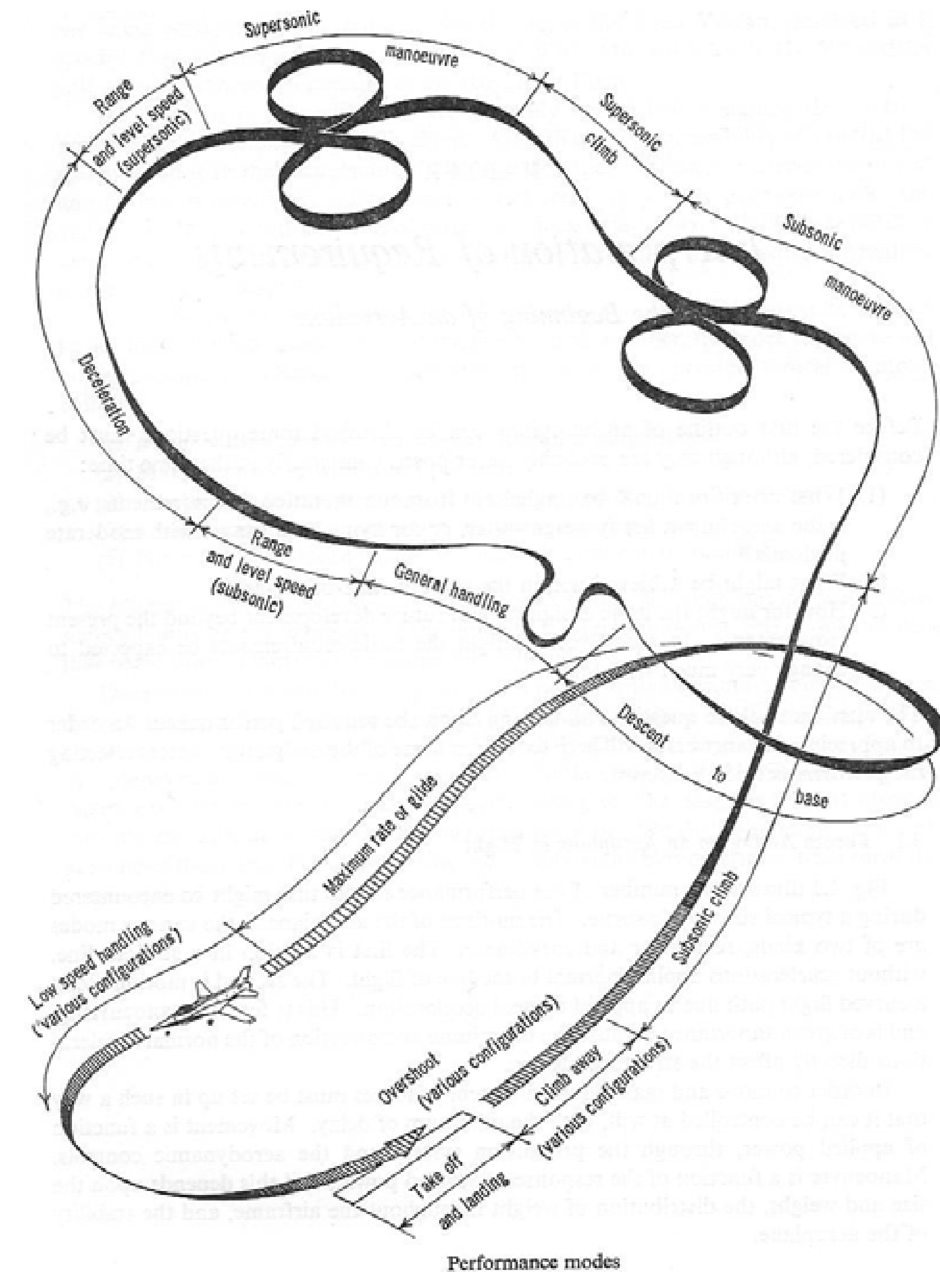
Comsim-Platform

an Object-Oriented Basis for Computing Aircraft Performance and Flight Trajectories

ANDERS HASSELROT



Source: Darrol Stinton (1966)

FOI, Swedish Defence Research Agency, is a mainly assignment-funded agency under the Ministry of Defence. The core activities are research, method and technology development, as well as studies conducted in the interests of Swedish defence and the safety and security of society. The organisation employs approximately 1000 personnel of whom about 800 are scientists. This makes FOI Sweden's largest research institute. FOI gives its customers access to leading-edge expertise in a large number of fields such as security policy studies, defence and security related analyses, the assessment of various types of threat, systems for control and management of crises, protection against and management of hazardous substances, IT security and the potential offered by new sensors.

FOI
Defence Research Agency
Defence & Security, Systems and Technology
SE-164 90 Stockholm

Phone: +46 8 555 030 00
Fax: +46 8 555 031 00

www.foi.se

FOI-R--1846--SE Technical report
ISSN 1650-1942 June 2007

Defence & Security, Systems and Technology

Anders Hasselrot

Comsim-Platform
an Object-Oriented Basis for Computing Aircraft Performance and Flight Trajectories

FOI-R--1846 --SE
ISSN 1650-1942

Technical report
June 2007

Defence & Security, Systems and Technology

Issuing organization	Report number, ISRN	Report type
FOI – Swedish Defence Research Agency Defence & Security, Systems and Technology SE-164 90 Stockholm	FOI-R--1846--SE	Technical report
	Research area code 2. Operational Research, Modelling and Simulation	
	Month year June 2007	**Project no.** B66066
	Sub area code 21 Modelling and Simulation	
	Sub area code 2	
Author/s (editor/s) Anders Hasselrot	**Project manager** Adam Jirasek	
	Approved by Maria Sjöblom	
	Sponsoring agency HISAC (a European Commission project)	
	Scientifically and technically responsible	

Report title

Comsim-Platform an Object-Oriented Basis for Computing Aircraft Performance and Flight Trajectories

Abstract

A new flight simulation package, named Comsim-Platform and written in the program language C++, has been created to replace the LISP-based PcP (being in use for environmental flight studies at FOI). It re-uses the aircraft model, Platform, of the latter, and it has been provided with a new flight management system, including navigation. All numerical integrations and event notices are handled by the existing C++ based simulation package Comsim.
The Comsim-Platform package is intended to act as a basis for various types of flight simulations. The Flight/Navigation application that has been created for testing the functionality of Comsim, with special views on flight/navigation management, has proved its capability. This report documents the usage and functionality of Comsim-Platform and Flight/Navigation.
The ease of reproducing the detailed procedures of recorded flights has been demonstrated. This has been used in the validation of the capability of the Comsim-Platform package. Using accurate data for the airframe and its engine(s) of a studied flight, this report proves and documents the soundness of the flight-mechanic basis of the extended Platform model.
The final conclusion is that the Comsim-Platform package is ready to be used, with existing version of Flight/Navigation, or with some type of modification/utilization of the latter modules.

Keywords

Flight, simulation, aircraft, platform, program package, computer programming, navigation, validation, flight-mechanics

Further bibliographic information	Language English
ISSN 1650-1942	**Pages** 56 p.
	Price acc. to pricelist

Utgivare	Rapportnummer, ISRN	Klassificering
FOI - Totalförsvarets forskningsinstitut Försvars- och säkerhetssystem 164 90 Stockholm	FOI-R--1846--SE	Teknisk rapport
	Forskningsområde 2. Operationsanalys, modellering och simulering	
	Månad, år Juni 2007	**Projektnummer** B66066
	Delområde 21 Modellering och simulering	
	Delområde 2	
Författare/redaktör Anders Hasselrot	**Projektledare** Adam Jirasek	
	Godkänd av Maria Sjöblom	
	Uppdragsgivare/kundbeteckning HISAC (ett European Commission projekt)	
	Tekniskt och/eller vetenskapligt ansvarig	

Rapportens titel

Comsim-Platform - en objektorienterad bas för beräkning av prestanda och flygbanor för flygplan

Sammanfattning

Ett nytt flygsimuleringspaket, som kallas Comsim-Platform och som är skrivet i programspråket C++, har skapats för att ersätta det LISP-baserade PcP (använd i samband med miljörelaterade flygsimuleringar vid FOI). Det återanvänder flygplansmodellen Platform hos det senare programmet, och det har försetts med ett nytt flyghanteringssystem, inklusive navigering. Alla numeriska integrationer och händelsenotiser hanteras med hjälp av det existerande C++-baserade simuleringspaketet Comsim.

Comsim-Platform-paketet är avsett att fungera som grund för olika typer av flygsimuleringar. Flight/Navigation-applikationen, som har skapats för att testa funktionaliteten hos Comsim, med speciellt fokus på flygbane- och navigeringshantering, har visat sin förmåga. Denna rapport dokumenterar användningen av och funktionaliteten hos Comsim-Platform och Flight/Navigation.

Enkelheten att reproducera detaljerade procedurer från inspelade flygdata har demonstrerats. Detta har utnyttjats i valideringen av kapaciteten hos Com-Platform-paketet. Genom att använda noggranna data för bland annat aerodynamik och motorer för ett flygplan, tillsammans med data för en inspelad flygning, bevisar och dokumenterar denna rapport tillförlitligheten hos den flygmekaniska basen i den utökade Platform-modellen.

Slutsatsen är att Comsim-Platform-paketet kan tagas i bruk, med existerande version av Flight/Navigation eller någon typ av modifiering/användning av de senare modulerna.

Nyckelord

Flyg, simulering, flygplan, plattform, programpaket, datorprogrammering, navigering, validering, flygmekanik

Övriga bibliografiska uppgifter	**Språk** Engelska
ISSN 1650-1942	**Antal sidor:** 56 s.
Distribution enligt missiv	**Pris: Enligt prislista**

Contents

Nomenclature 7
Notations 7
Abbreviations 9
1 Introduction 11
1.1 Purpose 11
1.2 Scope 11
1.3 Background 11
1.4 Summary of Capability of Comsim-Platform-Flight 13
2 The Platform Model 15
2.1 Philosophy and Definitions 15
2.2 The Aircraft 18
2.2.1 Aerodynamics 18
2.2.2 Engine Characteristics 19
2.2.3 Other Aircraft Characteristics 19
2.3 The Atmosphere 20
2.4 Flight-Mechanics 20
3 The Navigation Module 23
3.1 Path Modelling 23
4 The Simulation Package (Information for the Software Developer) 25
4.1 The Comsim Structure 25
4.1.1 Class cslink 25
4.1.2 Class objectattribute : public cslink 25
4.1.3 Class variable : public objectattribute 26
4.1.4 Class continuous : public objectattribute 26
4.1.5 Class reporter : public objectattribute 26
4.1.6 Class object : public objectattribute 26
4.1.7 Class process : public cslink 27
4.1.8 Class eventnotice : public cslink 27
4.1.9 Class monitor : public process 27
4.1.10 Class integrator : public monitor 28
4.1.11 Combined Simulation Module, Comsim 28
4.2 Comsim Usage 28
4.2.1 Class Flight : public continuous 28
4.2.2 Class Pilot : public process 31
4.2.3 Class Report : public reporter 31
4.2.4 Module Integration 31
4.3 Comsim Flow 34
5 Flight Application (Information for the Software User) 35
5.1 Flight Profile and Navigation 35
5.1.1 Input for Flight Profile 35
5.1.2 Input for Navigation 38
5.1.3 Interaction between Flight Profile and Navigation 39
5.1.4 Example 40
6 Validation (Information for the Generalist) 43
6.1 Modelled Data 43
6.1.1 Aerodynamic Data 43
6.1.2 Engine Data 43
6.1.3 Miscellaneous Data 43
6.1.4 Flight Data 45
7 Conclusions 51
References 53
Appendix A. General Instructions on Creating C++ Classes from FORTRAN Codes 55

Nomenclature

Notations

[a]	The aerodynamically related frame of reference (X-Z plane = aircraft symmetry plane, X along speed vector)	
[b]	The body fixed frame of reference (X-Z plane = aircraft symmetry plane, X along nose)	
[e]	The fixed earth frame of reference (X-Z plane = vertical plane, X to the north)	
[w]	The wind related frame of reference (X-Z plane = vertical plane, X along speed vector)	
a	Velocity of sound	[m/s]
A	Afterburner (reheater) On/Off	A/-
acc	Aircraft acceleration	[m/s^2]
CAS	Calibrated Air Speed [m/s] (ideally, without position errors)	[m/s]
C_D	Aircraft drag coefficient, defined as D/(q*S)	[-]
C_{Di}	C_D induced by C_L	[-]
C_{Dmin}	Minimum C_D, i.e. at C_L^*	[-]
C_L	Aircraft lift coefficient, defined as L/(q*S)	[-]
C_L^*	C_L at Cdmin	[-]
C_{Lmax}	Maximally allowed C_L limit, usually corresponding to C_L at 1.2*V_{stall} (take-off) or 1.3*V_{stall} (landing)	[-]
C_p	Specicic heat at constant pressure	[J/kg/K]
C_v	Specicic heat at constant volume	[J/kg/K]
D	Aircraft drag force	[N]
ΔC_{DH}	C_D increment due to altitude, relatively to sea-level	[-]
$\Delta C_{D\,item}$	C_D increment due to effect of an *item*: high-lift flaps or external load	[-]
ΔK	Non-parabolic adjustment to K, to account for a special effect, such as trim drag when balancing the centre-of-gravity effect	
ΔT	Temperature increment to be used with standardized atmosphere model, to account for local conditions	[K]
F	Ground friction force due to rolling or braking	[N]
g	Acceleration due to gravity	[m/s^2]
H	Altitude	[m]
$\dot{H}$	Rate-of-climb (rate-of-descent when negative)	[m/s]
IAS	Indicated Air Speed as seen on instrument (with positional errors), strictly. However used here synonymously with CAS.	[m/s]
Item	Aircraft configuration element: external load (mainly military application) or wing flap state (mainly civil application)	
K	Parabola factor with non-parabolic adjustment for computing C_{Di}	
K'	Parabolic part of K	
$K_{deg\text{-}m}$	Factor for translating movement on earth from [degrees] to [m]	
L	Aircraft lift force	[N]
lat	Latitude	[degrees]
$long$	Longitude	[degrees]
M	Mach number, = V/a	[-]
m	Aircraft mass	[kg]
$\dot{m}_{fuel}$ or FC	Fuel flow	[kg/s]

N	Normal force due to ground reaction	[N]
n_H	Loadfactor due to horizontal turn	[-]
n_T	Turnfactor, defined as vector sum of gravity factor (cosγ) and n_z	[-]
n_V	Loadfactor due to vertical turn	[-]
n_Z	Loadfactor, defined as L/(m*g)	[-]
p	Static pressure	[Pa]
q	Dynamic pressure = $0.5 * \rho * V^2 = 0.7 * p * M^2$ (assuming $\kappa = 1.4$)	[Pa]
R	Molar gas constant in the gas law (=287 for air)	[J/kg/K]
r	Horizontal turn radius	[m]
S	Reference area, usually given by the wing contour extended to the centre-line	[m^2]
S	Horizontal, accumulated distance of flight path	[m]
$\dot{S}$	Horizontal translation speed	[m/s]
T	Absolute air temperature [K]	[N]
T	Installed engine thrust	[N]
throttle	Part thrust, ratio of T to maximum T, both at the current H and M	[-]
time	Simulation time	[s]
V	Air speed	[m/s]
W	Aircraft weight	[N]
X	North pointing axis of [e] frame of reference, primarily used together with Y for the navigation	[m]
Y	East pointing axis of [e] frame of reference, primarily used together with X for the navigation	[m]
α	Angle-of-attack, i.e. between nose and speed vector (in "clean" flight)	[degrees]
χ	Course angle, i.e. angle to the right from the north direction for the horizontal translation (projection of speed vector). Forms an Euler (semi-orthogonal) system with μ and γ	[degrees]
δ	Ratio, defined by $p/p_{sea\text{-}level}$	[-]
$\Delta\alpha$	α shift of C_L-curve, due to change of configuration, such as deployment of high-lift devices (flaps)	[degrees]
γ	Climb (elevation) angle for the speed vector. Forms an Euler (semi-orthogonal) system with μ and χ.	[degrees]
κ	Ratio of C_p/C_v (=1.4 for air)	[-]
μ	Bank angle, by which the vertical aircraft symmetry plane is tilted to the right along the speed vector. Forms an Euler (semi-orthogonal) system with γ and χ	[degrees]
μ_g	Ground friction coefficient during rolling or braking	[-]
θ	Attitude angle, i.e. angle of the aircraft nose axis relative the horizontal plane. Forms an Euler (semi-orthogonal) system with ϕ (roll angle) and ψ	[degrees]
ρ	Air density	[kg/m^3]
τ	Turnplane angle, the angle between the vertical plane containing the speed vector and the plane containing the sidewise-upward turn direction.	[degrees]
ψ	Angle to the right from the north direction for the horizontal nose direction (projection of the nose axis). Forms an Euler (semi-orthogonal) system with ϕ (roll angle) and θ	[degrees]

Abbreviations

ALGOL	ALGOrithmic Language, a high level language designed in late 50s specifically for programming scientific computations, formalized in reports ALGOL 58, ALGOL 60 and ALGOL 68. It never attained popularity, mainly due to lack of input/output definition. It had a profound influence on structure of later generation languages, of which SIMULA was one.
C++	"C with classes" , as C++ was released in 1983 by Bjarne Stroustrup at Bell Labs, implements some of the object-oriented technology of SIMULA. It is a superset of ANSI C. It supports procedural programming, data abstraction, object-oriented programming, and generic programming. Among the features are: virtual functions, operator overloading, multiple inheritance, templates, and exception handling. The current standard is ISO/IEC 14882:2003.
C	A procedural computer language, oriented towards system ("close to the machine") programming, started its life at Bell Telephone Laboratories in 1969 when the language and the operating system UNIX was developed together. It has a small kernel and relies on external libraries. There are different flavours of C, such as the one of "Kernighan and Ritchie" and the ANSI standard (mid 80s). C of the latter type forms the basis of C++.
f2c	A free and extremely well-debugged program that converts standard Fortran-77 code to standard C code. It comes with all GNU/Linux (free UNIX-like operating system) distributions.
FFA-APP	FFA Aircraft Performance Program (in FORTRAN)
FORTRAN	FORmula TRANslation, a computer programming language that was released in 1957 by John Backusin at IBM. Each new standard implied increased capability. Well-known standards are FORTRAN 4 and 77. These were basically supporting procedural programming until FORTRAN 95 with object-oriented technology was released.
LISP	LISt Processing, a family of computer programming languages closely connected with the artificial intelligence research community. An ANSI standard exists: Common LISP.
LTO	Landing and Take-Off, a flight procedure covering all segments below 3000 feet of altitude. ICAO defines a standard procedure as a basis for computing engine emissions. The standard consists of four segment-thrust level-time combinations: take-off-100%-0.7 min, climb out-85%-2.2 min, approach-30%-4min, and idle-7%-26 min. At FOI, more physically related LTOs are often used: times and thrusts determined by reached speed, altitude (the first two segments), force equilibrium (third and fourth segments).
PcP	Aircraft flight simulation program written in Common LISP: Programmable Civil Pilot
SIMULA	SIMULA originally developed in the 60s at Norsk regnesentral, Oslo, by Ole-Johan Dahl and Kristen Nygaard as a computer language for simulation, starting from ALGOL. The release of SIMULA 67, as a general programming language, introduced the concepts of object-orientation, inheritance and classes. This version is often regarded as the first object-oriented language. Both Smalltalk and C++ have been inspired by SIMULA 67.
Smalltalk	An object-oriented language that Alan Kay began developing in 1971 at Xerox Palo Alto Research Center (PARC), inspired by some aspects of SIMULA (message passing). Lateron it incorporated development environment with class library browser and SIMULA-like class inheritance. The first released version outside PARC, Smalltalk-80, introduced metaclasses to support the paradigm of "everything is an object".

1 Introduction

In order to satisfy the need of an easily modified software for high-level aircraft flight simulation – examples of this are emissions and noise computation along flight paths – FOI has created a batch-oriented (console) application, using input with flight profile and map navigation specifications. The application is based on a general simulation package and an aircraft module. The latter also includes modules for handling flight-mechanics. Having this package means that it can be re-used in other flight applications. This package has been named Comsim-Platform.

The mentioned application was created to be useful for civil applications, but also to verify the capability of Comsim-Platform. For this validation, flight recorder data were supplied by an airline operator, of which one flight with known aircraft and engine data was chosen.

1.1 Purpose

This report is intended for three main audiences:

- Generalists, who wish to have knowledge on aircraft modelling and validation aspects. Chapters 2, 3, and 6 are recommended.
- Software users of the present application or its derivatives. Chapters 2, 3, and 5 are recommended. Note that the exact description of the input format for the aircraft and its engine is found in Hasselrot *et al*. (1987).
- Software developers, who wish to develop applications based on the Comsim-Platform package, or to modify the present application. Chapters 2, 3, and 4 are recommended. Note that for a fuller description of the Comsim part, the reader should turn to Aronsson (1991). The source code is not included in this report.

1.2 Scope

The layout of the report has been aimed at giving an overall view of modelling, simulation, validation, and usage. Hereby most questions of general nature should be answered. In addition, examples of code utilization and input data have been included to aid the software developer/user.

1.3 Background

Hitherto FOI had been using the LISP and FORTRAN based PcP software [Stehlin, 1999] for high-level environment simulations, but it was time-consuming to use (due to many types of preparations) and difficult to modify for special needs (due to lack of LISP kowledge). As there were several planned/ongoing projects that called for easier simulation software, the environment group of FOI decided to perform a quick measure to ease the situation: create an application using existing modules, the C++ based Comsim package [Aronsson, 1991] and the aircraft module from FFA-APP [Hasselrot *et al.*] written in FORTRAN.

Comsim contains a translation – a work performed at FFA as a thesis for ME degree - of corresponding parts in NYCOND simulation package [Righard, 1981], which in itself has ancestors in CONDIS [Månsson, 1980] and Combinedsimulation [Heldsgaun, 1978-9]. Indeed, it is a long FOA-FFA-FOI heritage! The language of the original codes is SIMULA 67, commonly regarded as the first object-oriented language. Comsim has implemented only a few of the integration methods that are available in the SIMULA versions, but this is not a serious problem, as our application will not simulate rapid processes.

The idea of adopting the FFA-APP module is attractive, as it is being used in the PcP software (translated into C which Common LISP can handle). This module was named Platform. Thus the database of aircraft created for the PcP can be reused. Using the Comsim package is attractive, because hereby the object-oriented philosophy can be introduced: general class declarations and specific of class objects. There is a viable idea of creating a class version of Platform. Then it will be possible to create several Platform objects (one for each aircraft), without the risk of internal data collision. This is useful for cases where several aircraft act/interact in the same simulation, i.e. when studing collision risks in dense air traffic or air combat.

The focus for the new FOI application, Flight, which may be regarded as a first attempt in creating something useful, has been creating a re-useable package, with unmodified Comsim and a class version of Platform. In creating the latter, the FORTRAN code (after some small modification) was automatically translated to C by means of a public domain software 'f2c' (run on a PC), see Feldman (1990), whereupon the result was edited into a class version. This editing was performed according to a procedure described in Appendix A.

Figure 1 shows an overview of how Comsim and Platform packages are utilized in an application (Flight). The Comsim structure and its usage are described in chapter 4.1 and 4.2, respectively. The FORTRAN based Platform has been extended, in the form of a C++ subclass PlatformX, to include a comprehensive set of flight-mechanic relationships (to aid implementation of Flight software and future versions). Note that Navigation is a free-standing class that is called in our present Flight application. The models of Platform, PlatformX and Navigation are described in chapters 2, 2.4, and 3, respectively.

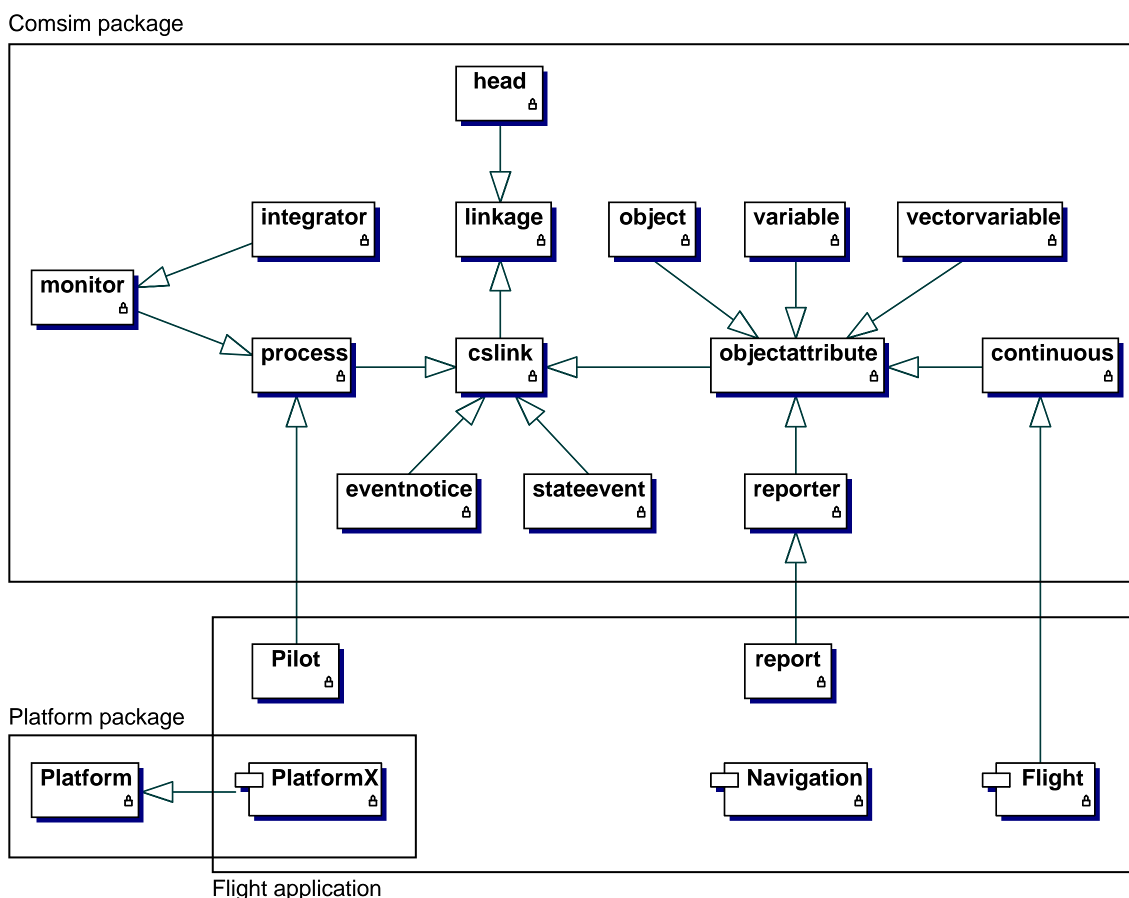


Figure 1. Class diagram (C++ structure) of Comsim and Platform packages including a Flight application.

1.4 Summary of Capability of Comsim-Platform-Flight

Reference: Hasselrot, A. (2007): “Comsim-Platform — an Object-Oriented Basis for Computing Aircraft Performance and Flight Trajectories”, FOI-R--1846-SE (2007).

Comsim-Platform is a general aircraft simulation package written in the language C++, with defined aircraft and simulation handling. Aircraft input is in the form of aerodynamic coefficients, thrust- and fuel flow tables, and weights. The aircraft is treated kinematically as a masspoint, on the assumption that the aircraft flies ideally and aerodynamically balanced.

Flight is a specific software, designed primarily for civil applications and based on Comsim-Platform, to handle one aircraft and its flight procedure. Apart from calling for aircraft data from Comsim-Platform, Flight requires input in the form of detailed flight profile data and map coordinates for the navigation. Flight can easily be modified to manage several aircraft.

Simulation, not only of flight but also of ground operations, with Flight is performed by calling an aircraft—engine combination and running a batch of flight and navigation commands residing in an input file.

Flight commands are given as ‘taxi’, ‘takeoff’, ‘climb’, ‘accelerate’, ‘cruise’, ‘descend’, ‘approach’, and ‘land’, in any order. Special setup commands are ‘setstate’ and ‘calibrate’. An LTO procedure can be defined using some of these commands. Ground operations, where the rolling friction coefficient and the angle-of-attack are specified, are treated with regard to partial lift depending on the current speed. Each Flight command is performed with its own set of controlling parameters, such as thrust level, calibrated airspeed, Mach number, etc., and it is finished when a certain, specified flight condition is met, such as a speed or an altitude has been reached. The flight simulation is then continued with the next Flight command.

The execution of Navigation commands is performed in parallel with the flight procedure, where the accumulated flight distance will guide through the pre-defined map path in the form of lines and arcs. The geometrical path elements are derived from the coordinates given in the batch file, which can be specified as local, metric coordinates or as global coordinates as latitude and longitude. The coordinates are useful for both small-scale navigation, such as taxiing, and larger scales up to global level, when great circle movement, i.e. along shortest paths around Earth, can be specified.

The flight simulation results in an output, containing a time history, second by second, of various flight states, such as the thrust state, the speed as calibrated airspeed or Mach number, the altitude, the path angle, the rate of climb, the fuel rate and state, etc.

2 The Platform Model

The basis for the Platform model is FFA-APP [Hasselrot *et al.*, 1987], from which a subset for the aircraft and its input has been re-used. In this document the modelling and data handling are already well described. This model is written in FORTRAN but is used in C++ translated form. An extension, written in C++, of this model covers flight-mechanical relations, which are created to facilitate writing flight programs.

This chapter at first gives an overall view of the forces acting on the aircraft. The frames of reference are also briefly introduced. Hereupon more detailed descriptions of the aircraft and atmosphere models follow. Finally, the basic relationships for the flight-mechanics are presented.

2.1 Philosophy and Definitions

The present simulation package is aimed to satisfy the research needs at high system level (no detailed modelling of movements initiated by the control system). It can be seen as appropriate when studying flight paths to represent the aircraft as a mass point, and perform the simulation through numerical integration of accelerations (acc) based on forces acting on the mass point (m) (Newton's second law):

$$force = acc \cdot m$$

The forces are assumed to reach their states without transients (=immediately). Aerodynamic data are supposed to be in trimmed condition (the pitching moment balanced out, with effects of the centre-of-gravity and the elevator angle) and with no side force ("clean" flying). Figure 2 shows the kinds of forces that are most important in three different situations.

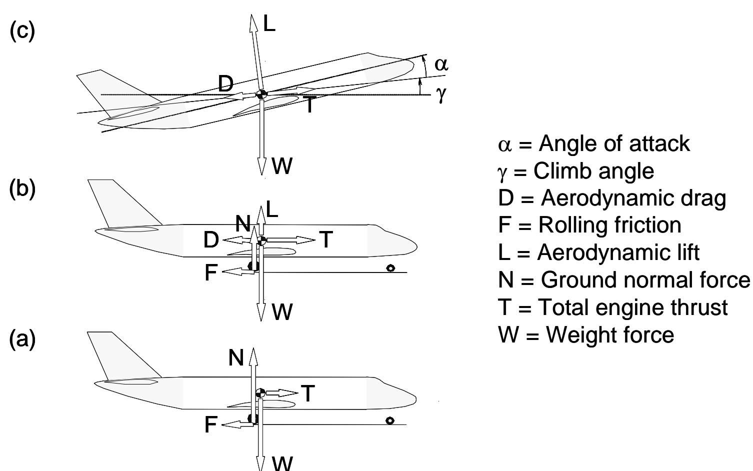


Figure 2. Forces acting on the aircraft in different situations: (a) low speed rolling on ground, (b) high speed rolling (with aerodynamic effects), and (c) airborne flight along path at angle γ (note that the thrust is simplified to be acting in the path direction).

From aerodynamics point of view it is convenient to separate forces into longitudinal (along the path) and transversal (across the path) components, and treat the accelerations separately. To illustrate this, Figures 3(a) and 3(b) show force projections into aircraft symmetry plane and transversal (normal to the speed vector) plane, respectly, for the case of an aircraft performing a climb-turn. Note that the aircraft is banked with its fin in the aerodynamic lift direction (satisfying the assumption of no side force) and is moving sideways along the transversal acceleration force.

Another way of viewing Figure 3(b), where forces are shown, is looking at dimension-less factors as in Figure 4. Here the three forces – the lift force, the gravity force, and the transversal acceleration force – are translated to factors by relating them to the aircraft mass and the gravity acceleration. The loadfactor n_Z acts in the lift direction and is maximized to n_{Zmax}, usually due to the aircraft structure or a pilot related limit.

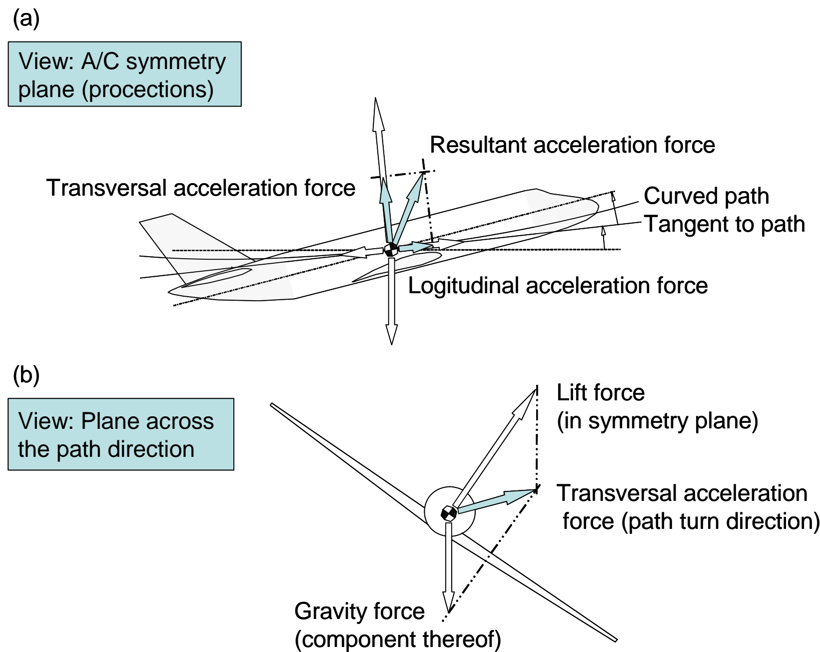


Figure 3. Separation of forces into path related longitudinal and transversal components.

The gravity factor is given by cos(γ). The force factor triangle is valid in any aircraft and path attitude, provided that the lift acts in the fin direction. By the turnplane is meant the plane that is formed when the vertical plane, with the path vector in it, is rotated around this vector by the amount of the angle τ. The actual turn is then, of course, executed in this plane.

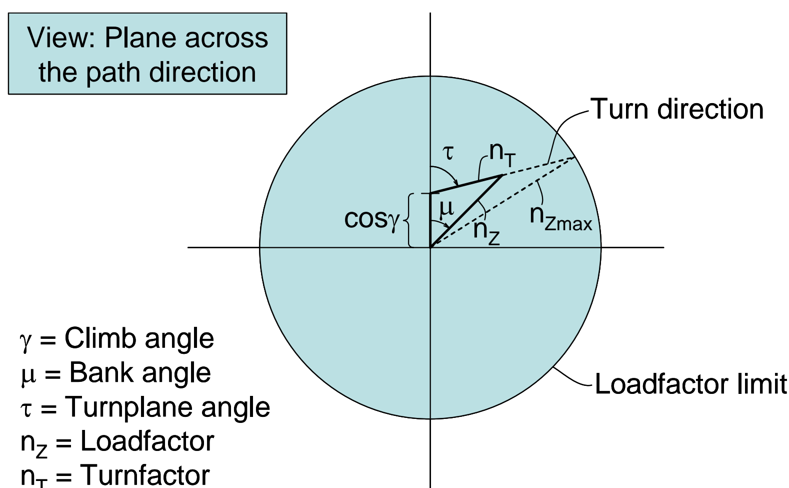


Figure 4. Transversal loadfactor diagram for an aircraft in any path orientation.

Mathematically, the following can be extracted from Figure 4, where n_V and n_H in the equations below are components of n_T. The magnitude of n_T is determined by the path lateral acceleration,

which means that the components are given by the path velocity V, and the path angular rates $\dot{\gamma}$ (vertical) and $\dot{\chi}$ (horizontal), see equations that follow:

Vertical equations

$$n_V = n_T \cdot cos\,\tau$$

$$n_V + cos\,\gamma = n_Z \cdot cos\,\mu$$

$$n_V = \frac{V}{g} \cdot \dot{\gamma}$$

Horizontal equations

$$n_H = n_T \cdot sin\,\tau$$

$$n_H = n_Z \cdot sin\,\mu$$

$$n_H = \frac{V}{g} \cdot \dot{\chi}$$

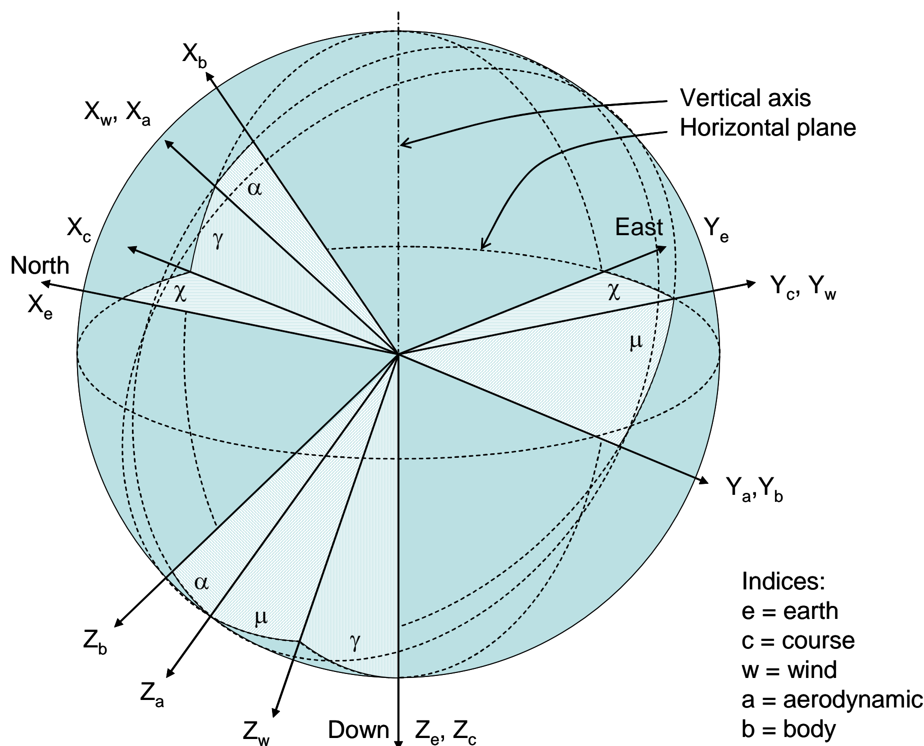


Figure 5. Relationship between reference frames [X_e, Y_e, Z_e], [X_c, Y_c, Z_c], [X_w, Y_w, Z_w], [X_a, Y_a, Z_a], and [X_b, Y_b, Z_b], derived through rotation serially around one axis (X_x, Y_x, or Z_x) at a time. Note that only the [X_e, Y_e, Z_e] and [X_c, Y_c, Z_c] systems are orthogonal.

Finally, there are several coordinate systems to keep track on. Please look at Figure 5. The basic frame of reference is of course the fixed earth system [e], where the origin is determined by its latitude and longitude position, and where the X_e-axis is pointing to the north and the Y_e-axis to the east. The aircraft mass point primarily moves along a path, and hence, the wind-related system [w] is the relevant frame of reference. The orientation of this system relative to the earth system is given by the course (heading) angle χ and the path elevation angle γ. In order to conveniently treat the aircraft aerodynamics, the wind system is rotated around the X_w-axis until the wind vector lies in the X-Z-plane of the new system, i.e. until the bank angle μ is reached. Now the system is an aerodynamic frame of reference [a]. The final system rotation (here) is performed around the Y_a-

axis to yield the angle-of-attack α. This system is now body-fixed [b], where X_b points forward along the aircraft nose and Y_b points to the right in the normal direction of the X_b-Z_b-plane.

In simulations in general, path oriented data are the parameters primarily in view. Sometimes, however, aircraft attitudes govern the simulation. In civil applications this often happens during takeoff and landing, where the nose attitude relative to the air field may be important to check. In military simulations aiming at a target is such a situation. The aircraft attitude, i.e. that of the engine, is also important, when noise effects of the engine jet are studied. The aircraft attitude is measured as in the case with the path (the corresponding name in parenthesis): ψ (χ) and θ (γ). For these two parameters the following expressions are given, using angle-of-attack α and bank angle μ, see also Figure 4:

$$\psi = \chi + \alpha \cdot \sin \mu$$
$$\theta = \gamma + \alpha \cdot \cos \mu$$

2.2 The Aircraft

The aircraft description is separated into three parts: aerodynamics, engine characteristics, and configuration data.

2.2.1 Aerodynamics

The following presentation is mainly based on information found in Hasselrot *et al.* (1987).

The aerodynamic tables are based on a coefficient modelling that allows a relatively accurate description of realistic characteristics. The lift curve can be composed by piecewise linear segments. The drag polar can have a non-parabolic behaviour by means of a lift-depending correction of the parabola factor. In this context it is possible to use an arbitrary parameter to account for trim drag changes due to different centre-of-gravity positions or other effects. The maximum lift is specified separately from the lift slope definition.

A new feature, already introduced in later versions of PcP [Stehlin, 1999], is the raising of the whole lift curve, to represent the effect of wing flaps, by using an angle-of-attack change.

The aerodynamic model is thus governed by the following, where *M* and *H* are Mach number and altitude, respectively:

$$\underline{Minimum\ drag\ and\ altitude\ and\ load-configuration\ increments}$$

$$C_{D\,min} = f(M)$$
$$\Delta C_{DH} = f(M, H)$$
$$\Delta C_{Ditem} = f(M, item)$$

$$\underline{Induced\ drag}$$

$$C_{Di} = K \cdot (C_L - C_L^*)^2$$
$$K = Parabola\ factor$$
$$K = K' + \Delta K$$
$$K' = C_{Di} / (C_L - C_L^*)^2 = f(M, C_L)$$
$$\Delta K = \Delta C_{Di} / (C_L - C_L^*)^2 = f(M, arbitrary\ parameter)$$
$$C_L^* = f(M)$$

Total drag

$$C_D = C_{D\,min} + \Delta C_{DH} + C_{Di} + \sum \Delta C_{D\,item} = f(M, H, C_L, item)$$

Angle of attack with increment and maximimum lift

$$\alpha = f(M, C_L, item)$$

$$\Delta\alpha = f(item)$$

$$C_{L\,max} = f(M, H, item)$$

The term *item* refers to an identifiable factor that affects the aircraft lift and drag. It can be interpreted as an external military load, such as a set-up of missiles, bombs, etc.. It can also be a configuration, such as a flap setting.

2.2.2 Engine Characteristics

The format description is found in Hasselrot *et al.* (1987).

The tables for the engine characteristics simply hold pre-computed data for a chosen atmosphere model, in the form of thrust and fuel rate as a function of several parameters: *throttle* level (in a scale chosen by the user), basic operating condition (*A*, reheating/afterburning or not), temperature adjustment ΔT (for the standard atmosphere), altitude *H*, and Mach number.

Mathematically this can be expressed as:

$$T = f(throttle, A, \Delta T, H, M, scalefactor1)$$

$$\dot{m}_{fuel} = f(throttle, A, \Delta T, H, M, scalefactor2)$$

It should be noted that *T* and $\dot{m}_{fuel}$ for each given parameter state (combination of *throttle, A, ΔT, H, and M*) can be scaled individually (1 is default for *scalefactor1* and *scalefactor2*). It is also possible to set a global *scale factor* for the engine installation, and this function is available programmatically. Hence, there is a means for adjusting with regard to installation effects on *T* and $\dot{m}_{fuel}$.

2.2.3 Other Aircraft Characteristics

In order to complete the specification for an aircraft, additional parameters are needed: reference area (*S*) and basic components for computing aircraft mass (*m*), such as operating empty mass, maximum fuel mass, maximum payload/external load mass, and maximum takeoff mass.

Although the basic input system in the Platform module can handle *S* and some rudimentary mass specifications, the new PlatformX has an extended treatment for this. Here multiple configurations can be defined, i.e. be given names, be coupled to the *items* (specified as a number) with the aerodynamic characteristics and to a mass value. Changing the aircraft configuration will then be performed by giving its name. The initial fuel state is set during the definition of the flight plan. The actual fuel mass state is derived from the fuel rate due to the engine characteristics by performing a numerical integration.

Mathematically the mass build-up can be expressed as:

$$m = m_{op.empty} + m_{payload} + m_{ext.load} + m_{fuel}$$

$$m_{fuel} = f(m_{init.fuel}, \dot{m}_{fuel}, time)$$

2.3 The Atmosphere

The atmosphere properties play a central role for the characteristics of both the engine thrust and the aerodynamic forces. These vary with the location on earth, the season viewed, and even on day-to-day basis. To bring an order of this, standardized atmosphere models are used. The Platform package comes with a model that is based on an old ICAO standard (ICAO, 1954). Later on, the standard has been extended, mainly with data for higher altitudes. This work has been pursued by both International Civil Aviation Organization (ICAO) and U.S. Committee on Extension to the Standard Atmoshpere (COESA, established 1953, published standards 1958, 1962, and 1976).

To what degree is the ICAO (1954) standard, modelled in Platform, compatible with the U.S. (1976) standard? Based on what our model produces and on table lookups in the 1976 standard, the following observations can be made:

1. The pressure and temperature data are identical for altitudes up to 20 km.
2. The pressures continue to be compatible for altitudes up to 32 km
3. The temperature profiles differ: (a) In the ICAO (1954) standard, the tropopause continues until 25 km, whereupon a temperature rise starts with a rate of 3 degrees per km. (b) In the U.S. (1976) standard, the temperature starts rising by 1 degree per km until 25 km and then by 2.8 degrees per km until 32 km.

The atmosphere model in Platform has a modifying parameter in the form of temperature adjustment ΔT . The model functionality is as follows:

$$\delta = Pressurefactor = f(H)$$

$$T = Absolute\ temperature = f(H, \Delta T)$$

$$p = Staticpressure = const1 \cdot \delta$$

$$\rho = Air\ density = const2 \cdot \delta / T$$

$$a = Velocity\ of\ sound = \sqrt{\kappa \cdot R \cdot T}$$

$$\kappa = Ratio\ of\ specific\ heats$$

2.4 Flight-Mechanics

Basic conditions

Moving in the air results in aerodynamic forces, lift *L* and drag *D*, which grow roughly with square of the speed *V* along the path, and inversely with the air density ρ. For this reason the dynamic pressure, together with the reference area *S*, is generally used in conjunction with lift and drag coefficients, C_L and C_D, respectively. The coefficients are determined by the aircraft modelling, see the description below.

$$S = Wing\ area\ (arbitrary, but\ usually\ formed\ by\ the\ external\ projected\ area$$
$$plus\ the\ part\ extended\ to\ the\ vertical\ symmetry\ plane)$$

$$q = Dynamic\ pressure = .5 \cdot \rho \cdot V^2 = .7 \cdot p \cdot M^2$$

$$V = Path\ velocity$$

$$M = Mach\ number = V \cdot a$$

$$L = C_L \cdot q \cdot S$$

$$D = C_D \cdot q \cdot S$$

Conditions while airborne

The magnitude of L, which acts at right angle to the path, depends on the wing loadfactor n_Z and the thrust component in the L direction (lift to weight and thrust components):

$$L = n_Z \cdot m \cdot g - T \cdot sin\alpha$$

where the angle-of-attack α is given by the aircraft lifting characteristics, see section 2.2.1. The thrust term is usually small when compared to L and thus is often neglected. The n_Z, in its turn, is linked to the gravity factor $n_G = cos\gamma$ and the turnfactor n_T in the way that is shown in Figure 4 and equations in section 2.1.

The longitudinal effect (in the velocity V direction) of L is the aerodynamic drag D, which is determined through the modelling of the C_D, see below, and equations in section 2.2.1. The longitudinal acceleration acc is given by the excess thrust in this direction:

$$m \cdot acc = T \cdot cos\,\alpha - D - m \cdot g \cdot sin\,\gamma$$

Conditions on ground

During ground-run the force relationship changes, which is shown in Figure 2. The following equations are then valid, where the angle-of-attack is assumed to govern the aerodynamic lift and drag and μ_G is the ground (rolling) friction coefficient:

$$L = f(M, \alpha)$$
$$D = g(M, L)$$
$$F = Ground\ \ friction = \mu_G \cdot (m \cdot g - L) \cdot cos\,\gamma, \quad where\ L \leq m \cdot g$$
$$m \cdot acc = T \cdot cos\,\alpha - D - m \cdot g \cdot sin\,\gamma - F$$

Flight or movement on the ground always implies variations (continuous or discontinuous) of state parameters, such as:

$$T, acc, \gamma, \alpha, \theta, CAS, M, \dot{H}, \mu_G,$$

where CAS is the Calibrated Air Speed (=Indicated Airspeed when instrument and place errors are zero. From here on IAS, whenever it occurs in the text and the code, will be treated as CAS.

These parameters are interdepending, which means that only a few can be chosen as control parameters, leaving the others in depending state. For the airborne case, it is possible to specify conditions for two parameters. For the ground case four are chosen. For example, a climb, or more commonly an approach, may be required to follow a certain path angle γ. Then there is still room for making a requirement on another parameter, in our example the thrust level T (e.g. maximum) or the longitudinal acceleration acc (e.g. zero=constant speed), or one other parameter in the above list. After the state of, in all, two parameters have been given their values, all other parameters are functions of these two. This constitutes a flight condition. As there are many possible flight parameter combinations, there are theoretically also many flight conditions. Table 1 presents several, not all, flight conditions that may be envisioned. Many of these have been implemented in Platform as case alternatives in the C++ method 'flight_cond' during the work with the present flight application (reported here). The last two lines show four chosen parameters, which may seem illogical. However, this applies for the ground case, when two additional degrees-of-freedom are introduced and hence another two parameters must be locked.

Table 1. Useful flight conditions

Required locked states	Case name in flight_cond (C++ code)	Implemented	Usage in Flight application
throttle, acc	Pt_acc	x	Cruise-Climb
throttle, γ	Pt_gamma	x	Acceleration
throttle, α	Pt_alpha		
throttle, θ	Pt_theta		
throttle, IAS	Pt_ias	x	Climb, descent
throttle, M	Pt_mach	x	Climb, descent
γ, *IAS*	Gamma_ias	x	Approach
γ, *M*	Gamma_mach	x	Approach
acc, γ	Acc_gamma	x	Cruise
acc, α	Acc_alpha		
acc, θ	Acc_theta		
$\dot{H}$, *IAS*	Hr_ias	x	Rate-limited descent
$\dot{H}$, *M*	Hr_mach	x	Rate-limited descent
throttle, μ_G, γ, α	Pt_mu_gamma_alpha	x	Taxi, start, land (ground acceleration/retardation)
acc, μ_G, γ, α	Acc_mu_gamma_alpha	x	Taxi

The difference between the terms ‘descent’ and ‘approach’ is only technical, both being useable at all altitudes: the former is associated with prescribed *throttle*, and the latter with prescribed climb angle (γ).

The present development state of Comsim-Platform reflects the focus on civil applications, where the application development has been performed along two lines: the first to define a vertical flight profile, and the second to flex this profile along a prescribed projected pathway (on a map). The whole flight profile is executed by going through the various flight-conditions, each of which delivering a longitudinal acceleration (*acc*) value for the numerical integration. This results in a longitudinal velocity (V) value. As the climb angle (γ) is always either a prescribed value or an answer of the flight-condition, the following rates for the altitude and horizontal distance, $\dot{H}$ and $\dot{S}$, respectively, are established for the flight profile state:

$$\dot{H} = V \cdot \sin\gamma$$
$$\dot{S} = V \cdot \cos\gamma$$

for which the numerical integrations are performed, thus resulting in new H and S states. The S value is then used for establishing the actual X and Y positions on the map. This is possible, as the prescribed path projection is a purely a geometrical definition. It should be noted that flexing the vertical flight profile implies turnfactor effects, which is easily implemented by means of the path geometrical curvature and speed in the horizontal plane. Another note is the effect due to the fuel flow $\dot{m}_{fuel}$, which primarily acts on the aircraft mass m. Of course fuel flow is also numerically integrated into an actual state, which also affects the total aircraft mass, see below.

For the future development, the performance-related military applications require additional, not yet defined flight conditions, which may even need new state (controlling) parameters. It is easy to predict loadfactor- or turnfactor-oriented flight-conditions, and their combinations with the already present flight parameters.

3 The Navigation Module

Navigation capability is important in many FOI tasks for computing distributed engine emissions and noise pollution. Also, the flight-recordings used for the comparisons, see chapter 6, contain path coordinates. Inclusion of navigation aspects leads to a better validation process. Therefore the navigation module was created.

The navigation module is in principle a completely isolated code, with its own input (hence it can be used in contexts other than this). Its contact with flight environment is through the *S* variable, see section 2.4, via the argument list.

3.1 Path Modelling

The basis for the navigation is letting the projection of the flight path follow a prescribed map path, and assuming the path be defined as straight lines and circular arcs. The input then consists in specifying the corner-points (*latitude, longitude*) of a polygon-train and radii (*r*) to round-off the corners:

$$(lat, long, r)_1, (lat, long, r)_2, \ ..., (lat, long, r)_N$$

Before these data are analyzed into alternating lines (shorter than the corresponding polygon elements) and arcs, the *lat* and *long* are translated into metric coordinates of *X* (positive northwards) and *Y* (positive eastwards), using the following:

$$X = X_0 + (lat - lat_0) \cdot K_{\deg-m}$$
$$Y = Y_0 + (long - long_0) \cdot \sqrt{\cos(lat) \cdot \cos(lat_0)} \cdot K_{\deg-m}$$
$$K_{\deg-m} = 10000000/90 \, [m/^\circ]$$

where the index 0 refers to an origin along the map path, at the start or locally defined.

An idea of a typical map path is shown in section 5.2. In the navigation module, the distances of the lines and the arcs are noted. To identify what line or arc is being processed during the actual flight simulation, the accumulating sum of these distances is compared with the simulated distance (*S*). With the segment identified, it is possible to compute the actual metric position as a function of *S*:

$$X = f(S)$$
$$Y = g(S)$$

Of course, these values can be converted back to *lat* and *long*, using the above expressions.

In the navigation module there is a variant of the “straight” segment in the path definition, which is intended for long distances due to Earth curvature. In this context “straight” is the shortest distance on the surface, the so-called the great-circle distance. This is treated in the module mainly by means of translating *lat* and *long* position into Earth centre angle movement, and from there to accumulated distance using the value of Earth radius.

4 The Simulation Package (Information for the Software Developer)

Comsim [Aronsson, 1991] has been chosen to form one of main pillars (the other being the Platform model) for the new flight simulation system, mainly due to its capability of handling continuous simulation (based of derivatives) and discontinuous event management. In addition it is considered in this context a merit that it is written in C++. It is, in fact, a relatively direct translation of parts in NYCOND [Righard, 1981], written in SIMULA, commonly regarded as the first object-oriented language (no longer being maintained).

Comsim is well described in Aronsson (1990), but as the present report is intended to mediate the experiences of using the package, a general overview is given about its structure and usage (this chapter) before proceeding to the Comsim application (next chapter).

4.1 The Comsim Structure

Comsim contains classes to enable handling of *continuous* simulation, i.e. numerically integrating conditions for user defined *variable* set, and treatment of *process* events, which may be user-defined (in the application). A typical event is reporting variable states, for which Comsim provides a class *reporter* to aid the application writer. All events imply stopping and re-starting the simulation.

This chapter is primarily intended for those who need an in-depth knowledge of the workings of the package. The reader may jump directly to 4.2, where application-oriented aspects are presented.

The more in-depth interested reader can also go to Aronsson (1991) for the complete theory and the implementation notes.

In the following, please note that words in *italics* are reserved words in C++ or concepts in Comsim. The main concepts are described under sections below. Note also that these sections have form of *class* definitions, the first being simple, and the others shown as child – parent relationship. The parent part is indicated by *': public'*.

Concepts, marked as italics:
virtual: a method defined within a class can have this prefix, say:
class parent { virtual void method_a(); … }
then it is possible to redefine this method without changing its name:
class child : public parent { void method_a() { <own actions> }; }

4.1.1 Class cslink

During simulation many objects are handled dynamically in various levels. For example, *variables* are such objects that need updating in the integration calculation. These objects need to have ability to be placed into some queue, and later to be picked one after another for the appropriate treatment. Comsim therefore includes definitions for the class *cslink (*renamed from *link* due to name clash in g++ compilation, where 'cs' stands for Comsim), with facilities for queue handling. This class is intended mainly for internal (but also external) usage: direct object declaration, or as the parent of subclasses

4.1.2 Class objectattribute : public cslink

Comsim has many kinds of dynamic objects that, apart from having queueing capability, also need to be started and stopped. This occurs for example, when a *reporter* at intervals needs to check *variable* states (i.e. for an interpolation). Therefore a class *objectattribute*, based on the

class *cslink*, has been provided in Comsim, where the following as *virtual* defined methods are included: *start(), stop()*.

4.1.3 Class variable : public objectattribute

Class *variable* is a subclass to class *objectattribute*, with additional storage attributes to hold intermediate state conditions for the numerical integration. Apart from starting and stopping the handling for the *variable* object (using the methods *start()* and *stop()*), queueing methods (*intoobject()* and *outofobject()*) are available to place the objects into an environment instance of the class *object*.

There also exists an extension of class *variable*: *vectorvariable*, which is based on three-dimensional vectors. Each element is handled as a *variable*. In the Comsim package there is a fairly complete set of class definitions for vector and matrix (up to three dimensions) and their operation methods.

4.1.4 Class continuous : public objectattribute

Class *continuous* is subclass to class *objectattribute*, with additional facilities for handling *variables* and their numerical integration.

It is normally used as *parent* for *subclass* extension, from which a *continuous* simulation object is created. From here (or externally) *variable* objects are created, queued here and continuously managed. The place within this class where in principle the rates are computed is: *virtual compute()*. However, this method is empty. The idea is then letting the programmer/user redefine the method, to contain the proper derivative computing, in a subclass of *continuous*.

The numerical integration of all *variable* objects is performed by *integrator*, where objects are accessed from a common environment instantiation of the class *object*. For this the methods are available: *intoobject()* and *outofobject()*.

4.1.5 Class reporter : public objectattribute

Class *reporter* is a subclass to class *objectattribute*, with additional methods for handling reports. Four types of *reporters* can be specified: *eventrep, time_steprep, timerep, time_eventrep*, i.e. based on repetition of time, event, step, or a combination thereof. Apart from methods for time and frequency handling and queueing, the following re-definable methods are handled:

```
virtual void prelude() {;}
virtual void start();
virtual void stop();
virtual void compute() {;}
virtual void endcompute() {;}
```

The programmer/user must define a subclass of *reporter*, in order to compose a report layout, for which the methods *prelude*, *compute*, and *endcompute* may be modified. Note that special computing can be performed from here on the *variable* objects (provided that access is granted).

4.1.6 Class object : public objectattribute

Class *object* is a subclass to class *objectattribute*, with additional facilities for handling objects of the following classes: *variable*, *vectorvariable*, *continuous*, and *reporter*. It manages queues for these objects, and also has a time setting method. It is in fact a basis for a complete simulation environment.

The programmer/user has the responsibility of inserting the different kinds of objects into their respective queues, by means of the method: *intoobject()*, defined in the respective classes. The reverse is of course: *outofobject()*.

4.1.7 Class process : public cslink

Class *process* is a subclass to class *cslink*, with additional facilities for activating *process* objects and *eventnotice* handling. The following methods are available:

```
friend void _diractivate( bool, process* );
friend void _befactivate( bool, process*, bool, process* );
friend void _delactivate( bool, process*, double, bool);
friend void _atactivate( bool, process*, double, bool);
friend void _passivate();
friend void _wait( head* );
friend void _cancel( process* );
```

4.1.8 Class eventnotice : public cslink

Class *eventnotice* is a subclass to class *cslink*. An instance of this class is created whenever a new object of the class *process* (or a subclass of it) is instantiated, i.e. this event is placed into the time sequence (schedule) by means of the method: *intosqs()*. The class eventnotice also allows re-scheduling, or in other words control where to place the event on the time axis. This is controlled by a number of external methods, as the following:

```
friend void runsimulation();
friend void _diractivate( bool,process* );
friend void _befactivate( bool,process*, bool, process* );
friend void _delactivate( bool,process*, double, bool );
friend void _atactivate( bool, process*, double, bool );
friend void _passivate();
friend void _hold( double );
friend void _wait( head* );
friend void _cancel( process* );
```

4.1.9 Class monitor : public process

Class *monitor* is a subclass to class *process*, with storage attributes to hold *time* status and coming *time* and *process events* for the process of the *integrator* and the *reporter* handling, and for the internal management of *continuous* (*variable*) objects. The following methods are available/planned (modelled after SIMULA application, see Righard):

void compute_rates();	//Executes all objects' *Continuous->compute*
virtual void simplestep() {;}	//Takes an integration-step with the Euler-, Simpson-, fixAdams- or Trapez method.
virtual void stiffstep() {;}	//Takes an integration-step with Gear's method.
virtual void adamsstep() {;}	//Takes an integration-step with Adam's method.
virtual void rkstep() {;}	//Takes an integration-step with Runge-Kutta-England's method.
virtual void stiffini() {;}	//Initiates for *stiffstep()*.
virtual void adamsini() {;}	//Initiates for *adamsstep()*.
virtual void seterrcoeff() {;}	//Assigns the errortest-coefficient-matrix values used in *stiffstep()*.
virtual void interpolate() {;}	//Interpolates variable-states at *lasttime* + *dt*.
virtual void coset_interpolate() {;}	//Interpolates variable-states at *lasttime* + *dt* and calculates the interpolation-coefficients.
virtual void happening();	//Contains the code for the simulation-control.

where only the first one has the functionality defined: usage of the *continuous* method *compute()*. This constitutes the *monitor* functionality. The rest of the methods are envisioned integration methods, see Righard (1981), of which some are implemented in class *integrator*.

This class accesses an appropriate instantiation of the class *object*, where the *variable* objects are stored.

It is in this place both *continuous* (based on derivatives) and *process* (based on happening) objects are managed in the same dynamic context, i.e. a combined simulation is executed. (Hence the name of the simulation package: Comsim.)

4.1.10 Class integrator : public monitor

Class *integrator* is a subclass to class *monitor*, with some of the envisioned integration methods implemented. These are:

void simplestep();
void interpolate();
void coset_interpolate();

Presently, this class is the basis (the only one, in fact) for creating a *monitor* object for the integration process. If additional methods are to be implemented, then a new class must be defined, using ': public integrator' to retain the earlier implementations.

4.1.11 Combined Simulation Module, Comsim

This module is the final product that contains all references to the package elements. A simulation application must always include Comsim, in order to have access to all the functionality. The module has two functions defined:

Initcombinedsimulation();	//Calls *initsimulation()* and creates an instance of the *class Integrator* (*theMonitor*), which is activated instantaneously.
Runcombinedsimulation();	//Ractivates *theMonitor* instantaneously because the user might have activated some processes before the simulation has started. *TheMonitor* must be the first event in SQS (on the time axis) when the simulation starts. Finally, *runsimulation()* is started.

4.2 Comsim Usage

The usage of the Comsim structure in a simulation program is considerably simpler than it may seem by looking at the Comsim structure, as described above. How it would appear in an application is best shown by an example, which is an extract of the application program described in chapter 5. We will follow a step by step build-up of the program (each step under its own header).

The example is based on the availability of Comsim, with the structure as described above, and of the class PlatformX, whose software structure is not described here.

Using Comsim requires that subclass extensions of the following classes are defined by the programmer: *continuous*, *process*, and *reporter*. In our example, these are:

4.2.1 Class Flight : public continuous

Here our problem-oriented *variables* are declared and initialized. In order to compute their *rates* (derivatives), the *continuous* method *compute* has to be redefined. All this is shown in List 1 and List 2.

List 1. Sketch of Flight.h file

```
#ifndef _FLIGHT_H
#define _FLIGHT_H

#include "comsim.h"
#include "platformx.h"

class Flight : public continuous
{
        friend class Pilot; //to let objects of Pilot have access to Flight attributes and methods
        friend class Report; //to let objects of Report have access to Flight attributes and methods

//Flight attributes
        PlatformX *thisPlatform; //to hold actual aircraft data
        object *thisObject; //thisObject is the simulation environment
        variable *v, *s, *h, *fc; //variable declarations (all that are needed!; explained in the
               flight.cpp scetch)
        float g=9.80665, acc_, ga_, al_, chi_, bankangle_, psi_, th,  x_, y_, x0_, y0_, r;
        ...
//Flight methods
        void FlightCondition(); //entry point to the following flight segments
        void Setstate( char *config, int no, float takeofffuel, float fc, float dist, float h,
                float v, float vi, float mach );
        void Calibrate( char *config, float takeoff_fuelmass, float pt, float paftt, float alt,
                float vel, float ias, float mach );
        void Taxi( char *config, float pt, float dist, float dur, float iaslim, float mu,
                float acc, float ga, float al );
        void Start( char *config, float pt, float paftt, float iaslim, float mu, float ga,
                float al, float cllim );
        void Climb( char *config, float pt, float paftt, float hlim, float hrlim, float dur,
                float machlim, float iaslim, float acc, float ga, float cllim );
        void Accelerate( char *config, float pt, float paftt, float dist, float dur,
                float machlim, float iaslim, float ga, float cllim );
        void Cruise( char *config, float pt, float paftt, float dist, float dur, float machlim,
                float iaslim, float acc, float ga, float cl_max );
        void Descend( char *config, float pt, float hlim, float hrlim, float dur, float machlim,
                float iaslim, float cl_max );
        void Approach( char *config, float hlim, float acc, float machlim, float iaslim, float ga,
                float cllim );
        void Land( char *config, float pt, float iaslim, float mu, float ga, float al );
        ...
        void compute(); //this is the prototype for the redefined method (of the one
               in class continuous)
     public:
        void readFlightInstr(char *flightFile); //must be public, as it is called in Main
};
#endif //_FLIGHT_H
```

List 2. Scetch of Flight.cpp file

```
#include "flight.h"

...//Method implenentations for the class Flight
//The implementations for FlightCond, Fuel, Taxi, Start, Rotate, Climbout, Flapsin, Climb,
//Cruise, Descend, Flapsout, Approach, Land, and Unspecified are placed here.

void Flight::vars_intoobject(object *refObject) //this method must be called before starting
                                                //the Flight simulation, see main() program
{ //create the variables and place them into the environment (thisObject)
        thisObject = refObject;
        v = new variable(0);        //to hold the velocity value
        v->intoobject(thisObject);
        s = new variable(0);        //to hold the horizontal distance along the path from the start
        s->intoobject(thisObject);
        h = new variable(0);        //to hold the altitude value
        h->intoobject(thisObject);
        fc = new variable(0);       //to hold the amount of fuel consumption
        fc->intoobject(thisObject);
}

void Flight::compute() //the implementation of the redfined method
{
        //Save the previous state values:
        float x0_ = x_;
        float y0_ = y_;
        float chi0_ = chi_;
        float time0 = time_;

        //Establish the current flight condition,
        //and compute the resulting longitudinal acceleration (acc_):
        FlightCondition(); //delivers acc_ value

        //Set the speed rate along the path (to be numerically integrated):
        v->rate = acc_;

        //Set the horizontal speed (to be numerically integrated):
        s->rate = v->state * cos( ga_ );

        //Set the altitude rate (to be numerically integrated):
        h->rate = v->state * sin( ga_ );

        //Set the rate of the fuel consumption (to be numerically integrated):
        fc->rate = ff_;
        fuelmass_ = takeoff_fuelmass_ - fc->state;

        //Check the loadfactor requirement:
        nz_in = thisNavigation->get_loadfactor();
        <Here the nz_in value is checked against the maximum loadfactor value that is allowed
        for thisPlatform, which may be limited by aerodynamics or structure.
        The structural reference data is found through thisPlatform->maxloadfactor, while
        the aerodynamically related value is computed by means of thisPlatform->lift_and_drag.
        After nz_in has been adjusted, the corresponding turn radius, r_, is computed.>

        //Let thisNavigation compute positional data based on the accumulated horizontal distance:
        thisNavigation->handle_pathsegment( s->state, r_ ); //note the radius input!

        //Extract the positional data including the actual heading course, chi_:
        thisNavigation->get_pos_data( la_, lo_, x_, y_, chi_ );

        chi_rate_ = ( chi_ - chi0_ ) / ( time_ - time0_ );
        turnfactor_ = sqrt( nz_in * nz_in - cos( ga_ ) * cos( ga_ ) );
        loadfactor_ = sqrt( turnfactor_ * turnfactor_ + cos( ga_ ) * cos( ga_ ) );
        bankangle_ = asin( turnfactor_ / loadfactor_ );
        th_ = ga_ + al_ * cos( bankangle_ );
        psi_ = chi_ + al_ * sin( bankangle_ );

        //Check the break condition:
        <In this prototype application of ComsimPlatform, the check for end of simulation is
        performed here. This is not in the spirit of the Comsim package (=clumsy)! This should
        be implemented in a (class) process object, for example our Pilot. This remains to be
        done.>
}
```

4.2.2 Class Pilot : public process

Here the discrete-oriented events are defined. The idea of our envisioned example is letting the discrete happenings be governed by the class Pilot (as the pilot naturally has this role), but for the time being this has not been implemented yet. The conditions for the happenings are specified in the redefined *process* method *happening()*. To give an example of happening, the total total simulation time has been specified in List 4 (the *hold* instruction). The simulation can also be controlled by *variable* states. As in our example *variable* objects are declared in the *continuous* class Flight, and in order for Pilot to have access to these, a reference to the Flight object has to be transferred to thePilot, see how this is done in List 3 and List 4.

List 3. Pilot.h

```
#ifndef _PILOT_H
#define _PILOT_H

#include "comsim.h"
#include "flight.h"

class Pilot : public process
{
 private:
       Flight *thisFlight;
 public:
       Pilot(Flight *refFlight)
       {
              thisFlight = refFlight;
       }
       void happening();
};

#endif //_PILOT_H
```

List 4. Pilot.cpp

```
#include "pilot.h"

void Pilot::happening()
{
 thisFlight->simtime = tm_time();
 switch (_pos)
 {
  case 1:
/*    dtMin = 0.00001; dtMax = 100;
    maxAbsError = maxRelError = 0.00001;*/
    hold(36000);
  case 2:
  default: ;
 }

 lastline: ;
}
```

4.2.3 Class Report : public reporter

As variable objects are defined as problem-oriented, their presentations must also be handled accordingly. In our example, Report contains the problem-specific output, see Lists 5 and 6.

4.2.4 Module Integration

Having defined the necessary extensions to the Comsim *classes*: *continuous*, *process*, and *reporter*, it is time to bring the components together. As in all programs, the main program is where all things collected and controlled. Let us see how this is done in our example, using Comsim, Platform, and Flight, please look at List 7.

List 5. report.h file

```
#ifndef __REPORT_H
#define __REPORT_H

#include "comsim.h"
#include "flight.h"
#include <fstream.h>
#include <iomanip.h>

class report : public reporter
{
 private:// double cyclus;
        Flight *thisFlight;
        ofstream out;
 public:
        report(Flight *refFlight, char *resultFile, reportertype r = timerep, double f = 0.1) :
reporter(r,f)
        {
                thisFlight = refFlight;//     ofstream out;
                out.open( resultFile );
                out << "\ntime[s] F[kN] Frel00 V[m/s] X[km] Y[km] H[km] Psi[deg] Theta[deg]\n" ;
        }
        void compute();
};


#endif __REPORT_H
```

List 6. report.cpp file

```
#include "report.h"
#include <stdio.h>

float simtime;

void report::compute()
{
        float kN = .001;
        float km = .001;
        float thrust(float alt, float mach, float pt, float paftt, integer lim ); //primarily to
be used externally
        simtime = tm_time(); //
        float maxthr00 = thisFlight->thrust( 0., 0., 1., 0., 0. );
        float relthr00 = thisFlight->thr_ / maxthr00;
        out << "\n" << simtime << " " << setprecision(4) << thisFlight->thr_*kN << " "
            << relthr00 << " " << thisFlight->v->state << " " << thisFlight->x_*km << " "
            << thisFlight->y_*km << " " << thisFlight->h->state*km << " "
            << thisFlight->psi_*deg << " " << thisFlight->th_*deg << endl;
}
```

By following the explanations (after "//") in List 7, the reader should be able to identify the important elements, already presented above. Examples of such are the objects of the classes PlatformX, Flight, Pilot, Report, and object (= theEnvironment). Also, how they, including the variable objects, are introduced into theEnvironment, and how to specify the file transfers to or from the different objects. However, the most important is including Comsim into the main program, which is done with *#include "comsim.h"* at the beginning of *main()*. This file and its implementation file are shown in List 8 and List 9, respectively.

These files contain two routines (not formally methods, as they do not belong to a class): initcombinedsimulation() and runcombinedsimulation(). The first one creates an instace of the class integrator, theMonitor, and sets up the same in inactive state. The second one activates theMonitor and starts simulation. As expected the two routines are called in main(), in beginning and at the end, respectively, see List 7.

List 7. C++ main program

```
#include "comsim.h"//makes available Comsim package
#include "flight.h"//makes available extension of Comsim class continuous
#include "pilot.h"//makes available extension of Comsim class process
#include "report.h"// makes available extension of Comsim class reporter


int main(int argc, char* argv[])
{
    Platform *thePlatform = new Platform; //create a Platform object
    char aircraft[20];
    cout << "\nGive name of an aircraft type: "; //specify a specific aircraft (frame and engine)
    cin >> aircraft;
    thePlatform->readPlatformData(aircraft); //read aircraft data to thePlatform

    initcombinedsimulation(); //create an integrator object (theMonitor), see #include "comsim.h"

    Flight *theFlight = new Flight(thePlatform); //create a Flight object and refer to thePlatform
    char flightprofile[20];
    cout << "\nGive name of a flight profile: "; //specify a filename to flight profile data
    cin >> flightprofile;
    theFlight->readFlightInstr(flightprofile); //read flight profile data to theFlight

    Pilot *thePilot = new Pilot(theFlight);
    _diractivate(false,thePilot);

    fixadams = true; //choose an implemented method for numerical integration, here fixaddams

    dtmin = 0.00001; dtmax = .1 //set conditions for the integration
    maxabserror = maxrelerror = 0.00001;

    object *theEnvironment = new object;
           //create an common environment where all simulation objects
           //are stored (variable, Flight, Report etc.)
    theFlight->intoobject(theEnvironment);
    theFlight->vars_intoobject(theEnvironment);
    theFlight->gotoFirstInstr();

    char resultfile[20];
    cout << "\nGive name of a result file: "; //specify a filename for storing result data
    cin >> resultfile;
    Report * theReport = new Report(theFlight, resultfile );
    //create a report object, with reference
                                                  //to theFlight and the filename
    theReport->setfrequency( timerep, 1 ); //set reporter conditions
    theReport ->intoobject(theEnvironment);

    theEnvironment->start(); //
    runcombinedsimulation(); //activate theMonitor and start the simulation,
                             //see #include "comsim.h"
    return 0;
}
```

List 8. Comsim.h

```
#ifndef __COMSIM_H
#define __COMSIM_H

// This file is to be included in continuous-
// and discrete-time event-driven simulations,
// i.e. combinedsimulations.
// By Johan Aronsson. Revision 1990-02-06.
// © Johan Aronsson.
// File comsim.h.

#include "integrator.h"

void initcombinedsimulation();
void runcombinedsimulation();

#endif __COMSIM_H
```

List 9. Comsim.cpp

```
// By Johan Aronsson. Revision 1990-02-06.
// © Johan Aronsson.
// File comsim.cpp.

#include "comsim.h"

void initcombinedsimulation()
{
 initsimulation();
 themonitor = new integrator;
 _diractivate( false,  themonitor );
}

void runcombinedsimulation()
{
 _diractivate( true, themonitor );
 runsimulation();
}
```

4.3 Comsim Flow

When initiated the simulation stays within *runcombinedsimulation()*, i.e. in the routine *runsimulation()*, until it exits itself due to pre-defined conditions in a *continuous* object, a *process* object, or a *reporter* object. In our application program, this would theoretically imply any of the following class objects: *theFlight*, *thePilot*, or *theReport*. Actually, we have let *theFlight* take care of the intelligent program flow and the final exit condition.

The running simulation is completely controlled by Comsim methods, which may be redefined (when possible). Most of the time will be spent in the *integrator* object (*theMonitor*), going through each *variable* in the queue for their appropriate integration treatment (according to the chosen method). This will involve a call of the method *compute* (redefined in Flight) for computing the *variable* derivatives. Occasionally, programmer/user defined conditions are satisfied to allow a break in order to handle a special task, such as a *reporter event* (from our *theReport*) or a *process event* (from our *thePilot*; not implemented). The end of simulation happens either when the flow has reached a point when there is nothing more to execute, or when a pre-defined condition for an *event* has been met.

Let us return to our main program in List 7. Before initiating the simulation, there are some necessary steps to be performed: the objects *thePlatform*, *theFlight* and *theReport* need have their data and be linked to each other in one context. This context is defined by *theEnvironment* (object of class *object*), into which these objects are introduced by means of *intoobject*. Although indirectly through *vars_intoobject* (defined in List 2), variables are also inputted into *theEnvironment* in the same way. *Process* objects, such as *thePilot* are introduced in the simulation process by activation, i.e. through *_diractivate*, as shown in List 7 for *thePilot* (and the the *process* descendent *theMonitor*, see List 9). Other steps before the simulation can start are: setting time steps and error tolerances (Pilot may be a suitable place for this) and setting report trigging characteristics. Finally *theEnvironment*, i.e. the *variable* objects (which are started internally), must be started, before *theMonitor* is activated.

5 Flight Application (Information for the Software User)

The flight application, here called Flight, has been created to be useful in the FOI inventory, and, at the same time, be of use when validating the methods against reference data. Both viewpoints indicated creating a package for civil applications: the needed PcP replacement and the existing civil flight-recorder data. As the flight-recordings contained positional information and this also was needed in FOI computation of distributed engine emissions and noise pollutions, a navigation module was written to aid the comparisons.

Technically, Flight has already been sketched in the example listings of 4.2. Essentially the flight-specific part of the application contains the implementations of a set of flight commands, their switching, and the integration control. The method *compute* takes care of the derivatives. Table 1 and List 2 indicate the set of implemented flight commands. Almost all of these are based on the same set of flight-mechanic equations, the difference being only which two “independent” parameters are selected in each case, see 2.4. The working of the navigation module is described in chapter 3.

Flight, as it stands today, is a rather complete simulation program for civil flight missions. For military applications additional flight commands are needed, especially where maximum performance governs the flight movement. Extending Flight for this kind of mechanics should be easy, as much is already prepared for this. For example the Platform part of the package can handle external loads.

The focus of this chapter will be on user aspects, such as input handling including the navigational side.

5.1 Flight Profile and Navigation

From the user point of view, the definition of the flight profile is isolated from that of the navigation, in most aspects. How the coupling will be done will be high-lighted later on. Despite the isolatedness, all inputs for the flight profile and for the navigation are given in the same file. Rows belonging to the former are marked with an ‘F’ and to the latter with an ‘N’.

5.1.1 Input for Flight Profile

Flight Profile Segments and Parameters

An overview of the available flight segments and their purposes is shown in Table 2. The controlling parameters for these segments are presented in Table 3.

There are some concepts that need to be explained, when exploring these tables:

Variable is a Comsim entity that is being updated continuously during the simulation. Four of this are defined: *V*, *S*, *H*, and *FC*. These hold states of velocity (along the path), horizontal distance (from the start), altitude, and consumed fuel, respectively.

Parameter is a flight segment entity that remains constant during the execution of the actual segment. It is user-specified. By means of the defaulting mechanism the parameter value can be transferred to the next segment.

Default value is given to any parameter when no value is specified in a flight segment. The mechanism varies with the segment

Table 2. Available flight segments.

Segment	Explanation	Notes
setstate	Set initial condition for flight (position, speed, fuel, etc)	The only place where the simulation *variables V*, *S*, *H*, and *FC* can be set
calibrate	Adjust zero-lift drag indirectly according to known flight state	The achieved drag coefficient level will be used for segments that follow from then on
taxi	Taxi at specified speed and distance or duration	Performs ground acceleration initially to the required speed
start	Accelerate on ground until lift-off speed	
climb	Climb and follow the minimum speed of IAS and Mach number	Performs *accelerate* initially to the required speed
accelerate	Accelerate in flight	The segment is used indirectly by *climb*, *cruise*, *descend*, and *approach*
cruise	Cruise or cruise-climb at required condition	Performs *accelerate* initially to the required speed
descend	Descend and follow the minimum speed of IAS and Mach number, and follow the required rate-of-descent	Performs *accelerate* initially to the required speed
approach	Descend at constant path angle (strictly)	Performs *accelerate* initially to the required speed
land	Retard (break) on ground	

Of the segments in Table 2, two serve special purposes:

setstate is normally the first command among the flight specifications on the file, where the purpose is initializing the flight simulation *variables*, to be left untouched for the rest of the flight simulation. *setstate* can be used repeatedly with the same number argument, to return to the flight state of the first occurrence of setstate-number combination. This mechanism enables studies of alternative flight sequences. Using a different number implies several branching levels.

calibrate is used when a flight simulation is required to reproduce a known performance out of known flight condititions. As the performance and all the conditions are pre-set, then something else has to be changed. Here an increment to the zero-lift drag coefficient is added (invisibly to the user), to be unchanged for the rest of the flight simulation. The increment is also valid for all speeds, despite only one speed condition is used with *calibrate*.

The workings of the remaining segments do not need further explanations, apart from the information given in Table 2 and section 2.4 (about flight-mechanics and following certain flight conditions).

There is a one-to-one coupling between the flight segments in the file specification and the methods in the Flight code, both having the same name, and hence the reference to Flight in Table 3. An overview of the methods with their arguments (attributes in C++ terminology) has already been listed in List 1. The names of these are listed in column 1. Column 2 will be referred to in the input section.

In the right-most column, the names with '_rq' are used internally within Flight and represent values set in the previous segment in the default system.

Table 3. Available segment parameters.

Parameter in Flight method list	**Parameter code in input file**	**Explanation**	**Segment application**	**Default value**
acc	(ac)	Acceleration when changing speed	*taxi, climb, cruise, approach*	*calibrate, taxi, start, cruise, approach*: 0; *climb*: 10;
al	(al)	Angle-of-attack [deg]	*taxi, start, land*	taxi, start, land: 0
config	(cf)	Configuration name (found in .plf file)	*All segments*	Current config_rq
cllim, cl	(cl)	Lift coefficient to determine IAS, for cruise: cruise-climb parameter	*start, climb, accelerate, cruise, descend, approach*	4
dist	(di)	Max. distance [m] for segment, for setstate: startpoint in flight profile	*setstate, taxi, accelerate, cruise*	40000000
dur	(du)	Max. duration [s] for segment	*taxi, climb, accelerate, cruise, descend*	3600
fc	(fc)	Initial state of consumed fuel, in [kg] or fraction of internal capacity (value in .plf file)	*setstate*	0
takeoff_fuelmass	(fu)	Tanked fuel, in [kg] or fraction of internal capacity (value in .plf file)	*calibrate, setstate*	Max. internal capacity
ga	(ga)	Elevation angle [deg] of path	*taxi, start, climb, accelerate, cruise, approach, land*	*calibrate, taxi, start, accelerate, land*: 0; *climb, approach*: current ga_;
hrlim	(hr)	Min. rate-of-climb [m/s]; for climb: end of segment when reached, for descend: hold the reached neg. value	*climb, descend*	*climb*: 0.5; *descend, approach*: -25
alt, h, hlim	(hs)	(Final) altitude state [m]	*calibrate, setstate, climb, descend, approach*	*climb*: 1000; *descend, approach*: 0;
vi, ias, iaslim	(is)	(Final) IAS state [m/s]	*calibrate, setstate, taxi, start, climb, accelerate, cruise, descend, approach, land*	*Start:* 1.2*stall speed, *land*: 0; else current ias_rq
mach, machlim	(ma)	(Final) Mach number state	*calibrate, setstate, climb, accelerate, cruise, descend, approach*	Current mach_rq
mu	(mu)	Friction coefficient	*taxi, start, land*	*taxi, start*: 0.02; land: 0.4
paftt	(pa)	Part thrust (<=1) of max. afterburning state	*calibrate, start, climb, accelerate, cruise*	Current paftt_rq
pt	(pt)	Part thrust (<=1) of max. non-afterburning state	*calibrate, taxi, start, climb, accelerate, cruise, descend, land*	Current pt_rq
v, vel	(vs)	(Final) absolute speed [m/s]	*calibrate, setstate, land*	*land*: 0
no	(no)	Setstate ID no; defined at 1st occurrence, use as return reference at later ocurrences	*setstate*	0

Input Format

Flight segment commands are given in the order they are to be exequted, one command per row in the file. The format for the command follows the pattern:

F <segment> (<par1>) <val1> (<par2>) <val2> (<par3>) <val3> … //comment

where <*segment*> is a name as above; *par1, par2, par3 …* are two-letter codes according to Table 5.2; *val1, val2, alv3 …* are the corresponding values (numeric except for the textual *'(cf)'*); *comment* is any string cannot be interpreted as a parameter code.

The parameter codes with the corresponding value can be placed in any order.

5.1.2 Input for Navigation

The navigation module in the flight application uses a pre-defined map path, which divided into consequtive lines and arcs. The input consists in specifying the corner points and radii of a polygon train, which are used by the module to create the necessary lines and arcs. Figure 6 illustrates this.

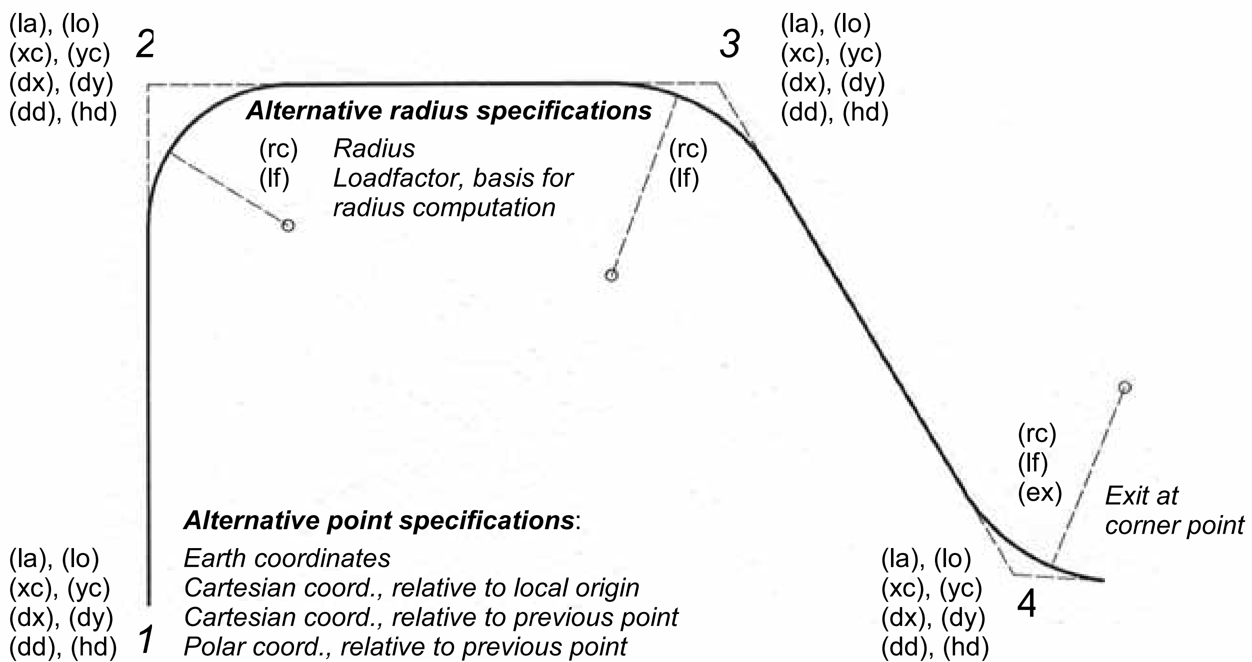


Figure 6. Alternative ways of specifying the navigation path, consisting of straight lines and circular arcs. The path starts at '1'. The parameter codes within parenthesis are explained in the report text.

In the figure four corner points are shown, where the coordinates of each can be specified in four alternative ways in the input file. The two-letter codes within parentheses are used in the file specification.

<u>Absolute Coordinates</u>

It is possible to use *absolute coordinates (latitudes and longitudes)* only, which is necessary for long (global) distances and also useful for short distances. Then all corner points are specified in the same format, one corner per row in the file, where *dd mm ns/ew* stand for degrees, minutes and one of the letters *n*, *s*, *e*, and *w* for north, south, east, and west, respectively, and *r* is a radius value [m]:

N (la) dd mm ns (lo) dd mm ew (rc) r //comment about the location

When specifying a location from which a global flight is to be executed, it is of interest that the path will be the shortest possible, i.e. the great-circle path be followd. This is done by giving the mode parameter *(md)* the value *g* (for global/great-circle):

N (md) g (la) dd mm ns (lo) dd mm ew (rc) r //comment about the location

Local Coordinates
It is also possible to use *local coordinates (X and Y with an implicitly defined origin)* only, which is only useful for short distances. For long distances these coordinates become meaningless due to the curvature of Earth. Hereafter all corner points are specified in the same format, each per row in the file, where *x*, *y*, and *r* are metric values [m]:

N (xc) x (yc) y (rc) r //comment about the location

Alternatively a corner point can be specified relatively to the previous corner point, either as increments in X- and Y-directions [m], or a distance increment [m] and a heading direction [degrees]:

N (dx) dx (dy) dy (rc) r // comment about the location

or

N (dd) dd (hd) hd (rc) r // comment about the location

Mix of Absolute and Local Coordinates
Finally, it is possible to mix absolute coordinates with local coordinates, even with several local systems. Then it is mandatory to couple each local system to a latitude – longitude pair, which is done by specifying both absolute and local coordinates on the same row as:

N (la) dd mm ns (lo) dd mm ew N (xc) x (yc) y (rc) r //comment about the location

After such a row either absolute or local coordinates can be specified. Internally in the navigation module, if absolute coordinates are specified then the local ones will be computed, or vice versa.

Finishing Navigation
The last navigational location, regardless of how it is specified, need not have the radius *(rc)* specification. Instead an exit command (ex) must be given:

N ... (ex)

5.1.3 Interaction between Flight Profile and Navigation

Up to now the the navigation (5.1.2) has been treated as completely independent of the flight profile (5.1.1). The navigation path is traversed while performing the flight simulation. As the paths of the navigation and the flight profile have implicit lengths, there is a risk for mismatching: the flight may end before the navigation is finished, or vice versa. Hence, there is a need for synchronizing.

Distance Synchronization
The main synchronizing mechanism is having the flight profile (F) and navigation (N) commands in the same file, and let an occurred navigational condition trigger a pre-mature end of a flight segment, in order to start the next following flight segment. Presently, only the basically horizontal flight segments *taxi* and *cruise* allow for pre-mature breaks. Thus the following will enable the interaction:
F cruise ... //(di) not to be specified here
N (la) dd mm ns (lo) dd mm ew (rc) r (do) dist_offset //provided cruise has been started

Not introduced before in 5.1.2 is the *distance offset (do)*, which is an offset [m] the user can specify relative to the location on the row, along the "straight" path from the previous given

location. By "straight" is meant literally straight (as handled in a local system) or along the great-circle path (as handled in the absolute system, when global is specified). The interaction mechanism is triggered when the specified location plus the offset (a negative value) is reached, which means that the on-going cruise will be broken. Hereafter the the navigation will be continued to the location (regardless of the offset), while executing the next (or more) segment(s).

Load Factor Interaction

Another type of interaction is through the aircraft load factor. Whenever the aircraft follows an arc of the navigation path, the flight speed together with the turn radius of the path cause an increase of the load factor, when compared to a straight flight. This increase of course implies a drag rise. This handling is automatic without the user action.

There is, however, a situation when the user needs to be aware of the load factor state. This is when the factor approaches the maximum level that the aircraft is allowed to perform due to aerodynamic, structural or pilot-related limit. If that happens then the specified radius is too small. The consequence is that the flight path cannot be followed. To circumvent this, instead of giving a radius (rc) value, a load factor (lf) can be specified:

N ... (lf) nn ..

This interaction is performed just prior to the entry of the arc associated with the assigned load factor. It is only then the state information about the altitude and speed is available, on which to base a computatiom of loadfactor and the corresponding ard radius.

5.1.4 Example

The final example that is being described here, to illustrate the usage of the flight and navigation commands, is input for the validation case, where the simulation results will be discussed in chapter 6. As this input is complicated, a simpler example will be introduced first, using the same aircraft modelling.

Input File Structure

The flight application using the Comsim-Platform package requires the following input files:

1. <aircraft>.aer_dat
2. <aircraft>.eng_dat
3. <aircraft>.plf
4. <flight_navigation>.fn

The first two files, which are inputs for the Platform part of the package, contain the aerodynamic coefficient basis and the engine tables with thrusts and fuel rates. The format for the files are described in Hasselrot et al. (1987). No example of aircraft data, i.e. .aer_dat and .eng_dat, will be presented here.

The third file serves as a master file, from which the *.aer_dat* and *.eng_dat* files are reached. Here the basic weights are given, the configurations are defined, and finally, the static sea-level thrust and the reference area for aerodynamics are specified. The last two items imply scaling, as these are also given in the *.eng_dat* and *.aer_dat* files. It is to be described in this report, later on while going through the example.

The fourth file is the true simulation file, as it is unique for each flight simulation. The input rules for the flight profile and the navigation are described in 5.1.2.

Example of Platform File

List 10 shows an example of a platform file (.plf), in which the basic data (*.aer_dat* and *.eng_dat*) for the MD90-30 are referred. The values in this file are complementary data, needed for the simulation. They are essentially self-explanatory, but the configuration handling is worth commenting. The *.aer_dat* file contains the configuration building blocks and the built configurations, which are used to form the actual named configurations, see *configlist* and *confignolist*. The names are referred to in all simulation files that use this aircraft as basis. The values at *addmasslist* are the used passenger payload [kg].

List 10. Contents of a Platform file (.plf): data for the McDonnell Douglas MD90-30

```
//file: md90-30.plf
//date: 2005-12-20
//basic aircraft and engine data (only first name; data must have extensions .aer_dat
and .eng_dat)
basicaircraft:  md90-30a
wingarea:       113.0   //default found in .aer_dat file: 113.0 m2; other value implies
                        //using this as new reference (=scaling the size)
maxbasicthrust: 222192. //default: determined by static sea-level data and engine
scale.
maxtakeoffmass: 70760.  //default found in .aer_dat file
emptyopmass:    39916.  //empty operating mass
maxfuelmass:    16964.  //default found in .aer_dat file
maxpayload:     13880.  //max payload
maxloadfactor:  1.5     //default: 7.
configlist:     clean   mid-flaps  mid-flaps+gear  max-flaps  max-flaps+gear
//the names in this list are defined here; these are referred to in the flight
instruction file
confignolist:   1       4          5               8          7
//configs are defined in the last lines of .aer_dat file; a config thus may be composed
//by several elements defined earlier in the file
addmasslist:    9284.   9284.      9284.           9284.      9284.
//the config system is used for flap settings; hence weights are used for payload
```

Example of Flight and Navigation File

To illustrate how a typical flight–navigation file would look like, List 11 is shown. The flight profile and navigation aspects are marked with *F* and *N*, respectively. During the preparation of this file, these aspects are viewed separately at first. Afterwards some N rows are moved among suitable F rows to satisfy synchronization needs.

As shown there are traces of all the *N* rows having been collected. Later on a couple of the last *N* rows were moved, in order to satisfy the need of starting the descent, here simplified as an *approach*, early enough to be in *Copenhagen* when the descent is finished. Through a couple of experiments (the flight software helps with information) it was found that the descent of -4 degree slope had to be started 85000 m before reaching the beacon *Fyr Kemax*. The row containing this beacon was therefore moved to a position immediately after the *F* row containing *cruise*, to enable the break interaction, see 5.1.3. Of course the remaining *N* rows must also be moved (the place is uncritical).

The flight profile, described by the *F* rows, is relatively easy to follow, as both the segments (*taxi*, *start*, etc.) and their parameters (names in parenthesis and values) are described in 5.1.1. Some parameters have values transported from the previous *F* row, and therefore need not be explicitly specified. This is the effect of the defaulting system. The default values in the different situations are presented in Table 3. For example, no *(is)* is specified for start, which implies 1.2 times the stalling speed is used. Defaulting from the previous segments are used in the example for *(cf)*, *(is)*, and *(ma)*. Finally to note, cruise has no *(di)* specified, as would seem logical. This is intentional, because it is to be broken through interaction by the follwing *N* row.

List 11. Contents of a Flight and Navigation file (.fn): data for a Stockholm – Copenhagen flight

```
N      (la) 59 39 n   (lo) 17 55 e    (dd) 0      //Stockholm/Arlanda
N      (la) 59 12 n   (lo) 17 01 e    (rc) 5000   //beacon Fyr Dunker
N      (la) 58 18 n   (lo) 15 43 e    (rc) 5000   //beacon Fyr Vassen
N      (la) 57 33 n   (lo) 14 44 e    (rc) 5000   //beacon Fyr Shilling
F      setstate       (fu) 7800       (hs) 0      (is) 0
F      taxi   (cf) mid-flaps+gear     (pt) .2     (is) 10      (du) 500
F      start     (pt) 1
F      climb     (hs) 500             (is) 80
F      climb     (cf) mid-flaps       (pt) .95 (hs) 1500
F      climb     (cf) clean           (pt) .95 (hs) 10668  (is) 160     (ma) .77
F      cruise    //no distance value set, cruise will stop at 85000 m before Fyr Kemax
N      (la) 56 08 n   (lo) 13 13 e    (rc) 5000   (do) -85000  //beacon Fyr Kemax
F      approach  (ga) -4              (hs) 1500   (is) 130
F      approach  (cf) mid-flaps       (ga) -4     (hs) 500     (is) 100
F      approach  (cf) max-flaps+gear  (ga) -4     (hs) 0       (is) 69
F      land      (pt) 0               (is) 10
F      taxi      (pt) .2              (is) 10     (di) 2000
N      (la) 55 37 n   (lo) 12 39 e    (rc) 5000   (ex)         //Copenhagen/Kastrup
```

6 Validation (Information for the Generalist)

Validation in our context is stating the soundness of the various aspects of the modelling.

The idea of creating the Comsim-Platform package, to be used in flight applications, was re-using well-tried modules. The main constituents, as the package name indicates, are simulation package and the aircraft management. This basis can be assumed to be validated. The foundation for the platform side is specified aerodynamic and engine characteristics, and this means that the performance of the platform model depends on the quality of the input.

To be useful for flight application, the Comsim-Platform package has been provided with newly developed modules: a platform extension with a set of flight-mechanic relations, and a navigation module with user-defined map paths. To test this and to be of practical usage, a flight management program has been written. The functionality of the flight-mechanics in the different flight profile segments and the management system is an area, where validation is needed.

The validation idea is using a flight recorder data from an actual flight as a reference, and select a case having an aircraft with well described model data to enable accurate flight simulation. Thus the aim is to compare the results, with focus on the flight performance. The Scandinavian flight operator SAS has kindly provided us with a number of flight-recordings, among which we have chosen one: a Stockholm – Copenhagen flight with the aircraft MD90-30 and V2525-D5 engines.

6.1 Modelled Data

As the validation aims at verifying flight data and for it to be credible, the methods for creating the underlying data must be presented. These concern the aerodynamics for the aircraft, the engine thrust and fuel rate during operation, and weight data for the actual referred flight. Only then it is meaningful to compare the conditions and results of the flight itself.

6.1.1 Aerodynamic Data

Aerodynamic data for the airframe of the MD90-30 were not available at FOI. However, there existed manufacturers' data for a related aircraft: DC-9-40. Externally, the wing geometry of the two aircraft is about the only difference. The MD90-30 has almost the same fuselage size as the older aircraft, but the wing has been increased (=higher aspect ratio). Essentially the adjustments consist in the following:

1. The zero-lift drag of the DC9 has been adjusted with regard to the overall wetted area.
2. The parabola factor (K, see 2.2.1) of the DC9 has been inversely proportioned with respect to the aspect ratio.

6.1.2 Engine Data

FOI did not have access to real thrust and fuelflow characteristics for the V2525-D5. In connection with some environment-related work, considerable effort had been laid down at FOI in accurately computing data, using the commercial software GasTurb [Kurzke, 1988], for a related engine: the V2527-A5. In the ICAO databank [ICAO, 1995] for engine emissions, these two engine versions are presented with identical thrust and fuelflow properties. Thus we accepted all data that were created for the V2527-A5 to be valid for the V2525-D5.

6.1.3 Miscellaneous Data

The third main data category for the aircraft is the mass components that sum up into the total flight weight. This varies with every flight, which is due to the passenger and fuel loads. Fuel is usually tanked for the flight in mind and allowing for reserves. From simulation point of view it is only the initial total aircraft weight that matters. In the SAS flight-recordings no aircraft mass is specified. This is noted in other loggings. For the chosen Stockholm – Copengagen flight, FOI

remembered making an extract of the logging for the actual flight, where the take-off mass was noted: 57000 kg.

Other data that have a potential of influencing the performance results are when configuration changes, i.e. the retraction/deployment of the high-lift system and the landing gear, are performed. The effects are drag changes. For the chosen flight-recording there was no information on this. Recordings of other flights with the actual aircraft type did have this. Thus approximate times for for the configuration changes could be had. In any case the errors in applying the different drag levels at slightly wrong times are very small (a few seconds), when viewing the fuel consumption on flight basis.

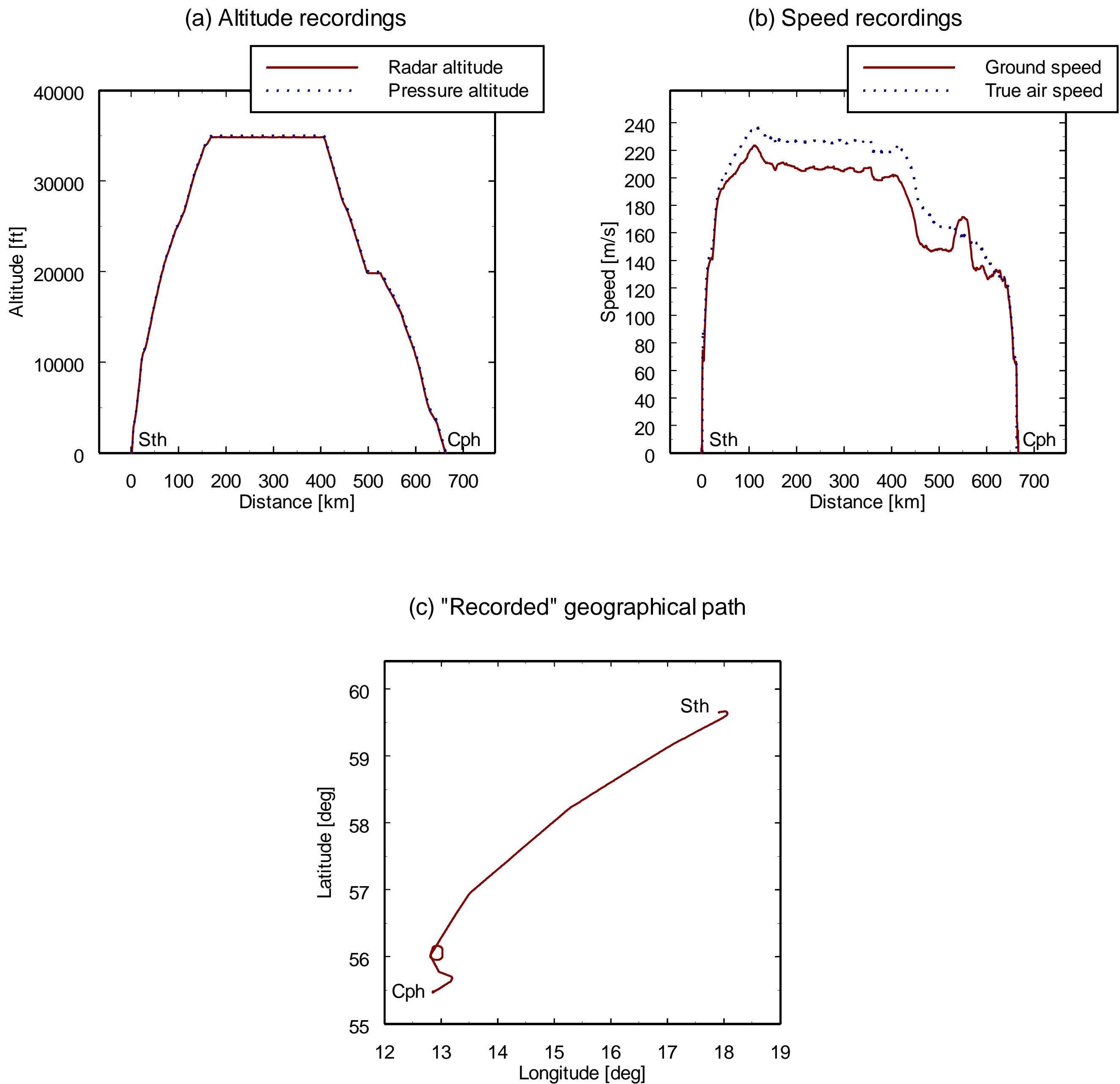


Figure 7. Check of Flight Recorder Data.

6.1.4 Flight Data

The selected flight recorder data, used for the validation process, consist of time-recorded (every fouth second) data for many parameters, of which only a few have been used in the comparisons. The reproduction of the complicated flight procedures has been performed largely by observing how the thrust (or fuel) and the speed change with the altitude specifying the corresponding commands for the simulation. In descents, path angles and descent rates have been studied alternately. The final reproduced flight, from which the comparison data are shown in this chapter, has been created with the flight/navigation file that is reproduced in List 12. This is a considerably more complicated version of List 11, which was explained in 5.1.4.

List 12. Contents of a Flight and Navigation file (.fn): data for a Stockholm – Copenhagen flight

```
F      setstate      (cf)   mid-flaps+gear      (fu)   7800   (hs)   0      (is)   0
N      (la)   59 39 n       (lo)   17 55 e      (xc)   0      (yc)   0      (dd)   0
N      (dd)   35     (hd)   267    (rc)   10
N      (dd)   450    (hd)   356    (rc)   10
N      (dd)   241    (hd)   20     (rc)   10
N      (dd)   255    (hd)   40     (rc)   10     //runway end
N      (dd)   470    (hd)   73     (rc)   3000
N      (dd)   5600   (hd)   79     (rc)   3000
N      (dd)   5200   (hd)   180    (rc)   3000
N      (la)   59 9 n (lo)   17 01 e      (rc)   5000   //near beacon Fyr Dunker
N      (la)   58 14 n       (lo)   15 18 e      (rc)   5000   //near beacon Fyr Vassen
N      (la)   56 57 n       (lo)   13 30 e      (rc)   5000
N      (la)   56 00 n       (lo)   12 49 e      (rc)   1000   //near beacon Fyr Kemax
N      (dd)   11000  (hd)   90     (rc)   1000
N      (dd)   15000  (hd)   0      (rc)   1000
N      (dd)   11000  (hd)   270    (rc)   1000
N      (dd)   15000  (hd)   180    (rc)   1000
N      (la)   55 46 n       (lo)   13 00 e      (rc)   1000   //exact
N      (la)   55 41 n       (lo)   13 12 e      (rc)   1000   //exact
N      (la)   55 28 n       (lo)   12 52 e      (rc)   1000   //exact
N      (la)   55 29 n       (lo)   12 52 e      (rc)   1000   //exact
N      (dd)   10000  (hd)   100    (rc)   1000
F      taxi   (cf)   mid-flaps+gear      (pt)   .2     (is)   .9     (di)   1010
F      start  (pt)   1      //(ma) .25    //(is) 1081
F      climb  (hs)   517    (is)   100    (ma)   .25
F      climb  (cf)   mid-flaps     (pt)   .99    (hs)   850
F      climb  (pt)   .98    (hs)   1300   (is)   125    (ma)   1
F      climb  (pt)   .97    (hs)   1422   (is)   127
F      climb  (cf)   clean  (pt)   .96    (hs)   3300   (is)   130
F      climb  (pt)   .95    (hs)   3700   (is)   160    (ma)   .77
F      climb  (pt)   .93    (hs)   4200   (is)   160    (ma)   .77
F      climb  (pt)   .91    (hs)   5000   (is)   160    (ma)   .77
F      climb  (pt)   .89    (hs)   6000   (is)   160    (ma)   .77
F      climb  (pt)   .87    (hs)   7000   (is)   160    (ma)   .77
F      climb  (pt)   .86    (hs)   8750   (is)   160    (ma)   .77
F      climb  (pt)   .84    (hs)   10668  (is)   160    (ma)   .765
F      cruise (pt)   1      (ma)   .765   (di)   164350
F      cruise (pt)   1      (ma)   .741   (di)   54900                //58650
F      descend       (pt)   0      (hs)   9750   (is)   135.5  (hr)   -11    (ma)   .753
F      descend       (pt)   0      (hs)   9500   (is)   135.5  (hr)   -11
F      descend       (pt)   0      (hs)   8400   (is)   135.5  (hr)   -11
F      descend       (pt)   0      (hs)   8100   (is)   122.5  (hr)   -6
F      descend       (pt)   0      (hs)   6100   (is)   122.5  (hr)   -8
F      cruise (pt)   .3     (is)   122.5  (di)   26000
F      descend       (pt)   0      (hs)   4600   (is)   123    (hr)   -5
F      descend       (pt)   0      (hs)   3600   (is)   123    (hr)   -7
F      descend       (pt)   0      (hs)   2650   (is)   120    (hr)   -7
F      descend       (pt)   0      (hs)   2300   (is)   120    (hr)   -9
F      descend       (cf) mid-flaps      (pt)   0      (hs)   1600   (is)   120    (hr)   -9
F      descend       (cf) mid-flaps      (pt)   0      (hs)   1400   (is)   120    (hr)   -7
F      descend       (pt)   0      (hs)   1000   (is)   120    (hr)   -4
F      approach      (ga)   -3.5   (hs)   750    (is)   103
F      approach      (ga)   -3.2675       (hs)   400    (is)   82
F      approach      (cf)   max-flaps+gear      (ga)   -3.2675       (hs)   0      (is)   69
F      land   (pt)   0      (is)   10
F      taxi   (pt)   .2     (is)   10     (di)   3000
```

Checking Flight Conditions in Recordings

Reproducing a recorded flight through a computer simulation requires that all the flight conditions are copied, if a meaningful comparison is to be performed. First of all, the flight is subjected to the current meteorogical properties. The use of the Standard Atmosphere in the simulation has to be checked, and also the expected headwind conditions.

In the chosen flight recorder data, two kinds of altitude presentations were available, which enable us to judge the correctness of using the Standard Atmosphere for the modelling, without or with temperature adjustment. The first is the radar altitude, which based on geometric measurement. The other is pressure altitude, which is a translation of static pressure into metric altitude, using the altitude—pressure relationship of the Standard Atmosphere. If the two kinds of altitude values co-incide, then it is proper to use Standard Atmosphere. If not, a temperature adjustment is applied so that the pressure—altitude characteristics of the flight are reproduced. Figure 7(a) shows that the former is valid, i.e. no temperature correction is needed.

The current headwind condition is easily established by taking the difference between the true air speed (TAS) and the ground speed. None of these is is directly accessible from the recordings. TAS is computed by combining Mach number recording and the sound speed given by the Standard Atmosphere. The ground speed is derived by using the path (metric coordinates) and time recordings. Figure 7(b) reveals that a head wind is encountered. Another proof of the current headwind is the peak at around 550 km flown distance, which is an effect of a performed waiting loop, see Figure 7(c).

Reproducing Flight Conditions in Simulations

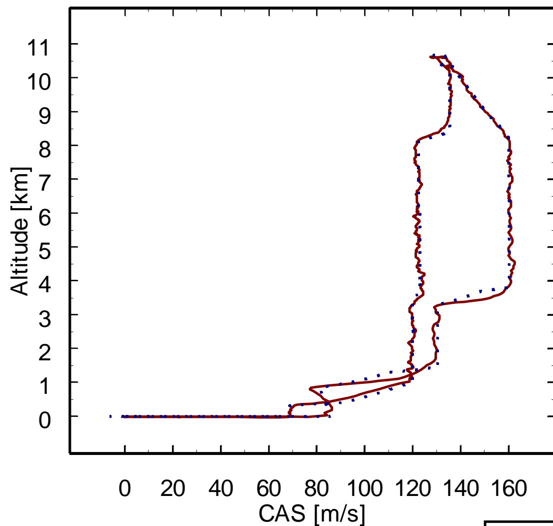
Flight is essentially governed by performance of the engines and aircraft aerodynamics. The thrust is usually kept high at takeoff and climb. However, there may be reasons for holding it lower: noise abatement, stress relax (due to lower burning temperature), etc. Aerodynamically, low speed may require flap deployment (safety), or the speed may be chosen to reach an optimum flight condition (lower fuel comsumption). Therefore it is important to study the thrust levels and speed states, in order to reproduce these sequences in the flight simulation.

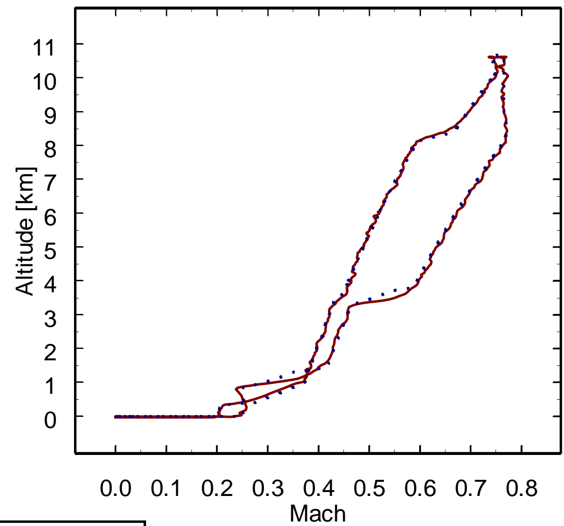
How can we best present the reproduction? Due to the identified headwind condition, which can not be simulated (in the present state of the Flight program, based on the Comsim-Platform package), the distance cannot be used as an independent parameter. We need a parameter that is not influenced by the headwind. The altitude fulfills this role almost perfectly, as we will use only quantities that are derived from pressure measurings, such as the calibrated air speed (CAS) and the Mach number.

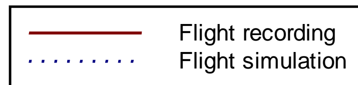
The speed is readily available in the flight recording, as CAS and as Mach number, see Figures 8(a) and 8(b). The thrust has to be computed from recorded fuel flow states, as the former is not presented. For this the thrust—fuel flow characteristics of the estimated engine data are used. We are happy about the correctness of using the Standard Atmosphere for the actual simulation, because the engine data have also been computed under this assumption. Thus the result of the thrust adaption is presented as fuel flow comparisons, see Figure 8(c).

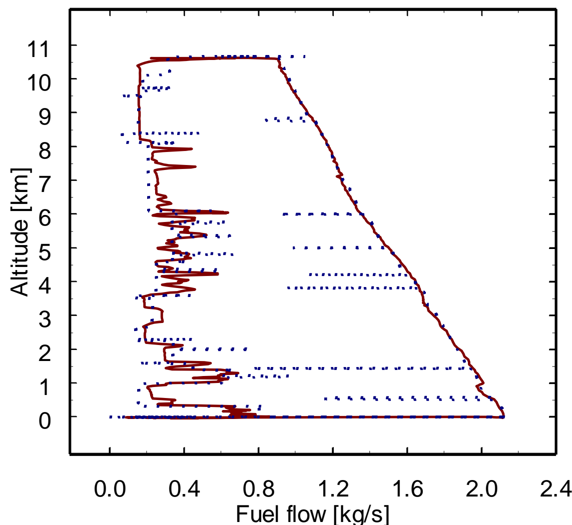
The thrust affects the climb performance, which may be viewed as rate of climb (r.o.c.), see Figure 8(d). Negative r.o.c. is to be interpreted as rate of descent (r.o.d.). It is apparent that the climb is well reproduced. The exception is visible within altitudes 8—9.5 km. Here the encountered headwind leads to inertial effect of gained altitude. The equilibrium is again established above this region.

The descent adaption has been focused on reproducing the r.o.d., which means the thrust level is an answer. It is possible to specify an r.o.d. value (=negative r.o.c.) in the software to control the descent. It should be noted that the actual flap settings in take-off and landing procedures for this flight are not known. Such were available, however, for other similar flights. We have assumed a mid-flaps setting at the altitude of 2.3 km during the landing, and maximum setting and gear down from 0.4 km. For the descent the resulting trust (=fuel flow) levels are well reproduced, see Figure 8(c).









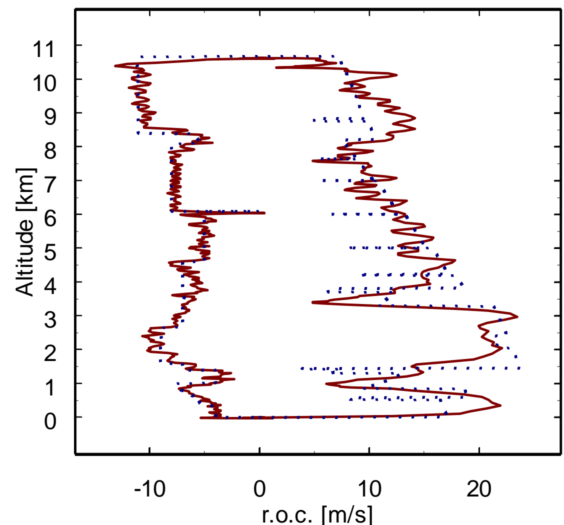


Figure 8. Reproduction of flight conditions.

<u>Comparing Flight Performance</u>

Having established the controlling flight parameters, i.e. the flap settings, the thrust and speed sequences, it is time to compare the path characteristics. Although neither the flight distance nor the flight time is fully suitable for the comparison of the recorded and similated flights, studying the effects of these leads to interesting observations. Viewing the distance is natural, since the flight is between two points, whereas the time is better for headwind independent studies.

Figures 9(a) and 9(c) (left vertical) show the fuel flow level (corresponding to a thrust) and the numerically integrated altitude as a function of the covered flight distance. Here it is evident that the recorded flight has a climb gain due to the headwind, and that the cruise starts earlier. Figures 9(b) and 9(d) (right vertical) show similar graphs, but now as a function of the accumulated flight time. Here it is very clear, when viewing only states relative to the free air without regard to the headwind effect, that the climb and the cruise (as long the smallest of the two cases lasts) are very much in agreement. It should be noted that the length of the simulated cruise has been adjusted, in order to have the descent parts of the recorded and simulated flights to match. The simulated end of cruise thus occurs earlier as a consequence.

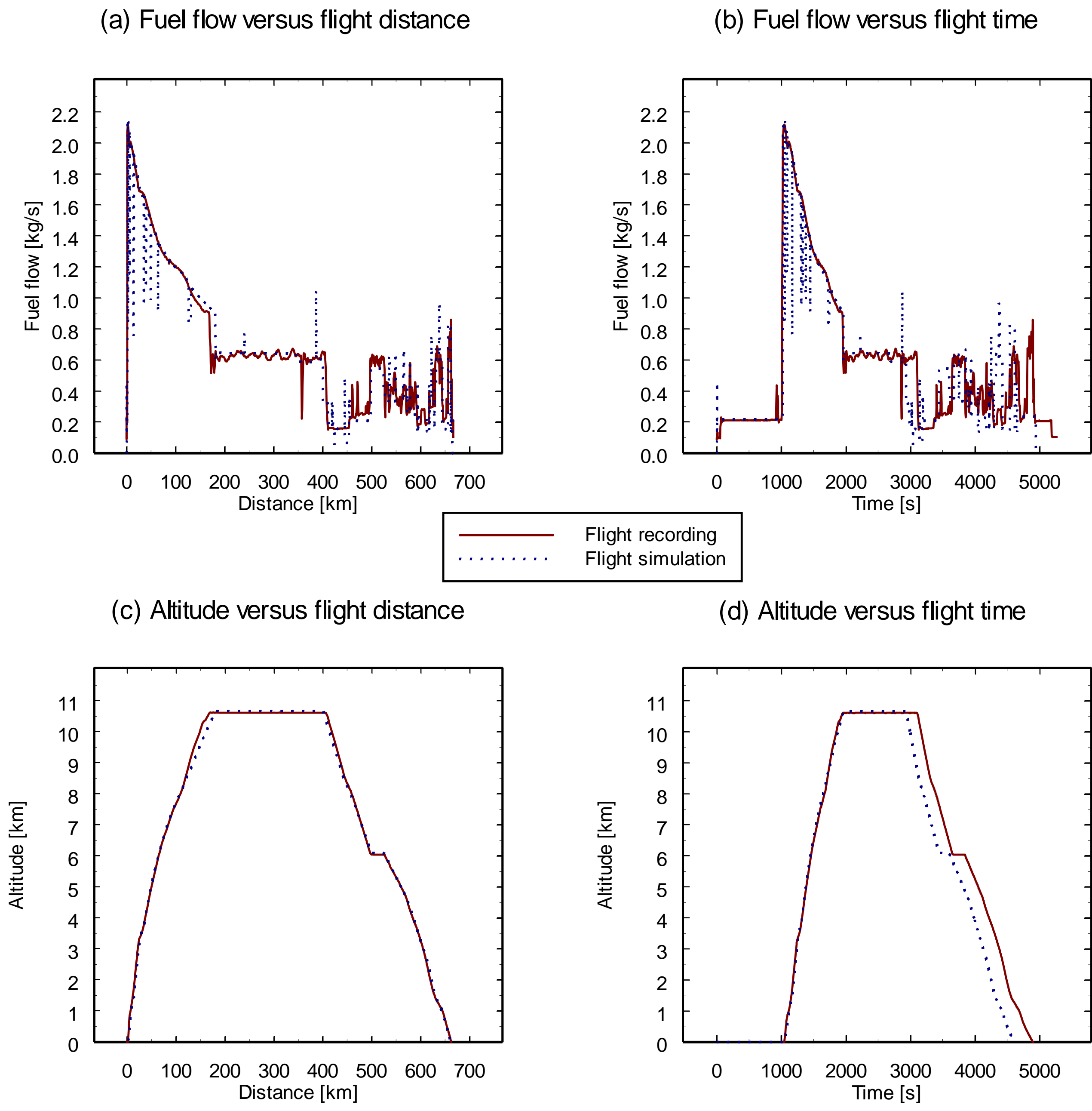


Figure 9. Comparison of engine thrusts (=proportional to fuel flows) and flight paths.

The good agreement between the fuel flows in cruise can be taken as a sign of good estimation of the aerodynamic drag. The reason is that during the cruise the flight speed is constant, which means that the drag level is almost in steady-state (as the weight decrease due to the fuel consumption is negligible) and thus also the thrust level (balancing the drag). This of course is under the assumption of correctly estimated thrust—fuel flow characteristics.

The relative merits of the path characteristics are also reflected in the accumulated fuel consumption, see Figures 10(a) and 10(b). As with the time-history of the climb and cruise, we have extreme agreement between the corresponding accumulated fuel consumption in the same phases. The first part in Figure 10(b) is the effect of the taxi-out phase, where the assumption of the rolling friction coefficient of 0.02 seems to be adequate (defaulted to this value in the flight specification file; another can be given). The difference in the consumed fuel versus the covered distance starts diverging when the recorded flight ancounters the headwind at about 8 km, by then the distance is about 100 km. From here on the diverging continues until about the approach phase. This is as it should be.

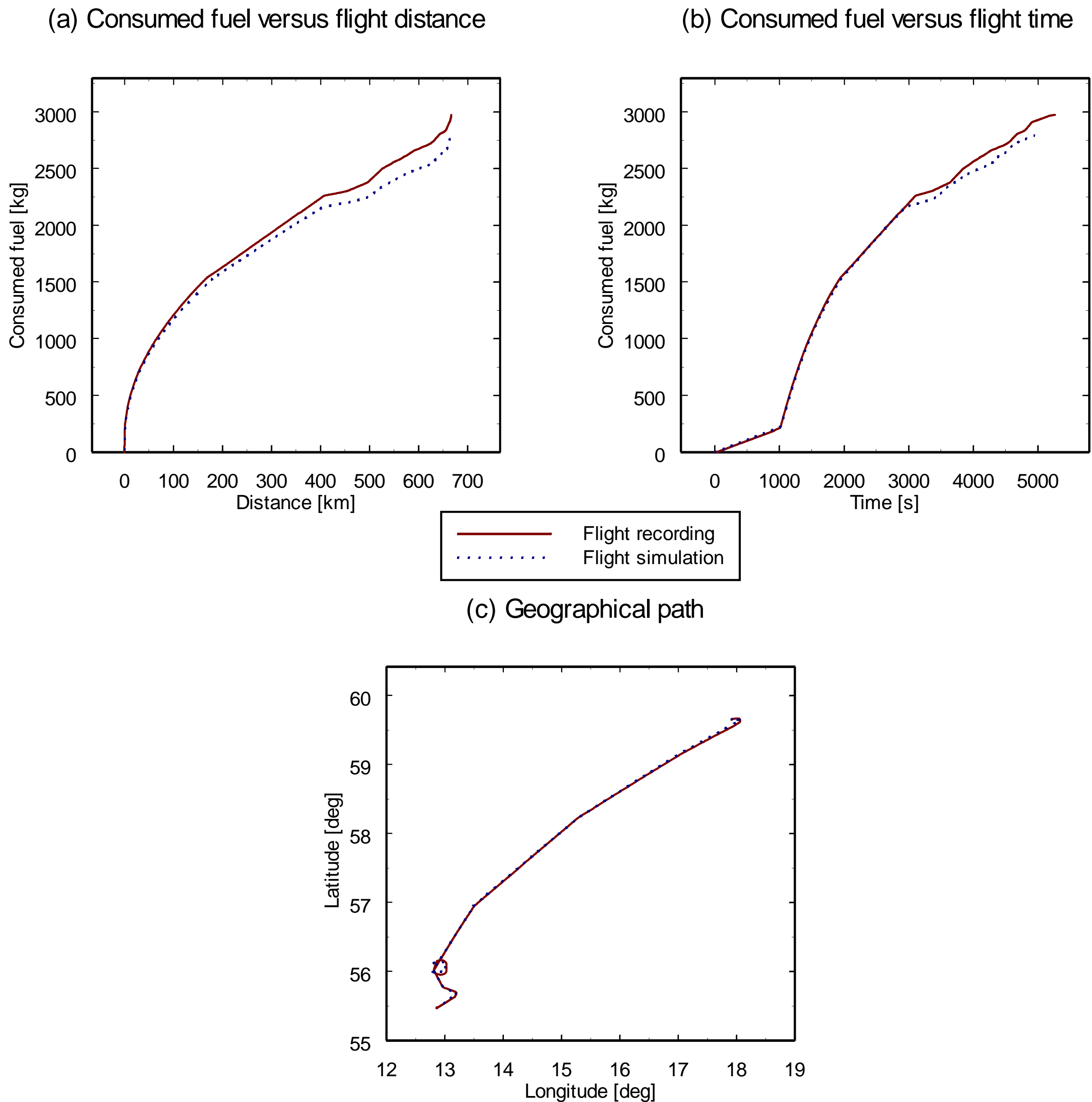


Figure 10. Comparison of integrated fuel consumptions and positions.

Finally, it is of interest to show the capability of the Comsim-Platform program (newly created and described in this report) regarding the management of the navigational path. Figure 10(c) shows the comparisons between recorded and simulated map paths, using latitude and longitude coordinates, where the former were translated from metric coordinates in a post-process.

7 Conclusions

A new flight simulation package, named Comsim-Platform and written in the program language C++, has been created to replace the LISP-based PcP (being in use for environmental flight studies at FOI). It re-uses the aircraft model, Platform, of the latter, and it has been provided with a new flight management system, including navigation. All numerical integrations and event notices are handled by the existing C++ based simulation package Comsim.

The Comsim-Platform package is intended to act as a basis for various types of flight simulations. The Flight/Navigation application that has been created for testing the functionality of Comsim, with special views on flight/navigation management, has proved its capability. The usage and functionality of Comsim-Platform and Flight/Navigation have been documented.

The ease of reproducing the the detailed procedures of recorded flights has been demonstrated. This has been used in the validation of the capability of the Comsim-Platform package. Using accurate data for the airframe and its engine(s) of a studied flight, the soundness of the flight-mechanic basis of the extended Platform model has been proved and documented.

The final conclusion is that the Comsim-Platform is ready to to be used, with existing version of Flight/Navigation, or with some type of modification/utilization of the latter modules.

References

Aronsson, J. (1991): "Comsim – a Module for Combined Discrete and Continuous Simulation in C++", Thesis for a Master's Degree in Computer Engineering at Lund Institute of Technology (1991).

Feldman S. I., Gay D. M., Maimone M. W., Schryer N. L. (1995): "A Fortran-to C Converter", Computing Science Technical Report No. 149, AT&T Bell Laboratories, Murray Hill, NJ 07974 (1995).

Hasselrot, A., Marklund A. (1987): "FFA-APP – a Computer Program for Estimating Flight Performance of Military Aircraft", FFA TN 1986-44 (1987).

Heldsgaun, K. (1978-9): "Combinedsimulation", reports (in Danish): "Introduktion" (Rapport nr 4, 1978), "Brugerhåndbog" (Rapport nr 5, 1978), and "Dokumentation" (Rapport nr 6, 1979). Dep of Computer Science, Roskilde University Center.

ICAO (1954): "Manual of the ICAO Standard Atmosphere", ICAO Document 7488 (1954).

ICAO (1995): "ICAO Engine Exhaust Databank", first edition (1995), International Civil Aviation Organization.

Kurzke, J. (1988): "User's Manual – GasTurb 8.0 for Windows, a Program to Calculate Design and Off-Design Performance of Gas Turbines", www.gasturb.de.

Månsson, L. (1980): "CONDIS, en vidareutveckling av simuleringspaketet Combinedsimulation" ("a Further Development of the Simulation Package Combinesimulation"; in Swedish), FOA Rapport C 20357-E3 (1980).

Righard, T (1981): "NYCOND, en omarbetning av CONDIS" ("a Revision of CONDIS"), FOA Rapport C 20419-E3 (1981).

SAS (1999): Flight Recorder Data for a selection of flights and aircraft operated by SAS, confidential data (1999).

Stehlin, P. (1999): "PcP: The Programmable Commercial Pilot", FFA TN 1999-02.

Stinton, D. (1966): "The Anatomy of the Aeroplane", G.T. Foulis & Co Ltd 1966, London.

US (1976): "U.S. Standard Atmosphere", 1976, U.S. Government Printing Office, Washington, D.C.

Appendix A. General Instructions on Creating C++ Classes from FORTRAN Codes

by Anders Hasselrot, FOI, September 2005

Translating FORTRAN to C/C++

Feldman *et al.* (1995) gives the basic information on the f2c translation. Using "f2c -C++ <FORTRAN code>", the proper calling format of functions/routines for C++ applications are created. In principle the generated code is directly ready to be compiled. It is also convenient to give the switch -c in order to include extractions of the original FORTRAN codes as comments.

Conditioning the FORTRAN code for C++ classes

If the intention is translating the code into C++ classes, it may be convenient to perform some minor source editings. The ‘common’ blocks will need careful treatment. All unlabelled blocks should be given names, as this will facilitate placing into classes. Also, let all the ‘common’ blocks with the same label have the same variable names, which apply all occurrences in the code, i.e. in all subroutines and functions.

Creating C++ classes

Some editing of the f2c-generated code is required, if class structures are to be generated. The generated code assumes that immediate instantiation is performed, i.e. that the program can be used. The class definition is a code that awaits the instantiation, i.e. a class object must be created before the code can be used.

The class is structured with "attributes" (constants and variables) and "methods" (functions and subroutines). Both "attributes" and "methods" can have different access levels: 'private' (default), 'protected' and 'public'. Of these 'public' is the most important. Under the 'public:' specification all "methods" that may be of interest to be run separately are placed here. Regarding the "attributes", almost all are to be kept under 'private:'. If some "attributes" need to be extracted, do this through 'public' "methods".

The C++ programming technique is based on separaion of code into .h and .cpp files. The former type is used for the basic 'class' definitions within which the "attributes" are placed according to their access level. In addition, "prototypes" (function/subroutine name with arguments) of the "methods", also according to their access level, are put here. In the .cpp file the full "method" definitions, taken directly from the f2c generated code, are placed, where however the '< class name>::' is inserted just berore each "method" name, in order to qualify the "method" for the class in view.

FORTRAN 'common' statements are translated into structs, with names based on the FORTRAN labels. The ‘struct’ concept is very similar to class concept in C++ (as opposed to C ‘structs’): they can have "attributes" and "methods", which have the same access possibilities as the class ditto. However the default access level is 'public'. However, structs are are mostly used for data only applications. Structs in classes cannot be initialized in the way it may be done in C and somple C++ applications. The only way is assigning each variable explicitly, i.e. using the '=' operator. This has to be reviewed in the generated f2c code.

During the code generation, constants that f2c detects are treated as static variables. These may be placed outside of the class scope. Problematic static variables are those that are created by local (defined in a function/subroutine) and DATA-set variables. These are static and initialized. Being static, such a variable cannot be created in several instances, i.e. two (or more) class objects will share the same static variable. The DATA-statement (as translated from FORTRAN) will take

effect during the first object creation, but not in the second. To sum up: initializing a local static variable must be done through assinging it the initial value (not :'static <type> <parameter>=<value>', but: 'static <type> <parameter>' and '<parameter>=<value>').

The f2c automatically generates 'extern "C"' to avoid name mangling during linking. This applies to the explicitly defined functions/subroutines. However, there are other functions that are created during the translation process of specific FORTRAN characteristics, such as open, read and write statements. These are not declared with 'extern "C"' (which they should), and hence causing name mangling problems. There are also other name referencing problems that the f2c code generates: in functions/subroutines where calls of other routines in the package are performed, for which f2c has defined extern declarations. These are not needed and should be commented out.

Reference

Feldman S. I., Gay D. M., Maimone M. W., Schryer N. L. (1995): “A Fortran-to C Converter”, Computing Science Technical Report No. 149, AT&T Bell Laboratories, Murray Hill, NJ 07974 (1995).