      if (tau < 0) {
            return -1.0;
      } else {
            double xCO2Avg = (yCO2 * (28.95 / 44.0) *
                        (fire.getQ(tau) / (massFlowAir * fire.getChi() *
                                    fire.getHeatOfCombustion())));
            return xCO2Avg;
      }
}

/** Returns the vertical, average mole fraction carbon
 * monoxide at a position x m downstream the fire at the
 * time t in s. */
public double getMoleFractionCO(int t, double x) {
      double yCO = fire.getYieldCO();
      int tau = this.getTau(t, x);
      if (tau < 0) {
            return -1.0;
      } else {
            double xCOAvg = (yCO * (28.95 / 28.0) *
                        (fire.getQ(tau) / (massFlowAir * fire.getChi() *
                                    fire.getHeatOfCombustion())));
            return xCOAvg;
      }
}

/** Returns the vertical, average mole fraction hydrogen
 * cyanide at a position x m downstream the fire at the
 * time t in s. */
public double getMoleFractionHCN(int t, double x) {
```

```
        double yHCN = fire.getYieldHCN();
        int tau = this.getTau(t, x);
        if (tau < 0) {
                return -1.0;
        } else {
                double xHCNAvg = (yHCN * (28.95 / 27.0) *
                                (fire.getQ(tau) / (massFlowAir * fire.getChi() *
                                                fire.getHeatOfCombustion())));
                return xHCNAvg;
        }
}

/** Returns the vertical, average mole fraction oxygen
 * at a position x m downstream the fire at the
 * time t in s. */
public double getMoleFractionO2(int t, double x) {
        int tau = this.getTau(t, x);
        if (tau < 0) {
                return -1.0;
        } else {
                double xO2Avg = (0.2095 - ((28.95 / 32.0) *
                                (fire.getQ(tau) / (massFlowAir * 13100))));
                return xO2Avg;
        }
}

/** Returns the heat flux in kW/m2 at a position x m downstream
 * the fire at the time t in s. */
public double getHeatTransport(int t, double x) {
        double temp = this.getTavgX(t, x);
        double qRadFire = this.getRadiationFromFire(t, x);
        double qSum = 0.0;
        if (temp == -1.0) {
                if (x < 50) {
                        qSum = qRadFire;
                        return qRadFire / 1000.0;
                } else {
                        return 0.0;
                }
        } else {
                double qRadSmoke = (0.5 * 5.67 * Math.pow(10.0, -8.0)
                                * (Math.pow((temp + 273.0), 4.0) -
                                                Math.pow(tunnel.getTemp(), 4.0))); // W/m2
                double qConvSmoke = (8.0 * (temp -
                                (tunnel.getTemp() - 273.0))); // W/m2
                if (x < 50) {
                        qSum = qRadFire + qRadSmoke + qConvSmoke;
                } else {
                        qSum = qRadSmoke + qConvSmoke;
                }
                return qSum / 1000.0;
        }
}

/** Returns tau, i.e., the time in s at which the smoke particles
 * starts to flow from the source of the fire (x = 0). */
private int getTau(int t, double x) {
        int tau = t - (int) Math.round(x / windSpeed);
        return tau;
}

/** Returns Cp in kJ/kg K for air at a given temperature in K.
 * Correlation derived by fitting regression line to data. */
private double getAirCp(double T) {
        double cp = (Math.pow(10, -13) * Math.pow(T, 4))
                        - (5 * Math.pow(10, -10) * Math.pow(T, 3))
                        + (8 * Math.pow(10, -7) * Math.pow(T, 2))
                        - (0.0003 * T) + 1.0321;
        return cp;
}

/** Returns the density in kg/m^3 for air at a given temperature
 * in K. Correlation derived by fitting regression line to data. */
private double getAirDens(double T) {
        double dens = (2 * Math.pow(10, -18) * Math.pow(T, 6))
                        - (2 * Math.pow(10, -14) * Math.pow(T, 5))
                        + (5 * Math.pow(10, -11) * Math.pow(T, 4))
                        - (6 * Math.pow(10, -8) * Math.pow(T, 3))
                        + (5 * Math.pow(10, -5) * Math.pow(T, 2))
```

```
                - (0.0216 * T) + 4.5049;
        return dens;
}

/** Returns the distance in m between the fire
 * and the tunnel entrance. */
public double getDistanceFromEntrance() {
        return pos;
}

/** Returns the distance between the fire and the
 * tunnel exit in m. */
public double getDistanceFromExit() {
        return (tunnel.getLength() - pos);
}

/** Returns the direction of the wind, i.e., if the wind
 * is blowing with (true) or against (false) the direction
 * of the traffic. */
public boolean getWindDirection() {
        return windTowardsExit;
}

/** Returns the fraction of an incapacitating
 * dose of CO for a 1-second exposure to the gas
 * concentration in ppm. The calculation is performed
 * at a time t in s at a defined position x in m downstream
 * the fire. Thus, note that the user has to specify not a
 * position of the person in the tunnel, but the relative
 * position compared to the fire.
 *
 * If person is not incapacitated (i.e., FID < 1.0),
 * calculation is done according to Eq. 7 in SFPE handbook
 * (p. 2-117, 4th Ed.) for activity level of a light
 * working person. If person is incapacitated (i.e.,
 * FID => 1.0), calculation is done similarly but for
 * activity level of a resting or sleeping person. */
private double getFedIncapacitationCo(int t, double x,
            boolean incapacitated) {
        double xCOAvg = this.getMoleFractionCO(t, x);
        if (xCOAvg == -1.0) {
                return 0.0;
        } else {
                double coPpm = (xCOAvg * Math.pow(10.0, 6.0));
                if (!incapacitated) {
                        double fedIncapacitationCo = ((8.2925 *
                                        Math.pow(10.0, -4.0) *
                                        Math.pow(coPpm, 1.036) *
                                        (1.0 / 60.0)) / 30.0);
                        return fedIncapacitationCo;
                } else {
                        double fedIncapacitationCo = ((2.81945 *
                                        Math.pow(10.0, -4.0) *
                                        Math.pow(coPpm, 1.036) *
                                        (1.0 / 60.0)) / 40.0);
                        return fedIncapacitationCo;
                }
        }
}

/** Returns the fraction of an incapacitating
 * dose of HCN for a 1-second exposure to the gas
 * concentration in ppm. The calculation is performed
 * at a time t in s at a defined position x in m downstream
 * the fire. Thus, note that the user has to specify not a
 * position of the person in the tunnel, but the relative
 * position compared to the fire.
 *
 * Calculation is done according to Eq. 11 in SFPE handbook
 * (p. 2-119, 4th Ed.). */
private double getFedIncapacitationHcn(int t, double x) {
        double xHCNAvg = this.getMoleFractionHCN(t, x);
        if (xHCNAvg == -1.0) {
                return 0.0;
        } else {
                double hcnPpm = (xHCNAvg * Math.pow(10.0, 6.0));
                double fedIncapacitationHcn =
                                (((Math.exp(hcnPpm / 43.0) / 220.0) -
                                0.0045) * (1.0 / 60.0));
```

```
            return fedIncapacitationHcn;
      }
}

/** Returns the fraction of an incapacitating
 * dose of low-oxygen hypoxia for a 1-second exposure to
 * the gas concentration in ppm. The calculation is
 * performed at a time t in s at a defined position x in
 * m downstream the fire. Thus, note that the user has to
 * specify not a position of the person in the tunnel, but
 * the relative position compared to the fire.
 *
 * Calculation is done according to Eq. 12-13 in SFPE handbook
 * (p. 2-121, 4th Ed.). */
private double getFedIncapacitationO2(int t, double x) {
      double xO2Avg = this.getMoleFractionO2(t, x);
      if (xO2Avg == -1.0) {
            return 0.0;
      } else {
            double o2Percent = (xO2Avg * Math.pow(10.0, 2.0));
            double fedIncapacitationO2 = ((1.0 / 60.0) /
                        (Math.exp(8.13 - (0.54 * (20.9 - o2Percent)))));
            return fedIncapacitationO2;
      }
}

/** Returns the multiplication factor for CO2-induced
 * hyperventilation. The calculation is performed
 * at a time t in s at a defined position x in m downstream
 * the fire. Thus, note that the user has to specify not a
 * position of the person in the tunnel, but the relative
 * position compared to the fire.
 *
 * Calculation is done according to Eq. 18 in SFPE handbook
 * (p. 2-122, 4th Ed.). */
private double getHyperVentilationFactor(int t, double x) {
      double xCO2Avg = this.getMoleFractionCO2(t, x);
      if (xCO2Avg == -1.0) {
            return 0.0;
      } else {
            double co2Percent = (xCO2Avg * Math.pow(10.0, 2.0));
            double vCo2 = Math.exp(co2Percent / 5.0);
            return vCo2;
      }
}

/** Returns the fraction of an incapacitating
 * dose of all asphyxiant gases. The calculation is performed
 * at a time t in s at a defined position x in m downstream
 * the fire. Thus, note that the user has to specify not a
 * position of the person in the tunnel, but the relative
 * position compared to the fire.
 *
 * Calculation is done according to Eq. 21 in SFPE handbook
 * (p. 2-124, 4th Ed.). */
public double getFractionOfIncapacitationAsphyxiants(int t, double x,
            boolean incapacitated) {
      double fedIn = (((this.getFedIncapacitationCo(t, x, incapacitated) +
                  this.getFedIncapacitationHcn(t, x)) *
                  this.getHyperVentilationFactor(t, x)) +
                  this.getFedIncapacitationO2(t, x));
      return fedIn;
}

/** Returns the fractional incapacitating dose of heat acquired
 * from radiant and convected components of fire and smoke.
 * Assumption: Evacuating person is immersed in smoke.
 *
 * If person is within 50 m of fire , the radiant heat flux from
 * the fire is calculated according to Eq. 7.16 in Enclousure
 * Fire Dynamics (p. 156), and then added to the total heat flux.
 *
 * Calculation of the radiant and convected components of the
 * smoke are calculated according to Eq. 36 in SFPE handbook
 * (p. 2-145, 4th Ed.). Assumptions: Emissivity = 0.5, material
 * surface temperature = ambient temperature in tunnel,
 * convective heat transfer factor = 8 W/m2 K. */
public double getFractionOfIncapacitationHeatDownstream(int t, double x) {
      double temp = this.getTavgX(t, x);
```

```
        double qRadFire = this.getRadiationFromFire(t, x);
        double qSum = 0.0;
        double timeToIncapacitation = 0.0;
        double fraction = 0.0;
        if (temp == -1.0) {
              if (x < 50) {
                    qSum = qRadFire / 1000.0;
                    timeToIncapacitation = (10.0 / Math.pow(qSum, 1.33));
                    fraction = ((1.0 / timeToIncapacitation) * (1.0 / 60.0));
                    return fraction;
              } else {
                    return 0.0;
              }
        } else {
              double qRadSmoke = (0.5 * 5.67 * Math.pow(10.0, -8.0)
                          * (Math.pow((temp + 273.0), 4.0) -
                                      Math.pow(tunnel.getTemp(), 4.0)));
              double qConvSmoke = (8.0 * (temp -
                          (tunnel.getTemp() - 273.0)));
              if (x < 50) {
                    qSum = (qRadFire + qRadSmoke + qConvSmoke) / 1000.0;
              } else {
                    qSum = (qRadSmoke + qConvSmoke) / 1000.0;
              }
              timeToIncapacitation = (10.0 / Math.pow(qSum, 1.33));
              fraction = ((1.0 / timeToIncapacitation) * (1.0 / 60.0));
              return fraction;
        }
}

/** Returns the fractional lethal dose of heat acquired
 * from radiant and convected components of fire and smoke.
 * Assumption: Evacuating person is immersed in smoke.
 *
 * If person is within 50 m of fire , the radiant heat flux from
 * the fire is calculated according to Eq. 7.16 in Enclousure
 * Fire Dynamics (p. 156), and then added to the total heat flux.
 *
 * Calculation of the radiant and convected components of the
 * smoke are calculated according to Eq. 36 in SFPE handbook
 * (p. 2-145, 4th Ed.). Assumptions: Emissivity = 0.5, material
 * surface temperature = ambient temperature in tunnel,
 * convective heat transfer factor = 8 W/m2 K. */
public double getFractionOfLethalHeatDownstream(int t, double x) {
        double temp = this.getTavgX(t, x);
        double qRadFire = this.getRadiationFromFire(t, x);
        double qSum = 0.0;
        double timeToIncapacitation = 0.0;
        double fraction = 0.0;
        if (temp == -1.0) {
              if (x < 50) {
                    qSum = qRadFire / 1000.0;
                    timeToIncapacitation = (16.7 / Math.pow(qSum, 1.33));
                    fraction = ((1.0 / timeToIncapacitation) * (1.0 / 60.0));
                    return fraction;
              } else {
                    return 0.0;
              }
        } else {
              double qRadSmoke = (0.5 * 5.67 * Math.pow(10.0, -8.0)
                          * (Math.pow((temp + 273.0), 4.0) -
                                      Math.pow(tunnel.getTemp(), 4.0)));
              double qConvSmoke = (8.0 * (temp -
                          (tunnel.getTemp() - 273.0)));
              if (x < 50) {
                    qSum = (qRadFire + qRadSmoke + qConvSmoke) / 1000.0;
              } else {
                    qSum = (qRadSmoke + qConvSmoke) / 1000.0;
              }
              timeToIncapacitation = (16.7 / Math.pow(qSum, 1.33));
              fraction = ((1.0 / timeToIncapacitation) * (1.0 / 60.0));
              return fraction;
        }
}

/** Returns the fractional incapacitating dose of heat acquired
 * from radiant component of fire.
 *
 * If person is within 50 m of fire , the radiant heat flux from
```

```
 * the fire is calculated according to Eq. 7.16 in Enclousure
 * Fire Dynamics (p. 156), and then added to the total heat flux. */
public double getFractionOfIncapacitationHeatUpstream(int t, double x) {
      if (x < 50) {
            double qRadFire = this.getRadiationFromFire(t, x) / 1000.0;
            double timeToIncapacitation = (10.0 /
                        Math.pow(qRadFire, 1.33));
            double fraction = ((1.0 / timeToIncapacitation) *
                        (1.0 / 60.0));
            return fraction;
      } else {
            return 0.0;
      }
}

/** Returns the fractional lethal dose of heat acquired
 * from radiant component of fire.
 *
 * If person is within 50 m of fire , the radiant heat flux from
 * the fire is calculated according to Eq. 7.16 in Enclousure
 * Fire Dynamics (p. 156), and then added to the total heat flux. */
public double getFractionOfLethalHeatUpstream(int t, double x) {
      if (x < 50) {
            double qRadFire = this.getRadiationFromFire(t, x) / 1000.0;
            double timeToIncapacitation = (16.7 /
                        Math.pow(qRadFire, 1.33));
            double fraction = ((1.0 / timeToIncapacitation) *
                        (1.0 / 60.0));
            return fraction;
      } else {
            return 0.0;
      }
}

/** Returns the radiant heat flux from the fire in W/m2,
 * calculated according to Eq. 7.16 in Enclousure
 * Fire Dynamics (p. 156). */
public double getRadiationFromFire(int t, double x) {
      if (x < 50) {
            double qRadFire = (((fire.getQ(t) * 0.33) /
                        (4 * Math.PI * Math.pow(x, 2))));
            return (qRadFire * 1000.0);
      } else {
            return 0.0;
      }
}

/** Returns the delta temperature rise due to gas temperature
 * and radiation from the fire. Input is the time t in s and
 * the relative position x in m from the fire.
 *
 * If the operation is performed upstream the fire, the
 * consideration is only taken to the radiation form the source. If,
 * on the other hand, the operation is done downsteram the fire, also
 * gas temperature is considered.
 *
 * Evaluation of the delta temperature rise is done based on
 * Eq. 5.3 in (Ingason, Bergqvist, Lönnermark, Frantzich and
 * Hasselrot, 2005). */
public double getFireFighterBodyTemperatureRise(int t, double x) {
      double f = 1.3;
      double A = 1.85;
      double iRad = this.getRadiationFromFire(t, x) * 1000;
      double fEff = 0.71;
      double rArC = 0.15;
      double tG = this.getTavgX(t, x);
      double tB = 37.0;
      double rC = 0.465;
      double M = 300.0;
      double pS = 5940.0;
      double pA = 700.0;
      double rE = 75.0;
      double m = 75.0;
      double cP = 3480.0;
      double deltaTime = 1.0;

      double deltaTemperatureRise = 0.0;
      if (iRad < 5000) {
```

```
                deltaTemperatureRise = ((f * A * ((iRad * fEff * rArC) + ((tG - tB) / rC) + (M / f)))
- (f * A * ((pS - pA) / rE))) / (m * cP * deltaTime);
            } else {
                deltaTemperatureRise = 2.5;
            }

            return deltaTemperatureRise;
        }

        /** Returns the current fire. */
        public Fire getFire() {
            return fire;
        }

        /** Returns the current tunnel. */
        public Tunnel getTunnel() {
            return tunnel;
        }

        /** Returns a string with current version number
         * of TuFT. */
        public String versionText() {
            String text = "Output generated with TuFT (v 14.10.22.1). Anyone who uses the results
produced by TuFT does so on his/her own responsibility.";
            return text;
        }

}
```

C.4 Measures.java

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;

/** This Java class is a part of TuFT, a decision support
 * tool for tunnel fires. The tool was developed by Karl Fridolf
 * and Håkan Frantzich at Lund University during 2013-2014.
 * Among other things, the tool offers the possibility to
 * simulate fires in tunnels, and furthermore, the effects
 * on evacuation and rescue operation possibilities.
 *
 * The tool's sub models, as well as the conclusions and
 * the results produced by the tool, have not been quality
 * controlled, verified or validated to the extent necessary
 * for quality assurance. TuFT must hence be used with caution.
 * Anyone who uses the results produced by TuFT does so on
 * his/her own responsibility */
public class Measures {
        private FireEvent fireEvent;
        private Tunnel tunnel;
        private ArrayList<Integer> time;
        private ArrayList<Double> positions;

        public Measures (FireEvent fireEvent, int timeStart,
                int timeEnd, ArrayList<Double> positions) {
            this.fireEvent = fireEvent;
            this.tunnel = fireEvent.getTunnel();
            this.positions = positions;

            time = new ArrayList<Integer>();
            for (int i = timeStart; i < timeEnd + 1; i++) {
                time.add(i);
            }
        }

        /** Generates output file related to average gas
         * temperature in tunnel for pre-defined positions
         * downstream the fire for every s of the
         * fire development. */
        public void printTempToFile() {
            String fileName = "tunnelTemp.txt";
            double temp = 0.0;
```

```
        try {
                FileWriter fileWriter = new FileWriter(fileName);
                BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

                bufferedWriter.write(fireEvent.versionText());
                bufferedWriter.newLine();
                bufferedWriter.newLine();

                bufferedWriter.write("TIME");
                for (int i = 0; i < positions.size(); i++) {
                        bufferedWriter.write(",");
                        bufferedWriter.write("x = ");
                        bufferedWriter.write(Double.toString(positions.get(i)));
                        bufferedWriter.write(" m");
                }
                for (int i = 0; i < time.size(); i++) {
                        bufferedWriter.newLine();
                        bufferedWriter.write(Integer.toString(time.get(i)));
                        for (int j = 0; j < positions.size(); j++) {
                                bufferedWriter.write(",");
                                temp = fireEvent.getTavgX(i, positions.get(j));
                                if (temp == -1.0) {
                                        bufferedWriter.write(Double.toString(tunnel.getTemp() - 273.0));
                                } else {
                                        bufferedWriter.write(Double.toString(temp));
                                }
                        }
                }
                bufferedWriter.close();
        }

        catch (IOException ex) {
        System.out.println(
            "Error writing to file '"
            + fileName + "'");
        }
}

/** Generates output file related to average gas
 * mole fractions in tunnel for pre-defined positions
 * downstream the fire for every s of the
 * fire development. */
public void printGasToFile() {
        String fileName = "tunnelGasMoleFraction.txt";
        double temp = 0.0;

        try {
                FileWriter fileWriter = new FileWriter(fileName);
                BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

                bufferedWriter.write(fireEvent.versionText());
                bufferedWriter.newLine();
                bufferedWriter.newLine();

                bufferedWriter.write("TIME");
                for (int i = 0; i < positions.size(); i++) {
                        bufferedWriter.write(",");
                        bufferedWriter.write("X_CO2 | x = ");
                        bufferedWriter.write(Double.toString(positions.get(i)));
                        bufferedWriter.write(" m");
                        bufferedWriter.write(",");
                        bufferedWriter.write("X_CO | x = ");
                        bufferedWriter.write(Double.toString(positions.get(i)));
                        bufferedWriter.write(" m");
                        bufferedWriter.write(",");
                        bufferedWriter.write("X_O2 | x = ");
                        bufferedWriter.write(Double.toString(positions.get(i)));
                        bufferedWriter.write(" m");
                        bufferedWriter.write(",");
                        bufferedWriter.write("X_HCN | x = ");
                        bufferedWriter.write(Double.toString(positions.get(i)));
                        bufferedWriter.write(" m");
                }
                for (int i = 0; i < time.size(); i++) {
                        bufferedWriter.newLine();
                        bufferedWriter.write(Integer.toString(time.get(i)));
                        for (int j = 0; j < positions.size(); j++) {
                                bufferedWriter.write(",");
                                temp = fireEvent.getMoleFractionCO2(i, positions.get(j));
```

```
                        if (temp == -1.0) {
                              bufferedWriter.write("");
                        } else {
                              bufferedWriter.write(Double.toString(temp));
                        }
                        bufferedWriter.write(",");
                        temp = fireEvent.getMoleFractionCO(i, positions.get(j));
                        if (temp == -1.0) {
                              bufferedWriter.write("");
                        } else {
                              bufferedWriter.write(Double.toString(temp));
                        }
                        bufferedWriter.write(",");
                        temp = fireEvent.getMoleFractionO2(i, positions.get(j));
                        if (temp == -1.0) {
                              bufferedWriter.write("");
                        } else {
                              bufferedWriter.write(Double.toString(temp));
                        }
                        bufferedWriter.write(",");
                        temp = fireEvent.getMoleFractionHCN(i, positions.get(j));
                        if (temp == -1.0) {
                              bufferedWriter.write("");
                        } else {
                              bufferedWriter.write(Double.toString(temp));
                        }
                  }
            }
            bufferedWriter.close();
      }

      catch (IOException ex) {
      System.out.println(
          "Error writing to file '"
          + fileName + "'");
      }

}

/** Generates output file related to average extinction
 * coefficient in tunnel for pre-defined positions
 * downstream the fire for every s of the
 * fire development. */
public void printExtinctionCoeffToFile() {
      String fileName = "tunnelExtinction.txt";
      double temp = 0.0;

      try {
            FileWriter fileWriter = new FileWriter(fileName);
            BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

            bufferedWriter.write(fireEvent.versionText());
            bufferedWriter.newLine();
            bufferedWriter.newLine();

            bufferedWriter.write("TIME");
            for (int i = 0; i < positions.size(); i++) {
                  bufferedWriter.write(",");
                  bufferedWriter.write("x = ");
                  bufferedWriter.write(Double.toString(positions.get(i)));
                  bufferedWriter.write(" m");
            }
            for (int i = 0; i < time.size(); i++) {
                  bufferedWriter.newLine();
                  bufferedWriter.write(Integer.toString(time.get(i)));
                  for (int j = 0; j < positions.size(); j++) {
                        bufferedWriter.write(",");
                        temp = fireEvent.getExtinctionCoefficient(i, positions.get(j));
                        if (temp == -2.0) {
                              bufferedWriter.write(" ");
                        } else {
                              bufferedWriter.write(Double.toString(temp));
                        }
                  }
            }
            bufferedWriter.close();
      }

      catch (IOException ex) {
```

```
        System.out.println(
            "Error writing to file '"
            + fileName + "'");
        }
    }

    /** Generates output file related to average
     * visibility in tunnel for pre-defined positions
     * downstream the fire for every s of the
     * fire development. */
    public void printVisibilityToFile() {
        String fileName = "tunnelVisibility.txt";
        double temp = 0.0;

        try {
            FileWriter fileWriter = new FileWriter(fileName);
            BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

            bufferedWriter.write(fireEvent.versionText());
            bufferedWriter.newLine();
            bufferedWriter.newLine();

            bufferedWriter.write("TIME");
            for (int i = 0; i < positions.size(); i++) {
                bufferedWriter.write(",");
                bufferedWriter.write("x = ");
                bufferedWriter.write(Double.toString(positions.get(i)));
                bufferedWriter.write(" m");
            }
            for (int i = 0; i < time.size(); i++) {
                bufferedWriter.newLine();
                bufferedWriter.write(Integer.toString(time.get(i)));
                for (int j = 0; j < positions.size(); j++) {
                    bufferedWriter.write(",");
                    temp = fireEvent.getVisibility(i, positions.get(j));
                    if (temp == -1.0) {
                        bufferedWriter.write(" ");
                    } else {
                        bufferedWriter.write(Double.toString(temp));
                    }
                }
            }
            bufferedWriter.close();
        }

        catch (IOException ex) {
        System.out.println(
            "Error writing to file '"
            + fileName + "'");
        }
    }

    /** Generates output file related to average heat
     * transfer in tunnel for pre-defined positions
     * downstream the fire for every s of the
     * fire development. */
    public void printHeatTransferToFile() {
        String fileName = "tunnelHeatTransfer.txt";
        double temp = 0.0;

        try {
            FileWriter fileWriter = new FileWriter(fileName);
            BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

            bufferedWriter.write(fireEvent.versionText());
            bufferedWriter.newLine();
            bufferedWriter.newLine();

            bufferedWriter.write("TIME");
            for (int i = 0; i < positions.size(); i++) {
                bufferedWriter.write(",");
                bufferedWriter.write("x = ");
                bufferedWriter.write(Double.toString(positions.get(i)));
                bufferedWriter.write(" m");
            }
            for (int i = 0; i < time.size(); i++) {
                bufferedWriter.newLine();
                bufferedWriter.write(Integer.toString(time.get(i)));
                for (int j = 0; j < positions.size(); j++) {
```

```
                        bufferedWriter.write(",");
                        temp = fireEvent.getHeatTransport(i, positions.get(j));
                        bufferedWriter.write(Double.toString(temp));
                  }
            }
            bufferedWriter.close();
      }

      catch (IOException ex) {
      System.out.println(
          "Error writing to file '"
          + fileName + "'");
      }
}

/** Generates output file related to average radiation
 * flux in tunnel for pre-defined positions
 * downstream the fire for every s of the
 * fire development. */
public void printFireRadiationToFile() {
      String fileName = "tunnelFireRadiation.txt";
      double temp = 0.0;

      try {
            FileWriter fileWriter = new FileWriter(fileName);
            BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

            bufferedWriter.write(fireEvent.versionText());
            bufferedWriter.newLine();
            bufferedWriter.newLine();

            bufferedWriter.write("TIME");
            for (int i = 0; i < positions.size(); i++) {
                  bufferedWriter.write(",");
                  bufferedWriter.write("x = ");
                  bufferedWriter.write(Double.toString(positions.get(i)));
                  bufferedWriter.write(" m");
            }
            for (int i = 0; i < time.size(); i++) {
                  bufferedWriter.newLine();
                  bufferedWriter.write(Integer.toString(time.get(i)));
                  for (int j = 0; j < positions.size(); j++) {
                        bufferedWriter.write(",");
                        temp = fireEvent.getRadiationFromFire(i, positions.get(j)) / 1000.0;
                        bufferedWriter.write(Double.toString(temp));
                  }
            }
            bufferedWriter.close();
      }

      catch (IOException ex) {
      System.out.println(
          "Error writing to file '"
          + fileName + "'");
      }
}

/** Generates output file related to average backlayering
 * distance in tunnel for every s of the
 * fire development. */
public void printBackLayeringDistance() {
      String fileName = "tunnelBacklayeringDistance.txt";
      double temp = 0.0;

      try {
            FileWriter fileWriter = new FileWriter(fileName);
            BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

            bufferedWriter.write(fireEvent.versionText());
            bufferedWriter.newLine();
            bufferedWriter.newLine();

            bufferedWriter.write("TIME");
            bufferedWriter.write(",");
            bufferedWriter.write("DISTANCE [m]");
            for (int i = 0; i < time.size(); i++) {
                  bufferedWriter.newLine();
                  bufferedWriter.write(Integer.toString(time.get(i)));
                  bufferedWriter.write(",");
```

```
                        temp = fireEvent.getBackLayeringLength(i);
                        bufferedWriter.write(Double.toString(temp));
                    }
                    bufferedWriter.close();
                }

                catch (IOException ex) {
                System.out.println(
                    "Error writing to file '"
                    + fileName + "'");
                }
        }

}
```

C.5 Person.java

```
import java.util.ArrayList;
import java.util.Random;

/** This Java class is a part of TuFT, a decision support
 * tool for tunnel fires. The tool was developed by Karl Fridolf
 * and Håkan Frantzich at Lund University during 2013-2014.
 * Among other things, the tool offers the possibility to
 * simulate fires in tunnels, and furthermore, the effects
 * on evacuation and rescue operation possibilities.
 *
 * The tool's sub models, as well as the conclusions and
 * the results produced by the tool, have not been quality
 * controlled, verified or validated to the extent necessary
 * for quality assurance. TuFT must hence be used with caution.
 * Anyone who uses the results produced by TuFT does so on
 * his/her own responsibility */
public class Person {
      private FireEvent fireEvent;
      private double startPos;
      private ArrayList<Double> position;
      private ArrayList<Double> fedAsphyxia;
      private double sumFedAsphyxia;
      private ArrayList<Double> fedHeat;
      private double sumFedHeat;
      private ArrayList<Double> fldHeat;
      private double sumFldHeat;
      private int responseTime;
      private int recognitionTime;
      private int timeToLeaveVehicle;     //Give for road tunnel, leave to 0 for train tunnel.
      private boolean towardsTunnelExit;
      private double safePosition;
      private ArrayList<Double> movementSpeed;
      private boolean incapacitated;
      private boolean dead;
      private boolean chooseEmergencyExit;
      private boolean deterministic;
      private ArrayList<Double> temperatures;
      private ArrayList<Double> visibilities;
      private ArrayList<Double> gasConcentrationsCo;
      private ArrayList<Double> gasConcentrationsCo2;
      private ArrayList<Double> gasConcentrationsHcn;
      private ArrayList<Double> gasConcentrationsO2;

      /** Creates a person with a start position startPos in
       * the in a tunnel Tunnel (0 < startPos < tunnel length).
       * The person has a pre-defined recognition time as well
       * as a response time specified in s. Time to leave vehicle
       * is independent of recognition and response times (a person
       * may exit his/her vehicle before deciding to evacuate).
       * If the person is supposed to evacuate through the
       * nearest emergency exit (always away from the fire),
       * chooseEmergencyExit is true, else false.
       * The person is evacuating in a tunnel fire
       * event fireEvent.
       *
       *  Modelling of movement speed is done deterministic or
       *  probabilistic according to Fridolf et al. (2014)
       *  (doi: 10.1002/fam.2217) Table 2 and Figure 7, depending
```

```
 *  on if determinstic is true or false (= probabilistic). */
public Person(double startPos, int recognitionTime,
            int responseTime, int timeToLeaveVehicle,
            boolean chooseEmergencyExit, boolean deterministic,
            Tunnel tunnel, FireEvent fireEvent) {
        this.startPos = startPos;
        this.recognitionTime = recognitionTime;
        this.responseTime = responseTime;
        this.timeToLeaveVehicle = timeToLeaveVehicle;
        this.chooseEmergencyExit = chooseEmergencyExit;
        this.deterministic = deterministic;
        this.fireEvent = fireEvent;
        position = new ArrayList<Double>();
        position.add(0, startPos);
        fedAsphyxia = new ArrayList<Double>();
        fedAsphyxia.add(0, 0.0);
        fedHeat = new ArrayList<Double>();
        fedHeat.add(0, 0.0);
        fldHeat = new ArrayList<Double>();
        fldHeat.add(0, 0.0);
        incapacitated = false;
        dead = false;
        movementSpeed = new ArrayList<Double>();

        temperatures = new ArrayList<Double>();
        temperatures.add(0, tunnel.getTemp() - 273.0);
        visibilities = new ArrayList<Double>();
        visibilities.add(0, Double.POSITIVE_INFINITY);
        gasConcentrationsCo = new ArrayList<Double>();
        gasConcentrationsCo.add(0, 0.0);
        gasConcentrationsCo2 = new ArrayList<Double>();
        gasConcentrationsCo2.add(0, 0.0);
        gasConcentrationsHcn = new ArrayList<Double>();
        gasConcentrationsHcn.add(0, 0.0);
        gasConcentrationsO2 = new ArrayList<Double>();
        gasConcentrationsO2.add(0, 0.2095);

        // DETERMINES MOVEMENT SPEED //
        if (deterministic) {
              movementSpeed.add(1.3);
              movementSpeed.add(1.0);
              movementSpeed.add(0.8);
              movementSpeed.add(0.7);
              movementSpeed.add(0.55);
              movementSpeed.add(0.3);
              movementSpeed.add(0.2);
        } else {
              Random rand = new Random();
              double t = rand.nextGaussian();
              movementSpeed.add((t * 0.1) + 1.3);
              movementSpeed.add((t * 0.11) + 1.02);
              movementSpeed.add((t * 0.26) + 1.00);
              movementSpeed.add((t * 0.18) + 0.83);
              movementSpeed.add((t * 0.27) + 0.78);
              movementSpeed.add((t * 0.17) + 0.42);
              movementSpeed.add((t * 0.1) + 0.2);
        }

        // DETERMINES SAFE LOCATION //
        if (chooseEmergencyExit) {
              if (startPos > fireEvent.getDistanceFromEntrance()) {
                    safePosition = tunnel.getLength();
                    towardsTunnelExit = true;
                    for (int i = 0; i < tunnel.getEmergencyExits().size(); i++) {
                          if (tunnel.getEmergencyExits().get(i) >
                          fireEvent.getDistanceFromEntrance()) {
                                if (tunnel.getEmergencyExits().get(i) > startPos) {
                                      if (tunnel.getEmergencyExits().get(i) <
                                                  tunnel.getLength()) {
                                            if (Math.abs(startPos -
tunnel.getEmergencyExits().get(i)) <
                                                        Math.abs(startPos - safePosition)) {
                                                  safePosition = tunnel.getEmergencyExits().get(i);
                                            }
                                      }
                                }
                          }
                    }
              } else if (startPos < fireEvent.getDistanceFromEntrance()) {
```

```
                        safePosition = 0.0;
                        towardsTunnelExit = false;
                        for (int i = 0; i < tunnel.getEmergencyExits().size(); i++) {
                              if (tunnel.getEmergencyExits().get(i) <
                                          fireEvent.getDistanceFromEntrance()) {
                                    if (tunnel.getEmergencyExits().get(i) < startPos) {
                                          if (tunnel.getEmergencyExits().get(i) > 0.0) {
                                                if (Math.abs(startPos -
tunnel.getEmergencyExits().get(i)) <
                                                            Math.abs(startPos - safePosition)) {
                                                      safePosition = tunnel.getEmergencyExits().get(i);
                                                }
                                          }
                                    }
                              }
                        }
                  }
            } else if (!chooseEmergencyExit) {
                  if (startPos > fireEvent.getDistanceFromEntrance()) {
                        safePosition = tunnel.getLength();
                        towardsTunnelExit = true;
                  } else if (startPos < fireEvent.getDistanceFromEntrance()) {
                        safePosition = 0.0;
                        towardsTunnelExit = false;
                  }
            }
      }

      /** Returns the start position in m (relative
       * to the tunnel entrance) of the person. */
      public double getStartPosition() {
            return startPos;
      }

      /** Returns array of saved average gas temperatures in °C
       * based on the position of the evacuee for each time step
       * of the evacuation. */
      public double getTemperature(int t) {
            return temperatures.get(t);
      }

      /** Adds an average gas temperatures in °C
       * based on the position of the evacuee for time step t s
       * and position x m in tunnel (relative to fire). */
      public void addTemperature(int t, double x) {
            temperatures.add(t, x);
      }

      /** Returns array of saved average visibilities in m
       * based on the position of the evacuee for each time step
       * of the evacuation. */
      public double getVisibility(int t) {
            return visibilities.get(t);
      }

      /** Adds an average visibility in m
       * based on the position of the evacuee for time step t s
       * and position x m in tunnel (relative to fire). */
      public void addVisibility(int t, double x) {
            visibilities.add(t, x);
      }

      /** Returns array of saved average gas concentrations for CO2 in vol-%
       * based on the position of the evacuee for each time step
       * of the evacuation. */
      public double getConcentrationCo2(int t) {
            return gasConcentrationsCo2.get(t);
      }

      /** Adds an average gas concentration of CO2 in vol-%
       * based on the position of the evacuee for time step t s
       * and position x m in tunnel (relative to fire). */
      public void addConcentrationCo2(int t, double x) {
            gasConcentrationsCo2.add(t, x);
      }

      /** Returns array of saved average gas concentrations for CO in vol-%
       * based on the position of the evacuee for each time step
       * of the evacuation. */
```

```java
public double getConcentrationCo(int t) {
	return gasConcentrationsCo.get(t);
}

/** Adds an average gas concentration of CO in vol-%
 * based on the position of the evacuee for time step t s
 * and position x m in tunnel (relative to fire). */
public void addConcentrationCo(int t, double x) {
	gasConcentrationsCo.add(t, x);
}

/** Returns array of saved average gas concentrations for HCN in vol-%
 * based on the position of the evacuee for each time step
 * of the evacuation. */
public double getConcentrationHcn(int t) {
	return gasConcentrationsHcn.get(t);
}

/** Adds an average gas concentration of HCN in vol-%
 * based on the position of the evacuee for time step t s
 * and position x m in tunnel (relative to fire). */
public void addConcentrationHcn(int t, double x) {
	gasConcentrationsHcn.add(t, x);
}

/** Returns array of saved average gas concentrations for O2 in vol-%
 * based on the position of the evacuee for each time step
 * of the evacuation. */
public double getConcentrationO2(int t) {
	return gasConcentrationsO2.get(t);
}

/** Adds an average gas concentration of O2 in vol-%
 * based on the position of the evacuee for time step t s
 * and position x m in tunnel (relative to fire). */
public void addConcentrationO2(int t, double x) {
	gasConcentrationsO2.add(t, x);
}

/** Returns the position in m (relative
 * to the tunnel entrance) of the person at a
 * defined time in s. */
public double getPosition(int t) {
	return position.get(t);
}

/** Returns the latest position in m (relative
 * to the tunnel entrance) of the person. */
public double getLastPosition() {
	return position.get(position.size() - 1);
}

/** Adds a position x in m (relative to the
 * tunnel entrance) at a time t in s. */
public void addPosition(int t, double x) {
	position.add(t, x);
}

/** Adds a fraction of an incapacitating dose
 * of all asphyxiant gases at a time in s. The
 * calculation must be performed in a main method,
 * as this operation only adds the value. */
public void addFedAsphyxia(int t, double fed) {
	fedAsphyxia.add(t, fed);
}

/** Returns the fraction of an incapacitating dose
 * of all asphyxiant gases at a time in s. */
public double getFedAsphyxia(int t) {
	return fedAsphyxia.get(t);
}

/** Returns the accumulated fraction of an
 * incapacitating dose of all asphyxiant gases
 * at a time in s. */
public double getAccumulatedFedAsphyxia(int t) {
	double tempSum = 0.0;
	for(int i = 0; i < (t + 1); i++) {
		tempSum += fedAsphyxia.get(i);
```

```
        }
        return tempSum;
}

/** Returns the sum of all fractions of incapacitating
 * doses related to asphyxiant gases. */
public double getSumFedAsphyxia() {
        sumFedAsphyxia = 0.0;
        for(int i = 0; i < fedAsphyxia.size(); i++) {
                sumFedAsphyxia += fedAsphyxia.get(i);
        }
        return sumFedAsphyxia;
}

/** Adds a fraction of an incapacitating dose
 * of heat at a time in s. The calculation must be
 * performed in a main method,as this operation
 * only adds the value. */
public void addFIDHeat(int t, double fed) {
        fedHeat.add(t, fed);
}

/** Returns the fraction of an incapacitating dose
 * of heat at a time in s. */
public double getFIDHeat(int t) {
        return fedHeat.get(t);
}

/** Returns the accumulated fraction of an
 * incapacitating dose heat at a time in s. */
public double getAccumulatedFIDHeat(int t) {
        double tempSum = 0.0;
        for(int i = 0; i < (t + 1); i++) {
                tempSum += fedHeat.get(i);
        }
        return tempSum;
}

/** Returns the sum of all fractions of incapacitating
 * doses related to heat. */
public double getSumFIDHeat() {
        sumFedHeat = 0.0;
        for(int i = 0; i < fedHeat.size(); i++) {
                sumFedHeat += fedHeat.get(i);
        }
        return sumFedHeat;
}

/** Adds a fraction of a lethal dose
 * of heat at a time in s. The calculation must be
 * performed in a main method,as this operation
 * only adds the value. */
public void addFLDHeat(int t, double fed) {
        fldHeat.add(t, fed);
}

/** Returns the fraction of a lethal dose
 * of heat at a time in s. */
public double getFLDHeat(int t) {
        return fldHeat.get(t);
}

/** Returns the accumulated fraction of a lethal
 * dose heat at a time in s. */
public double getAccumulatedFLDHeat(int t) {
        double tempSum = 0.0;
        for(int i = 0; i < (t + 1); i++) {
                tempSum += fldHeat.get(i);
        }
        return tempSum;
}

/** Returns the sum of all fractions of lethal
 * doses related to heat. */
public double getSumFLDHeat() {
        sumFldHeat = 0.0;
        for(int i = 0; i < fldHeat.size(); i++) {
                sumFldHeat += fldHeat.get(i);
        }
```

```
        return sumFldHeat;
    }

    /** Returns the person's recognition time in s. */
    public int getRecognitionTime() {
        return recognitionTime;
    }

    /** Returns the person's response time in s. */
    public int getResponseTime() {
        return responseTime;
    }

    /** Returns the movement speed in m/s of the person for a
     * the given visibility condition. Input is time in s
     * and the position x of the person relative to the fire
     * in m. Thus, note that it is not the actual position of
     * the person but the relative position compared to the
     * fire. */
    public double getMovementSpeed(int t, double x) {
        double visibility = fireEvent.getVisibility(t, x);
        double movementSpeedTemp = 0.0;
        if (visibility >= 5.0 || visibility == -1.0) {
            movementSpeedTemp = movementSpeed.get(0);
        } else if (visibility < 5.0 && visibility >= 2.0) {
            movementSpeedTemp = movementSpeed.get(1);
        } else if (visibility < 2.0 && visibility >= 1.39) {
            movementSpeedTemp = movementSpeed.get(2);
        } else if (visibility < 1.39 && visibility >= 1.11) {
            movementSpeedTemp = movementSpeed.get(3);
        } else if (visibility < 1.11 && visibility >= 0.83) {
            movementSpeedTemp = movementSpeed.get(4);
        } else if (visibility < 0.83 && visibility >= 0.55) {
            movementSpeedTemp = movementSpeed.get(5);
        } else if (visibility < 0.55 && visibility > 0) {
            movementSpeedTemp = movementSpeed.get(6);
        }
        return movementSpeedTemp;
    }

    /** Returns the unimpeded movement speed in m/s
     * of the person. */
    public double getUnimpededMovementSpeed() {
        return movementSpeed.get(0);
    }

    /** Returns the evacuation direction of the person. If
     * true, the person is evacuating towards the tunnel exit,
     * and if false, the person is evacuating towards the tunnel
     * entrance. Note, the person can still be headed for an
     * emergency exit, but direction is always defined in relation
     * to the tunnel entrance or the tunnel exit. */
    public boolean evacuationToTunnelExit() {
        return towardsTunnelExit;
    }

    /** Returns the nearest safe position, either the closest
     * emergency exit (away from the fire) or the tunnel
     * entrance/exit.*/
    public double getSafePosition() {
        return safePosition;
    }

    /** Returns an answer to whether the person is downstream
     * (true) or upstream (false) the fire. */
    public boolean isPersonDownstreamFire() {
        boolean isPersonDownstreamFireTemp;
        if (((startPos > fireEvent.getDistanceFromEntrance()) &&
                    fireEvent.getWindDirection()) ||
                    ((startPos < fireEvent.getDistanceFromEntrance()) &&
                                !fireEvent.getWindDirection())) {
            isPersonDownstreamFireTemp = true;
        } else {
            isPersonDownstreamFireTemp = false;
        }
        return isPersonDownstreamFireTemp;
    }

    /** Is person incapacitated? If true, the answer is yes.
```

```
 * If false, the answer is no. A person is defined
 * incapacitated if any FID-value related to
 * asphyxiants and heat is >= 1.0.
 *
 * Assumption is based on text in Section 2,
 * Chapter 6 of SFPE handbook (4th Ed.), see for example
 * discussion on p. 2-148 among others. */
public boolean isIncapacitated() {
      if (getSumFedAsphyxia() < 1.0 &&
                  getSumFIDHeat() < 1.0) {
            incapacitated = false;
      } else {
            incapacitated = true;
      }
      return incapacitated;
}

/** Is person dead? If true, the answer is yes. If false,
 * the answer is no. A person is defined
 * dead if any FID-value related to
 * asphyxiants and heat is >= 2.0.
 *
 * Assumption is based on text in Section 2,
 * Chapter 6 of SFPE handbook (4th Ed.), see for example
 * discussion on p. 2-148 among others. */
public boolean isDead() {
      if (getSumFedAsphyxia() < 2.0 && getSumFLDHeat() < 1.0) {
            dead = false;
      } else {
            dead = true;
      }
      return dead;
}

/** Did person succesfully evacuate the tunnel?
 * If true, the answer is yes (the person
 * managed to evacuate). If false, the answer is no, and
 * the person has not yet reached a safe position. */
public boolean isEvacuated() {
      boolean isEvacuated;
      if (towardsTunnelExit) {
            if (this.getLastPosition() < safePosition) {
                  isEvacuated = false;
            } else {
                  isEvacuated = true;
            }
      } else {
            if (this.getLastPosition() > safePosition) {
                  isEvacuated = false;
            } else {
                  isEvacuated = true;
            }
      }
      return isEvacuated;
}

/** Returns the time in s until the simulation ended for this
 * person. It could either be the time to death, or the
 * time it took the person to reach a safe position (however,
 * note that this function doesn't gives the reason, only the
 * time in s).*/
public int timeToTheEnd() {
      return position.size() - 1;
}

/** Returns the array of defined movement speeds for the
 * person */
public ArrayList<Double> getMovementSpeedTotal() {
      return movementSpeed;
}

/** Returns exit preference of the evacuee. */
public boolean getExitPreference() {
      return chooseEmergencyExit;
}

/** Returns walking speed calculation method
 * of the evacuee. */
public boolean getDeterministic() {
```

```
            return deterministic;
      }

      /** Returns the person's time to leave vehicle in s. */
      public int getTimeToLeaveVehicle() {
            return timeToLeaveVehicle;
      }

}
```

C.6 Group.java

```
import java.util.ArrayList;
import java.util.Random;

/** This Java class is a part of TuFT, a decision support
 * tool for tunnel fires. The tool was developed by Karl Fridolf
 * and Håkan Frantzich at Lund University during 2013-2014.
 * Among other things, the tool offers the possibility to
 * simulate fires in tunnels, and furthermore, the effects
 * on evacuation and rescue operation possibilities.
 *
 * The tool's sub models, as well as the conclusions and
 * the results produced by the tool, have not been quality
 * controlled, verified or validated to the extent necessary
 * for quality assurance. TuFT must hence be used with caution.
 * Anyone who uses the results produced by TuFT does so on
 * his/her own responsibility */
public class Group {
      private int count;
      private double startPosIntervalFrom;
      private double startPosIntervalTo;
      private int recognitionTimeIntervalFrom;
      private int recognitionTimeIntervalTo;
      private int responseTimeIntervalFrom;
      private int responseTimeIntervalTo;
      private int timeToLeaveVehicleIntervalFrom;
      private int timeToLeaveVehicleIntervalTo;
      private boolean chooseEmergencyExit;
      private boolean deterministic;
      private Tunnel tunnel;
      private FireEvent fireEvent;
      private ArrayList<Person> people;

      /** Creates a group (array) of count number of people in
       * the tunnel Tunnel interval measured from startPosIntervalFrom
       * to startPosIntervalTo. People are randomly (uniform)
       * distributed within the interval.
       *
       * Recognition and response times are also randomly
       * (uniform) distributed within the defined intervals
       * for respectively parameter.
       *
       * chooseEmergencyExit determines whether or not the group
       * of people will use emergency exits or not (if available).
       *
       * Modelling of movement speed is done deterministic or
       * probabilistic according to Fridolf et al. (2014)
       * (doi: 10.1002/fam.2217) Table 2 and Figure 7, depending
       * on if determinstic is true or false (= probabilistic). Values
       * are assigned to each person individually. */
      public Group(int count, double startPosIntervalFrom,
                  double startPosIntervalTo, int recognitionTimeIntervalFrom,
                  int recognitionTimeIntervalTo, int responseTimeIntervalFrom,
                  int responseTimeIntervalTo,   int timeToLeaveVehicleIntervalFrom,
                  int timeToLeaveVehicleIntervalTo, boolean chooseEmergencyExit,
                  boolean deterministic, Tunnel tunnel, FireEvent fireEvent) {
            this.count = count;
            this.startPosIntervalFrom = startPosIntervalFrom;
            this.startPosIntervalTo = startPosIntervalTo;
            this.recognitionTimeIntervalFrom = recognitionTimeIntervalFrom;
            this.recognitionTimeIntervalTo = recognitionTimeIntervalTo;
            this.responseTimeIntervalFrom =  responseTimeIntervalFrom;
            this.responseTimeIntervalTo = responseTimeIntervalTo;
```

```
        this.timeToLeaveVehicleIntervalFrom = timeToLeaveVehicleIntervalFrom;
        this.timeToLeaveVehicleIntervalTo = timeToLeaveVehicleIntervalTo;
        this.chooseEmergencyExit = chooseEmergencyExit;
        this.deterministic = deterministic;
        this.tunnel = tunnel;
        this.fireEvent = fireEvent;

        people = new ArrayList<Person>();
        Random rand = new Random();

        for (int i = 0; i < count; i++) {
              double t = rand.nextDouble();
              double personPosition = (startPosIntervalFrom * (1 - t)) +
                          (startPosIntervalTo * t);
              double u = rand.nextDouble();
              double recognitionTime = (recognitionTimeIntervalFrom * (1 - u)) +
                          (recognitionTimeIntervalTo * u);
              double v = rand.nextDouble();
              double responseTime = (responseTimeIntervalFrom * (1 - v)) +
                          (responseTimeIntervalTo * v);
              double w = rand.nextDouble();
              double timeToLeaveVehicle = (timeToLeaveVehicleIntervalFrom * (1 - w)) +
                          (timeToLeaveVehicleIntervalTo * w);
              if (timeToLeaveVehicle > (recognitionTime + responseTime)) {
                    timeToLeaveVehicle = recognitionTime + responseTime;
              }

              people.add(new Person(personPosition,
                          (int) Math.round(recognitionTime),
                          (int) Math.round(responseTime),
                          (int) Math.round(timeToLeaveVehicle),
                          chooseEmergencyExit, deterministic,
                          tunnel, fireEvent));
        }
  }

  /** Returns the number of individuals in the group. */
  public int getCount() {
        return count;
  }

  /** Returns first value of start position interval for the group (m). */
  public double getStartPosIntervalFrom() {
        return startPosIntervalFrom;
  }

  /** Returns last value of start position interval for the group (m). */
  public double getStartPosIntervalTo() {
        return startPosIntervalTo;
  }

  /** Returns first value of recognition time interval for the group (s). */
  public int getRecognitionTimeIntervalFrom() {
        return recognitionTimeIntervalFrom;
  }

  /** Returns last value of recognition time interval for the group (s). */
  public int getRecognitionTimeIntervalTo() {
        return recognitionTimeIntervalTo;
  }

  /** Returns first value of response time interval for the group (s). */
  public int getResponseTimeIntervalFrom() {
        return responseTimeIntervalFrom;
  }

  /** Returns last value of response time interval for the group (s). */
  public int getResponseTimeIntervalTo() {
        return responseTimeIntervalTo;
  }

  /** Returns first value of time to leave vehicle interval for the group (s). */
  public int getTimeToLeaveVechicleFrom() {
        return timeToLeaveVehicleIntervalFrom;
  }

  /** Returns last value of time to leave vehicle interval for the group (s). */
  public int getTimeToLeaveVechicleTo() {
        return timeToLeaveVehicleIntervalTo;
```

```
	}

	/** Returns exit choice preference for the individuals of the group. */
	public boolean getChooseEmergencyExit() {
		return chooseEmergencyExit;
	}

	/** Returns walking speed calculation method for the individuals of the group. */
	public boolean getDeterministic() {
		return deterministic;
	}

	/** Returns current tunnel. */
	public Tunnel getTunnel() {
		return tunnel;
	}

	/** Returns current fire event. */
	public FireEvent getFireEvent() {
		return fireEvent;
	}

	/** Returns the array of persons. */
	public ArrayList<Person> getPeople() {
		return people;
	}

}
```

C.7 Train.java

```
import java.util.ArrayList;

/** This Java class is a part of TuFT, a decision support
 * tool for tunnel fires. The tool was developed by Karl Fridolf
 * and Håkan Frantzich at Lund University during 2013-2014.
 * Among other things, the tool offers the possibility to
 * simulate fires in tunnels, and furthermore, the effects
 * on evacuation and rescue operation possibilities.
 *
 * The tool's sub models, as well as the conclusions and
 * the results produced by the tool, have not been quality
 * controlled, verified or validated to the extent necessary
 * for quality assurance. TuFT must hence be used with caution.
 * Anyone who uses the results produced by TuFT does so on
 * his/her own responsibility */
public class Train {
	private double position;
	private double length;
	private int numberOfExits;
	private double exitWidth;
	private int numberOfPassengers;
	int recognitionTime;
	int responseTime;
	boolean chooseEmergencyExit;
	boolean deterministic;
	Tunnel tunnel;
	FireEvent fireEvent;
	private ArrayList<Double> trainExitPositions;
	private ArrayList<Person> passengers;

	/** Creates a train with one end at position x m from
	 * tunnel portal and the other end at position + length m.
	 * Number of exits determines the number of available exits from
	 * the train and is evenly distributed on one side of the train.
	 *
	 * Exit width determines how quickly the train is emptied, number
	 * of passengers the count of individuals on the train. Recognition and
	 * response times defined for all individuals. Time to exit vehicle (train)
	 * is dependent on how quickly each individual exits the train. In a train,
	 * no individuals can leave the train prior to recognition and response
	 * times over. Exit choice and walking speed calculation done
	 * with same assumptions as for individuals and defined similarly. */
```

```
	public Train(double position, double length, int numberOfExits,
			double exitWidth, int numberOfPassengers, int recognitionTime,
			int responseTime, boolean chooseEmergencyExit, boolean deterministic,
			Tunnel tunnel, FireEvent fireEvent) {
		this.position = position;
		this.length = length;
		this.numberOfExits = numberOfExits;
		this.exitWidth = exitWidth;
		this.numberOfPassengers = numberOfPassengers;
		this.recognitionTime = recognitionTime;
		this.responseTime = responseTime;
		this.chooseEmergencyExit = chooseEmergencyExit;
		this.deterministic = deterministic;
		this.tunnel = tunnel;
		this.fireEvent = fireEvent;

				// Calculates positions of train exits //
		trainExitPositions = new ArrayList<Double>();
		double distance = (length / (numberOfExits + 1));
		for (int i = 0; i < numberOfExits; i++) {
			trainExitPositions.add(position + (distance * (double) (i + 1)));
		}

		// Creates each individual //
		passengers = new ArrayList<Person>();
		int passengersPerExit = (numberOfPassengers / numberOfExits);
		int temp = 0;
		int j = 0;
		double startPos = 0;
		int cumTrainExitTime = 0;
		for (int i = 0; i < numberOfPassengers; i++) {
			j++;
			startPos = trainExitPositions.get(temp);
			cumTrainExitTime += (1 / (0.2 * exitWidth)); // Fridolf, K., Nilsson, D., & Frantzich,
H. (2014). The Flow Rate of People during Train Evacuation in Rail Tunnels: Effects of Different Train
Exit Configurations. Safety Science, 62(C), 515-529. doi: 10.1016/j.ssci.2013.10.008
			passengers.add(new Person(startPos, recognitionTime,
				(responseTime + cumTrainExitTime), (recognitionTime + responseTime +
cumTrainExitTime),
				chooseEmergencyExit, deterministic,
				tunnel, fireEvent));
			if (j > passengersPerExit) {
				j = 0;
				temp++;
				cumTrainExitTime = 0;
			}
		}
	}

	/** Returns position of the train (closest to entrance) in m. */
	public double getTrainPosition() {
		return position;
	}

	/** Returns length of train in m. */
	public double getTrainLength() {
		return length;
	}

	/** Returns no of train exits. */
	public int getNumberOfTrainExits() {
		return numberOfExits;
	}

	/** Returns width of train exits. */
	public double getTrainExitWidth() {
		return exitWidth;
	}

	/** Returns no of passengers on board train. */
	public int getNumberOfPassengers() {
		return numberOfPassengers;
	}

	/** Returns array including all train passengers. */
	public ArrayList<Person> getArrayOfPassengers() {
		return passengers;
	}
```

```
	/** Returns passengers recognition time in s. */
	public int getRecognitionTime() {
		return recognitionTime;
	}

	/** Returns passengers response time in s. */
	public int getResponseTime() {
		return responseTime;
	}

	/** Returns passengers exit choice preference. */
	public boolean getChooseEmergencyExit() {
		return chooseEmergencyExit;
	}

	/** Returns modelling of walking speed method. */
	public boolean deterministic() {
		return deterministic;
	}

}
```

C.8 EvacSim.java

```
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Collections;
import java.io.*;

/** This Java class is a part of TuFT, a decision support
 * tool for tunnel fires. The tool was developed by Karl Fridolf
 * and Håkan Frantzich at Lund University during 2013-2014.
 * Among other things, the tool offers the possibility to
 * simulate fires in tunnels, and furthermore, the effects
 * on evacuation and rescue operation possibilities.
 *
 * The tool's sub models, as well as the conclusions and
 * the results produced by the tool, have not been quality
 * controlled, verified or validated to the extent necessary
 * for quality assurance. TuFT must hence be used with caution.
 * Anyone who uses the results produced by TuFT does so on
 * his/her own responsibility */
public class EvacSim {
	private Fire fire;
	private FireEvent fireEvent;
	private ArrayList<Person> persons;
	private int preMovementTime;
	private int timeToLeaveVehicle;
	private boolean downstreamFire;
	private boolean towardsTunnelExit;
	private double tempFedAsph;
	private double tempFIDHeat;
	private double tempFLDHeat;
	private double tempMovementSpeed;
	private double tempDistance;
	private double tempPosition;
	private boolean tempIncapacitated;

	/** Prepares evacuation simulation */
	public EvacSim(Fire fire, FireEvent fireEvent, ArrayList<Person> persons) {
		this.fire = fire;
		this.fireEvent = fireEvent;
		this.persons = persons;
		preMovementTime = 0;
		timeToLeaveVehicle = 0;
		downstreamFire = true;
		towardsTunnelExit = true;
		tempFedAsph = 0.0;
		tempFIDHeat = 0.0;
		tempFLDHeat = 0.0;
		tempMovementSpeed = 0.0;
		tempDistance = 0.0;
		tempPosition = 0.0;
		tempIncapacitated = false;
	}
```

```
	/** Performs evacuation simulation */
	public void performEvacSim() {
		for (int i = 0; i < persons.size(); i++) {
			downstreamFire = persons.get(i).isPersonDownstreamFire();
			towardsTunnelExit = persons.get(i).evacuationToTunnelExit();
			preMovementTime = persons.get(i).getRecognitionTime() +
persons.get(i).getResponseTime();
			timeToLeaveVehicle = persons.get(i).getTimeToLeaveVehicle();
			for (int t = 1; t < fire.getTotalTime(); t++) {
				if (downstreamFire) {
					tempIncapacitated = persons.get(i).isIncapacitated();
					if (t < preMovementTime) {
						if (t < timeToLeaveVehicle) {
							tempPosition = persons.get(i).getPosition(t - 1);
							persons.get(i).addPosition(t, tempPosition);
							persons.get(i).addFedAsphyxia(t, 0.0);
							persons.get(i).addFIDHeat(t, 0.0);
							persons.get(i).addFLDHeat(t, 0.0);
							if (fireEvent.getTavgX(t,
Math.abs(fireEvent.getDistanceFromEntrance() - persons.get(i).getPosition(t))) == -1.0) {
								persons.get(i).addTemperature(t,
fireEvent.getTunnel().getTemp() - 273.0);
								persons.get(i).addVisibility(t,
Double.POSITIVE_INFINITY);
								persons.get(i).addConcentrationCo(t, 0.0);
								persons.get(i).addConcentrationCo2(t, 0.0);
								persons.get(i).addConcentrationHcn(t, 0.0);
								persons.get(i).addConcentrationO2(t, 0.2095);
							} else {
								persons.get(i).addTemperature(t, fireEvent.getTavgX(t,
Math.abs(fireEvent.getDistanceFromEntrance() - persons.get(i).getPosition(t))));
								persons.get(i).addVisibility(t,
fireEvent.getVisibility(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
								persons.get(i).addConcentrationCo(t,
fireEvent.getMoleFractionCO(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
								persons.get(i).addConcentrationCo2(t,
fireEvent.getMoleFractionCO2(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
								persons.get(i).addConcentrationHcn(t,
fireEvent.getMoleFractionHCN(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
								persons.get(i).addConcentrationO2(t,
fireEvent.getMoleFractionO2(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
							}
						}
						else {
							tempPosition = persons.get(i).getPosition(t - 1);
							persons.get(i).addPosition(t, tempPosition);
							tempFedAsph =
fireEvent.getFractionOfIncapacitationAsphyxiants(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t)), tempIncapacitated);
							persons.get(i).addFedAsphyxia(t, tempFedAsph);
							tempFIDHeat =
fireEvent.getFractionOfIncapacitationHeatDownstream(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t)));
							persons.get(i).addFIDHeat(t, tempFIDHeat);
							tempFLDHeat = fireEvent.getFractionOfLethalHeatDownstream(t,
Math.abs(fireEvent.getDistanceFromEntrance() - persons.get(i).getPosition(t)));
							persons.get(i).addFLDHeat(t, tempFLDHeat);
							if (fireEvent.getTavgX(t,
Math.abs(fireEvent.getDistanceFromEntrance() - persons.get(i).getPosition(t))) == -1.0) {
								persons.get(i).addTemperature(t,
fireEvent.getTunnel().getTemp() - 273.0);
								persons.get(i).addVisibility(t,
Double.POSITIVE_INFINITY);
								persons.get(i).addConcentrationCo(t, 0.0);
								persons.get(i).addConcentrationCo2(t, 0.0);
								persons.get(i).addConcentrationHcn(t, 0.0);
								persons.get(i).addConcentrationO2(t, 0.2095);
							} else {
								persons.get(i).addTemperature(t, fireEvent.getTavgX(t,
Math.abs(fireEvent.getDistanceFromEntrance() - persons.get(i).getPosition(t))));
								persons.get(i).addVisibility(t,
fireEvent.getVisibility(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
```

```
                                                persons.get(i).addConcentrationCo(t,
fireEvent.getMoleFractionCO(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
                                                persons.get(i).addConcentrationCo2(t,
fireEvent.getMoleFractionCO2(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
                                                persons.get(i).addConcentrationHcn(t,
fireEvent.getMoleFractionHCN(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
                                                persons.get(i).addConcentrationO2(t,
fireEvent.getMoleFractionO2(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
                                            }
                                        }
                                    } else {
                                        if (!tempIncapacitated) {
                                            tempMovementSpeed = persons.get(i).getMovementSpeed(t,
Math.abs(fireEvent.getDistanceFromEntrance() - persons.get(i).getPosition(t - 1)));
                                            tempDistance = tempMovementSpeed * 1.0;
                                            if (towardsTunnelExit) {
                                                tempPosition = persons.get(i).getPosition(t - 1) +
tempDistance;
                                            } else {
                                                tempPosition = persons.get(i).getPosition(t - 1) -
tempDistance;
                                            }
                                            persons.get(i).addPosition(t, tempPosition);
                                            tempFedAsph =
fireEvent.getFractionOfIncapacitationAsphyxiants(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t)), tempIncapacitated);
                                            persons.get(i).addFedAsphyxia(t, tempFedAsph);
                                            tempFIDHeat =
fireEvent.getFractionOfIncapacitationHeatDownstream(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t)));
                                            persons.get(i).addFIDHeat(t, tempFIDHeat);
                                            tempFLDHeat = fireEvent.getFractionOfLethalHeatDownstream(t,
Math.abs(fireEvent.getDistanceFromEntrance() - persons.get(i).getPosition(t)));
                                            persons.get(i).addFLDHeat(t, tempFLDHeat);
                                            if (fireEvent.getTavgX(t,
Math.abs(fireEvent.getDistanceFromEntrance() - persons.get(i).getPosition(t))) == -1.0) {
                                                persons.get(i).addTemperature(t,
fireEvent.getTunnel().getTemp() - 273.0);
                                                persons.get(i).addVisibility(t,
Double.POSITIVE_INFINITY);
                                                persons.get(i).addConcentrationCo(t, 0.0);
                                                persons.get(i).addConcentrationCo2(t, 0.0);
                                                persons.get(i).addConcentrationHcn(t, 0.0);
                                                persons.get(i).addConcentrationO2(t, 0.2095);
                                            } else {
                                                persons.get(i).addTemperature(t, fireEvent.getTavgX(t,
Math.abs(fireEvent.getDistanceFromEntrance() - persons.get(i).getPosition(t))));
                                                persons.get(i).addVisibility(t,
fireEvent.getVisibility(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
                                                persons.get(i).addConcentrationCo(t,
fireEvent.getMoleFractionCO(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
                                                persons.get(i).addConcentrationCo2(t,
fireEvent.getMoleFractionCO2(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
                                                persons.get(i).addConcentrationHcn(t,
fireEvent.getMoleFractionHCN(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
                                                persons.get(i).addConcentrationO2(t,
fireEvent.getMoleFractionO2(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
                                            }
                                        } else {
                                            tempPosition = persons.get(i).getPosition(t - 1);
                                            persons.get(i).addPosition(t, tempPosition);
                                            tempFedAsph =
fireEvent.getFractionOfIncapacitationAsphyxiants(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t)), tempIncapacitated);
                                            persons.get(i).addFedAsphyxia(t, tempFedAsph);
                                            tempFIDHeat =
fireEvent.getFractionOfIncapacitationHeatDownstream(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t)));
                                            persons.get(i).addFIDHeat(t, tempFIDHeat);
```

```
                                        tempFLDHeat = fireEvent.getFractionOfLethalHeatDownstream(t,
Math.abs(fireEvent.getDistanceFromEntrance() - persons.get(i).getPosition(t)));
                                        persons.get(i).addFLDHeat(t, tempFLDHeat);
                                        if (fireEvent.getTavgX(t,
Math.abs(fireEvent.getDistanceFromEntrance() - persons.get(i).getPosition(t))) == -1.0) {
                                              persons.get(i).addTemperature(t,
fireEvent.getTunnel().getTemp() - 273.0);
                                              persons.get(i).addVisibility(t,
Double.POSITIVE_INFINITY);
                                              persons.get(i).addConcentrationCo(t, 0.0);
                                              persons.get(i).addConcentrationCo2(t, 0.0);
                                              persons.get(i).addConcentrationHcn(t, 0.0);
                                              persons.get(i).addConcentrationO2(t, 0.2095);
                                        } else {
                                              persons.get(i).addTemperature(t, fireEvent.getTavgX(t,
Math.abs(fireEvent.getDistanceFromEntrance() - persons.get(i).getPosition(t))));
                                              persons.get(i).addVisibility(t,
fireEvent.getVisibility(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
                                              persons.get(i).addConcentrationCo(t,
fireEvent.getMoleFractionCO(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
                                              persons.get(i).addConcentrationCo2(t,
fireEvent.getMoleFractionCO2(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
                                              persons.get(i).addConcentrationHcn(t,
fireEvent.getMoleFractionHCN(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
                                              persons.get(i).addConcentrationO2(t,
fireEvent.getMoleFractionO2(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t))));
                                        }
                                  }
                            }
                            if (persons.get(i).isDead() || persons.get(i).isEvacuated()) {
                                  break;
                            }
                      } else if (!downstreamFire) {
                            tempIncapacitated = persons.get(i).isIncapacitated();
                            if (t < preMovementTime) {
                                  if (t < timeToLeaveVehicle) {
                                        tempPosition = persons.get(i).getPosition(t - 1);
                                        persons.get(i).addPosition(t, tempPosition);
                                        persons.get(i).addFedAsphyxia(t, 0.0);
                                        persons.get(i).addFIDHeat(t, 0.0);
                                        persons.get(i).addFLDHeat(t, 0.0);
                                        persons.get(i).addTemperature(t,
fireEvent.getTunnel().getTemp() - 273.0);
                                        persons.get(i).addVisibility(t, Double.POSITIVE_INFINITY);
                                        persons.get(i).addConcentrationCo(t, 0.0);
                                        persons.get(i).addConcentrationCo2(t, 0.0);
                                        persons.get(i).addConcentrationHcn(t, 0.0);
                                        persons.get(i).addConcentrationO2(t, 0.2095);
                                  } else {
                                        tempPosition = persons.get(i).getPosition(t - 1);
                                        persons.get(i).addPosition(t, tempPosition);
                                        persons.get(i).addFedAsphyxia(t, 0.0);
                                        tempFIDHeat =
fireEvent.getFractionOfIncapacitationHeatUpstream(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t)));
                                        persons.get(i).addFIDHeat(t, tempFIDHeat);
                                        tempFLDHeat = fireEvent.getFractionOfLethalHeatUpstream(t,
Math.abs(fireEvent.getDistanceFromEntrance() - persons.get(i).getPosition(t)));
                                        persons.get(i).addFLDHeat(t, tempFLDHeat);
                                        persons.get(i).addTemperature(t,
fireEvent.getTunnel().getTemp() - 273.0);
                                        persons.get(i).addVisibility(t, Double.POSITIVE_INFINITY);
                                        persons.get(i).addConcentrationCo(t, 0.0);
                                        persons.get(i).addConcentrationCo2(t, 0.0);
                                        persons.get(i).addConcentrationHcn(t, 0.0);
                                        persons.get(i).addConcentrationO2(t, 0.2095);
                                  }
                            } else {
                                  if (!tempIncapacitated) {
                                        tempMovementSpeed =
persons.get(i).getUnimpededMovementSpeed();
                                        tempDistance = tempMovementSpeed * 1.0;
                                        if (towardsTunnelExit) {
```

```
                                                tempPosition = persons.get(i).getPosition(t - 1) +
tempDistance;
                                        } else {
                                                tempPosition = persons.get(i).getPosition(t - 1) -
tempDistance;
                                        }
                                        persons.get(i).addPosition(t, tempPosition);
                                        persons.get(i).addFedAsphyxia(t, 0.0);
                                        tempFIDHeat =
fireEvent.getFractionOfIncapacitationHeatUpstream(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t)));
                                        persons.get(i).addFIDHeat(t, tempFIDHeat);
                                        tempFLDHeat = fireEvent.getFractionOfLethalHeatUpstream(t,
Math.abs(fireEvent.getDistanceFromEntrance() - persons.get(i).getPosition(t)));
                                        persons.get(i).addFLDHeat(t, tempFLDHeat);
                                        persons.get(i).addTemperature(t,
fireEvent.getTunnel().getTemp() - 273.0);
                                        persons.get(i).addVisibility(t, Double.POSITIVE_INFINITY);
                                        persons.get(i).addConcentrationCo(t, 0.0);
                                        persons.get(i).addConcentrationCo2(t, 0.0);
                                        persons.get(i).addConcentrationHcn(t, 0.0);
                                        persons.get(i).addConcentrationO2(t, 0.2095);
                                } else {
                                        tempPosition = persons.get(i).getPosition(t - 1);
                                        persons.get(i).addPosition(t, tempPosition);
                                        persons.get(i).addFedAsphyxia(t, 0.0);
                                        tempFIDHeat =
fireEvent.getFractionOfIncapacitationHeatDownstream(t, Math.abs(fireEvent.getDistanceFromEntrance() -
persons.get(i).getPosition(t)));
                                        persons.get(i).addFIDHeat(t, tempFIDHeat);
                                        tempFLDHeat = fireEvent.getFractionOfLethalHeatDownstream(t,
Math.abs(fireEvent.getDistanceFromEntrance() - persons.get(i).getPosition(t)));
                                        persons.get(i).addFLDHeat(t, tempFLDHeat);
                                        persons.get(i).addTemperature(t,
fireEvent.getTunnel().getTemp() - 273.0);
                                        persons.get(i).addVisibility(t, Double.POSITIVE_INFINITY);
                                        persons.get(i).addConcentrationCo(t, 0.0);
                                        persons.get(i).addConcentrationCo2(t, 0.0);
                                        persons.get(i).addConcentrationHcn(t, 0.0);
                                        persons.get(i).addConcentrationO2(t, 0.2095);
                                }
                        }
                        if (persons.get(i).isDead() || persons.get(i).isEvacuated()) {
                                break;
                        }
                }
        }
    }
    }

    /** Saves the position of each person
     * for each time step during the simulation
     * in a comma separated .txt file. */
    public void saveEvacPosition() {
        String fileName = "evacPosition.txt";

        int max = 0;
        for (int j = 0; j < persons.size(); j++) {
            if (persons.get(j).timeToTheEnd() > max) {
                max = persons.get(j).timeToTheEnd();
            }
        }

        try {
            FileWriter fileWriter = new FileWriter(fileName);
            BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

            bufferedWriter.write(fireEvent.versionText());
            bufferedWriter.newLine();
            bufferedWriter.newLine();

            bufferedWriter.write("TIME");
            for (int i = 0; i < persons.size(); i++) {
                bufferedWriter.write(",");
                bufferedWriter.write("PERSON " + Integer.toString(i + 1));
            }

            for (int j = 0; j < max + 1; j++) {
                bufferedWriter.newLine();
```

```
                        bufferedWriter.write(Integer.toString(j));
                        for (int k = 0; k < persons.size(); k++) {
                              bufferedWriter.write(",");
                              if (j >= persons.get(k).timeToTheEnd()) {

      bufferedWriter.write(Double.toString(persons.get(k).getPosition(persons.get(k).timeToTheEnd() -
1)));
                              }
                              else {

      bufferedWriter.write(Double.toString(persons.get(k).getPosition(j)));
                              }
                        }
                  }

                  bufferedWriter.close();
            }
            catch (IOException ex) {
            System.out.println(
                "Error writing to file '"
                + fileName + "'");
            }
      }

      /** Saves the accumulated total FID for asphyxiants
       *  of each person for each time step
       *  during the simulation in a comma separated
       *  .txt file. */
      public void saveEvacFedAsphyxia() {
            String fileName = "evacFidAsphyxia.txt";

            int max = 0;
            for (int j = 0; j < persons.size(); j++) {
                  if (persons.get(j).timeToTheEnd() > max) {
                        max = persons.get(j).timeToTheEnd();
                  }
            }

            try {
                  FileWriter fileWriter = new FileWriter(fileName);
                  BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

                  bufferedWriter.write(fireEvent.versionText());
                  bufferedWriter.newLine();
                  bufferedWriter.newLine();

                  bufferedWriter.write("TIME");
                  for (int i = 0; i < persons.size(); i++) {
                        bufferedWriter.write(",");
                        bufferedWriter.write("PERSON " + Integer.toString(i + 1));
                  }

                  for (int j = 0; j < max + 1; j++) {
                        bufferedWriter.newLine();
                        bufferedWriter.write(Integer.toString(j));
                        for (int k = 0; k < persons.size(); k++) {
                              bufferedWriter.write(",");
                              if (j >= persons.get(k).timeToTheEnd()) {

      bufferedWriter.write(Double.toString(persons.get(k).getAccumulatedFedAsphyxia(persons.get(k).time
ToTheEnd())));
                              }
                              else {

      bufferedWriter.write(Double.toString(persons.get(k).getAccumulatedFedAsphyxia(j)));
                              }
                        }
                  }

                  bufferedWriter.close();
            }
            catch (IOException ex) {
            System.out.println(
                "Error writing to file '"
                + fileName + "'");
            }
      }

      /** Saves the accumulated total FID for heat
```

```
 *  of each person for each time step
 *  during the simulation in a comma separated
 *  .txt file. */
public void saveEvacFidHeat() {
      String fileName = "evacFidHeat.txt";

      int max = 0;
      for (int j = 0; j < persons.size(); j++) {
            if (persons.get(j).timeToTheEnd() > max) {
                  max = persons.get(j).timeToTheEnd();
            }
      }

      try {
            FileWriter fileWriter = new FileWriter(fileName);
            BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

            bufferedWriter.write(fireEvent.versionText());
            bufferedWriter.newLine();
            bufferedWriter.newLine();

            bufferedWriter.write("TIME");
            for (int i = 0; i < persons.size(); i++) {
                  bufferedWriter.write(",");
                  bufferedWriter.write("PERSON " + Integer.toString(i + 1));
            }

            for (int j = 0; j < max + 1; j++) {
                  bufferedWriter.newLine();
                  bufferedWriter.write(Integer.toString(j));
                  for (int k = 0; k < persons.size(); k++) {
                        bufferedWriter.write(",");
                        if (j >= persons.get(k).timeToTheEnd()) {

bufferedWriter.write(Double.toString(persons.get(k).getAccumulatedFIDHeat(persons.get(k).timeToTh
eEnd())));
                        }
                        else {

bufferedWriter.write(Double.toString(persons.get(k).getAccumulatedFIDHeat(j)));
                        }
                  }
            }

            bufferedWriter.close();
      }
      catch (IOException ex) {
      System.out.println(
          "Error writing to file '"
          + fileName + "'");
      }

}

/** Saves the accumulated total FLD for heat
 *  of each person for each time step
 *  during the simulation in a comma separated
 *  .txt file. */
public void saveEvacFldHeat() {
      String fileName = "evacFldHeat.txt";

      int max = 0;
      for (int j = 0; j < persons.size(); j++) {
            if (persons.get(j).timeToTheEnd() > max) {
                  max = persons.get(j).timeToTheEnd();
            }
      }

      try {
            FileWriter fileWriter = new FileWriter(fileName);
            BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

            bufferedWriter.write(fireEvent.versionText());
            bufferedWriter.newLine();
            bufferedWriter.newLine();

            bufferedWriter.write("TIME");
            for (int i = 0; i < persons.size(); i++) {
                  bufferedWriter.write(",");
```

```
                        bufferedWriter.write("PERSON " + Integer.toString(i + 1));
                    }

                    for (int j = 0; j < max + 1; j++) {
                        bufferedWriter.newLine();
                        bufferedWriter.write(Integer.toString(j));
                        for (int k = 0; k < persons.size(); k++) {
                            bufferedWriter.write(",");
                            if (j >= persons.get(k).timeToTheEnd()) {

bufferedWriter.write(Double.toString(persons.get(k).getAccumulatedFLDHeat(persons.get(k).timeToTh
eEnd())));
                            }
                            else {

bufferedWriter.write(Double.toString(persons.get(k).getAccumulatedFLDHeat(j)));
                            }
                        }
                    }

                    bufferedWriter.close();
                }
                catch (IOException ex) {
                System.out.println(
                    "Error writing to file '"
                    + fileName + "'");
                }

            }

            /** Saves the end position for each
             * person included in the simulation. */
            public void saveEvacEndPosition() {
                String fileName = "evacEndPosition.txt";

                try {
                    FileWriter fileWriter = new FileWriter(fileName);
                    BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

                    bufferedWriter.write(fireEvent.versionText());
                    bufferedWriter.newLine();
                    bufferedWriter.newLine();

                    bufferedWriter.write("PERSON [ID]");
                    bufferedWriter.write(",");
                    bufferedWriter.write("TIME WHEN REACHED END [s]");
                    bufferedWriter.write(",");
                    bufferedWriter.write("POSITION WHEN REACHED END");
                    bufferedWriter.write(",");
                    bufferedWriter.write("STATUS WHEN REACHED END ");

                    for (int i = 0; i < persons.size(); i++) {
                        bufferedWriter.newLine();
                        bufferedWriter.write(Integer.toString(i + 1));
                        bufferedWriter.write(",");
                        bufferedWriter.write(Integer.toString((int) persons.get(i).timeToTheEnd()));
                        bufferedWriter.write(",");

bufferedWriter.write(Double.toString(Math.round(persons.get(i).getLastPosition())));
                        bufferedWriter.write(",");
                        if (persons.get(i).isDead()) {
                            bufferedWriter.write("DEAD");
                        } else {
                            bufferedWriter.write("ALIVE");
                        }
                    }
                    bufferedWriter.close();
                }
                catch (IOException ex) {
                System.out.println(
                    "Error writing to file '"
                    + fileName + "'");
                }
            }

            /** Saves the time when each evacuated
             * person managed exited the tunnel. */
            public void saveEvacuatedPerTime() {
                String fileName = "evacEvacuatedPerTime.txt";
```

```
            try {
                  FileWriter fileWriter = new FileWriter(fileName);
                  BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

                  bufferedWriter.write(fireEvent.versionText());
                  bufferedWriter.newLine();
                  bufferedWriter.newLine();

                  bufferedWriter.write("TIME");
                  bufferedWriter.write(",");
                  bufferedWriter.write("ACCUMULATED NO. OF EVACUATED");
                  bufferedWriter.write(",");
                  bufferedWriter.write("PROPORTION OF ALL EVACUEES");
                  bufferedWriter.newLine();
                  bufferedWriter.write("0");
                  bufferedWriter.write(",");
                  bufferedWriter.write("0");
                  bufferedWriter.write(",");
                  bufferedWriter.write("0");
                  int counter = 0;

                  ArrayList<Integer> sortedTimes = new ArrayList<Integer>();
                  for(int i = 0; i < persons.size(); i++) {
                        if (!persons.get(i).isDead()) {
                              sortedTimes.add(persons.get(i).timeToTheEnd());
                        }
                  }
                  Collections.sort(sortedTimes);
                  for (int i = 0; i < sortedTimes.size(); i++) {
                        bufferedWriter.newLine();
                        bufferedWriter.write(Integer.toString(sortedTimes.get(i)));
                        bufferedWriter.write(",");
                        counter++;
                        bufferedWriter.write(Integer.toString(counter));
                        bufferedWriter.write(",");
                        bufferedWriter.write(Double.toString(((double) counter) / ((double)
persons.size())));
                  }
                  bufferedWriter.close();
            }

            catch (IOException ex) {
            System.out.println(
                "Error writing to file '"
                + fileName + "'");
            }
      }

      /** Saves the averaged temperature of the tunnel
       * environment for the position the person is at
       * for each person and for each time step during
       * the simulation in a comma separated .txt file. */
      public void saveEvacTemperature() {
            String fileName = "evacTemperature.txt";

            int max = 0;
            for (int j = 0; j < persons.size(); j++) {
                  if (persons.get(j).timeToTheEnd() > max) {
                        max = persons.get(j).timeToTheEnd();
                  }
            }

            try {
                  FileWriter fileWriter = new FileWriter(fileName);
                  BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

                  bufferedWriter.write(fireEvent.versionText());
                  bufferedWriter.newLine();
                  bufferedWriter.newLine();

                  bufferedWriter.write("TIME");
                  for (int i = 0; i < persons.size(); i++) {
                        bufferedWriter.write(",");
                        bufferedWriter.write("PERSON " + Integer.toString(i + 1));
                  }

                  for (int j = 0; j < max + 1; j++) {
                        bufferedWriter.newLine();
```

```
                        bufferedWriter.write(Integer.toString(j));
                        for (int k = 0; k < persons.size(); k++) {
                              bufferedWriter.write(",");
                              if (j >= persons.get(k).timeToTheEnd()) {

      bufferedWriter.write(Double.toString(persons.get(k).getTemperature(persons.get(k).timeToTheEnd()
- 1)));
                              }
                              else {

      bufferedWriter.write(Double.toString(persons.get(k).getTemperature(j)));
                              }
                        }
                  }

                  bufferedWriter.close();
            }
            catch (IOException ex) {
            System.out.println(
                "Error writing to file '"
                + fileName + "'");
            }
      }

      /** Saves the averaged visibility of the tunnel
       * environment for the position the person is at
       * for each person and for each time step during
       * the simulation in a comma separated .txt file. */
      public void saveEvacVisibility() {
            String fileName = "evacVisibility.txt";

            int max = 0;
            for (int j = 0; j < persons.size(); j++) {
                  if (persons.get(j).timeToTheEnd() > max) {
                        max = persons.get(j).timeToTheEnd();
                  }
            }

            try {
                  FileWriter fileWriter = new FileWriter(fileName);
                  BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

                  bufferedWriter.write(fireEvent.versionText());
                  bufferedWriter.newLine();
                  bufferedWriter.newLine();

                  bufferedWriter.write("TIME");
                  for (int i = 0; i < persons.size(); i++) {
                        bufferedWriter.write(",");
                        bufferedWriter.write("PERSON " + Integer.toString(i + 1));
                  }

                  for (int j = 0; j < max + 1; j++) {
                        bufferedWriter.newLine();
                        bufferedWriter.write(Integer.toString(j));
                        for (int k = 0; k < persons.size(); k++) {
                              bufferedWriter.write(",");
                              if (j >= persons.get(k).timeToTheEnd()) {

      bufferedWriter.write(Double.toString(persons.get(k).getVisibility(persons.get(k).timeToTheEnd() -
1)));
                              }
                              else {

      bufferedWriter.write(Double.toString(persons.get(k).getVisibility(j)));
                              }
                        }
                  }

                  bufferedWriter.close();
            }
            catch (IOException ex) {
            System.out.println(
                "Error writing to file '"
                + fileName + "'");
            }
      }

      /** Saves the averaged gas concentration CO of the tunnel
```

```
     * environment for the position the person is at
     * for each person and for each time step during
     * the simulation in a comma separated .txt file. */
    public void saveEvacConcentrationCo() {
          String fileName = "evacConcentrationCo.txt";

          int max = 0;
          for (int j = 0; j < persons.size(); j++) {
                if (persons.get(j).timeToTheEnd() > max) {
                      max = persons.get(j).timeToTheEnd();
                }
          }

          try {
                FileWriter fileWriter = new FileWriter(fileName);
                BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

                bufferedWriter.write(fireEvent.versionText());
                bufferedWriter.newLine();
                bufferedWriter.newLine();

                bufferedWriter.write("TIME");
                for (int i = 0; i < persons.size(); i++) {
                      bufferedWriter.write(",");
                      bufferedWriter.write("PERSON " + Integer.toString(i + 1));
                }

                for (int j = 0; j < max + 1; j++) {
                      bufferedWriter.newLine();
                      bufferedWriter.write(Integer.toString(j));
                      for (int k = 0; k < persons.size(); k++) {
                            bufferedWriter.write(",");
                            if (j >= persons.get(k).timeToTheEnd()) {

    bufferedWriter.write(Double.toString(persons.get(k).getConcentrationCo(persons.get(k).timeToTheEn
d() - 1)));
                            }
                            else {

    bufferedWriter.write(Double.toString(persons.get(k).getConcentrationCo(j)));
                            }
                      }
                }

                bufferedWriter.close();
          }
          catch (IOException ex) {
          System.out.println(
              "Error writing to file '"
              + fileName + "'");
          }
    }

    /** Saves the averaged gas concentration CO of the tunnel
     * environment for the position the person is at
     * for each person and for each time step during
     * the simulation in a comma separated .txt file. */
    public void saveEvacConcentrationCo2() {
          String fileName = "evacConcentrationCo2.txt";

          int max = 0;
          for (int j = 0; j < persons.size(); j++) {
                if (persons.get(j).timeToTheEnd() > max) {
                      max = persons.get(j).timeToTheEnd();
                }
          }

          try {
                FileWriter fileWriter = new FileWriter(fileName);
                BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

                bufferedWriter.write(fireEvent.versionText());
                bufferedWriter.newLine();
                bufferedWriter.newLine();

                bufferedWriter.write("TIME");
                for (int i = 0; i < persons.size(); i++) {
                      bufferedWriter.write(",");
                      bufferedWriter.write("PERSON " + Integer.toString(i + 1));
```

```
                }

                for (int j = 0; j < max + 1; j++) {
                      bufferedWriter.newLine();
                      bufferedWriter.write(Integer.toString(j));
                      for (int k = 0; k < persons.size(); k++) {
                            bufferedWriter.write(",");
                            if (j >= persons.get(k).timeToTheEnd()) {

      bufferedWriter.write(Double.toString(persons.get(k).getConcentrationCo2(persons.get(k).timeToTheE
nd() - 1)));
                            }
                            else {

      bufferedWriter.write(Double.toString(persons.get(k).getConcentrationCo2(j)));
                            }
                      }
                }

                bufferedWriter.close();
            }
            catch (IOException ex) {
            System.out.println(
                "Error writing to file '"
                + fileName + "'");
            }
      }

      /** Saves the averaged gas concentration CO of the tunnel
       * environment for the position the person is at
       * for each person and for each time step during
       * the simulation in a comma separated .txt file. */
      public void saveEvacConcentrationHcn() {
            String fileName = "evacConcentrationHcn.txt";

            int max = 0;
            for (int j = 0; j < persons.size(); j++) {
                  if (persons.get(j).timeToTheEnd() > max) {
                        max = persons.get(j).timeToTheEnd();
                  }
            }

            try {
                  FileWriter fileWriter = new FileWriter(fileName);
                  BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

                  bufferedWriter.write(fireEvent.versionText());
                  bufferedWriter.newLine();
                  bufferedWriter.newLine();

                  bufferedWriter.write("TIME");
                  for (int i = 0; i < persons.size(); i++) {
                        bufferedWriter.write(",");
                        bufferedWriter.write("PERSON " + Integer.toString(i + 1));
                  }

                  for (int j = 0; j < max + 1; j++) {
                        bufferedWriter.newLine();
                        bufferedWriter.write(Integer.toString(j));
                        for (int k = 0; k < persons.size(); k++) {
                              bufferedWriter.write(",");
                              if (j >= persons.get(k).timeToTheEnd()) {

      bufferedWriter.write(Double.toString(persons.get(k).getConcentrationHcn(persons.get(k).timeToTheE
nd() - 1)));
                              }
                              else {

      bufferedWriter.write(Double.toString(persons.get(k).getConcentrationHcn(j)));
                              }
                        }
                  }

                  bufferedWriter.close();
            }
            catch (IOException ex) {
            System.out.println(
                "Error writing to file '"
                + fileName + "'");
```

```
		}
	}

	/** Saves the averaged gas concentration CO of the tunnel
	 * environment for the position the person is at
	 * for each person and for each time step during
	 * the simulation in a comma separated .txt file. */
	public void saveEvacConcentrationO2() {
		String fileName = "evacConcentrationO2.txt";

		int max = 0;
		for (int j = 0; j < persons.size(); j++) {
			if (persons.get(j).timeToTheEnd() > max) {
				max = persons.get(j).timeToTheEnd();
			}
		}

		try {
			FileWriter fileWriter = new FileWriter(fileName);
			BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

			bufferedWriter.write(fireEvent.versionText());
			bufferedWriter.newLine();
			bufferedWriter.newLine();

			bufferedWriter.write("TIME");
			for (int i = 0; i < persons.size(); i++) {
				bufferedWriter.write(",");
				bufferedWriter.write("PERSON " + Integer.toString(i + 1));
			}

			for (int j = 0; j < max + 1; j++) {
				bufferedWriter.newLine();
				bufferedWriter.write(Integer.toString(j));
				for (int k = 0; k < persons.size(); k++) {
					bufferedWriter.write(",");
					if (j >= persons.get(k).timeToTheEnd()) {

	bufferedWriter.write(Double.toString(persons.get(k).getConcentrationO2(persons.get(k).timeToTheEn
d() - 1)));
					}
					else {

	bufferedWriter.write(Double.toString(persons.get(k).getConcentrationO2(j)));
					}
				}
			}

			bufferedWriter.close();
		}
		catch (IOException ex) {
		System.out.println(
		    "Error writing to file '"
		    + fileName + "'");
		}
	}

	/** Saves summary of evacuation simulation
	 * to a comma separated .txt file. */
	public void saveEvacSummary() {
		String fileName = "evacSummary.txt";
		String tempString = null;

		try {
			FileWriter fileWriter = new FileWriter(fileName);
			BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

			bufferedWriter.write(fireEvent.versionText());
			bufferedWriter.newLine();
			bufferedWriter.newLine();

			bufferedWriter.write("TUNNEL LENGTH:," + fireEvent.getTunnel().getLength() + " m");
			bufferedWriter.newLine();
			bufferedWriter.write("POSITION OF FIRE:," + fireEvent.getDistanceFromEntrance());
			bufferedWriter.newLine();
			bufferedWriter.newLine();
			bufferedWriter.write("PERSON [no.],");
			for (int i = 0; i < persons.size(); i++) {
				bufferedWriter.write(Integer.toString(i + 1));
```

```
                        bufferedWriter.write(",");
                    }
                    bufferedWriter.newLine();
                    bufferedWriter.write("RECOGNITION TIME [s],");
                    for (int i = 0; i < persons.size(); i++) {
                        bufferedWriter.write(Integer.toString((int)
persons.get(i).getRecognitionTime()));
                        bufferedWriter.write(",");
                    }
                    bufferedWriter.newLine();
                    bufferedWriter.write("RESPONSE TIME [s],");
                    for (int i = 0; i < persons.size(); i++) {
                        bufferedWriter.write(Integer.toString((int) persons.get(i).getResponseTime()));
                        bufferedWriter.write(",");
                    }
                    bufferedWriter.newLine();
                    bufferedWriter.write("TIME TO LEAVE VEHICLE [s],");
                    for (int i = 0; i < persons.size(); i++) {
                        bufferedWriter.write(Integer.toString((int)
persons.get(i).getTimeToLeaveVehicle()));
                        bufferedWriter.write(",");
                    }
                    bufferedWriter.newLine();
                    bufferedWriter.write("START POSITION [m],");
                    for (int i = 0; i < persons.size(); i++) {
                        bufferedWriter.write(Integer.toString((int) persons.get(i).getStartPosition()));
                        bufferedWriter.write(",");
                    }
                    bufferedWriter.newLine();
                    bufferedWriter.write("END POSITION [m],");
                    for (int i = 0; i < persons.size(); i++) {
                        bufferedWriter.write(Integer.toString((int) persons.get(i).getLastPosition()));
                        bufferedWriter.write(",");
                    }
                    bufferedWriter.newLine();
                    bufferedWriter.write("DISTANCE WALKED [m],");
                    for (int i = 0; i < persons.size(); i++) {
                        bufferedWriter.write(Integer.toString(Math.abs((int)
Math.round(persons.get(i).getLastPosition() - persons.get(i).getStartPosition()))));
                        bufferedWriter.write(",");
                    }
                    bufferedWriter.newLine();
                    bufferedWriter.write("FED(TOT) ASPHYXIANTS,");
                    for (int i = 0; i < persons.size(); i++) {
                        bufferedWriter.write(Double.toString(persons.get(i).getSumFedAsphyxia()));
                        bufferedWriter.write(",");
                    }
                    bufferedWriter.newLine();
                    bufferedWriter.write("FID(TOT) HEAT,");
                    for (int i = 0; i < persons.size(); i++) {
                        bufferedWriter.write(Double.toString(persons.get(i).getSumFIDHeat()));
                        bufferedWriter.write(",");
                    }
                    bufferedWriter.newLine();
                    bufferedWriter.write("FLD(TOT) HEAT,");
                    for (int i = 0; i < persons.size(); i++) {
                        bufferedWriter.write(Double.toString(persons.get(i).getSumFLDHeat()));
                        bufferedWriter.write(",");
                    }
                    bufferedWriter.newLine();
                    bufferedWriter.write("EVACUATED?,");
                    for (int i = 0; i < persons.size(); i++) {
                        if (persons.get(i).isDead()) {
                            tempString = "NO";
                        } else {
                            tempString = "YES";
                        }
                        bufferedWriter.write(tempString);
                        bufferedWriter.write(",");
                    }
                    bufferedWriter.newLine();
                    bufferedWriter.write("TIME UNTIL DEATH/SAFETY [s],");
                    for (int i = 0; i < persons.size(); i++) {
                        bufferedWriter.write(Integer.toString(persons.get(i).timeToTheEnd()));
                        bufferedWriter.write(",");
                    }

                    bufferedWriter.close();
```

```
            }

            catch (IOException ex) {
            System.out.println(
                "Error writing to file '"
                + fileName + "'");
            }
      }

      /** Saves temporary summary of evacuation simulation
       * to command prompt (not regularly used in TuFT). */
      public void saveEvacSimDataTemp() {
            DecimalFormat df = new DecimalFormat("#.##");
            String tempString = new String();

            System.out.println("RESULTS");
            System.out.print("PERSON [no.]" + "\t" + "\t" + "\t");
            for(int i = 0; i < persons.size(); i++) {
                  System.out.print(" " + (i + 1) + "\t");
            }
            System.out.print("\n");
            System.out.print("START POSITION [m]" + "\t" + "\t");
            for(int i = 0; i < persons.size(); i++) {
                  System.out.print((int) persons.get(i).getStartPosition() + "\t");
            }
            System.out.print("\n");
            System.out.print("END POSITION [m]" + "\t" + "\t");
            for(int i = 0; i < persons.size(); i++) {
                  System.out.print((int) persons.get(i).getLastPosition() + "\t");
            }
            System.out.print("\n");
            System.out.print("DISTANCE WALKED [m]" + "\t" + "\t");
            for(int i = 0; i < persons.size(); i++) {
                  System.out.print(Math.abs((int) (persons.get(i).getLastPosition() -
persons.get(i).getStartPosition())) + "\t");
            }
            System.out.print("\n");
            System.out.print("FED ASPHYXIA" + "\t" + "\t" + "\t");
            for(int i = 0; i < persons.size(); i++) {
                  System.out.print(df.format(persons.get(i).getSumFedAsphyxia()) + "\t");
            }
            System.out.print("\n");
            System.out.print("FID HEAT" + "\t" + "\t" + "\t");
            for(int i = 0; i < persons.size(); i++) {
                  System.out.print(df.format(persons.get(i).getSumFIDHeat()) + "\t");
            }
            System.out.print("\n");
            System.out.print("FLD HEAT" + "\t" + "\t" + "\t");
            for(int i = 0; i < persons.size(); i++) {
                  System.out.print(df.format(persons.get(i).getSumFLDHeat()) + "\t");
            }
            System.out.print("\n");
            System.out.print("DID PERSON SURVIVE?" + "\t" + "\t");
            for(int i = 0; i < persons.size(); i++) {
                  if (persons.get(i).isDead()) {
                        tempString = "NO";
                  } else {
                        tempString = "YES";
                  }
                  System.out.print(tempString + "\t");
            }
            System.out.print("\n");
            System.out.print("TIME UNTIL DEATH/SAFETY [s]" + "\t");
            for(int i = 0; i < persons.size(); i++) {
                  System.out.print(persons.get(i).timeToTheEnd() + "\t");
            }
      }
}
```

C.8 FireFighterPair.java

```
import java.util.ArrayList;

/** This Java class is a part of TuFT, a decision support
 * tool for tunnel fires. The tool was developed by Karl Fridolf
```

```
 * and Håkan Frantzich at Lund University during 2013-2014.
 * Among other things, the tool offers the possibility to
 * simulate fires in tunnels, and furthermore, the effects
 * on evacuation and rescue operation possibilities.
 *
 * The tool's sub models, as well as the conclusions and
 * the results produced by the tool, have not been quality
 * controlled, verified or validated to the extent necessary
 * for quality assurance. TuFT must hence be used with caution.
 * Anyone who uses the results produced by TuFT does so on
 * his/her own responsibility */
public class FireFighterPair {
	private RescueOperation operation;
	private double startPos;
	private int actionTime;
	private ArrayList<Double> deltaTemperatureRises;
	private ArrayList<Double> position;

	/** Defines a fire fighter pair. Input is a
	 * previously defined RescueOperation operation and
	 * a start position for the pair. The start position should
	 * be expressed as a absolute position in the tunnel in m. Thus,
	 * not relative to the fire. */
	public FireFighterPair(RescueOperation operation, double startPos) {
		this.operation = operation;
		this.startPos = startPos;

		if (operation.doWeHaveMuchAir()) {
			this.actionTime = 40 * 60;
		} else {
			this.actionTime = 25 * 60;
		}

		this.deltaTemperatureRises = new ArrayList<Double>();
		deltaTemperatureRises.add(0, 0.0);
		this.position = new ArrayList<Double>();
		position.add(0, startPos);
	}

	/** Returns the available action time of
	 * the fire fighter pair in s. */
	public int getActionTime() {
		return actionTime;
	}

	/** Defines a new action time in s for the
	 * fire fighter pair. */
	public void setActionTime(int newActionTime) {
		actionTime = newActionTime;
	}

	/** Adds a delta temperature rise deltaT in °C
	 * for a time time in s (into the fire event). */
	public void addDeltaTemperatureRise(int time, double deltaT) {
		deltaTemperatureRises.add(time, deltaT);
	}

	/** Returns the accumulated temperature rise of the
	 * fire fighter pair in °C. */
	public double getSumDeltaTemperatureRise() {
		double sum = 0.0;
		for (int i = 0; i < deltaTemperatureRises.size(); i++) {
			sum+= deltaTemperatureRises.get(i);
		}
		return sum;
	}

	/** Adds a position pos in m for a time time
	 * in s (in the fire event). */
	public void addPosition(int time, double pos) {
		position.add(time, pos);
	}

	/** Returns the latest known position of the fire
	 * fighter pair in the tunnel [m]. */
	public double getLastPosition() {
		return position.get(position.size() - 1);
	}
```

```
	/** Returns the position in m in the tunnel at which
	 * the fire fighter pair began their operation. */
	public double getStartPosition() {
		return startPos;
	}

	/** Returns the array of positions for the fire fighter
	 * pair during their operation in the tunnel. */
	public ArrayList<Double> getArrayOfPosition() {
		return position;
	}

	/** Returns the walking speed in m/s when walking in
	 * smoke free conditions, e.g., upstream the fire. */
	public double getUnimpededMovementSpeed() {
		return 1.0;
	}

	/** Returns the walking speed in [m/s] of the fire fighters for a
	 * the given visibility condition. Input is time in s
	 * and the position x of the fire fighter relative to the fire
	 * in m. Thus, note that it is not the actual position of
	 * the person but the relative position compared to the
	 * fire. */
	public double getMovementSpeed(int t, double x) {
		double visibility = operation.getFireEvent().getVisibility(t, x);
		double movementSpeedTemp = 0.0;
		if (operation.doWeHaveThermalImaging()) {
			if (visibility >= 4.0) {
				movementSpeedTemp = 1.0;
			} else {
				movementSpeedTemp = (1.55 - (1.1 *
operation.getFireEvent().getExtinctionCoefficient(t, x)));
				if (movementSpeedTemp < 0.6) {
					movementSpeedTemp = 0.6;
				}
			}
		} else {
			if (visibility >= 4.0) {
				movementSpeedTemp = 1.0;
			} else if (visibility < 4.0 && visibility > 1.5) {
				movementSpeedTemp = (1.55 - (1.1 *
operation.getFireEvent().getExtinctionCoefficient(t, x)));
			} else if (visibility <= 1.5) {
				movementSpeedTemp = 0.1;
			}
		}
		return movementSpeedTemp;
	}

}
```

C.9 RescueOperation.java

```
import java.util.ArrayList;

/** This Java class is a part of TuFT, a decision support
 * tool for tunnel fires. The tool was developed by Karl Fridolf
 * and Håkan Frantzich at Lund University during 2013-2014.
 * Among other things, the tool offers the possibility to
 * simulate fires in tunnels, and furthermore, the effects
 * on evacuation and rescue operation possibilities.
 *
 * The tool's sub models, as well as the conclusions and
 * the results produced by the tool, have not been quality
 * controlled, verified or validated to the extent necessary
 * for quality assurance. TuFT must hence be used with caution.
 * Anyone who uses the results produced by TuFT does so on
 * his/her own responsibility */
public class RescueOperation {
	private FireEvent fireEvent;
	private Tunnel tunnel;
	private int preparationTime;
	private boolean upstream;
	private boolean portal;
```

```java
private int numberOfFireFighters;
private boolean thermalImaging;
private boolean muchAir;
private double goal;
private ArrayList<Double> position;
private int typeOfAction;

private double hoseLengthLong;
private double hoseLengthShort;
private int connectTimeLong;
private int connectTimeShort;

private double startPos;
private boolean towardExit;

private int timeSpent;
private double distanceMoved;

/** Creates a RescueOperation in a previously
 * defined FireEvent fireEvent. The input variables
 * are described below:
 *
 * fireEvent = (FireEvent) A defined fireEvent (fire in a tunnel)
 * preparationTime = (Integer) Time in min to prepare before
 * operation can start, including, e.g., driving time to the tunnel
 * upstream = (Boolean) Describes whether operation should be done
 * upstream (true) or downstream (false) fire
 * portal = (Boolean) Describes whether operation should be done
 * through one of the tunnel portals (true) or through the closest
 * emergency exit to the fire (false)
 * numberOfFireFighters = (Integer) The number of fire fighters
 * that can take part in the operation inside the tunnel
 * themalImaging = (Boolean) Describes whether the fire fighters
 * do have thermal imaging equipment available (true) or not (false)
 * muchAir = (Boolean) Describes whether the fire fighter do have
 * large (true) or small (false) air tubes (implicitly defines
 * the available action time for each fire fighter pair)
 * hoseLengthLong = (Double) Describes the length of a long fire hose
 * hoseLengthShort = (Double) Describes the length of a short fire hose
 * connectTimeLong = (Integer) Describes the time it takes to connect
 * a branch
 * connectTimeShort = (Integer) Describes the time it takes to connect
 * two hoses */
public RescueOperation(FireEvent fireEvent, int preparationTime,
            boolean upstream, boolean portal, int numberOfFireFighters,
            boolean thermalImaging, boolean muchAir, double hoseLengthLong,
            double hoseLengthShort, int connectTimeLong, int connectTimeShort) {
      this.fireEvent = fireEvent;
      this.tunnel = fireEvent.getTunnel();
      this.preparationTime = preparationTime * 60;
      this.upstream = upstream;
      this.portal = portal;
      this.numberOfFireFighters = numberOfFireFighters;
      this.thermalImaging = thermalImaging;
      this.muchAir = muchAir;
      this.goal = fireEvent.getDistanceFromEntrance();
      this.timeSpent = 0;
      this.distanceMoved = 0.0;
      this.hoseLengthLong = hoseLengthLong;
      this.hoseLengthShort = hoseLengthShort;
      this.connectTimeLong = connectTimeLong;
      this.connectTimeShort = connectTimeShort;
      this.setStartPosAndDirectionForOperation();
      this.typeOfAction = 1;
      this.position = new ArrayList<Double>();
      position.add(0, startPos);
}

/** Returns the Fire Event in which the operation
 * is taking place. */
public FireEvent getFireEvent() {
      return fireEvent;
}

/** Returns the preparation time for the operation
 * in min. */
public double getPreparationTime() {
      return preparationTime;
}
```

```java
/** Returns the number of fire fighters taking part
 * in the operation. */
public int getNumberOfFireFighters() {
      return numberOfFireFighters;
}

/** Returns an answer to the question if
 * the fire fighters are equipped with thermal
 * imaging cameras. If true, they do, if false,
 * they don't. */
public boolean doWeHaveThermalImaging() {
      return thermalImaging;
}

/** Returns an answer to the question if
 * the fire fighters are equipped with large
 * air tubes. If true, they do, if false,
 * they don't. */
public boolean doWeHaveMuchAir() {
      return muchAir;
}

/** Returns the start position of the operation in m.
 * Determined by whether or not the operation is
 * carried out upstream or downstream the fire,
 * and if the tunnel is entered through any of the
 * two portals or through the closest emergency
 * exit. */
public double getStartPos() {
      return startPos;
}

/** Returns the position in m of the operation
 * front at a time defined in s. */
public double getPosition(int t) {
      return position.get(t);
}

/** Returns the last position in m of the operation
 * front. */
public double getLastPosition() {
      return position.get(position.size() - 1);
}

/** Returns the array of positions of the operation
 * for all times of the operation. */
public ArrayList<Double> getArrayOfPosition() {
      return position;
}

/** Adds the operation front's position x in m
 * for a time t in s into the fire event. */
public void addPosition(int t, double x) {
      position.add(t, x);
}

/** Returns an answer to the question if
 * the operation is performed upstream the
 * fire. If true, the operation is done upstream,
 * if false, downstream. */
public boolean operatingUpstream() {
      return upstream;
}

/** Returns an answer to the question if
 * the operation is performed toward the tunnel
 * exit. If true, the operation is done toward
 * the tunnel exit, if false, toward the tunnel
 * portal. */
public boolean goingTowardExit() {
      return towardExit;
}

/** Returns the goal position for the operation in m,
 * i.e., the position of the fire. */
public double getGoal() {
      return goal;
}
```

```
/** Returns the type of action that the fire fighters
 * should perform. */
public int getAction() {
      return typeOfAction;
}

/** A report function used by a fire fighter pair pair.
 * State time t in s and position x in m in the tunnel.
 * The function then determines what is the next action
 * for the fire fighter pair. */
public void reportingIn(FireFighterPair pair, int t, double x) {
      timeSpent = timeSpent + t;
      distanceMoved = distanceMoved + x;

      if (typeOfAction ==  1) {
            if (Math.round(distanceMoved) >= hoseLengthLong) {
                  typeOfAction = 2;
                  timeSpent = 0;
                  distanceMoved = 0.0;
            }
      } else if (typeOfAction == 2) {
            if (timeSpent >= connectTimeLong) {
                  typeOfAction = 3;
                  timeSpent = 0;
                  distanceMoved = 0.0;
            }
      }  else if (typeOfAction == 3) {
            if (Math.round(distanceMoved) >= hoseLengthShort) {
                  typeOfAction = 4;
                  timeSpent = 0;
                  distanceMoved = 0.0;
            }
      } else if (typeOfAction == 4) {
            if (timeSpent >= connectTimeShort) {
                  typeOfAction = 1;
                  timeSpent = 0;
                  distanceMoved = 0.0;
            }
      }

}

/** Returns the delta temperature rise due to gas temperature
 * and radiation from the fire. Input is the time t in s and
 * the relative position x in m from the fire.
 *
 * If the operation is performed upstream the fire, the
 * consideration is only taken to the radiation form the source. If,
 * on the other hand, the operation is done downsteram the fire, also
 * gas temperature is considered.
 *
 * Evaluation of the delta temperature rise is done based on
 * Eq. 5.3 in (Ingason, Bergqvist, Lönnermark, Frantzich and
 * Hasselrot, 2005). */
public double getFireFighterBodyTemperatureRise(int t, double x) {
      double f = 1.3;
      double A = 1.85;
      double iRad = fireEvent.getRadiationFromFire(t, x);
      double fEff = 0.71;
      double rArC = 0.15;
      double tG = fireEvent.getTavgX(t, x);
      double tB = 37.0;
      double rC = 0.465;
      double M = 300.0;
      double pS = 5940.0;
      double pA = 700.0;
      double rE = 75.0;
      double m = 75.0;
      double cP = 3480.0;
      double deltaTime = 1.0;

      double deltaTemperatureRise = 0.0;

      if (upstream) {
            if (iRad < 5000) {
                  deltaTemperatureRise = ((f * A * ((iRad * fEff * rArC) + ((tunnel.getTemp() -
273 - tB) / rC) + (M / f))) - (f * A * ((pS - pA) / rE))) / (m * cP * deltaTime);
            } else {
```

```
                        deltaTemperatureRise = 2.5;
                    }
            } else {
                    if (iRad < 5000) {
                        deltaTemperatureRise = ((f * A * ((iRad * fEff * rArC) + ((tG - tB) / rC) + (M /
f))) - (f * A * ((pS - pA) / rE))) / (m * cP * deltaTime);
                    } else {
                        deltaTemperatureRise = 2.5;
                    }
            }
            return deltaTemperatureRise;
        }

        /** Private method that defines the start position
         * as well as the direction of the rescue operation.
         * Definition is done based on the defined variables
         * portal and upstream, i.e., if the rescue operation
         * is done through one of the tunnel's two portals or
         * the tunnel's emergency exit (closest one to the fire),
         * and if it is done upstream or downstream the fire. */
        private void setStartPosAndDirectionForOperation() {
            startPos = 0.0;
            towardExit = true;

            if (fireEvent.getWindDirection()) {
                    if (upstream && portal) {
                        startPos = 0.0;
                        towardExit = true;
                    } else if (!upstream && portal) {
                        startPos = tunnel.getLength();
                        towardExit = false;
                    } else if (upstream && !portal) {
                        startPos = 0.0;
                        towardExit = true;
                        for (int i = 0; i < tunnel.getEmergencyExits().size(); i++) {
                            if (tunnel.getEmergencyExits().get(i) <
fireEvent.getDistanceFromEntrance()) {
                                if (tunnel.getEmergencyExits().get(i) > 0.0) {
                                    if (Math.abs(tunnel.getEmergencyExits().get(i) -
fireEvent.getDistanceFromEntrance()) < Math.abs(startPos - fireEvent.getDistanceFromEntrance())) {
                                        startPos = tunnel.getEmergencyExits().get(i);
                                    }
                                }
                            }
                        }
                    } else if (!upstream && !portal) {
                        startPos = tunnel.getLength();
                        towardExit = false;
                        for (int i = 0; i < tunnel.getEmergencyExits().size(); i++) {
                            if (tunnel.getEmergencyExits().get(i) >
fireEvent.getDistanceFromEntrance()) {
                                if (tunnel.getEmergencyExits().get(i) < tunnel.getLength()) {
                                    if (Math.abs(tunnel.getEmergencyExits().get(i) -
fireEvent.getDistanceFromEntrance()) < Math.abs(startPos - fireEvent.getDistanceFromEntrance())) {
                                        startPos = tunnel.getEmergencyExits().get(i);
                                    }
                                }
                            }
                        }
                    }
            } else if (!fireEvent.getWindDirection()) {
                    if (upstream && portal) {
                        startPos = tunnel.getLength();
                        towardExit = false;
                    } else if (!upstream && portal) {
                        startPos = 0.0;
                        towardExit = true;
                    } else if (upstream && !portal) {
                        startPos = tunnel.getLength();
                        towardExit = false;
                        for (int i = 0; i < tunnel.getEmergencyExits().size(); i++) {
                            if (tunnel.getEmergencyExits().get(i) >
fireEvent.getDistanceFromEntrance()) {
                                if (tunnel.getEmergencyExits().get(i) < tunnel.getLength()) {
                                    if (Math.abs(tunnel.getEmergencyExits().get(i) -
fireEvent.getDistanceFromEntrance()) < Math.abs(startPos - fireEvent.getDistanceFromEntrance())) {
                                        startPos = tunnel.getEmergencyExits().get(i);
                                    }
                                }
```

```
                                }
                        }
                } else if (!upstream && !portal) {
                        startPos = 0.0;
                        towardExit = true;
                        for (int i = 0; i < tunnel.getEmergencyExits().size(); i++) {
                                if (tunnel.getEmergencyExits().get(i) <
fireEvent.getDistanceFromEntrance()) {
                                        if (tunnel.getEmergencyExits().get(i) > 0.0) {
                                                if (Math.abs(tunnel.getEmergencyExits().get(i) -
fireEvent.getDistanceFromEntrance()) < Math.abs(startPos - fireEvent.getDistanceFromEntrance())) {
                                                        startPos = tunnel.getEmergencyExits().get(i);
                                                }
                                        }
                                }
                        }
                }
        }
    }

}
```

C.10 RescueSim.java

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;

/** This Java class is a part of TuFT, a decision support
 * tool for tunnel fires. The tool was developed by Karl Fridolf
 * and Håkan Frantzich at Lund University during 2013-2014.
 * Among other things, the tool offers the possibility to
 * simulate fires in tunnels, and furthermore, the effects
 * on evacuation and rescue operation possibilities.
 *
 * The tool's sub models, as well as the conclusions and
 * the results produced by the tool, have not been quality
 * controlled, verified or validated to the extent necessary
 * for quality assurance. TuFT must hence be used with caution.
 * Anyone who uses the results produced by TuFT does so on
 * his/her own responsibility */
public class RescueSim {
      private RescueOperation operation;
      private FireEvent fireEvent;
      private Fire fire;
      private ArrayList<FireFighterPair> pairs;
      private int availableFireFighters;
      private int availableFireFighterPairs;
      private int totalTime;
      private double tempDistance;
      private double tempMovementSpeed;
      private double tempPosition;
      private int tempTime;
      private double tempDeltaT;
      private int maxTime;

      /** Prepares the simulation of a RescueOperation
       * operation during a fire in a tunnel, described
       * by FireEvent fireEvent. */
      public RescueSim(RescueOperation operation,
                  FireEvent fireEvent) {
            this.operation = operation;
            this.fireEvent = fireEvent;
            this.fire = fireEvent.getFire();
            this.pairs = new ArrayList<FireFighterPair>();
            this.availableFireFighters =
                        operation.getNumberOfFireFighters();
            this.availableFireFighterPairs =
                        availableFireFighters / 2;
            this.totalTime = 0;
            this.tempDistance = 0.0;
            this.tempMovementSpeed = 0.0;
            this.tempPosition = 0.0;
            this.tempTime = 0;
            this.tempDeltaT = 0.0;
            this.maxTime = fire.getTotalTime();
      }

      /** Initiates the simulation. */
      public void performRescueSim() {

            for (int t = 1; t < operation.getPreparationTime();
                        t++) {
                  totalTime++;
                  operation.addPosition(totalTime,
                              operation.getLastPosition());
            }

            for(int i = 0; i < availableFireFighterPairs; i++) {
                  pairs.add(i, new FireFighterPair(operation,
                              operation.getLastPosition()));
                  availableFireFighters = availableFireFighters - 2;
                  if (i > 0) {
                        pairs.get(i).setActionTime((int)
                                    Math.round(pairs.get(i).getActionTime() * 0.85));
                  }

                  for (int t = 1; t < pairs.get(i).getActionTime() + 1; t++) {
```

```
                        if (operation.getAction() == 1) {
                              if (operation.operatingUpstream()) {
                                    tempMovementSpeed = pairs.get(i).getUnimpededMovementSpeed();
                                    tempDistance = tempMovementSpeed * 1.0;
                                    if (operation.goingTowardExit()) {
                                          tempPosition = operation.getLastPosition() + tempDistance;
                                    } else {
                                          tempPosition = operation.getLastPosition() - tempDistance;
                                    }
                                    tempDeltaT = operation.getFireFighterBodyTemperatureRise(totalTime,
Math.abs(fireEvent.getDistanceFromEntrance() - operation.getLastPosition()));
                              } else {
                                    tempMovementSpeed = pairs.get(i).getMovementSpeed(totalTime,
Math.abs(fireEvent.getDistanceFromEntrance() - operation.getLastPosition()));
                                    tempDistance = tempMovementSpeed * 1.0;
                                    if (operation.goingTowardExit()) {
                                          tempPosition = operation.getLastPosition() + tempDistance;
                                    } else {
                                          tempPosition = operation.getLastPosition() - tempDistance;
                                    }
                                    tempDeltaT = operation.getFireFighterBodyTemperatureRise(totalTime,
Math.abs(fireEvent.getDistanceFromEntrance() - operation.getLastPosition()));
                              }
                              tempTime = 1;
                        } else if (operation.getAction() == 2) {
                              tempMovementSpeed = 0.0;
                              tempDistance = tempMovementSpeed * 1.0;
                              tempPosition = operation.getLastPosition();
                              tempTime = 1;
                              if (operation.operatingUpstream()) {
                                    tempDeltaT = operation.getFireFighterBodyTemperatureRise(totalTime,
Math.abs(fireEvent.getDistanceFromEntrance() - operation.getLastPosition()));
                              } else {
                                    tempDeltaT = operation.getFireFighterBodyTemperatureRise(totalTime,
Math.abs(fireEvent.getDistanceFromEntrance() - operation.getLastPosition()));
                              }
                        } else if (operation.getAction() == 3) {
                              if (operation.operatingUpstream()) {
                                    tempMovementSpeed = pairs.get(i).getUnimpededMovementSpeed();
                                    tempDistance = tempMovementSpeed * 1.0;
                                    if (operation.goingTowardExit()) {
                                          tempPosition = operation.getLastPosition() + tempDistance;
                                    } else {
                                          tempPosition = operation.getLastPosition() - tempDistance;
                                    }
                                    tempDeltaT = operation.getFireFighterBodyTemperatureRise(totalTime,
Math.abs(fireEvent.getDistanceFromEntrance() - operation.getLastPosition()));
                              } else {
                                    tempMovementSpeed = pairs.get(i).getMovementSpeed(totalTime,
Math.abs(fireEvent.getDistanceFromEntrance() - operation.getLastPosition()));
                                    tempDistance = tempMovementSpeed * 1.0;
                                    if (operation.goingTowardExit()) {
                                          tempPosition = operation.getLastPosition() + tempDistance;
                                    } else {
                                          tempPosition = operation.getLastPosition() - tempDistance;
                                    }
                                    tempDeltaT = operation.getFireFighterBodyTemperatureRise(totalTime,
Math.abs(fireEvent.getDistanceFromEntrance() - operation.getLastPosition()));
                              }
                              tempTime = 1;
                        } else if (operation.getAction() == 4) {
                              tempMovementSpeed = 0.0;
                              tempDistance = tempMovementSpeed * 1.0;
                              tempPosition = operation.getLastPosition();
                              tempTime = 1;
                              if (operation.operatingUpstream()) {
                                    tempDeltaT = operation.getFireFighterBodyTemperatureRise(totalTime,
Math.abs(fireEvent.getDistanceFromEntrance() - operation.getLastPosition()));
                              } else {
                                    tempDeltaT = operation.getFireFighterBodyTemperatureRise(totalTime,
Math.abs(fireEvent.getDistanceFromEntrance() - operation.getLastPosition()));
                              }
                        }

                        totalTime++;
                        operation.addPosition(totalTime, tempPosition);
                        pairs.get(i).addPosition(t, tempPosition);
                        pairs.get(i).addDeltaTemperatureRise(t, tempDeltaT);
                        operation.reportingIn(pairs.get(i), tempTime, tempDistance);
```

```
                        if (totalTime >= maxTime - 1 || Math.abs(operation.getGoal() -
operation.getLastPosition()) < 5.0 || pairs.get(i).getSumDeltaTemperatureRise() >= 2.5) {
                              break;
                        }
                  }

                  if (totalTime >= maxTime - 1 || Math.abs(operation.getGoal() -
operation.getLastPosition()) < 5.0) {
                        break;
                  }

            }

      }

      /** Saves the results of the rescue operation
       * in terms of the position of the frontier
       * fire fighter pair for each time step. Save
       * is done to a comma separated text file
       * termed operationPosition.txt. */
      public void saveOperationPosition() {
            String fileName = "operationPosition.txt";

            try {
                  FileWriter fileWriter = new FileWriter(fileName);
                  BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

                  bufferedWriter.write(fireEvent.versionText());
                  bufferedWriter.newLine();
                  bufferedWriter.newLine();

                  bufferedWriter.write("TIME" + "," + "POSITION [m]" + "," + "DISTANCE [m]");
                  for (int i = 0; i < operation.getArrayOfPosition().size(); i++) {
                        bufferedWriter.newLine();
                        bufferedWriter.write(Integer.toString(i));
                        bufferedWriter.write(",");
                        bufferedWriter.write(Double.toString(operation.getPosition(i)));
                        bufferedWriter.write(",");
                        bufferedWriter.write(Double.toString(Math.abs(operation.getPosition(i) -
operation.getStartPos())));
                  }
                  bufferedWriter.close();
            }
            catch (IOException ex) {
      System.out.println(
          "Error writing to file '"
          + fileName + "'");
            }
      }

      /** Saves a summary of the rescue operation.
       * Save is done to a comma separated text file
       * termed operationSummary.txt. */
      public void saveOperationSummary() {
            String fileName = "operationSummary.txt";

            try {
                  FileWriter fileWriter = new FileWriter(fileName);
                  BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

                  bufferedWriter.write(fireEvent.versionText());
                  bufferedWriter.newLine();
                  bufferedWriter.newLine();

                  bufferedWriter.write("TUNNEL LENGTH:," + fireEvent.getTunnel().getLength() + " m");
                  bufferedWriter.newLine();
                  bufferedWriter.write("POSITION OF FIRE:," + fireEvent.getDistanceFromEntrance());
                  bufferedWriter.newLine();
                  bufferedWriter.newLine();
                  bufferedWriter.write("FIRE FIGHTER PAIR [no],");
                  for (int i = 0; i < pairs.size(); i++) {
                        bufferedWriter.write(Integer.toString(i + 1));
                        bufferedWriter.write(",");
                  }
                  bufferedWriter.newLine();
                  bufferedWriter.write("ACTION TIME [s],");
                  for (int i = 0; i < pairs.size(); i++) {
                        bufferedWriter.write(Integer.toString(pairs.get(i).getActionTime()));
```

```
                        bufferedWriter.write(",");
                    }
                    bufferedWriter.newLine();
                    bufferedWriter.write("START POS [m],");
                    for (int i = 0; i < pairs.size(); i++) {
                        bufferedWriter.write(Double.toString(pairs.get(i).getStartPosition()));
                        bufferedWriter.write(",");
                    }
                    bufferedWriter.newLine();
                    bufferedWriter.write("END POS [m],");
                    for (int i = 0; i < pairs.size(); i++) {
                        bufferedWriter.write(Double.toString(pairs.get(i).getLastPosition()));
                        bufferedWriter.write(",");
                    }
                    bufferedWriter.newLine();
                    bufferedWriter.write("DISTANCE MOVED [m],");
                    for (int i = 0; i < pairs.size(); i++) {
                        bufferedWriter.write(Double.toString(Math.abs(pairs.get(i).getLastPosition() -
pairs.get(i).getStartPosition())));
                        bufferedWriter.write(",");
                    }
                    bufferedWriter.newLine();
                    bufferedWriter.write("TIME IN TUNNEL [s],");
                    for (int i = 0; i < pairs.size(); i++) {
                        bufferedWriter.write(Integer.toString(pairs.get(i).getArrayOfPosition().size() -
1));
                        bufferedWriter.write(",");
                    }

                    bufferedWriter.close();
                }

                catch (IOException ex) {
                System.out.println(
                    "Error writing to file '"
                    + fileName + "'");
                }
            }

}
```

C.11 Sim.java

```
import java.util.ArrayList;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

import org.json.simple.JSONArray;
import org.json.simple.JSONObject;
import org.json.simple.parser.JSONParser;
import org.json.simple.parser.ParseException;

/** This Java class is a part of TuFT, a decision support
 * tool for tunnel fires. The tool was developed by Karl Fridolf
 * and Håkan Frantzich at Lund University during 2013-2014.
 * Among other things, the tool offers the possibility to
 * simulate fires in tunnels, and furthermore, the effects
 * on evacuation and rescue operation possibilities.
 *
 * The tool's sub models, as well as the conclusions and
 * the results produced by the tool, have not been quality
 * controlled, verified or validated to the extent necessary
 * for quality assurance. TuFT must hence be used with caution.
 * Anyone who uses the results produced by TuFT does so on
 * his/her own responsibility */
public class Sim {
    private Boolean measure;
    private Boolean evacuation;
    private Boolean operation;
    private String tunnelType;
    private EvacSim evacSim;
    private RescueSim rescueSim;
    private Tunnel tunnel;
    private RescueOperation rescueOperation;
```

```java
	private Fire fire;
	private FireEvent fireEvent;
	private Train train;
	private Measures measures;
	private ArrayList<Person> individuals;
	private ArrayList<Group> groups;
	private ArrayList<Person> passengers;
	private ArrayList<Person> allEvacuees;
	private ArrayList<Double> measurementPositions;

	/** Prepares full simulation by reading from settings.JSON file. */
	public Sim(String settingsFileName) {

		JSONParser parser = new JSONParser();

		try {

			Object obj = parser.parse(new FileReader(settingsFileName));
			JSONObject jsonObject = (JSONObject) obj;

			// TYPE OF SIMULATION //
			JSONObject jsonSimulation = (JSONObject) jsonObject.get("simulation");

			measure = (Boolean) jsonSimulation.get("measure");
			evacuation = (Boolean) jsonSimulation.get("evacuation");
			operation = (Boolean) jsonSimulation.get("operation");

			// TUNNEL //
			JSONObject jsonTunnel = (JSONObject) jsonObject.get("tunnel");

			tunnelType = (String) jsonTunnel.get("tunnelType");
			double length = (Double) jsonTunnel.get("length");
			double width = (Double) jsonTunnel.get("width");
			double height = (Double) jsonTunnel.get("height");
			double windSpeed = (Double) jsonTunnel.get("windSpeed");
			double ambientTemperature = (Double) jsonTunnel.get("ambientTemperature");
			double distanceBetweenEmergencyExits = (Double)
jsonTunnel.get("distanceBetweenEmergencyExits");

			tunnel = new Tunnel(length, width, height, windSpeed, ambientTemperature,
distanceBetweenEmergencyExits);

			// FIRE //
			JSONObject jsonFire = (JSONObject) jsonObject.get("fire");
			String fireType = (String) jsonFire.get("fireType");

			if (fireType.equals("linear")) {
				double alfaGrowth = (Double) jsonFire.get("alfaGrowth");
				double alfaDecay = (Double) jsonFire.get("alfaDecay");
				boolean linearGrowthRate = true;
				double qMax = (Double) jsonFire.get("qMax");
				int timeFullBurn =  ((Long) jsonFire.get("timeFullBurn")).intValue();
				double massOpticalDensity = (Double) jsonFire.get("massOpticalDensity");
				double heatOfCombustion = (Double) jsonFire.get("heatOfCombustion");
				double chi = (Double) jsonFire.get("chi");
				double yieldCo2 = (Double) jsonFire.get("yieldCo2");
				double yieldCo = (Double) jsonFire.get("yieldCo");
				double yieldHcn = (Double) jsonFire.get("yieldHcn");
				fire = new Fire(linearGrowthRate, alfaGrowth, alfaDecay,
						qMax, timeFullBurn, massOpticalDensity, heatOfCombustion,
						chi, yieldCo2, yieldCo, yieldHcn);
			} else if (fireType.equals("squared")) {
				double alfaGrowth = (Double) jsonFire.get("alfaGrowth");
				double alfaDecay = (Double) jsonFire.get("alfaDecay");
				boolean linearGrowthRate = false;
				double qMax = (Double) jsonFire.get("qMax");
				int timeFullBurn =  ((Long) jsonFire.get("timeFullBurn")).intValue();
				double massOpticalDensity = (Double) jsonFire.get("massOpticalDensity");
				double heatOfCombustion = (Double) jsonFire.get("heatOfCombustion");
				double chi = (Double) jsonFire.get("chi");
				double yieldCo2 = (Double) jsonFire.get("yieldCo2");
				double yieldCo = (Double) jsonFire.get("yieldCo");
				double yieldHcn = (Double) jsonFire.get("yieldHcn");
				fire = new Fire(linearGrowthRate, alfaGrowth, alfaDecay,
						qMax, timeFullBurn, massOpticalDensity, heatOfCombustion,
						chi, yieldCo2, yieldCo, yieldHcn);
			} else if (fireType.equals("exponential")) {
				double qMaxExp = (Double) jsonFire.get("qMaxExp");
				double eTot = (Double) jsonFire.get("eTot");
```

```
                        double tMax = (Double) jsonFire.get("tMax");
                        double massOpticalDensity = (Double) jsonFire.get("massOpticalDensity");
                        double heatOfCombustion = (Double) jsonFire.get("heatOfCombustion");
                        double chi = (Double) jsonFire.get("chi");
                        double yieldCo2 = (Double) jsonFire.get("yieldCo2");
                        double yieldCo = (Double) jsonFire.get("yieldCo");
                        double yieldHcn = (Double) jsonFire.get("yieldHcn");
                        fire = new Fire(qMaxExp, eTot, tMax,
                                    massOpticalDensity, heatOfCombustion,
                                    chi, yieldCo2, yieldCo, yieldHcn);
                    }

                    // FIRE EVENT //
                    JSONObject jsonFireEvent = (JSONObject) jsonObject.get("fireEvent");

                    double firePos = (Double) jsonFireEvent.get("firePos");
                    boolean windDirExit = (Boolean) jsonFireEvent.get("windDirExit");

                    fireEvent = new FireEvent(tunnel, fire, firePos, windDirExit);

                    // INDIVIDUALS //
                    if (evacuation && tunnelType.equals("road")) {
                        JSONArray jsonIndividuals = (JSONArray) jsonObject.get("individuals");
                        individuals = new ArrayList<Person>();

                        for(int i = 0; i < jsonIndividuals.size(); i++) {
                            JSONObject jsonIndividual = (JSONObject) jsonIndividuals.get(i);
                            double personPosition = (Double) jsonIndividual.get("personPosition");
                            int recognitionTime = ((Long)
jsonIndividual.get("recognitionTime")).intValue();
                            int responseTime = ((Long) jsonIndividual.get("responseTime")).intValue();
                            int timeToLeaveVehicle = ((Long)
jsonIndividual.get("timeToLeaveVehicle")).intValue();
                            boolean chooseEmergencyExit = (Boolean)
jsonIndividual.get("chooseEmergencyExit");
                            boolean deterministic = (Boolean) jsonIndividual.get("deterministic");

                            individuals.add(new Person(personPosition, recognitionTime,
                                        responseTime, timeToLeaveVehicle, chooseEmergencyExit,
                                        deterministic, tunnel, fireEvent));
                        }
                    }

                    // GROUPS //
                    if (evacuation && tunnelType.equals("road")) {
                        JSONArray jsonGroups = (JSONArray) jsonObject.get("groups");
                        groups = new ArrayList<Group>();

                        for(int i = 0; i < jsonGroups.size(); i++) {
                            JSONObject jsonGroup = (JSONObject) jsonGroups.get(i);
                            int count = ((Long) jsonGroup.get("count")).intValue();
                            double personPositionIntervalFrom = (Double)
jsonGroup.get("personPositionIntervalFrom");
                            double personPositionIntervalTo = (Double)
jsonGroup.get("personPositionIntervalTo");
                            int recognitionTimeIntervalFrom = ((Long)
jsonGroup.get("recognitionTimeIntervalFrom")).intValue();
                            int recognitionTimeIntervalTo = ((Long)
jsonGroup.get("recognitionTimeIntervalTo")).intValue();
                            int responseTimeIntervalFrom = ((Long)
jsonGroup.get("responseTimeIntervalFrom")).intValue();
                            int responseTimeIntervalTo = ((Long)
jsonGroup.get("responseTimeIntervalTo")).intValue();
                            int timeToLeaveVehicleFrom = ((Long)
jsonGroup.get("timeToLeaveVehicleFrom")).intValue();
                            int timeToLeaveVehicleTo = ((Long)
jsonGroup.get("timeToLeaveVehicleTo")).intValue();
                            boolean chooseEmergencyExit = (Boolean)
jsonGroup.get("chooseEmergencyExit");
                            boolean deterministic = (Boolean) jsonGroup.get("deterministic");

                            groups.add(new Group(count, personPositionIntervalFrom,
personPositionIntervalTo,
                                        recognitionTimeIntervalFrom, recognitionTimeIntervalTo,
                                        responseTimeIntervalFrom, responseTimeIntervalTo,
                                        timeToLeaveVehicleFrom, timeToLeaveVehicleTo,
                                        chooseEmergencyExit, deterministic, tunnel, fireEvent));
                            }
                    }
```

```
                // TRAIN //
                if (evacuation && tunnelType.equals("rail")) {
                        JSONObject jsonTrain = (JSONObject) jsonObject.get("train");

                        double position = (Double) jsonTrain.get("position");
                        double trainLength = (Double) jsonTrain.get("trainLength");
                        int numberOfExits = ((Long) jsonTrain.get("numberOfExits")).intValue();
                        double exitWidth = (Double) jsonTrain.get("exitWidth");
                        int numberOfPassengers = ((Long)
jsonTrain.get("numberOfPassengers")).intValue();
                        int recognitionTime = ((Long) jsonTrain.get("recognitionTime")).intValue();
                        int responseTime = ((Long) jsonTrain.get("responseTime")).intValue();
                        boolean chooseEmergencyExit = (Boolean) jsonTrain.get("chooseEmergencyExit");
                        boolean deterministic = (Boolean) jsonTrain.get("deterministic");

                        train = new Train(position, trainLength, numberOfExits,
                                    exitWidth, numberOfPassengers, recognitionTime,
                                    responseTime, chooseEmergencyExit, deterministic,
                                    tunnel, fireEvent);
                        passengers = train.getArrayOfPassengers();
                }

                // RESCUE OPERATION //
                if (operation) {
                        JSONObject jsonOperation = (JSONObject) jsonObject.get("operation");

                        int preparationTime = ((Long) jsonOperation.get("preparationTime")).intValue();
                        boolean upstream = (Boolean) jsonOperation.get("upstream");
                        boolean portal = (Boolean) jsonOperation.get("portal");
                        int numberOfFireFighters = ((Long)
jsonOperation.get("numberOfFireFighters")).intValue();
                        boolean thermalImaging = (Boolean) jsonOperation.get("thermalImaging");
                        boolean muchAir = (Boolean) jsonOperation.get("muchAir");
                        double hoseLengthLong = (Double) jsonOperation.get("hoseLengthUneven");
                        double hoseLengthShort = (Double) jsonOperation.get("hoseLengthEven");
                        int connectTimeLong = ((Long)
jsonOperation.get("connectTimeUneven")).intValue();
                        int connectTimeShort = ((Long) jsonOperation.get("connectTimeEven")).intValue();

                        rescueOperation = new RescueOperation(fireEvent, preparationTime, upstream,
                                    portal, numberOfFireFighters, thermalImaging, muchAir,
hoseLengthLong,
                                    hoseLengthShort, connectTimeLong, connectTimeShort);
                }

                // MEASURES //
                if (measure) {
                        JSONArray jsonMeasurementPositions = (JSONArray)
jsonObject.get("measurementPositions");
                        measurementPositions = new ArrayList<Double>();
                        for (int i = 0; i < jsonMeasurementPositions.size(); i++) {
                                JSONObject jsonMeasurementPosition = (JSONObject)
jsonMeasurementPositions.get(i);
                                double pos = (Double) jsonMeasurementPosition.get("pos");
                                measurementPositions.add(pos);
                        }

                        measures = new Measures(fireEvent, 0, fire.getTotalTime() - 1,
measurementPositions);

                }

            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            } catch (ParseException e) {
                e.printStackTrace();
            }
        }

        /** Initiates simulation. */
        public void run() {
            this.saveHrrToFile();

            if (evacuation) {
                allEvacuees = new ArrayList<Person>();
```

```
                if (tunnelType.equals("road")) {
                        allEvacuees.addAll(individuals);
                        for (int i = 0; i < groups.size(); i++) {
                                allEvacuees.addAll(groups.get(i).getPeople());
                        }
                }

                if (tunnelType.equals("rail")) {
                        allEvacuees.addAll(passengers);
                }

                this.evacSim = new EvacSim(fire, fireEvent, allEvacuees);
                evacSim.performEvacSim();
                this.saveAllEvacToFile();
        }

        if (operation) {
                this.rescueSim = new RescueSim(rescueOperation, fireEvent);
                rescueSim.performRescueSim();
                this.saveAllOperationToFile();
        }

        if (measure) {
                this.saveMeasuresToFile();
        }

    }

    /** Saving evacuation simulation to files. */
    private void saveAllEvacToFile() {
        evacSim.saveEvacFedAsphyxia();
        evacSim.saveEvacFidHeat();
        evacSim.saveEvacPosition();
        evacSim.saveEvacEndPosition();
        evacSim.saveEvacuatedPerTime();
        evacSim.saveEvacTemperature();
        evacSim.saveEvacVisibility();
        evacSim.saveEvacConcentrationCo();
        evacSim.saveEvacConcentrationCo2();
        evacSim.saveEvacConcentrationHcn();
        evacSim.saveEvacConcentrationO2();
        evacSim.saveEvacSummary();
    }

    /** Saving rescue operation simulation to files. */
    private void saveAllOperationToFile() {
        rescueSim.saveOperationSummary();
        rescueSim.saveOperationPosition();
    }

    /** Saving HRR to file. */
    private void saveHrrToFile() {
        fire.printHrr();
    }

    /** Saving prediction measurements to files. */
    private void saveMeasuresToFile() {
        measures.printTempToFile();
        measures.printGasToFile();
        measures.printExtinctionCoeffToFile();
        measures.printVisibilityToFile();
        measures.printHeatTransferToFile();
        measures.printFireRadiationToFile();
        measures.printBackLayeringDistance();
    }

    /** Saving temporary evacuation simulation data to
     * command prompt (only done for verification of TuFT). */
    public void saveEvacSimTemp() {
        evacSim.saveEvacSimDataTemp();
    }

}
```

C.12 App.java

```
/** This Java class is a part of TuFT, a decision support
 * tool for tunnel fires. The tool was developed by Karl Fridolf
 * and Håkan Frantzich at Lund University during 2013-2014.
 * Among other things, the tool offers the possibility to
 * simulate fires in tunnels, and furthermore, the effects
 * on evacuation and rescue operation possibilities.
 *
 * The tool's sub models, as well as the conclusions and
 * the results produced by the tool, have not been quality
 * controlled, verified or validated to the extent necessary
 * for quality assurance. TuFT must hence be used with caution.
 * Anyone who uses the results produced by TuFT does so on
 * his/her own responsibility */
public class App {

      public static void main(String[] args) {

            Sim simulation = new Sim("settings.JSON");
            simulation.run();

      }

}
```