AE3212-II

Simulation, Verification and Validation

Lecture Notes

February 2021

E. Mooij, Z. Papp, W. van der Wal

Faculty of Aerospace Engineering

Delft University of Technology

Simulation, Verification and Validation

Lecture Notes

E. Mooij, Z. Papp, W. van der Wal

February 2021

Faculty of Aerospace Engineering · Delft University of Technology

Delft University of Technology

Table of Contents

Chapter 1

Introduction

The development and testing of large and expensive engineered systems, such as cars, aircraft, spacecraft, and weapons systems, has become increasingly reliant on simulations. Such large complex systems consist of so many interdependent parts that it is impossible to determine the system's collection of behaviors from just physical tests. Even if it were possible, the cost of doing so would be prohibitive. Imagine having to execute a number of crash tests with a brand-new car for dozens of different speeds and configurations at every stage of development just to check if you are still heading in the right direction. That car would never be finished. Developers and designers therefore use computer simulations for all these intermediate steps.

To simulate the behavior of something, the process, system or structure under consideration must be modeled somehow. This model should be appreciably less complex than the system it represents, but at the same time provide results that describe and predict the behavior of the system being modeled with sufficient accuracy. Modeling a complex system takes away the necessity of testing the system itself at every step of the way. However, such a model cannot be too complex, because it would take too much time and energy to develop, and would take too much computing time to deliver results. There is no point in developing a model that is equally complex as the original system; this would take away the benefit of developing a model to begin with. However, it is equally important that the model still represents the system accurately enough; otherwise the simulation results are useless. An example of a model is the engineering model of a satellite. It looks like the actual satellite, but it does not have a functioning electrical subsystem. The goal is to test the satellite for launch conditions and therefore it is important that the moment of inertia is correct, but not that payload is fully functioning. Models are thus used to make predictions about characteristics that cannot all be tested for practical reasons, such as performance, safety and reliability. In this course, we focus on computer models.

Depending on the purpose of the simulation, the model only has to predict the behavior of a system to a certain degree of accuracy. For some purposes the error margin may be 0.1%, while for many others it may be sufficient to deliver predictions within 10% or 20% of the

Figure 1-1: The structural and thermal model of Bepi Colombo in the Large Space Simulator at Estec, Noordwijk, source: ESA.

system's actual response. Depending on these accuracy requirements, the model can be made as simple or complex as necessary. If a simulation has an accuracy of 1% while only 10% was required, then the model is probably too complex and too costly. The accuracy of the model should be such that it is adequate for its intended use. Simplifying a model is done through assumptions. Assumptions for numerical simulations can reduce computational time and still provide a convenient and sufficiently accurate method to predict the behavior of many systems and structures. Sometimes assumptions are necessary to be able to produce a model at all. However, assumptions introduce uncertainty and affect the accuracy of the model.

During the development of a model or computer simulation, a lot of effort is involved in checking if everything works correctly. The National Institute of Standards and Technology found that software engineers spend on average 70 to 80% of their time on testing and debugging ([12]). Just because a program runs without errors does not mean that it delivers the right results. So how does one determine whether the simulation results obtained from a certain model are to be trusted? Verification and validation (V&V) are two very important processes focused on just that: determining and demonstrating the credibility of a model. An easy way to define the two is by the questions each process aims to answer. Verification can be expressed by the question: 'Are we building the model right?' and validation by: 'Are we building the right model?' Note that verification and validation can also apply to products. The NASA System engineering handbook describes verification as follows ([17]): "Verification proves that a realized product for any system model within the system structure conforms to the build-to requirements (for software elements) or realize-to specifications and design descriptive documents (for hardware elements, manual procedures etc.). The specifications or requirements are important here, those are the criteria that determine when a product is validated. In the following mostly verification and validation of computer models is discussed. In this course we will focus on models, and not discuss V&V of products.

The way the above questions for V&V are phrased also implies another important aspect of V&V, namely that it should be done concurrently with model development. In the long

term, it saves a lot of time to perform V&V during the development of a program and find and fix errors early on. Because V&V determines the quality of a product, V&V procedures are required by certifying authorities, such as flight qualification for airplanes. Because of the importance for the quality of the product in industry and science, V&V should be an important part of any engineering program, and it is hoped that this course gives a handle for applying V&V. Actual implementation of V&V differs from one institute to another, therefore V&V will always be an important part of learning on the job. However, it is very useful to go through the exercise of applying V&V, more useful than learning the theoretical concepts of V&V. To learn how to apply V&V, the model must be complex enough so that V&V becomes necessary. A systematic V&V for a simple exercise is usually overkill. Therefore, this course consists of two large group assignments. V&V is also an integral part of the Design Synthesis Exercise, and the MSc thesis, which are also complex enough that they require systematic V&V.

These lecture notes start with a description of some general concepts (Chapter 1). How to actually perform a simulation is explained in Chapter 2. Verification is detailed in Chapter 3, while validation is discussed in Chapter 4. This is followed by examples of simulation, verification and validation in aerospace engineering (Chapter 5). Then the deliverables for the course are discussed, the simulation plan (Chapter 6), and the report (Chapter 7).

1-1 General Concepts

The words code, model and simulation are often used in engineering, sometimes interchangeably. This will also be the case in this reader, but for clarity the official definitions as formulated by the American Society of Mechanical Engineers (ASME) [6] are given:

Code : The computer implementation of algorithms developed to facilitate the formulation and approximate solution of a class of problems.

Model : The conceptual, mathematical, and numerical representations of the physical phenomena needed to represent specific real-world conditions and scenarios. Thus, the model includes the geometrical representation, governing equations, boundary and initial conditions, loadings, constitutive models and related material parameters, spatial and temporal approximations, and numerical solution algorithms.

Simulation results : The output generated by the computational model.

The definitions for verification and validation as formulated by the U.S. Department of Defense [20] and the American Institute of Aeronautics and Astronautics (AIAA) [2] are:

Verification The process of determining that a computational model accurately represents the underlying mathematical model and its solution.

Validation The process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model.

Basically, *verification* is necessary to confirm that the computational implementation of a model and its solutions are correct (*'are we building the product(model) right?'*). *Validation* is the process confirming that the model chosen to represent a system or structure is appropriate; this can done by comparing simulation results to experimental data (*'are we building the right product(model)?'*). Verification and validation both focus on the intended use of the model in question; for different uses, the same model might need to be verified and validated differently. The following example illustrates this.

Example: Crash test

An intended use of a model could be to predict the response of a Fiat 500 in frontal impacts against a wall at velocities up to 60 km/h (see Figure 1-2). Validation might consist of predicting the deceleration of the inside compartment to within 10% for crash tests at velocities of 20, 40 and 60 km/h. Once this model is validated by performing the crash tests at the given velocities and measuring the decelerations, it could then be used to predict the decelerations at other impact velocities up to 60 km/h as well. However, this same model could not be used to predict the Fiat's performance at velocities above 60 km/h, or its response to rear-end collisions; for the same velocity range, it would also not be able to predict the behavior of a BMW M5. Therefore the model, once validated, is only useful for its intended use ("predict the response of a Fiat 500 in frontal impacts against a wall at velocities up to 60 km/h"). It might still be accurate in predicting the response of a similar car (e.g., a Ford Ka), but it would have to be validated for that purpose first before being put to use. For a different car, the model might have to be revised entirely before it is validated again. A model's intended use should be specified in detail before verification and validation takes place. Furthermore, accuracy criteria ("within 10%") should be defined.

Figure 1-2: Fiat 500 crash test [1]

1-2 Activities

General approach

Most real-world structures and systems that require simulations are complex. It helps to split the process of designing a model into a number of steps. Figure 1-3 identifies the activities in this systematic approach; the left branch represents the process of designing a model, while the right branch represents the process of obtaining experimental data. The simulation outcome and the experimental outcome are compared for validation of the model. In the following section, each step will be explained in more detail.

Reality of interest

As a first step when designing a model, the *reality of interest* must be defined. The reality of interest is the entire system and its environment; this should all be taken into account when developing the model. The reality of interest is therefore much larger than the system itself, which in the case of an aircraft is basically just its structural components. All factors ranging from weather conditions, collisions with wildlife/ debris and varying runway quality could be incorporated, as well as the passengers. Thus, the reality of interest can be quite broad.

It is important to recognize that most complex systems have a hierarchy. An airplane seat is not impacted by weather conditions, but for the engines this is a crucial factor that needs to be incorporated in its design. Conversely, for the engines the maximum expected weight of a single passenger is not a relevant factor, but for an airplane seat it is. The top-down decomposition of a system helps to identify the various components of the model and will result in a multi-level set of individual models. For each level, the activities in the flowchart shown in Figure 1-3 are repeated, starting with the definition of the reality of interest in each case. For software testing we will also see that hierarchy is an important concept (Chapter 2).

Conceptual model

When the reality of interest for the level/component being modeled is defined, it has to be abstracted into something less complex and more workable. The result of this abstraction is the *conceptual model*. As a first step, one makes assumptions and simplifications which will not degrade the results too much. For instance, when modeling the behavior of an aircraft on short flights, one can generally make the assumption that the Earth is flat while still meeting accuracy requirements of the model. However completely disregarding crosswinds for the sake of simplicity would in most cases lead to an unacceptable error. It requires skill and effort to distill what parts of the reality of interest need to be included in the model and to find out to what extent assumptions affect the result.

An important part of the conceptual model are boundary conditions. This can be clarified with an example of air flow in a wind tunnel. The air flow is considered to be a continuum which leads to certain governing equations as we will see in the next section on the

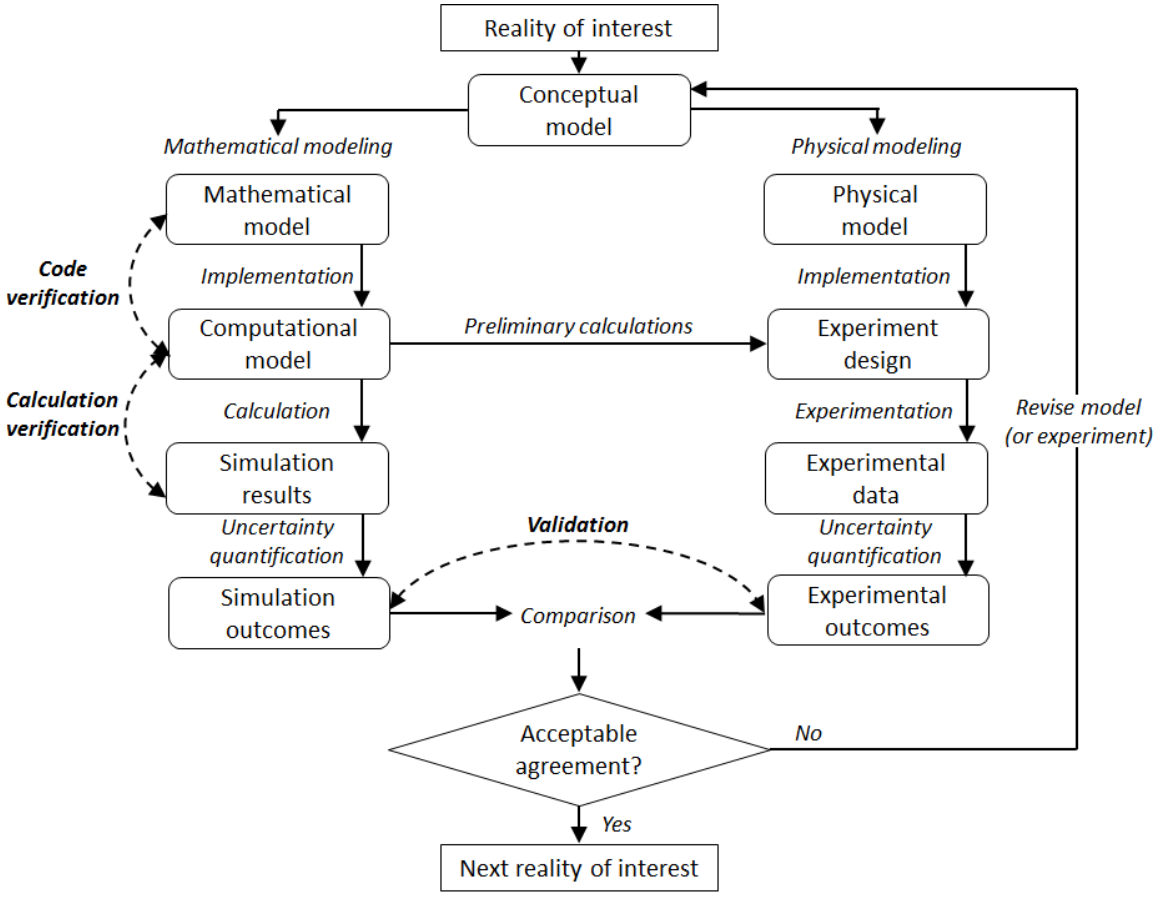

Figure 1-3: Simulation, verification and validation activities [5]

mathematical model. The following boundary conditions can be used:

- no slip at the boundaries
- no flow through the boundaries - the walls of the wind tunnel are not porous
- flow is incompressible

These boundary conditions will be expressed in equations.

Mathematical model

The conceptual model represents a simplified and more workable version of the reality of interest. Once the conceptual model has been formulated, the next step is to describe this conceptual model in a mathematical sense. This is done by formulating governing equations, boundary conditions and initial conditions.

Following the example of flow in a wind tunnel from the previous section the mathematical model for flow of a continuum are the Navier Stokes equation. The boundary conditions corresponding to the assumptions in the previous section are:

- $u_{tan} = 0$ at the boundaries
- $u_{perp} = 0$ at the boundaries
- div velocity $= 0$

The governing equations are likely not be solvable in an analytical way. A numerical method may be required which use discretization or linearization. As such the solution requires iteration (for example, in the case of linearization) or requires discretization (for example, in the case of finite-element modeling). To be able to check if the method is implemented correctly you would like to have an independent method, see the next section on verification. This could be an analytical model. Such a model has the advantage that the solution is known exactly. However, to obtain an analytical solution of the mathematical model, additional assumptions and simplifications may have to be made which means you are not able to check the full numerical model. Another numerical method could be a way to check the correctness. However, a disadvantage is that both models will have numerical error, so any differences between the two methods can not be attributed to one of the models. Note that the distinction between analytical and numerical model does not have to be based on whether you need a computer to solve it. Software such as Maple and Mathematica can manipulate symbolic language to produce closed-form solutions, and you can use a program to plot the results of the analytical model, neither of which makes the model numerical. On the other hand, numerical solutions, i.e. after discretization, can sometimes be solved with pen and paper although this is not practical.

Verification

Deriving the numerical method from the mathematical model and implementing it results in the computational model (Figure 1-3). The implementation step is not simply writing a program and calling it a day. Programs rarely function on the first try, and even after spending a lot of time on debugging it is possible that the program still does not deliver the correct result (just that it delivers *some* result).

This is where the question 'Are we building the product right?' needs to be answered via verification. Verification consists of two parts. First, it is necessary to determine that the code is indeed correct and does not contain any errors (i.e., if the numerical model was implemented correctly). Second, it must also be ascertained that the program is actually calculating the right thing (i.e., if the numerical model itself is correct). These two verification steps are called *code verification* and *calculation verification*, respectively. They will be discussed in more detail in 2.

Validation

The right branch of the chart in Figure 1-3 concerns the development of the experiments that will be used to validate the model. The purpose of validation experiments is to compare the experimental results with the simulation results and thereby assess the accuracy of the numerical model, keeping in mind that experimental data will have errors as well.

One of the most difficult steps in the entire experiment design process is establishing a relationship between validation experiments and the reality of interest. For what set of cases should the model be accurate enough? Before designing and doing an experiment, these cases need to be clearly defined. Experiment design can be quite difficult, but can be simplified by taking into account the results of some preliminary calculations of the computational model. This way a clearer picture may be formed of what needs to be measured and what values can be expected. Once the experiments have been defined, they can be used to obtain raw data. Experiments may need to be repeated multiple times to be able to quantify uncertainty. Once experimental and simulation outcomes have been obtained, a quantitative comparison between the two sets of outcomes takes place.

The scope of this course is not experiment design, but rather performing validation correctly using experimental data that is provided. Data will be provided for the Structural Analysis assignment. For the flight dynamics assignment, data will be collected during the flight test. However, it is important to question the experimental data. How were the data obtained? Is the error known? If not, you may have to come up with a reasonable error margin yourself. You can test data for outliers and remove these. After this, the data can be used for validation. This is done by comparing experimental and simulation data **for the same case**. If the model does not simulate the same case, you are seeing effects of a change in input parameters and you are not evaluating the quality of the model itself. Then, the question is whether the model is validated, given the required simulation accuracy level and error in the data. Such questions are formalized in so-called *hypothesis testing* which is taught in the course Probability and Statistics (WI2180LR-II). To be more specific, you need to formulate a null-hypothesis ('the model is correct') which can be rejected or not based on the data. Incorrectly rejecting a null

hypothesis that the model is correct (*Type I error*) is usually acceptable. However, accepting the null hypothesis that the model while it is not (*type II error*) is usually worse and should be avoided.

1-3 Feedback

Figure 1-3 might create the impression that model design is mostly linear, but this is not the case. The process of designing a model is cyclic, which can be better seen in Figure 1-4. This figure represents the many relations between the main components of model design. The model building, verification, and validation arrows all point in both directions. Verification may lead to the conclusion that either the mathematical model or the computational model, or both, need to be adapted. The same is the case for validation: either the (already verified) computational model needs to be adapted, or the "reality", i.e., the experimental results/ setup. A clear example of how validation can influence the use of a model can be understood from safety factors that are applied to aerospace structures. The safety factor provides a certain safety margin by designing the structure to withstand larger loads than are thought to occur. Part of the safety margin takes into account that the structural analysis modeling is inaccurate, and the value of the safety factor or safety margin can be based on the validation of the simulation.

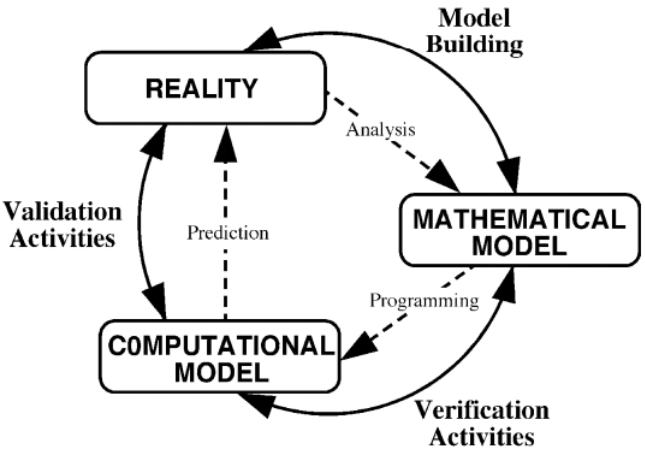

Figure 1-4: The interaction of verification and validation in the numerical modeling process [5]

1-4 Conclusion

This chapter makes clear that verification and validation are crucial for providing trust in the model. It is not enough to know the theory or to be good at programming; you also need to demonstrate the accuracy of your model. Secondly, verification and validation during the model development process can save a lot of time.

Figure 1-3 gives an overview of the main steps taken during the model building process. While

this process is highly systematic, it is important to note that it is most definitely not linear (see Figure 1-4).

The very first step in designing a model consists of defining the reality of interest; this encompasses the entire system and its environment. Large, complex systems consist of many components and have a certain hierarchy. The reality of interest should be defined for each component or level in the hierarchy and each should be modeled separately using the systematic approach shown in Figure 1-3.

Once the reality of interest has been defined, it can be distilled into a workable conceptual model. This model contains all assumptions, approximations and simplifications of the reality of interest, and the effects this has on the accuracy of the simulation.

The conceptual model can then be formulated mathematically in the mathematical model. This model can have an analytical and numerical formulation, or two different numerical implementations, one of which is the one that will produce your final results. Numerical models could involve discretization and linearization.

During the implementation of the numerical model into the computational model, verification must occur. Verification answers the question 'Are we building the product right?' Code verification is the process of finding and removing all errors in the code. Calculation verification is the process of determining whether the numerical model has been implemented correctly. This can be done by calculating the solution to a simplified problem using an analytical model and the numerical model and comparing the results for the same input parameters. Or it can be done by comparing two independent, or partially independent, numerical models. Most likely you will not use one model or calculation, but you will verify parts of the code separately. Still, there is usually one model that is used to verify the largest part of the code. This will be called the verification model in the following; in most cases it will be an analytical model.

Once the computational model has been verified, validation takes place. Validation answers the question 'Are we building the right product?'. This is done by comparing simulation results to "real-world"-data, i.e., experimental results. During verification and validation steps, errors or discrepancies will be found, either in the code or the numerical model itself. The main work in verification is in designing the comparison between analytical and numerical model, and in finding errors when the test shows a mismatch. The main task in validation is in analyzing the differences to find out which assumptions in the numerical model cause these differences. The iterative nature of model building using verification and validation is shown in Figure 1-4.

Chapter 2

Simulation

Simulation is the process of developing a computational model from the conceptual model. A basic outline of this process is shown in Figure 2-1. The conceptual model represents a simplified and more workable version of the reality of interest as explained in the previous chapter in section 1-2. It contains all assumptions and approximations for the expected accuracy loss they will result in. The mathematical model is the mathematical representation of the conceptual model and will be explained in Section 2-1. Finally, the computational model is derived from the mathematical model. This will be used to perform simulations (Section 2-2). In Section 2-4, an example will be introduced concerning the explosion of the first Ariane 5, which will highlight the topics presented in this chapter.

2-1 Physical model

The first step in the simulation process is to define the conceptual model in a mathematical sense. This is done by formulating governing equations based on theory, as well as defining boundary conditions and initial conditions.

Before the physical model can be developed, the following have to be clearly defined:

- Reality of interest
- Intended use of the model
- Required response features
- Required accuracy

A good way to investigate the reality of interest is to draw a free-body diagram. Based on this, and knowledge of the theory the system of equations can be formed. Important choices

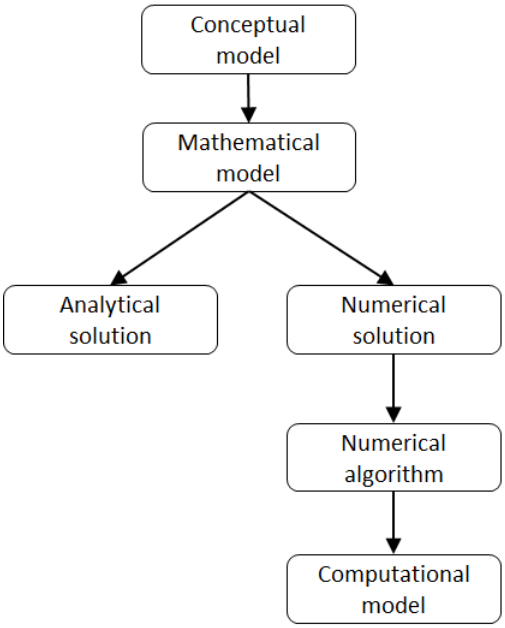

Figure 2-1: From conceptual model to computational model [5]

are made here, such as which physical processes will have significant effect on the results and will be included, and which physical processes are negligible and are excluded from the model. This choice needs to be well documented. But apart from this, the effect of assumptions on the results should also be quantified. We will see later that verification and validation results in discrepancies between benchmark model and numerical model, and between experimental results and model predictions. We can only trust the numerical model if we understand where the discrepancies come from, and how they will behave when the model is applied in a different environment. Therefore, the assumptions should be quantified as much as possible.

2-2 Computational model

The computational model can take different forms: the analytical solution and the numerical solution. At this point it is clear if an analytical solution can be found. In most cases an analytical solution does not exist and a numerical solution is sought. An analytical solution for a simplified case will be useful for the verification.

To obtain a numerical solution, the equations have to be put in algebraic form. This involves discretization of the physics (for example replacing a gradient by a numerical derivative) and discretization of the solution method (for example using a finite step size). The numerical solution is the solution that will be implemented into the program.

Finally the physical model (more specifically the algebraic system) has to be put into a computer program. For your own benefit in verification and for collaborators and future users, it should be documented how the algebraic model has been implemented computationally. You should choose the best form for that. Mathematical equations are precise but hard to

understand. Text is easier to understand but imprecise. For that reason *pseudocode* is sometimes used, where the meaning of a program is described in a mix between normal language and computer commands for loops and if-statements. If you want to show what separately identifiable code blocks there are, and how they are related? you can use a *flowchart*. It gives the reader an immediate overview of the structure of the program.

2-2-1 Effect of assumptions

When the simulation is producing results you can use it to study the effect of some of the assumptions. Here it is useful to distinguish uncertainties and errors. Uncertainties are due to a phenomena that is neglected, or unknown (the unknown unknowns). To estimate the effect is difficult, because it can be estimated, it could also be included in the physical model. For this reason validation is important; some effects can not be simulated. Errors are due to approximations used, for example discretizations, the use of constant coefficients instead of time-varying ones. These errors can typically be estimated. this is useful because the errors typically show up in verification. When you find a discrepancy in verification step you can use the error estimates to explain the discrepancy. If you can not explain it, then it is hard to convince the user that you fully understand the model. A small discrepancy might be due to two effects that happen to cancel eachother for the case of the simulation, but one of them might grow much larger in a realistic situation with different conditions (e.g. stress concentrations, discontinuous forces). A simulation without accuracy information is not useful for the user, and can not be verified or validated.

As example of error estimate, the effect of a first order approximation can be estimated by computing the second order term. The effect of discretization can be estimated by reducing element or step size, a so-called order of accuracy test (see the course Computational Modelling). For certain integration methods a certain decrease of the numerical error with step size is expected. If this is not observed, then some other effects play a role, such as round-off or linearization errors.

To get an idea of the effect of uncertainty in input parameters you can vary the input parameters within the acceptable range for that input parameter. Doing this according to the assumed statistical distribution of the input parameters yield Monte Carlo simulation.

2-3 Software Specification

Next step in the computational model is the translation from the algebraic equations to the software. It is likely that these parts are done by separate people. Or that you are given an existing program. (in practice it is more likely that you are working on an existing program than that you are developing your own program from scratch). Can you start right away with checking if it works properly? No, you need to know what the program is supposed to do; you need to have a *specification* of the program. Even if you are the one making the algebaric equations as well as the computer program, it will be very useful for designing verification tests to have a good specification of the program. There are different ways in which you can give a specification, that will be made clear with an example taken from the book 'Software

Verification and Analysis', by Laski and Stanley (2009) ([13]).

SPECIFICATION : *Given is an array A[1 to n] of integers. The function returns the length of a longest strictly increasing monotone sequence in A.*

It is assumed that you know what strictly increasing means. Still, it is not immediately obvious if [2 2 2] is a strictly increasing sequence, and a 'sequence in A', could refer to its indices or to the elements of array A. Thus, the text is not enough to fully understand what the program is supposed to do. In general, text is ambiguous. The good thing about text is that it is readable. Every text that is added to code is helpful, even when you are the only one using the code!

To give information in a more structured way, think of specification as a model of the program. Here you treat the program more as a black box. You have to describe the interface between what the program is supposed to do, how this is done, and the environment:

- Input parameters (variables that may be used)
- Output parameters (variables that are important to know once the program has been executed)
- Global variables (variables that are considered part of the state)

Additionally you may want to put restrictions on the input and output. For example, you want the program to work for an array with a size up to 100000 elements. Most important is to specify the desired relation between input and output, to have some expression for the output as function of input. Such a relation can be given in mathematical form. Mathematics are very precise, but not easy to read. The code can also be represented with clear text in the form of instructions. This is sometimes called *pseudocode* because it resembles the instructions of a computer program but does not conform to a programming language. An example from [13] is the following:

Compute the set S of all monotonically increasing sequences in the subarray A [1 to n]; Find the longest length of the sequences in S;

Better yet than text might be visuals. That is why we ask in the simulation plan and in the report to add a flow chart, which shows input and output and the relations between them. Such a flow chart will help you to identify missing variables and it will also help you in testing the program. More on that in the next section.

Finally software specification should also include units. The well known case of the loss of the Mars Climate Orbiter shows the importance of units. The software that calculated total impulse provided as output pound-force seconds whereas the trajectory software expected Newton seconds, a factor of 4.45 differences. The spacecraft was much closer to Mars than the wrong calculations showed, and burned in the atmosphere, or skipped towards outer space. This example seems to call for the use of SI units, but it is of course not the case that the engineers were not aware that an imperial or SI unit could be used. The Software Interface Specification required input and output to be given in SI units. Most of the blame is not placed with the person or the company (Lockheed Martin) that committed the error,

but with the system of 'checks and balances'. As a NASA administrator put it [14]: "People sometimes make errors, The problem here was not the error; it was the failure of NASA's systems engineering, and the checks and balances in our processes, to detect the error. That's why we lost the spacecraft."

2-4 Example: Ariane 5

Ariane 5's first test flight (Flight 501) on 4 June 1996 failed due to a malfunction in the control software. About 37 seconds into the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded. The error occurred due to a small oversight: the control program of the Ariane IV had been adjusted to this new rocket; however, the 64-bit values required to correctly represent the state of the Ariane 5 did not fit into the 16-bit space used by the control program. Sixteen bits had been enough to represent the state of the (much smaller) Ariane IV; in the case of the Ariane 5, however, data values were cut off and subsequently incorrectly interpreted by the control program, causing the rocket to veer off course. The only course of action left was to self destruct, causing the loss of its payload, the Cluster satellite mission.

Figure 2-2: Ariane-5 explosion

During the development of launch systems, many risk analyses are conducted that will predict impact footprints. Of course, whenever a launcher explodes, the debris footprint should be outside inhabited areas. The approach we follow in this example is that for worst-case scenarios an explosion of the launcher is simulated at different moments in the trajectory, taking into account different launcher velocities, positions and environmental conditions.

2-4-1 Conceptual model

In formulating the physical model, the reality of interest has to be distilled into a workable model using assumptions and simplifications. Possible assumptions that could be used in the

Ariane 5 example are:

- Atmospheric conditions (e.g., constant density, wind)
- Particle versus finite body (point-mass motion or also attitude dynamics?)
- Aerodynamics (drag only? Will lift be required?)
- (Non-)Rotating Earth
- Spherical or flat Earth
- Constant gravity, or a variation with altitude?

2-4-2 Physical model

Analytical solution

For simplicity we assume that the trajectory of the rocket is ballistic, resulting in parabolic flight. From the diagram in Figure 2-3, the following well-known equations of motion can be derived:

$$V_x = V_{0,x} \tag{2-1}$$

$$V_y = V_{0,y} + gt \tag{2-2}$$

$$x = x_0 + V_{0,x}t \tag{2-3}$$

$$y = y_0 + V_{0,y}t + \tfrac{1}{2}gt^2 \tag{2-4}$$

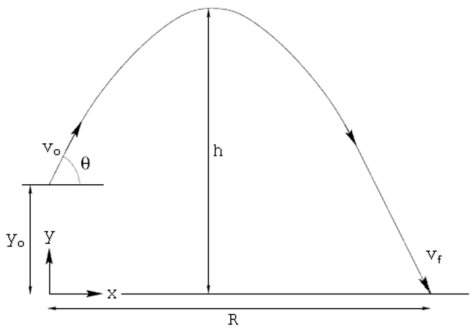

Figure 2-3: Sketch of the situation

These equations represent the analytical solution to a simplified case where, atmospheric effects, wind, etc. are not taken into account and the motion is purely driven by gravity. This solution can be used to predict a single-particle trajectory (2D). To simulate the debris footprint, one can calculate a trajectory for any number of particles.

Numerical solution

To set up the numerical solution we need to define the complete set of non-linear equations of motion, starting from Newton's Laws of Motion. The formulation of these equations depends on the chosen state variables [16].

A possible formulation of the numerical solution is, given a mass point m that moves w.r.t the rotating Earth (with angular velocity ω):

$$\dot{V} = -\frac{D}{m} + g\sin\gamma + \omega^2 r\cos\delta\left(\sin\gamma\cos\delta - \cos\gamma\sin\delta\cos\chi\right) \tag{2-5}$$

$$\begin{aligned} V\dot{\gamma} =& \frac{L\cos\sigma}{m} - g\cos\gamma + 2\omega V\cos\delta\sin\chi + \frac{V^2}{r}\cos\gamma + \\ &+\omega^2 r\cos\delta\left(\cos\delta\cos\gamma + \sin\gamma\sin\delta\cos\chi\right) \end{aligned} \tag{2-6}$$

$$\begin{aligned} V\cos\gamma\dot{\chi} =& \frac{L\sin\sigma}{m} + 2\omega V\left(\sin\delta\cos\gamma - \cos\delta\sin\gamma\cos\chi\right) + \\ &+\frac{V^2}{r}\cos^2\gamma\tan\delta\sin\chi + \omega^2 r\cos\delta\sin\delta\sin\chi \end{aligned} \tag{2-7}$$

$$\dot{r} = \dot{h} = V\sin\gamma \tag{2-8}$$

$$\dot{r} = \frac{V\sin\chi\cos\gamma}{r\cos\delta} \tag{2-9}$$

$$\dot{\delta} = \frac{V\cos\chi\cos\gamma}{r} \tag{2-10}$$

where both position and velocity are defined by spherical components: distance r, longitude τ and latitude δ for the position, and velocity modulus V, flight-path angle γ and heading χ for the velocity. The external forces are the aerodynamic drag D and lift L, whereas g is the acceleration due to gravity. Finally, the bank angle σ is defined to allow for a lift orientation away from the vertical plane.

As the above solution is defined for a single debris particle, we can take the distribution of flight-path angle and velocity into account, as well as wind profiles and shape-depending aerodynamics by simulating each debris particle separately. For each debris particle, the motion is then given by such a set of ordinary differential equations (ODEs).

2-4-3 Computational model

The computational model concerns the dispersion of debris over an altitude range (i.e., not omnidirectional). The particle sizes and masses are distributed randomly. The solution will be obtained by integrating the ODEs given in the numerical solution (Equations 2-5 to 2-10).

Possible output variables are:

- Impact point of debris part (latitude/longitude or the x- and y-position)
- Mass/size of debris part (following from the explosion model – not covered here)

- Number of parts (explosion model)
- Probability distribution of parts (explosion model)

Each of these output variables may be a function of launch direction, type of launcher, environmental conditions, etc.

Possible functional code blocks for the computational model of the Ariane-5 problem could be:

- Global initialization
- Explosion (generate initial conditions and masses for n particles)
- Main loop for each particle:
  - Evaluate environment
    - Atmosphere
    - Gravity field
    - Earth shape
    - Wind
  - Calculate flight parameters (Mach number, state-derived variables)
  - Calculate aerodynamic properties
  - Calculate forces and moments
  - Calculate accelerations (using properties of each particle)
  - Integration of acceleration into velocity and position
  - Calculate (and store) required output
- Loop until stop criterion is met (e.g., $h \leq 0$) (for all particles)
- Write data to file and terminate program

In a report these can be presented in a flow chart with all inputs and outputs (data flows) defined. In smaller flowcharts these data flows should be included in the figure itself. When the flowchart becomes to cluttered, the inputs and outputs can also be listed in a table, clearly identifying the code block they refer to. A typical example of a flow chart for a general-purpose ascent-flight simulator is shown in Figure 2-4.

Note: to save development time, and finding coding mistakes or thinking errors, it is recommended to *always* make a flow chart before starting to program. This is especially useful if more people are working on the same code. This phase is called the architectural design of the software. In an industrial environment, this phase is documented in the so-called Architectural Design Document.

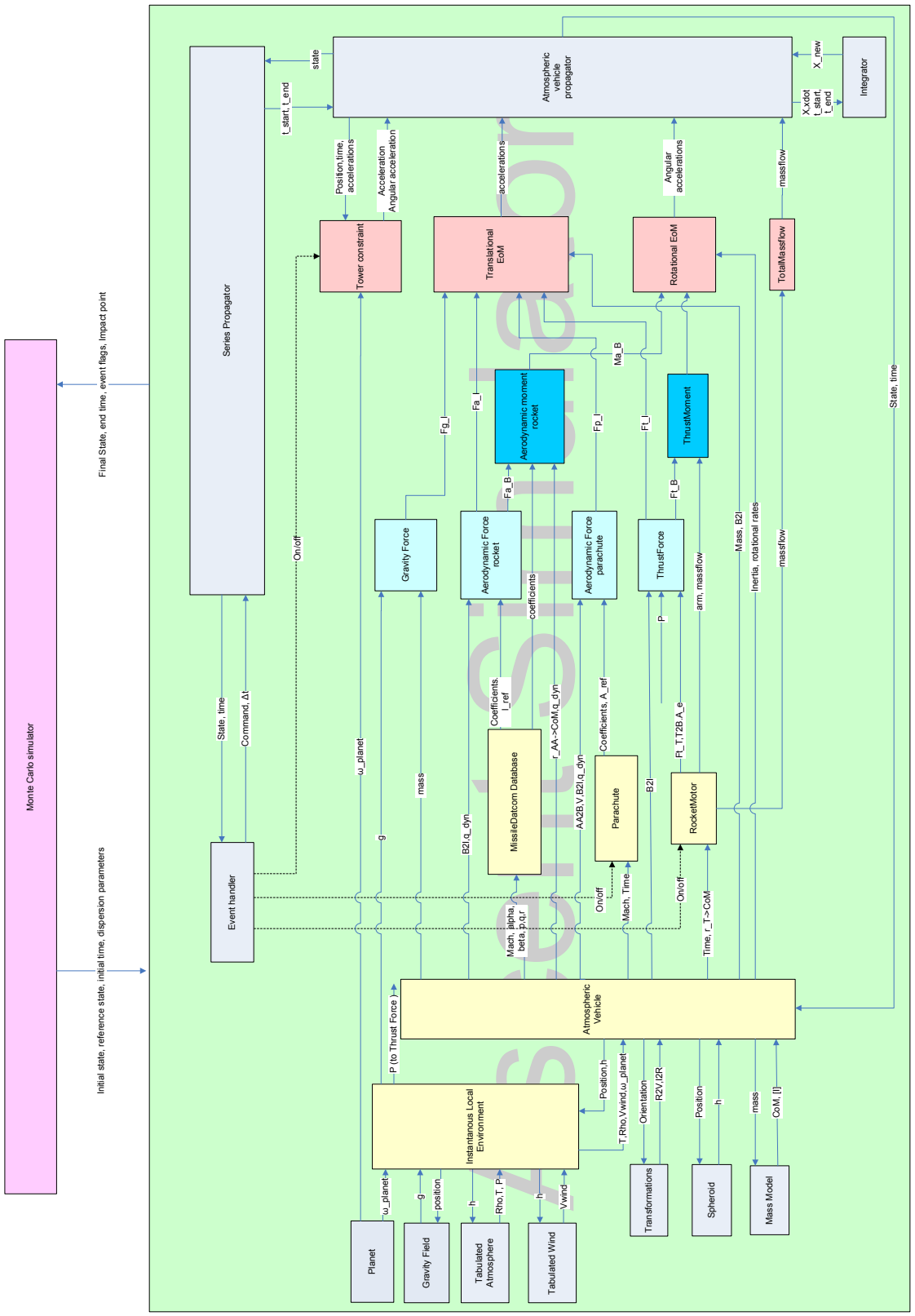

Figure 2-4: Flow diagram of a general-purpose ascent simulator [9]

Chapter 3

Verification

Verification answers the question 'Are we building the product right?' Again, in this course we will focus on building a software simulation. Large computer programs play an important role in industrial and scientific applications. The more people are working on a software package, the more important it is to have a good procedure for verification in place. In this course you will develop a numerical model to simulate the behaviour of an airplane structure and motion of a full aircraft. But also during MSc thesis work considerable amount of time is spent on code verification and validation. Formal and automated procedures are out of the scope of this course, but some concepts will be introduced to make clear that there are simple methods for verifying software in a systematic way.

To find out if the program is working properly, you first need to know what the program is doing as described in the previous chapter in section 2-3. Verification consists of two parts. First, it is necessary to determine that the code is indeed correct and does not contain any errors (i.e., if the numerical model was implemented correctly); this is explained in Section 3-1. Second, it must also be ascertained that the program is actually calculating the right thing (i.e., if the numerical model itself is correct); this is presented in Section 3-2. Finally, in Section 3-3 verification will be applied to the Ariane-5 example introduced previously in Section 2-4.

3-1 Code verification

Code verification is the process of finding and fixing the programming errors present in the code. Initially this can be done with the help of your compiler; many compilers will give error messages when the syntax is incorrect (e.g. due to a typo or a forgotten semicolon). Once all syntax errors have been found, it is also important to find mistakes that may not lead to syntax errors, but will make your program function differently than was intended.

With clear specification, you should have an idea of what the software is supposed to do. To

test this, an intuitive, and widely used, way is to try different inputs and see if the output is what you expect. For example, increasing strength of a material should lead to decreased deformation. Can extensive testing determine if a program is correct? Technically speaking it can't, at least not for a program of a meaningful size. The number of possible inputs and possible paths through the code is often simply too large to test each one. It looks like a purely semantic discussion, but testing is a way to show *incorrectness* of a program. Looking at it another way, verification can be positive, negative, or inconclusive. Testing gives you an inconclusive answer. The program is likely to be correct because you didn't find any error. It is important to realize that the amount of testing is usually insufficient to conclude with absolute certainty that the program is completely free of errors.

Is there another way to guarantee that a program is free of errors? Such a procedure is called *formal verification* and this is outside the scope of the course. Formal verification requires good knowledge of the solution of the problem, and it uses formal methods of mathematics. A lot of attention in literature is given to such methods, partly because they can be automated. However, so far testing is the most common way for software verification. This means that luckily humans still have an important role in software development, because testing requires intuition and common sense.

Software testing should have the following properties [13]:

- high potential to detect faults
- practically feasible to execute
- it should result in high confidence in the software when all tests are passed

The last item needs more explanation because it is less obvious than it seems. A test usually treats the program as black-box, only considering its input and output. For example, given the following tests for our monotone increasing function:

t1 = [5,4,3,2,1]

t2 = [1,2,3,4,5]

According to the program introduced at the end of section 2-3 for *t1* the outcome of the function is 1, this is the longest sequence that is increasing. For *t2* the outcome is 5. These are two distinct types of inputs, another one would be a sequence which is increasing first and then decreasing. Common sense would say that if you test all distinct types of inputs the test covers a large part of the code. This is a way to quantify how good the testing is, by keeping track of how many instructions are involved in the test, or how many branches. For example, test *t1* covers 88 percent of the instructions, and with test *t2* this increases to 92 percent. 67 and 75 percent of branches are covered, respectively.

Debugging

Few activities are more frustrating than debugging code can be. The fun is usually in creating something, not in endless testing. People express frustration that they spend 2 hours on coding

for an assignment and 10 hours on debugging. Some of that frustration can be taken away but part of it is a fact of life. To give it a positive spin, you could say that debugging gives you a deeper understanding of the program. In devising tests you need to know what the program is supposed to do, for debugging you need to know how the program is meant to achieve its goal. At this stage you will find that comments in the code are particularly useful. Mostly, debugging is done by going through the program line by line. Modern compilers allow to walk through the code while having access to current values of each variable (e.g. debugging mode in MATLAB or Python compiler). More basic compilers require the user to write output to a log file to analyze afterwards.

Experienced programmers usually have a feeling of where the error might be and test that particular part of the code. The activity of debugging can be done in a somewhat more systematic way. A first and obvious step is narrowing down the part of the program where the error could occur, by looking only at the statements that are executed (because of if-statements, for example, not all parts of the code have to be executed in every test). This is part of *static analysis*, analysing without executing the code. Other questions that can be asked are, what is the path that leads to a certain instruction? What are the dependencies, i.e. which variable may change as a results of a change in a certain instruction? In *dynamic testing*, actually running the code, easy strategies are available to narrow down where the error should be: check the output of the program halfway. If you found out that the error occurs in the first or second half of the program, you can divide the search area in half, so-called *binary searching*.

The verification process is designed to check the software implementation of the numerical solution. Division of the numerical analysis program in units gives a good starting point for verifying the complete process. Each different unit can be verified separately through a so-called *unit test*, resulting in a clear indication of which parts of the program function correctly. During this unit verification process one also checks for singularities, for example, what happens if the angle is 90 degrees? After the unit tests, larger tests are performed in which several units are combined. Finally, a system test is performed in which most of the code is tested.

Verification is a vital part of projects, for software and products. In industry a large part (up to 50%) of a projects budget in terms of both time and money is spent on verification and validation [4]. It is therefore not surprising that you will also spend a large amount of time verifying and validating your programs.

Wikibook Computer programming principles [21] lists several steps to reduce debugging time which are more of an advice on having the right character, such as starting with the right attitude: assume that your program will contain mistakes. Test systematically by changing one thing at a time. Indeed it takes discipline to keep testing systematically, especially when the part of the code looks simple. However, experience will show that errors usually arise in an 'easy' statement, such as an array index. When you work with multiple people on software you have the advantage of separating the coding from the testing by assigning these tasks to different people.

3-2 Calculation verification

Once code verification has taken place and it has been determined that no programming errors remain, the simulation can be run to produce some results. However, you still do not know whether your numerical model is correct; just that it has been programmed correctly. What if the underlying theory is incorrect? *Calculation verification* is the second part of the verification process. The purpose of calculation verification is to determine whether the numerical model does not contain any errors or inaccuracies from numerical methods. Calculation verification can be done by using the program to solve a simplified problem of which the exact solution is known, a benchmark solution. It is important that the model is complicated enough that all terms in the governing equation are activated. For example, for CFD problems the advection term should be activated, which is not the case if you use a Poiseuille problem as benchmark for example ([22]). Comparison with an analytical model shows the error in the numerical model, and the error in the solution procedure, for example numerical integration. This means that discrepancy can have multiple causes, such as linearization error or round-off error. To learn more about the errors introduced by individual steps in the procedure it is useful to create a graph of the discretization error as a function of step size and see if this shows expected behaviour: numerical discretization should be convergent. A simple example, taken from the lecture notes of AE2220-I [23] is shown in figure 3-1. The error is halved for a doubling in number of steps over the interval, which results in a line in the log-log plot.

However, this will only be possible for a simplified model. It is also possible to verify a numerical model with an alternative numerical model. However, in that case differences between the two solutions will emerge that are more difficult to ascribe to one of the solutions.
.

Again, depending on the complexity of the program, it is smart to not only do calculation verification for the entire program, but also for individual code blocks (i.e., "unit tests") to make sure that they do not individually contain programming errors that might not be detectable when the program in its entirety is under consideration. These errors may accidentally cancel or obscure other error for the specific problem being solved, but may lead to incorrect results for another situation. After successful unit testing, the units can be integrated into a single program and a larger test can be performed in which a larger part of the model is tested.

3-3 Example: Ariane 5

Once a model has been developed for the debris dispersion for the Ariane 5 example (see Section 2-4), a possible way to verify the model is presented in the following. First, unit tests can be performed:

- For an atmosphere model it is known that the density at surface level ($h = 0$ m) should be approximately $\rho = 1.225$ kg/m^3. Also one can find density as a function of altitude in references. Then the "atmospheric properties" block could be checked by simply giving it an input of $h = 0$ m, and checking whether it outputs (approximately) $\rho = 1.225$

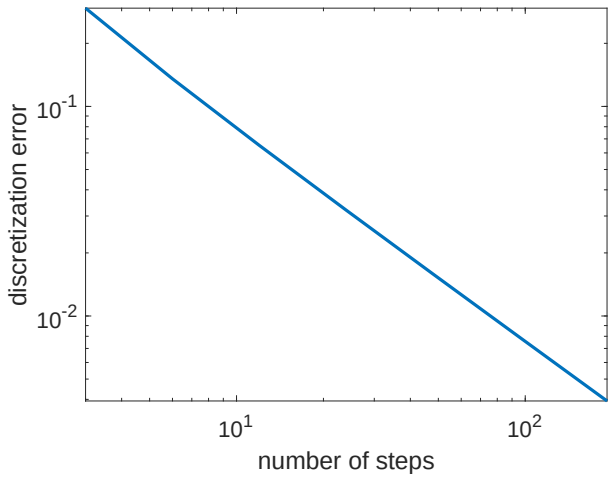

Figure 3-1: discretization error versus number of steps when forward Euler-Cauchy method is used to discretize the initial value problem $y' = x - y/2$, $y(0) = 1$ over the interval $[0, 3]$. Taken from Example 8.3 from the lecture notes of Numerical Analysis AE2220-I [23]

kg/m^3. The same can be repeated for other altitudes and the solutions can be compared to tabulated values found in literature.

- Equations of motion: a 1 N force is applied to a 1 kg mass, the result should be a constant acceleration of 1 m/s^2. This gives a linear increase of velocity and quadratic increase of positions. Providing such a unit input to a block is a good "sanity check"; if the block gives an obviously illogical answer, you can immediately start debugging.
- Singularities should always be checked for (usually zero-input, or 90° for angles). These can provide problems that are hard to spot because they don't occur for standard cases.

After successful unit testing, the units can be integrated into a larger program unit and a *system test* can be devised to check that the units interface properly.

For the system test, the analytical solution can be used to calculate a particle trajectory (time and location of impact, particle velocity) for a simple situation. The computational model should then reproduce this solution using the exact same input and assumptions (e.g., no drag). Values for impact time, position, velocity and trajectory shape can be compared to verify that the computational model is working correctly for the simplified case, see figure 3-2. The differences should be at the level of the computer precision if assumptions for both models are the same.

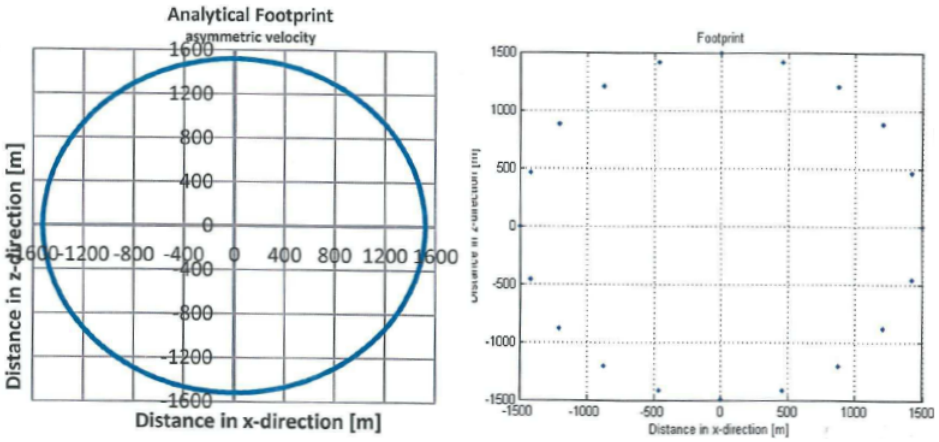

Figure 3-2: Analytical vs. numerical footprint with launcher flight path angle 90°and symmetric velocity increment due to explosion

Chapter 4

Validation

The goal of validation is to determine the predictive capability of a computational model for its intended use. This is done by comparing computational predictions (i.e., simulation outcomes) to observations (experimental outcomes). Prerequisites for validation are:

- Clear definition of the model's intended use
- Verification has been performed

If the computational model predicts the experimental outcomes within the predetermined accuracy requirements, the model is considered validated for its intended use. Validation experiments are discussed in Section 4-1, where experiment design and selection are explained in more detail. Section 4-2 deals with assessing the uncertainty of validation experiments. Section 4-3 discusses validation in general, and Section 4-4 applies validation to the Ariane 5 example introduced in Section 2-4.

4-1 Validation experiments

Validation experiments are performed to generate data to compare to the outcomes of the computational model. The situation being tested, as well as the initial conditions, boundary conditions, and input parameters should be prescribed for the experiment to be useful. The validation experiments and measurement set should be designed to leave as few unknown parameters as possible.

4-1-1 Experiment design

Experiments can have many purposes and are generally focused on assessing system performance relative to certain criteria or on exploring a system's response to certain inputs.

Therefore, the measurement sets obtained from experiments might be different from the measurement sets required for validation. For example, a test may show that a component fails at a higher load than a predefined acceptable threshold and therefore the component should be considered acceptable for use. However, the test may not have taken into account deformations as a result of the applied force because that measurement was not required for the purpose of the experiment. If both the component-failure measurement and the deformation-measurement are necessary to validate a numerical model, the test measuring only component failure cannot be used for validation.

Validation experiments and measurement sets should be designed such that as few parameters are left unknown as possible. If some significant parameters are not measured, calculations have to be performed to compare with the experiments by varying the values of those parameters. You cannot arbitrarily select a parameter value within the accepted range and base the validation comparison on that measurement only. If all calculation results are within the acceptable tolerance for validation, only then validation may be claimed. However, experiments also have their inaccuracies. It is always important to consider where experimental data come from and how trustworthy they are.

4-1-2 Measurement selection

Selecting the quantities to measure should be based on the response features of interest (ideally the same as the output of the computational model). When possible, these features should be measured directly rather than derived from other measurements. For example, if velocity can be measured directly, that approach would be better than integrating a measurement of acceleration or differentiating a measurement of displacement. However, it may be useful to measure other quantities as well to ascertain that the measurement of the quantity of interest is correct (e.g., measuring displacement or acceleration to corroborate measurement of velocity).

Additionally, measuring additional variables and locations during the experiment other than those specified in the validation requirements may provide extra confidence in the model if they agree with simulation results. Although some quantities may be of secondary importance, accurate predictions of these quantities by the numerical model provide additional evidence that the model accurately calculates the primary quantities. For example, confidence in a model whose primary purpose is to predict the central deflection of a beam is enhanced if it also accurately predicts the strains or displacements along the entire length of the beam - even if the central deflection is the only quantity relevant for the model's intended use. Thus, validation experiments should produce a variety of data so that multiple aspects of the model can be assessed.

4-1-3 Sources of error

It is important to calibrate all measurement equipment that will be used for validation experiments and to carefully document their inaccuracies. If certain equipment is sensitive to the environment, it should be calibrated in the same environment in which the experiments are done. If the environment changes rapidly during the test, the equipment's sensitivity

to the environment should already have been established so the measurement data can be corrected to account for this sensitivity. Additionally, one needs to determine and account for such effects as compliance and inertia of the test equipment if these effects contribute to the measurements.

4-1-4 Redundant measurements

Redundant measurements are necessary to establish the precision of the experiments and improve the quantification of uncertainty in the measurements. One approach for obtaining redundant measurements is to simply repeat the test. The test-to-test scatter could have contributions from differences in specimens, material properties, boundary conditions, measurement equipment and data acquisition. For example, consider bending tests on several members of a set of beams where the response is measured with strain gauges mounted on the tension and compression surfaces. Not only would the beams be different, but they might be installed slightly differently in the testing machine, and the strain gauges would show different scatter in location and orientation. Another approach for obtaining redundant measurements is to perform the same test repeatedly on the same specimen. This option is also used when testing is very expensive or not many test specimens are available.

4-2 Uncertainty assessment

When determining the uncertainty of an experiment, the effects of measurement errors, design tolerances, construction uncertainty etc. are quantified. It is important to report the uncertainty in measurements in validation experiments so the simulation results can be judged appropriately.

Errors are usually classified as either random or systematic. A random error contributes to the scatter of the data in redundant measurements or repeat experiments. Random errors are inherent to the experiment and can be reduced with additional testing. Sources include variations in measurement location and equipment tolerances which typically have a normal distribution. Systematic errors can produce a bias in the experimental measurements that is difficult to detect and estimate. Sources include calibration errors and data acquisition errors.

4-3 Validation

In the validation process one aims at matching the simulated data with real-life data and at explaining possible deviations from these data. These differences should be related to the assumptions made to obtain the conceptual/computational model, and ideally quantified. A sensitivity analysis may help in quantifying uncertainties in, for instance, input data. Validation data may consist of flight manuals, wind-tunnel data, test data from laboratory tests, or data from an actual flight test. Since it is expected that validation data are not in the same format as the output data, the first step would be to convert the real-life data to a format in which it can be analyzed and compared to the numerical output data. This would

mean matching the real-life data format with the numerical output data format.

4-4 Example: Ariane 5

Once the computational model for the Ariane 5 example has been programmed (Section 2-4) and verified (Section 3-3), it also has to be validated against real-world data.

The results of the simulation can be compared with the actual footprint of the Ariane 5. Figure 4-1 shows a sketch of the analytically calculated, numerically simulated and actual debris footprint of the Ariane 5. The discrepancies between the actual and both calculated footprints can be explained by the assumptions made to obtain both the analytical and numerical solutions. The analytical solution, for instance, did not consider drag and wind, while the numerical solution did. Additionally, the numerical solution also took into account that not all particles were dispersed at the same time; this is why the footprint's contour looks jagged. However, the numerical solution did not consider such nonlinear effects as wind gusts, etc., which may have caused the actual footprint to be off-center with respect to the analytical and numerical footprint.

It is important to relate discrepancies to assumptions and if possible, quantify them. Otherwise no conclusions can be drawn about the validity of the model, and it is unclear where adjustments could be made if necessary. A conclusion can be drawn about the model's adequacy based on previously defined accuracy requirements.

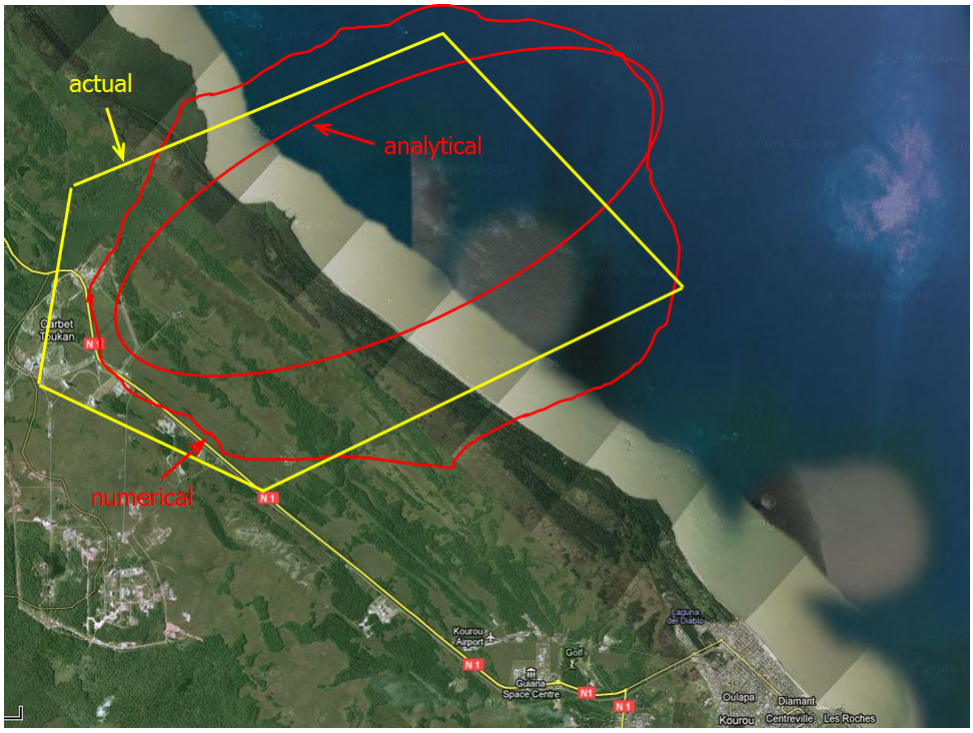

Figure 4-1: Analytical vs. numerical vs. actual footprint

Chapter 5

Simulation, Verification and Validation in Practice

In the following chapter, two examples of real-life applications of simulation, verification and validation will be presented. First, in Section 5-1 (based on [11]), the load-bearing capacity of a new carbon fiber reinforced plastic (CFRP) material is predicted and tested. This same material is also used in the A380. Section 5-2 (based on [3]) explains the safety measures taken by Airbus and Engine Alliance based on simulation and testing.

5-1 Aircraft structures - a new CFRP design

The allowable load bearing capacity of thin-walled stringer-stiffened carbon fiber reinforced plastic (CFRP) panels loaded in compression is currently limited by its buckling load [11]. The extension to a new design scenario - to permit postbuckling under ultimate load - requires validated simulation procedures.

The reduction of structural weight without adverse effect on costs and structural life has been the main target for research on primary aircraft parts. A possible approach to reach that target is to use CFRP material for fuselage structures. For thin-walled structures loaded in compression to permit postbuckling up to ultimate load could further improve structural performance.

To visualize the approach, typical load-shortening curves of two panel designs are depicted in Figure 5-1. Buckling (usually local buckling of the skin between the stringers) and collapse (maximum load) are indicated by dots and crosses, respectively. Design 1 is constrained by the limit load (the first buckling load is close to limit load), which is common practice. Design 2, with reduced structural weight, is constrained by an ultimate load definition (ultimate load is slightly below collapse). This new design scenario, utilizing the load carrying capacity in the postbuckling area, requires accurate simulation up to the deep postbuckling region.

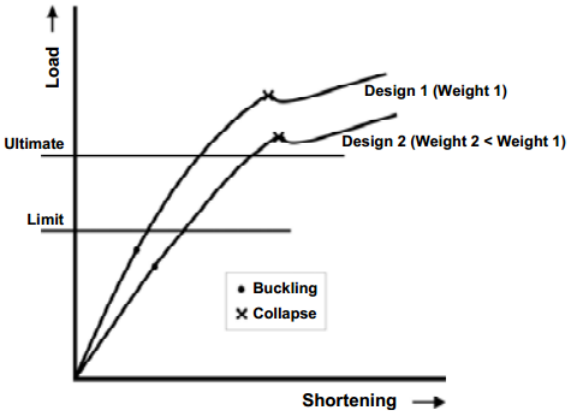

Figure 5-1: Different structural design scenarios [11]

Simulation can only be relied upon when it is subjet to an a validation procedure, which requires extensive experimental data.

As numerical model the finite element software ABAQUS/Standard was used, which is capable of non-linear analysis to simulate the postbuckling behavior. ABAQUS is a commercial software package which itself has been subject to verification. It is first used to guide the set-up of the experiments, as described in the following section.

5-1-1 Testing

Pre-test analyses

Due to the experiments being time-consuming and therefore expensive, a substantial amount of work was spent on detailed pre-test analysis and planning. Several nonlinear analyses were conducted with the finite element software ABAQUS/Standard to identify the postbuckling behavior. ABAQUS is a software package which itself has been subject to verification. at this stage ABAQUS was used for a sensitivity study which can be assumed part of simulation, that helped to prepare the validation tests. The preliminary numerical studies of boundary conditions along the longitudinal edges of the panels revealed that the clamping width of the attached longitudinal supports has a significant influence on the postbuckling behavior. Based on these pre-test analyses, a good understanding of the nonlinear behavior of the CFRP test structures was obtained, which determined the placement of sensors (e.g. strain gauges) and examination of critical areas within the experiments. An important conclusion is that pre-test simulation (with a verified model) and planning is crucial for effective validation of numerical results.

The experiment

Eight stringer-stiffened panels were manufactured, and tested in the DLR buckling test facility. The blade-shaped stringers were manufactured separately and subsequently bonded to the

skin. Ultrasonic inspections were conducted to ensure the quality of the panels, especially the bonding of skin and stringer flange. The locations of the strain gauges were based on the aforementioned pre-test analysis. An important assumption in the modelling is the effect of imperfections in the material, and part of the validation focused on establishing the effect of these.

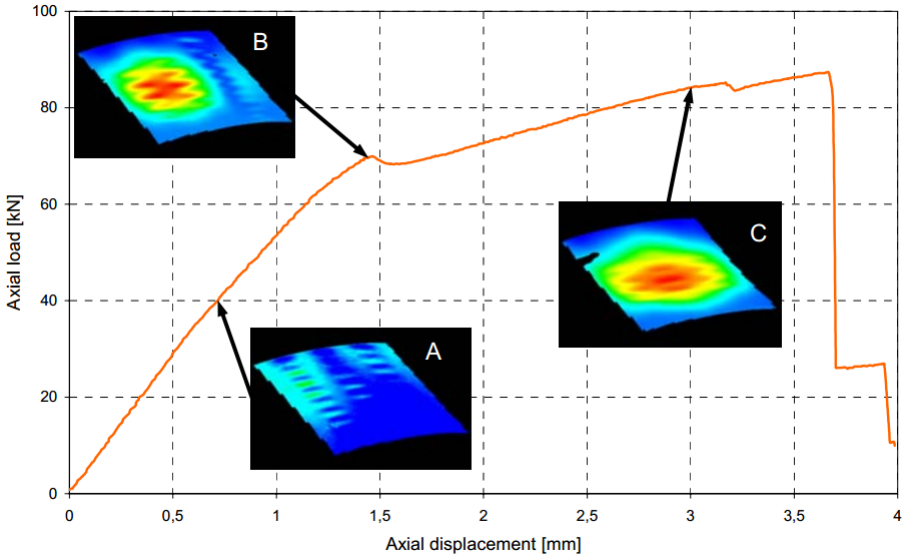

Figure 5-2: Load-shortening curve and selected deformation patterns (experimental data) [11]

During the experiment, an optical measurement system (ARAMIS) was utilized to capture digital images of the deformed structure. After postprocessing, these images show the displacements at nodes of a fine optical mesh representing the surface of the panel. Three of these displacement patterns at characteristic locations (local skin buckling (A), 2/3 versus 1/3 global buckle (B) and single global buckle (C)) along the load-displacement curve are shown in Figure 5-2.

5-1-2 The computational model

To analyze the pre- and postbuckling behavior of the panels, the finite element tool ABAQUS/Standard was again employed. Figure 5-3 depicts part of the geometry of the FE-model.

The approach to conduct the FE-analysis consists basically of four stages (Figure 5-4): preprocessing, a linear eigenvalue analysis, a nonlinear analysis utilizing the Newton-Raphson technique with adaptive/artificial damping, and finally post-processing.

Figure 5-5 depicts the load-displacement curves which were obtained utilizing the analysis procedure described in Figure 5-4 with and without initial geometric imperfections. Comparable to the experimental results displayed in Figure 5-2, characteristic deformation patterns (local skin buckling (A), 2/3 versus 1/3 global buckle (B) and single global buckle (C)) are shown for the FE analysis without imperfections. It can be seen that the FE curve closely follows the experimental results but can still under- or overestimate strength compared to

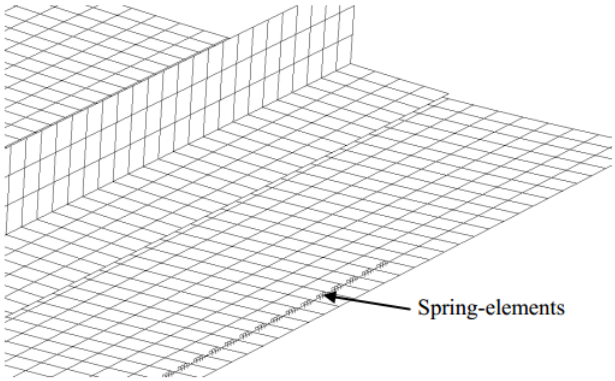

Figure 5-3: Details of the FE model [11]

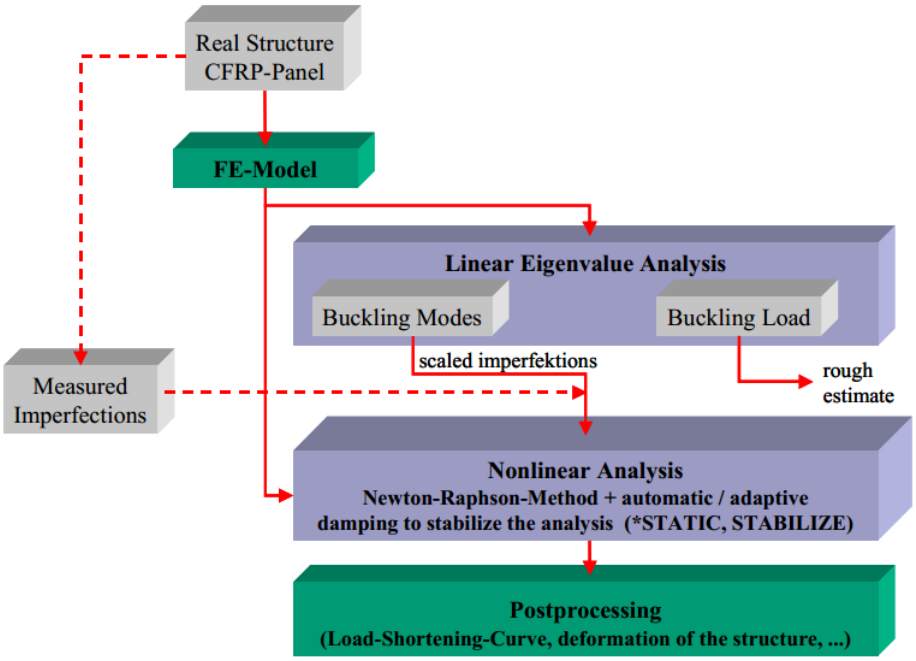

Figure 5-4: Analysis procedure of a CFRP panel [11]

the experimental results. The differences are deemed within acceptable range. A final and important step of validation is to understand the differences between simulation and experiment. To this extent imperfections where introduced in the modelling, to see if these lead to variation in the modeling results that explains the differences with respect to the experiments. This is not the case; the influence of imperfections is small.

A possibility at this point would be to extend validation to different configurations of stringer stiffened CFRP panels (stringer pitch, stringer height, radius of curvature).

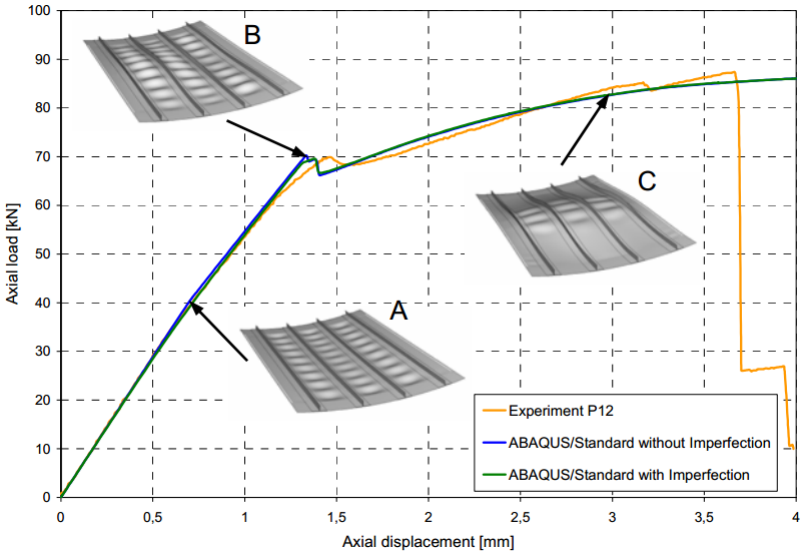

Figure 5-5: Load-displacement curve, nonlinear finite element analysis versus experiment [11]

5-2 The Airbus A380

The new CFRP panel design discussed in Section 5-1 is used on the Airbus A380. The following section will detail V&V procedures used by Airbus and Engine Alliance.

5-2-1 Safety case

The main objective of the aircraft certification process is to make sure that the aircraft complies with airworthiness requirements. In order to meet most regulatory guidelines, the aircraft manufacturer must build a safety case as a means of documenting the safety justification of a system. The safety case is an important document used to support certification. It contains a set of arguments supported by analytical and experimental evidence concerning the safety of a design. Items that should be included in the safety case include specification of safety requirements, results of hazard and risk analysis, the verification and validation strategy, and results of all validation and verification activities).

5-2-2 A380 System Safety Assessment Process

The system safety assessment process includes requirements generation and verification which supports the aircraft development activities. This process is normally conducted during concept and design phases. It determines whether all possible associated hazards have been addressed in the aircraft functions and systems. For the first time on a large-scale commercial aircraft, Airbus implemented aircraft system safety assessment processes at the aircraft level. The V&V process played an important part in the A380 safety system assessment process. In the A380's structural design computational solid mechanics play a large role in designing the aircraft structure. The processes of V&V in computational mechanics are intended to provide and quantify confidence in the numerical modeling. This was challenging because Airbus introduced more composite materials in major aircraft parts. An example is the case of the new CFRP panel which was discussed in Section 5-1.

5-2-3 V&V of the GP7200 engine

Reliability at entry into service is one of the requirements that the world's major airlines expect from manufacturers. Engine Alliance has followed the V&V process alongside with the design in order to achieve the target reliability of the engine before entry into service. Design and manufacturing phases of the GP7200 engine are:

- Phase 0: Detailed design
  - Code verification between mathematical model and computational model
  - Calculation verification between computational model and simulation outcomes
- Phase I: Engine certification
  - Engine validation testing
    - 747 flying test bed
    - Endurance
    - Fan blade out
    - Bird strike
- Phase II: Service readiness
  - FAA certification
  - A380/ GP7200 first flight
- Phase III: Service and support
  - GP7200 entry into service

The simulations used during the detailed design phase (Phase 0) of the GP7200 engine include mission/propulsion cycle simulation, aerothermodynamics modeling, structural assessment and controls modeling and evaluation. Code verification and calculation verification was conducted in this phase and resulted in simulation outcomes. To establish confidence

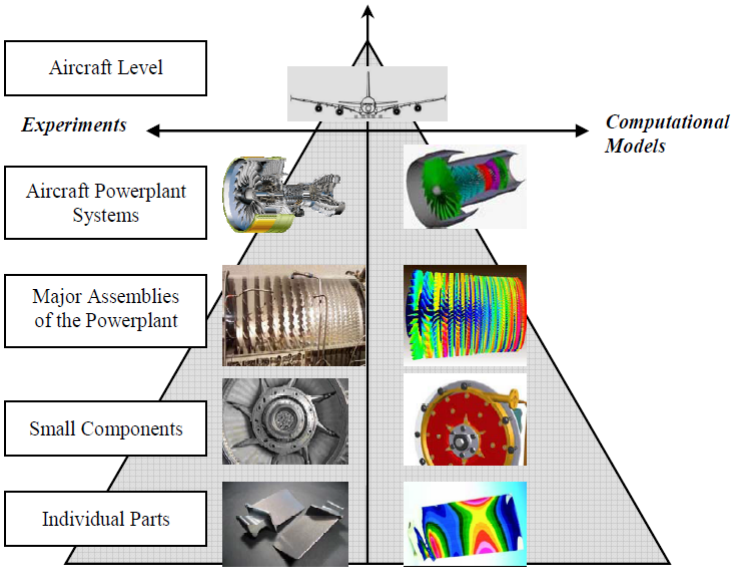

Figure 5-6: GP7200 engine validation pyramid [3]

that the computational model was correct, experiments were conducted. Comparison of the computational model predictions with the results of these physical tests were performed as validation activities. The validation and certification tests demonstrate compliance with the specification and flight safety regulations.

Figure 5-6 shows the validation pyramid. The validation process of the GP7200 engine was performed on different levels, from the aircraft level to individual parts. The Engine Alliance conducted the GP7200 component rig test as part of the reliability strategy to continually validate performance and mature technology. On the aircraft level, the EA validates the GP7200 engine's endurance by performing flight tests which run on ETOPS (extended operations). Additionally, a Boeing 747 flying test bed was used to evaluate engine throttle response and altitude characteristics, and optimize clearance control schedules. On the engine level, the GP7200 was tested in a wind tunnel in order to survey the mechanical stress levels in engine components and vibrations. Intensive tests of the major assemblies of the engine also took place, such as swept fan testing, core engine testing, HP compressor and HP turbine testing. Finally the individual parts validation could be done in the test lab for the stress analysis of the materials and structures.

Chapter 6

Simulation Plan

6-1 Developing a simulation plan

Simulation, verification and validation activity should be planned before starting to develop the model. In that sense. the simulation plan is more a project plan, but we will follow the name simulation plan. The general layout of a simulation plan can be as follows:

1. Introduction (including problem statement)
2. Group organization
3. Method (using equations, pseudocode, or a flow chart)
4. Numerical solution
5. Verification model (analytical model)
6. Verification
7. Validation

A simulation plan explains and documents all the steps you will take when actually developing a simulation. As explained in section 2-3, to build the code and the test the code it needs to be stated *what* needs to be solved, and *how* it needs to be solved. This is especially important when the problem statements, coding and debugging are split over different persons, as in this course, and most likely in simulations developed in industry. The description needs to be unambiguous and readable, and it is best to use a combination of methods to convey the information: text, equations and figures. In the following chapter, several items in the simulation plan will be explained in more detail . Section 6-1-1 explains what needs to be defined and introduced first. Section 6-1-2 explains how group organization should take place,

discusses Gantt charts and what should be included in the simulation plan (a Gantt chart is not mandatory). Section 6-1-3 details what should be included in terms of the mathematical model. Section 6-1-4 explains how the verification model should be included. Section 6-1-6 describes the creation of the input and output flow chart and Section 6-1-5 describes the numerical solution. Verification and validation are explained in Sections 6-1-7 and 6-1-8, respectively.

6-1-1 Introduction

Problem statement

The problem statement defines what the model is being built for. Examples of a problem statement for the Ariane 5 example:

1. What is the probability that debris of a failed rocket launch will fall outside the safety zone of a certain area?
2. What is the optimal launch direction such that the debris footprint after a launch failure will not overlap with inhabited areas?
3. Given uncertainties in environmental and operational conditions, does the nominal safety zone still hold?

6-1-2 Group organization

Group organization (in the scope of this course) entails dividing the work into work packages and estimating the time it takes to complete each package. Group members are assigned to work packages in a logical and fair way such that every group member spends approximately the same amount of time on the project (on average). It is not necessary to assign only one person per task but it is required that each student works on at least two out of the three main elements simulation, verification and validation. Work should be divided such that everyone can work parallel to each other to complete their tasks (i.e. horizontal work division). Keep in mind that programming and verification go hand-in hand because testing is easiest when small parts are tested and added to the code. When planning, keep in mind that that time spent on tasks such as programming and debugging is hard to predict, and take into account that there might be delays. However, several verification tests can be designed an executed in parallel to other tasks.

Typical tasks are:

- Verification model
- Numerical solution
- Verification
- Validation

Table 6-1: Group organization and time spent

	Student	S1	S2	S3	S4	S5	S6
Work package	Time (h)						
WP1	12	5	3	4			
WP2	8				3	3	2
WP3	16				4	5	7
WP4	8		4	4			
WP5	4	4					
WP6	6	1	2			3	
WP7	8			2	3		3
Total time (h)	62	10	9	10	10	11	12

- Reporting

A simple work package distribution is shown in the Table 6-1. In the simulation plan, provide a task distribution which shows the work you have put in the simulation plan **on a per person basis** and also give an expected task distribution for the future work related to the report **on a per person basis**.

Making a Gantt chart can be very useful at this time, but it is not a required component of the simulation plan. A Gantt chart is a type of bar chart that shows the different activities and elements of the work packages, with time along the horizontal axis. The order in which the different activities need to be completed are shown and the progress can be tracked. The Gantt chart can be made in a dedicated program such as Microsoft Project or in a more general table program such as Microsoft Excel. The different tasks can be assigned to the different group members and the dependencies of the different tasks can also be highlighted. Determine possible bottlenecks and delays beforehand, and plan such that these can be mitigated. The whole group waiting for one person to solve a bug in the code is not an efficient use of time (and uncomfortable for the programmer).

An example of a (section of a) Gantt chart is shown in Figure 6-1. This Gantt chart is part of the Gantt chart of a master thesis project on the cost of delay of an aircraft and the environmental impact of delay. The order in which the tasks have to be completed are clearly visible as are the different stages. As this Gantt chart is for a large project the duration of particular tasks can be large. For a specific phase a more detailed Gantt chart can be made.

6-1-3 Method

Assumptions

Assumptions should be listed, and most importantly, their effect on simulations results should be quantified as much as possible. This can be done by using the numerical model for a sensitivity study as described in 1. Where this is not possible, the assumptions can maybe be ranked according to their expected impact. Some assumptions have a major impact on the results (e.g., ignoring wind when it is known that usually there are relatively strong winds).

	Task Name	Duration	Start	Finish
1	**MSc. Thesis**	**212 days**	**Mon 18-2-13**	**Tue 10-12-13**
2	**Initial Phase**	**20 days**	**Mon 18-2-13**	**Fri 15-3-13**
3	Research Goal Definition	4 days	Mon 18-2-13	Thu 21-2-13
4	Generate Project Description	1 day	Thu 21-2-13	Thu 21-2-13
5	Project Planning for Next Phase	1 day	Thu 21-2-13	Thu 21-2-13
6	**Literature Phase**	**52 days**	**Mon 18-3-13**	**Tue 28-5-13**
7	Perform Literature Research	40 days	Mon 18-3-13	Fri 10-5-13
8	Write Literature Report	20 days	Wed 1-5-13	Tue 28-5-13
9	Hand-in Literature Report	0 days	Tue 28-5-13	Tue 28-5-13
10	Kick-off review	0 days	Tue 14-5-13	Tue 14-5-13
11	**Detailed Research Phase**	**80 days**	**Wed 19-6-13**	**Tue 8-10-13**
12	Detailed definition of research scenarios	5 days	Wed 19-6-13	Tue 25-6-13
13	Familiarize with software	5 days	Wed 26-6-13	Tue 2-7-13
14	**MILP modelling**	**65 days**	**Wed 3-7-13**	**Tue 1-10-13**
15	Define en-route sector	10 days	Wed 3-7-13	Tue 16-7-13
16	Set-up sector schedule	5 days	Wed 17-7-13	Tue 23-7-13
17	Model different aircraft	5 days	Wed 24-7-13	Tue 30-7-13
18	Incorporate conflict detection & resolution algorithms	15 days	Wed 31-7-13	Tue 20-8-13
19	Mid-term review	0 days	Wed 31-7-13	Wed 31-7-13
20	Set-up emission index calculations	10 days	Wed 21-8-13	Tue 3-9-13
21	Develop fairness algorithm	10 days	Wed 4-9-13	Tue 17-9-13
22	Read-in BADA performance diagrams	5 days	Wed 18-9-13	Tue 24-9-13
23	Finetune different research scenario's	5 days	Wed 25-9-13	Tue 1-10-13
24	Perform tests and generate results/plots	5 days	Wed 2-10-13	Tue 8-10-13
25	**Reporting Phase**	**20 days**	**Wed 9-10-13**	**Tue 5-11-13**

Figure 6-1: Gantt chart

Others have a minor impact on the results, but are not negligible (e.g., for a low-altitude rocket explosion the curvature of the Earth is less important).

Reference frames

When constructing a model, it should be clearly defined in which reference frame calculations are done. When converting from one reference frame to another are made, errors are easily made because coordinate axes and angles generally use generic symbols (for example α for angles). Thus, it is important to clearly define frames and angles.

Equations of motion

The equations of motion should for both the analytical and numerical solutions need to be given in separate state and auxiliary equations, as well as the assumptions for which the equations are valid. The state variables and external forces/ moments should be defined clearly. Finally, the input data (e.g. boundary or initial conditions) needed to solve the equations should be stated.

6-1-4 Numerical model (your own)

All assumptions made to arrive at the verification model should be stated, as well as their impact on the solution, as described above in section 6-1-3 . All equations used should be derived or a proper reference should be given. The verification model will be used later during the verification stage to solve a simplified version of the problem at hand.

6-1-5 developer model

Similarly to your own model, all assumptions made to arrive at the developer model should be clearly stated, as well as their effects. Further guidelines will be provided in the assignment. The simulation plan should describe the functional blocks and the algorithm. Additionally, a flow chart has to be made.

6-1-6 Flow-chart design

The flow chart will show what information is required at which stage to complete each step of the simulation. For the purpose of dividing the work and testing the software, the simulation needs to be divided in logically separated functional blocks, which will correspond to different units of code (which are sometimes called functions, subroutines, objects). Each block should have only one (or possibly two) “functionalities” to avoid subroutines/functions with many lines of code. A large function is difficult to test and debug. Special attention should be paid to data flows and input/output interfaces (e.g., a variable should not be used in a function if it has not been calculated before). Also variables should be initialized first.

The flow chart makes clear what the inputs and outputs are for each of the different units so that the finished units can be easily merged and the merging can be tested after each unit has been tested. A flow chart is also very useful in larger projects where you are not working face-to-face all the time. The overview can help in structuring the reporting and can shorten the report text because you can refer to items in the flow-chart. An example flow chart is presented in Figure 6-2.

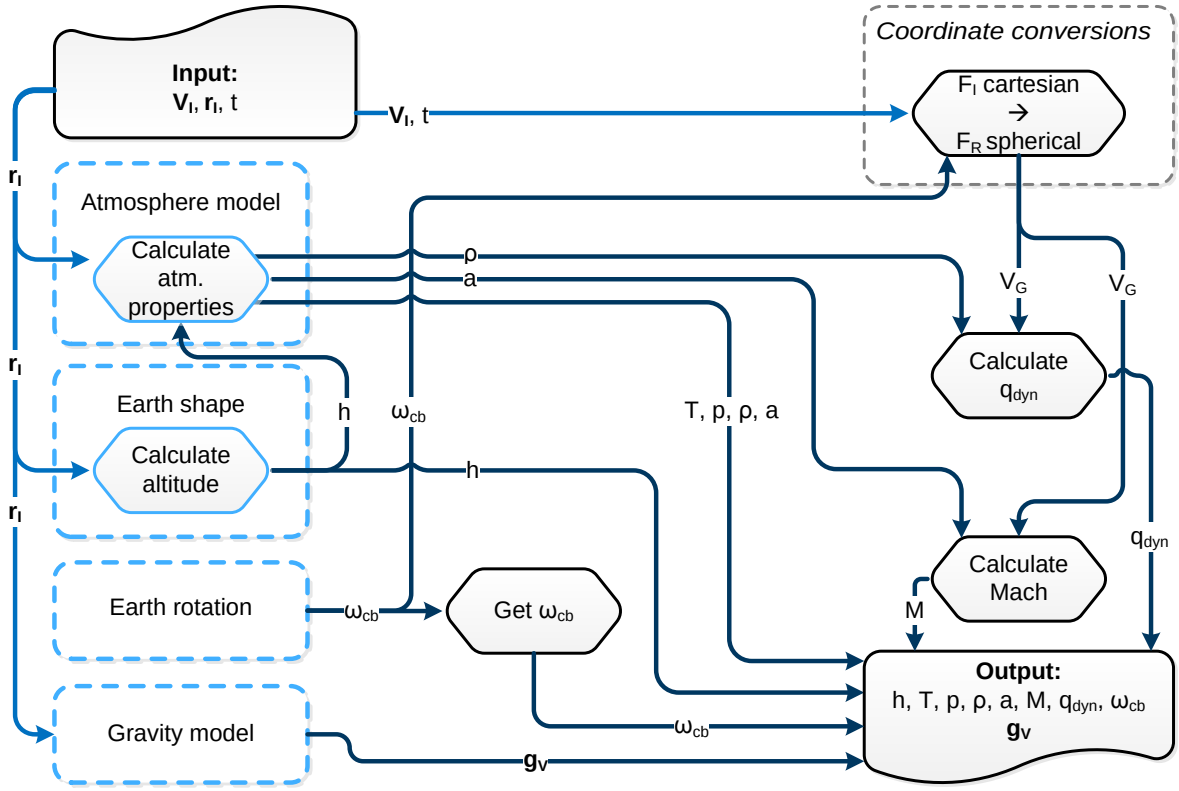

Figure 6-2: Example flow diagram of an environment module, taken from a flight simulator.

The module depicted by the flow chart is an environment module, which is part of a simulator to study entry trajectories into the Earth's atmosphere. Because this module is part of a much larger software, it is necessary to indicate the input (inertial state variables and time) and output (main results from this module) variables to and from this module. Individual code blocks that can be discerned are:

1. Calculate atmospheric properties
2. Calculate altitude
3. Gravity model
4. Coordinate transformation from the inertial (Cartesian state) to the rotating frame (spherical coordinates)
5. Calculate dynamic pressure

6. Calculate Mach number

The complexity of most of these units is low, and verification can thus easily be done with hand calculations (Units #2, #5, and #6). Verifying Unit #1 can be done by comparing the output with documented values in the literature, or data from (meteorological) institutes. The same would apply to Unit # 3, although usually this model is relatively simple for re-entry studies and could also be checked by hand calculations. Note that if one would need a more complex gravity model, or a different atmosphere model, the modular layout of the software allows for an easy replacement of the corresponding units, but attention should be paid to possible changes in the input/output interfaces. It is clear from this example that verification does not simply entail comparing one model to another. Every unit should be verified as well as the integration of multiple units.

6-1-7 Verification

In the verification part of the simulation plan, tests have to be proposed that give high confidence that the developer model is accurate. It should be made clear why these tests are chosen (see section 3-1) and where the test data is coming from (e.g. hand calculations, literature). Tests of small units typically result in two numbers that agree to computer precision, but larger tests might show discrepancies with respect to the reference data because no reference data is available or an analytical model can not be generated for exactly the part of the program that is being verified. It should be explained why you expect that discrepancies arise and how they affect verification (for example, which error is considered acceptable for verification?). Verification will take place for many parts of the software and will point at errors in the code. A strategy should be given for locating where errors in the code are, how to correct them, and how to test if the errors are eliminated.

6-1-8 Validation

The validation chapter in your simulation plan should include a description with the experimental data (which is to some extent provided with the assignment). A plan should be formulated to address discrepancies between the provided data and the output from the developer model and relate them to the effects of the assumptions for the developer model. Additionally, a plan has to be formulated for addressing discrepancies found after validation. When do you consider them acceptable, given the effects of assumptions you estimated before? If the discrepancies are not acceptable, can you improve the accuracy of the numerical model, for example with denser discretization? For the Flight Dynamics Assignment it is more difficult to improve the model. Here it is asked to modify parameters in the model by trial-and-error to better fit to the data. This is no longer part of validation, it is *model improvement*, where the data are used to improve the physical model. At that point the data can no longer be used as independent validation.

Chapter 7

Technical Report

After you have completed the simulation plan, the project work proceeds along the lines you have put down in the simulation plan. A well thought-out and documented simulation plan covering each of the steps simulation (building), verification and validation will save you time during the development of the simulation.

A typical report that will summarize the project work that you did, will cover the following points:

1. Introduction
2. Problem analysis
3. Verification model solution
4. Numerical solution
5. Verification
6. Validation
7. Conclusion

The different sections will be briefly described below.

Introduction

As with all other reports the introduction presents the purpose of the report, its main focus and the problem statement.

Problem analysis

This section should discuss what governing equations are to be used for numerical model and what assumptions have been made.

Verification model

This section should briefly present the theory that has been applied in the model that was suggested or described in the assignment. Most important is to discuss the different assumptions that have been made and to quantify (if possible) their effects on results. Results of the verification model can be presented together with those of the numerical model in the section on verification.

Verification

Show how you verified the separate modules of the program and the integration of separate modules. Explain the test cases and how they cover all or all reasonable paths through the code. Discuss and explain the differences that you encounter between the numerical model and the verification model results.

Validation

This section discusses the validation of the numerical results obtained by the full simulation program with the reference data. The most important part of validation is the discussion and explanation of differences, and how these related to assumptions in the modeling or in obtaining the data.

Numerical solution

This section should present the numerical results insofar they are asked in the assignment, together with their accuracy based on verification and validation.

Conclusion

Link to the problem statement and summarize the main findings and recommendations.

Besides the above, include a table with division of tasks plus time spent per person per work package.

Bibliography

[1] Ultimate Car Blog 2012, Fiat 500 gets three stars in NHTSA safety test, Available: `http://www.ultimatecarblog.com/2012-fiat-500-gets-three-stars-in-nhtsa-safety-test/`, Last accessed 07-02-2014.

[2] AIAA, 1998, *Guide for Verification and Validation of Computational Fluid Dynamics Simulations*, AIAA G-077-1998, American Institute of Aeronautics and Astronautics, Reston, VA.

[3] ARP 4754, *Certification and considerations for highly-integrated or complex aircraft systems*, Systems Integration Requirements Task Group AS-1C, ASD SAE April 10, 1996.

[4] Andersson, C., Runeson, P., *Verification and validation in industry-a qualitative survey on the state of practice*, Proceedings of ISESE, 2002, 37-47.

[5] ASME, 2006, *ASME Standards Committee on Verification and Validation in Computational Solid Mechanics*, The American Society of Mechanical Engineers, New York, NY.

[6] ASME, 2006, *Guide for Verification and Validation in Computational Solid Mechanics*, ASME V&V 10-2006, The American Society of Mechanical Engineers, New York, NY.

[7] Boehm, B.W. (1984). *Verifying and Validating Software Requirements and Design Specifications*, IEEE Software, January: 75-88

[8] The European Cooperation for Space Standardization, 2009, "*Verification*", ECSS-E-ST-10-02C, ECSS, `www.ecss.nl`

[9] Engelen, F.M., "Quantitative risk analysis of unguided rocket trajectories", MSc. thesis, Delft University of Technology, 2013.

[10] IEEE (2004). *IEEE Standards for Software Verification and Validation*, IEEE Standard 1012-1998.

[11] Kling, A., Degenhardt, R., Klein, H. Teßmer, J. and Zimmermann, R., "*Novel stability design scenario for aircraft structures -simulation and experimental validation*", Proceedings of the 5th International Conference on Computation of Shell and Spatial Structures, June 1-4 2005, Salzburg, Austria

[12] Ko, A.J. and Myers, B.A., *A framework and methodology for studying the causes of software errors in programming systems.* Journal of Visual Languages & Computing, 16(1), 41-84.

[13] Laski, J., Stanley, W., *Software verification and analysis: An integrated, hands on approach*, 2009, Springer Science & Business Media.

[14] Oberg, J., (1999), *Why the Mars probe went off course [accident investigation]*, IEEE Spectrum 36(12), 34–39,

[15] Megson T.H.G., *Aircraft Structures for Engineering Students*, 4th Edition 2007, Elsevier

[16] Mulder, J.A., Staveren, W.H.J.J van, Vaart, J.C. van der,Weerdt, E. de, in 't Veld, A.C., and Mooij, E. "*Flight Dynamics: Lecture Notes*, Delft University of Technology, Delft, 2013.

[17] Kapurch, S.J., ed., "*NASA Systems Engineering Handbook*, Diane Publishing, 2010.

[18] NASA, 2008-2009, *NASA Independent Verification & Validation Program Value Report*

[19] NASA Independent Verification & Validation Program, February 2009, *Independent Verification and Validation*, IVV 09-1 +

[20] U.S. Department of Defense, May 13, "*DoD Modeling and Simulation (MS) Verification, Validation and Accreditation (VV&A)*", DoD Instruction 5000.61, Defense Modeling and Simulation Office, Washington, DC.

[21] `https://en.wikibooks.org/wiki/Computer_Programming_Principles`, Last accessed 02-2019.

[22] Roache, P.J., (2019), *The method of manufactured solutions for code verification*, in: Computer Simulation Validation, 295-318, Springer.

[23] Klees, R., R.P. Dwight, (2020), *Applied Numerical Analysis (AE2220-I)*, Lecture Notes,