This paper was published in the International Journal of Elevator Engineers. Volume 4 No.2 (2002). It is reproduced with permission from The International Association of Elevator Engineers. This web version

Current Technology and Future Developments in Elevator Simulation

Dr Richard D Peters
Peters Research Ltd, Boundary House, Missenden Road,
Great Kingshill, Bucks HP15 6EB, UK
Fax: +44 (0)1494 716647 E-mail: richard.peters@peters-research.com

ABSTRACT

Elevate is traffic analysis and simulation software used by consultants, elevator companies and researchers world-wide. The program runs under Windows, reflecting the dominance of this platform, and customers expectations for easy to use graphical user interfaces. Other platforms may need to be considered in future years. A detailed description of the main simulation classes provides an outline specification for Elevate's object orientated elevator simulation. Elevator simulation is becoming increasingly more flexible and powerful. Current and possible future developments to Elevate are discussed.

INTRODUCTION

Elevator simulation models of varying sophistication have been written and applied for many years. The continuing improvements in computer technology and software development tools make increasing complex and comprehensive simulation models feasible.

The author writes primarily from his own experience in developing the traffic analysis and simulation software package, Elevate[1]. Other simulation programs have different approaches, but most of the key issues, variables and functions will be similar.

The discussion will be of interest to those who want to understand the basic principles of how an elevator simulation works. For aspiring authors of elevator simulations, the class descriptions and flow diagram provide a good starting point for development.

SOFTWARE TECHNOLOGY

When writing a simulation, choosing the software technology to apply is an early and important decision.

The author began writing elevator simulation programs in the 1980's. At that time, most engineering programs were being written for the IBM PC with Microsoft DOS. In

the early 1990's Microsoft Windows emerged as the dominant operating system and users began to expect easy to use, graphical user interfaces.

Elevate is written using Microsoft Visual C++. This is the author's favoured development tool as Microsoft Windows is currently the most widely used operating system. C++ allows the developer to apply objected oriented programming as discussed in following sections. It also produces very fast and portable (re-usable) code.

Future development tools and platforms for elevator simulation will be determined by the continued dominance or loss in popularity of Windows. If Windows looses favour, alternatives will need to be considered. Two possibilities are:

- Java applications – the Java language is closely related to C++, but runs on a "virtual machine" that is available for different operating systems. So, a single version of the software can run under all popular operating systems. Other single source to multiple platform development tools are emerging, and will be popular with developers.

- Internet applications can be run on the user's machine (client side) and on the computer hosting the web site (server side). For an Internet elevator simulation program, the author envisages a client side application in Java to enter data, view the simulation display and present the results. The client side application would link to a server side application, which would perform the simulation calculations in C++ or other language.

Both alternatives are already technically feasible but the development platforms are less mature, so some advanced functionality would not be available, and the programs would run slower than native Windows applications.

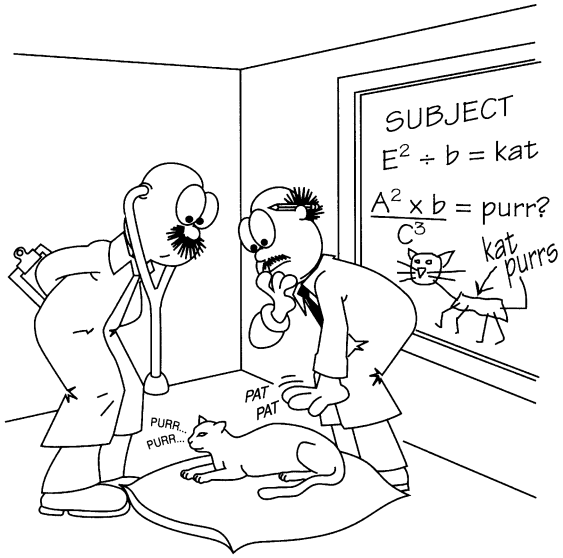
Software development tools are continually improving, so the best tools for elevator simulation need to be kept under review.

Introducing objects

Traditional structural programming techniques break a program into several smaller tasks by defining a set of functions. Object oriented programming (OOP) builds on this by introducing objects. In an object, both the variables and functions are grouped together. The behaviour (i.e. the variables and functions) of an object is defined by the class to which it belongs. Each object is an "instance" of a class.

Object oriented programming uses abstraction to allow the programmer to consider the important details of the problem in hand, and to ignore unnecessary complexities. Encapsulation is applied to hide the details of a solution so that the solution is easier to understand.

For an example of how OOP is mimicking the real world, consider *Ginger* the cat in Figure 1.



The world has a class *cat*. Everything in the *cat* class has a set of the same variables (no of paws, age, sex, etc.) and a range of functions (if you chase it runs; if you pat it, it purrs). *Ginger* is an object, and an instance of the cat class. He has all the functions and variables of a cat. The *cat* class utilises abstraction and encapsulation: If we feed Ginger, he will eat without us having to understand the complexities of his digestive system; we can concentrate on the tasks in hand such as preparing his food and stroking him.

Figure 1 Ginger the cat graphic from [2]

Returning to elevators, we can define the class *elevator* with variables such as *capacity* and *speed*, and functions such as *StartJourney()*. We can create as many elevator objects as we need; each elevator object is independent, but may use all the variables and functions defined by the class.

OOP helps break down complex problems into manageable parts that are easy to work with as they represent familiar ideas or components. The approach works extremely well for elevator simulation. The author's original simulation program in Fortran became more difficult to enhance as the program became larger and more complex. Elevate is object orientated and currently has over 20 000 lines of software code, yet adding additional functionality is relatively straightforward.

CLASS DESCRIPTIONS

Elevate has over 30 classes. Many are related to the user interface and other supporting features. The main simulation classes, their principle variables and functions are discussed in the following subsections.

Building class

The *building* class defines the building in terms of number of stories and story heights. The variables and functions are summarised in Table 1.

Class Information	Description
member variables	
int m_NoFloors;	no of floors in building
double m_FloorPositions[MAX_FLOORS];	array of floor heights
functions	
double BuildingHeight();	calculates building height

Table 1 Building class variables and functions

Motion class

The main purpose of the *motion* class is to enable an *elevator* object to determine its current position and speed while travelling. It can also tell the elevator when it will arrive at its next stop, and whether or not it can stop in time if a new call is registered in front of its next scheduled stop. The variables and functions for the *motion* class are defined in Table 2. Implementation is based on ideal elevator kinematics formulae[3]. Examples of the velocity – time plots generated by the *motion* class are given in Figure 2.

Class Information	Description
member variables	
double m_d;	journey distance,(+ for up travel, - for down) (m)
double m_D;	absolute value of m_d (m)
double m_v;	rated speed, (always +) (m/s)
double m_a;	rated acceleration, (always +) (m/s/s)
double m_j;	rated jerk (always +) (m/s/s/s)
double m_Tstart;	motor start up delay (s)
double m_t;	time elapsed since journey commenced (s)
double m_StartTime;	time journey commenced (s past ref.)
double m_CurrentTime;	current time (s past ref.)
double m_StartPosition;	start position (m above ref. height)
functions	
double JourneyTime();	journey time for trip (s)
char Condition();	journey condition (A, B, or C)
int Slice();	calculates which time slice journey is in
double Distance();	calculates the current distance travelled (m)
double Velocity();	calculates the current velocity (m/s)
double Acceleration();	calculates the current acceleration (m/s/s)
double Jerk();	calculates the current jerk (m/s/s/s)
double Position();	calculates current position (m above ref.)
double EndTime();	time when journey will be complete (s past ref.)
double MinDistance();	calculates minimum journey distance if elevator begins slowing down immediately (m)
int ConfirmDestination();	confirmation that elevator can no longer change destination, that MinDistance() is same as m_D (1- confirmed, 0 - may change)
void DataChecks();	data checks called by constructor

Table 2 Motion class, variables and functions

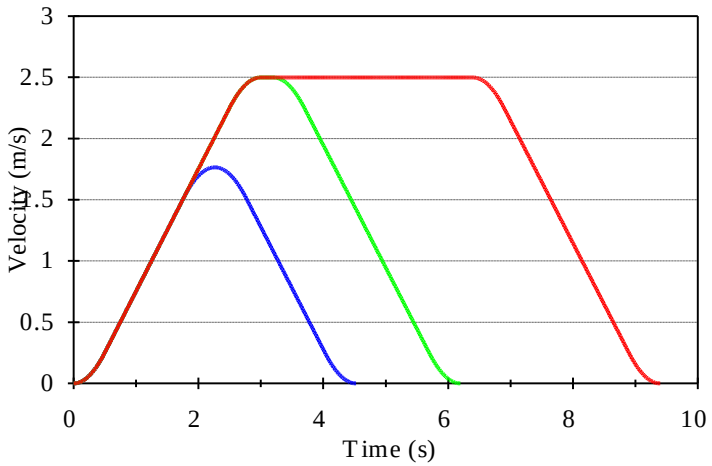


Figure 2 Velocity-time plots generated by *motion* class

Elevator class

The *elevator* class defines an elevator (rated speed, capacity, floors served, etc.) and its current status (position, speed, load, etc.). The *motion* class is applied to enable the elevator to move according to the selected journey profile. The *elevator* class includes algorithms to allow elevators to answer landing and car calls according to the principles of directional collective control. (Most elevator control systems adopt a directional collective control strategy regardless of the complexities of the dispatcher algorithms.) The main *elevator* class variables and functions are defined in Tables 3 and 4.

Class Information	Description
about the elevator	
int m_Capacity;	nominal elevator capacity (kg)
double m_Velocity;	rated elevator velocity (m/s)
double m_Acceleration;	rated elevator acceleration (m/s/s)
double m_Jerk;	rated elevator jerk (m/s/s/s)
double m_MotorStartDelay;	motor start up delay (s)
double m_DoorPreOpen;	door pre-opening (s)
double m_DoorOpen;	door open time (s)
double m_DoorClose;	door closing time (s)
double m_DoorDwell1;	door dwell time 1 (s) (time doors will wait until closing if beam not broken)
double m_DoorDwell2;	door dwell time 2 (s) (time doors will wait until closing after beams have been broken/cleared)
int m_DoorBeams;	flag for status of door beams (corresponding to passenger transfer - 1 beams broken, 0 clear)
how the elevator serves the building	
int m_NoFloors;	no of floors in building
int m_Home;	home floor/default parking position
double m_FloorPositions[MAX_FLOORS];	positions of floors in building (m above ref.)
int m_FloorsServed[MAX_FLOORS];	floors served by elevator (1 yes, 0 no)
about the current status of the elevator	
int m_CarCall[MAX_FLOORS];	car calls registered (1 registered, 0 not)
int m_ParkCall[MAX_FLOORS];	parking calls; doors do not open doors on arrival
int m_ParkOpenCall[MAX_FLOORS];	parking calls, elevator parks with doors open
int m_UpLandingCalls[MAX_FLOORS];	up landing calls allocated to elevator by dispatcher
int m_DownLandingCalls[MAX_FLOORS];	down landing calls allocated by dispatcher
int m_TravelStatus;	travel status, (1 travelling, 0 at floor)
int m_Direction;	direction of travel (-1 down, 0 neither, 1 up)
double m_DestinationPosition;	current destination position (m above ref.)
double m_StartPosition;	position current journey started (m above ref.)
double m_JourneyStart;	time elevator journey started (s past ref.)
int m_CurrentLoad;	current car load (kg)
int m_DoorStatus;	door status (1 fully open, 2 closing, 3 fully closed, 4 opening)
double m_DoorsStart;	time doors started opening/closing (s past ref.)
double m_TimerT1;	time timer T1 began (s past ref.)
double m_TimerT2;	time timer T2 began (s past ref.)
double m_PersonStart;	time current person began loading/unloading (s past ref.)
double m_CurrentTime;	current time (s past ref.)
double m_DestinationTime;	arrival time next planned stop (s after ref.)
double m_CurrentPosition;	current position (m above ref.)
double m_CurrentDistance;	distance travelled on current trip (m)
double m_CurrentVelocity;	current velocity (m/s)
double m_CurrentAcceleration;	current acceleration (m/s/s)
double m_CurrentJerk;	current jerk (m/s/s/s)
double m_QuickestStopPosition;	next stop elevator can make (m above ref.)
int m_DestinationFloor;	current destination floor no.

Table 3 Elevator class variables

Class Information	Description
void Reset(building b);	sets elevator to home position, cancels all calls, etc.
int StartJourney(int floor);	start journey to destination "floor"
int ChangeJourney(int floor);	change journey, new destination, "floor"
void UpdateDestination();	check for calls allocated to elevator and set destination
void SetDestination();	set destination/direction travel
void Update(double CurrentTime);	update time (s); this function updates the status of the elevator (position, speed, door operation, etc.)
void RemoveLandingCall(int direction, int floor);	removes landing call - called by class when elevator arrives at landing
int LowestFloorServed();	returns number of lowest floor served by elevator
int HighestFloorServed();	returns number of highest floor served by elevator
int FloorAt();	return floor no if not travelling
int FloorNo(double position);	returns floor no at position
double QuickestStopPosition();	next stop elevator could make (m above reference)
double QuickestStopTime();	time of next stop elevator could make (s after ref.)
int QuickestFloorStopFloor();	floor of next stop elevator could make
double QuickestFloorStopPosition();	position of next stop elevator could make (m above reference)
double QuickestFloorStopTime();	time of next stop elevator could make (s after ref.)

Table 4 Elevator class functions

Dispatch class

The *dispatch* class defines rules for allocating which elevator serves which calls. When a passenger presses a landing call, the *dispatch* class decides which elevator should serve the call. The dispatch class variables and functions are defined in Table 5.

Class Information	Description
member variables	
int m_Algorithm;	dispatcher algorithm no. selected
int m_NoFloors;	number of floors in building
int m_NoElevators;	number of elevators
double m_FloorPositions[MAX_FLOORS];	floor positions (m above reference)
int m_UpLandingCalls[MAX_FLOORS];	up landing calls registered with dispatcher
int m_DownLandingCalls[MAX_FLOORS];	down landing calls registered with dispatcher
member functions	
void CancelLandingCalls(elevator l[MAX_ELEVATORS]);	cancel landing call when elevator arrives at floor
void Reset(building b,int NoElevators,elevator l[MAX_ELEVATORS]);	resets dispatcher, sets up member variables
int Update(double CurrentTime,elevator l[MAX_ELEVATORS],motor m[MAX_ELEVATORS], double SimulationTimeStep);	update dispatcher; this function updates the status of the dispatcher, allocating calls, etc.

Table 5 Dispatch class functions and variables

Person class

The *person* defines a person, what time he/she arrives at the landing station, where he/she wants to go, their mass, etc. Once the journey is complete, the class provides details about passenger waiting and transit times. Variables and functions of the person class are defined in Table 6.

Class Information	Description
member variables	
double m_TimeArrived;	time passenger arrived at landing (s past reference) (taken to be when call button pressed).
int m_ArrivalFloor;	arrival floor
int m_Destination;	destination floor
int m_Mass;	passenger mass (kg)
int m_LoadingThreshold;	threshold determining whether passenger will get into this elevator or wait for the next (%) e.g. 80% means that passenger will not load elevator if the elevator will then be >80% full
double m_LoadingTime;	passenger loading time (s)
double m_UnloadingTime;	passenger unloading time (s)
double m_TimeBeganTransfer;	variable used to store when passenger transfer (loading and unloading) began (s past reference)
int m_CurrentStatus;	current status of passenger's journey; 1 yet to arrive, 2 waiting, 3 loading, 4 travelling, 5 unloading, 6 journey completed
int m_ElevatorUsed;	elevator used by passenger
double m_TimeElevatorArrived;	time responding elevator arrived, taken from when the doors began to open (s past reference)
double m_TimeReachedDestination;	time responding elevator reached destination, taken from when the doors began to open (s past reference)
member functions	
void NewLandingCalls(double CurrentTime,dispatch& d);	registers new landing calls when passenger arrives
void Update(double CurrentTime,int NoElevators,elevator l[MAX_ELEVATORS],dispatch& d);	update status of passengers, adjust elevator load, break/clear beams, etc.
int Direction();	returns direction of call (1 up, -1 down)
double WaitingTime();	passenger waiting time (s)
double TransitTime();	passenger transit time (s)

Table 6 Person class functions and variables

FLOW DIAGRAM

Elevate is a time slice simulation; it calculates the status (position, speed, etc.) of the elevators, increments the time, re-calculates status, increments time, and so on. A simplified flow diagram of simulation is given in Figure 3.

FUTURE DEVELOPMENTS

There are many possible developments to an elevator simulation program such as Elevate. Below are some of the enhancements that are in progress or under consideration.

Closer integration with installed control systems

Elevate allows users to program their own dispatcher algorithms into a dynamic link library (DLL) which is called by the program. There is ever increasing interest in and demand for this feature; simulation is an excellent tool for developing, testing and demonstrating control systems.

Currently customers are programming dispatch algorithms in a format suitable for Elevate. In the future we envisage closer and closer links between Elevate and installed control systems, so that algorithms can be exchanged with compatible systems at the click of a button.

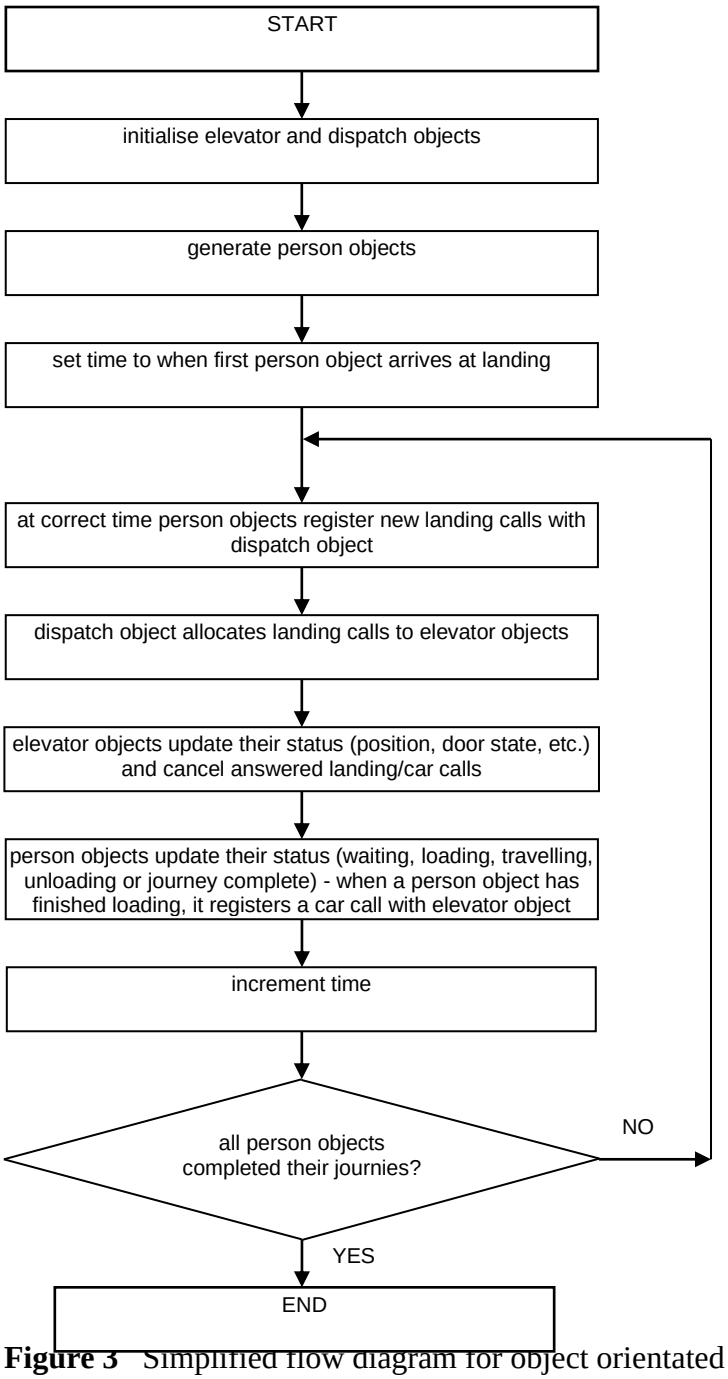


Figure 3 Simplified flow diagram for object orientated elevator simulation

Total building models

There are a number of pedestrian modelling software tools. These have been developed to model the evacuation of buildings in an emergency, and the flow of people in transport terminals such as airports and train stations.

These programs currently have either very crude or no elevator model. We are considering the possibility of linking programs so that the elevator model in Elevate can contribute a total models of building circulation.

The more advanced pedestrian modelling software programs are object orientated, so the interface between programs is conceptually simple. In its normal mode, Elevate generates its person objects which push the landing buttons, wait for the elevator, get in and press the car call buttons, etc. In a total building model the person objects will be created by the pedestrian modelling software. A person object will move around the building until he/she needs to use an elevator, when he/she will be introduced into the elevator simulation model. Once their elevator trip is complete, the person object will be returned to the total building model at the new floor level.

Traffic analysers

In the assessment of passenger service quality, the most important traffic analysis results are average passenger waiting and transit times. Using simulation we can measure these results as we know at what time every passenger arrives, and how and when they are transported.

Traffic analysers can be interfaced with an installed elevator control system to record the time every landing and car call is made and cleared. Many modern control systems incorporate similar functionality. A range of traffic and performance measures can be determined, for example:

- average response time to landing calls by time of day
- distribution of response times
- distribution of car calls by floor

A traffic analysers does not measure average waiting and transit times as an elevator does not know when someone arrives in a lobby; it only knows when a landing or car call button are pressed. There is often more than one person behind a call, but the traffic analyser will not know this. (For this reason, it is generally unreliable to use a conventional traffic analyser results to assess the demand on an existing system, or to evaluate the benefits of modernisation. The author recommends that designers carry out surveys counting *people* as opposed to *calls*.)

Within a simulation program it is straightforward to implement a compete traffic analyser. This will have a number of applications:

- if using a simulation to model existing installations with an installed traffic analyser, the simulation traffic analyser should present similar results.

- there is a theoretical relationship between the passenger arrival rate and, the time from landing calls being cleared to being re-registered[4]. With simulation it will be possible to investigate this relationship further and possibly develop software that can estimate actual traffic flow from traffic analyser data.

Modelling of new technology

Established technology dictates that an elevator moves in one dimension, up and down a shaft. And that there should only be one elevator per shaft. This is a major limitation, especially in high rise buildings where the relative core space required by the elevators is high. The ultimate solution it to have multiple elevators in a single shaft, and for them to be able to overtake though moving in at least two dimensions (side to side as well as up and down). Elevate is currently being extended to model this so that one of our customers can develop an appropriate control system for a research project. The drive technology to apply this control system is yet to be developed, but recent developments in self-propelled elevators [5] suggest that the concept is feasible.

Multiple deck elevators

Elevate currently models single deck elevators. Double deck elevators will be added to Elevate in due course. If required, it would be feasible to model triple, quadruple or even n deck elevators (where n is any number).

CONCLUSIONS

Elevator simulation is now readily available and increasingly popular for traffic analysis and control system design.

In recent years software technology has developed so that complex programs are easier to develop. Object orientated programming technology is extremely helpful to the developer, and graphical user interfaces help make programs easy to use.

An elevator simulation program requires many classes. We have discussed Elevate's principle simulation classes, which are building, motion, elevator, dispatch and person.

Elevate and other simulation programs will continue to be developed to provide increased functionality. Some current and future possible developments have been discussed. The author welcomes other comments and suggestions.

REFERENCES

1. Elevate traffic analysis and simulation software, distributed by *Elevator World, Inc*.
2. Perry G, Ross J *Visual C++ By Example* (Indianapolis: Que Publishing) (1994)
3. Peters R D *Ideal Lift Kinematics: Derivation of Formulae for the Equations of Motion of a Lift* International Journal of Elevator Engineers, Volume 1 No 1 (1996)
4. Peters R D *Green Lifts?* Proceedings of CIBSE National Conference 1994 (The Chartered Institution of Building Services Engineers) (1994) (republished by Elevator World, June 1995 and by Elevation, Autumn 1995)
5. SchindlerMobile elevator, www.schindler.co.uk

TRADEMARKS

BIOGRAPHICAL DETAILS

Dr Richard D Peters has a degree in Electrical Engineering, and a Doctorate for research in Vertical Transportation. He is a Chartered Engineer, Member of the Institution of Electrical Engineers, and Member of the Chartered Institution of Building Services Engineers. He is Chairman of the Lifts Group of the Chartered Institution of Building Services Engineers and a visiting lecturer at UMIST.

Dr Peters worked for ten years with international engineering consultants Ove Arup & Partners. In 1997, he set up his own company, Peters Research Ltd to provide software development and engineering consultancy. Dr Peters provides advice on a range of vertical transportation issues to clients and has notable expertise in the mathematical modelling of vertical transportation systems. Dr Peters' traffic analysis software used world-wide.