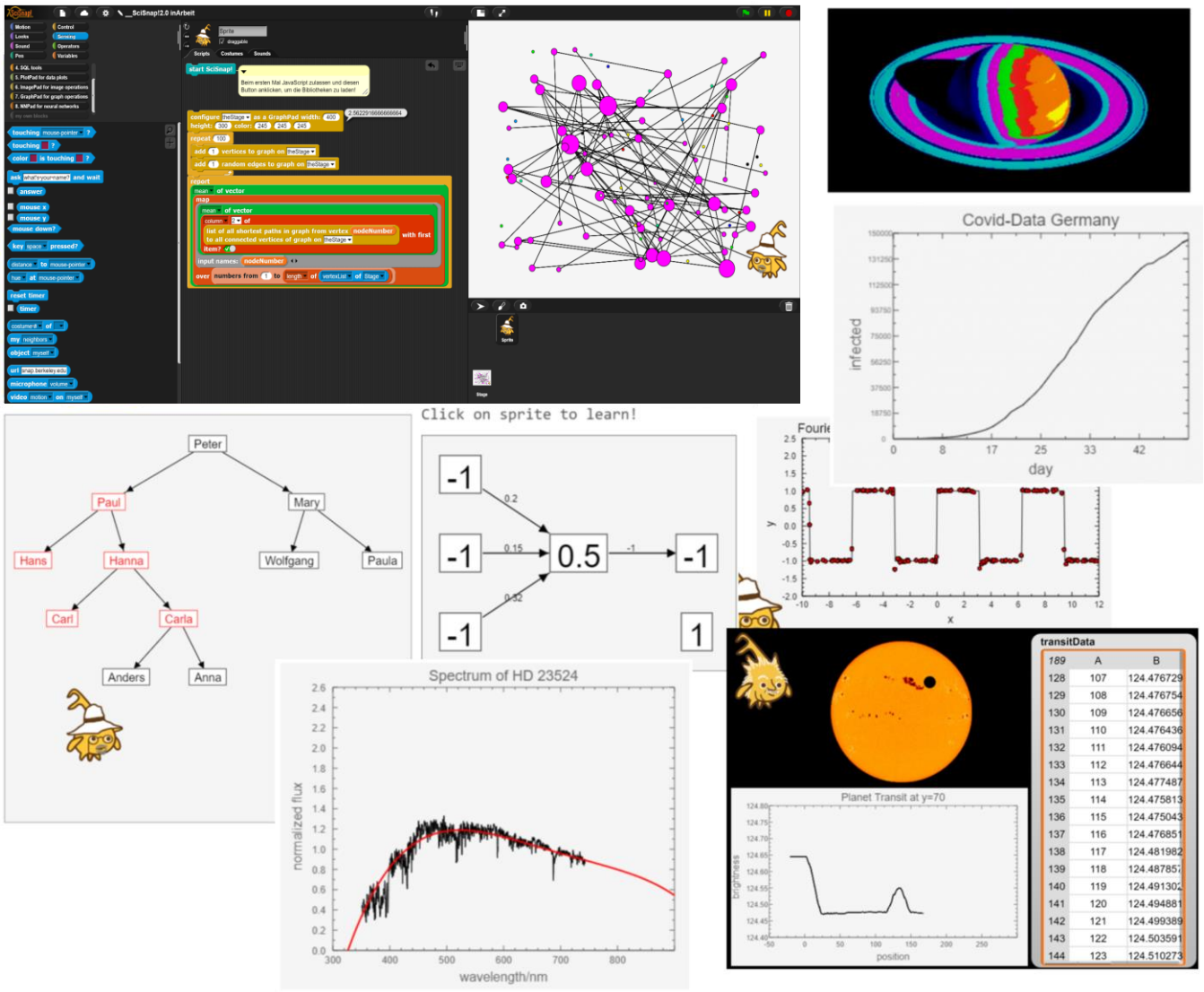
Eckart Modrow

Programmieren

mit SciSnap! Vers. 2



emodrow@informatik.uni-goettingen.de

Die Beispiele und dieses Skript können geladen werden von http://emu-online.de/SciSnap2Examples.zip

http://emu-online.de/ SciSnap2Examples.zip bzw.
http://emu-online.de/ProgrammierenMitSciSnap2.pdf

SciSnap!2 selbst unter
https://snap.berkeley.edu/snap/snap.html#present:Username=emodrow&ProjectName=SciSnap!2&editMode

Prof. Dr. Modrow, Eckart:

Programmieren mit *SciSnap!2*

Wenn Sie mit diesem Buch zufrieden sind und ihre Anerkennung in Form einer Spende zeigen möchten, dann können Sie das auf folgendem PayPal-Konto tun:

emodrow@emu-online.de
Verwendungszweck: SciSnap!-Buch

Vorwort

Die Entwicklung von informatischen Werkzeugen, insbesondere auch auf dem Gebiet der Programmiersprachen, hat in den letzten Jahrzehnten rapide Fortschritte gemacht. Beispielsweise sind grafische Programmiersprachen entwickelt worden, die es Anfänger*innen möglich machen, sehr schnell eigenständig kleine Projekte zu bearbeiten, ohne sich um Syntaxeigenheiten usw. groß kümmern zu müssen. Steht nur eine begrenzte Zeit für das Erlernen des Programmierens zur Verfügung, dann ist das ein entscheidender Fortschritt, weil sich das Verhältnis zwischen der Einübung des Umgangs mit dem Programmierwerkzeug (der Programmierumgebung incl. -sprache) und der inhaltlichen Arbeit praktisch umdreht. Entsprechend erfolgreich werden Werkzeuge wie *Scratch*[1] vom *MIT* oder *Snap!*[2] von der *UCB* in der Schule und an Universitäten eingesetzt.

Obwohl sich bei den Werkzeugen also einiges getan hat, sieht es bei den Inhalten in Programmierkursen erstaunlich unverändert aus. Es werden einfache „Aufgaben" gestellt, die weitgehend nur dazu dienen, den Umgang mit algorithmischen Grundstrukturen und Datenstrukturen einzuüben, ohne darüber hinauszuweisen. Dazu kommen oft Arbeitstechniken, die für große Projekte mit vielen Beteiligten ihren Sinn haben, die aber von den Anfänger*innen kaum als hilfreich erfahren werden. Als Beispiel mag das Zeichnen von Nassi-Shneiderman-Diagrammen[3] (Struktogrammen) dienen, die manchmal anzufertigen sind, bevor Skripte z. B. in Scratch entwickelt werden – und das, obwohl grafische Sprachen die algorithmische Struktur durch ihre Blöcke selbst veranschaulichen. Genau dafür wurden sie ja (u. a.) entwickelt. Die Begeisterung bei den Lernenden, z. B. einige Zahlen addieren zu lassen und dazu die Mehrwertsteuer zu berechnen oder als „lustige" Einlage alle „r" in einem Text durch „l" zu ersetzen und so „chinesische" Texte zu produzieren, kann man sich vorstellen. Folgerichtig sind die „Erfolge" im Programmierunterricht auch weitgehend unverändert. Weil eigenständige Problemlösungen mit dem daraus folgenden Produktstolz ebenso selten sind wie sinnvolle Anwendungen, die Teile der Lebenswelt der Lernenden erklären, fühlen sich oft nur die Lernenden angesprochen, die sich sowieso „für Computer" interessieren. Die anderen, also die meisten, erfüllen zwar auch die Anforderungen, aber sie fragen sich zu Recht: „Was soll das?"

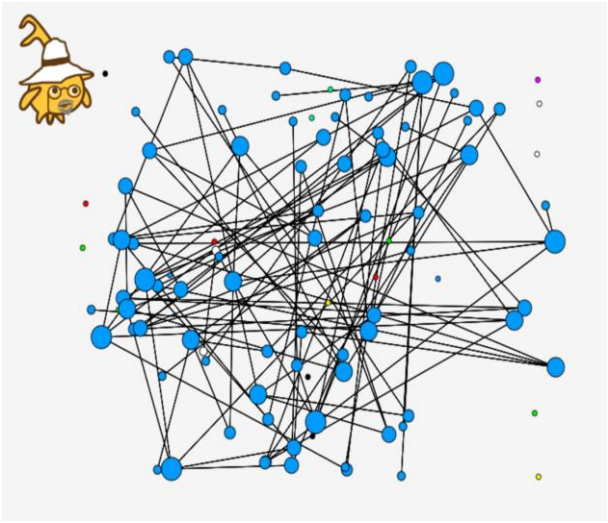
Der Einwand, dass man mit Anfänger*innen nun einmal nur elementare Beispiele behandeln kann, ist nicht von der Hand zu weisen. Obwohl mit den grafischen Sprachen viel mehr Zeit für das eigentliche Problemlösen zur Verfügung steht, sind die Algorithmen, die auf dieser Stufe des Lernens selbstständig entwickelt werden, ziemlich einfach. Meist handelt es sich um eine Befehlsfolge, oft innerhalb einer Schleife, die einige Alternativen nacheinander aufzählt: „Wenn dieses der Fall ist, dann tue das." Sollen solche Skripte trotzdem aussagekräftige Erfahrungen ermöglichen, dann müssen die – wenigen – verwendeten Elementarbefehle „mächtig" sein, und es ist bei den Lehrenden Fantasie gefragt, um „interessante" Probleme auf einem elementaren Niveau zu unterrichten.

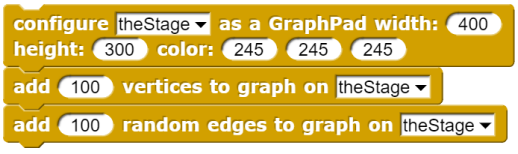
Das ist keine neue Erkenntnis: Zu Zeiten der Nassi-Shneiderman-Diagramme war es eine anspruchsvolle Aufgabe, auf einem technischen Gerät wie einem Bildschirm oder Drucker eine schräge Linie zeichnen zu lassen. Seit entsprechende Grafikbefehle entwickelt waren, handelte es sich um ein triviales Problem, das mit einer Anweisung erledigt wird. Noch vor wenigen Jahren war die Messwerterfassung mit Computern etwas für Spezialisten. Heute verfügt fast jede Schule über einen Satz von Sensorboards, mit denen Kinder arbeiten. Es ist eigentlich kaum zu verstehen, weshalb ausgerechnet im Bereich der Algorithmik die neuen Möglichkeiten kaum in Erscheinung treten. Es werden nach wie vor ein paar Zufallszahlen sortiert oder Worte in Großbuchstaben umgewandelt, statt mit ähnlich einfachen Skripten in Datenmengen zu „wühlen" oder Bücher oder Bilder auf charakteristische Strukturen zu durchsuchen. Um präzise zu sein: können mit einem Befehl in einer Datenmenge charakteristische Merkmale

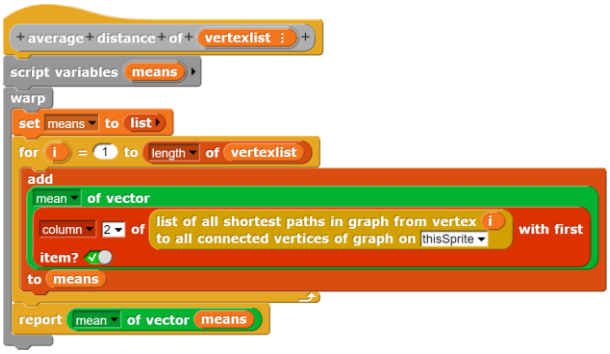
[1] https://scratch.mit.edu/
[2] https://snap.berkeley.edu/
[3] Dieser Diagrammtyp wurde 1972 entwickelt. Zur zeitlichen Einordnung: ein Jahr später kam mit dem Intel 4004 der erste Mikroprozessor auf den Markt. Das kann für die zeitlose Bedeutung der Struktogramme sprechen, aber auch darauf hindeuten, dass nach 50 Jahren Änderungen in Erwägung gezogen werden könnten.

wie Mittelwerte, Standardabweichungen, … gruppiert nach Merkmalen wie Wohnort, Geschlecht oder Beruf der Eltern ermittelt werden, dann ist im Rest des Algorithmus genügend Raum z. B. für die Suche nach Korrelationen oder die grafische Darstellung der Zusammenhänge, ebenfalls mit einem oder wenigen Befehlen. Vor allem aber lassen sich mit diesen Möglichkeiten aktuelle und offensichtlich bedeutsame Fragestellungen aufwerfen, deren Antworten die Lernenden selbst betreffen.

1. Ein Ziel von *SciSnap!* besteht deshalb darin, entsprechende Bibliotheken auf verschiedenen Gebieten wie Bildbearbeitung, Diagrammerstellung, Mathematik, Datenanalyse und Datenbanken, Graphen oder Neuronalen Netzen bereitzustellen.







Hierzu ein (leider immer noch) aktuelles Beispiel: wirtschaftliche, geografische, soziale oder inhaltliche Beziehungen lassen sich durch Graphen darstellen. Stehen entsprechende Befehle zur Verfügung, dann erfordert die Erzeugung und Darstellung eines solchen (hier: random) Graphen in *SciSnap!* nur drei Befehle: *„konfiguriere ein Sprite, erzeuge n Knoten und danach n zufällig gewählte Kanten“*[4]. Betrachten wir die Verbindungen als Kontakte zwischen Personen, dann stellt sich die Frage, über wieviel Zwischenkontakte sich Infektionen in Pandemiezeiten ausbreiten können. Wir berechnen also die kürzesten Wege zwischen den Knoten: *„Berechne für jeden Knoten die kürzesten Wege zu allen anderen Knoten und trage diese in eine Liste ein“*. Für diese Ergebnisse berechnen wir in einer einfachen Schleife die Mittelwerte pro Knoten und daraus den Gesamtmittelwert.[5] Algorithmisch handelt es sich um ein typisches Anfängerproblem: *„durchlaufe eine einfache Schleife“*. Inhaltlich haben wir Wege zu Diskussionen über ein aktuelles gesellschaftliches Problem, über „small worlds“[6], soziale Netzwerke, Freundschaften oder Kunden-Lieferanten-Beziehungen gefunden. Der Unterricht hat an Relevanz gewonnen.

Ab *Snap! 7.0* gibt es die Möglichkeit, zusätzliche Paletten zu erzeugen oder zu löschen. Das erhöht deutlich die Übersicht. Niemand wird alle der acht teilweise umfangreichen Bibliotheken von *SciSnap!2* sowie die Kategorie *my own blocks* gleichzeitig benötigen. Ich habe sie trotzdem nicht weiter aufgespalten, weil man mit einem Klick (*Remove a category…* aus dem Datei-Menü) die nicht benötigten abschalten kann. Auf die ersten drei (*SciSnap! globals*, *Math tools* und *Data tools*) sowie einige neue Blöcke in den Standardpaletten sollte man allerdings nicht verzichten.

Es ist nicht das Ziel von *SciSnap!*, fertige Anwendungen vorzugeben. Vielmehr werden leistungsfähige Befehle bereitgestellt, mit denen sich Anwendungen erzeugen lassen. Ein Beispiel dafür sind die „Sketchpads“: Kostüme für beliebige Sprites oder die Bühne, auf denen sich schnell Skizzen erzeugen lassen. Funktionsgraphen, Bilder,

[4] Pseudocode: *configure GraphPad, add 100 vertices, add 100 edges*
[5] Pseudocode: *means=list, for i=1 to 100(add mean of row i of distances to means), mean=mean of means*
[6] https://de.wikipedia.org/wiki/Kleine-Welt-Ph%C3%A4nomen

Diagramme oder Histogramme lassen sich mit wenigen Befehlen erstellen, durch Skalen ergänzen – und wieder löschen, wenn es zu voll wird. Damit kann man z. B. einfach mathematische Zusammenhänge illustrieren, etwa die Wirkung von Operatoren auf komplexe Zahlen zeigen. Es wird gehofft, dass diese Beispiele die Lernenden dazu anregen, selbst andere, vielleicht bessere Anwendungen zu erzeugen, die algorithmische Methoden auf unterschiedlichen Gebieten benutzen.

2. *SciSnap!* soll sowohl als Tool wie als Entwicklungsumgebung nutzbar sein.

Snap! ist nicht nur ein fantastisches Entwicklungswerkzeug, sondern es basiert auf einem fantastischen Konzept. Als grafische Neuimplementierung der Ausbildungssprache *Scheme*[7] des *MIT*, basierend auf der „Informatikbibel“ *„Struktur und Interpretation von Computerprogrammen“*[8] von Abelson et.al., ist es konzeptionell vielen der gängigen Programmiersprachen weit überlegen. Seine eingebauten Visualisierungsmöglichkeiten machen es ideal für Simulationen. Die eingesetzte prototypische Vererbung macht informatische Grundkonzepte direkt erfahrbar. Trotzdem wird es, wohl wegen der Ähnlichkeit seiner Oberfläche zu *Scratch*, weitgehend unterschätzt. Ich hoffe deshalb, dass sich die Bereitstellung von Bibliotheken, die eher für Projekte in der Oberstufe bzw. in den ersten Semestern des Studiums bestimmt sind, positiv auf die Verbreitung in diesen Altersstufen auswirkt. Mal sehen …

3. *SciSnap!* ist für höhere Jahrgangsstufen der Schule sowie für das Grundstudium gedacht.

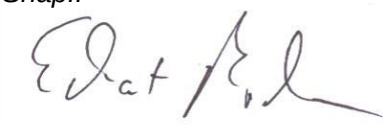
Das vorliegende Skript enthält eine Beschreibung der Möglichkeiten von *SciSnap!2* sowie einige Beispiele, die den gedachten Einsatz erläutern. Die Bibliotheken beruhen auf den Erfahrungen zum *Maschinellen Lernen* mit *Arthur&Ina*[9] und der *Snap!*-Variante *SQL-Snap!*[10]. Sie wurden um zahlreiche mathematische Operatoren, die Sketchpads, Neuronale Netze und Graphen ergänzt. Eine ausführliche Beschreibung von *Snap!* mit vielen Beispielen findet man unter *„Informatik mit Snap!“*[11] – und natürlich im *Snap!*-Manual[12]. Die vorgestellten Konzepte wurden und werden im Unterricht und in Anfängervorlesungen der Universität eingesetzt.

Ach ja, und da gibt es natürlich auch zwei kleine Helfer, die Sie bei der Arbeit mit *SciSnap!* unterstützen werden. Je nach Anwendung werden sich die beiden ablösen. *Alberto*[13] wird sich um die eher naturwissenschaftlichen Anwendungen kümmern, *Hilberto*[14] um mathematische und datenorientierte. Ist der eine tätig, dann kann sich der andere etwas ausruhen. Beide behaupten, entfernte Verwandte von *Alonzo*, dem *Snap!*-Maskottchen, zu sein. Ob das stimmt? Man weiß es nicht!

Ich bedanke mich bei Jens Mönig und besonders bei Rick Hessman für seine Beiträge z. B. zum *PlotPad*, ihre Unterstützung und die zahlreichen Diskussionen und Anregungen.

Ich wünsche Ihnen viel Freude bei der Arbeit mit *SciSnap!*.

Göttingen, am 22.3.2022



[7] https://en.wikipedia.org/wiki/MIT/GNU_Scheme
[8] https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html
[9] ebenso wie andere Materialien auf http://emu-online.de
[10] http://snapextensions.uni-goettingen.de/
[11] http://ddi-mod.uni-goettingen.de/InformatikMitSnap.pdf
[12] https://snap.berkeley.edu/snap/help/SnapManual.pdf
[13] er arbeitet in der Astrophysik bei Rick Hessman
[14] https://de.wikipedia.org/wiki/David_Hilbert

Inhalt

1 Starten von SciSnap!

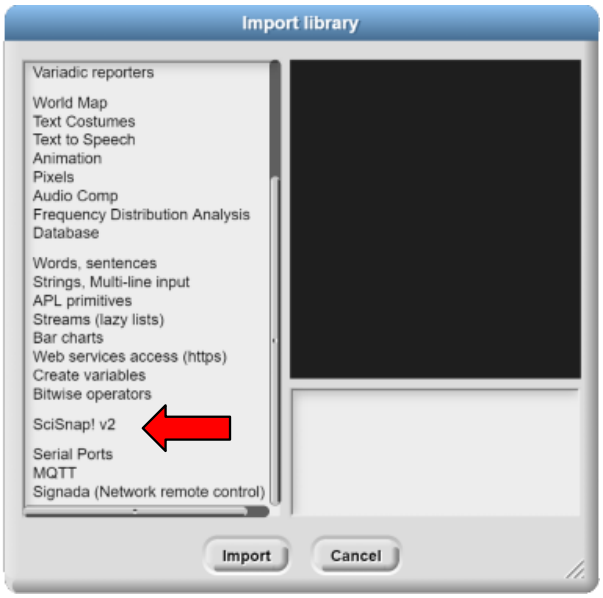
SciSnap! besteht aus einer Sammlung von ganz normalen *Snap!*-Blöcken, die in acht Paletten geordnet sind, die sich in ihrer Funktionalität unterscheiden. Eine weitere Palette *My own blocks* ist für die eigenen Programme gedacht. Hinzu kommt ein Sprite namens *Hilberto* mit ein paar Kostümen, das aber für das Funktionieren der Blöcke nicht notwendig ist. Natürlich benötigt niemand alle Paletten gleichzeitig; ich habe sie trotzdem in einem Paket gelassen, weil es ab *Snap!7.0* mit ein paar Klicks möglich ist, die nicht benötigten Teile schnell zu löschen und die veränderte Konfiguration als neue Arbeitsumgebung zu speichern.

Die ersten vier Paletten enthalten Blöcke, die direkt benutzt werden können. Bis auf die SQL-Befehle sollten sie nicht gelöscht werden. Die Befehle der weiteren vier Paletten beziehen sich jeweils auf Sprites, die mit dem ersten Block der Palette konfiguriert worden sind – zu *SketchPads* für Diagramme, Bilder, Graphen oder Neuronale Netze. Diese Paletten können je nach Bedarf im Projekt gelöscht werden.

Die einfachste Art *SciSnap!* zu starten ist es, einfach den entsprechend Link aufzurufen.

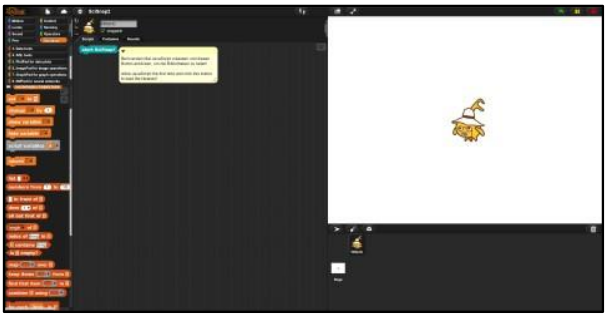
https://snap.berkeley.edu/snap/snap.html#present:Username=emodrow&ProjectName=SciSnap!2.0&editMode

Stattdessen können natürlich auch die *SciSnap!*-Blöcke geladen werden – dann allerdings ohne *Hilberto*. Das zweite empfiehlt sich, wenn ein vorhandenes Projekt um *SciSnap!*-Funktionalitäten erweitert werden soll.

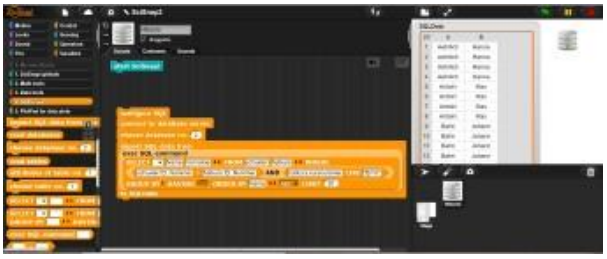


SciSnap! arbeitet mit einem Satz von JavaScript-Bibliotheken, die auf einem Server gespeichert sind. Sie werden vom Block *start SciSnap!* automatisch geladen, bevor *Snap!* zu *SciSnap!* umkonfiguriert wird, was man z. B. am veränderten Logo erkennt. Sie finden *SciSnap!* bei den *Snap!*-Bibliotheken im Datei-Menü. Sollten Sie die Blöcke von einem anderen Server laden, dann muss ausdrücklich der Zugriff auf JavaScript-Erweiterungen im Settings-Menü zugelassen werden, falls *Snap!* den Server nicht für vertrauenswürdig hält.

Wir gehen das anhand eines Beispiels durch: Ziel ist es, mit den *SQL-Blöcken* und der Palette für *eigene Blöcke* zu arbeiten. Auch *Diagramme* sollen erstellt werden.



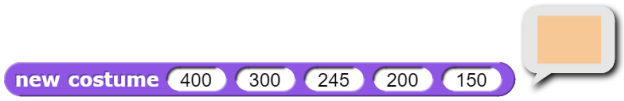
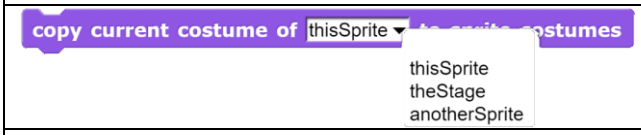
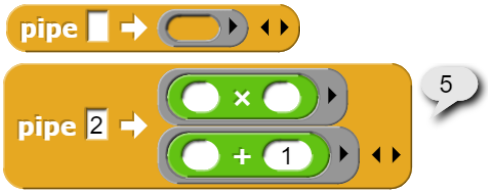
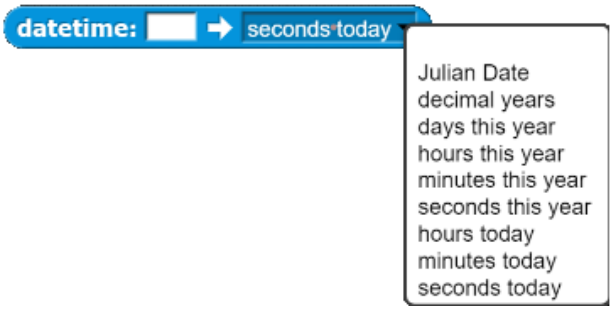
1. Schritt: Entweder laden wir *SciSnap!* als normale *Snap!*-Bibliothek (ohne *Hilberto*) oder wir laden *SciSnap!* aus der *Snap!*-Cloud
https://snap.berkeley.edu/snap/snap.html#present:Username=emodrow&ProjectName=SciSnap!2&editMode
(als Projekt mit *Hilberto*). Wir erhalten den nebenstehenden Bildschirm. Dort schalten wir ggf. JavaScript frei.

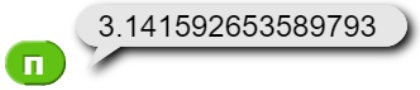
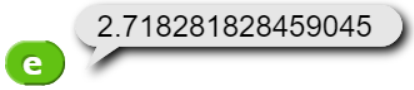
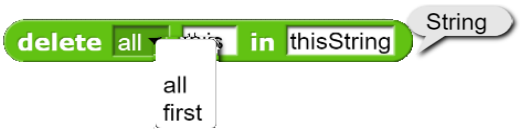
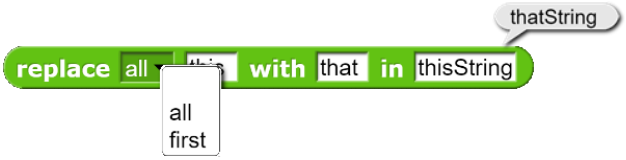


2. Schritt: Wir löschen die letzten drei *SciSnap!*-Paletten mithilfe des Dateimenüs (*Remove a category…*). **Auf jeden Fall klicken wir auf den *start SciSnap!*-Button**. Als Ergebnis erhalten wir eine SQL-Arbeitsumgebung, im Bild mit dem Ergebnis einer ersten Abfrage.

2 Neue Blöcke in den Standardpaletten

Einige Blöcke von *SciSnap!* befinden sich in den Standardpaletten, weil sie logisch einfach dorthin gehören. Sie ergänzen meist die sowieso schon vorhandene Funktionalität. Es handelt sich um die folgenden Blöcke:

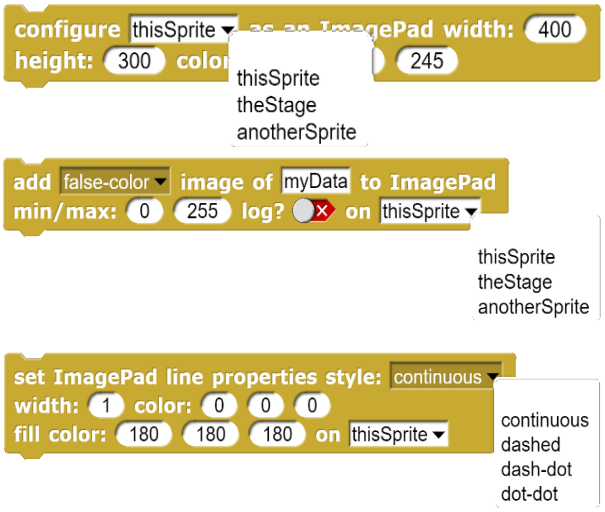
	Liefert das Kostüm eines Sprites, etwa für Pooling-Operationen.
	Liefert eine Kopie eines Kostüms.
	Liefert ein Kostüm der angegebenen Größe und Farbe.
	Fügt eine Kopie des aktuellen Kostüms der Kostümliste des Sprites oder der Bühne hinzu.
	Importiert ein Sprite, das als XML-Datei exportiert wurde.
	Löscht das aufrufende Sprite.
	Ein Block aus den *Snap!*-Libraries: Ersetzt eine geschachtelte Blockfolge durch eine etwas übersichtlichere Struktur.
	Datum und Zeit in Standarddarstellung, beispielsweise für astronomische Zwecke.
	Liefert Datum und Zeiten auf Basis der Standarddarstellung.
	Einfacher Eingabedialog per Auswahl der gewünschten Antwort mit der Maus.

	Erzeugt Zufallszahlen zwischen 0 und 1.
	Liefert die Zahl π.
	Liefert die Eulersche Zahl.
	Rundet eine Zahl auf die angegebene Zahl von Nachkommastellen.
	Liefert die Fakultät einer natürlichen Zahl.
	Berechnet einen Binomialkoeffizienten.
	Liefert den angegebenen Teil einer Zeichenkette.
	Löscht eine Teilzeichenkette in einer anderen, entweder an allen Stellen oder der ersten.
	Wandelt eine Zeichenkette in Großschrift um.
	Wandelt eine Zeichenkette in Kleinschrift um.
	Speichert einen Text in der angegebenen Datei im Download-Verzeichnis des Browsers.
	Liefert die Position des ersten Zeichens des ersten Auftretens der angegebenen Teilzeichenkette in einer Zeichenkette.
	Ersetzt eine Teilzeichenkette in einer anderen, entweder an allen Stellen oder der ersten.
	Erzeugt aus einer Spalte mit Texten eine Beschriftung für Koordinatensystemachsen.

3 Die Struktur der SciSnap!-Sprites

Der erste Teil von *SciSnap!* besteht aus vier Bibliotheken, die generell in *Snap!*-Skripten einsetzbar sind (*SciSnap! globals*, *Math tools*, *Data tools*, *SQL tools*). Die anderen vier Bibliotheken arbeiten mit besonders konfigurierten Sprites (*PlotPad*, *ImagePad*, *GraphPad*, *NNPad*). Der Grund dafür besteht in den *Properties*, die für ein *NeuralNet* nun einmal sehr anders sind als für ein *PlotPad*. Zusätzlich müssen solche „Spezial-Sprites“ über eigene Datenbereiche verfügen, in denen z. B. Bilddaten gespeichert sein können. Ein globaler Datenbereich findet sich in der Variablen *SciSnap!Data*, mit der z. B. die Blöcke der *Data*-Bibliothek im Default-Fall arbeiten. Für die Erstellung von Diagrammen ist es dagegen sinnvoller, dass ein *PlotPad* einen eigenen Datenbereich besitzt, der unabhängig von z. B. dem des *ImagePads* ist, auf das sich die Diagramme beziehen.

Jedes Sprite und auch die Bühne lassen sich als „Spezial Sprites“ konfigurieren, z. B. als *ImagePad*. Dabei werden die lokalen Variablen *myProperties* und *myData* erzeugt und einige sinnvolle Voreinstellungen bei den Eigenschaften vorgenommen. Die Properties sind meist zu Gruppen, z. B. über Kostümeigenschaften (*costumeProperties*) oder die Art, Linien zu zeichnen (*lineProperties*), zusammengefasst. Alle Blöcke, die eine bestimmte Konfiguration erfordern, fragen anfangs die Eigenschaft *typeOfConfiguration* des Sprites, mit dem sie arbeiten sollen, ab. Stimmt diese nicht, erfolgt eine Fehlermeldung. Die Gruppen von Eigenschaften werden mit den entsprechenden Blöcken geändert.



Der Aufbau der *SciSnap!Sprites* orientiert sich also an der Idee von dokumentierten Datensätzen, die aus zwei Teilen bestehen: den *Metadaten*, die die Struktur und den inhaltlichen Kontext der Daten beschreiben (z. B. Zahlenformat, Bilddimensionen, Aufnahmegerät, Aufnahmedatum, ...) und den dazugehörigen reinen *Datensegmenten*. Metadaten bestehen gewöhnlich aus *Dictionaries* - Namen mit zugewiesenen Werten (z. B. "Aufnahmedatum: 24.12.2018"). Beispiele für diese Struktur sind FITS-Dateien [FITS], die in der Astrophysik Standard sind, aber auch in der Vatikanischen Bibliothek Verwendung finden, oder JPEG Bilder vom Handy. Auch hier gibt es Metadaten (Bildgröße, Kompressionsgrad, Aufnahmedatum, …), ohne die eine Bilderzeugung nicht möglich wäre.

Wir adaptieren diese Struktur, indem wir einem *SciSnap!-SketchPad* zwei lokale Variable verpassen, die jeweils die Daten (*myData*) und die Datenbeschreibung (*myProperties*) enthalten. Diese Variablen können einerseits durch den Import von Daten aus unterschiedlichen Quellen (SQL-Abfrage, Textdatei, CVS-Datei, JSON-Datei, FITS-Datei, direkte Zuweisung, …) gefüllt werden, wobei die Eigenschaften *myProperties* den jeweiligen Daten anzupassen sind. Andererseits kann das auch „per Hand“ geschehen. Mithilfe dieser Eigenschaften können Daten in grafische Darstellungen (Graph, Datenplot, Histogramm, Bild, …) umgesetzt werden, wobei als Quelle entweder *myData* oder eine andere geeignete Tabelle gewählt wird.

Wichtig ist, dass die Bilderzeugung die Originaldaten nicht verändert. Wird also z. B. eine Aufnahme des Jupiters benutzt, um die Abstände seiner Monde zu bestimmen, dann müssen diese zumindest im Bild sichtbar sein. Dafür kann nach dem Einstellen einiger Parameter z. B. ein Falschfarbenbild erzeugt werden. In diesem wird der Jupiter selbst ziemlich unstrukturiert erscheinen. Will man dagegen das „Auge“ des Planeten genauer untersuchen, dann müssen die Parameter ganz anders gewählt werden, sodass die Monde wiederum kaum zu sehen sind. Alle diese Änderungen müssen in den Pixeln des aktuellen Kostüms des *Snap!*-Sprites geschehen, ohne die Bilddaten selbst zu beeinflussen.

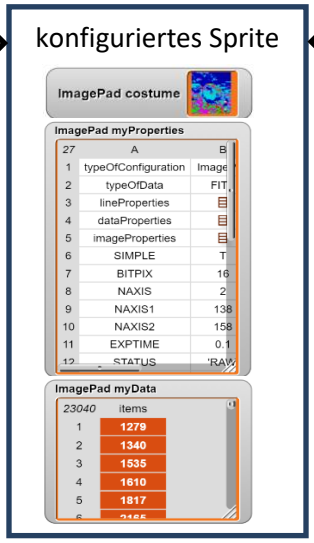
Weil Tabellen in *Snap!* sehr schön dargestellt werden können, ist diese Darstellungsform nicht zusätzlich implementiert. Dafür ist der Datentyp *table* mit zahlreichen der im Bereich von *Data Science* üblichen Operationen (Tabellenoperationen, Korrelationsberechnung, affine Transformationen, Lösen linearer Gleichungssysteme, …) implementiert, der ausreichend schnell auch mit größeren Datenmengen umgehen kann.

Da *SciSnap!* (derzeit) etwa 250 neue Blöcke enthält, wurden diese nach ihrer Funktionalität gruppiert und auf verschiedene Bibliotheken und Sprite-Konfigurationen verteilt: eine *Math-tools*-Bibliotheken für unterschiedliche Gebiete der Mathematik (60 Blöcke), eine *Data-tools*-Bibliothek (38 Blöcke) zum Umgang mit den eigentlichen Daten, ein *ImagePad* zur Bildbearbeitung (25 Blöcke), ein *PlotPad* für grafische Darstellungen (27 Blöcke), ein *NeuralNetPad* für Perzeptron-Netze (11 Blöcke), eine *SQL*-Bibliothek für Datenbankanfragen (26 Blöcke) und ein *GraphPad* für Anwendungen der Graphentheorie (35 Blöcke). Hinzu kommen die schon genannten Blöcke in den Standardpaletten von *Snap!*. Alle Blöcke sind global und enthalten das Ziel der Operation (*thisSprite*, *theStage* oder den *Namen* eines anderen Sprites). Objektorientierte Aufrufe sind deshalb weitgehend unnötig.

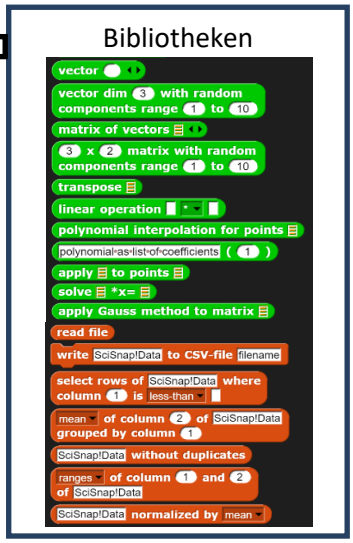
Die konfigurierten Sprites haben die folgende Struktur:

Sie importieren mithilfe der Bibliotheken Daten aus …

- Bild-Dateien
- Text-Dateien
- SQL-Abfragen
- JSON-Dateien
- CSV-Dateien
- …



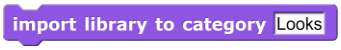
Die Bibliotheken stellen Blöcke zur grafischen Darstellung der Daten, zur Bearbeitung von Tabellen, Lösen von Gleichungssystemen, Anwendung statistischer Operationen, … bereit.



Die meisten Blöcke erhalten ihre Parameter (Bildgröße, Wertebereiche, Farben, …) aus dem Dictionary *myProperties*. Die voreingestellten Eigenschaften ermöglichen es, ohne allzu viele Parameter Blöcke zum Erstellen von Grafiken, Diagrammen, … zu benutzen. Passen die Werte nicht, dann werden die Eigenschaften entweder einzeln oder in Gruppen geändert.

4 Die SciSnap!-Bibliotheken

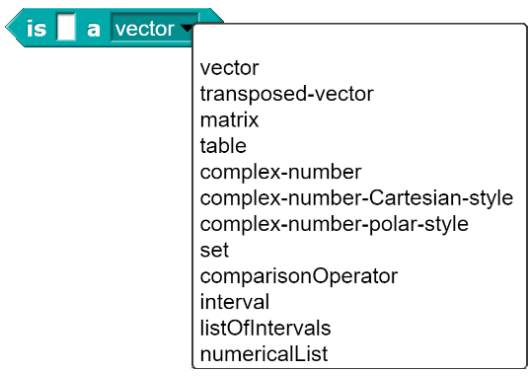
Im Folgenden werden die Bibliotheken tabellarisch vorgestellt. Umfangreichere Beispiele, die meist mehrere Bibliotheken nutzen, folgen danach.

Die *SciSnap!*-Bibliotheken wurden – wie alle Block-Bibliotheken von *Snap!* – Paletten zugeordnet und gespeichert. Ist die Palette eines Blocks noch nicht vorhanden, dann wird sie beim Laden erzeugt. Soll eine Bibliothek in eine andere Palette geladen werden, dann kann dafür der Block 

benutzt werden. *My own blocks* ist für die eigenen Blöcke gedacht.

4.1 Blöcke der Palette SciSnap! globals

In dieser Palette befinden sich Blöcke für die Konfiguration von *SciSnap!*.

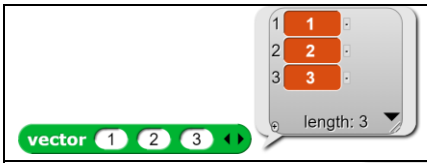
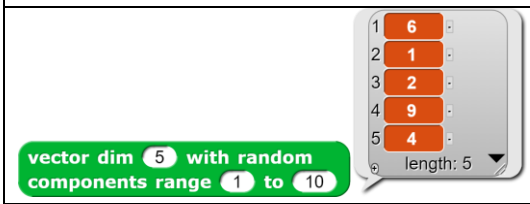
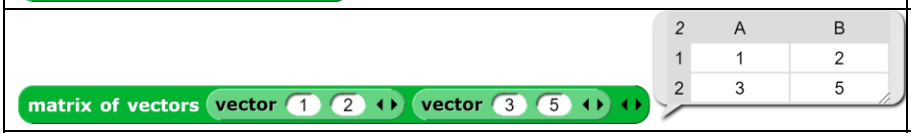
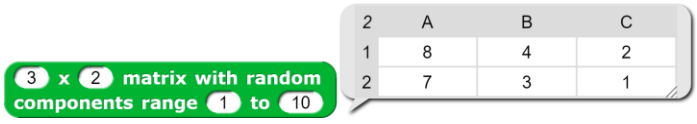
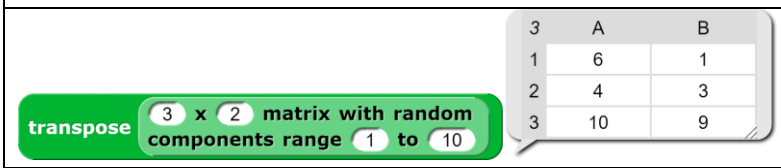
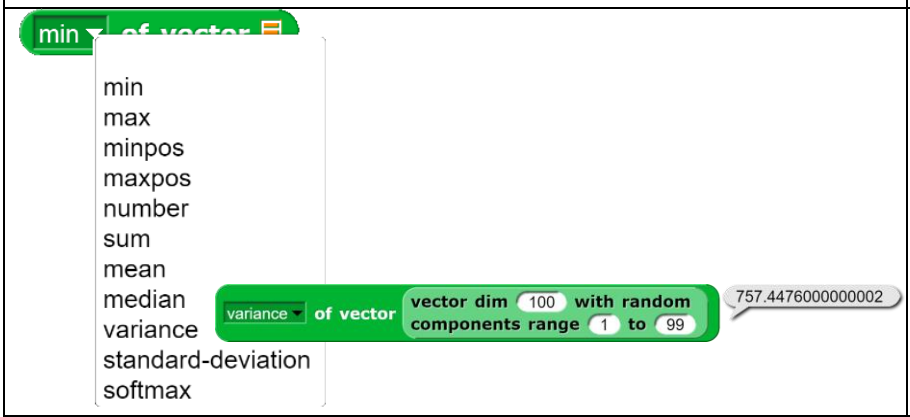
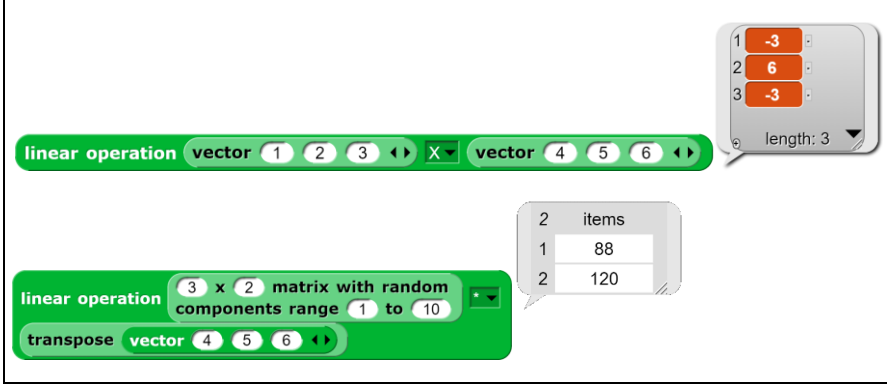
	Lädt die JavaScript-Bibliothek von *SciSnap!*. Erzeugt das *SciSnap!*-Logo und die abgebildeten globalen Variablen. Vergrößert die Bühne auf 800x600 Pixel.
	Erzeugt die globalen *SciSnap!*-Variablen und setzt einige *SciSnap!*-Eigenschaften.
	Liest eine Eigenschaft.
	Erlaubt das Verändern oder Erzeugen einer Eigenschaft.
	Erzeugt ein Meldungsfenster in der Mitte des Bildschirms.
	Gibt einen Fehler, wenn möglich, über das aktuelle Sprite aus und trägt ihn in *SciSnap!Messages* ein.
	Testet ein Element darauf, ob es zu einem *SciSnap!*-Typ gehört.

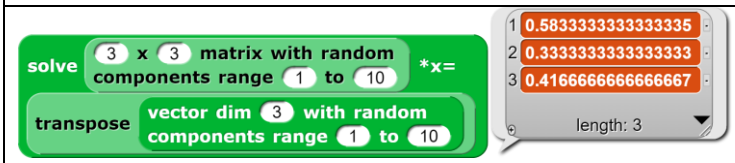
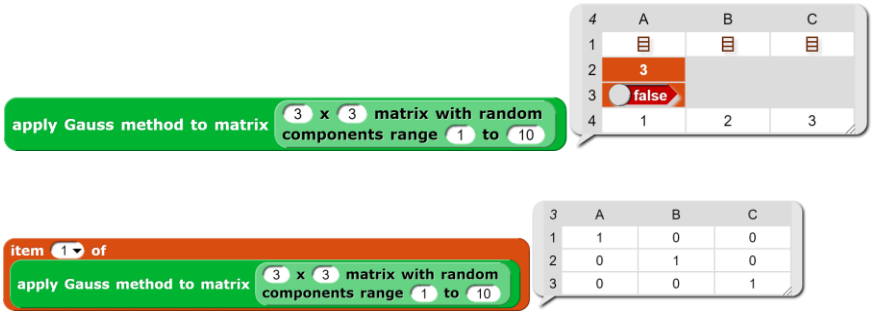
	Testet, ob eine globale oder lokale Eigenschaft vorhanden ist und ob sie den angegebenen Wert hat.
	Ersetzt das *Snap!*-Logo durch das *SciSnap!*-Logo.
	Importiert eine Bibliothek in die angegebene Palette. Alle Blöcke sollten zur gleichen Kategorie gehören!

4.2 Die Mathematik-Bibliothek

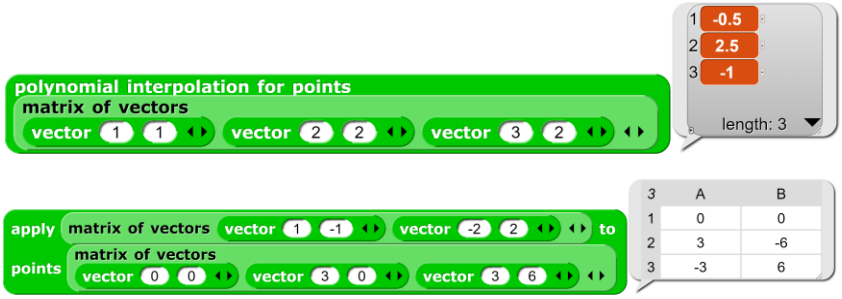
4.2.1 Lineare Algebra

SciSnap! kennt Vektoren und Matrizen sowie die gängigen Operationen mit ihnen. Beide werden als Listen bzw. Listen von Listen dargestellt, und beide können in transponierter Form vorliegen. Die folgenden Blöcke arbeiten mit ihnen:

	Liefert einen Vektor beliebiger Dimension mit vorgegebenen Werten.
	Liefert einen Vektor beliebiger Dimension mit Zufallswerten aus dem angegebenen Bereich.
	Liefert die Matrix aus den angegebenen Vektoren.
	Liefert eine Matrix beliebiger Dimension mit Zufallswerten aus dem angegebenen Bereich.
	Matrizen und Vektoren können transponiert werden.
	Liefert eine der angegebenen Eigenschaften eines Vektors.
	Zwischen Skalaren, Vektoren und Matrizen können die zugelassenen Operationen ausgeführt werden. Da Vektoren mit den arithmetischen Standard-Operatoren von *Snap!* bearbeitet werden können, gibt es dafür keine gesonderten Blöcke.

	Wendet eine Matrix auf eine Punktliste an (s. Beispiel).
	Berechnet die Lösung eines linearen Gleichungssystems.
	Der Block liefert mehrere Daten der übergebenen Matrix: die ggf. diagonalisierte Matrix, deren Rang, ob Spaltenvertauschungen stattgefunden haben und die jetzige Reihenfolge der Spalten. Interessiert nur die diagonalisierte Matrix, dann betrachten wir das erste Element des Resultats.
	Berechnet die Koeffizienten des Polynoms durch n Punkte.
	Für Polynome kann man Funktionswerte berechnen.
	Führt eine affine Transformation in der Ebene für eine Punktliste, z. B. ein Bild, aus, die durch die Zuordnung von drei Punkten zu drei anderen beschrieben ist (s. Beispiel).

Beispiele:

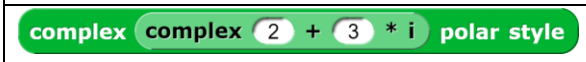
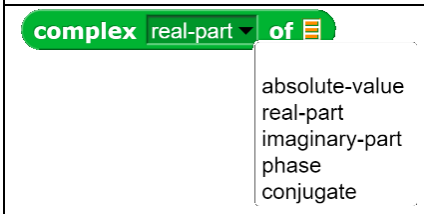
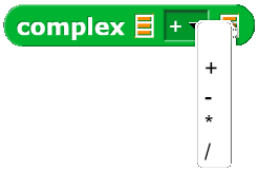


4.2.2 Komplexe Zahlen

Falls Sie umfangreichere Operationen mit komplexen Zahlen planen, sollten Sie sich überlegen, die *Scheme*-Bibliothek von *Snap!* zu benutzen (*Bignums*-library). *SciSnap!* ist eher für komplexe Arithmetik sowie die Veranschaulichung der Operationen gedacht. In *SciSnap!* werden komplexe Zahlen als 3-elementige Listen dargestellt, wobei der erste Eintrag das Format der Zahl bezeichnet: entweder den „kartesischen“ Stil $z = a + b \cdot i$ oder die Polarform $z = r \cdot e^{i\varphi}$. Für diese beiden Formen gibt es Eingabeblöcke:

	Liefert eine komplexe Zahl in kartesischer Form.
	Liefert eine komplexe Zahl in Polarform.

Bei Bedarf können diese Darstellungsformen ineinander umgewandelt werden, und man kann arithmetische Operationen durchführen.

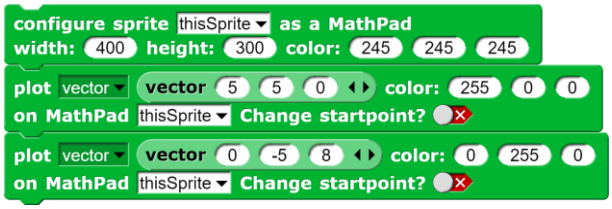
	Liefert eine komplexe Zahl in kartesischer Form.
	Liefert eine komplexe Zahl in Polarform.
	Beispiel für eine Formatumwandlung.
	Auf die Komponenten einer komplexen Zahl kann – unabhängig von ihrem Format – zugegriffen werden.
	Und man kann natürlich mit ihnen rechnen.

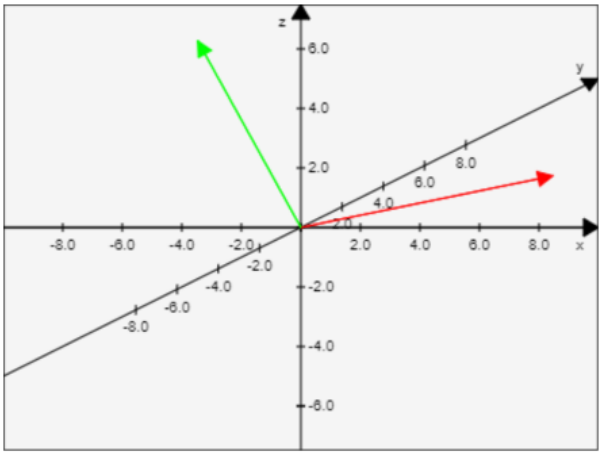
Beispiel für eine Multiplikation:

4.2.3 Das MathPad

configure sprite thisSprite as a MathPad width: 400 height: 300 color: 245 245 245	Konfiguriert ein Sprite als MathPad und zeichnet ein 3-dimensionales Koordinatensystem zentriert in der Mitte.
is thisSprite a MathPad?	Test auf MathPad-Konfiguration.
set MathPadProperty costumeProperties of thisSprite to	Setzt eine MathPad-Eigenschaft.
MathPadProperty costumeProperties of thisSprite	Liest eine MathPad-Eigenschaft.
set MathPad costume properties width: 400 height: 300 color: 245 245 245 offsets: 0 0 on thisSprite	Setzt die Kostüm-Eigenschaften eines MathPads auf die angegebenen Werte.
set MathPad properties lineWidth: 1 onlyPoints? dimension: 3 maxValue: 10 startPoint: 0 0 0 on thisSprite	Setzt die Linien-Eigenschaften eines MathPads auf die angegebenen Werte. Ist „onlyPoints“ gesetzt, werden nur die Endpunkte gezeichnet.
add centered axes to a MathPad on thisSprite	Zeichnet das Koordinatensystem auf ein MathPad.
plot vector … 0 0 on MathPa … ange startpoint? vector complex-number line-to object-of	Zeichnet einen Vektor, eine komplexe Zahl, eine Linie oder ein Objekt, das durch eine Liste von Vektoren beschrieben wird.

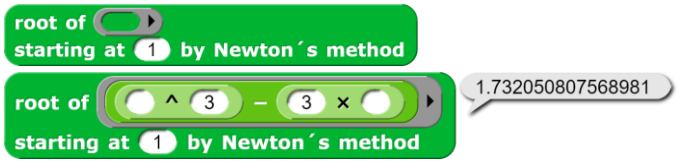
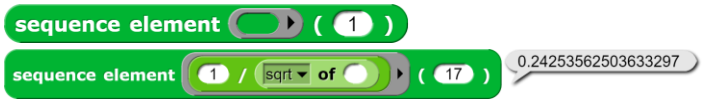
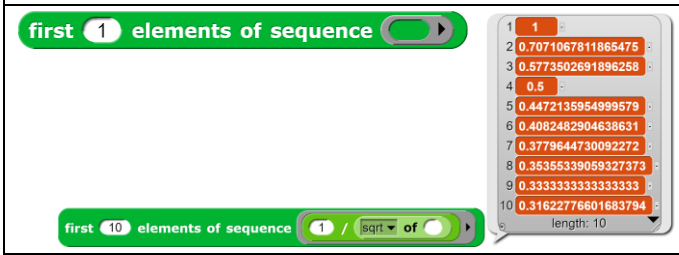
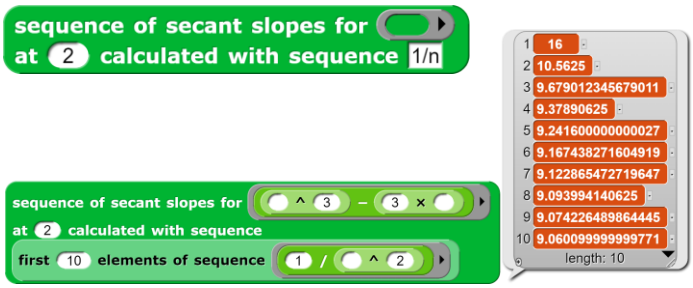
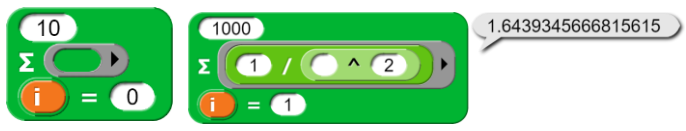
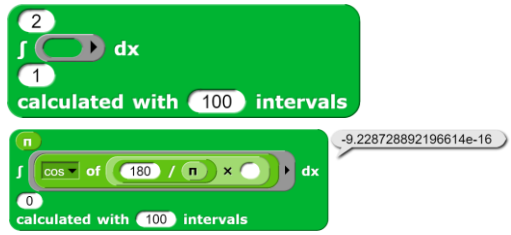
Beispiel:

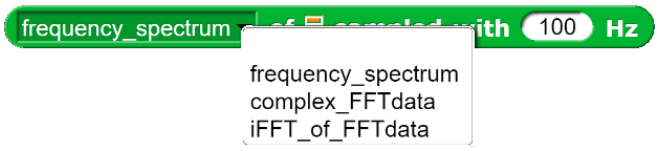




4.2.4 Numerische Verfahren

Die Bibliothek enthält einige Blöcke zum Umgang mit Folgen, Reihen, Sekanten, Integralen und Nullstellen sowie der Berechnung von Ableitungen an einer bestimmten Stelle oder schnelle Fourier-Transformationen (FFT).

	Nullstellenberechnung nach dem Newton-Verfahren. Der Term muss mit grauem Ring eingegeben werden („ringified“). Beispiel: Berechnung der Nullstelle von $f(x) = x^3 - 3x$, Start bei x=1.
	Berechnung eines Folgenelements. Der Term muss mit grauem Ring eingegeben werden („ringified“). Beispiel: das 17. Element der Folge $\frac{1}{\sqrt{n}}$.
	Liefert die ersten n Glieder einer Folge als Liste. Beispiel: Die ersten 10 Elemente der Folge $\frac{1}{\sqrt{n}}$.
	Folge von Sekantensteigungen in einem Punkt. Die Folge kann auch explizit in Form einer Liste angegeben werden. Der Term muss mit grauem Ring eingegeben werden („ringified“). Beispiel: 10 Sekantensteigungen in der Nähe des Punkts mit x=2 für $f(x) = x^3 - 3x$, berechnet mit der Folge $\frac{1}{n^2}$.
	Numerische Bestimmung der Ableitung in einem Punkt. Der Term muss mit grauem Ring eingegeben werden („ringified“). Beispiel: Berechnung der Ableitung von $f(x) = x^3 - 3x$ für x=0.
	Berechnet die Summe einer endlichen Reihe. Der Term muss mit grauem Ring eingegeben werden („ringified“). Beispiel: Die Summe der ersten 1000 Glieder der Reihe $\sum_{i=1}^{1000} \frac{1}{i^2}$.
	Numerische Berechnung eines Integrals mithilfe des Trapezverfahrens. Der Term muss mit grauem Ring eingegeben werden („ringified“). Beispiel: Berechnung des Integrals F=$\int_0^{\pi} \cos x \, dx$ (mit Umrechnung ins Gradmaß)

	Berechnung schneller Fourier-Transformationen (FFT) sowie deren Inverser (iFFT). Alternativ werden die Daten des Frequenz-Spektrums berechnet.

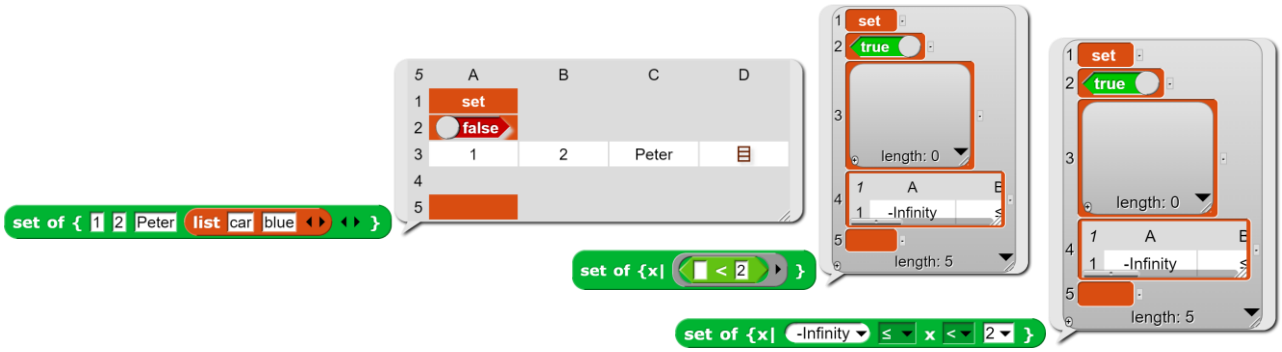
4.2.5 Statistik

Für statistische Anwendungen enthält *SciSnap!* eine Reihe von Verteilungen. Korrelationsberechnung, Varianzen etc. sind in der Datenbibliothek implementiert, Binomialkoeffizienten und Fakultäten findet man in der Operatoren-Palette.

	Wahrscheinlichkeiten der Binomialverteilung $b(N,p,k) = \binom{N}{k} \cdot p^k \cdot (1-p)^{N-k}$
	Berechnet die kumulierte Verteilungsfunktion der Binomialverteilung.
	Wahrscheinlichkeiten der hypergeometrischen Verteilung $h(N,M,n,k) = \frac{\binom{M}{k} \cdot \binom{N-M}{n-k}}{\binom{N}{n}}$
	Berechnet die kumulierte Verteilungsfunktion der hypergeometrischen Verteilung.
	Wahrscheinlichkeiten der Poisson-Verteilung $p(\theta,k) = \frac{\theta^k \cdot e^{-\theta}}{k!}$
	Berechnet die kumulierte Verteilungsfunktion der Poisson-Verteilung.
	Wahrscheinlichkeiten der Pareto-Verteilung $pareto(x_{min},k,x) = \frac{k \cdot {x_{min}}^k}{x^{k+1}};$ $x \geq x_{min}; 0\ sonst$
	Wahrscheinlichkeiten der Normalverteilung $n(x,\mu,\sigma) = \frac{e^{-\frac{(x-\mu)^2}{2\sigma^2}}}{\sqrt{2\pi\sigma^2}};$

4.2.6 Mengen

Mengen sind in *SciSnap!* als 5-elementige Listen implementiert. Bei der Implementierung war es mein Ziel, möglichst weitgehend mit unendlichen Mengen umzugehen. Dafür wurden Mengen, die durch Prädikate definiert sind, etwas vernachlässigt. Um eine Menge zu erzeugen, kann man die Elemente aufzählen {1,Auto,true}, über Prädikate definieren {x|x<5} oder Bereiche eingeben {-Infinity<x<=2}. Im ersten Element einer Menge steht der Typ („set"), im zweiten ein Prädikat, das angibt, ob es sich um eine „numerische" Menge handelt (also alle Elemente Zahlen sind), das dritte ist entweder leer oder es werden Elemente aufgezählt, im vierten steht ggf. eine Liste der Intervalle, die die Mengenelemente enthalten, und im letzten steht ggf. ein Prädikat. Zusammengesetzte Prädikate sind wiederum Listen, die im ersten Element den booleschen Operator („NOT", „OR", „AND") und danach ein oder zwei Prädikate enthalten, die wiederum zusammengesetzt sein können. Nach Möglichkeit werden die Listenelemente durch Intervalle beschrieben. Dafür werden z. B. Prädikate in Intervall-Listen umgesetzt.



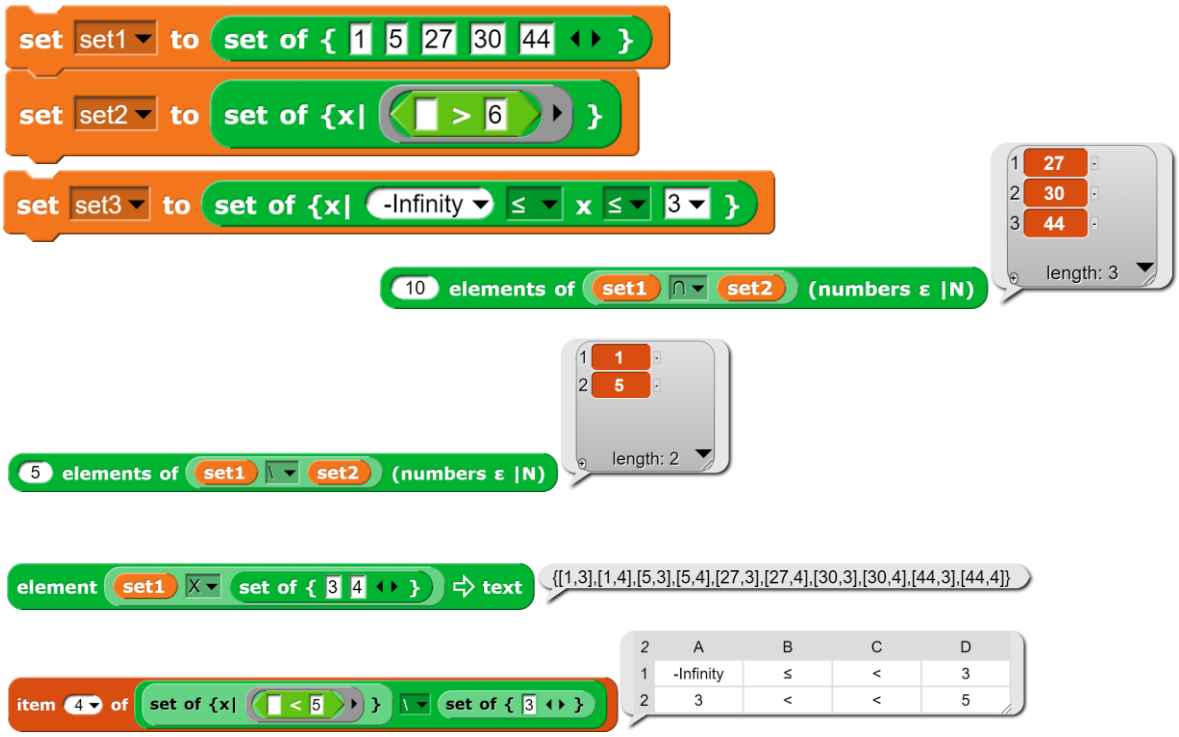
Für die weitere Arbeit mit Mengen stehen die bekannten Mengenoperationen zur Verfügung. Benutzt man Prädikate, dann sind als Elemente weitgehend nur Zahlen und Zeichenketten sinnvoll. Lassen sich die Mengen durch Intervalle beschreiben, dann sind die Operationen nicht beschränkt. Etwa 20 Hilfsblöcke für z. B. Intervall-Operationen tauchen nicht in den Paletten auf. Man findet sie, wenn man Blöcke zum Export auswählt.

set of { } set of {x| -Infinity ≤ x ≤ Infinity } set of {x| }	Blöcke zur Definition von Mengen (wie oben beschrieben).
element ε set ?	Liefert „wahr", falls das Element ein Element der Menge ist, sonst „falsch".
set1 ∩ set2 (∩ ∪ \ X)	Die grundlegenden Mengenoperationen zur Berechnung der Schnittmenge, der Vereinigungsmenge, der Differenzmenge und des kartesischen Produkts.
is set1 ⊆ set2 ?	Liefert „wahr", falls set1 Teilmenge von set2 ist, sonst „falsch".
is set1 = set2 ?	Liefert „wahr", falls set1 = set2 ist, sonst „falsch".
element set ⇨ text {2,3,Petra,[car,blue],0}	Stellt die Elemente einer Menge „etwas lesbarer" dar.
text 1,2,[3,4],{6,7,8..12} ⇨ elements	Erzeugt aus einem Text die entsprechende Liste von Elementen. Listen werden im Text „eckig" geklammert, Mengen „geschweift".

Einige Operationen müssen mit endlichen Mengen ausgeführt werden, z. B. um Mengenelemente aufzuzählen. Aus diesem Grund gibt es eine obere Schranke für Mengenelemente, die durch die *SciSnap!Properties* festgelegt wird.

	Setzt die Grenze, bis zu der ggf. Prädikate bzw. Intervallgrößen überprüft werden.
	Liefert die ersten n Elemente einer Menge als Liste.

Beispiele:



4.3 Die Daten-Bibliothek

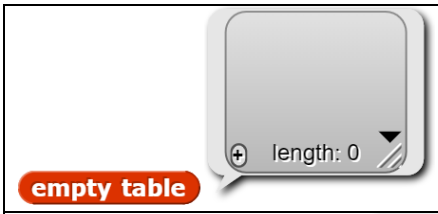
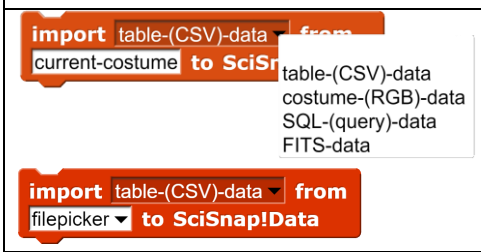
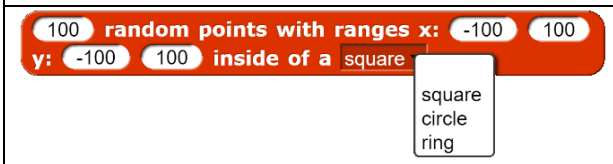
Die Daten-Bibliothek von *SciSnap!* dient einerseits zur direkten Manipulation auch größerer Datenmengen, andererseits zur Auswertung von Daten, z. B. zur Berechnung von statistischen Größen wie der Kovarianz oder Korrelationen. Hinzu kommen einige Blöcke für die Standardverfahren des maschinellen Lernens.

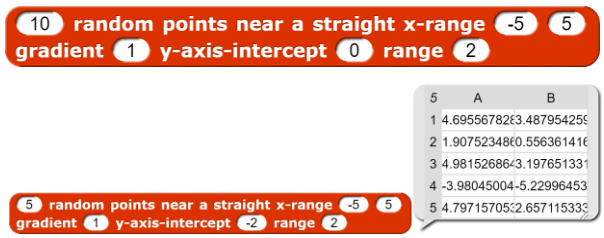
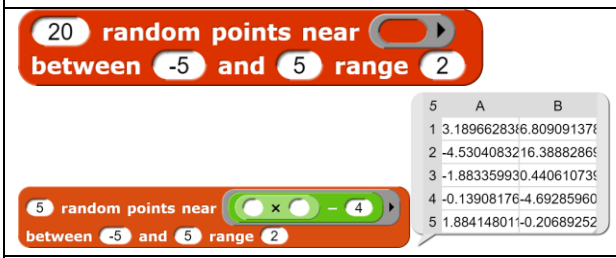
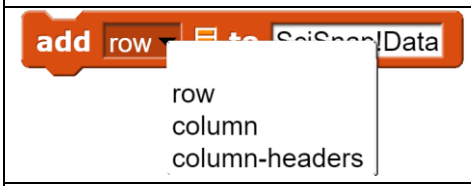
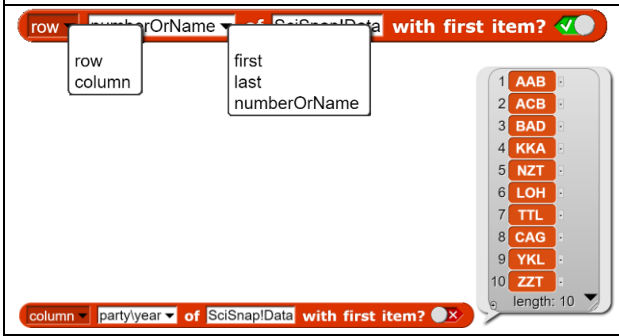
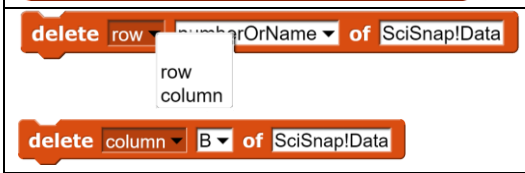
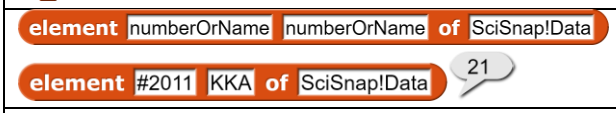
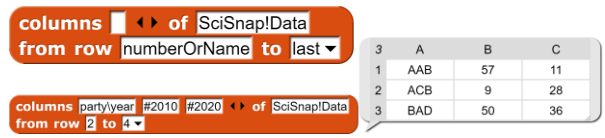
Weil Zahlenstrukturen wie Vektoren und Matrizen in den Mathematik-Bibliotheken implementiert sind, beschränkt sich die Daten-Bibliothek weitgehend auf Tabellen (*tables*) als zusätzliche Struktur. Deren Zeilen und Spalten können entweder über ihre Nummern oder die Bezeichner der ersten Spalte bzw. Zeile identifiziert werden. Spalten können zusätzlich mit großen Buchstaben (*A..Z*) benannt werden. Wird eine Zahl als Bezeichner benötigt (also nicht die Spalten- oder Zeilennummer), dann muss ein Doppelkreuz (#) vor die Zahl als Bezeichner gesetzt werden, z. B. *#123*.

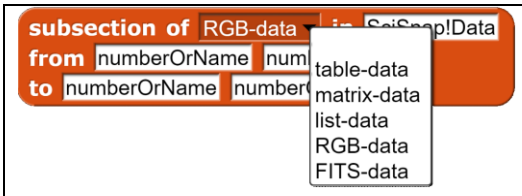
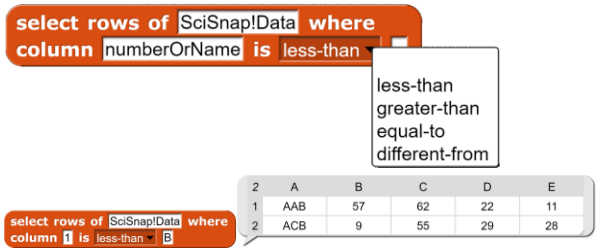
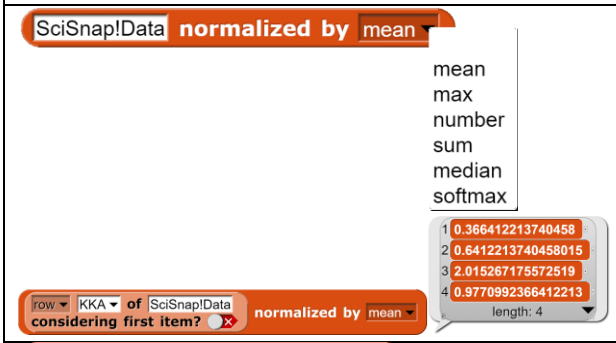
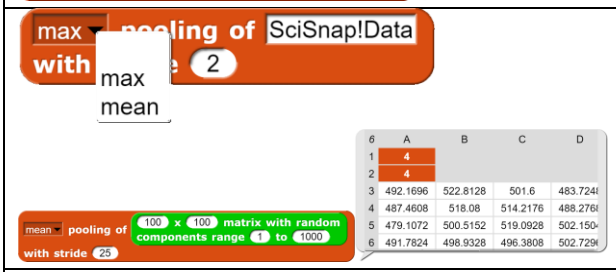
SciSnap!Data

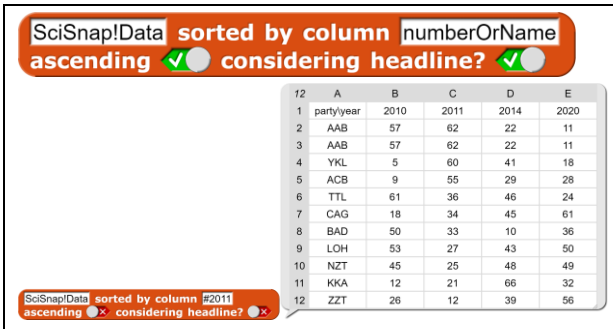
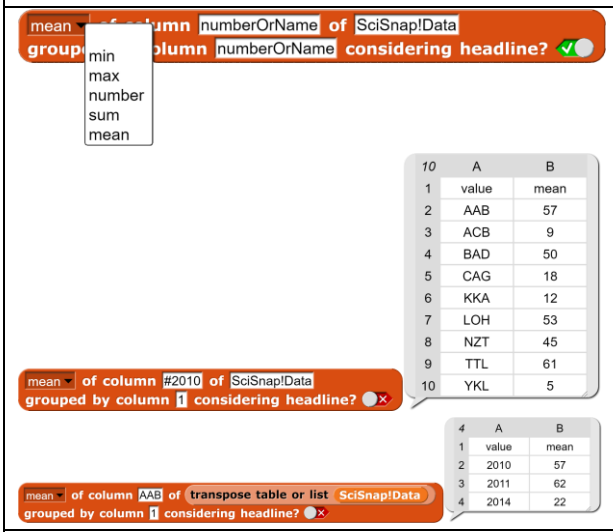
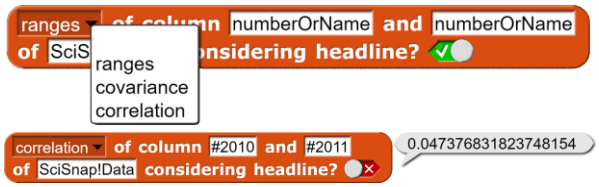
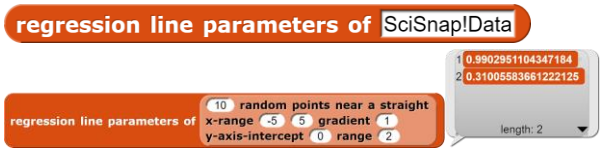
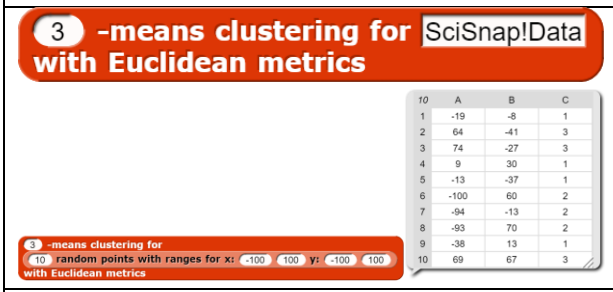
11	A	B	C	D	E
1	party\year	2010	2011	2014	2020
2	AAB	57	62	22	11
3	ACB	9	55	29	28
4	BAD	50	33	10	36
5	KKA	12	21	66	32
6	NZT	45	25	48	49
7	LOH	53	27	43	50
8	TTL	61	36	46	24
9	CAG	18	34	45	61
10	YKL	5	60	41	18
11	ZZT	26	12	39	56

Die folgenden Beispiele beziehen sich auf eine Tabelle, die „Parteinamen“ als erste Spalte und danach „Wahlergebnisse“ in den angegebenen Jahren enthält.

	Liefert eine leere Tabelle (als Startstruktur für weitere Tabellenoperationen).
	Liefert eine Tabelle der angegebenen Größe, initialisiert mit einem Startwert.
	Liefert eine Tabelle der angegebenen Größe ohne Inhalte, aber mit Spaltenüberschriften.
	Liefert eine Kopie einer Liste oder Tabelle. Da *Snap!* Listen als Referenzen zuweist, kann man mit diesem Block unbeabsichtigte Änderungen am Original vermeiden.
	Der Block importiert Tabellen, Bilddaten oder SQL-Daten in die globale Variable *SciSnap!Data*, mit der die anderen Blöcke der Bibliothek per Voreinstellung arbeiten. Beispiel: Import einer Tabelle mithilfe des Datei-Auswahldialogs.
	Schreibt eine Tabelle in eine Datei des Downloadbereichs des Browsers unter dem angegebenen Namen.
	Liefert n unterschiedlich verteilte Punkte aus dem angegebenen Bereich.

	Liefert n Punkte, die um eine Gerade, gegeben durch Steigung und y-Achsenabschnitt, in einem Bereich streuen, der durch „range“ begrenzt wird. Dient meist zu Testzwecken. Beispiel: 5 Punkte, die um $f(x) = x - 2$ streuen.
	Liefert n Punkte, die um eine beliebige Funktion, gegeben durch ihre „ringified“ Operatoren, in einem Bereich streuen, der durch „range“ begrenzt wird. Dient meist zu Testzwecken. Beispiel: 5 Punkte, die um $f(x) = x^2 - 4$ streuen.
	Liefert als Ergebnis eine transponierte Tabelle oder Liste, also eine, bei der Zeilen und Spalten vertauscht wurden.
	Fügt der angegebenen Tabelle eine Zeile, eine Spalte oder Spalten-Überschriften hinzu. Fehlende Elemente werden mit leerem Inhalt ergänzt, „überstehende“ werden ignoriert.
	Liefert eine Zeile oder Spalte der angegebenen Tabelle. Beispiel: Die erste Spalte der Beispieltabelle ohne die Überschrift.
	Löscht eine Zeile oder Spalte aus der angegebenen Tabelle. Beispiel: Löscht die Spalte für das Jahr 2010 aus der Beispieltabelle.
	Liefert das angegebene Tabellenelement. Beispiel: Ergebnis der Partei KKA im Jahr 2011.
	Setzt den Wert in einer Tabellenzelle.
	Liefert den angegebenen Zeilenbereich. Beispiel: Ergebnisse der drei angegebenen Parteien aus 2010 und 2020.

	Liefert einen Ausschnitt aus einer Tabelle, einer Matrix, einer Liste oder aus Bilddaten, der durch die beiden Punkte „links-oben“ und „rechts-unten“ gegeben ist.
	Liefert ausgewählte Zeilen einer Tabelle, die dem genannten Kriterium genügen. Beispiel: Alle Wahlergebnisse mit Parteien, die mit „A“ anfangen.
	Zählt das Vorkommen der Attribute einer Liste.
	Bestimmt die Entropie einer Liste.
	Entfernt Duplikate aus einer Liste.
	Normalisierung eines Vektors durch Teilen durch den Mittelwert, den Maximalwert, die Anzahl, Summe, Median seiner Werte oder durch Anwendung der Softmax-Funktion. Beispiel: Ergebnisse der Partei KKA, „normalisiert“ durch den Mittelwert.
	Komprimiert einen Vektor oder eine Matrix mit dem angegebenen Faktor durch Mittelwertbildung.
	Liefert eine Tabelle, die durch max- oder mean-Pooling mit der Schrittweite n komprimiert wurde. Vor dem Resultat werden die Dimensionen der neuen Tabelle übergeben. Gut anwendbar auf Bilddaten. Beispiel: Eine Matrix aus 100x100 Zufallszahlen wird mit der Schrittweite 25 komprimiert.
	Sortiert eine Liste mithilfe des angegebenen Prädikats.

	Liefert eine nach der angegebenen Spalte auf- oder absteigend sortierte Tabelle. Beispiel: Die Wahlergebnisse sortiert nach dem Jahr 2011.
	Liefert Minimum, Maximum, Anzahl, Summe oder Mittelwert der ersten angegebenen Spalte der Daten, gruppiert nach der zweiten. Die Überschriften können einbezogen werden – oder nicht. Beispiele: Die Mittelwerte des Jahres 2010, gruppiert nach Parteien. Die Mittelwerte des Partei AAB, gruppiert nach Jahren.
	Liefert Bereiche, Kovarianz und Korrelation zwischen zwei Tabellenspalten. Beispiel: Korrelation zwischen den Jahren 2010 und 2011.
	Liefert Steigung und y-Achsenabschnitt der Regressionsgerade durch die angegebenen Daten. Beispiel: Regressionsgerade durch 10 Zufallspunkte, die um eine Gerade streuen.
	k-Nächste-Nachbarn-(kNN)-Verfahren in zwei Dimensionen für maschinelles Lernen. Beispiel: HR-Diagramm
	Liefert das Ergebnis einer Faltung (convolution), angewandt entweder auf eine Tabelle oder auf Bilddaten. Beispiele: Kantenerkennung, CNN
	Clustering von n-dimensionalen Daten mit der k-means-Methode. Als Metrik wird der Euklidische Abstand genommen. Cluster-Nummern werden an die Daten angehängt.
	Clustering mit beliebiger Metrik. Beispiel: DNS-Clustering mit Levenshtein-Metrik

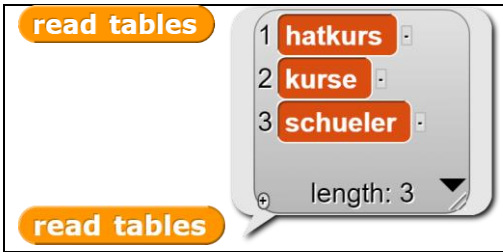
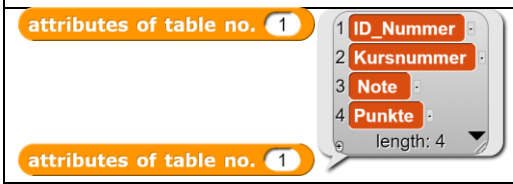
	Liefert die Levenshtein-Distanz zwischen zwei Zeichenketten.
	Clustert Daten nach dem dichtebasierten DBSCAN-Verfahren.
	Liefert einen Entscheidungsbaum für die angegebenen Daten, der nach dem ID3-Verfahren konstruiert wurde. Beispiel: Klassifizierung von Singvögeln
	Klassifiziert Daten mithilfe eines ID3-Baums.

4.4 Die SQL-Bibliothek

Die *SQL*-Bibliothek enthält die meisten Befehle, die für SQL-Abfragen (*Select*) erforderlich sind. Andere *SQL*-Anweisungen können direkt in den *exec SQL-command*-Block eingegeben werden. Allerdings muss man dann auch über die entsprechenden Zugriffsrechte verfügen. Die Bibliothek arbeitet mit der globalen Variablen *SQLData.* Dadurch soll verhindert werden, dass die darin gespeicherten Daten mit denen anderer Sprites in Konflikt geraten. Die Variable wird automatisch bei der Konfiguration erzeugt.

Ähnlich wie die Mathematik- und Data-Blöcke arbeiten die *SQL*-Blöcke nicht mit einem bestimmten Sprite oder der Bühne. Bei jedem Aufruf wird aber zuerst geprüft, ob *Sci-Snap!* erfolgreich für *SQL*-Zugriffe konfiguriert wurde. Ist das nicht der Fall, erfolgt eine Fehlermeldung – bei *Reporter*-Blöcken als Ergebnis des Funktionsaufrufs, bei *Command*-Blöcken als Ausgabe des rufenden Sprites sowie in der *SciSnap!*-Sammelbox für Fehlermeldungen *SciSnap!Messages*.

configure SQL	Konfiguriert *SciSnap!* für *SQL*-Zugriffe. Dafür wird die globale Variable *SQLData* erzeugt und die Anfangs-Eigenschaften werden gesetzt. Das Sprite, das den Befehl ausführt, nimmt das Kostüm *SQLDisconnected* an, falls es vorhanden ist.
is SQL configured?	Liefert bei korrekter Konfiguration „wahr“, sonst „falsch“.
connect to database server	Verbindet mit einem Datenbank-Server, dessen Adresse im Skript steht. Dieser sollte neu eingestellt werden, z. B. als *localhost*, wenn nicht der voreingestellte Server benutzt wird. Ist der Verbindungsversuch erfolgreich, nimmt das ausführende Sprite das Kostüm *SQLConnected* an, falls es vorhanden ist.
import SQL-data from ▤ to SQLData import SQL-data from exec SQL-command SELECT * FROM Kurse WHERE to SQLData	Importiert eine Tabelle in den Datenbereich *SQLData*. Beispiel: Import eines Abfrageergebnisses.
read databases 1 snapex_example 2 snapex_school 3 snapex_world 4 snapextensions_db1 5 snapextensions_db2 6 test length: 6 read databases	Liefert eine Liste der aktuell verfügbaren Datenbanken.
choose database no. 2	Wählt eine der vorhandenen Datenbanken aus.

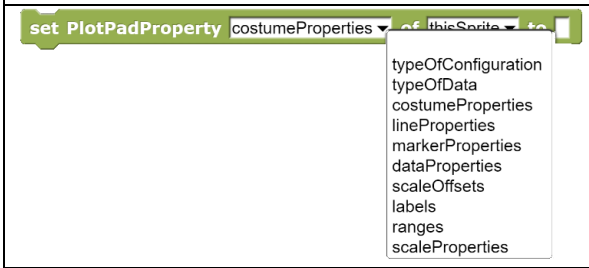
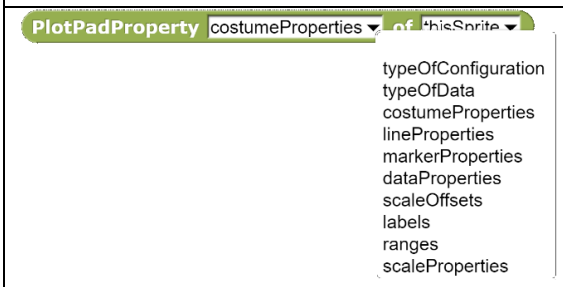
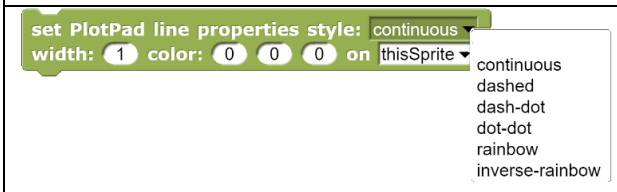
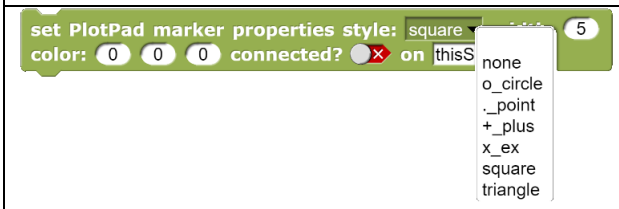
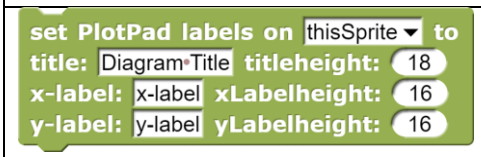
	Liefert eine Liste der Tabellen der ausgewählten Datenbank.
	Wählt eine der vorhandenen Tabellen aus.
	Liefert eine Liste der Attribute der angegebenen Tabelle.
	Block zur Erzeugung einfacher SQL-Abfragen, die vom Block *exec SQL-command* ausgeführt werden können.
	Block zur Erzeugung komplexer SQL-Abfragen, die vom Block *exec SQL-command* ausgeführt werden können.
	Block zur Ausführung von SQL-Anweisungen für eine Datenbank. Die Anweisungen können durch *Select*-Blöcke erzeugt oder direkt eingegeben werden.
	Prädikate zur Ausführung von Vergleichen.
	Prädikate zur Ausführung logischer Verknüpfungen.
	Prädikat zur Überprüfung eines Zeichenkettenmusters.
	Prädikat zum Überprüfen, ob ein Element in einer Aufzählung enthalten ist.
	Aggregatfunktionen.

Die folgenden *SciSnap!*-Bibliotheken arbeiten mit lokal konfigurierten *Spezial-Sprites*. Diese sind konzeptionell von *Sketchpads* abgeleitet, dienen also zur Anfertigung von Skizzen, zum Experimentieren, zum Ausprobieren der Wirkung von Befehlen usw. Ist das *Pad* zu voll, dann wird eben eine neue „Seite“ davon genommen und weitergearbeitet.

Jedes Sprite und die Bühne können als *Spezial-Sprite* dienen. Dafür werden sie entsprechend konfiguriert, indem zwei lokale Variable *myData* und *myProperties* erzeugt und mit Anfangswerten gefüllt werden. Die Befehle, die sich auf ein *Spezial-Sprite* beziehen, enthalten alle das Ziel der Operation – also ein Sprite oder die Bühne. Vor der Ausführung der Anweisungen wird jeweils überprüft, ob das Ziel richtig konfiguriert worden ist. Danach wird mit den lokalen Daten und Eigenschaften des Ziels gearbeitet. Dieses Vorgehen erübrigt einerseits weitgehend objektorientierte Aufrufe, die die Befehlsblöcke sehr verlängern, wenn Daten übergeben werden müssen, andererseits hält es die Daten lokal bei den Sprites, die die Operationen betreffen. Werden z. B. Daten in einem Bild gemessen und in einem Diagramm dargestellt, dann können das *ImagePad* und das *PlotPad* unabhängig voneinander mit ihren jeweils eigenen Daten und Eigenschaften arbeiten. Die Anfangseinstellungen ermöglichen es, mit halbwegs sinnvollen Voreinstellungen zu arbeiten. Stellt sich im Ergebnis heraus, dass andere Einstellungen sinnvoller wären, dann werden die Voreinstellungen geändert – aber auch nur dann. Diese Arbeitsweise ermöglicht es, bei den einzelnen Blöcken mit relativ wenigen Parametern auszukommen. Die Properties sind weitgehend zu Gruppen zusammengefasst, etwa bei den Eigenschaften der zu zeichnenden Linien. Damit können sie „auf einen Schlag“ übergeben oder gelesen werden, und ihre Anzahl hält sich in Grenzen.

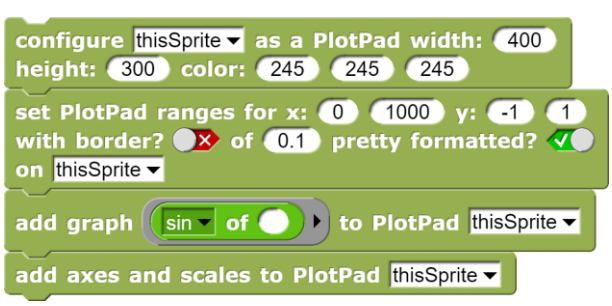
4.5 Die PlotPad-Bibliothek

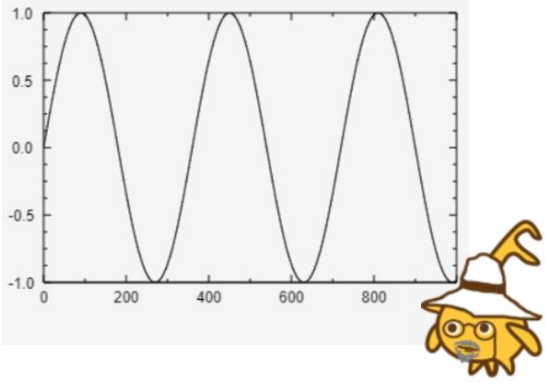
PlotPads dienen zur Darstellung von Diagrammen, Histogrammen usw. Die aktuelle *Bibliothek* wurde stark von *Rick Hessman* gestaltet, insbesondere das *PrettyPrinting* und *Linienstile* stammen von ihm. Vielen Dank dafür!

	Konfiguriert ein Sprite oder die Bühne als *PlotPad*. Der Name von *„anotherSprite“* muss bei Bedarf angegeben werden. Der Befehl ist einmal auszuführen, bevor mit einem Sprite als *PlotPad* gearbeitet werden kann. Das Ziel des Aufrufs nimmt ein rechteckiges Kostüm mit den angegebenen Maßen und Farben an.
	Liefert bei korrekter Konfiguration „wahr“, sonst „falsch“.
	Setzt eine der Eigenschaften in *myProperties* auf den angegebenen Wert.
	Liefert eine der Eigenschaften in *myProperties*.
	Setzt die Kostüm-Eigenschaften des angegebenen Ziels.
	Setzt die Linien-Eigenschaften des angegebenen Ziels.
	Setzt die Datenpunkt-Eigenschaften des angegebenen Ziels.
	Setzt die Skalen-Eigenschaften des angegebenen Ziels.
	Setzt die Beschriftungs-Eigenschaften des angegebenen Ziels.
	Berechnet die Abstände der Koordinatenachsen von den Rändern.

set PlotPad ranges for x: -10 10 y: -10 10 with border? of 0.1 pretty formatted? on thisSprite	Setzt die Zahlenbereiche der Achsen des Koordinatensystems.
add graph ringified operator or polynomial to PlotPad thisSprite	Fügt dem *PlotPad* einen Funktionsgraphen hinzu, der als Liste von Polynom-Koeffizienten oder als „ringified“ Term gegeben ist.
add dataplot of numeric data: myData to PlotPad thisSprite	Fügt dem *PlotPad* einen Datenplot für eine zweidimensionale Datentabelle hinzu.
add dataplot of mixed data: myData y-scale? x-scale? to PlotPad thisSprite	Fügt dem *PlotPad* einen Datenplot für eine zweidimensionale Tabelle hinzu, die in der ersten Spalte Texte, in der zweiten Zahlenwerte enthält.
add histogram of myData with 10 groups pretty formated? to PlotPad thisSprite	Fügt dem *PlotPad* ein Histogramm hinzu.
add axes and scales to PlotPad thisSprite	Fügt dem *PlotPad* Achsen und Beschriftungen hinzu. Je nach Geschmack kann man diesen Block auch den eigentlichen Plot-Befehlen am Ende hinzufügen.
clear plot of thisSprite	Löscht das *PlotPad*, ohne die anderen Voreinstellungen zu verändern.
set pretty ranges on PlotPad thisSprite	Setzt die Zahlenbereiche so, dass „schöne“ Beschriftungen an den Achsen stehen.
pretty values for a PlotPad from -10 to 10 with 6 intervals	Liefert Werte für „schöne“ Beschriftungen der Achsen.
get ranges for PlotPad thisSprite from myData with border 0.1	Bestimmt die Zahlenbereiche neu, berechnet aus den Daten.
ranges of 2-dim table	Liefert die Zahlenbereiche einer zweidimensionalen Tabelle.
convert value 100 to coordinate xp of PlotPad thisSprite (xp, yp, x, y)	Liefert die Umrechnung eines Zahlenwerts in Koordinaten des *Plotpads* oder des Koordinatensystems.
PlotPad costume-coordinates … for mouse (costume-coordinates, graph-coordinates)	Rechnet die Mausposition in Kostüm- bzw. Graph-Koordinaten um.
Example 3: Simple plot of data: x: 0 y: 0 width: 600 height: 400 title: labels: line: continuous marker: square color: 0 0 0	Einfaches Plotprogramm für Daten.

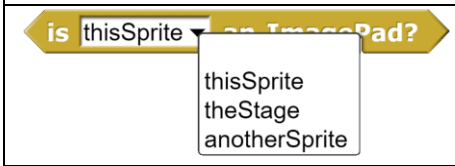
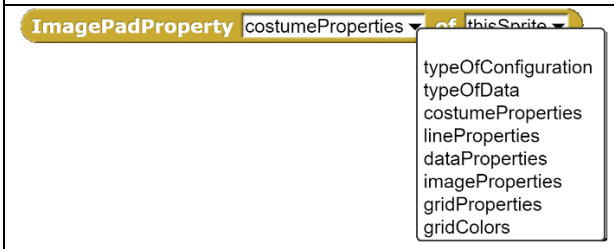
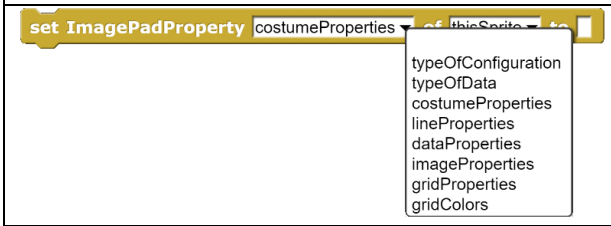
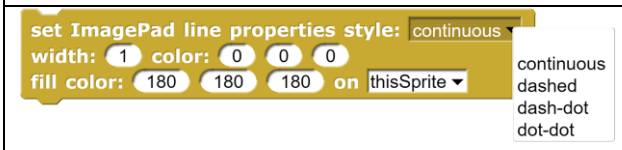
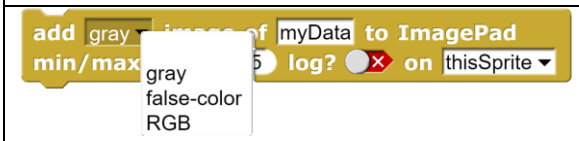
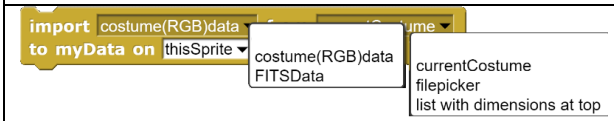
Beispiel:

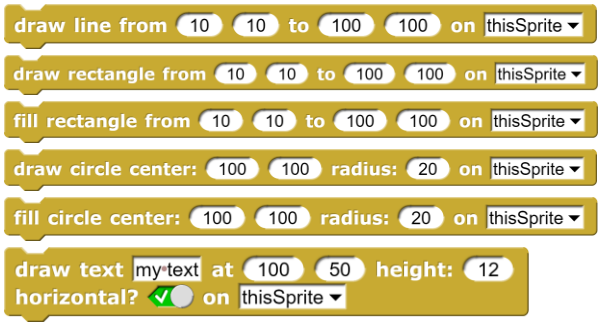
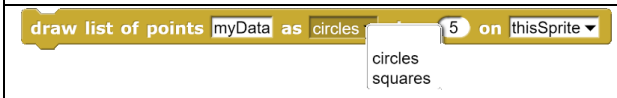
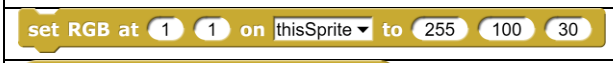
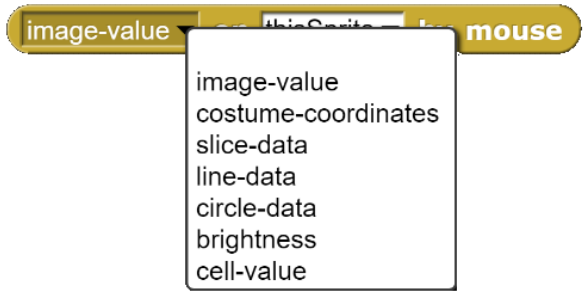
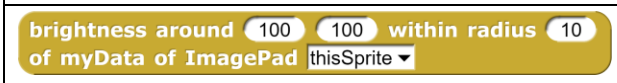
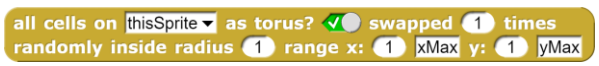
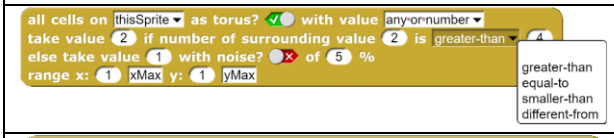
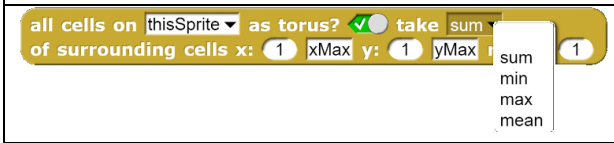


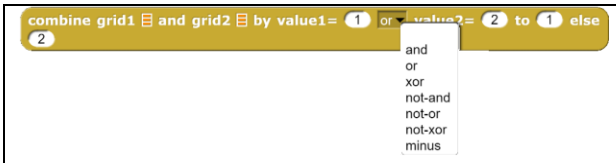


4.6 Die ImagePad-Bibliothek

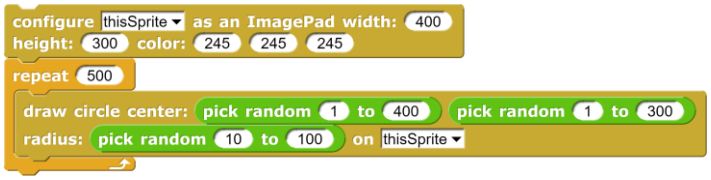
ImagePads dienen zur Darstellung von Bilddaten, also zur Erzeugung von Bildern. In diesen Bildern wird dann mithilfe der Maus gemessen – z. B. kann ein Schnitt durch das Bild gelegt werden. Zusätzlich stehen einige der üblichen Operationen zum Zeichnen von Linien, Rechtecken, Kreisen und Texten zur Verfügung. Zusätzlich können Gitter dargestellt und Operationen auf Gittern ausgeführt werden. Das Koordinatensystem auf *ImagePads* ist das für Bilder übliche: der Ursprung liegt in der oberen linken Ecke und die y-Achse ist nach unten gerichtet.

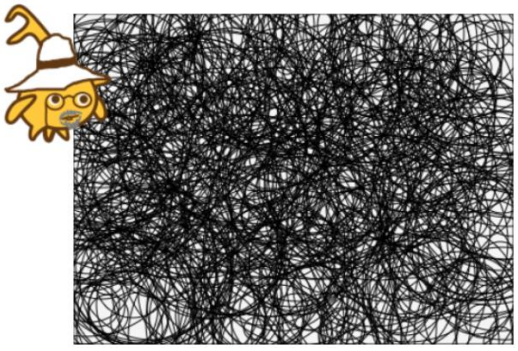
	Konfiguriert ein Sprite oder die Bühne als *ImagePad*. Der Name von „*anotherSprite*“ muss bei Bedarf angegeben werden. Der Befehl ist einmal auszuführen, bevor mit einem Sprite als *ImagePad* gearbeitet werden kann. Das Ziel des Aufrufs nimmt ein rechteckiges Kostüm mit den angegebenen Maßen und Farben an.
	Liefert bei korrekter Konfiguration „wahr“, sonst „falsch“.
	Liefert eine der Eigenschaften in *myProperties*.
	Setzt eine der Eigenschaften in *myProperties* auf den angegebenen Wert.
	Setzt die Kostüm-Eigenschaften des angegebenen Ziels.
	Setzt die Linien-Eigenschaften des angegebenen Ziels.
	Setzt die Werte für die Dimensionen des Gitters.
	Erzeugt aus den Bilddaten ein Bild auf einem *ImagePad*.
	Stellt die Daten des Gitters in *myData* oder einer anderen Datei auf dem *ImagePad* dar.
	Importiert Bilddaten in den Bereich *myData* des ausgewählten Sprites.

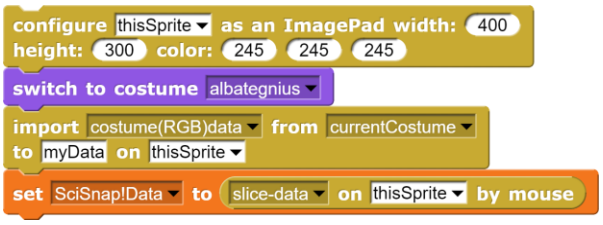
	elementare Zeichenoperationen
	Zeichnet eine Liste von "Punkten" als Kreise oder Quadrate. Achtung! Es werden JS-Koordinaten verwendet!
	Setzt ein Pixel des Kostüms auf den angegebenen Wert.
	Liefert den Wert eines Pixels des Kostüms.
	Setzt einen Bildpunkt im Datenbereich auf den angegebenen Wert.
	Liefert den Wert eines Bildpunktes aus dem Datenbereich.
	Liefert Bilddaten mithilfe der Maus: einzelne Bildwerte, Bildkoordinaten, Schnitte durch das Bild, Endpunkte einer Linie oder Daten eines Kreises sowie Helligkeitswerte aus einem Bereich oder einen Gitterwert. Die Bilddaten müssen sich in *myData* befinden.
	Liefert das Ergebnis einer affinen Transformation eines Kostüms, die durch die Angabe von drei Originalpunkten und drei Bildpunkten beschrieben wird.
	Liefert die Gesamthelligkeit um einen Bildpunkt im angegebenen Radius.
	Füllt die Zellen eines Gitters zufällig mit einer der angegebenen Zahlen.
	Setzen von Zellinhalten mithilfe der Maus.
	Direktes Setzen von Zellinhalten.
	Bestimmt die angegebene Nachbarschaft einer Gitterzelle.
	Vertauscht zufällig die Werte von Gitterzellen mit denen von Nachbarzellen aus dem angegebenen Bereich.
	Ersetzt die Zellwerte aus dem angegebenen Bereich unter den angegebenen Bedingungen.
	Ersetzt die Zellwerte aus dem angegebenen Bereich durch das Ergebnis der angegebenen Aggregatfunktion.

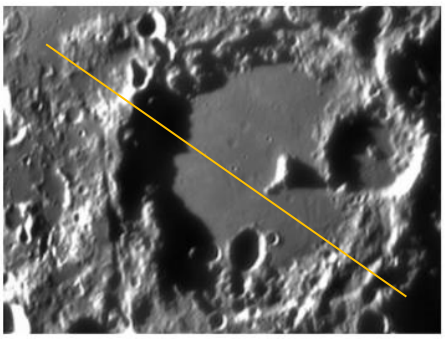
	Überlagert zwei Gitter unter Anwendung der angegebenen Operation.
	Wendet die Operationen des angegebenen linearen Wolfram-Automaten auf ein Gitter von der ersten Zeile an auf alle folgenden an.

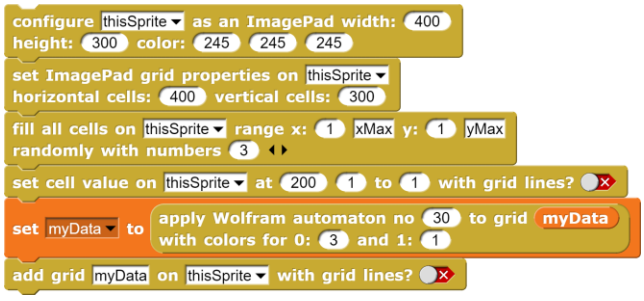
Beispiele:

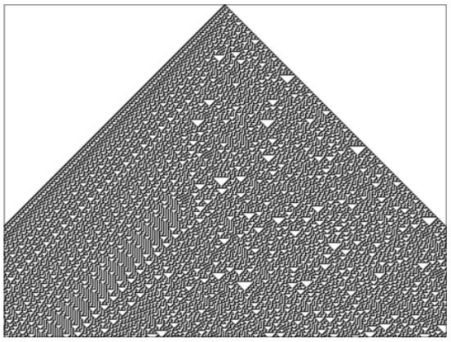






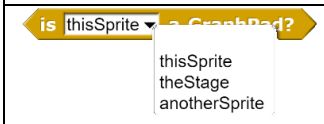






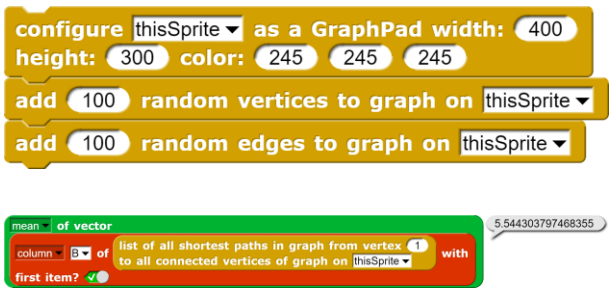
4.6 Die GraphPad-Bibliothek

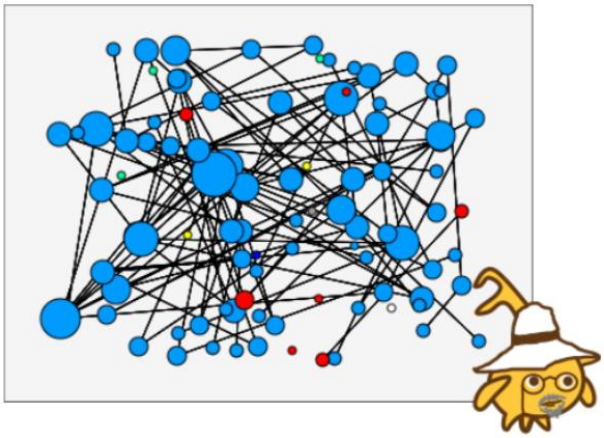
Graphen sind eines der mächtigsten Modelle der Informatik. Mit ihrer Hilfe lassen sich komplexe Systeme, vor allem aber die Auswirkung der Vernetzung zahlreicher ähnlicher Teilsysteme, studieren. Die dafür erforderlichen Algorithmen wie Breitensuche oder Routing sind selbst aber nicht trivial, sodass es sinnvoll erscheint, diese als Basisbefehle zur Verfügung zu stellen, um die Arbeit auf die Modellierung selbst zu konzentrieren. Genau das ist der Zweck des *GraphPads*.

	Konfiguriert ein Sprite oder die Bühne als *GraphPad*. Der Name von *„anotherSprite“* muss bei Bedarf angegeben werden. Der Befehl ist einmal auszuführen, bevor mit einem Sprite als *GraphPad* gearbeitet werden kann. Das Ziel des Aufrufs nimmt ein rechteckiges Kostüm mit den angegebenen Maßen und Farben an.
	Liefert bei korrekter Konfiguration „wahr“, sonst „falsch“.
	Setzt eine der Eigenschaften in *myProperties* auf den angegebenen Wert.
	Liefert eine der Eigenschaften in *myProperties*.
	Setzt die Kostüm-Eigenschaften des angegebenen Ziels.
	Setzt die Knoten-Eigenschaften des angegebenen Ziels.
	Setzt die Kanten-Eigenschaften des angegebenen Ziels.
	Fügt dem Graph n Knoten an Zufallspositionen hinzu.
	Fügt dem Graph einen Knoten an der angegebenen Position hinzu.
	Bewegt einen Knoten zu der angegebenen Position.
	Fügt dem Graph n zufällig gewählte Kanten hinzu.
	Fügt dem Graph eine Kante zwischen den genannten Knoten hinzu.
	Zeichnet den Graph. Färbt verbundene Knoten in der gleichen Farbe. Wird auf der Bühne gezeichnet, dann bleibt das anfängliche Hintergrundbild erhalten, z. B. um Karten benutzen zu können.
	Löscht einen Knoten des Graphen.
	Löscht eine Kante des Graphen.

weight of edge from vertex 1 to vertex 2 of graph on thisSprite	Liefert das Gewicht einer Kante, wenn möglich.
change weight of edge from vertex 1 to vertex 2 to 1 of graph on thisSprite	Verändert das Gewicht einer Kante, wenn möglich.
ask for new weight of graph on thisSprite	Erfragt ein neues Kantengewicht.
ask for new start vertex width of graph on thisSprite	Erfragt eine neue Start-Knotengröße.
content of vertex 1 of graph on thisSprite	Liefert den Inhalt eines Knotens.
change content of vertex 1 to of graph on thisSprite	Verändert den Inhalt eines Knotens.
ask for new vertex content in graph on thisSprite	Erfragt einen neuen Knoteninhalt.
set marker of vertex 1 of graph on thisSprite	Markiert einen Knoten.
remove marker of vertex 1 of graph on thisSprite	Löscht die Markierung eines Knotens.
remove all markers of graph on thisSprite	Löscht alle Markierungen im Graph.
depth first search of content starting at vertex 1 of graph on thisSprite	Tiefensuche ab Knoten mit der genannten Nummer nach einem Inhalt.
breadth first search of content starting at vertex 1 of graph on thisSprite	Breitensuche ab Knoten mit der genannten Nummer nach einem Inhalt.
distance on thisSprite from vertex 1 to vertex 2	Räumlicher Abstand zweier Knoten auf dem Sprite.
shortest path in graph from vertex 1 to vertex 2 on thisSprite	Kürzester Weg zwischen zwei Knoten im Graph.
list of all shortest paths in graph from vertex 1 to all connected vertices of graph on thisSprite	Liste aller kürzesten Wege eines Knotens zu allen verbundenen Knoten im Graph.
vertexnumber at 100 50 of graph on thisSprite	Nummer eines Knotens an der genannten Position auf dem Sprite.
point 0 0 on sprite/stage ⇨ point on graph thisSprite	Rechnet Sprite/Stage-Koordinaten in Graph-(JS)-Koordinaten um.
vertexnumber of Peter in graph of thisSprite	Liefert die Nummer des Knotens mit dem angegebenen Inhalt.
vertexnumber of graph on thisSprite at mouse position (thisSprite, theStage, anotherSprite)	Liefert die Nummer des Knotens an der Mausposition.

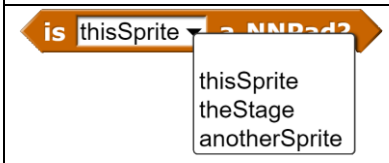
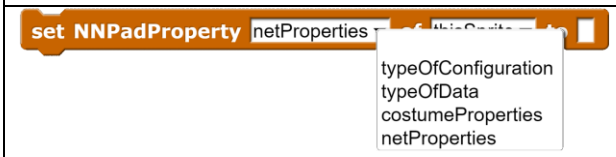
Beispiel:



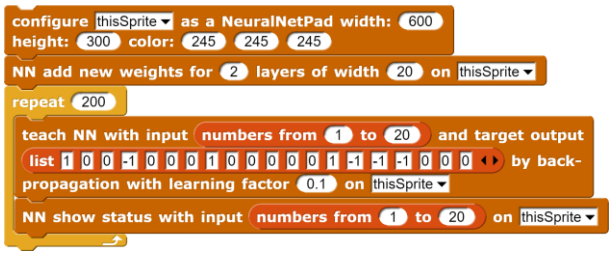


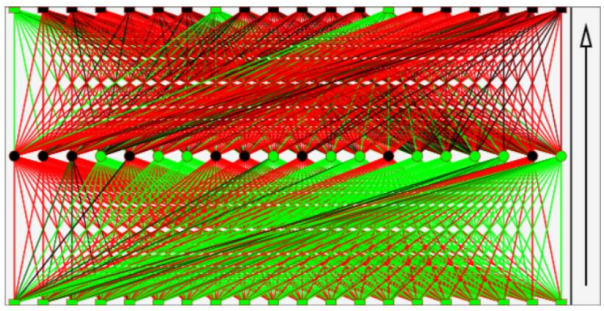
4.7 Die NeuralNetPad-Bibliothek

Die Grundprinzipien Neuronaler Netze sind zwar einfach und einleuchtend, die Lernverfahren der Systeme aber mit ihrem diskreten Gradientenabstieg und den damit verbundenen partiellen Ableitungen für Mathematikferne schwer zu durchdringen. Insbesondere ist nicht so leicht zu verstehen, was ein Neuronales Netz eigentlich gelernt hat. Das *NNPad* soll den Umgang mit Neuronalen Netzen auf einer Zwischenebene zwischen sehr kleinen und wirklich großen Netzen ermöglichen. Sie veranschaulicht die Belegung der Kanten durch Färbungen, wobei positive Werte in grünen und negative in roten Farben dargestellt werden. Kleine Werte sind eher schwarz. Die Bibliothek erleichtert das Erzeugen und das Training voll verbundener Perzeptron-Netze.

	Konfiguriert ein Sprite oder die Bühne als *NNPad*. Der Name von *„anotherSprite“* muss bei Bedarf angegeben werden. Der Befehl ist einmal auszuführen, bevor mit einem Sprite als *NNPad* gearbeitet werden kann. Das Ziel des Aufrufs nimmt ein rechteckiges Kostüm mit den angegebenen Maßen und Farben an.
	Liefert bei korrekter Konfiguration „wahr“, sonst „falsch“.
	Setzt eine der Eigenschaften in *myProperties* auf den angegebenen Wert.
	Liefert eine der Eigenschaften in *myProperties*.
	Setzt die Kostüm-Eigenschaften des angegebenen Ziels.
	Setzt die Schicht-Eigenschaften des angegebenen Ziels.
	Fügt zufällige Gewichte für ein NN der angegebenen Breite und Tiefe ein.
	Liefert die Ausgabe der n-ten Schicht des NN beim angegebenen Eingabevektor.
	Zeigt den Status des Netzes beim angegebenen Eingabevektor farblich kodiert an.
	Trainiert das NN beim angegebenen Eingabevektor zur Erreichung des gegebene Ausgabevektors.

Beispiel: Ein Neuronales Netz wird trainiert, das angegebene Muster zu erzeugen, wenn die Zahlen 1 bis 20 an den Eingängen anliegen.





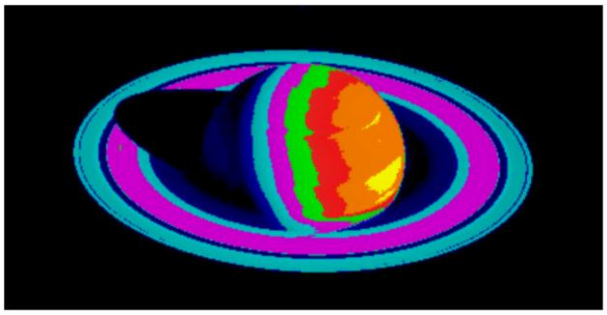
5 Datenimport und -export

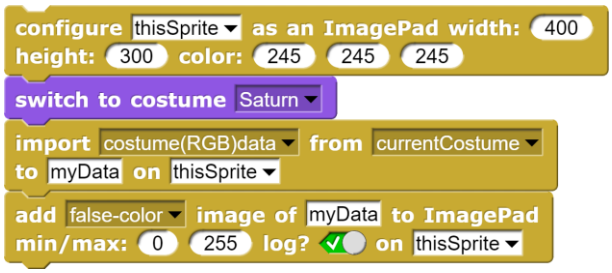
Snap! kann eine Reihe von Datenformaten direkt importieren. Das kann geschehen, indem man entsprechende Dateien auf das *Snap!*-Fenster „fallen“ lässt oder sie durch Rechtsklick auf einen Variablen-Watcher[15] importiert. Beides klappt gut mit Text-, CSV- und JSON-Dateien. Andere Text-Dateiformate wie FITS kann man ebenfalls so importieren, wobei nachgefragt wird, ob man es ernst meint. Das Exportieren funktioniert auf die gleiche Art. Die Daten landen dann im Download-Ordner des Browsers. Will man den Datenimport programmgesteuert durchführen, dann benutzt man die Option ***filepicker*** bei den Importblöcken. Es erscheint ein Dateimanager-Fenster, in dem man die Datei wie üblich auswählt. Danach werden die Daten importiert.

Als wesentliche Aufgabe bleibt anschließend, diese Daten der *Sci-Snap!Data*-Variablen zuzuweisen und die entsprechenden Eigenschaften in *SciSnap!Properties* zu setzen. Das wird von dem Import-Block erledigt, der Daten von außen in den *SciSnap!Data*-Bereich importiert. Dabei kann es sich um Bilddaten, Tabellendaten oder die Daten des aktuellen Kostüms handeln. Dieses wird als Tabelle von RGB-Werten gespeichert.

Beispiel: Falschfarbenbild

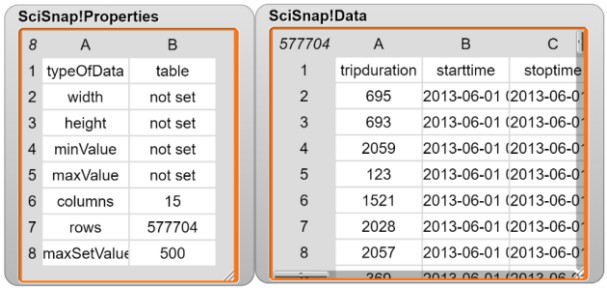
Durch ein *ImagePad* wird ein Bild (Quelle: [NASA]) gespeichert und mit Falschfarben neu dargestellt.





Beispiel: CSV-Import

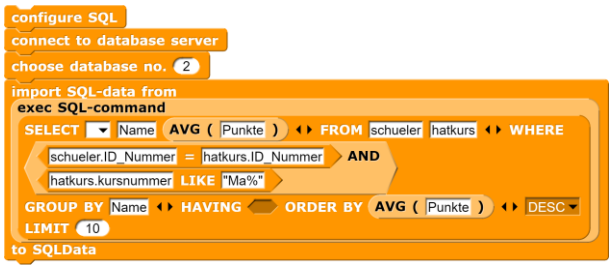
Knapp 600000 Datensätze aus einer CSV-Datei werden in etwa 10 Sekunden eingelesen. Die Eigenschaften werden gesetzt.



[15] Einen Variablen-Watcher erhält man, wenn man im Kästchen neben der Variablen einen Haken setzt.

Beispiel: SQL-Import

Haben wir Zugang zu einem SQL-Server, dann können wir auch von dort Daten einlesen. In unserem Fall importieren wir mithilfe der *SciSnap! SQL-tools* die Ergebnisse einer Abfrage in die Variable *SQLData*. Dabei werden die Daten in Tabellenform umgesetzt und ihre relevanten Eigenschaften wie Anzahl der Spalten und Zeilen, … in den *SQL-Properties* neu gesetzt.



SciSnap!Properties

16	A	B
1	typeOfData	table
2	width	not set
3	height	not set
4	minValue	notSet
5	maxValue	notSet
6	columns	2
7	rows	10
8	maxSetValue	500
9	typeOfConfig	SQL
10	connection	https://snape
11	connected	true
12	databases	
13	currentDatab	snapex_schc
14	tables	
15	currentTable	
16	attributes	

SQLData

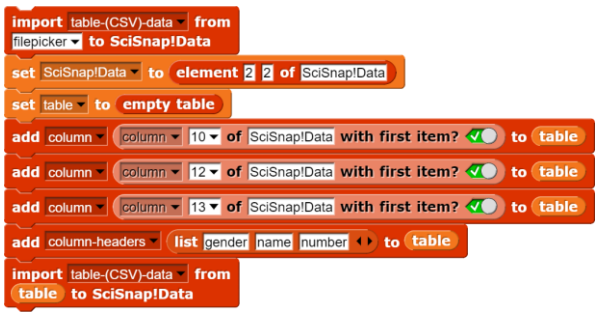
10	A	B
1	Kirsche	13.7500
2	Pogenberg	13.5000
3	Rassin	13.5000
4	Karbel	12.7500
5	Krahn	12.0000
6	Rawe	12.0000
7	Gallus	11.7500
8	Ruf	11.5000
9	Boemmel	11.5000
10	Vestmann	11.0000

Beispiel: JSON-Import

Der einfachste Weg ist auch hier, eine JSON-Datei einfach ins *Snap!*-Fenster „fallen“ zu lassen. Es geht aber auch automatisiert. Zuerst einmal suchen wir uns interessante JSON-Daten und wählen dafür natürlich die Statistik der Baby-Namen in New York City – was sonst. Der geeignete Block dafür ist wieder ***import <table data> from <filepicker> to SciSnap!Data***. Das Ergebnis ist eine Liste mit zwei Spalten und zwei Reihen, den Metadaten und den eigentlichen Daten. Weil wir uns für die interessieren, ersetzen wir die Originaldaten durch das Element (2|2) der Tabelle. Natürlich haben wir uns vorher die einzelnen Elemente in Tabellenform angesehen, um zu prüfen, was wir da überhaupt geladen haben. Von den vielen Spalten kopieren wir die drei interessanten in eine neue Tabelle, fügen Spaltenüberschriften hinzu und importieren das Ergebnis wieder in *SciSnap!Data*.

SciSnap!Data

2	A	B
1	meta	
2	data	



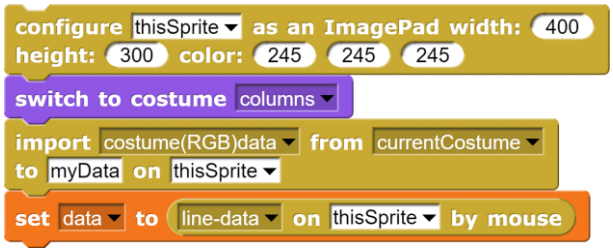
table

19419	A	B	C
1	gender	name	number
2	FEMALE	Olivia	172
3	FEMALE	Chloe	112
4	FEMALE	Sophia	104
5	FEMALE	Emily	99
6	FEMALE	Emma	99
7	FEMALE	Mia	79
8	FEMALE	Charlotte	59
9	FEMALE	Sarah	57
10	FEMALE	Isabella	56
11	FEMALE	Hannah	56
12	FEMALE	Grace	54
13	FEMALE	Angela	54
14	FEMALE	Ava	53
15	FEMALE	Joanna	49

Das Ergebnis: 19419 Babynamen - Wer hätte das gedacht!

Beispiel: Datenimport mit der Maus

In vielen Fällen ist es gerade bei Bildern vorteilhaft, Daten mithilfe der Maus einzulesen. Dafür verfügen *ImagePads* über einen Block, mit dem Bildwerte, Bildkoordinaten, die Daten auf einem Schnitt durchs Bild, Anfangs- und Endpunkt einer Linie, Mittelpunkt und Radius eines Kreises und die summierten Helligkeitswerte zusammen mit deren Zahl in einem Kreis bestimmt werden können. Als Beispiel soll die Höhe antiker Säulen vermessen werden. Dazu wird das Kostümbild des *ImagePads* mit den Säulen importiert und anschließend mit der Maus ausgemessen (gelbe Linie).



data

2	A	B
1	122	42
2	125	172

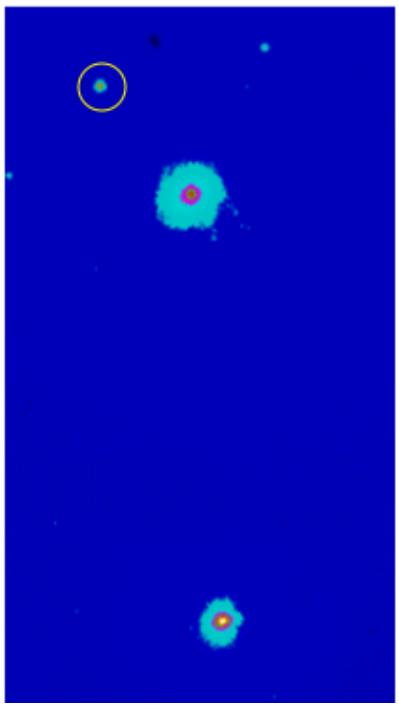
Als zweites Beispiel für das Messen mit der Maus wollen wir die Gesamthelligkeit innerhalb eines Kreises um ein Sternfoto messen (Quelle: [HOU]).

Bei Graustufenbildern wie FITS erhalten wir die Gesamthelligkeit und die Zahl der gemessenen Pixel, bei RGB-Bildern die Helligkeit der drei Farben und die Pixelzahl.

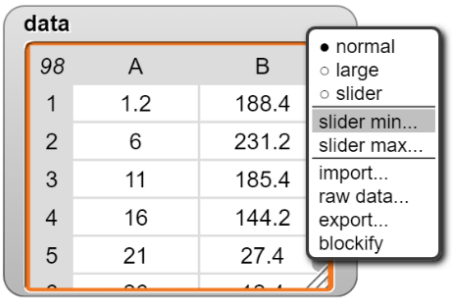
data

1	952
2	15

length: 2



Der Export von Daten kann wiederum direkt aus einem Variablen-Watcher geschehen.



Für Skripte gibt es zwei neue Blöcke ***write <table> to CSV file <filename>*** sowie ***write string <string> to file <filename>***. Die Ergebnisse landen wie in *Snap!* üblich jeweils im Download-Ordner des Browsers. Die beiden Blöcke gestatten es, den Datenaustausch mit Tabellenkalkulationsprogrammen bzw. über Textdateien zu automatisieren, beispielsweise um Ergebnisse der Datenverarbeitung zu sichern.

6 Mathematikbezogene Beispiele

6.1 Darstellung komplexer Zahlen

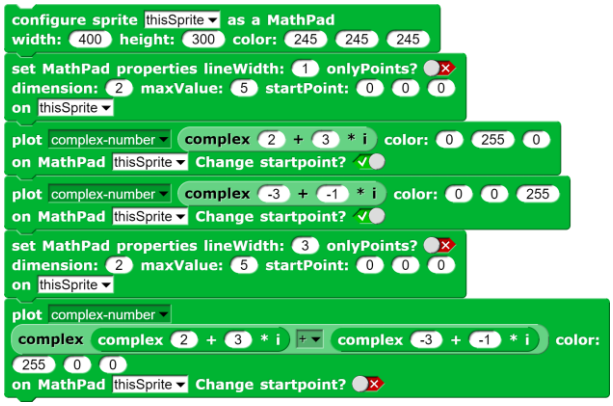
Die Operationen mit komplexen Zahlen können in *SciSnap!* leicht veranschaulicht werden, indem man das *MathPad* benutzt: eine Sprite-Konfiguration, mit der man z. B. komplexe Zahlen als Pfeile darstellen kann. Da die komplexe Ebene zwei Dimensionen hat, müssen wir die Voreinstellung (3 Dimensionen) ändern, danach stellen wir zwei komplexe Zahlen und deren Summe verschiedenfarbig dar.

Das aktuelle Sprite als MathPad in der angegebenen Größe und Farbe erzeugen. Danach Dimension usw. einstellen.

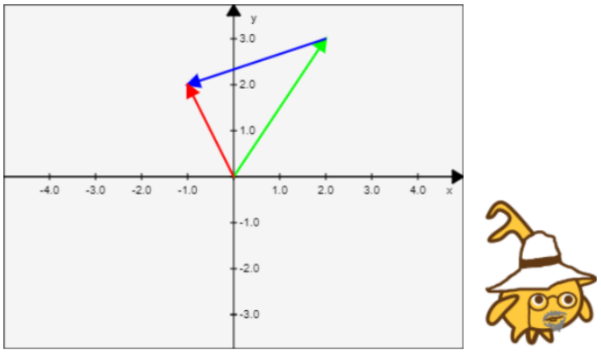
Erste Zahl in Grün zeichnen, dabei Startpunkt verschieben.

Zweite Zahl in Blau zeichnen. Danach Startpunkt wieder in den Ursprung verschieben.

Die Summe in Rot vom Ursprung aus zeichnen.



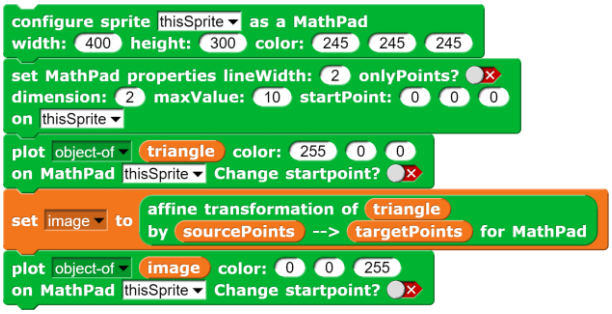
Da es sich um ein mathematisches Beispiel handelt, muss die Mitwirkung von Hilberto natürlich durch seine Mit-Darstellung angemessen gewürdigt werden.

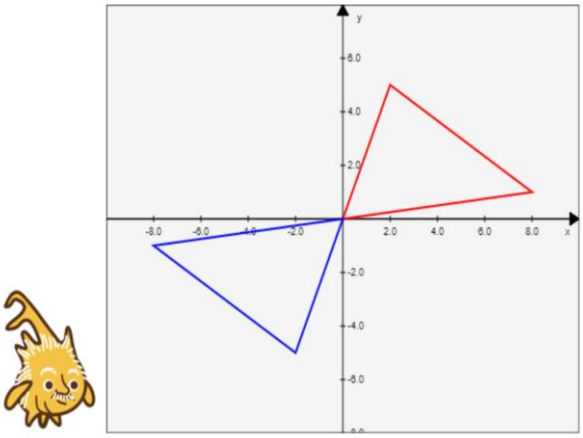


6.2 Affine Transformation eines Dreiecks im R^2

Wir definieren ein Dreieck durch seine Ortsvektoren. Dann definieren wir drei Punkte in der Ebene sowie drei Punkte, auf die diese abgebildet werden sollen.

Danach erzeugen wir ein zweidimensionales *MathPad*, ändern den Maximalwert der Achsen und zeichnen das Dreieck in Rot. Auf seine Ortsvektoren wenden wir die affine Transformation an und zeichnen das Ergebnis in Blau. Da Koordinatentransformationen eher etwas für die Physik sind, stellt *Alberto* das Ergebnis vor.

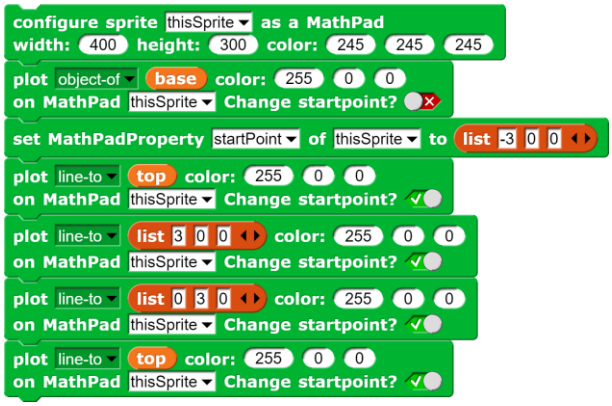


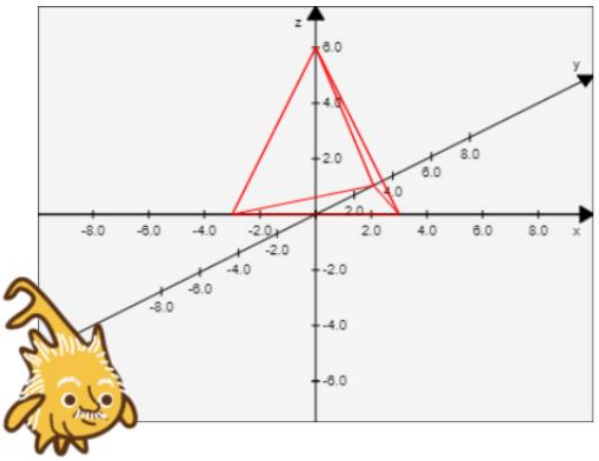


6.3 Drehung einer Pyramide im R^3

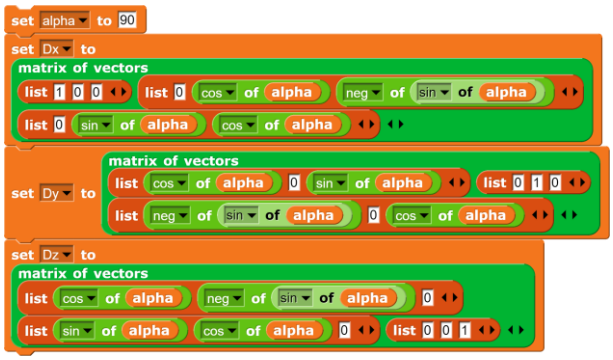
Zuerst wollen wir natürlich eine Pyramide zeichnen. Wir definieren die Grundfläche und die Spitze durch Ortsvektoren:

Wir lassen sie zeichnen, indem zuerst die Grundfläche gezeichnet wird, danach Linien von deren Ecken zur Spitze.

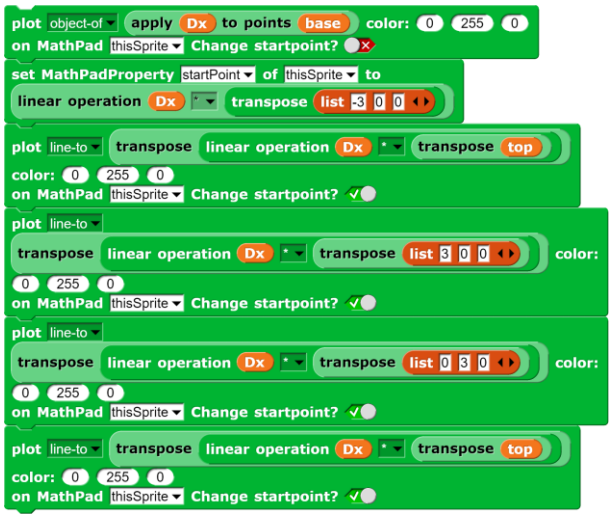


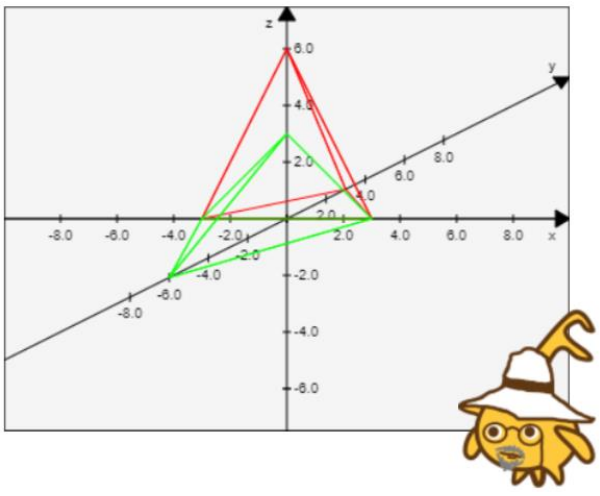


Für Drehungen um die Achsen benötigen wir die drei Drehmatrizen D_x, D_y und D_z. Für eine Drehung um die x-Achse um 90° können wir die Matrix direkt auf die Grundfläche anwenden. Danach lassen wir die Seitenlinien zur gedrehten Spitze zeichnen, indem wir die Drehmatrix mit den transponierten Ortsvektoren der Eckpunkte multiplizieren. Da wir – mathematisch korrekt – Matrizen mit Spaltenvektoren multiplizieren und als Resultate Spaltenvektoren erhalten, müssen wir diese nochmals transponieren, um zu normalen Ortsvektoren zu kommen.



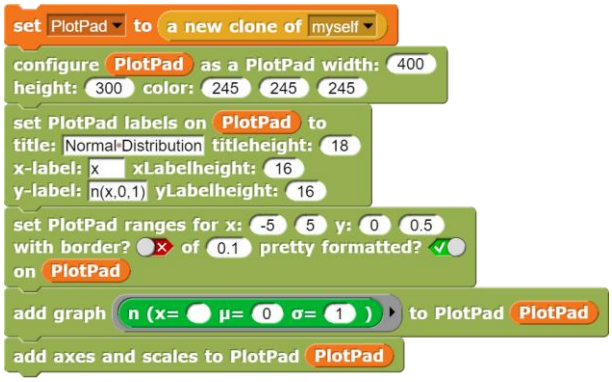
Insgesamt erhalten wir:

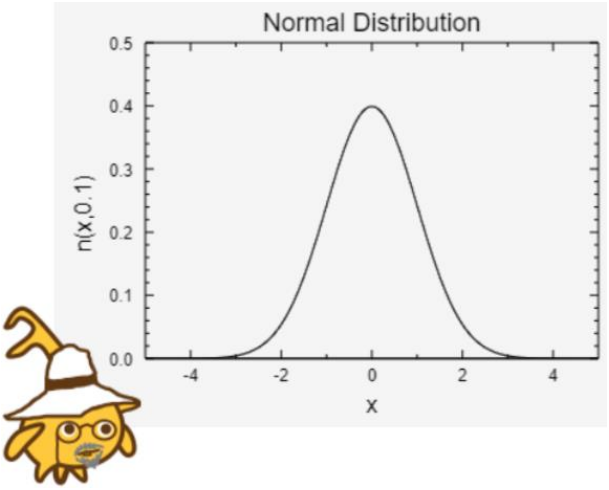




6.4 Graph der Normalverteilung

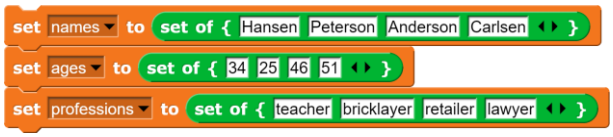
Mithilfe eines *PlotPads* wird der Graph der Normalverteilung gezeichnet. Dazu erzeugen wir einen Klon von *Hilberto,* konfigurieren den als *PlotPad* und erstellen den Graph.



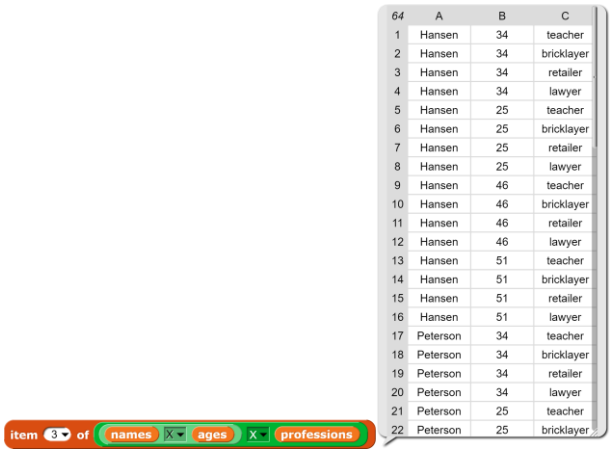


6.5 Kartesisches Produkt dreier Mengen

Zuerst einmal erzeugen wir drei Mengen mit Namen, möglichen Altern und Berufen:



Aus diesen Mengen können wir nun das kartesische Produkt aus Namen, Alterswerten und Berufen bilden:

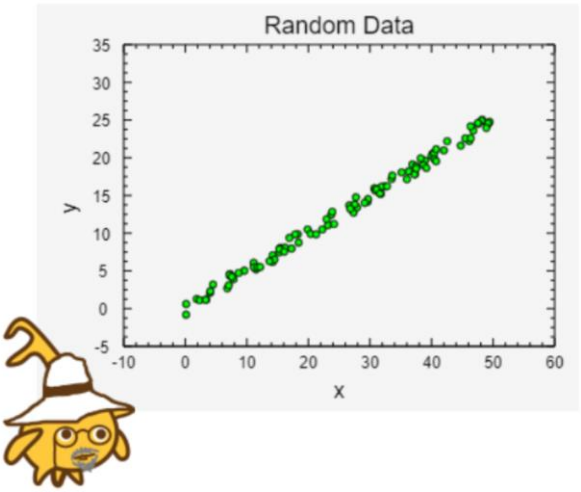


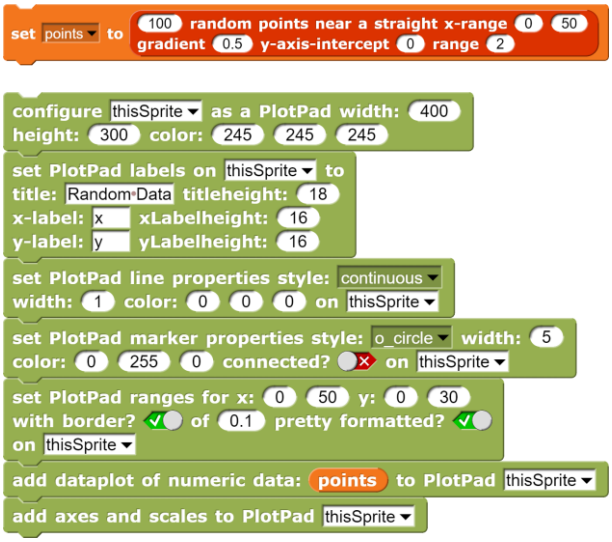
64	A	B	C
1	Hansen	34	teacher
2	Hansen	34	bricklayer
3	Hansen	34	retailer
4	Hansen	34	lawyer
5	Hansen	25	teacher
6	Hansen	25	bricklayer
7	Hansen	25	retailer
8	Hansen	25	lawyer
9	Hansen	46	teacher
10	Hansen	46	bricklayer
11	Hansen	46	retailer
12	Hansen	46	lawyer
13	Hansen	51	teacher
14	Hansen	51	bricklayer
15	Hansen	51	retailer
16	Hansen	51	lawyer
17	Peterson	34	teacher
18	Peterson	34	bricklayer
19	Peterson	34	retailer
20	Peterson	34	lawyer
21	Peterson	25	teacher
22	Peterson	25	bricklayer

Und damit steht einem Übergang z. B. zum Thema „relationale Datenbanken“ nichts mehr im Wege.

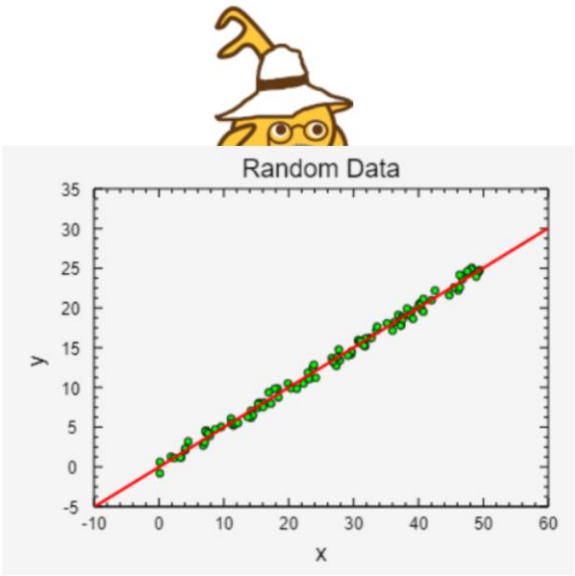
6.6 Darstellung einer Punktmenge und der Regressionsgeraden

Wir erzeugen mithilfe der *Data tools* 100 Zufallspunkte, die um eine Gerade mit der Steigung *m=0.5* und dem Achsenabschnitt *b=0* streuen. Die erhaltenen Punkte stellen wir in einem Diagramm dar.





Zusätzlich wird jetzt noch die Regressionsgerade eingezeichnet.

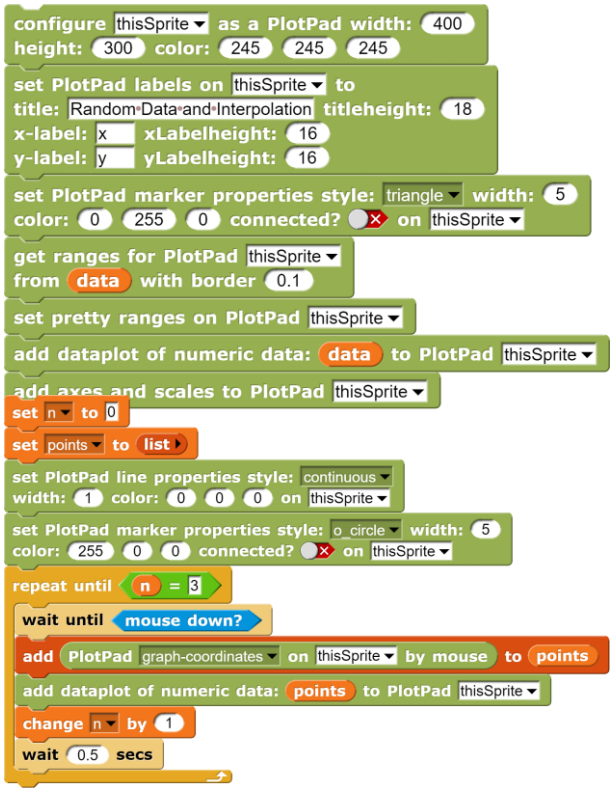


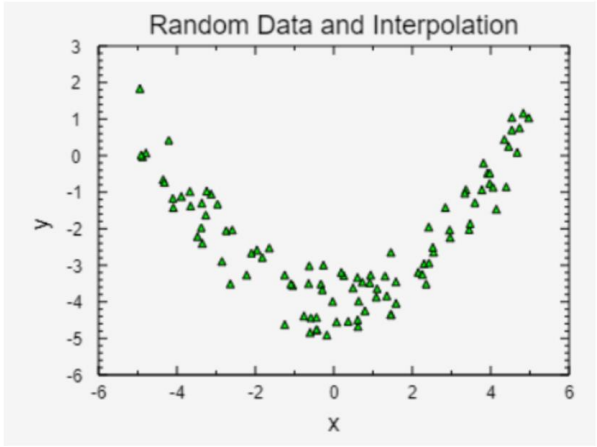
6.7 Interpolationspolynom durch n Punkte

Wir wollen eine Datenmenge von 100 Punkten erzeugen, die um eine vorgegebene Funktion streuen. Diese Punkte stellen wir in einem Diagramm dar. Anschließend wählen wir drei Punkte aus, indem wir mit der Maus die entsprechenden Orte anklicken und dort eine rote Markierung setzen. Durch diese drei Punkte zeichnen wir anschließend ein Interpolationspolynom in Rot. Da es sich um ein „mathematisches“ Projekt handelt, ist *Hilberto* dafür zuständig.

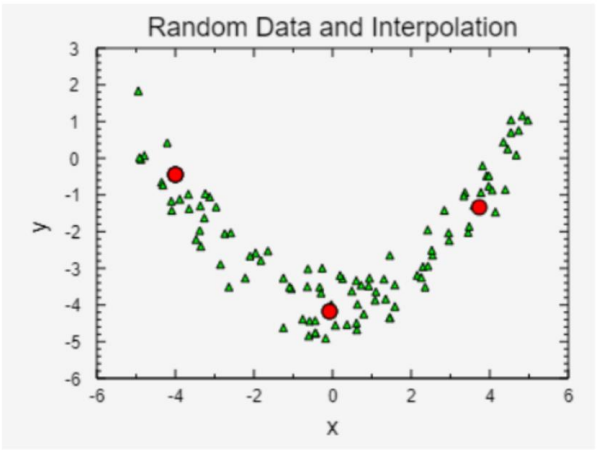
Zuerst einmal die Zufallspunkte:

Danach konfigurieren wir das aktuelle Sprite zum *PlotPad* und zeichnen die Punktmenge.

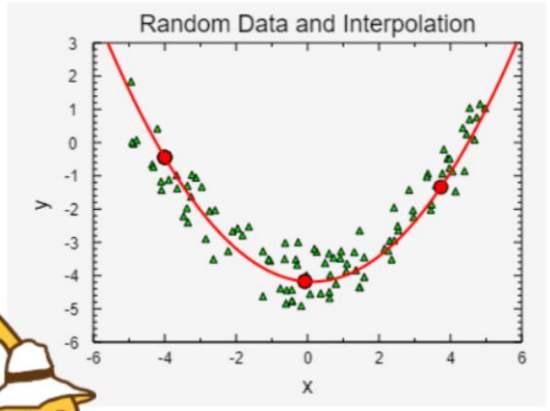




Jetzt wählen wir die drei Punkte und stellen sie gleich im Diagramm dar.



Und zuletzt fügen wir das Interpolationspolynom hinzu.



Aufgaben:

1. a: Erzeugen Sie „Punktwolken“, die um andere ganzrationale Funktionsgraphen streuen.

   b: Legen Sie wie im Beispiel einige Punkte in diesen Wolken fest, durch die ein Interpolationspolynom gezeichnet werden soll.

   c: Lassen Sie diese Polynome zeichnen.

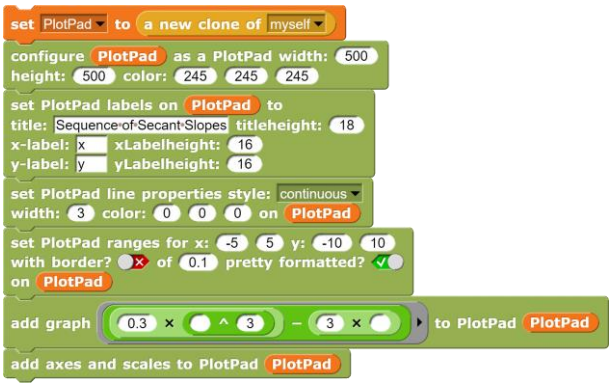
2. a: Experimentieren Sie mit der Anzahl der ausgewählten Punkte. Werden die Ergebnisse besser, wenn Sie mehr Punkte wählen?

   b: Erzeugen Sie „Punktwolken“, die um nicht ganzrationale Funktionsgraphen (trigonometrische, …) streuen. Können Sie auch diese durch Interpolationspolynome beschreiben?

   c: Formulieren Sie eine Regel, wann und wie man Interpolationspolynome sinnvoll einsetzen kann - und weshalb gerade so.

6.8 Approximation einer Tangente durch Sekanten

Wir wollen zeigen, dass eine Folge von Sekantensteigungen gegen die Steigung der Tangente in einem Punkt konvergiert. Dazu konfigurieren wir ein Sprite namens *PlotPad* als *PlotPad* und zeichnen den Graph einer Funktion, hier: $y = 0{,}3 \cdot x^3 - 3 \cdot x$.



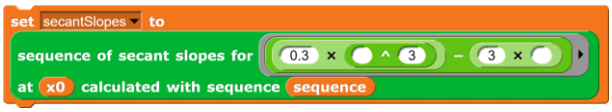
Wir wollen in der Nähe des rechten Minimums eine Folge von Sekanten zeichnen lassen, die sich der Tangente nähern. Dafür benötigen wir natürlich erstmal einen Punkt $(x_0 | y_0)$ in der Nähe des Minimums:

Den können wir auch gleich einzeichnen.

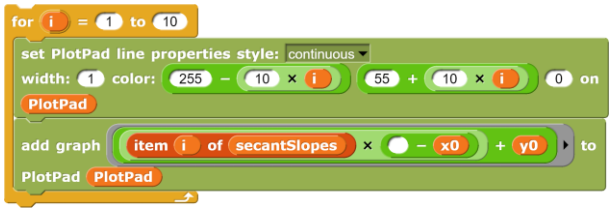
Als Folge zur Annäherung an den Punkt wählen wir $a_n = \frac{2}{n}$

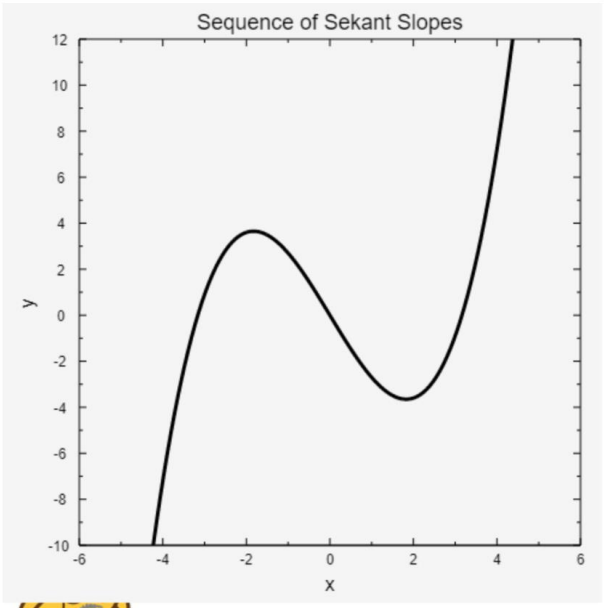
Von der lassen wir die ersten 20 Glieder erzeugen.

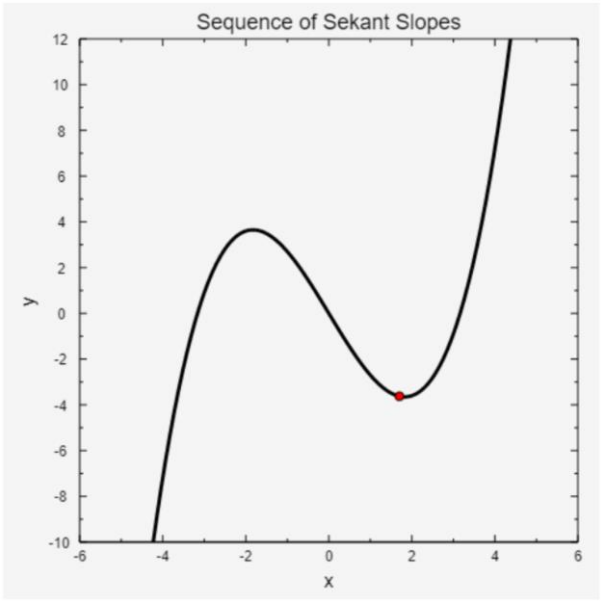
Der Rest ist genauso einfach: wir lassen die Sekantensteigungen berechnen …



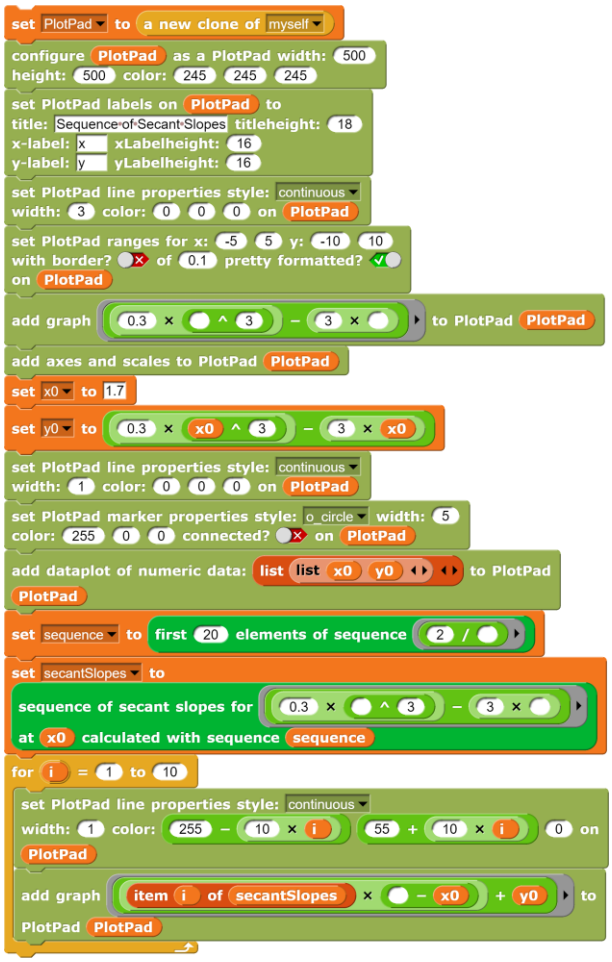
… und die Sekanten in Farbabstufungen zeichnen.

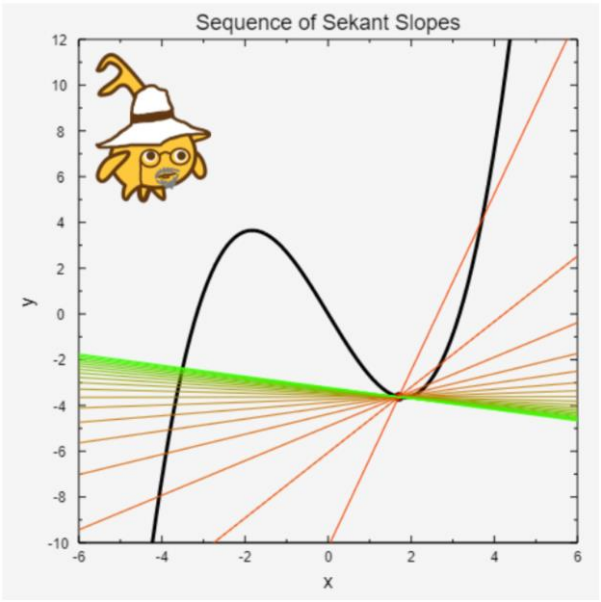






Das Ganze – mit Ergebnis – noch einmal in einem Stück:





Aufgaben:

1. Lassen Sie zusätzlich die „richtige“ Tangente in das Diagramm einzeichnen.
2. Wählen Sie als Folge für die Sekantenberechnung andere Folgen, die sich dem Punkt $(x_0|y_0)$ von der anderen bzw. beiden Seiten nähern.
3. a: Veranschaulichen Sie auf ähnliche Weise die Nullstellenberechnung nach dem Newton-Verfahren.

   b: Wählen Sie einige Fälle, in denen das Verfahren gut funktioniert bzw. kaum noch oder gar nicht.

6.9 Endliche Reihen

Wir wollen einige der üblichen mathematischen Konstanten und Funktionen über Reihenentwicklungen annähern – und auch mal ausprobieren, wie lange man eigentlich rechnen muss, um gute Ergebnisse zu erhalten.

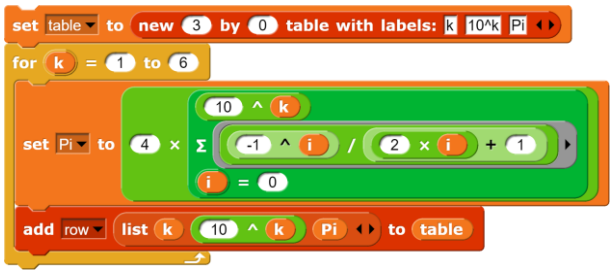
Beginnen wir einmal mit π. In einer Formelsammlung[16] oder bei Wikipedia[17] finden wir eine Formel zur Berechnung von π: die *Leibniz-Reihe*:

$$\frac{\pi}{4} = \sum_{i=0}^{n} \frac{(-1)^i}{(2 \cdot i + 1)}$$

Die können wir direkt in *SciSnap!* umsetzen.

Aber wann ist das Ergebnis eigentlich „gut“?

Zur Beantwortung dieser Frage erstellen wir eine Tabelle.

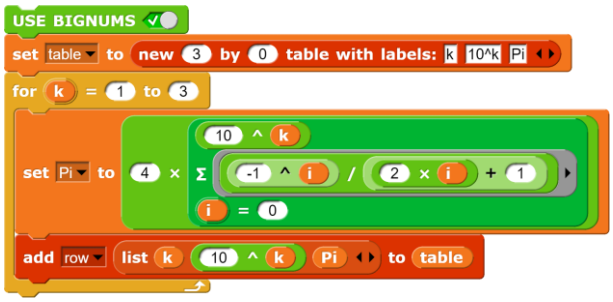


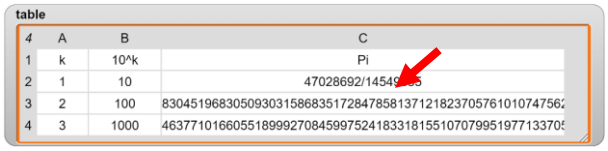
table

7	A	B	C
1	k	10^k	Pi
2	1	10	3.232315809405594
3	2	100	3.1514934010709914
4	3	1000	3.1425916543395442
5	4	10000	3.1416926435905346
6	5	100000	3.1416026534897203
7	6	1000000	3.1415936535887745

Bei einer Million Summanden muss man schon etwas auf das Ergebnis warten! 😉

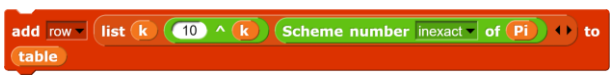
Eigentlich ist es doch seltsam, dass man für so wenig verbesserte Genauigkeit so lange rechnen muss. So etwas kann man vielleicht im Jahre 1673 im verregneten Hannover machen, um nicht spazieren gehen zu müssen – aber jetzt? Wir versuchen es einfach mal mit der *BIGNUM*-Library von *Snap!* für exakte Rechnungen mit *Scheme*-Zahlen. Dann dauert das Rechnen noch länger und wir erhalten erstaunliche Ergebnisse, die nicht nach π aussehen.





table

4	A	B	C
1	k	10^k	Pi
2	1	10	47028692/14549
3	2	100	830451968305093031586835172847858137121823705761010747562
4	3	1000	463771016605518999270845997524183318155107079951977133705



Nach langem Suchen entdecken wir den Bruchstrich in der Mitte, wobei der schon in der zweiten Zeile nicht mehr zu sehen ist. *Scheme*-Zahlen sind nun mal exakte Brüche und keine Gleitkommazahlen.

Wir lassen deshalb die exakten *Scheme*-Zahlen in die inexakte Gleitkomma-Darstellung umwandeln, bevor wir sie in die Tabelle eintragen.

[16] Fragen Sie mal ihren Opa, was das ist. 😉
[17] https://de.wikipedia.org/wiki/Leibniz-Reihe

Trotz des Aufwands sind die Ergebnisse fast die gleichen wie vorher. Das schlechte Konvergenz-Verhalten der Reihe liegt also nicht an den Ungenauigkeiten der Standard-Arithmetik einer Programmiersprache (hier: *JavaScript*), sondern an ihrem Aufbau. Gut zu wissen! 😉

table

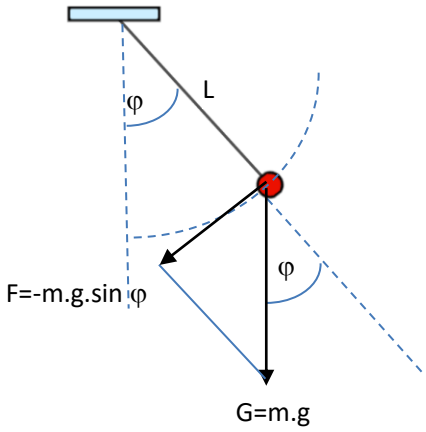
4	A	B	C
1	k	10^k	Pi
2	1	10	3.2323158094055926
3	2	100	3.15149340107099
4	3	1000	3.1425916543393395

Aufgaben:

1. Informieren Sie sich über die Bedeutung der Begriffe „*Scheme-Zahlen*“ und „*Gleitpunktzahlen*“.
2. Suchen Sie sich andere Reihenentwicklungen für π, die besser konvergieren als die Leibniz-Reihe.
3. Suchen und implementieren Sie eine Reihenentwicklung für die Eulersche-Zahl e.
4. Informieren Sie sich über Gründe, mit der Genauigkeit von Scheme-Zahlen zu rechnen, statt der für Gleitpunktzahlen.
5. Schreiben Sie Skripte für die Reihenentwicklung trigonometrischer Funktionen, z. B. *sin(x)*, *cos(x)*, … Beachten Sie, dass dabei der Winkel im Bogenmaß angegeben werden muss.

6.10 Anwendung der Taylor-Reihe beim mathematischen Pendel

Eine sehr anschauliche Anwendung von Reihenentwicklungen ist die Simulation eines mathematischen Pendels, also eines Fadenpendels. Üblicherweise arbeitet man dort mit der Näherung, dass für kleine Winkel der Wert des Sinus (im Bogenmaß) in etwa dem Wert seines Arguments entspricht – d. h. man bricht die Reihenentwicklung der Sinusfunktion nach dem ersten Summanden ab. Wir sehen uns das mal genauer an: Die Kraft F, die die Kugel auf der Kreisbahn beschleunigt, erhalten wir als $F = -G \cdot \sin\varphi = -m \cdot g \cdot \sin\varphi$. Nach der Grundgleichung der Mechanik ist diese Kraft gleich der Trägheitskraft $m \cdot \ddot{s} = m \cdot a$. Benutzen wir die Beziehung für den Winkel im Bogenmaß $\varphi = \frac{s}{L}$, dann erhalten wir

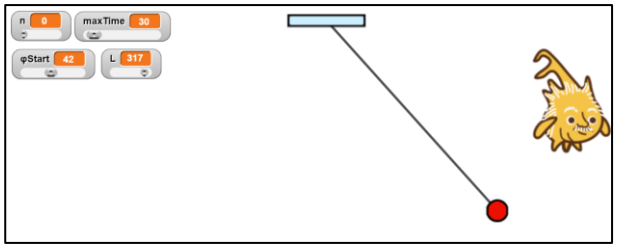


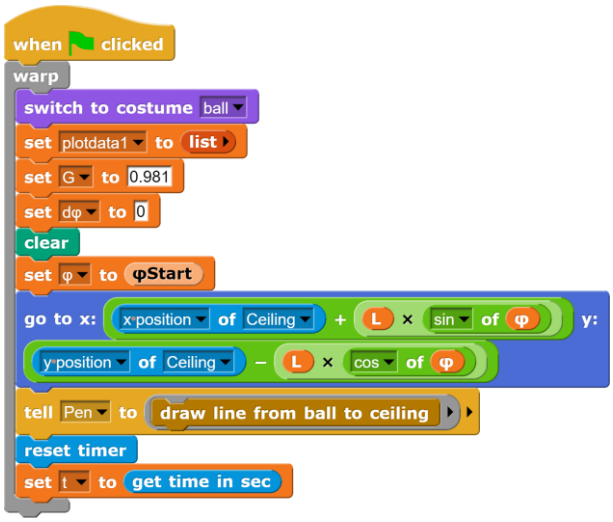
$$\ddot{\varphi} = -\frac{g}{L} \cdot \sin\varphi$$

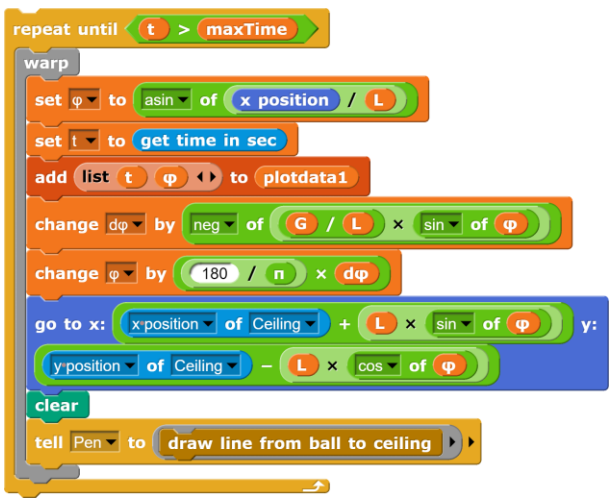
Für kleine Winkel gilt genähert $\sin\varphi \approx \varphi$ und damit

$$\ddot{\varphi} \approx -\frac{g}{L} \cdot \varphi$$

Mal sehen, ob das funktioniert. Wir simulieren zuerst einmal die „echte“ Pendelbewegung, indem wir die Beschleunigung aus der aktuellen Situation entnehmen, daraus die Änderung der Geschwindigkeit bestimmen und daraus wiederum die neue Position. Die Daten nehmen wir in eine Liste *plotdata1* auf. Einige weitere Größen wie die Pendellänge und die Anfangsauslenkung geben wir über Variable in der Slider-Darstellung vor. Zuerst einmal zeichnen wir die Anfangssituation: Ein Stift zeichnet erforderliche Linien, und das Pendel hängt natürlich an der Decke des Labors.

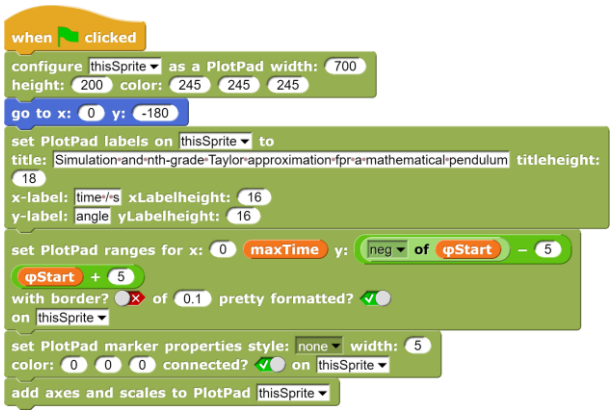


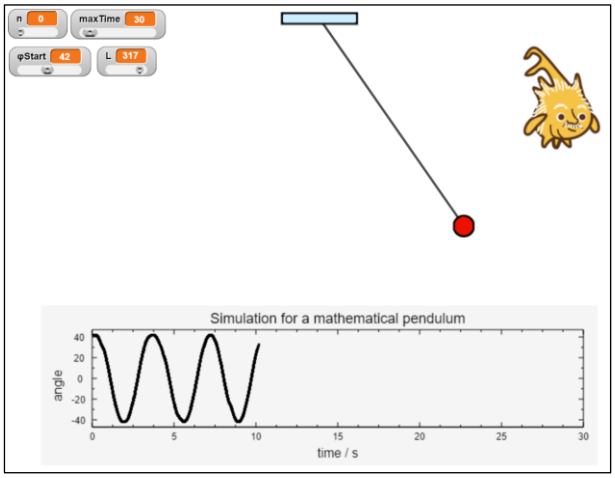




Jetzt geht es los: Wir messen die aktuelle Auslenkung und die Zeit und schreiben beide Werte in die Liste der Messwerte. Danach berechnen wir die aktuelle Beschleunigung, die die Winkelgeschwindigkeit ändert, und daraus den neuen Winkel. Dann lassen wir die neue Situation zeichnen. Und das immer wieder.

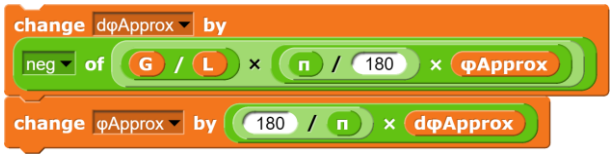
Dieser Anordnung spedieren wir ein weiteres Sprite als PlotPad: den *Plotter*. Der ist schnell genug, die aktuellen Daten in Echtzeit darzustellen. Deshalb fügen wir den Block dafür in die Simulationsschleife des Pendels ein.





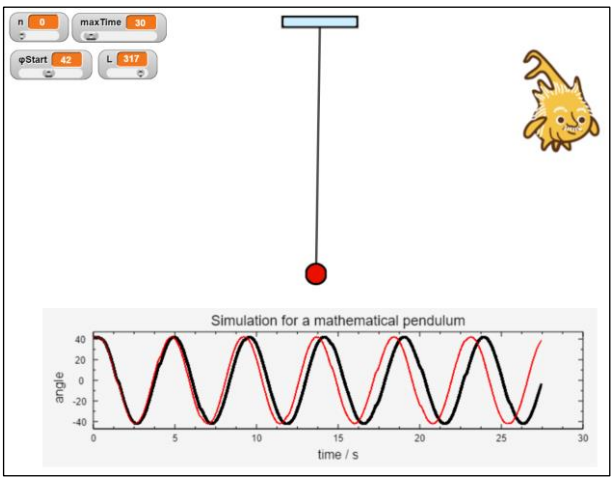
Alberto ist sichtbar begeistert, dass das so gut klappt!

Das eigentliche Ziel war es aber, sich mal anzusehen, wie gut die Näherung ist. Dafür führen wir eine zweite Datenliste *plotdata2* ein sowie einen „genäherten“ Winkel $\varphi Approx$ und die entsprechende Winkelgeschwindigkeit. Der „echte“ Winkel φ und der genäherte starten mit dem gleichen Wert. Danach wird jeweils die „echte“ Auslenkung des Pendels gemessen und der genäherte Wert berechnet.



Beides zeichnen wir in das Diagramm ein – den berechneten Wert in Rot.

Man sieht, dass die Näherung anfangs ganz gut ist, aber mit zunehmender Zeit mit den Echtzeitwerten auseinander läuft.



Das kann man besser machen!

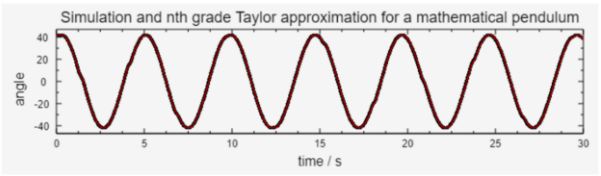
Statt der linearen Näherung wählen wir die Taylorreihe des Sinus, von der wir *n* Summanden als Näherung verwenden. *n* geben wir als Slider-Variable vor.

$$\sin\varphi = \sum_{i=0}^{n} (-1)^i \cdot \frac{\varphi^{2i+1}}{(2i+1)!} = \varphi - \frac{\varphi^3}{3!} + \frac{\varphi^5}{5!} - \cdots$$

Das können wir direkt in *SciSnap!* abschreiben:

Zusammen mit der Umrechnung von Winkeln ins Bogenmaß erhalten wir für die Winkelbeschleunigung:

Schon bei einer Erweiterung der Näherung um ein Reihenglied (also $-\frac{\varphi^3}{3!}$) sind die Abweichung von den Messwerten im Diagramm nicht mehr sichtbar.



Aufgaben:

1. Lassen Sie die Simulation länger laufen und stellen Sie fest, wann sich – in Abhängigkeit von der Zahl der Summanden der Taylorreihe – Abweichungen zeigen.
2. Machen Sie das Gleiche für unterschiedliche Startwinkel des Pendels.

6.11 Fourier-Entwicklung für ein Rechtecksignal mit numerischer Integration

Eine Möglichkeit zur Fourier-Darstellung einer 2π-periodischen Funktion $f(x)$ ist

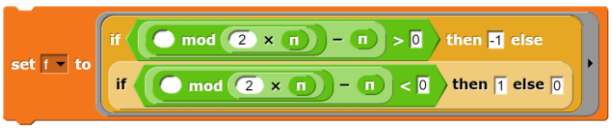
$$f(x) \cong a_0 + \sum_{k=1}^{\infty}(a_k \cdot \cos(k \cdot x) + b_k \cdot \sin(k \cdot x)) \qquad a_0 = \frac{1}{2\pi} \cdot \int_0^{2\pi} f(x) \cdot dx$$

$$a_k = \frac{1}{\pi} \cdot \int_0^{2\pi} f(x) \cdot \cos(k \cdot x) \cdot dx \qquad b_k = \frac{1}{\pi} \cdot \int_0^{2\pi} f(x) \cdot \sin(k \cdot x) \cdot dx\,;\ k = 1,2,3,\ldots$$

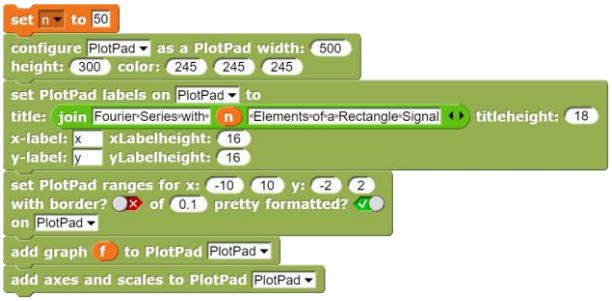
Wir wollen die Koeffizienten für die Reihenentwicklung eines Rechtecksignals rein numerisch ermitteln und dabei testen, wie sich die Länge der Reihenentwicklung auf die Genauigkeit der Ergebnisse auswirkt.

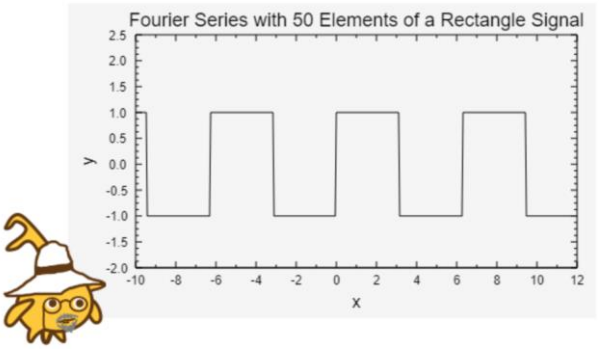
Zuerst einmal benötigen wir ein Rechtecksignal mit der Periode 2π. Eine Funktion, die das liefert, wäre z. B.

$$f(x) = \begin{cases} -1 & falls\ (x\ mod\ 2\pi - \pi) > 0 \\ 1 & falls\ (x\ mod\ 2\pi - \pi) < 0 \\ 0 & sonst \end{cases}$$



Das sehen wir uns natürlich erst einmal im Diagramm an, bevor wir glauben, dass es sich um ein Rechtecksignal handelt:

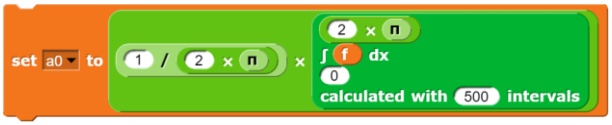




Stimmt also. 😉

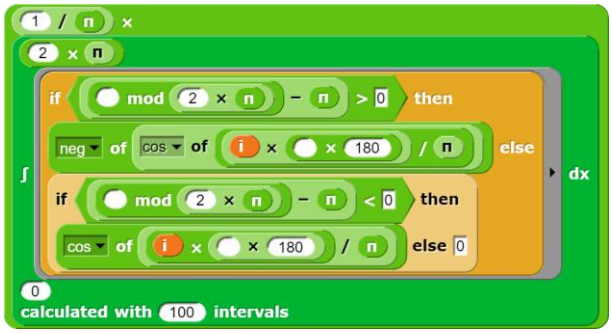
Jetzt müssen wir für jeden Wert von *x* die ersten *n* Koeffizienten der Fourier-Reihe berechnen. Dafür benutzen wir den Block für die numerische Integration.

Probieren wir es also mal für $a_0 = \frac{1}{2\pi} \cdot \int_0^{2\pi} f(x) \cdot dx$. Da wir den „ringified term“ für die Funktion *f* schon haben, können wir einfach abschreiben:

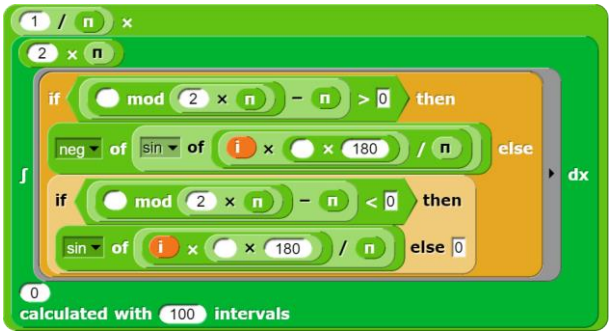


Bei den anderen Koeffizienten müssen wir die trigonometrischen Terme ergänzen und erhalten für

a_k:

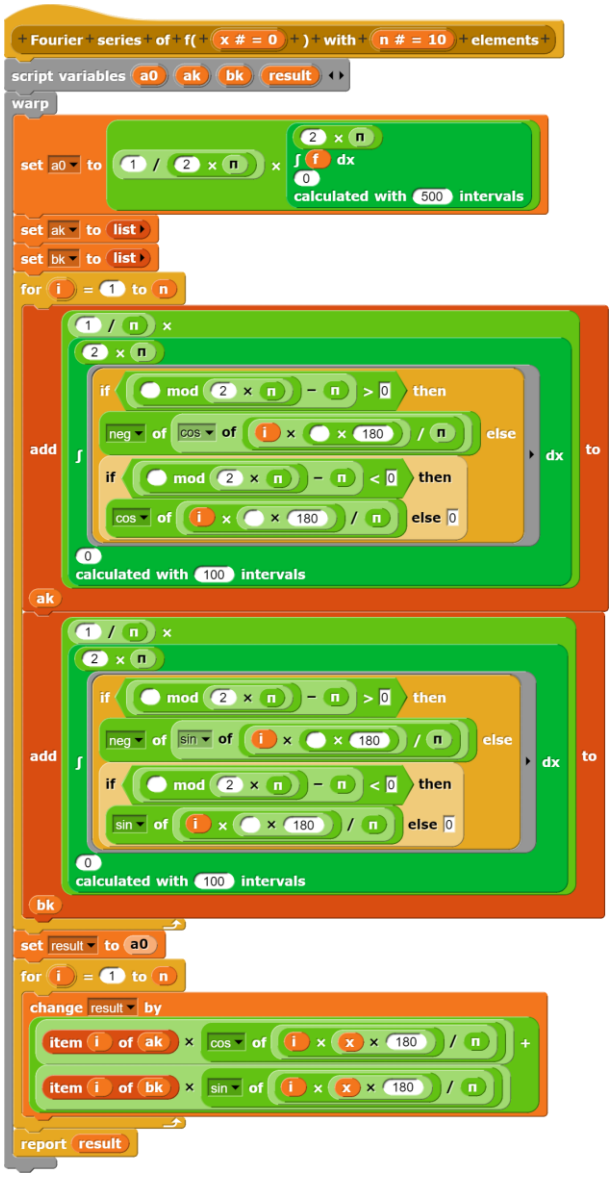


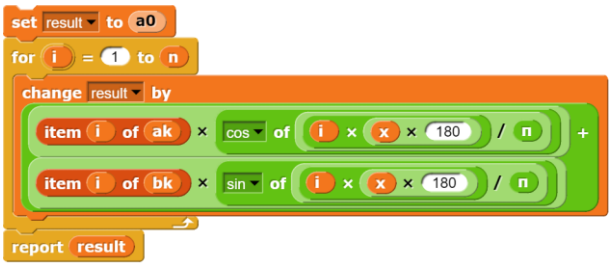
und b_k:



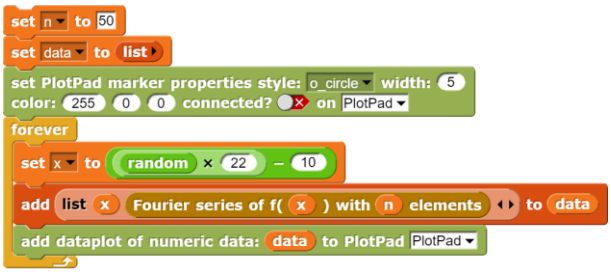
Diese Terme müssen jetzt nur noch summiert werden:

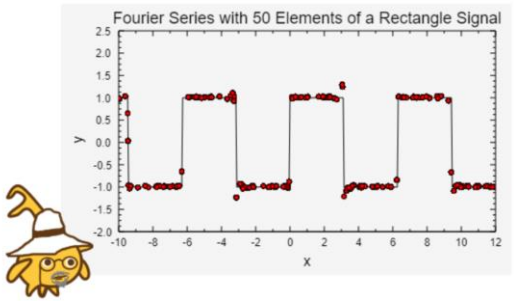
Damit erhalten wird insgesamt das folgende Skript.

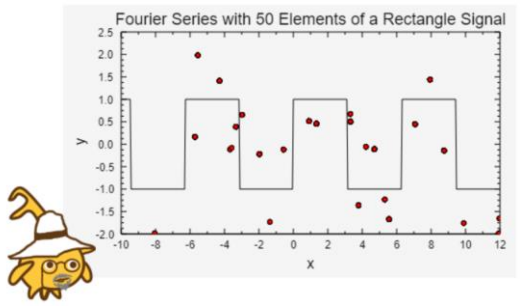
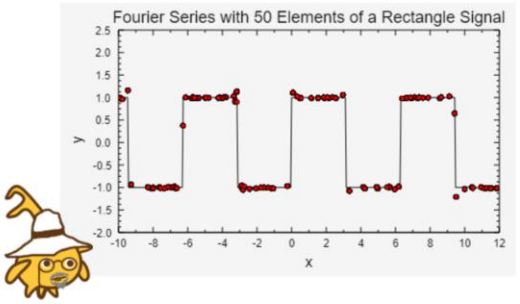
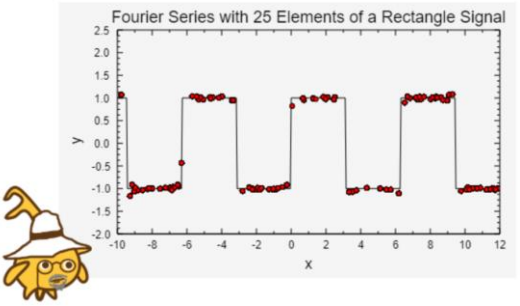
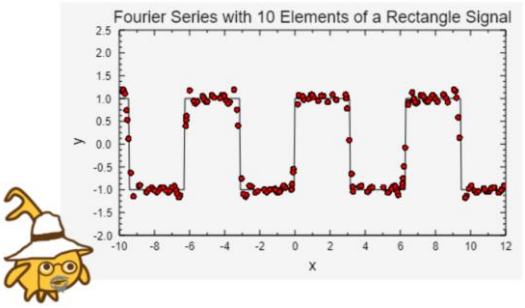
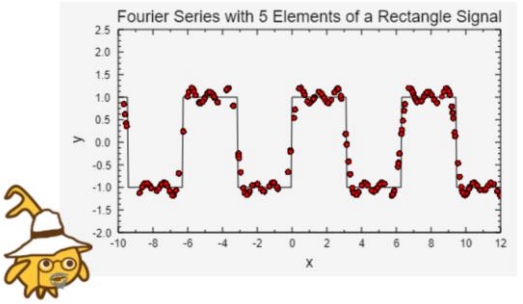




Wie haben darin zwei „Schrauben“, an denen wir drehen können: einerseits kann die Zahl *n* der Terme der Fourier-Reihe geändert werden, andererseits die Anzahl *i* der Intervalle bei der numerischen Integration. Die Auswirkungen von Änderungen lassen sich leicht beurteilen, wenn wir zufällig Werte aus dem Wertebereich von *x* ziehen und das Ergebnis zusammen mit der Funktion *f* plotten lassen. Abweichungen zeigen sich dann sofort.



n Summanden	i Intervalle	Ergebnis	Kommentar
50	100		Eigentlich ganz gut bis auf „Ausrutscher“, die an den Sprungstellen der Funktion liegen.

50	50		Das dürfte eindeutig sein.
50	200		Besser, aber nur wenig besser als mit 100 Summanden.
25	200		Schlechter, aber nur unwesentlich schlechter als mit 50 Summanden.
10	100		Vielleicht ein ganz guter Kompromiss zwischen Genauigkeit und Rechenzeitbedarf.
5	75		In vielen Fällen wohl auch noch vertretbar.

Man sieht, dass es wohl von der Aufgabenstellung abhängt, welche Genauigkeit benötigt wird. Sind z. B. Ausrutscher an den Sprungstellen vertretbar oder kommt es gerade da auf Präzision an? Man sieht auch, dass man mit gleichem Aufwand sehr unterschiedliche Wirkungen erzielen kann.

Aufgaben:

1. a: Geben Sie Funktionsterme für ein Dreieckssignal, ein Signal aus Peaks, … an. Lassen Sie die Funktionsgraphen zeichnen.

   b: Berechnen Sie die Fourierreihen für die Signale.

   c: Experimentieren Sie wie gezeigt, um einen vertretbaren Kompromiss zwischen Genauigkeit und Rechenzeitbedarf zu finden.

2. a: Erzeugen Sie Tabellen mit den Daten für ein Dreieckssignal, ein Signal aus Peaks, … mithilfe der Funktionsterme.

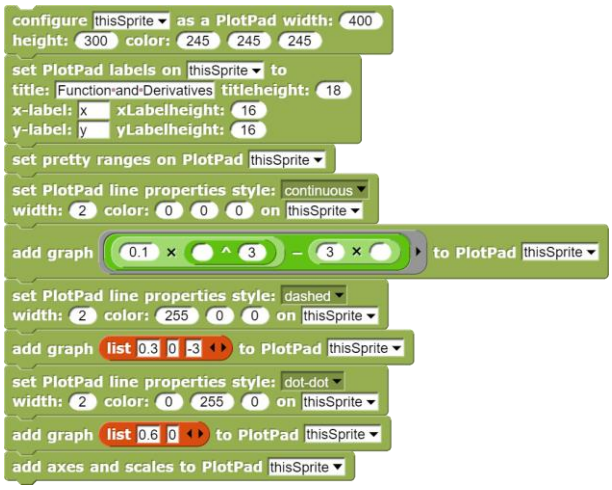
   b: Geben Sie die Signale über den Lautsprecher aus.

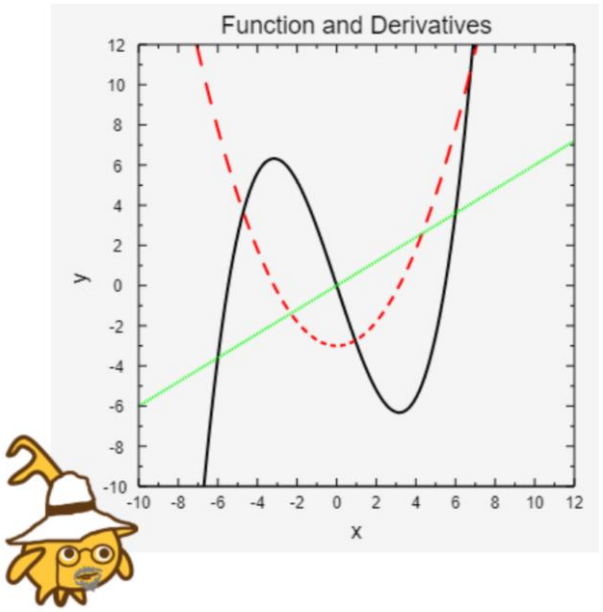
   c: Erzeugen Sie die entsprechenden Tabellen mithilfe unterschiedlich guter Fourier-Reihen und hören Sie sich auch diese als Töne an. Nehmen Sie Unterschiede wahr?

3. Informieren Sie sich über Einsatzgebiete von Fourier-Reihen.

6.12 Zeichnen einer Funktion und ihrer Ableitungen

Wir wollen ein *SciSnap!PlotPad* dazu benutzen, eine Funktion und ihre ersten beiden Ableitungen in unterschiedlichen Farben und Linienarten darzustellen. Dazu erzeugen wir ein neues Sprite, wechseln in seinen Skriptbereich und konfigurieren es geeignet. Die Funktionsterme werden einmal als „ringified“ Operator, dann zweimal als Liste von Polynom-Koeffizienten angegeben.





Aufgaben:

1. Stellen Sie unterschiedliche Funktionstypen (trigonometrische Funktionen, Logarithmen, Polynome, …) als Graph auf einem *PlotPad* dar.
2. Ergänzen Sie die Funktionsgraphen durch deren Ableitungen.
3. Wählen Sie für die Darstellungen unterschiedliche Wertebereiche, Genauigkeiten der Zahlendarstellung und Textgrößen und Beschriftungen.

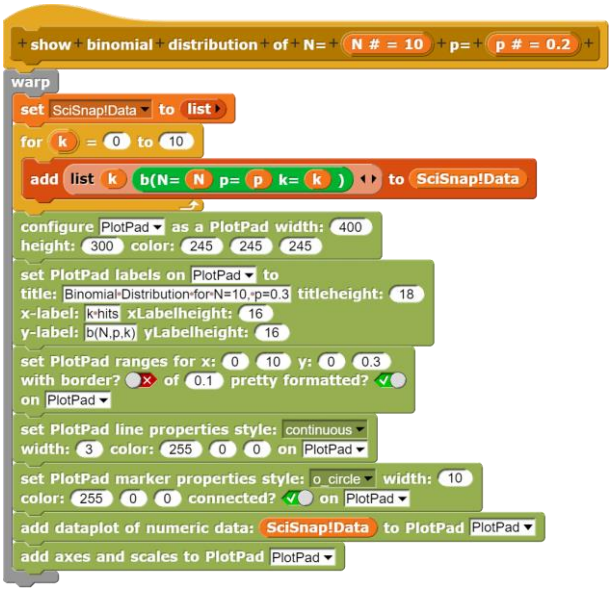
6.13 Zufallsexperimente zur Binomialverteilung

Da hier richtig mathematisch gearbeitet wird, bitten wir *Gundolf de Jong*, den begabten jungen Mathematiker, um seine Mithilfe. Er soll einerseits Zufallsexperimente durchführen, andererseits die Experimente auswerten und mit der entsprechenden Binomialverteilung vergleichen. Dazu erzeugt er natürlich ein Diagramm.

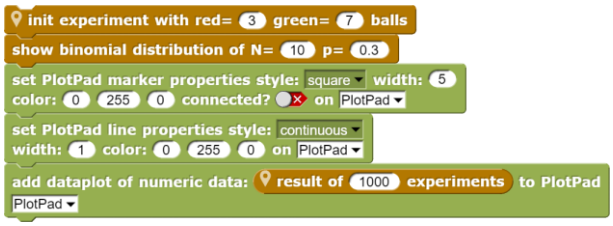
Für die Experimente benötigt *Gundolf* rote und grüne Bälle, die nummeriert werden können. Die liefert ihm ein Sprite namens *Ball*, das man um entsprechende Exemplare bitten kann.

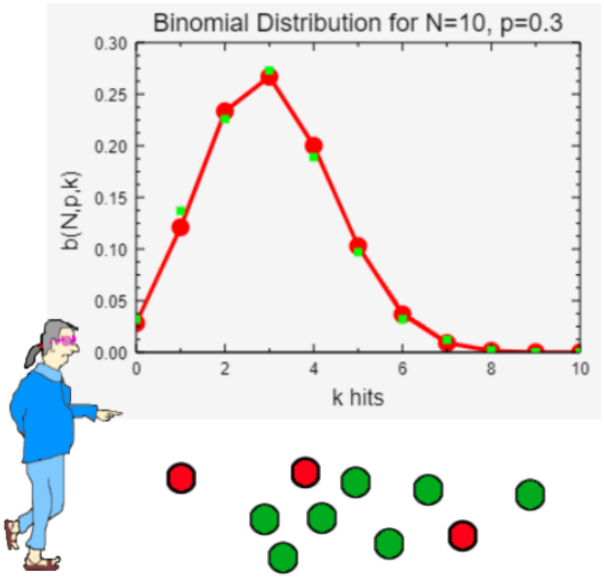
Nach der Ausführung von 

steht Gundolf noch ziemlich alleine mit seinen Bällen im Raum. Er beginnt deshalb zu experimentieren: von seinen 10 Bällen zieht er zufällig einen und überprüft seine Farbe. Ist die Rot, dann zählt er den Ball und legt ihn zurück. Das macht er jeweils 10-mal und nennt das Ganze ein Experiment. Diese Experimente führt er n-mal aus und teilt die Anzahlen der jeweils gezogenen roten Bälle durch die Gesamtzahl der Experimente. Das Ergebnis gibt er zurück.



Danach kann er ein Diagramm auf einem Sprite *PlotPad* zeichnen, das die entsprechende Binomialverteilung in Rot zeigt. Das Ergebnis von n=1000 Experimenten zeichnet er dann in Grün darüber.





Aufgaben:

1. Ändern Sie die Spielregeln: Legen Sie die gezogenen Bälle nicht wieder zurück, ändern Sie die Zahl der Bälle insgesamt, pro Farbe oder bezogen auf die Zahl der gezogenen Bälle. Definieren Sie, was als „Erfolg“ gilt.
2. Tragen Sie die Ergebnisse in Diagramme ein. Informieren Sie sich über die zu erwartende Verteilung und tragen Sie auch diese ein. Diskutieren Sie ggf. Abweichungen.

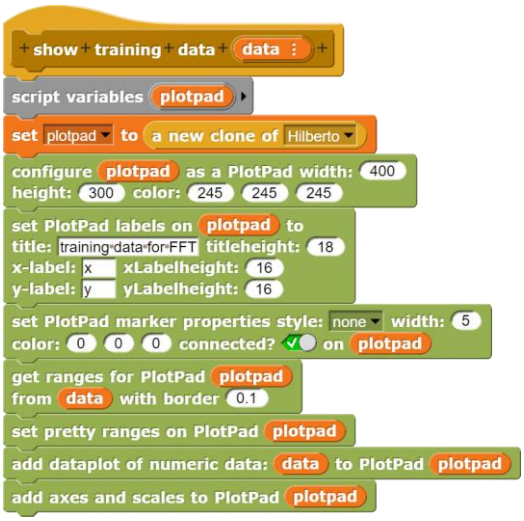
6.14 Schnelle Fourier Transformation (FFT)

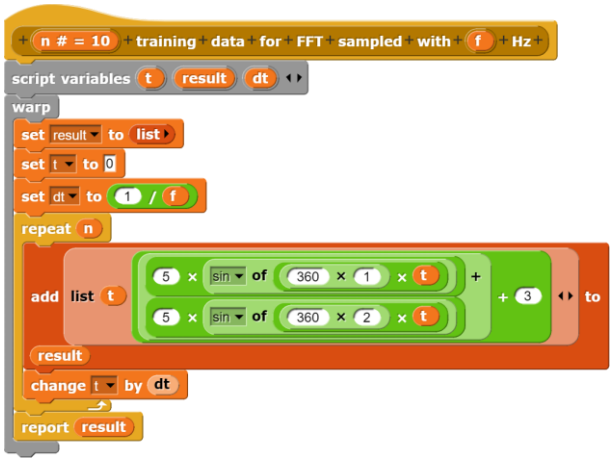
Die *FFT* (Fast Fourier Transform) ist ein effizienter Algorithmus, um aus einem Vektor dessen diskrete Fourier Transformierte (*DFT*) zu berechnen. Dabei wird den Daten ein diskretes periodisches Frequenzspektrum zugeordnet. Es handelt sich um eines der Standardverfahren z. B. bei naturwissenschaftlich-technischen Anwendungen. Informieren Sie sich besser an anderer Stelle über die Einzelheiten, bevor Sie den entsprechenden *SciSnap!*-Block einsetzen. An dieser Stelle wollen wir das Vorgehen durch ein *Snap!*-Skript beschreiben und danach an einigen Beispielen die Möglichkeiten des *SciSnap!*-Blocks demonstrieren.

Gegeben sei der Verlauf einer Funktion durch eine Folge von Funktionswerten. Wir können uns deren Zustandekommen vorstellen als ein Ablesen des Funktionswertes in regelmäßigen Abständen, z. B. bei einem zeitabhängigen Signal in bestimmten zeitlichen Abständen. Man bezeichnet so etwas als *Sampling*. In die Berechnung der FFT gehen aber nur die Funktionswerte, nicht die „Stützstellen“ ein. Dieser Wertevektor wird nun wiederholt durch Aufteilung auf gerade und ungerade Indizes halbiert, und zwar solange, bis nur einzelne Funktionswerte übrig sind. Aus diesen wird dann die DFT durch Neukombinationen erzeugt. Da diese Aufteilung nur für Vektorlängen von 2^N möglich ist, muss ein Vektor anderer Länge vor der Verarbeitung auf diese Länge durch Anfügen z. B. von Nullen ergänzt werden. Der Algorithmus arbeitet dann wie folgt[18]:

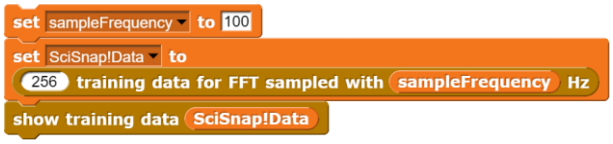
```
FFT(vektor) //Index beginnt mit 0
  n = Länge von vektor
  Falls n<=1 ergebnis = vektor als komplexe Zahl
  sonst teile vektor in zwei vektoren gerade und ungerade mit den geraden bzw.
        ungeraden Indizes auf und wende auf diese FFT an:
        gerade    = FFT(Werte mit geradem Index)
        ungerade  = FFT(Werte mit ungeradem Index)
        für m von 0 bis n/2-1 tue
            ergebnis_m     = gerade_m+ungerade_m*e^(-2πim/n
            ergebnis_n/2+m = gerade_m-ungerade_m*e^(-2πim/n
  gib ergebnis zurück
```

Zur Darstellung der Vorgänge benötigen wir erst einmal Trainingsdaten, an denen wir die FFT testen können. Der Einfachheit halber überlagern wir zwei Sinusfunktionen und addieren den Wert 3. Das Ergebnis sehen wir uns einmal an.

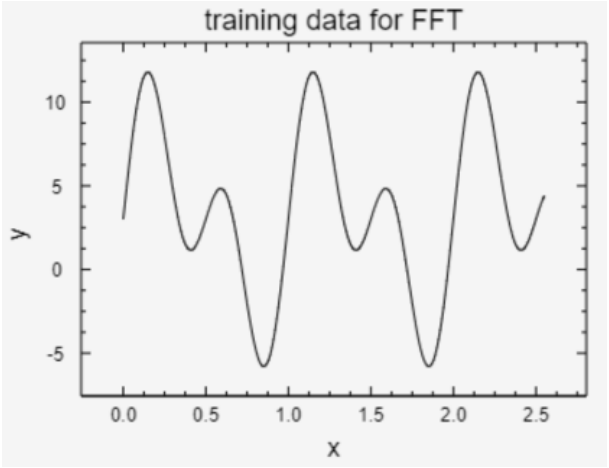




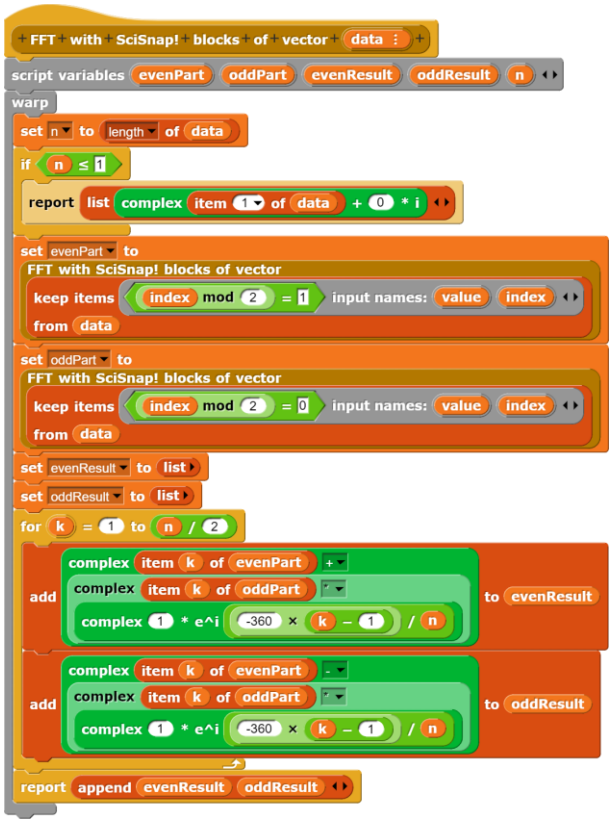
Wir setzen die Samplefrequenz auf 100 Hz und bestimmen die Trainingsdaten.



[18] Quelle: Wikipedia https://de.wikipedia.org/wiki/Schnelle_Fourier-Transformation



So sieht die Funktion also aus. Auf die zweite Spalte der Trainingsdaten – die abgetasteten Funktionswerte – wenden wir den FFT-Algorithmus wie beschrieben an. Wir können ihn einfach abschreiben.



Hat der übergebene Vektor die Länge Eins, wird er als komplexe Zahl im SciSnap!-Format zurückgegeben.

Die Daten werden in zwei Teile mit geraden bzw. ungeraden Indizes aufgeteilt, auf die jeweils die FFT rekursiv angewandt wird.

Die beiden Hälften des Ergebnisses werden wie beschrieben zusammengesetzt …

… und aneinandergeheftet zurückgegeben.

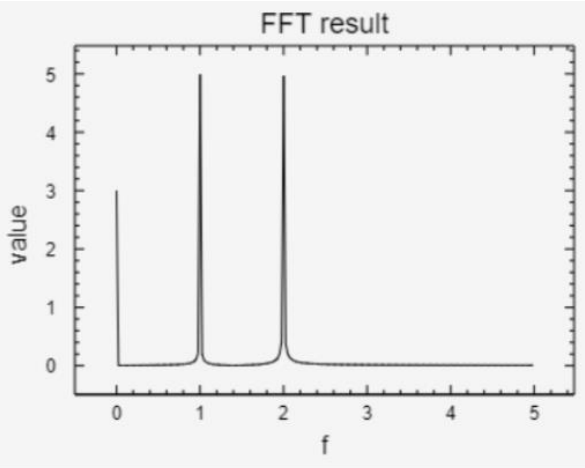
Das Ergebnis sehen wir uns für (hier: nur) 8 Funktionswerte an.

8	A	B	C
1	complexNumberCartesianStyle	48.96931370990699	0
2	complexNumberCartesianStyle	-3.8922516608711244	8.164322519753798
3	complexNumberCartesianStyle	-3.4812560903797785	3.334362009251132
4	complexNumberCartesianStyle	-3.4121839517945776	1.377837140911721
5	complexNumberCartesianStyle	-3.3979303038160324	0
6	complexNumberCartesianStyle	-3.4121839517945767	-1.377837140911721
7	complexNumberCartesianStyle	-3.4812560903797785	-3.334362009251132
8	complexNumberCartesianStyle	-3.8922516608711235	-8.164322519753798

Aber was sollen wir damit nur anfangen?

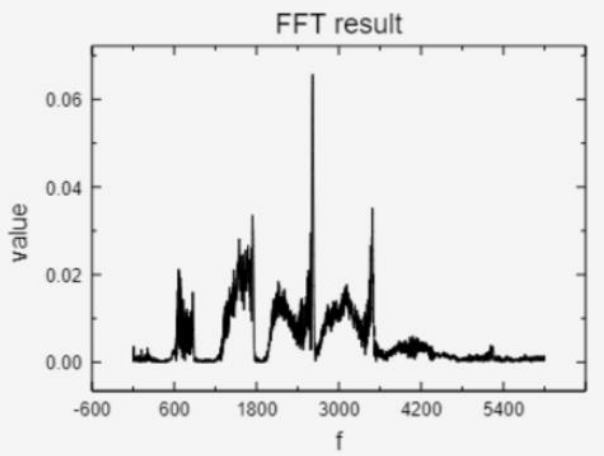
Für ein Frequenzspektrum interessieren nicht die gesamten komplexen Zahlen, sondern (meist) nur deren Beträge. Die erhalten wir z. B., indem wir die Zahlen in die Polarform überführen und dann den imaginären Teil abspalten – und die erste Spalte mit dem Typ gleich mit. Diese Werte skalieren wir so, dass sich die Amplituden der Teilfunktionen richtig ergeben, und vor diesen Werten fügen wir die zugehörigen Frequenzen ein.

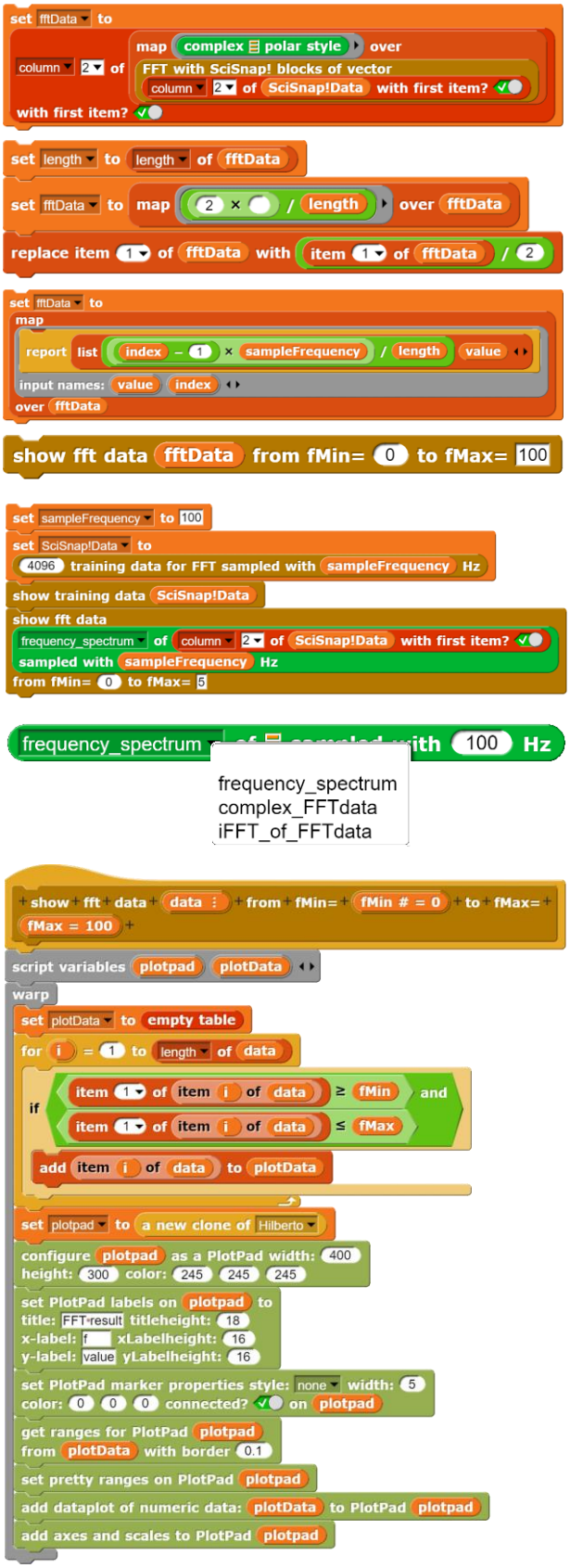
Mit dem Ergebnis können wir dann das Frequenzdiagramm zeichnen lassen. Samplen wir 4096 Werte mit 100 Hz und lassen das Verfahren wie beschrieben ablaufen, dann erhalten wir ein schönes Spektrum.



Das absolute Glied mit dem Wert 3 sowie die Sinusfunktionen von 1 Hz und 2 Hz mit den Amplituden 5 sind klar zu erkennen.

Fouriertransformationen muss man auch umkehren können (*iFFT*). Dafür sind die imaginären Anteile, die Phasen, dann doch erforderlich. Der *SciSnap!*-Block für FFTs enthält deshalb die drei entsprechenden Optionen. Mit dessen Hilfe erhalten wir das gesuchte Spektrum sehr viel einfacher. Und auch das Miauen aus den *Snap!*-Sounds („cat“) lässt sich schnell analysieren.

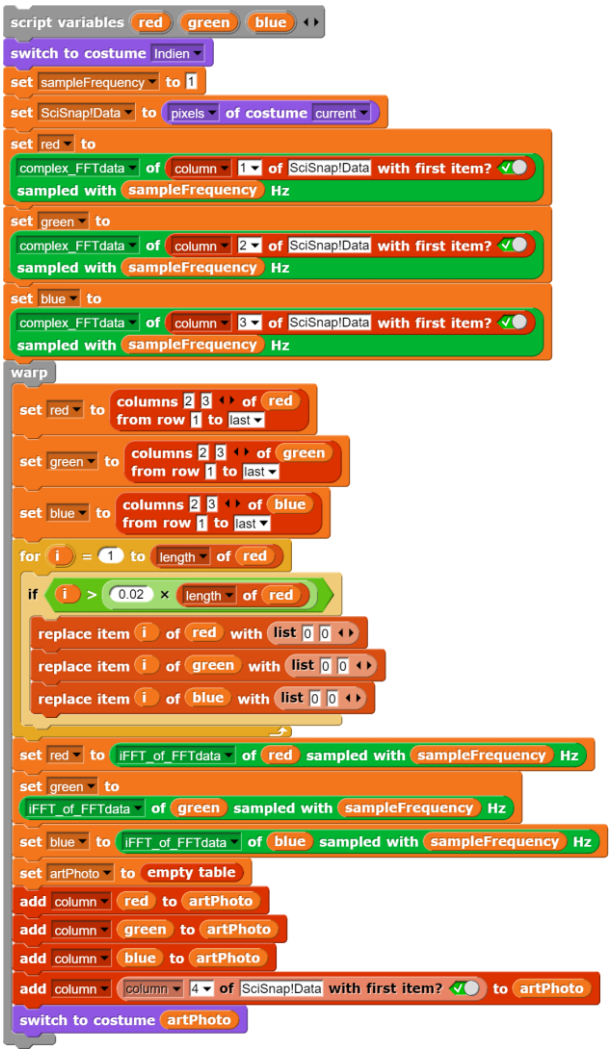




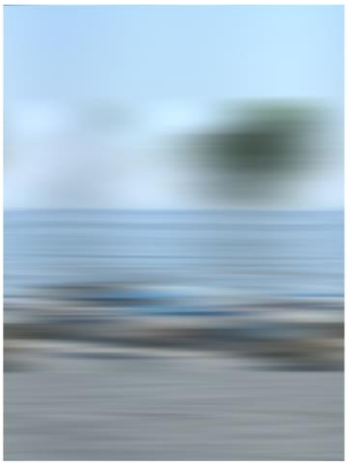
FFTs werden nicht nur bei Tönen, sondern auch zur Bildverarbeitung eingesetzt. Wir fassen dazu die drei Farbkanäle eines Bildes als Vektoren auf, wenden jeweils die FFT-Operation an und schneiden z. B. die höheren Frequenzen ab. Das Ergebnis wird dann aus drei Farbkanälen zusammengesetzt, die durch iFFT aus den veränderten Frequenzwerten entstehen. Die Ergebnisse sind durchaus „modern“. 😉

Lässt man nur 5% der niedrigen Frequenzen übrig, dann kann man noch erstaunlich viel erkennen. Die höheren Frequenzen sind also für die Details zuständig.

Bei 2% wird es schon schwieriger: ein „Osterbild“.



Und bei 1%? Raten Sie mal! 😉



<u>Aufgaben</u>:

1. Erzeugen Sie aus Sinus-/Cosinusfunktionen unterschiedliche Funktionsverläufe. Experimentieren Sie dann mit unterschiedlichen Sampleraten und Anzahlen der Abtastpunkte, um Frequenzspektren zu erzeugen. Vergleichen Sie jeweils ihre Ergebnisse mit dem zu erwartenden Resultat – das Sie ja kennen.
2. Erzeugen Sie Spektren unterschiedlicher Klänge, z. B. von Instrumenten, Stimmgabeln, Sprache, Gesang, …

6.15 Ein einfaches Bild-Kompressionsverfahren mit FFT

Im letzten Abschnitt haben wir gesehen, dass mit FFT bearbeitete Bilder eigentlich noch ganz gut erkennbar sind, obwohl ein großer Teil der höheren Frequenzen „abgeschnitten“ wurde. Wir können das ausnutzen, um den Speicherbedarf von Bildern zu reduzieren, ohne allzu viel von der Bildqualität zu verlieren. Es handelt sich aber trotzdem um ein verlustbehaftetes Kompressionsverfahren. Teile der JPEG-Kompression[19] arbeiten ähnlich.

Unser Verfahren soll wie folgt arbeiten: Wir geben eine Kompressionsrate *compressionRate* und das Kostüm eines Sprites als Parameter *image* vor. Danach bestimmen wir die Pixel des Bildes als *imageData*. Jetzt können wir ausrechnen, wie viele Frequenzen bei der gewünschten Kompression „valide“ sein sollen. Das Ergebnis nennen wir *numberOfValidFrequencies*. Die restlichen *numberOfEmptyFrequencies* werden später auf Null gesetzt und zum „Auffüllen“ des FFT-Ergebnisses auf die richtige Länge benutzt. Um den Kontrastbereich des Originalbildes halbwegs zu erhalten, bestimmen wir zusätzlich die Wertebereiche der drei Farbkanäle *rangesRed, rangesGreen* und *rangesBlue*. Zusammen mit Breite und Höhe des Bildes speichern wir diese Daten, aus denen später das Bild rekonstruiert wird, in einer Liste ab. Diese ergänzen wir um die FFTs der drei Farben, von denen wir aber nur die „valide“ Anzahl speichern. Das komprimierte Bild hat damit das Format:

width	height
numberOfValidFrequencies	numberOfEmptyFrequencies
min red	max red
min green	max green
min blue	max blue
data of the three color channels as complex numbers	

Diese Daten können wir dann z. B. einer Variablen zuweisen und deren Inhalt exportieren, oder wir können direkt in eine Datei schreiben.

Bei der Dekomprimierung des Bildes müssen wir die gespeicherten Daten auslesen und die drei Farbkanäle mit der komplexen Zahl Null auffüllen. Danach werden sie mit *iFFT* zurück in reelle Zahlenreihen verwandelt und zu Pixeln zusammengesetzt. Wir leisten uns zusätzlich etwas Luxus: da wir die ursprünglichen Wertebereiche der Farbkanäle gespeichert haben, strecken wir die durch die reduzierte Anzahl der Frequenzen in ihrem Wertebereich ebenfalls reduzierten Farben wieder auf den ursprünglichen Bereich.

[19] https://de.wikipedia.org/wiki/JPEG

Bildmaße auslesen und löschen

Zahl der gültigen bzw. gelöschten Frequenzen lesen und löschen

ursprüngliche Farbbereiche auslesen und löschen

eine Liste mit komplexen Nullen der richtigen Länge anlegen

iFFT für den Rotkanal

aktuelle Grenzen des roten Farbkanals bestimmen

erweitern des Rotkanals auf die ursprünglichen Werte und einfügen in die Pixelliste

iFFT für den Grünkanal

aktuelle Grenzen des grünen Farbkanals bestimmen

erweitern des Grünkanals auf die ursprünglichen Werte und einfügen in die Pixelliste

iFFT für den Blaukanal

aktuelle Grenzen des blauen Farbkanals bestimmen

erweitern des Blaukanals auf die ursprünglichen Werte und einfügen in die Pixelliste

einfügen der Transparenzwerte und Export des Kostüms

Zum Test verpassen wir *Hilberto* ein Bild als Kostüm und komprimieren es zunehmend.

Geht doch! 😉

Aufgaben:

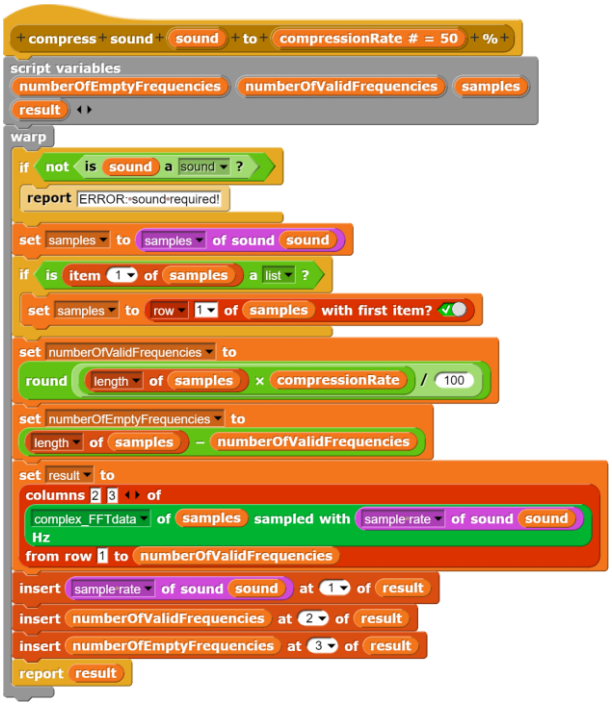
1. Experimentieren Sie mit unterschiedlichen Bildtypen (Portrait, Landschaftsaufnahme, …) und Kompressionsraten.
2. Informieren Sie sich über Kompressionsverfahren für Bilder und andere Daten.
3. Erfinden Sie bessere Kompressionsverfahren für Bilder, z. B. indem Sie nicht einfach Frequenzbereiche abschneiden, sondern indem Sie sich auf die „wesentlichen“ Frequenzen beschränken, …

6.16 Ein einfaches Ton-Kompressionsverfahren mit FFT

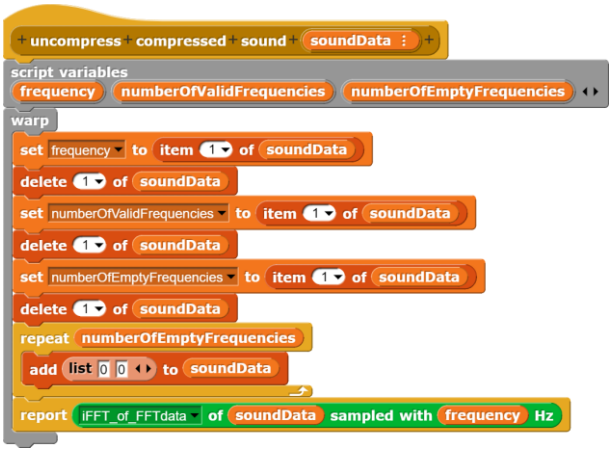
Da alle Daten in digitaler Form vorliegen müssen, ist es eigentlich egal, welche Art von Daten mit der *FFT* verarbeitet werden. Wir können also genauso gut Sounddaten komprimieren. Es ist sogar noch einfacher, wenn wir uns auf einen Kanal beschränken. Es handelt sich wieder um ein verlustbehaftetes Kompressionsverfahren. Teile der MP3-Kompression[20] arbeiten ähnlich.

Unser Verfahren soll wie folgt arbeiten: Wir geben eine Kompressionsrate *compressionRate* und einen Sound als Parameter *sound* vor. Danach bestimmen wir die Samples des Bildes als *samples*. Sollte es sich um einen Sound mit mehreren Kanälen handeln, dann nehmen wir von diesen nur den ersten. Jetzt können wir ausrechnen, wie viele Frequenzen bei der gewünschten Kompression „valide“ sein sollen. Das Ergebnis nennen wir *numberOfValidFrequencies*. Die restlichen *numberOfEmptyFrequencies* werden später auf Null gesetzt und zum „Auffüllen“ des FFT-Ergebnisses auf die richtige Länge benutzt. Der komprimierte Sound hat damit das Format:

sample rate
numberOfValidFrequencies
numberOfEmptyFrequencies
sound data as complex numbers



Bei der Dekomprimierung des Sounds müssen wir die gespeicherten Daten auslesen und die Sounddaten mit der komplexen Zahl Null auffüllen. Danach werden sie mit *iFFT* zurück in reelle Zahlenreihen verwandelt und als rekonstruierter Sound zurückgegeben.



Bis zu einer Kompression auf etwa 15% höre ich jedenfalls kaum Unterschiede. Aber vielleicht sind Sie ja jünger. 😉

Aufgaben:

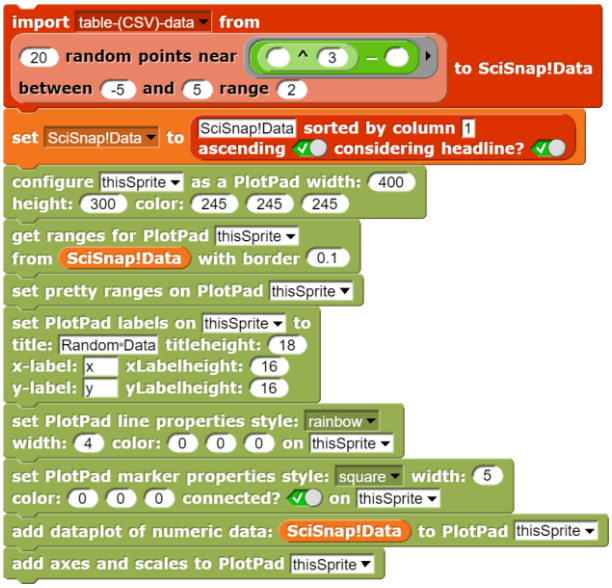
1. Nehmen Sie Musik, Sprache, andere Geräusche als Sounds auf und wandeln Sie die in WAV-Dateien um. Importieren Sie die Sounds nach *SciSnap!*.
2. Experimentieren Sie mit dem Kompressionsverfahren. Lassen Sie die die Ergebnisse von verschiedenen Personen beurteilen.
3. Informieren Sie sich über das MP3 (und andere) Kompressionsverfahren für Sounds.

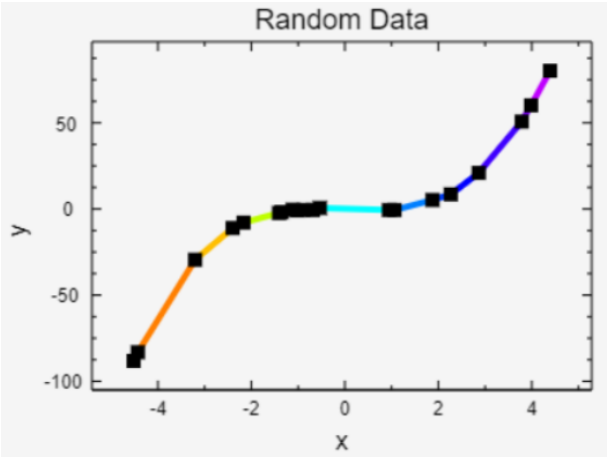
[20] https://de.wikipedia.org/wiki/MP3

7 Datenbezogene Beispiele

7.1 Datenplot von Punkten, die um einen Funktionsgraphen streuen

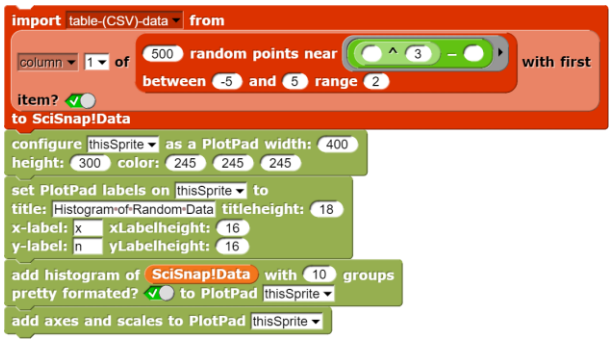
Wir erzeugen einige Datenpunkte in der Nähe des Graphen zu $f(x) = x^3 - x$ und stellen sie auf einem *PlotPad* dar. Damit sie schön geordnet auftreten, sortieren wir die Punkte vor der Erstellung des Diagramms, und weil heute Sonntag ist, werden sie in Regenbogenfarben verbunden.

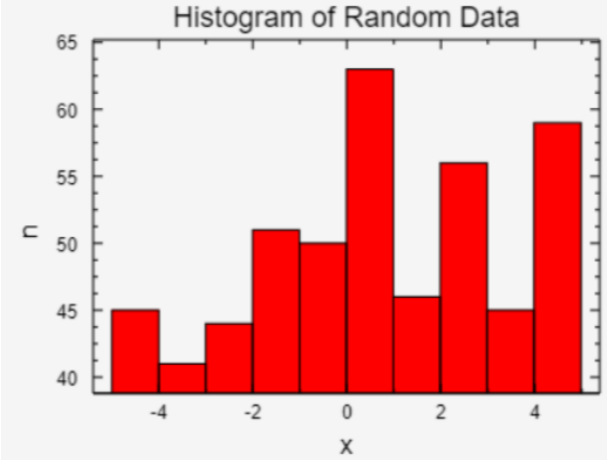




7.2 Histogramm von Zufallswerten

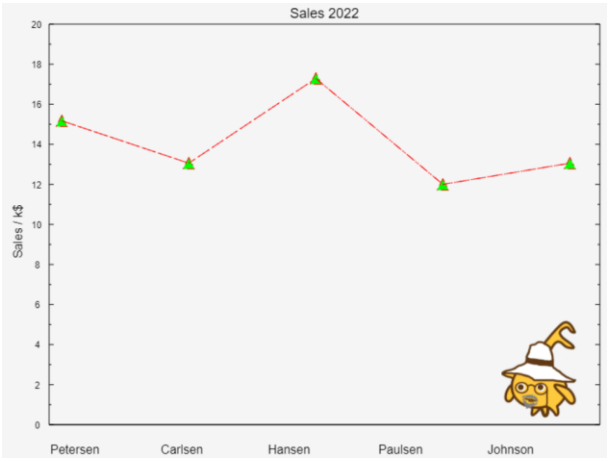
Wir erzeugen wiederum Zufallspunkte, die um den Graphen zu $f(x) = x^3 - x$ streuen, wählen aber diesmal nur die erste Spalte als Datenmenge. Von diesen Werten lassen wir ein Histogramm mit 10 Säulen erstellen.

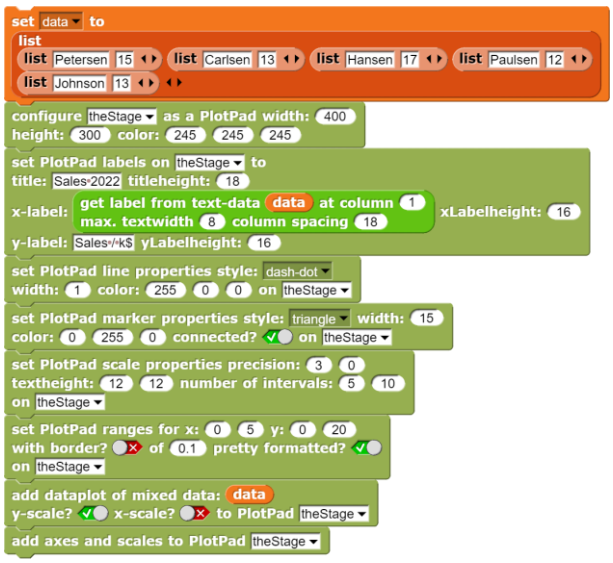




7.3 Darstellung gemischter Daten

Oft werden Textdaten mit numerischen Daten zusammengefasst. Ein Beispiel wären die Umsatzdaten verschiedener Vertreter in einem Jahr in einem Bereich. Wollen wir die grafisch darstellen, dann muss z. B. die x-Achse mit Textdaten beschriftet werden, während die y-Achse wie gehabt behandelt wird. Zum Erzeugen des Diagramms benutzen wir den Block *add dataplot of mixed data* des *PlotPads*. In diesem Fall soll die Bühne als *PlotPad* dienen.

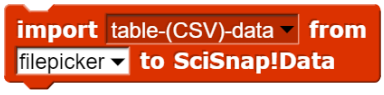


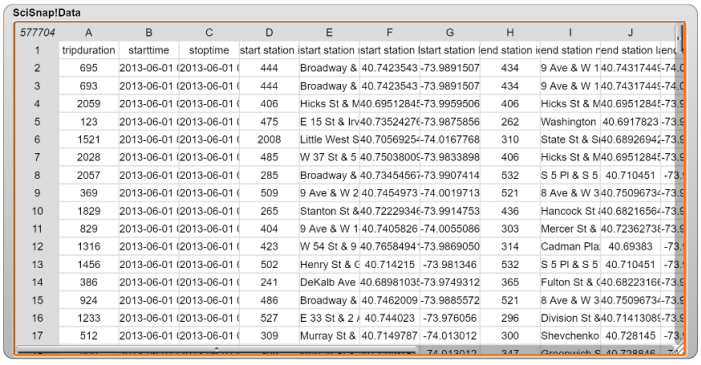


7.46 NY CitiBike Tripdata 1: Korrelationen

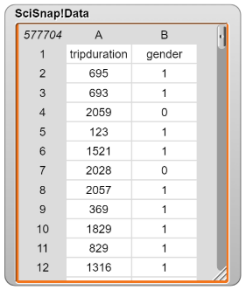
Als Beispiel für die Anwendung der Blöcke wollen wir etwas in einer größeren, frei verfügbaren Datenmenge „wühlen“: den Entleihdaten von New York Citi Bike (NY citibike tripdata: https://www.citibikenyc.com/system-data).

Wir laden die Daten, entpacken sie und erhalten CSV-Daten ziemlicher Größe, die wir „auf einen Schlag“ in den Datenbereich von *SciSnap!*, die Variable *SciSnap!Data*, laden. Wir erhalten knapp 600.000 Datensätze.



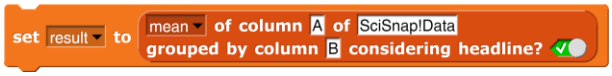


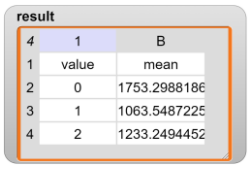
Deren Spaltenüberschriften speichern wir.



Diese Daten lassen sich in sehr unterschiedlicher Art auswerten. Wir werden das später auch noch tun, beschränken uns aber erst einmal auf die Frage, ob es eine Korrelation zwischen Geschlecht und Ausleihdauer gibt. Dafür benötigen wir offensichtlich nur die Spalten 1 und 15.

Zuerst einmal sehen wir uns die Mittelwerte für die unterschiedlichen Geschlechter an (0: unbekannt, 1 männlich, 2: weiblich):





Das haben wir uns doch gedacht! 😉

Und wie sieht es nun mit der Korrelation aus? Wir werfen zuerst die Daten mit dem „unbekannten“ Geschlecht raus. Es bleiben immer noch ca. 340.000 Datensätze. Für diese berechnen wir den Korrelationskoeffizienten zwischen Spalte 1 und 2 – und erhalten das nebenstehende Resultat.

Und was will uns diese Zahl nun sagen??? Wir wissen es nicht – aber wir können ja nachlesen und es lernen! 😉

7.5 New York Citibike Tripdata 2: Radnutzung

Wir wollen mal nachsehen, wer in New York eigentlich Rad fährt. Dazu laden wir uns die Entleihdaten von NY Citibike eines Monats auf den Rechner, das sind die schon erwähnten knapp 600.000 Datensätze. Die sehen wir uns genauer an.

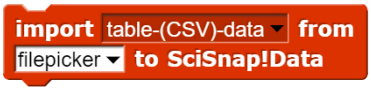


Table view

577704	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	tripduration	starttime	stoptime	start station i	start station	start station	start station l	end station i	end station	end station n	end station l	bikeid	usertype	birth year	gender
2	695	2013-06-01 (	2013-06-01 (	444	Broadway &	40.7423543	-73.9891507	434	9 Ave & W 1	40.7431744	-74.0036644	19678	Subscriber	1983	1
3	693	2013-06-01 (	2013-06-01 (	444	Broadway &	40.7423543	-73.9891507	434	9 Ave & W 1	40.7431744	-74.0036644	16649	Subscriber	1984	1
4	2059	2013-06-01 (	2013-06-01 (	406	Hicks St & M	40.6951284	-73.9959506	406	Hicks St & M	40.6951284	-73.9959506	19599	Customer	NULL	0
5	123	2013-06-01 (	2013-06-01 (	475	E 15 St & Irv	40.7352427	-73.9875856	262	Washington	40.6917823	-73.9737299	16352	Subscriber	1960	1
6	1521	2013-06-01 (	2013-06-01 (	2008	Little West S	40.7056925	-74.0167768	310	State St & Si	40.6892694	-73.9891286	15567	Subscriber	1983	1
7	2028	2013-06-01 (	2013-06-01 (	485	W 37 St & 5	40.7503800	-73.9833898	406	Hicks St & M	40.6951284	-73.9959506	18445	Customer	NULL	0
8	2057	2013-06-01 (	2013-06-01 (	285	Broadway &	40.7345456	-73.9907414	532	S 5 Pl & S 5	40.710451	-73.960876	15693	Subscriber	1991	1

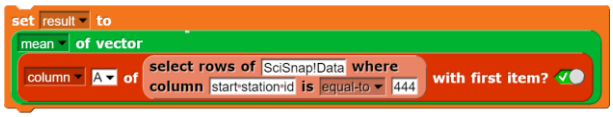
Natürlich müssen wir uns bei der Quelle noch darüber informieren, was die Daten eigentlich bedeuten – also die Metadaten ansehen. Für das Geschlecht erfahren wir, dass 0: *unknown*, 1: *male* und 2: *female* bedeutet. Für die Spalten „*tripduration*“ und „*gender*“ ermitteln wir ein paar Daten:

result

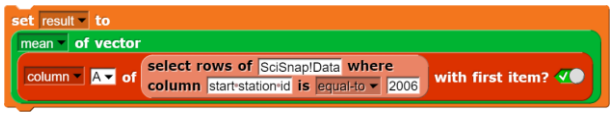
4	A	B
1	value	mean
2	0	1753.2988186817752
3	1	1063.5487225418608
	2	1233.249445298994

Die mittlere Entleihdauer, bezogen auf das Geschlecht, kennen wir ja schon aus dem letzten Beispiel.

Wir sehen mal nach, ob die am Broadway fauler sind:



Aha. Wahrscheinlich ist es Central Park noch schlimmer!



Na gut. Alle Vorurteile müssen ja nicht stimmen. 😉

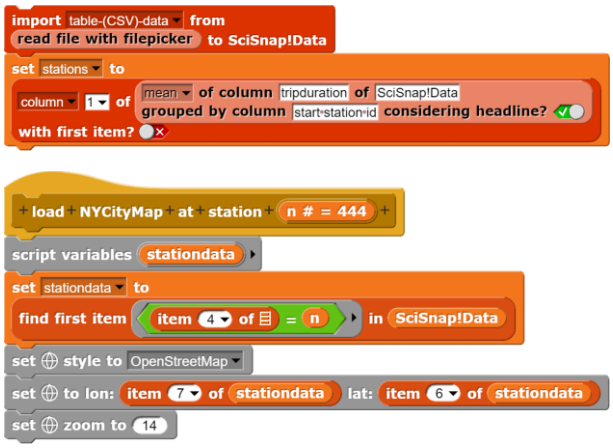
Aufgaben:

1. Vielleicht fahren aber nur die Frauen am Central Park mehr Rad. Überprüfen Sie das.
2. Am Central Park gibt es ja nicht nur eine Entleihstation. Ermitteln Sie geeignete Mittelwerte für den ganzen Bereich.
3. Gibt es eigentlich auch Entleihdaten für andere Stadtteile? Suchen Sie mal und vergleichen Sie die Ergebnisse mit Manhattan.
4. Ermitteln Sie die mittleren Entleihdauern pro Wochentag, insgesamt und für einzelne Stationen. Gibt es da Unterschiede? Weshalb?
5. Oben wurde die mittlere Entleihdauer bezogen auf das Geschlecht berechnet. Man könnte das auch umgekehrt machen. Wäre das völliger Unsinn oder gibt es Fragestellungen, bei denen das sinnvoll wäre?

7.6 New York Citibike Tripdata 3: World Map Library

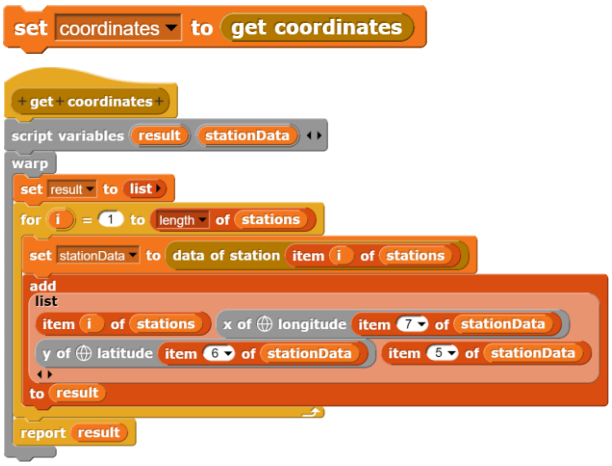
Selbst in New York ist das Fahrradfahren inzwischen „hip“ geworden und die Entleihdaten können als CSV-Dateien geladen werden. Wir tun das wie in den vorherigen Beispielen mit diesem Datensatz. Da wir auch noch Grafiken erstellen wollen, konfigurieren wir ein Sprite als *PlotPad*.

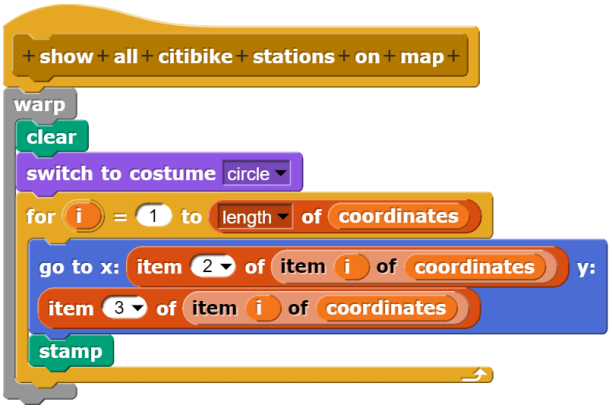
Wir wollen doch mal sehen, wo man Fahrräder entleihen kann. Für die Übersicht extrahieren wir die Entleihstationen aus der Gesamtliste, z. B. indem wir sie nach dem Namen der Startstation (Spalte 5) gruppieren lassen und nur diese Spalte als Ergebnis wählen. Wir erhalten immerhin 337 Stationen. Da geografische Länge und Breite der Entleihstationen angegeben sind, bietet es sich an, die *World Map Library* von *Snap!* einzusetzen. Wir importieren die entsprechende *Snap!*-Bibliothek und schreiben dafür einen kleinen Block, der die Umgebung einer Entleihstation als Karte darstellt.

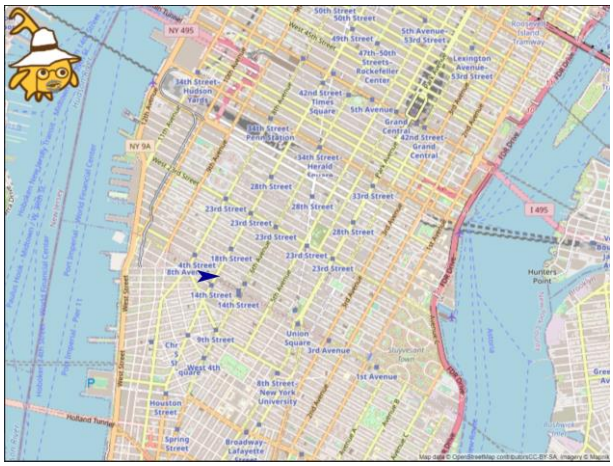


Danach suchen wir die Daten einer Station zusammen …

… und stellen so die Koordinatenliste der Stationen auf.





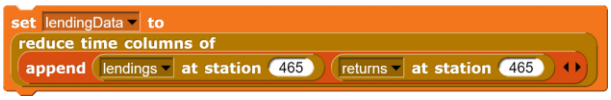


Mit diesen Daten können wir *Hilberto* zu den einzelnen Positionen schicken und dort mit dem *stamp*-Block z. B. zum Hinterlassen von Kreisen auffordern.

Zumindest in Midtown Manhattan müssen wir uns wohl keine Sorge darum machen, ob wir eine Entleihstation finden!

7.7 New York Citibike Tripdata 4: Ausleihdiagramme

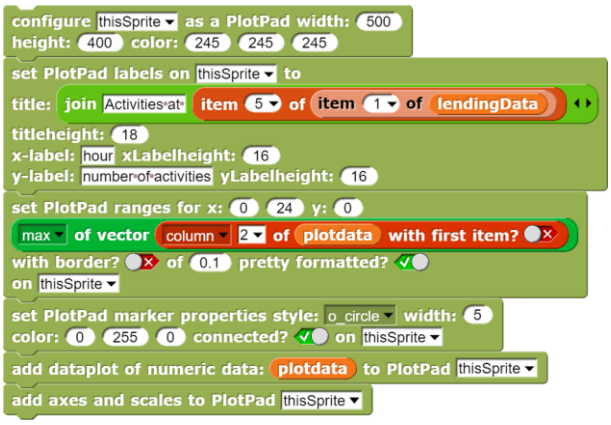
Wir wollen uns jetzt einmal die Verleihstation Broadway – Ecke 41 Street (Nr. 465) genauer ansehen. Dazu ziehen wir alle Datensätze aus der Gesamtliste, die an dieser Station starten oder enden. Das sind in diesem Monat 5054 Vorgänge. Zeiten sind in dieser Liste zusammen mit dem Datum eingetragen. Das können wir herauswerfen (Abtrennen mit *split* mit „ “) und auf die Stunde reduzieren (*split* mit „:“). Wir haben danach eine numerische Skala mit der Einheit „Stunde“. Jetzt können wir sehen, was in den einzelnen Stunden des Tages an der Station los ist.

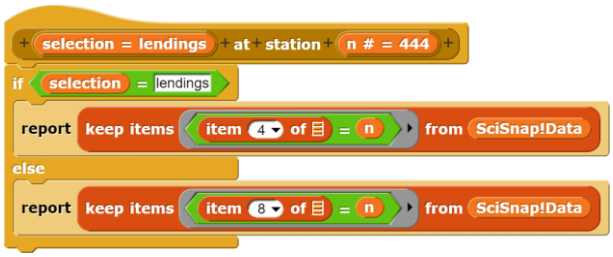


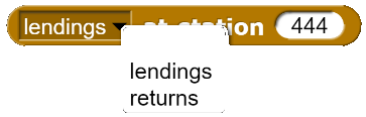
lendingData

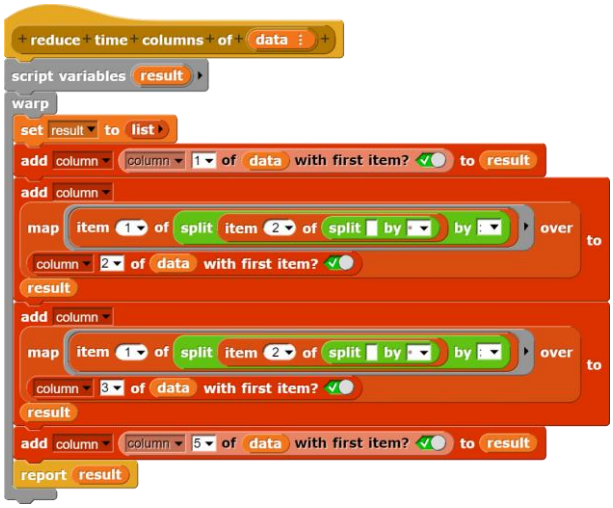
5054	A	B	C
1	627	00	00
2	586	00	00
3	601	00	00
4	993	05	05
5	34966	07	17
6	2314	10	11
7	843	11	11
8	250	12	12
9	707	12	12
10	344	13	13
11	2177	13	13
12	2156	13	13
13	670	13	13
14	2100	14	15
15	1394	15	15
16	820	16	16
17	13649	16	20

Daraus bauen wir ein Diagramm zusammen.

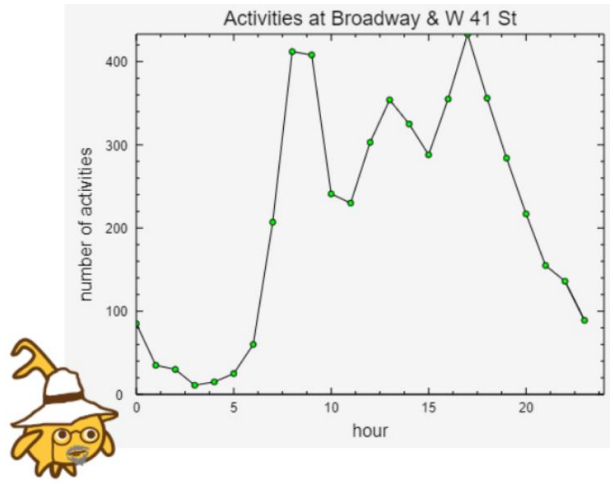




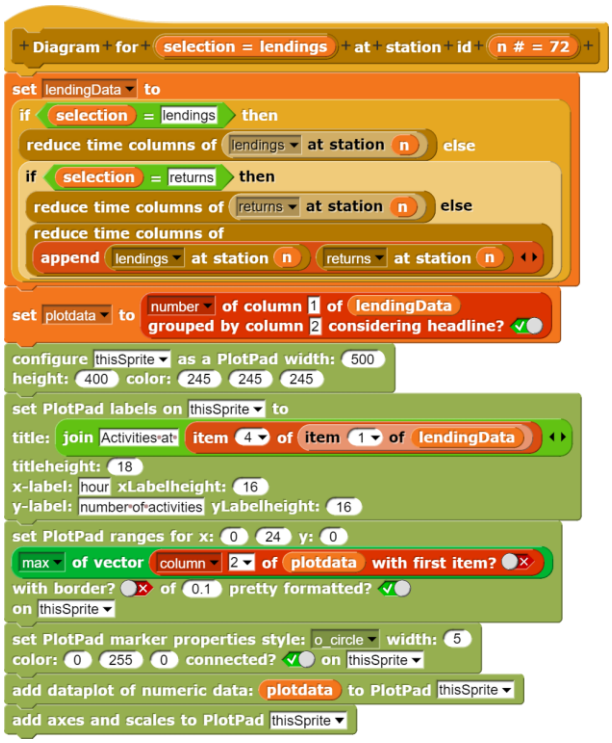




Diese Daten können wir wie üblich grafisch darstellen. Wir zählen einfach, wie viele es in den einzelnen Tagesstunden gab.



Das können wir natürlich zu einem Block zusammenfassen, wobei wir uns noch die Entscheidung offenlassen, ob wir Entleihungen, Rückgaben oder beides erfassen wollen.



Ausleihen

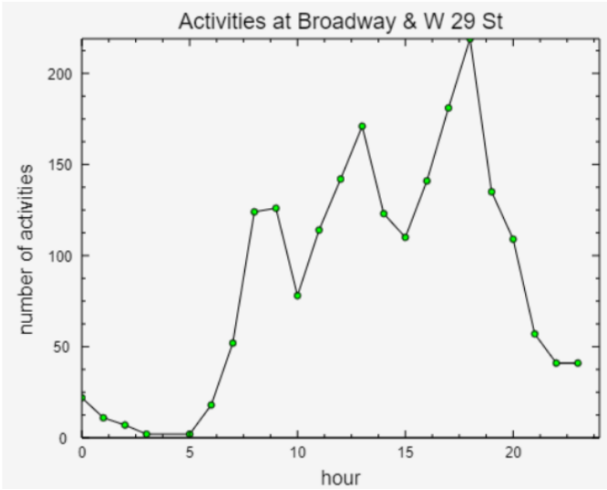
Rückgaben

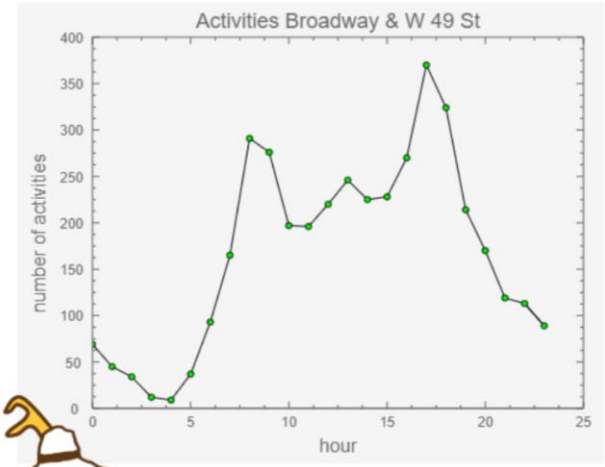
beides

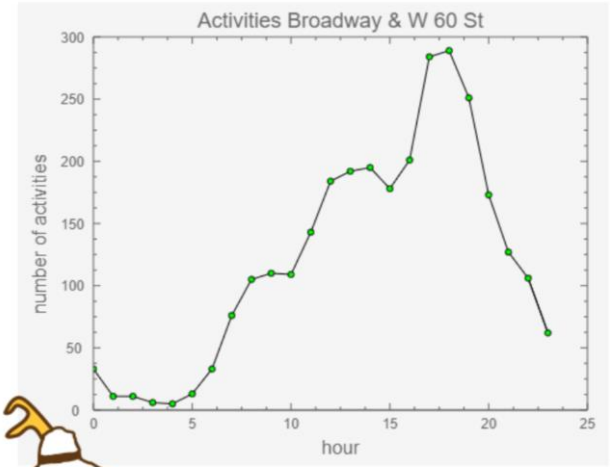
Daten aggregieren

Diagramm erstellen

Ein paar Straßen weiter sieht es ganz ähnlich aus. Ist das ein allgemeines Muster?







Nun gut, am Central Park stehen die Leute später auf und die Touristen sind noch nicht da. Dafür schließen die Museen immer zur gleichen Zeit.

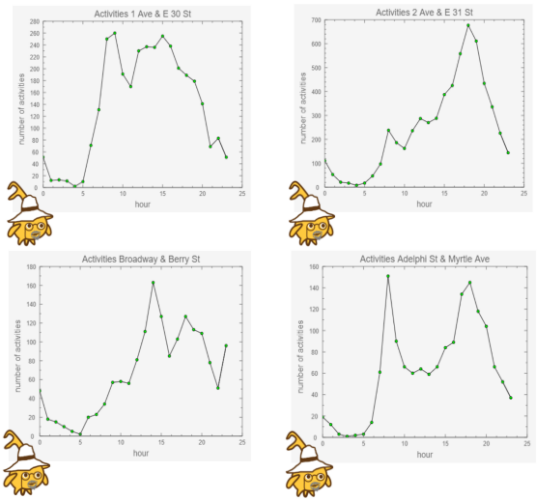
Was können unsere Programme nun aus diesen Daten lernen?

- *Wir könnten z. B. aus den üblichen Ab- und Zugängen sowie dem Istbestand voraussagen, ob an einer Station noch rechtzeitig genügend Fahrräder zurückgegeben werden oder ob es besser wäre, einige dorthin zu transportieren.*
- *Wir könnten aus den mittleren Weglängen ermitteln, welche Akkus für eBikes gebraucht werden.*
- *Wir könnten feststellen, ob eher Frauen oder Männer zu einer bestimmten Tageszeit die Räder entleihen und dann dafür sorgen, dass das Angebot stimmt. Das Entsprechende könnten wir für das Alter der Entleihenden machen.*
- *Wir könnten die Entleihdaten pro Rad ermitteln und voraussagen, wann Reparaturen fällig sein werden. Wir könnten das z. B. auch in Abhängigkeit von der Lage der Stationen machen.*
- *Wir könnten versuchen, Verteilungen von einigen Stationen so zu verallgemeinern, dass sich Prognosen für andere daraus ableiten lassen. Wenn also am Central Park die Museen schließen, kann das Programm aus den alten Daten „lernen“, in welchen Bezirken die Räder wann vermutlich abgegeben werden, und warnen, falls dort nicht genug freie Slots zur Verfügung stehen.*

usw.

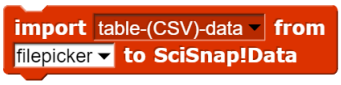
Aufgaben:

1. Gliedern Sie die Aktivitäten der Stationen nach Zu- und Abgängen auf.
2. Schreiben Sie eine Prognosefunktion, die warnt, wenn in den nächsten Stunden ein Radmangel an einer Station droht.
3. Stellen Sie für bestimmte Stationen durch direkte Linien die Verbindungen zu den meist gewählten Abgabestationen graphisch auf der Karte dar. Wählen Sie die Liniendicke entsprechend der Anzahl der Entleihvorgängen und die Farben je nach Station. Bilden sich Cluster?
4. Stellen Sie mithilfe von Korrelationen fest, ob es Zusammenhänge im Entleihverhalten (z. B. bzgl. der Tageszeiten, dem Ort, …) mit dem Geschlecht, dem Alter, dem Status der Entleihenden gibt. Ggf. müssen Sie die Daten vorher durch numerische Daten ersetzen – ähnlich wie bei den Zeiten. Diskutieren Sie mögliche Konsequenzen.
5. Ermitteln Sie für einen kleinen Abschnitt von Midtown (dort, wo alles schön rechteckig ist) die Koordinaten der Straßenecken. Entwickeln Sie dann einen Router, der den kürzesten Weg zur nächsten Citibike-Station anzeigt.
6. Die Entleihzahlen abhängig von der Tageszeit zeigen in unterschiedlichen Bereichen Manhattans ziemliche Unterschiede. Untersuchen Sie Gemeinsamkeiten und Unterschiede systematisch und versuchen Sie, die Ergebnisse zu erklären.



7.8 Einkommensdaten aus dem US Census Income Dataset (Quelle: [Census])

Wir wollen etwas in Daten wühlen und laden uns deshalb den *Census Income Dataset* aus dem Netz.[21] Die entsprechende CSV-Datei lässt sich aus dem Dateiverzeichnis in ***SciSnap!Data*** laden und sofort anzeigen. Sie umfasst 32562 Datensätze. Ein Doppelklick oder ein Rechtsklick darauf und die Wahl von „*open in dialog…*“ zeigt alle Spalten.



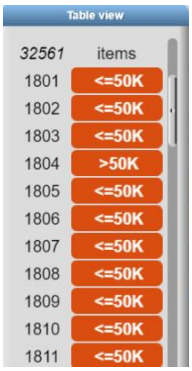
32562	A	B	C	D	E	F	G	H	I	J	K	L	M
1	age	workclass	final weight	education	education-n	marital-statu	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-v
2	39	State-gov	77516	Bachelors	13	Never-marri	Adm-clerica	Not-in-famil	White	Male	2174	0	40
3	50	Self-emp-nc	83311	Bachelors	13	Married-civ-	Exec-manag	Husband	White	Male	0	0	13
4	38	Private	215646	HS-grad	9	Divorced	Handlers-cl	Not-in-famil	White	Male	0	0	40
5	53	Private	234721	11th	7	Married-civ-	Handlers-cl	Husband	Black	Male	0	0	40
6	28	Private	338409	Bachelors	13	Married-civ-	Prof-special	Wife	Black	Female	0	0	40
7	37	Private	284582	Masters	14	Married-civ-	Exec-manag	Wife	White	Female	0	0	40
8	49	Private	160187	9th	5	Married-spc	Other-servic	Not-in-famil	Black	Female	0	0	16
9	52	Self-emp-nc	209642	HS-grad	9	Married-civ-	Exec-manag	Husband	White	Male	0	0	45
10	31	Private	45781	Masters	14	Never-marri	Prof-special	Not-in-famil	White	Female	14084	0	50
11	42	Private	159449	Bachelors	13	Married-civ-	Exec-manag	Husband	White	Male	5178	0	40
12	37	Private	280464	Some-colleg	10	Married-civ-	Exec-manag	Husband	Black	Male	0	0	80

Welche Zusammenhänge könnten sich nun darin zeigen?

Unsere Daten-Blöcke helfen erstmal nicht so sehr weiter, weil sie meist numerische Daten verarbeiten. Wollen wir sie einsetzen, dann müssen wir die Spalten so skalieren, dass sich numerische Inhalte ergeben. Im einfachsten Fall ersetzen wir Texte einfach durch Zahlenwerte – und sollten uns dabei gut überlegen, welche Folgen das bezüglich ihrer Interpretation haben könnte.

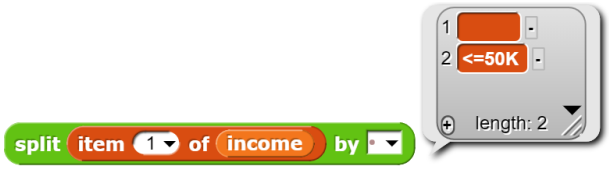
Fangen wir mit der letzten Spalte an: Die Einkommenswerte werden nur für zwei Bereiche angegeben: kleiner oder größer als 50.000$. Wir ordnen diesen Bereichen die Werte 1 und 2 zu. (Oder 0 und 1, oder -1 und +1, oder 0 und 100, oder ….

Hätten diese Änderungen Konsequenzen? Um die Originaldaten nicht zu verändern, erzeugen wir eine Variable ***income*** und speichern dort die veränderten Werte, indem wir die Spalte 15 (income) ohne den ersten Wert (die Überschrift) in diese Variable kopieren und dann mithilfe des ***map…over…***-Blocks die Inhalte verändern. Jedenfalls versuchen wir das. Leider erhalten wir nur die unveränderte Spalte 13, wenn wir uns das Ergebnis wieder als Tabelle ansehen.



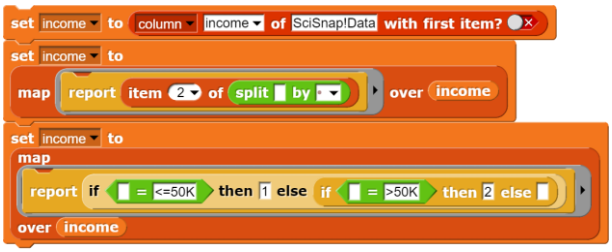
Was ist los? Wir sehen uns das erste Element von ***income*** an und überprüfen, ob es sich um eine Zeichenkette handelt. Das ist der Fall, aber sie ist länger als gedacht:

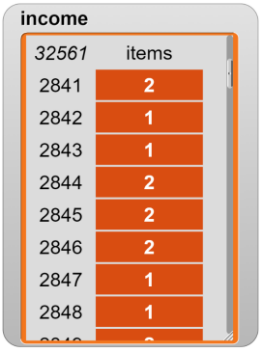
Spalten wir sie bei den Leerzeichen auf, dann sehen wir, was los ist: es haben sich führende Leerzeichen eingeschlichen. Diese Gauner!



[21] Dabei handelt es sich um einen der Trainingsdatensätze für Maschinelles Lernen.

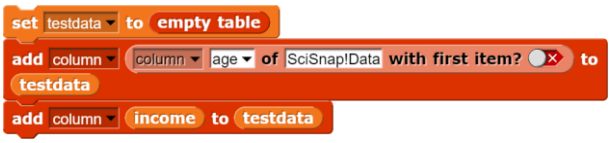
Wir müssen also vorher die führenden Leerzeichen rauswerfen. Das klappt jetzt: unsere Variable ***income*** enthält danach nur noch die Werte 1 und 2, wie wir durch Ansehen schnell überprüfen können.





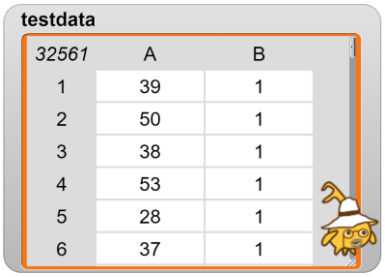
Wovon hängt dieses Einkommen nun ab?

Vielleicht vom Alter? Wir kombinieren die Spalte 1 (Alter) und unsere modifizierte Einkommensspalte zu einer neuen Tabelle namens ***testdata***.



Den Zusammenhang zwischen Alter und Einkommen beschreiben wir durch den Korrelationskoeffizienten. Die Berechnung ist einfach:

Und was bedeutet das?



Aufgaben:

1. Informieren Sie sich über die Bedeutung des Korrelationskoeffizienten und die Interpretation des erhaltenen Werts. Was bedeutet der Wert „0,2340…“?
2. Hängt der Korrelationskoeffizient in diesem Fall von der Art der numerischen Skalierung der Daten (1 und 2, 1 und 0, -1 und 1, …) ab? Überprüfen Sie das.
3. Ermitteln Sie weitere Korrelationskoeffizienten, z. B. zwischen Ausbildung und Einkommen, Herkunftsland und Einkommen, Familienstand und Einkommen, Herkunftsland und Beruf, …
4. Informieren Sie sich darüber, ob und wann die Skalierung nichtnumerischer Daten einen Einfluss auf das Ergebnis haben kann.

7.9 Auswertung von Covid-19-Daten

Wir laden die Covid-19-Zahlen der Johns-Hopkins-Universität vom 1.3.2020 bis 19.4.2020 für vier Länder in den Datenbereich von *SciSnap!* und erhalten:

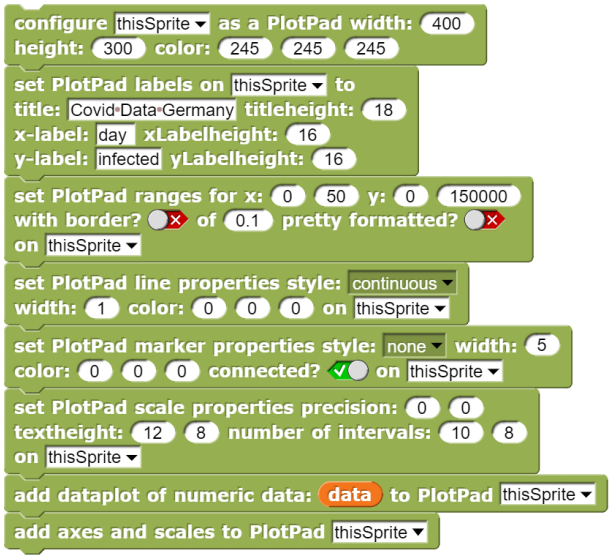
55	A	B	C	D	E	F
1		Covid-10 Infections from 1.3.2020				
2		origin: Johns-Hopkins-University				
3						
4			Infected			
5	Date	Day	Germany	Italy	USA	China
6	1.3.	1	119	1694	43	79826
7	2.3.	2	152	2036	67	80026
8	3.3.	3	190	2502	88	80151
9	4.3.	4	264	3089	117	80271
10	5.3.	5	402	3858	186	80422
11	6.3.	6	641	4636	232	80573
12	7.3.	7	797	5883	351	80652
13	8.3.	8	900	7375	469	80699
14	9.3.	9	1146	9172	535	80735
15	10.3.	10	1567	10149	892	80757
16	11.3.	11	1968	12462	1214	80785
17	12.3.	12	2747	12462	1596	80793
18	13.3.	13	3677	17660	1647	80801
19	14.3.	14	4587	21157	2656	80827
20	15.3	15	5815	24747	3338	80848

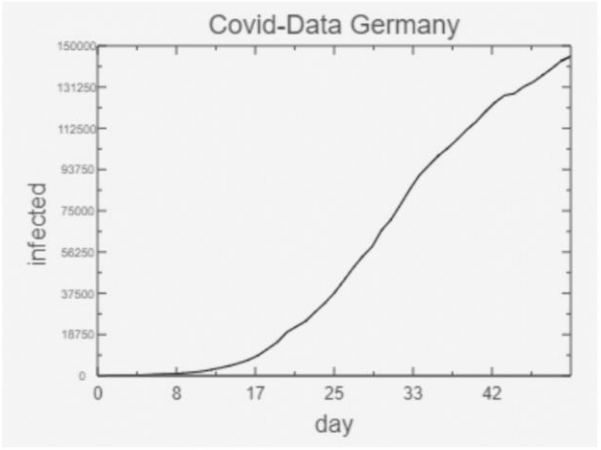
Da wir uns nur für die reinen Daten interessieren, greifen wir den relevanten Datenbereich für ein Land heraus.

data

51	A	B
1	Day	Germany
2	1	119
3	2	152
4	3	190
5	4	264
6	5	402

Darüber verschaffen wir uns erstmal einen Überblick:

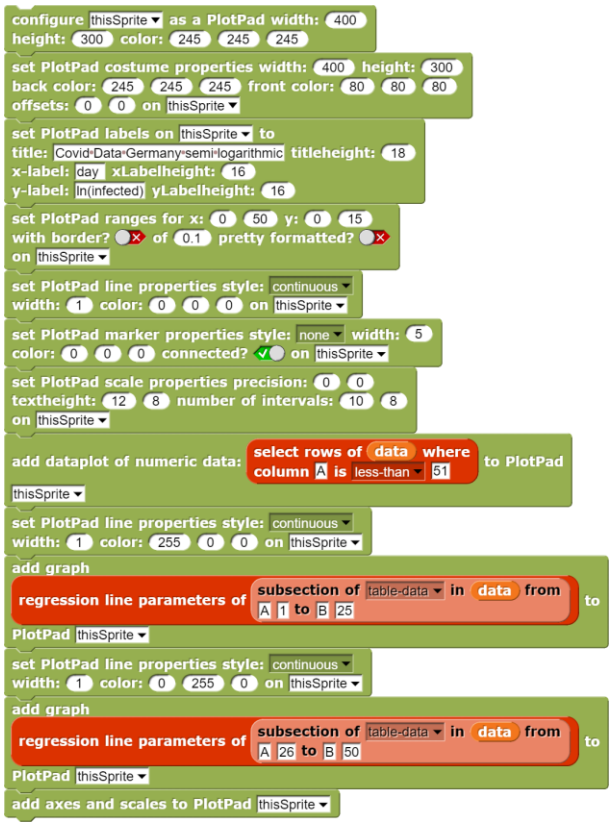


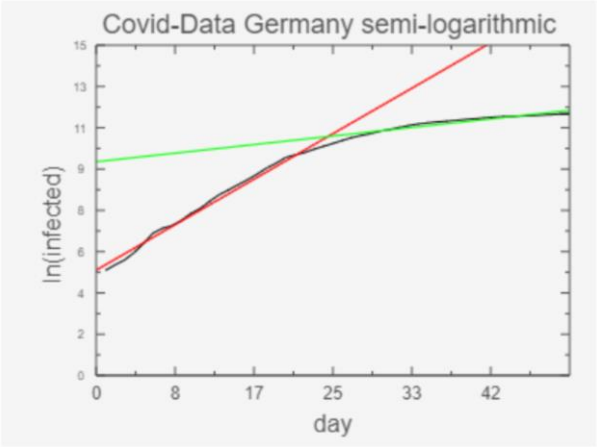


Dann probieren wir es doch einmal mit einer halblogarithmischen Darstellung …

… greifen die beiden interessanten Spalten heraus …

… und passen das Diagramm an: Wir zeigen die halblogarithmisch aufgetragenen Daten und die Regressionsgeraden für die beiden Hälften der Daten.





Da kam Hoffnung auf!

Aufgaben:

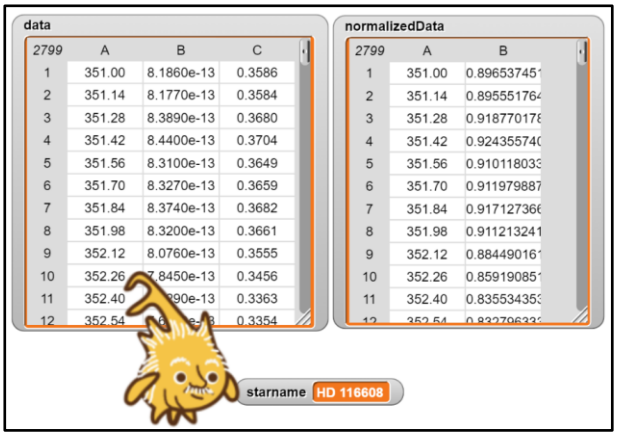
1. Stellen Sie die Daten für die anderen Länder ebenfalls grafisch dar.
2. Versuchen Sie festzustellen, ob es Zusammenhänge zwischen den Datenreihen gibt.

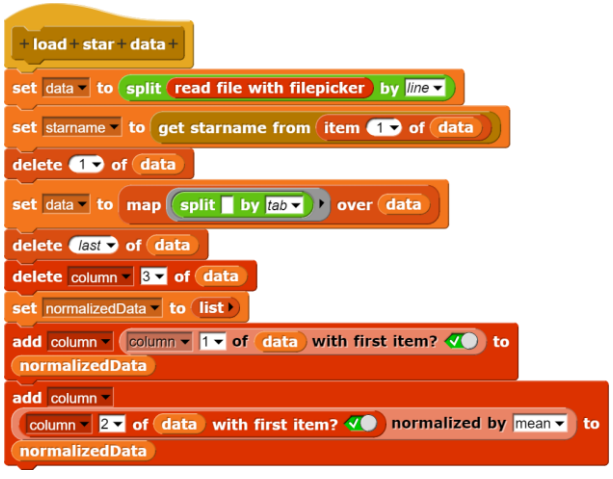
7.10 Sternspektren [UniGOE]

Sterne leuchten in unterschiedlichen Farben, weil sie unterschiedliche Temperaturen haben. Zusätzlich unterscheiden sich die Spektren in ihren Absorptionslinien. Das wollen wir etwas genauer untersuchen.

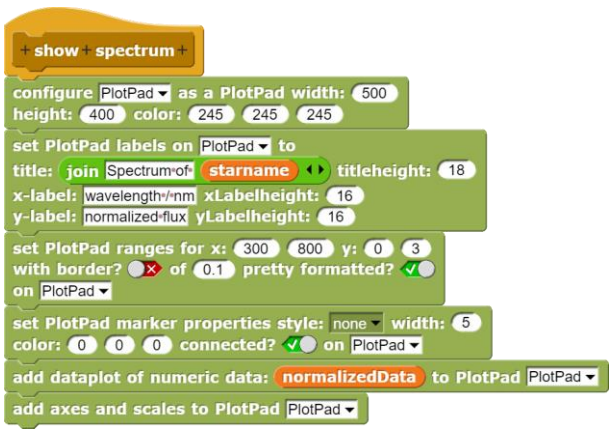
Wir besorgen uns einige Sternspektren (Quelle: [UniGOE]) und speichern sie als Textdatei. Solch eine lesen wir ein und zerlegen sie gleich in eine Liste *data*. In der ersten Zeile steht nach den Spaltenbeschriftungen der Sternname. Den isolieren wir und speichern ihn als *starname*.

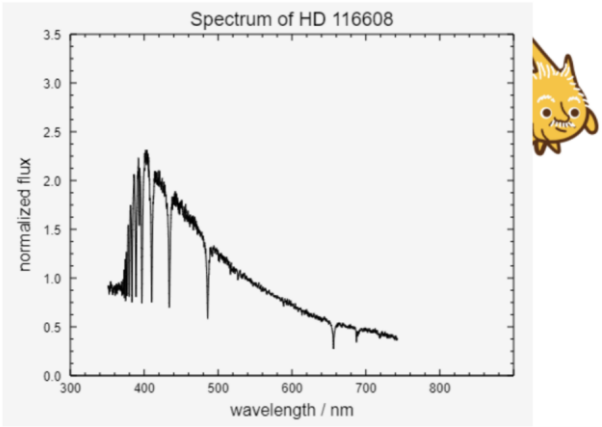
Wir wissen jetzt, wie der Stern heißt. Wenn Sie im Internet danach suchen, finden Sie eine Fülle von Informationen darüber. Damit wir den Ladevorgang mit anderen Daten wiederholen können, kapseln wir ihn in einem eigenen Block. Nach dessen Ausführung liegen die eigentlichen Sterndaten als leicht aufbereitete Tabelle vor. Unschön daran ist die stark unterschiedliche Größenordnung der Daten in den beiden Spalten. Wir normalisieren deshalb die zweite Spalte mithilfe des Mittelwerts und speichern das Ergebnis als *normalizedData*.



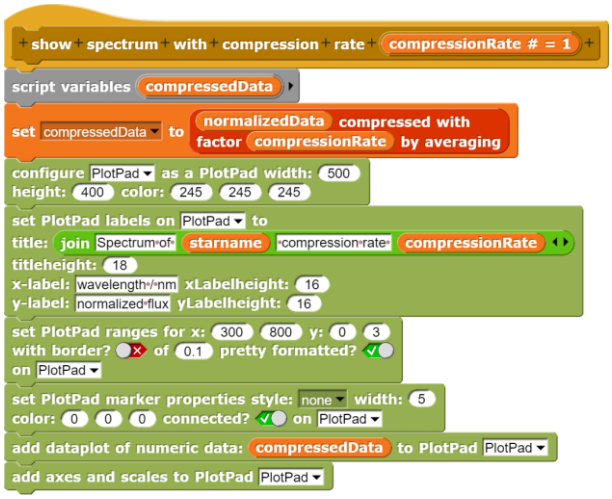


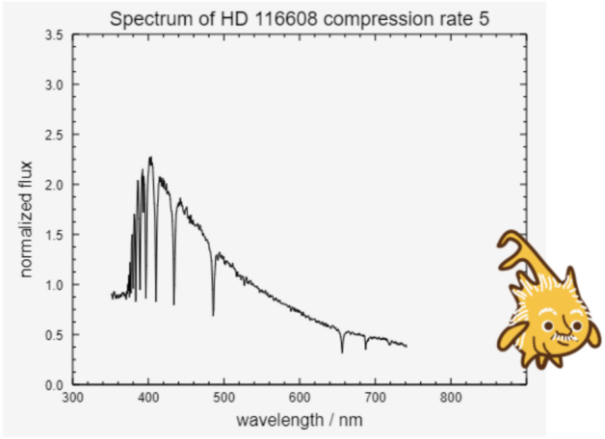
Mit den normalisierten Daten lässt sich schnell ein Diagramm auf einem *PlotPad* erstellen.



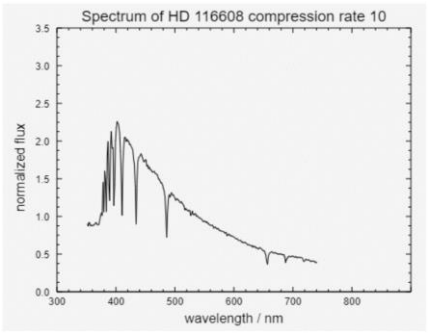


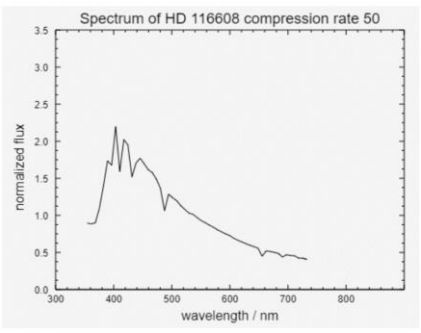
Man erkennt gut den abfallenden Verlauf mit einigen markanten Absorptionslinien. Aber braucht man für diese Erkenntnis überhaupt alle Spektraldaten? Vielleicht genügt es ja, durch Mittelwertbildung die Datenmenge zu reduzieren. Wir führen einen Kompressionsfaktor *compressionRate* ein und ergänzen das Skript vor der Diagrammerstellung.

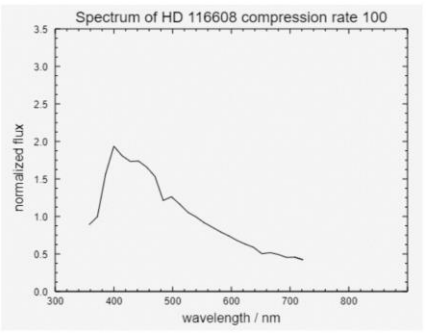




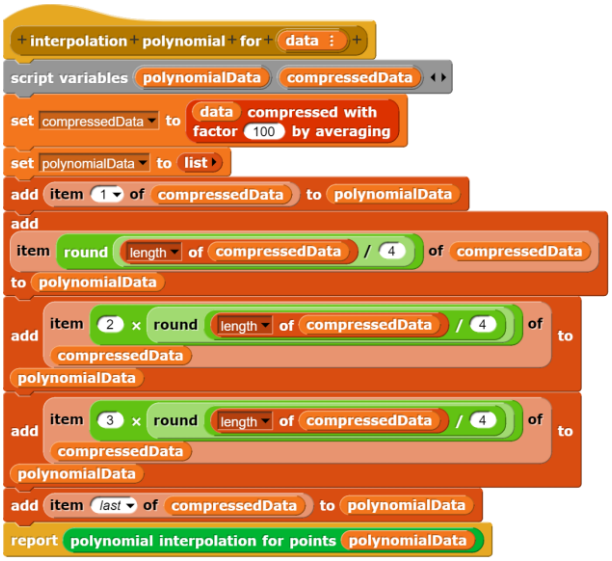
Der Faktor 5 ändert nicht viel. Probieren wir also weiter.

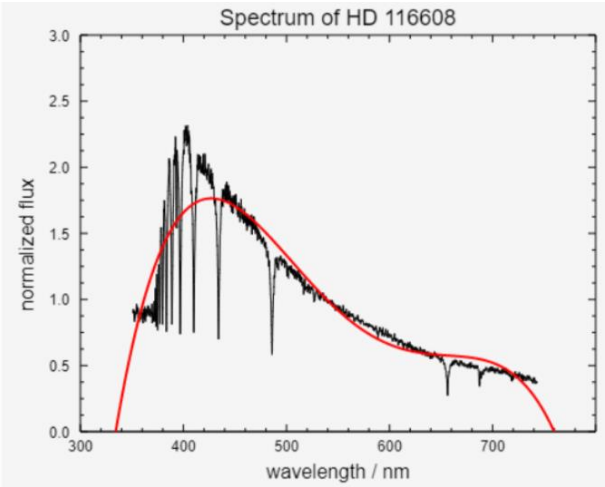






Man sieht, dass der temperaturabhängige Verlauf des Spektrums kaum verändert wird. Nur die Absorptionslinien gehen verloren. Somit sollte sich der Typ des Spektrums durch ein Interpolationspolynom z. B. 4. Grades beschreiben lassen.

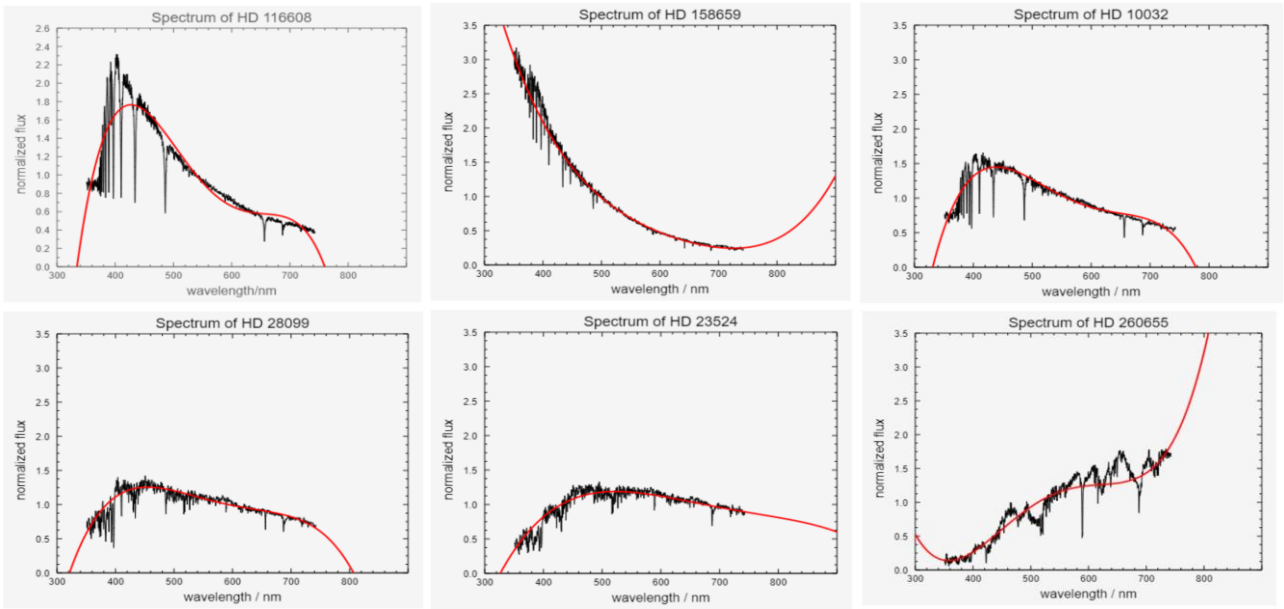




Das funktioniert also hervorragend! Protokollieren wir die Polynomparameter bei der Untersuchung gleich mit, dann können wir anhand der Parameterbereiche die Sterntypen leicht unterscheiden.

7	A	B	C	D	E	F
1	star name	a4	a3	a2	a1	a0
2	HD 116608	-1.2493580327340172e-9	0.0000028868621087800814	-0.002459579857425176	0.9103088865090065	0.9103088865090065
3	HD 158659	1.565259017017166e-10	-3.963032080178107e-7	0.0003879661846290463	-0.17622312866994078	-0.17622312866994078
4	HD 10032	-7.27005023271818e-10	0.00000169492984799126 4	-0.001462425925103779	0.5500801501694278	0.5500801501694278
5	HD 28099	-4.0018935572381893e-10	9.399457129604694e-7	-0.0008209889485783107	0.3141072191721327	0.3141072191721327
6	HD 23524	-8.18301248511472e-11	2.3253458278204257e-7	-0.00024615800544876965	0.11348374829256708	0.11348374829256708
7	HD 260655	6.248027476637483e-10	-0.000001337322548726115	0.0010450333683869723	-0.3486709605339992	-0.3486709605339992

Füttert man mit den Polynomkoeffizienten ein Neuronales Netz, dann lernt dieses schnell, ein Diagramm grob einem Sterntyp zuzuordnen. Das Programm kann anhand der alten Daten also „lernen“, welche Parameterintervalle zu welchen Sternklassen gehören. Gibt man die Daten eines neuen Sterns ein, dann ermittelt es die Koeffizienten des Polynoms und gibt danach eine gut begründete Prognose ab, um welche Art von Stern es sich handeln könnte.

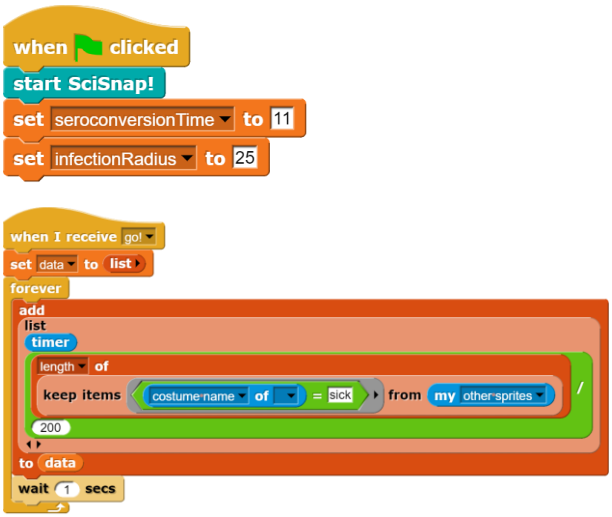


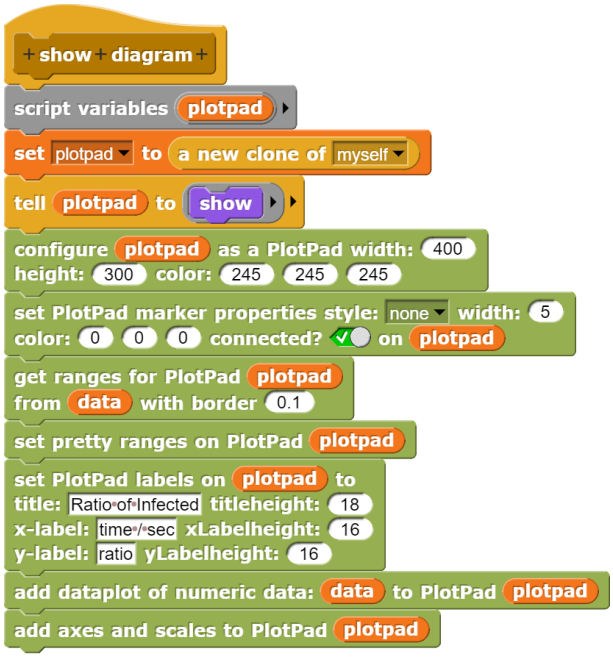
Aufgaben:

1. Stellen Sie für die nicht komprimierten Spektrumsdaten jeweils ein Interpolationspolynom möglichst niedrigen Grades auf. Welche Punkte sollten dafür gewählt werden? Ergeben sich Unterschiede zwischen diesen Polynomen und den Ergebnissen des oben gezeigten Verfahrens?
2. Entwickeln Sie ein Skript, das ein unbekanntes Spektrum einem der bisher auftretenden Typen zuordnet.
3. Entwickeln Sie ein Verfahren, um die markantesten Absorptionslinien genauer zu untersuchen. Stellen Sie diese für Sterne der gleichen Klasse vergrößert dar und versuchen Sie, Unterschiede „automatisch“ zu bestimmen. Diskutieren Sie Ihre Ideen vor der Realisierung.

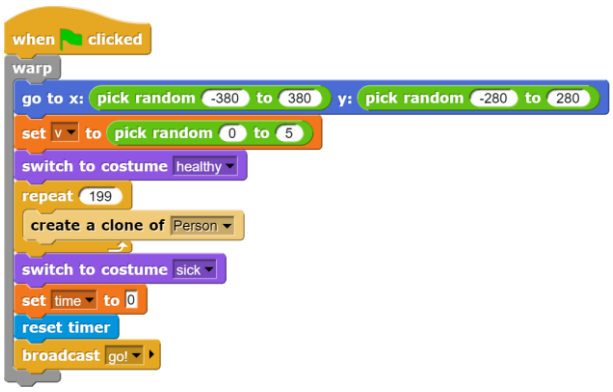
7.11 Simulation einer Grippewelle

Wir wollen auf möglichst einfache Art eine Grippewelle simulieren, die durch nur zwei Parameter gesteuert wird: Die *Serokonversionszeit* ist die Zeit zwischen Infektion und Übergang zur Immunität, der *Infektionsradius* gibt an, in welchem Abstand andere Personen infiziert werden. Andere Parameter wie z. B. die Infektionswahrscheinlichkeit sollten ergänzt werden. Die Simulation besteht aus 200 „Personen“ nur einer Art, die durch farbige Kreise symbolisiert werden: Grün für Gesunde, Rot für Infizierte und Gelb für Immune. Die Personen wuseln in der Gegend herum, begegnen und infizieren sich. Nach der Seronkonversionszeit werden sie immun. *Hilberto* hat wenig zu tun: er setzt ein paar Anfangswerte und protokolliert den Anteil der Infizierten im Sekundenabstand. Bei Bedarf kann er ein Diagramm der aufgenommenen Daten erzeugen.





Etwas aktiver ist eine *Person*, also ein Sprite, das als Vorlage für die anderen 199 Klone dient. Da diese anfangs gesund sein sollen, nimmt es das „gesunde“ Kostüm an, erzeugt die Klone, setzt die Zeitmessung zurück und gibt ein Startsignal. Außerdem wechselt sie zum Kostüm „krank“. Damit ist sie der Ausgangspunkt der Infektion.



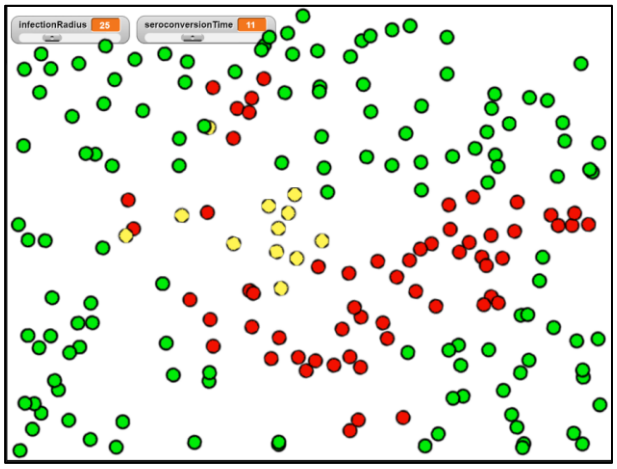
Die erzeugten Klone nehmen eine zufällige Position ein, setzen ihre Geschwindigkeit auf einen Zufallswert und die bisherige Krankheitsdauer auf -1, denn sie sind ja noch gesund.

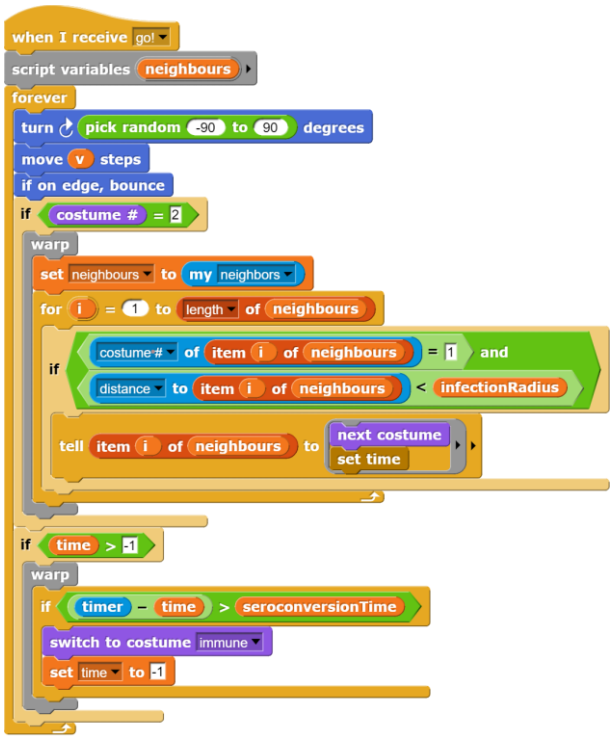
Nach Empfang des Startsignals dreht sich eine Person etwas und bewegt sich. Trifft sie den Rand, dann prallt sie von diesem ab.

Sollte die Person krank sein, dann bestimmt sie ihre Nachbarn. Sind diese gesund und nahe genug, dann erkranken sie.

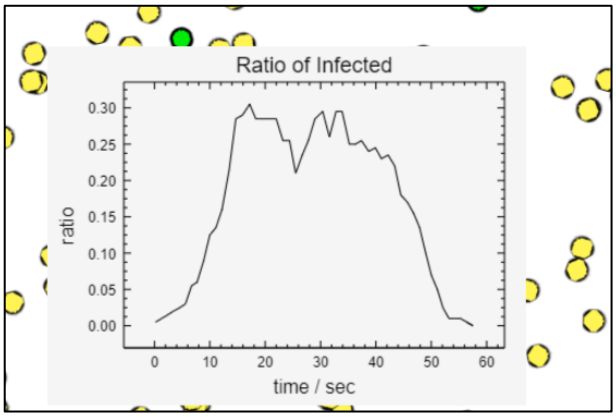
Ist die Person schon eine Zeitlang krank, dann sieht sie nach, ob die Serokonversionszeit um ist. Falls das der Fall ist, wird sie immun.

Das wars!





Ein typisches Diagramm des Infektionsverlaufs:



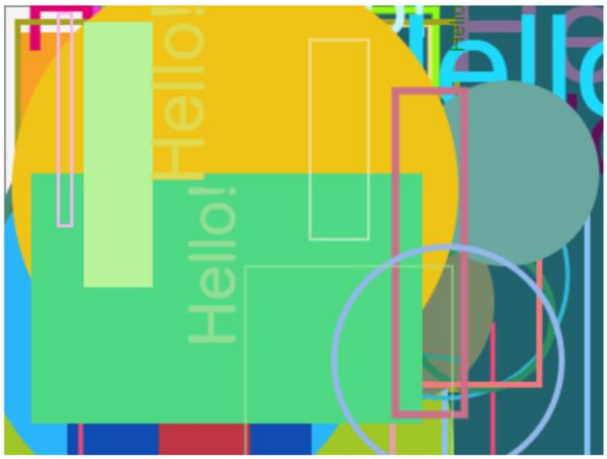
Aufgaben:

1. Nicht bei jedem Kontakt mit Erkrankten erkranken Personen. Führen Sie eine Infektionswahrscheinlichkeit ein.
2. Es gibt unterschiedliche Personen, z. B. solche, die Schutzmasken tragen – oder nicht. Berücksichtigen Sie das bei der Simulation.
3. Auch die Beweglichkeit von Personen ist unterschiedlich, einige bleiben meist zuhause, während andere in der Welt herumreisen. Berücksichtigen Sie das in der Simulation.
4. Auch die Viren gehen mit der Zeit. Ab und zu entstehen Mutationen, die eine andere Infektionswahrscheinlichkeit bewirken. Berücksichtigen Sie das in der Simulation.

8 Grafikbezogene Beispiele

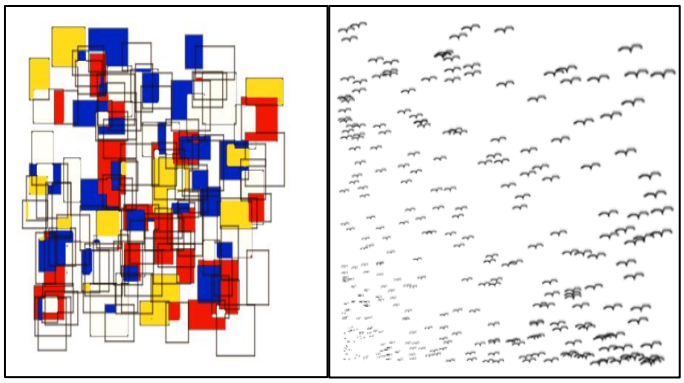
8.1 Einfache Zufallsgrafik

Wir zeichnen einfach 100 zufällig gewählte Grafikelemente übereinander.



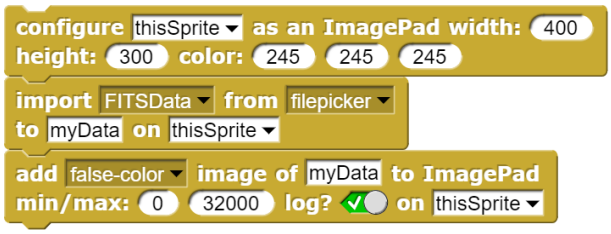
Aufgaben:

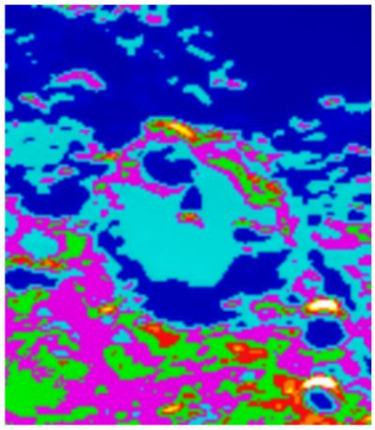
1. Suchen Sie im Netz nach Bildern von Piet Mondrian. Versuchen Sie, ähnliche Zufallsbilder auf dem *ImagePad* zu erzeugen.
2. Mithilfe eines „Fluchtpunktes“ lassen sich Bilder erzeugen, in denen sich Objekte scheinbar „von hinten nach vorne“ bewegen. Versuchen Sie es.



8.2 Falschfarbenbild eines Mondkraters

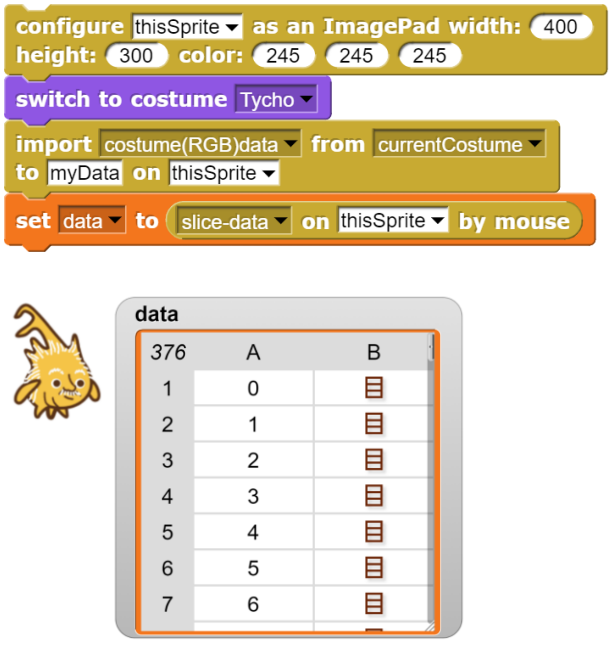
Wir importieren die Bilddaten aus einer FITS-Datei und stellen sie anschließend als Falschfarbenbild dar.

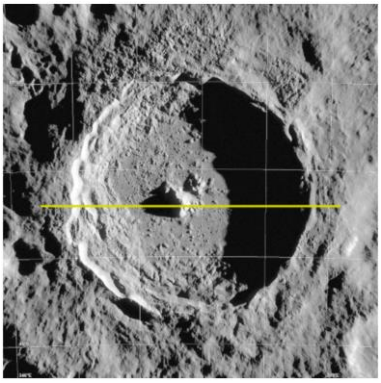




8.3 Schnitt durch ein Bild des Mondkraters Tycho

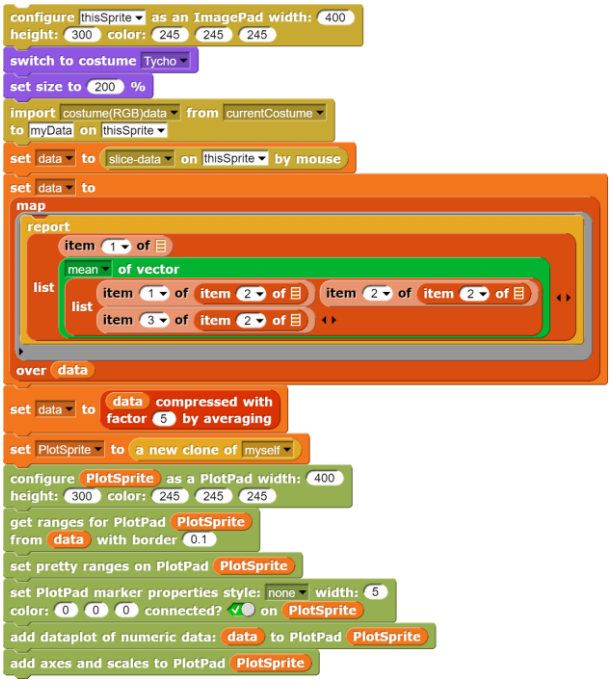
https://www.spektrum.de/fm/912/thumbnails/Mond0.jpg.2996657.jpg





8.4 Schattenlängen im Mondkrater Tycho

Wir erzeugen wie oben gezeigt ein Bild des Mondkraters auf einem *ImagePad,* importieren dessen Daten und legen mithilfe der Maus einen Schnitt durch das Bild. Die Datenwerte der Schnittlinie stellen wir auf einem *PlotPad* dar. Aus diesen und einigen Zusatzdaten können die Schattenlängen berechnet werden.

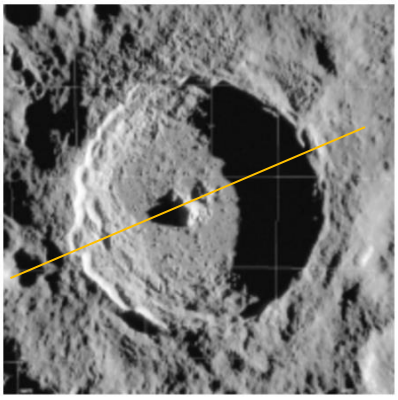


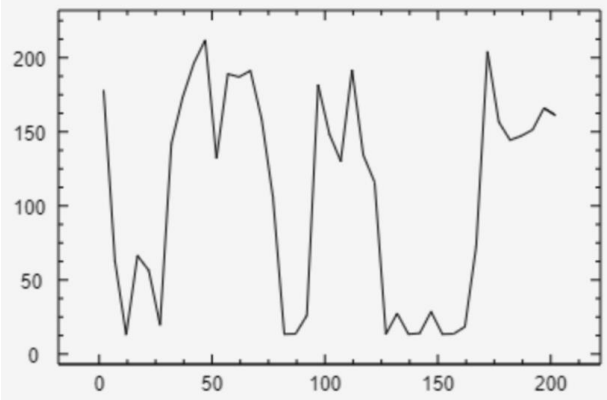
Darstellung des Kraterbildes auf dem ImagePad.

Datenaufnahme mit der Maus.
Umrechnung der RGB-Werte in Grauwerte.

Datenkompression mit dem Faktor 5.

Erzeugung eines Diagramms auf einem zweiten Sprite, das als PlotPad konfiguriert wird.

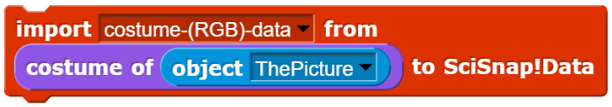




8.5 Darstellung von Bilddaten als Histogramm

Ein RGB-Bild wird geladen, in Graustufen zerlegt, und die normalisierte Verteilung der Bildwerte wird als Histogramm auf einem neuen *PlotPad* dargestellt. Das eigentliche Bild finden wir als Kostüm eines zusätzlichen Sprites namens „*ThePicture*“.

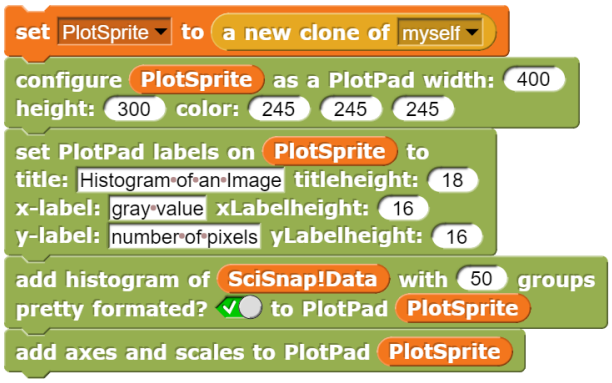
Zuerst einmal laden wir das Bild in den Datenbereich *SciSnap!Data*:

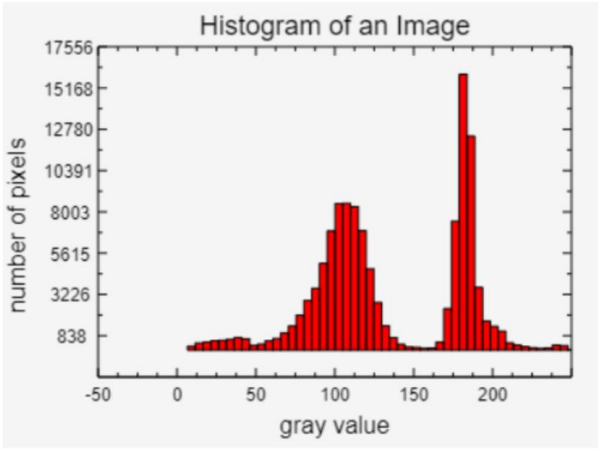


Wir erhalten 120.000 RBG-Werte.

Diese wandeln wir in Grauwerte um.

Danach erzeugen wir ein *PlotSprite* als Klon des aktuellen Sprites und konfigurieren es als *PlotPad*. Auf diesem erzeugen wir das Histogramm.



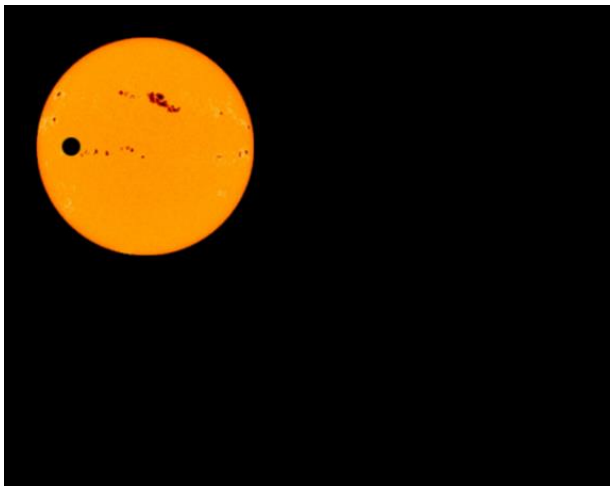


Aufgaben:

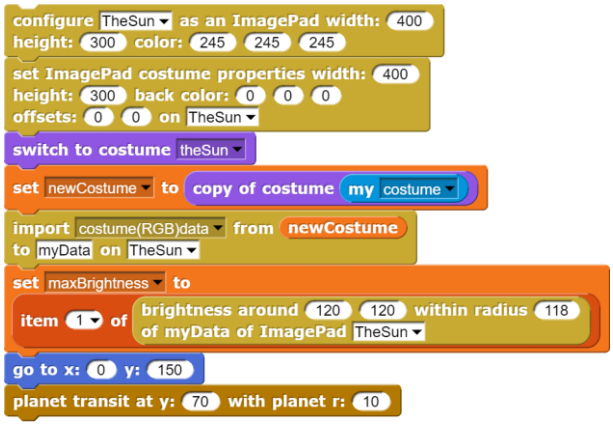
1. Suchen Sie im Netz unterschiedliche Datenmengen. Stellen Sie diese oder Teile von diesen grafisch dar.
2. Automatisieren Sie die Histogrammerstellung durch einen neuen Block ***histogram of*** *<costume>*. Vergleichen Sie die Histogramme typischer Bildtypen. Inwieweit ist ein Vergleich von Bildern auf diese Art möglich bzw. wo könnten Schwierigkeiten auftreten?
3. Stellen Sie im gleichen Diagramm die drei Farben eines RGB-Bildes durch Graphen und/oder Histogramme dar.

8.6 Simulation eines Planeten-Transits vor der Sonne

Wir suchen uns ein schönes Bild der Sonne (Quelle hier: [Schul Astro]) und laden es als Kostüm eines Sprites. Damit es mehr nach Weltall aussieht, vergrößern wir die Bühne und färben sie schwarz. Zeichnen wir noch den Planeten, dann erhalten wir das nebenstehende Bild.

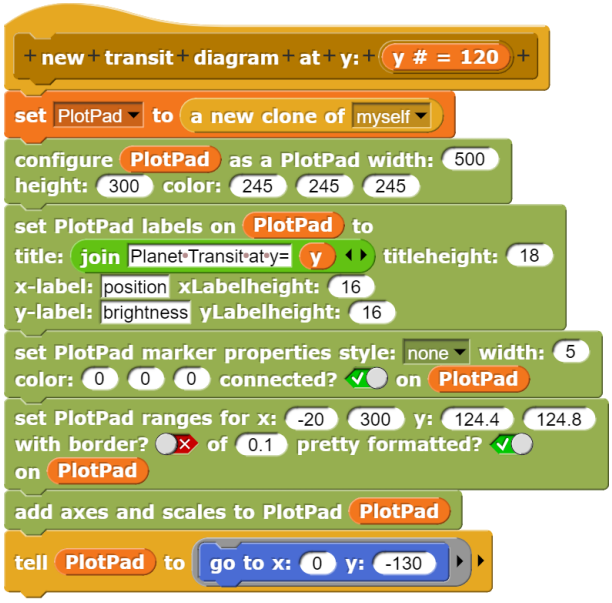


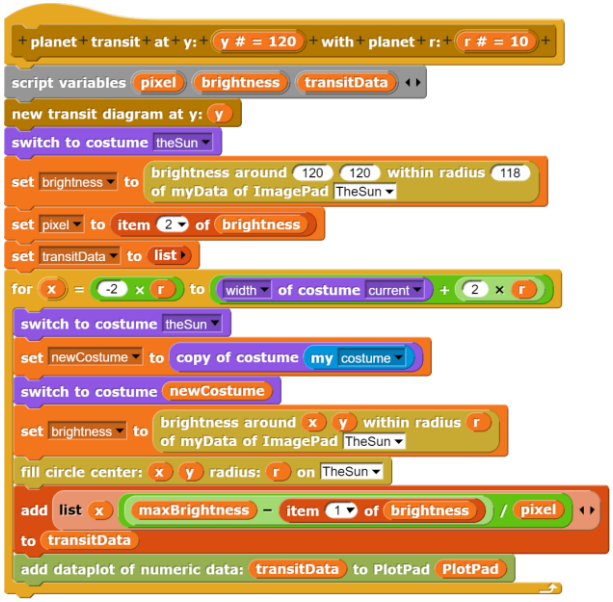
Der Planet soll als schwarzer Kreis vor der Sonne vorbeiziehen. Wenn wir einen solchen Kreis malen, dann verändern wir das eigentliche Sonnenbild. Von diesem ziehen wir deshalb eine Kopie *newCostume*, auf der wir dann zeichnen. Unser Planet soll sich von ganz links etwas außerhalb des Bildes (***x***=*-2r*) nach ganz rechts (***x***=*Bildbreite+2r*) auf der Höhe ***y*** bewegen. Auch den Radius ***r*** des Planeten können wir vorgeben.



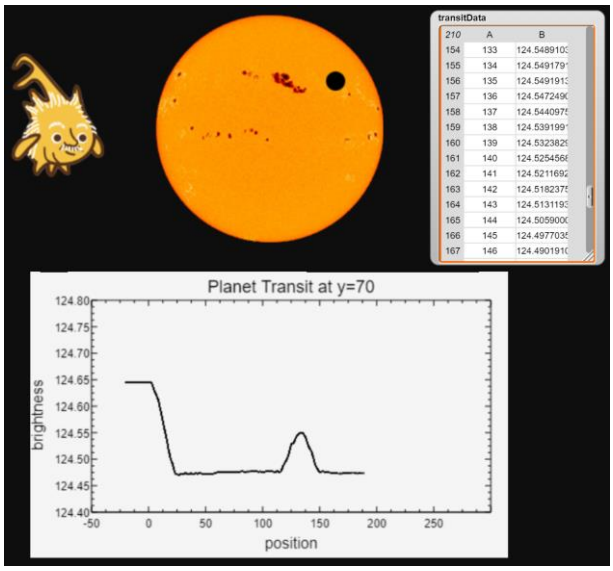
Die aktuelle Helligkeit dieser Anordnung können wir ohne allzu viele Kopiervorgänge bestimmen, indem wir von der anfangs bestimmten Gesamthelligkeit jeweils die Helligkeit der vom Planeten verdeckten Pixel abziehen. Dazu wird anfangs das Bild der Sonne in den Datenbereich *myData* importiert und die Helligkeit um den Bild-Mittelpunkt im Radius „halbe Bildbreite“ sowie die Anzahl der beteiligten Pixel bestimmt. *brightness around* liefert den summierten Grauwert sowie diese Anzahl. Aus diesen Größen berechnen wir jeweils die mittlere Helligkeit der „leicht verdunkelten“ Sonne und speichern sie zusammen mit der aktuellen Position in der Variablen *transitData*.

Parallel zum Transit soll ein Diagramm erstellt werden, in dem wir die Ergebnisse „live“ verfolgen können. Dazu erzeugen wir ein weiteres Sprite, das wir *PlotPad* nennen und das wir entsprechend konfigurieren. Die dafür erforderlichen Befehle verpacken wir in einem Block namens *new transit diagram*.





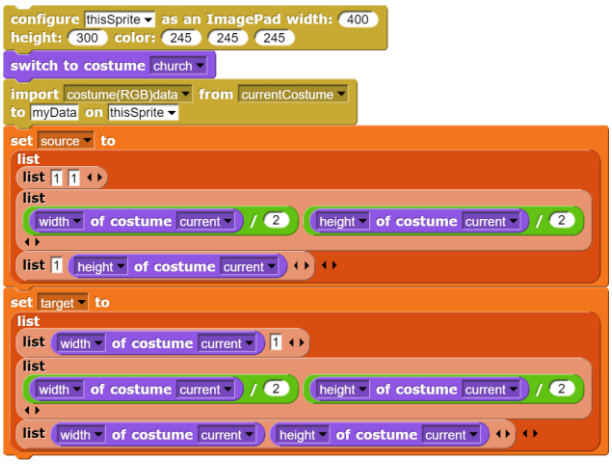
Danach schreiben wir einen Block, der die Helligkeitsdaten wie beschrieben ermittelt und parallel das Diagramm auffrischt. Das Ergebnis entspricht in etwa einer der Methoden, mit denen Exo-Planeten gefunden werden.



8.7 Affine Transformation eines Bildes

Beim *ImagePad* finden wir einen Block, der es gestattet, affine Transformationen in einem Bild vorzunehmen, indem man drei Punkte auf drei andere abbildet – und alle anderen Punkte entsprechend. Als Bild wird das aktuelle Kostüm eines Sprites genommen.

Wir wollen ein Bild an der Mittellinie vertikal spiegeln. Wir laden das Bild – hier: einer Kirche – und wählen dazu entsprechende Punkte an den Rändern aus. Diese fassen wir zu den beiden Listen *source* und *target* zusammen.

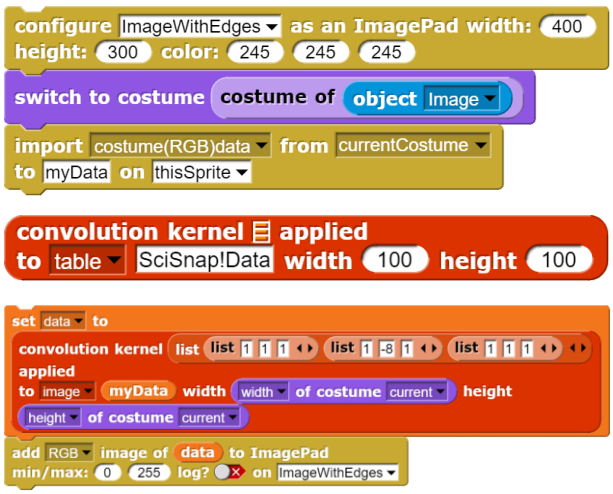


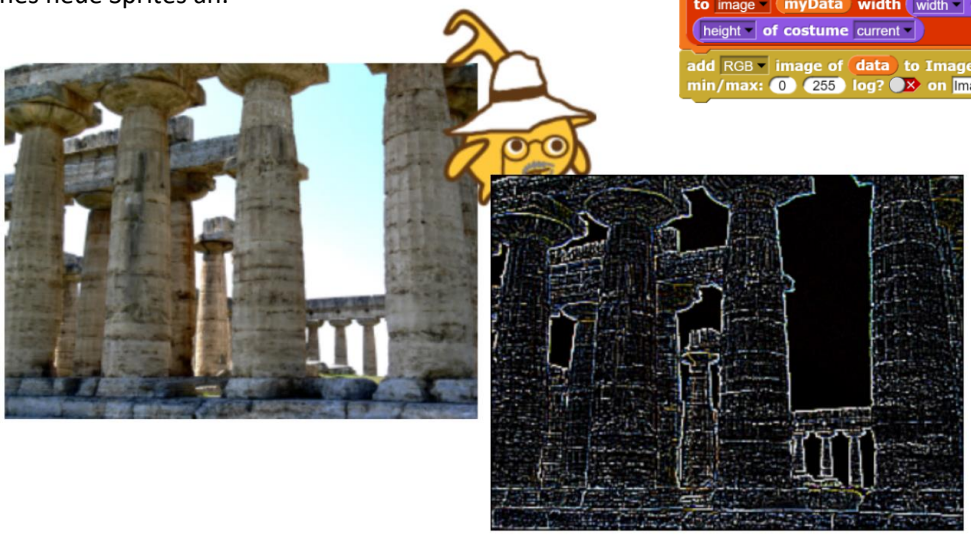
Zuletzt erzeugen wir einen Klon des *ImagePads* und bitten diesen, das transformierte Bild als Kostüm anzuzeigen.

8.8 Kernel-Anwendung zur Kantenerkennung

Wir konfigurieren ein Sprite als *ImagePad* und wechseln zum Kostüm eines antiken Tempels. Dieses Bild importieren wir in den Datenbereich *myData* des *ImagePads*.

Auf diese Bilddaten wenden wir den *Laplace-Kernel* $\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ mithilfe des Blocks *convolution kernel applied …* der *DataTools* an und speichern das Ergebnis in der Variablen *data*. Das Ergebnis zeigen wir als Kostüm eines neue Sprites an.





Aufgaben:

1. Bilder sind manchmal etwas „flau“. Das liegt daran, dass sie nicht den vollen Wertebereich für die drei Farbkanäle von 0 bis 255 ausnutzen.
   a: Entwickeln Sie eine Methode, die Wertebereiche eines Bildes zu ermitteln und anzeigen zu lassen.
   b: Entwickeln Sie eine Methode; den vollen Wertebereich auszuschöpfen, also schwarze Pixel auf 0, helle auf 255 abzubilden.
   c: Fassen Sie das Verfahren in einem neuen Block zusammen, dem das Kostüm eines Sprites übergeben wird und der das verbesserte Kostüm als Ergebnis zurückgibt.

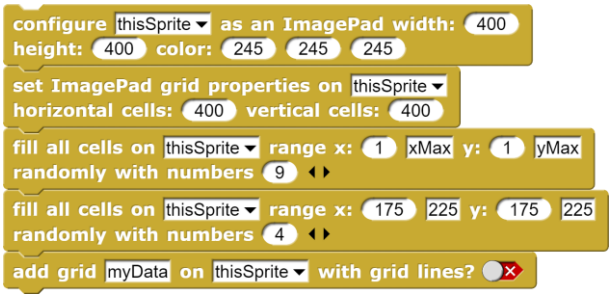
2. a: Auf Bildern kann man versuchen, „Gesichter“ zu finden, indem man zusammenhängende Bereiche eines Farbbereichs, z. B. „orange“, hervorhebt und den Rest des Bildes löscht. Versuchen Sie, dafür einen neuen Block zu entwickeln.
   b: Mithilfe eines Kernels können die Ränder solcher Bereiche isoliert werden. Informieren Sie sich z. B. im Netz über geeignete Kernels und erproben Sie diese bzgl. des genannten Zwecks.
   c: Gesichter sind oft „oval“. Versuchen Sie auf diesem Wege Gesichter von anderen „orangenen“ Gegenständen zu unterscheiden.

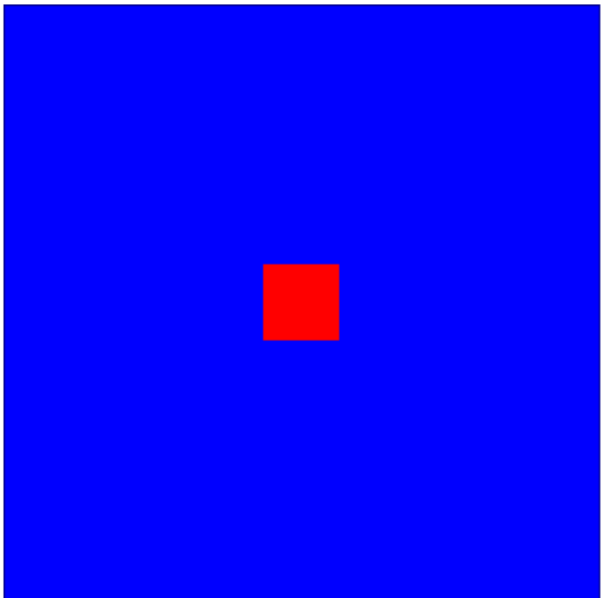
3. a: Richtig künstlerische Fotos sind natürlich schwarz-weiß. Wenn man keine hat, kann man aus RGB-Bildern Graustufenbilder erzeugen. Tun Sie das.
   b: Noch künstlerischer wirkt es, wenn die Fotos „hart“ sind, also einen sehr starken Kontrast haben. Experimentieren Sie mal ein bisschen!

8.9 Diffusion im Gittermodell

Bringt man etwas Farbe in einen Wasserbehälter, dann verteilen sich die Farbteilchen langsam in das sie umgebende Wasser. Man nennt das *Diffusion*. Wir können uns den Vorgang als einen Austauschprozess vorstellen, in dem Farbteilchen ihren Platz mit Wasserteilchen tauschen – und natürlich tauschen auch Teilchen der gleichen Sorte ihre Plätze, aber das sieht man nicht.

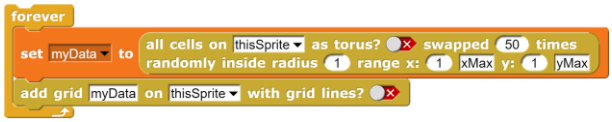
Im Gittermodell symbolisieren wir Wasserteilchen mit dem Wert für Blau (9) und Farbe mit dem Wert für Rot (4). Wir konfigurieren ein Sprite als *ImagePad* und initialisieren das Gitter mit den Dimensionen 400x400. Danach erhalten alle Gitterzellen den Wert 9 und dann die in der Mitte den Wert 4. Das Gitter kann nun dargestellt werden- mit einem roten „Klotz Farbe“ im Zentrum.



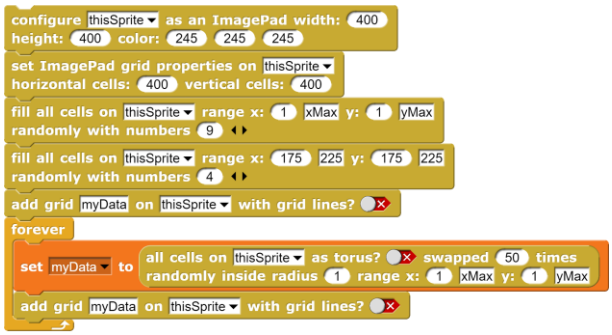


Danach setzt die Diffusion ein, indem die Werte der Gitterzellen zufällig mit Nachbarn getauscht werden. Für eine Zelle ist das ein einfacher Prozess, für unsere 160.000 Zellen aber schon etwas aufwändig. *SciSnap!* hat deshalb einen eigenen Block, der den Vorgang dieser „Wärmebewegung“ für alle Zellen n-mal ausführt:

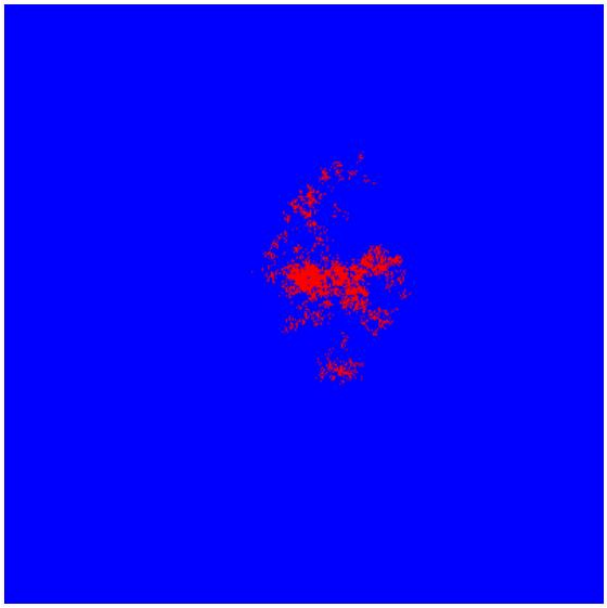
Das Ergebnis weisen wir einfach dem Datenbereich des Gitters *myData* zu und stellen das Ergebnis neu dar. Diese Operationen wiederholen wir immer wieder.



Insgesamt lautet unser Skript:

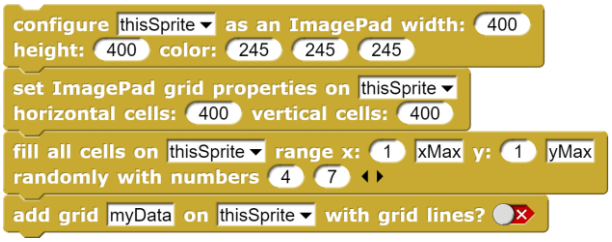


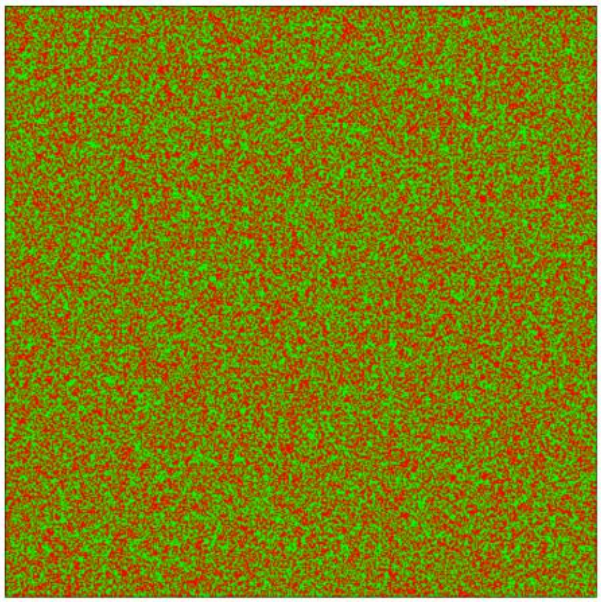
Und nach einigen Durchläufen sieht unser Farbfleck dann z. B. so aus:



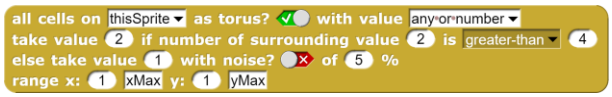
8.10 Ferromagnetismus als Gitterautomat

In zellulären Automaten treten oft Selbstorganisationsprozesse auf, die auf das Zusammenwirken benachbarter Zellen zurückzuführen sind. Wir wollen eine vereinfachte Version des *Ising-Modells* [22] des Ferromagnetismus bauen. Dazu stellen wir uns ein Gitter aus kleinen Elementarmagneten, z. B. Atomen mit einem magnetischen Moment vor, die zufällig nach oben oder unten ausgerichtet sind. Die Richtung kodieren wir mit den Farben Grün (7) und Rot (4). Der Anfang des Skripts ähnelt also dem vorangegangen sehr.



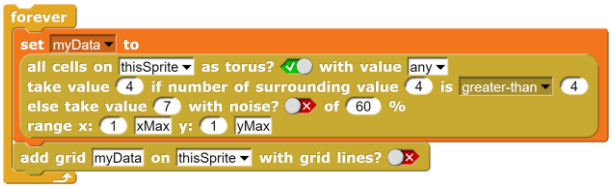


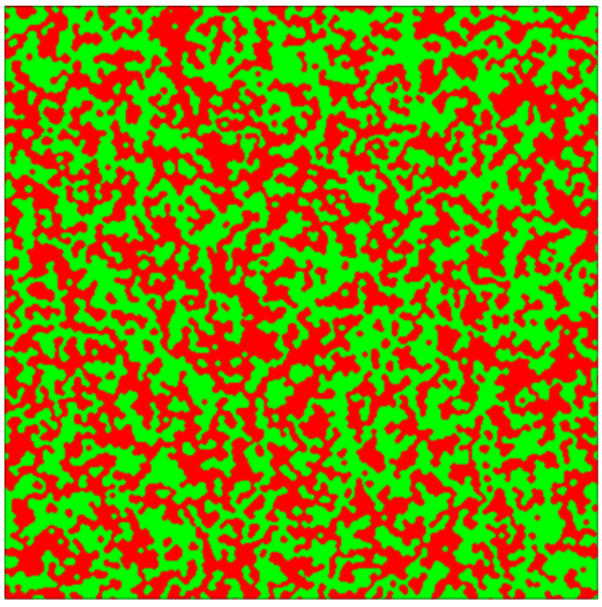
Jetzt nehmen wir an, dass Elementarmagnete Opportunisten sind: sie richten sich in ihrem Verhalten einfach nach der Mehrheit ihrer Nachbarn. Ist die Mehrheit der Elementarmagnete in der Nachbarschaft nach oben ausgerichtet, dann richtet sich der kleine Magnet auch nach oben aus, sonst nach unten. Weil solche Abfragen in Gitterautomaten häufig sind, gibt es einen eigenen *SciSnap!*-Block.

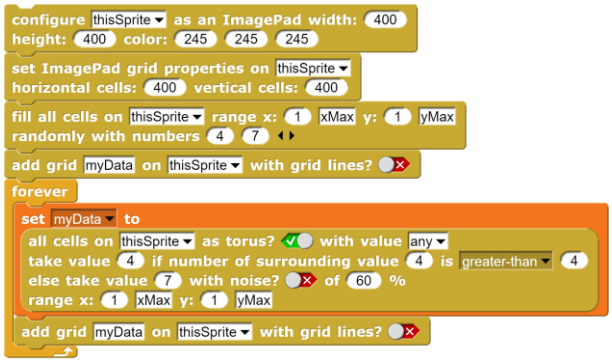


Als „noise“ bezeichnet man ein Rauschen, das den angegebenen Prozentsatz der Zellen zufällig „kippt“.

Damit lassen sich wie beim letzten Beispiel die neuen Zustände des Gitterautomaten wiederholt berechnen und anzeigen. Insgesamt erhalten wir das folgende Skript und nach einigen Durchgängen das angezeigt Ergebnis, in dem sich ineinander „verschlungene“ größere Bereiche ergeben, die sich in einem äußeren Magnetfeld ausrichten und dieses verstärken. Das Gitter verhält sich wie ein Ferromagnet. Durch starkes Rauschen, was physikalisch einer erhöhten Temperatur entspricht, wird die Magnetisierung gestört, bis sie bei hohen Werten zusammenbricht.







[22] https://de.wikipedia.org/wiki/Ising-Modell

8.11 Conways Game of Life

Ein berühmtes Beispiel für zelluläre Automaten ist das *Game of Life*[23] von *John Horton Conway* aus dem Jahr 1970(!!). Es besteht aus einem zweidimensionalen zellulären Automaten, der lebendige Zellen (schwarz) und tote Zellen (weiß) enthält. Das Spiel entwickelt sich nach drei Regeln:

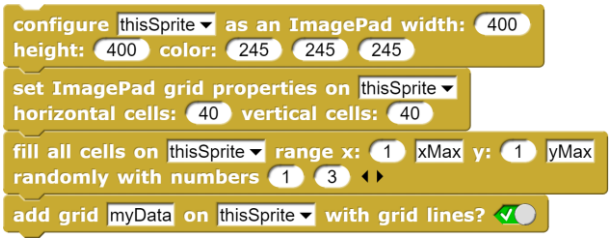
Regel 1: eine weiße Zelle mit genau drei schwarzen Nachbarn wird schwarz (lebendig).

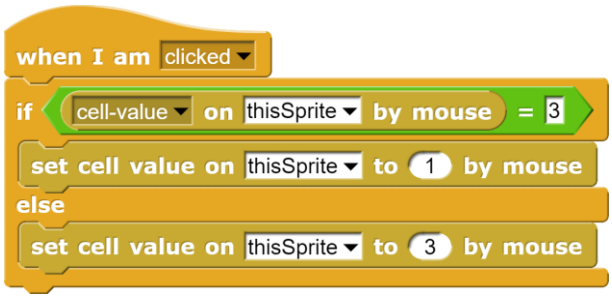
Regel 2: schwarze Zellen mit weniger als zwei schwarzen Nachbarn sterben an Einsamkeit, werden also weiß.

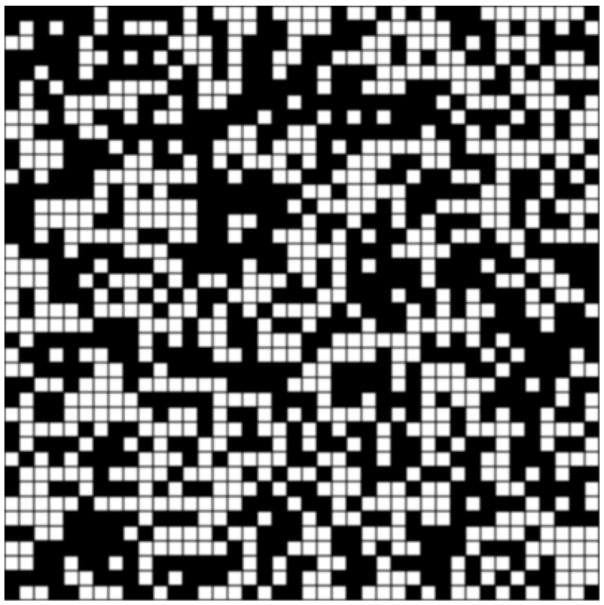
Regel3: schwarze Zellen mit mehr als drei schwarzen Nachbarn sterben an Überbevölkerung, werden also weiß.

Andere Zellen ändern sich nicht.

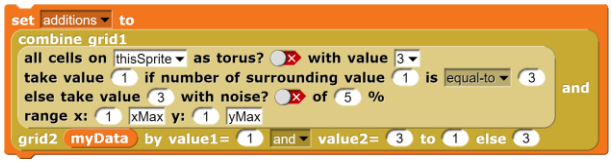
Mit den *SciSnap!*-Blöcken für Gitter ist es einfach, einen Conway-Automaten zu erzeugen. Statt ihn zufällig mit Werten zu füllen, könnten wir auch nur weiße Zellen erzeugen und danach mit der Maus Zellen verändern:



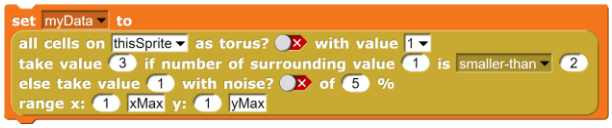


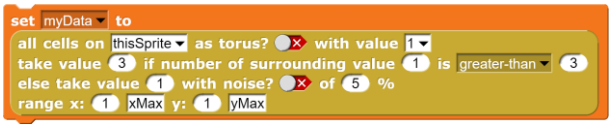


Da sich die drei Regeln erst auf die folgende Generation auswirken sollen, müssen wir die Ergebnisse teilweise zwischenspeichern. Wir erzeugen dafür eine Variable *additions*. Diese füllen wir mit den neu erzeugten Zellen. Dafür wenden wir die Regel 1 auf das Gitter an und ziehen die alten Werte ab:



Es folgen die „Robinson-Regel“ 2 und die Regel 3 für die „Lemminge“:

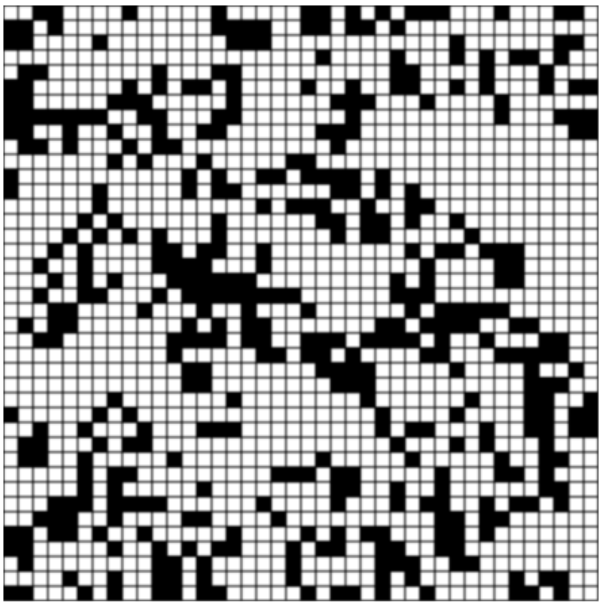


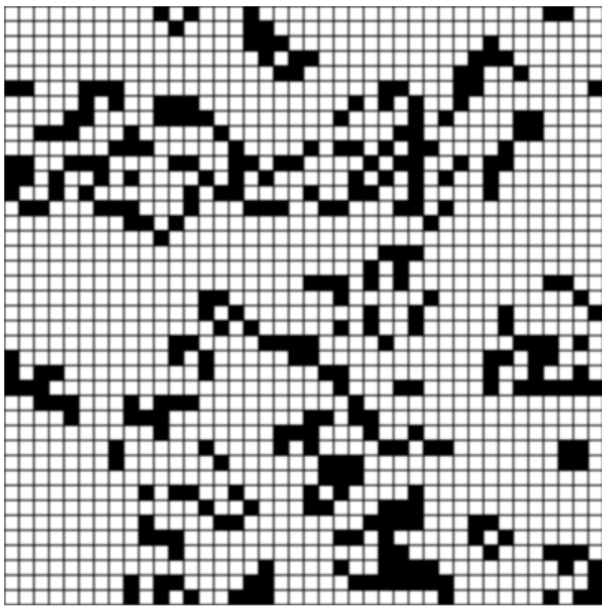


Nach den vielen Toten kommen die neuen Zellen zum Ergebnis hinzu:

[23] https://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens

Daraus bauen wir das Gesamtskript zusammen und lassen das Spiel laufen. Nach einigen Generationen bilden sich stabile Muster und Objekte, die sich bewegen. Erzeugen Sie mal mit der Maus Muster, von denen Sie sich Besonderes erwarten (oder suchen Sie solche im Netz). 😉

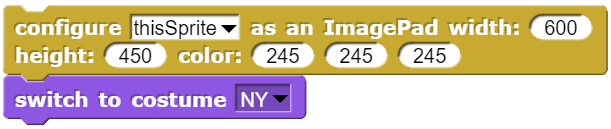




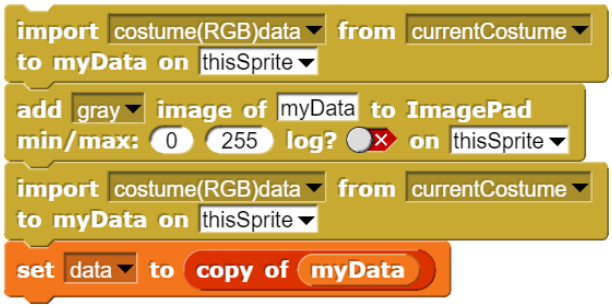
8.12 Ein zellulärer Automat als Weichzeichner

In manchen Fällen ist es wünschenswert, ein Bild im Ganzen oder in Teilen zu verändern, z. B. einen *Weichzeichner* darüber laufen zu lassen, der eine einstellbare Unschärfe erzeugt. Ein Grund könnte sein, Gesichter oder PKW-Kennzeichen unerkennbar zu machen. Wir probieren das mit unseren Gitteroperationen aus.

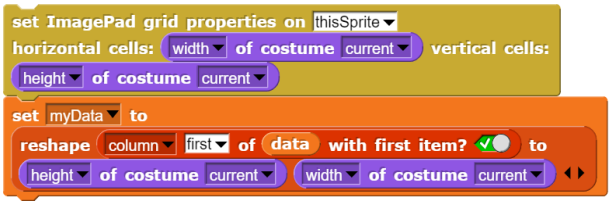
Zuerst einmal suchen wir ein schönes Bild aus, hier vom nächtlichen New York mit den Dimensionen 600x400, und stellen es als Sprite-Kostüm dar.



Da unser Gitter nur einfache Zahlen aufnimmt, verwandeln wir das Bild in ein Graustufenbild, verwenden dieses als neues Kostüm und importieren es zurück in *myData*. Das Ergebnis sichern wir in der Variablen *data*.



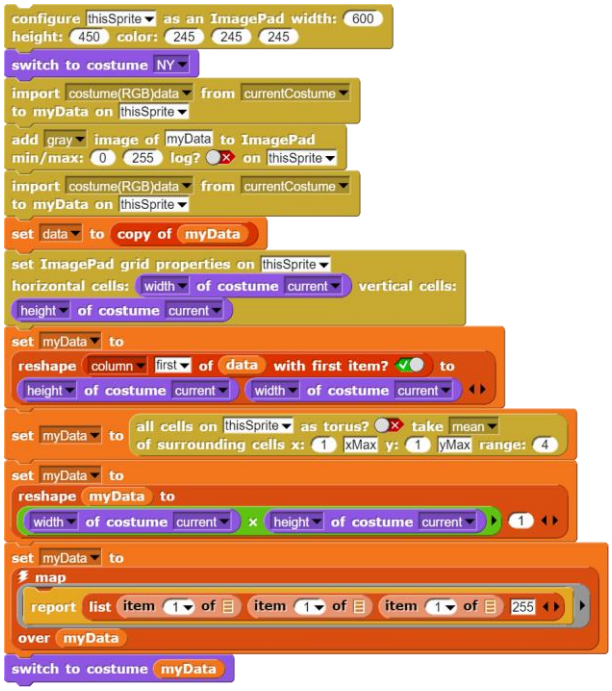
Jetzt wird es ernst. Wir erzeugen ein Gitter mit den Bilddimensionen, wählen die erste Spalte der gesicherten Bilddaten aus (die restlichen beiden Farbspalten sind ja identisch) und verwandeln diese zurück in eine Matrix – mit *reshape*. Das Ergebnis landet in *myData*. Damit sind geeignete Gitterwerte vorhanden.



Auf dieses Gitter können wir jetzt eine der Grid-Operationen anwenden. Wir entscheiden uns dafür, die Mittelwerte der umgebenden Zellwerte im Radius 4 als neue Zellwerte zu nehmen. Damit sollte etwas Unschärfe entstehen, ohne das Bild unerkennbar zu machen.

Aus den Gitterwerten bauen wir jetzt wieder ein Bild zusammen, indem wir das Gitter in eine einzelne Spalte zurückverwandeln (mit *reshape*), daran zwei identische Spalten und den Transparenzwert pro Zeile 255 anhängen. Das Ergebnis entspricht dann einer *Snap!*-Pixeldarstellung eines Kostüms. Wir stellen es deshalb direkt auf dem Sprite dar.

Das gesamte Skript lautet dann:



range 4, max

range 4, mean

range 10, mean

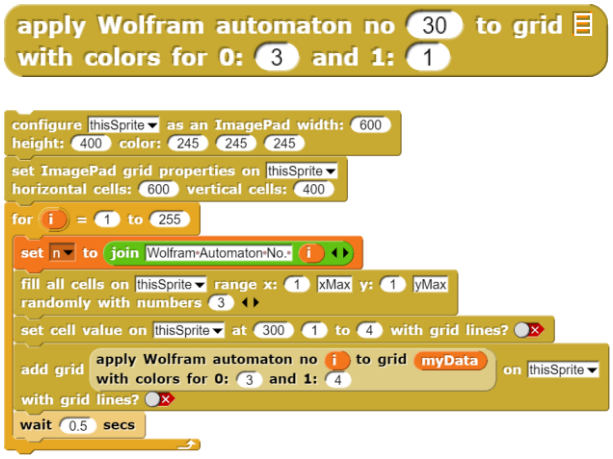
Aufgaben:

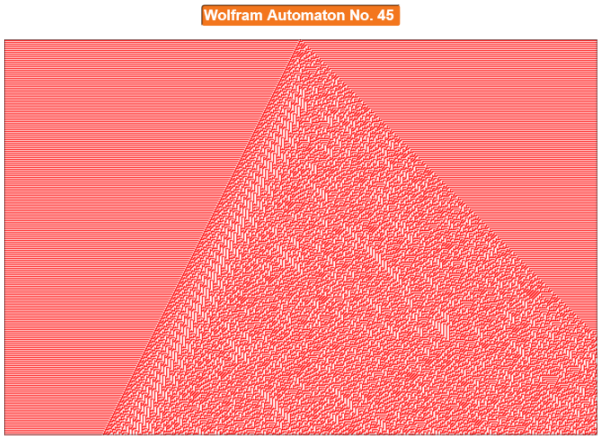
1. Skalieren Sie die Grauwerte auf den Bereich 0 bis 9 (den der neun Grid-Farben) und stellen Sie das Bild als Farbbild im Gitter dar.
2. Suchen Sie ein Bild mit Gesichtern. Machen Sie nur die Gesichter unkenntlich.
3. Suchen Sie ein Bild mit PKW-Kennzeichnen. Machen Sie nur die Nummernschilder unkenntlich.
4. Experimentieren Sie mal herum. Vielleicht wird es echte Kunst! 😉

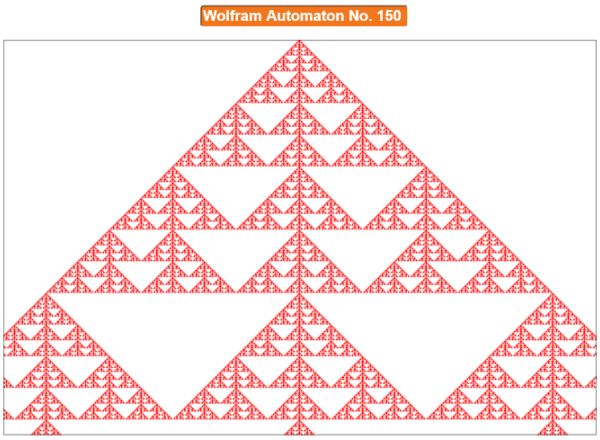
8.13 Lineare Wolfram-Automaten

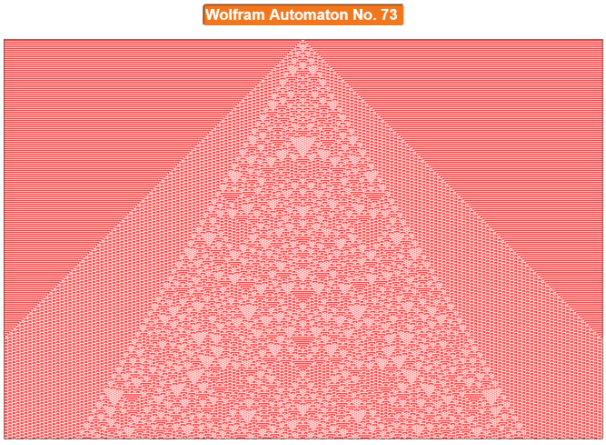
Eines der bekanntesten Beispiele aus dem Bereich der zellulären Automaten sind die linearen Automaten von *Stephen Wolfram*[24]. Ein linearer Gitterautomat ist eine Reihung von Zellen, die zwei unterschiedliche Werte annehmen können, die wir zu 0 und 1 wählen können. Wolfram betrachtet immer drei benachbarte Zellen, konstruiert aus den drei Zellwerten eine Dualzahl – die in diesem Fall nur zwischen 0 und 7 liegen kann – und ordnet dem Folgezustand der mittleren Zelle dieser Kombination eine Zahl zu, die entsprechend 0 oder 1 ist. Mit diesen Vorgaben gibt es 2^8=256 verschiedene lineare Automaten dieser Art. Die Nummer des Automaten gibt direkt seine Funktionalität an. Wolfram zeichnet untereinander die jeweilige Belegung der Gitterzellen. Die zeitliche Reihenfolge verläuft dann von oben nach unten und ergibt insgesamt ein zweidimensionales Gitter. Informieren Sie sich an anderer Stelle über die Details der Abläufe.

Da Wolfram-Automaten ziemlich rechenaufwändig (und so interessant sind 😉), gibt es dafür einen eigenen Block in *SciSnap!* Mit dessen Hilfe erzeugen wir schnell ein Skript, das nacheinander alle Wolfram-Automaten erzeugt und anzeigt. Als „Samen“ für den Automaten wählen wir jeweils eine einzelne 1 in der Mitte der ersten Zeile. Anbei einige Ergebnisse:









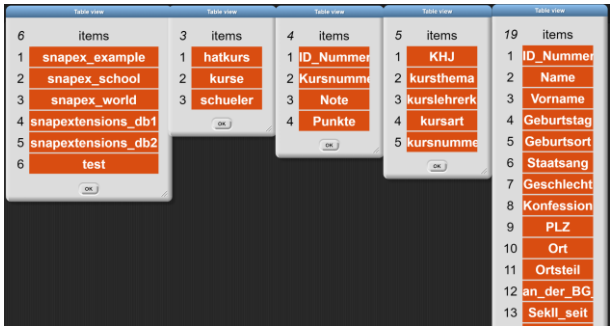
[24] [Wolfram]

9 SQL Beispiele

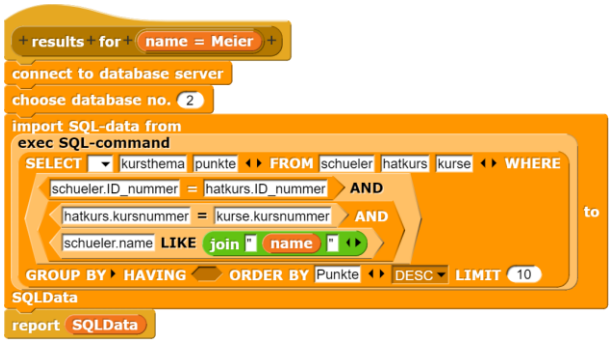
9.1 Umgang mit der SQL-Bibliothek

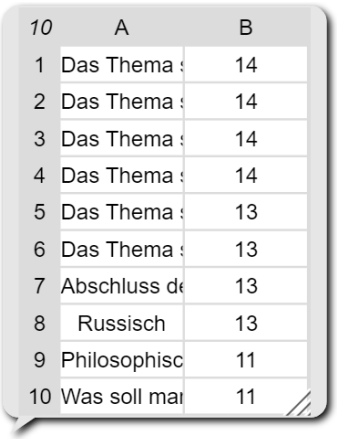
Mithilfe der *SQL tools* kann man entweder auf eine der Beispieldatenbanken oder andere Datenquellen zugreifen. Der Block *configure SQL* erzeugt eine globale Variable *SQLData* und nimmt einige Voreinstellungen vor. Der Block *connect to database server* verbindet mit dem Server der Beispieldatenbanken. Steht das Sprite *Hilberto* mit seinen Kostümen zur Verfügung, dann verwandelt sich das in ein Serversymbol und zeigt durch eine „Lampe“ an, ob die Verbindung steht. Will man andere Server benutzen, dann müssen in diesem Block die Server-Adresse und andere Einstellungen verändert werden. Mit *read databases* erhält man eine Liste der verfügbaren Datenbanken. Von diesen wählt man eine mit *choose database no. …* aus.

In der Praxis benötigt man dauernd die Details der Tabellen der benutzen Datenbank. Weist man nacheinander die Tabellen-Attribute einer Variablen zu und lässt diese mit *„open in dialog“* anzeigen, dann kann man die entsprechenden Tabellendaten geeignet im *SciSnap!*-Fenster platzieren und mit den Abfragen beginnen.



Beispiel: Die Kurstitel und Bewertung für alle Kurse eines Lernenden, absteigend sortiert nach der Note, werden gesucht.





Beispiel: Für statistische Zwecke soll die ***schueler***-Tabelle nach unterschiedlichen Kriterien durchsucht werden.

9.2 Eine einfache SQL-Abfrage

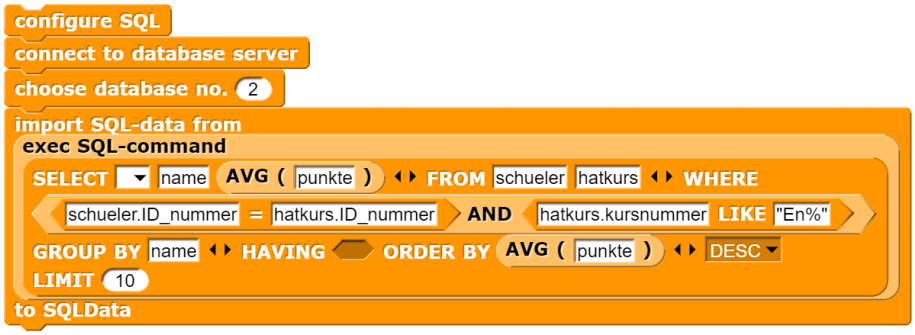
Wir wollen die Themen aller Englischkurse einer Schuldatenbank erfragen.

SQLData

22	A	B
1	Problems an	En 31
2	Problems an	En 32
3	Landeskund	en 31
4	Fantasy and	en 32
5	Fantasy and	en 33
6	Landscapes	En 41
7	Landscapes	En 42
8	Shakespear	en 41
9	Education in	en 42
10	Education in	en 43
11	The Short St	En 11

9.3 Eine komplexere SQL-Abfrage

Anfrage an eine Schuldatenbank: Suche der besten Ergebnisse in Englisch.



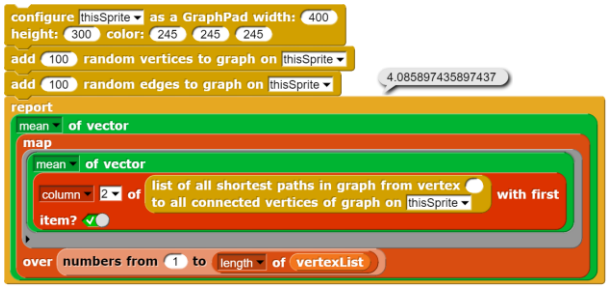
SQLData

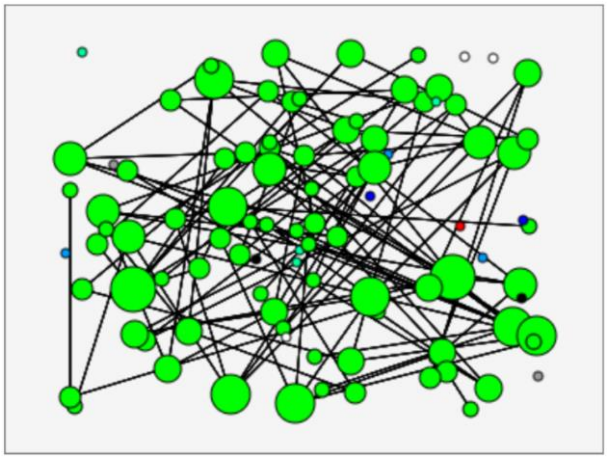
10	A	B
1	Rassin	12.5000
2	Schmitt	12.5000
3	Wuchtig	12.2500
4	Frugich	12.0000
5	Knusel	12.0000
6	Pfaffner	11.7500
7	Zinn	11.7500
8	Reinsberg	11.5000
9	Krahn	11.2500
10	Tohler	11.2500

10 Beispiele mit Graphen

10.1 Mittlerer Abstand in einem Random-Graph (Small Worlds)

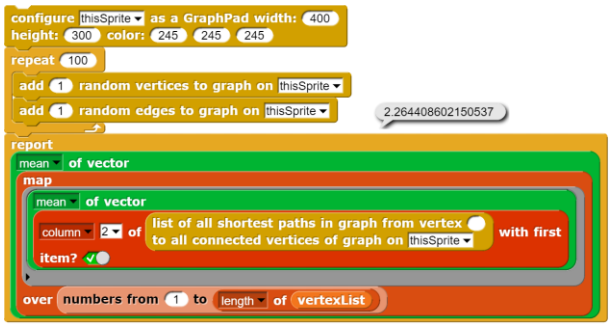
Wir berechnen den mittleren Abstand der Knoten in einem Graphen, bei dem zuerst alle Knoten und danach alle Kanten erzeugt wurden.

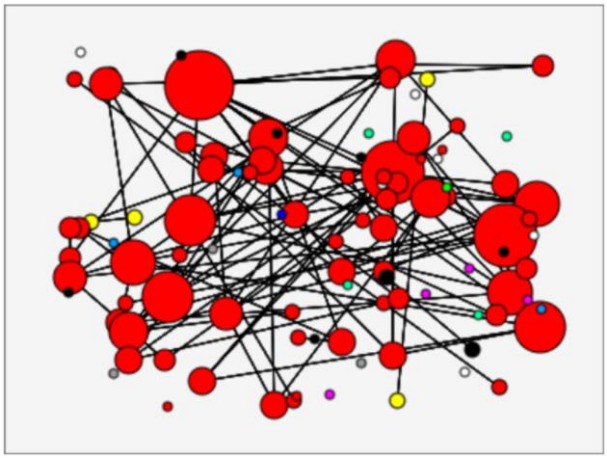




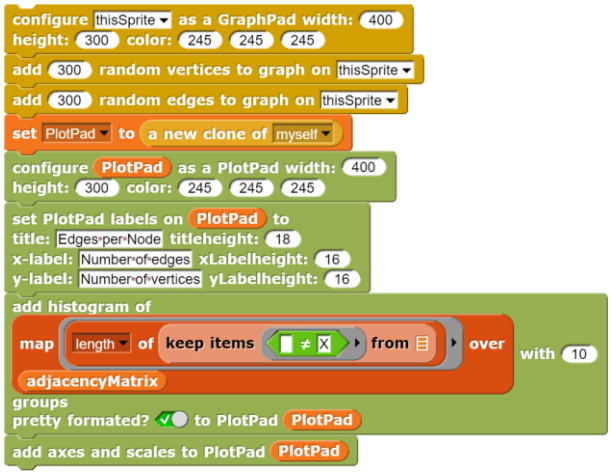
10.2 Mittlerer Abstand in einem Scalefree-Graph (Small Worlds)

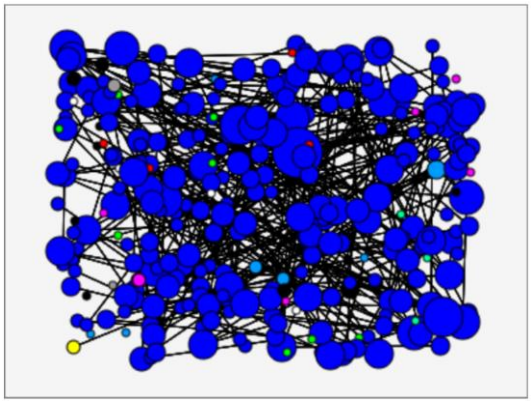
Wir berechnen den mittleren Abstand der Knoten in einem Graphen, bei dem abwechselnd Knoten und Kanten erzeugt wurden.

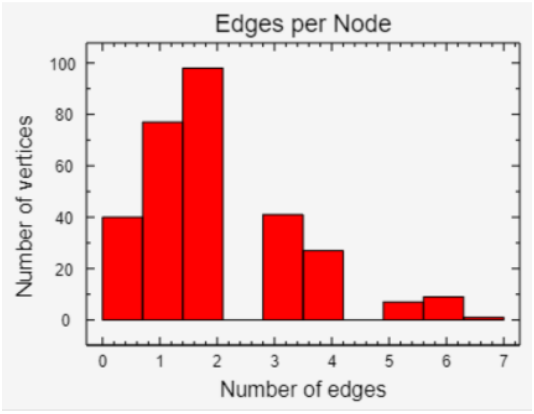




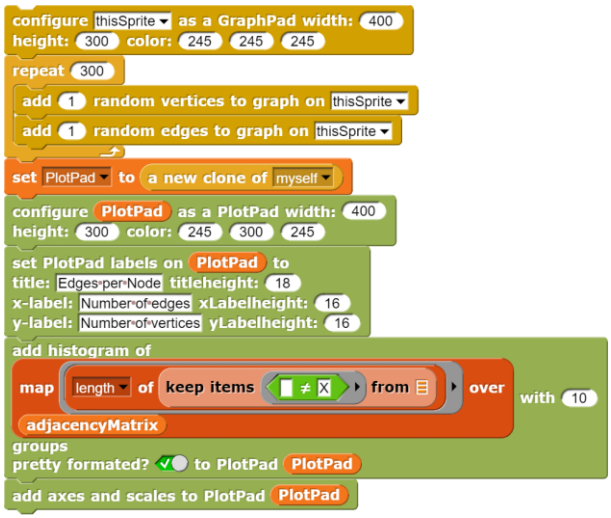
10.3 Histogramm „Kanten pro Knoten“ in einem Random-Graph

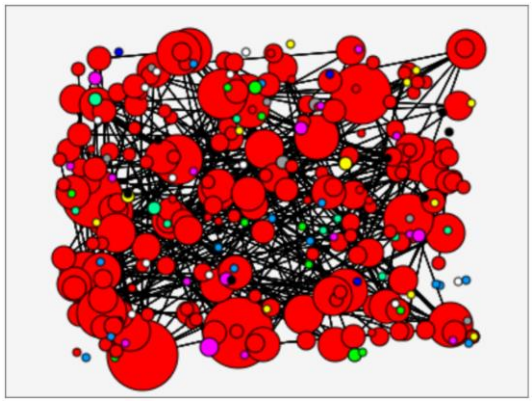


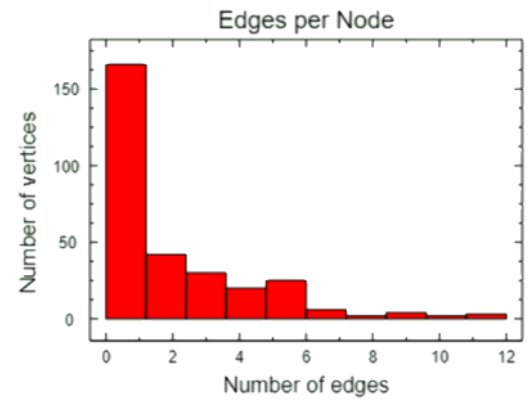




10.4 Histogramm „Kanten pro Knoten“ in einem Scalefree-Graph

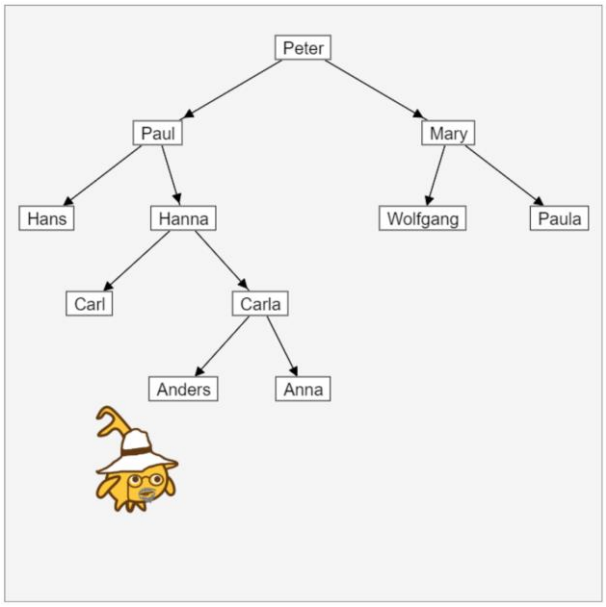






10.5 Breiten- und Tiefensuche im Stammbaum

Wir erzeugen einen Stammbaum als gerichteten Graphen ohne Kantengewichte einfach durch Anordnung und Verbindung der entsprechenden Knoten. Das ist umständlich, aber einfach. Eben etwas Gefummel.




```
+ new + family + tree +
configure FamilyTree as a GraphPad width: 700 height: 700 color: 245 245 245
set GraphPad vertex properties minSize: 10 growing? showsContent? on FamilyTree
set GraphPad edge properties lineWidth: 1 color: 0 0 0 directed? weighted? showsWeight? on FamilyTree
new vertex at 0 300 content: Peter on graph of FamilyTree
new vertex at -170 200 content: Paul on graph of FamilyTree
add edge from vertex 1 to vertex 2 to graph on FamilyTree
new vertex at 170 200 content: Mary on graph of FamilyTree
add edge from vertex 1 to vertex 3 to graph on FamilyTree
new vertex at -300 100 content: Hans on graph of FamilyTree
add edge from vertex 2 to vertex 4 to graph on FamilyTree
new vertex at -140 100 content: Hanna on graph of FamilyTree
add edge from vertex 2 to vertex 5 to graph on FamilyTree
new vertex at 140 100 content: Wolfgang on graph of FamilyTree
add edge from vertex 3 to vertex 6 to graph on FamilyTree
new vertex at 300 100 content: Paula on graph of FamilyTree
add edge from vertex 3 to vertex 7 to graph on FamilyTree
new vertex at -250 0 content: Carl on graph of FamilyTree
add edge from vertex 5 to vertex 8 to graph on FamilyTree
new vertex at -50 0 content: Carla on graph of FamilyTree
add edge from vertex 5 to vertex 9 to graph on FamilyTree
new vertex at -140 -100 content: Anders on graph of FamilyTree
add edge from vertex 9 to vertex 10 to graph on FamilyTree
new vertex at 0 -100 content: Anna on graph of FamilyTree
add edge from vertex 9 to vertex 11 to graph on FamilyTree
```

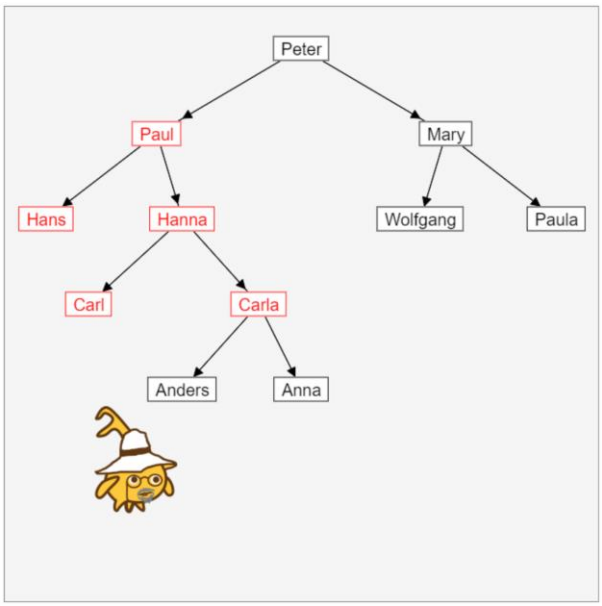
In diesem Baum können wir nun alle möglichen Aufgaben lösen. Wir könnten z. B. fragen, ob eine Person Vorfahr von einer anderen ist. Dazu benötigen wir die Nummer des Startknotens, die wir mit `vertexnumber of Peter in graph of FamilyTree` ermitteln. Den Rest erledigt entweder der Block zur Breitensuche oder der zur Tiefensuche.

```
+ is + parent = Carla + ancestor + of + child = Mary + ? +
script variables result
set result to
breadth first search of content parent
starting at vertex vertexnumber of child in graph of FamilyTree of graph on FamilyTree
draw graph on FamilyTree
report result
```

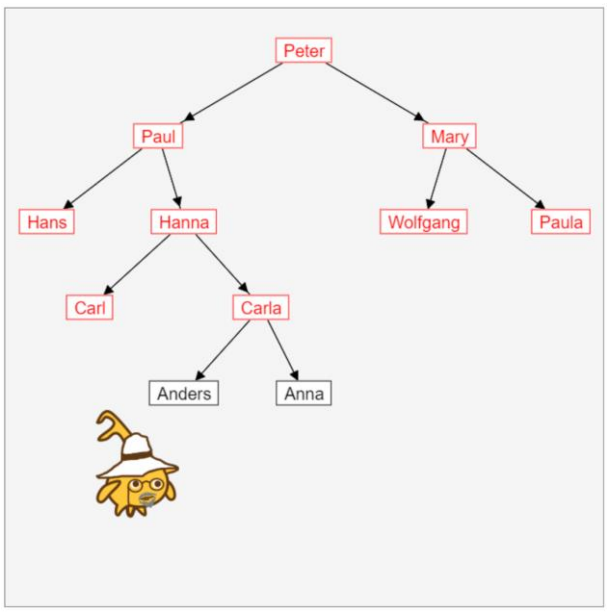
```
is Carla ancestor of Peter ?
```

Bei der Breiten- und Tiefensuche werden die besuchten Knoten markiert und bei einem erneuten Zeichnen des Graphen rot dargestellt. Das zeigt die Unterschiede der beiden Suchverfahren ziemlich deutlich.

Tiefensuche



Breitensuche



Aufgaben:

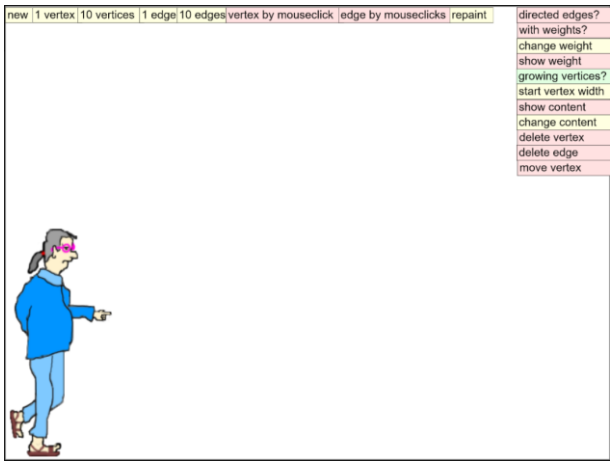
1. Ermitteln Sie, über wie viele Generationen die Verwandtschaft besteht, wenn sie denn besteht.
2. Stellen Sie Listen von Verwandten einer Person auf, z. B. von Eltern, Großeltern, Tanten, Schwippschwagern 😉, …
3. a: Entwickeln Sie ein Skript zur Erstellung eines Entscheidungsbaums, z. B. zur Klassifikation von Tieren oder Pflanzen: Es wird jeweils gefragt, ob es sich um ein bestimmtes Exemplar handelt oder, falls nicht, durch welche Frage man das genannte von dem aktuellen unterscheiden kann („Hat es vier Beine?“). Es wird entweder das Exemplar oder die Frage in den Baum eingetragen.

   b: Lassen Sie ihr Ergebnis von anderen testen. Versuchen Sie abzuschätzen, ab welcher Datenmenge im Baum so ein Programm sinnvoll einsetzbar wäre.

   c: Entscheidungsbäume spielen eine Rolle bei den bestimmten Anwendungen des maschinellen Lernens (*Decision Tree Classification*). Informieren Sie sich über das Verfahren und seine Einsatzgebiete.

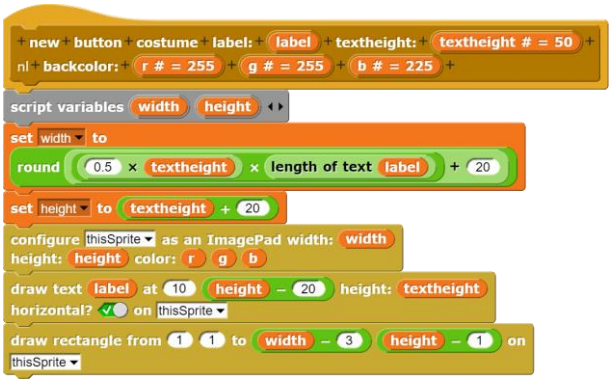
10.6 Ein Graph-Labor

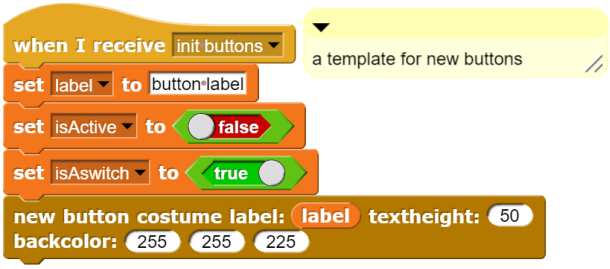
Um mit Graphen einfach experimentieren zu können, bauen wir uns ein kleines Graph-Labor zusammen, das es gestattet, die Parameter des Graphen – also *gewichtet*, *gerichtet*, … zu sein – direkt zu setzen. Das Labor benutzt die Bühne als *GraphPad*. Gesteuert wird alles von *Gundolf de Jong*, dem begabten jungen Mathematiker.



Wir wollen ein paar Knöpfe einfügen, die entweder als einfacher Knopf oder als Schalter zu benutzen sind. Damit es nicht zu kompliziert wird, bauen wir eine einfache Vorlage. Jeder Knopf hat eine Aufschrift, kann aktiviert sein (oder nicht) und kann als Schalter dienen (oder nicht). Nach diesen Angaben kann ein Kostüm der richtigen Größe erzeugt werden, das der Button, als *ImagePad* konfiguriert, anzeigt. Bei den Knöpfen sollte die Möglichkeit abgeschaltet werden, sie zu verschieben (*draggable*).

Die Aktivität des Knopfes wird weitgehend durch Aufrufe der *GraphPad*-Blöcke realisiert. Diese werden einfach an das entsprechende Skript angehängt. Bei Mausklicks auf die Bühne reagiert diese dann auf die Voreinstellungen.

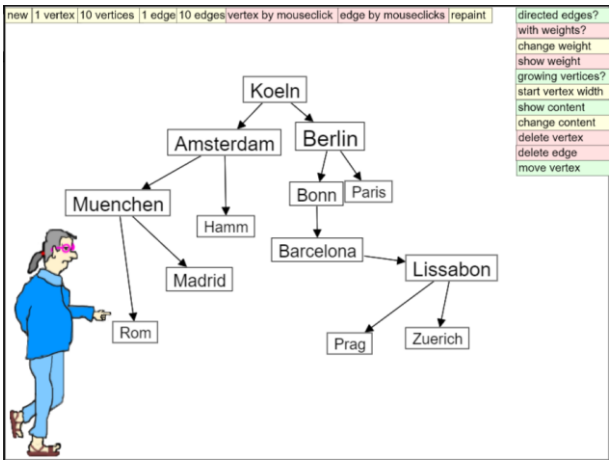




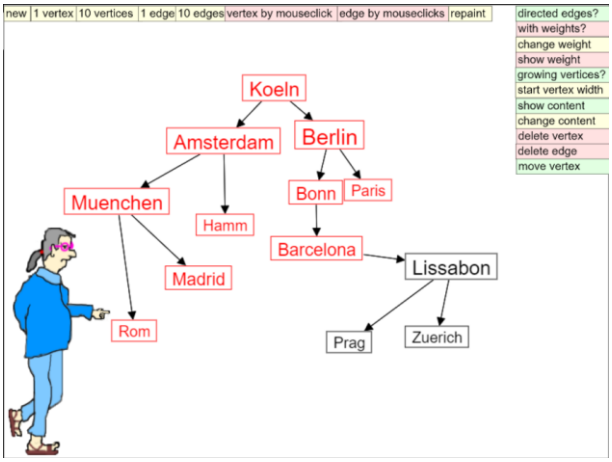
Ist z. B. der Knopf zum Einfügen von Knoten an der Stelle eines Mausklicks aktiviert, dann kann die Bühne das Entsprechende veranlassen.

10.7 Suchen im Baum mit dem Graph-Labor

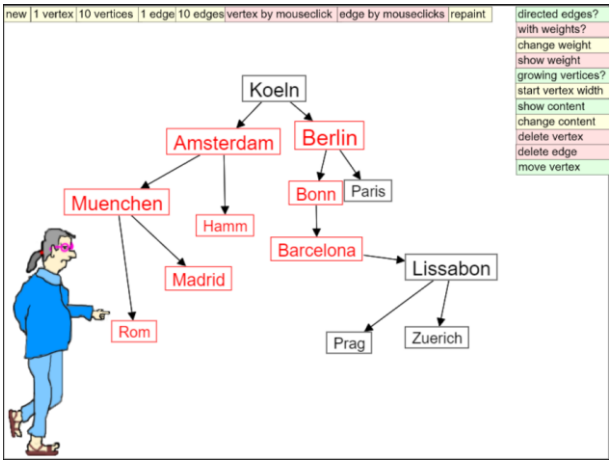
Wir stellen die Parameter für einen Baum ein: gerichtete Kanten, Start-Knotengröße 10. Dann positionieren wir einige Knoten mithilfe der Maus (*vertex by mouseclick*), geben deren Inhalte ein (*change content*), die nicht geordnet sind, und ordnen die Knoten anschließend mit der Maus so an, dass die Kanten sichtbar und die Inhalte lesbar sind (*move vertex*).



In diesem Baum suchen wir ausgehend von der Wurzel den Knoten mit dem Inhalt Barcelona, einmal per Breitensuche, dann per Tiefensuche. Dazu rufen wir den entsprechenden Block auf und lassen per *repaint* den Baum neu zeichnen. Die beim Durchlaufen markierten Knoten werden dann in Rot angezeigt. Mithilfe des Blocks *remove all markers of Graph* können sie wieder entfernt werden.



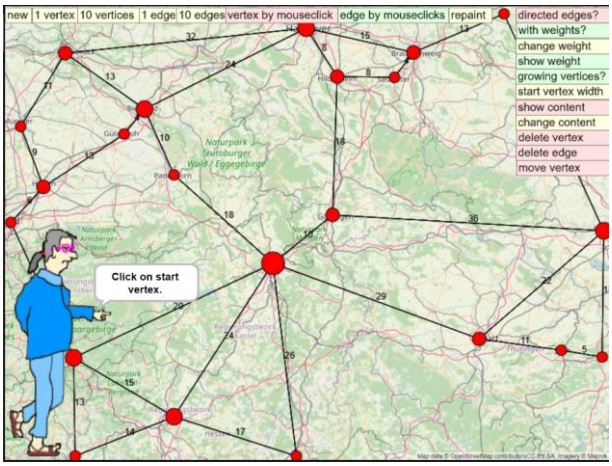
Dasselbe mit Tiefensuche:

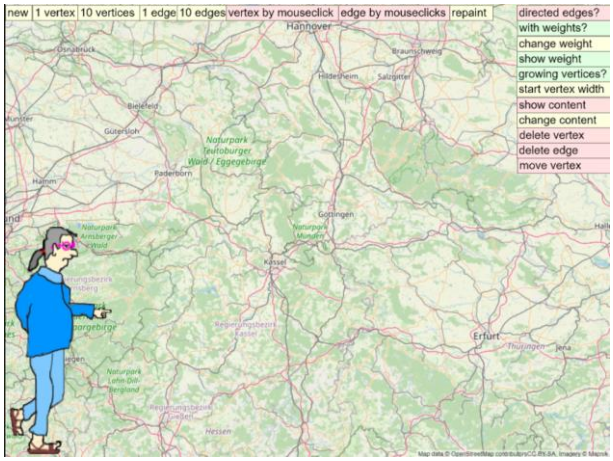


10.8 Das Graph-Labor mit World Map Library

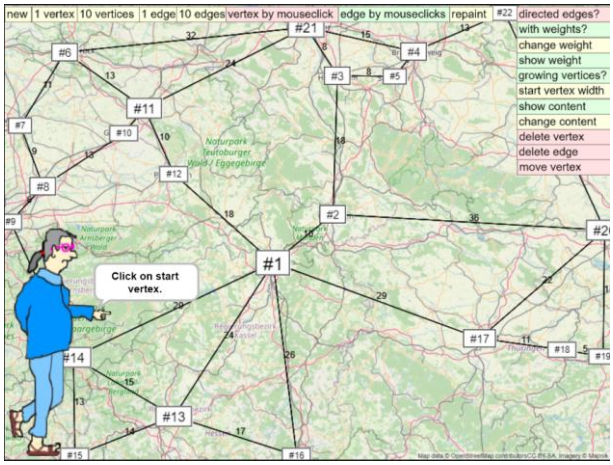
Wir laden die Word Map Library, stellen den aktuellen Ort und einen geeigneten Zoomfaktor ein. Darin wollen wir den Graph der näheren Umgebung erzeugen, in dem wir dann die günstigsten Wege zwischen den Orten bestimmen. Der Graph soll nicht gerichtet sein, aber die relativen Entfernungen grob anzeigen. Wir erhalten das folgende Bild.

In diesem klicken wir zuerst die Orte an, die für unser Verkehrsnetz relevant sind (*vertex by mouseclick*), vor allem die Städte. Danach erzeugen wir die gewünschten Kanten (*edge by mouseclick*).





Wobei wir natürlich auch die Knotennummern anzeigen können (*show content*).



Mithilfe des Graphen können jetzt diverse Fragen beantwortet werden:

- Wie weit ist es (in unseren Einheiten) von Braunschweig (#4) nach Siegen (#14)?

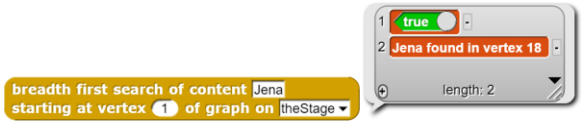
- Wie weit liegt von Hannover (#21) von Erfurt (#17) entfernt?

- Nachdem wir mindestens einen Ortsnamen eingegeben haben

  können wir eine Tiefensuche von Münster (#7) aus starten.

- Wohin kann man von Jena (#18) aus fahren?

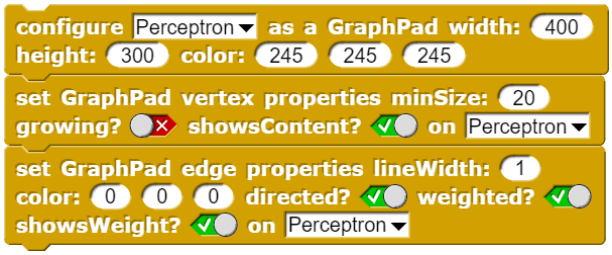
  

22	A	B
1	1	40
2	2	50
3	3	68
4	4	67
5	5	71
6	6	81
7	7	92
8	8	85
9	9	89
10	10	72
11	11	68
12	12	58
13	13	64
14	14	69
15	15	78
16	16	66
17	17	11
18	18	0
19	19	5
20	20	22
21	21	76
22	22	54

11 Beispiele zum Maschinellen Lernen

11.1 Ein einfaches Perzeptron als Graph

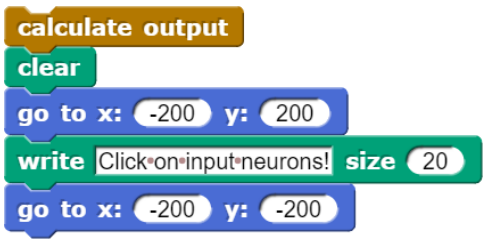
Wir wollen *Hilberto* bitten, ein *GraphPad* zu benutzen, um die Funktionsweise eines einfachen Perzeptrons zu veranschaulichen. Dazu soll er drei Eingabeknoten links im Bild platzieren. in der Mitte sitzt das eigentliche Perzeptron mit einer Sprungfunktion, das entweder +1 oder -1 an das Ausgabeneuron rechts im Bild übermittelt. Insgesamt handelt es sich also um einen *gerichteten Graphen* mit *Kantengewichten*. Den richtet *Hilberto* schnell ein: er erzeugt ein neues Sprite namens *Perceptron* und konfiguriert es richtig.



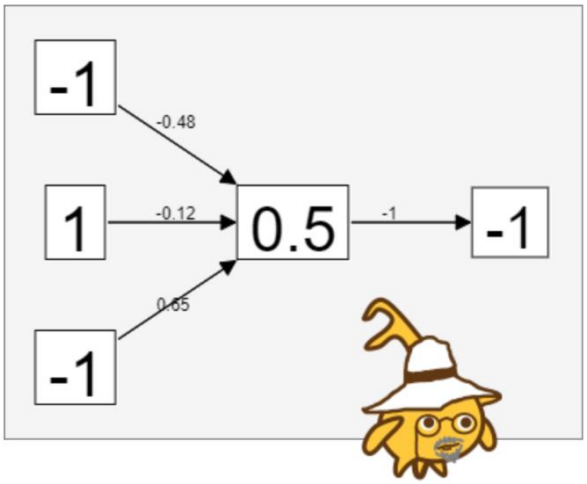
Danach fügt er angegebenen Knoten des Perzeptronnetzes hinzu:

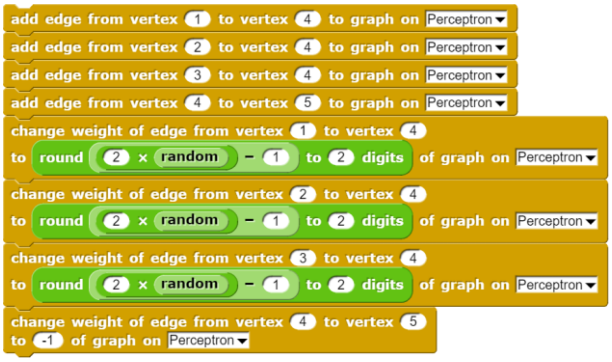
Es fehlen nur noch die Kanten, zuerst mit zufälligen Gewichten:

Hilberto fügt die Blöcke zu einem Skript zusammen und beschriftet das Ganze, und natürlich sorgt er für eine konsistente Situation, indem er das Perzeptron einmal durchrechnen lässt.




```
Click on input neurons!
```

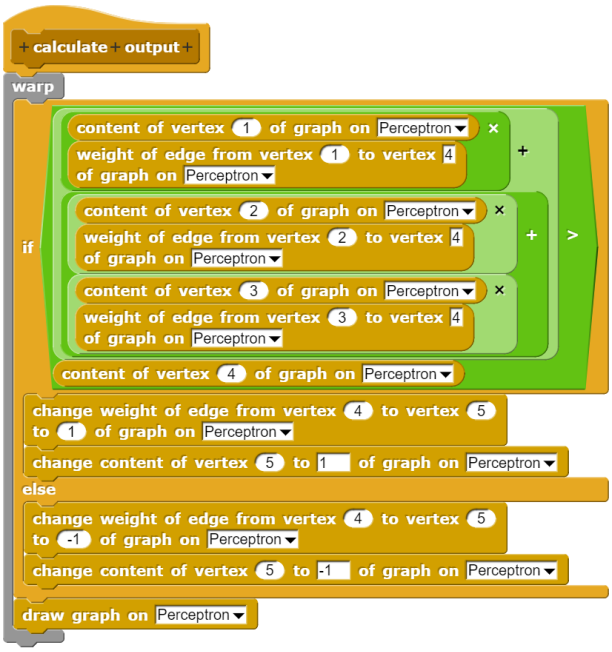




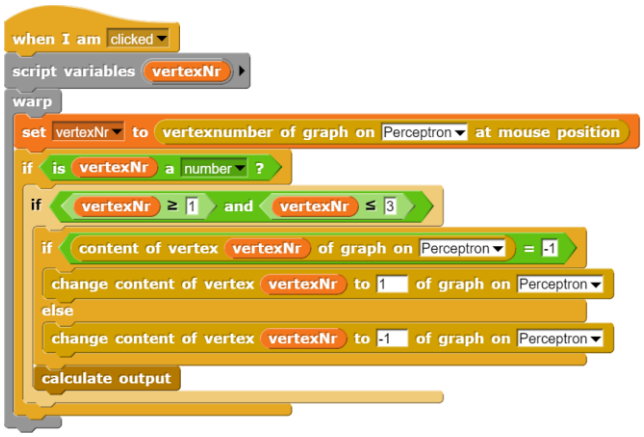
Die Ausgabe des Netzes wird bestimmt, indem die Werte der Eingabeneuronen mit den entsprechenden Kantengewichten multipliziert und die Ergebnisse addiert werden. Das Resultat wird dann mit dem Wert der Sprungfunktion in Neuron 4 verglichen. Je nach Ergebnis wird der Wert der letzten Kante und der Wert des Ausgabeneurons gesetzt.

Allerdings soll das Perzeptron auch noch arbeiten, und zwar auf folgende Weise: wenn ein Eingabe-Neuron, also ein Knoten auf der linken Seite, angeklickt wird, dann soll er seinen Wert ändern. Wir rechnen dazu die Mauskoordinaten beim Klick in Sprite-Koordinaten um und fragen anschließend nach der Knotennummer. Ist es eine der drei Eingabeneuronen, dann ändern wir deren Wert – und lassen neu rechnen.

Fertig.



Hilberto ist selbst erstaunt, dass das so einfach geht. 😉



Aufgaben:

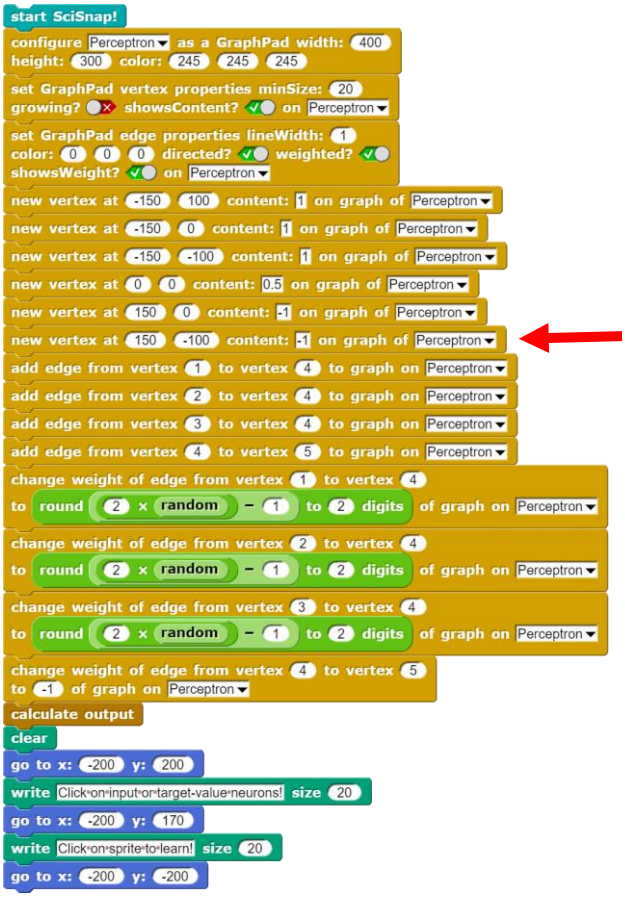
1. Ergänzen Sie eine Möglichkeit, den Wert der Sprungfunktion des zentralen Neurons zu ändern.
2. Ergänzen Sie eine Möglichkeit, die Kantengewichte der drei Eingabeneuronen zum zentralen Neuron zu ändern.
3. Ändern Sie die Kantengewichte und/oder die Sprungfunktion so, dass das Perzeptron als UND arbeitet.
4. Ändern Sie die Kantengewichte und/oder die Sprungfunktion so, dass das Perzeptron als ODER arbeitet.
5. Ändern Sie die Kantengewichte und/oder die Sprungfunktion so, dass das Perzeptron als NAND arbeitet.
6. Ändern Sie die Kantengewichte und/oder die Sprungfunktion so, dass das Perzeptron als NOR arbeitet.
7. Kann das Perzeptron auch als XOR arbeiten? Probieren Sie es aus!
8. Suchen Sie in der Literatur Gründe, weshalb sich einige Schaltungen gut durch Perzeptrons realisieren lassen, und andere nicht.

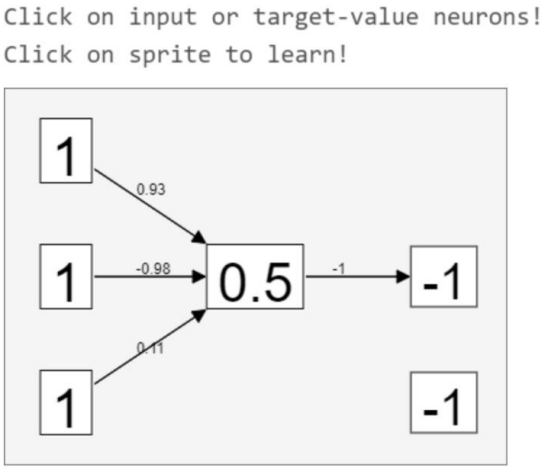
11.2 Ein einfaches lernendes Perzeptron

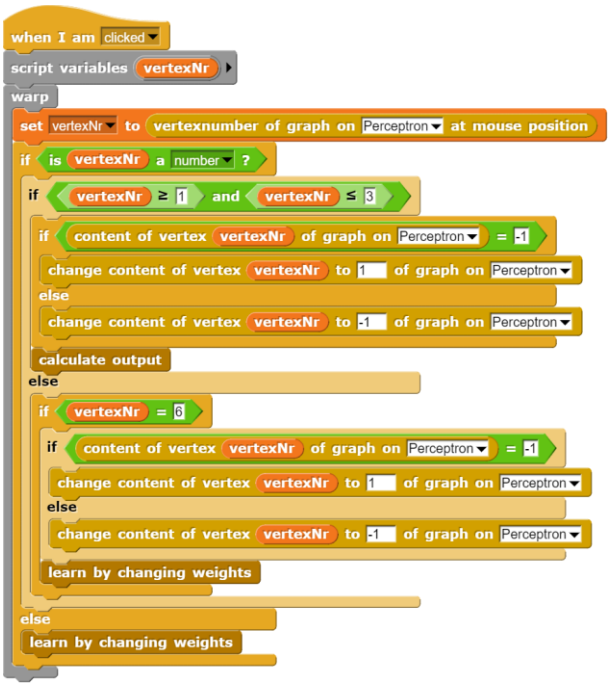
Wir ändern jetzt die Konfiguration ein wenig, um dem Perzeptron das Lernen beizubringen. Dazu führen wir einen weiten Knoten, einen „Zielknoten“, unter dem Ausgabeknoten ein, der „die richtigen“ Ergebnisse anzeigt, die wiederum durch Anklicken geändert werden können. Zeigt sich ein Unterschied zwischen den Werten von Ausgabe- und Zielknoten, dann werden die Gewichte solange geändert, bis sich das „richtige“ Ergebnis ergibt.

Was ist zu ändern?

Bei der Erzeugung des Netzes muss nur der neue Knoten eingefügt werden.





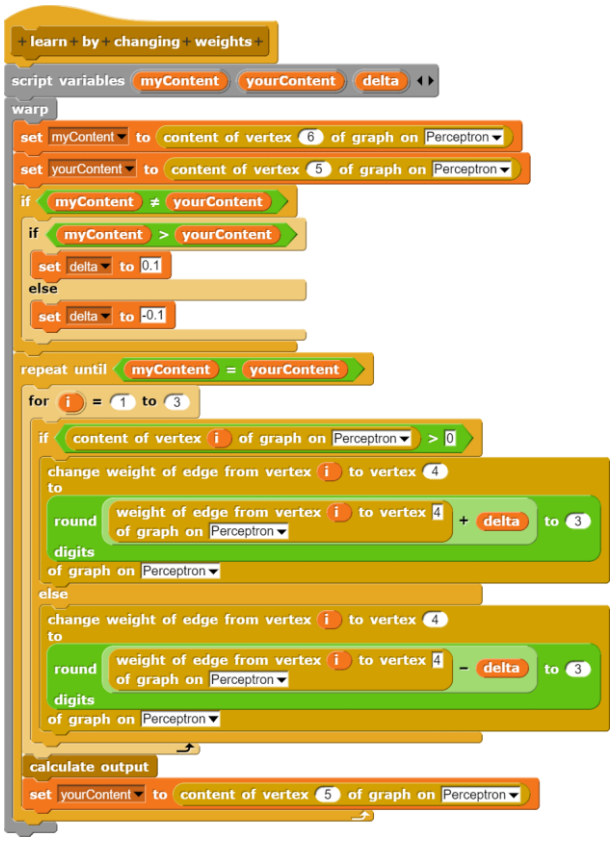


Und beim Klick-Ereignis auf das *GraphPad* muss nachgesehen werden, ob ein Knoten oder das Pad getroffen wurde. Im zweiten Fall startet das Lernen.

Und wie wird gelernt?

Wenn sich die Werte der beiden Neuronen rechts im Bild unterscheiden, dann ändern wir die Gewichte an den Kanten der Eingabeneuronen solange, bis sich das richtige Ergebnis ergibt.

Genauer: Wir geben den drei Eingabeneuronen jeweils einen Wert. Danach stellen wir den gewünschten Wert am Zielneuron ebenfalls durch Anklicken ein. Zuletzt klicken wir irgendwo auf das GraphPad und sehen ihm beim Lernen zu.

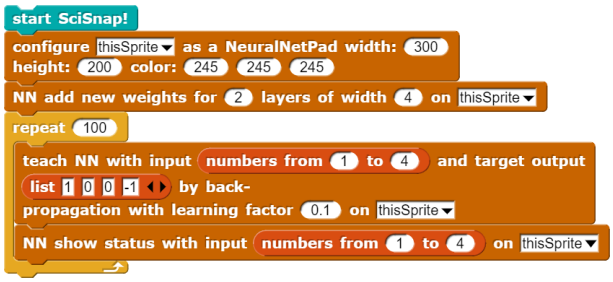


Aufgaben:

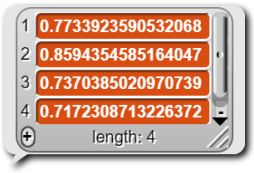
1. Ergänzen Sie eine Möglichkeit, das Lernen durch Änderungen am Wert der Sprungfunktion im inneren Neuron zu realisieren. Klappt das auch immer?
3. Trainieren Sie das Netz so, dass das Perzeptron als UND arbeitet.
4. Trainieren Sie das Netz so, dass das Perzeptron als ODER arbeitet.
5. Trainieren Sie das Netz so, dass das Perzeptron als NAND arbeitet.
6. Trainieren Sie das Netz so, dass das Perzeptron als NOR arbeitet.
7. Kann das Perzeptron auch als XOR arbeiten? Probieren Sie es aus!

11.3 Training eines Neuronalen Netzes

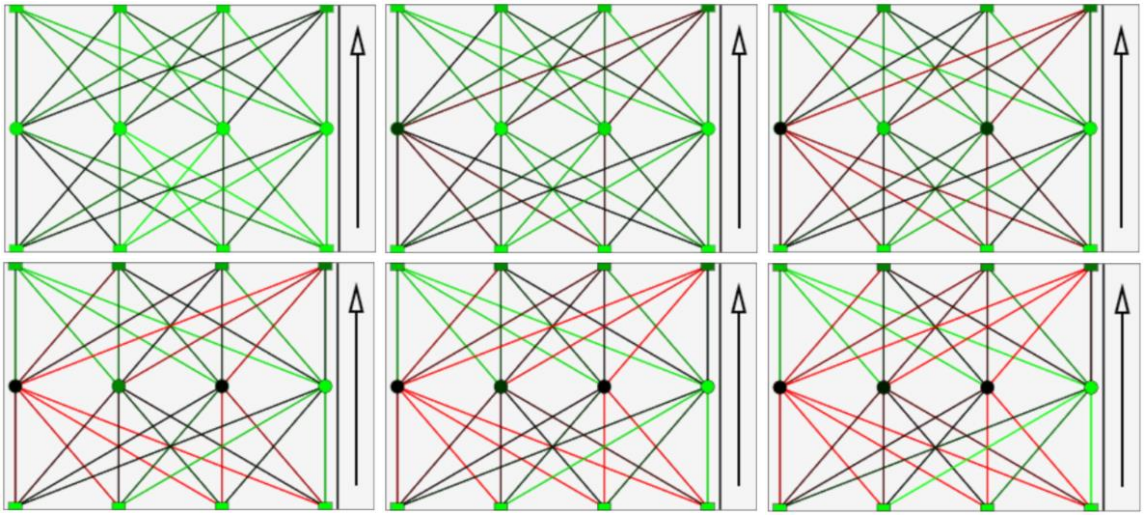
Ein Neuronales Netz des Breite 4 mit zwei Schichten (plus Eingabeschicht) wird erzeugt und darauf trainiert, bei Eingabe des Vektors aus den Zahlen 1 bis 4 als Ausgabe links eine 1 und rechts eine -1, dazwischen Nullen, zu liefern. Zu beachten ist, dass nicht die exakten Ausgabewerte, sondern links der größte und rechts der kleinste geliefert werden müssen.



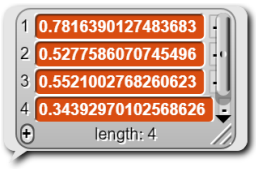
Die Ausgabe des Netzes vor dem Training:



Trainingszustände nach jeweils 20 Trainingsschritten:



Die Ausgabe des Netzes nach dem Training:



Die Positionen des größten bzw. kleinsten Werts der Ausgabe können direkt mithilfe der *SciSnap!MathTools* bestimmt werden:

11.4 Verkehrszeichenerkennung mit einem Neuronalen Netz von Perzeptren

„Tiefe“ Neuronale Netze prägen die Diskussion über die aktuelle „künstliche Intelligenz“. Dabei handelt es sich meist um „fully connected“ Netze aus mehreren Perzeptren-Schichten. „Fully connected“ bedeutet, dass alle Neuronen einer Schicht mit allen der nächsten Schicht verbunden sind. Jede Verbindung ist mit einem Gewicht versehen, aus dem sich sein Einfluss auf das verbundene Perzeptron ergibt – aber das lesen Sie besser woanders nach.

Betrachten wir dazu einmal ein Netz aus drei Lagen, das als Eingabe die Pixel eines aktuellen 20 M-Pixel-Fotos erhält, also $2x10^7$ Pixel. Die Eingabeschicht besteht aus $3x2x10^7$ MB Zahlenwerten zwischen 0 und 255 (wenn wir das Transparenz-Byte weglassen). Zur nächsten Schicht gibt es dann $(6x10^7)^2=3,6x10^{15}$ Verbindungen – und das dann noch zweimal. Insgesamt wären $3x3,6x10^{15}$, also etwa 10^{16} Gewichte zu bestimmen – eine für „normale“ Rechner völlig utopische Aufgabe. Wir werden uns also auf etwas kleinere Neuronale Netze beschränken müssen.

Eine Möglichkeit, Perzeptron-Netze zu trainieren, besteht darin, ihnen Eingabevektoren zu präsentieren und die gewünschte Ausgabe gleich dazu. Das Netz berechnet dann die Ausgabe, die sich aus den vorhandenen, anfangs zufällig gewählten Gewichten ergibt, und bestimmt die Differenz zum vorgegebenen Ergebnis. Von der letzten Ergebnisschicht ausgehend korrigiert es dann die Gewichte „rückwärtsgehend“ so, dass seine Ausgabe „etwas besser“ zum vorgegebenen Ergebnis passt. Das Verfahren nennt sich *Backpropagation*. Auch hierzu sollten Sie sich an anderer Stelle informieren. Aus vielen solcher Korrekturen ergibt sich das trainierte Netz. „Lernen“ bedeutet also, anhand vieler Beispiele die Parameter (die Gewichte) anzupassen. Mithilfe dieser Parameter bestimmt das Netz aus dem Eingabevektor einen Ausgabevektor: es berechnet einen Funktionswert. Unser *NeuralNetPad* kann solche Perzeptron-Netze simulieren und trainieren.

Die Gewichte bilden insgesamt einen *Tensor* mit *m* Schichten, die aus *nxn-Matrizen* bestehen. *NeuralNetPads* sollten deshalb die lineare Algebra beherrschen. Neu ist nur die *Softmax*-Funktion 

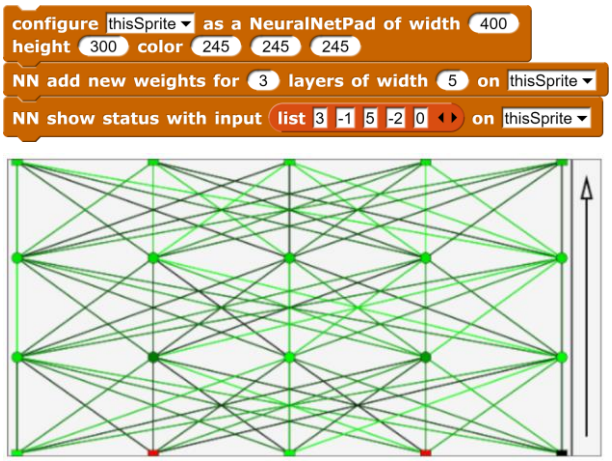
, mit der man z. B. Eingabevektoren skalieren kann. Auch hierzu sollten Sie sich informieren.

Die Dimensionierung und Anfangsbelegung des Netzes erfolgen im Block *add new weights*. Mit diesem Block können wir ein neues Neuronales Netz beliebiger Größe mit zufälligen Anfangsgewichten erzeugen. In diesem Fall hat es die Breite 3 und die Tiefe 2.

Da die Anzeige der vielen Zahlen ziemlich unübersichtlich und auch kaum informativ wäre, werden die Verbindungslinien (die *Kanten*) entsprechend den Werten der zugehörigen Gewichte farbcodiert: von vollem Grün für große positive Werte über Schwarz für kleine Beträge hin zu roten negativen Gewichten. Da anfangs nur positive Zahlen per Zufallsgenerator gezogen werden, ist ein neues Netz überwiegend grün. Es zeigt an, was sich aus den Berechnungen mit dem anzugebenden Eingabevektor ergibt.

Die *Knoten* des Netzes werden wie die Kanten farbkodiert. Unten stehen die Elemente des Eingabevektors als kleine Rechtecke. Die inneren Schichten bilden farbige Kreise und die letzte Schicht wird als Ausgabeschicht wieder rechteckig dargestellt. Die Richtung der Berechnung von unten nach oben zeigt der Pfeil ganz rechts. Da man Sprites aber einfach drehen kann, kann die Richtung natürlich auch anders dargestellt werden.

Eine einfache Anfangskonfiguration ergäbe sich damit zu:

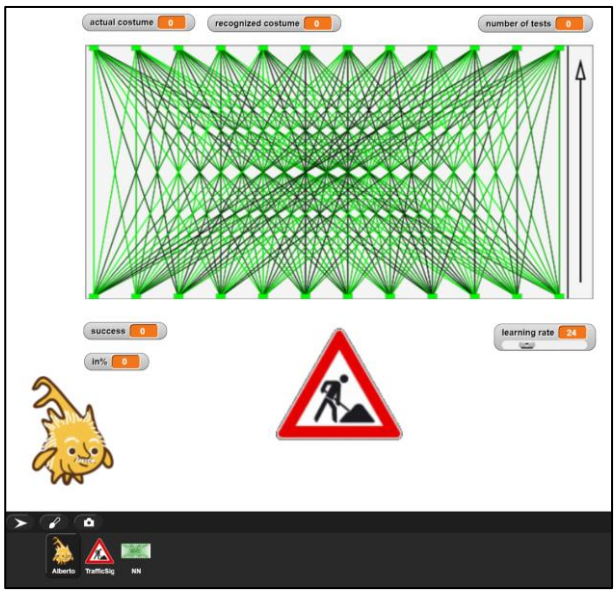


Oft benötigt man die Ergebnisse der letzten oder auch einer inneren Schicht des Netzes. Die können mithilfe des Blocks *output of…* bei vorgegebener Eingabe berechnet werden. Da die Farbcodierung nicht unbedingt das größte oder kleinste Element klar anzeigt, kann dieses mithilfe des *…of vector…*-Blocks bestimmt werden.

Das Training des Netzes erfolgt mithilfe des Blocks *teach NN …* durch Backpropagation mit einem anzugebenden Lernfaktor. Der darf anfangs durchaus etwas größer sein, um dann verkleinert zu werden.

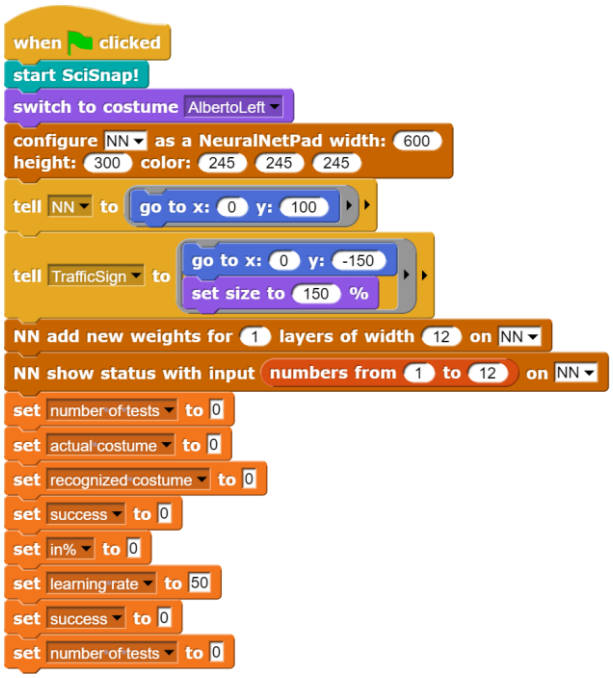
Wir wollen ein neuronales Netz (NN) so trainieren, dass es 12 unterschiedliche Verkehrszeichen erkennt. Dazu suchen wir Bilder von diesen Verkehrszeichen im Netz und verkleinern sie auf das Format 100 x 100 Pixel. Man kann sie jetzt zwar gut am Bildschirm darstellen, aber die 10000 Pixel sind als Eingänge für ein NN natürlich viel zu viel.

Um die Datenmenge in erträgliche Grenzen zu bringen, verkleinern wir die Pixel auf ein 2x2-Format durch *mean-pooling*, d. h. wir bilden jeweils die Mittelwerte der Farbpixel in den vier Quadranten des Bildes. Aus den 30000 Farbwerten des Verkehrszeichenbildes werden so 12.

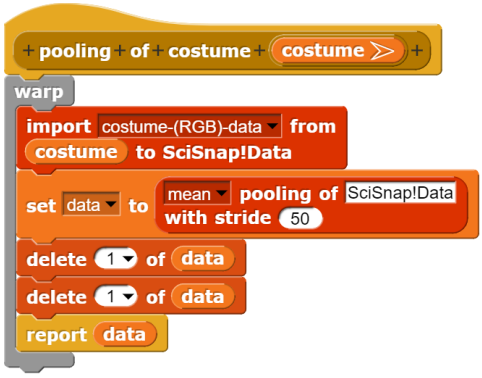


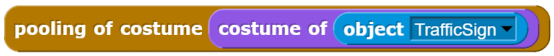
Weil es sich um ein schwieriges Problem handelt, übernimmt diesmal *Alberto* die Gesamtsteuerung.

Zum Start verpasst *Alberto* dem NN-Sprite ein neues Kostüm. Danach erzeugt er im NN die Gewichte für ein neues (hier: 12x2) Netz. Dieses lässt er mit einer (noch unsinnigen) Eingabe zeichnen. Danach schickt er das NN an einen gut gewählten Platz in der oberen Mitte und macht dasselbe mit dem Verkehrszeichen darunter. Zuletzt werden einige Variable auf 0 gesetzt. Die brauchen wir später.



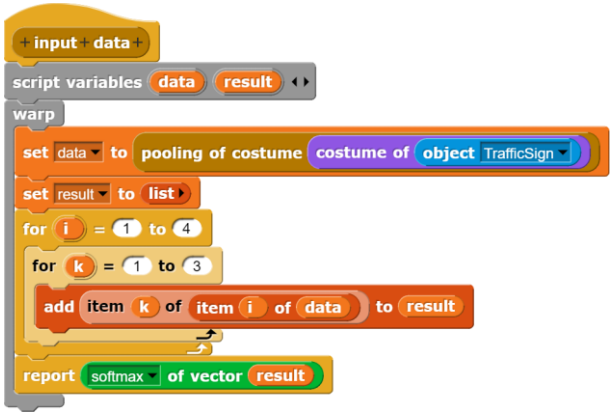
Alberto setzt muss zuerst einmal mithilfe der *Pooling*-Operation die Menge der Bildwerte reduzieren. Um die Operation anwenden zu können, muss er die Bilddaten importieren. Danach kann es sie umrechnen, die vorne in der Liste angegebenen Dimensionen des verkleinerten Bildes löschen und das Ergebnis zurückgeben. Wir fassen alles in einem neuen Block *pooling of <costume>* zusammen.





4	A	B	C	D
1	176.2016	61.1332	59.3068	200.328
2	176.0848	59.7848	59.4004	200.43
3	171.4432	53.4512	53.4504	199.512
4	169.4336	43.096	45.4232	199.41

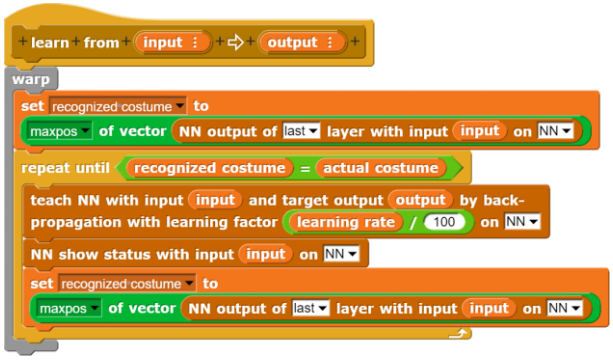
Die Farbwerte des reduzierten Bildes werden von *Alberto* zu einem Eingabevektor zusammengesetzt. Diese werden abschließend mit der *Softmax*-Funktion der *Data tools* modelliert, um ungünstige Eingangswerte auszuschließen.



Entsprechend lässt sich der Trainingsvektor in *training data* mit den gesuchten Ausgabewerten des NN ermitteln. In unserem Fall sollen alle Werte 0 sein bis auf den, der der Kostümnummer des Verkehrszeichens entspricht.

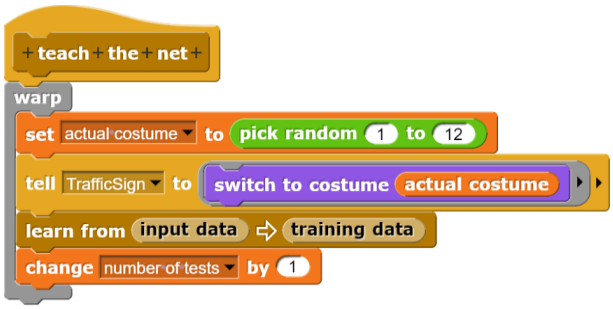
Das NN erhält zwei neue Methoden *learn from…* und *test with…* Beim Lernen wird die Position des Platzes mit dem größten Ausgabewert des NN ermittelt und mit der aktuellen Kostümnummer des Verkehrszeichens verglichen. Stimmen diese Werte nicht überein, dann wird weiter gelernt.

Zum Testen wird dieselbe Operation nur einmal ausgeführt.

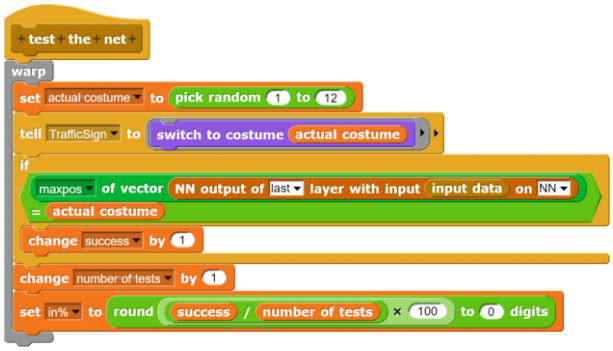


Jetzt haben wir alles zusammen, um *Alberto* vernünftig arbeiten zu lassen.

Ein Lehrvorgang besteht aus der Ermittlung einer zufälligen Kostümnummer mit dem entsprechenden Kostümwechsel. Danach wird der Lernvorgang des NN mit neuen Eingabe- und Zieldaten gestartet. Die Durchgänge werden mitgezählt.

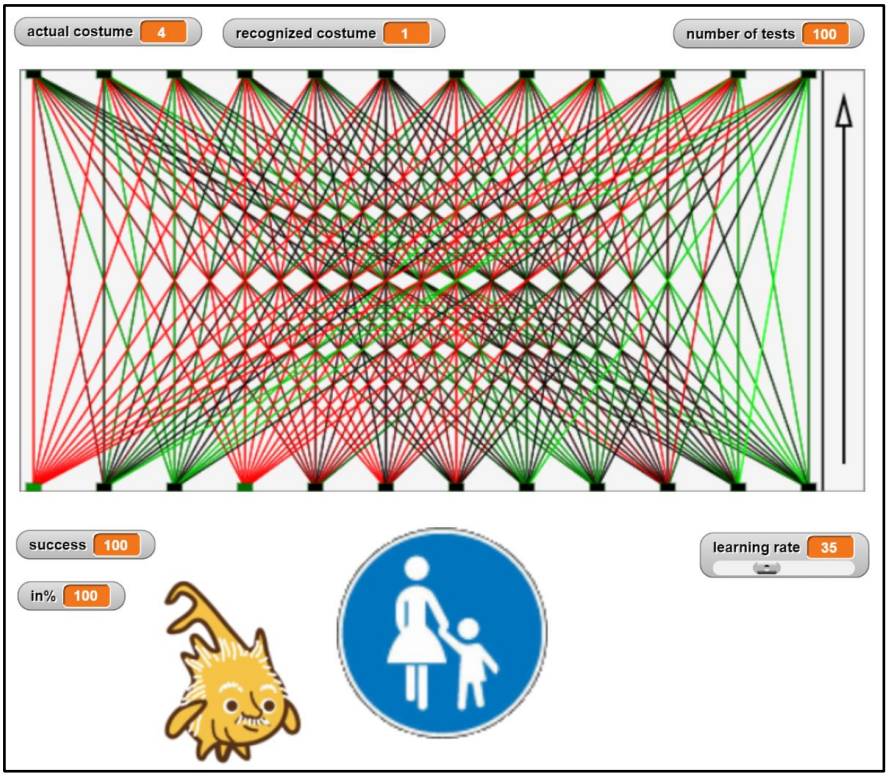


Zum Testen wird ähnlich vorgegangen: Das Kostüm wird gewechselt und geprüft, ob das NN die richtige Kostümnummer berechnet. Ist das der Fall, dann freuen sich alle. Der Prozentsatz der richtigen Versuche wird danach ermittelt.



Mehrere Lern- und Testläufe lassen sich dann leicht auslösen.

Nach einigen Hundert Trainingsläufen mit einer höheren Lernrate und nochmal mit einer geringen zur Feinabstimmung erreichen wir Erkennungsraten von bis zu 100%.



Aufgaben:

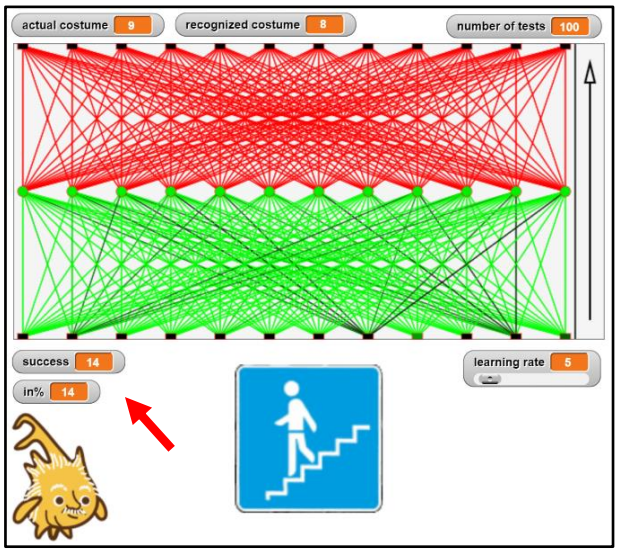
1. Trainieren Sie ein einschichtiges Netz mit unterschiedlichen Lernraten und Zahlen der Lerndurchläufe. Ermitteln Sie jeweils prozentual die Erkennungsrate.
2. Stellen Sie die Ergebnisse aus 1. grafisch mithilfe eines *PlotPads* dar.
3. Experimentieren Sie mit mehrschichtigen NNs. Werden die Ergebnisse besser?
4. Vergrößern Sie die Länge des Eingabevektors durch verändertes Pooling. Werden die Ergebnisse besser?
5. Vergrößern Sie die Anzahl der erkennbaren Schilder, indem Sie mehr als eine 1 in der Ausgabe zulassen.

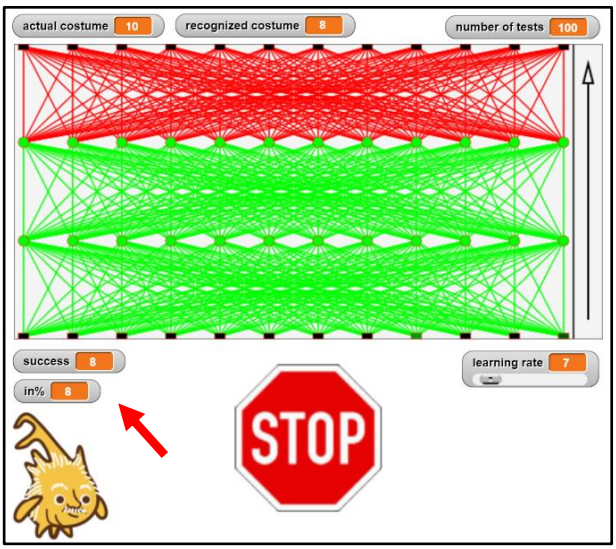
11.5 Under- und Overfitting

Maschinelles Lernen passt die Parameter einer Funktion mithilfe von Trainingsdaten so an, dass andere Werte gut prognostiziert werden – wenn alles klappt. Man baut also ein Prognoseinstrument, so eine Art „Fernrohr“ für Daten.

Man könnte nun meinen, dass so eine Funktion desto besser ist, je mehr anpassbare Parameter sie enthält. Dem ist aber nicht so. Einerseits erfordern (1.) mehr Parameter auch mehr Trainingsdaten und Trainingsläufe, also mehr Lernzeit; anderseits kann auch (2.) eine „unpassende“ Zahl der Parameter „gute“ Lösungen verhindern. Für beides geben wir jetzt ein Beispiel.

Zu 1: Beim Neuronalen Netz zur Verkehrszeichenerkennung (Beispiel 48) erzielen wir mit einer Schicht sehr gute Ergebnisse. Erhöhen wir die Zahl der Schichten und lassen die Zahl der Trainingsläufe gleich, dann verschlechtert sich drastisch die Erkennungsrate.





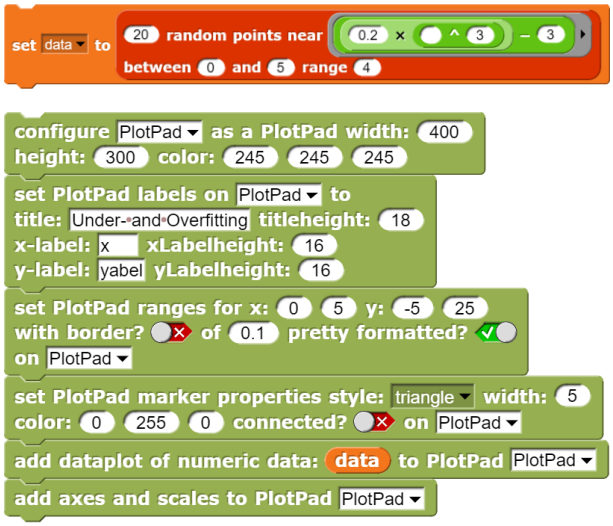
Zu 2: Wenn die Trainingsdaten von der Funktion gut reproduziert werden, dann heißt das noch lange nicht, dass das auch für andere Daten gilt. Es hängt sehr von der Art der Funktion ab, die erzeugt wird. Als Anwendung wählen wir das Beispiel *Polynominterpolation*.

Die Aufgabe lautet: *Mithilfe von Trainingsdaten werden die Koeffizienten eines Polynoms so angepasst, dass ANDERE Daten möglichst gut prognostiziert werden.*

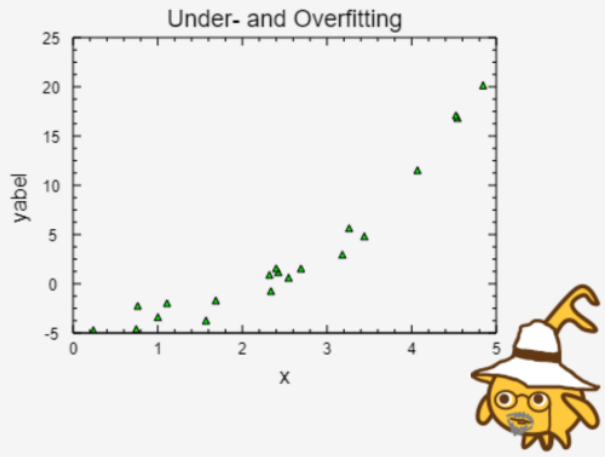
Dafür müssen wir Daten erzeugen, mit deren Hilfe ein Interpolations-Polynom berechnet wird. Die Aufgabe übernimmt diesmal *Hilberto*. Er erzeugt Daten, die um die Parabel $0{,}5 * x^2 - 3$ streuen.

Wir konfigurieren ein zweites Sprite neben *Hilberto* namens *PlotPad* als PlotPad und stellen die Daten darauf dar.

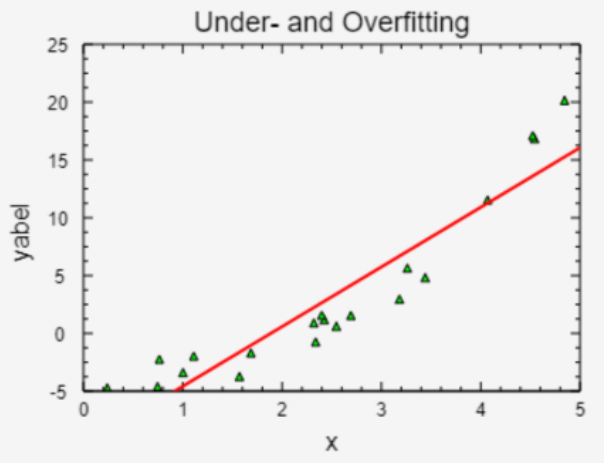
Als „Arbeitspferd“ wählen wir das *PlotPad*. Benötigte Funktionalität aus anderen Bibliotheken importieren wir bei Bedarf von dort.



Das genügt schon, um die Daten darzustellen.



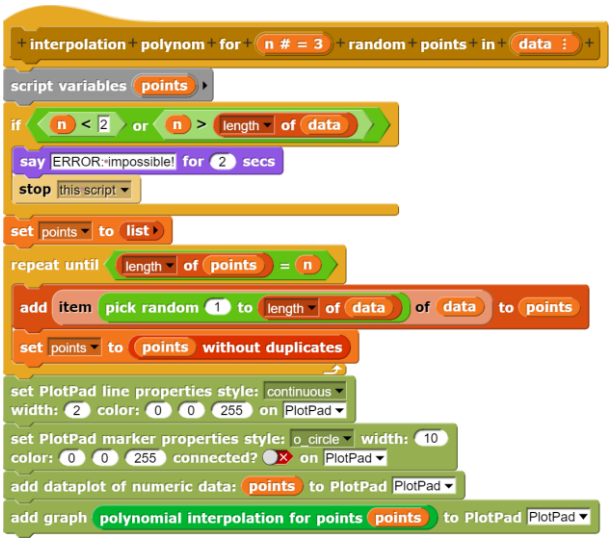
Die Interpolation probieren wir erst einmal mit einer Regressionsgeraden.

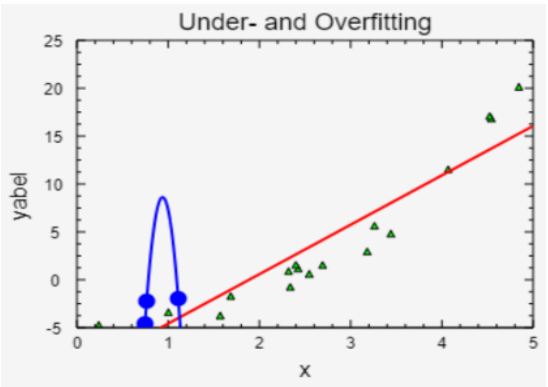


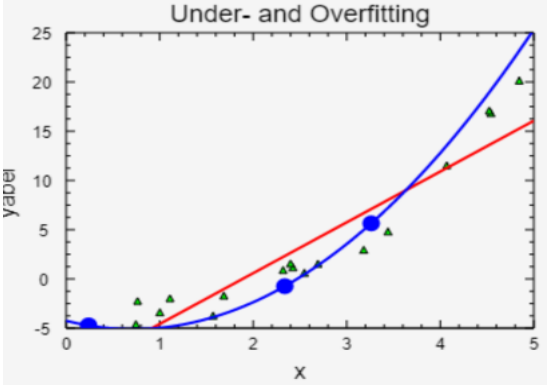
Das sieht eigentlich ganz nett aus, aber an den Seiten passt es nicht so recht.

Wir probieren es also mit einer Polynom-Interpolation.

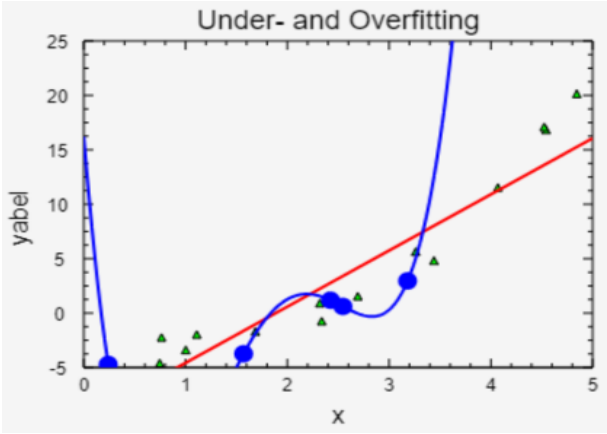
Zuerst einmal wählen wir drei Zufallspaare aus den Trainingsdaten, bestimmen daraus das Interpolationspolynom und zeichnen es. Weil wir weiter experimentieren wollen, verallgemeinern wir die Lösung gleich zu einem Polynom durch n Punkte. Die Ergebnisse hängen davon ab, welche Punkte erwischt wurden. Anbei ein schlechtes und ein ganz gutes Ergebnis.



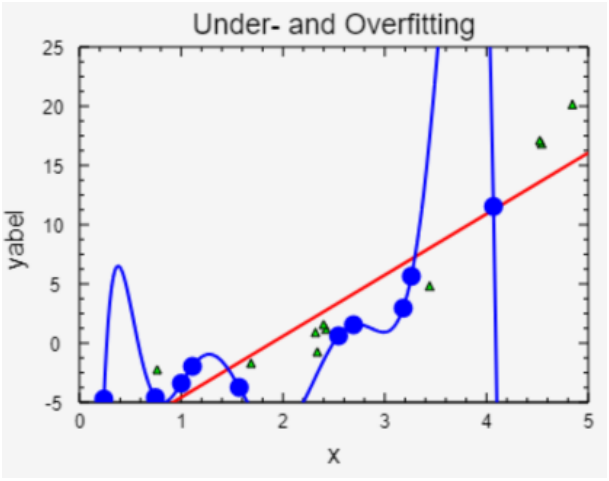


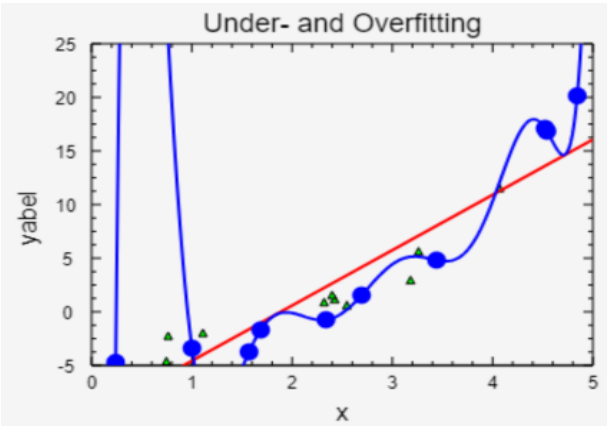


Jetzt werden wir mutig! Statt drei Punkten wählen wir 5. Wir wollen ja schließlich gute Arbeit abliefern! Das klappt halbwegs in der Mitte, und dann – upps!

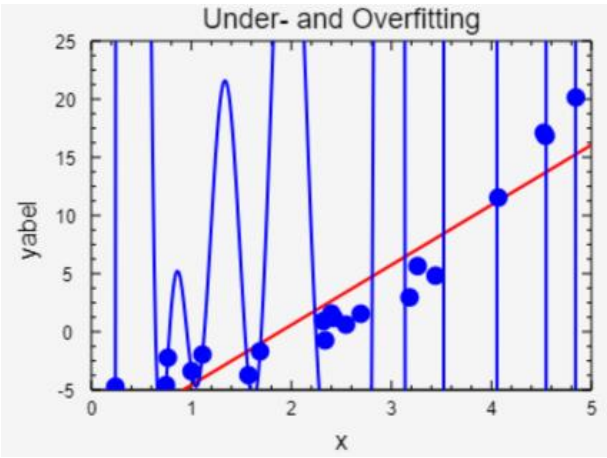


Vielleicht müssen wir ja einfach mehr Punkte nehmen. Versuchen wir es mit 10. Die Polynome verlaufen zwar durch mehr Punkte, aber an den Rändern „hauen sie ab“.





Na, dann mit allen Punkten!



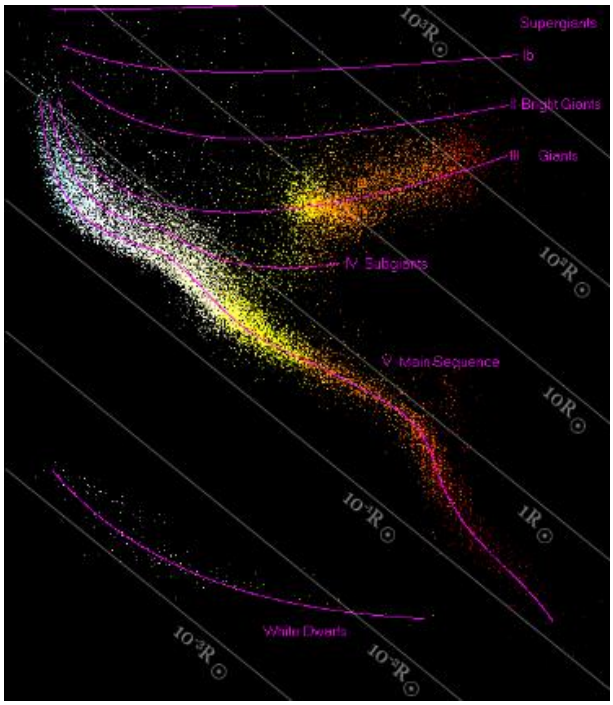
Man sieht, dass mit zunehmendem Grad des Polynoms zwar mehr Trainingsdaten direkt auf dem Graph liegen, dass aber dazwischen durch die wilden Oszillationen des Polynoms nur noch unsinnige Werte „prognostiziert“ werden.
Die Qualität des Gelernten hängt also sehr davon ab, wie wir mit Abweichungen umgehen. Wir müssen entscheiden, welche Ungenauigkeiten im Detail tolerierbar sind, damit die Prognose insgesamt zuverlässig wird. Ist der Grad des Polynoms zu klein, dann spricht man von *Underfitting*, ist er zu hoch, von *Overfitting*.

Aufgaben:

1. Diskutieren Sie unterschiedliche Möglichkeiten, einen „guten“ Grad des Interpolationspolynom (also seine höchste Potenz) festzulegen.
2. Formulieren Sie ihre Ergebnisse so präzise, dass sie sich als Skripte realisieren lassen.
3. Testen Sie die Skripte an unterschiedlichen Datensätzen.

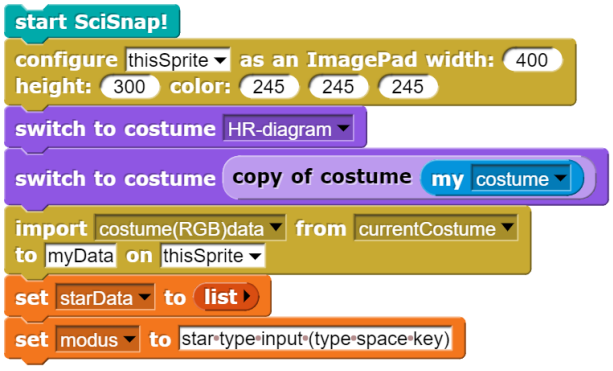
11.6 Klassifizierung mit dem kNN-Verfahren

Im Hertzsprung-Russel-Diagramm (s. Wikipedia) wird die Leuchtkraft von Sternen über ihrer Sternklasse aufgetragen. Es ergibt sich einen Art Linie von links-oben-nach rechts-unten, die „Hauptreihe“. Auf dieser halten sich Sterne wie die Sonne überwiegend auf. Rechts-oben über der Hauptreihe finden wir die Roten Riesen, links-unten unter der Hauptreihe die Weißen Zwerge. Das reicht erstmal. (Bildquelle: [HR])

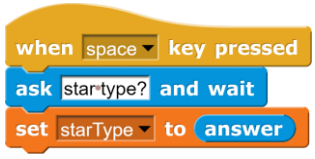


Wir wollen neue Sterne in diesem Diagramm klassifizieren, indem wir das Verfahren der *k-Nächsten-Nachbarn* (*kNN*) verwenden: Wir erzeugen als Trainingsdaten eine Liste von Sternen mit deren Koordinaten (einfach als Bildkoordinaten im Diagramm) und deren Typ. Wollen wir einen neuen Stern klassifizieren, dann bestimmen wir seine Position im Diagramm und suchen die nächsten *k* (z. B. *k=5*) Nachbarn. Danach bestimmen wir den am häufigsten auftauchenden Sterntyp in dieser Liste. Den weisen wir dem neuen Stern zu.

Zuerst einmal benötigen wir ein Bild des Hertzsprung-Russel-Diagramms ([HR]). Dieses importieren wir in *Snap!* als Kostüm eines *ImagePads* und erzeugen daraus die benötigten Daten. Da wir auf dem Bild zeichnen wollen, arbeiten wir mit einer Kopie des HR-Diagramms, um das Original nicht zu verändern.

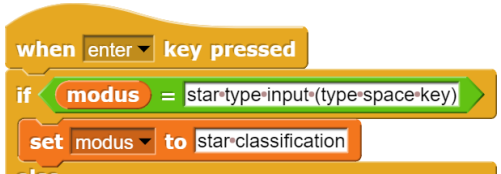


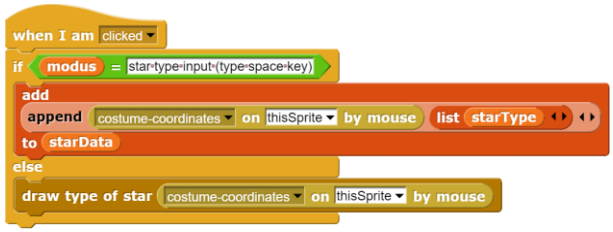
Die Trainingsdaten erhalten wir, indem wir einen Sterntyp vorgeben und danach einige Punkte im Diagramm anklicken, die diesem Typ entsprechen.



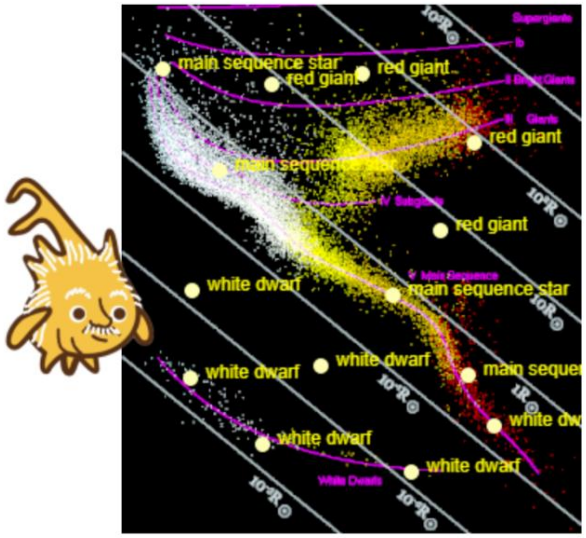
Danach können wir neue Sterne klassifizieren, indem wir sie anklicken und beschriften.

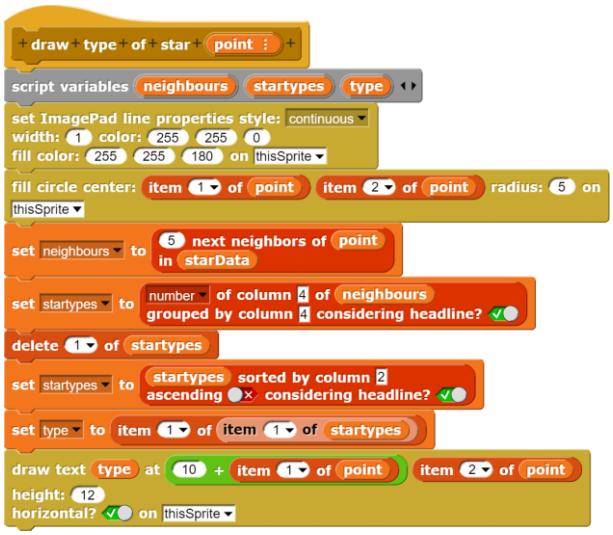
Wir stellen dazu einige Properties für die Darstellung ein und zeichnen einen Kreis am Ort des Sterns. Danach bestimmen wir die fünf nächsten Nachbarn und die Anzahlen des Auftretens ihres Typs. Im Ergebnis löschen wir die Überschriften und sortieren die Liste absteigend. Der Typ des neuen Sterns steht dann als erstes Element in der ersten Zeile. Diesen schreiben wir neben den Stern.





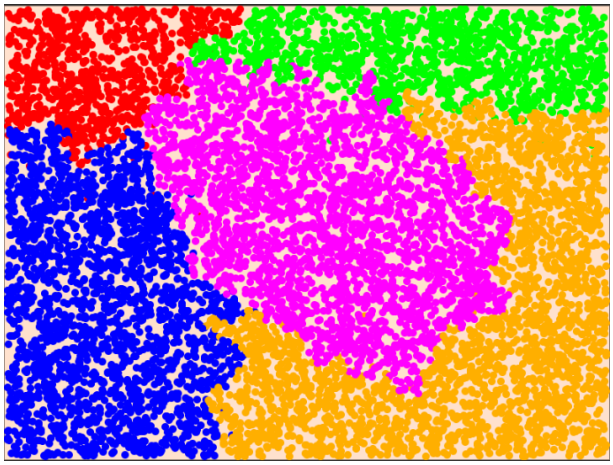
Das Ergebnis:





Aufgaben:

1. Fügen Sie die neu klassifizierten Sterne der Beispielliste hinzu, sodass sie bei weiteren Klassifizierungen mit hinzugezogen werden.
2. Zeichnen Sie für die unterschiedlichen Sternarten unterschiedlich farbige Punkte an den richtigen Stellen auf das Sprite, statt sie zu beschriften.
3. Lassen Sie den Prozess für zufällig ausgewählte Punkte ablaufen. Entsteht immer das gleiche Muster? Entstehen völlig verschiedene oder ähnliche Muster? Wovon hängt das ab?



11.7 Entscheidungsbäume nach dem ID3-Verfahren

Ein Entscheidungsbaum bildet das Verfahren nach, mit dem „Experten“ zu einer Entscheidung kommen. Ein einfaches Beispiel dafür ist das Kinderspiel „Tiere raten“, bei dem nacheinander Fragen gestellt werden (*„Hat es vier Beine?“, „Hat es Fell?, „Ist es der Osterhase?“, …*), die immer mehr Tiere ausschließen, bis entweder eines übrig bleibt oder bis es Streit gibt. Das Expertenwissen manifestiert sich in unserem Fall in Form von klassifizierten Daten, bei denen die Klassifizierung in der letzten Spalte stehen soll – der Einfachheit halber als binäre Entscheidung. Aus diesen Datensätzen wird ein Entscheidungsbaum konstruiert, den andere, noch nicht „gelabelte“ Daten durchlaufen können, um klassifiziert zu werden. Ein Verfahren zur Konstruktion solcher Bäume ist das *ID3-Verfahren* (*Iterative Dichotomiser 3*)[25]. Es erzeugt breite Bäume, die entsprechend schnell durchlaufen werden können, indem es die Entropie der einzelnen Attribute berechnet.

Ein Entscheidungsbaum besteht aus Knoten und Kanten, die von einer *Wurzel* (*root*) ausgehen. Bei den durch Kanten schichtweise verbundenen Knoten handelt es sich entweder um *innere Knoten* (*node*), von denen noch weitere Kanten ausgehen, oder um *Blätter* (*leaf*), die das Ende eines Zweiges markieren. Bei der Baumkonstruktion muss entschieden werden, in welcher Reihenfolge die Attribute benutzt werden, welches an einer bestimmten Stelle den größten Informationsgewinn verspricht. Üblicherweise wird der Informationsgewinn mithilfe der *Entropie* berechnet, die Auskunft über die „Unordnung“ eines Systems gibt. Kennen wir den Anteil p eines Attributwerts, dann ist die Entropie S definiert als $S(p) = -p \cdot \ln p$. Für ein Attribut A mit n Attributwerten mit den Anteilen p_i ist die Entropie dann die Summe der Teilentropien: $S(A) = \sum_{i=1}^{n} S(p_i) = -\sum_{i=1}^{n} p_i \cdot \ln p_i$. Der *Informationsgewinn* (*gain*), der durch die Entscheidung über einen Attributwert entsteht, ergibt sich aus der gewichteten Entropie $gain \approx \sum_{i=1}^{n} \frac{n_i}{n} \cdot p_i \cdot \ln p_i$. Sie sollten sich an anderer Stelle über die Details des Verfahrens informieren!

Zur Konstruktion und Befragung von Entscheidungsbäumen verfügt *SciSnap!* über drei Blöcke in den *Data tools*. Die Entropie einer Liste bestimmt der Block 

, 

konstruiert den Baum, und abgefragt wird der Baum mit 

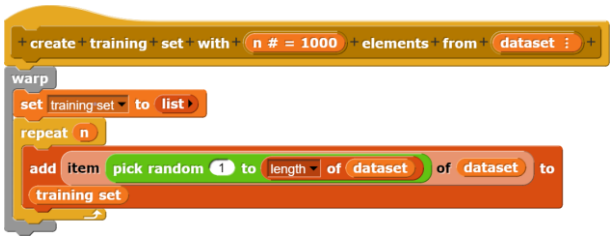
.

Als Beispiel wollen wir untersuchen, ob sich in den NYCitibike-Daten, die Sie schon kennengelernt haben, „geheime“ Zusammenhänge verstecken, die sich vor unseren ungeübten Augen verbergen. Es könnte z. B. sein, dass sich aus den Entleihdaten schließen lässt, welches Geschlecht die Entleihenden haben. Versuchen wir es einmal.

Wir laden den vollen Datensatz mit 577703 Elementen, den wir etwas „glätten“, indem wir die Ortsdaten (*latitude* und *longitude*) weglassen, weil sich die Entleihstationen aus den anderen Daten ergeben. Die Zeitskalen verkürzen wir wie in den anderen Beispielen auf die Tagesstunde und die nicht *gelabelten* Daten (*Geschlecht unbekannt* („0“)) streichen wir.

[25] J.R. Quinlan, 1986

Zum *Training* benutzen einen verkürzten Datensatz mit 1000 Entleihdaten, also zur Konstruktion des Entscheidungsbaums. Dazu ziehen wir einfach 1000 Datensätze aus dem Gesamt-Datensatz.



Dann wird der ID3-Baum aus den Daten erzeugt. Das dauert etwas, hier: 35,2 Sekunden.

Danach testen wir den Baum mit zufällig gewählten Daten aus dem Trainings-Datensatz. Damit sollte es ja klappen.

Tut es auch.

Jetzt benutzen wir den vollen Datensatz mit allen Datensätzen, von denen nach dem Filtern immer noch 336955 übriggeblieben sind. Vielleicht zeigt sich da ja die unheimliche Kraft der „künstlichen Intelligenz“?

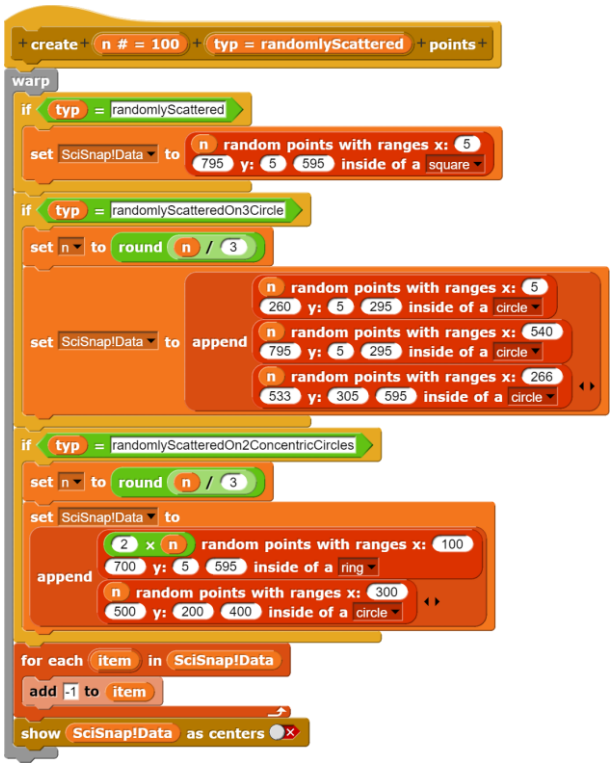
Na ja. Durch reines Raten wären wir wesentlich besser gewesen! Entweder verfügt die von uns geschaffene KI, der Entscheidungsbaum, über gar keine unheimliche Kraft, oder die Daten waren für die Fragestellung ungeeignet, oder wir haben es mit einem typischen Overfitting zu tun, denn die Trainingsdaten werden perfekt identifiziert, der Rest aber fast gar nicht. Suchen Sie mal nach dem Grund!

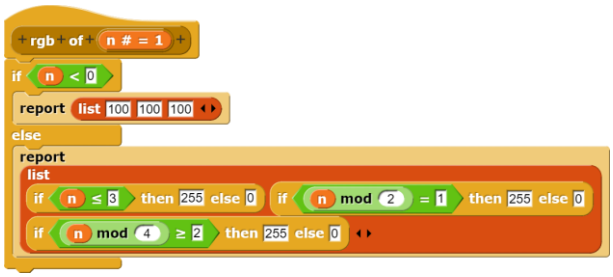
Wenigstens haben wir gelernt, wie ein Entscheidungsbaum konstruiert und benutzt wird. Man muss dankbar sein! 😉

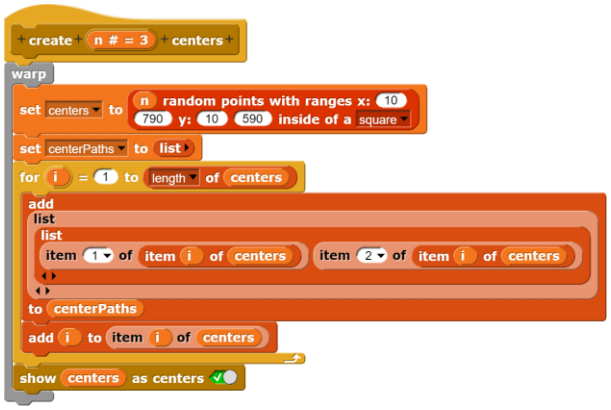
11.8 k-means-Clustering

Die *Data tools*-Palette stellt zwar Blöcke zum *k-means-Clustering* zur Verfügung, aber sie liefern (natürlich) nur das Endergebnis. Das Verfahren dient dazu, *k* Zentren so auf n Datenpunkte zu verteilen, dass die entstehenden *k* Gruppen möglichst gleichverteilt sind und die Abstände zu den Zentren minimal. Als Beispiel mag eine Wohnsiedlung dienen, in der k Verteilerzentren z. B. für Telefonanschlüsse oder elektrische Energie errichtet werden sollen. In diesem Fall wird der euklidische Abstand genommen werden, aber auch andere Metriken sind möglich, z. B. entlang vorhandener Wege oder die Levenshtein-Distanz.

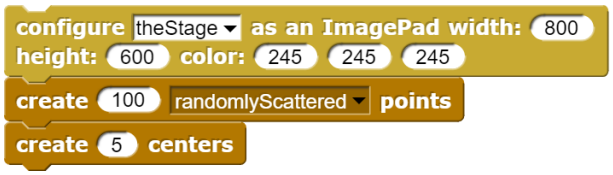
Wir wollen hier einmal den gesamten Prozess verdeutlichen. Dazu erzeugen wir eine Menge von z. B. 100 Zufallspunkten mit „JavaScript-Koordinaten“, an die wir jeweils noch die Clusternummer anhängen. Die ist anfangs noch „-1“, weil bisher kein Clustering stattgefunden hat. Dazu erzeugen wir „k“, z. B. 5, „Zentren“. Punkte und Zentren stellen wir als Kreise bzw. Quadrate auf der Bühne dar. Die Farben, anfangs grau, bestimmen die Clusternummern.

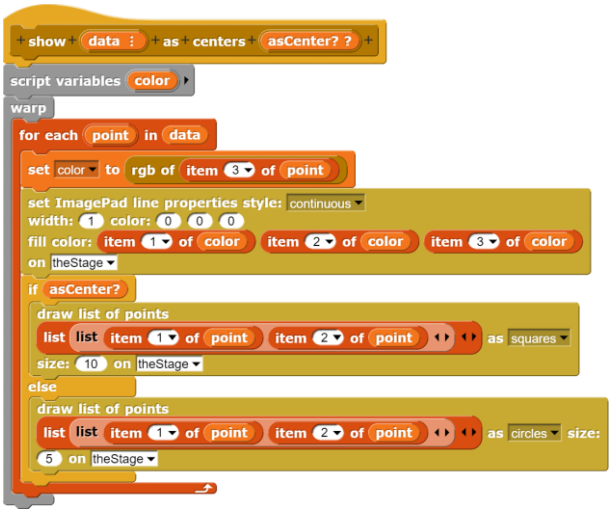




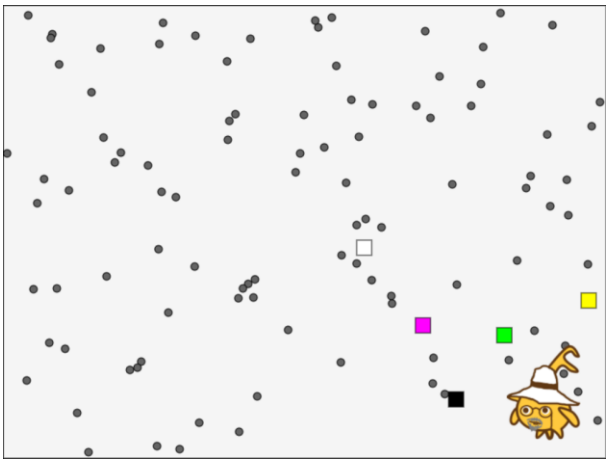


Die Befehlsfolge



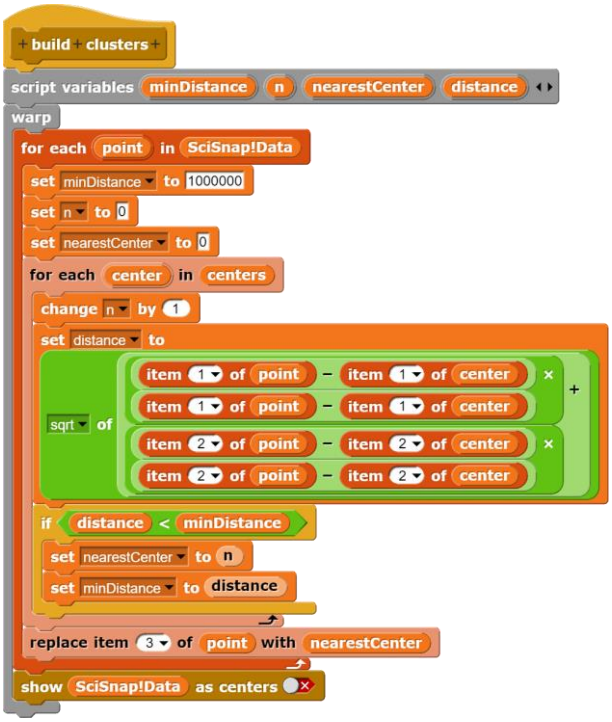


ergibt z. B. das folgende Bild.

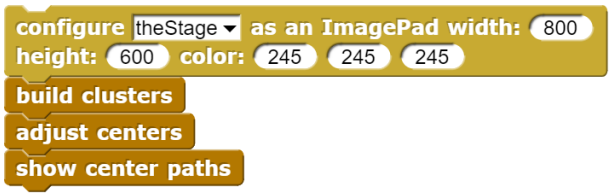


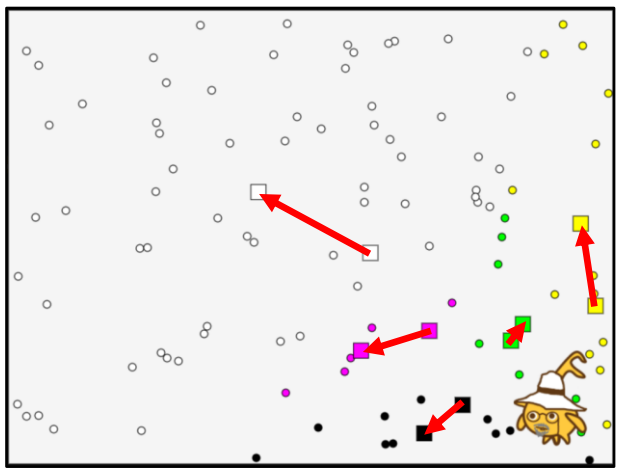
Wir nummerieren die Zentren und tragen ihre Koordinaten in eine Liste *centerPaths* ein, damit wir ihre Bewegungen nachvollziehen können.

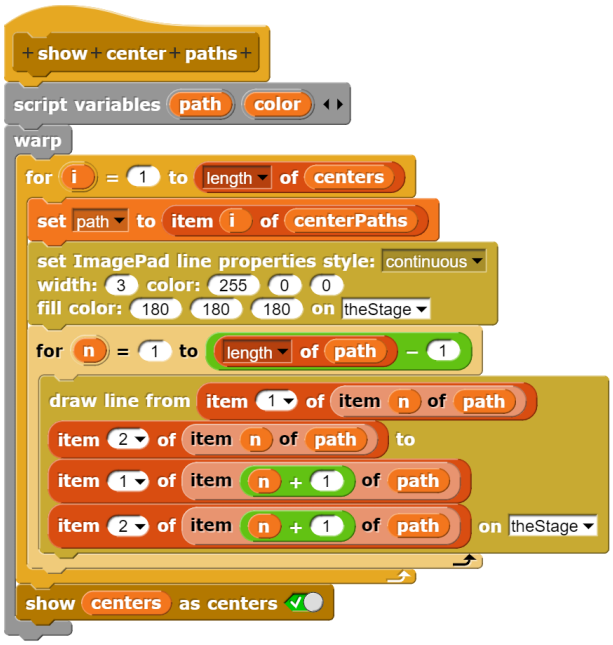
Das k-means-Verfahren bestimmt nun für jeden Punkt das nächstgelegene Zentrum und färbt die Punkte in dessen Farbe ein.



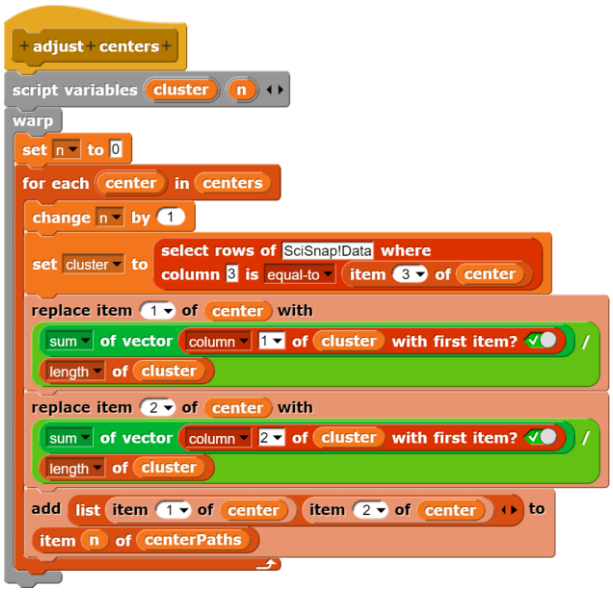
Zusammengefasst erhalten wir:



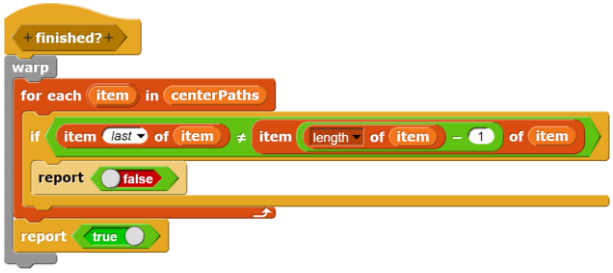




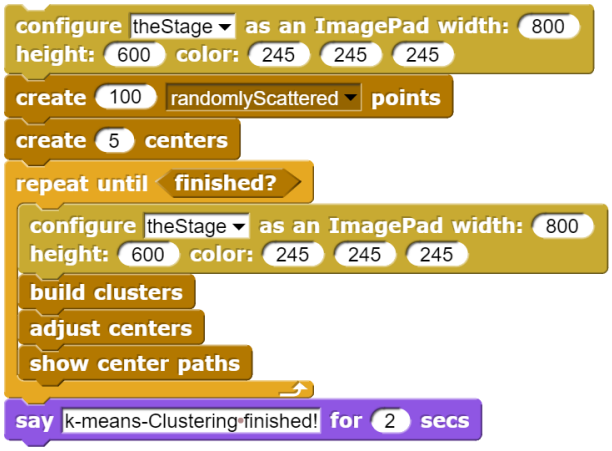
Danach werden die Zentren in den Mittelpunkt der ihnen zugeordneten Punktmenge verschoben. Die neuen Positionen werden in die Positionsliste eingetragen.

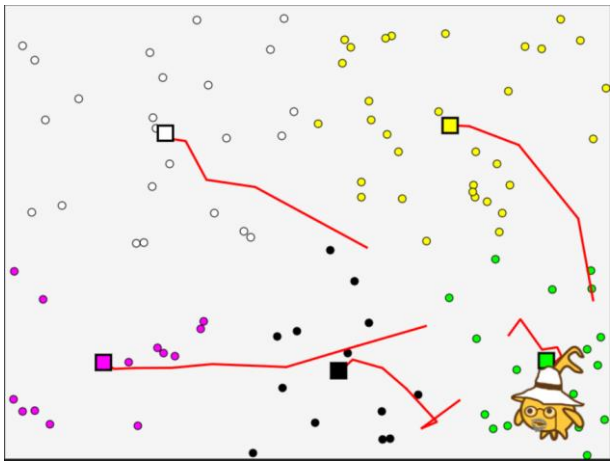


Damit haben wir meistens natürlich noch keine gleichmäßige Verteilung auf die Zentren erreicht. Deshalb wird das Verfahren solange fortgesetzt, bis sich keine Verschiebungen bei den Zentren mehr ergeben.

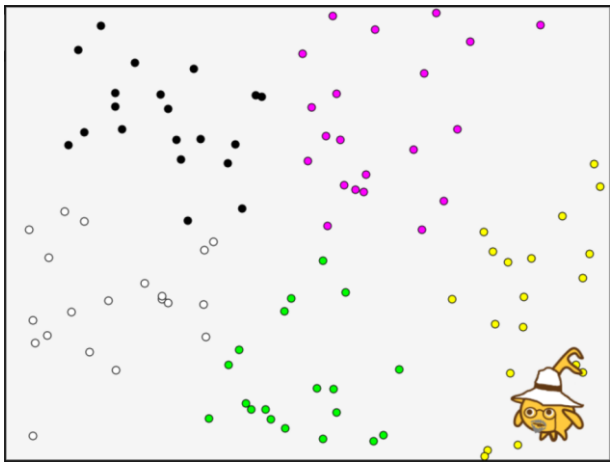


Wir erhalten das Ergebnis:

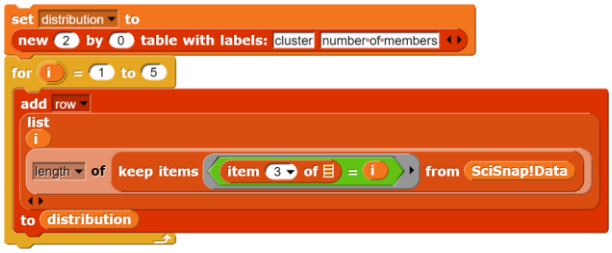




Unter Verwendung der vorhandenen Blöcke hätten wir das Ergebnis natürlich auch etwas einfacher erhalten können – aber eben ohne Darstellung des Prozesses:



Sehen wir uns noch kurz die Verteilung der Punkte auf die fünf Cluster an:



distribution

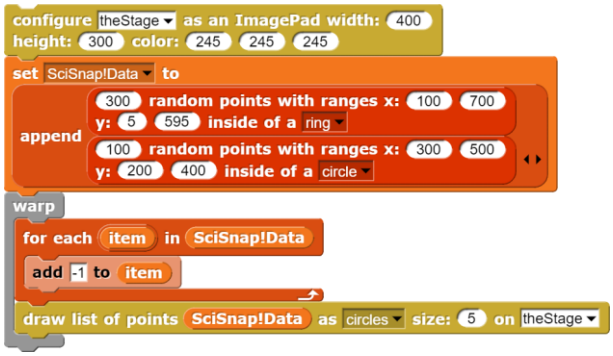
6	A	B
1	cluster	number of members
2	1	20
3	2	21
4	3	20
5	4	20
6	5	19

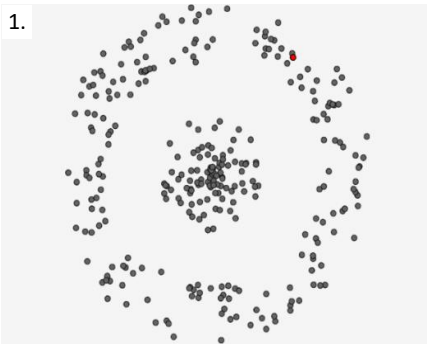
Das Ergebnis ist erfreulich. Ginge es um die Verteilung von Telefonzentralen, dann könnten die Bautrupps jetzt kommen. 😉

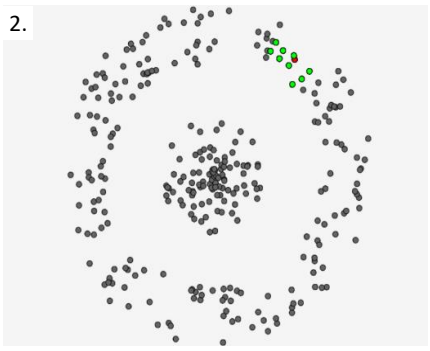
11.9 Clustering mit dem DBSCAN-Verfahren

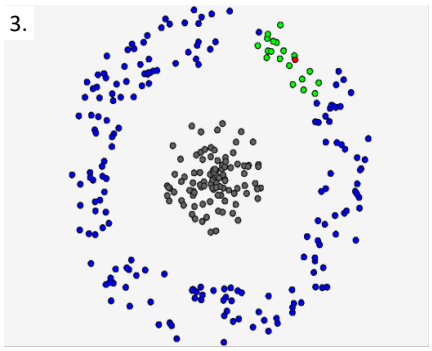
Das *DBSCAN*-(*density based spatial clustering of applications with noise*)-*Verfahren* versucht „Dichte-Inseln" im Datenraum zu identifizieren. Vor dem Start des Verfahrens müssen zwei Parameter gewählt werden: ein *Radius* gibt an, bis zu welchem Abstand Punkte als „benachbart" angesehen werden sollen, und eine *minimaleGruppengröße* legt fest, ab wann von einer Punktgruppe zu reden ist. Der Prozess durchläuft alle Datenpunkte und bestimmt mithilfe des Radius´ deren Nachbarpunkte. Sind genügend vorhanden, dann werden alle gefundenen Nachbarn wiederum auf Nachbarschaften untersucht. Sind diese groß genug, dann werden sie der ursprünglichen hinzugefügt. Auf diese Art entsteht eine zusammenhängende „Punktwolke" zusammengehöriger Punkte – ein Cluster. Sind danach noch nicht alle Datenpunkte erfasst, dann wird mit dem nächsten nicht verarbeiteten und einer neuen Clusternummer weitergemacht.

Wir erzeugen eine Punkteverteilung und wählen daraus einen beliebigen Punkt. Wir zählen dessen Nachbarn innerhalb des Radius (Bild 2) und erhalten 5 Nachbarn. Sind das genügend, dann fügen wir deren Nachbarschaf-ten zur ursprünglichen hinzu – wenn sie groß genug sind (Bild 3).

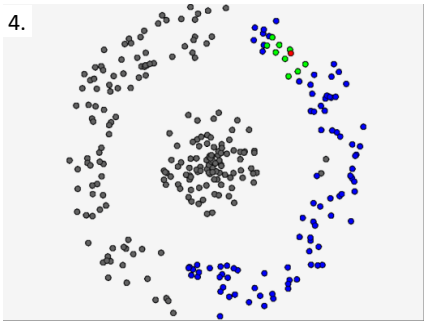








Dieser Prozess setzt sich so lange fort, bis keine genügend große neue Nachbarschaften gefunden werden (Bild 4). Haben einige Punkte nicht genügend Nachbarn, dann werden sie als Ausreißer (*noise*) bezeichnet und nicht geclustert (Bild 4, graue Punkte).



Wie man sieht, bildet *DBSCAN* wie in diesem Fall oft andere Cluster als das *k-means-Verfahren*. Benutzt man Clustering, um unklassifizierte Daten zu klassifizieren, dann „lernt" ein System beim unüberwachten Lernen möglicherweise ganz unterschiedliche Dinge – je nach eingesetztem Verfahren!

Für etwas größere Datenmengen bietet es sich an, den in SciSnap! vorhandenen Block *DBSCAN clustering for …* DBSCAN clustering for SciSnap!Data radius 50 minMembers 5 zu benutzen, der nach dem im Folgenden geschilderten Verfahren arbeitet.

Implementieren wir also das Verfahren.

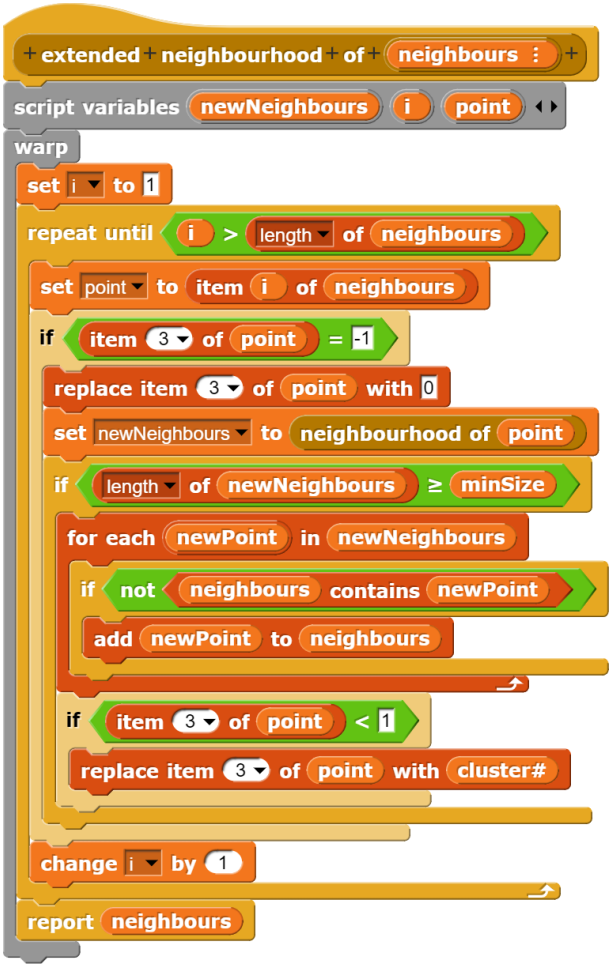
Zuerst wird die Punktmenge erzeugt und der Variablen SciSnap!Data zugewiesen. Die Punkte werden gezeichnet.

Dann setzen wir einige Startwerte und zeichnen den Startpunkt.

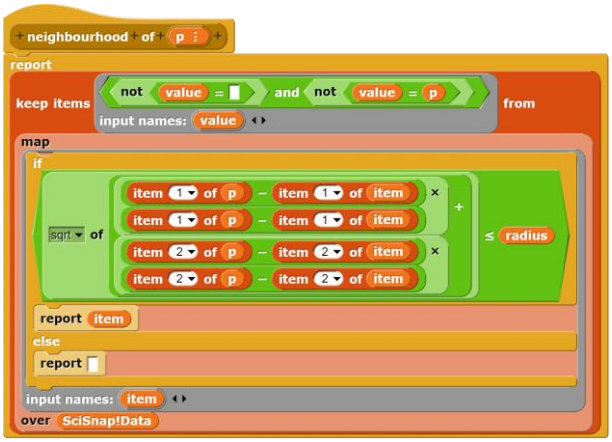
Die erste Nachbarschaft wird berechnet und angezeigt.

Dann startet das DBSCAN-Verfahren: Für alle Punkte der Nachbarschaft werden deren Nachbarschaften bestimmt, überprüft und ggf. in die alte Nachbarschaft übernommen. Bei Bedarf wird die Erzeugung des nächsten Clusters gestartet.

Die beiden zusätzlich benutzten Blöcke sind …



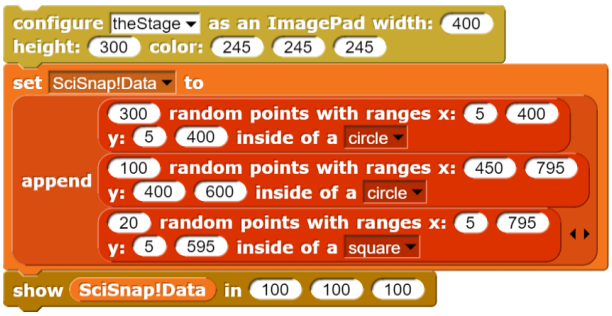
… und



11.10 Ausreißer-Erkennung mit dem DBSCAN-Verfahren

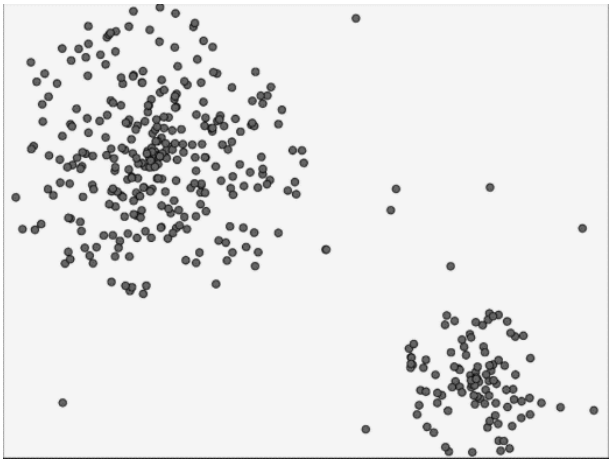
In vielen Fällen sind nicht die Cluster interessant, sondern eher die Ausreißer, also Daten, die „nicht ins Raster passen“, die auffällig sind. Das DBSCAN-Verfahren eignet sich gut dafür, weil es „abseits“ liegende Punkte nicht clustert (*outlier detection*).

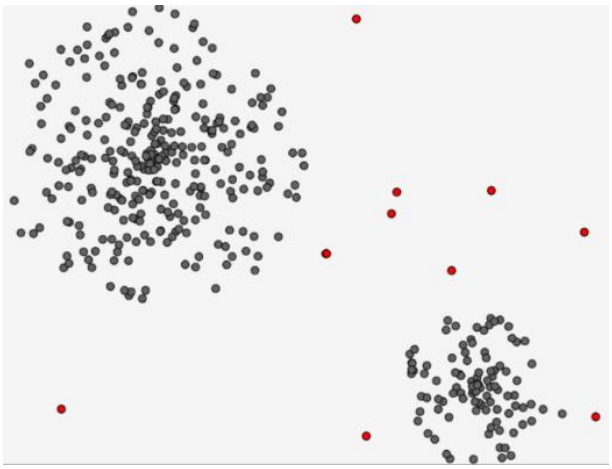
Wir erzeugen geeignete Zufallsdaten, also zwei „Haufen“ und einige über die Bühne verstreute Punkte, von denen einige außen liegen. Diese Daten clustern wir mithilfe des DBSCAN-Blocks.



397	635.999983	528.288293	2
398	606.007918	513.330165	2
399	558.459258	512.274097	2
400	656.989110	590.342389	2
401	780	537	-2
402	79	527	-2
403	643	243	-2
404	519	245	-2
405	61	201	1
406	426	325	-2
407	98	204	1
408	591	347	-2
409	165	304	1

Aus dieser Liste greifen wir die Ausreißer als nicht geclusterten „noise“ heraus und lassen die Punkte in Rot anzeigen. Die Blöcke fassen wir dafür zusammen.





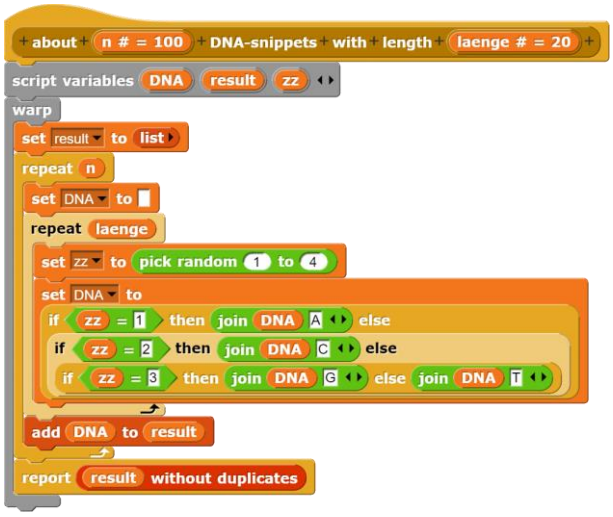
Das Beispiel ist in seiner Kürze natürlich unrealistisch. Wenn wir uns aber z. B. Parameter von E-Mails ansehen wie die Struktur des Headers, Anzahl der Adressaten, Art der Anhänge, Schlüsselworte, …, dann kommen wir in den Bereich realer Anwendungen, etwa bei der Spam-Erkennung.

Aus diesen Überlegungen folgen zwei Dinge:

- Einerseits müssen die Daten offensichtlich stark aufbereitet werden, um in einem geeigneten hochdimensionalen Raum dargestellt zu werden. Es müssen Verfahren entwickelt werden, die den Datenpunkten einen Vektor zuweisen – und umgekehrt. Das eigentliche Clustern steht dann gar nicht mehr im Mittelpunkt: es ist nur ein Schritt innerhalb des Prozesses.
- Andererseits nimmt mit der Zahl der Parameter die Anzahl der Dimensionen des Datenraums zu, sodass die vorhandenen Daten kaum noch Nachbarn um sich haben. Diesem *Fluch der Dimensionalität (curse of dimensionality)* kann nur begegnet werden, wenn extrem viele Daten zur Verfügung stehen. Wir haben ein weiteres Beispiel dafür, dass es beim maschinellen Lernen nicht nur auf die Technik der Computer, sondern ebenso auf die Verfügbarkeit von Daten ankommt.

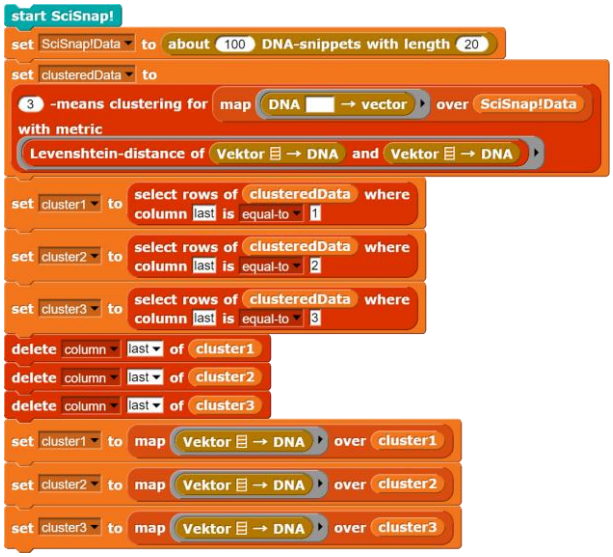
11.11 DNA-Clustering mit Levenshtein-Distanz

DNA-Untersuchungen beruhen oft darauf, die DNA zu zerlegen, DNA-Schnipsel zu untersuchen und ggf. die DNA wieder zusammenzusetzen. In diesem Fall wollen wir eine „Suppe“ mit unterschiedlichen DNA-Schnipseln in drei ähnliche Gruppen zerlegen, wobei der Abstand der DNA-Sequenzen mithilfe der Levenshtein-Distanz für Zeichenketten gemessen wird.



DNAs bestehen aus Folgen der Basen A, C, G und T (s. Literatur). Da unsere Blöcke mit Mittelwertbildung und damit mit Zahlen arbeiten, bilden wir solche Folgen auf einen n-dimensionalen Raum ab, dessen Koordinaten nur die Werte 1 bis 4 annehmen. Wir müssen also DNAs auf solche Vektoren abbilden, diese verarbeiten und zurück transformieren.

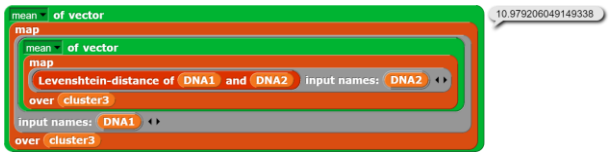
Wir beginnen mit der Erzeugung der DNA-Schnipsel. Wir begrenzen sie alle auf die Länge von 20 und bilden drei Cluster mithilfe der Levenshtein-Metrik.

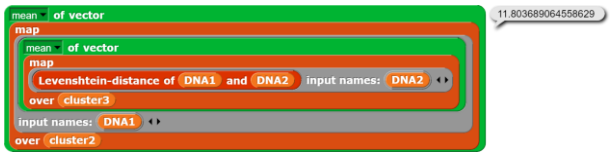


Danach werden die drei Cluster extrahiert und die Clusternummer wird abgetrennt.

Danach können die Vektoren der Cluster zurück in Basenfolgen übersetzt werden.

Zum Test bilden wir die mittleren Levenshtein-Entfernungen innerhalb eines Clusters und zwischen zwei Clustern.



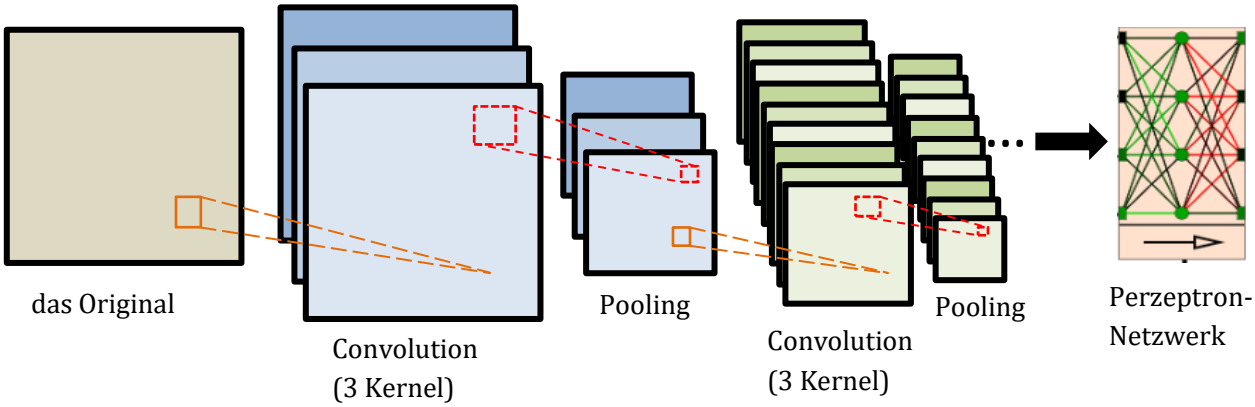


Na gut - wenigstens etwas unterscheiden sie sich.

11.12 Zeichenerkennung mit einem Convolutional Neural Network

Die immense Zahl von Parametern in vollständig verbundenen Perzeptron-Netzen und der daraus folgende Bedarf an riesigen Mengen von Trainingsdaten hat zu anderen Netz-Varianten geführt, um diese Zahl drastisch zu reduzieren. Eine davon sind die *Convolutional Neural Networks (CNNs)*, bei denen die Eingabedatenmenge für das Perzeptron-Netz reduziert wird. Diese Art von Netzen wird z. B. in der Bild- und Spracherkennung sehr erfolgreich eingesetzt.

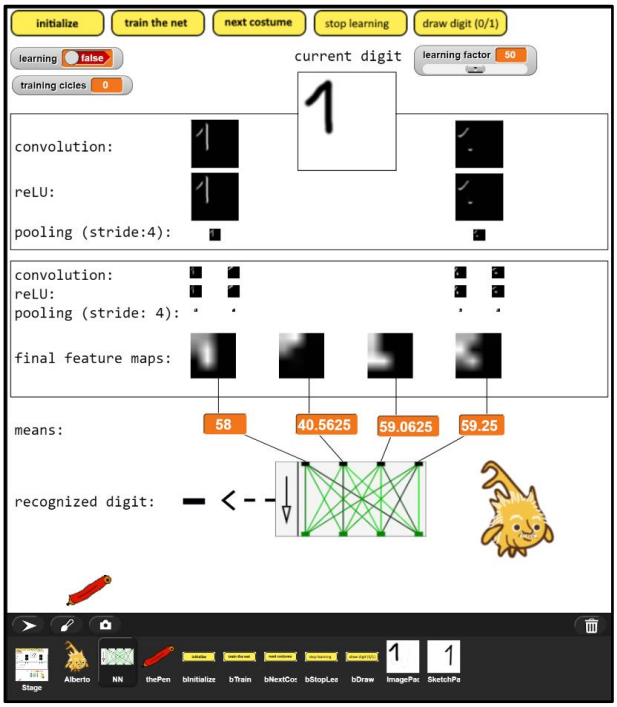
CNNs reduzieren die Datenmenge, indem in einem mehrstufigen Prozess zuerst mehrere Kernels angewandt werden, die bestimmte Eigenschaften z. B. eines Bildes (Kanten, ovale Flächen, …) herausfiltern und so zu mehreren *Feature-Maps* führen, die üblicherweise die gleiche Größe haben wie das Original. Das vergrößert erstmal die Datenmenge. Anschließend wird meist eine nichtlineare Aktivierungs-Funktion (*reLU*) auf die Feature-Maps angewandt und anschließend eine *Pooling*-Operation, die die Datenmenge wieder verkleinert. Meist handelt es sich um *Max-Pooling*, bei dem der Maximalwert aus einem Ausschnitt der Daten bestimmt wird. Macht man das mit einem „Fenster“, das mit einer bestimmten Schrittweite (*stride*) über die Feature-Map bewegt wird, dann erzeugt jeder Pooling-Schritt einen Wert der nächsten, verkleinerten Feature-Map.



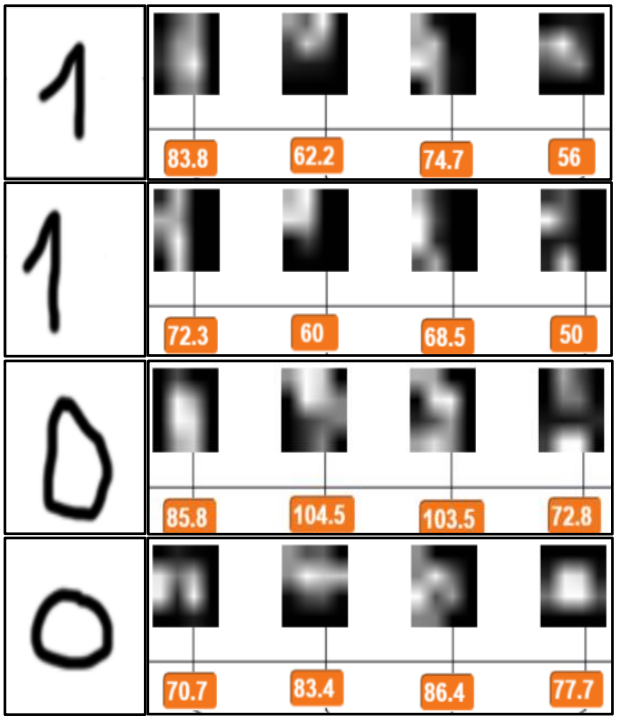
Als Beispiel nehmen wir einen Kernel, der senkrechte Linien filtert: er färbt einen Punkt weiß, wenn sich neben dem Punkt ein zweiter Bildpunkt befindet, sonst schwarz. Im „gefalteten“ Bild können wir dann senkrechte Linien des Originals als helle Flecken erkennen. Kommt es nicht so sehr darauf an, wo genau diese Linien sind, dann verlieren wir nicht allzu viel Information beim Pooling. Ein weißer Punkt in einer Feature-Map nach diversen Pooling-Prozessen bedeutet dann: *„In diesem Bereich befand sich irgendwo eine senkrechte Linie.“* Anhand solcher Daten aus mehreren Feature-Maps kann dann z. B. abgeleitet werden, dass sich dort auch eine horizontale Line, also eine Ecke befand. Hätten wir nach „beigen“ Bereichen sowie „ovalen“ Formen gesucht, dann wäre die Chance, Gesichter zu identifizieren, gar nicht so schlecht.

Wir wollen jetzt ein Modell für so ein CNN bauen, das die handgeschriebenen Ziffern *Null* und *Eins* unterscheiden kann. Dazu benutzen wir die *Data tools* für Hilfsoperationen, ein *ImagePad* für das eigentliche Bild und – natürlich – ein *NeuralNetPad* für das Perzeptron-Netzwerk am Ende der Kette. Ein weiteres, „normales“ Sprite namens *Alberto* soll die Abläufe steuern. Damit das Modell leichter zu bedienen ist, ergänzen wir es um einige Buttons sowie einen Stift, um die Oberfläche übersichtlich zu gestalten. Im Screenshot befindet sich das zu analysierende Bild oben im Kästchen, das Neuronale Netz zeigt sein Ergebnis unten an. Dazwischen werden von oben nach unten die verschiedenen Zwischenschichten durchlaufen und angezeigt. Als Zugabe enthält das Modell noch die Möglichkeit, eigene Ziffern zu zeichnen. Zu betonen ist, dass die Faltungsoperationen *„senkrechte Linie suchen“* bzw. *„horizontale Line suchen“* nicht gut zur Ziffernerkennung geeignet, dafür aber im Bild gut nachvollziehbar sind.

Unser CNN wird mit je 10 Ziffern aus je 64x64 Pixeln für die Nullen und Einsen trainiert. Anschließend soll es diese sowie andere handgeschriebene „erkennen“. Eigentlich müssten wir mehrere Kernel unseres CNN speziell für diese Aufgabe trainieren. Stattdessen nehmen wir nur zwei bekannte Kernel zur Erkennung vertikaler und horizontaler Linien, weil sich durch die Beschränkung auf zwei alles am Bildschirm darstellen lässt und die Ergebnisse sogar halbwegs zu interpretieren sind. (Die Erkennungsrate leidet darunter allerdings heftig!) Trainiert wird also nur das Perzeptron-Netz mit vier Eingangswerten.

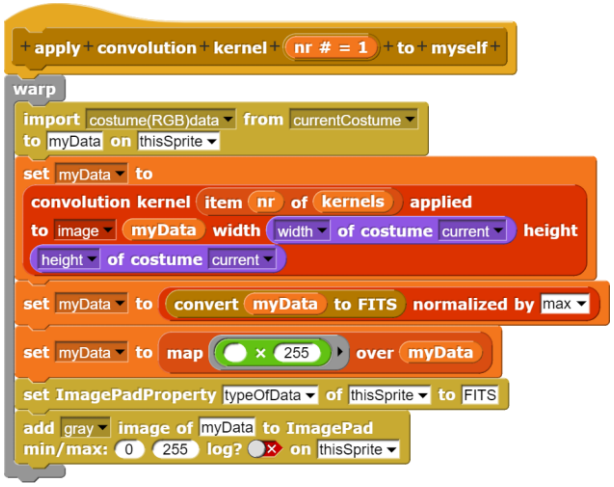


Im nebenstehenden Bild sind nach zwei Stufen der Reduktion vier Feature-Maps von je 16x16 Pixeln übrig, bei denen jeweils zweimal die Operationen *Convolution → reLU → Max-Pooling* durchlaufen wurden: ganz links mit dem Kernel für senkrechte Linien, dann mit beiden Kernels in unterschiedlicher Reihenfolge und zuletzt zweimal mit dem Kernel für horizontale Linien. Die Ziffern darunter geben den Mittelwert der Helligkeit gemessen über das gesamte Bild an. Wenden wir dieses auf verschiedene Ziffern an, dann zeigt sich die Möglichkeit, trotz des sehr ein fachen Verfahrens Unterschiede zwischen Nullen und Einsen zu messen.

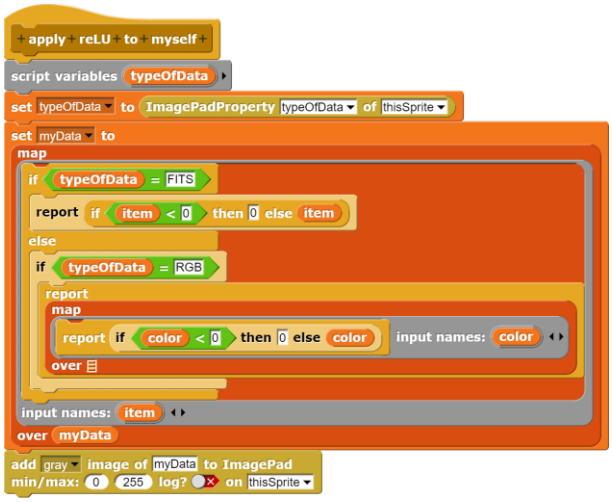


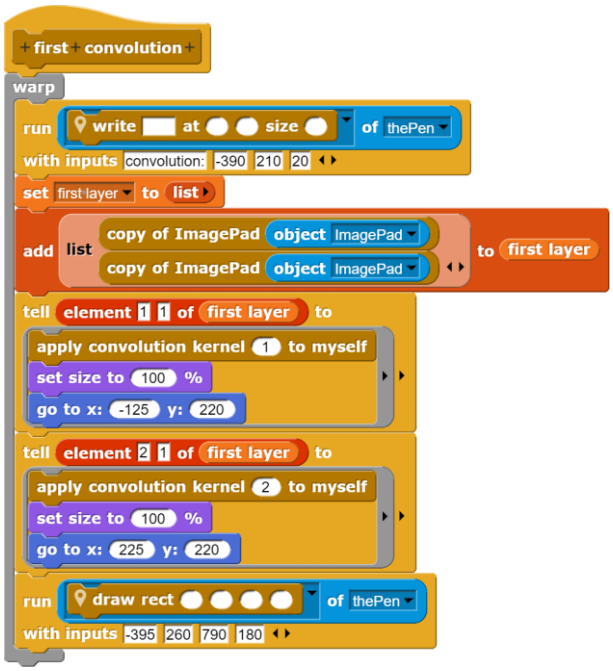
Betrachten wir die Funktionalitäten der Objekte:

Das *ImagePad* stellt die Daten eines neuen Kostüms als Grundlage für die Analyse bereit. Dazu erzeugt es eine „erste Schicht“ als Liste *first layer*, die aus zwei Kopien seiner selbst besteht. Auf diesen lässt es dann jeweils eine Faltung (*Convolution*) mit zwei verschiedene Kernen anwenden. Danach werden ein paar Linien gezeichnet.

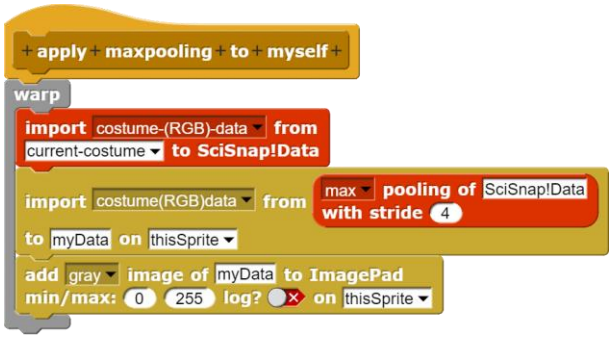


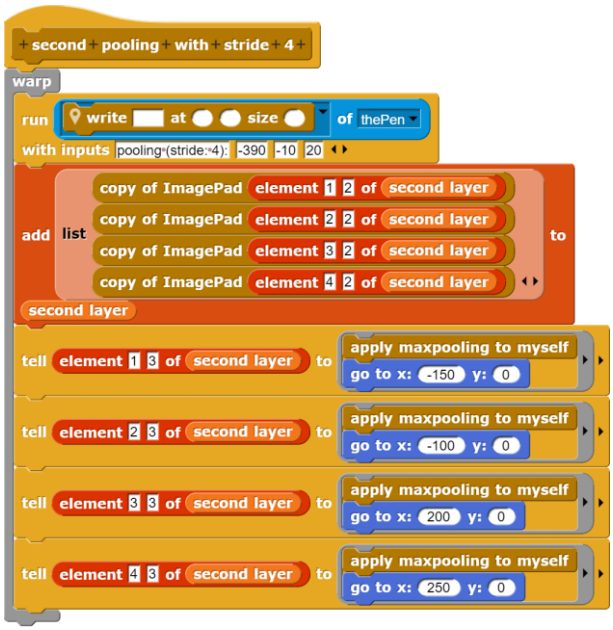
Danach muss jede Kopie eine *reLU* (*rectified linear unit*) durchlaufen, die als Aktivierungsfunktion dient. In diesem Fall werden einfach negative Werte auf Null gesetzt.



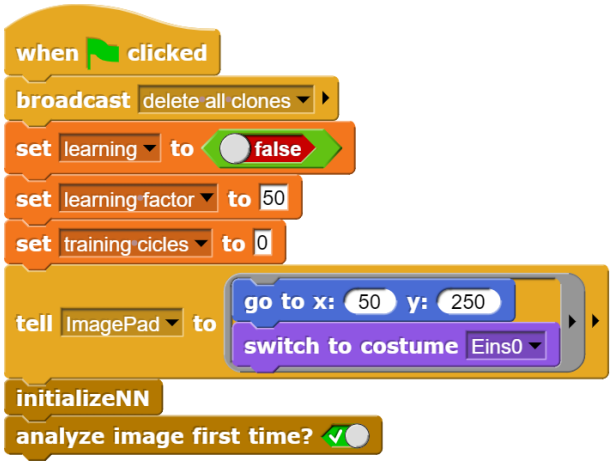


Zum Schluss wird jeweils eine Pooling-Operation zur Reduktion der Datenmenge ausgeführt. Als Beispiel ist der Pooling-Vorgang mit den vier Sprites der zweiten Ebene angegeben.





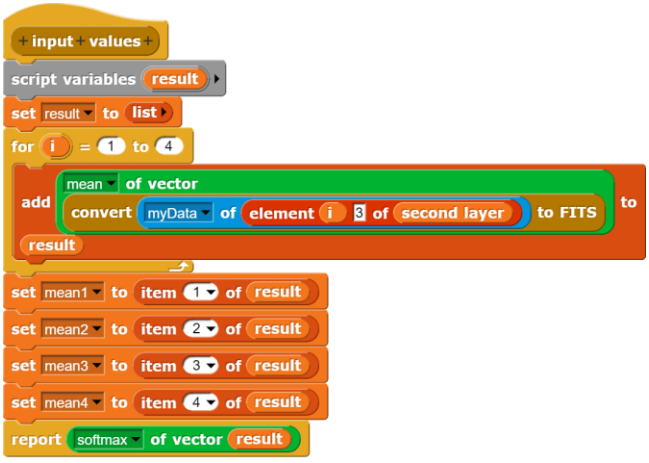
Alberto als Kontrolleur des Ganzen muss das *ImagePad* bitten, das Kostüm zu wechseln und dieses danach zu analysieren. Dabei hält er sich streng an die Vorgaben für CNNs.

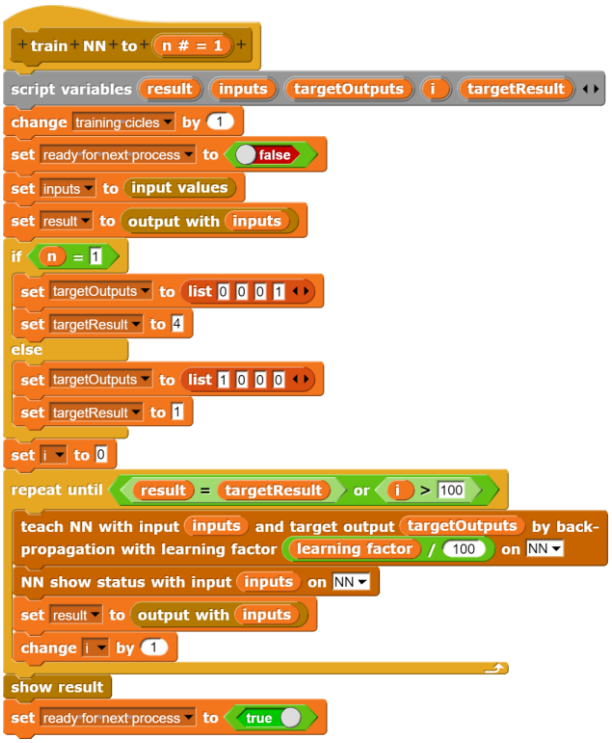


Die Methode *initialize* sorgt nur für das Zeichnen der Linien auf der Bühne. Die weiteren Methoden arbeiten mit zwei Ebenen des CNN, *first layer* und *second layer*, die jeweils die auf der Bühne erscheinenden Versionen der Ziffern enthalten. Damit sich die nicht gegenseitig stören, wird meist mit Kopien des *ImagePad* gearbeitet, nicht mit Klonen.

Nachdem die erforderlichen Kopien erzeugt wurden, werden diese von *Alberto* gebeten, die jeweilige CNN-Operation auszuführen. Zuletzt werden die inzwischen ziemlich mickrigen (4x4 Pixel) Klone der letzten Ebene als *„final feature maps“* stark vergrößert dargestellt. Mit diesen wird dann das Neuronale Netz trainiert.

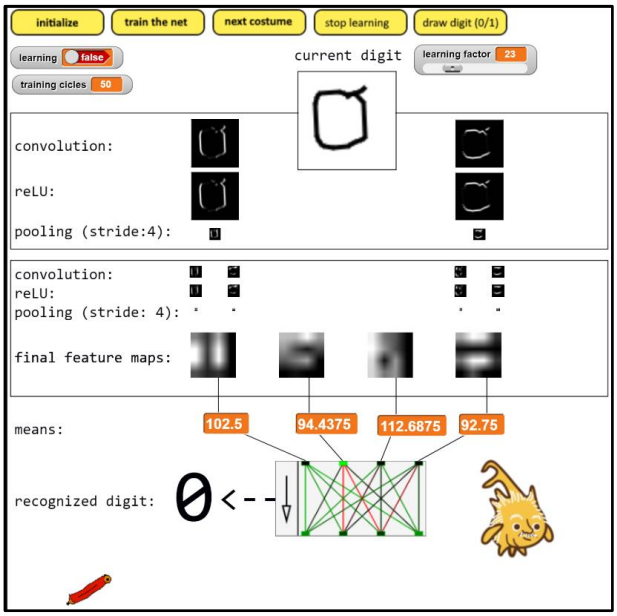
Das Neuronale Netz in Form eines *NeuralNetPads* soll bei Nullen die größte Ausgabe am Ausgang 1, bei Einsen am Ausgang 4 erzeugen. Das ist natürlich völlig willkürlich. Den aktuellen Ausgabewert ermittelt die Funktion *output with <input>*. Mit dessen Komponenten kann das Netz trainiert werden, wenn es uns gelingt, aus der letzten Ebene von *second layer* die Mittelwerte zu bestimmen. Diese modellieren wir noch passend mit der *softmax*-Funktion.





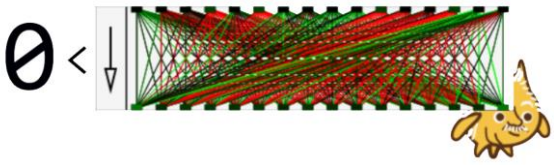
Und – hat das Netz was gelernt? Wir schreiben eine Ziffer:

Glück gehabt!



Aufgaben:

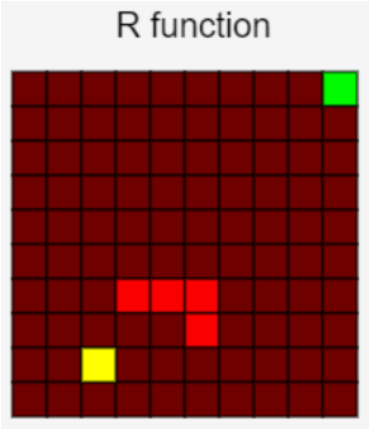
1. Generieren Sie aus den *final feature maps* eine Liste von 16 Werten.
2. Analysieren Sie diese Liste durch ein Neuronales Netz der Breite 16.
3. Testen Sie, ob die Erkennungsrate insbesondere von neu geschriebenen Ziffern steigt. Falls ja: wie begründen Sie den Effekt?
4. Experimentieren Sie auch mit mehrschichtigen Neuronalen Netzen.



11.13 Bestärkendes Lernen / Q-Learning

Beim *bestärkenden Lernen* (*reinforcement learning*) lernt ein *„Agent“* aus Erfahrungen, d. h. er führt *Aktionen* aus, auf die *Reaktionen* erfolgen. Aus diesen lernt er. Die Reaktionen müssen für jeden Zustand des Agenten in seiner Umwelt und jede seiner Aktionen in dem Moment zur Verfügung stehen, in dem der Agent die entsprechende Aktion ausführt. Meist werden sie als Belohnungen (*reward*) bezeichnet, und sie können positiv oder negativ sein. In ihrer Gesamtheit definieren Sie das Ziel des Lernprozesses. Eine Möglichkeit des bestärkenden Lernens ist das Q-Lernen. Informieren Sie sich in der Literatur zu den Themen *Reinforcement Learning*, dem *Optimalitätsprinzip von Bellman*, *Markow-Ketten* und *Q-Learning*.

In unserem Beispiel wollen wir einen kleinen Roboter (als gelbes Quadrat) in einer Umwelt (realisiert als Gitter) als Agenten herumfahren lassen, wobei er bei Bedarf eine Ladestation (grünes Quadrat) anfahren soll, ohne dabei durch das Tulpenbeet (rote Quadrate) zu pflügen. Dieses *Ziel* formulieren mithilfe einer Belohnungsfunktion (*R-function*), die für jedes Gitterelement eine Belohnung bereithält, die der Roboter bekommt, wenn er das Feld betritt. Für „normale“ Felder definieren wir die Belohnung als *-1* (wegen der notwendigen Fahrtstrecke), für die Ladestation als *+10* (sic!) und für das Tulpenbeet als *-15* (weil es dort beim Betreten gewaltig Ärger gibt). Versucht der Roboter den Garten zu verlassen, erhält er zur Strafe *-10*, bleibt aber im Feld.

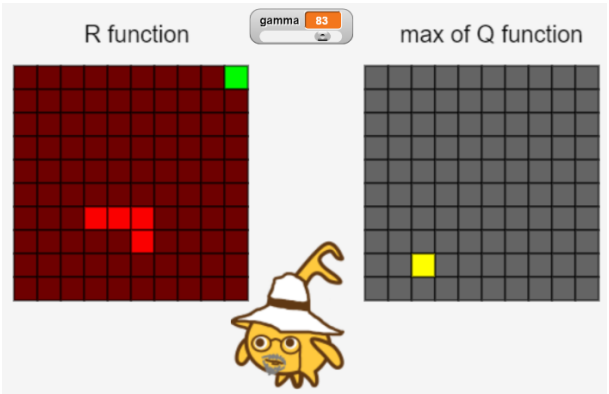


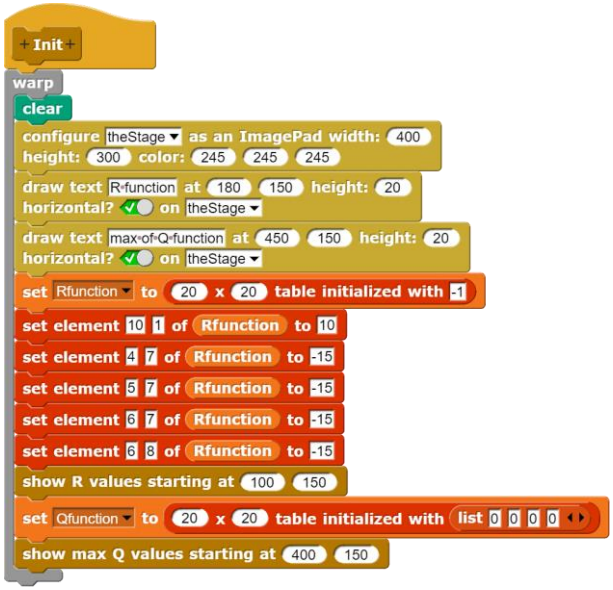
Die Aktionen, die der Roboter ausführen kann, beschränken wir auf Schritte nach links, rechts, oben und unten (in dieser Reihenfolge). Für jede dieser Aktionen muss jetzt entschieden werden, was passiert. Wir wählen als *Strategie* die *Q-Funktion*, die versucht, den *Gewinn* des Roboters zu maximieren, also negative Belohnungen klein zu halten und positive zu sammeln. Der Roboter wählt jeweils die Aktion, die ihm maximalen Gewinn verspricht. Lesen Sie dazu in der Literatur nach!

$$Q(Zustand, Aktion) = Belohnung(Zustand, Aktion) + \gamma \cdot Q(Folgezustand, optimale\ Aktion\ im\ Folgezustand)$$

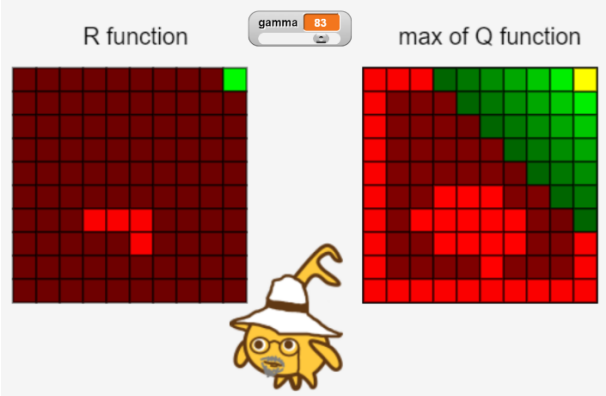
(γ ist ein Dämpfungsfaktor zwischen 0 und 1)

Wir realisieren unser Projekt in einem 10x10-Gitter, das wir auf der Bühne darstellen, die wir als ImagePad konfigurieren. Neben die R-Funktion zeichnen wir den (positiven oder negativen) Maximalwert der vier Q-Werte in der gleichen Farbdarstellung. Den Roboter transferieren wir dabei auf die Seite der Q-Funktion.



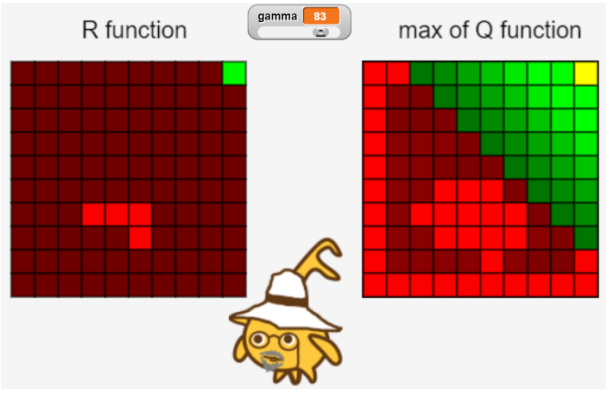


Roby muss nun erstmal seine Umgebung erkunden, d. h. er läuft im Gitter herum und berechnet jeweils die optimalen Q-Werte, soweit das mit den bisherigen Daten möglich ist. Damit das nicht zu lange dauert, lassen wir ihn von oben rechts an schleifenförmig die Zeilen des Gitters durchlaufen. Nach einem Durchgang stellt sich die Q-Funktion wie folgt dar.



Man sieht, dass die Information über die positive Belohnung oben-rechts von dort aus im Gitter „zerlaufen“ ist, also sich abgeschwächt auf die umliegenden Zellen verbreitet hat. In deren Q-Funktion ist jetzt die Information gespeichert, in welche Richtung sich Roby möglichst bewegen sollte, um Erfolg zu haben. Auch die Umgebung des Tulpenbeets wurde als äußerst unangenehm markiert, und dort, wo sonst nichts los ist, werden wenigstens die negativen Erfahrungen mit dem Rand erfasst.

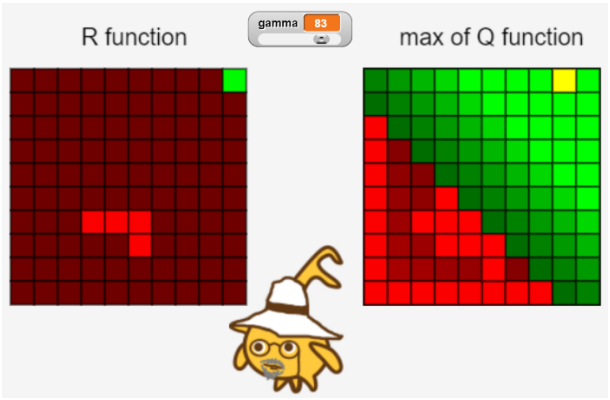
Machen wir noch einen Durchgang!



Die positiven Nachrichten haben sich (wegen des Dämpfungsfaktors) langsam weiter verbeitet, während es im Rest des Feldes nicht mehr schlimmer kommen kann.

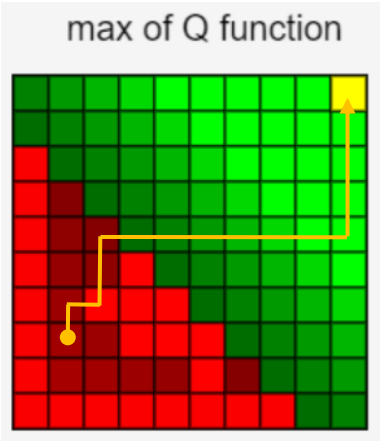
Von diesen Suchläufen können wir noch ein paar dranhängen.

Nach mehreren Durchgängen ändert sich nichts mehr an der Darstellung. Die Q-Funktion ist konstruiert. Wir können jetzt testen, ob Roby schnell zur Ladestation findet, ohne unterwegs großen Schaden anzurichten.



Der Befehl

führt zu folgendem Weg:



Roby hat also gelernt, schnell zum Ziel zu kommen, ohne Schaden anzurichten. Hilberto freut sich über den Kleinen!

Hinweise

1. *SciSnap!* ist nicht gerade für kleine Displays gedacht, aber auf einem größeren Monitor läuft es prima. 😉
2. Die Beispiele in diesem Skript sollen vor allem unterschiedliche Einsatzmöglichkeiten der *SciSnap!*-Bibliotheken zeigen. Es ist nicht ihre Aufgabe, Beispiele für guten Unterricht oder gute Lehre zu geben, aber sie geben hoffentlich Hinweise, auf welcher Ebene gearbeitet werden kann.
3. Dementsprechend fehlen in diesem Skript weitgehend Beispiele, anhand derer die Lernenden eigene Problemfelder und Lösungen finden und bearbeiten können. Sollten Sie da „best-practice“-Beispiele haben, dann wäre ich auf Hinweise darauf dankbar. Vielleicht könnte eine Sammlung daraus entstehen.
4. Die Bibliotheken enthalten sicherlich noch Fehler und Verbesserungsmöglichkeiten. Für Hinweise darauf wäre ich ebenfalls dankbar.

Ansonsten: Legen Sie los!

Literaturhinweise und Quellen

[ABELSON] Abelson, Sussman, Sussman: Structure and Interpretation of Computer Programs, MIT Press

[Census] https://archive.ics.uci.edu/ml/datasets/census+income

[DBV] Burger, W., Burge, M.-J-: Digitale Bildverarbeitung – Eine Einführung mit Java und ImageJ, Springer 2006

[FITS] de.wikipedia.org/wiki/Flexible_Image_Transport_System

[HOU] Hands-On Universe: handsonuniverse.org/

[HR] https://studylibde.com/doc/2985884/hertzsprung-russell--und-farb-helligkeits

[JSON] Popular Baby Names: https://catalog.data.gov/dataset/most-popular-baby-names-by-sex-and-mothers-ethnic-group-new-york-city-8c742

[NYcitibike] https://www.citibikenyc.com/system-data

[SchulAstro] www.schul-astronomie.de

[SQL] Modrow, Eckart: Informatik mit Snap!, http://ddi-mod.uni-goettingen.de/InformatikMitSnap.pdf

[UniGOE] Institut für Astrophysik, Universität Goettingen

[Wolfram] Wolfram, Stephen: A new kind of science, Stephen Wolfram LLC, 2002