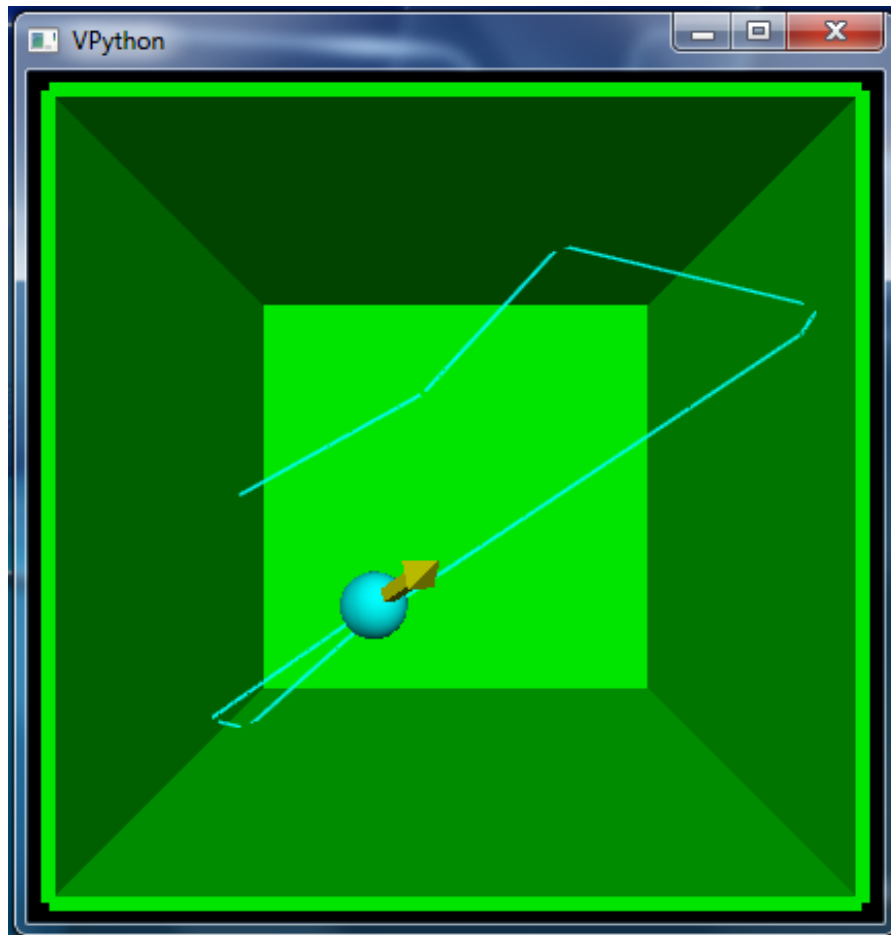
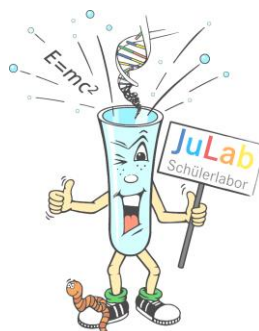


# A Ball in a Box

## Programmieren mit VPython



<b>EINLEITUNG</b>	<b>1</b>
<b>1. EIN ERSTES KURZES PROGRAMM</b>	<b>2</b>
<b>2. EIN BLICK DURCH DIE KAMERA</b>	<b>3</b>
<b>3. KOORDINATEN</b>	<b>4</b>
<b>4. AKTUALISIERUNG DER POSITION</b>	<b>5</b>
<b>5. KONTINUIERLICHES AKTUALISIEREN DER POSITION: WHILE</b>	<b>7</b>
<b>6. DEN BALL ABPRALLEN LASSEN: IF</b>	<b>10</b>
<b>7. VISUALISIERUNG DER GESCHWINDIGKEIT</b>	<b>12</b>
<b>8. AUTOSCALING</b>	<b>14</b>
<b>9. DIE BAHNKURVE</b>	<b>15</b>



# Einleitung

Der Artikel „Physik“ unter [www.wikipedia.de](http://www.wikipedia.de) enthielt 2012 die folgende Sätze:

„Die Arbeitsweise der Physik besteht im Allgemeinen in einem Zusammenspiel experimenteller Methoden und theoretischer Modellbildung, welche weitgehend mathematische Methoden verwendet. Physikalische Theorien bewähren sich in der Anwendbarkeit auf Systeme der Natur, indem sie bei Kenntnis von Anfangszuständen derselben möglichst genaue Vorhersagen über resultierende Endzustände erlauben.“

Ein wichtiges Instrument des oben genannten Zusammenspiels zwischen Experiment und Theorie ist die Computersimulation. Sie erlaubt – gerade bei sehr komplexen Sachverhalten – eine Vorhersage aufgrund theoretischer Grundüberlegungen. Andererseits kann man mit Computersimulationen experimentelle Ergebnisse nachmodellieren und daraus wichtige theoretische Erkenntnisse gewinnen.

Simulationsrechnen als auch die Bereitstellung der dazu erforderlichen Rechenleistung durch Supercomputer sind im Forschungszentrum Jülich in einem Institut, dem Institute for Advanced Simulation (IAS), vereint.

Link:

[http://www.fz-juelich.de/portal/DE/UeberUns/Organisation/Institute/InstituteAdvancedSimulation/\\_node.html;jsessionid=040E22D0986DE7580A90D24106E2A7F3](http://www.fz-juelich.de/portal/DE/UeberUns/Organisation/Institute/InstituteAdvancedSimulation/_node.html;jsessionid=040E22D0986DE7580A90D24106E2A7F3)

Das Programm „A Ball in a Box“ stellt ebenfalls eine Simulation dar. Mit Hilfe dieses Manuskripts könnt ihr das Programm selbst schreiben, und die einzelnen Schritte der Programmierung nachvollziehen und verstehen. Da es „nur“ darum gehen soll, einen Einblick ins Programmieren zu bekommen, ist das Bewegungsproblem so einfach wie möglich behandelt und auch nur grob modelliert.

Wir werden zum Programmieren VPython nutzen. VPython ist eine Programmiersprache, die leicht zu lernen ist und geeignet dazu, 3D-Modelle von physikalischen Systemen zu erstellen. VPython ist kostenlos unter [www.vpython.org](http://www.vpython.org) im Internet erhältlich. Neben Download-Möglichkeiten sind dort auch Dokumentationen und Tutorials zu finden.<sup>1</sup>

VPython hat drei Komponenten, mit denen ihr direkt zu tun haben werdet:

- **Python:** eine Programmiersprache
- **Visual:** ein 3D Graphik Modul für Python. Visual erlaubt euch, 3D-Objekte zu kreieren und zu animieren, und in einer 3D-Szene durch Drehen und Zoomen (mit der Maus) zu navigieren.
- **VIDLE:** eine graphische Benutzeroberfläche

Ziel der Übung ist es die Bewegung eines Balls, der in einer Box gefangen ist, aufgrund von bekannter Startposition und bekannter Startgeschwindigkeit zu berechnen und eine 3D-Animation der Bewegung zu erstellen.



Die englische Originalbeschreibung, die diesem Skript zugrunde liegt, findet ihr unter [www.vpython.org/contents/docs/VPython\\_Intro.pdf](http://www.vpython.org/contents/docs/VPython_Intro.pdf)

# 1. Ein erstes kurzes Programm



- Startet „VIDLE for Python“ (auf dem Desktop)
- In dem erscheinenden leeren Fenster schreibt ihr:

```
from visual import *
```



Dieser Satz fordert Python auf das Visual-Graphikmodul zu benutzen



- Als zweite Zeile eures Programms, tippt den folgenden Satz:

```
sphere ( )
```



Bevor ihr das Programm ausführt, soll es zunächst gespeichert werden. Wichtig ist, dass ihr die Datei mit der Endung **.py** abspeichert. (Ihr könnt natürlich im weiteren Verlauf auch immer zwischendurch speichern: im File-Menü „Save“ oder „Save as...“ auswählen).



- Speichert euer Programm im Ordner „Save“ ab!
- Startet euer Programm mit F5 (oder durch „Run Module“ im Run-Menü)



Es sollte nun ein Fenster erscheinen, dass mit „VPython“ betitelt ist, und eines mit dem Namen „Python Shell“. Es ist sinnvoll, das Shell-Fenster an eine Stelle des Bildschirms zu setzen, wo es nicht im Weg, aber trotzdem noch sichtbar ist.

Das VPython-Fenster sollte wie in Abbildung 1 aussehen.

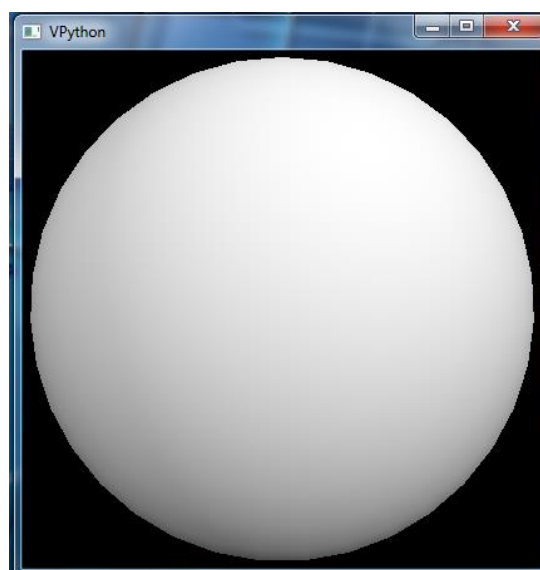
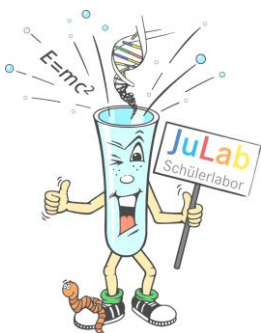


Abbildung 1: Eine Kugel in VPython



## 2. Ein Blick durch die Kamera



- Klickt gleichzeitig mit der rechten **und** linken Maustaste in das VPython-Fenster und bewegt die Maus!



Was beobachtet ihr?



- Haltet nun nur die rechte Maustaste gedrückt und bewegt die Maus!




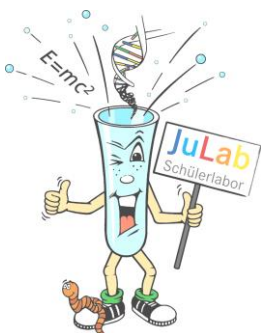
Was könnt ihr nun beobachten? [Ihr könnt statt der Kugel einen Quader erstellen, indem ihr `box ( )` als Kommandozeile in euer Programm schreibt (statt `sphere ( )` für die Kugel) und den Vorgang auch beim Quader beobachten. Falls der Quader in dem VPython-Fenster nicht zu erkennen sein sollte, versucht es mit Zoomen!]




Für uns ist folgendes wichtig: Wenn wir Zoomen oder Drehen, verändern wir nicht die Position oder räumliche Orientierung des Objektes. Es ist so, als ob wir das Objekt mit einer „Kamera“ beobachten. Wenn wir also zoomen und drehen, „bewegt“ sich nur unsere Kamera, nicht das Objekt!



- Stoppt das Programm, indem ihr  oben rechts im Python-Fenster anklickt. Lasst das Shell-Fenster geöffnet, denn wir brauchen es für unser nächstes Programm.
- Klickt im File-Menü auf „New Window“



### 3. Koordinaten

-  Um Objekte an gewünschte Stellen im VPython-Fenster zu platzieren, werden kartesische Koordinaten ( $x$ ,  $y$ ,  $z$ ) benutzt. Der Ursprung des Koordinatensystems liegt in der Mitte des Fensters. Die positive  $x$ -Achse zeigt nach rechts, die positive  $y$ -Achse nach oben, und die positive  $z$ -Achse kommt euch aus dem Monitor entgegen (siehe Abbildung 2).

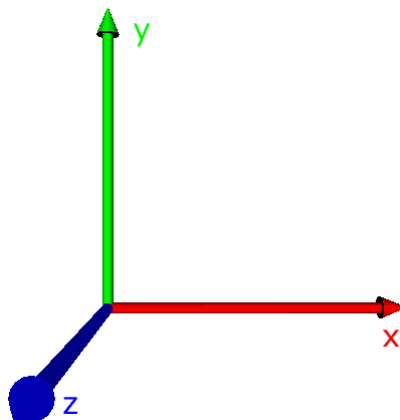


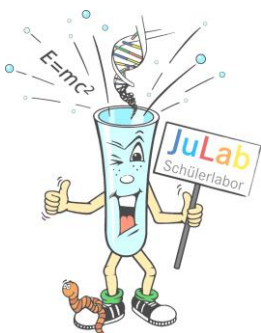
Abbildung 2: Koordinatensystem



- Schreibt das folgende Programm!

```
from visual import *
ball = sphere (pos=(-5,0,0), radius=0.5, color=color.cyan)
wallR = box(pos=(6,0,0), size=(0.2,12,12), color=color.green)
```

- Speichert das Programm!
- Startet das Programm!
- Macht euch die Bedeutung der einzelnen Befehle und deren Aufbau klar. Ausprobieren, d.h. Ändern der Einstellungen ist ausdrücklich erlaubt! (Mögliche Farben: red, green, blue, yellow, magenta, cyan, orange, black, white)



## 4. Aktualisierung der Position



Der Ball soll sich nun über den Bildschirm bewegen und von der grünen Wand abprallen. Wie stellen wir das an? Wir können uns vorstellen, dass unsere oben erwähnte „Kamera“ zu verschiedenen, aufeinanderfolgenden Zeitpunkten Schnappschüsse schießt, während der Ball über den Bildschirm wandert. Um angeben zu können, wie weit der Ball zwischen zwei einzelnen Schnappschüssen gewandert ist, müssen wir die Geschwindigkeit des Balls vorgeben und den Zeitraum, der zwischen zwei Schnappschüssen vergangen ist.

Die Geschwindigkeit des Balls wird von uns in unserem Programm `ball.velocity` genannt. Der Name ist willkürlich gewählt, stellt also noch kein Kommando in der Programmiersprache dar. Da unser Ball sich später in drei Dimensionen bewegen können dürfen soll, macht es Sinn, dass die Geschwindigkeit schon jetzt als einen dreidimensionalen Vektor mit drei Komponenten  $v_x$ ,  $v_y$  und  $v_z$  zu definieren.



- Definiert in eurem Programm den Geschwindigkeitsvektor `ball.velocity`!

```
ball.velocity = vector(25,0,0)
```



Das Zeitintervall zwischen zwei Schnappschüssen nennen wir `dt`. Um im Auge behalten zu können, wie viel Gesamtzeit `t` während der Bewegung verstreicht, wird die Gesamtzeit zum Start der Bewegung gleich Null gesetzt.



- Bestimmt das Zeitintervall `dt` und setzt den Zeitpunkt `t` gleich Null!

```
dt = 0.005  
t = 0
```

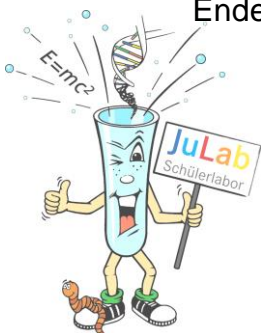
- Startet das Programm!



Der Ball bewegt sich immer noch nicht, da wir dem Programm noch nicht gesagt haben, wie es aus der Geschwindigkeit die neue Ballposition bestimmen soll. Eine einfache Beziehung (die ihr auch in *Der Supercomputer als Teleskop – Die Sternenbewegung im Zeitraffer* gefunden habt), welche die Position aus Geschwindigkeit und Zeitintervall berechnen kann ist :

$$\vec{r}_E = \vec{r}_A + \vec{v}\Delta t$$

Diese Beziehung gilt, wenn die Geschwindigkeit während des Zeitintervalls konstant bleibt (und zwar in Betrag und Richtung). Falls die Geschwindigkeit in Betrag und/oder Richtung nicht konstant sein sollte, muss das Zeitintervall so gewählt werden, dass die Geschwindigkeit zu Beginn des Zeitintervalls ungefähr gleich der Geschwindigkeit am Ende des Intervalls ist.

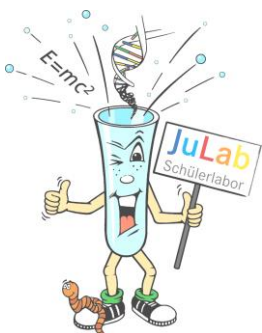


In der Sprache von VPython lautet die Beziehung  $\vec{r}_E = \vec{r}_A + \vec{v}\Delta t$

`ball.pos = ball.pos + ball.velocity*dt`

Weil die Geschwindigkeit ein Vektor ist, muss die Position des Balls, `ball.pos`, ebenfalls ein Vektor sein. Wie in den meisten Programmiersprachen, hat das Gleichheitszeichen eine andere Bedeutung als in der gewöhnlichen algebraischen Notation. In VPython bezeichnet das Gleichheitszeichen eine Zuweisung. Das heißt, die obige Zeile weist dem Vektor `ball.pos` einen neuen Wert zu, welcher sich aus dem gegenwärtigen Wert von `ball.pos` plus der Verschiebung `ball.velocity*dt` zusammensetzt.

- Schreibt die Beziehung  $\vec{r}_E = \vec{r}_A + \vec{v}\Delta t$  in der Sprache von VPython in euer Programm!



## 5. Kontinuierliches Aktualisieren der Position: while



- Kontrolliert, ob euer Programm folgendermaßen aussieht:

```
from visual import *  
ball = sphere(pos=(-5,0,0), radius=0.5, color=color.cyan)  
wallR = box(pos=(6,0,0), size=(0.2,12,12), color=color.green)  
ball.velocity = vector(25,0,0)  
dt = 0.005  
t = 0  
ball.pos = ball.pos + ball.velocity*dt
```

- Startet euer Programm!



Immer noch findet keine Bewegung statt! Das Problem ist, dass wir bisher nur ein einziges Zeitintervall berechnet haben. Um die Bewegung darstellen zu können, brauchen wir aber viele Zeitintervalle! Um viele Zeitintervalle zu berechnen, nutzen wir eine **while**-Schleife. Eine while-Schleife instruiert den Computer eine Reihe von Kommandos immer wieder und wieder durchzuführen bis wir sagen, dass er aufhören soll.



- Gebt die folgende Zeile vor eurer Positionsaktualisierung ein:

```
while t < 3:
```



Achtet auf den Doppelpunkt! Wenn ihr nun Enter drückt, erscheint der Cursor eine Zeile weiter an einer eingerückten Position. Die eingerückten Zeilen, die einer while-Schleife folgen, befinden sich innerhalb der Schleife, d.h. sie werden immer und immer wieder wiederholt. In diesem Fall werden sie solange wiederholt bis unsere „Stoppuhr“ eine Zeit kleiner als 3 Sekunden „misst“. Merkt euch, dass wenn der Cursor sich am Ende der while-Zeile befindet und ihr anschließend Enter drückt, die nächsten Zeilen automatisch eingerückt sind. Alternativ könnt ihr TAB drücken, um eine schon existierende Zeile einzurücken. Alle eingerückten Zeilen hinter einer while-Zeile werden immer ausgeführt, sobald die Schleife ausgeführt wird.







- Rückt eure Positionsaktualisierung unter der while-Schleife ein!
- Kontrolliert, ob euer Programm folgendermaßen aussieht und ergänzt die fehlende Zeile:

```
from visual import *
ball = sphere(pos=(-5,0,0), radius=0.5, color=color.cyan)
wallR = box(pos=(6,0,0), size=(0.2,12,12), color=color.green)
ball.velocity = vector(25,0,0)
dt = 0.005
t = 0
while t < 3:
    ball.pos = ball.pos + ball.velocity*dt
    t = t + dt
```



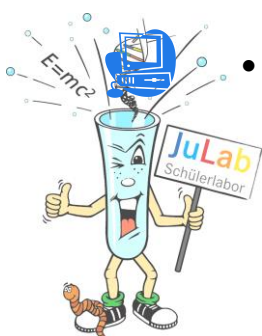
Was bewirkt die Zeile `t = t + dt`?



- Startet das Programm!



Abhängig von der Geschwindigkeit des ausführenden Computers, kann es sein, dass man keine Bewegung sieht, sondern nur die Wand und den Ball, wie sie aus weiter Entfernung dargestellt werden. Warum? Der Computer führt die Animation so schnell aus, dass wir nur die letzte Szene unserer Animation sehen, bei der das Programm schließlich stoppt. Aber warum sehen wir die Wand und den Ball aus weiter Entfernung? VPython versucht standardmäßig alle Objekte sichtbar zu halten. Das heißt, wenn sich der Ball weit vom Koordinatenursprung entfernt, bewegt VPython „die Kamera“ zurück, so dass Wand und Ball in der Ferne verschwinden (**autoscaling**). Wir lassen das autoscaling weiter eingeschaltet, verlangsamen aber unsere Animation folgendermaßen:



- Schreibt die folgende Zeile in die while-Schleife:

```
rate (100)
```



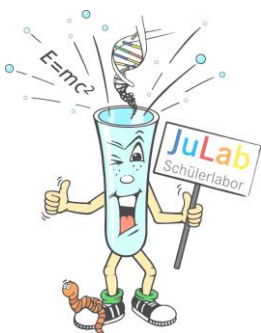
Das Kommando `rate(100)` gewährleistet, dass die Schleife nicht mehr als 100 mal pro Sekunde ausgeführt wird, auch wenn der Computer in der Lage ist, die Schleife weit mehr als 100 pro Sekunde auszuführen.



- Startet euer Programm!



Ihr solltet nun sehen, wie sich der Ball langsam nach rechts auf die Wand zu bewegt. Trotzdem bewegt sich der Ball durch die Wand hindurch in den leeren Raum hinein! Genau das bekommt der Computer gesagt. Da VPython keine Ahnung von Physik hat, müssen wir ihm sagen, dass der Ball von der Wand abprallen muss!



## 6. Den Ball abprallen lassen: if



Um den Ball von der Wand abprallen lassen zu können, muss auf irgendeine Art und Weise der Ort der Kollision detektiert werden. Eine einfache Methode ist, die x-Koordinate des Balls mit der x-Koordinate der Wand zu vergleichen. Hat sich der Ball zu weit nach rechts bewegt, wird die x-Komponente der Ballgeschwindigkeit einfach umgedreht. In VPython können den einzelnen Komponenten eines Vektors Eigenschaften zugeordnet werden. So ist beispielsweise `ball.pos` ein Vektor und `ball.pos.x` die x-Komponente dieses Vektors. Genauso ist `ball.velocity.x` die x-Komponente des Vektors `ball.velocity`.



- Fügt die folgenden Kommandozeilen in eure while-Schleife (vor der Positionsaktualisierung):

```
if ball.pos.x > wallR.pos.x:  
    ball.velocity.x = -ball.velocity.x
```



Die eingerückte Zeile nach dem if-Kommando wird nur ausgeführt, wenn eine Überprüfung des Wahrheitsgehalts der vorhergehenden Zeile „wahr“ ergibt. Falls die Überprüfung „falsch“ ergibt (d.h. wenn die x-Koordinate des Balls nicht größer ist als die x-Koordinate der Wand), wird die eingerückte Zeile übersprungen.



- Startet euer Programm!

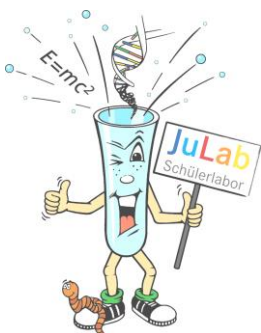


Ihr solltet nun beobachten können, wie der Ball nach recht gegen die Wand läuft, dort abprallt und sich nach links in den leeren Raum hinein bewegt.

Diese Art der Modellierung ist natürlich stark verbesserungswürdig. `ball.pos.x` liegt in der Mitte des Balls und `wallR.pos.x` in der Mitte der Wand. Wenn man genau hinsieht, kann man erkennen, dass der Ball bei der „Kollision“ ein wenig in die Wand eindringt. Eine Verbesserung des Modells wäre, den Radius des Balls und die Dicke der Wand zu berücksichtigen. Da es hier aber in erster Linie um das Kennenlernen von VPython gehen soll, akzeptieren wir einfach die Vereinfachung.



- Fügt eine weitere Wand an die linke Seite und gebt ihr den Namen `wallL`.
- Lasst den Ball auch von dieser Wand abprallen!



? Warum ist es unsinnig, die Wände innerhalb der while-Schleife zu erzeugen?



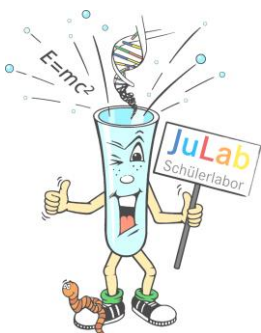
- Gebt der Anfangsgeschwindigkeit eine y-Komponente, die nicht Null ist!
- Startet das Programm!



Der Ball prallt nun auch an den Stellen ab, wo gar keine Wand mehr vorhanden ist! Der Grund hierfür ist wiederum, dass der Computer nur das macht, was man ihm sagt.



- Vervollständigt die Box so, wie auf dem Frontblatt des Skripts dargestellt! Der Ball soll von allen Wänden bei Kontakt abprallen!
- Gebt der Anfangsgeschwindigkeit eine z-Komponente und probiert euer Programm aus!



## 7. Visualisierung der Geschwindigkeit



Oftmals möchte man Vektorgößen, wie z.B. die Ballgeschwindigkeit in unserem Problem, sichtbar machen. Dazu können wir einen Pfeil benutzen, dessen Orientierung in Geschwindigkeitsrichtung zeigt, und dessen Länge dem Geschwindigkeitsbetrag entspricht. Da der Betrag in unserem Problem konstant bleibt, müssen wir hier nur auf die Richtungsänderung achten.



- Erzeugt mit dem folgenden Befehl einen Pfeil (an einer Stelle eures Programms, an der der Befehl Sinn macht):

```
varr = arrow(pos=ball.pos, axis=ball.velocity, color=color.yellow)
```



Was für eine Bedeutung hat der Inhalt der Klammer?

pos :

axis:



- Startet euer Programm!



Ihr solltet nun einen gelben Pfeil sehen, dessen hinteres Ende sich an der Startposition des Balls befindet. Der Pfeil ist sehr groß und dominiert die Szene. Geschwindigkeit und Position sind zwei verschiedene Größen, so dass wir den Geschwindigkeitspfeil der Größenordnung unserer Raumkoordinaten anpassen sollten. Wir wählen eine neue Skala für die Geschwindigkeit, indem wir den Geschwindigkeitsvektor mit einer Zahl multiplizieren. Die Multiplikation eines Vektors mit einer Zahl ändert nur den Betrag des Vektors, aber nicht dessen Richtung.

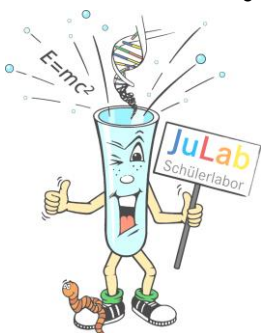


- Ändert die Längeskala des Geschwindigkeitsvektors folgendermaßen:

```
vscale = 0.1
```

```
varr = arrow(pos=ball.pos, axis=vscale*ball.velocity, color=color.yellow)
```

- Startet das Programm!

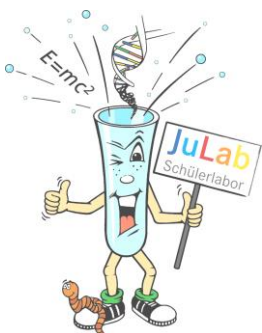




Der Pfeil sollte nun eine vernünftige Größe haben. Allerdings bewegt er sich noch nicht mit dem Ball. Die Position des Pfeils muss wie die Position des Balls ständig neu berechnet werden ([varr.pos](#)). Die Richtung des Pfeils wird durch [varr.axis](#) bestimmt und muss ebenfalls ständig aktualisiert werden.



- Ergänzt euer Programm so, dass sich der Geschwindigkeitspfeil mit dem Ball bewegt, und immer in Richtung der aktuellen Geschwindigkeit zeigt.



## 8. Autoscaling

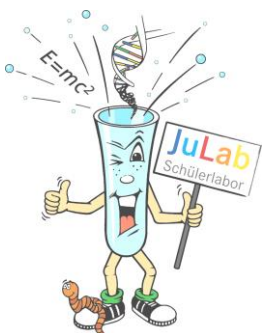


VPython skaliert die Szene automatisch, indem es die „Kamera“ vor- oder rückwärts bewegt. Falls gewünscht, könnt ihr die Autoskalierung nach der Startszene abschalten.

- Vor der while-Schleife, aber nach Erstellen der ersten Szene, könnt ihr die Autoskalierung folgendermaßen ausschalten:



```
scene.autoscale = False
```



## 9. Die Bahnkurve



Oftmals ist man an der Bahnkurve eines sich bewegenden Objektes interessiert.



- Schreibt folgende Zeile nach Erstellen des Balls, aber noch vor der while-Schleife:

```
ball.trail = curve(color=ball.color)
```

- Schreibt innerhalb der while-Schleife, nach der Aktualisierung der Ballposition:

```
ball.trail.append(pos=ball.pos)
```

