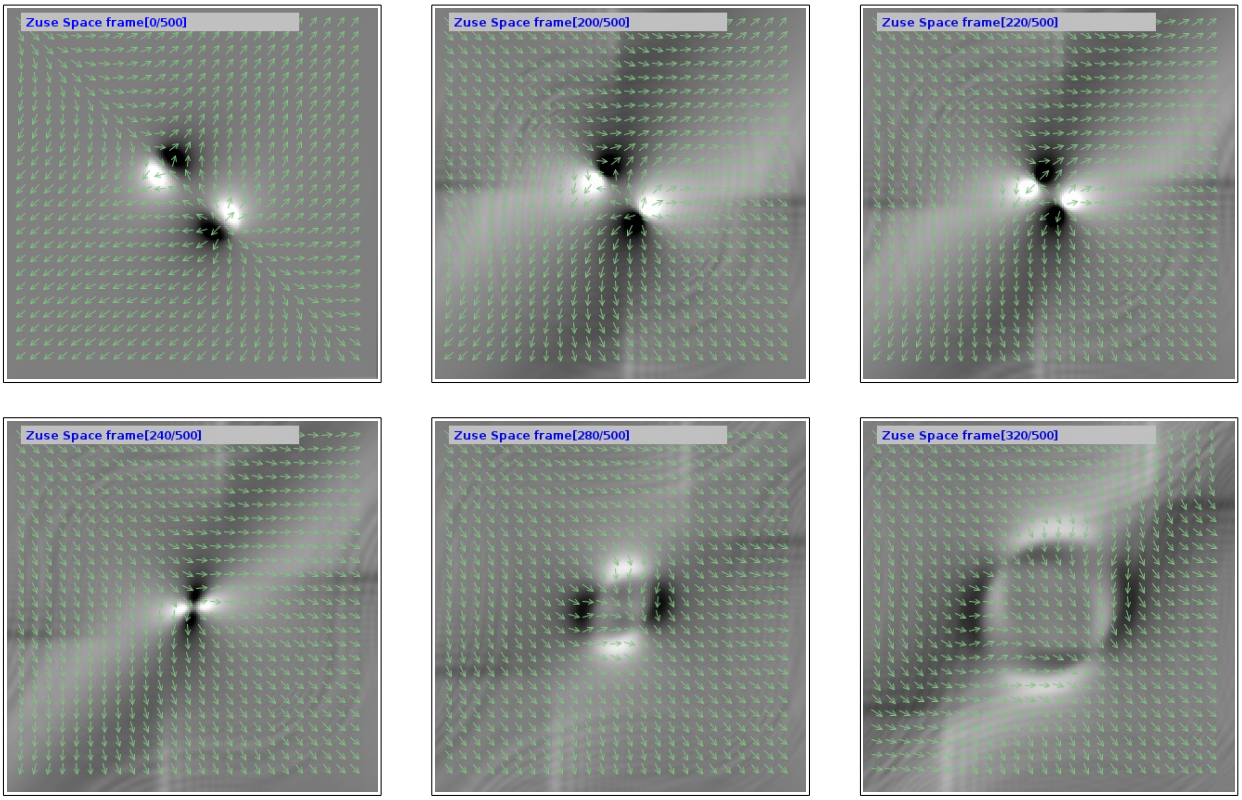
Zuse Space Annihilation

Wellen, Teilchen und Kräfte in einem diskreten Raum mit diskretem Zeittakt

Zusammenfassung

Der Autor hat vor längerer Zeit den historischen Zeitungsartikel [ZUSE_RR] gelesen. Das vorliegende Skript simuliert einen ähnlichen Raum in 2D - der Solitonen, ermöglicht durch die Sinus-Gordon-Gleichung, initialisiert und durch die in den Raum ausgesendeten Felder und wirkenden Kräfte zur Kollision bringt.



Das Beispiel fasst die Welt als eine Theaterbühne, bestehend aus kleinen Würfeln wie in einer Schachtel Würfelzucker, auf. In jedem Würfel (Pixel) befindet sich eine Kompassnadel die ihren Drehwinkel an die Nachbarn anpasst. Das Zusammenspiel mehrerer dieser, selbst unbeweglichen, Pixel kann ein Teilchen (Elektron) bilden. Das Elektron sendet ein Feld aus und kann sich fortbewegen indem es sich auf Nachbarpixel überträgt. Der Raum funktioniert durch schrittweises Berechnen des neuen Pixelzustandes aus den alten Pixelzuständen der Nachbarschaft. Somit könnte die Welt (und damit der Mensch auch) ganz aus den Zahlenwerten der Pixel gebildet sein. Alles materielle wäre ein Softwarekonstrukt.

© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.

Sinus-Gordon-Gleichung

Die Sinus-Gordon-Gleichung beschreibt einen Raum, mit einer Dimension, bei dem sich an jedem Ort ein zugeordneter Winkel befindet. Die Gesamtheit der Winkel in dem Raum bildet ein Skalarfeld. Für die Winkel in dem Raum bestehen zwei Tendenzen:

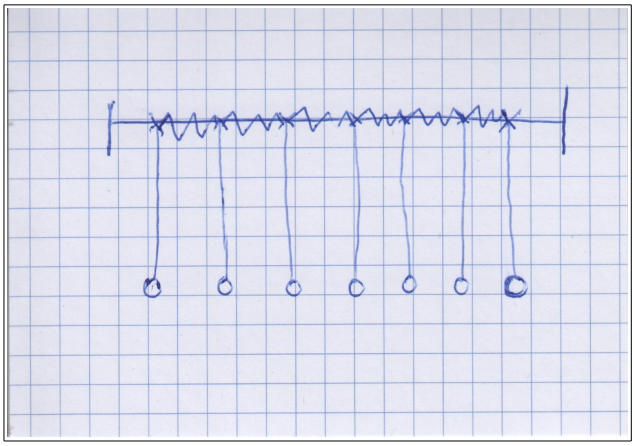
- Ein Winkel strebt dem Mittelwert der Winkel seiner direkten Umgebung zu.
- Ein Winkel strebt dem Wert 0 (NULL Bogenmaß) zu.

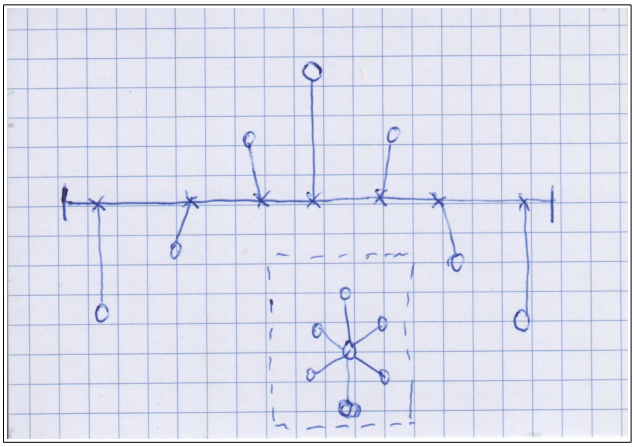
Dazwischen werden Kompromisse gemacht. In dem Raum können sich sowohl Wellen ausbreiten als auch Teilchen befinden. Die Gleichung lautet folgendermaßen:

$$\varphi_{tt} - \varphi_{xx} + \sin(\varphi) = 0$$

Dabei ist φ_{tt} die zweite Ableitung von φ nach der Zeit (also wie sehr eine Drehbewegung der Kompassnadel der Länge EINS beschleunigt wird). φ_{xx} ist die zweite Ableitung von φ nach dem Ort (also wenn es kein Winkel wäre - die Krümmung, Biegung). Ohne den Summand sin(φ), Rückstellbeschleunigung, liegt eine Wellengleichung vor.

Das Skript [WELLEN_M] bildet den Raum der Sinus-Gordon-Gleichung mechanisch als diskreten Raum mit kontinuierlicher Zeit nach. Es benutzt eine Kette aus etwa 100 Pendeln worin benachbarte Pendel mit Drehfedern in einander ähnlicher Position gehalten werden.





In der linken Abbildung befinden sich die Pendel in Ruhestellung (sin(φ) = 0). In der rechten Abbildung sind die Pendel jeweils um einen Winkelbetrag gleicher Richtung einander verdreht. Dieses Gebiet hat Eigenschaften eines Teilchens. Teilchen werden durch ihre Drehrichtung in eine der beiden Möglichkeiten „Kink“ und „Antikink“ unterteilt. Auch kann das Teilchen im Versuch „weitergeschraubt“ werden.

Die im kontinuierlichen Raum verwendeten Ableitungen sollen hier für sehr kleine aber endliche Differenzen synonym verwendet werden. φ_x entspricht der Winkeldifferenz zweier benachbarter Pendel und ist innerhalb des Teilchens unterschiedlich. φ_{xx} ist die zweite Ableitung nach dem Ort, die Differenz der Winkeldifferenzen und entspricht im

© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.

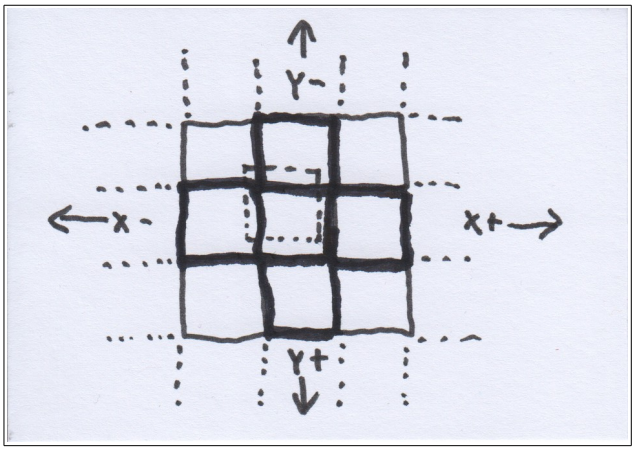
stabilen Zustand (wenn sich die Pendel nicht bewegen) auch der Auslenkung des Pendels. Da sin(φ) auch in der obersten Position des Pendels 0 ist könnten die Pendel für kurze Zeit diese instabile Stellung annehmen. φ_t ist die Drehgeschwindigkeit und φ_{tt} die Beschleunigung der Drehbewegung. sin(φ) ist ein Maß für die Auslenkung. Es ist gleich der senkrechten Gewichtskraft des Massestücks am Ende des Pendels.

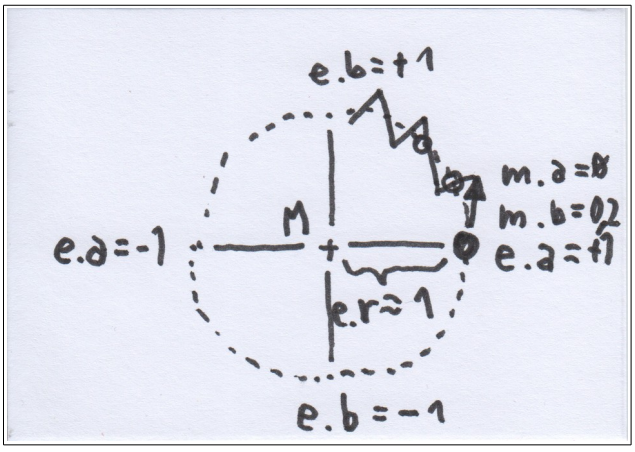
Die Teilchen dieses Raumes sind elastisch gegen Fehler durch Verbiegung und somit stabile Zustände. Sie können ihre Position verändern. Erzeugt werden sie von außen durch Paarbildung. Auch kann man sie (obwohl einander abstoßend) zur Annihilation bringen. Die Ausprägungen Kink und Antikink hängen mit dem Begriff „Elektronenspin“ zusammen. Im 1D - Raum üben sie zunächst keine Kräfte aufeinander aus. Spezielle Solitonen in Räumen mit 2 Dimensionen (vielleicht auch 3) nennt man Skyrmionen. Sie kommen in magnetischen Werkstoffen vor. Auch Elektronen könnten wie Skyrmionen aufgebaut sein.

Wellen im Raum benötigen mindestens zwei Variablen an einem Ort und werden durch die Position φ und die Geschwindigkeit φ_t gespeichert, repräsentiert. Davon lassen sich die übrigen Variablen berechnen. In dem Experiment kann man die Welle durch Anstubsen eines Pendels auslösen (La-Ola Welle). Am Ende eines nicht kreisförmigen Raumes wird die Welle reflektiert. Das könnte man durch Terminieren verhindern indem man dafür sorgt dass $\varphi = \varphi_t$ ist.

Simulation eines Pixels

Der Raum für die Simulation der Solitonen hat eine oder mehrere Längenabmessungen, die Dimensionen. In jeder der Dimensionen hat ein Pixel zwei Nachbarn die durch die Richtungsangaben KATA (weniger, z.B. nach links) und ANA (mehr, z.B. nach rechts) unterschieden werden.





In der linken Abbildung ist ein gerades Gitter mit 2 Dimensionen (x und y) skizziert wie es für die Simulation in diesem Skript verwendet wird. Jeder Pixel hat 4 Nachbarn im Sinne der Turm-Nachbarschaft aus dem Schachspiel.

© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.

Zur Simulation von Wellen nach Art der FDTD (einer Simulation mit diskretem Raum und diskreter Zeit) werden für jeden Pixel zwei Variablen benötigt, weil Wellen räumlich und zeitlich einen sinusförmigen Verlauf haben und die Sinusfunktion im Vollkreis des Bogenmaßes für einen Vorgängerpunkt mehrdeutige Nachfolgerpunkte hat. Der Sachverhalt ist nicht mit nur einer örtlichen Variable vereinbar wenn im zeitlichen Verlauf nur ein Vorgänger des Raumes im Speicher gehalten wird. Jede der beiden Variablen für den Pixel speichert einen Winkel je Dimension, hat also für den 2D-Raum Speicher für zwei Winkel. Das oben erwähnte „gerade Gitter“ bedeutet dass beide Variablen (wie durch den gestrichelten mittleren Pixel angedeutet) genau aufeinander liegen (für zweite Ableitung). Im Gegensatz dazu gibt es auch ein ungerades Gitter, das hier nicht verwendet wird, in diesem liegt die eine Variable örtlich genau auf dem Vierländereck der anderen Variable, ist also in beiden Koordinaten um die Hälfte versetzt (für erste Ableitung).

Für jeden Pixel werden somit zwei Variablen im Speicher gehalten. Der Simulator arbeitet nach dem Double-Buffer-Prinzip. In jedem der beiden Schritte des Double-Buffers wird für alle Pixel eine der beiden Variablen aus der anderen Variable von den benachbarten Pixeln berechnet. Die Nachbarschaftsbeziehungen zwischen den Pixeln existieren rein rechnerisch und legen den Raum aus dem Nichts fest. Pixel können im Hauptspeicher des Computers sogar völlig durcheinander gespeichert sein solange die Verweise zwischen den Nachbarn gegeben sind.

Damit man die Wellen am Rand des Raumes nicht gegen Reflexion terminieren muss kann man auch einen endlosen Kreis-Raum bauen. Für 1D ergibt sich ein gewöhnlicher Kreis. Für 2D kommt man auf einen Donut (wie in „Bernd das Brot“ der das Bild auf der einen Seite verlässt um auf der anderen wieder reinzukommen). Eine Kugel als Raum führt zu Schwierigkeiten bei Pixel-Nachbarschaften; es sind Vierer-Nachbarschaften während sich an Nordpol und Südpol Dreiecke ergeben.

Für jedes einzelne Pixel wird eine Variable für den Winkel unter dem Namen „electric“, kurz „e“ und eine Variable für die Geschwindigkeit unter dem Namen „magnetic“, kurz „m“ gespeichert.

Electric enthält die Information über eine Anzahl Winkel; die Anzahl der Winkel entspricht den Raum-Dimensionen. Die Winkel werden als ein Ortsvektor der Länge EINS gespeichert dessen Anzahl der Komponenten um eine Komponente größer ist als die Anzahl der Winkel. Alle Punkte, die mit dem Vektor erreicht werden können, beschreiben je nach Dimensionalität einen Kreis oder eine Kugel. Die Verwendung von Winkeln hat den Vorteil dass in dem Raum keine Variablen überlaufen können. Die Repräsentation als Vektor vermeidet einen Überlauf des Winkels und hält ihn gleichzeitig stetig. Sie hat Eigenschaften komplexer Zahlen. Bei mehr als zwei Vektor-Komponenten funktioniert die Winkeladdition durch Multiplikation aber nicht mehr.

© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.

Magnetic speichert eine Drehgeschwindigkeit des Ortsvektors von electric. Es wird durch die Subtraktion von der Komponente electric aus zwei oder mehr Pixeln berechnet. Um die Drehung von electric auszuführen wird magnetic aus dem selben Pixel zu ihm dazu addiert. Dabei verändert sich jedoch geringfügig des Radius von electric. Dem wird durch Addition eines Korrektursummanden zu magnetic Rechnung getragen.

Es folgt ein Beispiel für ein Pixel mit einem Positionswinkel für electric aus einem eindimensionalen Raum. Die beiden Vektoren electric und magnetic haben jeweils zwei Komponenten {a und b}. Der Pixel wird von dem Raum losgelöst betrachtet. Anfangs wird eine feste Geschwindigkeit in magnetic vorgegeben:

- Double-Buffer-Schritt 1: $\overline{e}_{i+1} := \overline{e}_i + \overline{m}_i$
- Double-Buffer-Schritt 2: $\overline{m}_{i+1} := \overline{m}_i + \overline{e}_{i+1} * (1 - |\overline{e}_{i+1}|)$ … + $\overline{m}_{i,t}$ + $\overline{back}_i$

Im ersten Schritt wird der Ortsvektor von electric gedreht. Im zweiten Schritt erfolgt die Korrektur der Geschwindigkeit in magnetic wie im obigen rechten Bild angedeutet. Der zusätzliche Summand $\overline{m}_{i,t}$ ist die zeitliche Änderung von magnetic in Abhängigkeit hier nicht berücksichtigter benachbarter Pixel. $\overline{back}_i$ eine Rückstellbeschleunigung.

Eine Simulation als Tabellenkalkulation ergab folgende Werte (mit p.e.r als Quadrat des Radius):

#	p.e.a	p.e.b	p.m.a	p.m.b	p.e.r
1	+1,000	+0,000	+0,010	+0,020	1,000
2	+1,010	+0,200	-0,051	+0,188	1,060
3	+0,959	+0,388	-0,119	+0,161	1,071
4	+0,841	+0,548	-0,125	+0,153	1,008
5	+0,716	+0,705	-0,131	+0,150	1,009
6	+0,584	+0,855	-0,174	+0,088	1,072
7	+0,411	+0,943	-0,198	+0,033	1,058
8	+0,213	+0,977	-0,197	+0,034	0,999
9	+0,016	+1,011	-0,197	+0,012	1,022
10	-0,182	+1,023	-0,183	-0,069	1,079
11	-0,365	+0,954	-0,168	-0,110	1,043
12	-0,533	+0,844	-0,170	-0,106	0,996
13	-0,703	+0,737	-0,144	-0,134	1,037
14	-0,846	+0,603	-0,076	-0,182	1,080
15	-0,922	+0,421	-0,050	-0,194	1,027
16	-0,972	+0,227	-0,053	-0,193	0,997

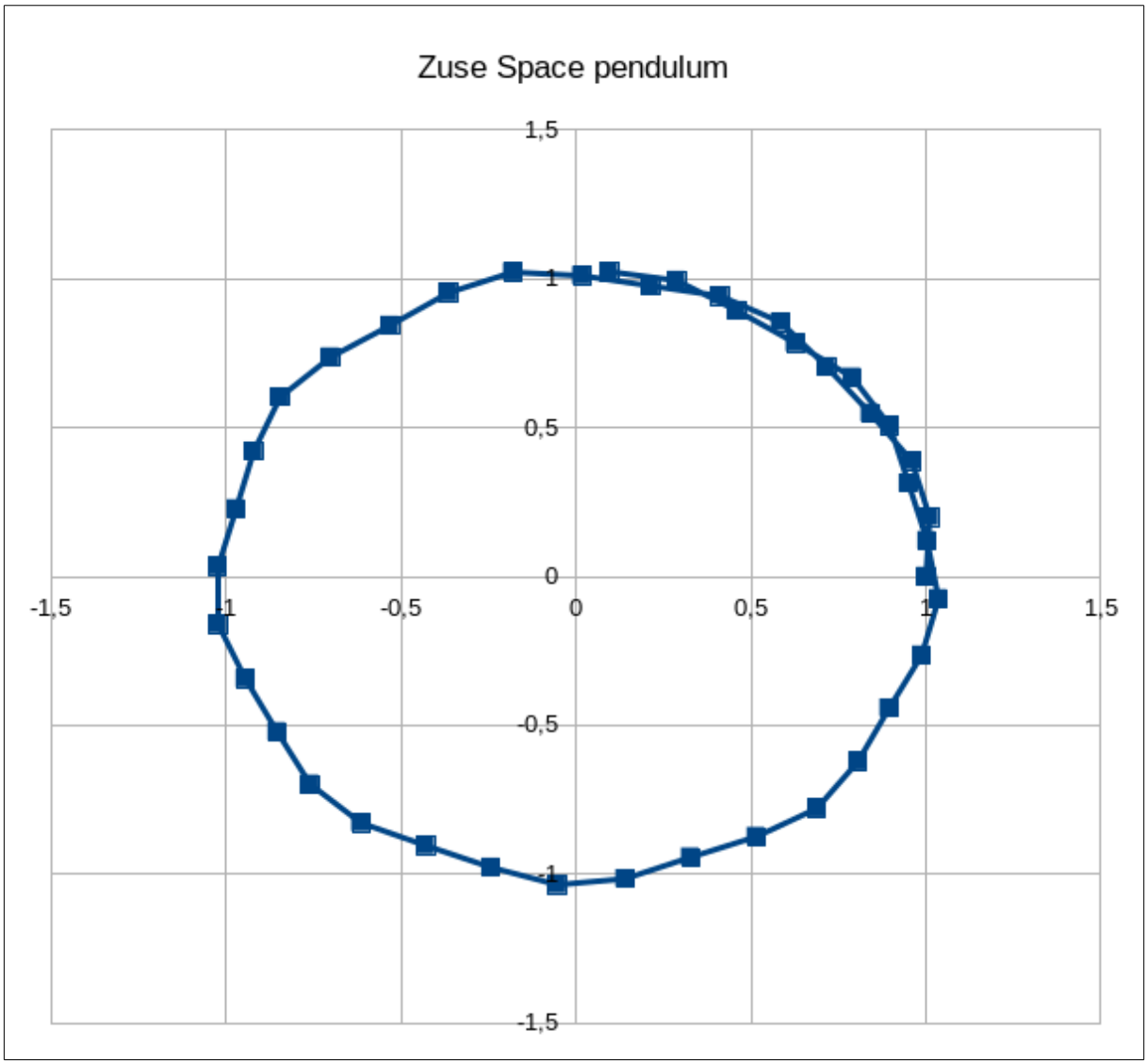
© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.

#	p.e.a	p.e.b	p.m.a	p.m.b	p.e.r
17	-1,026	+0,034	+0,001	-0,195	1,053
18	-1,024	-0,161	+0,078	-0,183	1,075
19	-0,046	-0,344	+0,091	-0,178	1,013
20	-0,855	-0,522	+0,094	-0,176	1,004
21	-0,761	-0,699	+0,145	-0,130	1,067
22	-0,616	-0,828	+0,185	-0,076	1,065
23	-0,431	-0,904	+0,186	-0,073	1,003
24	-0,244	-0,977	+0,190	-0,059	1,015
25	-0,054	-1,036	+0,194	+0,020	1,076
26	+0,140	-1,016	+0,187	+0,072	1,051
27	+0,327	-0,943	+0,188	+0,069	0,997
28	+0,515	-0,874	+0,173	+0,095	1,030
29	+0,688	-0,779	+0,118	+0,158	1,081
30	+0,805	-0,622	+0,089	+0,179	1,035
31	+0,894	-0,442	+0,093	+0,178	0,996
32	+0,987	-0,265	+0,048	+0,190	1,045
33	+1,036	-0,075	-0,033	+0,195	1,079
34	+1,003	+0,120	-0,053	+0,193	1,020
35	+0,950	+0,313	-0,053	+0,193	1,000
36	+0,896	+0,506	-0,107	+0,163	1,060
37	+0,789	+0,669	-0,163	+0,115	1,071
38	+0,626	+0,784	-0,168	+0,109	1,008
39	+0,459	+0,894	-0,172	+0,101	1,009
40	+0,287	+0,995	-0,193	+0,023	1,072
41	+0,094	+1,024	-0,198	-0,030	1,058

Zunächst wird electric mit einem Winkel in Richtung der Abszisse initialisiert. Jegliche Differenz zu einem nahegelegenen anderen Winkel müsste in magnetic einen dazu senkrechten Vektor ergeben. Magnetic ist hierbei aber, davon abweichend, mit einem zusätzlichen geringen Anteil von 0,01 ebenfalls in Richtung Abszisse festgelegt. In der Simulation baut jede nachfolgende Zeile auf der vorhergehenden auf. Bei der Korrektur von magnetic wird aber electric und Radius aus der aktuellen Zeile verwendet. Während hier Radius aus der Summe der Komponentenquadrate von electric besteht ist es auch möglich davon noch die Quadratwurzel zu ziehen. Entscheidend dafür ob die Simulation überhaupt funktioniert ist lediglich das Monotonieverhalten und die Gleichheit bei EINS.

© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.

Die beiden Komponenten a und b von electric verhalten sich jetzt wie ein Sinus-Kosinus-Paar mit dem Radius von 1 - er weicht teils geringfügig ab und oszilliert auch wegen der vorhin erwähnten Komponente in Richtung der Abszisse in magnetic. Magnetic steht mit kleinerem Betrag stets etwa senkrecht zu electric.



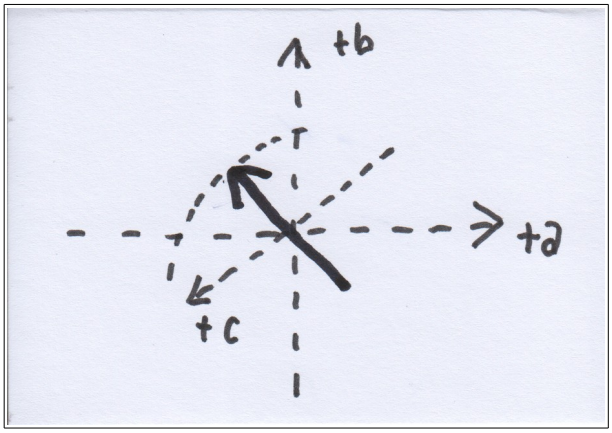
p.e.a und p.e.b sind als zeilenweise Punktepaare in ein X-Y-Diagramm eingetragen worden. Wie gewöhnlich wenn man das mit Sinus und Kosinus macht entsteht ein Kreis. Der Kreis hier weist eine Schwingung des Radius (wie oben erklärt auf). Das Verfahren ist in Puncto Radius selbstregulierend und behält Radius und Punkteabstand selbst bei vielen Umdrehungen noch bei.

Das hier im Beispiel mit der Double-Buffer-Technik simulierte Pendel kann sowohl hin-und-her pendeln als auch jeden Punkt auf dem Kreis erreichen. Für die Simulation im zweidimensionalen Raum wird es um die Komponenten p.e.c und p.m.c erweitert. Auch vier und mehr Komponenten pro Vektor sind möglich.

© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.

Skyrmionen in 2D

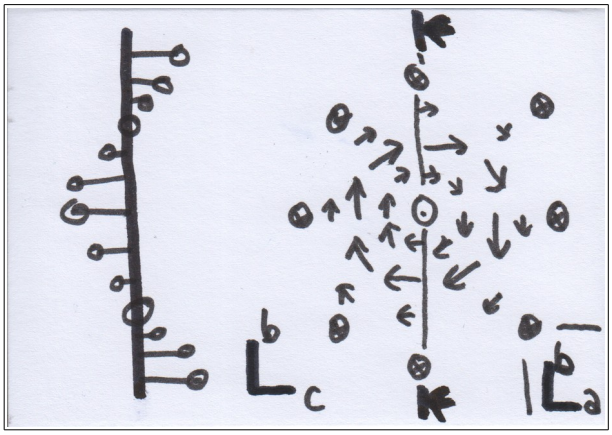
Um das Konzept aus [WELLEN_M] von 1D auf 2D zu erweitern wird das Pendel in jedem Pixel von den zwei Koordinaten {a, b} auf die drei Koordinaten {a, b und c} erweitert. Diese Koordinaten des Pendels sind von den Koordinaten {x und y} des 2D Pixelrasters unabhängig. In Zeichnungen in denen Pendelkoordinaten und Rasterkoordinaten gemeinsam, übereinander gezeichnet, vorkommen, wird a in der Richtung von x und b in der Richtung von y eingezeichnet. „c“ ist die besondere Pendel-Koordinate mit der Rückstellbeschleunigung. Immer wenn c ungleich NULL ist erhöht der Simulator die Bewegungsgeschwindigkeit des Pendels in Richtung NULL. Es sind zwei Möglichkeiten des Nullpunktes von c bekannt:



1. NULL gilt für c = -1 (Pol)
2. NULL gilt für c = 0 (Äquator)

In einem leeren Raum zeigen alle Pendel in eine Richtung. In einem Raum mit Wellen weichen die Pendel wellenförmig von dieser NULL-Richtung ab. In einem Raum mit einem Teilchen der Sinus-Gordon-Gleichung zeigt das Pendel in der Mitte des Teilchens in die entgegengesetzte Richtung von NULL und außen in die Richtung von NULL; dazwischen gibt es einen stetigen Übergang. Diese Teilen sind im 1D-Raum spezielle Solitonen und im 2D-Raum Skyrmionen. Skyrmionen sind aus einem Medium bekannt das aus Pixeln magnetischer Dipolnadeln in ferromagnetischen Werkstoffen besteht.

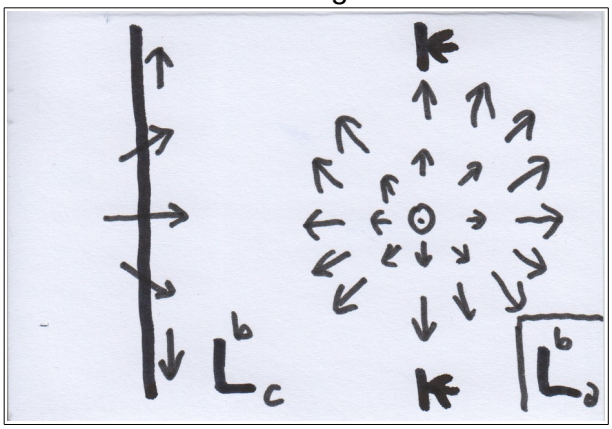
Das nebenstehende Bild zeigt den Aufbau eines Skyrmions mit c = -1 als NULL. Links steht eine Schnittdarstellung der Ebene b-c und rechts eine der Ebene a-b. Es wird in [ANTISKYRMION] benutzt um Kräfte zu simulieren und durch die Landau-Lifschitz-Gilbert-Gleichung (LLG) erzeugt. Mit der hier noch zu entwickelnden Pendel-Simulation kann es nicht erzeugt werden, denn es treten stark kontrahierende Kräfte auf und das Teilchen zieht sich zusammen und verschwindet. Um es dennoch zu retten kann man mit einem ungeraden Gitter arbeiten muss aber die Koordinaten zwischen Pendel und Pixelraster gleich halten. Das Arbeiten mit der LLG, im Gegensatz dazu, ist motiviert durch praktische Anwendungen mit ferromagnetischen Werkstoffen. Kräfte zwischen Skyrmionen wären mit



© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.

der Simulation obiger Pendelvorschrift nicht möglich da die Rückstellbeschleunigung die Auslenkung der Pendel in einem Skalarfeld mit abnehmender Entfernung „r“ exponentiell als $\varphi = e^{-r}$ gegen NULL bringt. Diese Kräfte berücksichtigt die LLG deshalb mit einem Layer des magnetischen Feldes (in diesem Kontext wirklich vorhanden) der aus verschiedenen Termen besteht und mit dem Layer für die Winkel der magnetischen Dipole wechselwirkt. Skyrmionen dieses Pol-Typs unterteilen sich wieder in Kink und Antikink. Ein Spin kann nicht modelliert werden.

Der andere Typ Skyrmion, mit c = 0 als NULL ist ebenfalls bereits bekannt, er taucht z.B. im [VOLOVIK] - Dokument auf Seite 6 auf, das der Autor aber nicht ausgewertet hat. Das Pendel in der Mitte des Skyrmions zeigt auf einen der beiden Pole während die Pendel außerhalb auf den Äquator zeigen und mit dem verbleibenden Freiheitsgrad einen Stern bilden. Die Rechenschritte des Simulators arbeiten wieder nach dem Double-Buffer-Prinzip. Das Pendel ist ein Vektor mit den Koordinaten {a, b und c} und hat die Nachbarindexe left, right, up und down. Nach dem Generator-Iterator-Prinzip werden erst ein oder zwei Skyrmionen im Simulator angelegt (Generator). Diese entsprechen aber nicht dem Zustand der minimalen Energie (zu schwierig zu berechnen). Während der Simulation stabilisieren sie sich unter Aussendung von Wellen. Da Terminieren schwierig ist werden sie in der vorliegenden Simulation vom Rand zurückgeworfen. Als Iteration werden nun die folgenden Schritte abwechselnd durchgeführt:



1. Double-Buffer-Schritt 1: $\overline{e}_{i+1} := \overline{e}_i + \overline{m}_i$
2. Double-Buffer-Schritt 2: $\overline{m}_{i+1} := \overline{m}_i + 0.25 * (\overline{e}_{left,\,i+1} + \overline{e}_{right,\,i+1} + \overline{e}_{down,\,i+1} + \overline{e}_{up,\,i+1}) - \overline{e}_{i+1} + \overline{e}_{i+1} * (1 - |\overline{e}_{i+1}|) - \{0; 0; 0{,}01\} * \overline{e}_{i+1}$

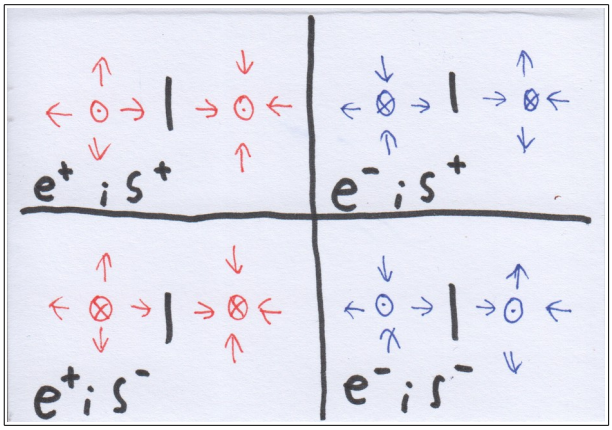
Im Schritt 2 hebt sich $\overline{e}_{i+1}$ auf. Die Rechnung mit den Nachbarn ist eine zweite Ableitung der Pendelposition (Wölbung). Der letzte Summand ist die Rückstellbeschleunigung und besteht aus electric bei dem außer c die beiden anderen Komponenten weg gefiltert werden. Dies ist die erweiterte obige Pendelrechenvorschrift mit dem Einfluss der benachbarten Pendel und der Rückstellbeschleunigung. Sie kann für einen Raum mit 3 Dimensionen {x, y und z} auf ein Pendel mit 4 Koordinaten {a, b, c und d} erweitert werden. Es können dann bei hoher CPU-Zeit noch kleine 3D Skyrmionen simuliert werden.

Es sollen jetzt nochmal die verschiedenen Möglichkeiten der Initialisierung (des Generators) am Anfang der Simulation unterschieden werden. Die Skyrmionen des 3D-Raumes mit Äquator als NULL können nach Ladung e^+ und e^- entsprechend Skyrmion und

© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.

Antiskyrmion und nach Spin entsprechend dem Pol in der Mitte als c = -1 und c = +1 unterschieden werden. Die Pendel am Rand des Skyrmions weisen an entgegengesetzten Seiten des Teilchens einen Unterschied von +/- π entsprechend +/- 180° auf. Umrundet man das Teilchen einmal so durchfährt der Winkel der Pendel einen Vollkreis. Die Richtung des Kreises ist bei Skyrmion und Antiskyrmion entgegengesetzt. Dabei kann man ein beliebiges Winkeloffset dazu addieren solange nur die Differenzen der Winkel untereinander bestehen bleiben (Windrichtung). Die Begriffe Skyrmion und Antiskyrmion stehen für die elektrische Ladung und bedeuten die Umkehr einer der beiden Koordinaten im Teilchen für das Antiskyrmion. Sie bleibt bei jeder Änderung der oben erwähnten Windrichtung bestehen. Spin bedeutet für das eine Vorzeichen dass die Ladung und der Pol des mittleren Pendels z.B. gleich sind während sie für das andere Vorzeichen verschieden sind.

Die Abbildung listet die verschiedenen Kombinationen von Ladung und Spin noch einmal auf. Punkt steht dafür, dass das mittlere Pendel aus dem Papier herausragt während es für Kreuz in das Papier herein zeigt. Für jede Ladung gibt es beliebig fein unterteilte Windrichtungen. Es sind jeweils zwei entgegengesetzte Windrichtungen eingezeichnet. Im kommenden Abschnitt geht es dann um mehrere Skyrmionen in einem Simulator. Die Windrichtungen der Skyrmionen müssen derart übereinstimmen, dass zwischen ihnen die Pendelwinkel stetig sind. Fall es eine Stelle gibt bei der im Generator zwei Pendel entgegengesetzt aneinander stoßen so entsteht dort ein neues Teilchen. Bei geringen Abweichungen richten sich die Pendel aneinander aus.



Kräfte zwischen den Skyrmionen

Ein Raum ohne Skyrmionen oder mit nur einem Skyrmion ist frei von Kräften. Zwischen zwei oder mehreren Skyrmionen treten beschleunigende elektrostatische Kräfte auf. In einem Raum ohne Felder sind alle Pendel auf den Äquator ausgerichtet und sie zeigen alle in eine beliebige aber unter allen gleiche Richtung, z.B. {a=1, b=0 und c=0}.

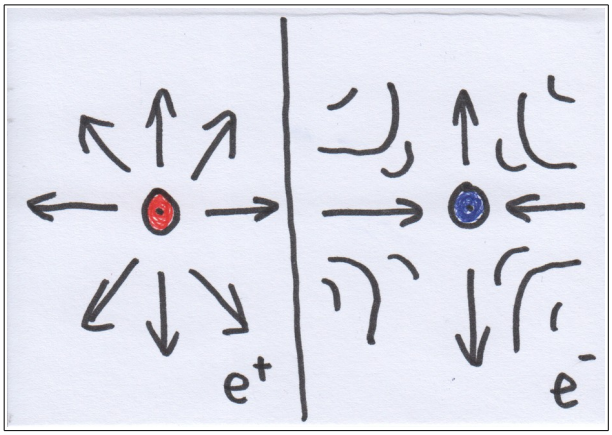
Zwischen den ähnlich ausgerichteten Pendeln lassen sich echte Feldlinien denken. Davon kann man elektrostatische Feldlinien ableiten. Aus den echten Feldlinien lassen sich die abgeleiteten elektrostatischen Feldlinien auf direktem Weg schwer ablesen. Am besten man verwendet dazu einen Probekörper, ein geladenes Skyrmion.

Zunächst soll mit dem Generator einmal ein Skyrmion allein in den Raum gesetzt werden. Ein anderes mal soll es ein Antiskyrmion sein. Das alleinige Teilchen richtet sogleich alle umgebenden Pendel gemäß seinem Inneren aus. Das soll so weiter gehen bis an das

© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.

Ende des Raumes. Der Raum wäre, wie im codierten Beispiel, an den Enden begrenzt aber nicht terminiert.

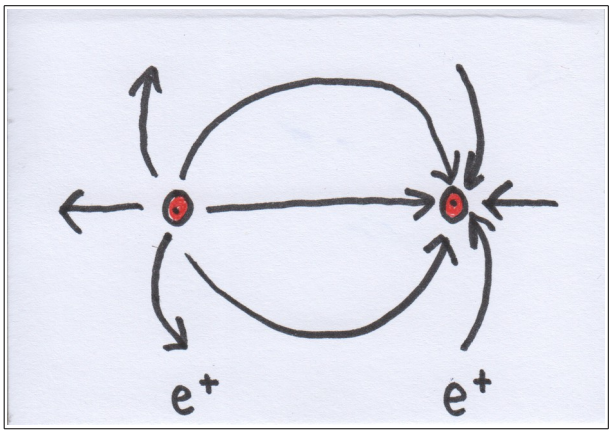
Zu Beginn würde also die nicht optimale Form des Teilchens noch als Welle abgestrahlt und vom Rand etwas reflektiert. Möglich wäre auch ein Donut als Raum. Die Welle würde darin noch ewig weiter kreisen. Weil auf der gegenüberliegende Seite im Raum die Feldlinien aufeinanderstießen entstünde ein Spiegelteilchen gleicher Ladung aber umgekehrter Windrichtung.



Zurück zum begrenzten Raum. Um zwei Teilchen zu kombinieren gibt es zwei Möglichkeiten: Gleiche Ladung und entgegengesetzte Ladung. Prinzipiell kann jedes der beiden Teilchen nicht, wie oben, die Pendel seiner Umgebung nach seinem Inneren ausrichten. Das hängt damit zusammen dass beide Teilchen um die dazwischenliegenden Pendel konkurrieren. Jedes der beiden Teilchen merkt, dass da etwas im Raum ist und wird bis in sein Innerstes von dem anderen Teilchen deformiert. Die Mittelpunkte in den Teilchen verschieben sich und die Teilchen werden beschleunigt. Sie verlagern sich auf benachbarte Pendel im Raum.

Umfährt man ein Gebiet mit mehreren Teilchen so beschriebt die Draufsicht auf den Winkel des Teilchens eine Anzahl von Vollkreisen die der Summe der Ladungen der eingeschlossenen Teilchen entspricht. Dadurch und durch die Entfernung von der Mitte wird das ausgesendete elektrostatische Feld bestimmt. Es breitet sich mit endlicher Geschwindigkeit aus.

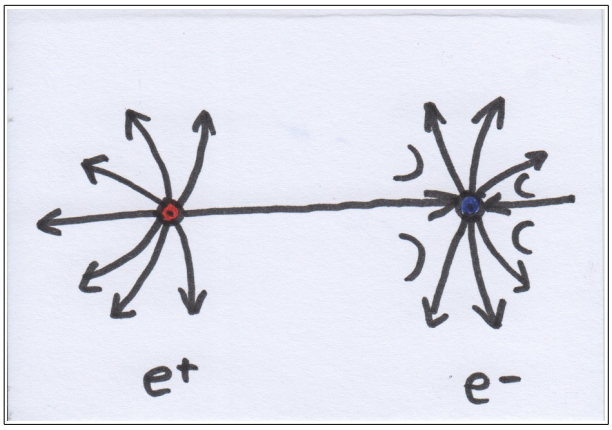
Nebenstehendes Beispiel zeigt zwei gleich geladene Teilchen entgegengesetzter Windrichtung die einander abstoßen. Die Winkel der Pendel sind stetig. Die Spins sind gleich. Zwischen den Spins kommt es zu keiner Wechselwirkung da die Teilchen sich voneinander entfernen. Die umschlossene Winkeländerung entspricht zwei Vollkreisen. Statt der beispielsweise positiven Ladung arbeitet die negative Ladung gleich.



Begegnen sich unterschiedlich geladene Teilchen so wirken anziehend beschleunigende Kräfte. Die Windrichtung ist auch wieder entgegengesetzt. Das Beispiel funktioniert nicht nur wenn beide Teilchen nicht waagerecht oder senkrecht zueinander angeordnet sind

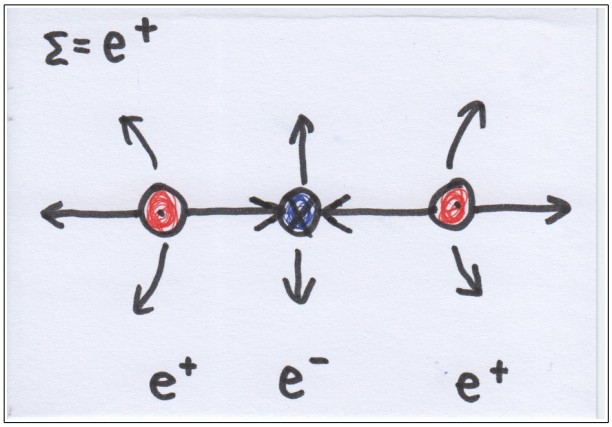
© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.

sondern auch wenn sie einen beliebig schrägen Winkel zueinander einnehmen. Die umschlossene Winkeländerung ist NULL, beide Ladungen heben einander auf.



In diesem Beispiel haben die Teilchen unterschiedliche Spins. Die mittleren Pendel zeigen auf den gleichen Pol. Bei der Kollision der Teilchen geht in der Mitte der Teilchen die c-Komponente des Pendels nicht durch NULL. Es entsteht ein Parapositronium, ein minimales Atom. Es annihiliert sehr schnell.

Möglich sind auch gleiche Spins der beiden Teilchen. Damit zeigen die mittleren Pendel auf entgegengesetzte Pole. Bei der Kollision der Teilchen geht in der Mitte der Teilchen die c-Komponente des Pendels durch NULL. Somit muss in der Nähe der Mitte die Wölbung NULL sein. Es entsteht ein Orthopositronium, die zweite Variante des Positroniums. Es annihiliert um ein Vielfaches langsamer als ein Parapositronium.



Während bei einem Orthopositronium die NULL in der Mitte beider Teilchen liegt, kann man sie davon weg verlagern wenn man drei Teilchen kombiniert. Das entstandene Konstrukt ist stabiler als ein Orthopositronium. Falls man die Rückstellbeschleunigung für c sehr klein macht, so entsteht ein recht großes Teilchen und man braucht entsprechend viele Pixel bei der Simulation. Bei vergrößerten Teilchen ist das Gebilde aber stabil und ähnelt einem Proton.

Zumindest Repulsion und Attraktion von Teilchenpaaren lässt sich im 3D-Raum auch simulieren. Das ist in dem codierten Beispiel nicht enthalten. In 2D ist der Querschnitt des Feldes zwischen den Teilchen eine Linie während es sich bei einem 3D Raum um eine kreisförmige Fläche handelt. Ein Proton würde hingegen für 3D einen Simulator größer als ein PC erfordern. Das liegt nicht in den Möglichkeiten des Autors.

Wertung

Die vorliegende Simulation zeigt dass der zur Verfügung stehende Raum tatsächlich, wie in [ZUSE_RR], aus einem Gitter kleinster Teilchen bestehen könnte. Somit gäbe es absolute Positionen im Raum - nur es gäbe keine Möglichkeit eine Landmarke auf ihnen zu setzen. Alle Teilchen sind stets beweglich. Dazu gibt es in der Relativitätstheorie auch keinen Widerspruch weil dort nur gesagt wird dass absolute Positionen nicht messbar sind (und sie daher aus der Rechnung entfernt wurden). Das Michelson-Morley-Experiment

© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.

sollte überprüfen, ob sich der Einfluss des absoluten Raumes (Äther) indirekt messen lässt. Es kam heraus dass eine absolute Geschwindigkeit zwar nicht messbar ist, aber dass sich bei absoluter Geschwindigkeit die Experimentieranordnung derart verbiegt dass sie deren Einfluss im Messergebnis aufhebt. Weiter erweckt der rechtwinklige Aufbau des Gitters den Eindruck bevorzugter Richtungen im Raum. Jedoch sieht im Simulator von weitem alles „rund“ aus.

Es könnte durchaus sein, dass ein Gott diesen Raum erst geschaffen hat. Somit könnte er die drei Richtungen Länge, Breite und Höhe willkürlich festgelegt haben. Allerdings scheint es dem Autor, wenn er sich in seinem Simulator so als Gott versucht, schwierig zu sein, ein Skyrmion zur Laufzeit zu erzeugen. Damit hätte Gott zwar großen Einfluss zu Beginn der Welt aber mittendrin könnte er nur wenig manipulieren.

In dem bereitgestellten FDTD-Simulator werden öfters Kreise simuliert. Ein einfachere Form wäre der numerische Kreiskontrakt mit (re := 1; im := 0):

- re := re + im/2
- im := im - re/2

der, wiederum nach dem Double-Buffer-Prinzip ausgeführt, im Bereich der komplexen Zahlen, einen etwas unrunden Kreis erzeugt. Dieser Kreis wäre eine Art Urform des Funktionspaares sin/cos, das in Wirklichkeit nicht existiert sondern ein technisches Hilfsmittel ist um die in dem echten Raum in vielen Einzelschritten errechneten künftigen Werte mit Funktionstabellen schon vorher zu kennen.

Bei der oben erwähnten und auch codiert bereitgestellten Annihilation von Teilchen tritt kein Lichtteilchen (Photon) auf. Photonen scheinen auch in Wirklichkeit nicht zu existieren. In der Quantenmechanik treten Quanten im Zusammenhang damit auf, dass ein Elektron seine Schale wechselt. Die damit verbundene Energie führt zu Licht bestimmter Farbe. Stets handelt es sich um Wellenlängen nahe des sichtbaren Lichtes. Nie Langwellen oder Gammaquanten. Es könnte sich eventuell nur um Versuche mit einer größeren Zahl von Elektronen handeln, nicht um ein einziges Elektron im ganzen Versuch. Grundlage der „Photonen“ ist der Photoelektrische Effekt, bei dem ein Elektron aus dem Atom bei Auftreffen von Licht, das kürzer als eine bestimmte Wellenlänge ist, freigesetzt wird. Dabei verschwindet das Photon. Zweifel lassen sich wie folgt anbringen: Sichtbares Licht umfasst beinahe den gesamten Bereich möglicher Quantensprünge im Atomkern. Ein solches Photon hat eine Wellenlänge von 500 nm (Nanometer) und sollte mindestens einen Wellenzug beinhalten. Das gesamte Atom ist nicht größer als 0,5 nm also tausendmal kleiner als das Photon. Somit ist es zu klein um das Photon zu schlucken. Hier eine alternative Erklärung: Zunächst kreist das Elektron um das Proton. Dann durchdringt eine Lichtwelle das Atom. Das ist eine transversale elektromagnetische Welle (Scherwelle). Ähnlich dem Prinzip eines elektromagnetischen Synchronmotors bestimmt die Lichtwelle die Drehzahl des Elektrons indem sie das Elektron hin- und herschiebt. Das

© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.

kann man nachstellen wenn man eine Kugel in eine Schüssel legt und die Schüssel hin- und herschiebt so dass die Kugel kreist. Ab einer bestimmten Frequenz des Lichtes hat das Elektron eine so hohe Drehzahl dass es die Schale verlässt. Der Photoelektrische Effekt tritt messbar sofort auf während das Elektron von der Lichtwelle erst beschleunigt werden muss. Allerdings wäre die Zeit zum Beschleunigen so kurz dass man sie nicht messen könnte.

In dem obigen Abschnitt über das Proton kommt heraus dass es aus zwei gleichen und einem davon verschiedenen Elektron bestehen könnte. Seine Stabilität im 3D-Raum konnte nicht ermittelt werden. Falls es stabil wäre so wären „Quarks“ in Wirklichkeit Elektronen deren Feld durch die räumlich gedrungene Anordnung sehr stark verbogen ist. Tatsächlich haben Quarks eine ähnliche Masse wie Elektronen auch und nur die Kombination zu einen Proton macht das Proton tausendfach schwerer.

Java-Formativ

```
// Begin file ZSpace.java
import java.io.*;
import java.awt.*;
import java.awt.image.*;
import javax.imageio.*;

// Command line example to render with MENCODER under LINUX.
// mencoder mf://img/*.png -mf fps=25:type=png -ovc lavc -lavcopts vcodec=mpeg4 -o ZSpace.avi

/**
 * FDTD simulator to generate a picture sequence in subdir .../img/*.png which
 * shows Zuse's Calculating Space with Skyrmions.
 *
 * Grayscale shows the sum of the electrical field components x and y.
 * Arrows show the real field lines of the pendulums.
 *
 * Provide a subdirectory ./img/ and compile and start with javac ZSpace.java and java ZSpace.
 *
 * @author Claus Wimmer
 * 2018-11-05
 *
 * Copy without fees granted.
 */
public class ZSpace {

    static final int
        width = 200,    // Width of the calculating space in pixels.
        height = 200,   // Height of the calculating space in pixels.
        zoom = 2,       // Number of rendered picture pixels per dimension and space pixel.
        numBatches = 8, // Number of parallel threads on render time.
        render = 4,     // Number of simulation steps for one render step.
        loop = 500,     // Number of render steps at all.
        arrowSize = 15; // Size of the real field line arrows.
    // Enables measuring of the consumed render time.
    static long time = System.currentTimeMillis();

    static int
        simulationStepIndex = 0,    // Counter for the simulation step index.
        renderIndex = 0;            // Counter for the index of the rendered image (seen in filename).
    // Arrows over the picture showing the real field lines.
    static Arrow arrows[] = new Arrow[((width * zoom) / arrowSize) * ((height * zoom) / arrowSize)];
    // Simulation area.
    static Space space = new Space();
    // Thread wrappers to simulate and render parallel.
    static Thread threads[] = new Thread[numBatches];
    /**
     * Data structure containing a pendulum with 3 coordinates which is equal to
     * 2 angles. Must not be initialized during each step because of CPU time
     * consumption.
     */
    static class Vector3 {
        // Components.
        public double a = 0, b = 0, c = 0;
        // Constructor with length of zero. Much CPU time consumption.
        public Vector3() {}
        // Initializing constructor. Much CPU time consumption.
        public Vector3(double a, double b, double c) {
            this.a = a; this.b = b; this.c = c;
        }
        // Adds the argument and multiplies its copy before addition.
        public void multAdd(double k, Vector3 arg) {
```

© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.

```
        this.a += k * arg.a; this.b += k * arg.b; this.c += k * arg.c;
    }
    // Multiply each component by factor.
    public void mult(double arg) {
        a *= arg; b *= arg; c *= arg;
    }
    // Returns the EUCLIDIAN length.
    public double getLength() {
        return Math.sqrt(a * a + b * b + c * c);
    }
    // Make the EUCLIDIAN length to one but to nothing if it is Zero on call.
    public void norm() {
        // Copy of the length.
        double l = getLength();
        // Do nothing to avoid division by zero on next step.
        if(0 == l) return;
        // Normalize each component.
        a /= l; b /= l; c /= l;
    }
    // Set to a length of zero. Purpose: Avoid CPU consuming constructor.
    public void reset() {
        a = b = c = 0.0;
    }
    // Cross product of two pendulums to calculate the electrical field.
    public void crossProd(Vector3 arg0, Vector3 arg1) {
        // Make a buffer copy.
        double
            a_0 = arg0.a, b_0 = arg0.b, c_0 = arg0.c,
            a_1 = arg1.a, b_1 = arg1.b, c_1 = arg1.c;
        // Calculate and assign.
        a = b_0 * c_1 - c_0 * b_1;
        b = c_0 * a_1 - a_0 * c_1;
        c = a_0 * b_1 - b_0 * a_1;
    }
    // Calculate the angle of the projection of the pendulum with c=zero.
    public double getPhi()
    {
        if(0.0 != a || 0.0 != b) {
            return Math.atan2(b, a); // Arcus angle.
        }else {
            return 0.0; // Not defined.
        }
    }
}
/**
 * Pixel in 2D space.
 */
static class Pixel {
    public int
        x, y, // Position of the pixel in space.
        alpha,red, green, blue, argb; // Colors of the pixel when rendered as picture.
    public Vector3
        electric = new Vector3(), // Position of the pendulum.
        magnetic = new Vector3(), // Turning speed of the pendulum.
        a_0 = new Vector3(), a_1 = new Vector3(); // Helper variables.
    public Pixel left, right, up, down; // Neighbors.
    // Constructor storing the position.
    public Pixel(int x, int y) { this.x = x; this.y = y; }
}
/**
 * Thread wrapper for parallel rendering. Helps to render a number of pixels
 * in double buffer mode.
 */
static class Batch {
    // Split the number of pixels into threads.
    public int len = (width * height) / numBatches;
    // Pixel references.
    public Pixel pixels[];
    // Constructor with thread index.
    public Batch(int i) {
        // Take reminding pixels to the last batch.
        if(i == numBatches - 1)
        {
            len += (width * height) % numBatches;
        }
        // Initialize once during startup.
        pixels = new Pixel[len];
    }
}
/**
 * 2D space simulation area.
 */
static class Space {
    // Instances of pixels in space.
    public Pixel pixels[] = new Pixel[width * height];
    // Instances of render batches.
    public Batch batches[] = new Batch[numBatches];
    // Constructor called once on startup.
    public Space() {
        // Initialize render batches.
        for(int i = 0; i < batches.length; ++i) batches[i] = new Batch(i);
        // Create pixels and store the in the space and in the render batches.
        { // Begin batch block.
            int batchindex = 0, pixelindex = 0;

            for(int x = 0; x < width; ++x) {
```

© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.

```
                for(int y = 0; y < height; ++y) {
                    Pixel p = new Pixel(x, y);
                    pixels[y * width + x] = p;
                    // Count batch index.
                    if(batches[batchindex].len <= pixelindex) {
                        pixelindex = 0;
                        ++batchindex;
                    }
                    // Store.
                    batches[batchindex].pixels[pixelindex++] = p;
                }
            } // End second for.
        } // End batch block.
        // Assign references of the borders of the pixels.
        for(Pixel p : pixels) {
            // Border pixels have itself as neighbor on the side or on the two sides of the border.
            p.left = p.right = p.up = p.down = p;
            // Otherwise neighbors are assigned by x and y  index.
            if(0 < p.x) p.left = pixels[p.y * height + p.x - 1];
            if(width - 1 > p.x) p.right = pixels[p.y * height + p.x + 1];
            if(0 < p.y) p.up = pixels[(p.y - 1) * width + p.x];
            if(height - 1 > p.y) p.down = pixels[(p.y +  1) * width + p.x];
        } // End third for.
        // Associate arrows with pixels.
        for(int x = 0; x < (width * zoom) / arrowSize; ++x) {
            for(int y = 0; y < (height * zoom) / arrowSize; ++y) {
                arrows[y * ((width * zoom) / arrowSize) + x] =
                    new Arrow(pixels[width * ((y * arrowSize) / zoom) + (x * arrowSize) / zoom], x * arrowSize, y * arrowSize);
            }
        } // End fourth for.
    }
    // Initial generator of a skyrmion in space. More than one skyrmion in space
    // is made by addition of the particles and needs normalization once
    // on program startup.
    public void setParticle(int x, int y, double phi, int polarity, int spin) {
        // Makes wind direction of the whole space to be compatible with the particle.
        for(Pixel p: pixels)
        {
            final double R = 10.0; //Size.
            double
                // Distance of the current rendered pixel to the middle of the particle.
                r = Math.sqrt(Math.pow((double)p.x - x, 2.0) + Math.pow((double)p.y - y, 2.0)),
                // Angle of the pendulum relating to its projection with c=zero.
                _phi = Math.atan2((double)p.y - y, (double)p.x -x) + phi,
                // c component of the pendulum.
                corpus = Math.max(0.0, (R - r) / R * Math.PI / 2.0);
            // Data of the new pendulum.
            Vector3 electric = new Vector3(
                Math.cos(_phi) * Math.cos(corpus) * polarity,
                Math.sin(_phi) * Math.cos(corpus),
                Math.sin(corpus) * spin);
            // Make farer pixels shorter in length to enable merging of
            // pixels in space with compatible wind direction.
            if(r > R) electric.mult(R / r);
            // Merge one or more pixels.
            p.electric.multAdd(1.0, electric);
        }
    }
}
/**
 * Marker arrow to show real field lines.
 */
static class Arrow {
    // Position over picture.
    int x, y;
    // Referenced data source.
    Pixel p;
    // Constructor.
    public Arrow(Pixel p, int x, int y) {this.p = p; this.x = x; this.y = y; }
    // Draw the arrow to the graphics context of the picture.
    public void render(Graphics g) {
        // Do not draw an arrow with undefined angle.
        if(0.999 < Math.abs(p.electric.c) || x == 0 || y == 0) return;

        double
            phi = p.electric.getPhi(),  // Angle of the arrow.
            alpha = phi + Math.PI,      // Angle of the arrow with opposite direction.
            _f = 0.4 * arrowSize, _sin = Math.sin(phi) * _f, _cos = Math.cos(phi) * _f; // Helper variables.
        // Line of the arrow.
        g.drawLine((int)(x + _cos), (int)(y + _sin), (int)(x - _cos), (int)(y - _sin));
        // One head of the arrow.
        g.drawLine((int)(x + _cos), (int)(y + _sin),
            (int)(x + (_cos +  Math.cos(alpha + 0.5) * _f)),
            (int)(y + (_sin +  Math.sin(alpha + 0.5) * _f)));
        // The other head of the arrow.
        g.drawLine((int)(x + _cos), (int)(y + _sin),
            (int)(x + (_cos + Math.cos(alpha - 0.5) * _f)),
            (int)(y + (_sin + Math.sin(alpha - 0.5) * _f)));

    }
}
/**
 * Double buffer step one. Turns the pendulum.
 */
static class Turn extends Thread {
    // Back reference.
```

© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.

```
    public Batch batch;
    // Constructor.
    public Turn(Batch batch) {
        this.batch = batch;
    }
    // Task of the thread.
    public void run() {
        try {
            for(Pixel p : batch.pixels) {
                // Turn by magnetic speed.
                p.electric.multAdd(1.0, p.magnetic);
            } // End for.
        } catch(Exception ex) {
            // Error handling.
            System.out.println("Exception in Turn thread = " + ex.getMessage());
            ex.printStackTrace();
        }
    }
}
/**
 * Double buffer step two. Increments the turning speed.
 */
static class Derive extends Thread {
    // Back reference.
    public Batch batch;
    // Constructor.
    public Derive(Batch batch) {
        this.batch = batch;
    }
    // Task of the thread.
    public void run() {
        try {
            for(Pixel p : batch.pixels) {
                // Second derive done with neighbors.
                p.magnetic.multAdd(0.25, p.left.electric);
                p.magnetic.multAdd(0.25, p.right.electric);
                p.magnetic.multAdd(0.25, p.up.electric);
                p.magnetic.multAdd(0.25, p.down.electric);
                // Begin merged subtrahend from second derive and track radius correction.
                p.magnetic.multAdd(-p.electric.getLength(), p.electric);
                // End merged subtrahend.
                p.magnetic.c -= 1e-2 * p.electric.c;
                // Calculate background colors by electric field.
                preRender(p);
            } // End for each pixel.
        } catch(Exception ex) {
            // Error handling.
            System.out.println("Exception in Derive thread = " + ex.getMessage());
            ex.printStackTrace();
        }
    }
}
/**
 * Main entry of the script. Does not take command line arguments.
 */
public static void main(String args[]) throws Exception {
    clear(); // Delete old image files.
    // Add first particle.
    space.setParticle(width / 2 + 15, height / 2 + 15, 0*-Math.PI / 4.0, +1, +1);
    // Add second particle.
    space.setParticle(width / 2 - 15, height / 2 - 15, -Math.PI / 2.0, -1, +1);
    for(Pixel p : space.pixels){
        p.electric.norm(); // Make length of each pixel to be one.
        preRender(p); // Render the backgroudn of the first image.
    }
    // Render the first image.
    render();
    // Render the sequence.
    for(renderIndex = 1; renderIndex < loop; ++renderIndex) {
        // Derive task of the double buffer.
        for(int j = 0; j < numBatches; ++j) {
            Derive d = new Derive(space.batches[j]);
            threads[j] = d;
            d.start();
        }
        // Wait until ready.
        for(Thread t : threads) t.join();
        // Turn task of the double buffer.
        for(int j = 0; j < numBatches; ++j) {
            Turn t = new Turn(space.batches[j]);
            threads[j] = t;
            t.start();
        }
        // Wait until ready.
        for(Thread t : threads) t.join();
        // Render the image.
        render();
    }
    // Print rendering duration time.
    System.out.println("Duration = " + ((System.currentTimeMillis() - time) / 1000) + " seconds.");
}
/** Prepare the render process of a single pixel without field line arrows.
 * Pixel render preparation can be done parallel on some CPU kernels but arrow rendering not.
 * The buffered image class is not thread safe.
 * @param p Reference of the pixel.
 */
```

© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.

```
    static void preRender(Pixel p)
    {
        if(0 == (renderIndex % render)){ // Render each n-th frame.
            // Render the sum of both electrical field components.
            p.a_0.crossProd(p.left.electric, p.right.electric);
            p.a_1.crossProd(p.up.electric, p.down.electric);
            // Scale and make sum.
            double a = 2e0 * (p.a_0.c + p.a_1.c);
            // Transparency.
            p.alpha = 0xff;
            // Color components.
            p.red = p.green = p.blue = (int)(127 * (1.0 + 2e0 * a));
            // Set pixel in image.
            p.argb =
                ((clipByte(p.alpha) & 0xff) << 24) |
                ((clipByte(p.red) & 0xff) << 16) |
                ((clipByte(p.green) & 0xff) << 8) |
                (clipByte(p.blue) & 0xff);
        }
    }
    /**
     * Draw the picture.
     */
    static void render() throws Exception {
        // Render each n-th pixel.
        if(0 == (renderIndex % render)){
            // Make the file name. Renderer requires numbers incremented by one.
            File f = new File("img" + File.separator + String.format("%06d", simulationStepIndex++) + ".png");
            // Image object.
            BufferedImage image = new BufferedImage(width * zoom, height * zoom, BufferedImage.TYPE_INT_ARGB);
            // Copy pixel colors.
            for(Pixel p : space.pixels) {
                for(int xi = 0; xi < zoom; ++xi) {
                    for(int yi = 0; yi < zoom; ++yi) {
                        image.setRGB(zoom * p.x + xi, zoom * p.y + yi, p.argb);
                    } // End for zoom y.
                } // End for zoom x.
            } // End for each pixel.
            // Graphics context.
            Graphics g = image.getGraphics();
            // Extended graphics context.
            Graphics2D g2 = (Graphics2D)g;
            // Prepare the context for arrows.
            g.setColor(new Color(128, 196, 128));
            g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,  RenderingHints.VALUE_ANTIALIAS_ON);
            // Iterate through the arrows.
            for(Arrow  a : arrows) a.render(g);
            // Render the under title of the image.
            g.setColor(Color.lightGray); g.fillRect(15, 5, 300, 20);
            g.setColor(Color.BLUE); g.setFont(new Font(Font.SANS_SERIF, Font.BOLD, 12));
            g.drawString("Zuse Space frame[" + renderIndex + "/" + loop + "]", 20, 20);
            // Store the image.
            ImageIO.write(image, "png", f);
            // Debug console.
            System.out.println("Scene[" + renderIndex + "] rendered.");
        }
    }
    // Saturate an integer to a byte.
    static int clipByte(int arg) {
        return Math.max(0, Math.min(255, arg));
    }
    // Delete the old files.
    static void clear() {
        File files [] = new File("img").listFiles();
        for(File file : files) file.delete();
    }
}
// End file ZSpace.java
```

Literatur

- [ZUSE_RR] - Konrad Zuse: „Rechnender Raum“ in Elektronische Datenverarbeitung Band 8, 1967, S. 336-344; ftp://ftp.idsia.ch/pub/juergen/zuse67scan.pdf; In einem rechteckigen Pixelgitter breiten sich kreisförmige Wellen aus.
- [WELLEN_M] - Markus Dietrich, Hans-Josef Patt : „Wellenmaschine zur Demonstration und Messung harmonischer und anharmonischer Wellenphänomene (Solitonen)“; http://www.uni-saarland.de/fak7/patt/pdf/bre_diet.pdf; Auf einer Kette aus gekoppelten Pendeln bildet sich durch Überschlag ein fächerförmiges dauerhaftes Teilchen, ein Soliton.

© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.

- [ANTISKYRMION] - Wataru Koshibae & Naoto Nagaosa: „Theory of antiskyrmions in magnets“; https://www.nature.com/articles/ncomms10542; Bewegliche Skyrmionen werden mit der LLG simuliert und zur Annihilation gebracht. Es wird ein dem Skyrmion entgegengesetztes Antiteilchen mit entgegengesetzter Ladung vorgestellt.
- [VOLOVIK] - G. Volovik: „Topological defects and anomalies in topological superfluids“; http://inac.cea.fr/Pisp/mike.zhitomirsky/ts2015/Slides_Volovik.pdf; Obwohl nicht Thema des Dokumentes so werden Skyrmionen abgebildet mit einem Äquator als Nullstelle statt einem Pol als Nullstelle.

© Claus Wimmer 2018; Kopieren ohne Gebühr gestattet.