Pedestrian Traffic Simulation Platform

Fredrik Johansson
Förord

Föreliggande notat beskriver en plattform för simulering av fotgängarflöden. Simuleringssplattformen har utvecklats av Fredrik Johansson inom ramen för projektet ‘‘Utvärdering av stora bytespunkter med hjälp av simulering’’ finansierat av Trafikverket.

Linköping, April 2013

Andreas Tapani
Projektledare
Kvalitetsgranskning


Quality review

Review seminar was carried out on 19 Mars 2013 where Johan Olstam reviewed and commented on the report. Fredrik Johansson has made alterations to the final manuscript of the report 10 April 2013. The research director of the project manager Maud Göthe-Lundgren examined and approved the report for publication on (18 April 2013).
Content

Summary ................................................................. 5
Sammanfattning ..................................................... 7
1 Introduction .......................................................... 9
1.1 Purpose ............................................................ 9
1.2 Outline ........................................................... 10
1.3 Notation and terminology ....................................... 10
2 Structure ............................................................. 11
2.1 Abstract model structure ....................................... 11
2.2 Implementation structure ....................................... 12
3 The GUI ............................................................... 14
3.1 Main Window ...................................................... 14
3.2 SimDisplay ......................................................... 14
4 The Simulator ........................................................ 15
5 The operational level - The Social Force Model .......... 16
5.1 The social force .................................................. 16
5.2 Physical extent ................................................... 17
5.3 The preferred velocity ......................................... 17
5.4 Interaction with obstacles ..................................... 18
5.5 The equations of motion ....................................... 18
6 The tactical level - Navigation ..................................... 20
6.1 The Fast Marching Method ..................................... 20
7 The strategical level - Activity Planning ...................... 22
7.1 Paths ............................................................... 22
7.2 Origins ............................................................ 22
7.3 Destinations ...................................................... 22
7.4 Waiting areas ..................................................... 23
Appendix A Parameters
Appendix B Doxygen generated documentation
Pedestrian Traffic Simulation Platform

av Fredrik Johansson
VTI (Swedish National Road and Transport Research Institute)
581 95 Linköping, Sweden

Summary

This is documentation of a simulation platform for simulation of pedestrian traffic, such as at public transport interchange stations. The simulation platform takes a microscopic perspective, and thus simulates each pedestrian separately and uses as a base the so-called “social force model”. This document describes the model and the implementation of it in detail to facilitate a discussion of the methodology, expose it to review, and simplify for others to implement similar models.
**Sammanfattning**

1 Introduction

Efficient transfers between modes of transport is important for the functionality of a public transport system, since many trips include such transfers. A large part of the transfers in a PT system takes place at the central interchange station, which means that the functionality of the system is dependent on that the central interchange station is designed to allow efficient transfers. This usually means making the walking distance between the stops at the station short, but short distances means less space to move, and higher probability for congestion problems, which leads to delays. This makes the problem of designing an efficient interchange station nontrivial, and suitable tools are needed to predict the traffic situation at the station, before it is constructed.

Delays are not the only effect of congestion. Another important effect is the discomfort it causes. For public transport to be a real alternative to the private car, it needs not only to compete with its travel speed, but also with its comfort, and for many public transport trips this includes the comfort during the transfer. One problem with discomfort is that it is hard to measure, especially if the infrastructure is not yet built.

The problem of predicting the delays and the discomfort experienced by pedestrians at future facilities, and comparing the results of different designs, can be approached by use of microscopic simulation. This document is a detailed documentation of a microscopic simulation platform constructed to address this and related problems.

The simulation platform was developed during a research project at the Swedish National Road and Transport Institute (VTI) and Linköping University, Department of Science and Technology, and funded by the Swedish National Transport Administration (Trafikverket). The project was also supported with expertise from Linköping municipality, through Jörgen Haslum, and the regional public transport administration, Östgötatrafiken, through Lars Flintzberg.

1.1 Purpose

The purposes of this document are manifold, but the main purpose is to provide a detailed description of a method, microscopic simulation of pedestrian traffic. This method consists of two separate stages, model development and implementation, of which the first is the most commonly occurring in the scientific literature, while the other often is only briefly discussed or even completely neglected. This is unfortunate since the implementation stage clearly is a part of the methodology, and the critical examination of it is essential for the assessment of microscopic simulation of pedestrian traffic as a reliable scientific method. Thus the primary purpose of the development and publication of this document is to describe the method and expose it to the review of the scientific community. Without detailed descriptions of the various possible implementation strategies, we cannot hope to have a constructive discussion on the correct or preferable strategy.

The second purpose of this document is to facilitate the development of, and discussion around, models for pedestrian traffic simulation. Since a discussion of a model is rather limited if no implementation of it is available, and since the results of a model is highly dependent on the implementation details, serious discussions of models cannot take place without reference to a well documented implementation. Thus the second and third purposes of this document are, respectively, to simplify the implementation of similar simulation platforms to facilitate the comparison of model predictions, and to serve as reference document whenever the simulation platform is used in research.

The last purpose of this document is of internal character. The documentation guaran-
tees that the code is transferable to another user or developer.

The targeted readers of this document can be coarsely divided into two categories with somewhat differing interests in different parts of the document. The first category consists of readers that have developed or are developing their own simulation platform, and are interested to compare or be inspired by the model implementation. The second category are readers that have been referred to this document from an article where the lack of available space prohibited the discussion of all details of the model or the implementation. Due to this expected difference in reader interests, a brief presentation of the outline, to guide the readers of the document, is given next.

1.2 Outline

In section 2, an overview of the structure of the PTSP is given. First in section 2.1, from a model perspective and then, in section 2.2, from an implementation perspective. Here are some central concepts introduced, and the relations between important software components described. This is a good starting point for any reader.

In section 3, the graphical user interface and its components are briefly discussed. Then in section 4 the core of the simulation, the simulator class, is described in detail.

The remainder of the main document, section 5 to section 7, is structured according to the three model levels, operational, tactical, and strategical, discussed in section 2.1. These three sections specify both the model and its implementation, starting with the operational level.

Appendix B is a document generated automatically from the commented source code by Doxygen. The document is a classical software documentation, listing all classes and methods with short descriptions. Inheritance and collaborations graphs are also given. In the end of the documentation there is an alphabetic index of classes and methods which may be useful to refer to during the reading of the main document.

1.3 Notation and terminology

This text contains some notation related to the description of program code to help clarify without lengthy explanations. All references to code follow the same notation as in the code, with some typographic help to make it even clearer. A class is denoted by ClassName, functions by functionName, virtual functions by virtualFunctionName, objects, including simple variables, by className.

The code is based on the Qt framework, which means that it uses many of its classes and functions Qt-Project 2013. These will not be explained here, instead the reader is referred to the excellent Qt documentation. Components of the Qt framework can easily be recognized by the prefix q, or Q.
2 Structure

The overall structure of the simulation platform can be viewed in two ways, either from the model point of view or from the implementation point of view. In this chapter the structure of PTSP is presented, first from a model perspective, and then from the perspective of the implementation.

2.1 Abstract model structure

From the model point of view we can see the simulation platform as a three level structure, as discussed by Hoogendoorn et al. 2004. Figure 1 presents the notion of the three levels, relates them to the corresponding behaviors, and summarizes the current state of the PTSP in the rightmost column. The underlying idea to the structure is that the three levels represent different mental processes, ranging from instinctive at the operational level to premeditated at the strategical level. Since the behavior at the different levels are manifestations of different mental processes, it is motivated to use different models to simulate the different behaviors.

![Figure 1 The general structure of the simulation platform from a model perspective.](image)

The operational level, which corresponds to evasive maneuvers, is modeled using the social force model introduced by Helbing et al. 1995. This model is used both in research and commercial software and has been examined extensively in scientific publications during the last two decades, see for example Moussaïd et al. (2009), Ma et al. (2010), and A. Johansson (2009). The basic idea behind the social force model is that pedestrians have a personal sphere into which they prefer that no one else enters and they control their acceleration in order to achieve this.

The tactical level, corresponding to path choice, is modeled using a shortest path algorithm called the fast marching method (FMM), developed by Sethian 1996. The algorithm is executed in advance of the simulation and results in a field of preferred directions for each destination.

The strategical level corresponds to activity planning and is in this work considered input data. A predetermined number of walkers is created at each origin and a predetermined fraction of these walk to each destination. Thus the model does not include any modeling of activity planning. The PTSP can be extended to include activity planning, if such model is given, either by a separate program generating input data to the PTSP, or by extension of the platform by construction of an activity planning module.
2.2 Implementation structure

The layered structure from the previous section is respected in the implementation due to the object oriented approach chosen for the implementation. In this section an overview of the most important classes will be given.

There are two classes in PTSP that represents real physical objects: Walker representing pedestrians, and Obstacle representing any kind of obstacle, that is any point where it is not possible to walk. These two classes are described in detail below, following a brief description of the two “book-keeping” classes: the Simulator and the SimWindow.

A simulator runs the main simulation loop, and owns all the objects in the simulated area. The simulation is time controlled, discrete in time and continuous in space. The simulator contains a list of the walkers that are currently in the simulated area and communicates the information needed to calculate the forces. It also contains lists of origins, destinations and obstacles, and handles data output. The Simulator is further described in section 4 and in the appendix.

The simulator is owned by an instance of the programs master class, the simWindow, which inherits QMainWindow. The SimWindows main purpose is to define the graphical user interface (GUI), but it also includes descriptions of the standard scenarios and the procedure to loop over several simulation runs with different random seeds.

2.2.1 The Walkers

A walker is the representation of a pedestrian in the model. A walker has a set of variables describing its current state: position, velocity, orientation, etc., a discrete variable determining its current activity, Walking or Waiting, and a set of parameters specifying the model by which it interacts with its surroundings. These parameters and their default values can be found in appendix A. The Walker is here introduced by describing the life of a walker object from its construction until its destruction.

A walker is created by an origin (section 7.2). At its creation it is given an individual preferred speed and a walkPath (an ordered list of destinations to visit, section 7). The preferred speed is the speed the walker will keep if it is not perturbed by any other walkers or any obstacles.

On its way toward the next destination, the walker navigate according to a vector field of preferred directions leading it the shortest path to the destination. This navigation field of preferred directions is discretized on a grid with 1 dm × 1 dm cells, but the walker moves in continuous space.

If the walker encounters any other walkers it will interact with them to avoid collisions. The form of this interaction is specified by the social force model (section 5). The social force model is responsible not only for avoidance of other walkers and obstacles, but also for how the walker adopts to its preferred velocity.

As the walker arrives at the destination, the walker checks if the destination is the last in its walkPath, if so, the walker is destroyed, otherwise it obtains a waiting time from a distribution specific to the destination and spends that time waiting, before continuing towards its next destination.
2.2.2 Obstacles

The obstacles constitute the map of the simulated area; everything that is not an obstacle belongs to the walkable area. The obstacles are represented, in different ways, on both the tactical and the operational level. On the operational level the obstacles are represented by a force field directed away from the obstacles. These forces guarantee that the walkers do not collide with the obstacles. On the tactical level the obstacles influences the construction of the navigation field of preferred directions. The field is constructed such that its integral lines are the shortest walkable paths to the destination, that is the shortest paths not intersecting any obstacles. Both the navigation fields and the obstacle force field are defined on a grid with cell size 1 dm × 1 dm, and calculated before the simulation starts.

Thus the obstacles themselves do not directly affect the walkers; their effect is only indirect through the obstacle force field and the navigation field of preferred directions. The role of the obstacles are thus to make the construction of these fields possible. The obstacles are created as part of the scenario specification by the simWindow, which calls the addWall() method of the Simulator. An obstacle is specified by its two end points, and consists of the line segment between them. Complex obstacles can be represented by several elementary obstacles. When an obstacle is created, the force from it is added to the forcefield of all obstacles. This field is read at each time step during the simulation by the walkers to determine if they are affected by any obstacle.
3 The GUI

The main purpose of the Graphical User Interface (GUI) is to simplify the verification of the model and to visualize the traffic situation. This is done by providing a `simDisplay` which displays a partially interactive map of the simulated area, and also has the ability to display an animated representation of the simulation in real time or quicker.

The second purpose to include a GUI is to enforce a code structure that simplifies a future implementation of a full GUI for end users.

3.1 Main Window

The `mainWindow` is the actual graphical main window of the application, but it also owns the main process and indirectly all other components.

The graphical window consists mainly of the `simDisplay`, but also some menus and buttons to control the simulation. The interface is incomplete, and for efficient use of the application access to the source code is needed for scenario specification.

In the constructor of the `mainWindow` the `simulator` is created and the scenario specified.

3.2 SimDisplay

The `simDisplay` contains the graphical representation of the simulated system. These graphical representations are implemented as objects separate from the objects of the model, i.e. a `walker` object represents a pedestrian in the model, and a `walkerDot` is the graphical representation of a `walker`, and similarly for other objects. This construction facilitates running the simulation without animation, with no computational overhead from the animation.
4 The Simulator

A simulator completely defines all characteristics of a simulation run, except the run time. A simulation run is executed through either the run ($T_{\text{run}}$) method or the rawRun ($T_{\text{run}}$) method, depending on whether animation should be provided or not. An overview of the main loop can be seen in figure 2.

![Diagram of the main simulation loop, started by run() or rawRun().](image)

The Update State phase loops over the list of walkers and updates their position, velocity etc. The Create walker phase loops over the list of origins and creates a walker if a certain time gap has passed. The Update forces phase loops over all pairs of walkers and calculates the forces by which the walkers interacts. It also calculates the other forces. When the loop stops, the control is handed back to the simWindow.

The simulator has a list of origins, a list of destinations, and a list of walkers, all implemented as QVectors. The simulator also has a vector field with the obstacle forces calculated according to equation (9) at a square grid, implemented as a QVector< QVector<QVector2D> >, and a scalar field on the same grid containing the distance to the closest obstacle, also implemented using a QVector combination.

The simulator also handles the output through DataFile, one file for each of: coordinates, velocities, forces, and discomfort, which are collected every time step by every walker, and one file for data that only needs to be collected once per walker, such as preferred speed and creation time. The data is stored by each walker until it is destroyed; then it is written to the files. The DataFile class is simply a wrapper for the QFile class, with different formatting included in the overloaded "<<" operator for the different output data types.

Further the Simulator loads model parameters from xml files using classes inheriting the Settings class. This class converts both ways between xml and the QStandardItemModel, with entries defined by the child. This structure facilitates a future implementation of GUI support for manipulation of the parameters, in contrast to the present need to manipulate the xml files directly.
5  The operational level - The Social Force Model

The operational level of the model is concerned with modeling the instinctive behavior of the pedestrians, basically evasive maneuvers, and how to adapt to their preferred speed. This behavior is modeled using a version of the social force model suggested by A. Johansson et al. 2008, with some additions to model waiting pedestrians.

The implementation of the social force model is completely contained in the Walker class, including the additional waiting models. The only result of the higher level models that the operational level is dependent upon is the activity and the preferred velocity of the walkers. The activity of a walker is set when the walker enters or exits a waiting area. The preferred direction of a walker is read by the walker at each time step from the navigation field produced in advance, see Section 6.

There are several versions of the social force model; in this section the version used in the PTSP is described.

5.1 The social force

The social force model postulates that the behavior of walker \( j \) is affected by the presence of walker \( i \) according to a force given by a potential of the form

\[
V_{ij} = V(r_{ij}, v_{ij}) = F_0 \sigma e^{-b(r_{ij}, v_{ij})/\sigma},
\]

where \( r_{ij} \) is the position of walker \( j \) relative to walker \( i \), \( v_{ij} \) is the velocity of walker \( i \) relative to walker \( j \), and \( F_0 \) and \( \sigma \) are model parameters describing the magnitude and the range scale of the force, respectively, both assumed to be constant over the population. The function \( b(r_{ij}, v_{ij}) \) determines the dependence of the force on the relative orientations of the walkers. For an overview of proposed forms of \( b(r_{ij}, v_{ij}) \), see A. Johansson 2009. The specification implemented in the PTSP is an elliptical version with

\[
b(r_{ij}, v_{ij}) = \frac{1}{2} \sqrt{(r_{ij} + |r_{ij} - v_{ij}T_s|)^2 - (T_s v_{ij})^2},
\]

where \( T_s \) is the look-ahead time, i.e. the extent to which the affected walker extrapolates the relative position of the affecting walker. This extrapolation is what gives the potential its elliptic form. \( T_s \) is assumed to be constant over the population.

Since the force is the gradient of the potential,

\[
F_{ij}^s = -\nabla r V[b(r_{ij}, v_{ij})],
\]

and,

\[
\nabla r b(r_{ij}, v_{ij}) = \frac{1}{4b(r_{ij}, v_{ij})} (r_{ij} + |r_{ij} - v_{ij}T_s|) \left( \frac{r_{ij}}{r_{ij}} + \frac{r_{ij} - T_s v_{ij}}{|r_{ij} - T_s v_{ij}|} \right),
\]

the explicit form of the force is

\[
F_{ij}^s = \frac{(r_{ij} + |r_{ij} - v_{ij}T_s|)}{4b(r_{ij}, v_{ij})} \left( \frac{r_{ij}}{r_{ij}} + \frac{r_{ij} - v_{ij}T_s}{|r_{ij} - v_{ij}T_s|} \right) F_0 e^{-b(r_{ij}, v_{ij})/\sigma}
\]

Since the forces are social, not physical, they should only affect the walker to the extent it is aware of the existence of its source. Thus a walker should be less affected by
walkers behind it, or in general, outside of its field of vision. This is modeled by introducing weights \( w(\varphi_{ij}) \) depending on the angle \( \varphi_{ij} \) between the preferred direction of the affected walker and the relative position of the affecting walker. The weights are assumed to have the form,

\[
w(\varphi_{ji}) = \left( \lambda + (1 - \lambda) \frac{1 - \cos(\varphi_{ji})}{2} \right),
\]

where \( \lambda \) is a model parameter, describing the strength of the isotropy.

Observe that this specification of the social force, with \( b(r_{ij}, v_{ij}) \) according to equation (2), and \( F^s_0, \sigma, \) and \( T_s \) constant over the population, implies that \( F^s_{ij} = -F^s_{ji} \). But since this is a special feature of this specification of the social force potential, the symmetry is not utilized in the implementation to allow for future versions of the PTSP to implement other specifications of the social force model, for example by allowing the parameters \( F^s_0, \sigma, \) and \( T_s \) to vary over the population. The calculation of the walker–walker social force is implemented in the Walker method reactTo(), and can easily be changed to some other specification. To speed up the execution the square of distance between the walkers are calculated, and reactTo() is called only if the distance is smaller than a threshold value, by default set to 5 meters.

5.2 Physical extent

To model the physical extent of the pedestrians an additional force between each pair of walkers is added,

\[
F^{phys}_{ij} = F^{phys}_0 e^{(2R - r_{ij})/\sigma^{phys} r_{ij}},
\]

where \( R \) is the radius of the walkers, assumed equal for all walkers. The scale of the range of the force, \( \sigma^{phys} \), is much shorter than the scale of the social force term equation (5), and is assumed to be equal for all walkers A. Johansson 2009.

The physical force is calculated in the same phase as the social force by the reactTo() method of the Walker class.

5.3 The preferred velocity

Pedestrian \( i \) adopts to its preferred velocity by supplying a force,

\[
F^p_i = \frac{1}{\tau} (v^p_i - v_i),
\]

where \( \tau \) is the time scale for the adaptation to the preferred velocity, \( v_i \) is the current velocity of the pedestrian and \( v^p_i \) is the preferred velocity of the pedestrian. The direction of \( v^p_i \) is given as the preferred direction from the tactical level and the size given by the preferred speed \( v^p_i \) of the pedestrian. The preferred speed is drawn from a Gaussian distribution with a mean of 1.37 m/s and a standard deviation of 0.3 m/s, constrained to the interval 0.5 m/s to 2.5 m/s. This distribution is approximately adopted from TRB 2010. The model parameter \( \tau \), taken as constant over the population, determines how quickly the walkers adopt to their preferred speed: the lower the value, the quicker and more aggressively the walkers adopt to their preferred velocity.

In the update state phase of the main simulation loop, after the walker has moved to its new position, the walker requests the preferred direction at its position from the current destination in its destination list. This together with the preferred speed of the walker becomes its new preferred velocity. Later, in the update force phase of the main

VTI notat 16-2013 17
simulation loop, $F_{i}^{p}$ is calculated according to equation (8) and added to the total force acting on the walker.

### 5.4 Interaction with obstacles

The obstacle avoidance behavior is partly encoded in the navigation model, which makes sure that the preferred velocity of a *walker* is parallel to, or directed away from, the boundary of the obstacle, close to the boundary. This is, however, not enough since the presence of other *walkers* may give a resulting force directed towards the obstacle. To make sure that *walkers* do not collide with obstacles, an obstacle force similar to the *walker–walker* social force is defined as

$$
F_{in}^{obst} = F_{0}^{obst} e^{-r_{in}/\sigma_{obst}} \frac{r_{in}}{r_{in}},
$$

where $r_{in}$ is the position of *walker* $i$ relative to the closest point of the closest obstacle $n$, and $F_{0}^{obst}$ and $\sigma_{obst}$ are the magnitude and range scale of the force, respectively.

An obstacle force field is calculated in advance of the simulation on the same grid as the navigation fields of the tactical level. During the simulation the *walkers* read from the field at each time step after the calculation of the *walker–walker* interactions.

### 5.5 The equations of motion

The sum of all the forces affecting the walker determines the desired acceleration of the walker,

$$
\ddot{a}_i = \sum_j \left( w(\varphi_{ji}) F_{ji}^{s} + F_{ji}^{phys} \right) + F_{in}^{obst} + F_{i}^{p} + F_{i}^{rand},
$$

where $F_{i}^{rand}$ is a random force that models any variations in the behavior not included in the other terms, such as accidental deviation from intended motion or failure to assess the current situation correctly. The desired acceleration $\dot{a}_i$ is the acceleration *walker* $i$ would like to apply. But the *walker* has a maximum speed $v_{max}$ above which it cannot accelerate, thus the desired acceleration $\dot{a}_i$ is not always the actual acceleration.

The equations of motion for the *walkers* are

$$
\frac{d\tilde{v}_i}{dt} = \dot{a}_i,
$$

and

$$
\frac{dx_i}{dt} = v_i,
$$

where

$$
v_i = \frac{\tilde{v}_i}{\dot{v}_i} \min \left( \tilde{v}_i, v_i^{max} \right),
$$

so that the actual speed $v_i$ is guaranteed not to exceed the maximum speed $v_i^{max}$ of the *walker*. The maximum speed of the *walkers* is set to a constant factor times the preferred speed of each *walker*. Due to this speed limit, the actual acceleration $d\tilde{v}/dt = a \leq \dot{a}$.

The forces are added to the acceleration as they are calculated. When all forces have been calculated the simulation proceeds to the next simulation step and calculates the new position from the velocity calculated in the previous step, the new $\tilde{v}$ from $\dot{a}$ in the previous step and the new velocity from equation (13). For each *walker*, this calculation is independent of the state of the other *walkers*, so this calculation only require a
single loop over the list of walkers and are thus of negligible computational expense compared to the calculation of the social force.

The equations of motion are solved using the Euler method with a default time step $\Delta t = 0.05$ s. Thus the “Update state” box in figure 2 represents the following pseudo code:

For each walker:

$$x = x + v \Delta t$$

$$\dot{v} = \dot{v} + a \Delta t$$

$$v_i = \frac{\dot{v}_i}{\bar{v}_i} \min (\bar{v}_i, v_i^{max})$$

Calculate/read $v^p$.

If arrived to the destination, or waiting time ended: change activity.

End.
6  The tactical level - Navigation

The tactical level corresponds to planning the route between the origin and a destination, taking into account any obstacles that the pedestrian is aware of, and possibly replanning if the conditions demand. In the current version of the simulation platform the tactical level is static.

The navigation to a given destination is implemented as a shortest path algorithm in the Destination class. Each destination calculates the shortest walkable distance from each point in the simulated area to any point of its own area by use of the Fast Marching Method (FMM) by Sethian 1996. The method and its implementation is described in section 6.1. The result is represented as a scalar field, the gradient of which gives the preferred direction of a walker walking towards its destination.

6.1 The Fast Marching Method

The Fast Marching Method is a numerical scheme to solve equations of the form

\[ |\nabla u(x)| = F(x), \]

where \( u(x) \) is sought for \( x \) in \( \Omega \subset \mathbb{R}^n \). \( F(x) > 0 \) is known in \( \Omega \) and \( u(x) \) is known only on a subset \( \omega \subset \Omega \). The solution of this equation corresponds to evolving a wavefront from \( \omega \) with the speed \( 1/F \). If \( F(x) \equiv 1 \) on the walkable area, \( F(x) \equiv \infty \) on obstacles, and \( u(x) \equiv 0 \) on the destination, the solution \( u(x) \) in the walkable area is the distance to the destination along the shortest walkable path. If the walkable area is not simply connected or the destination area is non-convex, there are points where the gradient is not well defined. In such cases the FMM chooses a solution that everywhere equals the shortest distance to the destination area.

The key idea in the FMM is to use an approximation of the gradient that allows the solution to be calculated "outwards" from the already known values, that is, away from the destination. One such approximation of the gradient is

\[ \frac{\partial u}{\partial x} \approx \max(D_{i}^{-x}u, -D_{i}^{+x}u, 0), \]

where \( D_{i}^{-x} \) and \( D_{i}^{+x} \) are the finite difference operators,

\[ D_{i}^{-x}u = \frac{u_{i} - u_{i-1}}{h}, \]

and

\[ D_{i}^{+x}u = \frac{u_{i+1} - u_{i}}{h}, \]

where the \( x \)-coordinate is discretized according to \( x = ih \). \( u_{i} \) and \( u_{i+1} \) are the values of \( u(x) \) at \( x = (ih, jh) \) and \( x = ((i + 1)h, jh) \), respectively.

With this approximation of the gradient equation (14) becomes

\[ \left( \max(D_{i}^{-x}u, -D_{i}^{+x}u, 0) \right)^2 + \left( \max(D_{j}^{-y}u, -D_{j}^{+y}u, 0) \right)^2 = F^2. \]

If this equation is used to calculate the value of the function \( u \) at the grid point \((i, j)\) from the value the function at the four neighboring grid points, and the largest of the two roots of the equation is chosen, the calculated value will only depend on the grid points with a lower value than the calculated. Furthermore the solution will converge to the correct one, even at points where \( u(x) \) is non-differentiable, for proof see Sethian.
1996. Thus we can start from the destination where the solution is known to be vanishing and construct the solution outwards, with new, higher values, only depending on the previous, lower values.

Pseudo code: Initiation:

1. Set the distance at points in the destination area to zero and tag them as Frozen.
2. Tag all non-Frozen points in the walkable area Far, and all points in obstacles as Obstacles.
3. Tag points neighboring Frozen points as Close and set the distance to each Close point to the largest solution of equation (18).
4. Create a list with the Close points always kept sorted according to their distance with the smallest first.

Loop:

1. Remove the first entry (smallest) in the list of Close points and tag it as Frozen.
2. Set the distance in its non-Frozen, non-Obstacle neighbors to the largest solution of equation (18) and tag them as Close and add them to the list if they were Far.
3. Goto 1 if the list of Close points is non-empty.

The list of Close points is implemented as a QMultiMap< float, QPoint >, since QMultiMap is automatically kept sorted in increasing order according to the key.

In the method described above, no consideration is taken to the obstacle avoiding model of the social force model. This model states that pedestrians tend to avoid walking close to obstacles and models this avoidance as a force directed away from obstacles. But the shortest path model described above results in preferred directions for the walkers that aims them very close to obstacles. This results in a contradictory desire for the walkers: according to the tactical level they want to walk close to obstacles to take the shortest possible path to their destinations, but when they come close to an obstacle the operational level forces them away from the obstacle. This conflict between the tactical and operational level must be resolved to avoid unrealistic behavior. In PTSP this is achieved by expanding all obstacles by 4 dm in all directions before applying the FMM.
7 The strategical level - Activity Planning

The strategical level corresponds to activity planning, that is, choice of origin, destination, and potentially intermediate destinations, and is considered input to the simulation platform. This means that to specify a scenario to simulate, the user must specify where and when walkers should appear and where they wish to go.

The origins and destinations are implemented as two classes, Origins and Destinations. The objects of these classes are collected into two lists owned by the simulator.

7.1 Paths

A *walkPath* is an ordered list of *destinations*. Note that the *origins* are not included in the *walkPaths*. The *simWindow* owns a list of all paths in the simulation, created as a part of the scenario specification. Each *walker* is given a copy of a *walkPath* by the *origin* that created the *walker*. The *walkers* use the *WalkPath* methods *next()* and *dest()* to get the next and current *destination*. When a *walker* reaches the last *destination* in its *walkPath* it is destroyed.

7.2 Origins

The *Origin* class main purpose is to produce *Walkers*. Any place where a pedestrian can enter the simulated area is represented in the model by an *origin*.

The frequency of which an *origin* creates *walkers* is determined in terms of a distribution of the time gap between two creations. In each time step of the main simulation loop the *simulator* iterates through its list of *Origins* and calls the *Origin* method *createWalker()* . This checks if the time elapsed from the last creation is larger than the last realization of the *distribution* representing the time gap. If so it creates a new walker and draws a new time gap. The distribution of time gaps are specified at the creation of the *origin*.

From a model point of view there are no restrictions on the shape of an *origin*, but so far the implementation constrains them to be rectangles centered at a given point and with a given width and height. The *origin* creates the *walkers* in the area covered, according to two independent *distributions*, one for the y-coordinate and one for the x-coordinate. The defaults are uniform distributions over the covered intervals.

An *origin* have a list of all the paths that *walkers* created by it can take, paired with the probability that a *walker* will take the path. When a *walker* is created it is assigned a path randomly according to the probabilities. The list of paths and probabilities is implemented as an instance of QHash<<*WalkPath*, qreal>, and is given to the *Origin* constructor, and specified in the scenario description.

7.3 Destinations

The *Destination* class represents both the exits from the simulated area and intermediate goals that pedestrians may have, such as waiting areas or ticket machines, but can also be used as way points to impose a route choice that differs from the route choice model. When created a *walker* is given a path specifying an ordered list of *destinations* that it should visit.

In principle, there are no constraints on the shape of a *destination*, it can even be
disconnected and still be compatible with the FMM, but it is implemented as a rectangle with edges parallel to the coordinate axes. When created, a destination is specified as a rectangle and a focus point, see section 7.4. It is also passed a field representing a map of all the obstacles in the simulated area, by at each point specifying the distance to the closest obstacle, that is, points where the field vanishes are obstacles. This field is used in the navigation model, described in section 6, which returns a vector field describing the preferred direction at each point.

The method of calculating a navigation field for each destination implies that each destination is independent of the location of any other destination, and also of the location of any origin. The destinations are only dependent on the map of the simulated area with all obstacles and can thus be constructed as soon as the map is available, before any origins are specified.

7.4 Waiting areas

A waiting area is a destination with a waiting time distribution, a focus point, and a waiting position distribution. The focus point represents an information sign or similar, towards which the waiting pedestrians have their attention. As a walker enters the waiting area, its state is changed from walking to waiting and a waiting time is drawn from the waiting time distribution of the destination, which can be set using the setWaitDist() method. When a walker is waiting it behaves according to a slightly modified model, called waiting model. The current version of the PTSP have two waiting models implemented. They are described in the following sections.

7.4.1 Preferred position waiting model

When the preferred position waiting model is used, the walker draws a number from the waiting position distance distribution. The walker then choose its preferred position as the point on the line between its current position and the focus point, at a distance equal to the drawn waiting position distance from the focus point.

When waiting the preferred direction of the walker is no longer determined by the navigation field, but instead always directed towards the preferred position. If the walker is within a certain distance \(d\) from its preferred position, the preferred speed is set to be proportional to the distance to the preferred position, in order to avoid oscillatory behavior. Outside this circle the preferred speed is equal to the walkers regular preferred speed. The distance \(d\) is chosen such that the motion of the walker, when approaching its preferred position unperturbed from afar, is over critically damped.

The preferred direction and the looking direction of the walker are normally the same, but is separated during waiting. The looking direction \(e_i\) is now set to the direction towards the focus point \(x^f\) of the destination.

The preferred position waiting model can thus be summarized by equation (19) - equation (21).

\[
v_{pw_i} = \begin{cases} 
  v^p (x_{pw_i} - x_i) / |x_{pw_i} - x_i|, & |x_{pw_i} - x_i| > d_i, \\
  v^p (x_{pw_i} - x_i) / d, & |x_{pw_i} - x_i| \leq d_i,
\end{cases}
\]

(19)

where, \(x_{pw_i}\) is the preferred waiting position, \(v_i^p\) is the ordinary preferred speed, \(x_i\) the current position, and where \(d_i\) is chosen as,

\[
d_i = 4\tau v_i^p.
\]

(20)
Finally, the looking direction is

$$e_i = x^f - x_i.$$  \hfill (21)

### 7.4.2 Preferred velocity waiting model

When the preferred velocity waiting model is used, a waiting \textit{walker} is not assigned a preferred position, but instead the preferred velocity is set to zero. As in the preferred position model the looking direction is set toward the focus point. Thus the equations of the preferred velocity waiting model are simply

$$v^p_i = 0,$$  \hfill (22)

and

$$e_i = x^f - x_i.$$  \hfill (23)

### 7.4.3 Alternative versions

The two models described above share the feature that they only alter the preferred velocity of the \textit{walker}. This means that they do not interfere with the core of the social force model, since the social force model has the preferred velocity as an external variable, given by the tactical level. In this way the waiting models discussed above are contained within the tactical level of the model hierarchy. Thus, given a calibrated social force model, the waiting models above may be calibrated without interfering with the parameters of the social force model.

However the waiting models discussed so far have some drawbacks. The waiting \textit{walkers} react too strongly on the presence of other, walking, \textit{walkers}. In both the above models, the waiting \textit{walkers} are very polite and move out of the way when another \textit{walker} approaches. This problem can be solved by reducing the force exerted by a walking \textit{walker} on a waiting \textit{walker} to $F = (1 - \beta)F$. For this reduction of the force not to result in too close encounters, the force exerted by a waiting \textit{walker} on a walking \textit{walker} is increased by the corresponding factor. The parameter $\alpha$ needs to be estimated from observations and will hopefully be the only additional parameter in need of calibration when including waiting models, but in the worst case all the parameters of the social force model need to be calibrated separately for waiting \textit{walkers}. 
References


Qt-Project (Apr. 2013). Qt. URL: http://qt-project.org/.


## Parameters

<table>
<thead>
<tr>
<th>Object</th>
<th>Parameter name</th>
<th>Sym.</th>
<th>Distr.</th>
<th>Value</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Walker</strong></td>
<td>Preferred speed</td>
<td>$v_p^0$</td>
<td>N(1.37,0.3)</td>
<td>(0.5,2.25) ms$^{-1}$</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>Social force strength</td>
<td>$F_0^s$</td>
<td>const.</td>
<td>0.25 ms$^{-2}$</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>Walker radius</td>
<td>$R$</td>
<td>const.</td>
<td>0.15 m</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Max speed factor</td>
<td>$\lambda$</td>
<td>const.</td>
<td>1.2</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>Force anisotropy</td>
<td>$T_a$</td>
<td>const.</td>
<td>1.27 s</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>Look-ahead time</td>
<td>$\tau$</td>
<td>const.</td>
<td>0.6 s</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>Social force scale</td>
<td>$\sigma$</td>
<td>const.</td>
<td>0.59 m</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>Pref. force time scale</td>
<td>$\tau$</td>
<td>const.</td>
<td>0.6 s</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>Random force</td>
<td>$F_{rand}$</td>
<td>N(0,0.1)</td>
<td>(-1,1) ms$^{-2}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Phys. force strength</td>
<td>$F_{phys}^0$</td>
<td>const.</td>
<td>3 ms$^{-2}$</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>Phys. force scale</td>
<td>$\sigma_{phys}$</td>
<td>const.</td>
<td>0.2 m</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Waiting force factor</td>
<td>$\beta$</td>
<td>const.</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>Simulator</strong></td>
<td>Time step</td>
<td>dt</td>
<td>const.</td>
<td>0.05 s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cell size</td>
<td></td>
<td>const.</td>
<td>0.1 m</td>
<td></td>
</tr>
<tr>
<td><strong>Obstacle</strong></td>
<td>Obst. force strength</td>
<td>$F_{obst}^0$</td>
<td>const.</td>
<td>10 ms$^{-2}$</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>Phys. force scale</td>
<td>$\sigma_{obst}$</td>
<td>const.</td>
<td>0.2 m</td>
<td></td>
</tr>
<tr>
<td><strong>Origin</strong></td>
<td>Time gap distr</td>
<td>N(2.3,0.2)</td>
<td></td>
<td>(1,10) s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pos distr</td>
<td>U(within)</td>
<td></td>
<td>(within)</td>
<td></td>
</tr>
<tr>
<td><strong>Destination</strong></td>
<td>Wait time distr</td>
<td>N(30,5)</td>
<td></td>
<td>(2,30) s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wait dist distr</td>
<td>N(3,0.5)</td>
<td></td>
<td>(0.5,5) m</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Obst ext. dist</td>
<td>const.</td>
<td></td>
<td>0.4 m</td>
<td></td>
</tr>
</tbody>
</table>

*Table 1  Table of parameters (in a broad sense) in the model, and their default values. The references are to the source of the default values, a denotes A. Johansson 2009, and b, TRB 2010. The default distribution of preferred speeds are approximately adopted from a distribution of speeds of pedestrians crossing a signalized pedestrian crossing. A parenthesis in the value column indicates the interval of possible values.*

VTI notat 16-2013
Appendix B

Contents

1 Class Index .................................................. 1
  1.1 Class List .................................................. 1

2 Namespace Documentation .................................... 2
  2.1 myMath Namespace Reference ............................. 2
    2.1.1 Detailed Description .................................. 2

3 Class Documentation .......................................... 2
  3.1 DataFile Class Reference ................................. 2
    3.1.1 Detailed Description .................................. 3
    3.1.2 Member Data Documentation ............................ 3
  3.2 Destination Class Reference .............................. 3
    3.2.1 Detailed Description .................................. 4
    3.2.2 Constructor & Destructor Documentation ............... 4
  3.3 myMath::Distribution Class Reference .................. 5
    3.3.1 Detailed Description .................................. 5
    3.3.2 Constructor & Destructor Documentation ............... 5
  3.4 ModelParameters Struct Reference ...................... 5
    3.4.1 Detailed Description .................................. 5
  3.5 ModelSettings Class Reference .......................... 5
    3.5.1 Detailed Description .................................. 6
  3.6 Obstacle Class Reference ................................ 6
    3.6.1 Detailed Description .................................. 6
    3.6.2 Constructor & Destructor Documentation ............... 7
  3.7 Origin Class Reference ................................... 7
    3.7.1 Detailed Description .................................. 7
    3.7.2 Constructor & Destructor Documentation ............... 7
  3.8 Settings Class Reference .................................. 8
    3.8.1 Detailed Description .................................. 8
  3.9 SimDisplay Class Reference .............................. 8
    3.9.1 Detailed Description .................................. 8
  3.10 SimSettings Class Reference ........................... 8
    3.10.1 Detailed Description .................................. 9
  3.11 Simulator Class Reference .............................. 9
    3.11.1 Detailed Description .................................. 12
    3.11.2 Constructor & Destructor Documentation ............... 12
    3.11.3 Member Function Documentation ....................... 12
    3.11.4 Member Data Documentation ............................ 13
  3.12 SimWindow Class Reference ............................. 14
Appendix B

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.12.1</td>
<td>Detailed Description</td>
<td>15</td>
</tr>
<tr>
<td>3.12.2</td>
<td>Constructor &amp; Destructor Documentation</td>
<td>15</td>
</tr>
<tr>
<td>3.12.3</td>
<td>Member Function Documentation</td>
<td>15</td>
</tr>
<tr>
<td>3.13</td>
<td>ToolMap Class Reference</td>
<td>15</td>
</tr>
<tr>
<td>3.13.1</td>
<td>Detailed Description</td>
<td>15</td>
</tr>
<tr>
<td>3.14</td>
<td>ToolSim Class Reference</td>
<td>16</td>
</tr>
<tr>
<td>3.14.1</td>
<td>Detailed Description</td>
<td>16</td>
</tr>
<tr>
<td>3.15</td>
<td>ToolWalker Class Reference</td>
<td>16</td>
</tr>
<tr>
<td>3.15.1</td>
<td>Detailed Description</td>
<td>16</td>
</tr>
<tr>
<td>3.16</td>
<td>Walker Class Reference</td>
<td>16</td>
</tr>
<tr>
<td>3.16.1</td>
<td>Detailed Description</td>
<td>19</td>
</tr>
<tr>
<td>3.16.2</td>
<td>Constructor &amp; Destructor Documentation</td>
<td>20</td>
</tr>
<tr>
<td>3.16.3</td>
<td>Member Function Documentation</td>
<td>20</td>
</tr>
<tr>
<td>3.16.4</td>
<td>Member Data Documentation</td>
<td>21</td>
</tr>
<tr>
<td>3.17</td>
<td>walkerDot Class Reference</td>
<td>21</td>
</tr>
<tr>
<td>3.17.1</td>
<td>Detailed Description</td>
<td>21</td>
</tr>
<tr>
<td>3.18</td>
<td>WalkerOriginDot Class Reference</td>
<td>21</td>
</tr>
<tr>
<td>3.18.1</td>
<td>Detailed Description</td>
<td>22</td>
</tr>
<tr>
<td>3.19</td>
<td>WalkerParameters Struct Reference</td>
<td>22</td>
</tr>
<tr>
<td>3.19.1</td>
<td>Detailed Description</td>
<td>22</td>
</tr>
<tr>
<td>3.20</td>
<td>WalkerSettings Class Reference</td>
<td>22</td>
</tr>
<tr>
<td>3.20.1</td>
<td>Detailed Description</td>
<td>22</td>
</tr>
<tr>
<td>3.21</td>
<td>WalkPath Class Reference</td>
<td>23</td>
</tr>
<tr>
<td>3.21.1</td>
<td>Detailed Description</td>
<td>23</td>
</tr>
<tr>
<td>3.21.2</td>
<td>Member Function Documentation</td>
<td>23</td>
</tr>
<tr>
<td>3.22</td>
<td>XmlFile Class Reference</td>
<td>23</td>
</tr>
<tr>
<td>3.22.1</td>
<td>Detailed Description</td>
<td>23</td>
</tr>
</tbody>
</table>
1 Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

- **DataFile**
  - Class to handle data output

- **Destination**
  - Represents destinations of walkers

- **myMath::Distribution**
  - Class for handling statistical distributions

- **ModelParameters**
  - Struct with the parameters of the SFM

- **ModelSettings**
  - **Settings** subclass to read SFM parameters from XML file

- **Obstacle**
  - Represents a wall or any other fixed obstacle

- **Origin**
  - Class to represent walker origins

- **Settings**
  - Class for handling different kinds of settings

- **SimDisplay**
  - GUI. The area visualizing the simulation

- **SimSettings**
  - Not yet implemented

- **Simulator**
  - The main **Simulator**

- **SimWindow**
  - The main window of the application

- **ToolMap**
  - GUI class

- **ToolSim**
  - GUI class

- **ToolWalker**
  - GUI class

- **Walker**
  - Objects of the **Walker** class represents pedestrians

- **walkerDot**
  - GUI. The visual representation of a walker

- **WalkerOriginDot**
  - GUI. Graphical representation of an origin
WalkerParameters
Parameters describing the walker

WalkerSettings

WalkPath
An ordered list of destinations

XmlFile
Class to handle Xml input/output

2 Namespace Documentation

2.1 myMath Namespace Reference
Contains pure general purpose math tools.

Classes

• class Distribution
  Class for handling statistical distributions.

Enumerations

• enum distShape
  Shape of distribution, Normal (Gaussian) or Flat (Uniform).

2.1.1 Detailed Description
Contains pure general purpose math tools. So far just the Distribution class

3 Class Documentation

3.1 DataFile Class Reference
Class to handle data output.

Public Member Functions

• DataFile (const QString filename)
  Constructor if the data to be stored is qreals.

• DataFile (const QString filenameX, const QString filenameY)
  Constructor if the data to be stored is QVector2Ds.

Static Private Attributes

• static const int mZeros = 3000
3.1.1 Detailed Description

Class to handle data output.

Data is written to disc using the `<<` operator. The class only supports a few data types. The data is formatted to easy load into MATLAB, a row of zeros is added to the beginning of the file to simplify this.

3.1.2 Member Data Documentation

3.1.2.1 const int DataFile::mZeros = 3000 [static, private]

Number of zeros in the leading row.

3.2 Destination Class Reference

Represents destinations of walkers.

Collaboration diagram for Destination:

```
myMath::Distribution
  | mWaitPosDist
  | mWaitTimeDist
  | Destination
```

Public Member Functions

- **Destination** (QRectF dest, QVector<QVector<qreal>> &distToWall, qreal pixw, QVector2D focus)
  Constructs a destination at dest.
- void **calcNavField** (const QVector<QVector<qreal>> &distToWall)
  Calculates the navigation field.
- qreal **distAt** (int x, int y)
  Returns the shortest walkable distance to the destination from (x,y).
- QVector2D **dirAt** (QVector2D pos)
  Returns the preferred direction at pos.
- bool **arrived** (QVector2D pos)
  Returns True if pos is within the destination area.
- int **getDestNumber** ()
  Returns the identification number of the destination.
- QVector2D **focus** ()
  Returns the point that walkers look at when waiting.
- qreal **waitTime** ()
  Generates and returns a waiting time from the waiting time distribution.
- qreal **drawWaitPos** ()
Generates and returns a distance to the focus, at which to wait.

- void setWaitDist ( myMath::Distribution ∗ dist)
  Sets the waiting time distribution.
- void setWaitPosDist ( myMath::Distribution ∗ dist)
  Sets the waiting distance distribution.
- QVector< QVector< QVector2D > > navField ()
  Returns the navigation field.

Private Types

- enum State
  Labels for the FMM.

Private Attributes

- QRectF mDestRect
  The destination area.
- qreal mPixw
  The cell size of the navigation field.
- int mDestNumber
  The identification number of the destination.
- QVector< QPoint > neighbors
  Definition of neighboring cells in the FMM.
- QVector2D mFocus
  The point at which waiting walkers look. Represents e.g. an information sign.
- QVector< QVector< QVector2D > > mDistToDestField
  The shortest walkable distances to the destination from every point.
- QVector< QVector< QVector2D > > mNavField
  The navigation field (preferred direction at every point).

Static Private Attributes

- static int destNumber = 0
  The number of destinations.

3.2.1 Detailed Description

Represents destinations of walkers.

An object of this class represents a destination, which can be a final destination or a waiting area of walkers. A Destination is defined by a rectangle within the walkable area, and a focus, if it may be a waiting area. The destination also calculates the navigation field to itself using the FMM.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 Destination::Destination ( QRectF dest, QVector< QVector< qreal > > & distToWall, qreal pixw, QVector2D focus )

Constructs a destination at dest.
The destination calculates the navigation field to itself given the location of obstacles in distToWall.
Appendix B page 5(25)

3.3 myMath::Distribution Class Reference

Class for handling statistical distributions.

Public Member Functions

- **Distribution (distShape shape, qreal mean, qreal min=0, qreal max=0, qreal stdDev=0)**
  Constructs a distribution as specified by the arguments.
- **qreal draw ()**
  Returns a random number from the distribution.

Private Attributes

- **qreal mMean**
  Mean of the non truncated distribution.
- **qreal mStdDev**
  Standard deviation of the non truncated distribution.

3.3.1 Detailed Description

Class for handling statistical distributions.
So far just Normal (Gaussian) and flat, both can be truncated.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 myMath::Distribution::Distribution ( distShape shape, qreal mean, qreal min = 0, qreal max = 0, qreal stdDev = 0 )

Constructs a distribution as specified by the arguments.
The shape can either be Flat or Normal. A sample from the distribution is produced by calling draw.

3.4 ModelParameters Struct Reference

Struct with the parameters of the SFM.

3.4.1 Detailed Description

Struct with the parameters of the SFM.

3.5 ModelSettings Class Reference

Settings subclass to read SFM parameters from XML file.
3.5.1 Detailed Description

*Settings* subclass to read SFM parameters from XML file.

3.6 Obstacle Class Reference

Represents a wall or any other fixed obstacle.

Public Member Functions

- **Obstacle** (QVector2D point_a, QVector2D point_b, qreal pixwidth=0.1)
  
  *Constructs an* Obstacle *between the points point_a and point_b.*

- **void addToField** (QVector<QVector2D> &field, QVector<QVector<QVector2D>> &ffield)
  
  *Adds the obstacle to the fields field and ffield.*

3.6.1 Detailed Description

Represents a wall or any other fixed obstacle.

An obstacle consist of a line segment between two points. More complex Obstacles are represented as several obstacles. The class handles calculating the distance from the obstacle to nearby points and modify the field.
representing all obstacles. The obstacle itself is not used during the simulation, only the total field with the distance to the closest obstacle at each point, and the field with the social forces from all obstacles.

### 3.6.2 Constructor & Destructor Documentation

#### 3.6.2.1 Obstacle::Obstacle (QVector2D point_a, QVector2D point_b, qreal pixwidth = 0.1)

Constructs an Obstacle between the points point_a and point_b.

**Parameters**

<table>
<thead>
<tr>
<th>point_a, point_b</th>
<th>Endpoints of the Obstacle.</th>
</tr>
</thead>
<tbody>
<tr>
<td>pixwidth</td>
<td>The lattice constant of the navigation field.</td>
</tr>
</tbody>
</table>

### 3.7 Origin Class Reference

Class to represent walker origins.

#### Public Member Functions

- **Origin (QVector2D pos, qreal width, qreal height, qreal firstGap, QMap<WalkPath*, float> paths, int N, QColor color)**

  Creates an Origin at pos.

  - void createWalker (Simulator *sim)
    
    Creates a walker if its time to do so.

  - void reset ()
    
    Resets the number of created walkers etc.

- **QVector2D pos ()**

  Returns the position of the origin.

- **void setRate (qreal rate)**

  Sets the mean of the time gap distribution.

- **void setGapDist (Distribution *dist)**

  Sets the time gap distribution.

#### 3.7.1 Detailed Description

Class to represent walker origins.

An origin is an area where walkers are created, that is areas where pedestrians enter the simulated area.

#### 3.7.2 Constructor & Destructor Documentation

#### 3.7.2.1 Origin::Origin (QVector2D pos, qreal width, qreal height, qreal firstGap, QMap<WalkPath*, float> paths, int N, QColor color)

Creates an Origin at pos.

Walkers are created in a rectangular area centered around pos according to a flat probability distribution.

**Parameters**

<table>
<thead>
<tr>
<th>firstGap</th>
<th>The time at which the origin starts producing walkers.</th>
</tr>
</thead>
<tbody>
<tr>
<td>paths</td>
<td>List of paths and relative intensities for the created walkers.</td>
</tr>
<tr>
<td>N</td>
<td>Number of walkers to create.</td>
</tr>
<tr>
<td>color</td>
<td>Color of the created walkers.</td>
</tr>
</tbody>
</table>
3.8 Settings Class Reference

Class for handling different kinds of settings.

Inheritance diagram for Settings:

```
Settings
\_\_\_\_
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ModelSettings</td>
</tr>
<tr>
<td>SimSettings</td>
</tr>
<tr>
<td>WalkerSettings</td>
</tr>
</tbody>
</table>
```

3.8.1 Detailed Description

Class for handling different kinds of settings.

Base class for SimSettings, ModelSettings, WalkerSettings, etc. Reads and write XML files

3.9 SimDisplay Class Reference

GUI. The area visualizing the simulation.

Public Member Functions

- void addWalkerDot (Walker ∗w, qreal pixw, QColor color)
  Constructs a visual representation of a walker.
- void addWall (QVector2D posA, QVector2D posB)
  Constructs a visual representation of an obstacle.
- void addWalkerSourceDot (QVector2D pos, qreal pixw)
  Constructs a visual representation of an origin.

3.9.1 Detailed Description

GUI. The area visualizing the simulation.

Inherits QWidget, is a Q_OBJECT

3.10 SimSettings Class Reference

Not yet implemented.
Inheritance diagram for SimSettings:

```
Settings
   ^
SimSettings
```

Collaboration diagram for SimSettings:

```
Settings
   ^
SimSettings
```

3.10.1 Detailed Description

Not yet implemented.

3.11 Simulator Class Reference

The main Simulator.
Public Member Functions

- **Simulator (SimWindow *simwin)**
  
  *The constructor is given a pointer to the SimWindow.*

- **void run (qreal runTime)**
  
  *Runs the simulator runTime sec.*

- **void rawRun (qreal runTime)**
  
  *Runs the simulation runTime sec without animation.*

- **void addWalker (QVector2D pos, WalkPath *path, QColor color)**
  
  *Creates a walker.*

- **void addWall (QVector2D a, QVector2D b)**
  
  *Constructs an Obstacle between the points a and b.*

- **void connectDisp (SimDisplay *disp)**
  
  *Connects the SimDisplay to the Simulator.*

- **void updDisp ()**
  
  *Updates the display.*

- **void updTime (qreal timestep=0)**
  
  *Updates the Simulators clock.*

- **void printField ()**
  
  *Stores the navigation fields to file.*

- **void addOrigin (QVector2D pos, qreal width, qreal height, qreal firstGap, QMap< WalkPath *, float > paths, int N, QCColor color)**
  
  *Constructs an Origin.*

- **void addDestination (QRectF dest, QVector2D focus)**
  
  *Constructs a Destination.*

- **Destination *getDestination (int i)**
  
  *Returns Destination i in mDestList.*

- **void calcNavFields ()**
  
  *Calculates the navigation fields.*

- **QVector< Origin > originList ()**
  
  *Returns a list of all Origins.*

- **QVector< Destination > destList ()**
  
  *Returns a list of all Destinations.*

- **QVector< QVector< qreal > > distField ()**
  
  *Returns mDistField.*
• QVector<QVector2D> wallForceField ()
  Returns the field of obstacle forces.
• QVector<WalkPath> paths ()
  Returns the list of WalkPaths.
• void setOriginList(QVector<Origin> originList)
  Sets the list of origins.
• void setDestList(QVector<Destination> destList)
  Sets the list of destinations.
• void setDistField(QVector<QVector<qreal>> distField)
  Sets the field of distances to obstacles.
• void setWallForceField(QVector<QVector<QVector2D>> wallForceField)
  Sets the field of obstacle forces.
• void setPathList(QVector<WalkPath> paths)
  Sets the list of WalkPaths.
• void resetSources ()
  Sets the number of produced walkers at all origins to zero.
• void collectData (bool coldata)
  Toggles if data is collected or not.
• void addOrigin (Origin s)
  Adds the Origin s to the simulation.
• void addDestination (QRectF dest, QVector2D focus, Distribution ∗dist)
  Creates and adds a Destination to the simulation.
• void addDestination (QRectF dest, QVector2D focus, Distribution ∗dist, Distribution ∗posdist)
  Creates and adds a Destination to the simulation.
• void setBeta (qreal beta)
  Sets the parameter beta.

Private Member Functions

• void updForces ()
  Updates every walkers force.
• void updState ()
  Updates the position of each walker.

Private Attributes

• QVector<Walker> mList
  The list of all Walkers.
• QVector<Origin> mOriginList
  The list of all Origins.
• QVector<Destination> mDestList
  The list of all Destinations.
• QVector<WalkPath> mPathList
  The list of all WalkPaths.
• ModelParameters mModParams
  The model parameters.
• QString mSettingsPath
  The path of the xml file containing the settings.
• qreal dt
  The time step of the simulation.
Appendix B page 12(25)

- **qreal** pixw
  
  Linear size [m] of a cell in the fields, default 0.1 m.

- **qreal** cutoff_dist2
  
  Cutoff distance squared of the SFM.

- **DataFile** mCoordsDataFile
  
  Output data file for the position.

- **DataFile** mForcesDataFile
  
  Output data file for the force.

- **DataFile** mDiscomfortDataFile
  
  Output data file for the discomfort.

- **DataFile** mVelocitiesDataFile
  
  Output data file for the velocity.

- **DataFile** mOtherdataDataFile
  
  Output data file for various additional data.

- **DataFile** mDelayDataFile
  
  Output data file for the delay.

### 3.11.1 Detailed Description

The main **Simulator**.

The simulator object owns all walkers and structures. A fixed time simulation is performed using **run()**.

### 3.11.2 Constructor & Destructor Documentation

#### 3.11.2.1 Simulator::Simulator

The constructor is given a pointer to the **SimWindow**.

The constructor reads model parameters and settings from XML files in a fixed path directory using classes inheriting the **Settings** class.

### 3.11.3 Member Function Documentation

#### 3.11.3.1 void Simulator::addDestination

Constructs a **Destination**.

**Parameters**

<table>
<thead>
<tr>
<th><strong>dest</strong></th>
<th>Specifies the destination area</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>focus</strong></td>
<td>Specifies the point the walkers look at.</td>
</tr>
</tbody>
</table>

#### 3.11.3.2 void Simulator::addOrigin

Constructs an **Origin**.

The constructed **Origin** is added to **mOriginList**.

**Parameters**

<table>
<thead>
<tr>
<th><strong>pos</strong></th>
<th>Center position of the origin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>paths</strong></td>
<td>is a QMap of <strong>WalkPath</strong>, float pairs specifying the proportion of walkers created by this origin that should take the paths.</td>
</tr>
<tr>
<td><strong>N</strong></td>
<td>Limit on the number of Walkers created by the <strong>Origin</strong></td>
</tr>
<tr>
<td><strong>color</strong></td>
<td>the color of the <strong>Walker</strong> created by the <strong>Origin</strong>.</td>
</tr>
</tbody>
</table>
3.11.3.3 \texttt{void \ Simulator::addWalker ( QVector2D pos, WalkPath \* path, \ QColor color )}

Creates a walker.  
Creates a walker and if disp\_connected also creates its graphical representation.

3.11.3.4 \texttt{void Simulator::addWall ( QVector2D a, \ QVector2D b )}

Constructs an Obstacle between the points a and b.

**Parameters**

\begin{itemize}
\item \texttt{a,b} \quad \texttt{QVector2D} giving the endpoints of the Obstacle
\end{itemize}

3.11.3.5 \texttt{void \ Simulator::calcNavFields ( )}

Calculates the navigation fields.  
Used in SimWindow when all Destinations and all Obstacles have been created.

3.11.3.6 \texttt{QVector< QVector< qreal >> Simulator::distField ( )}

Returns mDistField.  
the field of distances to the closest Obstacle.

3.11.3.7 \texttt{Destination \* \ Simulator::getDestination \ ( int i )}

Returns Destination i in mDestList.  
Used in createPath to construct WalkPath

3.11.3.8 \texttt{void Simulator::printField ( )}

Stores the navigation fields to file.  
Iterates through mDestList and stores all navigation fields to disk using DataFile

3.11.3.9 \texttt{void Simulator::run ( \ qreal runTime )}

Runs the simulator runTime sec.  
Runs the main simulation loop for runTime sec and then updates the display, if disp\_connected.

3.11.3.10 \texttt{void \ Simulator::updDisp ( )}

Updates the display.  
Iterates through all walkers and updates the position of each dot. Used in run() if disp\_connected.

3.11.3.11 \texttt{void \ Simulator::updForces ( ) \ [private]}

Updates every walkers force.  
Iterates through all walkers and clears the force, calculates prefered velocity, and the corresponding force. Then makes a double loop to calculate the w-w interaction force. Used in run()

3.11.3.12 \texttt{void Simulator::updState ( ) \ [private]}

Updates the position of each walker.  
Iterates through all walkers and updates the position, w and velocity of each. Used in run()

3.11.4 \ Member Data Documentation
3.11.4.1 **ModelParameters** Simulator::mModParams [private]

The model parameters.
Struct of lambda, $T_s$, $\sigma$, $F_0$, $\tau$

### 3.12 SimWindow Class Reference

The main window of the application.

Collaboration diagram for SimWindow:

![Collaboration diagram for SimWindow](image)

#### Public Slots

- void **start** ()

  *Starts a set of simulations with animation.*

- void **rawLoop** ()

  *Starts a set of simulations without animation.*

#### Public Member Functions

- **SimWindow** (QWidget *parent=0)

  *The constructor initiates a lot of stuff.*

- QTime **getTime** ()

  *Returns the elapsed time in the simulation.*

- void **createPath** (int dest1, int dest2=-1, int dest3=-1, int dest4=-1)

  *Creates a path from the destinations in the destination list.*

#### Private Member Functions

- void **rawRun** ()

  *Runs a simulation without animation.*

- void **resetSimulator** ()

  *Resets the Simulator to its initial state.*

#### Private Attributes

- int **mNumberRuns**

  *Number of consecutive simulation runs to run.*

- int **mSimDuration**

VTI notat 16-2013

Generated on Thu Apr 11 2013 11:32:18 for PTSP by Doxygen
The duration of a simulation in seconds.

- Simulator * mSim
  The Simulator object.
- qreal dt
  The timestep of the Simulator.
- QVector< WalkPath > mPathList
  The list of WalkPaths.

3.12.1 Detailed Description

The main window of the application.
Responsible for the GUI.
The scenarios are described in methods of this class which are called from the constructor. This should be done in a different way in the future.

3.12.2 Constructor & Destructor Documentation

3.12.2.1 SimWindow::SimWindow ( QWidget * parent = 0 )

The constructor initiates a lot of stuff.
For example the various components of the GUI, the Simulator, the scenarios, etc.

3.12.3 Member Function Documentation

3.12.3.1 void SimWindow::createPath ( int dest1, int dest2 = -1, int dest3 = -1, int dest4 = -1 )

Creates a path from the destinations in the destination list.

Parameters

| dest1, ..., dest4 | ints that refers to the placement of the Destinations in the DestList. The order in the parameter list is the order in the WalkPath. |

3.12.3.2 void SimWindow::rawLoop ( ) [slot]

Starts a set of simulations without animation.
See also

Simulator::rawRun

3.12.3.3 void SimWindow::rawRun ( ) [private]

Runs a simulation without animation.
Resets the simulator to its initial state if mIsFirstRun is False.

3.13 ToolMap Class Reference

GUI class.

3.13.1 Detailed Description

GUI class.

VTI notat 16-2013

Generated on Thu Apr 11 2013 11:32:18 for PTSP by Doxygen
3.14 ToolSim Class Reference

GUI class.

3.14.1 Detailed Description

GUI class.

3.15 ToolWalker Class Reference

GUI class.

3.15.1 Detailed Description

GUI class.

3.16 Walker Class Reference

Objects of the Walker class represents pedestrians.

Collaboration diagram for Walker:

```
myMath::Distribution

mRandForceDist
mWaitPosDist
mWaitTimeDist

mDot
mPath
mDest

Walker

walkerDot

Destination

WalkPath

Public Member Functions

- Walker (QVector2D pos, WalkPath *path, QTime time, bool colDta, ModelParameters modParams, WalkerParameters walkParams, qreal beta)
```
The constructor.

- `QVector2D vel()` 
  Returns the current velocity of the walker.

- `QVector2D pos()` 
  Returns the position of the walker.

- `QVector2D prefVel()` 
  Returns the preferred velocity of the walker.

- `bool arrived()` 
  Returns True if the walker have arrived at its current destination.

- `bool isOnFinalLink()` 
  Returns true if the next destination is the last in the path.

- `void wait()` 
  The walker starts waiting.

- `void walk()` 
  The walker starts walking.

- `void reactTo (Walker &w, QVector2D relpos, qreal dx)` 
  Adds the force from walker w.

- `void addRandomForce()` 
  Adds the random force to the total force.

- `void addPrefForce()` 
  Calculates and adds the force connected to the preferred velocity.

- `void calcPrefVel()` 
  Calculates the preferred velocity.

- `void calcVel()` 
  Calculates the preferred velocity.

- `void calcW (qreal dt)` 
  Calculates the desired velocity.

- `void move (qreal dt)` 
  Updates the position of the Walker.

- `void storeCoordinates (bool)` 
  Toggles if the data should be collected or not.

- `void printVelocities (QTextStream *stream)` 
  Writes data to file.

- `void printForces (QTextStream *stream)` 
  Writes data to file.

- `void printUncomfort (QTextStream *stream)` 
  Writes data to file.

- `void printCoords (QTextStream *stream)` 
  Writes data to file.

- `QVector2D fovFilter (QVector2D force, QVector2D relpos)` 
  The function representing the field of view of the Walker.

- `QVector<QVector2D> *coordVec()` 
  Returns the data buffer, written to file by Simulator before destruction.

- `QVector<QVector2D> *forceVec()` 
  Returns the data buffer, written to file by Simulator before destruction.

- `QVector<qreal> *dicomfortVec()` 
  Returns the data buffer, written to file by Simulator before destruction.

- `QVector<QVector2D> *velocityVec()` 
  Returns the data buffer, written to file by Simulator before destruction.

- `QHash<QString, qreal> *otherData()` 
  Returns the data buffer, written to file by Simulator before destruction.
• QVector<qreal> * delayVec ()
  Returns the data buffer, written to file by Simulator before destruction.

• Walker::Activity activity ()
  Returns the current Activity of the walker.

Static Public Attributes

• static int mN_walkers = 0
  The number of walkers created.

Private Member Functions

• QVector2D sfmForce (QVector2D relpos, QVector2D w_vel, qreal dx)
  Function calculating the social force.

• QVector2D physForce (QVector2D relpos, qreal R, qreal dist)
  Function calculating the physical force.

Private Attributes

• qreal mBeta
  Parameter specifying the modification to the sfm by the waiting model.

• WaitModel mWaitModel
  The waiting model used by the walker.

• Distribution * mRandForceDist
  Distribution specifying the random force $F_{rand}$.

• int mSamp
  How many timestep to wait between data samplings.

• int mT
  Number of timesteps since creation.

• Activity mActivity
  The current Activity of the Walker.

• bool mCollectData
  Collecting data or not.

• int mWalkerNumber
  The number of the Walker in order of creation.

• qreal mMaxSpeed
  The maximum speed of the Walker.

• qreal mPrefSpeed
  The preferred speed of the Walker.

• QTime mTime
  The simulation time at creation of the Walker.

• qreal mWaitTime
  The wait time at a Destination.

• QVector<QVector2D> * mAccelerations
  Data buffer, written to file by Simulator before destruction.

• QVector<QVector2D> * mCoordinates
  Data buffer, written to file by Simulator before destruction.

• QVector<QVector2D> * mVelocities
  Data buffer, written to file by Simulator before destruction.

• QVector<qreal> * mUncomfort
  Data buffer, written to file by Simulator before destruction.
Data buffer, written to file by Simulator before destruction.

- QHash<QString, qreal> * mOtherData
  Data buffer, written to file by Simulator before destruction.
- QVector<qreal> * mDelay
  Data buffer, written to file by Simulator before destruction.
- QVector2D mForce
  The force affecting the Walker.
- Destination * mDest
  The current Destination of the Walker.
- WalkPath mPath
  The WalkPath of the Walker.
- QVector2D mPrefVel
  The preferred velocity \( \mathbf{v}_p \) of the Walker.
- QVector2D mPos
  The Position of the walker.
- QVector2D mVel
  The velocity of the Walker.
- QVector2D mW
  The desired velocity of the Walker.
- qreal mR
  The radius of the Walker.
- walkerDot * mDot
  The graphical representation of the Walker.
- qreal lambda
  The unisotropy parameter of the field of view.
- qreal T_s
  The look ahead parameter (a.k.a the step size)
- qreal sig
  The range scale of the social force.
- qreal mA
  The size of the social force.
- qreal tau
  The relaxation time of the adaptation to the preferred velocity.
- qreal mUncAdd
  The cumulative discomfort of the current timestep.
- QVector2D mWaitPos
  The direction of the center of the field of view.

3.16.1 Detailed Description

Objects of the Walker class represents pedestrians. The walker class handles the interaction between walkers, ie. implements the SFM. Each walker collects its own data: position, delay discomfort, velocity, and acceleration.
3.16.2 Constructor & Destructor Documentation

3.16.2.1 Walker::Walker ( QVector2D pos, WalkPath * path, QTime time, bool colDta, ModelParameters modParams, WalkerParameters walkParams, qreal beta )

The constructor.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pos</td>
<td>The starting position.</td>
</tr>
<tr>
<td>path</td>
<td>The WalkPath to be taken.</td>
</tr>
<tr>
<td>time</td>
<td>The moment of creation.</td>
</tr>
<tr>
<td>colDta</td>
<td>Bool determining if data should be collected.</td>
</tr>
<tr>
<td>modParams</td>
<td>Parameters of the SFM.</td>
</tr>
<tr>
<td>walkParams</td>
<td>Parameters of the individual walker.</td>
</tr>
<tr>
<td>beta</td>
<td>Parameter modifying the SFM when waiting.</td>
</tr>
</tbody>
</table>

3.16.3 Member Function Documentation

3.16.3.1 void Walker::calcPrefVel ( )

Calculates the preferred velocity.
Reads the navigation field of the current destination and scales with the preferred speed. Also sets the looking direction Depends on the waiting model

3.16.3.2 void Walker::calcVel ( )

Calculates the velocity of the Walker.
The speed is minimum of the desired speed and the maximum allowed speed.

3.16.3.3 void Walker::calcW ( qreal dt )

Calculates the desired velocity.
The desired speed is the old speed updated by the acceleration.

3.16.3.4 QVector2D Walker::fovFilter ( QVector2D force, QVector2D relpos )

The function representing the field of view of the Walker.
The fovFilter takes the force and the relative position of the affecting Walker and returns the weighted force.

3.16.3.5 void Walker::move ( qreal dt )

Updates the position of the Walker.
The new position is the old updated with the actual speed.

3.16.3.6 void Walker::reactTo ( Walker & w, QVector2D relpos, qreal dx )

Adds the force from walker w.
The relative position and the distance between the walkers are passed to this function of performance reasons.
This function depends on the waiting model.

3.16.3.7 void Walker::wait ( )

The walker starts waiting.
If not already waiting a waiting time is drawn from the waiting distribution of the current Destination.
3.16.3.8 void Walker::walk()

The walker starts walking.
If not already walking the walker updates its current destination to the next in the path.

3.16.4 Member Data Documentation

3.16.4.1 Activity Walker::mActivity [private]
The current Activity of the Walker.
Can take the values Walking or Waiting

3.16.4.2 QVector2D Walker::mWaitPos [private]
The preferred waiting position of the walker.

3.17 walkerDot Class Reference

GUI The visual representation of a walker.
Collaboration diagram for walkerDot:

![Collaboration diagram for walkerDot](image)

Public Member Functions

- walkerDot (Walker *w, qreal pixw, QColor color)

  *Constructs a walkerDot connected to the walker w.*

3.17.1 Detailed Description

GUI The visual representation of a walker.
A isosceles triangle pointing in the looking direction of the walker. Inherits QGraphicsItem

3.18 WalkerOriginDot Class Reference

GUI. Graphical representation of an origin.
3.18.1 Detailed Description

GUI. Graphical representation of an origin.

3.19 WalkerParameters Struct Reference

Parameters describing the walker.

3.19.1 Detailed Description

Parameters describing the walker.

3.20 WalkerSettings Class Reference

Inheritance diagram for WalkerSettings:

![Inheritance Diagram](image)

Collaboration diagram for WalkerSettings:

![Collaboration Diagram](image)

3.20.1 Detailed Description

Settings subclass

Reads parameters of the walker population from XML file.
3.21 WalkPath Class Reference

An ordered list of destinations.

Public Member Functions

- **Destination ∗ next ()**
  Returns next destination in the path.
- **Destination ∗ dest ()**
  Returns the current destination.
- **void append (Destination ∗ dest)**
  Append a new final destination.

3.21.1 Detailed Description

An ordered list of destinations.

3.22 XmlFile Class Reference

Class to handle Xml input/output.

Public Member Functions

- **XmlFile (const QString docType)**
  Constructs an XmlFile object with !DOCTYPE docType.
- **void operator<< (QStandardItemModel ∗ model)**
  Copy the data in model to the XmlFile object.
- **void operator>>(QStandardItemModel ∗ model)**
  Copy the data in the XmlFile object to model.
- **bool readFile (const QString fileName)**
  Reads an xml file into the XmlFile obj.
- **bool writeFile (const QString fileName)**
  Writes the XmlFile object to an .xml file.

3.22.1 Detailed Description

Class to handle Xml input/output.

The XmlFile class handles I/O between a file in xml-format and a tree structure stored as a QStandardItemModel.
Index

addDestination
  Simulator, 12
addOrigin
  Simulator, 12
addWalker
  Simulator, 13
addWall
  Simulator, 13
calcNavFields
  Simulator, 13
calcPrefVel
  Walker, 20
calcVel
  Walker, 20
calcW
  Walker, 20
createPath
  SimWindow, 15
DataFile, 2
  mZeros, 3
Destination, 3
  Destination, 4
distField
  Simulator, 13
Distribution
  myMath::Distribution, 5
fovFilter
  Walker, 20
getAddression
  Simulator, 13
mActivity
  Walker, 21
mModParams
  Simulator, 13
mWaitPos
  Walker, 21
mZeros
  DataFile, 3
ModelParameters, 5
ModelSettings, 5
move
  Walker, 20
myMath, 2
myMath::Distribution, 5
  Distribution, 5
next
  WalkPath, 23
Obstacle, 6
  Obstacle, 7
Origin, 7
printField
  Simulator, 13
rawLoop
  SimWindow, 15
rawRun
  SimWindow, 15
reactTo
  Walker, 20
run
  Simulator, 13
Settings, 8
SimDisplay, 8
SimSettings, 8
SimWindow, 14
  createPath, 15
  rawLoop, 15
  rawRun, 15
  SimWindow, 15
  SimWindow, 15
Simplator, 9
  addDestination, 12
  addOrigin, 12
  addWalker, 13
  addWall, 13
  calcNavFields, 13
  distField, 13
  getDestination, 13
  mModParams, 13
  printField, 13
  run, 13
  Simulator, 12
  updDisp, 13
  updForces, 13
  updState, 13
ToolMap, 15
ToolSim, 16
ToolWalker, 16
updDisp
  Simulator, 13
updForces
  Simulator, 13
updState
  Simulator, 13
wait
  Walker, 20
walk
  Walker, 20
WalkPath, 23
  next, 23
  Walker, 16
calcPrefVel, 20
calcVel, 20
calcW, 20
fovFilter, 20
mActivity, 21
mWaitPos, 21
move, 20
reactTo, 20
wait, 20
walk, 20
Walker, 20
walkerDot, 21
WalkerOriginDot, 21
WalkerParameters, 22
WalkerSettings, 22

XmlFile, 23
VTI är ett oberoende och internationellt framstående forskningsinstitut som arbetar med forskning och utveckling inom transportsektorn. Vi arbetar med samtliga trafikslag och kärnkompetensen finns inom områdena säkerhet, ekonomi, miljö, trafik- och transportanalys, beteende och samspelet mellan människa-fordon-transportsystem samt inom vägkonstruktion, drift och underhåll. VTI är världslidande inom ett flertal områden, till exempel simulatorteknik.

VTI har tjänster som sträcker sig från förstudier, oberoende kvalificerade utredningar och expertutlåtanden till projektledning samt forskning och utveckling. Vår tekniska utrustning består bland annat av körsimulatorer för väg- och järnvägstrafik, väglaboratorium, däckprovningsanläggning, krockbanor och mycket mer. Vi kan även erbjuda ett brett utbud av kurser och seminarier inom transportområdet.

VTI är ett oberoende och internationellt framstående forskningsinstitut som arbetar med forskning och utveckling inom transportsektorn. Vi arbetar med samtliga trafikslag och kärnkompetensen finns inom områdena säkerhet, ekonomi, miljö, trafik- och transportanalys, beteende och samspelet mellan människa-fordon-transportsystem samt inom vägkonstruktion, drift och underhåll. VTI är världslidande inom ett flertal områden, till exempel simulatorteknik.

VTI har tjänster som sträcker sig från förstudier, oberoende kvalificerade utredningar och expertutlåtanden till projektledning samt forskning och utveckling. Vår tekniska utrustning består bland annat av körsimulatorer för väg- och järnvägstrafik, väglaboratorium, däckprovningsanläggning, krockbanor och mycket mer. Vi kan även erbjuda ett brett utbud av kurser och seminarier inom transportområdet.

VTI is an independent, internationally outstanding research institute which is engaged on research and development in the transport sector. Our work covers all modes, and our core competence is in the fields of safety, economy, environment, traffic and transport analysis, behaviour and the man-vehicle-transport system interaction, and in road design, operation and maintenance. VTI is a world leader in several areas, for instance in simulator technology.

VTI provides services ranging from preliminary studies, highlevel independent investigations and expert statements to project management, research and development. Our technical equipment includes driving simulators for road and rail traffic, a road laboratory, a tyre testing facility, crash tracks and a lot more. We can also offer a broad selection of courses and seminars in the field of transport.