

***Modellierung und Simulation eines Raketenflugs im Vergleich von
einstufigen und zweistufigen Raketen***

Studienarbeit



für das 4. Praxissemester
des Studienganges Mechatronik
an der Dualen Hochschule Baden-Württemberg Mannheim von

Felix Bräunling

11.01.2016

Bearbeitungszeitraum

12 Wochen

Matrikelnummer, Kurs

2034474, TMT13AM2

Ausbildungsfirma

Schaeffler Technologies AG & Co. KG

Betreuer der Dualen Hochschule

Prof. Dr. rer. nat. Rolf Litzenberger

Selbstständigkeitserklärung

Erklärung

gemäß §5 (3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 22. September 2011.

Ich habe die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegeben Hilfsmittel und Quellen verwendet.

Eppelheim, den 12. Februar 2016

Felix Bräunling

Abstract

In this student research paper a model for a rocket flight is developed. The goal is to give an answer to the question, whether a single-stage rocket or a two-stage one is better suited for reaching greater heights, with the result, that single-stage rockets are capable reaching greater heights because of their reduced weight compared to two-stage rockets. This fact is easily shown while optimizing two-stage rockets: the result is converging towards a single-stage rocket solution.

Vorwort

Die vorliegende erste Studienarbeit entstand während des fünften Theoriesemesters an der Dualen Hochschule Baden-Württemberg Mannheim am Standort Eppelheim. Ziel war die Modellierung und Simulation einer Rakete zur Beantwortung der Frage, ob zweistufige Raketen oder eine einstufige Rakete vorteilhafter ist beziehungsweise wie eine zweistufige Rakete für einen optimalen Flug aufgebaut sein muss.

An dieser Stelle möchte ich Herrn Prof. Dr. rer. nat. Rolf Litzenberger für die Betreuung meiner Studienarbeit und seine Unterstützung danken.

Inhaltsverzeichnis

Selbstständigkeitserklärung	I
Abstract	II
Vorwort	III
Inhaltsverzeichnis	IV
Abkürzungsverzeichnis	VII
Symbolverzeichnis	VIII
1 Simulation einer Rakete	1
1.1 Vergleich von zweistufigen mit einstufigen Raketen	1
1.2 Grundbegriffe der Raketentechnik und des Raketenfluges	1
1.2.1 Aufbau von modernen Raketen	1
1.2.2 Der Flug	6
1.3 Konkretisierung der Fragestellung	8
2 Physikalische Grundlagen eines Raketenflugs	9
2.1 Begrenzung der Betrachtung	9
2.2 Bestimmung der Position	11
2.3 Bestimmung der auf die Rakete resultierenden Kraft	12
2.3.1 Schub	13
2.3.2 Luftwiderstand	15
2.3.3 Gravitationskraft	16
2.4 Bestimmung der Umgebungswerte	16
2.4.1 Umgebungstemperatur	16

2.4.2 Umgebungsdruckruck	18
2.4.3 Dichte des umgebenden Mediums	19
3 Rechnerische Umsetzung	20
3.1 Modellierung der Rakete	20
3.1.1 Raketenteile	22
3.1.2 Tanks und Antriebe	23
3.2 Modellierung der Umwelt	25
3.2.1 Der Planet	25
3.2.2 Die Atmosphärenschichten	26
3.2.3 Die Atmosphäre	27
3.3 Umsetzung der Formeln	28
3.4 Speicherung der gewonnenen Daten	28
3.5 Ablauf der Simulation	29
3.5.1 Ablauf eines Simulationsschritts	29
3.5.2 Setup der einer Simulation	31
4 Parameter der Simulation	33
4.1 Die Aerobee 150	33
4.1.1 Geschichte der Aerobee 150	33
4.1.2 Aufbau der Aerobee 150	34
4.2 Umgebungsbedingungen	36
5 Ergebnisse der Simulation	37
5.1 Simulation der einstufigen Rakete	37
5.1.1 Die Startphase	38
5.1.2 Die Flugphase	43
5.1.3 Die Wiedereintrittsphase	44
5.2 Vergleich verschiedener zweistufiger Raketen	44
5.2.1 Gründe für die unterschiedliche Simulationsergebnisse	46

5.2.2 Schlussfolgerung aus den Ergebnissen	49
6 Validierung der Ergebnisse	52
7 Weitere Verwendung der Simulation	54
Abbildungsverzeichnis	XI
Quellcodeverzeichnis	XII
Literaturverzeichnis	XIII
Anhang	A

Abkürzungsverzeichnis

- A150 Aerobee 150
- ANFA Anilin-Furfuryalkohol-Treibstoff
- CSV kommaseparierte Datei (Comma-separated values)
- OOP Objektorientierte Programmierung (Object-oriented programming)
- RCS Reaction Control System
- RFNA Rot rauchende Salpetersäure (Red fuming nitric acid)
- VA Vought Astronautics

Symbolverzeichnis

A_n	m^2	Fläche der Antriebsdüse
A_{Rakete}	m^2	Querschnittsfläche der Rakete
a	$\frac{K}{m}$	Atmosphärischer Temperaturgradient
a_{Rakete}	$\frac{m}{s^2}$	Beschleunigung der Rakete
c_w	1	Widerstandsbeiwert
d_{Rakete}	m	Abstand des Raketenmassepunkt zum Planetenmassepunkt
F_D	N	Luftwiderstand
F_G	N	Gravitationskraft
F_{Res}	N	Resultierende, auf die Rakete wirkende Kraft
F_T	N	Schub
h_{Rakete}	m	Höhe der Rakete über der Planetenoberfläche
m_{Planet}	kg	Masse des Planeten
m_p	kg	Masse des Treibstoffes
\dot{m}_p	$\frac{kg}{s}$	Änderung der Treibstoffmasse
m_{Rakete}	kg	Masse der Rakete
p_{amb}	Pa	Druck in der Umgebung
p_{h_0}	Pa	Druck am unteren Ende der Schicht
p_n	Pa	Druck an der Antriebsdüse
pos_{Rakete}	m	Position der Rakete
T_{amb}	K	Umgebungstemperatur
r_{Planet}	m	Radius des Planeten
T_{h_0}	K	Temperatur am unteren Ende der aktuellen Schicht
t	s	Zeitschritt
v_{Rakete}	$\frac{m}{s}$	Geschwindigkeit der Rakete
v_e	$\frac{m}{s}$	Geschwindigkeit der Abgase am Antrieb

α *rad* Winkel der Rakete zur x-Achse ρ_{amb} $\frac{kg}{m^3}$ Dichte des umgebenden Mediums

1 Simulation einer Rakete

1.1 Vergleich von zweistufigen mit einstufigen Raketen

Schon im 16. Jahrhundert verwendete der Feuerwerkstechniker Johann Schmidlap eine Technik, die heute als „Multistaging“ bezeichnet wird. Im Gegensatz zu einstufigen Raketen, bei denen Antrieb, Tank und Nase mit Nutzlasten in einem Teil zusammengefasst sind beziehungsweise nur ein abtrennbarer Tank mit Antrieb vorhanden ist, werden hier mehrere Stufen verwendet. Unter einer Stufe versteht man einen oder mehrere Antriebe mit den dazugehörigen Tanks, die gleichzeitig in Betrieb genommen werden. Diese Technologie findet heutzutage bei fast allen modernen orbitalen und teilweise suborbitalen Raketenflügen Anwendung. So benutzen die russischen Soyuz-Raketen und die für die Apollo Missionen verwendeten Saturn V-Raketen jeweils drei Stufen (siehe [1]). Angesichts dessen stellt sich die Frage, ob mehrstufige Raketen einen Vorteil gegenüber einstufigen Raketen besitzen und wie sie aufgebaut sein müssen, um optimale Flüge zu absolvieren.

1.2 Grundbegriffe der Raketentechnik und des Raketenfluges

Im Folgenden werden für das Verständnis der Arbeit wichtige Grundbegriffe der Raketentechnik sowie des Raketenflugs erklärt. Des Weiteren soll auch auf den typischen Aufbau einer Rakete eingegangen werden.

1.2.1 Aufbau von modernen Raketen

In Abbildung 1.1 ist der typische Aufbau einer Rakete dargestellt. An der Spitze der Rakete befindet sich die sogenannte Nase. Sie enthält die Nutzlast und Telemetrieinstrumente. Darunter folgt die Stufe der Rakete, die zuletzt zündet. Sie besteht aus einem Tank, der mit Brennstoff und Oxidationsmittel gefüllt ist. An dessen Ende, geschützt in der Zwischenstufe, befindet sich der Antrieb für diese letzte Stufe, der für größere Höhen ausgelegt ist. Dazu wurde sie für die Treibstoffverbrennung bei geringem Druck und 5

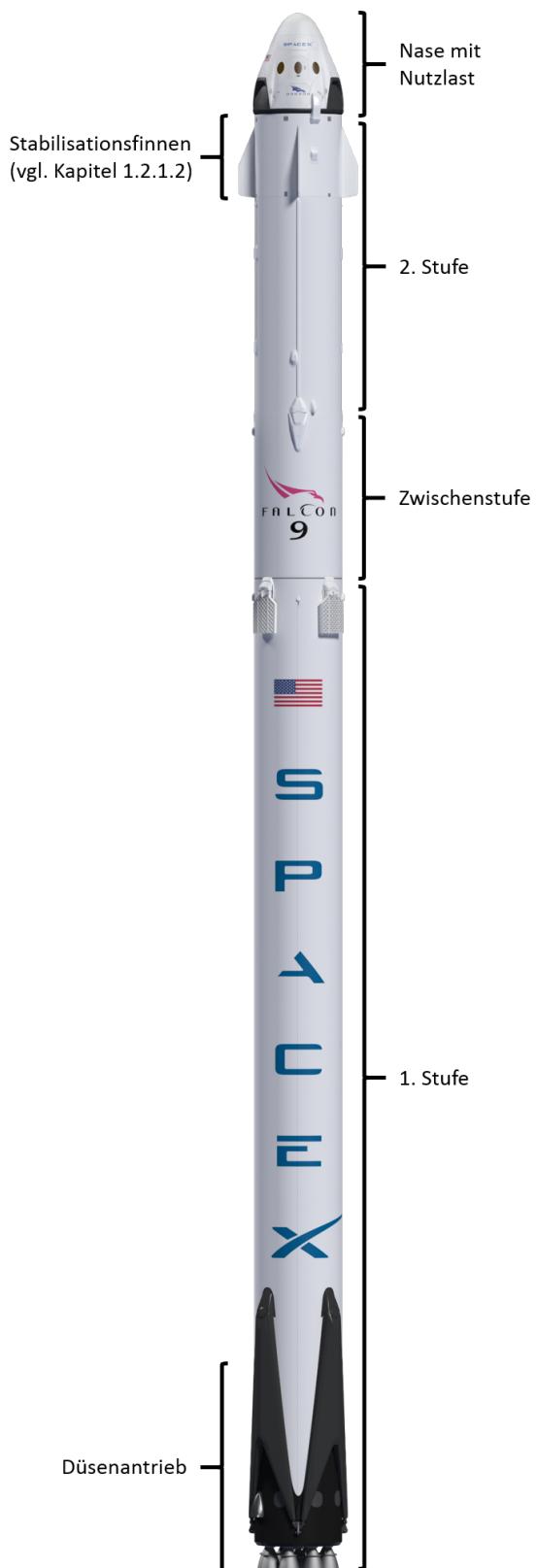


Abbildung 1.1: Falcon 9 Rakete von Space X mit gekennzeichnetem Aufbau nach [2]

geringer Temperatur ausgelegt, die in höheren Atmosphärenschichten vorherrschen. Die Zwischenstufe dient der Abtrennung der ersten Stufe und dem Schutz des Antriebs der zweiten während des Fluges. Sie erfolgt entweder über ein hydraulisches System oder eine gezielte Absprengung.

Abgeschlossen wird die Rakete nach unten von der ersten Stufe. Diese muss den meisten Schub aufbringen, da sie sämtliche weiteren Stufen sowie die Nutzlast beschleunigen muss. Des Weiteren ist sie auch für die Überwindung des Luftwiderstandes verantwortlich. Aufgrund des hohen Treibstoffverbrauchs brennen diese Stufen allerdings nur kurz (siehe [2]).

1.2.1.1 Der Antrieb

Der Antrieb liefert den für die Beschleunigung der Rakete nötigen Schub. Bei Raketen kommen dabei, je nach Einsatz, verschiedene Technologien zur Anwendung.

Die einfachste Form stellen dabei Feststoffantriebe dar. Diese Antriebsart lässt sich schon bei Silvesterraketen finden. Sie wird aber auch mit anderen Brennstoffen in der Raumfahrt eingesetzt. Bei Feststoffantrieben wird ein Granulat aus Treibstoff, Oxidations- und Bindungsmittel gezündet, das dann kontrolliert abbrennt. Die dabei entstehenden Gase werden durch eine Düse gebündelt und gerichtet abgeführt; hierdurch wird der Schub erzeugt. Diese Art Raketeantrieb kann hohe Schubkräfte für kurze Zeiträume bereitstellen und wird deswegen hauptsächlich als Booster beim Start eingesetzt. Der Brennstoff im Booster wird dabei von unten nach oben verbrannt bis eine leere Hülle übrig bleibt. Auf Abbildung 1.2 ist der Start eines Space Shuttle zu sehen; bei den weißen Antrieben an den Seiten handelt es sich um Feststoffbooster (siehe [1, 3]).

Für die zweite Stufe werden überwiegend Flüssigtreibstoffantriebe verwendet. Hierbei werden entweder ein Treibstoff (Mono-Propellant) wie Wasserstoffperoxid oder ein Gemisch aus zwei Treibstoffen (Bi-Propellant) bestehend aus Brennstoff und Oxidationsmittel verwendet. In letzterem Fall werden beide Mittel erst im Antrieb miteinander



Abbildung 1.2: Start eines Space Shuttle mit aktiven Feststoffantrieben aus [4]

gemischt. Diese Antriebe erweisen sich als die effizientesten Brennstoffantriebe, die zum Einsatz kommen. Der orangefarbige Tank in Abbildung 1.2 sowie die drei Hauptantriebe des Space Shuttles sind Bi-Propellant-Antriebe, die Wasserstoff und Sauerstoff verwenden. Zur Entleerung der Tanks wird Helium als druckerzeugendes Gas in die Tanks gefüllt, wodurch der Druck während des Treibstoffverbrauchs aufrecht erhalten wird (siehe [1, 3]).

Eine weitere Antriebsart sind elektrische Antriebssysteme, bei denen der Schub durch den Ausstoß von Ionen oder atomarer Partikeln erzeugt wird. Elektrische Antriebe sind sehr treibstoffeffizient, liefern allerdings nur Schubkräfte im Millinewtonbereich. Sie werden deshalb hauptsächlich für Satelliten oder Sonden bei Missionen in großen Entfernung eingesetzt; allerdings benötigen diese Systeme eine elektrische Energieversorgung, die bei Satelliten und erdnahen Missionen hauptsächlich durch Solarzellen

gewährleistet. Bei größeren Entfernungen zur Sonne ist dies auf Grund der geringen Sonnenstrahlung nicht mehr möglich, weshalb nukleare oder thermonukleare Antriebe verwendet werden (siehe [1, 3]). Ein Beispiel für die Verwendung eines solchen Antriebs war die Sonde „Deep Space 1“, deren Ziel der Test eines Xenon-Ionenantriebs war (siehe Abbildung 1.3).

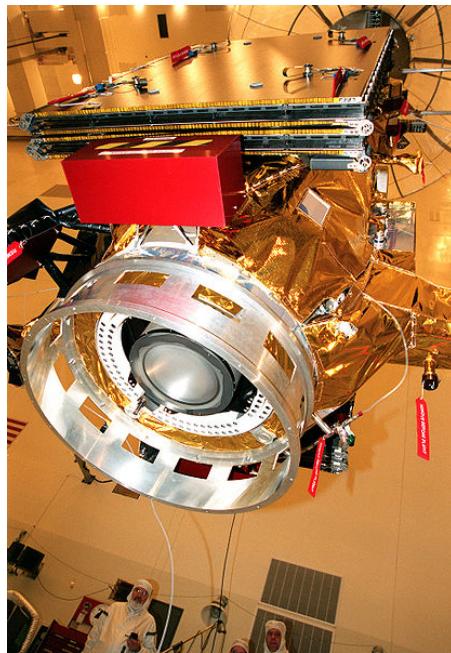


Abbildung 1.3: Deep Space 1 Sonde mit ihrem Ionenantrieb aus [5]

Diese Arbeit beschränkt sich ausschließlich auf Feststoff und Bi-Propellant Antriebe.

1.2.1.2 Die Steuerung

Die Steuerung einer Rakete erfolgt über verschiedene Technologien. Zur Stabilisierung und Steuerung durch die Atmosphäre werden Steuerfinnen eingesetzt wie sie in Abbildung 1.1 zu sehen sind.

Zusätzlich dazu kann bei Antrieben, bei denen es sich nicht um Feststoffantriebe handelt, eine Richtungskorrektur durch das Verstellen der Düsen erfolgen.

Des Weiteren werden zur Stabilisierung oder Positionierung im dreidimensionalen Raum Schwungscheiben eingesetzt, die eine Rotation um die Rotationsachse der Rakete hervorrufen oder verhindern können.

Auch mit Hilfe kleiner Schubdüsen, dem sogenannten Reaction Control System (RCS), kann die Rakete in drei Achsen positioniert werden. Ein solches System ist in Abbildung 1.4 zu sehen.

Eine besondere Art der Positionierung stellt die Stabilisierung mittels magnetischer Momente dar. Hierzu wird an der Seite von Satelliten ein Magnetband befestigt, das sich am Magnetfeld der Erde ausrichtet und somit eine konstante Ausrichtung sicherstellt (vgl. [1]).

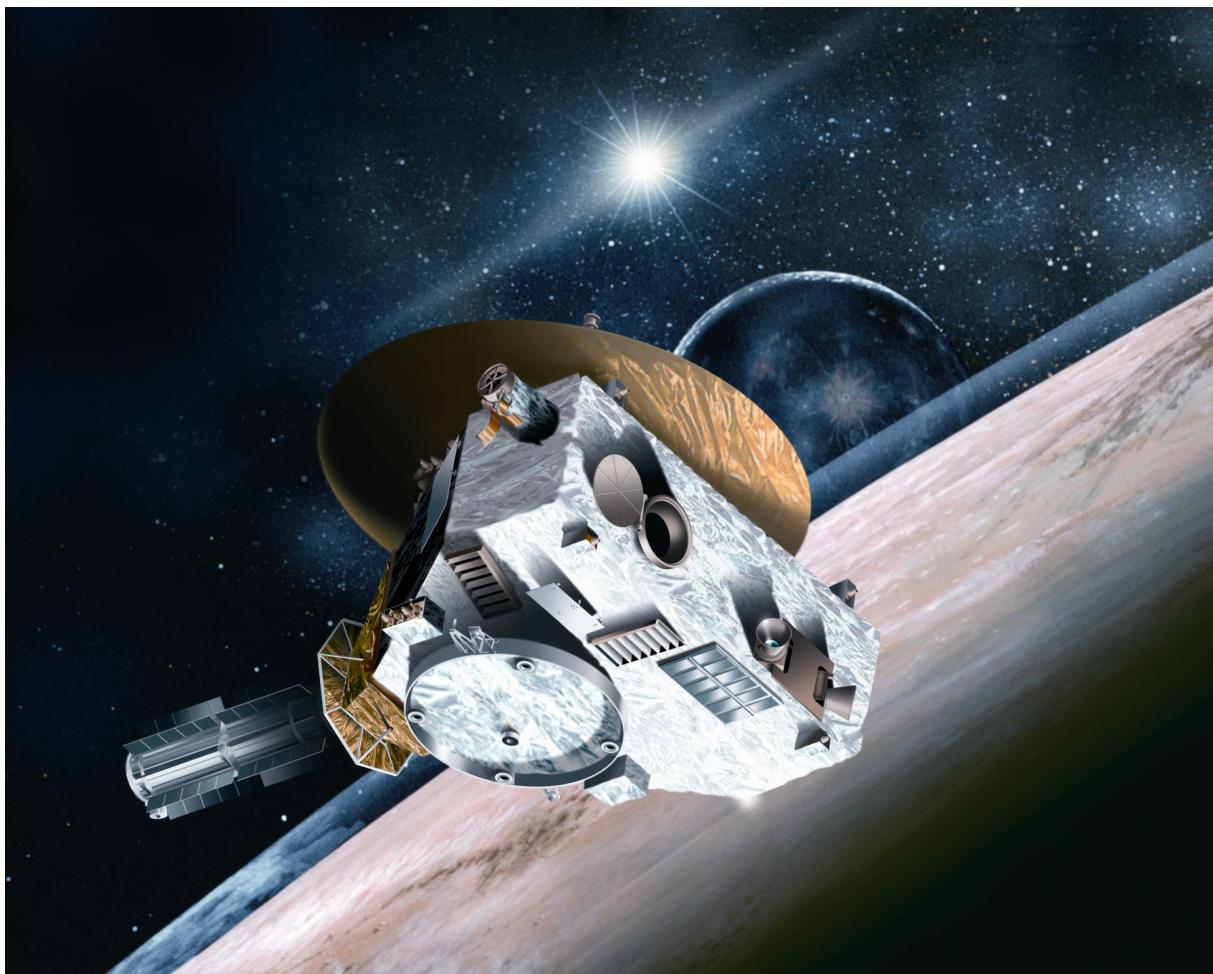


Abbildung 1.4: Render der Sonde New Horizon aus [6]

1.2.2 Der Flug

Bei einem Raketenflug wirkt zusätzlich zur vertikalen Beschleunigung noch die Rotationsgeschwindigkeit des Planeten auf die Rakete. Dies und das langsame Kippen

der Rakete, ausgelöst durch eine kurze Richtungskorrektur und die Verschiebung des Schwerpunkts der Rakete während des Fluges, führen zu einer parabolischen Flugbahn. Sie wird durch eine weitere horizontale Beschleunigung der Rakete zu einer orbitalen, elliptischen Flugbahn erweitert (siehe [7]), die in Abbildung 1.5 zu sehen ist.

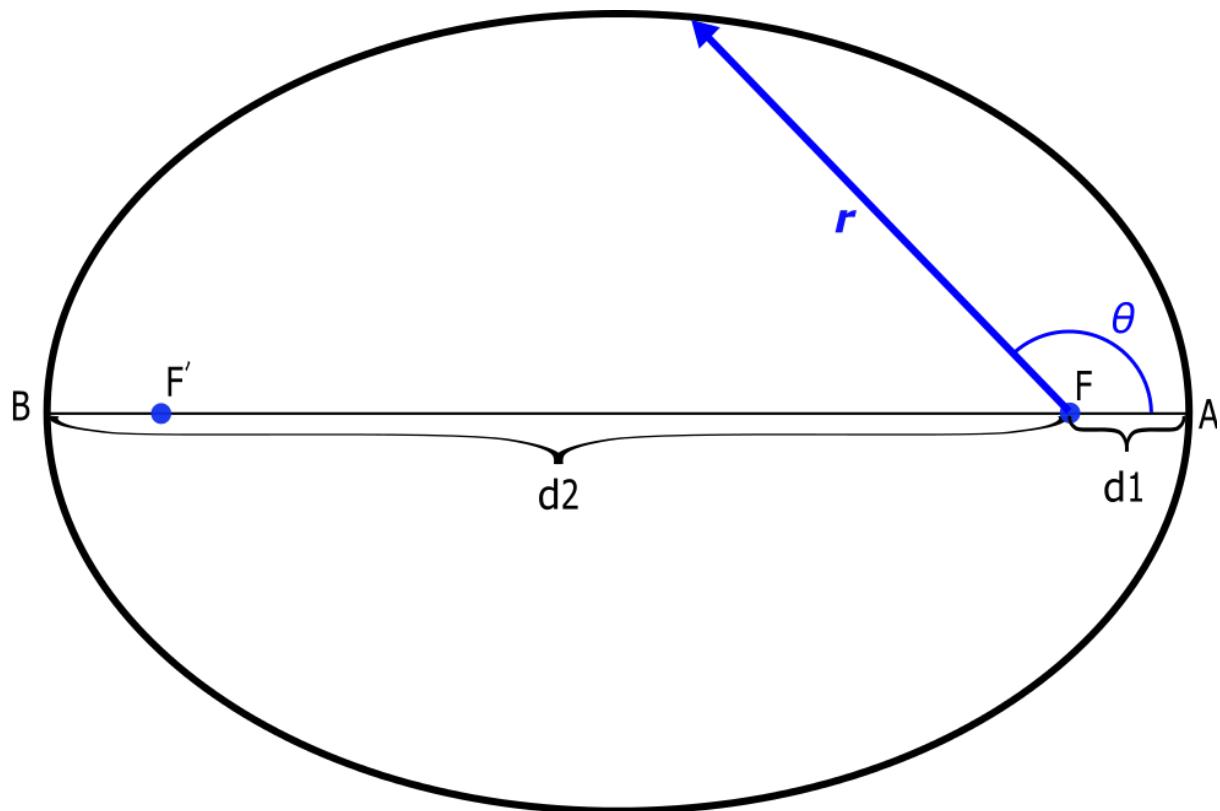


Abbildung 1.5: Elliptische Flugbahn mit markierten Apsiden (A & B) [8]

Die elliptische Form entsteht aus dem Zwei-Körper-System, das die Rakete und der umflogene Körper bilden. Das Gravitationszentrum liegt hierbei im Brennpunkt F der Ellipse. Charakterisiert wird diese dabei vor allem durch die Apoapsis B und die Periapsis A . Erstere stellt den vom Gravitationszentrum am weitest entfernten (Strecke d_2) Punkt der Flugbahn dar. Dies ist auch gleichzeitig der Punkt mit der niedrigsten Geschwindigkeit der Rakete, analog dem Extremum einer Wurfparabel. Dem entgegengesetzt ist die Periapsis, der Punkt der Ellipse, der dem Gravitationszentrum am nächsten (Strecke d_1) ist; an diesem Punkt ist die Geschwindigkeit der Rakete am höchsten (siehe [7]). Die Strecken d_1 und d_2 sind allerdings nicht mit der Entfernung der Apsiden

vom Planetenmittelpunkt zu verwechseln. Das Massenzentrum beziehungsweise der Planetenmittelpunkt sind nicht identisch mit dem Gravitationszentrum. Dieses liegt auf der Strecke zwischen dem Massenzentrum des Planeten und dem Massenzentrum des umkreisenden Körpers. Diese Tatsache kann allerdings bei Objekten mit stark unterschiedlichen Massen vernachlässigt werden, da das Gravitationszentrum sehr nahe am Mittelpunkt des massereicheren Körpers liegt.

1.3 Konkretisierung der Fragestellung

Die Beantwortung der Fragestellung, ob eine zweistufige Rakete Vorteile gegenüber einer einstufigen hat und wie eine zweistufige Rakete aufgebaut sein muss, um einen optimalen Flug zu gewährleisten, muss konkretisiert werden.

Als Kriterium für den Vergleich zweier Flüge wird die erreichte Höhe gewählt. Das Augenmerk liegt hierbei auf der Apoapsis, also dem am höchsten erreichbaren Punkt. Ein Flug gilt dann als besser oder optimaler, wenn er eine größere Höhe erreicht.

Unter dem Aufbau einer zweistufigen Rakete wird Verteilung des Tankvolumens auf beide Stufen verstanden. Das bedeutet, dass der Tankinhalt der einstufigen Rakete auf zwei Tanks aufgeteilt wird, sodass die Summe der Tankinhalte dem Tankvolumen des einstufen Tanks entspricht.

Im Folgenden werden mehrere Raketenflüge simuliert, um die Fragestellung unter den gegebenen Bedingungen zu beantworten. Zuerst soll allgemein der Raketenflug unter seinen physikalischen Gesetzmäßigkeiten mathematisch modelliert und beschrieben werden. Anschließend soll dieses Modell rechnerisch umgesetzt werden, um so eine Simulation zu ermöglichen. Abschließend sollen der Flug einer einstufigen Rakete mit verschiedenen mehrstufigen Raketen aufbauten verglichen werden, was die gegebene Fragestellung beantworten wird.

2 Physikalische Grundlagen eines Raketenflugs

In Kapitel 2 wird ein geeignetes physikalisches Modell erstellt, das einen Raketenflug mathematisch beschreibt.

2.1 Begrenzung der Betrachtung

Die Modellierung der Rakete kann nur bis zu einem bestimmten Detailgrad erfolgen: Dies hat zum einen den Grund, dass ein zu hoher Detailgrad die Anforderungen an den Rechner stark erhöht und somit zu einer langen, nicht mehr tragbaren, Simulationsdauer führt. Darüber hinaus ist ein zu hoher Detailgrad nicht gerechtfertigt, da es durch Rundungsungenauigkeiten und die numerische Berechnung trotzdem zu einer Abweichung des Ergebnisses kommt.

Aus diesem Grund müssen Vereinfachungen gemacht werden. Zum einen wird der Planet, von dem aus die Rakete startet nur als Punktmasse betrachtet. Dieser Massenpunkt liegt, da es sich, idealisiert betrachtet, um einen sphärischen Körper mit symmetrischer Massenverteilung handelt, im Mittelpunkt des Planeten. Der Radius des Planeten wird insofern berücksichtigt, dass er die Startdistanz der Rakete zum Massenpunkt beschreibt.

Zum anderen wird auch die Rakete als Punktmasse betrachtet. Dies ist dadurch gerechtfertigt, dass der Volumen- und Massenunterschied zwischen Planeten und Rakete sehr groß ist. Dieser Massenpunkt der Rakete wird am untersten Ende der Rakete platziert. Damit entspricht er allerdings nicht dem Schwerpunkt der Rakete, an dem normalerweise die Gravitationkraft ansetzt. Der Schwerpunkt wird nicht verwendet, da für seine Berechnung die genaue Massenverteilung der Rakete sowie deren Geometrie bekannt sein muss (siehe [9]). Des Weiteren müsste der Schwerpunkt in jedem Schritt erneut berechnet werden, da sich durch den Verbrauch von Treibstoff und das Abkoppen ausgebrannter Raketenstufen dieser in jedem Simulationsschritt ändert, was den

Rechenaufwand zusätzlich erhöht.

Beide Faktoren haben allerdings keinen großen Einfluss auf das Simulationsergebnis: Da es sich bei einer Rakete um einen rotationssymmetrischen Körper handelt, bewegt sich der Schwerpunkt auf der Rotationsachse des Körpers (siehe [9]); also erfolgt die Bewegung nur auf einer festen Linie. Da die Rakete in der Simulation senkrecht zum Horizont des Planeten startet, wirken alle Kräfte parallel zu dieser Rotationsachse auf einen Punkt der auf ihr liegt. Kräfte können allerdings entlang ihrer Wirkungslinien verschoben werden (siehe [9]). Dies legitimiert eine Positionierung des Massenpunkts am unteren Ende der Rakete. In Abbildung 2.1 ist die Rotationsachse (schwarz) und ein beispielhafter Schwerpunkt mit seiner Bewegungsrichtung während des Fluges (rot) eingezeichnet.

Als weitere Vereinfachung wird die Rakete in dieser Simulation nur senkrecht zur Planetenoberfläche beschleunigt. Die durch die Erddrehung bereits eingebrachte Kreisgeschwindigkeit wird ebenfalls außer Acht gelassen, da sie bei einem rein senkrechten Flug keinerlei Auswirkungen hat. Ihre Modellierung als Winkelgeschwindigkeit ist allerdings komplex. Hierdurch würde sich auch die erreichbare Höhe nur quantitativ und nicht qualitativ verändern, da die Rakete unter Berücksichtigung der Planetenrotation eine ballistische beziehungsweise parabolische Flugbahn anstatt einer geradlinigen verfolgt.

Die folgende Arbeit beschränkt sich auf zweistufige Raketen und deren Vergleich mit einstufigen Raketen.

Auch werden nur suborbitale Flüge betrachtet. Dies hat den Grund, dass orbitale Flüge bei einer Vernachlässigung der Planetenrotation wesentlich höhere Kräfte benötigen. Außerdem ist eine rein senkrechte Beschleunigung bei diesen nicht möglich. Die Flugdauer und damit die Simulationsdauer ist bei ihnen ebenfalls drastisch erhöht. Aller-



Abbildung 2.1: Antares Rakete während der Vorbereitung mit eingezeichneter Rotationsachse und Schwerpunkt (vgl. [10])

dings würde sich hier das Ergebnis ebenfalls nur quantitativ verändern, indem generell größere Höhen erzielt würden.

2.2 Bestimmung der Position

Für die Bestimmung der erreichten Höhe während eines Fluges muss die genaue Position der Rakete berechnet werden. Sie wird durch Koordinaten im zweidimensionalen Raum dargestellt:

$$\overrightarrow{pos}_{Rakete} = \begin{bmatrix} pos_x \\ pos_y \end{bmatrix}. \quad (2.1)$$

Dies ist ausreichend, da keine Kräfte an der Rakete in Süd-Nord- beziehungsweise z-Richtung wirken. Zur Berechnung werden die Bewegungsgleichungen herangezogen.

Der Weg berechnet sich gemäß [11] durch

$$pos_{Rakete\ i} = pos_{Rakete\ i-1} + v_{Rakete\ i} \cdot t + \frac{1}{2} \cdot a_{Rakete\ i} \cdot t^2. \quad (2.2)$$

Dies geschieht jeweils für die x und die y Koordinate. Abgeleitet nach der Zeit ergibt sich somit die Formel für die Geschwindigkeit.

$$v_{Rakete\ i} = v_{Rakete\ i-1} + a_{Rakete\ i} \cdot t \quad (2.3)$$

Die Berechnung der Beschleunigung erfolgt nach dem zweiten Newtonschen Gesetz. F_{Res} stellt dabei die auf die Rakete wirkende Kraft, m_{Rakete} die aktuelle Masse der Rakete dar.

$$a_{Rakete\ i} = \frac{F_{Res\ i}}{m_{Rakete\ i-1}} \quad (2.4)$$

Die Berechnung erfolgt mit diskreten Werten im Rahmen der computergestützten Simulation, der Index i bedeutet, dass der Wert aus dem aktuellen Zeitschritt stammt. Der Index $i - 1$ zeigt an, dass der Wert aus dem vorherigen Zeitpunkt verwendet wird.

2.3 Bestimmung der auf die Rakete resultierenden Kraft

Die auf die Rakete resultierenden Kraft $F_{Res\ i}$ berechnet sich aus der Summe der auf die Rakete wirkenden Kräfte. Dies sind der Schub F_T , die Gravitationskraft F_G und der Luftwiderstand F_D (siehe [7]). Deren Berechnung wird in den folgenden Kapiteln erläutert.

Zur Berechnung der Position muss diese durch einen Vektor ausgedrückt werden und

in x- beziehungsweise y-Komponenten berechnet werden als

$$\vec{F}_{Res} = \begin{bmatrix} F_{Res\ x} \\ F_{Res\ y} \end{bmatrix}. \quad (2.5)$$

Hierzu werden die x- und y-Komponenten der einzelnen Kräfte berechnet nach [9] unter Zuhilfenahme des Winkels α , den die Rotationsachse der Rakete mit der x-Achse einschließt.

$$F_x = F \cdot \cos(\alpha) \quad (2.6)$$

$$F_y = F \cdot \sin(\alpha) \quad (2.7)$$

F_{Res} berechnet sich damit durch

$$\vec{F}_{Res} = \begin{bmatrix} (F_D + F_T + F_G) \cdot \cos(\alpha) \\ (F_D + F_T + F_G) \cdot \sin(\alpha) \end{bmatrix}. \quad (2.8)$$

In Abbildung 2.2 sind die drei Kräfte, die auf die Rakete wirken, eingezeichnet.

2.3.1 Schub

Der Schub ist die Kraft, die der Antrieb einer Rakete erzeugt und sie somit beschleunigt. Er berechnet sich gemäß [7] mit Hilfe der Kenndaten des Antriebes zu:

$$F_T = \dot{m}_p \cdot v_e + A_n \cdot (p_n - p_{amb}). \quad (2.9)$$

\dot{m}_p beschreibt hierbei den Massenstrom des Treibstoffs durch die Düse des Antriebs. Dieser beschreibt die Veränderung der Masse des Treibstoffs m_p pro Zeiteinheit; der Wert ist charakteristisch für den Antrieb. Bei einigen Antrieben kann er sogar kontrolliert werden, was eine Möglichkeit zur Beschleunigungsregulierung darstellt.

v_e stellt die Geschwindigkeit dar, mit der die ausgestoßene Masse sich relativ zur Rakete bewegen. Das Produkt aus Massenstrom und Teilchengeschwindigkeit bildet den

sogenannten Düzenschub (vgl. [7]).



Abbildung 2.2: Start der SpaceX Falcon 9 mit eingezeichneten Kraftvektoren (vgl. [12])

Der zweite Summand beschreibt den Druckschub, der durch den Druckunterschied am Düsenausgang p_n und dem Druck der Umgebung p_{amb} entsteht. Der Druck an der Düse ist antriebsspezifisch. Der Umgebungsdruck dagegen ist abhängig von der Höhe der Rakete. Wie dieser berechnet wird, wird in Kapitel 2.4.2 erläutert. Bei einigen Antrieben lässt sich allerdings die Düsenfläche A_n verändern, um so auch den Druckschub zu optimieren. Die einzelnen Größen der Formel sind in Abbildung 2.3 illustriert.

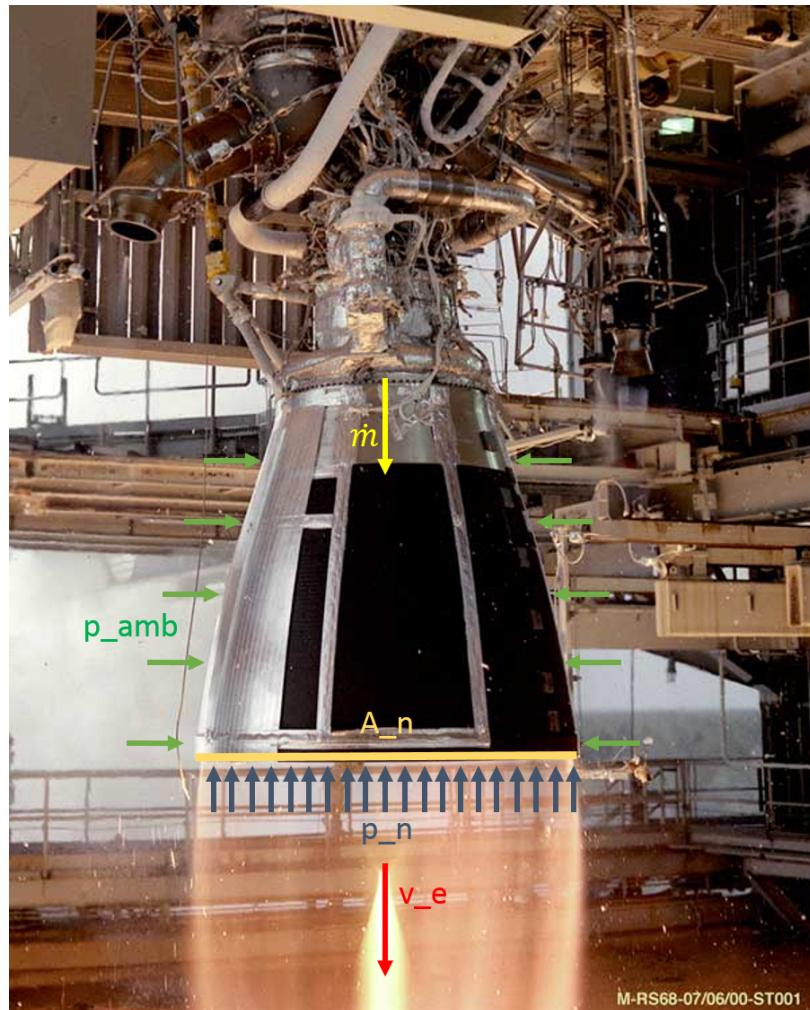


Abbildung 2.3: RS-68 Antrieb bei einem NASA Test mit eingezeichneten Kenngrößen auf Basis von [13]

2.3.2 Luftwiderstand

Der Luftwiderstand ist diejenige Kraft, die durch die Relativbewegung eines Körpers und eines ihn umströmenden Mediums entsteht. Nach [14] berechnet sich dieser zu:

$$F_D = c_w \cdot \frac{1}{2} \cdot \rho_{amb} \cdot v_{Rakete}^2 \cdot A_{Rakete}. \quad (2.10)$$

Der Widerstandsbeiwert c_w wird durch Windkanalversuche ermittelt und charakterisiert die Flächenreibung zwischen Medium und Raketenoberfläche (siehe [14]). Er bildet zusammen mit der Fläche der Rakete A_{Rakete} den Luftwiderstandsindex.

Der Faktor $\frac{1}{2} \cdot \rho_{amb} \cdot v_{Rakete}^2$ beschreibt den sogenannten dynamischen Druck oder Staudruck (vgl. [15]). Die Dichte des umgebenden Mediums ist abhängig vom Druck und der Temperatur des Mediums. Die Berechnung der Dichte wird in Kapitel 2.4.3 erläutert.

2.3.3 Gravitationskraft

Die Gravitationskraft ist diejenige Kraft, mit der sich zwei Massen gegenseitig anziehen. Diese Kraft ist nach dem Wechselwirkungsgesetz auf beide Körper gleich (siehe [9]). Die Gravitationskraft wird mit dem Newtonschen Gravitationsgesetz nach [7] berechnet:

$$F_G = G \cdot \frac{m_{Rakete} \cdot m_{Planet}}{d_{Rakete}^2}. \quad (2.11)$$

Dabei ist $G = 6,67408 \cdot 10^{-11} \frac{\text{m}^3}{\text{kg} \cdot \text{s}^2}$ die universelle Gravitationskonstante, d_{Rakete} beschreibt den Abstand zwischen dem Massenpunkt der Rakete und dem Massenpunkt des Planeten. Liegt der Mittelpunkt des Planeten im Ursprung des Koordinatensystems, kann der Betrag des Positionsvektors der Rakete einfach berechnet werden durch:

$$\|\vec{pos}_{Rakete}\| = \sqrt{pos_x^2 + pos_y^2}, \quad (2.12)$$

Ist dies nicht der Fall, kann die Distanz durch den Betrag der Vektordifferenz des Positionsvektors des Planeten und der Rakete berechnet werden.

2.4 Bestimmung der Umgebungswerte

Die Kräfte sind von verschiedenen Umgebungsvariablen abhängig wie der Dichte des Mediums, in dem sich die Rakete befindet, und dem Druck in der Höhe der Rakete.

2.4.1 Umgebungstemperatur

Die Umgebungstemperatur ist abhängig von der Höhe, in der sich die Rakete befindet; ihr Temperaturverlauf bei wachsender Höhe ist allerdings nicht linear. Er ist abhängig

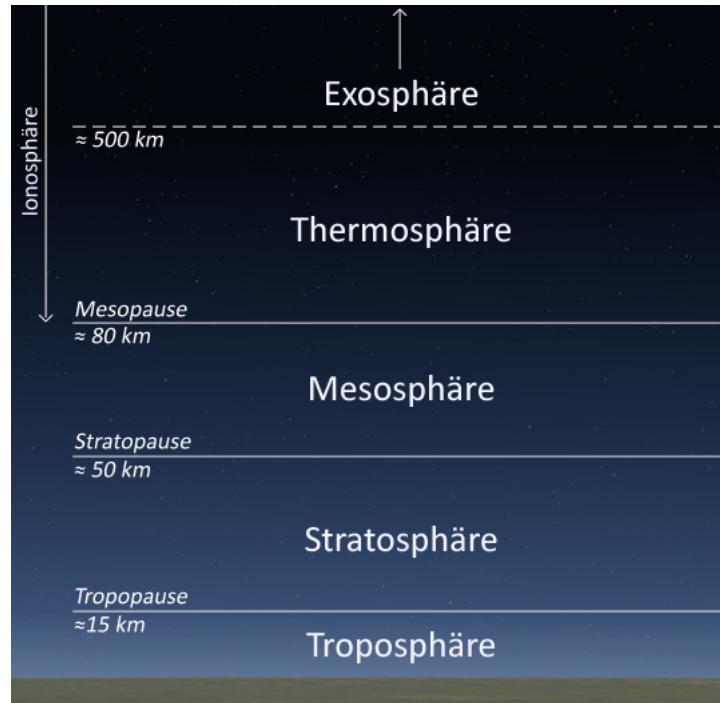


Abbildung 2.4: Schichten der Erdatmosphäre nach [16]

von der aktuellen atmosphärischen Schicht, in der sich die Rakete befindet. Innerhalb einer Schicht kann er idealisiert als linear betrachtet werden. Hierbei werden Wetterlagen und dadurch bedingte Temperaturschwankungen nicht berücksichtigt. Die aktuelle Umgebungstemperatur berechnet sich zu:

$$T_{amb} = T_{h_0} + a \cdot (h_{Rakete} - h_0). \quad (2.13)$$

h_0 ist dabei die Höhe, an der die aktuelle Schicht beginnt. Dementsprechend ist T_{h_0} die Temperatur in dieser Höhe. Die Konstante a beschreibt die lineare Temperaturänderung innerhalb einer Schicht.

Abbildung 2.4 zeigt die einzelnen Schichten der Erdatmosphäre.

2.4.2 Umgebungsdruckruck

Der Umgebungsdruck spielt eine wichtige Rolle bei der Berechnung des Schubs (vgl. Kapitel 2.3.1). Er berechnet sich aus der barometrischen Höhenformel ausgehend von der hydrostatischen Grundgleichung (siehe [17]):

$$p_{amb} = p_{h_0} \cdot \left(1 - \frac{a \cdot (h_{Rakete} - h_0)}{T_{h_0}}\right)^e, \quad (2.14)$$

mit e :

$$e = \frac{M_L \cdot g}{R \cdot a}. \quad (2.15)$$

Aus (2.14) lässt sich erkennen, dass ähnlich wie bei der Temperatur eine Abhängigkeit des Drucks von der aktuellen Luftsicht besteht. Der Druck am unteren Ende der Schicht ist p_{h_0} . Die Konstante $M_L = 0,02896 \frac{kg}{mol}$ ist die mittlere molare Masse der Gase in der Erdatmosphäre; $g = 9,807 \frac{m}{s^2}$ ist die Erdbeschleunigung; $R = 8,314 \frac{J}{K \cdot mol}$ stellt die universelle Gaskonstante dar.

In Tabelle 2.1 stehen die Kennwerte für die verschiedenen Schichten der Erdatmosphäre gefunden werden. Die Temperaturgradienten bilden sich aus der durchschnittlichen Temperaturänderung durch die Schicht. Die Drücke leiten sich aus der barometrischen Höhenformel her; diese Zwischenwerte könnten zwar auch während der Simulation berechnet werden, durch vorhergehende Berechnung kann die Simulation aber effizienter gestaltet werden.

Tabelle 2.1: Daten der atmosphärischen Schichten der Erde nach [18]

Schicht	a in $\frac{K}{m}$	T_{h_0} in K	p_{h_0} in Pa	h_0 in km
Troposphäre	-0,0065	288,15	101325	0
Stratosphäre	0,0031875	171,15	16901,37	18
Mesosphäre	-0,0033333	273,15	1,02	50
Thermosphäre	-0,00040512	173,15	0,04	80
Exosphäre	0	3	0	500

2.4.3 Dichte des umgebenden Mediums

Die Dichte eines Mediums hängt vom Druck und der Temperatur des Mediums ab.

Wird die Luft der Atmosphäre als ideales Gas betrachtet lässt sich die Dichte nach [17] berechnen zu:

$$\rho_{amb} = \frac{p_{amb} \cdot M_L}{R \cdot T_{amb}}. \quad (2.16)$$

Für eine nicht-ideale Betrachtung der Luftpfeuchte müsste die relative Luftpfeuchte des Mediums auf der Höhe bekannt sein. Diese unterliegt einer Reihe von Faktoren, die sich aufwendig oder gar nicht berechnen lassen, weshalb eine nicht-ideale Betrachtung nur exemplarisch möglich wäre und deshalb nicht in die Simulation einfließen kann (siehe [17]).

3 Rechnerische Umsetzung

Die Simulation soll computergestützt stattfinden; hierzu sollen die in Kapitel 2 erläuterten Formeln programmtechnisch umgesetzt werden. Außerdem sollen die Rakete, der Planet und die Atmosphäre modelliert werden. Die Umsetzung erfolgt in der Programmiersprache Python 2.7. Diese Programmiersprache ist immer weiter verbreitet für wissenschaftliche Simulationen, da sie im Gegensatz zu MatLab kostenlos und frei verfügbar ist sowie auf eine Vielzahl nützlicher Bibliotheken zugreifen kann, in ihrem Funktionsumfang aber die gleichen Möglichkeiten wie MatLab bietet. Sie ermöglicht objektorientierte Programmierung (OOP) und verfügt über eine einfach, gut verständlichen Syntax.

Der gesamte Code der Simulation befindet sich in den Anhängen A bis H und kann parallel unter <https://github.com/Gronner/RocketSim/tree/master> eingesehen werden.

3.1 Modellierung der Rakete

Die Rakete wird nach dem Prinzip der Objektorientierung programmiert. Hierzu wird eine Klasse „Rocket“ erstellt (siehe Quellcode 3.1). Im Folgenden soll auf die wichtigsten Funktionen dieser Klasse eingegangen werden. Der gesamte Code steht im Anhang F.

Quellcode 3.1: Klasse Rocket

```
1 class Rocket(object):  
2  
3     def __init__(self, pos, velocity, acceleration):  
4         ...  
5         self.rocket_parts = []  
6         self.pos = pos  
7         self.velocity = velocity  
8         self.acceleration = acceleration  
9         self.mass = 0.0  
10        self.surface = 0.0  
11        self.angle = 0
```

Die Einzelteile der Rakete werden in einer Liste „rocket_parts“ (siehe Zeile 1, Quellcode 3.1) gespeichert. Das erste Listenelement ist dabei immer die Raketenspitze, danach können Lastteile oder Tanks und Antriebe hinzugefügt werden. Das letzte Teil in der Liste ist die aktive Stufe.

In den Folgezeilen werden die Variablen der Rakete festgelegt, wobei die Position (pos), die Geschwindigkeit (velocity) und die Beschleunigung (acceleration) der Rakete als Vektor gespeichert werden, und die im Zeitschritt aktuellen Werte speichern.

Die Funktion „append_part“ (siehe Quellcode 3.2) ermöglicht das Hinzufügen von Raketenteilen und überprüft, ob es sich bei diesen auch um Tanks oder Raketenteile handelt (Zeile 3, Quellcode 3.2), sowie dass das erste Teil immer ein Raketenteil ist (Zeile 6, Quellcode 3.2). Das hinzuzufügende Teil wird als Argument übergeben.

Quellcode 3.2: Funktion append part

```
1 def append_part(self, new_part):
2     ...
3     if not issubclass(type(new_part), RocketPart):
4         print "Part has to be of type RocketPart or Subclass"
5         raise ValueError
6     if self.rocket_parts == [] and type(new_part) != RocketPart:
7         print "First part has to be of type RocketPart \
8             (e.g. nose of the rocket)"
9         raise ValueError
10    self.rocket_parts.append(new_part)
```

Mit „set_mass“ (siehe Quellcode 3.3) wird die aktuelle Masse der Rakete berechnet.

Dafür wird die Masse jedes Teiles in der Teileliste aufsummiert.

Quellcode 3.3: Funktion set mass

```
1 def set_mass(self):
2     ...
3     mass_sum = 0.0
4     for part in self.rocket_parts:
```

```
5     mass_sum += part.get_mass()  
6     self.mass = mass_sum
```

„decouple“ (siehe Quellcode 3.4) entfernt das letzte Teil in der Teileliste (Zeile 6, Quellcode 3.4), wenn es nicht das Teil auf Listenplatz eins ist (Zeile 3, Quellcode 3.4).

Quellcode 3.4: Funktion decouple

```
1 def decouple(self):  
2     ...  
3     if len(self.rocket_parts) == 1:  
4         pass  
5     else:  
6         self.rocket_parts.pop()
```

Diese Methoden ermöglichen es, die Rakete beliebig aufzubauen und mit verschiedenen Tanks oder Traglasten zu versehen. Im Folgenden soll nun die Modellierung der verschiedenen Teile, aus denen die Rakete aufgebaut werden kann, erläutert werden.

3.1.1 Raketenteile

Als Basisklasse und zur Modellierung von Traglasten wird die Klasse „RocketPart“ verwendet (siehe Quellcode 3.5). Ihr vollständiger Code ist im Anhang E.

Quellcode 3.5: Klasse Rocket Part

```
1 class RocketPart(object):  
2     ...  
3     def __init__(self, mass_part, surface_part,  
4                  drag_coefficient_part):  
5         ...  
6         if mass_part < 0:  
7             raise ValueError  
8         else:  
9             self.mass_part = mass_part  
10            if surface_part < 0:  
11                raise ValueError  
12            else:
```

```
13         self.surface_part = surface_part
14     if drag_coefficient_part < 0:
15         raise ValueError
16     else:
17         self.drag_coefficient_part = drag_coefficient_part
```

Hierzu werden dem Teil eine Masse (mass_ part), eine Querschnittsoberfläche (surface_ part) und ein Widerstandsbeiwert (drag_ coefficient_ part) zugewiesen. Alle drei Werte dürfen nicht Null betragen, weshalb Validitätskontrollen (Zeile 6, 10 und 14, Quellcode 3.5) eingebaut wurden, die zu einem Abbruch der Simulation führen und eine Fehlermeldung ausgeben. Die restlichen Methoden der Klasse dienen dazu, diese Variablen kontrolliert zu verändern oder auszulesen. Dies bedeutet, sollte während der Simulation eine der Variablen einen unzulässigen Wert ausgeben oder versucht werden, einen unzulässigen Wert zu setzen, bricht die Simulation mit einer Fehlermeldung ab.

3.1.2 Tanks und Antriebe

Da ein Tank ohne direkte Anbindung an einen Antrieb entweder auch als Traglast oder zusammengefasst zu einem anderen Tank modelliert werden könnte, werden ein Tank und ein Antrieb gemeinsam in einer Klasse zusammengefasst (siehe Quellcode 3.6). Die komplette Realisierung der Klassen kann im Anhang E eingesehen werden.

Quellcode 3.6: Klasse Tank

```
1 class Tank(RocketPart):
2     ...
3     def __init__(self, mass_part, surface_part,
4                  drag_coefficient_part, mass_propellant,
5                  mass_change_tank, velocity_exhaust_tank,
6                  surface_nozzle, pressure_nozzle):
7         RocketPart.__init__(self, mass_part, surface_part,
8                             drag_coefficient_part)
9         if mass_propellant < 0:
```

```
10         raise ValueError
11     else:
12         self.mass_propellant = mass_propellant
13     if mass_change_tank < 0:
14         raise ValueError
15     else:
16         self.mass_change_tank = mass_change_tank
17     if velocity_exhaust_tank < 0:
18         raise ValueError
19     else:
20         self.velocity_exhaust_tank = velocity_exhaust_tank
21     if surface_nozzle < 0:
22         raise ValueError
23     else:
24         self.surface_nozzle = surface_nozzle
25     if pressure_nozzle < 0:
26         raise ValueError
27     else:
28         self.pressure_nozzle = pressure_nozzle
29     self.thrust_level_tank = 1.0
```

Die Initialisierung der „RocketPart“ Klasse wird für die „Tank“ Klasse übernommen (Zeile 7, Quellcode 3.6). Zusätzlich dazu wird noch die Masse des Treibstoffes (mass_propellant) zur Modellierung des Tanks hinzugefügt. Die Eigenschaften des Antriebs werden durch die folgenden Variablen charakterisiert (Zeile 13-29, Quellcode 3.6). Die Variable „thrust_level_tank“ erlaubt die Anpassung des Massenstroms zur Regelung des Schubs wie schon in Kapitel 2.3.1 erwähnt. Aktuell wird diese Funktion jedoch nicht verwendet.

Die Funktion „get_mass“ gibt die Masse des Tankes aus (siehe Quellcode 3.7). Entsprechend der zusätzlichen Masse des Treibstoffs wird die Masse des Tanks aus der Summe der Teilemasse und der Treibstoffmasse berechnet.

Quellcode 3.7: Funktion get mass

```
1 def get_mass(self):
2     ...
3     mass_fueled = self.mass_propellant + self.mass_part
4     if mass_fueled < 0:
5         raise ValueError
6     else:
7         return mass_fueled
```

3.2 Modellierung der Umwelt

Um die Umwelt, in der sich die Rakete bewegt, zu modellieren muss, diese eine Atmosphäre mit ihren Schichten sowie einen Planeten beschreiben. Zur einfachen Erstellung dieser Objekte wird jedes durch eine Klasse dargestellt, das ihre typischen Attribute enthält.

3.2.1 Der Planet

Der Planet stellt den Himmelskörper dar, von dem die Rakete aus startet. Seine für die Simulation wichtigen Attribute werden mit seiner Initialisierung gesetzt und später nicht mehr geändert (siehe Quellcode 3.8, vgl. Anhang D).

Quellcode 3.8: Klasse Planet

```
1 class Planet(object):
2
3     def __init__(self, mass_planet, radius_planet, pos_planet):
4         ...
5         self.mass_planet = mass_planet
6         self.radius_planet = radius_planet
7         self.pos_planet = pos_planet
```

Das Positionsattribut (pos_planet) ermöglicht eine variable Platzierung des Planeten im Koordinatensystem, so ließen sich später auch Mehrkörpersysteme erstellen.

3.2.2 Die Atmosphärenschichten

Da die Schichten unterschiedliche Eigenschaften besitzen, werden sie durch eine eigene Klasse beschrieben (siehe Quellcode 3.9); so können Schichtobjekte schnell erstellt und einfach gehandhabt werden. Der gesamte Code befindet sich im Anhang C.

Quellcode 3.9: Klasse Layer

```

1 class Layer(object):
2
3     def __init__(self, width_layer, temp_gradient, temp_low,
4                  pressure_low):
5         ...
6         self.width_layer = width_layer
7         self.temp_gradient = temp_gradient
8         self.temp_low = temp_low
9         self.pressure_low = pressure_low

```

Die vertikale Ausdehnung der Schicht (width_layer) dient dazu, die Anfangs- und Endhöhe der verschiedenen Schichten zu bestimmen, mit absoluten Werten könnte keine variable Aufschichtung unterschiedlicher Schichten stattfinden. Des Weiteren wäre die Bestimmung der aktuellen Schicht erschwert. Diese wird im Kapitel 3.2.3 erläutert.

Die Schichtobjekte dienen auch der Berechnung der Dichte, des Drucks und der Temperatur der aktuellen Umgebung der Rakete (siehe Quellcode Quellcode 3.10). Hierzu werden drei Methoden verwendet, die wiederum auf die Formel in einem eigenen Modul zugreifen (siehe Kapitel 3.3).

Quellcode 3.10: Funktionen Layer

```

1 def get_pressure_now(self, height_rocket, height_layer_below):
2     ...
3     return Formula.pressure(self.pressure_low,
4                             self.temp_gradient, height_rocket,
5                             height_layer_below, self.temp_low)
6

```

```

7 def get_temperature_now(self, height_rocket,
8                         height_layer_below):
9     ...
10    return Formula.temperature(self.temp_low,
11                               self.temp_gradient, height_layer_below,
12                               height_rocket)
13
14 def get_density_now(self, height_rocket, height_layer_below):
15     ...
16     temperature_now = self.get_temperature_now(height_rocket,
17                                                 height_layer_below)
18     pressure_now = self.get_pressure_now(height_rocket,
19                                             height_layer_below)
20     return Formula.density(pressure_now, temperature_now)

```

Hierdurch wird die Übergabe von Parametern verringert, was den Code übersichtlicher und effizienter gestaltet.

3.2.3 Die Atmosphäre

Die Atmosphäre wird als Liste von Schichten modelliert, wobei hier das unterste Schichtenobjekt zu Beginn der Liste steht. Neue Schichten werden analog der Raketenteile hinten an die Liste angefügt.

Die Methode „clac_height_below“ berechnet die Höhe h_0 der aktuellen Schicht (siehe Quellcode 3.11). Dazu wird die vertikale Ausdehnung aller Schichten unter der aktuellen aufsummiert. Handelt es sich um die erste Schicht wird Null ausgegeben. Die Methode wird verwendet, um die aktuelle Schicht zu bestimmen.

Quellcode 3.11: Funktion calc height below

```

1 def calc_height_below(self, layer_index):
2     ...
3     height_below = 0
4     for i in range(0, layer_index):

```

```
5     height_below += self.layers[i].get_width()  
6 return height_below
```

Ist die aktuelle Höhe der Rakete größer als h_0 die aus „calc_height_below“ berechnete Höhe, wird die aktuelle Schicht ausgegeben (siehe Quellcode 3.12).

Quellcode 3.12: Funktion get_layer

```
1 def get_layer(self, height_rocket):  
2     ...  
3     current_layer = self.layers[0]  
4     for i in range(0, len(self.layers)):  
5         height_below = self.calc_height_below(i)  
6         if height_below > height_rocket:  
7             break  
8         current_layer = self.layers[i]  
9     return current_layer
```

Der restliche Code steht in Anhang B.

3.3 Umsetzung der Formeln

In Anhang A kann der Quellcode für die Umsetzung der Formeln aus Kapitel 2 gefunden werden. Alle Größen werden in SI-Einheiten übergeben und als doppelpräzise Gleitkommazahlen verarbeitet. Zur Umsetzung der Formeln wird auf die Python Bibliothek „math“ zugegriffen, die eine Vielzahl fertiger und performanceoptimierter Methoden für mathematischen Berechnungen bietet.

3.4 Speicherung der gewonnenen Daten

Auf die Klasse „Data“ (siehe Anhang H) soll nur kurz eingegangen werden. Sie dient hauptsächlich der Speicherung gewonnener Simulationsergebnisse, um diese später grafisch darzustellen und die Ergebnisse zu untersuchen. Hierzu werden die in jedem Schritt erzielten Ergebnisse in Listen gespeichert. Am Ende der Simulation werden diese Ergebnisse dann in eine kommasseparierte Datei (CSV) gespeichert.

Für eine spätere Be- oder Verarbeitung kann diese CSV-Datei auch durch eine Methode der Klasse wieder ausgelesen werden.

3.5 Ablauf der Simulation

Zum Vergleich unterschiedlicher Tankverteilungen zur Findung des optimalen Verhältnisses wird für jede zu simulierende Verteilung ein Flugobjekt der Klasse „Flight“ (vgl. Anhang G, Zeile 15 - 32) erstellt. Ein Flug besteht aus einem Planeten, einer Atmosphäre mit ihren Schichten, einer Rakete mit ihren Teilen und einem Datenobjekt zur Speicherung der Ergebnisse. Des Weiteren wird im Objekt die Größe des Zeitschrittes pro Simulationsschritt gespeichert.

In den Zeilen 34-37 wird beschrieben, wie einzelne Flugobjekte miteinander verglichen werden. Da als Kriterium für eine optimale Tankverteilung eine möglichst große erreichte Höhe ausgewählt wurde, wird die maximale Höhe während des Fluges aus dem Datenobjekt gesucht und mit der des anderen Objektes verglichen. Auf Basis der Funktion „__lt__“ sortiert und vergleicht Python Objekte. So können die Ergebnisse der Simulationen schnell miteinander verglichen werden.

3.5.1 Ablauf eines Simulationsschritts

In der Funktion „simulate“ ist der Algorithmus zur Berechnung eines Zeit- beziehungsweise Simulationsschrittes beschrieben (Zeilen 43-141). Alle gewonnenen Ergebnisse werden direkt in dem, dem Flug zugeordneten, Datenobjekt gespeichert.

Zu Beginn wird erst der aktuelle Zeitpunkt im dazugehörigen Datenarray gespeichert. Dazu wird der Zeitschritt zum im vorherigen Schritt gespeichert Zeitpunkt addiert. Anschließend wird die aktuelle Höhe berechnet (Zeilen 52-57). Hier wird die Position

der Rakete vektoriell addiert. Hieraus ergibt sich der Abstand des Planetenmittelpunkts zum Massenpunkt der Rakete. Vom diesem wird anschließend der Radius des Planeten abgezogen (3.1):

$$h_{Rakete} = \sqrt{pos_{Rakete\ x}^2 + pos_{Rakete\ y}^2} - r_{Planet}. \quad (3.1)$$

Mit h_{Rakete} lässt sich nun die Temperatur der Umgebung berechnen (Zeilen 59-65). Hierzu wird erst die aktuelle Schicht mit ihrem unteren Ende, ihrer Temperatur am unteren Ende und dem Temperaturgradienten bestimmt. Anschließend wird mit Hilfe von (2.13) die Temperatur auf der aktuellen Höhe berechnet.

Dies ermöglicht die Berechnung des Drucks auf der aktuellen Höhe (Zeilen 66-69) nach (2.14). Mit den so gewonnenen Werten können nun Gravitationskraft (Zeilen 70-72) mittels (2.11), Schub (Zeile 73-83) mit (2.9) und Luftwiderstand (Zeilen 84-94) mittels (2.10) berechnet werden.

Vor der Berechnung des Schubs wird die aktive Raketenstufe bestimmt, um so an die Antriebsdaten zu gelangen. Handelt es sich dabei allerdings um keinen Tank mit Antrieb, wird der Schub der Last genommen, der im Regelfall Null beträgt. Dies ist notwendig, da als letzte aktive Stufe immer die Raktenspitze ohne Antrieb vorhanden ist. An dieser Stelle werden damit Rechenoperationen eingespart sowie das Programmdesign der Raketenteile einfach gehalten.

Für die Berechnung des Luftwiderstandes wird vorher die Dichte berechnet mit den Werten der aktuellen Temperatur und des aktuellen Drucks (2.16).

In den Zeilen 95-103 werden die Kräfte in ihre x- und ihre y-Komponenten aufgeteilt. Dies erfolgt für den x-Anteil mittels (2.6) und für den y-Anteil mittels (2.7). In den folgenden Zeilen (104-109) wird jeweils die resultierende Kraft in x- und y-Richtung bestimmt, indem Gravitationskraft und Luftwiderstand vom Schub subtrahiert

werden. Zwar werden in (2.8) die Kräfte aufsummiert, hierbei wird allerdings davon ausgegangen, dass die Richtung der Kräfte in ihrem Wert enthalten ist. Da bei der senkrechten Flugbahn allerdings sowohl Luftwiderstand als auch Gravitationskraft entgegen des Schubs gerichtet sind, kann diese Vereinfachung gemacht werden, um Rechenoperationen einzusparen (siehe Abbildung 2.2).

Abschließend wird in den Zeilen 110-140, basierend auf den in Kapitel 2.2 beschriebenen Formeln, die Beschleunigung, die Geschwindigkeit und die neue aktuelle Position errechnet.

Des Weiteren wird die Masse der Rakete neu berechnet (Zeilen 116-124). Dazu wird zuerst überprüft, ob aktuell ein Antrieb aktiv ist; ist dies nicht der Fall findet keine Massenänderung statt. Ändert sich die Masse wird überprüft, ob dadurch der Tank keinen Treibstoff enthält. Dieser wird gegebenenfalls abgekoppelt oder die in ihm befindliche Treibstoffmenge angepasst.

In der letzten Zeile wird noch die aktuelle Distanz des Massenpunkts der Rakete bis zum Mittelpunkt des Planeten bestimmt.

3.5.2 Setup der einer Simulation

In der „main“-Methode werden die Eigenschaften der Simulationsobjekte festgelegt. Dies lässt sich ebenfalls im Anhang G (Zeilen 203-220) finden. Hierzu werden die Simulationsobjekte, also der Planet, die Atmosphäre mit ihren Schichten, die Rakete mit ihren Bauteilen, das Datenspeicherobjekt und der Flug als Instanzen der respektiven Klassen erzeugt.

Anschließend wird das Datenobjekt mit simulationsabhängigen Werten gefüllt und die Simulation durchgeführt (Zeilen 221-339). Abschließend wird die Methode „run_sim“ ausgeführt. Hierbei handelt es sich um eine Wrapper-Funktion, die die Einstellung der Simulation automatisch vor und nach der Simulation in das Datenobjekt überträgt. Auch wird die Methode „simulate“ des Flugobjektes so lange ausgeführt bis entweder die vorher festgelegte, maximale Anzahl an Simulationsschritten überschritten wird oder

die Entfernung der Rakete vom Mittelpunkt des Planeten kleiner als dessen Radius ist, was bedeutet, dass die Rakete sich wieder auf der Planetenoberfläche befindet und der Flug beendet ist.

4 Parameter der Simulation

Nachdem in Kapitel 2 das mathematische Modell für die Simulation der Rakete erläutert und in Kapitel 3 umgesetzt wurde, werden im Folgenden die Ergebnisse zur Beantwortung der Frage, ob eine zweistufige Rakete eine größere Höhe erreichen kann als eine einstufige und wie die Treibstoffverteilung in einer zweistufigen Rakete gestaltet sein muss, um eine optimale Höhe zu erreichen, vorgestellt. Die Ergebnisse werden auf Basis einer Aerobee 150 (A150) „Sounding Rocket“ berechnet.

4.1 Die Aerobee 150

Für realistische Simulationsergebnisse sollen Kenn- und Lastwerte einer realen Rakete verwendet werden. Hierzu wurde die für suborbitale Flüge entwickelte Aerobee 150 gewählt. Dies hat eine Vielzahl von Gründen. Zum Einen wurde sie zumeist senkrecht zur Erdoberfläche gestartet und verfolgte eine senkrechte Bahn während des Fluges. Die für heutige Raketen relativ geringe Reichweite verkürzt die Simulationsdauer, ohne allerdings die Aussagekraft der Ergebnisse einzuschränken. Des Weiteren sind die Lastdaten der Rakete und vor allem ihres Antriebs öffentlich verfügbar und gut dokumentiert.

4.1.1 Geschichte der Aerobee 150

Bei der Aerobee 150 handelt es sich um eine sogenannte „Sounding Rocket“. Dies sind einfache Raketen für suborbitale Flüge, die häufig nur im Bereich der Erdatmosphäre eingesetzt werden. Sie dienen dazu, Nutzlasten mit wissenschaftlichen Experimenten in höhere Atmosphärenschichten zu transportieren, um so zu vergleichsweise kostengünstigen Messwerten zu gelangen.

Die Aerobee 150 ist eine von zehn Raketen aus der Aerobee Serie, die eine Nutzlast von 150 lb (ca. 68 kg) transportieren kann. Sie wurde als simplere Alternative zur deutschen V2-Rakete entwickelt, die vor der Entwicklung der Aerobee als „Sounding Rocket“ von den Amerikanern eingesetzt wurde. Hauptsächlich wurde sie während der

50er und 60er Jahre des 20. Jahrhunderts eingesetzt (siehe [19]).

4.1.2 Aufbau der Aerobee 150

In Abbildung 4.1 ist eine einstufige Aerobee 150 abgebildet.

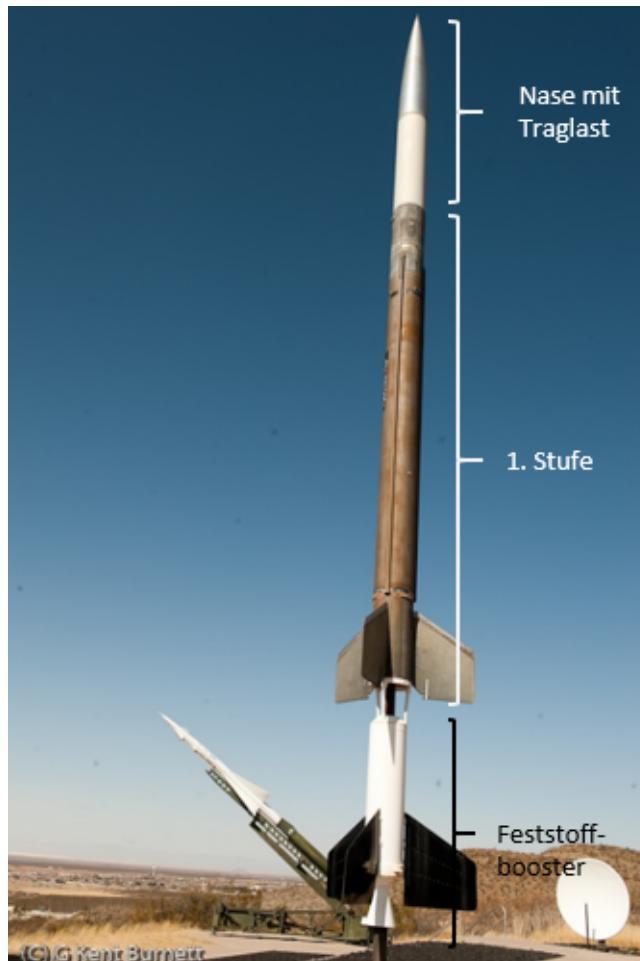


Abbildung 4.1: Aerobee 150 Rakete mit markiertem Aufbau auf Basis von [20]

Im Anhang I sind die Lastdaten der Aerobee 150 tabellarisch zusammengefasst. Alle Werte wurden aus dem angloamerikanischen System ins metrische auf ihre SI-Einheiten überführt. Im Folgenden soll nun auf die drei Standardteile der Rakete eingegangen werden.

4.1.2.1 Nose mit Traglast

Die Nase der Rakete besteht aus Aluminium und ist ogive geformt. Sie hat eine Länge von 2,23 m und eine Querschnittsfläche von $0,11 \text{ m}^2$. Ihr Eigengewicht beträgt 7 kg und kann bis zu 68 kg an Nutzlasten enthalten. Bei Bedarf kann die Nase durch Erweiterungen verlängert werden, um so das für Nutzlasten zur Verfügung stehende Volumen zu vergrößern. Nach Abbrennen der letzten Stufe bleibt nur noch die Nase mit Nutzlast von der Rakete übrig; die restlichen Stufen werden nach dem Abbrennen abgetrennt (vgl. [19]).

4.1.2.2 Flüssigtreibstofftank mit Antrieb

Hinter der Nase befinden sich die Flüssigtreibstoffantriebe mit ihren Tanks. Zum Antrieb der Rakete wird ein Gemisch aus einem Brennstoff und einem Oxidationsmittel verwendet. Bei der A150 wird für den Brennstoff ein Gemisch aus 65% Anilin und 35% Furfurylalkohol verwendet, das als ANFA bezeichnet wird. Als Oxidationsmittel wird rot rauchende Salpetersäure (RFNA) verwendet. Beide Stoffe werden getrennt in Tanks gelagert und im Düsenantrieb in einem festen Mischungsverhältnis verbrannt (siehe [19]).

Der Tank hat dabei ein Eigengewicht von 116 kg inklusive Antrieb und fasst 227 l Oxidationsmittel mit einem Gewicht von 347 kg sowie 130 l Brennstoff mit einem Gewicht von 138 kg. Die Länge des Tanks beträgt 4 m ohne Antrieb. Zusätzlich dazu kommen noch einmal 2 kg Helium, die zur Druckregulierung innerhalb des Tankes dienen (siehe [19]).

Der Antrieb hat eine Länge von 0,77 m. In ihm werden Brennstoff und Oxidationsmittel vermischt und die dabei entstehenden Gase durch die Antriebsdüse und dem im Tank herrschenden Druck nach außen gestoßen, um so Schub zu erzeugen. Der Nominalschub des Antriebs beträgt 18,23 kN. Unter dem Nominalschub versteht man den

Schub, den ein Antrieb auf Meeressniveau unter Standardbedingungen erzeugt, sodass mit diesem also nicht während des Fluges gerechnet werden kann.

Die Düsenfläche des Antriebs beträgt $0,02 \text{ m}^2$. Die Gase werden mit einer Geschwindigkeit von $1.417,8 \text{ m/s}$ ausgestoßen; der Massenstrom beträgt dabei $9,4 \text{ kg/s}$. Für den Düsendruck p_n wird kein Wert angegeben, er kann allerdings aus den gegebenen Werten errechnet werden, wenn Gleichung (2.9) nach p_n umgestellt wird. Hieraus ergibt sich ein Düsendruck, der dem Standardatmosphärendruck von 101325 Pa entspricht (vgl. [19, 21]).

4.1.2.3 Feststoffbooster

Vor allem beim Raketenstart sind hohe Kräfte notwendig, um die Rakete zu beschleunigen. Dies ist nötig, um die Gravitationskraft zu überwinden und die Rakete durch den dichtesten Teil der Atmosphäre, die sogenannte Troposphäre, zu beschleunigen. Flüssigtreibstoffantriebe liefern hierfür meist nicht genügend Schub, weshalb Feststoffbooster verwendet werden, die eine Art Brennpaste als Treibstoff verwenden. Für die A150 wird hierzu der Aerojet 2.5KS verwendet. Dieser liefert während einer Brennzeit von $2,5 \text{ s}$ einen Nominalschub von $82,7 \text{ kN}$. Mit einer Düsenfläche von $0,04 \text{ m}^2$ ist der Druckschub etwas höher als bei Flüssigtreibstoffantrieb, da der Düsendruck bei beiden Antrieben gleich ist. Der Massenstrom des Aerojet 2.5KS beträgt $47,1 \text{ kg/s}$. Die bei der Verbrennung entstehenden Gase werden mit einer Geschwindigkeit von $1.747,6 \text{ m/s}$ ausgestoßen. Der Antrieb hat ein Eigengewicht von 28 kg inklusive Tank, der 118 kg Treibstoff fasst (siehe [19, 21]).

4.2 Umgebungsbedingungen

Als Planet wird die Erde verwendet. Diese hat eine Masse von $5,974 \cdot 10^{24} \text{ kg}$ und einen Radius von 6.371 m ; auch wird die Erdatmosphäre verwendet. Die Bedingungen in dieser sowie ihre Kennwerte sind in Kapitel 2.4.2 und Tabelle 2.1 beschrieben

5 Ergebnisse der Simulation

In diesem Kapitel sollen die Ergebnisse der Simulation und die daraus resultierenden Schlussfolgerungen präsentiert werden. Allen Simulationen liegen die in Anhang I zu findenden Parameter zu Grunde und werden mit einem Zeitschritt von 0,1 s durchgeführt.

5.1 Simulation der einstufigen Rakete

Zuerst wird die Simulation auf Grundlage einer einstufigen A150 Rakete betrachtet.

In Abbildung 5.1 ist der Höhenverlauf des Fluges zu sehen.

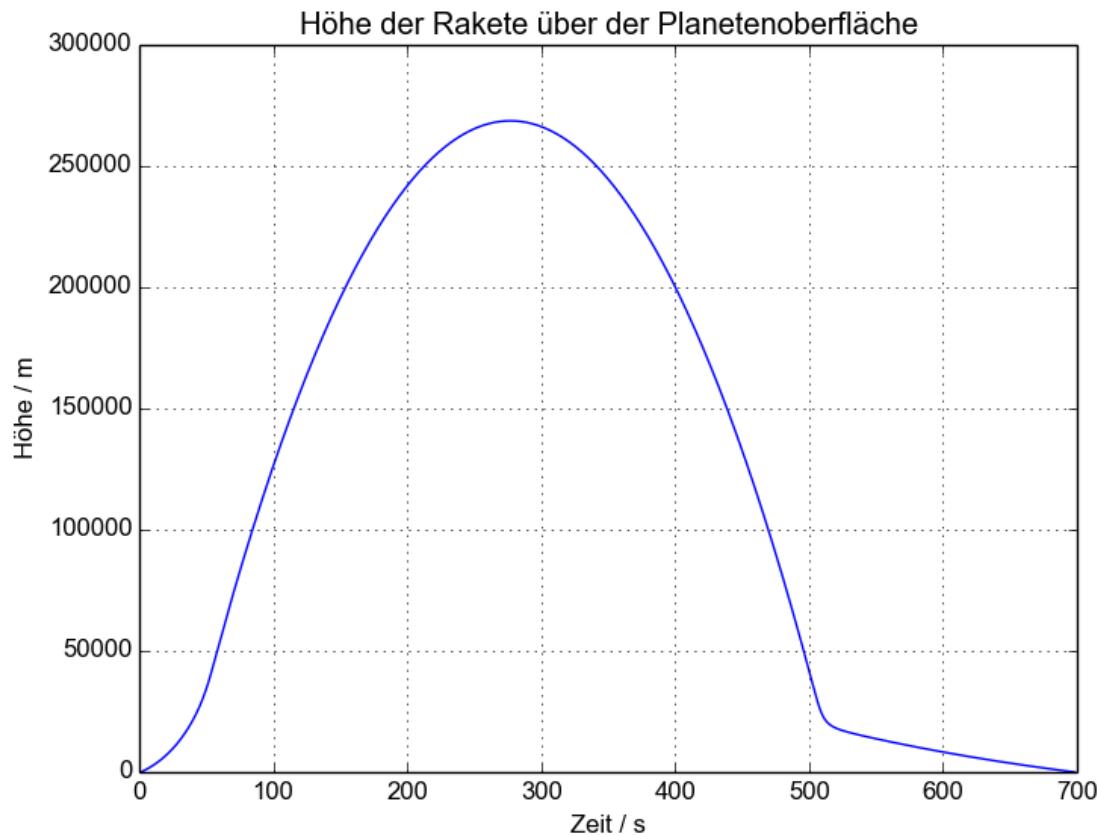


Abbildung 5.1: Höhenverlauf der einstufigen Rakete über die Zeit

Während des Fluges wird in der Simulation eine Höhe von 268.680 m erreicht. Die Apoapsis wird nach 277,1 s erreicht.

An der einstufigen Rakete wird auch das Verhalten der Rakete während der Simulation

beschrieben und anhand der physikalischen Gegebenheiten aus Kapitel 2 begründet.

5.1.1 Die Startphase

Zu Beginn beschleunigt die Rakete bis zu einem Zeitpunkt von 54,2 s. An ihm sind sowohl der Booster als auch die erste Stufe ausgebrannt. Die Phase von 0 s bis 54,2 s wird im Folgenden als Start oder Startphase bezeichnet. Dies lässt sich sowohl im Verlauf der Masse während der Anfangsphase (siehe Abbildung 5.2) als auch im Verlauf des Schubs der simulierten Rakete erkennen (siehe Abbildung 5.3).

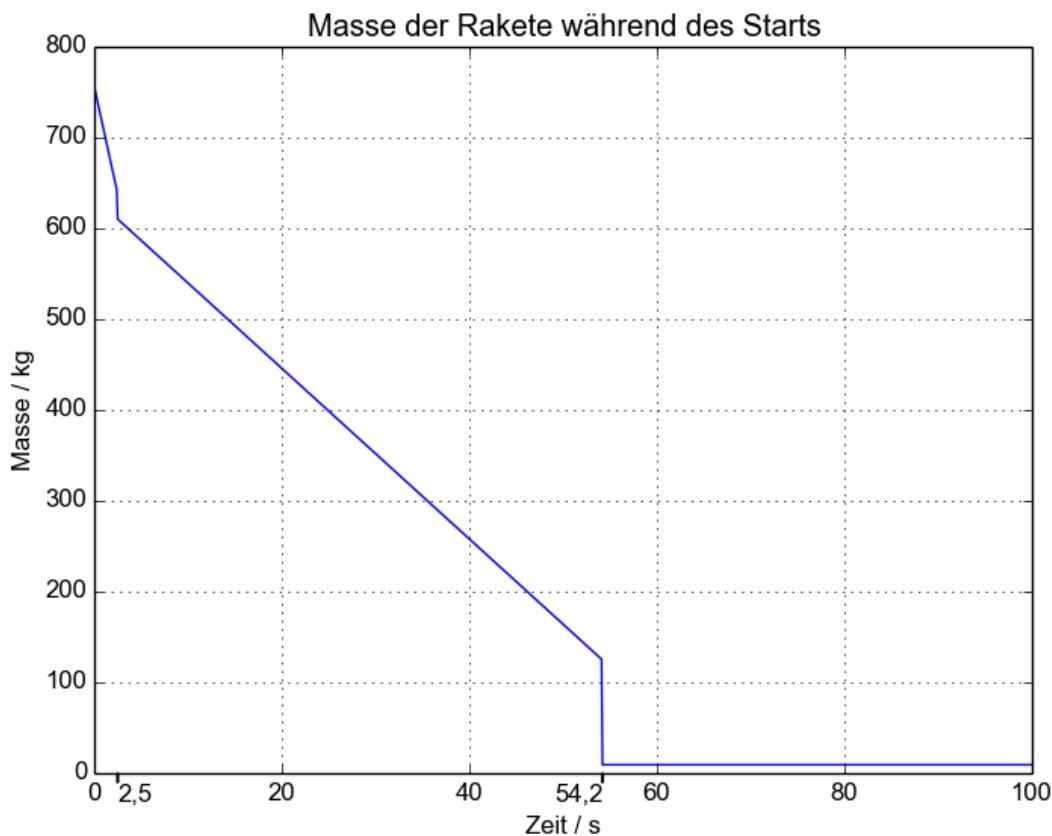


Abbildung 5.2: Verlauf der Masse der Rakete während des Starts

Auf Abbildung 5.2 lässt sich gut das lineare Abnehmen der Raketenmasse während des Abbrennens der Stufen erkennen. Dies geschieht infolge des Massenstroms durch den Antrieb. Auch die Abtrennung der ausgebrannten Stufen ist durch die sehr starke Änderung der Masse erkennbar. Der Booster wird bei 2,5 s abgetrennt, die erste 1. Stufe bei

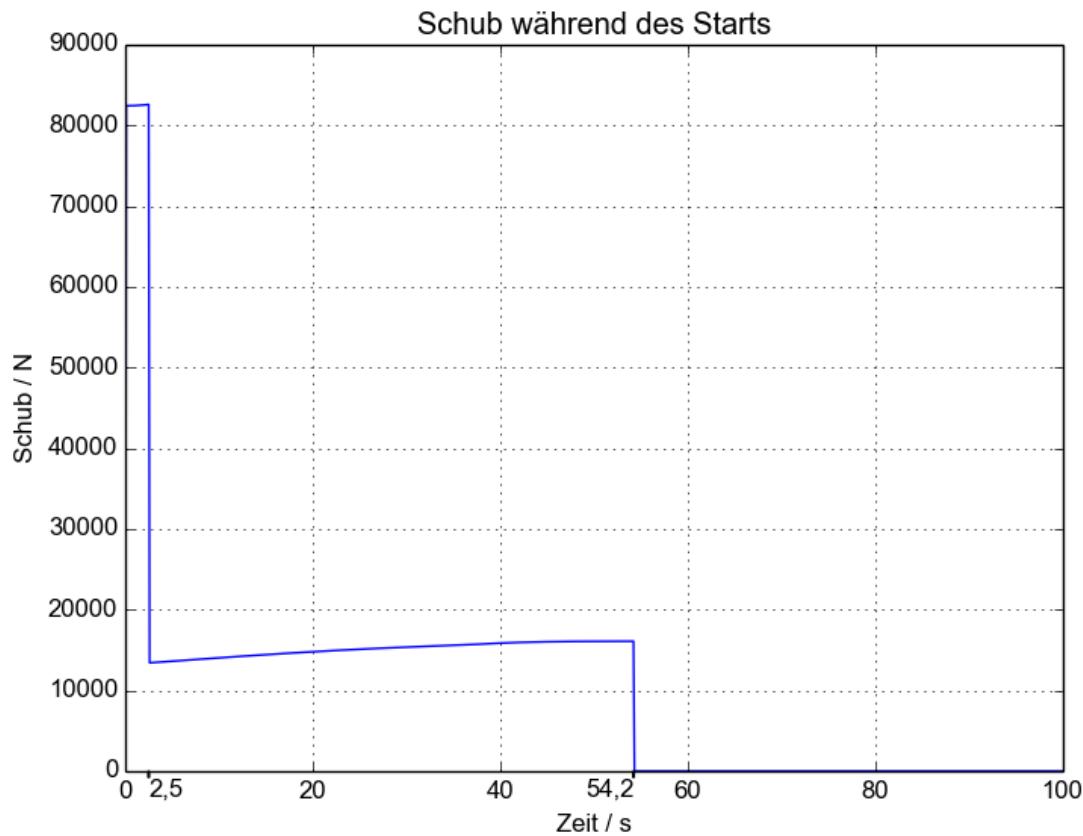


Abbildung 5.3: Verlauf der Schubkräfte während des Starts

54,2 s. Das Ausbrennen der Stufen ist auch im Verlauf der Schubkräfte in 5.3 erkennbar, da zu diesen Zeitpunkten ebenfalls eine schnelle Veränderung des Schubes stattfindet.

Ein weiteres Phänomen ist der nicht-lineare Anstieg des Schubs. Dieser entsteht aus dem Druckschub. Nach Gleichung (2.9) berechnet sich der Schub aus einem linearen Teil, bestehend aus dem Produkt des konstanten Massenstroms und der konstanten Abgasgeschwindigkeit. Der Druckschub wird über das Produkt der Düsenfläche mit der Druckdifferenz am Düsenausgang berechnet. Das nichtlineare Verhalten entsteht durch den logarithmischen Verlauf des Umgebungsdruckes (barometrischen Höhenformel (2.14)). Dessen Druckverlauf lässt sich in Abbildung 5.4 erkennen.

Während der Düsendruck p_n konstant bleibt verringert sich der Umgebungsdruck p_{amb} mit der Konsequenz, dass der Druckschub zunimmt. Dies führt insgesamt zu einem

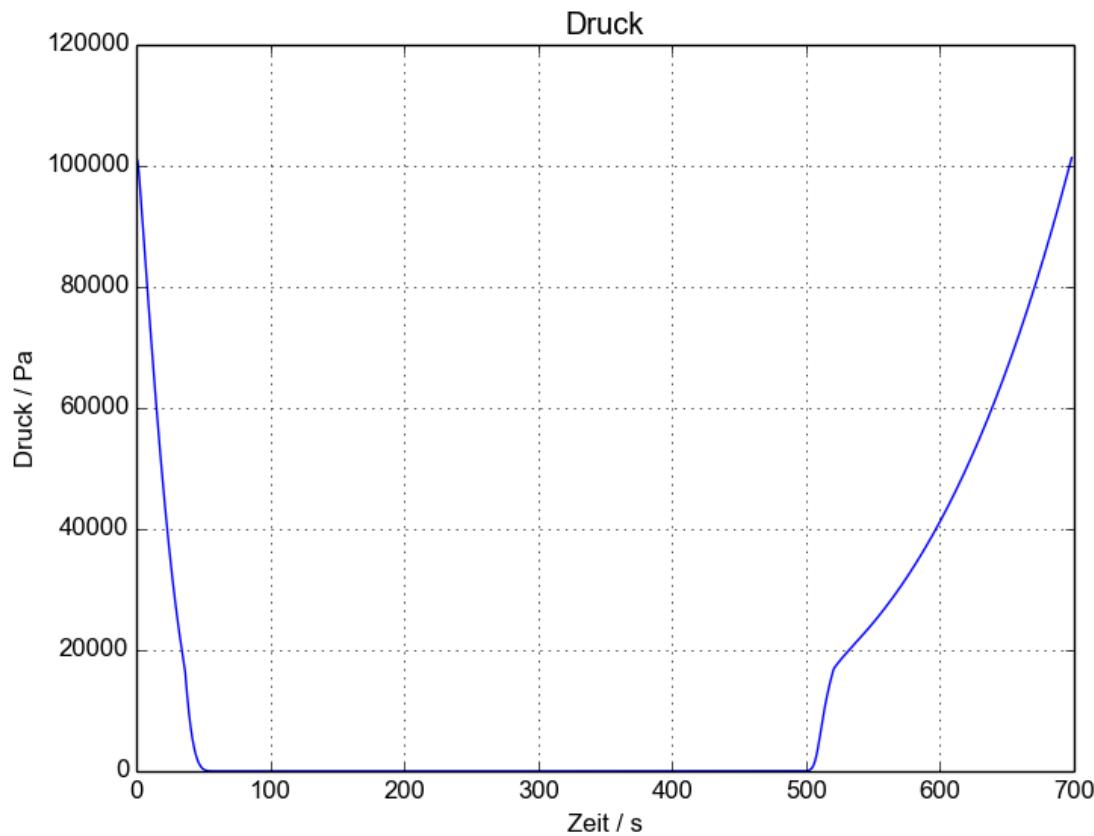


Abbildung 5.4: Verlauf des Druckes über die Simulationsdauer

Anstieg des Schubes.

Die Geschwindigkeit während der Startphase (siehe Abbildung 5.5) nimmt zu Beginn linear stark zu.

Der starke Anstieg erklärt sich durch die hohen Schubkräfte des Boosters. Der lineare Anstieg entsteht durch den konstanten Schub sowie den in dieser Höhe fast linearen Verlauf des Luftwiderstandes (siehe Abbildung 5.6) und der Gravitationskraft (siehe Abbildung 5.7). Im späteren Verlauf nimmt die Geschwindigkeit langsamer zu. Allerdings steigt die Beschleunigung mit der Zeit an (siehe Abbildung 5.8). Dies resultiert aus der Tatsache, dass die Gravitationskraft durch die größer werdende Entfernung zwischen Planetenmassenpunkt und Raketenmassenpunkt abnimmt; ebenso wie die Masse der Rakete durch den Verbrauch an Treibstoff. Auch der Luftwiderstand nimmt,

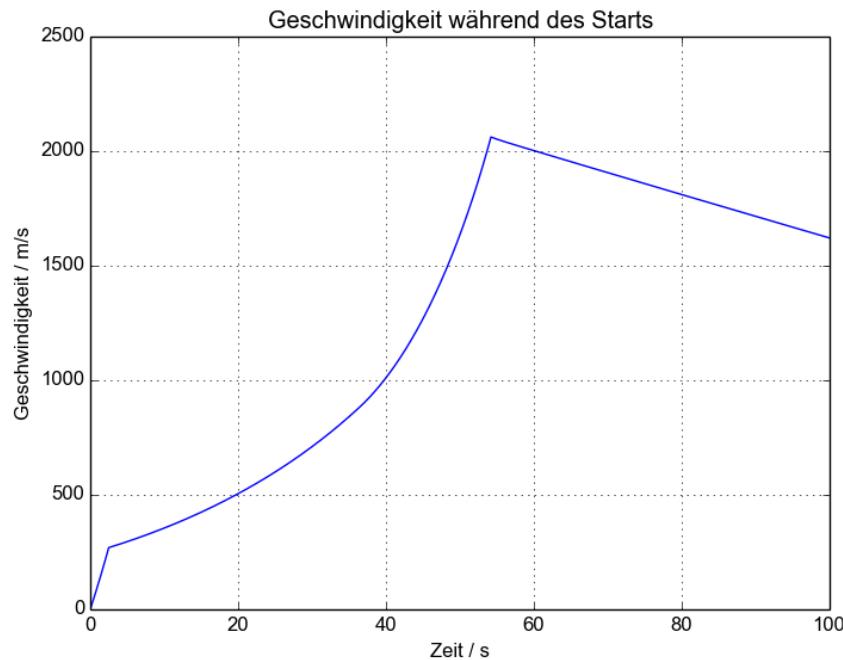


Abbildung 5.5: Verlauf der Geschwindigkeit während des Starts

nachdem er sich kontinuierlich durch die steigende Geschwindigkeit erhöht, durch die stark abfallende Dichte in Folge des sinkenden Drucks weiter ab.

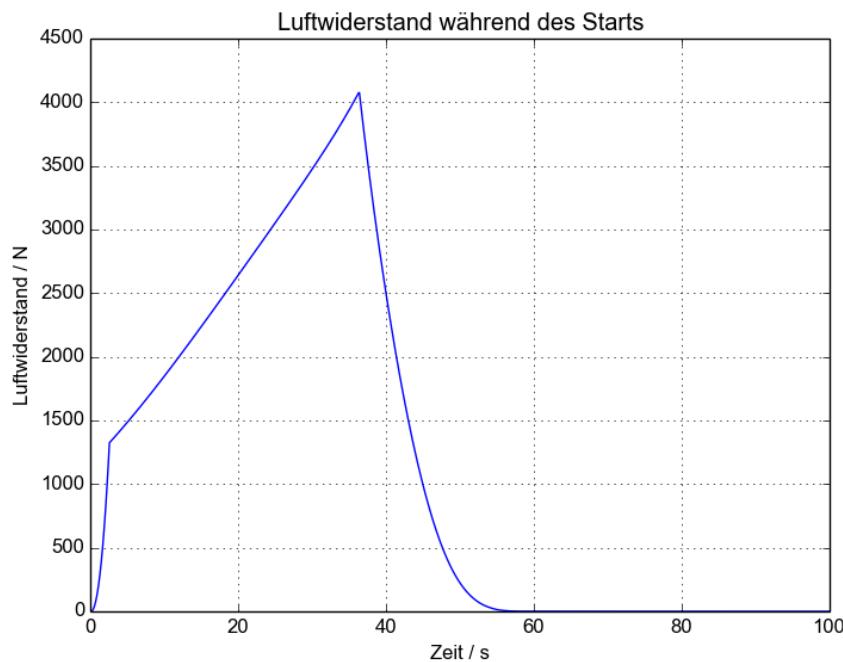


Abbildung 5.6: Verlauf des Luftwiderstands während des Starts

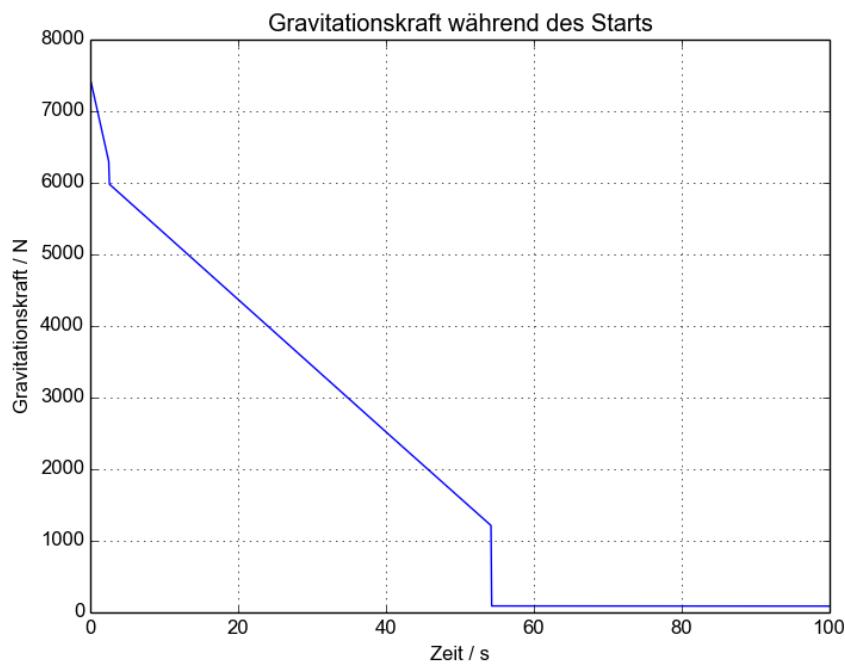


Abbildung 5.7: Verlauf der Gravitationskraft während des Starts

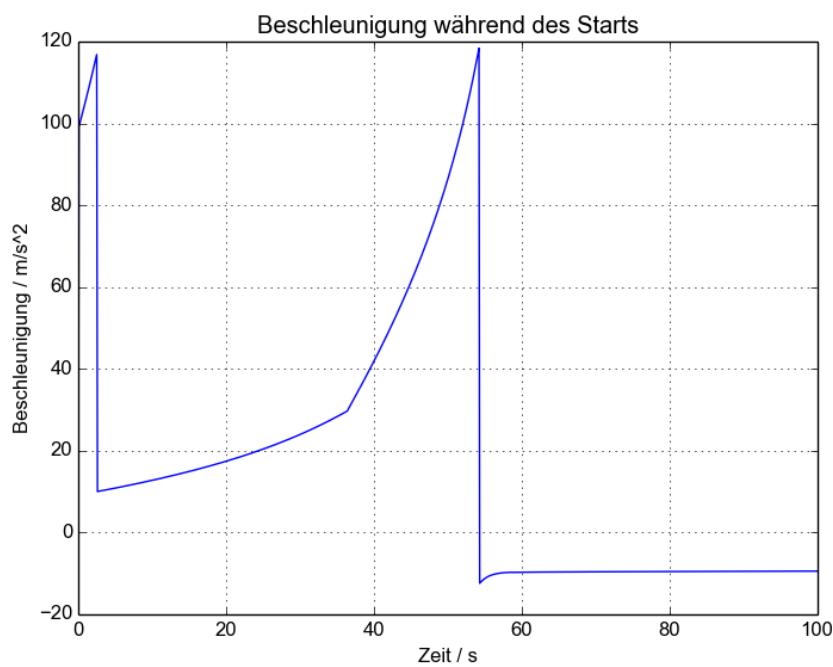


Abbildung 5.8: Verlauf der Beschleunigung während des Starts

5.1.2 Die Flugphase

Nach dem Ausbrennen der Stufen nimmt die Geschwindigkeit der Rakete langsam ab, bis sie an der Apoapsis 0 km/h ($t = 271,1$ s) erreicht. Der Geschwindigkeitverlauf ist in Abbildung 5.9 dargestellt. Die Geschwindigkeit verringert sich in Folge der Gravitationskraft und des Luftwiderstands, da keine Schubkraft, die diesen entgegenwirkt, mehr vorhanden ist.

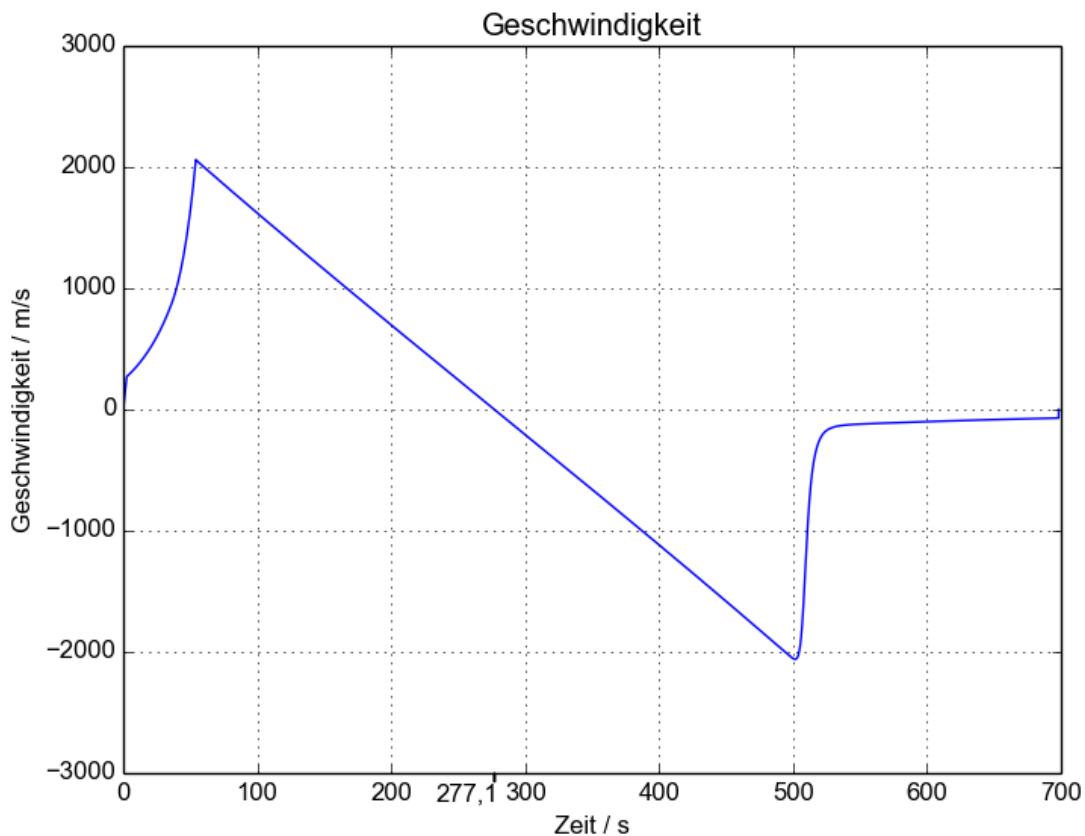


Abbildung 5.9: Verlauf der Geschwindigkeit über die Simulationsdauer

Nach der Apoapsis nimmt Geschwindigkeit weiter ab, d.h. sie nimmt eigentlich in negative Richtung zu. Dieses Verhalten setzt sich bis zum Wiedereintritt in die Troposphäre fort.

5.1.3 Die Wiedereintrittsphase

Durch den Eintritt in die Troposphäre wird der Betrag der Geschwindigkeit sehr schnell stark verringert. Dies hat seinen Grund im durch die schnell zunehmende Dichte stark ansteigenden Luftwiderstand. Dieser bremst die fallende Rakete auf die sogenannte maximale Fallgeschwindigkeit. Sie wird erreicht, wenn der Luftwiderstand der Gravitationskraft entspricht. Dadurch erreicht die auf die Rakete resultierende Kraft 0 N und somit erfährt die Rakete keine Beschleunigung mehr. Diese maximale Geschwindigkeit nimmt mit zunehmender Gravitationskraft, aber auch zunehmender Dichte, weiter ab, bis die Rakete auf der Planetenoberfläche aufschlägt.

5.2 Vergleich verschiedener zweistufiger Raketen

In Abbildung 5.10 sind die Höhenverläufe verschiedener zweistufiger Raketaufbauten zu sehen.

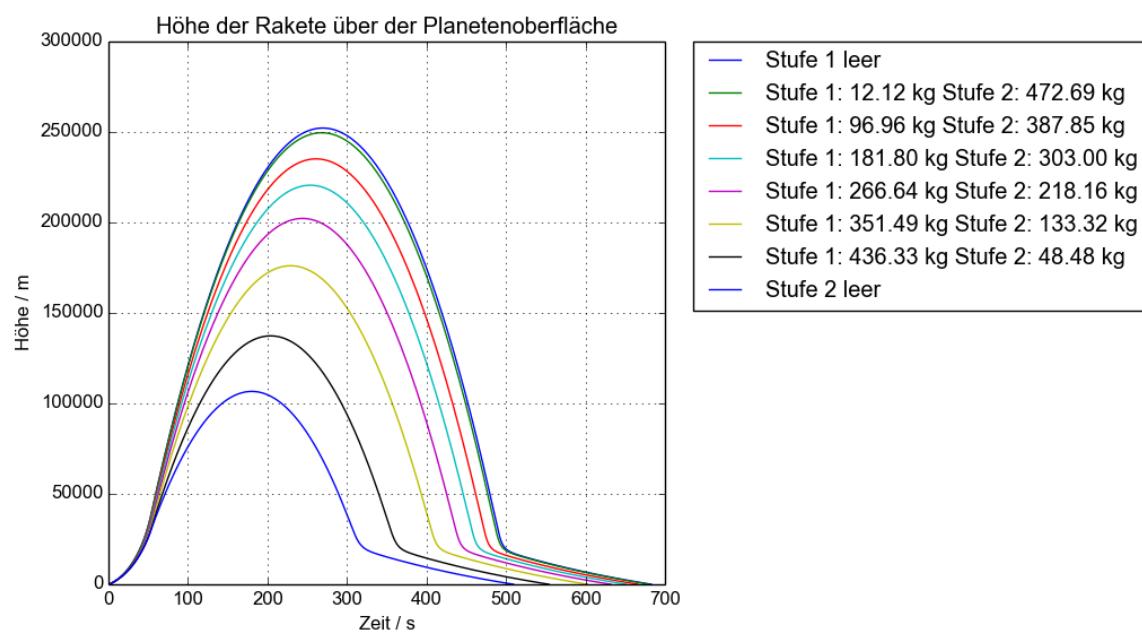


Abbildung 5.10: Die Höhenverläufe die mit verschiedenen zweistufigen Raketaufbauten erreicht wurden

Diese Raketaufbauten bestehen aus dem Booster der A150 und zwei Flüssigtreibstoffantrieben. Auf beide Flüssigtreibstofftanks wird der Treibstoff, der normalerweise

in einem Tank der A150 ist, auf beide Tanks verteilt. Die verschiedenen Kurven in Abbildung 5.10 begründen sich dadurch, dass dieses Verhältnis immer weiter verändert wird. Zu Beginn ist Stufe 1, das heißt die Stufe die nach dem Booster zündet, komplett leer, anschließend wird mit zunehmenden Treibstoff in der ersten Stufe simuliert.

Der Verlauf ähnelt dem der einstufigen Rakete in Abbildung 5.1. Er unterscheidet sich nur in der Höhe. Hierbei erreicht die Konfiguration, in der Stufe 2 komplett leer ist, eine Höhe von 106.625 m. Der Aufbau mit einer leeren Stufe 1 erreicht eine Höhe von 252.190 m. Die Apoapsiden der anderen Aufbauten liegen innerhalb dieses Intervalls. Aus den Werten lässt sich erkennen, dass die zweistufige Konfiguration niedrigere Höhen erreicht als die einstufige Rakete. Dies wird in Abbildung 5.11 illustriert.

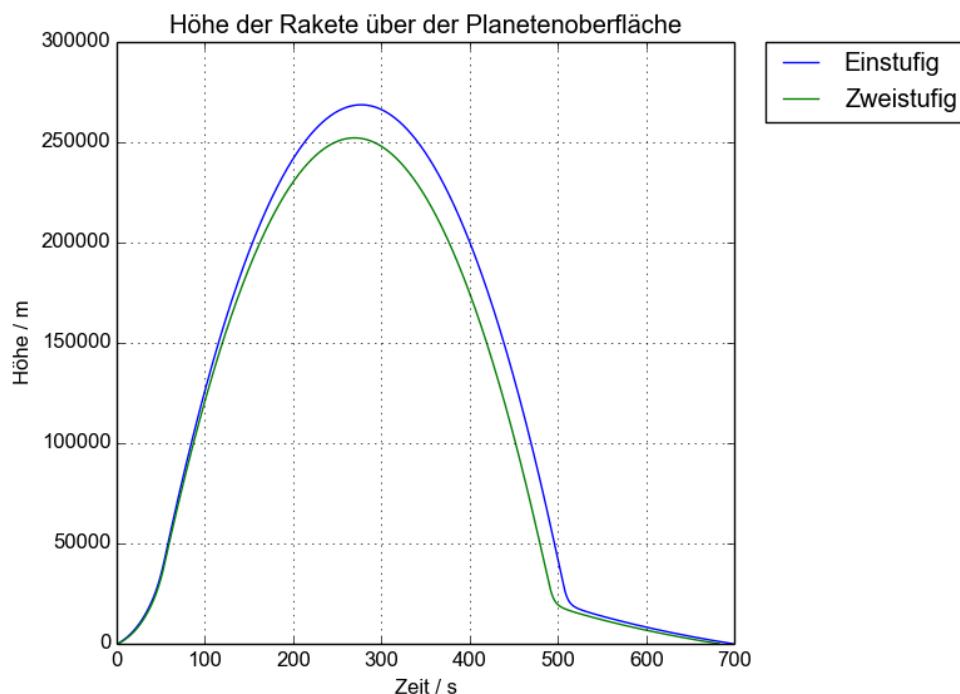


Abbildung 5.11: Vergleich des einstufigen Aufbaus mit dem besten zweistufigen Aufbau

5.2.1 Gründe für die unterschiedliche Simulationsergebnisse

Der Unterschied in den erreichten Höhen resultiert aus dem erhöhten Gewicht der zweistufigen Rakete. Dieses entsteht durch den zusätzlichen Tank und Antrieb, der bei einer zweiten Stufe vorhanden ist, und somit sein Eigengewicht zur Gesamtmasse der Rakete hinzufügt. So wiegt die einstufige Rakete nur 756,33 kg beim Start, die zweistufige hingegen 872,45 kg. Die Veränderung der Masse der Rakete für verschiedene Aufbauten ist in Abbildung 5.12 zu sehen.

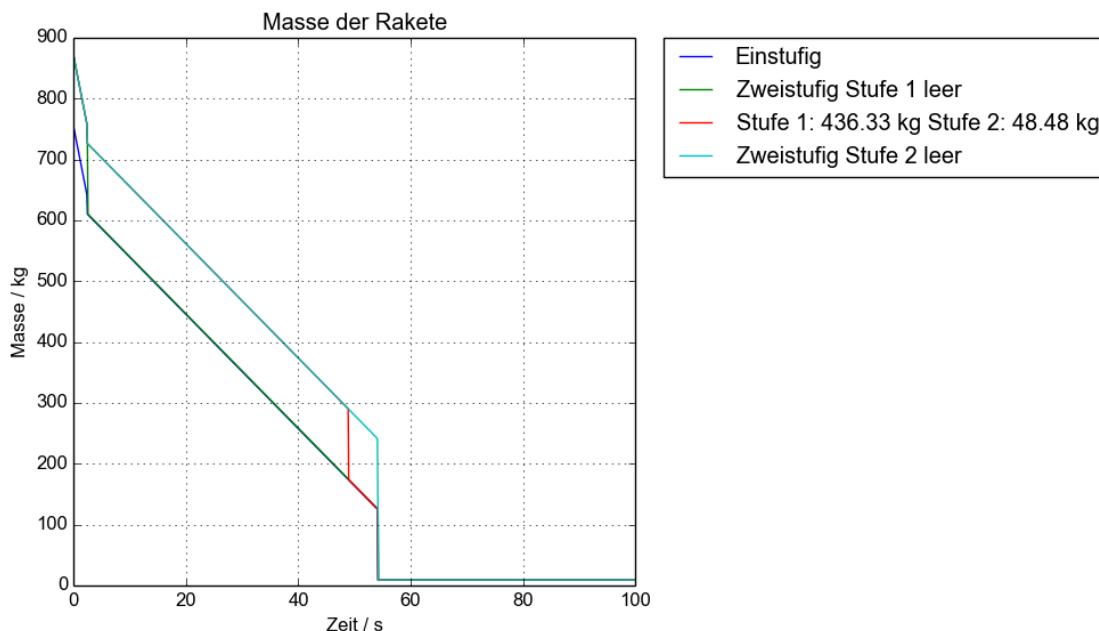


Abbildung 5.12: Verlauf der Massen verschiedener Aufbauten während der Startphase

In Abbildung 5.12 ist zu erkennen, dass die Masse der einstufigen Rakete beim Start geringer ist als die der zweistufigen Varianten. Dies setzt sich im weiteren Verlauf der Simulation fort, wobei die zweistufigen Aufbauten nach dem Abtrennen der ausgebrannten Stufen auch dem Kurvenverlauf der Masse der einstufigen Rakete folgen.

Die erhöhte Masse hat auch gleichzeitig eine erhöhte Gravitationskraft zur Folge. Dies ist in Abbildung 5.13 erkennbar.

Die Kurve der Gravitationskraft folgt, wie nach Gleichung (2.11) erwartet, dem Verlauf der Masse der Raketen. Besonders in niedrigen Höhen wirkt sich das erhöhte Gewicht

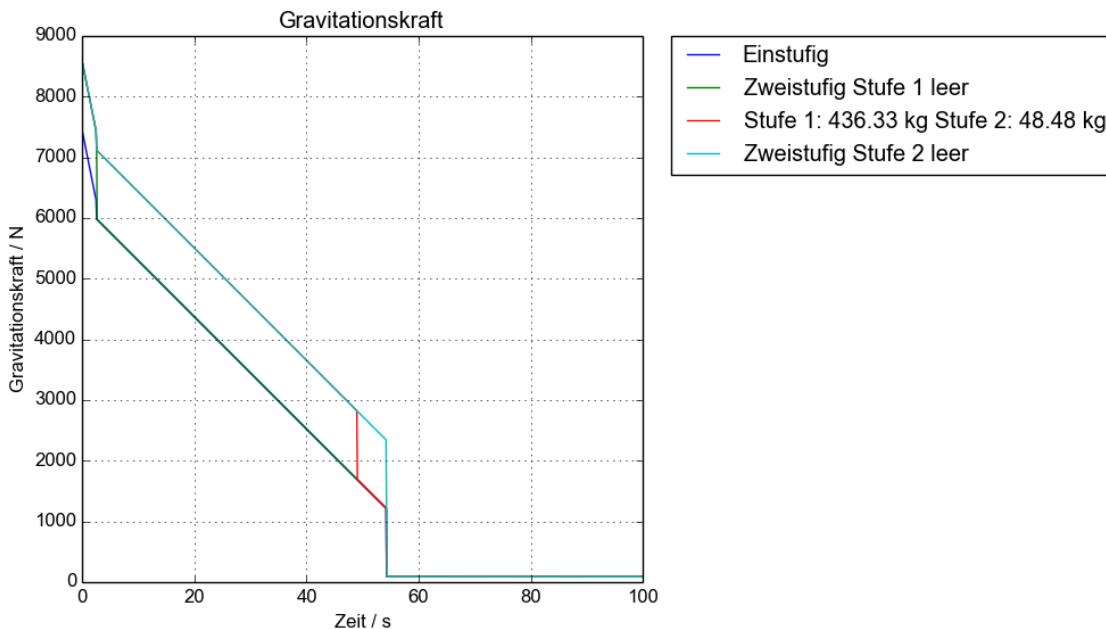


Abbildung 5.13: Verlauf der Gravitationskraft verschiedener Aufbauten während der Startphase

aus, da die Distanz d_{Rakete} der Massenpunkte von Planet und Rakete noch gering ist und sich quadratisch auswirkt, was die auf die Rakete resultierende Kraft verringert, die letztlich zur Beschleunigung der Rakete führt.

Betrachtet man den Schub der Raketen (siehe Abbildung 5.14) erkennt man, dass dieser für alle Raketen gleich ist, da die Brenndauer der Antriebe gleich bleibt. Nur die aktive Stufe ändert sich. Da der Abkopplungsprozess der Stufen in der Simulation keine Zeit in Anspruch nimmt, lässt sich der Stufenwechsel in der Schubkennlinie nicht erkennen.

Der Schub ist also nicht für die verringerte Apoapsis bei zweistufigen Raketen verantwortlich.

Die Kurve des Luftwiderstandes (siehe Abbildung 5.15) zeigt einen auf die zweistufige Rakete wirkenden, geringeren Luftwiderstand. Dies liegt allerdings an der geringeren Geschwindigkeit im Vergleich zur Simulation mit der einstufigen Rakete.

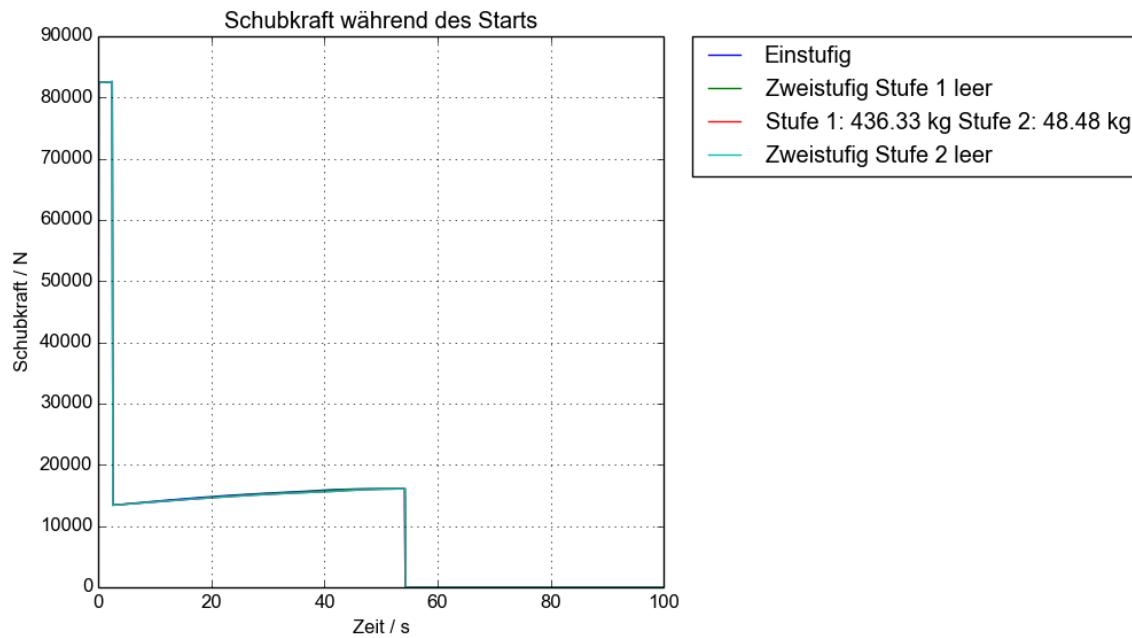


Abbildung 5.14: Verlauf der Schubkraft verschiedener Aufbauten während der Startphase

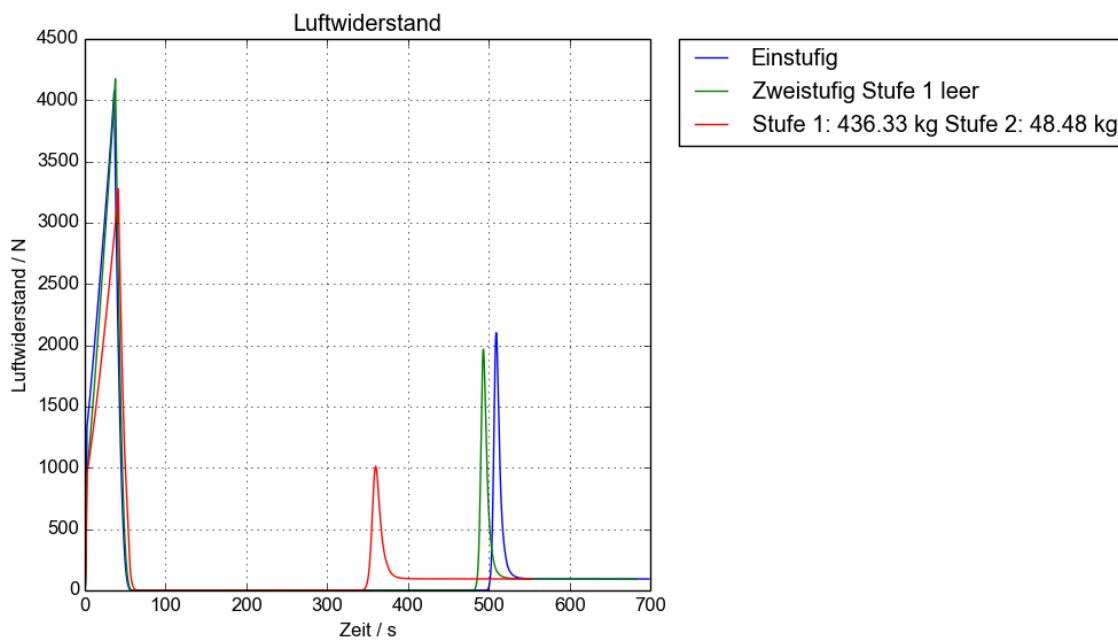


Abbildung 5.15: Verlauf des Luftwiderstands verschiedener Aufbauten während der Simulation

Der Verlauf der Geschwindigkeit lässt sich in Abbildung 5.16 erkennen. Auch die Tatsache, dass die zweistufigen Raketen eine kürzere Flugdauer haben, ist ablesbar.

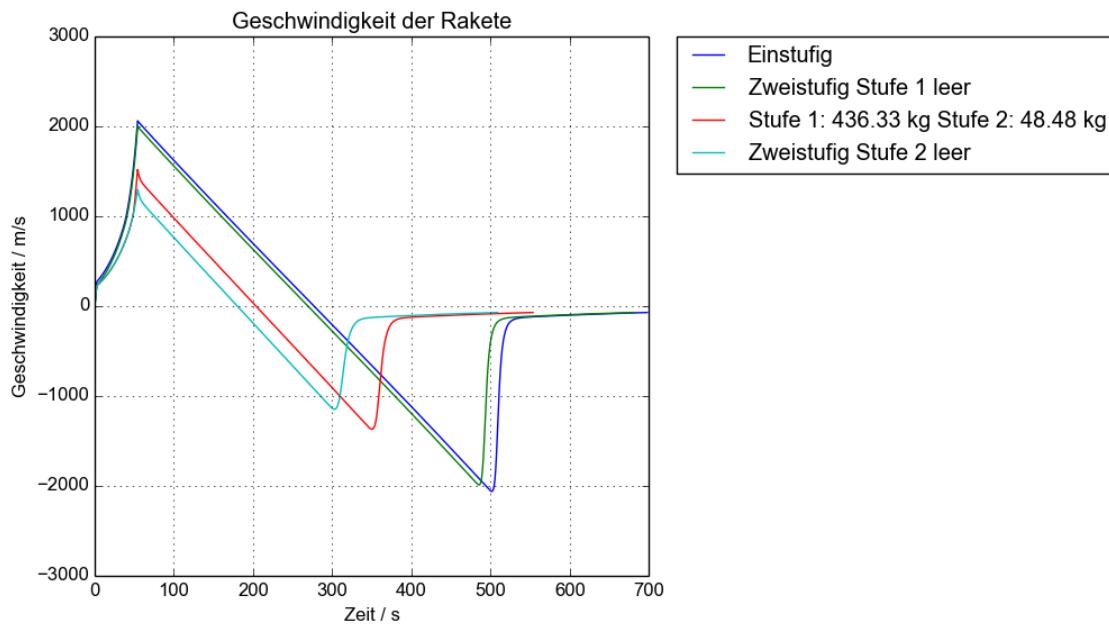


Abbildung 5.16: Verlauf der Geschwindigkeit verschiedener Aufbauten während der Simulation

5.2.2 Schlussfolgerung aus den Ergebnissen

Betrachtet man den Verlauf der Höhenverläufe mit zunehmendem Treibstoff in der zweiten Stufe, so lässt sich erkennen, dass diese Höhenverläufe gegen den Höhenverlauf der einstufigen Rakete konvergieren (vgl. Abbildung 5.17).

Dies Konvergenz lässt sich dadurch zeigen, dass die Masse der einstufigen Rakete um eine Nutzlast, die dem Eigengewicht einer zusätzlichen Stufe entspricht, ergänzt wird. Diese Nutzlast wird einmal an Stelle der ersten Stufe und einmal an Stelle der zweiten Stufe simuliert. Die Höhenverläufe dieser Simulation sind in Abbildung 5.18 zu sehen. In der Simulation sind einstufige Raketen also besser als zweistufige geeignet, um große Höhen zu erreichen. Dies liegt am zusätzlichen Gewicht, das durch den zweiten Antrieb und das Eigengewicht des Tankes entsteht. Das Ersetzen einer Stufe durch zwei äquivalente Stufen bringt also keine Vorteile für den Raketenflug.

Trotzdem finden sich in der modernen Raketenfahrt hauptsächlich mehrstufige Raketen. Dies hat seinen Grund darin, dass so unterschiedliche Antriebssysteme in einer

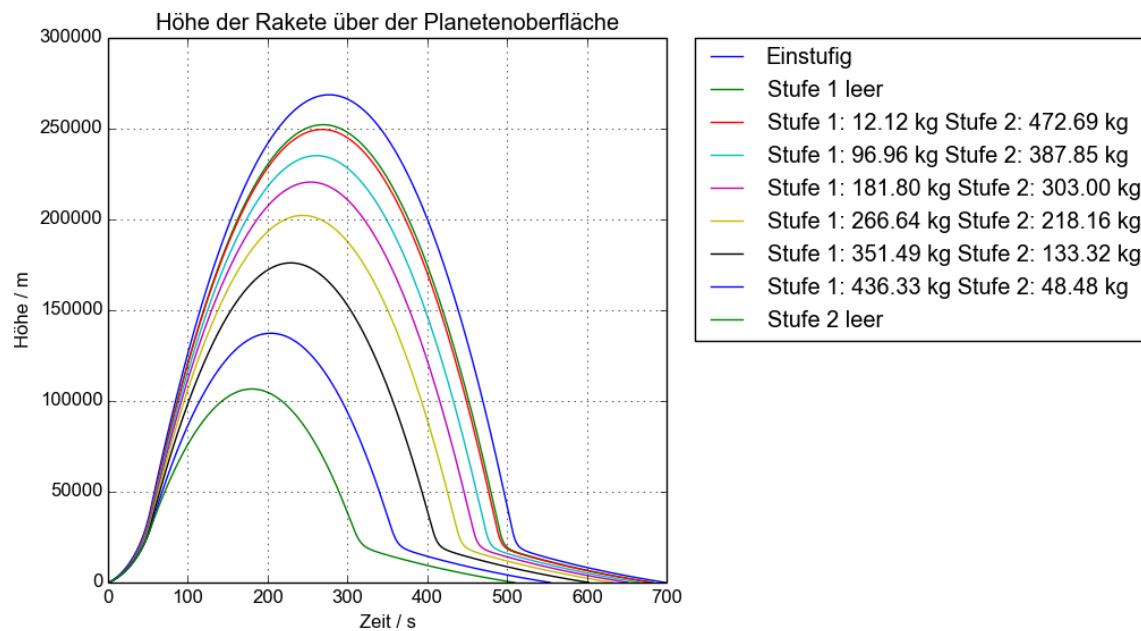


Abbildung 5.17: Höhenverläufe mit verschiedenen zweistufigen Aufbauten im Vergleich zum einstufigen Aufbau

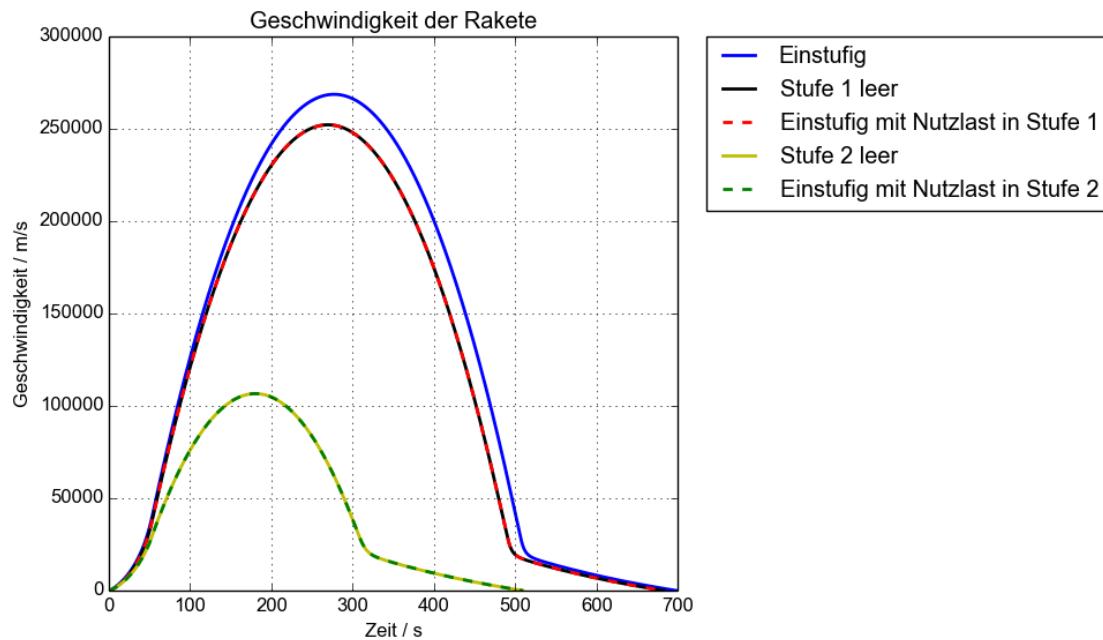


Abbildung 5.18: Höhenverläufe zweistufige Raketen im Vergleich zu einstufigen Raketen mit Nutzlast

Rakete kombiniert werden können. Dies ist zum Beispiel notwendig, um die hohen Kräfte beim Start aufzubringen. Dazu können Feststoffbooster (wie beim SpaceShuttle in Abbildung 1.2) oder Flüssigtreibstofftanks mit mehreren Antrieben verwendet wer-

den (wie bei der SpaceX Falcon 9-Rakete in Abbildung 1.1). Die Notwendigkeit dieser Art von Raketenaufbau lässt sich zeigen, indem in der Simulation der Booster der A150 weggelassen wird. Der daraus resultierende Höhenverlauf lässt sich in Abbildung 5.19 erkennen.

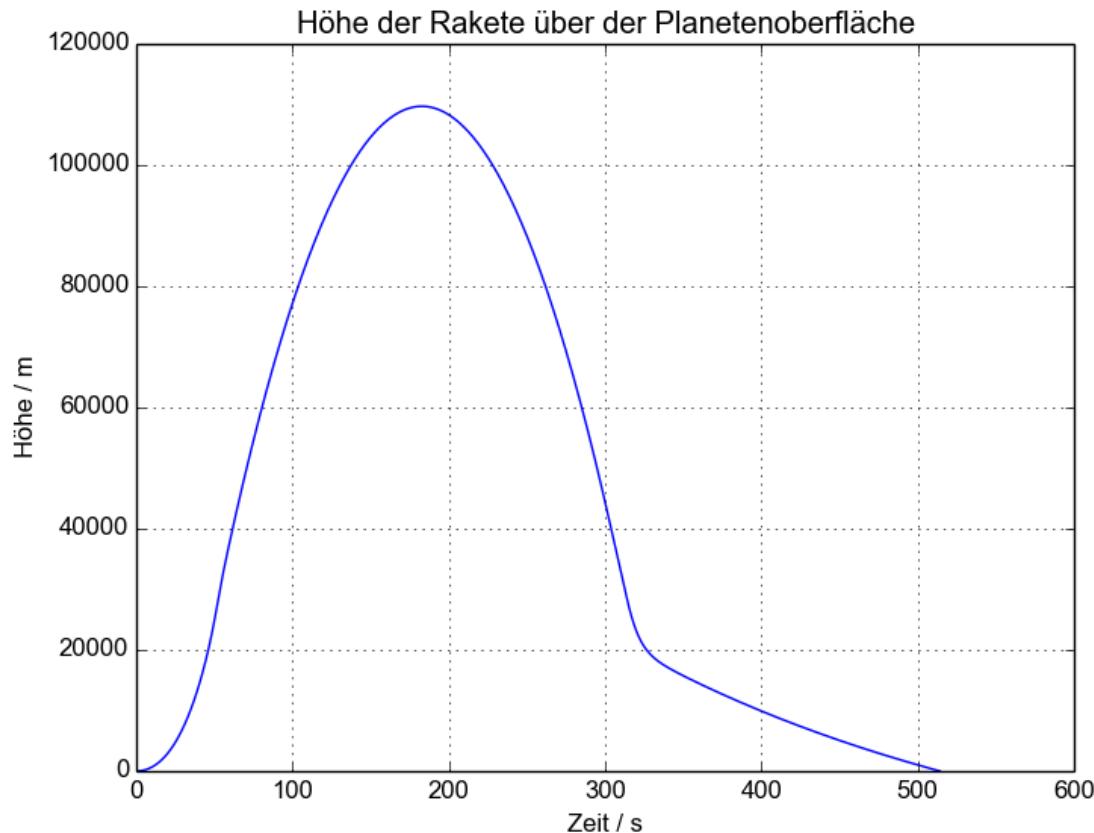


Abbildung 5.19: Höhenverläufe einer A150 ohne Booster während des Simulation

Diese erreicht nicht einmal ein Drittel der Höhe der normalen einstufigen Rakete. Des Weiteren können Antriebe mit geringem Schub verwendet werden, wenn keine große Beschleunigung notwendig ist. Dies ist zum Beispiel dann der Fall, wenn Sonden für ihre Flugbahn weit entfernt von der Erde beschleunigt werden.

6 Validierung der Ergebnisse

Im Folgenden sollen die Ergebnisse der Simulation validiert werden. Dazu werden die Ergebnisse mit einem Artikel über die A150 und ihre technischen Eigenschaften verglichen.

In [21] wird die Ausbrenndauer des ersten Boosters mit 2,5 s angegeben, ähnlich der Angabe in [19], hier wird eine Brenndauer von 2,3 s angegeben. Dies entspricht der Brenndauer in der Simulation, die 2,5 s beträgt.

Die zweite Stufe brennt in der Simulation bis 54,2 s mit einer Geschwindigkeit von 2.061 m/s. Nach [19] ist dies nach 52,85 s und bei einer Geschwindigkeit von 2.177 m/s der Fall. Diese Abweichung ist durch die kürzere Brenndauer in [19] zurückzuführen. Die Beschleunigung zu diesem Zeitpunkt in der Simulation beträgt 118,38 m/s², nach einer weiteren Sekunde wäre die Abweichung auf Rundungsfehler in der Simulation und auf die Einheitenkonvertierung zurückführbar. Dies stützt die Validität der Ergebnisse.

Vergleicht man die während des Fluges maximal erreichten Höhen findet sich eine größere Diskrepanz zwischen Simulation und [19]. Bei einem tatsächlichen Flug erreichte die A150 eine Höhe von 186,5 Landmeilen, was einer Höhe von 300.143 m entspricht; in der Simulation wird allerdings nur eine Höhe von 268.680 m erreicht. Vergleicht man dies allerdings mit der für den Flug vorhergesagten Apoapsis von 283.464 m, beträgt die Abweichung nur noch 14.784 m statt 31.756 m. Dies liegt in den Abweichungen während der Startphase begründet, da durch die höheren Endgeschwindigkeiten in der Startphase auch größere Höhen erreicht werden können. Mit einer Abweichung von 5 % bezogen auf die Simulationsapoapsis stimmt dies mit der 5 % Abweichung der Geschwindigkeiten während der Startphase überein und ist somit erklärbar. Des Weiteren handelt es sich bei dem in [19] behandelten Flug 4.30GG auch um den Flug, in dem die größte Höhe während eines Fluges der A150 erreicht wurde.

Die Plausibilität der Ergebnisse und des Verhaltens der Rakete während des Fluges wurde bereits in Kapitel 5.1 erläutert. Die Ergebnisse und das Verhalten sind in sich stimmig und unterstreichen somit die Validität der Simulation.

7 Weitere Verwendung der Simulation

Die Arbeit ist mit dem Ergebnis, dass einstufige Raketen vorteilhafter gegenüber äquivalenter zweistufiger Raketen sind, abgeschlossen. In Zukunft wäre es allerdings noch interessant, andere Raketen als die in dieser Arbeit betrachtete Aerobee 150 zu modellieren und zu simulieren. Auch die Betrachtung von Raketen mit mehr als zwei Stufen und mit verschiedenen Antriebstechnologien wäre möglich.

Eine Erweiterung des Programms, die zusätzlich auch Orbitalflüge simuliert, ist eine mögliche Weiterführung der Arbeit. Hierzu müssten allerdings Vereinfachungen, die im Rahmen der Modellierung gemacht wurden rückgängig gemacht werden. So ist für eine realitätsnahe Modellierung die Berücksichtigung des Schwerpunkts und der Planetenrotation notwendig.

Abbildungsverzeichnis

1.1 Falcon 9 Rakete von Space X mit gekennzeichnetem Aufbau nach [2]	2
1.2 Start eines Space Shuttle mit aktiven Feststoffantrieben aus [4]	4
1.3 Deep Space 1 Sonde mit ihrem Ionenantrieb aus [5]	5
1.4 Render der Sonde New Horizon aus [6]	6
1.5 Elliptische Flugbahn mit markierten Apsiden (A & B) [8]	7
2.1 Antares Rakete während der Vorbereitung mit eingezeichneter Rotationsachse und Schwerpunkt (vgl. [10])	11
2.2 Start der SpaceX Falcon 9 mit eingezeichneten Kraftvektoren (vgl. [12])	14
2.3 RS-68 Antrieb bei einem NASA Test mit eingezeichneten Kenngrößen auf Basis von [13]	15
2.4 Schichten der Erdatmosphäre nach [16]	17
4.1 Aerobee 150 Rakete mit markiertem Aufbau auf Basis von [20]	34
5.1 Höhenverlauf der einstufigen Rakete über die Zeit	37
5.2 Verlauf der Masse der Rakete während des Starts	38
5.3 Verlauf der Schubkräfte während des Starts	39
5.4 Verlauf des Druckes über die Simulationsdauer	40
5.5 Verlauf der Geschwindigkeit während des Starts	41
5.6 Verlauf des Luftwiderstands während des Starts	41
5.7 Verlauf der Gravitationskraft während des Starts	42
5.8 Verlauf der Beschleunigung während des Starts	42
5.9 Verlauf der Geschwindigkeit über die Simulationsdauer	43
5.10 Die Höhenverläufe die mit verschiedenen zweistufigen Raketaufbauten erreicht wurden	44
5.11 Vergleich des einstufigen Aufbaus mit dem besten zweistufigen Aufbau	45

5.12 Verlauf der Massen verschiedener Aufbauten während der Startphase	46
5.13 Verlauf der Gravitationskraft verschiedener Aufbauten während der Start- phase	47
5.14 Verlauf der Schubkraft verschiedener Aufbauten während der Startphase	48
5.15 Verlauf des Luftwiderstands verschiedener Aufbauten während der Si- mulation	48
5.16 Verlauf der Geschwindigkeit verschiedener Aufbauten während der Si- mulation	49
5.17 Höhenverläufe mit verschiedenen zweistufigen Aufbauten im Vergleich zum einstufigen Aufbau	50
5.18 Höhenverläufe zweistufige Raketen im Vergleich zu einstufigen Raketen mit Nutzlast	50
5.19 Höhenverläufe einer A150 ohne Booster während des Simulation	51

Quellcodeverzeichnis

3.1 Klasse Rocket	20
3.2 Funktion append part	21
3.3 Funktion set mass	21
3.4 Funktion decouple	22
3.5 Klasse Rocket Part	22
3.6 Klasse Tank	23
3.7 Funktion get mass	24
3.8 Klasse Planet	25
3.9 Klasse Layer	26
3.10 Funktionen Layer	26
3.11 Funktion calc height below	27
3.12 Funktion get layer	28

Literaturverzeichnis

- [1] Rogers, L. *It's only Rocket Science*. Isle of Wight: Springer Science, 2008.
- [2] (Hrsg.), SpaceX. *Falcon 9*. o.J. URL: <http://www.spacex.com/falcon9> (besucht am 28. 12. 2015).
- [3] Greatrix, D. R. *Powered Flight*. London: Springer Verlag, 2012.
- [4] Aeronautics, National / Administration, Space. *STS-120 Launch*. 2007. URL: <https://commons.wikimedia.org/wiki/File:STS120LaunchHiRes-edit1.jpg> (besucht am 28. 12. 2015).
- [5] Aeronautics, National / Administration, Space. *Deep Space 1 moves to CCAS for testing*. 1998. URL: <http://science.ksc.nasa.gov/gallery/photos/1998/captions/KSC-98PC-1191.html> (besucht am 28. 12. 2015).
- [6] Aeronautics, National / Administration, Space. *Pluto: Past and Future*. 2008. URL: https://www.nasa.gov/mission_pages/newhorizons/images/index.html?id=354382 (besucht am 28. 12. 2015).
- [7] Curtis, H. D. *Orbital Mechanics for Engineering Students*. Burlington: Butterworth-Heinemann, 2010.
- [8] (Psdn.), Stanmar. *Keplerellipse*. 2006. URL: <https://commons.wikimedia.org/wiki/File:EllipseInPolarCoords2.svg> (besucht am 28. 12. 2015).
- [9] Gross, D. u. a. Darmstadt: Springer Verlag, 2013.
- [10] National Aeronautics and Space Administration. *Antares Rocket Preparation*. 2013. URL: <https://www.flickr.com/photos/nasahqphoto/8664163104/> (besucht am 26. 12. 2015).
- [11] Gross, D. u. a. *Technische Mechanik 3 - Kinetik*. Darmstadt: Springer Verlag, 2006.

- [12] National Aeronautics and Space Administration. *KSC-2015-1358*. 2015. URL: <https://www.flickr.com/photos/nasakennedy/16318394040/in/album-72157650531580732/> (besucht am 07. 12. 2015).
- [13] National Aeronautics and Space Administration. *RS-68 being tested at NASA's Stennis Space Center*. 2000. URL: http://www.nasa.gov/images/content/148709main_d4_testing_08.jpg (besucht am 01. 12. 2015).
- [14] Böswirth, L. *Technische Strömungslehre*. Wiesbaden: Vieweg+Teubner, 2010.
- [15] Sigloch, H. *Technische Fluidmechanik*. Heidelberg: Springerverlag, 2014.
- [16] L., Niko. *Aufbau der Erdatmosphäre (Europa, Sommer)*. 2006. URL: https://upload.wikimedia.org/wikipedia/commons/9/9c/Atmosph%C3%A4re_Aufbau.jpg (besucht am 01. 12. 2015).
- [17] Faust, H. *Der Aufbau der Erdatmosphäre*. Wiesbaden: Springer Fachmedien Wiesbaden GmbH, 1968.
- [18] Diverse. *Atmosphärischer Temperaturgradient*. 2015. URL: https://de.wikipedia.org/wiki/Atmosph%C3%A4rischer_Temperaturgradient (besucht am 30. 12. 2015).
- [19] Busse, J. R. / Leffer, M. T. „The Aerobee 100, 150 and 300 Series Sounding Rockets“. *High Power Rocketry* 7 (1994), S. 67–106.
- [20] Burnett, K. *Aerobee 150 Rocket*. o.J. URL: http://www.pbase.com/kent_burnett/image/109986285 (besucht am 26. 12. 2015).
- [21] Vought Astronautics (Hrsg.) *Performance Summary for the Aerobee 150A*. Technischer Bericht. Vought Astronautics, 1961.

Anhang

Anhang A: Formula.py	B
Anhang B: Atmospheres.py	E
Anhang C: Layers.py	F
Anhang D: Planets.py	H
Anhang E: RocketParts.py	I
Anhang F: Rockets.py	L
Anhang G: RocketSim.py	O
Anhang H: Datas.py	U
Anhang I: Simulationswerte	W

Anhang A: Formula.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/Formula.py

Seite 1 von 3

```
1 """
2 Author:      Felix Bräunling
3 Description: This module contains the formulas for calculating the
4             necessary values for the simulation on the basis of
5             flight, orbital, gravitational mechanics, as well as
6             other physical influences like drag.
7 """
8
9 import math
10
11 G = 6.67408e-11    # Newton's gravity constant           [m^3/(kg*s^2)]
12 M = 0.02896         # Molar mass of atmosphere gasses       [kg/mol]
13 R = 8.314           # universal gas constant            [J/(K*mol)]
14 g = 9.807           # Earths gravitational acceleration     [m/s^2]
15
16
17 def thrust(mass_change, velocity_propellant, surface_nozzle, pressure_nozzle, pressure_ambient):
18     """
19     Calculates the thrust the engine is producing based on the following inputs
20     :param mass_change: Change of mass of propellant (Double)
21     :param velocity_propellant: Velocity of propellant (Double)
22     :param surface_nozzle: Surface area of the exit nozzle (Double)
23     :param pressure_nozzle: Pressure in the area of the nozzle (Double)
24     :param pressure_ambient: Pressure of the ambient (Double)
25     :return: Thrust produced by the engine (Double) [N]
26     """
27     return mass_change * velocity_propellant + surface_nozzle * (pressure_nozzle -
28     pressure_ambient)
29
30
31 def drag(density_ambient, velocity_rocket, surface_rocket, drag_coefficient):
32     """
33     Calculates the drag the rocket experiences based on the following inputs
34     :param density_ambient: Density of the medium the rocket is travelling in (Double)
35     :param velocity_rocket: Velocity of the rocket in travelling direction (Double)
36     :param surface_rocket: Surface area of the rocket that is exposed to drag (Double)
37     :param drag_coefficient: Drag coefficient of the rocket (Double)
38     :return: Drag experienced by the rocket (Double) [N]
39     """
40     return 1.0 / 2.0 * density_ambient * velocity_rocket**2 * surface_rocket * drag_coefficient
41
42
43 def gravity(mass_planet, mass_rocket, distance):
44     """
45     Calculates the gravitational force between the planet and the rocket,
46     raises an exception if distance is zero or force is negative
47     :param mass_planet: Mass of the planet (Double)
48     :param mass_rocket: Mass of the rocket (Double)
49     :param distance: Distance between the center of mass of the rocket and the planet (non-zero
50     Double)
51     :return: gravitational fore (positive Double) [N]
52     """
53     global G
54     gravity_now = G * (mass_planet * mass_rocket) / (distance**2)
55     if gravity_now < 0:
56         raise ValueError("No negative gravity possible!")
57     return gravity_now
58
59
60 def pressure(pressure_height_low, temp_gradient, height_rocket, height_low, temp_height_low):
61     """
62     Calculates the pressure at a certain height
63     :param pressure_height_low: Pressure at the lower end of the atmosphere layer (Double)
64     :param temp_gradient: Temperature gradient for the atmosphere layer (Double)
65     :param height_rocket: Height of the rocket above the lower end of the layer (Double)
66     :param height_low: Height of the lower end of the atmosphere layer (Double)
67     :param temp_height_low: Temperature at the lower end of the atmosphere layer (Double)
68     :return: Pressure at the height of the rocket [Pa]
69     """
70     global g
71     global M
72     global R
```

Anhang A: Formula.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/Formula.py

Seite 2 von 3

```
71     if temp_gradient == 0:
72         expo = 0
73     else:
74         expo = (M * g) / (R * temp_gradient)
75         height_diff = height_rocket - height_low
76         quot = (temp_gradient * height_diff) / temp_height_low
77         return pressure_height_low * abs(1 - quot)**expo
78
79
80 def temperature(temp_height_low, temp_gradient, height_low, height_rocket):
81     """
82     Calculates the temperature at a certain height
83     :param temp_height_low: Temperature at the lowest point of the atmosphere layer (Double)
84     :param temp_gradient: Temperature gradient for the atmosphere layer (positive or negative
Double)
85     :param height_low: Height of the lower end of the atmosphere layer (Double)
86     :param height_rocket: Height of the rocket (Double)
87     :return: Temperature at the height of the rocket (Double) [K]
88     """
89     height_diff = height_rocket - height_low
90     return temp_height_low + temp_gradient*height_diff
91
92
93 def density(pressure_medium, temp_height):
94     """
95     Calculates the atmospheres density at a certain height
96     :param pressure_medium: The pressure of the medium depending on the height (Double)
97     :param temp_height: The Temperature of the medium depending on the height (Double)
98     :return: Density of the medium depending on the height of the rocket (Double) [kg/m^3]
99     """
100    global R
101    global M
102    return (pressure_medium * M) / (R * temp_height)
103
104
105 def resulting_force(force_thrust, force_drag, force_gravity):
106     """
107     Calculates the resulting force from the three main forces
108     :param force_thrust: Force produced by thrust (Double)
109     :param force_drag: Force produced by drag (Double)
110     :param force_gravity: Force applied by gravity (Double)
111     :return: Sum of the three forces (Double) [N]
112     """
113     return force_thrust - force_drag - force_gravity
114
115
116 def angle(velocity_rocket, pos_change, radius_planet, height_rocket):
117     """
118     Calculates the change of the angle of the rocket to the horizon
119     :param velocity_rocket: Speed of the rocket (Double)
120     :param pos_change: Change of the position of the rocket (Double)
121     :param radius_planet: Radius of the planet the rocket is orbiting (Double)
122     :param height_rocket: Height of the rocket above the planet (Double)
123     :return: Returns the new angle of the rocket to the horizon (Double) [grad]
124     """
125     angle_now = math.acos(1 / velocity_rocket * pos_change * (radius_planet + height_rocket) /
radius_planet)
126     return math.degrees(angle_now)
127
128
129 def acceleration(force_result, mass_rocket):
130     """
131     Calculates the acceleration of the rocket based on its mass and the force it's experiencing
132     :param force_result: Resulting force of drag, thrust and gravity (Double)
133     :param mass_rocket: Mass of the rocket depending on parts and carried fuel (Double)
134     :return: Acceleration of the rocket (Double) [m/s^2]
135     """
136     return force_result / mass_rocket
137
138
139 def velocity(velocity_t0, duration_interval, acceleration_rocket):
140     """
```

Anhang A: Formula.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/Formula.py

Seite 3 von 3

```
141     Calculates the velocity of the rocket at a certain time interval
142     :param velocity_t0: Velocity at the beginning of the time interval (Double)
143     :param duration_interval: Duration of the interval (Double)
144     :param acceleration_rocket: Acceleration of the rocket during the interval (Double)
145     :return: Velocity of the rocket at the end of the interval (Double) [m/s]
146     """
147     return velocity_t0 + duration_interval * acceleration_rocket
148
149
150 def way(duration_interval, velocity_rocket, acceleration_rocket):
151     """
152     Calculates the change of the position of the rocket at a certain time
153     :param duration_interval: Duration of the interval (Double)
154     :param velocity_rocket: Velocity of the rocket during the interval (Double)
155     :param acceleration_rocket: Acceleration of the rocket during the interval (Double)
156     :return: Change in position of the rocket at the end of the time interval (Double) [m]
157     """
158     return duration_interval * velocity_rocket + 1.0 / 2.0 * acceleration_rocket *
159     duration_interval**2
160
161 def res_x(factor_result, angle_rocket):
162     """
163     Calculates the resulting force in x-direction
164     :param factor_result: resulting factor in rocket direction (Double)
165     :param angle_rocket: angle the rocket is facing to the horizon (Double) [grad]
166     :return: resulting factor in x-direction (Double) [N]
167     """
168     angle_rocket_rad = math.radians(angle_rocket)
169     return factor_result * math.cos(angle_rocket_rad)
170
171 def res_y(factor_result, angle_rocket):
172     """
173     Calculates the resulting force in y-direction
174     :param factor_result: resulting factor in rocket direction (Double)
175     :param angle_rocket: angle the rocket is facing to the horizon (Double) [grad]
176     :return: resulting factor in y-direction (Double) [N]
177     """
178     angle_rocket_rad = math.radians(angle_rocket)
179     return factor_result * math.sin(angle_rocket_rad)
180
181
182 def position(pos_t0, way_traveled):
183     """
184     Calculates the new position of the rocket
185     :param pos_t0: Last position of the rocket (Double)
186     :param way_traveled: Length traveled of the rocket (Double)
187     :return: New Position of the rocket
188     """
189     return pos_t0 + way_traveled
190
191
192 def vector_addition(vector):
193     """
194     Calculates the magnitude of a given vector of n = 2
195     :param vector: Vector of n = 2 (Tuple -> (Double, Double))
196     :return: Magnitude of a given vector Double [ ]
197     """
198     return math.sqrt(vector[0]**2 + vector[1]**2)
```

Anhang B: Atmospheres.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/Atmospheres.py

Seite 1 von 1

```
1 """
2 Author:          Felix Bräunling
3 Description:    This class describes the atmosphere of a planet consisting of different layers.
4 """
5
6 from Layers import Layer
7
8
9 class Atmosphere(object):
10     def __init__(self):
11         self.layers = []
12
13     def add_layer(self, new_layer):
14         """
15             Adds a new layer to the atmosphere, the new layer gets added at the end of the layer
16             list, this the new layer
17             is the highest one.
18             :param new_layer: A layer object describing an atmospheric layer (Layer)
19         """
20         if type(new_layer) != Layer:
21             raise ValueError
22         self.layers.append(new_layer)
23
24     def calc_height_below(self, layer_index):
25         """
26             Calculates the accumulated height of the layers below the indexed layer
27             :param layer_index: Index of the layer the height below is calculated (Integer)
28             :return: Accumulated height of the layers below the indexed one (Double) [m]
29         """
30         height_below = 0
31         for i in range(0, layer_index):
32             height_below += self.layers[i].get_width()
33
34     def get_layer(self, height_rocket):
35         """
36             Returns the layer the rocket is at the moment.
37             :param height_rocket: Height of the rocket above the planets surface (Double)
38             :return: The Layer the rocket currently is into (Layer)
39         """
40         current_layer = self.layers[0]
41         for i in range(0, len(self.layers)):
42             height_below = self.calc_height_below(i)
43             if height_below > height_rocket:
44                 break
45             current_layer = self.layers[i]
46
47         return current_layer
48
49     def get_layers(self):
50         """
51             Returns a list of the layers of this atmosphere object
52             :return: Layers of this atmosphere (List)
53         """
54
55         return self.layers
```

Anhang C: Layers.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/Layers.py

Seite 1 von 2

```
1 """
2 Author:      Felix Bräunling
3 Description: This class is used to describe different layers that make up an atmosphere.
4 """
5
6 import Formula
7
8
9 class Layer(object):
10
11     def __init__(self, width_layer, temp_gradient, temp_low, pressure_low):
12         """
13             Setup of an object to describe a layer of an atmosphere
14             :param width_layer: Height the layer is spanning over (Double) [m]
15             :param temp_gradient: Change in temperature over distance in the layer (Double) [K/m]
16             :param temp_low: Temperature at the lower end of the layer (Double) [K]
17             :param pressure_low: Pressure at the lower end of the layer (Double) [Pa]
18         """
19         self.width_layer = width_layer
20         self.temp_gradient = temp_gradient
21         self.temp_low = temp_low
22         self.pressure_low = pressure_low
23
24     def get_width(self):
25         """
26             :return: The width (vertical) of the atmospheric layer (Double) [m]
27         """
28         return self.width_layer
29
30     def set_width(self, new_width_layer):
31         """
32             Allows to set the width of the current layer
33             :param new_width_layer: New width of the atmospheric layer (Double) [m]
34         """
35         self.width_layer = new_width_layer
36
37     def get_temp_gradient(self):
38         """
39             :return: The temperature gradient of the current atmospheric layer (Double) [K/m]
40         """
41         return self.temp_gradient
42
43     def set_temp_gradient(self, new_temp_gradient):
44         """
45             Allows to set the temperature gradient of the atmospheric layer (Double)
46             :param new_temp_gradient: New temperature gradient of the atmospheric layer (Double) [K/
47             """
48         self.temp_gradient = new_temp_gradient
49
50     def get_temp_low(self):
51         """
52             :return: The temperature at the lower end of the layer (Double) [K]
53         """
54         return self.temp_low
55
56     def set_temp_low(self, new_temp_low):
57         """
58             Allows to set the temperature at the lower end of the atmospheric layer
59             :param new_temp_low: New temperature at the lower end of the layer (Double) [K]
60         """
61         self.temp_low = new_temp_low
62
63     def get_pressure_low(self):
64         """
65             :return: The pressure at the lower end of the layer (Double) [Pa]
66         """
67         return self.pressure_low
68
69     def set_pressure_low(self, new_pressure_low):
70         """
71             Allows to set the pressure at the lower end of the atmospheric layer
```

Anhang C: Layers.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/Layers.py

Seite 2 von 2

```
72     :param new_pressure_low: New pressure at the lower end of the atmospheric layer (Double)
73     [Pa]
74     """
75     self.pressure_low = new_pressure_low
76
77     def get_pressure_now(self, height_rocket, height_layer_below):
78         """
79             Calculates the pressure at the height the rocket currently is in
80             :param height_rocket: Height of the rocket above the planets surface (Double)
81             :param height_layer_below: Accumulated height of the atmospheric layers beneath the
82                 current layer (Double)
83             :return: Pressure at the current height in the layer (Double) [Pa]
84             """
85     return Formula.pressure(self.pressure_low, self.temp_gradient, height_rocket,
86     height_layer_below, self.temp_low)
87
88     def get_temperature_now(self, height_rocket, height_layer_below):
89         """
90             Calculates the temperature at the height the rocket currently is in
91             :param height_rocket: Height of the rocket above the planets surface (Double)
92             :param height_layer_below: Accumulated height of the atmospheric layers beneath the
93                 current layer (Double)
94             :return: Temperature at the current height in the layer (Double) [K]
95             """
96     return Formula.temperature(self.temp_low, self.temp_gradient, height_layer_below,
97     height_rocket)
98
99     def get_density_now(self, height_rocket, height_layer_below):
100        """
101            Calculates the density of the medium at the height the rocket is currently in
102            :param height_rocket: Height of the rocket above the planets surface (Double)
103            :param height_layer_below: Accumulated height of the atmospheric layers beneath the
                current layer (Double)
104            :return: Density of the medium the rocket is in at a given height (Double) [kg/m^3]
105            """
106            temperature_now = self.get_temperature_now(height_rocket, height_layer_below)
107            pressure_now = self.get_pressure_now(height_rocket, height_layer_below)
108            return Formula.density(pressure_now, temperature_now)
```

Anhang D: Planets.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/Planets.py

Seite 1 von 1

```
1 """
2 Author:          Felix Bräunling
3 Description:    This class describes the planet (or celestial body) the rocket is taking off
4 """
5
6
7 class Planet(object):
8
9     def __init__(self, mass_planet, radius_planet, pos_planet):
10
11         """
12             Setup of an object that describes a planet (or celestial body)
13             :param mass_planet: Mass of the planet (Double) [kg]
14             :param radius_planet: Radius of the planet (center to surface) (Double) [m]
15             :param pos_planet: Position of the planets center of mass as x and y Coordinates (Tuple -> (Double, Double))
16         """
17         self.mass_planet = mass_planet
18         self.radius_planet = radius_planet
19         self.pos_planet = pos_planet
20
21     def get_mass(self):
22
23         """
24             :return: Mass of the planet (Double) [kg]
25         """
26         return self.mass_planet
27
28     def get_radius(self):
29
30         """
31             :return: Radius of the planet (Double) [m]
32         """
33         return self.radius_planet
34
35     def get_pos(self):
36
37         """
38             :return: Position of the center of mass of the planet (Tuple -> (Double, Double))
39         """
40         return self.pos_planet
```

Anhang E: RocketParts.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/RocketParts.py

Seite 1 von 3

```
1 """
2 Author:          Felix Bräunling
3 Description:    Those classes are used to describe rocket parts in general and different
4 variants of rocket parts
5 """
6
7 class RocketPart(object):
8 """
9     Base class for rocket parts with their attributes mass_part, surface_part and
10    drag_coefficient_part
11 """
12     def __init__(self, mass_part, surface_part, drag_coefficient_part):
13         """
14             Sets up a rocket part with value validation
15             :param mass_part: Mass of the part (Not negative Double) [kg]
16             :param surface_part: Surface area of the part (Not negative Double) [m^2]
17             :param drag_coefficient_part: Drag coefficient of the part (Not negative Double) [1]
18         """
19         if mass_part < 0:
20             raise ValueError
21         else:
22             self.mass_part = mass_part
23         if surface_part < 0:
24             raise ValueError
25         else:
26             self.surface_part = surface_part
27         if drag_coefficient_part < 0:
28             raise ValueError
29         else:
30             self.drag_coefficient_part = drag_coefficient_part
31
32     def get_mass(self):
33         """
34             :return: Mass of the part (Not Negative Double) [kg]
35         """
36         if self.mass_part < 0:
37             raise ValueError
38         return self.mass_part
39
40     def get_surface(self):
41         """
42             :return: Surface area of the part (Not Negative Double) [m^2]
43         """
44         if self.surface_part < 0:
45             raise ValueError
46         return self.surface_part
47
48     def get_drag_coefficient(self):
49         """
50             :return: Drag coefficient of the part (Not Negative Double) [1]
51         """
52         if self.drag_coefficient_part < 0:
53             raise ValueError
54         return self.drag_coefficient_part
55
56     def set_mass(self, new_mass_part):
57         """
58             Changes the mass of the object and validates the value
59             :param new_mass_part: New Mass of the part (Not negative Double) [kg]
60         """
61         if new_mass_part < 0:
62             raise ValueError
63         self.mass_part = new_mass_part
64
65     def get_thrust(self):
66         """
67             The thrust of a non tank or engine part is always zero
68             :return: Returns 0.0
69         """
70         return 0.0
```

Anhang E: RocketParts.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/RocketParts.py

Seite 2 von 3

```
71
72
73 class Tank(RocketPart):
74     """
75     Class to describe rocket tanks with connected engine
76     """
77
78     def __init__(self, mass_part, surface_part, drag_coefficient_part, mass_propellant,
79                  mass_change_tank,
80                  velocity_exhaust_tank, surface_nozzle, pressure_nozzle):
81         """
82         Setup for a tank with value validation
83         :param mass_part: Mass of the tank (without propellant!) (Double) [kg]
84         :param surface_part: Surface of the tank (Double) [m^2]
85         :param drag_coefficient_part: Drag coefficient of the tank (Double) [1]
86         :param mass_propellant: Mass of the propellant stored in the tank (Double) [kg]
87         :param mass_change_tank: Maximum mass flow of the propellant (Double) [kg/s]
88         :param velocity_exhaust_tank: Velocity of the propellant at the nozzle [m/s]
89         :param pressure_nozzle: Static pressure around the nozzle [Pa]
90         """
91         RocketPart.__init__(self, mass_part, surface_part, drag_coefficient_part)
92         if mass_propellant < 0:
93             raise ValueError
94         else:
95             self.mass_propellant = mass_propellant
96         if mass_change_tank < 0:
97             raise ValueError
98         else:
99             self.mass_change_tank = mass_change_tank
100        if velocity_exhaust_tank < 0:
101            raise ValueError
102        else:
103            self.velocity_exhaust_tank = velocity_exhaust_tank
104        if surface_nozzle < 0:
105            raise ValueError
106        else:
107            self.surface_nozzle = surface_nozzle
108        if pressure_nozzle < 0:
109            raise ValueError
110        else:
111            self.pressure_nozzle = pressure_nozzle
112        self.thrust_level_tank = 1.0
113
114    def get_thrust_level(self):
115        """
116        :return: Thrust level of the engine, scales the mass of propellant flow (Double) [%]
117        """
118        if self.thrust_level_tank < 0 or self.thrust_level_tank > 1:
119            raise ValueError
120        return self.thrust_level_tank
121
122    def set_thrust_level(self, new_thrust_level_tank):
123        """
124        Allows to change the thrust level of the engine
125        :param new_thrust_level_tank: New thrust level of the engine
126        """
127        if new_thrust_level_tank < 0 or new_thrust_level_tank > 1:
128            raise ValueError
129        else:
130            self.thrust_level_tank = new_thrust_level_tank
131
132    def get_velocity_exhaust(self):
133        """
134        :return: Velocity of the propellant exhaust at the nozzle (Double) [m/s]
135        """
136        if self.velocity_exhaust_tank < 0:
137            raise ValueError
138        else:
139            return self.velocity_exhaust_tank
140
141    def get_mass(self):
142        """
```

Anhang E: RocketParts.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/RocketParts.py

Seite 3 von 3

```
142     Calculates the mass of the part from the sum of propellant and part mass
143     :return: Mass of the part filled with propellant (Double) [kg]
144     """
145     mass_fueled = self.mass_propellant + self.mass_part
146     if mass_fueled < 0:
147         raise ValueError
148     else:
149         return mass_fueled
150
151     def set_mass_propellant(self, new_mass_propellant):
152         """
153             Allows to change the mass of the propellant
154             :param new_mass_propellant: New mass of the propellant (Double)
155             """
156             if new_mass_propellant < 0:
157                 raise ValueError
158             else:
159                 self.mass_propellant = new_mass_propellant
160
161     def get_mass_change(self):
162         """
163             Returns the mass change of the tank
164             :return: Mass change of the tank (Double) [kg/s]
165             """
166             new_mass_change = self.mass_change_tank * self.thrust_level_tank
167             if new_mass_change < 0:
168                 raise ValueError
169             else:
170                 return new_mass_change
171
172     def get_surface_nozzle(self):
173         """
174             :return: Surface of the engine nozzle (Double) [m^2]
175             """
176             if self.surface_nozzle < 0:
177                 raise ValueError
178             else:
179                 return self.surface_nozzle
180
181     def get_pressure_nozzle(self):
182         """
183             :return: Static pressure at the nozzle (Double) [Pa]
184             """
185             if self.pressure_nozzle < 0:
186                 raise ValueError
187             else:
188                 return self.pressure_nozzle
189
190     def get_mass_propellant(self):
191         """
192             :return: Mass of the propellant contained in the tank (Double) [kg]
193             """
194             return self.mass_propellant
```

Anhang F: Rockets.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/Rockets.py

Seite 1 von 3

```
1 """
2 Author:      Felix Bräunling
3 Description: This class describes the assembled rocket with its position and physical
4 attributes
5 """
6 import math
7 from RocketParts import RocketPart
8 from RocketParts import Tank
9
10
11 class Rocket(object):
12
13     def __init__(self, pos, velocity, acceleration):
14         """
15             Set up of a rocket
16             :param pos: Position of the rocket as x and y coordinates (Tuple -> (Double, Double))
17             :param velocity: Velocity of the rocket in x and y direction (Tuple -> (Double, Double))
18             :param acceleration: Acceleration of the rocket in x and y direction (Tuple -> (Double,
19 Double))
20         """
21         self.rocket_parts = []
22         self.pos = pos
23         self.velocity = velocity
24         self.acceleration = acceleration
25         self.mass = 0.0
26         self.surface = 0.0
27         self.angle = 0
28
29     def get_pos(self):
30         """
31             :return: The position of the rocket as x and y coordinates (Tuple -> (Double, Double))
32             ([m], [m])
33         """
34         return self.pos
35
36     def get_velocity(self):
37         """
38             :return: The velocity of the rocket in x and y direction (Tuple -> (Double, Double)) ([m/
39 s], [m/s])
40         """
41         return self.velocity
42
43     def get_acceleration(self):
44         """
45             :return: The acceleration of the rocket in x and y direction (Tuple -> (Double, Double))
46             ([m/s^2], [m/s^2])
47         """
48         return self.acceleration
49
50     def set_pos(self, pos_x, pos_y):
51         """
52             Allows to set the position of the rocket as x and y coordinates
53             :param pos_x: X-position of the rocket (Double)
54             :param pos_y: Y-position of the rocket (Double)
55         """
56         self.pos = [pos_x, pos_y]
57
58     def set_velocity(self, velocity_x, velocity_y):
59         """
60             Allows to set the velocity of the rocket in x and y direction
61             :param velocity_x: Velocity in X direction of the rocket (Double)
62             :param velocity_y: Velocity in Y direction of the rocket (Double)
63         """
64         self.velocity = [velocity_x, velocity_y]
65
66     def set_acceleration(self, acceleration_x, acceleration_y):
67         """
68             Allows to set the acceleration of the rocket in x and y direction
69             :param acceleration_x: Acceleration in X direction of the rocket (Double)
70             :param acceleration_y: Acceleration in Y direction of the rocket (Double)
71         """
72
```

Anhang F: Rockets.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/Rockets.py

Seite 2 von 3

```
68         self.acceleration = [acceleration_x, acceleration_y]
69
70     def get_mass(self):
71         """
72             :return: Mass of the rocket (Double) [kg]
73         """
74         if self.mass < 0:
75             raise ValueError
76         else:
77             return self.mass
78
79     def append_part(self, new_part):
80         """
81             Adds a new part to the end of the Rocket, first part always has to be a RocketPart!
82             :param new_part: Part to be added at the end of the rocket (RocketPart or Subclass)
83         """
84         if not issubclass(type(new_part), RocketPart):
85             print "Part has to be of type RocketPart or Subclass"
86             raise ValueError
87         if self.rocket_parts == [] and type(new_part) != RocketPart:
88             print "First part has to be of type RocketPart (e.g. nose of the rocket)"
89             raise ValueError
90         self.rocket_parts.append(new_part)
91
92     def set_mass(self):
93         """
94             Calculates the mass of the rocket depending on the mass of the parts and changes the
95             attribute mass accordingly
96         """
97         mass_sum = 0.0
98         for part in self.rocket_parts:
99             mass_sum += part.get_mass()
100        self.mass = mass_sum
101
102    def get_surface(self):
103        """
104            :return: Surface area of the rocket (Double) [m^2]
105        """
106        if self.surface < 0:
107            raise ValueError
108        else:
109            return self.surface
110
111    def set_surface(self):
112        """
113            Calculates the surface of the rocket depending on the surface of the parts
114            and changes the attribute surface accordingly
115        """
116        surface_sum = 0.0
117        for part in self.rocket_parts:
118            surface_sum += part.get_surface()
119        self.surface = surface_sum
120
121    def get_angle(self):
122        """
123            Calculates the angle of the rocket to the horizon
124            :return Angle of the rocket (Double) [grad]
125        """
126        return self.angle
127
128    def set_angle(self, new_angle):
129        """
130            Allows to set the angle of the rocket to the horizon
131        """
132        self.angle = new_angle
133
134    def decouple(self):
135        """
136            Decouples the last added part in Rocket.rocket_parts
137        """
138        if len(self.rocket_parts) == 1:
139            pass
```

Anhang F: Rockets.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/Rockets.py

Seite 3 von 3

```
139         else:
140             self.rocket_parts.pop()
141
142     def get_current_stage(self):
143         """
144             Returns the current active stage
145             :return: Current active stage (Rocket Part or Tank)
146         """
147         return self.rocket_parts[-1]
```

Anhang G: RocketSim.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/RocketSim.py

Seite 1 von 6

```
1 """
2 Author: Felix Bräunling
3
4 Description: This file contains the Flight class, that describes a setup flight and also contains
5 """
6 import Formula
7 import os
8 from Atmospheres import Atmosphere
9 from Datas import Data
10 from Layers import Layer
11 from Planets import Planet
12 from RocketParts import RocketPart, Tank
13 from Rockets import Rocket
14
15
16 class Flight(object):
17     def __init__(self, time_delta, planet, rocket, atmosphere, data):
18         """
19             Setup of a flight with it's parameters
20             :param time_delta: Size of the time step the simulation use
21             :param planet: Planet the flight will take place
22             :param rocket: Rocket the flight will use
23             :param atmosphere: Atmosphere of the planet
24             :param data: Data object gained values will be saved in.
25             :return:
26         """
27         self.time_delta = time_delta
28         self.planet = planet
29         self.rocket = rocket
30         self.atmosphere = atmosphere
31         self.data = data
32         self.distance = Formula.vector_addition(self.rocket.get_pos())
33
34     def __lt__(self, other):
35         max_self = max(self.data.pos_rocket)
36         max_other = max(other.data.pos_rocket)
37         return Formula.vector_addition(max_self) < Formula.vector_addition(max_other)
38
39     def get_max_height(self):
40         return max(self.data.pos_x_rocket) - self.planet.get_radius()
41
42     def simulate(self):
43         """
44             Runs the simulation and starts the calculations
45         """
46         # Save time
47         time_now = self.data.time[-1] + self.time_delta
48         if time_now > 140.0:
49             pass
50         self.data.time.append(time_now)
51         # Get the current height the rocket is at
52         height_now = Formula.vector_addition(self.rocket.get_pos()) - self.planet.get_radius()
53         descending = False
54         if height_now < self.data.height_rocket[-1]:
55             descending = True
56         distance_now = Formula.vector_addition(self.rocket.get_pos())
57         self.data.height_rocket.append(height_now) # Save current height
58         # Get the temperature at the height of the rocket
59         temp_low_now = self.atmosphere.get_layer(height_now).get_temp_low()
60         temp_gradient_now = self.atmosphere.get_layer(height_now).get_temp_gradient()
61         index_layer = self.atmosphere.get_layers().index(self.atmosphere.get_layer(height_now))
62         height_low_now = self.atmosphere.calc_height_below(index_layer)
63         temperature_now = Formula.temperature(temp_low_now, temp_gradient_now, height_low_now,
64         height_now)
65         self.data.temperature.append(temperature_now) # Save current temperature
66         # Get the pressure at the height of the rocket
67         pressure_low_now = self.atmosphere.get_layer(height_now).get_pressure_low()
68         pressure_now = Formula.pressure(pressure_low_now, temp_gradient_now, height_low_now,
69         height_low_now, temp_low_now)
70         self.data.pressure.append(pressure_now) # Save current pressure
71         # Get the current force of gravity the rocket is experiencing
```

Anhang G: RocketSim.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/RocketSim.py

Seite 2 von 6

```

71     gravity_now = Formula.gravity(self.planet.get_mass(), self.rocket.get_mass(),
72         distance_now)
73     self.data.gravity.append(gravity_now)
74     # Get the thrust the rocket is producing
75     current_stage = self.rocket.get_current_stage()
76     change_prop_mass = False
77     if type(current_stage) == RocketPart:
78         thrust_now = current_stage.get_thrust()
79     else:
80         thrust_now = Formula.thrust(current_stage.get_mass_change(),
81             current_stage.get_velocity_exhaust(),
82             current_stage.get_surface_nozzle(),
83             current_stage.get_pressure_nozzle(),
84             pressure_now)
85         change_prop_mass = True
86     self.data.thrust.append(thrust_now)
87     # Get the drag the rocket is experiencing
88     density_now = Formula.density(pressure_now, temperature_now)
89     self.data.density.append(density_now) # Save density
90     velocity_before = Formula.vector_addition(self.rocket.get_velocity())
91     drag_coefficient = self.rocket.rocket_parts[0].drag_coefficient_part
92     if descending:
93         rocket_desc = -1
94     else:
95         rocket_desc = 1
96     drag_now = rocket_desc * Formula.drag(density_now, velocity_before,
97         self.rocket.get_surface(), drag_coefficient)
98     self.data.drag.append(abs(drag_now)) # Save current drag
99     # Get forces split in x and y direction
100    angle_rocket_now = self.rocket.get_angle()
101    self.data.angle_rocket.append(angle_rocket_now) # Save current angle of the rocket
102    thrust_now_x = Formula.res_x(thrust_now, angle_rocket_now)
103    thrust_now_y = Formula.res_y(thrust_now, angle_rocket_now)
104    gravity_now_x = Formula.res_x(gravity_now, angle_rocket_now)
105    gravity_now_y = Formula.res_y(gravity_now, angle_rocket_now)
106    drag_now_x = Formula.res_x(drag_now, angle_rocket_now)
107    drag_now_y = Formula.res_y(drag_now, angle_rocket_now)
108    # Get current resulting force and resulting forces in x and y direction
109    force_res_now_x = Formula.resulting_force(thrust_now_x, gravity_now_x, drag_now_x)
110    force_res_now_y = Formula.resulting_force(thrust_now_y, gravity_now_y, drag_now_y)
111    force_res_now = Formula.vector_addition([force_res_now_x, force_res_now_y])
112    self.data.force_res.append(force_res_now_x) # Save current resulting force
113    self.data.force_res_split.append([force_res_now_x, force_res_now_y]) # Save split
114    resulting force
115    # Get current acceleration
116    acceleration_x_now = Formula.acceleration(force_res_now_x, self.rocket.get_mass())
117    acceleration_y_now = Formula.acceleration(force_res_now_y, self.rocket.get_mass())
118    acceleration_now = Formula.vector_addition([acceleration_x_now, acceleration_y_now])
119    self.data.acceleration_rocket.append(acceleration_x_now) # Save current acceleration
120    self.rocket.set_acceleration(acceleration_x_now, acceleration_y_now)
121    # Change mass of the rocket
122    if change_prop_mass:
123        mass_propellant_now = current_stage.get_mass_propellant() -
124            current_stage.get_mass_change()*self.time_delta
125        if mass_propellant_now < 0:
126            self.rocket.decouple()
127        else:
128            current_stage.set_mass_propellant(mass_propellant_now)
129            self.rocket.set_mass()
130            self.data.mass_rocket.append(self.rocket.get_mass())
131            # Get current velocity
132            velocity_x_now = Formula.velocity(self.rocket.get_velocity()[0], self.time_delta,
133                acceleration_x_now)
134            velocity_y_now = Formula.velocity(self.rocket.get_velocity()[1], self.time_delta,
135                acceleration_y_now)
136            velocity_now = Formula.vector_addition([velocity_x_now, velocity_y_now])
137            self.rocket.set_velocity(velocity_x_now, velocity_y_now)
138            self.data.velocity_rocket.append(velocity_x_now) # Save velocity of the rocket
139            # Get the current position
140            way_traveled_x_now = Formula.way(self.time_delta, velocity_x_now, acceleration_x_now)
141            pos_x_now = Formula.position(self.rocket.get_pos()[0], way_traveled_x_now)
142            way_traveled_y_now = Formula.way(self.time_delta, velocity_y_now, acceleration_y_now)
143

```

Anhang G: RocketSim.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/RocketSim.py

Seite 3 von 6

```

135     pos_y_now = Formula.position(self.rocket.get_pos())[1], way_traveled_y_now)
136     self.data.pos_x_rocket.append(pos_x_now) # Save x position of the rocket
137     self.data.pos_y_rocket.append(pos_y_now) # Save y position of the rocket
138     self.rocket.set_pos(pos_x_now, pos_y_now)
139     # Get distance to planet core
140     self.distance = Formula.vector_addition([pos_x_now, pos_y_now])
141     print height_now
142
143     def get_distance(self):
144         """
145             :return: Distance of the rocket to the planets core [m]
146         """
147         return self.distance
148
149
150     def run_sim(sim_flight, flight_name, max_step, sim_time_step):
151         """Runs the simulation for one flight, with all setup and tear down operations
152         :param flight: Flight object, that will be simulated (Flight)
153         :param flight_name: Name of the flight (String)
154         :param max_step: Maximum iterations in on simulation (Integer)
155         :param sim_time_step: Time delta for each simulation step (Integer)
156         """
157         current_step = 0
158         sim_flight.data.pos_x_rocket.append(sim_flight.rocket.get_pos()[0])
159         sim_flight.data.pos_y_rocket.append(sim_flight.rocket.get_pos()[1])
160         sim_flight.data.velocity_rocket.append(sim_flight.rocket.get_velocity()[0])
161         sim_flight.data.acceleration_rocket.append(sim_flight.rocket.get_acceleration()[0])
162         sim_flight.data.mass_rocket.append(sim_flight.rocket.get_mass())
163         sim_flight.data.angle_rocket.append(sim_flight.rocket.get_angle())
164         sim_flight.data.gravity.append(Formula.gravity(sim_flight.planet.get_mass(),
165             sim_flight.rocket.get_mass(), sim_flight.planet.radius_planet))
166         sim_flight.data.force_res.append(-1*sim_flight.data.gravity[0])
167         sim_flight.data.temperature.append(sim_flight.atmosphere.get_layer(0.0).get_temp_low())
168         sim_flight.data.pressure.append(sim_flight.atmosphere.get_layer(0.0).get_pressure_low())
169         sim_flight.data.density.append(sim_flight.atmosphere.get_layer(0).get_density_now(0.0, 0.0))
170         print "Simulation starting"
171         while sim_flight.get_distance() >= sim_flight.planet.get_radius() and current_step <=
172             max_step:
173                 sim_flight.simulate()
174                 current_step += 1
175                 #Setting end point data
176                 sim_flight.data.time.append(sim_flight.data.time[-1]+sim_time_step)
177                 sim_flight.data.height_rocket.append(0.0)
178                 sim_flight.data.pos_x_rocket.append(sim_flight.planet.get_radius())
179                 sim_flight.data.pos_y_rocket.append(0.0)
180                 sim_flight.data.velocity_rocket.append(0.0)
181                 sim_flight.data.acceleration_rocket.append(0.0)
182                 sim_flight.data.mass_rocket.append(sim_flight.rocket.rocket_parts[-1].get_mass())
183                 sim_flight.data.angle_rocket.append(0.0)
184                 sim_flight.data.thrust.append(0.0)
185                 sim_flight.data.drag.append(0.0)
186                 sim_flight.data.gravity.append(sim_flight.data.gravity[0])
187                 sim_flight.data.force_res.append(sim_flight.data.gravity[0])
188                 sim_flight.data.force_res_split.append([sim_flight.data.gravity[0], 0.0])
189                 sim_flight.data.temperature.append(sim_flight.data.temperature[0])
190                 sim_flight.data.pressure.append(sim_flight.data.pressure[0])
191                 sim_flight.data.density.append(sim_flight.data.density[0])
192                 print "Simulation ended! Last Step: "+str(current_step)
193                 if not os.path.exists("./Results_2/{}".format(flight_name)):
194                     os.makedirs("./Results_2/{}".format(flight_name))
195                 sim_flight.data.write_csv()
196                 print "Saved to CSV! In Folder {}".format(flight_name)
197
198     def main():
199         """
200             Declare your planet, atmosphere with its layers, the rocket with its parts, the data object
201             and the time delta here
202             and run the simulation
203             :return:
204         """
205             # Setup here

```

Anhang G: RocketSim.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/RocketSim.py

Seite 4 von 6

```
204     sim_max_step = 100000 # Maximum time steps the simulation should run
205     sim_time_step = 0.1 # [s] Timestep the simulation uses
206     earth = Planet(pos_planet=[0.0, 0.0], mass_planet=5.974e+24, radius_planet=12756.32/2.0*1000)
207     earth_radius = earth.get_radius()
208     troposphere = Layer(pressure_low=101325.0, width_layer=18000.0, temp_gradient=-0.0065,
209                           temp_low=288.15)
210     stratosphere = Layer(pressure_low=16901.37, width_layer=32000.0, temp_gradient=0.0031875,
211                           temp_low=171.15)
212     mesosphere = Layer(pressure_low=1.02, width_layer=30000.0, temp_gradient=-0.003333,
213                           temp_low=273.15)
214     thermosphere = Layer(pressure_low=0.04, width_layer=420000.0, temp_gradient=-0.000405,
215                           temp_low=173.15)
216     exosphere = Layer(pressure_low=0.0, width_layer=999999999.0, temp_gradient=0.0,
217                           temp_low=3.0)
218     earth_atmosphere = Atmosphere()
219     earth_atmosphere.add_layer(troposphere)
220     earth_atmosphere.add_layer(stratosphere)
221     earth_atmosphere.add_layer(mesosphere)
222     earth_atmosphere.add_layer(thermosphere)
223     earth_atmosphere.add_layer(exosphere)
224     # Set maximum propellant mass
225     A150_mass_propellant = 484.8076
226     # Calculate A150 one tank:
227     sim_flight_name = "A150_OneTank"
228     sim_data = Data(data_file=".~/Results_2/{}~/Results_Data.csv".format(sim_flight_name))
229     sim_data.name = "Einstufig"
230     A150_payload = 0.0 # kg moegliche Nutzlast
231     A150_nose_cone = RocketPart(mass_part=7.0+2.31336+A150_payload, surface_part=0.1140,
232                                   drag_coefficient_part=0.27)
233     A150_liquid_tank = Tank(mass_part=116.1216, surface_part=0.0, drag_coefficient_part=0.0,
234                             mass_propellant=A150_mass_propellant, mass_change_tank=9.394,
235                             velocity_exhaust_tank=1417.32, surface_nozzle=0.0275,
236                             pressure_nozzle=101325.0)
237     A150 = Rocket(pos=[earth_radius, 0.0], velocity=[0.0, 0.0], acceleration=[0.0, 0.0])
238     A150_booster = Tank(mass_part=28.1232, surface_part=0.0, drag_coefficient_part=0.0,
239                          mass_propellant=117.6074, mass_change_tank=47.1733,
240                          velocity_exhaust_tank=1747.6074, surface_nozzle=0.0434,
241                          pressure_nozzle=101325.0)
242     A150.append_part(A150_nose_cone)
243     A150.append_part(A150_liquid_tank)
244     A150.append_part(A150_booster)
245     A150.set_mass()
246     A150.set_surface()
247     sim_flight = Flight(sim_time_step, earth, A150, earth_atmosphere, sim_data)
248     # Running the simulation
249     run_sim(sim_flight, sim_flight_name, sim_max_step, sim_time_step)
250
251     # Calculate A150 two tanks 1
252     sim_flight_name = "A150_RefTanka"
253     sim_data = Data(data_file=".~/Results_2/{}~/Results_Data.csv".format(sim_flight_name))
254     sim_data.name = "Stufe 1 leer"
255     A150_payload = 0.0 # kg moegliche Nutzlast
256     A150_nose_cone = RocketPart(mass_part=7.0+2.31336+A150_payload, surface_part=0.1140,
257                                   drag_coefficient_part=0.27)
258     A150_liquid_tank_one = Tank(mass_part=116.1216, surface_part=0.0, drag_coefficient_part=0.0,
259                                 mass_propellant=A150_mass_propellant, mass_change_tank=9.394,
260                                 velocity_exhaust_tank=1417.32, surface_nozzle=0.0275,
261                                 pressure_nozzle=101325.0)
262     A150_liquid_tank_two = Tank(mass_part=116.1216, surface_part=0.0, drag_coefficient_part=0.0,
263                                 mass_propellant=0, mass_change_tank=9.394,
264                                 velocity_exhaust_tank=1417.32, surface_nozzle=0.0275,
265                                 pressure_nozzle=101325.0)
266     A150 = Rocket(pos=[earth_radius, 0.0], velocity=[0.0, 0.0], acceleration=[0.0, 0.0])
267     A150_booster = Tank(mass_part=28.1232, surface_part=0.0, drag_coefficient_part=0.0,
268                          mass_propellant=117.6074, mass_change_tank=47.1733,
269                          velocity_exhaust_tank=1747.6074, surface_nozzle=0.0434,
270                          pressure_nozzle=101325.0)
271     A150.append_part(A150_nose_cone)
272     A150.append_part(A150_liquid_tank_one)
273     A150.append_part(A150_liquid_tank_two)
274     A150.append_part(A150_booster)
275     A150.set_mass()
```

Anhang G: RocketSim.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/RocketSim.py

Seite 5 von 6

```
259     A150.set_surface()
260     sim_flight_a = Flight(sim_time_step, earth, A150, earth_atmosphere, sim_data)
261     # Running the simulation
262     run_sim(sim_flight_a, sim_flight_name, sim_max_step, sim_time_step)
263
264     # Calculate A150 two tanks 1
265     sim_flight_name = "A150_RefTankB"
266     sim_data = Data(data_file=".//Results_2//{}//Results_Data.csv".format(sim_flight_name))
267     sim_data.name = "Stufe 2 leer"
268     A150_payload = 0.0 # kg moegliche Nutzlast
269     A150_nose_cone = RocketPart(mass_part=7.0+2.31336+A150_payload, surface_part=0.1140,
270                                   drag_coefficient_part=0.27)
271     A150_liquid_tank_one = Tank(mass_part=116.1216, surface_part=0.0, drag_coefficient_part=0.0,
272                                   mass_propellant=0, mass_change_tank=9.394,
273                                   velocity_exhaust_tank=1417.32, surface_nozzle=0.0275,
274                                   pressure_nozzle=101325.0)
275     A150_liquid_tank_two = Tank(mass_part=116.1216, surface_part=0.0, drag_coefficient_part=0.0,
276                                   mass_propellant=A150_mass_propellant, mass_change_tank=9.394,
277                                   velocity_exhaust_tank=1417.32, surface_nozzle=0.0275,
278                                   pressure_nozzle=101325.0)
279     A150 = Rocket(pos=[earth_radius, 0.0], velocity=[0.0, 0.0], acceleration=[0.0, 0.0])
280     A150_booster = Tank(mass_part=28.1232, surface_part=0.0, drag_coefficient_part=0.0,
281                           mass_propellant=117.6074, mass_change_tank=47.1733,
282                           velocity_exhaust_tank=1747.6074, surface_nozzle=0.0434,
283                           pressure_nozzle=101325.0)
284     A150.append_part(A150_nose_cone)
285     A150.append_part(A150_liquid_tank_one)
286     A150.append_part(A150_liquid_tank_two)
287     A150.append_part(A150_booster)
288     A150.set_mass()
289     A150.set_surface()
290     sim_flight_b = Flight(sim_time_step, earth, A150, earth_atmosphere, sim_data)
291
292     # Running the simulation
293     run_sim(sim_flight_b, sim_flight_name, sim_max_step, sim_time_step)
294
295     # Searching for optimal tank setup
296     divisor = 2
297     count = 0
298     sim_opt_max_step = 40
299
300     propellant_step = A150_mass_propellant/sim_opt_max_step
301     while count < sim_opt_max_step:
302         sim_flight_name = "A150_TankSetup_{0:03d}".format(count)
303         sim_data = Data(data_file=".//Results_2//{}//Results_Data.csv".format(sim_flight_name))
304         A150_mass_propellant_tank_one = propellant_step*count
305         A150_mass_propellant_tank_two = A150_mass_propellant - propellant_step*count
306         sim_data.name = "Stufe 1: {0:.2f} kg Stufe 2: {1:.2f} kg".format
307             (A150_mass_propellant_tank_two, A150_mass_propellant_tank_one)
308         A150 = Rocket(pos=[earth_radius, 0.0], velocity=[0.0, 0.0], acceleration=[0.0, 0.0])
309         A150_liquid_tank_one_split = Tank(mass_part=116.1216, surface_part=0.0,
310                                           drag_coefficient_part=0.0,
311                                           mass_propellant=A150_mass_propellant_tank_one,
312                                           mass_change_tank=9.394,
313                                           velocity_exhaust_tank=1417.32, surface_nozzle=0.0275,
314                                           pressure_nozzle=101325.0)
315         A150_liquid_tank_two_split = Tank(mass_part=116.1216, surface_part=0.0,
316                                           drag_coefficient_part=0.0,
317                                           mass_propellant=A150_mass_propellant_tank_two,
318                                           mass_change_tank=9.394,
319                                           velocity_exhaust_tank=1417.32, surface_nozzle=0.0275,
320                                           pressure_nozzle=101325.0)
321         A150_booster = Tank(mass_part=28.1232, surface_part=0.0, drag_coefficient_part=0.0,
322                           mass_propellant=117.6074, mass_change_tank=47.1733,
323                           velocity_exhaust_tank=1747.6074, surface_nozzle=0.0434,
324                           pressure_nozzle=101325.0)
325         A150.append_part(A150_nose_cone)
326         A150.append_part(A150_liquid_tank_one_split)
327         A150.append_part(A150_liquid_tank_two_split)
328         A150.append_part(A150_booster)
329         A150.set_mass()
330         A150.set_surface()
```

Anhang G: RocketSim.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/RocketSim.py

Seite 6 von 6

```
315     sim_flight_p = Flight(sim_time_step, earth, A150, earth_atmosphere, sim_data)
316     # Running the simulation
317     run_sim(sim_flight_p, sim_flight_name, sim_max_step, sim_time_step)
318
319     count += 1
320
321
322     # Calculate A150 one tank, no booster:
323     sim_flight_name = "A150_NoBooster"
324     sim_data = Data(data_file=".//Results_2/{}/Results_Data.csv".format(sim_flight_name))
325     sim_data.name = "Einstufig ohne Booster"
326     A150_payload = 0.0 # kg moegliche Nutzlast
327     A150_nose_cone = RocketPart(mass_part=7.0+2.31336+A150_payload, surface_part=0.1140,
328                                   drag_coefficient_part=0.27)
328     A150_liquid_tank = Tank(mass_part=116.1216, surface_part=0.0, drag_coefficient_part=0.0,
329                               mass_propellant=A150_mass_propellant, mass_change_tank=9.394,
329                               velocity_exhaust_tank=1417.32, surface_nozzle=0.0275,
329                               pressure_nozzle=101325.0)
330     A150 = Rocket(pos=[earth_radius, 0.0], velocity=[0.0, 0.0], acceleration=[0.0, 0.0])
331     A150.append_part(A150_nose_cone)
332     A150.append_part(A150_liquid_tank)
333     A150.set_mass()
334     A150.set_surface()
335     sim_flight_nb = Flight(sim_time_step, earth, A150, earth_atmosphere, sim_data)
336     # Running the simulation
337     run_sim(sim_flight_nb, sim_flight_name, sim_max_step, sim_time_step)
338
339     print "Optimization has ended after {} iterations, Results available".format(count-1)
340
341
342
343 if __name__ == "__main__":
344     main()
```

Anhang H: Datas.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/Datas.py

Seite 1 von 2

```
1 """
2 Author:      Felix Bräunling
3 Description: This class is used to store any values generated during the calculations and
4           saves
5           them for later plotting
6 """
7 import csv
8
9
10 class Data(object):
11
12     def __init__(self, data_file):
13         self.data_file = data_file
14         self.name = ""
15         self.time = [0] # [s]
16         self.pos_x_rocket = [] # [m]
17         self.pos_y_rocket = [] # [m]
18         self.velocity_rocket = [] # [m/s]
19         self.acceleration_rocket = [] # [m/s^2]
20         self.mass_rocket = [] # [kg]
21         self.height_rocket = [0] # [m]
22         self.angle_rocket = [] # [grad]
23         self.thrust = [0] # [N]
24         self.drag = [0] # [N]
25         self.gravity = [] # [N]
26         self.force_res = [] # [N]
27         self.force_res_split = [] # [(N, N)] for x and y direction
28         self.temperature = [] # [K]
29         self.pressure = [] # [Pa]
30         self.density = [] # [kg/m^3]
31
32     def add_data(self, time, pos_x_rocket, pos_y_rocket, velocity_rocket, acceleration_rocket,
33                 mass_rocket, height_rocket,
34                 angle_rocket, thrust, drag, gravity, force_res, force_res_split, temperature,
35                 pressure, density):
36         """
37             Appends new data values to their list
38             :param time: Time the simulation ran (Double)
39             :param pos_x_rocket: Position of the rocket as coordinates (Double)
40             :param pos_y_rocket: Position of the rocket as coordinates (Double)
41             :param velocity_rocket: Velocity of the rocket (Double)
42             :param acceleration_rocket: Acceleration of the rocket (Double)
43             :param mass_rocket: Mass of the rocket (Double)
44             :param height_rocket: Height of the above the planet surface (Double)
45             :param angle_rocket: Angle of the rocket to the horizon (Double)
46             :param thrust: Thrust of the rocket (Double)
47             :param drag: Drag the rocket is experiencing (Double)
48             :param gravity: Gravity between the rocket and the planet (Double)
49             :param force_res: Resulting force affecting the rocket (Double)
50             :param force_res_split: Resulting force split in x and y direction (Tuple -> (Double,
51                                         Double))
51             :param temperature: Temperature of the ambient (Double)
52             :param pressure: Pressure of the ambient (Double)
53             :param density: Density of the medium the rocket is in (Double)
54         """
55         self.time.append(time)
56         self.pos_x_rocket.append(pos_x_rocket)
57         self.pos_y_rocket.append(pos_y_rocket)
58         self.velocity_rocket.append(velocity_rocket)
59         self.acceleration_rocket.append(acceleration_rocket)
60         self.mass_rocket.append(mass_rocket)
61         self.height_rocket.append(height_rocket)
62         self.angle_rocket.append(angle_rocket)
63         self.thrust.append(thrust)
64         self.drag.append(drag)
65         self.gravity.append(gravity)
66         self.force_res.append(force_res)
67         self.force_res_split.append(force_res_split)
68         self.temperature.append(temperature)
69         self.pressure.append(pressure)
70         self.density.append(density)
```

Anhang H: Datas.py

Datei: /home/felix/Projekte/Studienarbeit/RocketSim/Datas.py

Seite 2 von 2

```
69      def write_csv(self):
70          """
71              Writes the saved data in a csv file
72          """
73          with open(self.data_file, 'wb') as csv_file:
74              data_writer = csv.writer(csv_file, delimiter=',')
75              data_writer.writerow(self.time)
76              data_writer.writerow(self.pos_x_rocket)
77              data_writer.writerow(self.pos_y_rocket)
78              data_writer.writerow(self.velocity_rocket)
79              data_writer.writerow(self.acceleration_rocket)
80              data_writer.writerow(self.mass_rocket)
81              data_writer.writerow(self.heighth_rocket)
82              data_writer.writerow(self.angle_rocket)
83              data_writer.writerow(self.thrust)
84              data_writer.writerow(self.drag)
85              data_writer.writerow(self.gravity)
86              data_writer.writerow(self.force_res)
87              data_writer.writerow(self.force_res_split)
88              data_writer.writerow(self.temperature)
89              data_writer.writerow(self.pressure)
90              data_writer.writerow(self.density)
91              data_writer.writerow(self.name)
92
93      def read_csv(self):
94          """
95              Reads saved data from an csv-file
96          """
97          with open(self.data_file, 'rb') as csv_file:
98              data_reader = csv.reader(csv_file, delimiter=',')
99              new_data = []
100             for row in data_reader:
101                 new_data.append(row)
102             self.time = map(float, new_data[0])
103             self.pos_x_rocket = map(float, new_data[1])
104             self.pos_y_rocket = map(float, new_data[2])
105             self.velocity_rocket = map(float, new_data[3])
106             self.acceleration_rocket = map(float, new_data[4])
107             self.mass_rocket = map(float, new_data[5])
108             self.heighth_rocket = map(float, new_data[6])
109             self.angle_rocket = map(float, new_data[7])
110             self.thrust = map(float, new_data[8])
111             self.drag = map(float, new_data[9])
112             self.gravity = map(float, new_data[10])
113             self.force_res = map(float, new_data[11])
114             self.force_res_split = new_data[12]
115             self.temperature = map(float, new_data[13])
116             self.pressure = map(float, new_data[14])
117             self.density = map(float, new_data[15])
118             temp_name = ""
119             for char in new_data[16]:
120                 temp_name += char
121             self.name = temp_name
```

Anhang I: Simulationswerte

Bezeichnung	Formelzeichen	SI-Einheiten	Angloamerikanisch ¹
Erdradius ²	r_{Planet}	6.371 m	-
Erdmasse ²	m_{Planet}	$5,974 \cdot 10^{24}$ kg	-
Troposphäre²			
Unterer Druck	p_{h_0}	101.325 Pa	-
Schichtbreite	-	18.000 m	-
Temperaturgradient	a	-0,0065 K/m	-
Untere Temperatur	T_{h_0}	288,15 K	-
Stratosphäre²			
Unterer Druck	p_{h_0}	6.700 Pa	-
Schichtbreite	-	32.000 m	-
Temperaturgradient	a	0,0031875 K/m	-
Untere Temperatur	T_{h_0}	171,15 K	-
Mesosphäre²			
Unterer Druck	p_{h_0}	2.000 Pa	-
Schichtbreite	-	30.000 m	-
Temperaturgradient	a	-0,003333 K/m	-
Untere Temperatur	T_{h_0}	273,15 K	-
Thermosphäre²			
Unterer Druck	p_{h_0}	0 Pa	-
Schichtbreite	-	420.000 m	-
Temperaturgradient	a	-0,000405 K/m	-
Untere Temperatur	T_{h_0}	173,15 K	-
Exosphäre²			
Unterer Druck	p_{h_0}	0 Pa	-
Schichtbreite	-	-	-
Temperaturgradient	a	-0,0 K/m	-
Untere Temperatur	T_{h_0}	3 K	-
Aerobee 150 Rakete³			
Gewicht			
Nase	-	7,03 kg	15,5 lb
Max. Nutzlast	-	68,04 kg	150 lb
Flüssigtank	-	116,12 kg	256 lb
Helium	-	2,31 kg	5,1 lb
Oxidationsmittel ⁴	-	356,73 kg	764,4 lb
Brennstoff ⁴	-	138,08 kg	304,4 lb
Treibstoffmasse ⁴	m_p	484,8 kg	-
Feststofftank	-	28,12 kg	62 lb
Festbrennstoff ⁴	m_p	117,94 kg	260 lb
Gesamtmasse ⁴	m_{Rakete}	756,33 kg	-
Querschnittsfläche	A_{Rakete}	0,11 m ³	1,23 ft ³

Anhang I: Simulationswerte

Flüssigantrieb			
Düsenfläche	A_n	0,03 m ²	42,75 in ²
Massenstrom	\dot{m}_p	9,39 kg/s	20,71 lb/s
Abgasgeschwindigkeit	v_e	1417,32 m/s	4650 ft/s
Düsendruck	p_n	101325 Pa	-
Nominalschub	-	18,24 kN	4100 lbs
Bezeichnung	Formelzeichen	SI-Einheiten	Angloamerikanisches ¹
Feststoffbooster			
Düsenfläche	A_n	0,04 m ²	67,2 in ²
Massenstrom	\dot{m}_p	47,17 kg/s	104 lb/s
Abgasgeschwindigkeit	v_e	1747,6 m/s	-
Nominalschub	-	82,73 kN	18.600 lbs

¹ Umrechnungswerte nach WolframAlpha

² Werte aus [18]

³ Werte aus [19, 21]

⁴ Verändert sich über Simulation