

Simulation Fallschirmspringer

Patrice Keusch, Hauptrasse 30, 5312 Döttingen, Tel. 056 535 99 63,

keuscpat@students.zhaw.ch

Severin Müller, Kesselstrasse 20, 8200 Schaffhausen, Tel. 052 620 40 09

muelles5@students.zhaw.ch

Softwareprojekt 2 mit Anwendungen von Methoden aus dem Bereich der
Nummerischen Mathematik

ZHAW

Kursverantwortlicher: Philippe Nahlik

Scrummaster: Lukas Eppler

Auftraggeber: Albert Heuberger

15.06.2012

Inhaltsverzeichnis

1	Einleitung.....	5
1.1	Ausgangslage	5
1.2	Aufgabenstellung.....	5
1.3	Eigenmotivation.....	5
1.4	Vorgehen	6
1.5	Teilziel 1 – Mathematische Grundlagen und GUI.....	7
1.6	Teilziel 2 – Berechnungen darstellen im GUI.....	7
1.7	Teilziel 3 – Code bereinigt und Ziel erreicht	7
1.8	Teilziel 4 – Dokumentation und Präsentation fertigstellen	7
2	Analyse	8
2.1	Luftwiderstand.....	8
2.2	Bewegungsgesetz von Newton.....	8
2.3	Differentialgleichung	9
2.4	Zustand des Springers.....	9
2.5	Widerstand des Windes.....	10
2.6	Ablauf der Simulation	10
2.6.1	Phase 1 – Flugzeug.....	10
2.6.2	Phase 2 – Absprung aus dem Flugzeug	11
2.6.3	Phase 3 – Beschleunigung bis zur maximalen Geschwindigkeit	11
2.6.4	Phase 4 – Fallschirm wird geöffnet.....	12
2.6.5	Phase 5 – Der Fallschirmspringer gleitet.....	12
2.6.6	Phase 6 – Landen am Zielpunkt	12
2.7	Runge-Kutta Verfahren.....	13
2.8	Exakte Lösungen finden.....	13
2.9	Grafische Darstellung	14
3	Umsetzung.....	15

3.1	SimulationObject	15
3.2	Vektor Berechnungen.....	15
3.3	Windfunktion.....	15
3.4	Widerstandfunktion.....	15
3.5	Springer	16
3.5.1	Schrittweite der Berechnung	16
3.5.2	Springer Flugbahn berechnen.....	17
3.6	Runge-Kutta Funktion.....	17
3.7	Berechnungen der Ableitungen.....	18
3.8	Exakte Lösungen finden.....	19
3.9	Grafische Darstellung	20
3.9.1	Hauptfenster	20
3.9.2	Flugzeug	20
3.9.3	Springer.....	21
4	Software (SM).....	22
4.1	Grundstruktur	22
4.2	Grafische Benutzeroberfläche (GUI).....	22
4.2.1	Grundgerüst	22
4.2.2	Validierungen.....	23
4.3	Kapselung der Parameter	24
4.4	Problem: Blockieren des GUI.....	25
4.5	Threads	25
5	Simulation - Kurze Flugzeit	27
6	Fazit	30
6.1	Sevi	30
6.2	Patrice.....	30
6.3	Danksagung	30
7	Anhang.....	31

7.1	Quellenverzeichnis	31
7.2	Abbildungsverzeichnis	31
7.3	Tabellenverzeichnis	32

1 Einleitung

1.1 Ausgangslage

Dieses Projekt wurde im Kurs „Softwareprojekt 2“ im vierten Semester des Studiengangs Informatik am Standort Zürich der ZHAW realisiert. Ziel des Kurses war, ein kleines Projekt zu realisieren, welches professionelle Standards erfüllt und im Inhalt eine Anwendung von Methoden aus den Bereichen Numerische Mathematik, Algorithmen und Datenstrukturen oder Theoretische Informatik sein soll. Die Aufgabenstellung und die fachliche Betreuung übernimmt ein Herr Heuberger in der Rolle des Auftraggebers. Herr Eppler übernimmt die Rolle des Scrummasters, welcher die Projektplanung und Projektumsetzung überwacht.¹

1.2 Aufgabenstellung

Als Aufgabenstellung haben wir uns die Simulation eines Fallschirmsprunges ausgesucht. Für eine realitätsnahe Simulation braucht es numerische Methoden, Differentialgleichungen und Physikalische Gleichungen.

Wir haben uns folgendes Szenario überlegt: Ein Fallschirmspringer sitzt im Flugzeug und macht sich bereit für seinen Absprung. Diverse Parameter wie Windstärke, Gravitation, Flugzeuggeschwindigkeit usw. sind bekannt. Der Fallschirmspringer möchte nun einen bestimmten Landepunkt erreichen. Er kann die Koordinaten des gewünschten Ziels im Programm eingeben und seine persönlichen Parameter ergänzen. (Gewicht, Fallschirmgrösse, nach welcher Zeit der Fallschirm geöffnet wird, etc.). Das Programm soll nun anhand der physikalischen Gleichungen und numerischen Verfahren zum Lösen von Differentialgleichungen den Absprungspunkt aus dem Flugzeug so berechnen, dass der Springer eine Punktlandung hinlegt. Mit Hilfe dieses Programmes können dann verschiedene Zeiten für das Öffnen des Fallschirmes simuliert werden. Je nach dem was gewünscht ist, kann so die längste oder kürzeste Flugzeit berechnet werden, ohne dass sich der Springer bei der Landung verletzt.

1.3 Eigenmotivation

Bei der Themenwahl konnten wir uns mit keinem der vorgegebenen Themen so richtig anfreunden. Wir überlegten uns daher eine eigene Aufgabenstellung. Die Idee für die Simulation eines Fallschirmsprunges kam uns durch ein Spiel, wo man einen Fallschirmspringer fallen lässt und dann im richtigen Augenblick den Fallschirm öffnen muss, so dass dieser auf der Zielplattform landet und auch nicht zu schnell beim Aufprall ist.

Zudem ist ein Freund von Patrice Keusch ein „Sky Diver“, was das Thema umso interessanter machte und man auch von realen Erfahrungen profitieren konnte.

Die Vorstellung einen natürlichen Vorgang wie der Fallschirmsprung mit Hilfe einer Differentialgleichung zu beschreiben und dies mit nummerischen Verfahren zu lösen, motivierte uns sehr. Wir haben uns bisher viel

¹ Nahlik [2012]

theoretisches numerisches Wissen angeeignet und waren nun gespannt darauf, wie sich dies bei einer realen Simulation verhält.

Schlussendlich hat das Software Projekt mit der Umsetzung der Simulation in einer Programmiersprache zu tun. In der Berufstätigkeit von Patrice Keusch kommt das Entwickeln jedoch zu kurz, was es interessant macht, sich intensiv mit der Programmierung zu beschäftigen. Severin Müller arbeitet zwar als Entwickler, jedoch nicht in einem Java Umfeld, was es für ihn ebenfalls spannend macht.

Unser Ziel war ganz klar, die Aufgabenstellung möglichst gut zu erfüllen und die berechneten Werte in einem GUI aussagekräftig darzustellen. Dafür mussten wir uns mit der Programmierung eines GUI auseinander setzen und mit der Umsetzung von numerischen Methoden in einer Hochsprache.

1.4 Vorgehen

Für die Umsetzung des Projekts haben wir uns für folgende Hilfsmittel entschieden:

- Iterationsplan mit User Stores und Tasks
- Dynamischer Iterationsplan (Google Docs)
- Dynamisches Fortschrittsdokument (Google Docs)

Als erstes haben wir zusammen mit dem Kunden die Aufgabenstellung festgelegt. Danach haben wir uns mit dem Thema auseinandergesetzt, um dann einen fixen Iterationsplan zu erstellen. Im Iterationsplan haben wir uns User Stories und die passenden Tasks dazu überlegt. Wir haben uns dabei auch Gedanken über die zur Verfügung stehende Zeit, Velocity und Task-Aufteilung gemacht. Für das Projekt müssen wir pro Person rund 60 Stunden aufwenden und wir haben gut 11 Wochen Zeit. Wenn wir nun die Velocity einberechnen und die Anzahl Wochen beachten, gibt dies in etwa 8 Stunden pro Woche, welche wir investieren müssen.

Um das Projektmanagement dynamisch zu halten, haben wir uns entschieden einen Iterationsplan und ein Fortschrittsdokument auf Google Docs zu erstellen. Im Iterationsplan ist jeweils der aktuelle Status der Tasks und User Stories ersichtlich. Im Fortschrittsdokument werden Meeting Ergebnisse, Tätigkeiten und ToDo-Listen abgelegt.

Bei der Erarbeitung des Iterationsplans haben wir uns für vier Teilziele entschieden. Wir wollten dabei vor allem das Risiko vermeiden, dass wir zuletzt noch vor grossen Problemstellungen stehen und nicht rechtzeitig fertig werden.

1.5 Teilziel 1 – Mathematische Grundlagen und GUI

Bevor wir mit der Programmierung begonnen haben, mussten zuerst die mathematischen Grundlagen und die physikalischen Gleichungen klar sein. Der Ablauf eines Fallschirmsprunges muss verstanden sein und Überlegungen zur Lösung mit Hilfe von nummerischen Methoden müssen gemacht werden.

Das Grundlayout des grafischen User Interfaces muss stehen und es muss klar sein, wie wir die Flugbahn grafisch darstellen können. Hochsprache und Frameworks müssen festgelegt sein.

1.6 Teilziel 2 – Berechnungen darstellen im GUI

Programmierung der nummerischen Methoden und erste Berechnungen müssen möglich sein. Einbeziehen der korrekten physikalischen Gleichungen. (Widerstand, Windstärke, etc.)

Die mathematischen Berechnungen und das GUI müssen zusammen agieren und es können im GUI erste Koordinaten gezeichnet werden. Im GUI können die Parameter definiert werden und es finden Validierungen statt.

1.7 Teilziel 3 – Code bereinigt und Ziel erreicht

Die Software wird komplettiert und verfeinert. Das GUI wird mit nützlichen Funktionen und Darstellungen erweitert. Die Flugbahn kann mit diversen Parametern berechnet und dargestellt werden. Die Lösung der Aufgabenstellung ist erreicht.

1.8 Teilziel 4 – Dokumentation und Präsentation fertigstellen

Die Konzeptdokumentation ist fertiggestellt. Der Programmcode ist bereinigt und dokumentiert. Die Abschlusspräsentation ist vorbereitet und einstudiert. Die CD für die Abgabe wird mit folgenden Inhalten erstellt:

- Handout der Präsentation
- Code (dokumentiert)
- Allenfalls notwendige Libraries zum Ausführen des Programms
- Konzeptdokument, welches die Anforderungen enthält
- Projektplan, allenfalls im Konzeptdokument vorhanden
- In einer üblichen Umgebung lauffähiges Programm²

² Nahlik [2012]

2 Analyse

In der Analyse wird die Problemstellung analysiert und einzelne Themen genauer angeschaut. Benötigte physikalische Gleichungen und numerische Verfahren werden erklärt. Entsprechende Funktionen werden gleich definiert.

2.1 Luftwiderstand

Der Luftwiderstand wird von vier Faktoren beeinflusst:

1. Faktor: Der Luftwiderstand ist proportional zu der grössten Querschnittsfläche A , die senkrecht zur Strömungsrichtung steht.
2. Faktor: Der Luftwiderstand ist proportional zur Geschwindigkeit v im Quadrat.
3. Faktor: Der Luftwiderstand ist proportional zur Luftdichte p .
4. Faktor: der Luftwiderstand hängt von der Form des Körpers ab.

Der Luftwiderstand kann daher durch diese Formel beschrieben werden:

$$F = \frac{1}{2} p A c_w v^2$$

Der Faktor c_w bezeichnet den sogenannten Formfaktor. Je nach Form des Objekts, ist dieser Wert höher oder niedriger. Dieser Wert muss normalerweise in echten Simulationen gemessen werden.³ In unserer Simulation starten wir mit einem Wert aus Dokumentationen für den freien Fall. Den c_w Wert für die Zeit während des Öffnens des Fallschirms und für diejenige während des Fluges mit offenem Fallschirm werden wir mit Programm Simulationen ermitteln.

2.2 Bewegungsgesetz von Newton

Für die Simulation brauchen wir das Bewegungsgesetz von Newton. Es wirken folgende Kräfte auf den Fallschirmspringer:

Schwerkraft: $F_1 = m * g$

Widerstandskraft: $F_2 = -\frac{1}{2} p A c_w v^2 = -kv^2$

Gesamtkraft: $F = F_1 + F_2 = mg - kv^2$

³ Kirchgraber [1993]

2.3 Differentialgleichung

Wir wissen nun, dass auf den Körper zwei Kräfte wirken: Die Gewichtskraft und die Widerstandskraft, die in die entgegengesetzte Richtung gerichtet ist.

Die Gesamtkraft können wir nun beschreiben:

$$ma = mg - cw \frac{1}{2} v^2 pA$$

Daraus können wir die Bewegungsgleichung aufstellen:

$$m \frac{dv}{dt} = mg - cw \frac{1}{2} v^2 pA$$

Formen wir diese Differentialgleichung um, erhalten wir eine passende Form für die zukünftigen Berechnungen der jeweiligen Beschleunigung zu der Zeit t . Dadurch können wir nun die Flugbahn ermitteln.⁴

$$a = \frac{dv}{dt} = g - \frac{cw \frac{1}{2} v^2 pA}{m}$$

2.4 Zustand des Springers

Der Springer ist das zentrale Objekt in unserer Berechnung. Er muss zu jedem Zeitpunkt seine Position und seine Geschwindigkeit kennen.

Der Zustand des Springers wird durch einen Vektor z in Abhängigkeit zu t beschrieben:

$$z(t) = \begin{pmatrix} x1 \rightarrow xPositionsVektor \\ y1 \rightarrow yPositionsVektor \\ v1 \rightarrow xGeschwindigkeitsVektor \\ v2 \rightarrow yGeschwindigkeitsVektor \end{pmatrix}$$

Die Positionsvektoren abgeleitet ergeben die Geschwindigkeitsvektoren und die Geschwindigkeitsvektoren abgeleitet ergeben die Beschleunigungsvektoren.⁵

$$z(t) = \begin{pmatrix} x1 \\ x2 \\ v1 \\ v2 \end{pmatrix} = \begin{pmatrix} x1' \\ y1' \\ v1' \\ v2' \end{pmatrix} = \begin{pmatrix} v1 \\ v2 \\ a1 \\ a2 \end{pmatrix} = \begin{pmatrix} v_x \\ v_y \\ a_x \\ a_y \end{pmatrix}$$

Mit der aufgestellten Differentialgleichung können wir demnach die benötigten Werte berechnen.

⁴ Maxim [2010]

⁵ Halliday [2007]

2.5 Widerstand des Windes

Der Wind ist ein weiterer Faktor aus der Realität. Der Wind wird als Windwiderstand betrachtet.

Die Windstärke ist definiert durch:

$$\begin{pmatrix} v_1 \rightarrow \text{Windstärke in } y\text{-Richtung} \\ v_2 \rightarrow \text{Windstärke in } x\text{-Richtung} \end{pmatrix}$$

Je nach Positionsgröße ist die Windstärke verschieden. Die Windfunktion $W(y)$ ist abhängig von der y-Koordinate. Um die Simulation interessant zu gestalten, haben wir verschiedene Werte angenommen.

$$W(y) = \begin{cases} \begin{pmatrix} -30 \\ 0 \end{pmatrix}, & y < 1000 \\ \begin{pmatrix} 25 \\ 0 \end{pmatrix}, & y < 2000 \\ \begin{pmatrix} 20 \\ 20 \end{pmatrix}, & y < 3000 \\ \begin{pmatrix} 10 \\ 0 \end{pmatrix}, & y < 4000 \\ \begin{pmatrix} 20 \\ 20 \end{pmatrix}, & y \geq 4000 \end{cases}$$

Um nun die Windfunktion einzuberechnen, wird diese jeweils von der aktuellen Geschwindigkeit subtrahiert, bevor danach die neue Geschwindigkeit mit Hilfe der Ableitungen berechnet wird:

$$\begin{pmatrix} x_1 \\ y_1 \\ v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \\ v_1 \\ v_2 \end{pmatrix} - \begin{pmatrix} x_1 \\ y_1 \\ W(y)[0] \\ W(y)[1] \end{pmatrix}$$

2.6 Ablauf der Simulation

Wir beschreiben den Ablauf der Simulation sehr detailliert. Bevor mit der Berechnung begonnen wird, muss der Ablauf verstanden sein.

2.6.1 Phase 1 – Flugzeug

Zuerst ist der Fallschirmspringer im Flugzeug. Er besitzt daher die Geschwindigkeit v_1 und die Koordinaten des Flugzeugs.

Für die Simulation der Flugbahn des Flugzeugs können wir eine einfache Funktion verwenden. Wir vernachlässigen hier den Luftwiderstand, die Gravitation und die weitere Einflussfaktoren. Wir gehen davon aus, dass sich das Flugzeug immer mit der gleichen Geschwindigkeit bewegt und in der gleichen Höhe hält.

Es ergibt sich daraus folgende Ortsfunktion:

$$\text{Flugzeug}(t) = \begin{pmatrix} x \rightarrow \text{horizPos} \\ y \rightarrow \text{vertPos} \end{pmatrix} + \begin{pmatrix} v1 \rightarrow \text{horizGeschw} \\ v2 \rightarrow \text{vertGeschw} \end{pmatrix} + t$$

Wobei in unserer Simulation $v2 = 0$ ist und $v1$ beliebig gesetzt werden kann.

2.6.2 Phase 2 – Absprung aus dem Flugzeug

Beim Absprung übernimmt der Springer die Werte des Flugzeugs zum Zeitpunkt t:

$$z(t) = \begin{pmatrix} x1 = xPositionFlugzeug \\ y1 = yPositionFlugzeug \\ v1 = xGeschwindigkeitFlugzeug \\ v2 = yGeschwindigkeitFlugzeug \end{pmatrix}$$

Der Springer wird danach immer schneller, zudem passt sich die $v1$ Geschwindigkeit der Windgeschwindigkeit in dieser Höhe an.

2.6.3 Phase 3 – Beschleunigung bis zur maximalen Geschwindigkeit

Der Springer erreicht seine maximale Geschwindigkeit nach ca. 10 Sekunden abhängig von Gewicht und der vertikalen Windstärke.⁶ Die maximale Geschwindigkeit kann berechnet werden, indem man die Beschleunigung auf 0 setzt:

$$a = 0$$

$$\begin{aligned} ma &= mg - cw \frac{1}{2} v^2 pA \\ 0 &= mg - cw \frac{1}{2} v^2 pA \end{aligned}$$

Durch umformen erhält man die Endgeschwindigkeit:

$$v_E = \sqrt{\frac{2mg}{cwpA}}$$

Je grösser die Fläche, desto kleiner wird die Endgeschwindigkeit. In unserer Simulation legen wir die Startfläche des Springers auf 0.5 m^2 . In der Realität ändert sich diese Fläche immer wieder, je nach Haltung des Springers. Durch eine möglichst senkrechte Haltung würde der Springer die höchste Geschwindigkeit erreichen und durch eine waagrechte Haltung mit gestreckten Armen eine relativ tiefe Endgeschwindigkeit.⁷

⁶ Wikipedia [2012]

⁷ Maxim [2012]

2.6.4 Phase 4 – Fallschirm wird geöffnet

In unserer Simulation können wir den Zeitpunkt des Öffnens vorgeben. Die komplette Entfaltungszeit des Fallschirms beträgt 2-5 Sekunden.⁸ In unserer Simulation haben wir den Wert auf 2 Sekunden gesetzt. In dieser Zeit ändert sich die Fläche A von $0,5 \text{ m}^2$ auf die eingestellte Fallschirm Fläche zum Beispiel 20 m^2 .

Ebenso ändert sich der cw-Wert in diesen zwei Sekunden. In unserer Simulation ist der cw-Startwert bei 0,5 und der Endwert bei 3,0.⁹

Die Widerstandsfunktion wird daher erweitert:

$$\text{Widerstand}(t) = -\frac{1}{2} p A(t) c_w(t) v^2$$

Bei jedem Berechnungsschritt wird daher die Funktion $A(t)$ und $c_w(t)$ neu berechnet. Abhängig der Schrittweite der Berechnung, wird A und c_w Wert linear erhöht, bis der Fallschirm offen ist.

$$A(t) = \begin{cases} A, & t < \text{toffen} \\ A + h, & \text{tgeöffnet} > t \geq \text{toffen} \\ A, & t > \text{tgeöffnet} \end{cases}$$

$$c_w(t) = \begin{cases} c_w, & t < \text{toffen} \\ c_w + h, & \text{tgeöffnet} > t \geq \text{toffen} \\ c_w, & t > \text{tgeöffnet} \end{cases}$$

2.6.5 Phase 5 – Der Fallschirmspringer gleitet

Da der Widerstand durch den offenen Fallschirm sehr gross ist, gleitet der Fallschirmspringer relativ langsam hinunter. Die Endgeschwindigkeit darf nun nicht mehr hoch sein, da der Springer mit dieser Geschwindigkeit auf dem Boden landen wird. Beim Fallschirmspringen wird die Grösse das Fallschirms so gewählt, das die Endgeschwindigkeit bei der Landung zwischen $3,5 \text{ m/s}$ und $5,0 \text{ m/s}$ liegt.¹⁰

Wie in allen Phasen nähert sich die Geschwindigkeit in x-Richtung der x-Geschwindigkeit des Windes zur gegebenen Höhe an, bis diese erreicht wird.

2.6.6 Phase 6 – Landen am Zielpunkt

Sobald die Koordinate $y = 0$ ist, ist der Springer gelandet. Dabei kommt es abrupt zur Verringerung der Geschwindigkeit auf 0.

⁸ Wikipedia [2012]

⁹ Kirchgraber [1993]

¹⁰ Wikipedia [2012]

2.7 Runge-Kutta Verfahren

Als nummerisches Verfahren zur Lösung der Differentialgleichung haben wir uns für das Runge-Kutta Verfahren entschieden. Für das Verständnis und die ersten Versuche haben wir ebenfalls das Euler-Verfahren verwendet. Das Euler-Verfahren ist einfacher anzuwenden, hat jedoch nur eine Konsistenz – und Konvergenzordnung von $p = 1$, wogegen das RK-Verfahren $p = 4$ besitzt. Daher werden wir das RK-Verfahren einsetzen, welches die wesentlich besseren Näherungen berechnet.¹¹

Wie das Euler und das RK-Verfahren in Java eingesetzt werden können, haben wir bei Herrn Heuberger in einer Schulung gelernt und dafür auch den entsprechenden Java Code erhalten. Wir haben den Code für unsere Berechnungen angepasst und erweitert.

Das Runge-Kutta Verfahren haben alle Studierenden im Kurs „Numerik 2“ kennen gelernt, daher wird der Algorithmus (Abb. 1) nicht näher erklärt.

Das Verfahren

$$\begin{aligned}k_1 &= f(t_n, y_n) \\k_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} k_1\right) \\k_3 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} k_2\right) \\k_4 &= f(t_n + h, y_n + h k_3) \\y_{n+1} &= y_n + h \frac{1}{6} (k_1 + 2 k_2 + 2 k_3 + k_4)\end{aligned}$$

heißt klassisches vierstufiges Runge²-Kutta³-Verfahren. Es besitzt die Konsistenz- und Konvergenzordnung $p = 4$.

Abbildung 1: Runge-Kutta Verfahren¹²

2.8 Exakte Lösungen finden

Um die exakte Lösung zu finden, können wir eine simple Überlegung anstellen. Wir berechnen die Werte mit dem RK-Verfahren und geben als Startwert eine mögliche Koordinate, wie zum Beispiel (1000,0) ein. Diese Berechnung wird uns einen Landepunkt ausgeben, wo die y-Koordinate = 0 ist. Nun haben wir einen berechneten Wert x . Wollten wir nun bei der Koordinate $x = 300$ landen, sind aber auf der Koordinate $x = 500$ gelandet, können wir das RK-Verfahren nochmals mit geändertem Startwert berechnen lassen. Wir würden die neue Berechnung in diesem Beispiel mit den Werten (1000,200) starten. Da die Berechnung

¹¹ Bärwolff [2011]

¹² Knorrenschild [2008]

symetrisch ist und die Parameter sich nicht mehr ändern, landet der Fallschirmspringer bei der zweiten Berechnung exakt am gewünschten Zielort.

2.9 Grafische Darstellung

Für die grafische Darstellung der Simulation wurden einige Szenarien ins Auge gefasst. Zunächst wollten wir ein Koordinatensystem zeichnen und die Simulation darauf ausführen. Dies hatte allerdings den Nachteil, dass das Darüberlegen von grafischen Objekten zu aufwändig war. Nach eingehender Analyse der Möglichkeiten haben wir uns dann dafür entschieden, dass wir eine Grafik mit einem Raster einbinden. Für die beweglichen Objekte benutzen wir Grafiken von einem Flugzeug und einem Fallschirmspringer. Für den Springer benötigten wir zwei Grafiken – eine ohne und eine mit Fallschirm.

Die Oberfläche für die Eingabe der Parameter war von vornherein klar. Allerdings sind im Laufe des Projektes einige Objekte dazugekommen. So wollten wir zum Beispiel die momentane Position des Springers als Koordinaten sowie die momentane Fallgeschwindigkeit auf dem GUI anzeigen lassen.

Für die beweglichen Objekte mussten wir uns überlegen, wie wir die am besten realistisch darstellen können. Schnell wurde klar, dass dies am besten geht, wenn wir die Flugbahn des Springers als Array haben, die einen Zeitwert beinhaltet.

3 Umsetzung

3.1 SimulationObject

Die Klasse SimulationObject.java verwenden wir als Daten Objekt. Darin sind über 25 Variablen deklariert, welche mit Getter-Methoden abgerufen und mit Setter-Methoden gesetzt werden können.

3.2 Vektor Berechnungen

Für die Vektor Berechnungen verwenden wir die Klasse linalg4_4.java. Diese Klasse hat uns Herr Heuberger zur Verfügung gestellt. Die Klasse Springer.java erbt die Methoden der Klasse linalg4_4.java, wodurch wir die benötigten Vektor Berechnungsmethoden aufrufen können.

3.3 Windfunktion

Zu jedem Zeitpunkt t setzen wir die Windstärke neu. Je nach Höhe (gespeichert in $z[1]$) wird ein anderer Wind festgelegt.

Ein Teil der `wind()` Funktion aus Springer.java

```
// Windfunktion wind(t,z)
public double[] wind (double t, double[] z)
{
    double[]res = new double[4];

    res[0] = simulationObject.getWindSpeed();
    res[1] = 0;

    // Höhe z[1] wird jeweils ausgelesen und verglichen. Je nach Höhe wird ein anderer Wind gesetzt.
    if(z[1] < 4000){
        res[0] = 10;
        res[1] = 0;
    }
}
```

Die Windgeschwindigkeit wird dann bei den Berechnungen vor den Ableitungen abgezogen. Wobei das Array $u[]$ die neuen Beschleunigungsvektoren speichert.

Ein Teil der `w()` Funktion aus Springer.java

```
u[0] = z[2]-wind(t,z)[0];
u[1] = z[3]-wind(t,z)[1];
```

3.4 Widerstandfunktion

Im Gegensatz nur Analyse verwenden wir hier für den Widerstand vorerst nur diese Gleichung:

$$Widerstand(t) = \frac{1}{2} pA(t) cw(t)^2$$

Da die Geschwindigkeit durch Vektoren definiert ist, lassen wir hier den Faktor v^2 noch weg, werden diesen aber in den Berechnungen der Ableitungen nachträglich mit einbeziehen.

Sobald der Fallschirm geöffnet wird, wird der cw und der Flächenwert schrittweise erhöht.

calcWiderstand() und *calcCW()* Funktion aus Springer.java

```
public void calcWiderstand(double t){

    r = calcCW(t) * 0.5 * 1.2 * calcFlaeche(t); // r(t) = cw(t) * 0.5 * p * A(t)
    //System.out.println("Widerstand: " + r + "Fläche: " + springerFlaeche + " " + " CW: " + cw + " ");

}

// CW Funktion, Abhängig von der Fläche des Fallschirmes und der Zeit
public double calcCW(double t){

    if(t >= simulationObject.getTOffen() && t < (simulationObject.getTOffen() + simulationObject.getTOeffnen()))
        cw = cw + cwAddierenBeimOeffnen;

    return cw;
}
```

3.5 Springer

Die Springer Klasse enthält sämtliche Berechnungen. Bei der Instanzierung einer Springer Klasse muss ein SimulationObject-Objekt mitgegeben werden. Die Klasse Springer wird dann mit allen wichtigen Werten initialisiert.

3.5.1 Schrittweite der Berechnung

Das Runge-Kutta Verfahren kann mit verschiedenen Schrittweiten berechnet werden, daher muss die Schrittweite für die Fallschirmfläche und den cw-Wert während des Öffnens vor jeder Berechnung neu ermittelt werden. Dies wird bei der Initialisierung ausgelöst.

CalcSchritte() Funktion aus Springer.java

```
public void calcSchritte(){

    // Schrittweite Berechnen für Flächen und cw Funktion
    anzahlSchritteBeimOeffnen = simulationObject.getTOeffnen() / simulationObject.getSchrittweiteH();

    // Schrittflächen berechnen für Flächenfunktion
    differenzFlaeche = simulationObject.getParachuteArea() - simulationObject.getSpringerFlaecheStart();
    flaecheAddierenBeimOeffnen = differenzFlaeche / anzahlSchritteBeimOeffnen;

    // Schrittflächen berechnen für cw Funktion
    cwDifferenz = simulationObject.getCwEnde() - simulationObject.getCwStart();
    cwAddierenBeimOeffnen = cwDifferenz / anzahlSchritteBeimOeffnen;

}
```

3.5.2 Springer Flugbahn berechnen

Die Flugbahnberechnung wird mit der Methode `calcSpringer()` gestartet.

Die Funktion `fTable()` startet das Runge-Kutta Verfahren und gibt danach ein 5-Array aus. Folgende Parameter werden dort mitgegeben:

Tabelle 1 Runge Kutta Parameter

Ausgabe tStart	Ab welchem t soll die Tabelle ausgegeben werden
Ausgabe Schrittweite	Welche Schrittweite für die Ausgabe
Ausgabe tEnde	Bis zur welcher Laufzeit soll die Ausgabe erstellt werden
tStart	Startzeit der Berechnungen
yAnfang	Werte für das Anfangswertproblem
yAnfang[0]	X Koordinate
yAnfang[1]	Y Koordinate
yAnfang[2]	Flugzeuggeschwindigkeit
yAnfang[3]	Springer Geschwindigkeit
yAnfang[4]	Variable für Laufzeit
Schrittweite	h Schrittweite für Runge-Kutta

Durch die zwei verschiedenen Schrittweiten kann erreicht werden, dass mit einer kleineren Schrittweite gerechnet wird, die Ausgabe jedoch eine grössere Schrittweite aufweist. Dadurch bekommt man genauere Resultate, jedoch muss pro Sekunde nicht 100-Mal das gleiche Objekt auf fast den gleichen Punkt gezeichnet werden.

Ein Teil der `CalcSpringer()` Funktion aus Springer.java

```
public void calcSpringer() {  
  
    // {xStart-Koordinate, yStart-Koordinate (Flughöhe), Flugzeuggeschwindigkeit = Springer Startgeschwindigkeit,  
    // yStart Geschwindigkeit Springer, Laufzeit}  
    double[] yAnfang = {0, simulationObject.getAltitude(), simulationObject.getPlaneSpeed(), 0, 0};  
  
    // Berechnungen starten, erste drei Werte für TabellenAusgabe: tStart, tSchrittweite, tEnde,  
    // Danach tStart, yAnfang Werte (siehe oben), Genauigkeit der Berechnungen.  
    double[][] result = fTable(0, simulationObject.getSchrittweiteResult(), simulationObject.getTEnde(),  
        0, yAnfang, simulationObject.getSchrittweiteH());  
}
```

3.6 Runge-Kutta Funktion

Die Runge-Kutta Funktion wird aus der `fTable()` Funktion aufgerufen und bekommt die Anfangswertprobleme mitgeliefert.

Dies ist ein Auszug aus einer Iteration im RK-Verfahren. Als erstes wird der aktuelle Widerstand mit der Funktion `calcWiderstand()` berechnet. Danach wird gemäss RK-Algorithmus vorgegangen und die K1-4

berechnet. Dort wird jeweils die Funktion `w()` aufgerufen mit der Laufzeit und dem Array `y[][]` mit den Anfangswertproblemen als Parameter. Die Funktion `w()` übergibt die neu berechneten Vektoren. Nachdem diese mit der Schrittweite multipliziert wurden, werden die neuen Vektoren zum alten Vektor addiert und man erhält so die neuen Werte.

Auszug von RK4 Funktion aus Springer.java

```
// Widerstand berechnen und abspeichern in r
calcWiderstand(t);

//K1 - RK4-Verfahren
ka = w(t,y);
ya = addVector(y, multScalarVector(h/2, ka));
ta = t + h/2;
|
//K2 - RK4-Verfahren
kb = w(ta,ya);
yb = addVector(y, multScalarVector(h/2, kb));
tb = t + h/2;
```

3.7 Berechnungen der Ableitungen

Die Funktion `w()` ist die komplizierteste Methode in unserer Software. Sie beinhaltet die Ableitungen für die vier Zustandsgrößen. Die uns bekannte Differentialgleichung für die Flugbahn im eindimensionalen Fall mussten wir nun so umschreiben, dass diese auch im zweidimensionalen Fall mit `x` und `y` Vektoren funktioniert.

Die Geschwindigkeitsvektoren ergeben die neuen Positionsvektoren und können daher einfach zugewiesen werden:

Auszug von w Funktion aus Springer.java

```
res[0] = z[2];
res[1] = z[3];
```

Wir berechnen den Betrag der Geschwindigkeitsvektoren und speichern diese in `uBetrag`. Mit Hilfe des Einheitsvektors, können wir die neuen Vektor berechnen.¹³

$$\text{Einheitsvektor von } x\text{Geschwindigkeitsvektor} = \frac{x\text{GeschwVektor}}{\text{GeschwBetrag}} = \frac{v1}{\sqrt{v1^2 + v2^2}}$$

$$x\text{Geschwindigkeitsvektor} = \frac{v1}{u\text{Betrag}} u\text{Betrag}^2$$

¹³ Massjung [2012]

Unter anderem kompletieren wir hier dadurch auch den Widerstandsfaktor r mit dem Faktor $u\text{Betrag}^2$. Für den Beschleunigungsvektor a_x müssen wir keine Gravitation beachten. Daher lassen wir sie dort weg.

Setzen wir diese Überlegungen in unsere Differentialgleichung ein, erhalten wir diese Berechnung.

$$\begin{pmatrix} a_x \\ a_y \end{pmatrix} = \begin{pmatrix} -r v1 u\text{Betrag} u\text{Betrag} \\ -g - \frac{r v2 u\text{Betrag} u\text{Betrag}}{m u\text{Betrag}} \end{pmatrix}$$

Wobei wir $u\text{Betrag}$ weglassen können.

$$\begin{pmatrix} a_x \\ a_y \end{pmatrix} = \begin{pmatrix} -r v1 u\text{Betrag} \\ -g - \frac{r v2 u\text{Betrag}}{m} \end{pmatrix}$$

Auszug von `w()` Funktion aus Springer.java

```
uBetrag = Math.sqrt(u[0]*u[0] + u[1]*u[1]);
res[2] = - r * (u[0]*uBetrag); // Ableitung der Geschwindigkeit ergibt Beschleunigung
res[3] = - g - (r * (u[1]*uBetrag))/simulationObject.getSpringerGewicht();
```

3.8 Exakte Lösungen finden

Nach dem ersten Durchlauf der Berechnungen wird der Landepunkt `KoNull` ausgelesen. Wenn dieser nicht mit dem gewünschten Landepunkt übereinstimmt, wird der Springer neu initialisiert und die Berechnung mit geänderten Anfangswerten nochmals gestartet.

Auszug von `calcSpringer` Funktion aus Springer.java

```
// Neuer SpringerAbsprungpunkt wird festgelegt
yAnfang[0] = yAnfang[0] - (KoNull - (simulationObject.getLandePunkt() + 1));
```

3.9 Grafische Darstellung

3.9.1 Hauptfenster

Das GUI wurde mit Swing von Java umgesetzt. Wir haben die Bereiche durch Container getrennt. In der Abb. 2 links sieht man die Box für die Parameter, rechts die Simulationsgrafik:

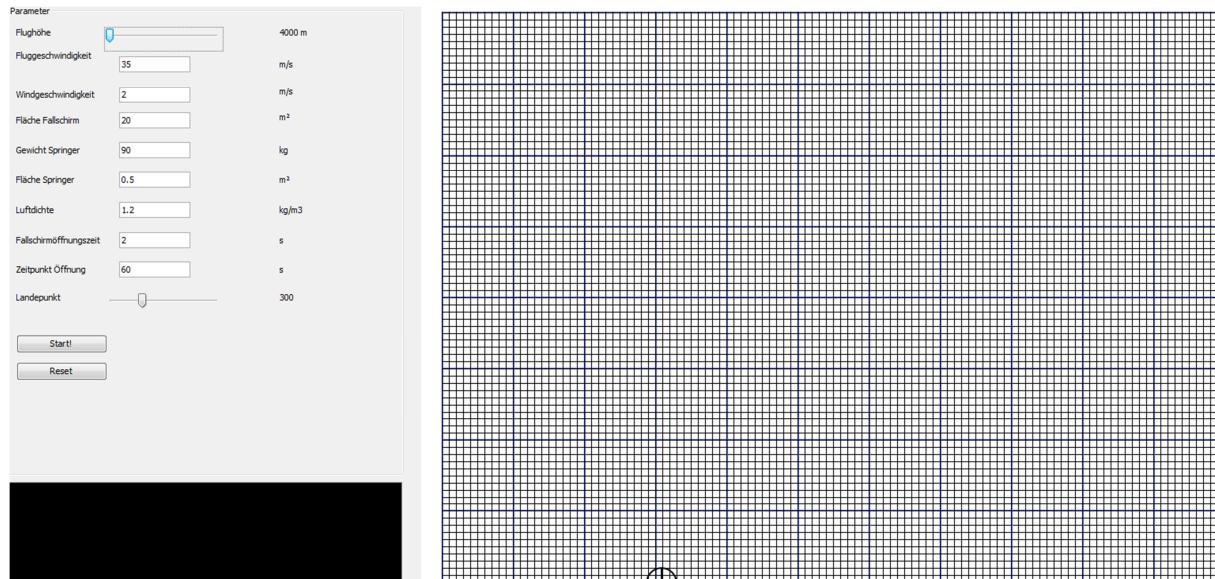


Abbildung 2: GUI mit Parameter

Unten links ist ein schwarzes Textfeld in der die mathematischen Berechnungen angezeigt werden. (Abb. 2)

Um zu verhindern, dass die Flughöhe und der Landepunkt des Springers den vorgesehenen Bereich verlassen, haben wir diese mit Reglern eingegrenzt. Mehr zu den Parameter-Werten in Kapitel 4.2.2.

3.9.2 Flugzeug

Um das Flugzeug zu bewegen, mussten wir zunächst die Flughöhe, die als Parameter festgesetzt wird, auf den Koordinatenbereich skalieren.

Auszug aus ChuteRunnable.java

```
double coordinatePerDot = (double) 700 / (double) 5000;
double yCoord = (700 - coordinatePerDot * sim.getAltitude());
double result[][] = this.sim.getResult();
```

Da das Koordinatenfenster auf 700 Pixel festgesetzt wurde, haben wir diesen Wert als Basis für die Skalierung genommen. Danach mussten wir ausrechnen, wie viele Pixel per Koordinate abgeflogen werden mussten.

Somit konnten wir die Position des Flugzeugs auf dem GUI festlegen. Da die Flugbahn des Fliegers horizontal und immer mit der gleichen Geschwindigkeit verläuft, waren keine Berechnungen notwendig.

Für die Geschwindigkeit mussten wir das Ganze auch horizontal skalieren. Dafür haben wir eine Schleife angelegt, die folgendes durchführt:

Auszug aus ChuteRunnable.java

```
planeLocation.setLocation((i * this.sim.getPlaneSpeed() / 1000), yCoord);
```

Damit bewegt sich das Flugzeug mit der korrekten Geschwindigkeit und in der korrekten Höhe.

3.9.3 Springer

Die Flugbahn des Springers kommt mit der Resultate-Tabelle aus den Berechnungen zurück. Die Skalierung funktioniert ähnlich wie beim Flugzeug:

Auszug aus ChuteRunnableTwo.java

```
    }
    jumperLocation.setLocation((result[i][0] / 10), 700 - coordinatePerDot * result[i][1]);
    moveJumper(jumperLocation);
```

Der Unterschied dabei ist, dass die Position nicht von der Flugzeuggeschwindigkeit abhängt, sondern von den Berechnungen.

4 Software (SM)

4.1 Grundstruktur

Die einzelnen Bereiche der Software sind getrennt. Wir haben eine grafische Benutzeroberfläche (graphical user interface, GUI), auf der die Parameter für die Simulation gesetzt werden. Die Werte werden dann an die Berechnungslogik weitergegeben. Dort wird die Flugbahn des Springers berechnet. Die Flugbahn wird als zeitabhängige Tabelle (Array) an das GUI zurückgegeben, welche dann zwei Threads startet, einen für die Flugzeugbewegung und einen für die Springerbewegung.

4.2 Grafische Benutzeroberfläche (GUI)

4.2.1 Grundgerüst

Auszug aus SimuchuteView.java

```
/*
 * Das Hauptfenster der Applikation.
 * @author Severin Mueller
 */
public class SimuchuteView extends FrameView {
```

Das GUI wird in der Klasse SimuchuteView definiert, die die Klasse Frameview erweitert. Die Klasse Frameview wird von der Entwicklungsumgebung zur Verfügung gestellt. Zunächst werden alle GUI Elemente definiert dann konstruiert.

Auszug aus der *initComponents()* aus SimuchuteView.java

```
landingPointValueLabel.setText(resourceMap.getString("landingPointValueLabel.text"));
landingPointValueLabel.setName("landingPointValueLabel"); // NOI18N

javax.swing.GroupLayout jPanel1Layout = new javax.swing.GroupLayout(jPanel1);
jPanel1.setLayout(jPanel1Layout);
jPanel1Layout.setHorizontalGroup(
    jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
    .addGroup(jPanel1Layout.createSequentialGroup()
        .addContainerGap()
```

Wird das Programm gestartet, können zunächst die entsprechenden Werte eingesetzt werden (Abb. 3):

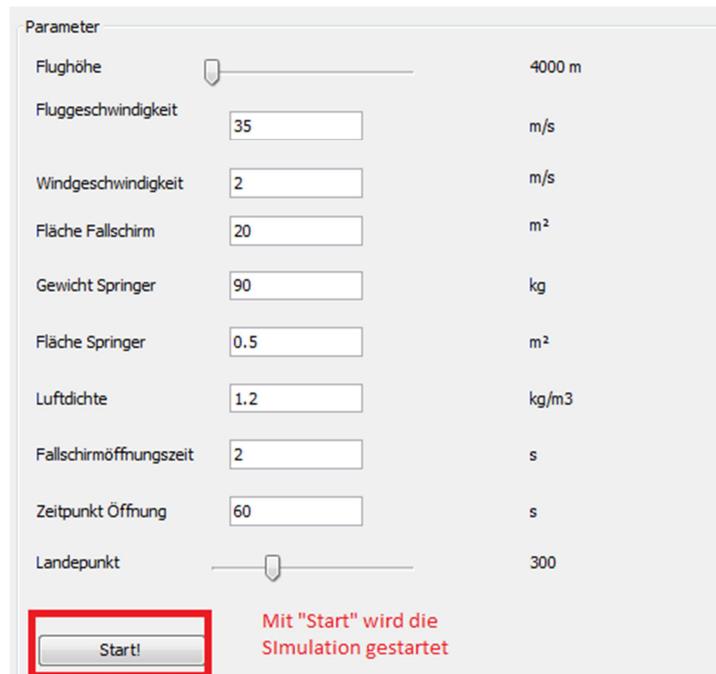


Abbildung 3: Parameter im GUI

4.2.2 Validierungen

Um sicherzustellen, dass nur sinnvolle Werte übertragen werden führen wir beim Starten eine Validierung durch.

Auszug aus den Validierungen in SimuchuteView.java

```
// Validierungen, damit nur korrekte Werte übergeben werden
if (!Tools.isDouble(view.flightSpeedValue.getText())) {
    JOptionPane.showMessageDialog(null, "Der Wert im Feld Fluggeschwindigkeit muss eine Zahl sein", "Fehler", JOptionPane.ERROR_MESSAGE);
    return null;
}
if (!Tools.isDouble(view.airSpeedValue.getText())) {
    JOptionPane.showMessageDialog(null, "Der Wert im Feld Windgeschwindigkeit muss eine Zahl sein", "Fehler", JOptionPane.ERROR_MESSAGE);
    return null;
}
if (!Tools.isDouble(view.parachuteAreaValue.getText())) {
    JOptionPane.showMessageDialog(null, "Der Wert im Feld Fallschirmfläche muss eine Zahl sein", "Fehler", JOptionPane.ERROR_MESSAGE);
    return null;
}
```

Für das korrekte Zahlenformat haben wir eine Helfer-Klasse *Tools* mit diversen Überprüfungsmethoden erstellt.

Die Methode *isInteger()* in Tools.java

```
public static boolean isInteger(String input) {
    try {
        Integer.parseInt(input);
        return true;
    } catch (Exception e) {
        return false;
    }
}
```

Wird ein Wert eingesetzt, der keinen Sinn macht, z.B. Buchstaben in einem Feld, in dem nur Zahlen erwartet werden, gibt das Programm eine Fehlermeldung zurück (Abb. 4):



Abbildung 4: Validierung Fehlermeldung

4.3 Kapselung der Parameter

Wie bereits im Kapitel 3.1 erwähnt, werden die Parameter des GUI als Objekt der Klasse `SimulationObject` gespeichert, damit mit den Werten gerechnet werden kann.

Auszug aus `doInBackground()` in `SimuchuteView.java`

```
SimulationObject sim = new SimulationObject();
sim.setAltitude(view.altitudeValue.getValue());
sim.setParachuteArea(new Double(view.parachuteAreaValue.getText()));
sim.setPlaneSpeed(new Double(view.flightSpeedValue.getText()));
sim.setWindSpeed(new Double(view.airSpeedValue.getText()));
sim.setLuftDichte(new Double(view.jumperAreaValue.getText()));
sim.setSpringerGewicht(new Integer(view.jumperWeightValue.getText()));
sim.setParachuteTimeToOpen(new Integer(view.timeToOpenValue.getText()));
sim.setSpringerFlaeche(new Double(view.jumperAreaValue.getText()));
sim.setLuftDichte(new Double(view.airDensityValue.getText()));
sim.setTOffen(new Double(view.timeWhenToOpenValue.getText()));
sim.setLandepunkt(new Double(view.landingPointValue.getValue() * 10));
// ...
```

Nun können die Werte an die Berechnungsroutine weitergereicht werden.

Auszug aus `doInBackground()` in `SimuchuteView.java`

```
Springer springer = new Springer(sim);
// Flugbahn etc berechnen
springer.calcSpringer();
// Object mit den berechneten Werten abholen,
sim = springer.getFlugbahnSpringer();
```

4.4 Problem: Blockieren des GUI

Standardmässig laufen die Benutzeroberflächenelemente auf dem Event Dispatch Thread (EDT)¹⁴. Werden weitere Routinen gestartet kommen sich die GUI-Elemente und die weiteren Programmabschnitte in die Quere und das GUI blockiert, weil es zunächst die Abarbeitung der weiteren Schritte abwarten muss.

Um dieses Problem zu lösen, haben wir einen sogenannten Hintergrund-Task eingebaut, auf dem alle weiteren Schritte laufen. Realisiert haben wir dies mit einer inneren Klasse *StartSimulationTask*.

Innere Klasse *StartSimulationTask* in SimuchuteView.java

```
private class StartSimulationTask extends org.jdesktop.application.Task<Object, Void> {

    private org.jdesktop.application.Application backupApp;
    private final SimuchuteView view;

    StartSimulationTask(org.jdesktop.application.Application app, SimuchuteView view) {
        super(app);
        backupApp = app;
        this.view = view;
    }
    // Wir verschieben die Simulation in den Hintergrund, damit die Berechnungen nicht auf dem Event Dispatch Thread laufen

    @Override
    protected Object doInBackground() {
        ...
    }
}
```

So können wir das Programm anweisen, das GUI auf dem EDT laufen zu lassen, während die weiteren Schritte in einem Hintergrundtask laufen.

4.5 Threads

Wenn die Flugbahn des Fallschirmspringers berechnet wurde, müssen wir die Simulation grafisch darstellen. Die Herausforderung hier war, dass wir das Flugzeug und den Springer gleichzeitig und unabhängig voneinander bewegen. Die Lösung heisst Threads. Wir haben für diesen Zweck zwei Klassen erstellt, die das Java-Interface *Runnable* implementieren. Der ganze Bewegungsablauf der zwei Objekte findet in diesen Klassen statt.

Die Klasse ChuteRunnable – Thread für das Bewegen des Flugzeugs

```
public class ChuteRunnable implements Runnable {

    private SimuchuteView view;
    private SimulationObject sim;

    public ChuteRunnable(SimuchuteView view, SimulationObject sim) {
        this.view = view;
        this.sim = sim;
    }
}
```

¹⁴ Oracle [2012]

Die Klasse ChuteRunnableTwo – Thread für das Bewegen des Springers

```
/**  
 * Zweite Runnable Klasse für den Springer-Thread  
 * @author Fish-Guts  
 */  
public class ChuteRunnableTwo implements Runnable {  
  
    private SimuchuteView view;  
    private SimulationObject sim;  
  
    public ChuteRunnableTwo(SimuchuteView view, SimulationObject sim) {  
        this.view = view;  
        this.sim = sim;  
    }  
}
```

Dadurch, dass wir alle Werte in einem Objekt der Klasse `SimulationObject` gespeichert haben, sind diese über den Konstruktor verfügbar und die Werte können nun verwendet werden. Weil die Threads unabhängig voneinander laufen, können wir zwei Objekte gleichzeitig bewegen lassen:

Auszug au `run()` Funktion in `ChuteRunnableTwo.java`

```
public void run() {  
    Point jumperLocation = new Point();  
    boolean changed = false;  
    double coordinatePerDot = (double) 700 / (double) 5000;  
    double yCoord = (700 - coordinatePerDot * sim.getAltitude());  
    // Startkoordinaten von Flugzeug
```

Mit der `run()` Methode werden die jeweiligen Threads gestartet.

5 Simulation - Kurze Flugzeit

Mit Hilfe der Software können wir den Absprungpunkt und die Zeit des Öffnens durch ausprobieren bestimmen.

Dazu haben wir ein paar Versuche gemacht. Das Ziel in dieser Simulation ist es, die kürzeste Flugzeit zu erreichen. Dass heisst, der Fallschirm wird möglichst spät geöffnet. Wichtig ist dabei, dass der Fallschirmspringer bei der Landung keine grosse Geschwindigkeit mehr hat, da er sonst den Aufprall nicht überleben würde. In der Realität landen Fallschirmspringer mit 2 m/s bis 5 m/s , wobei eine 5 m/s bereits eine harte Landung ist. Ab einer Geschwindigkeit höher als 5 m/s muss bereits mit Verletzungen gerechnet werden. Die Parameter für den ersten Versuch sind unten aufgelistet. (Tab. 2)

Tabelle 2: Simulation - Versuch 1

Simulation – Versuch 1			
Parameter		Resultate	
Flughöhe	4000 m	Flugzeit	61.9 s
Flugzeuggeschwindigkeit	20 m/s	Absprungpunkt	-123 m
Fläche Fallschirm	20 m ²	Geschwindigkeit bei Landung	-4.95 m/s
Gewicht Springer mit Ausrüstung	90 kg		
Gewünschter Landepunkt	300 m		
Zeitpunkt Fallschirm öffnen	68 s		

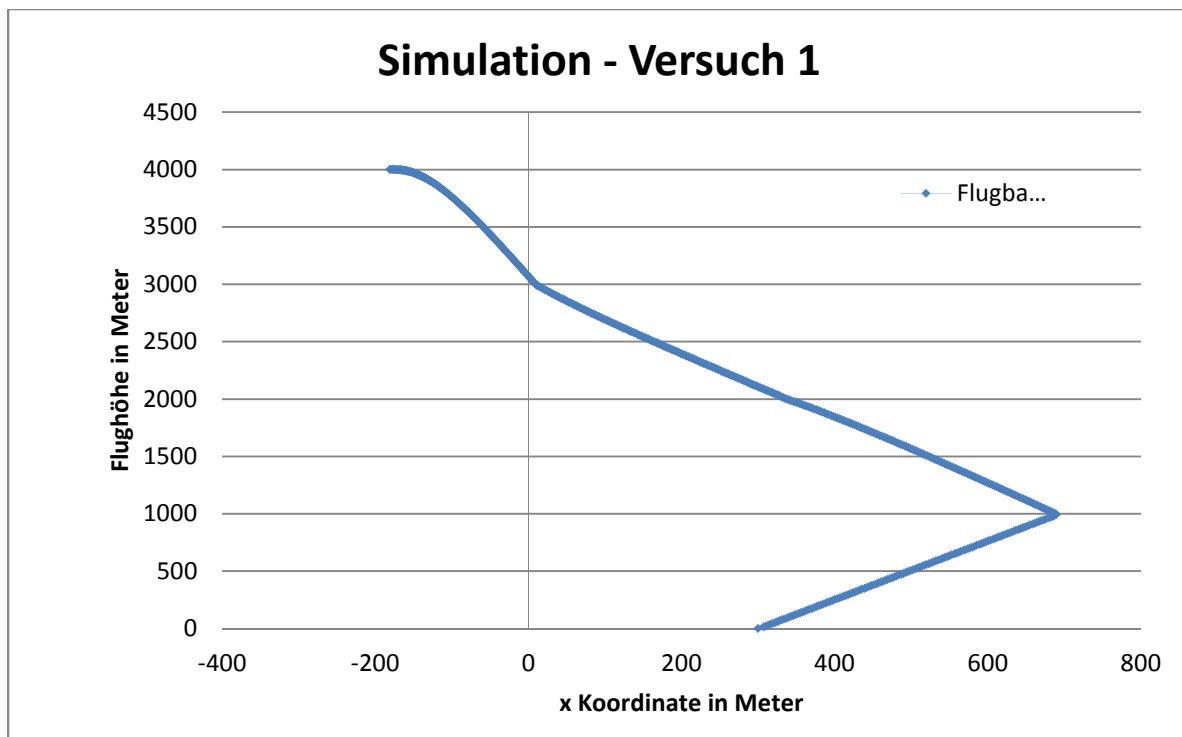


Abbildung 5: Simulation - Versuch 1

Da die Flugzeit gemäss Tab. 2 nur 61.9 Sekunden beträgt, kommt der Springer gar nicht zum Öffnen des Fallschirms, wodurch er mit voller Geschwindigkeit aufprallt. Was in der Abb. 5 gut ersichtlich ist, da die Steigung der Flugbahn bis zum Aufprall sehr steil ist. Dies wollen wir natürlich verhindern und ändern daher die Parameter für den zweiten Versuch (Tab. 3). Der Fallschirm benötigt zwei Sekunden um sich vollständig zu öffnen, wir versuchen es daher einmal mit 60 Sekunden.

Tabelle 3: Simulation - Versuch 2

Simulation – Versuch 2			
Parameter		Resultate	
Flughöhe	4000 m	Flugzeit	63.8 s
Flugzeuggeschwindigkeit	20 m/s	Absprungpunkt	-123 m
Fläche Fallschirm	20 m ²	Geschwindigkeit bei Landung	-4.95 m/s
Gewicht Springer mit Ausrüstung	90 kg		
Gewünschter Landepunkt	300 m		
Zeitpunkt Fallschirm öffnen	60 s		

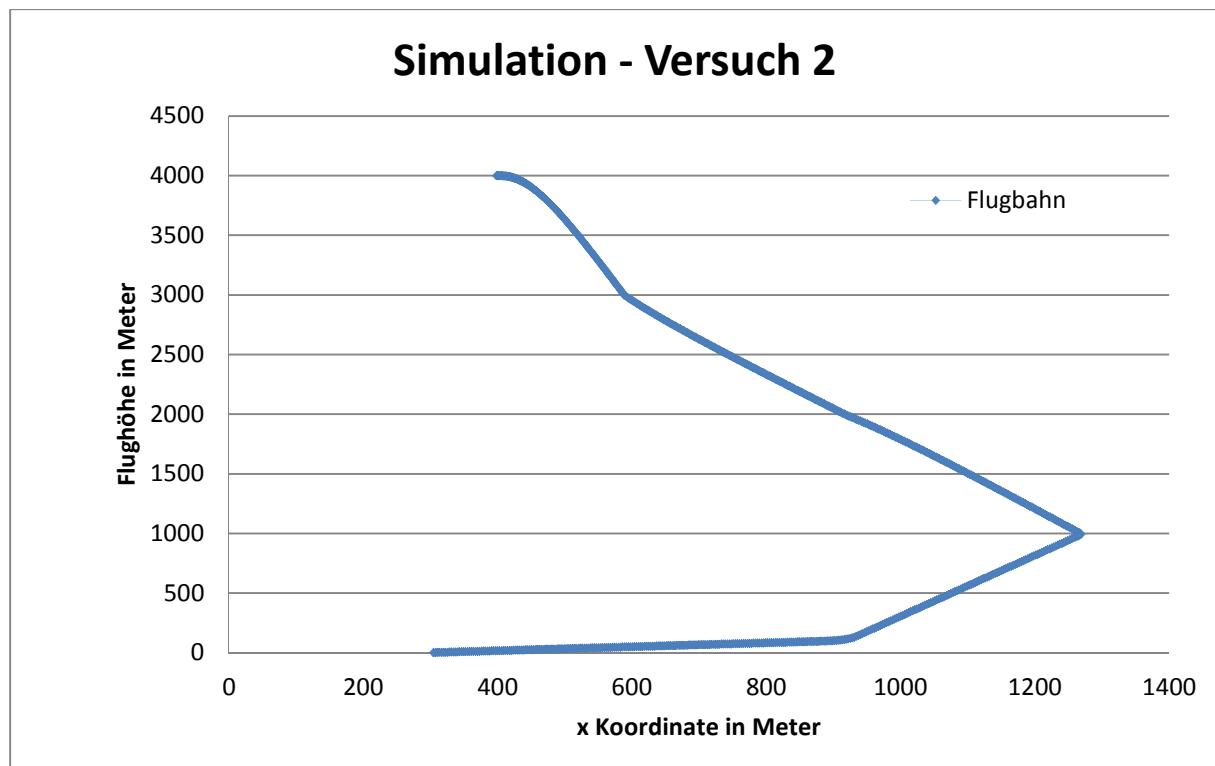


Abbildung 6: Simulation - Versuch 2

Diese Simulation ist schon besser (Abb. 6). Der Springer überlebt und kann eine Flugzeit von 81.2 Sekunden geniessen. Wir wollen nun aber noch mehr Risiko eingehen und öffnen den Fallschirm erst bei 61.2 Sekunden. (Tab. 4)

Tabelle 4: Simulation - Versuch 3

Simulation – Versuch 3			
Parameter		Resultate	
Flughöhe	4000 m	Flugzeit	63.8 s
Flugzeuggeschwindigkeit	20 m/s	Absprungpunkt	-123 m
Fläche Fallschirm	20 m ²	Geschwindigkeit bei Landung	-5.01 m/s
Gewicht Springer mit Ausrüstung	90kg		
Gewünschter Landepunkt	300 m		
Zeitpunkt Fallschirm öffnen	61.2 s		

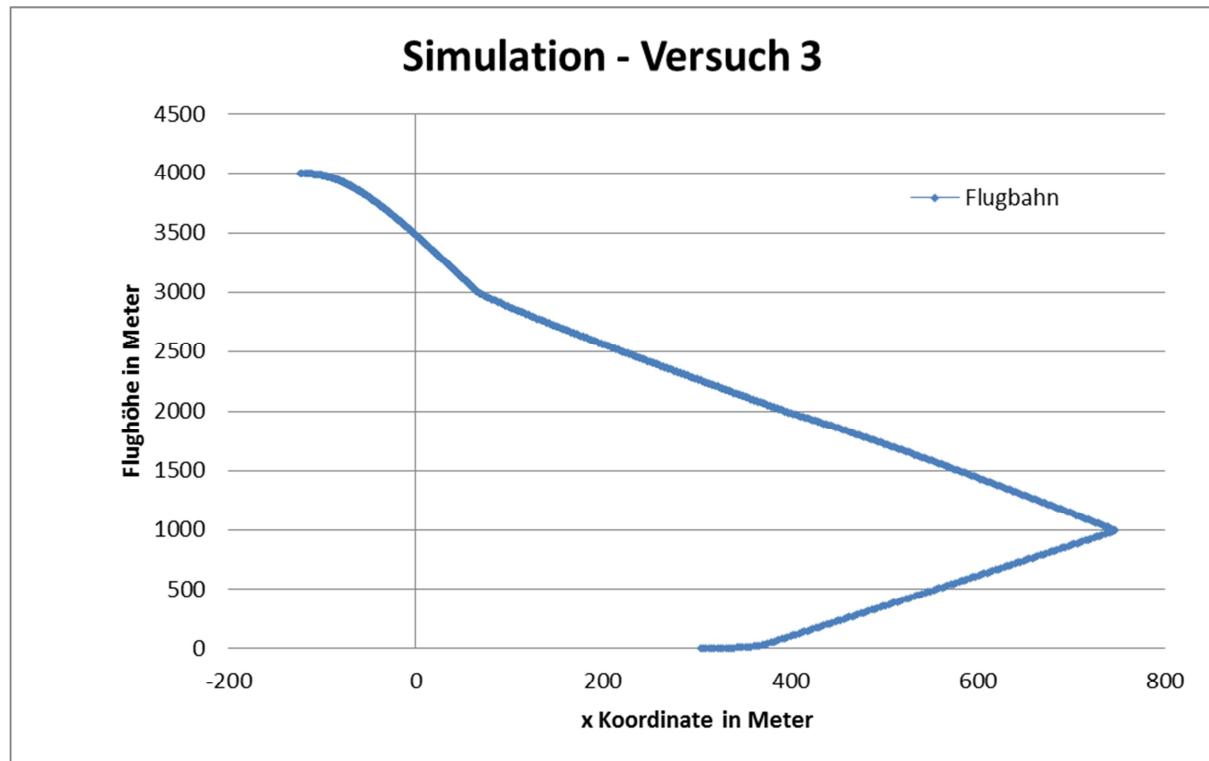


Abbildung 7: Simulation - Versuch 3

Und es funktioniert (Abb. 7). Theoretisch könnte der Springer den Fallschirm erst bei 61.2 Sekunden (Tab. 4) öffnen und er würde überleben. Durch die späte Öffnung des Fallschirms wird eine sehr kurze Flugzeit erreicht. Wir könnten nun versuchen die Parameter zu optimieren. Beim Öffnen bei 61.3 Sekunden haben wir dann aber eine Geschwindigkeit von 9.7m/s und bei 61.4 Sekunden bereits 25m/s. In realen Bedingungen sollte man daher nicht an die Grenzen gehen und eher ein paar Sekunden früher öffnen, denn nur wenige Zehntelsekunden können bereits über Leben und Tod entscheiden.

6 Fazit

6.1 Sevi

Dieses Projekt brachte einige neue Herausforderungen, denen wir uns stellen mussten. Ich habe viel über die GUI-Programmierung gelernt. Auch Threads waren absolutes Neuland. Der gesetzte zeitliche Rahmen schien realistisch zu sein, zumal wir ziemlich pünktlich mit dem Projekt fertig geworden sind.

Es war spannend zu erleben, wie man auch mal Mathematische Grundlagen in einer Software anwenden konnte, dies hat den Spassfaktor sicherlich erhöht.

6.2 Patrice

Ich habe mich in diesem Projekt intensiv mit dem Ablauf eines Fallschirmsprungs und dessen Berechnung beschäftigt. Das zusammentragen der Informationen und entsprechenden Gleichungen haben wir sehr grossen Spass gemacht. Mein grösstes Problem war die Berechnung der Ableitungen in einem zweidimensionalen Umfeld, jedoch konnte ich die Problemstellung mit Hilfe von Herrn Heuberger und entsprechenden Unterlagen lösen. Die Simulationen zum Schluss waren sehr interessant, vor allem da sich die Flugbahn extrem verändert hat, sobald man an den Parameter etwas verändert hat.

6.3 Danksagung

Besonderer Dank möchten wir an Herrn Albert Heuberger aussprechen. Er hat uns in mehreren Meetings die Zeit genommen, uns die nummerischen und mathematische Probleme zu erklären.

Vielen Dank ebenfalls an Herrn Lukas Eppler, welcher uns als Scrum Master begleitet und uns in den Iterationsmeetings projekttechnisch unterstützt hat.

7 Anhang

7.1 Quellenverzeichnis

Bärwolff, Günter. 2007. *Numerik für Ingenieure, Physiker und Informatiker*. Universität Berlin : s.n., 2007.

Grassl, Florian. www.fsr-club2000.de. [Online] [Zitat vom: 10. 06 2012.] http://www.fsr-club2000.de/ausbildung/Florian_Grassl_Facharbeit_Freier_Fall.pdf.

Halliday, David. 2007. *Halliday Physik*. 2007. ISBN: 978-3-527-40746-0.

Kirchgraber, U. 1993. www.educ.ethz.ch. [Online] 1993. [Zitat vom: 30. Mai 2012.]
<http://www.educ.ethz.ch/unt/um/mathe/gb/Fallschirmspringer.pdf>.

Knorrenchild, Michael. 2008. *Nummerische Mathematik*. 2008. ISBN: 978-3446412613.

Massjung, R. 2012. [Online] 22. Mai 2012. [Zitat vom: 15. Juni 2012.]
<http://elearning.zhaw.ch/moodle/mod/resource/view.php?id=276205>.

Maxim. 2010. www.virtual-maxim.de/. *Virtual Maxim*. [Online] 2010. [Zitat vom: 25. Mai 2012.]
<http://www.virtual-maxim.de/downloads/freier%20fall%20mit%20und%20ohne%20luftwiderstand.pdf>.

Nahlik, Philippe. 2012. [Online] 10. Februar 2012. [Zitat vom: 15. Juni 2012.]
http://elearning.zhaw.ch/moodle/file.php/5821/Dokumente/Konzept_SoftwareProjekte_2_Studenten.pdf.

Oracle. [Online] <http://docs.oracle.com/javase/tutorial/essential/concurrency/>.

Sierra, Kathy & Bates, Bert. *Java von Kopf bis Fuss*. ISBN: 978-3-89721-448-4.

Ullenboom, Christian. 2009. *Java ist auch eine Insel*. 2009. ISBN: 978-3-8362-1371-4.

Wikipedia. [Online] [Zitat vom: 10. 06 2012.] <http://de.wikipedia.org/wiki/Fallschirmspringen>.

7.2 Abbildungsverzeichnis

Abbildung 1: Runge-Kutta Verfahren	13
Abbildung 2: GUI mit Parameter	20
Abbildung 3: Parameter im GUI	23
Abbildung 4: Validierung Fehlermeldung.....	24
Abbildung 5: Simulation - Versuch 1	27
Abbildung 6: Simulation - Versuch 2	28

7.3 Tabellenverzeichnis

Tabelle 1 Runge Kutta Parameter	17
Tabelle 2: Simulation - Versuch 1.....	27
Tabelle 3: Simulation - Versuch 2.....	28
Tabelle 4: Simulation - Versuch 3.....	29