1.1: HELLO SPACE

Zu berechnen, wie sich Gegenstände unter dem Einfluss physikalischer Kräfte bewegen kann sehr komplex sein. Das hast du wahrscheinlich schon im Physikunterricht festgestellt. Wirken nur einige wenige Kräfte wie zum Beispiel die Schwerkraft und Federkraft an einem Objekt, kann man mathematisch exakte Lösungen finden. Aber gerade wenn viele verschiedene Kräfte auf ein Objekt einwirken, stösst man schnell an die Grenzen von dem was man exakt berechnen kann.

In dieser Unterrichtseinheit werden wir entdecken, wie wir den Computer dazu benutzen können, um solche Probleme approximativ zu Lösen. Wir teilen dafür die Bewegung in eine Folge von kleine Schritte auf, und schauen uns in jedem Schritt an, was gerade für Kräfte auf unseren Gegenstand wirken, und wo sie diesen hinziehen.

Wir verwenden dazu eine Methode, die zwar einfach, aber in der Wissenschaft und Industrie weit verbreitet ist. Man findet sie überall, von der Spielentwicklung bis zur Simulation von Sternensystemen. Wir werden auch darauf eingehen, wann diese Methode fehlschlägt und was wir dagegen tun können.

Wir werden uns langsam an diese Methode herantasten, indem wir die Fortbewegung vom kleinen Planeten Pluto physikalisch simulieren. Dabei fangen wir ganz einfach (und etwas physikalisch inkorrekt) an, und nähern uns dann Schritt für Schritt der physikalischen Wirklichkeit.

Das Ziel ist dabei, dass du am Ende selbstständig Probleme aus dem Physikunterricht mit dem Computer nachbilden und simulieren kannst und ein besseres Verständnis dafür gewinnst, wie Gegenstände durch Kräfte beeinflusst werden.

Und hier kommt schon eine erste kleine Aufgabe:

> **Aufgabe 1:** Navigation im Weltall-Koordinatensystem
>
> Bewege das XY-Weltall-Koordinatensystem auf der rechten Seite, indem du es mit der Maus ziehst. Bewege den Mauszeiger auf Pluto, um seine Position anzuzeigen. Wenn du seine Position dauerhaft anzeigen willst, klicke mit der Maus auf Pluto.

Wenn du dich genug mit dem Koordinatensystem vertraut gemacht hast, drücke oben auf `>` um zum nächsten Level zu gelangen!

1.2: ERSTE SCHRITTE

Fangen wir ganz einfach und physikalisch inkorrekt an. Wir können Pluto im XY-Weltall-Koordinatensystem bewegen, indem wir ihn Schritt für Schritt um einen kleinen Vektor verschieben.

Wie du in der letzten Aufgabe bereits gesehen hast, kannst du die aktuelle Position von Pluto anzeigen, indem du auf ihn mit der Maus klickst. Anstelle der in der Physik üblichen vertikal geschriebenen Vektoren benutzen wir eine normale Python-Liste um die Position `x = [0.0, 0.0]` darzustellen, wobei das erste Element die X-Koordinate und das zweite die Y-Koordinate ist.

Wie bewegen wir nun Pluto? Bei physikalischen Simulationen macht man von sogenannten *Schrittfunktionen* Gebrauch, um den alten Zustand eines Objektes auf einen neuen zu aktualisieren. In unserem Fall ist der Zustand einfach die Position `x` von Pluto.

Die Schrittfunktion `step` im rechten Fenster nimmt die alte Position `x` als Argument, berechnet dann die aktualisierte Position im Körper der Funktion indem sie zur X-koordinate `1` addiert, und gibt die neue Position am Ende zurück.

Der Knopf ⏭ führt die Schrittfunktion einmal aus und aktualisiert mit dem Rückgabewert die Position von Pluto.

✎ **Aufgabe 1:** Codeinterpretation

Führe die Funktion ein paar mal aus, und beschreibe in deinen eigenen Worten, wie sich Pluto bewegt.
Du kannst ↺ benutzen, um Pluto zum Ursprungszustand zurückzusetzen.

So weit so gut. Nun ist es deine Aufgabe, die Bewegung zu verändern:

✎ **Aufgabe 2:** Schritt für Schritt

Passe den Code im rechten Fenster so an, dass Pluto sich in jedem Schritt neu um den Vektor `[2,1]` bewegt. Drücke dann auf ⏭ um deine Implementation zu testen.
Wenn du sicher bist, das sie richtig funktioniert, drücke auf ✓ , um zu überprüfen, ob deine Implementierung mit der Lösung übereinstimmt.

✎ **Aufgabe 3:** Geschwindigkeit

Wenn du ⏭ fünf mal pro Sekunde drückst, mit welchem Geschwindigkeitsvektor bewegt sich Pluto dann?

Hat alles geklappt? Dann auf zum nächsten Level!

1.3: ERSTE SIMULATION

Auf der vorherigen Seite haben wir Pluto zum ersten mal Schritt für Schritt bewegt. Wir möchten aber natürlich nicht dauernd ⏭ drücken müssen, um eine gleichmässige Bewegung zu simulieren. Anstatt die Schrittfunktion also immer manuell auszuführen, wollen wir dies automatisch tun. Das ist genau was der Knopf ▷ rechts macht. Er führt `step` pro Sekunde 5 mal aus.

✎ **Aufgabe 1:** Änderung der Geschwindigkeit

Mit welchem Geschwindigkeitsvektor bewegt sich Pluto, wenn du auf ▷ drückst? Passe die Schrittfunktion so an, dass Pluto sich mit einem Geschwindigkeitsvektor von `[5,-2.5]`m/s bewegt. Du kannst die Zeit, die in der Simulation abgelaufen ist im Fenster neben ↺ ablesen.

Das ist unsere erste selbständig laufende Simulation von Pluto! Sie stockt allerdings noch etwas.

Aufgabe 2: Stockende Simulation

Weshalb bewegt sich Pluto in der Simulation nicht flüssig? Was könnte man tun, um das zu ändern?

Auf der nächsten Seite werden wir sehen, wie wir das beheben können.

1.4: FLÜSSIGE BEWEGUNG

Wenn wir momentan die Simulation ausführen, indem wir auf ▷ drücken, bewegt sich Pluto zwar mit der erwarteten Geschwindigkeit von `[5,-2.5]`m/s, das ganze stockt dabei aber etwas, da die Schrittfunktion nur alle 0.2 Sekunden ausgeführt wird.

Aufgabe 1: Änderung der Zeitschritte

Mit dem Regler unten kannst du anpassen, wie lange zwischen zwei aufrufen der Schrittfunktion gewartet wird. Wie verändert sich die Geschwindigkeit von Pluto in Abhängigkeit der gewarteten Zeit, dem sogenannten *Zeitschritt* `dt`? Warum ist das so?

Wie du in der Aufgabe oben gesehen hast, wird die Bewegung von Pluto zwar flüssiger, wenn man `dt` kleiner wählt, aber Pluto bewegt sich dann auch schneller als `[5,-2.5]`m/s. Das Problem ist dabei, das die Schrittfunktion noch unabhängig von `dt` ist. Deine Aufgabe ist es nun, das zu ändern:

Aufgabe 2: Simulation unabhängig von dt

Passe die Schrittfunktion so an, dass Pluto sich immer mit `[5,-2.5]`m/s bewegt, egal wie groß `dt` ist.

Tipp

Überlege dir, wie du vom neuen Argument `dt` der Schrittfunktion Gebrauch machen kannst.

Tipp

Pluto sollte sich immer gleichschnell bewegen, unabhängig davon wie du `dt` im Regler oben einstellst.

Tipp

Wie müsstest du die Schrittfunktion anpassen, wenn `dt = 0.4` wäre? Wie wenn `dt = 0.1` wäre? Wie kannst du das auf ein beliebiges `dt` verallgemeinern?

Wie du siehst, können wir eine flüssige, kontinuierliche Bewegung mit diskreten Schritten simulieren, indem wir den Zeitschritt einfach klein genug wählen, sodass man die einzelnen Schritte nicht mehr wahrnimmt.

Wieso wählen wir den Zeitschritt nicht einfach immer `dt = 0.01`? Zum einen kann es sehr rechenintensiv sein, wenn wir eine Komplizierte Schrittfunktion sehr oft in der Sekunde ausführen müssen. Zum anderen

werden wir in Kapitel 2 sehen, das die Korrektheit einer Simulation auch vom Zeitschritt abhängig sein kann.

Wenn du alle Aufgabe bis jetzt erfolgreich abgeschlossen hast, hast du schon einen guten Einblick in die Funktionsweise von Schrittfunktionen bekommen. Wir werden in den nächsten Kapiteln noch viel mehr mit Schrittfunktionen arbeiten, aber nichts grundlegendes mehr an der Funktionsweise ändern. Wir werden aber sehen, wie wir diese Funktionsweise nutzen können, um jede Menge andere physikalische Konzepte simulieren zu können!

2.1: KONSTANTE BESCHLEUNIGUNG

Im letzten Level haben wir es geschafft, Pluto Schritt für Schritt mit einem Diskretisationsschritt Δt mit konstanter Geschwindigkeit in eine Richtung zu bewegen. Für dieses Kapitel machen wir einen Abstecher weg vom Weltraum zurück zu uns auf der Erde.

Wie du weist, übt die Erde auf alle Objekte darauf eine Gravitationskraft aus. Die Gravitationskraft auf der Erdoberfläche beträgt ca. $9.81m/s^2$. Wenn wir zum Beispiel einen Fussball aus dem 5. Stock werfen, dann fällt er nicht mit gleichmässiger Geschwindigkeit Richtung Boden sondern beschleunigt sich auf dem Weg nach unten. Somit vergrössert sich die Geschwindigkeit je länger sich der Fussball im freien Fall befindet.

Im Editor auf der rechten Seite siehst du bereits ein Beispiel, in dem der Fussball mit einer Beschleunigung von $1m/s^2$ nach rechts beschleunigt wird.

> **Aufgabe 1:** Geschwindigkeit und Position des Fussballes
>
> Die Geschwindigkeit und die X-Koordinaten der Position des Fussballes sind zu Beginn des Levels bei 0. Was betragen die Geschwindigkeit und die Position des Fussballes nach dem ersten Schritt? Was betragen sie nach dem zehnten Schritt? Berechne sowohl die Geschwindigkeit und die Position des Fussballes von Hand für die ersten zehn Schritte mit einen Diskretisationsnschritt von $\Delta t = 0.01s$ und einer Beschleunigung von $1m/s^2$ in die X-Richtung wie im Beispiel. Zeichne dann sowohl die Geschwindigkeit als auch die Position in der X-Richtung in ein Koordinatensystem ein wobei die X-Achse die Zeit und die Y-Achse die Geschwindigkeit bzw. die Position sein soll. Was stellst du fest? Wie verändern sich die Geschwindigkeit und die Position in Abhängigkeit der Zeit? Überprüfe dann deine Lösungen mithilfe der Konsolenausgabe in der Simulation.

Deine eigentliche Aufgabe in diesem Level ist es, den Fussball, der vom 5. Stock ($20m$ über dem Boden) fallengelassen wird, zu simulieren. Dafür reicht es nun nicht mehr nur die Position in jedem Schritt zu aktualisieren sondern es muss auch die Geschwindigkeit aktualisiert werden.

> **Aufgabe 2:** Zeit bis zum Aufprall
>
> Versuche von Hand zu berechnen, wie lange es dauert bis der Fussball aus dem Stillstand von $20m$ auf dem Boden aufprallt. Um deine Berechnungen zu vereinfachen kannst du annehmen, dass die Erdbeschleunigung $10m/s^2$ beträgt.

Aufgabe 3: Simulation des freien Falles

Nun kannst du deine Berechnung der letzten Aufgabe mit Hilfe der Simulation überprüfen (verwende in der Simulation die Erdbeschleunigung von $9.81m/s^2$). Aktualisiere dazu in der Schrittfunktion zuerst die Geschwindigkeit (v) (wie im Beispiel) bevor du wie im letzten Kapitel die Position aktualisierst. Die Simulation wird automatisch stoppen, sobald der Fussball den Boden erreicht hat und du kannst die Zeit, die bis dahin vergangen ist im Feld neben dem Play Knopf ablesen. Vergleiche das Resultat der Simulation mit deinem berechneten. Was stellst du fest?

Um das Level abzuschliessen, drücke auf den Validierungsknopf ✓ um deine Lösung mit der Musterlösung zu vergleichen.

2.2: SPRINGENDER BALL

Der Fussball aus der letzten Aufgabe hat sich nach dem Aufprall auf dem Boden nicht mehr bewegt so als ob er am Boden festgeklebt worden wäre. In der Realität verhält sich ein Fussball jedoch ganz anders. Nach dem Aufprall prallt er vom Boden ab und hüpft dann mehrmals bis er zum Stillstand kommt. In diesem Level möchten wir genau dieses Verhalten in unsere Simulation miteinbauen.

Das Abprallen vom Boden nennt man in der Physik einen **elastischen Stoss**. Dabei wird die ganze kinetische Energie, welche der Ball während des freien Falles angesammelt hat in elastische Energie umgewandelt. Mit anderen Worten, der Ball verformt sich während des Aufprall wie eine Feder und gibt diese Energie dann wieder in Form von kinetischer Energie ab und fliegt in die entgegengesetzte Richtung davon. In unserer Simulation nehmen wir an, dass der Deformationsprozess unglaublich schnell passiert und der Ball direkt wieder mit entgegengesetzter Geschwindigkeit davon fliegt.

Aufgabe 1: Aufprall detektieren

Versuche herauszufinden, wie wir den Aufprall innerhalb der Simulation detektieren können. Welche Kondition muss **x[1]** dazu erfüllen?

Aufgabe 2: Aufprall simulieren

Implementiere die Kondition der vorangehenden Aufgabe. Falls die Kondition erfüllt ist, invertiere die Geschwindigkeit **v** sodass der Ball in die entgegengesetzte Richtung davonfliegt. Simuliere dann deine Implementierung und sieh dir das Resultat an. Was sagt deine Intuition? Entspricht das Resultat der Realität?

Wahrscheinlich hast du bemerkt, dass der Ball im Vergleich zu Realität viel zu hoch springt. Der Ball erreicht nach jedem Abpraller wieder dieselbe Höhe von der er fallengelassen wurde. In der Realität verringert sich die Höhe nach jedem Abpraller bis der Ball auf dem Boden liegen bleibt.

Unsere jetzige Implementierung entspricht in der Physik einem **perfekten elastischen Stoss**. In der Realität gibt es jedoch keine **perfekte elastische Stösse**. Bei jedem Aufpraller wird ein Teil der elastischen Energie in

thermische Energie (Wärme) umgewandelt und nur ein Teil der elastischen Energie wird in kinetische Energie zurückgewandelt. Versuchen wir nun dieses Verhalten in unserer Simulation umzusetzen.

Aufgabe 3: Aufprall mit Energieverlust

In der letzten Aufgabe hast du die Geschwindigkeit des Balles invertiert. In der Realität geht aber immer ein Teil der kinetischen Energie verloren. Reduziere nun den Teil der Geschwindigkeit welche invertiert wird. Probiere verschiedene Werte aus (90%, 70%, 50%, 30%). Welcher Wert entspricht am ehesten deiner Intuition eines Abprallers des Fussballes auf einer asphaltierten Strasse? Auf einer Rasenfläche? Auf einer Wasseroberfläche?

Um das Level abzuschliessen, setze den Abprallkoefizient auf 70% und drücke auf den Validierungsknopf ✓ um deine Lösung mit der Musterlösung zu vergleichen.

2.3: SCHIEFER WURF

Bisher haben wir den Fussball nur aus dem Fenster geworfen. In dieser Aufgabe verwenden wir ihn nun so, für das er gemacht wurde, nähmlich zum kicken. Als erstes simulieren wir den Fussball, welcher in einem Winkel von 20° zum Boden mit einer Geschwindigkeit von $15m/s$ losgekickt wird.

Aufgabe 1: Flug des Fussballes

Vervollständige die Schrittfunktion sodass der Fussball fliegen kann. Du erhälst die Initialgeschwindigkeit $\vec{\mathbf{v}}$ bereits als Eingabe in die Schrittfunktion. Vergiss nicht die Geschwindigkeit zu aktualisieren - auf der Erde herscht nachwievor eine Gravitation von $9.81m/s^2$.

Den Flug des Fussballes, welcher du soeben als Simulation implementiert hast nennt man in der Physik **Schierfer Wurf**. Eine wichtige Frage welche man sich im Sport häufig stellt ist die Wurfweite in Abhängigkeit des Wurfwinkels. Im schwarzen Kasten unterhalb der Programmcodeeingabe siehst du jeweils, wie weit der Fussball gefolgen ist.

Aufgabe 2: Optimaler Wurfwinkel

Nun wollen wir herausfinden, mit welchem Wurfwinkel wir die Wurfdistanz maximiert werden kann. Versuche mit dem Slider unterhalb den Winkel anzupassen und finde den Winkel mit dem der Fussball am weitesten fliegt.

Aufgabe 3: Knobelaufgabe: Optimaler Wurfwinkel - Mathematischer Beweis

Versuche deine Erkentnisse von der vorangehenden Aufgabe mathematisch zu beweisen. Die Formel für den schiefen Wurf lautet

$$\vec{\mathbf{x}}\,(t) = \vec{\mathbf{x_0}} + \vec{\mathbf{v_0}}t - \frac{1}{2}\,\vec{\mathbf{g}}\,t^2 = \begin{pmatrix}0\\0\end{pmatrix} + \begin{pmatrix}cos(\alpha)\\sin(\alpha)\end{pmatrix}v_0 t - \frac{1}{2}\begin{pmatrix}0\\g\end{pmatrix}t^2$$

wobei $g = 9.81m/s^2, v_0 = 15m/s$ und α der Wurfwinkel sind. Stimmt deine Berechnung mit dem

Resultat aus der Simulation überein?

Tipp: Schritt 1

Damit der Fussball auf dem Boden aufschlägt muss die y-Komponente des Positionsvektors $\vec{\mathbf{x}}$ null sein. Mit dieser Information kannst du die Zeit des Aufpralles $t_{Aufprall}$ in Abhängigkeit von g, v_0 und α berechnen.

Tipp: Schritt 2

Setze nun die Zeit des Aufpralles $t_{Aufprall}$ in die Gleichung der x-Komponenten ein. Nun haben wir das Problem in ein Optimierungsproblem umgeformt. Wir müssen also das α finden, für das die x-Komponente maximiert wird.

Tipp: Schritt 3

Berechne nun die Ableitung der x-Komponente nach dem Wurfwinkel α. Um das Maximum zu finden muss die Ableitung null betragen.

Tipp: Schritt 4

Für das Auflösen der Gleichung der Ableitung hilft dir die geometrische Identität $sin(\alpha)^2 + cos(\alpha)^2 = 1$ (trigonometrischer Pythagoras).

2.4: ZEIT DISKRETISIERUNG

In diesem Level beschäftigen wir uns tiefer mit dem der Schrittzeitkonstante Δt. Was für einen Einfluss hat die Konstante auf das Simulationsresultat? Welcher Wert dieser Konstante gibt uns das beste Resultat? Für unsere Untersuchung der Schrittzeitkonstanten verwenden wir wieder den schiefen Wurf des letzten Levels.

Die grüne Parabel zeigt die analytische Lösung des schiefen Wurfes (die Lösung, welche uns die physikalische Formel gibt). Wir wollen nun herausfinden, wie genau unsere Simulation in Abhängigkeit der Schrittzeitkonstanten ist.

Aufgabe 1: Schrittzeitkonstante

Simuliere den schiefen Wurf mit verschiedenen Schrittzeitkonstanten (Slider unterhalb dieser Aufgabe). Welche Schrittzeitkonstante führt zum besten Resultat? Weshalb? Was sind die Nachteile dieser Schrittzeitkonstanten?

Die Wahl der Schrittzeitkonstanten hat anscheinend einen erheblichen Einfluss auf die Genauigkeit der Simulation. Aber was ist überhaupt der Vorteil der Simulation? Anscheinend können wir auch eine viel genauere Lösung direkt mit den physikalischen Gleichungen berechnen und es erfordert erst noch viel weniger

Rechenleistung.

Für diese simpleren Beispiele gibt es tatsächlich analytische Lösungen, welche sowohl genauer sind und weniger Rechenaufwand benötigen, jedoch wird sich das in den nächsten Levels ändern wie wir sehen werden.

Aufgabe 2: Knobelaufgabe: Diskretisationsfehler

Wie wir in diesem Level feststellen mussten, beinhaltet die Simulation immer einen gewissen Fehler aufgrund der Diskretisation. Aber wie gross ist dieser Fehler? Versuchen wir die analytische Lösung mit der Lösung der Simulation anhand des schiefen Wurfes zu vergleichen. Als Ausgangslage dient uns die Formel unserer Schrittfunktion

$$\begin{aligned}\vec{\mathbf{v}}^{\,i+1} &= \vec{\mathbf{v}}^{\,i} + \vec{\mathbf{g}}\,\Delta t \\ \vec{\mathbf{x}}^{\,i+1} &= \vec{\mathbf{x}}^{\,i} + \vec{\mathbf{v}}^{\,i}\,\Delta t\end{aligned}$$

wobei

$$\begin{aligned}\vec{\mathbf{v}}^{\,0} &= \vec{\mathbf{v}}_0 \\ \vec{\mathbf{x}}^{\,0} &= \vec{\mathbf{x}}_0\end{aligned}$$

. Ausserdem haben wir die analytische Formel, welche die Position von $\vec{\mathbf{x}}(t)$ gibt:

$$\vec{\mathbf{x}}(t) = \vec{\mathbf{x}}_0 + \vec{\mathbf{v}}_0 t + \frac{1}{2}\vec{\mathbf{g}}\,t^2$$

. Die Diskretisation an der Stelle $\vec{\mathbf{x}}^i$ ergibt uns also eine Approximation für die analytische Lösung an der Stelle $\vec{\mathbf{x}}(\Delta t \cdot i)$. Versuche nun den Diskretisationsfehler zu bestimmen, also $\vec{\epsilon} = |\vec{\mathbf{x}}^i - \vec{\mathbf{x}}(\Delta t \cdot i)|$ Dafür musst du zuerst $\vec{\mathbf{x}}^i$ als Funktion von $\vec{\mathbf{v}}_0$ und $\vec{\mathbf{x}}_0$ umformen. Wie verhält sich der Fehler in Abhängigkeit zu i und Δt?

Tipp: Schritt 1

Finde $\vec{\mathbf{v}}^i$ als Funktion von $\vec{\mathbf{v}}_0, \vec{\mathbf{g}}$ und i.

Tipp: Schritt 2

Setze $\vec{\mathbf{v}}^i$ in die Funktion von $\vec{\mathbf{x}}^i$ ein und finde $\vec{\mathbf{x}}^i$ in Abhängigkeit von $\vec{\mathbf{v}}_0, \vec{\mathbf{g}}, \vec{\mathbf{x}}_0$ und i. Hinweis: Die Gausssche Summenformel lautet $\sum_{k=1}^{n} k = \frac{n(n-1)}{2}$.

Tipp: Schritt 3

Berechne die Differenz $\vec{\epsilon}$.

2.5: GENERISCHE BESCHLEUNIGUNG

Nachdem wir den Einfluss der Schrittkonstante genauer untersucht haben nähern wir uns dem Ende dieses Kapitels. In diesem Kapitel haben wir uns bisher immer auf der Erde befunden und die Beschleunigung war bisher immer konstant in die negative y-Richtung. Nun verlassen wir die Erde und kehren in den Weltraum zu Pluto zurück. Bald werden wir uns mit der Gravitationskraft, welche die Sonne auf Pluto ausübt, beschäftigen. Als Vorbereitung dazu wollen wir zuerst die Beschleunigung generisch implementieren.

Tipp

Die Beschleunigung in diesem Level kannst du selbst steuern. Wenn du rechts klickst, dann setzt du die Beschleunigung in die Richtung des Mauszeigers. Die Amplitude des Beschleunigungsvektors beträgt immer der Distanz zwischen deinem Mauszeiger und Pluto.

Aufgabe 1: Schrittfunktion mit generischer Beschleunigung

In dieser Schrittfunktion erhältst du zusätzlich zu den bekannten Parametern $\vec{\mathbf{x}}$, $\vec{\mathbf{v}}$ und Δt den Beschleunigungsvektor $\vec{\mathbf{a}}$ als Eingabe in die Schrittfunktion. Vervollständige die Schrittfunktion mit der generischen Beschleunigung und teste deine Implementierung indem du mit der rechten Maustaste in den Weltraum klickst.

Aufgabe 2: Erste Orbits

Versuche den Mauszeiger an Ort und Stelle zu halten und beobachte Pluto dabei. Was stellst du fest? Schau dir das Verhalten von verschieden grossen Orbits an. Inwiefern unterscheiden sich die Orbits von dieser Simulation von denjenigen der physikalischen Gravitationskraft?

3.1: MASSE-FEDER SIMULATION

Nun ist es endlich soweit. Wir werden die erste Simulation auf einem echten Beispiel durchführen. Dafür werden wir nochmals einen kleinen Abstecher zurück auf die Erde machen. In diesem Level werden wir eine Masse von $10kg$ an einer Feder befestigen. Unsere Masse wird deshalb drei verschiedenen Kräften ausgesetzt: Der Gravitationskraft auf der Erde, der Federkraft sowie einer Dämpfungskraft. Wir werden nun Schritt für Schritt diese drei Kräfte in unserer Simulation integrieren. Aber bevor wir mit der Implementierung der Kräfte starten können müssen wir die Beschleunigung berechnen, welche aus der Kraft resultiert.

Aufgabe 1: Statische Kraft

Deine erste Aufgabe besteht darin, die Kraft in Beschleunigung zu übersetzen. Damit du bereits ein erstes Ergebnis deiner Implementierung beobachten kannst, nehmen wir eine statische Kraft von $-1N$ in die y-Richtung an. Implementiere nun das zweite Gesetz von Newton $\vec{\mathbf{F}} = m\,\vec{\mathbf{a}}$ in der Funktion `step`.

Aufgabe 2: Einfluss der Masse

Experimentiere mit dem Slider der Masse der Kugel. Wie verändert sich die Simulation wenn du die Masse vergrösserst respektive verkleinerst?

Aufgabe 3: Gravitationskraft

Nun, da wir die Kraft in Beschleunigung übersetzen konnten wollen wir unsere erste Kraft implementieren. Als erstes widmen wir uns der Gravitationskraft auf der Erde. Die Gravitationskraft zeigt in die negative y-Richtung und aus der Physik kennen wir auch die Formel dazu: $\vec{\mathbf{F}_{\mathbf{grav}}} = m\,\vec{\mathbf{g}}$. Vervollständige nun die

Funktion `calculateGravityForce` und setze die Gesamtkraft $\vec{\mathbf{F}_{\mathbf{tot}}}$ auf die Gravitationskraft. Validiere dein Resultat in der Simulation. Welchen Einfluss hat die Masse nun auf das Simulationsergebnis? Weshalb?

Als nächstes wollen wir die Federkraft genauer unter die Lupe nehmen. Wie du wahrscheinlich aus der Physik weist, hat eine Feder immer einen Ruhezustand, in welchem sie weder gespannt noch komprimiert ist. Unsere Feder ist an der Decke einer Fabrikhalle auf $20m$ befestigt und im Ruhezustand ist sie $10m$ lang. Sobald wir jedoch unsere Masse an der Feder befestigen, wird sich die Feder verlängern, da das Gewicht unserer Masse die Feder nach unten drückt. Ausserdem hat jede Feder eine sogenannte Federkonstante. Diese gibt uns die Steifheit der Feder oder mit anderen Worten wieviel Kraft es braucht um die Feder zu komprimieren respektive auszudehnen. Die Federkonstante bezeichnen wir als k. Die Federkonstante gibt uns also die benötigte Kraft, um die Feder aus dem Ruhestand heraus zu bewegen.

Aufgabe 4: Federkraft

Vervollständige die Funktion `calculateSpringForce` . Die Formel für die Kraft lautet $\vec{\mathbf{F}_{\mathbf{spring}}} = k(\vec{\mathbf{x_0}} - \vec{\mathbf{x}})$. Vergiss nicht, die neue Kraft mit der Gravitationskraft zu addieren, denn die Gesamtkraft beträgt immer

$$\vec{\mathbf{F}_{\mathbf{total}}} = \sum_i \vec{\mathbf{F_i}} = \vec{\mathbf{F}_{\mathbf{spring}}} + \vec{\mathbf{F}_{\mathbf{grav}}}$$

Verändere das Gewicht unserer Masse. Welchen Einfluss hat das Gewicht mit der Federkraft? Weshalb? Ist diese Simulation deiner Meinung nach realistisch?

Unsere Simulation scheint nach wie vor nicht mit der Intuition übereinzustimmen. Nach unserer Intuition müsste die Oszillation der Masse abnehmen und nach einer gewissen Zeit einen Ruhezustand erreichen. Das liegt daran, dass unsere Masse zur Zeit nicht gedämpft wird. Wir müssen also eine weitere Kraft, die Dämpfungskraft implementieren.

Aufgabe 5: Dämpfungskraft

Die Dämpfungskraft ist von der Geschwindigkeit der Masse abhängig. Je schneller sich die Masse bewegt, je stärker wird sie gedämpft. Die Dämpfungskraft hängt Konstanten c ab. Je grösser c, je stärker wird die Masse gedämpft und erlangt ihren Ruhezustand. Die Kraft kann mit der folgenden Formel ausgedrückt werden:

$$\vec{\mathbf{F}_{\mathbf{damp}}} = -c\,\vec{\mathbf{v}}$$

Vervollständige die letzte Kraftfunktion `calculateDampingForce` und füge sie zur Schrittfunktion hinzu. Verändere nochmals das Gewicht unserer Masse. Welchen Einfluss hat das Gewicht auf die gesamte Simulation? Inwiefern kannst du die Oszilationsperiode damit beeinflussen?

3.2: GENERISCHE GRAVITATION

Nach dem Abstecher auf die Erde kehren wir wieder zum Weltraum zurück. Pluto befindet sich nicht im leeren Weltraum. Es gibt viele weitere Körper im Weltraum wie die Sonne, Planeten, Sterne etc. All diese Körper üben eine Kraft auf Pluto aus: Die Gravitationskraft. Dabei ist die Gravitationskraft der Sonne diejenige, welche Pluto mit Abstand am stärksten beeinflusst. Aus diesem Grund konzentrieren wir uns in diesem Level nur auf die Gravitationskraft der Sonne.

Die Formel für die Gravitationskraft lautet

$$F_{grav} = G\frac{m_1 m_2}{r^2}$$

wobei $G = 6.674 \cdot 10^{-11}$ die Gravitationskonstante ist, m_1 und m_2 die Massen der zwei Körper ist, welche die Gravitationskraft ausüben (in unserem Fall die Masse der Sonne respektive Pluto) und r ist die Distanz zwischen den zwei Körpern.

Aufgabe 1: Berechnung des Vektors von Pluto zu der Sonne

Um die Gravitationskraft zu berechnen, brauchen wir in einem ersten Schritt den Vektor, welcher von Pluto zur Sonne zeigt. Versuche in der Funktion `calculateGravityForce` den Vektor $\vec{\mathbf{r}}$ von Pluto zur Sonne zu berechnen.

Tipp

Ein Vektor von Punkt A zu Punkt B berechnet man mit der Formel

$$\mathbf{v}_{\mathbf{A}\vec{\rightarrow}\mathbf{B}} = \begin{pmatrix} B_x - A_x \\ B_y - A_y \end{pmatrix}$$

Aufgabe 2: Berechnung des Vektors von Pluto zu der Sonne

Als nächstes müssen wir den Betrag des Vektors $\vec{\mathbf{r}}$ berechnen. Als kleiner Zwischenschritt können wir das Quadrat $|\vec{\mathbf{r}}|^2$ berechnen. Benutze dafür den Satz des Pythagoras.

Tipp

Die Formel für das Quadrat des Betrages eines Vektors lautet

$$|\vec{\mathbf{v}}|^2 = v_x^2 + v_y^2$$

Nun müssen wir nur noch die Wurzel aus dem Quardrat des Betrages $|\vec{\mathbf{r}}|^2$ ziehen. In Python kann die Wurzel mit der Funktion `math.sqrt(x)` gezogen werden.

Aufgabe 3: Berechnung der Gravitationskraft

Berechne nun die Gravitationskraft $F_{grav} = G\frac{m_1 m_2}{r^2}$ mit den Resultaten der vorherigen Aufgaben. Dein Resultat wird nun ein Skalar sein. Da wir uns jedoch im zwei-dimensionalen Raum befinden, müssen wir die Kraft Vektorisieren. Aber in welche Richtung zeigt die Gravitationskraft eigentlich? Die Kraft wirkt immer in die Richtung vom angezogenen Körper (Pluto) zum kraftausübenden Körper (Sonne). Glücklicherweise haben wir bereits den Vektor $\vec{\mathbf{r}}$ ausgerechnet, welcher genau in diese Richtung zeigt. Jedoch beinhaltet dieser Vektor nicht nur die Richtung sondern auch die Distanz zwischen Pluto und Sonne. Wenn wir die

Gravitationskraft direkt mit diesem Vektor multiplizieren würden, würde das die Gravitationskraft mit der Distanz multiplizieren. Um das zu verhindern, müssen wir zuerst den Vektor $\vec{\mathbf{r}}$ **normalisieren**. Die **Normalisierung** setzt dabei den Betrag des Vektors auf eins (Einheitsvektor). Das Berechnen des normalisierten Vektors ist grundsätzlich sehr einfach: Wir müssen lediglich beide Komponenten des Vektors (x und y) durch den Betrag des Vektors dividieren. Berechne nun den normalisierten Richtungsvektor $\hat{\vec{\mathbf{r}}} = \frac{\vec{\mathbf{r}}}{|\vec{\mathbf{r}}|}$ und multipliziere diesen danach mit der skalaren Gravitationskraft um die gerichtete Gravitationskraft zu erhalten. Starte dann die Simulation und beobachte, wie Pluto um die Sonne rotiert.

Aufgabe 4: Verschiedene Pluto Orbits

Mit der jetzigen Parametrisierung der Simulation kreist Pluto in einem fast perfekten Kreis um die Sonne. Verändere nun die Koordinaten der Sonne `x_sun` auf `[10, 0]` . Wie verändert sich der Orbit von Pluto? An welcher Stelle ist Pluto am schnellsten bzw. am langsamsten unterwegs? Wie verhält sich seine Geschwindigkeit in Abhängigkeit zur Distanz zur Sonne?

3.3: ASTEROIDEN EINSCHLAG

Ein Asteroid nähert sich der Erde. Wir wollen in diesem Level herausfinden, wo der Asteroid auf der Erde einschlagen wird. Dafür wird die Gravitationskraft, welche wir in der letzten Aufgabe berechnet haben nützlich sein. Jedoch reicht die Gravitationskraft alleine noch nicht aus. Die Erde besitzt eine Atmosphere und die Partikel in der Atmosphere üben eine Luftwiderstandskraft auf den Asteroiden aus, welche den Asteroiden zusätzlich abbremst.

Die Formel für die Luftwiderstandskraft lautet

$$F_{drag} = -\frac{1}{2}\rho|\vec{\mathbf{v}}|^2 C_D A$$

wobei ρ die Dichte des Mediums ist (in unserem Fall die Luft in der Atmosphere). Die Dichte wird in $\frac{kg}{m^3}$ gemessen, also die Masse von Luft in einem Kubikmeter. $\vec{\mathbf{v}}$ ist die Geschwindigkeit des Asteroiden und C_D ist der Luftwiderstandskoeffizient. Der Luftwiderstandskoeffizient C_D beschreibt dabei, wie aerodynamisch eine bestimmte geometrische Form ist. Der Luftwiderstandskoeffizient einer perfekten Sphere liegt bei 0.47 und es ist derjenige, welchen wir für den Asteroiden annehmen. Im Gegensatz liegt z.B. der Luftwiderstandskoeffizient eines Flugzeugflügels bei lediglich 0.04. Zu guter Letzt ist A die Querschnittsfläche des Asteroiden. Da wir annehmen, dass der Asteroid eine perfekte Sphäre ist, ist die Querschnittsfläche ein perfekter Kreis.

Aufgabe 1: Querschnittsfläche

Berechne zuerst die Querschnittsfläche des Asteroiden innerhalb der Funktion `calculateDragForce` . Der Radius des Asteroiden beträgt $0.5m$.

Aufgabe 2: Luftwiderstandskraft

Berechne als nächstes die Luftwiderstandskraft mithilfe der Formel von oben. Die Luftwiderstandskraft ist dabei immer in die entgegengesetzte Richtung des Geschwindigkeitsvektors gerichtet. Nutze dazu

dasselbe Prinzip wie bei der Gravitationskraft um den Betrag und die Richtung des Geschwindigkeitsvektors zu isolieren.

Bisher haben wir angenommen, dass die Atmosphäre der Erde im ganzen Weltraum überall genau gleich gross ist. In der Realität ist das jedoch nicht der Fall und die Atmosphäre ist auf der Erdoberfläche auf Meeresebene am höchsten. Sobald man einen höheren Berg besteigt, nimmt die Dichte der Atmosphäre ab. Das kennt man im Alltag aus zwei verschiedenen Phänomenen. Zum einen Teil nimmt durch die abnehmende Dichte auch die Dichte des Sauerstoffes ab und deshalb haben die meisten Extremsportler, welche Berge wie der Everest besteigen zusätzliche Sauerstoffflaschen dabei. Zum anderen Teil nimmt auch der Siedepunkt von Wasser ab, je höher man in die Berge geht. Deshalb muss man dann auch Teigwaren und Gemüse länger kochen.

Die Dichte der Atmosphäre lässt sich mit dieser Formel als Funktion der Höhe über der Erdatmosphäre approximieren

$$\rho(h) = \rho_0(1 - \frac{r_e - h}{h_a})^{k_2}$$

, wobei r_e der Radius der Erde ist, h der Abstand zwischen dem Mittelpunkt der Erde und des Asteroiden ist und h_a die Höhe der Atmosphäre gemessen von der Erdatmosphäre ist. k_2 ist dabei eine Konstante, welche den Abfall der Atmosphärendichte bestimmt.

Aufgabe 3: Rand der Atmosphäre

Ausserhalb der Atmosphäre nehmen wir an, dass sich der Asteroid ungestoppt fortbewegen kann. Verändere die Funktion `getAirDensity` so, dass die Funktion eine Kraft von Null zurückgibt, falls sich der Asteroid ausserhalb der Atmosphäre befindet (also $h > h_a + r_e$).

Aufgabe 4: Variable Atmosphärendichte

Versuche nun als nächstes die Formel der Atmosphärendichte von oben zu implementieren. Die Atmosphäre ist am dichtesten, wenn der Asteroid sich direkt auf der Erdoberfläche befindet und nimmt exponentiell ab je weiter er sich von der Erdatmosphäre entfernt befindet. Welchen Einfluss hat die verbesserte Approximation der Erdatmosphäre auf die Simulation?