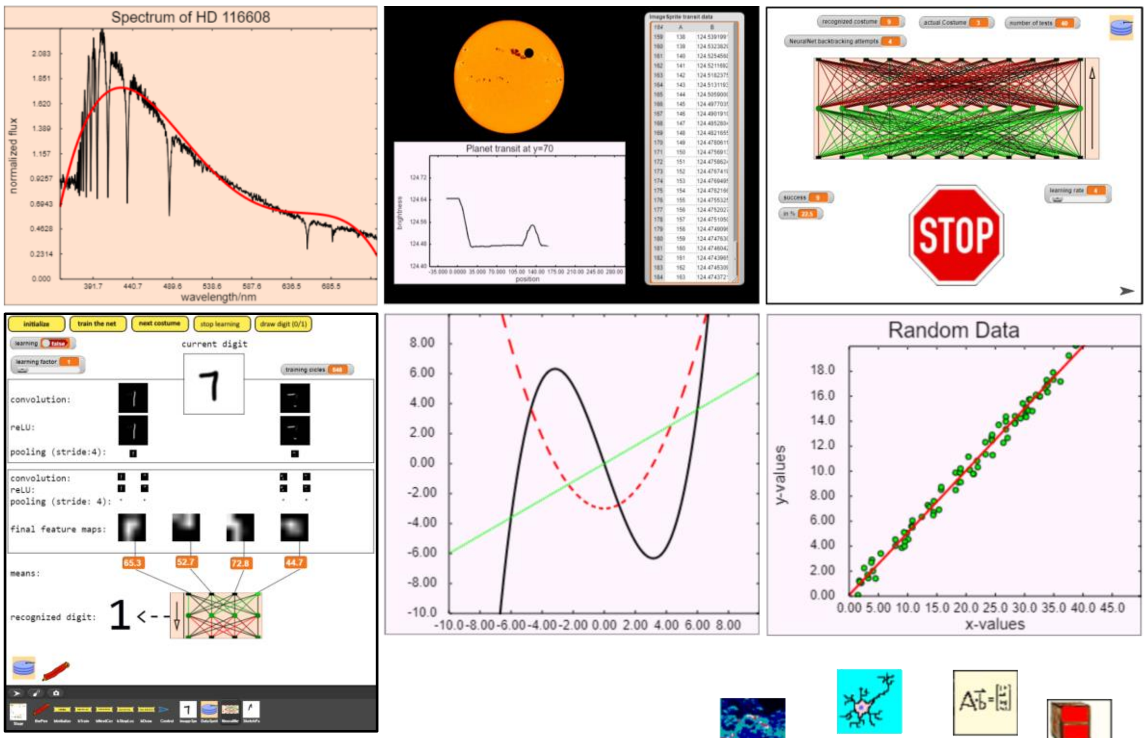
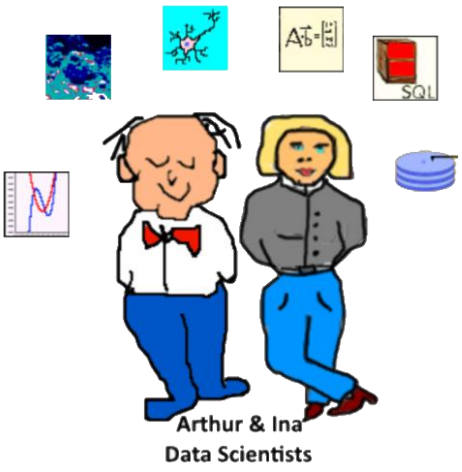
Eckart Modrow

Machine Learning with Snap! – ML.Sprites





emodrow@informatik.uni-goettingen.de

Prof. Dr. Modrow, Eckart:

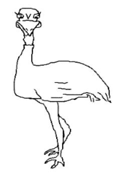
Machine Learning with *Snap!* – ML.Sprites

© emu-online Scheden 2020

All rights reserved

If this book is helpful for you and you would like to express your appreciation in form of a donation, you can do so at the following PayPal account:

emodrow@emu-online.de
Intended use: ML-Book



Preface

This script describes the *ML.Sprite-Library* with *Snap!* blocks, which is intended for (relatively) fast processing of large amounts of data. "Large" data volumes are almost never used in schools and initial university education - because they were hardly freely available some time ago, and money is scarce in education. In the meantime, however, there are large amounts of data in abundance, be it as a data collection on the Internet or as image files, because they are also "large". Education thus has the chance to deal with relevant data and thus find numerous points of contact with the field of "computer science and society". In the long run they are more important than any programming tricks in terms of general education.

Especially for beginners it is important to "see" what they are doing with their programming attempts. *Snap!*'s fantastic visualization capabilities are complemented by the *ML.Sprite-Library*, which includes library functions for graphics and images that, like the *Snap!* tables, quickly display the results of operations. Speed is important in this area because it supports experimental work in trial and error style. If you must wait too long, you won't try that much. The *ML.Sprite-Library* supports this approach by implementing most time-critical functions in JavaScript. Besides, these blocks also show how text-based programming can be senseful integrated into a graphical development environment.

The *ML.Sprite-Library* contains blocks from the area of data visualization and table handling, which is supported by the introduction of the data type *table*. In addition, functions of linear algebra with the data types *vector* and *matrix*, the solution of linear systems of equations and interpolation by polynomials are available. SQL queries are integrated, neural networks from perceptrons can be easily created and trained, image operations can be quickly executed via kernels as well as through vector and matrix operations. The examples show how this can be done. But they always show only one way - invent others and better ones for yourself!

Unlike the first version, the *ML.Sprite-Library* is divided into six prototypes, which can be loaded as attached parts of the overall sprite called "*Arthur&Ina*" or as individual parts. This limits the number of visible blocks. You can limit yourself to the prototypes that are currently needed. Each prototype contains a small example script that illustrates its use.

This book is a translation from German. Unfortunately, I do not speak English well, so it will be bumpy. I apologize for that. Be strong and hold it! Many thanks for the wonderful help of the *DeepL*[1] translation program. I would probably never have finished without these.

I would like to thank Jens Mönig and Rick Hessman very much for their support and the numerous discussions.

I wish you a lot of fun working with *Snap!* and the prototypes of the *ML.Sprite-Library* from Arthur and Ina!

Goettingen, 13 January 2020

[illegible]

[1] https://www.deepl.com/translator

Content

1 Artificial Intelligence and School

The term "artificial intelligence" is currently and for the foreseeable future more than current. In Germany, the Year of Science 2019 has been declared the "Year of Artificial Intelligence". In the field of "digitalization", the term is shaping discussions in the media, business and politics. Informatic didactic contributions are also increasingly being made on the subject.

In school informatics the topic is not really new. For three to four decades now, there have been examples of neural networks (NNs) suitable for use in schools, for example, which are developed and trained by the pupils themselves [Baumann] [Modrow1]. Such networks are clear and easy to understand, encourage students to work independently and then to discuss philosophical implications based on their professional experience [Modrow2]. Above all, however, they are small. This is exactly the difference to the current NNs: they are big. According to Ian Goodfellow [Deep Learning], one of the leading developers in this field, basically nothing has changed compared to the old small networks. The structure and methods have (almost) remained the same, but of course they have been improved. What has changed is the performance of the computers on which the NNs run and the amount of data available to train them. This, however, leaves older findings valid, such as Marvin Minsky's [Minsky] 1967 findings on the equivalence of NNs and finite automata. The result is no wonder, because the model of finite automata has its roots in the first NNs. However, such results help to classify a topic: if the term "learning" tends to bring brains to mind, the finite automaton tends to bring the field to the level of vending machines.

But we have a problem with that. In school, real applications are usually reduced to small model systems that still show some of the original properties. If, however, the property of being large in current neural networks is what makes the difference to earlier versions, the restriction to small networks is at least questionable. Such a thing could have been - and has been - done some decades ago. So, what is new about this topic?

The suggestions for treating large NNs in class often consist of training finished NNs using finished training data. Students then watch the net learn, slowly improving its results. You don't need a real NN for this experience, a video was enough. You can't see that the net is big and you can't see why this size is important from watching it. All you can see is that the results are improving. You don't learn anything from this experience alone from NNs. A discussion of the effects of NNs then is based on the information that they exist and that they can learn. Further technical basics are missing, so that this discussion could take place just as well in other subjects.

Let us compare the situation with an example from physics. The relatively new image of a black hole [SZ] shows that there are black holes and that they "swallow" matter. However, this information alone does not integrate the topic into the physics lesson, because a technical treatment of black holes is largely beyond the possibilities of the school. But within a subject area "gravitation", which contains numerous activities, historical and social references, typical problems of school physics, etc., the picture links school physics with "science after school", shows ways to a more profound occupation with it and, for example, encourages reflection on whether the learners see a personal perspective in this area - or not.

What do we learn from this?

The pure introduction of new technologies has no place in school - there are other channels for shows. The pure information that such technologies exist is also not enough to assign the topic to a specific subject. On the contrary, if you limit yourself to that, then it would be better to locate subjects in which, for example, the social or philosophical effects are discussed, and the topic is thus networked with other aspects. Only the didactic reduction of a question to a complexity level, on which the learners can work as independently and imaginatively as possible, makes the topic pedagogically fruitful.

In the field of artificial intelligence, it is not the passive observation of the learning of networks in schools that is important, but the active promotion of the understanding of human learners for the fundamentals and implications of this process.

New for the school are the tools we can use today. The visualization possibilities on the one hand and the use of powerful libraries on the other hand make it possible for the learners to explore simple first approaches on their own and thus experience the consequences of this expansion. In this work the meaning of the terms used becomes clear and thus assessable. The term "learning" has, for example, in the field of machine learning largely the meaning of "parameter adjustment". This does not quite correspond to the common meaning - and thus its use in the media, for example. The number of parameters in e.g. small perceptron networks increases immensely with their expansion - and thus the time required for training as well as the number of training data required. Extrapolation to really large networks therefore raises the question where these resources come from. When it comes to training duration, the parallelizability of the algorithms and the speed of the computers is decisive, but when it comes to data volumes, their sources - and that is often us - are decisive. So, we can't be indifferent to the application of AI systems, it directly leads to problems with data protection and is therefore a current socio-political topic. This also applies to the data itself. Working on machine learning topics quickly teaches us that there is not much to do with the freely available data itself. It often only becomes interesting when different data sources are linked. However, this linkage is usually not made by statistical quantities, but by the individual sources themselves - i.e. us. For example, does the spread of a disease have anything to do with the behaviour of population groups? We can actually only answer this conclusively when we know whether the disease occurs more frequently in precisely these people than in others. Otherwise, we will not get very far beyond suspicions.

Working with machine learning requires learning time - and this is only available to a limited extent. The more powerful the instruction set used, the more time is left for the question of the consequences. The ML.Sprite-library is designed to free up this time.

One more remark on this: I think that besides the usual learning of technical contents and methods, especially in the field of computer science, a good deal of creativity belongs in the classroom. Computer science provides wonderful tools such as *Snap!* for this very purpose. If the school concentrates exclusively on teaching facts and data as well as practicing the application of calculations, there is the danger that the students never experience what it is like to discover and understand connections and backgrounds themselves or to find and test their own solutions for interesting problems. This would be a bit sad, because a chance to develop a creative personality, aware of its possibilities and limits, would be missed at least in this field. The goal of the ML.Sprite-library is therefore to provide a toolbox for the learners, which is suitable for their own projects in the field of machine learning. It is explicitly not the goal to provide ready-made solutions for certain problems.

2 Machine Learning

The term "machine learning" is often used as a synonym for "artificial intelligence" or "neural networks". However, this limitation is not true. For example, the definition found on the SAP page [SAP] is more precise:

Machine learning technology teaches computers to perform tasks by learning from data instead of being programmed for the tasks.

"Learning from data" can be understood as adapting the parameters of a function. A data set (image, table, character string, ...) is presented as input vector ***E*** to a machine. It calculates an output value *k* from this, which assigns the input to a category ("It is a cat", "Feature present" (or not), "The word 'car'", ...).

$$f(\boldsymbol{E}) = k$$

This assignment can take place in very different ways. For example, you can adjust the parameters of a polynomial, search for similar input values ("k-next neighbours"), work with decision trees, use Bayesian filters, ... - or even train an NN. All these methods have in common that the "machine" contains a set of parameters that can be changed. The machine "learns from data" by repeatedly reading in a data set, calculating the output value from this using the current parameter set, and then comparing this output with the "desired" output value using some method. If there is a deviation, it changes the parameters so that the output at least approaches the "desired" value. "Desired" values may be known in advance ("The image is a cat image"), may come from outside e.g. from a "trainer" ("supervised learning") or may be generated by the machine itself ("unsupervised learning"), e.g. by extracting features from many training data ("clustering"). In all cases, the machine does not "learn" anything, but adapts parameters according to a given procedure.

This approach, too, has long been widespread in schools. "Learning Nimm-games" etc. can already be found in the first computer science textbooks. What is new again is the scope of the required training data. A large NN can have billions of parameters that need to be trained - and this requires "a lot of" training data. Another new feature is that these data are available on the net. So, if applications available "for free" are paid "with data", then we now also know how and why this happens.

If you look at common textbooks on machine learning [Grus] [Albon], you won't find much about NNs, but a lot about data handling. These must be normalized, for example, in order to make the many input data, which can come from very different sources, compatible. For example, if we photograph many dogs with an older digital camera and many cats with a newer one, then an NN would very likely learn from these images that dog images are smaller than cat images.

The preparation of data now is a very manual activity. It can be done step by step, tested and then automated with simple algorithms. Testing is greatly facilitated if the structure of the data is easy to visualize, e.g. in tables or as a graph. And algorithms are simple if they have a clear structure, e.g. if, after some preparation steps, they consist of a loop in which some alternatives with the corresponding instructions are enumerated. The power of the developed scripts does not depend so much on the algorithmic structure as on the power of the available commands. Or vice versa: if you have enough powerful commands, you can do a lot with simple programs. The parameters then can be adjusted in one of the usual ways. If the appropriate tools are available, the preparation of data is a very suitable topic for schools. The *ML.Sprite-Library* is intended as such a tool.

The instruction sets of the *ML.Sprites* contain solutions for a number of typical beginner problems, e.g. sorting, drawing a graph or displaying an image in false colors. This does not mean that there is nothing left to do - you can only tackle more complex problems. So instead of sorting a list, you can look for solutions to problems that require sorting, among other things. Part of the work will consist of assembling sequences of library functions, testing them, and then making them available as a new block that can be used by other sprites to perform other tasks. The access to resources of other objects can serve as a training ground for object-oriented programming - but it does not have to. Instead, simple procedures such as data exchange using global variables can be used.

3 The Structure of ML.Sprites

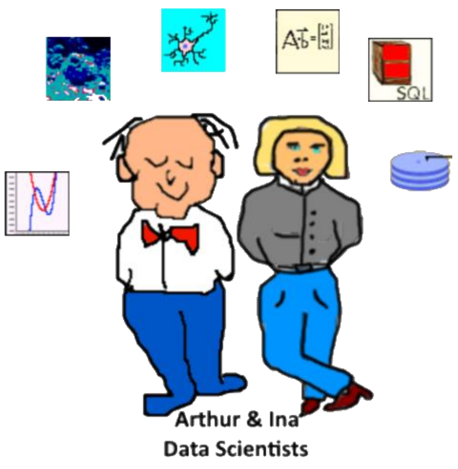
The structure of *ML.Sprites* is based on the idea of documented data sets consisting of two parts: the *metadata*, which describes the structure and context of the data (e.g. number format, image dimensions, recording device, recording date, ...) and the associated pure *data* segments. Metadata usually consists of dictionaries - names with assigned values (e.g. "Recording date: 24.12.2018"). Examples for this structure are FITS files [FITS], which are standard in astrophysics but are also used in the Vatican Library, or JPEG images from mobile phones. Also, here there are meta data (image size, compression degree, date of acquisition, often also GPS coordinates). Without these an image generation would not be possible. It is important that the image generation does not change the original data.

We adapt this structure by giving an *ML.Sprite* three local variables, each containing the data (*myData*), the data description (*myProperties*) and a collection basin for (error) messages of called blocks of the sprite (*myMessages*). These variables can be filled by importing data from different sources (SQL queries, text file, CVS file, JSON file, FITS file, direct assignment, ...), whereby the properties *myProperties* have to be adapted to the respective data. On the other hand, this can also be done "by hand". With the help of these properties, data can be converted into graphical representations (graph, data plot, histogram, image, ...), whereby either *myData* or another suitable table is selected as the source.

It is important that the image generation does not change the original data. If, for example, an image of Jupiter is used to determine the distances between its moons, then these must at least be visible in the image. Therefore, after adjusting some parameters, e.g. a false color image can be generated. In this image Jupiter itself will appear rather unstructured. If you want to examine the "eye" of the planet in more detail, the parameters must be selected completely differently, so that the moons are hardly visible. All these changes must be done in the pixels of the current *Snap!* sprite costume without affecting the image data itself.

Because tables can be displayed very nicely in *Snap!*, this display format is not additionally implemented. Therefore, the data type *table* is implemented with many of the operations commonly used in *data science* (table operations, correlation calculation, affine transformations, solving linear systems of equations, ...), which can handle larger amounts of data sufficiently quickly.

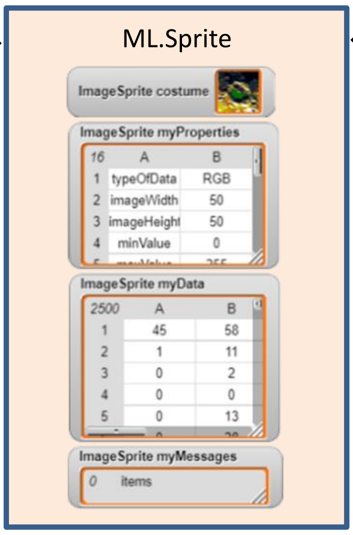
Since the library (currently) contains 118 new blocks, these have been grouped according to their functionality and distributed into a total of six sprites that can serve as prototypes for various tasks: a *DataSprite* for handling the actual data, an *ImageSprite* for image processing, a *PlotSprite* for graphical representations, a *NeuralNetSprite* for Perceptron nets, a *SQLSprite* for database queries and a *MathSprite* for linear algebra operations. In order to use them independently, some blocks were created in slightly different variants. A "pool" called *Arthur&Ina* contains these prototypes as parts. If you want to use them individually, they can be solved by *Arthur&Ina* and stored individually.



The overall structure is as follows:

imports data from …

- image files
- text files
- SQL-queries
- JSON files
- CSV files
- …



ImageSprite myProperties

16	A	B
1	typeOfData	RGB
2	imageWidth	50
3	imageHeight	50
4	minValue	0

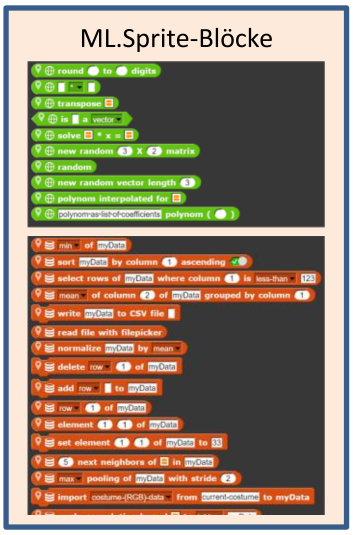
ImageSprite myData

2500	A	B
1	45	58
2	1	11
3	0	2
4	0	0
5	0	13

ImageSprite myMessages

0 items

provides blocks for the graphical representation of data, for editing tables, solving equation systems, statistical operations, ...



```
round ( ) to ( ) digits
[ ] [* ▾] [ ]
transpose
is [ ] a [vector ▾]
solve * x =
new random (3) X (2) matrix
random
new random vector length (3)
polynom interpolated for
[polynom-as-list-of-coefficients] polynom ( )

[min ▾] of [myData]
sort [myData] by column (1) ascending
select rows of [myData] where column (1) is [less-than ▾] [123]
[mean ▾] of column (2) of [myData] grouped by column (1)
write [myData] to CSV file [ ]
read file with filepicker
normalize [myData] by [mean ▾]
delete [row ▾] (1) of [myData]
add [row ▾] [ ] to [myData]
[row ▾] (1) of [myData]
element (1) (1) of [myData]
set element (1) (1) of [myData] to [33]
(5) next neighbors of in [myData]
[max ▾] pooling of [myData] with stride (2)
import [costume-(RGB)-data ▾] from [current-costume] to myData
```

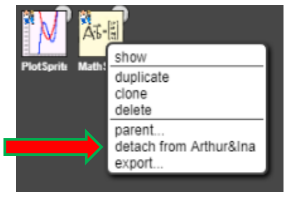
Most blocks get their parameters (image size, value ranges, colors, ...) from the dictionary *myProperties*. The properties preset with ***set properties*** make it possible to use blocks for creating graphics, diagrams, ... without too many parameters. If the values do not match, the properties are changed either individually (with ***set property***) or in groups (e.g. with ***set line attributes***).

The blocks of each sprite have a different symbol on the front to quickly distinguish them from each other and from the standard *Snap!* blocks. As soon as you can create your own new palettes in *Snap!*, the library blocks should be placed there. If this would already be done in an own *Snap!* variant, then you would either must adapt it continuously or decouple from the *Snap!* development. Both would be more unpleasant than the solution chosen now.

4 Working with ML.Sprite-Prototypes

4.1 Versions of ML.Sprites

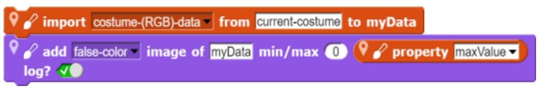
As experienced data scientists, *Arthur&Ina* have the full range of methods at their disposal. The corresponding prototypes buzz around their heads, and if you double click on them, you will reach the corresponding sprite. As we are professionally differently oriented than they are, we will hardly ever need all versions of the *ML.Sprites* at the same time - but which one we need depends on the chosen problem. Therefore, we can also solve all or single sprites of *Arthur&Ina* (right click on the sprite and select "*detach*") and save them individually ("*export*"). If we are lucky, someone has already done this, and we only import the needed prototypes (either with "*import...*" from the file menu or by "pulling" the file into the *Snap!* window). This is much faster than loading everything at once. After that we should set the stage to a suitable size (depending on the screen resolution) with "*Stage size...*" in the tool menu of *Snap!*, e.g. 800x600 pixels.



In each of the loaded sprites we find a small example that shows how they can be used. The new blocks are distributed on the usual palettes, mostly on *Looks*, *Operators* or *Variables*. They are located down there and have a common symbol to distinguish them from the standard blocks.

If we want, we can...

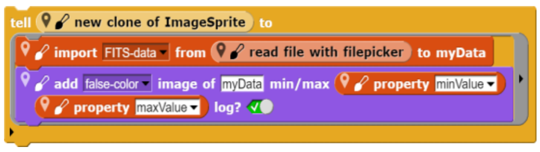
- ... work directly in the prototype, i.e. use the new blocks without further formalities.
  Example: The current costume, which was inserted from outside, for example, is read into the *myData* area and the corresponding properties are set. It is then printed in false-color and logarithmic representation, using the maximum value just determined during import.



```
import costume-(RGB)-data from current-costume to myData
add false-color image of myData min/max 0 property maxValue
log? ✓
```

- ... right click on the sprite to create a copy or a permanent clone of the prototype and work with it in the same way.

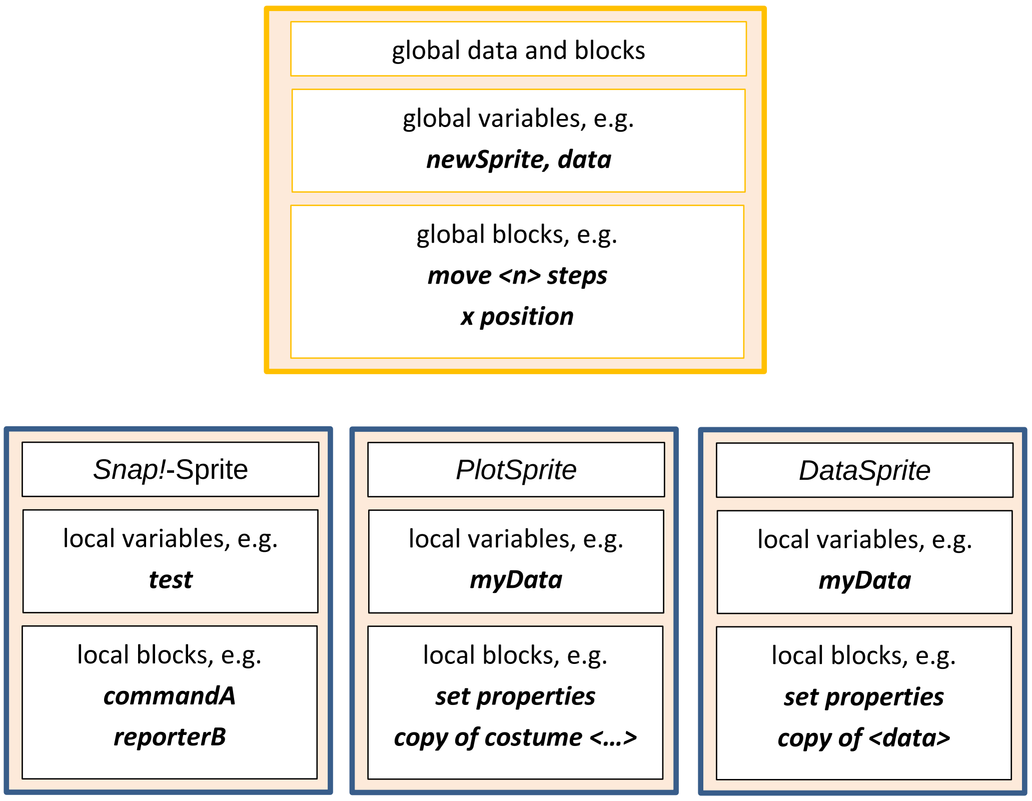
- ... create new temporary clones from the Control palette with the ***new clone of ...*** block and use them in the following.
  Example: First a temporary clone of the *ImageSprite* is created. This is asked to first read astronomical FITS data, whereby the file to be read is chosen by the user. Then the image is displayed as shown in the other example.



```
tell new clone of ImageSprite to
    import FITS-data from read file with filepicker to myData
    add false-color image of myData min/max property minValue
    property maxValue log? ✓
```

4.2 Access to ML.Sprites

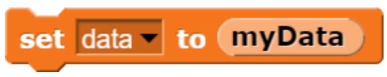
If you don't find the functionality you are looking for in the current *ML.Sprite*, the question arises how to access the data and/or methods of another *ML.Sprite* "from outside". Since these variables are all local, you cannot see them directly "from outside". On the one hand, this has the advantage of keeping the number of currently visible data and methods clear, but on the other hand it makes access more difficult. We will go through the different access options in sequence. We have to distinguish clearly between local and global data and methods. As an example, we choose the situation where a "normal" *Snap!* sprite wants to use the possibilities of two *Arthur&Ina* sprites, a *PlotSprite* and a *DataSprite*.



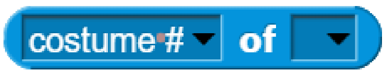
1. Access to the data of another sprite

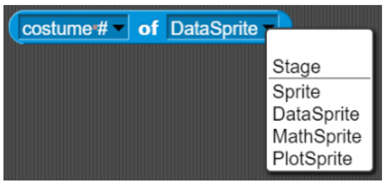
Example: The *Snap!* sprite needs the data of the *DataSprite*.

Solution 1: In the *DataSprite* you assign the value of the data (***myData***) to a global variable (***data***). This can be accessed directly everywhere, even in the *Snap!* sprite. (But this is not very elegant!)

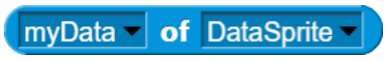


Solution 2: You can access directly the data of the *DataSprite* by using the ***of***-Block. First select the desired sprite in the right input field, here: the *DataSprite*. Afterwards, by clicking on the left input field the local variables (here: ***myData***, ...) and the local methods (here: set property, ...) are listed next to some standard attributes like position, size, ... (see next page). From these you can select the desired one.



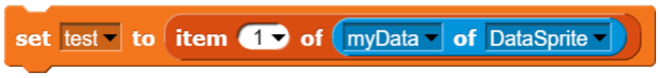


In our case it is the variable ***myData***.

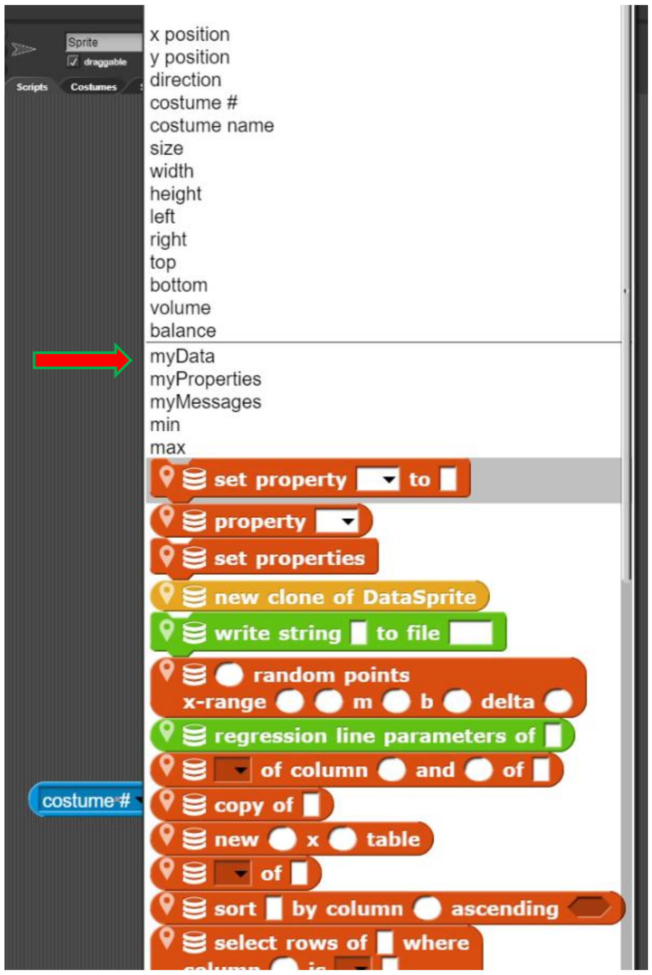


```
myData of DataSprite
```

You can work with this as usual, e.g. by assigning the first element to another variable.

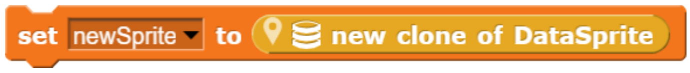


```
set test to item 1 of myData of DataSprite
```



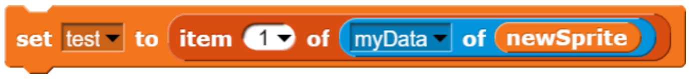
Example: The *Snap!* sprite needs the data of a clone of a *DataSprite*.

Solution: You bind the clone to a local or global variable. With this variable you can access the clone later.



```
set newSprite to new clone of DataSprite
```

In the following, we proceed as in the previous example, because only the properties of "named" sprites can be listed. Finally, replace the name of the selected sprite (here: *DataSprite*) with the variable that points to the clone.



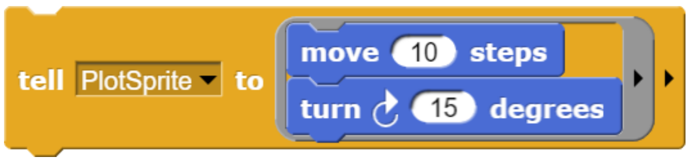
```
set test to item 1 of myData of newSprite
```

For methods we choose the following convention: global methods as well as methods of other sprites without parameters are called by ***tell*** and ***ask*** blocks, methods of other sprites with parameters are called by ***run*** and ***call*** blocks. (But sometimes we do it differently. 😉)

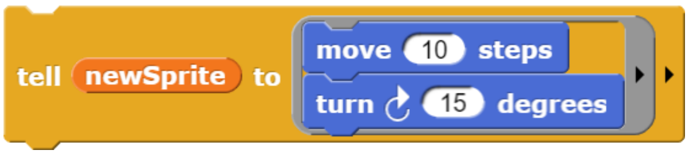
1. **Execution of a global method (command) by another sprite**

Example: The *Snap!* sprite tells the *PlotSprite* to move a little.

Solution: The *Snap!* sprite asks the *PlotSprite* to move by dragging the global block or blocks into the script area of a ***tell*** block. On the left side of the ***tell*** block the addressed sprite (here: the *PlotSprite*) is selected. The blocks are executed in the context of the other sprite, e.g. with its current position and direction.



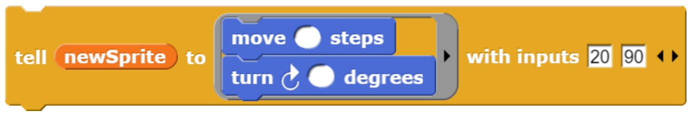
If the addressed sprite is a clone, proceed as above: replace the name of the plot sprite with the variable.



2. **Execution of a global method (command) with parameters by another sprite**

Example: The Snap! sprite tells the *PlotSprite* to move differently.

Solution: The ***tell*** block is expanded to the right by as many fields (small right arrow) as there are open parameters. The corresponding input fields for the methods must be completely empty!

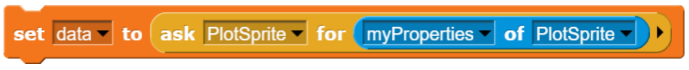


Note: There are other ways to do this and, above all, much more differentiated. Please read the Snap! manual.

3. **Call of a global method (reporter) by another sprite**

Example: The *Snap!* sprite asks for the properties of the *PlotSprite*.

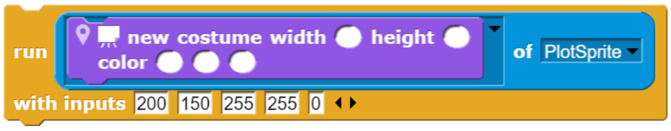
Solution: The ***ask*** block is used instead of the ***tell*** block. Otherwise as described above.



4. **Calling a local method by another sprite**

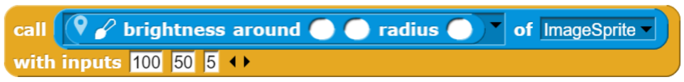
Example: The *Snap!* sprite changes the size and color of the *PlotSprite*.

Solution: The ***run*** block is used with a local method that is selected using the of block.



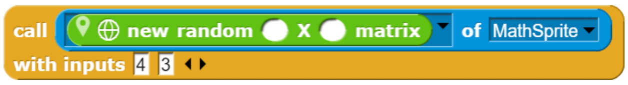
```
run (new costume width ( ) height ( ) color ( ) ( ) ( )) of PlotSprite
with inputs 200 150 255 255 0
```

Example: An *ImageSprite* is asked to determine the total brightness around the specified point (100|50) in a radius of 5 pixels.

Solution: 

```
call (brightness around ( ) ( ) radius ( )) of ImageSprite
with inputs 100 50 5
```

Example: A *MathSprite* is asked to provide a 4x3 matrix with random numbers.

Solution: 

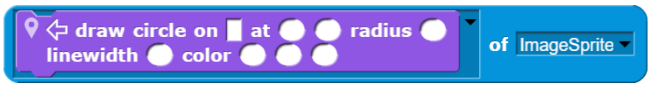
```
call (new random ( ) X ( ) matrix) of MathSprite
with inputs 4 3
```

5. **Calling the code of a local method by another sprite**

Example: The *PlotSprite* wants to execute the method of the *ImageSprite* to draw a circle on its own costume.

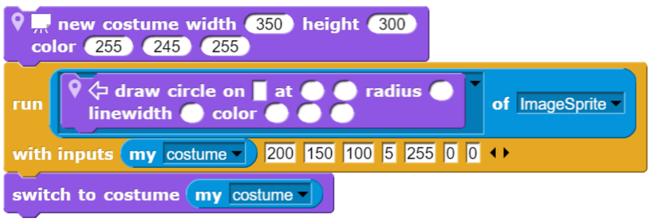
Solution: If local methods do not depend on local variables and/or other local methods, you can export the code and execute it in another context. This is typically the case with JavaScript functions. The drawing operations of the *ImageSprite*, which are sometimes also needed by other prototypes, serve as examples here. Since we cannot store global methods for the chosen way of binding functionality to sprites, the drawing methods of the *ImageSprite* are available in a second version, which has been marked as "*exportable*". Since the properties of the *ImageSprite* can no longer be accessed here, the number of required parameters "explodes". But for this you can export the code. 😉

In *PlotSprite* the code of the drawing function is accessible via the ***of***-Block.



```
(draw circle on [ ] at ( ) ( ) radius ( ) linewidth ( ) color ( ) ( ) ( )) of ImageSprite
```

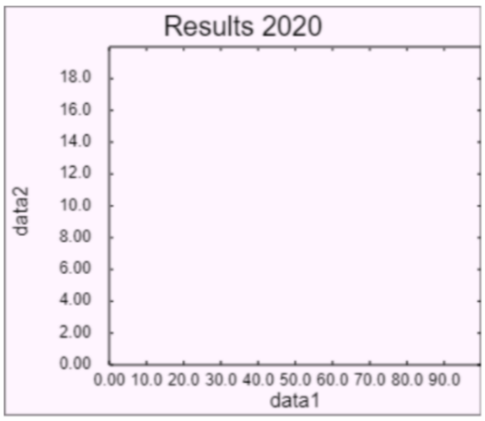
This code is executed in the context of the *PlotSprite*, whereby the required parameters must all be listed. Afterwards the changed costume is displayed again.

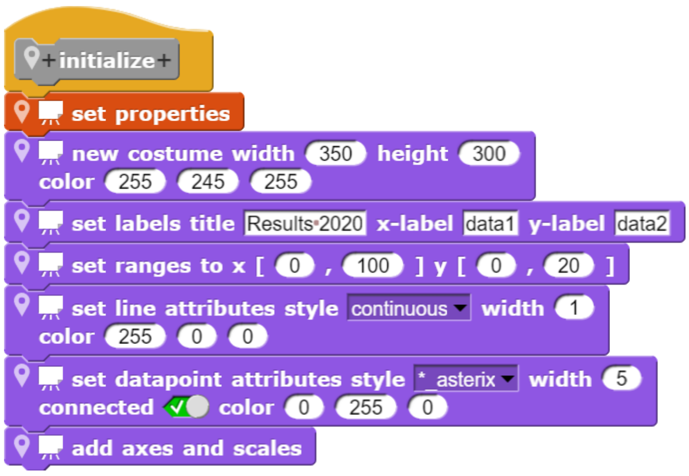


```
new costume width (350) height (300) color (255) (245) (255)
run (draw circle on [ ] at ( ) ( ) radius ( ) linewidth ( ) color ( ) ( ) ( )) of ImageSprite
with inputs (my costume) 200 150 100 5 255 0 0
switch to costume (my costume)
```

If you have to combine several such calls, it becomes a bit cumbersome. It is therefore recommended to write a local method within the addressed sprite that triggers the required actions. This method is then called from outside.

Example: The specifications for a diagram are set in *PlotSprite*. In another sprite you can simply call this method.

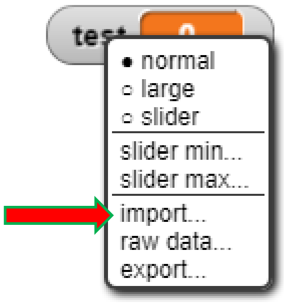




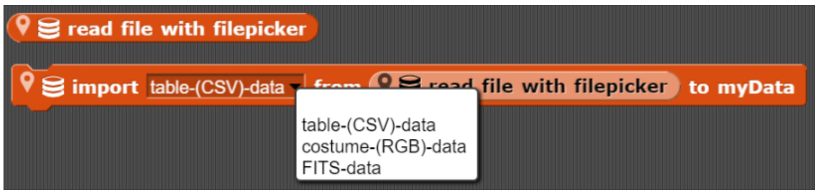
```
+initialize+
set properties
new costume width 350 height 300
color 255 245 255
set labels title Results 2020 x-label data1 y-label data2
set ranges to x [ 0 , 100 ] y [ 0 , 20 ]
set line attributes style continuous width 1
color 255 0 0
set datapoint attributes style * asterix width 5
connected ✓ color 0 255 0
add axes and scales
```

4.3 Importing and exporting data

Snap! can import a range of data formats directly. This can be done by "*dropping*" the corresponding files on the *Snap!* window or by right-clicking on a variable watcher[2]. Both works well with text, CSV and JSON files. Other text file formats such as FITS can also be imported in this way, where you are asked if you are serious. Exporting works the same way. If you want to do the same programmatically, use the reporter block ***read file with filepicker***. A file manager window appears, where you select the file as usual. Afterwards the data will be imported.

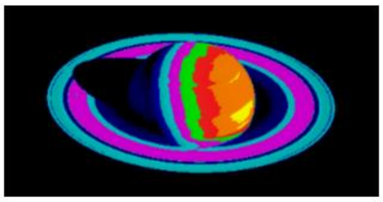


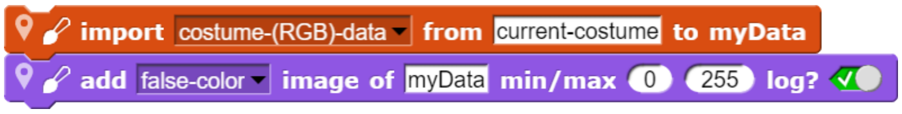
The main task is then to assign this data to the ***myData*** variable and set the corresponding properties in ***myProperties***. This is done by the following block, which imports data from outside into the ***myData*** area. This can be image data, table data or the data of the current costume. The costume is stored as a table of RGB values.



```
read file with filepicker

import [table-(CSV)-data ▾] from (read file with filepicker) to myData
```

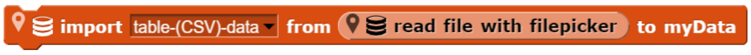
Example: The *ImageSprite* saves an image (source: [NASA]) and displays it with false colors.



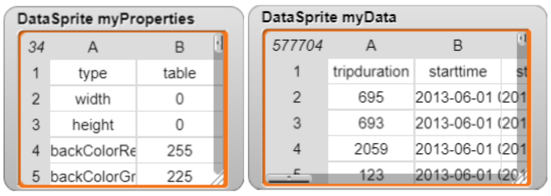


```
import [costume-(RGB)-data ▾] from [current-costume] to myData
add [false-color ▾] image of [myData] min/max (0) (255) log? ✓
```

Example: Almost 600.000 data records from a CSV file are read in about 10 seconds. The properties are set.



```
import [table-(CSV)-data ▾] from (read file with filepicker) to myData
```



DataSprite myProperties

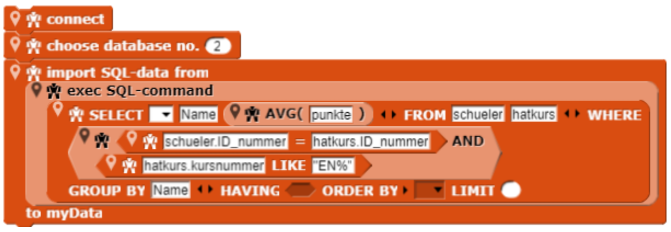
34	A	B
1	type	table
2	width	0
3	height	0
4	backColorRe	255
5	backColorGr	225

DataSprite myData

577704	A	B	
1	tripduration	starttime	st
2	695	2013-06-01 (201	
3	693	2013-06-01 (201	
4	2059	2013-06-01 (201	
5	123	2013-06-01 (201	

Example: SQL-import

If we have access to an SQL server, we can also read in data from there. In our case we use an *SQLSprite* to import the results of a query into the variable ***myData***. The data is converted into a table and its relevant properties such as number of columns and rows, ... are reset.

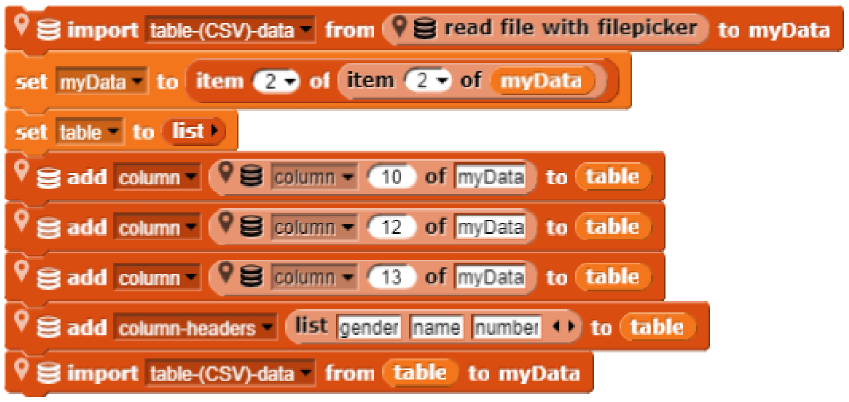


```
connect
choose database no. (2)
import SQL-data from
  exec SQL-command
    SELECT [ ▾] [Name] (AVG( [punkte] )) ◂▸ FROM [schueler] [hatkurs] ◂▸ WHERE
      < <[schueler.ID_nummer] = [hatkurs.ID_nummer]> AND
        <[hatkurs.kursnummer] LIKE ["EN%"]> >
    GROUP BY [Name] ◂▸ HAVING < > ORDER BY ▸ [ ▾] LIMIT ( )
to myData
```

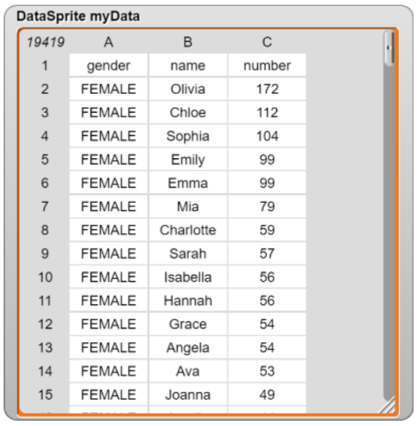
[2] You get a variable-watcher, if you set a checkmark in the box beside the variable.

Example: JSON-import

Again, the easiest way is to simply "*drop*" a JSON file into the *Snap!* window. But it also works automatically. First of all, we look for interesting JSON data and of course choose the statistics of baby names in New York City - what else. The appropriate block for this is again ***import** <table data> **from <read file with filepicker> to myData***. The result is a list with two columns and two rows, the metadata and the actual data. Because we are interested in these, we replace the original data with the element (2|2) of the table. Of course, we looked at the individual elements in table form beforehand to check what we loaded. Of the many columns, we copy the three interesting ones into a new table, add column headings and import the result back into ***myData***.



```
import table-(CSV)-data from read file with filepicker to myData
set myData to item 2 of item 2 of myData
set table to list
add column column 10 of myData to table
add column column 12 of myData to table
add column column 13 of myData to table
add column-headers list gender name number to table
import table-(CSV)-data from table to myData
```



DataSprite myData

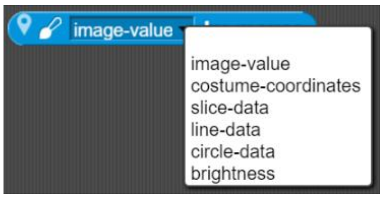
19419	A	B	C
1	gender	name	number
2	FEMALE	Olivia	172
3	FEMALE	Chloe	112
4	FEMALE	Sophia	104
5	FEMALE	Emily	99
6	FEMALE	Emma	99
7	FEMALE	Mia	79
8	FEMALE	Charlotte	59
9	FEMALE	Sarah	57
10	FEMALE	Isabella	56
11	FEMALE	Hannah	56
12	FEMALE	Grace	54
13	FEMALE	Angela	54
14	FEMALE	Ava	53
15	FEMALE	Joanna	49

The result: 19419 baby names

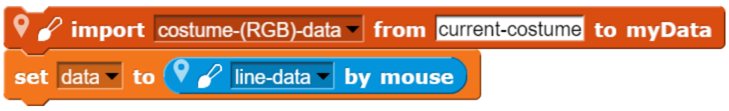
Who would have thought it!

Example: Importing data with the mouse

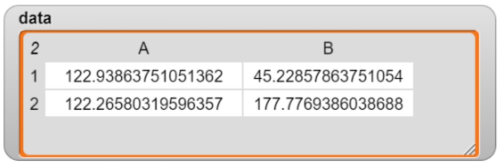
In many cases it is advantageous, especially with images, to read in data with the mouse. The *Sensing* palette of the *ImageSprite* and partly also of the *PlotSpite* offers blocks such as ***<...> by mouse***, with which image values, image coordinates, coordinates in the used coordinate system for graphs and/or data points, the data on a section through the image, starting and end point of a line, center and radius of a circle and the summed brightness values together with their number in a circle can be determined. As an example, the height of ancient columns is to be measured. Therefore, the costume image of the *ImageSprite* with the columns is imported and then measured with the mouse (yellow line).



Example: Measuring distances on an image

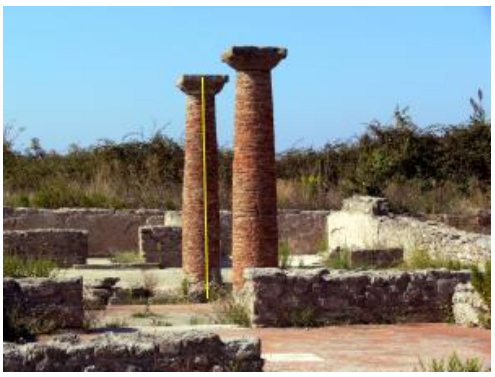


```
import costume-(RGB)-data from current-costume to myData
set data to line-data by mouse
```

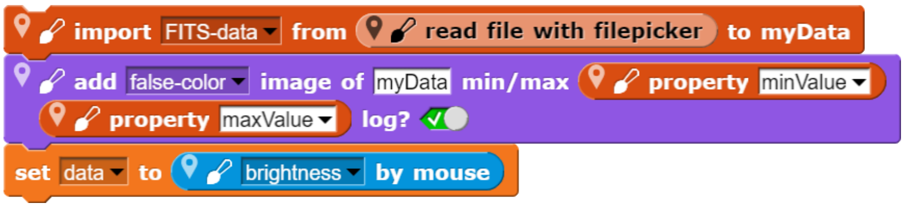


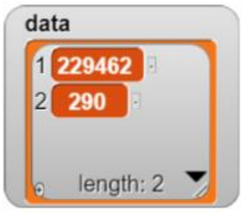
data

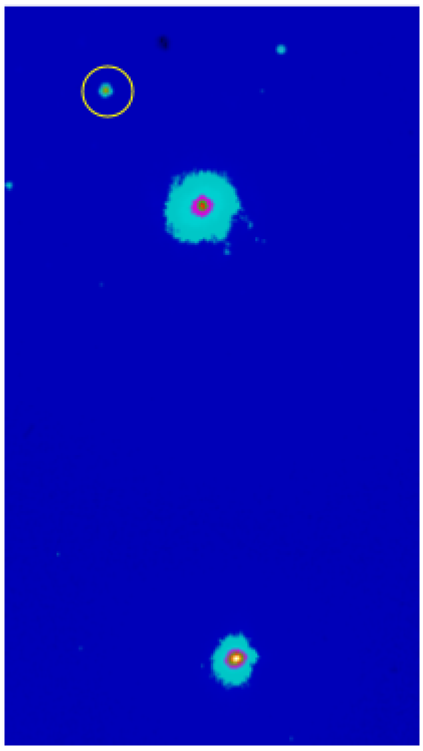
2	A	B
1	122.93863751051362	45.22857863751054
2	122.26580319596357	177.7769386038688



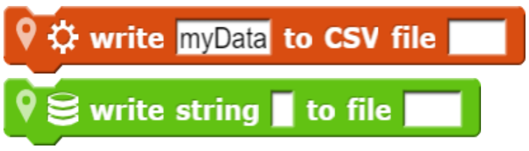
Example: Measure the total brightness around a pixel in an image (Source: [HOU])


```
import FITS-data from read file with filepicker to myData
add false-color image of myData min/max property minValue
    property maxValue log? ✓
set data to brightness by mouse
```


```
data
1 229462
2 290
length: 2
```

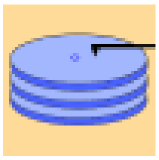


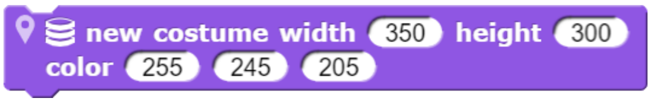
The export of data can be done directly from a Variable-Watcher.


```
write myData to CSV file
write string to file
```

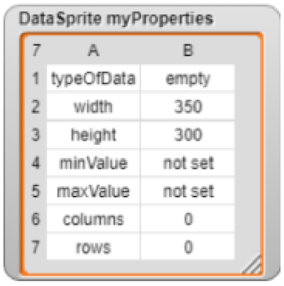
For scripts there are two new blocks ***write** <table> **to CSV file** <file-name>* and ***write string** <string> **to file** <filename>*. The results will end up in the download folder of the browser, as usual in *Snap!*. The two blocks allow you to automate data exchange with spreadsheet programs or text files, for example, to save the results of data processing.

4.4 The DataSprite

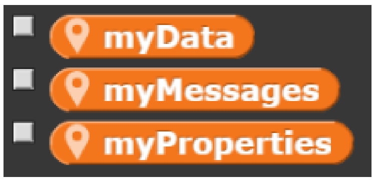




All *DataSprites* contain a block in the *Looks* palette that creates a costume of the specified size with the RGB values of the background color and switches to it. They also all have the same set of local variables: ***myData*** contains the actual data, ***myProperties*** its metadata, and ***myMessages*** any messages generated when the blocks are executed - usually error messages. If something does not work, you should look there.

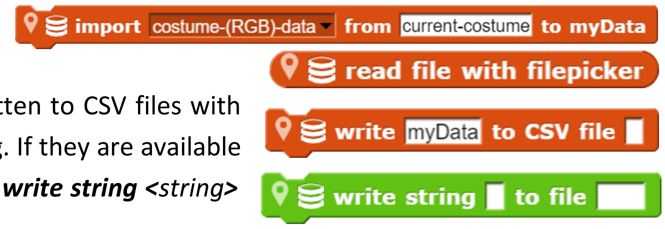

DataSprite myProperties

7	A	B
1	typeOfData	empty
2	width	350
3	height	300
4	minValue	not set
5	maxValue	not set
6	columns	0
7	rows	0

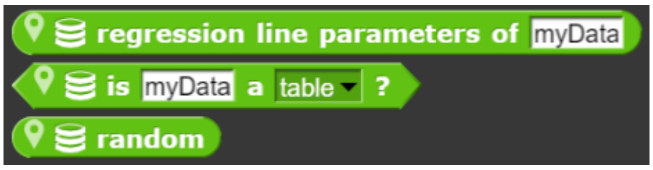


The *DataSprite* is used to manipulate table data. The properties are adapted accordingly: they describe the current state of the stored table.

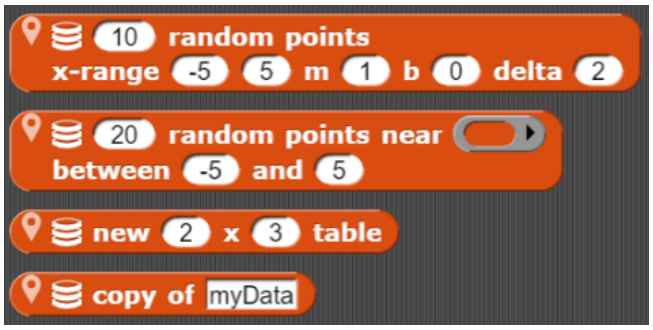
The block ***import** <data> **from** <source> **to myData***, which is often used together with ***read file with filepicker***, is used to read data. Existing data in table form can be written to CSV files with ***write** <table> **to CSV file** <filename>* for further processing. If they are available as a character string, they can be saved as a text file with ***write string** <string> **to file** <filename>*.


import costume-(RGB)-data from current-costume to myData
read file with filepicker
write myData to CSV file
write string to file

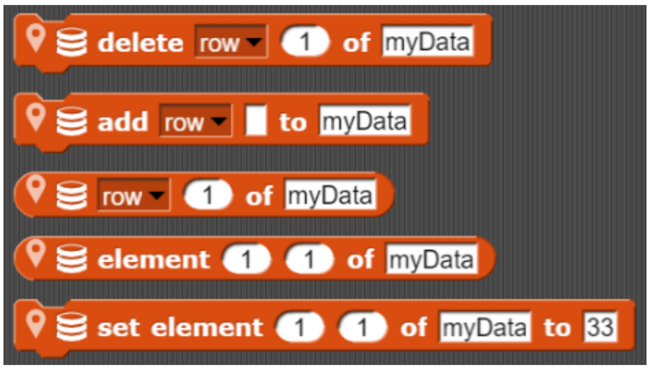
In the *Operators* palette we find three more blocks: ***random*** provides random numbers from the usual range between 0 and 1 with full accuracy, ***regression line parameters of** <source>* calculates the parameters of a regression line through the given data set. The predicate ***is** <source> **a** <type>* tests the input for whether it is a table, a vector or a matrix, the latter being allowed to contain only numerical values. The block is mainly used to intercept errors.



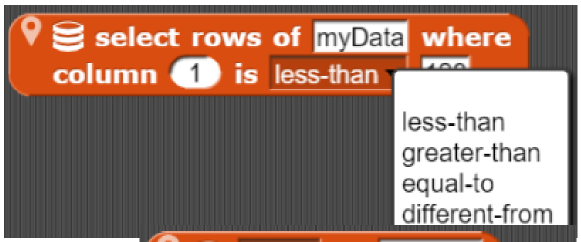
The main functions of the *DataSprite* can be found in the *Variables* palette. There are two blocks for generating test data: *<n> **random points with ranges** <xmin><xmax> **and** <ymin><ymax>* generates random points from the given range that spread around a straight line, *<n> **random points near** <term> **between** <xmin> **and** <xmax>* corresponding to points distributed around the given function graph. The block ***new** <n> **X** <m> **table*** creates a new empty table of the specified size and ***copy of** <source>* returns a copy of the passed data. This is sometimes necessary to prevent the operations on the data from altering the original data itself.



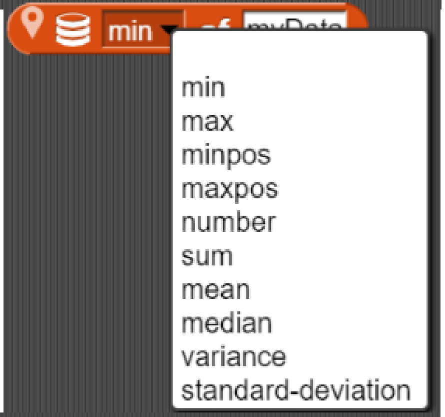
The next group of blocks refers to the elements of the specified table: ***delete <row/column>*** *<n>* ***of*** *<table>* deletes a column or row of the specified table, ***add <row/column>*** *<data>* ***to*** *<table>* appends the passed data as a new row or column to the table. With ***<row/column>*** *<n>* ***of*** *<source>* you can copy individual rows or columns of a table. The last two blocks allow access to individual table elements.



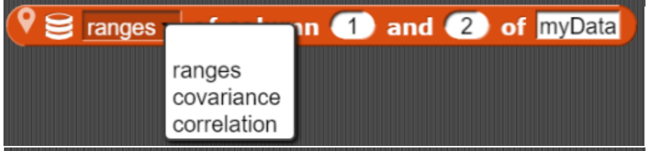
select rows of *<table>* ***where column*** *<n>* ***is*** *<predicate>* *<value>* allows to select table rows with certain properties.

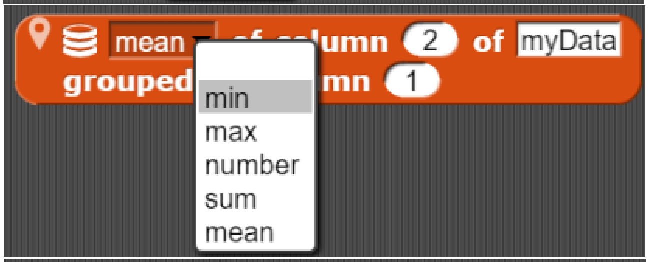


The block *<property>* ***of*** *<data>* determines the minimum or maximum of a vector and their positions, the number of elements or their sum, the mean value or median as well as the variance or standard deviation.

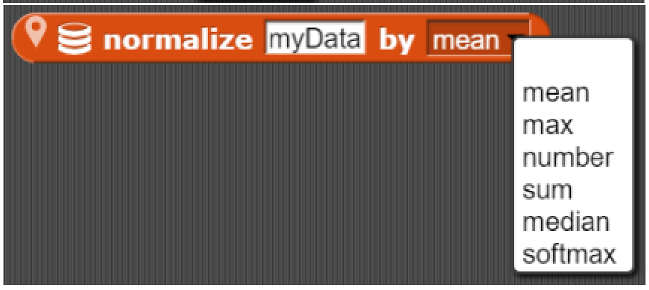


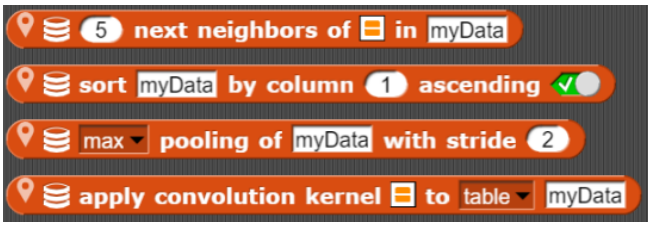
Relationships between two columns of a table can be determined with ***<property> of column*** *<n>* ***and*** *<m>* ***of*** *<table>*. ***ranges*** determines the value ranges that are needed for graphical representations, for example, the ***covariance*** and the ***correlation coefficient*** are used for statistical investigations. The next block groups the data of a column and calculates the specified values of the respective groups.





The block ***normalize*** *<data>* ***by*** *<option>* is intended for graphical representations and the comparison of data. Also, for neural networks it is sometimes needed with the option ***softmax***.

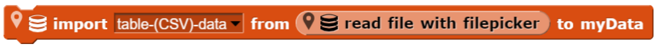




```
5 next neighbors of ☰ in myData
sort myData by column 1 ascending ✓
max pooling of myData with stride 2
apply convolution kernel ☰ to table myData
```

The last four blocks are mainly needed for examples from the field of machine learning. The reporter block ***<k> next neighbors of** <point>* ***in** <data>* determines the ***k*** nearest neighbors of a point in a point set. It can be used, for example, for clustering problems. ***sort <data> by column <n> <ascending/descending>*** allows you to sort a table by a column to be specified. The ***pooling*** block reduces the amount of data with a step size to be specified, e.g. for convolutional neural networks, and the actual convolution can be performed for images or neural networks with ***apply convolution kernel** <kernel>* ***to <table/image>***. Examples of this are given in the next section.

Example: Income data from the US Census income dataset (Source: [Census])

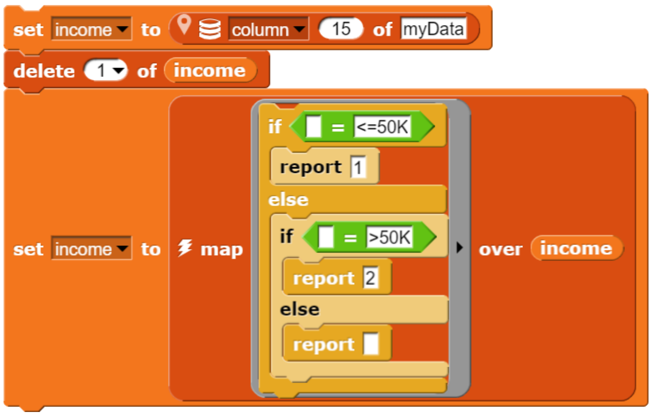


```
import table-(CSV)-data from read file with filepicker to myData
```

We want to dig a little bit in data and therefore download the *Census Income Dataset*[3] from the net. The corresponding CSV file can be loaded from the storage directory into ***myData*** and displayed immediately. It contains 32562 datasets. A right click on it and selecting "*open in dialog...*" shows all columns.

What connections could now become apparent in?

Our *DataSprite* blocks do not help much at first, because they mostly process numerical data. If we want to use them, we have to scale the columns so that numerical contents result. In the simplest case, we just replace texts with numerical values - and we should think carefully about the consequences this might have in terms of interpretation.



```
set income to column 15 of myData
delete 1 of income
set income to map (
    if ( [] = <=50K )
        report 1
    else
        if ( [] = >50K )
            report 2
        else
            report []
) over income
```

Let's start with the last column: The income values are given for only two ranges: less than or greater than $50000. We assign the values *1* and *2* to these ranges. (Or 0 and 1, or -1 and +1, or 0 and 100, or ... Would these changes have consequences)? In order not to change the original values, we create a variable ***income*** and store the changed values there by copying column 15 into this variable, deleting the first value (the heading) and then using the ***map...over...*** block to change the contents. At least that's what we're trying to do. Since there are quite a lot of values, we right-click on the map block and choose *just-in-time compilation* with "*compile...*". A small lightning symbol appears in front of map. Unfortunately we only get the unchanged column 13 when we look at the result as a table again.

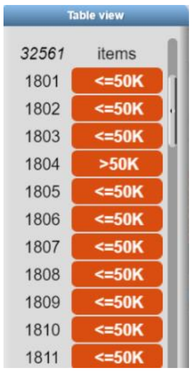
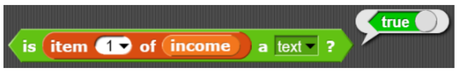


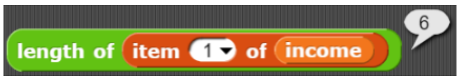
Table view

32561	items
1801	<=50K
1802	<=50K
1803	<=50K
1804	>50K
1805	<=50K
1806	<=50K
1807	<=50K
1808	<=50K
1809	<=50K
1810	<=50K
1811	<=50K

What's up? We look at the first element of ***income*** and check if it is a string. That is the case, but it is longer than we thought:



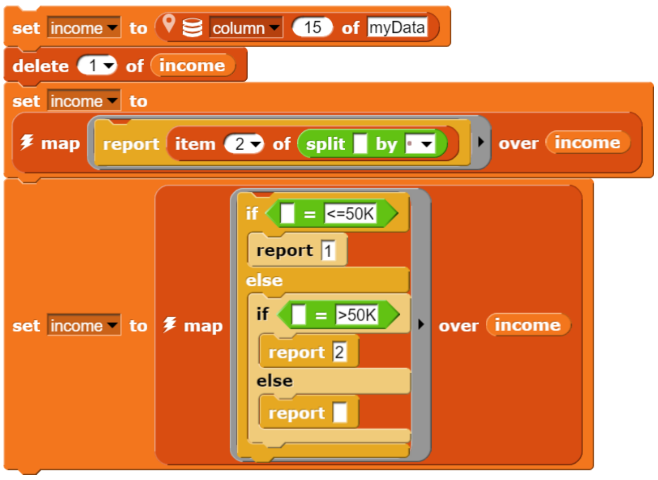
```
is item 1 of income a text ?
true
```



```
length of item 1 of income
6
```

[3] This is one of the training data sets for machine learning.

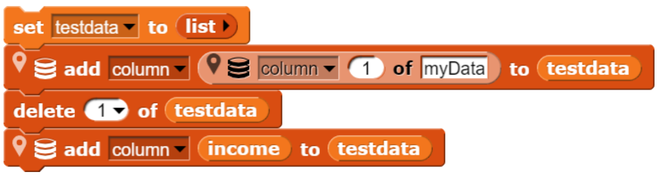
So, we have to throw out the leading spaces first. That works: our variable ***income*** only contains the values *1* and *2*, as we can quickly check by looking at it.



```
set income to column 15 of myData
delete 1 of income
set income to map (report item 2 of split [ ] by [ ]) over income
set income to map (
    if [ ] = <=50K
        report 1
    else
        if [ ] = >50K
            report 2
        else
            report [ ]
) over income
```

What does this income now depend on?

Maybe from age? We combine column 1 (age) and our modified income column into a new table called ***testdata***.



```
set testdata to list
add column (column 1 of myData) to testdata
delete 1 of testdata
add column income to testdata
```

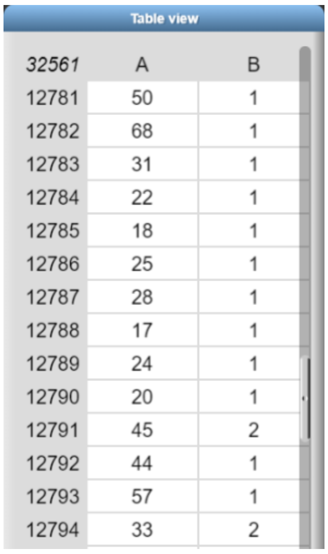
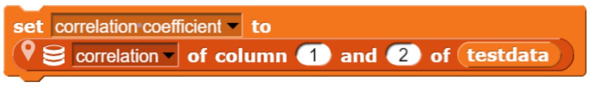


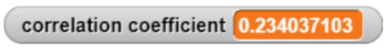
Table view

32561	A	B
12781	50	1
12782	68	1
12783	31	1
12784	22	1
12785	18	1
12786	25	1
12787	28	1
12788	17	1
12789	24	1
12790	20	1
12791	45	2
12792	44	1
12793	57	1
12794	33	2

The correlation between age and income is described by the *correlation coefficient*. The calculation is simple:



```
set correlation coefficient to correlation of column 1 and 2 of testdata
```



```
correlation coefficient 0.234037103
```

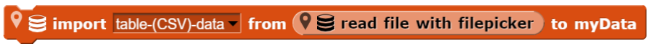
And what does that mean?

<u>**Tasks**</u>:

1. Find out about the meaning of the correlation coefficient and the interpretation of the value obtained. What does the value "0.2340..." mean?
2. Does the correlation coefficient in this case depend on the type of numerical scaling of the data (1 and 2, -1 and 1, ...)? Check that.
3. Determine other correlation coefficients, e.g. between education and income, country of origin and income, marital status and income, country of origin and occupation, ...
4. Find out if and when scaling of non-numerical data can have an influence on the result.

Example: New York Citibike Tripdata (Source: [NYcitibike])

Let's go see who actually rides a bike in New York. To do this, we download the NYCitibike rental data of one month onto our computer, that is the already mentioned almost 600000 data records. We take a closer look at them.



```
import table-(CSV)-data from read file with filepicker to myData
```

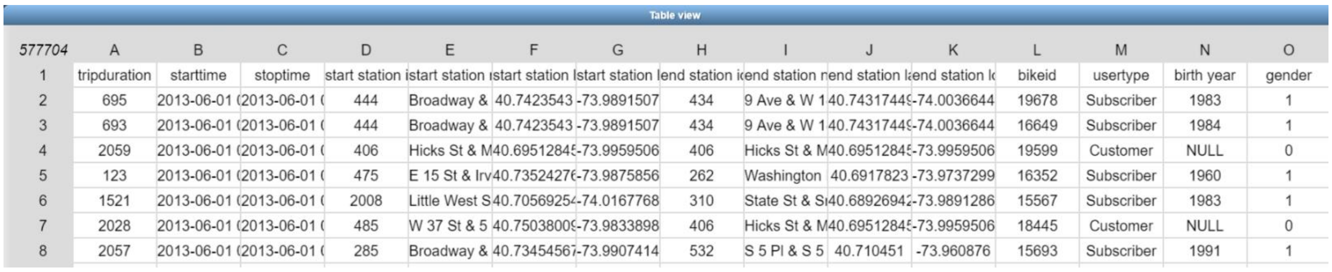
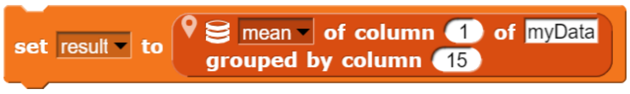


Table view

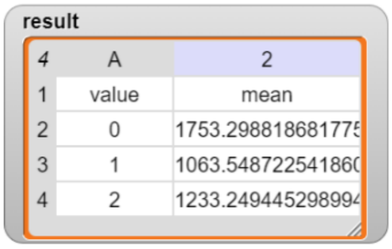
577704	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	tripduration	starttime	stoptime	start station	start station	start station	start station	end station	end station	end station	end station	bikeid	usertype	birth year	gender
2	695	2013-06-01 (	2013-06-01 (	444	Broadway &	40.7423543	-73.9891507	434	9 Ave & W 1	40.7431744	-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 & S	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

Of course, we still have to find out from the source what the data actually means - i.e. look at the metadata. For *gender* we learn that *0: unknown*, *1: male* and *2: female*. For the columns "*tripduration*" and "*gender*" we get some data:

The average loan period, based on gender:



```
set result to mean of column 1 of myData grouped by column 15
```

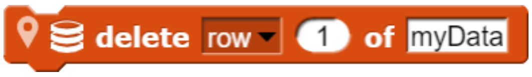


result

4	A	2
1	value	mean
2	0	1753.29881868177
3	1	1063.54872254186
4	2	1233.24944529899

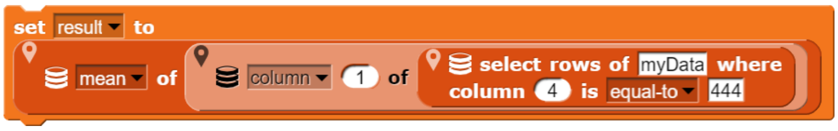
We figured as much!

For further calculations we delete the header line of the table ...

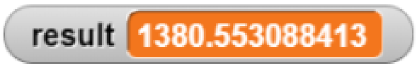


```
delete row 1 of myData
```

... and see if they're any lazier on Broadway:

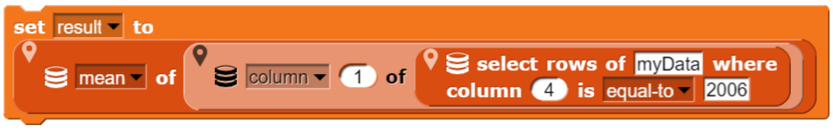


```
set result to mean of column 1 of select rows of myData where column 4 is equal-to 444
```

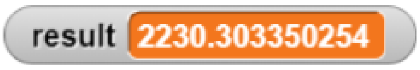


result 1380.553088413

I see. It's probably worse in Central Park!



```
set result to mean of column 1 of select rows of myData where column 4 is equal-to 2006
```



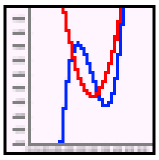
result 2230.303350254

All right. All the prejudices don't have to be true. 😉

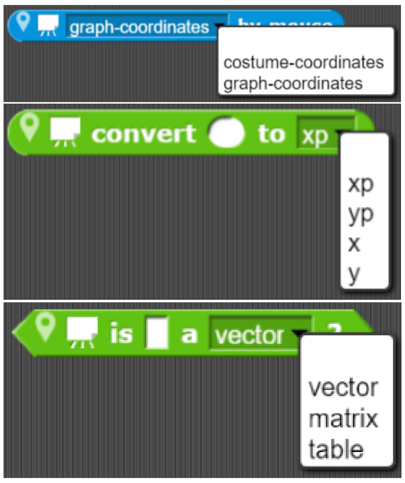
Tasks:

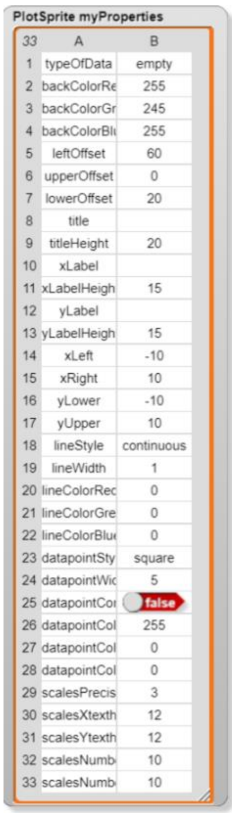
1. Maybe only women in Central Park ride bikes more. Check it out.
2. It's not like there's only one bike rental place in Central Park. Find out the appropriate mean values for the entire area.
3. Is there actually borrowing information for other parts of the city? Search and compare the results with Manhattan.
4. Determine the average borrowing times per weekday, overall and for individual stations. Are there any differences? What are the differences?
5. Above, the average borrowing duration was calculated based on gender. It could be done the other way around. Would that be complete nonsense or are there any questions where this would make sense?

4.5 The PlotSprite



The *PlotSprite* is used to create and display diagrams. Therefore, it is mostly used together with a *DataSprite*. In addition to the three blocks for managing its properties in the *Variables* palette, which primarily contain presets for generating the diagrams, there is only one new block ***<costume-/graph-coordinates> by mouse*** in the *Sensing* palette for the "*remeasurement*" of values with the mouse and two new blocks in the *Operators* palette ***convert <value> to <...>*** for the conversion of screen to graph coordinates and vice versa as well as the already known predicate block for type checking ***is <data> a <vector/matrix/table>?***.

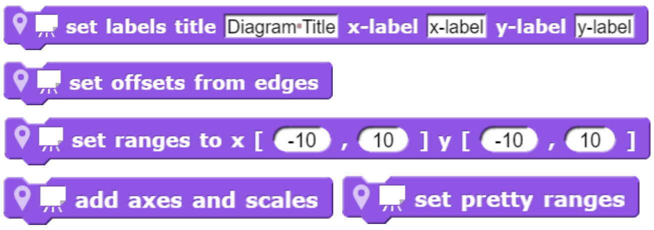



PlotSprite myProperties

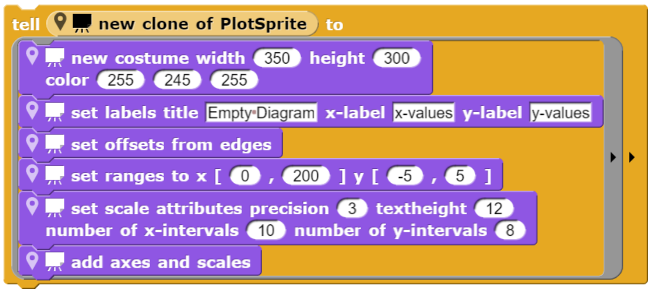
33	A	B
1	typeOfData	empty
2	backColorRe	255
3	backColorGr	245
4	backColorBlu	255
5	leftOffset	60
6	upperOffset	0
7	lowerOffset	20
8	title	
9	titleHeight	20
10	xLabel	
11	xLabelHeigh	15
12	yLabel	
13	yLabelHeigh	15
14	xLeft	-10
15	xRight	10
16	yLower	-10
17	yUpper	10
18	lineStyle	continuous
19	lineWidth	1
20	lineColorRec	0
21	lineColorGre	0
22	lineColorBlu	0
23	datapointSty	square
24	datapointWic	5
25	datapointCor	false
26	datapointCol	255
27	datapointCol	0
28	datapointCol	0
29	scalesPrecis	3
30	scalesXtexth	12
31	scalesYtexth	12
32	scalesNumb	10
33	scalesNumb	10

The actual new tools can be found at the bottom of the *Looks* palette. On the one hand, you can use them to change the preset properties and thus adapt them to the current problem, e.g. the diagram labels or the value ranges. On the other hand, you can create different types of diagrams.

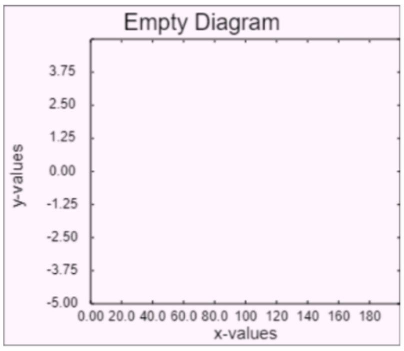
set labels ... set the values for the diagram title and the labels of the axes. Since the distances of the diagram axes from the edges depend on them, these values are also determined by ***set offsets from edges***. While we're at it, we can also reset the value ranges with ***set ranges ...*** and the display of the numbers on the axes with ***set scale attributes ...***. ***set pretty ranges*** automatically sets the number ranges with "*pretty*" limits. ***add axes and scales*** then draws the frame of the new diagram.

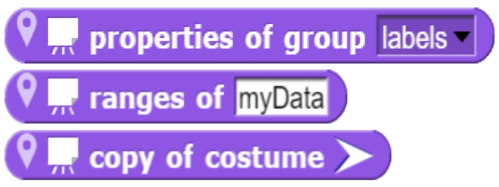

set labels title Diagram Title x-label x-label y-label y-label
set offsets from edges
set ranges to x [-10 , 10] y [-10 , 10]
add axes and scales
set pretty ranges

Since we often need several diagrams, we first create a clone of the *PlotSprite* and pass the necessary data to it. This clone will display the tastefully designed new diagram on its costume.



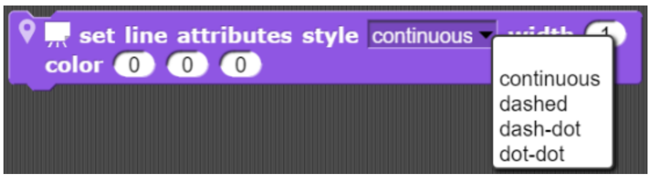
```
tell new clone of PlotSprite to
    new costume width 350 height 300
    color 255 245 255
    set labels title Empty Diagram x-label x-values y-label y-values
    set offsets from edges
    set ranges to x [ 0 , 200 ] y [ -5 , 5 ]
    set scale attributes precision 3 textheight 12
    number of x-intervals 10 number of y-intervals 8
    add axes and scales
```



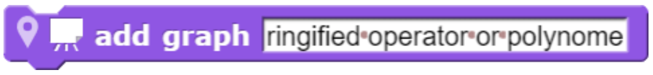


```
properties of group [labels ▾]
ranges of [myData]
copy of costume [>]
```

Three of the blocks in the *Looks* palette are for programming convenience: ***properties of group*** *<groupname>* combines several properties and thus facilitates the data transfer to *JavaScript functions*. ***ranges of*** *<data>* determines the value ranges of a two-dimensional table and ***copy of costume*** *<costume>* is needed if you want to switch quickly between two versions of a costume.



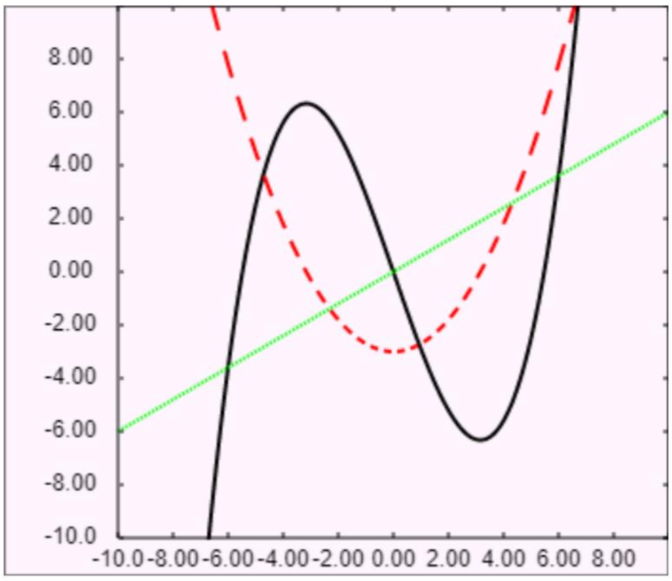
```
set line attributes style [continuous ▾] width (1)
color (0) (0) (0)
```

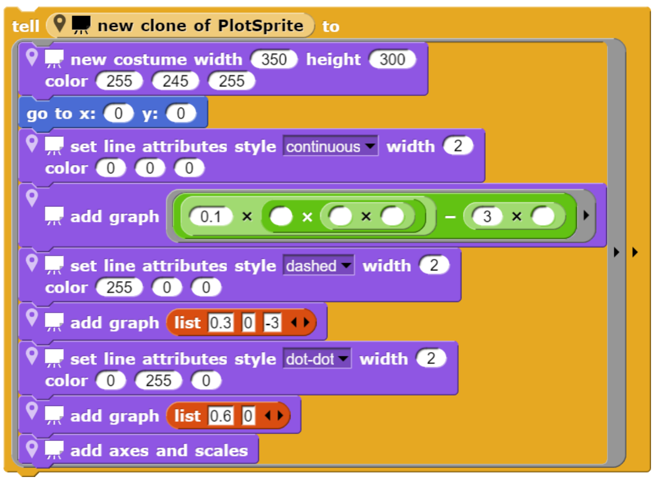


```
add graph [ringified operator or polynome]
```

The diagrams themselves are generated with the remaining new blocks. The properties of line diagrams are set with the block ***set line attributes ...*** Function graphs are drawn with these attributes using the ***add graph ...*** block, for example. The function term can be passed either as a list of the coefficients of a polynomial or as a "*ringified*" *Snap!* term.

Example: Drawing a function and its derivatives in different colors and line styles.



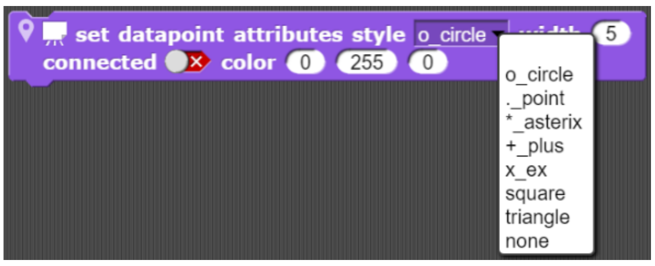


```
tell [new clone of PlotSprite] to
    new costume width (350) height (300)
    color (255) (245) (255)
    go to x: (0) y: (0)
    set line attributes style [continuous ▾] width (2)
    color (0) (0) (0)
    add graph ((0.1) × (( ) × ( ) × ( ))) − ((3) × ( ))
    set line attributes style [dashed ▾] width (2)
    color (255) (0) (0)
    add graph (list [0.3] [0] [-3])
    set line attributes style [dot-dot ▾] width (2)
    color (0) (255) (0)
    add graph (list [0.6] [0])
    add axes and scales
```

Tasks:

1. Plot different types of functions (trigonometric functions, logarithms, polynomials, ...) as a graph on a *PlotSprite.*
2. Extend the function graphs with their derivatives.
3. Choose different value ranges, accuracies of number representation, text sizes and labels for the representations.

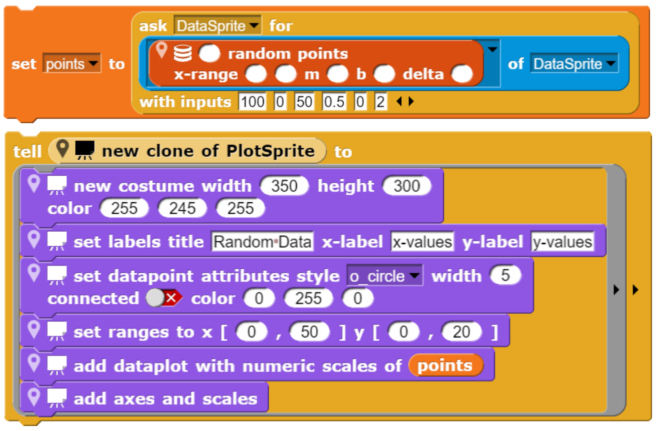
If we want to display the contents of a data table graphically, this can be done with the block ***add dataplot with numeric scales ...*** Scales and labels are again added with the block ***add axes and scales***. The way of displaying the data points can be adjusted very precisely with the block ***set datapoint attributes ...***. The lines in between keep the line attributes.



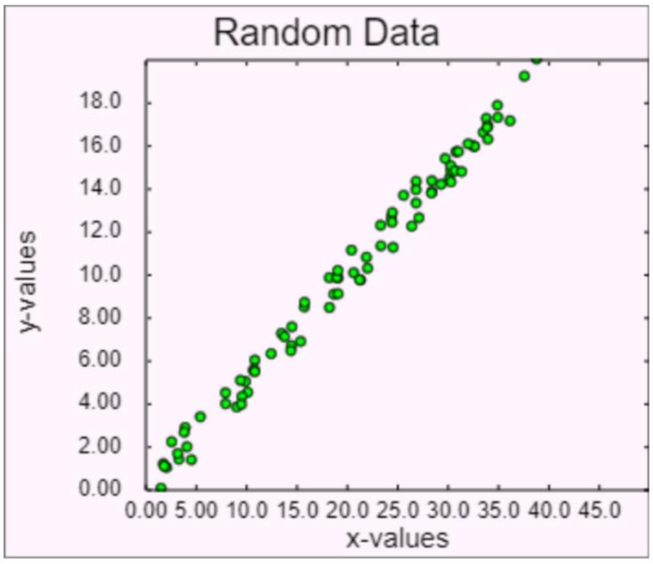
```
set datapoint attributes style o_circle width 5
connected color 0 255 0
```

Example: Representation of a point set

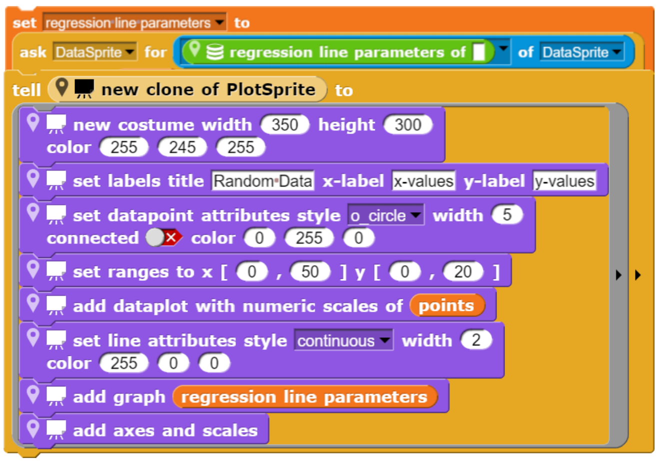
We ask a *DataSprite* to provide 100 random points scattered around a straight line with the slope m=0.5 and the intercept b=0. The obtained points are displayed in a diagram.



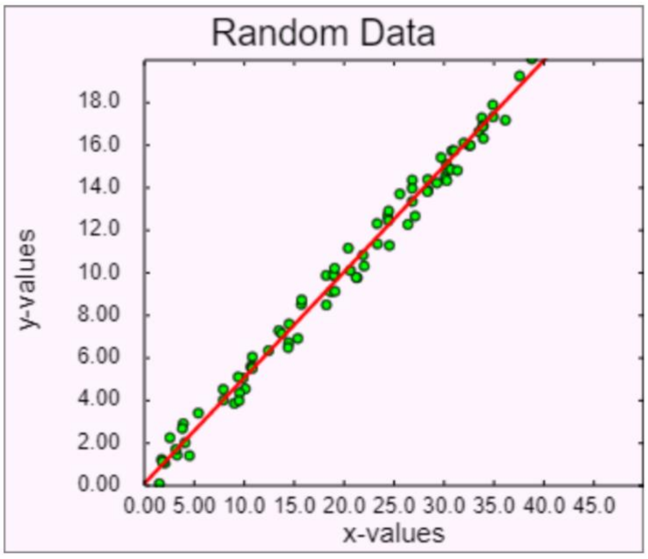
```
set points to ask DataSprite for (random points x-range ( ) ( ) m ( ) b ( ) delta ( ) of DataSprite) with inputs 100 0 50 0.5 0 2
tell (new clone of PlotSprite) to
    new costume width 350 height 300 color 255 245 255
    set labels title Random Data x-label x-values y-label y-values
    set datapoint attributes style o_circle width 5 connected color 0 255 0
    set ranges to x [ 0 , 50 ] y [ 0 , 20 ]
    add dataplot with numeric scales of points
    add axes and scales
```



Example: Additionally the regression line is now drawn.

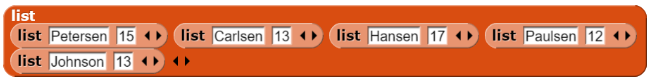


```
set regression line parameters to
ask DataSprite for (regression line parameters of ( ) of DataSprite)
tell (new clone of PlotSprite) to
    new costume width 350 height 300 color 255 245 255
    set labels title Random Data x-label x-values y-label y-values
    set datapoint attributes style o_circle width 5 connected color 0 255 0
    set ranges to x [ 0 , 50 ] y [ 0 , 20 ]
    add dataplot with numeric scales of points
    set line attributes style continuous width 2 color 255 0 0
    add graph regression line parameters
    add axes and scales
```

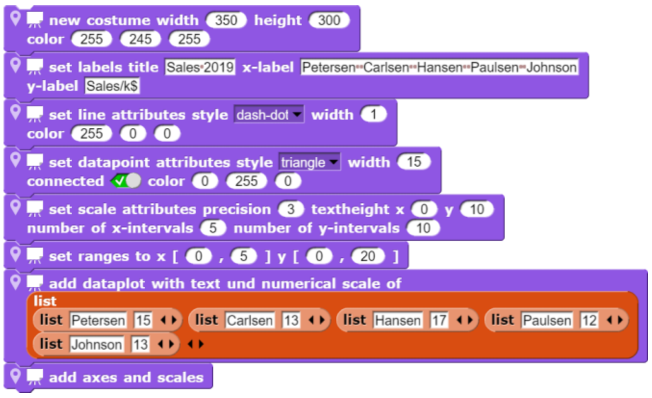


Example: Display of mixed data

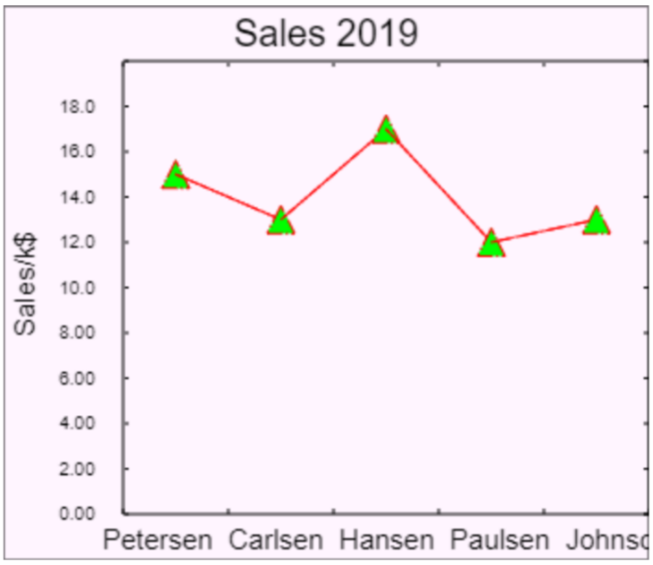
Text data is often combined with numerical data. An example would be the turnover data of different representatives in one year in one area. If we want to display them graphically, the x-axis, for example, must be labelled with text data, while the y-axis is treated as before. To create the diagram we use the block ***add dataplot with text and numerical scale***.



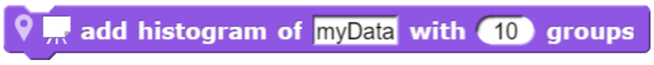
```
list
  list Petersen 15
  list Carlsen 13
  list Hansen 17
  list Paulsen 12
  list Johnson 13
```



```
new costume width 350 height 300 color 255 245 255
set labels title Sales 2019 x-label Petersen Carlsen Hansen Paulsen Johnson y-label Sales/k$
set line attributes style dash-dot width 1 color 255 0 0
set datapoint attributes style triangle width 15 connected ✓ color 0 255 0
set scale attributes precision 3 textheight x 0 y 10 number of x-intervals 5 number of y-intervals 10
set ranges to x [ 0 , 5 ] y [ 0 , 20 ]
add dataplot with text und numerical scale of
  list
    list Petersen 15
    list Carlsen 13
    list Hansen 17
    list Paulsen 12
    list Johnson 13
add axes and scales
```



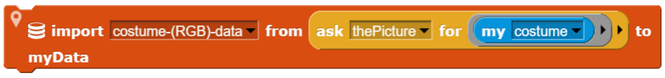
As the last of the new blocks of *PlotSprite* we consider ***add histogram of <data> with <n> groups***. Histograms can be generated and displayed directly from data sources.



```
add histogram of myData with 10 groups
```

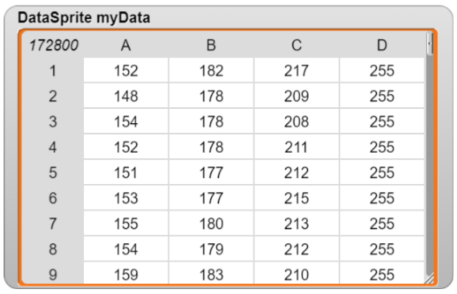
Example: An RGB image is loaded, decomposed into grayscale, and the normalized distribution of image values is displayed as a histogram on a new *PlotSprite*. We find the actual image as a costume of an additional sprite called "*thePicture*".

First of all we load the image into the data area of a *DataSprite*:



```
import costume-(RGB)-data from ask thePicture for my costume to myData
```

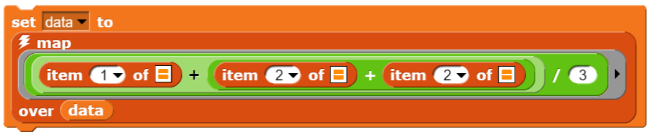
We get 172800 RBG values.



DataSprite myData

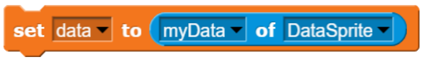
172800	A	B	C	D
1	152	182	217	255
2	148	178	209	255
3	154	178	208	255
4	152	178	211	255
5	151	177	212	255
6	153	177	215	255
7	155	180	213	255
8	154	179	212	255
9	159	183	210	255

We convert these into grey values. We use the compiled version of the ***map*** block.



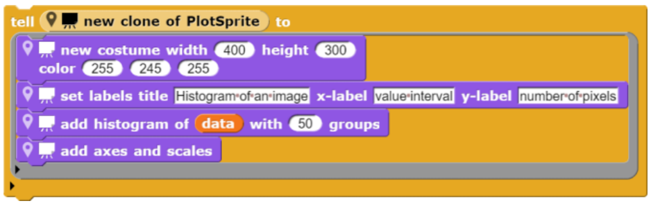
```
set data to
  map
    (item 1 of ▭ + item 2 of ▭ + item 2 of ▭) / 3
  over data
```

Then we switch to the *PlotSprite* and copy the loaded data of the *DataSprite*.

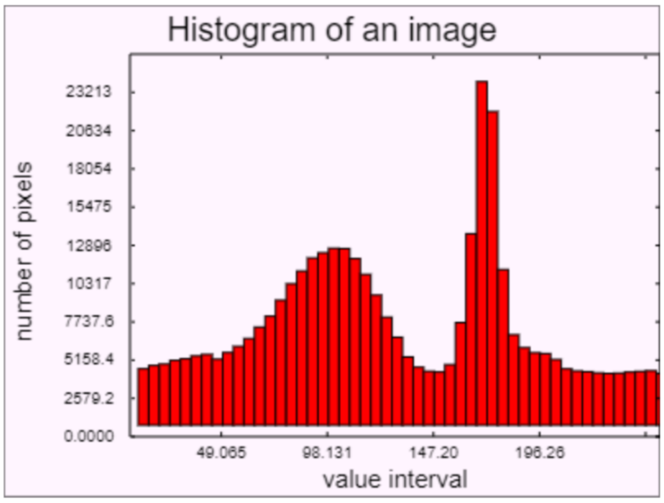


```
set data to myData of DataSprite
```

Now we can display the data as histogram e.g. on a new *PlotSprite*.



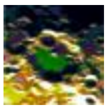
```
tell new clone of PlotSprite to
  new costume width 400 height 300 color 255 245 255
  set labels title Histogram of an image x-label value interval y-label number of pixels
  add histogram of data with 50 groups
  add axes and scales
```

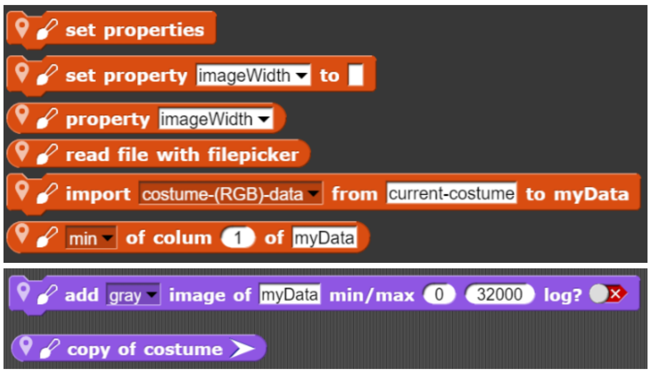


Tasks:

1. Search for different amounts of data in the network. Display these or parts of them graphically.
2. Automate the creation of histograms with a new block ***histogram of <costume>***. Compare the histograms of typical image types. To what extent is it possible to compare images in this way, or where could difficulties arise?
3. Represent the three colors of an RGB image in the same diagram by graphs and/or histograms.

4.6 The ImageSprite





set properties

set property imageWidth to

property imageWidth

read file with filepicker

import costume-(RGB)-data from current-costume to myData

min of colum 1 of myData

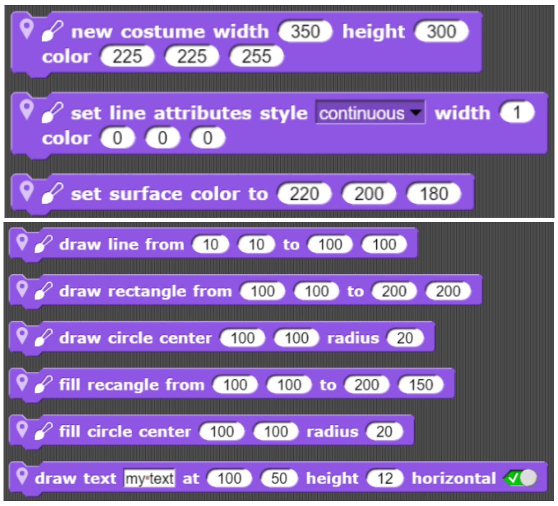
add gray image of myData min/max 0 32000 log?

copy of costume

The *ImageSprite* is used to display and manipulate images. For this purpose, it contains in the *Variables* palette, in addition to the obligatory blocks for managing the properties, the possibility to load images and determine the minimum and maximum values of the image data.

In the *Looks* palette we find the already known block ***add image ...***, which allows to display data on the costume as gray or false color image. With the help of the block ***copy of costume*** you can duplicate costumes if necessary.

In addition to the usual block for creating a new costume, available for all *ML.Sprites*, there are two blocks for setting line properties and fill color. With these and six blocks for drawing figure outlines and filled figures on the costume, you can easily create random graphics, for example.



new costume width 350 height 300 color 225 225 255

set line attributes style continuous width 1 color 0 0 0

set surface color to 220 200 180

draw line from 10 10 to 100 100

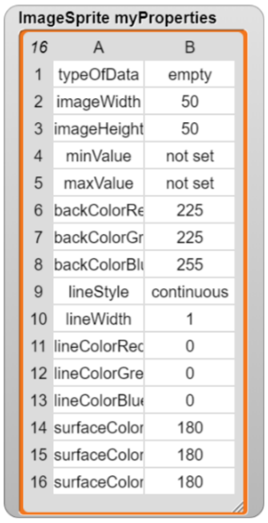
draw rectangle from 100 100 to 200 200

draw circle center 100 100 radius 20

fill recangle from 100 100 to 200 150

fill circle center 100 100 radius 20

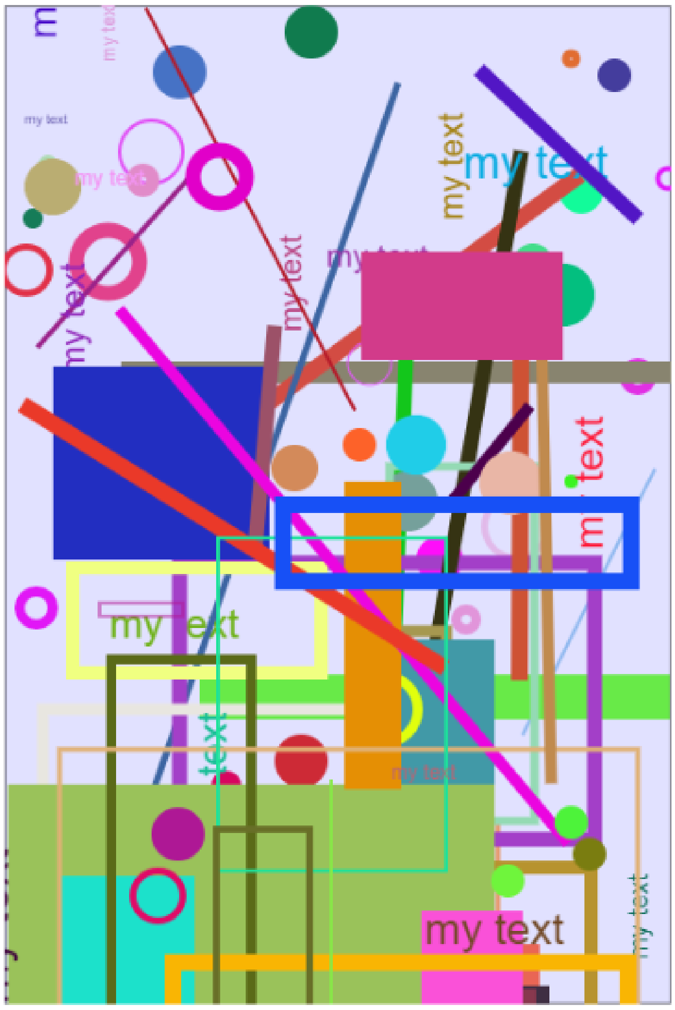
draw text my text at 100 50 height 12 horizontal

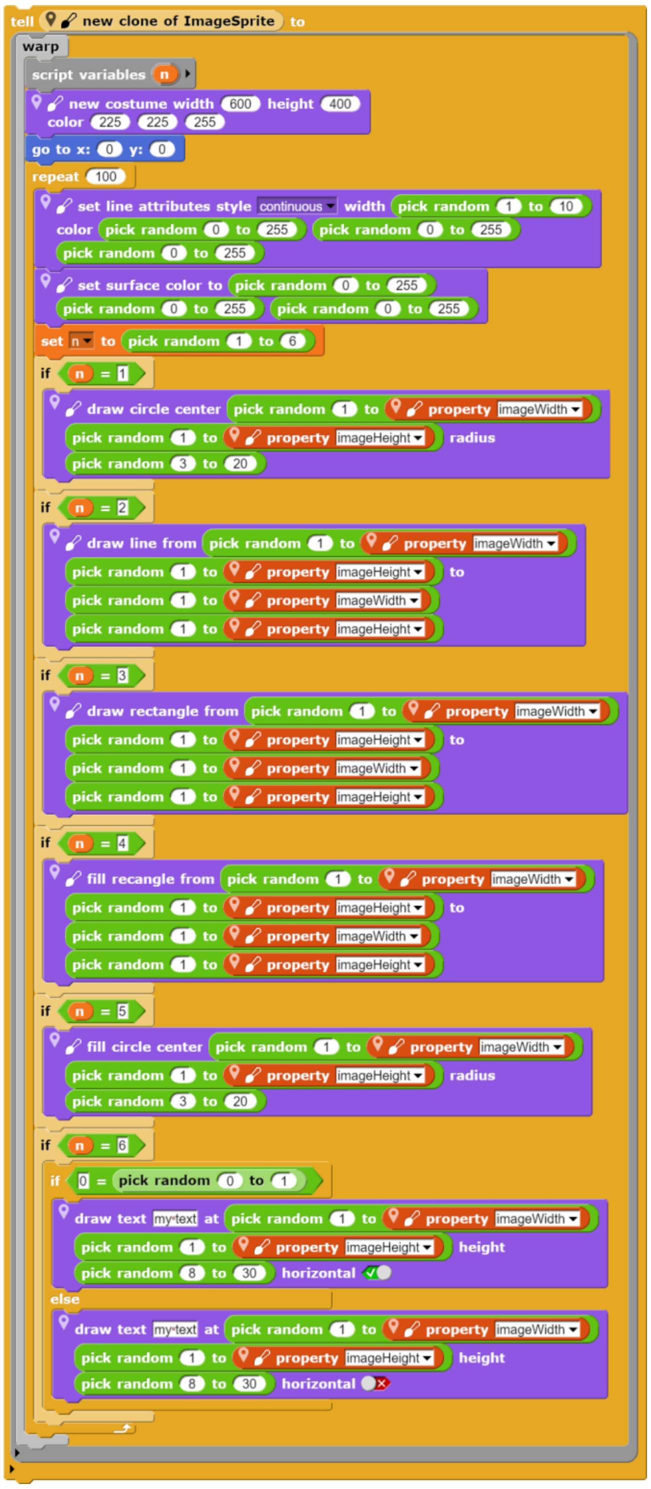


ImageSprite myProperties

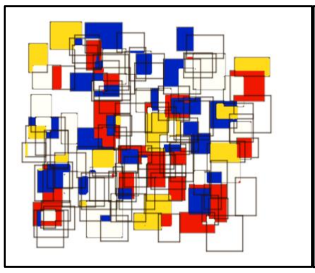
16	A	B
1	typeOfData	empty
2	imageWidth	50
3	imageHeight	50
4	minValue	not set
5	maxValue	not set
6	backColorRe	225
7	backColorGr	225
8	backColorBlu	255
9	lineStyle	continuous
10	lineWidth	1
11	lineColorRec	0
12	lineColorGre	0
13	lineColorBlue	0
14	surfaceColor	180
15	surfaceColor	180
16	surfaceColor	180

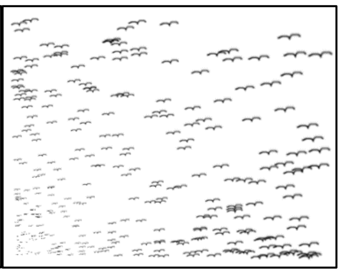
Example: Generation of random graphics





```
tell [new clone of ImageSprite] to
  warp
    script variables (n)
    new costume width (600) height (400)
      color (225) (225) (255)
    go to x: (0) y: (0)
    repeat (100)
      set line attributes style [continuous] width (pick random (1) to (10))
        color (pick random (0) to (255)) (pick random (0) to (255))
        (pick random (0) to (255))
      set surface color to (pick random (0) to (255))
        (pick random (0) to (255)) (pick random (0) to (255))
      set [n] to (pick random (1) to (6))
      if <(n) = [1]>
        draw circle center (pick random (1) to (property [imageWidth]))
          (pick random (1) to (property [imageHeight])) radius
          (pick random (3) to (20))
      if <(n) = [2]>
        draw line from (pick random (1) to (property [imageWidth]))
          (pick random (1) to (property [imageHeight])) to
          (pick random (1) to (property [imageWidth]))
          (pick random (1) to (property [imageHeight]))
      if <(n) = [3]>
        draw rectangle from (pick random (1) to (property [imageWidth]))
          (pick random (1) to (property [imageHeight])) to
          (pick random (1) to (property [imageWidth]))
          (pick random (1) to (property [imageHeight]))
      if <(n) = [4]>
        fill recangle from (pick random (1) to (property [imageWidth]))
          (pick random (1) to (property [imageHeight])) to
          (pick random (1) to (property [imageWidth]))
          (pick random (1) to (property [imageHeight]))
      if <(n) = [5]>
        fill circle center (pick random (1) to (property [imageWidth]))
          (pick random (1) to (property [imageHeight])) radius
          (pick random (3) to (20))
      if <(n) = [6]>
        if <[0] = (pick random (0) to (1))>
          draw text [my text] at (pick random (1) to (property [imageWidth]))
            (pick random (1) to (property [imageHeight])) height
            (pick random (8) to (30)) horizontal <✓>
        else
          draw text [my text] at (pick random (1) to (property [imageWidth]))
            (pick random (1) to (property [imageHeight])) height
            (pick random (8) to (30)) horizontal <✗>
```

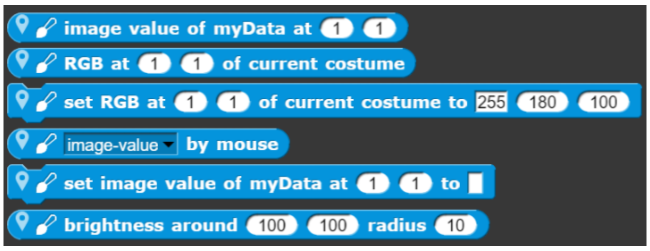




Tasks:

1. Search the web for pictures of Piet Mondrian. Try to create similar random images on the *ImageSprite*.
2. Use a "*vanishing point*" to create images in which objects appear to move "*from back to front*". Try it.

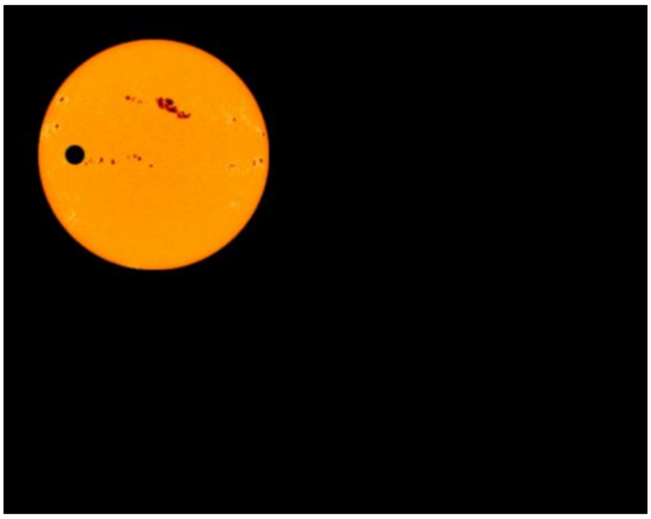
In the Sensing palette of a *ImageSprite* there are some new blocks for accessing pixel values. The block ***... by mouse*** has already been described above for data import. With ***image value ... at ...*** and ***set image value of ...*** individual pixels can be read or changed if they are located in the data area ***myData***. ***RGB at ...*** and ***set RGB at ...*** allow the same directly on the current costume. The block ***brightness around ...*** determines the total brightness around the specified point.



```
image value of myData at (1) (1)
RGB at (1) (1) of current costume
set RGB at (1) (1) of current costume to [255] (180) (100)
[image-value] by mouse
set image value of myData at (1) (1) to [ ]
brightness around (100) (100) radius (10)
```

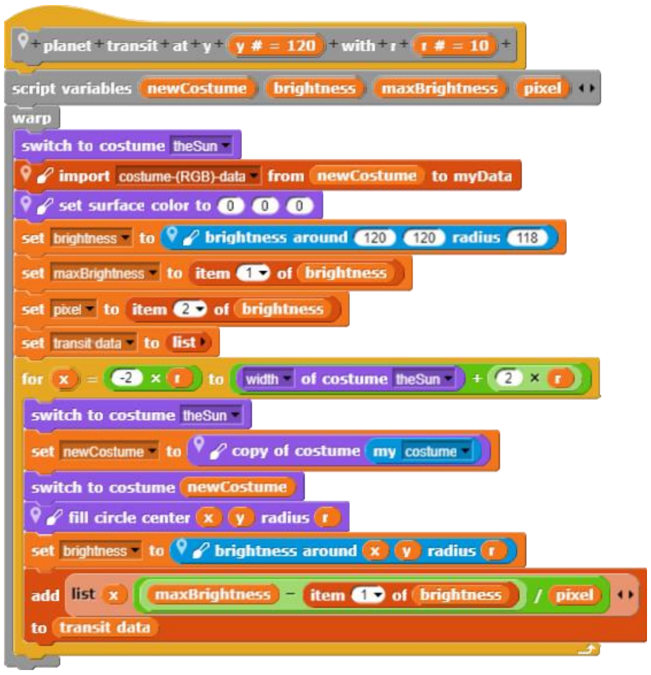
Example: Simulation of a planetary transit in front of the sun

We look for a nice image of the sun (source here: [Schul-Astro]) and load it as a costume of an *ImageSprite*. To make it look more like space, we enlarge the stage and color it black. If we still draw the planet, we get the image on the right.



The planet should pass in front of the sun as a black circle. If we paint such a circle, we change the actual image of the sun. From this we make a copy ***newCostume*** to draw on it. Our planet should move from the far left a little outside the image (***x***=-2r) to the far right (***x***=image width+2r) on the height ***y***. We can also specify the radius ***r*** of the planet.

We can determine the current brightness of this arrangement without too many copying processes by subtracting the brightness of the pixels covered by the planet from the total brightness determined at the beginning. To do this, we initially import the image of the sun into the data area ***myData*** and determine the brightness around the image center in the radius "*half image width*" as well as the number of pixels involved. ***brightness around*** returns the summed gray values as well as its number. From these values we calculate the average brightness of the "*slightly darkened*" sun and store it together with the current position in the variable ***transit data***. We package these operations in the new block ***planet transit at <y> with r <r>***.

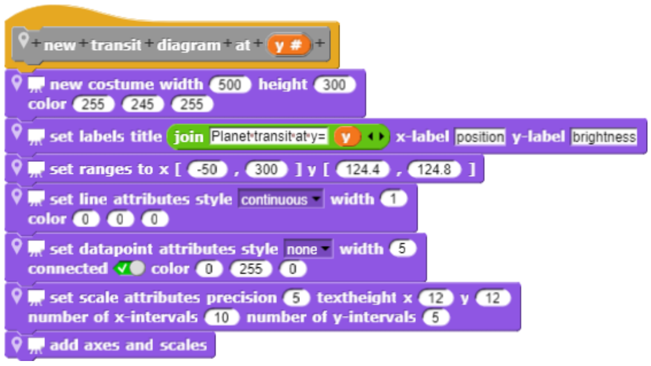


```
+planet+transit+at+y+(y # = 120)+with+r+(r # = 10)+
script variables (newCostume) (brightness) (maxBrightness) (pixel)
warp
  switch to costume [theSun]
  import [costume-(RGB)-data] from (newCostume) to myData
  set surface color to (0) (0) (0)
  set [brightness] to (brightness around (120) (120) radius (118))
  set [maxBrightness] to (item (1) of (brightness))
  set [pixel] to (item (2) of (brightness))
  set [transit data] to (list)
  for (x) = ((-2) × (r)) to (([width] of costume [theSun]) + ((2) × (r)))
    switch to costume [theSun]
    set [newCostume] to (copy of costume (my [costume]))
    switch to costume (newCostume)
    fill circle center (x) (y) radius (r)
    set [brightness] to (brightness around (x) (y) radius (r))
    add (list (x) (((maxBrightness) - (item (1) of (brightness))) / (pixel)))
    to (transit data)
```

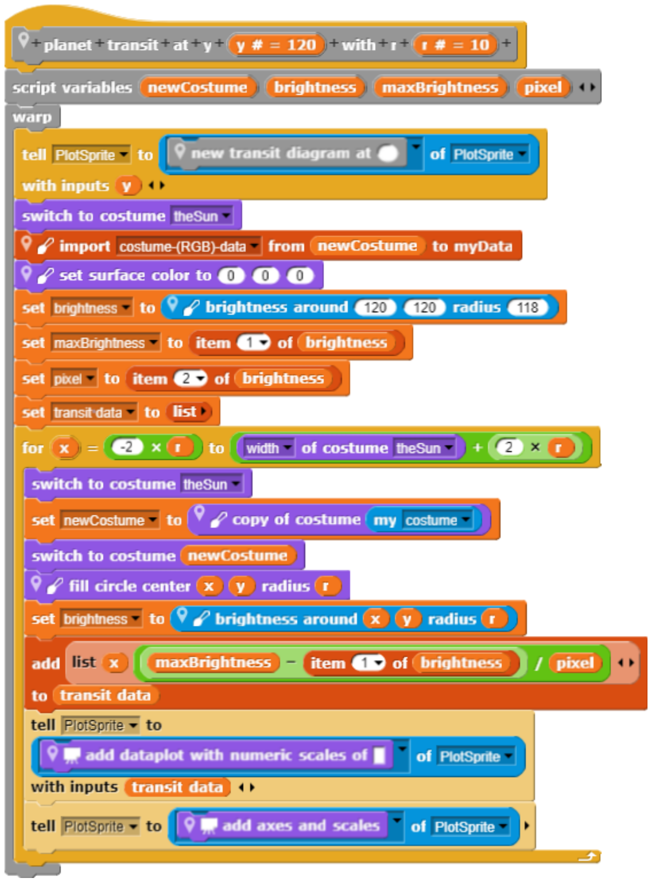
We now want to follow the planet transit "*live*" in a diagram. For this purpose, we load the *PlotSprite* and write a block for this plot to prepare the diagram parameters. This block can be called by the *ImageSprite*.

We insert the corresponding block at the beginning of our transit script and supplement it with two calls at the end of each loop to redraw the *PlotSprite* diagram with the new data.

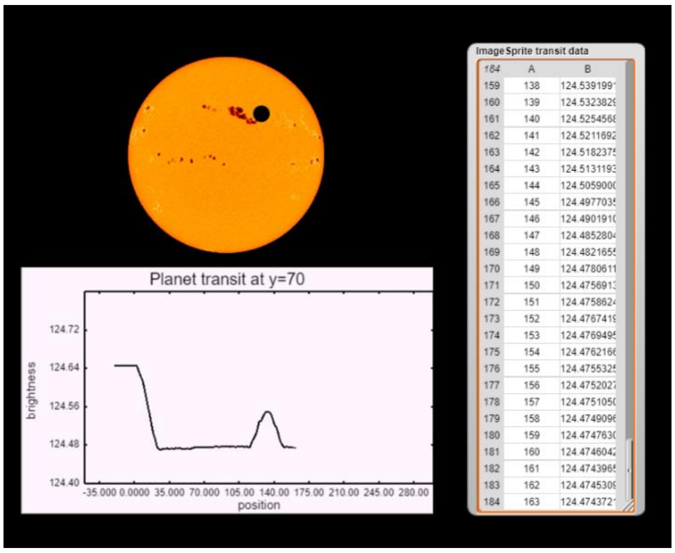
The result corresponds roughly to one of the methods used to find exo planets.



```
new transit diagram at (y #)
  new costume width (500) height (300) color (255) (245) (255)
  set labels title (join [Planet transit at y=] (y)) x-label [position] y-label [brightness]
  set ranges to x [ (-50) , (300) ] y [ (124.4) , (124.8) ]
  set line attributes style [continuous] width (1) color (0) (0) (0)
  set datapoint attributes style [none] width (5) connected ✓ color (0) (255) (0)
  set scale attributes precision (5) textheight x (12) y (12) number of x-intervals (10) number of y-intervals (5)
  add axes and scales
```



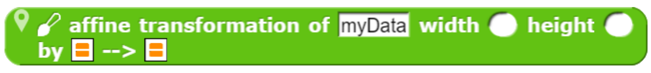
```
planet transit at y (y # = 120) with r (r # = 10)
script variables (newCostume) (brightness) (maxBrightness) (pixel)
warp
  tell [PlotSprite] to (new transit diagram at ( ) of [PlotSprite]) with inputs (y)
  switch to costume [theSun]
  import [costume-(RGB)-data] from (newCostume) to myData
  set surface color to (0) (0) (0)
  set [brightness] to (brightness around (120) (120) radius (118))
  set [maxBrightness] to (item (1) of (brightness))
  set [pixel] to (item (2) of (brightness))
  set [transit data] to (list)
  for (x) = ((-2) × (r)) to (([width] of costume [theSun]) + ((2) × (r)))
    switch to costume [theSun]
    set [newCostume] to (copy of costume (my [costume]))
    switch to costume (newCostume)
    fill circle center (x) (y) radius (r)
    set [brightness] to (brightness around (x) (y) radius (r))
    add (list (x) (((maxBrightness) - (item (1) of (brightness))) / (pixel))) to (transit data)
    tell [PlotSprite] to (add dataplot with numeric scales of [ ] of [PlotSprite]) with inputs (transit data)
    tell [PlotSprite] to (add axes and scales of [PlotSprite])
```



ImageSprite transit data

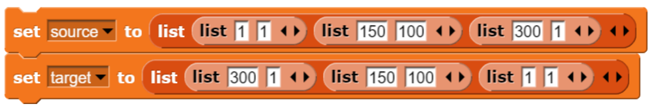
184	A	B
159	138	124.5391991
160	139	124.5323829
161	140	124.5254568
162	141	124.5211692
163	142	124.5182375
164	143	124.5131193
165	144	124.5059000
166	145	124.4977035
167	146	124.4901910
168	147	124.4852804
169	148	124.4821655
170	149	124.4780611
171	150	124.4756913
172	151	124.4758624
173	152	124.4767419
174	153	124.4769495
175	154	124.4762166
176	155	124.4755325
177	156	124.4752027
178	157	124.4751050
179	158	124.4749096
180	159	124.4747630
181	160	124.4746042
182	161	124.4743965
183	162	124.4745309
184	163	124.4743721

In the *Operators* palette, we find the ***is ... a ...*** predicate, which is used everywhere, and two somewhat more sophisticated blocks. The first allows to perform *affine transformations* in an image by mapping three points to three others - and all other points accordingly. Additionally, the dimensions of the image must be specified.



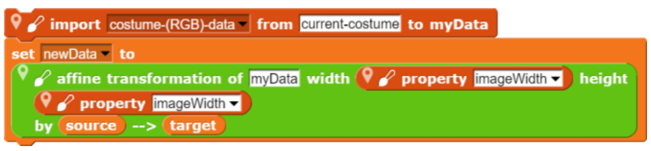
```
affine transformation of [myData] width ( ) height ( ) by [☰] --> [☰]
```

<u>Example</u>: Let's flip an image vertically along the center line. We load the image - here: of a church - and select corresponding points on the edges. These points are combined to the two lists ***source*** and ***target***.



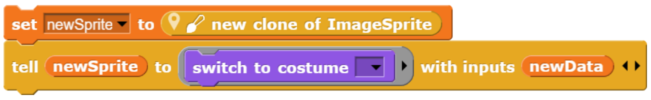
```
set [source] to (list (list [1] [1]) (list [150] [100]) (list [300] [1]))
set [target] to (list (list [300] [1]) (list [150] [100]) (list [1] [1]))
```

After that we import the image into the data area ***myData*** and execute the affine transformation with the two pointlists. We save the result in the variable ***newData***.

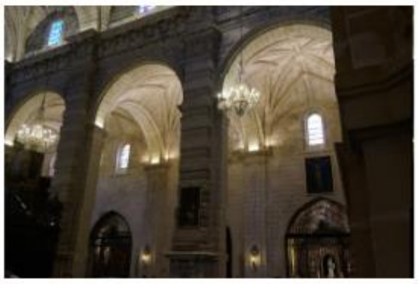


```
import [costume-(RGB)-data] from [current-costume] to myData
set [newData] to (affine transformation of [myData] width (property [imageWidth]) height (property [imageWidth]) by (source) --> (target))
```

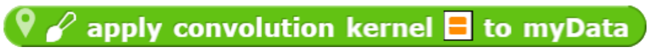
Finally, we create a clone of the *ImageSprite* and ask it to display the transformed image as a costume.



```
set newSprite to new clone of ImageSprite
tell newSprite to switch to costume with inputs newData
```



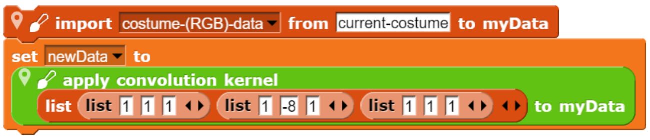
As last new block we find the ***apply convolution kernel … to myData*** – block to perform convolutions on images.



```
apply convolution kernel ☰ to myData
```

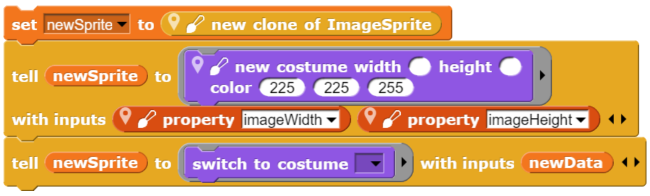
Example: Edge detection

We load a picture of an ancient temple and import its data into the data area of ***myData***. We apply the Laplace kernel $\begin{bmatrix} 1 & 1 & 1 \\ 1 & -4 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ to it and save the result in the variable ***new-Data***.

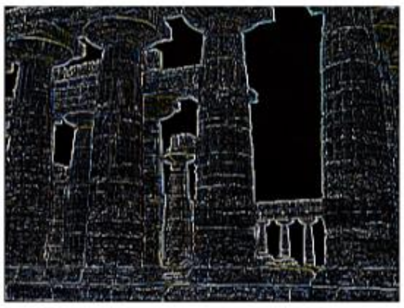


```
import costume-(RGB)-data from current-costume to myData
set newData to
    apply convolution kernel
        list (list 1 1 1) (list 1 -8 1) (list 1 1 1) to myData
```

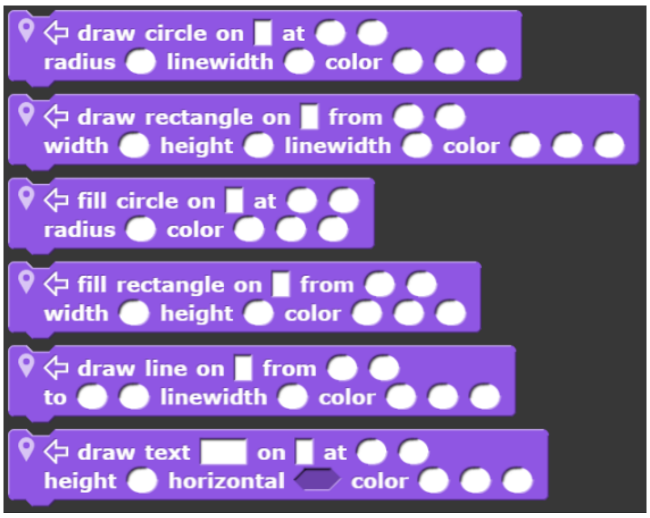
Next, we create a clone of the *ImageSprite*, resize it and let it show the changed image as its costume.



```
set newSprite to new clone of ImageSprite
tell newSprite to (new costume width ( ) height ( ) color 225 225 255)
    with inputs (property imageWidth) (property imageHeight)
tell newSprite to switch to costume with inputs newData
```



Since there is sometimes a need to execute the graphics methods in the context of another sprite (see above), these methods are available in an "*exportable*" version that does not rely on local sizes. They are labeled by a left arrow ⇦ . The usage is described in chapter 4.2.


⇦ draw circle on at radius linewidth color

⇦ draw rectangle on from width height linewidth color

⇦ fill circle on at radius color

⇦ fill rectangle on from width height color

⇦ draw line on from to linewidth color

⇦ draw text on at height horizontal color

<u>**Tasks**</u>:

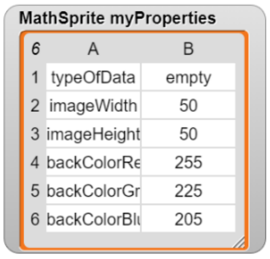
1. Pictures are sometimes a bit "*weak*". This is because they do not use the full range of values for the three color-channels from 0 to 255.
   a: Develop a method to determine and display the value ranges of an image.
   b: Develop a method to use the full range of values, i.e. to map black pixels to 0, bright pixels to 255.
   c: Summarize the method as a new block with the costume of a sprite as input and the improved costume as output.

2. a: You can try to find "*faces*" on images by highlighting contiguous areas of a color range, e.g. "*orange*", and deleting the rest of the image. Try to develop a new block for this.
   b: You can use a kernel to isolate the edges of such areas. Find out about suitable kernels on the net and test them for the purpose mentioned.
   c: Faces are often "*oval*". Try to distinguish faces from other "*orange*" objects in this way.

3. a: Really artistic photos are black and white of course. If you don't have any, you can create grayscale images from RGB images. Do that.
   b: It's even more artistic if the photos are "hard", i.e. have a very strong contrast. Experiment a little!

4.7 The MathSprite

The *MathSprite* can create clones and a new costume and manage its - few - properties, but otherwise it only has new blocks in the *Operators* palette. It is intended for services for the other sprite types. Essentially, it adds linear algebra capabilities to the operators of *Snap!*.

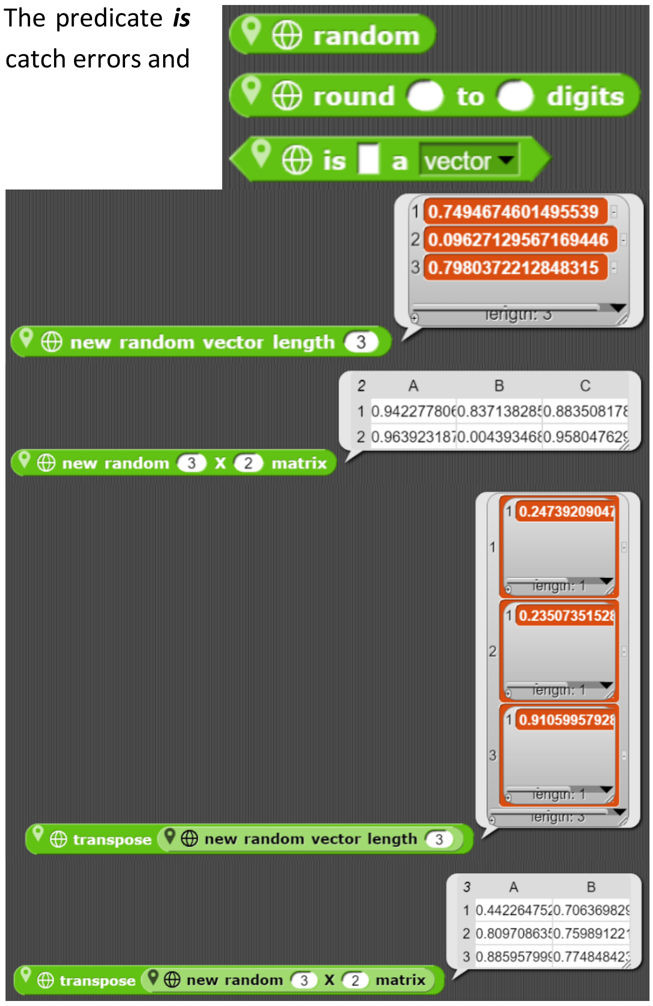

MathSprite myProperties

6	A	B
1	typeOfData	empty
2	imageWidth	50
3	imageHeight	50
4	backColorRe	255
5	backColorGr	225
6	backColorBlu	205

Three of the blocks are quickly explained: ***random*** returns a random number of the usual type, i.e. between 0 and 1 with full accuracy. ***round*** rounds numbers with a fractional part after the decimal point to the number of digits specified. The predicate ***is*** *<data>* ***a*** ***<vector/matrix/table>*** is mostly used in scripts to catch errors and has ***been*** used in this script many times.

The next two blocks should also be self-explanatory. ***new random vector*** generates a new vector from random numbers, which is sometimes needed for testing purposes. ***new random matrix*** does the same for matrices.

The *MathSprite* works with numbers, vectors and matrices. Since vectors and matrices are sometimes needed in transposed form, we also find a block ***transpose***. This block can handle both data types.


random
round to digits
is a vector
new random vector length 3
new random 3 X 2 matrix
transpose new random vector length 3
transpose new random 3 X 2 matrix

1	0.7494674601495539
2	0.09627129567169446
3	0.7980372212848315

length: 3

2	A	B	C
1	0.94227780	0.83713828	0.88350817
2	0.96392318	0.00439346	0.95804762

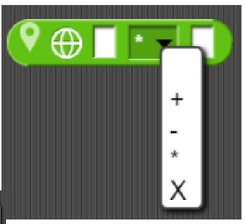
1	1 0.24739209047 length: 1
2	1 0.23507351528 length: 1
3	1 0.91059957928 length: 1

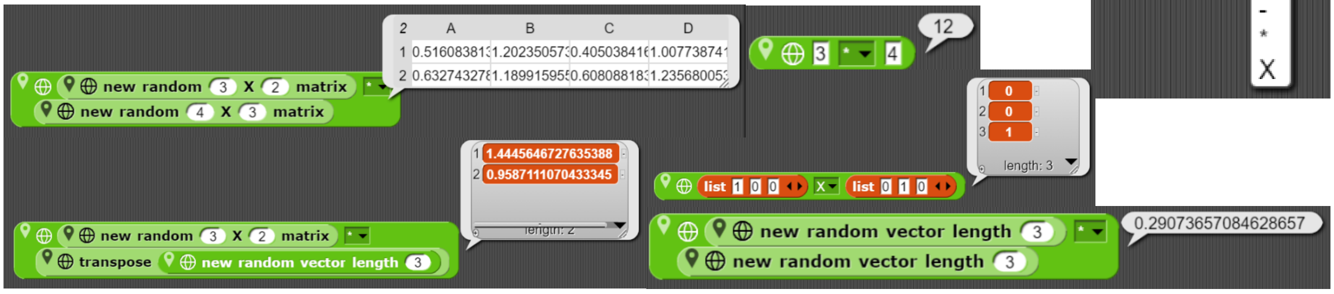
length: 3

3	A	B
1	0.44226475	0.70636982
2	0.80970863	0.75989122
3	0.88595799	0.77484842

More interesting are the next blocks.

The somewhat inconspicuous reporter block *<a>* ***operator*** *<b>* can perform the specified operation with numbers, vectors and matrices - if they are allowed. So it works between numbers, numbers and vectors, vectors and matrices, matrices and matrices, ...




new random 3 X 2 matrix *
new random 4 X 3 matrix
3 * 4
12
new random 3 X 2 matrix *
transpose new random vector length 3
list 1 0 0 X list 0 1 0
new random vector length 3 *
new random vector length 3
0.29073657084628657

2	A	B	C	D
1	0.51608381	1.20235057	0.40503841	1.00773874
2	0.63274327	1.18991595	0.60808818	1.23568005

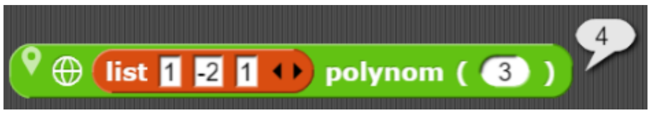
1	1.4445646727635388
2	0.9587111070433345

length: 2

1	0
2	0
3	1

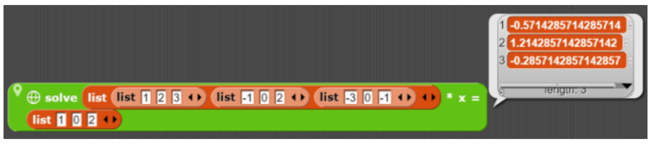
length: 3

Polynomials are processed by *MathSprite* as lists of their coefficients. The coefficient of the highest power is on the left. $x^2 - 2x + 1$ is written as `list 1 -2 1` The value for a given argument is calculated by the block ***<polynomial> polynomial (<argument>)***.



```
list 1 -2 1 polynom ( 3 )
```

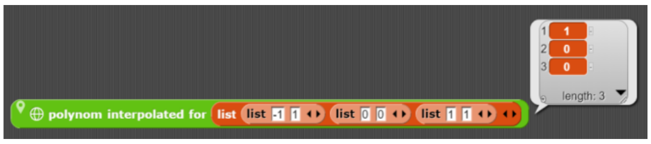
solve <matrix> * x = <vector> solves linear systems of equations - if there is a solution.



```
solve list (list 1 2 3) (list -1 0 2) (list -3 0 -1) * x = list 1 0 2
```

```
1 -0.5714285714285714
2 1.2142857142857142
3 -0.2857142857142857
length: 3
```

If a list of points is given, the block ***polynomial interpolated for <points>*** calculates a polynomial whose graph runs through these points.



```
polynom interpolated for list (list -1 1) (list 0 0) (list 1 1)
```

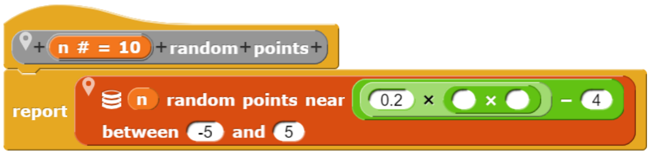
```
1 1
2 0
3 0
length: 3
```

Applications of these blocks follow.

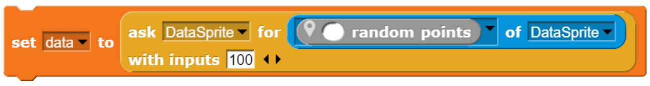
<u>Example</u>: Curve through ***n*** points

We need three sprites: a *DataSprite* to generate the random points, a *MathSprite* to calculate the interpolation polynomial and a *PlotSprite* to plot the results. Everything will be controlled by a fourth sprite called *Control*. The points for the interpolation are to be selected by mouse.

<u>Generation of the random data</u>: In the *DataSprite* we write a function to generate the points. We call this function from Control.

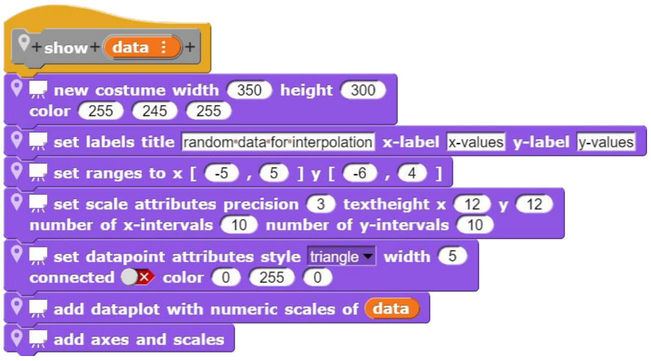


```
+ n # = 10 + random + points +
report ( n random points near ((0.2 × ( ) × ( )) − 4) between -5 and 5 )
```

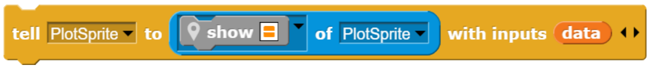


```
set data to ask DataSprite for ( random points ) of DataSprite with inputs 100
```

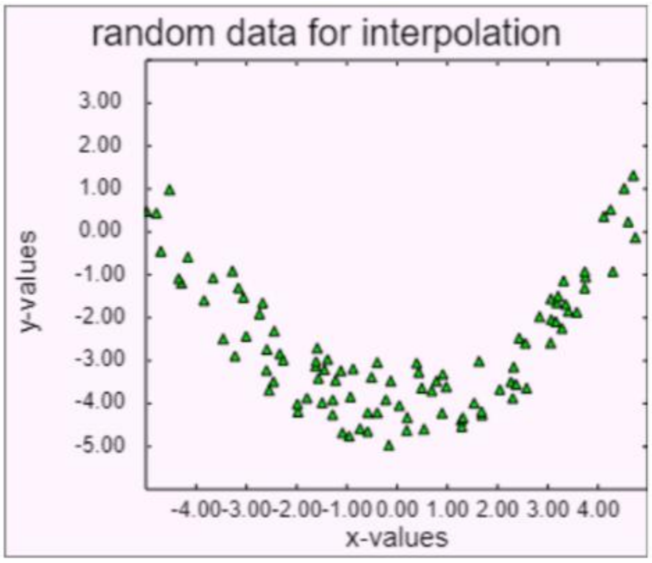
<u>Random data display</u>: We write a method in *PlotSprite* to display the data. We pass the data to this method from *Control*.



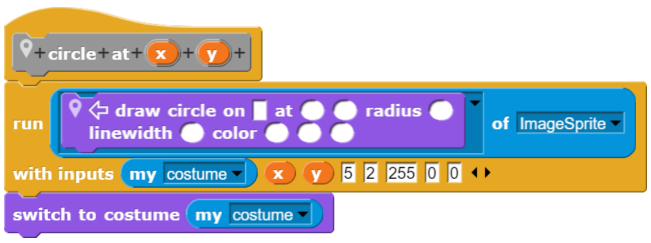
```
+ show + data +
new costume width 350 height 300 color 255 245 255
set labels title random data for interpolation x-label x-values y-label y-values
set ranges to x [ -5 , 5 ] y [ -6 , 4 ]
set scale attributes precision 3 textheight x 12 y 12 number of x-intervals 10 number of y-intervals 10
set datapoint attributes style triangle width 5 connected ✗ color 0 255 0
add dataplot with numeric scales of data
add axes and scales
```



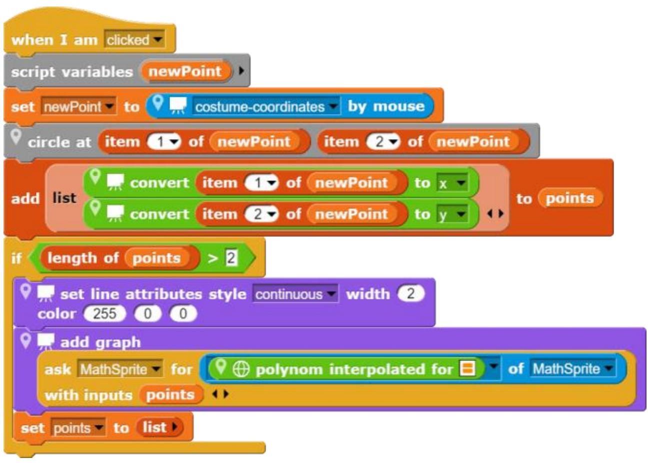
```
tell PlotSprite to ( show ) of PlotSprite with inputs data
```



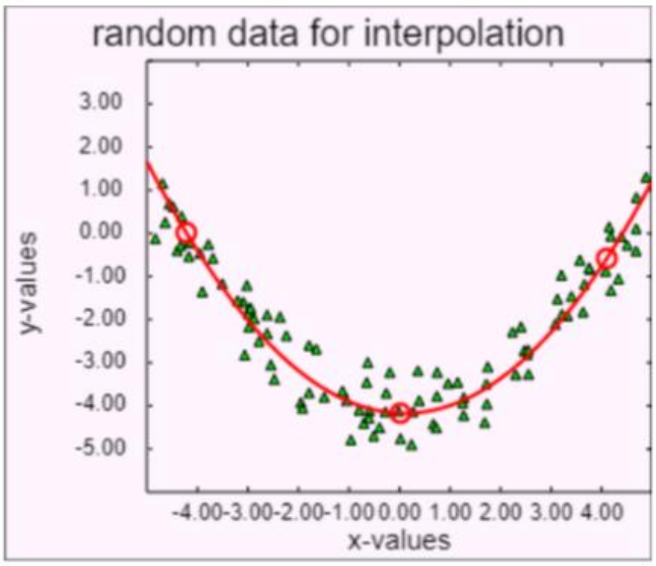
<u>Interpolation data collection with the mouse</u>: We add an event handling method to *PlotSprite* that reacts to mouse clicks. With ***<costume-coordinates> by mouse*** we get the costume coordinates and store them in the variable ***new-Point***. To see the points, we import the "*exportable*" ***draw circle*** method of the *ImageSprite*. Since it requires quite a lot of parameters and we only want to draw small red circles, we use a helper method ***circle at*** ***<x><y>***, which only needs the current coordinates - the rest is already filled in. Next, we save the costume coordinates converted to graph coordinates in the list ***points***. If we've clicked three times, we let the *MathSprite* calculate a polynomial through these points. We draw this polynomial a little bit thicker in red.



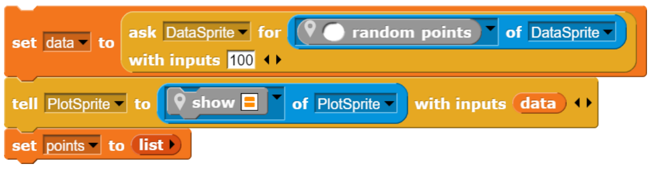
```
+circle+at+(x)+(y)+
run (draw circle on [] at () () radius () linewidth () color () () () of ImageSprite) with inputs (my costume) (x) (y) 5 2 255 0 0
switch to costume (my costume)
```



```
when I am clicked
script variables newPoint
set newPoint to (costume-coordinates by mouse)
circle at (item 1 of newPoint) (item 2 of newPoint)
add (list (convert (item 1 of newPoint) to x) (convert (item 2 of newPoint) to y)) to points
if (length of points) > 2
    set line attributes style continuous width 2 color 255 0 0
    add graph (ask MathSprite for (polynom interpolated for []) of MathSprite) with inputs points)
    set points to list
```



In *Control* we prepare everything in advance for the data acquisition.



```
set data to (ask DataSprite for (random points ()) of DataSprite) with inputs 100)
tell PlotSprite to (show []) of PlotSprite) with inputs data
set points to list
```

<u>**Tasks**</u>:

1. a: Create "*point clouds*" which scatter around other fully rational function graphs.

   b: As in the example, define some points in these clouds through which an interpolation polynomial is to be drawn.

   c: Have these polynomials drawn.

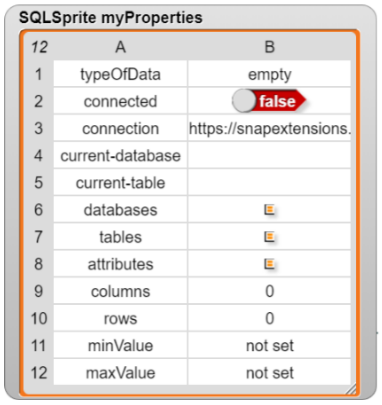
2. a: Experiment with the number of selected points. Will the results be better if you select more points?

   b: Create "*point clouds*" which scatter around non-rational function graphs (trigonometric, ...). Can you also describe them by interpolation polynomials?

   c: Formulate a rule, when and how interpolation polynomials can be used meaningfully - and why just like that.

4.8 The SQLSprite

Similar to the *MathSprite*, the *SQLSprite* is intended as a service provider for projects. It encapsulates the functionality for database access and should therefore only be loaded when needed.

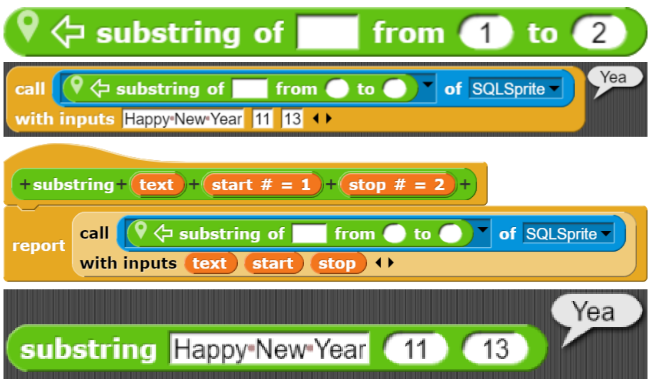


SQLSprite myProperties

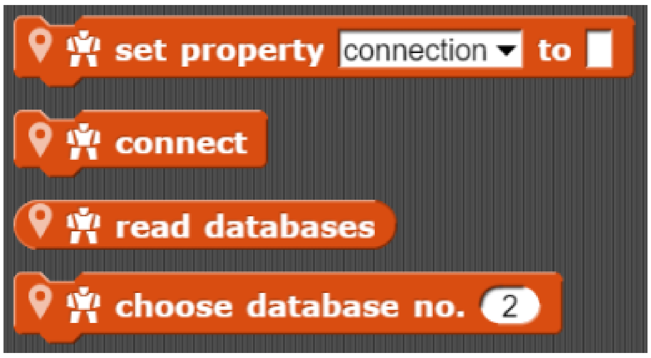
12	A	B
1	typeOfData	empty
2	connected	false
3	connection	https://snapextensions.
4	current-database	
5	current-table	
6	databases	
7	tables	
8	attributes	
9	columns	0
10	rows	0
11	minValue	not set
12	maxValue	not set

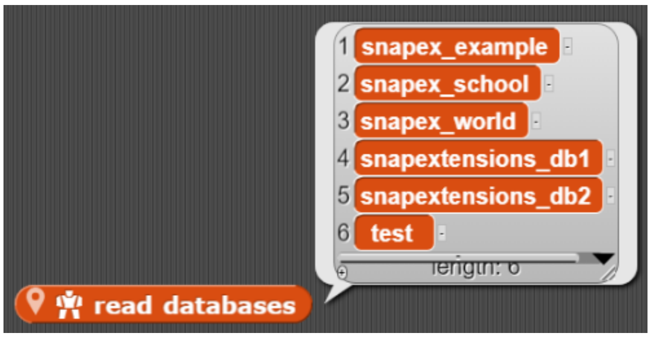
The few properties of the *SQLSprite* essentially contain the connection data to a sample database server and the current state of the connection (database used, current table, ...). It is possible to create clones of *SQLSprite*, but there will be few projects where this is necessary. Also new costumes are rarely needed. Most of the time the SQLSprite will just "lie around somewhere" and show by the color of its costume if the connection to the server is established.

Outside the *Variables* palette, there is only one operator block to determine the part of a string - this is needed from time to time to evaluate the query results. Since no local data is required, the block is marked as "*exportable*" by the left arrow. It can therefore be executed by all sprites regardless of context. If this happens often, you should write a global method ***substring*** to make the call easier.

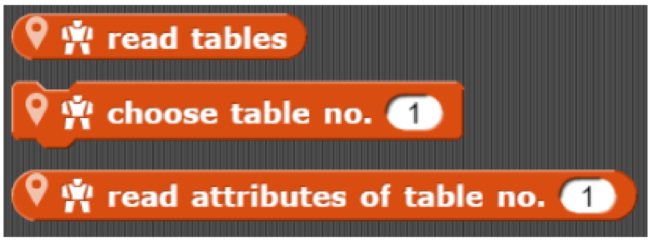

substring of from 1 to 2
call substring of from to of SQLSprite
with inputs Happy New Year 11 13
Yea
+substring+ text + start # = 1 + stop # = 2 +
report call substring of from to of SQLSprite
with inputs text start stop
Yea
substring Happy New Year 11 13

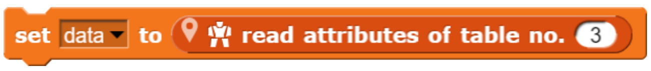
In the *Variables* palette you will find the actual SQL blocks. If you want to establish a connection, you can either use the default server or set a new connection with ***set property****<connection>* and the connection data. Then the block connect should establish the connection. If this works, the SQLSprite changes to the green costume. The available databases are listed with ***read databases***. One of them can be selected with ***choose database***.





The handling of tables is correspondingly. A specific one can also be selected and displayed.





```
set data to read attributes of table no. 3
```

In practice, the details of the tables in the database used are always needed. If you assign the table attributes to a variable one after the other and let it be displayed with "*open in dialog*", you can place the corresponding table data in the *Snap!* window and start the queries.

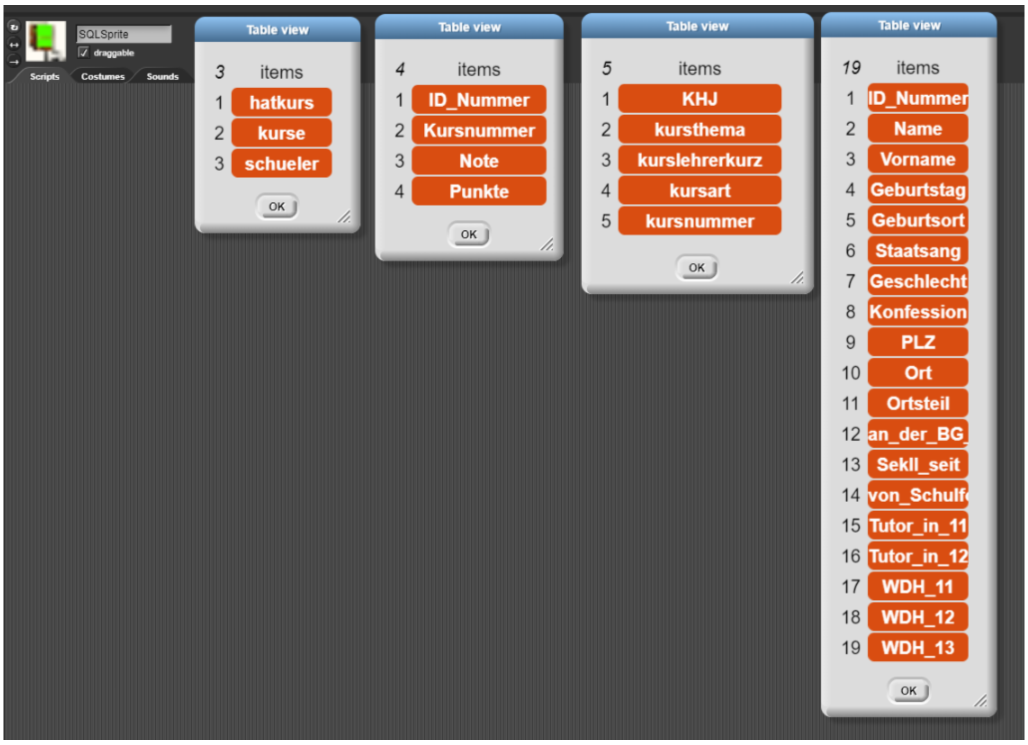


Table view

3	items
1	hatkurs
2	kurse
3	schueler

Table view

4	items
1	ID_Nummer
2	Kursnummer
3	Note
4	Punkte

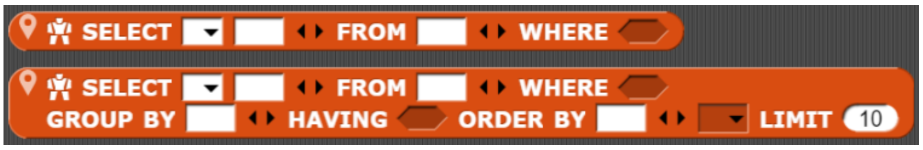
Table view

5	items
1	KHJ
2	kursthema
3	kurslehrerkurz
4	kursart
5	kursnummer

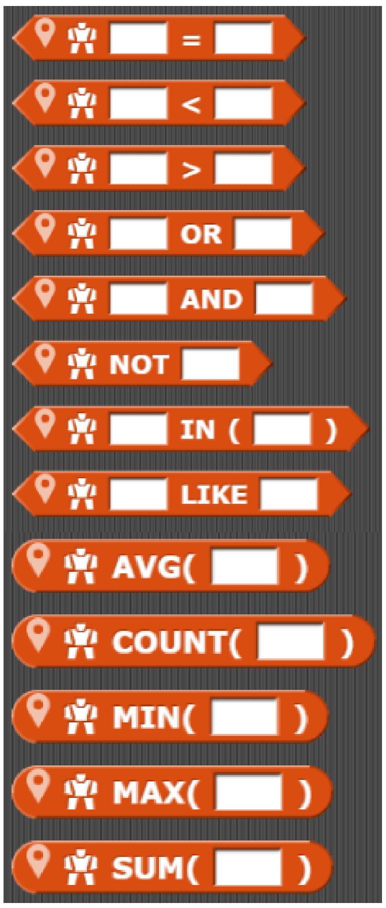
Table view

19	items
1	ID_Nummer
2	Name
3	Vorname
4	Geburtstag
5	Geburtsort
6	Staatsang
7	Geschlecht
8	Konfession
9	PLZ
10	Ort
11	Ortsteil
12	an_der_BG
13	SekII_seit
14	von_Schulf
15	Tutor_in_11
16	Tutor_in_12
17	WDH_11
18	WDH_12
19	WDH_13

SQL queries are composed as SELECT statements. There are two blocks for this: one for simple statements, one for (almost) complete statements.



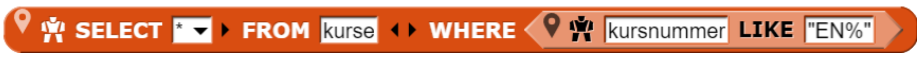
```
SELECT [ ▾] [ ] ◂▸ FROM [ ] ◂▸ WHERE < >

SELECT [ ▾] [ ] ◂▸ FROM [ ] ◂▸ WHERE < >
GROUP BY [ ] ◂▸ HAVING < > ORDER BY [ ] ◂▸ [ ▾] LIMIT (10)
```



```
[ ] = [ ]
[ ] < [ ]
[ ] > [ ]
[ ] OR [ ]
[ ] AND [ ]
NOT [ ]
[ ] IN ( [ ] )
[ ] LIKE [ ]
AVG( [ ] )
COUNT( [ ] )
MIN( [ ] )
MAX( [ ] )
SUM( [ ] )
```

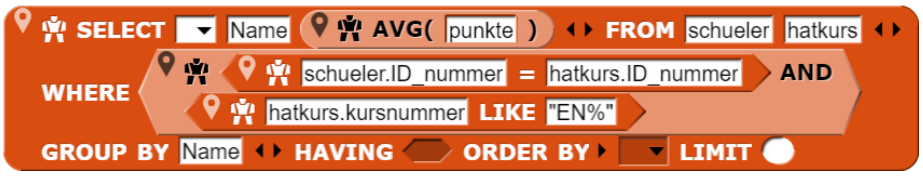
The standard predicates and functions of SQL are used to compose the queries.

Example: a simple SQL query



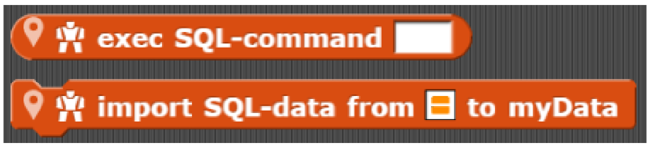
```
SELECT [* ▾] ▸ FROM [kurse] ◂▸ WHERE <[kursnummer] LIKE ["EN%"]>
```

Example: a more complex SQL query

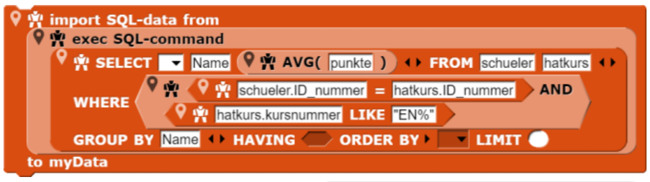


```
SELECT [ ▾] [Name] (AVG( [punkte] )) ◂▸ FROM [schueler] [hatkurs] ◂▸
WHERE < <[schueler.ID_nummer] = [hatkurs.ID_nummer]> AND
        <[hatkurs.kursnummer] LIKE ["EN%"]> >
GROUP BY [Name] ◂▸ HAVING < > ORDER BY ▸ [ ▾] LIMIT ( )
```

The two SELECT blocks generate the text of an SQL query, but do not execute it. The reason is simple: you should be able to view the query. If you are satisfied with it, it is executed with ***exec SQL-command***. In this block you can also enter other SQL commands, if you have the appropriate rights on the server. With the help of ***import SQL-data ...*** these are converted into tables and moved into the data area of the SQLSprite.

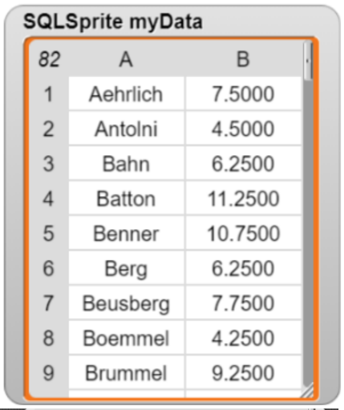


```
exec SQL-command [ ]
import SQL-data from [ ] to myData
```



```
import SQL-data from
  exec SQL-command
    SELECT [Name] (AVG( [punkte] )) FROM [schueler] [hatkurs]
    WHERE ( ( [schueler.ID_nummer] = [hatkurs.ID_nummer] ) AND ( [hatkurs.kursnummer] LIKE ["EN%"] ) )
    GROUP BY [Name] HAVING < > ORDER BY [ ] LIMIT ( )
to myData
```

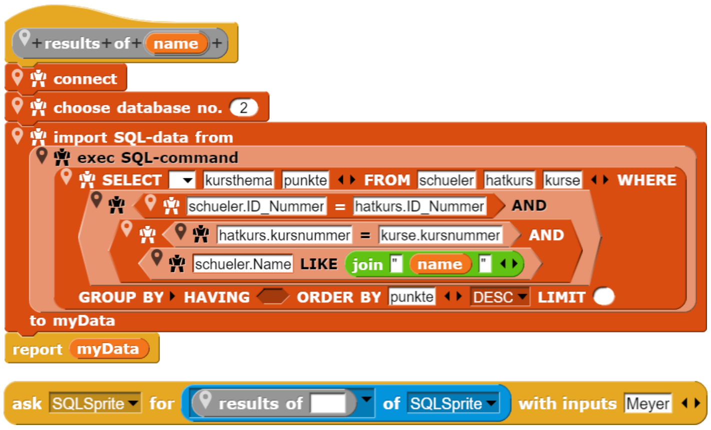
More complex instructions can be assembled and tested from these blocks. If the result meets the expectations, the query can be encapsulated in one block, which only contains the relevant parameters and can be used by other sprites without detailed knowledge of SQL.



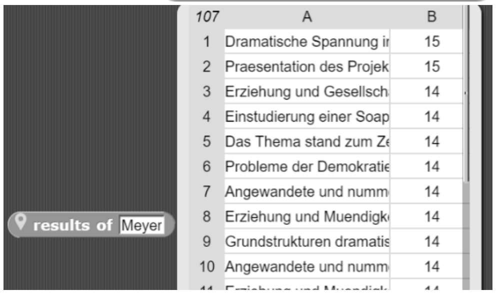
SQLSprite myData

82	A	B
1	Aehrlich	7.5000
2	Antolni	4.5000
3	Bahn	6.2500
4	Batton	11.2500
5	Benner	10.7500
6	Berg	6.2500
7	Beusberg	7.7500
8	Boemmel	4.2500
9	Brummel	9.2500

Example: The course titles and grades for all courses of a learner, sorted by grade in descending order, are searched.



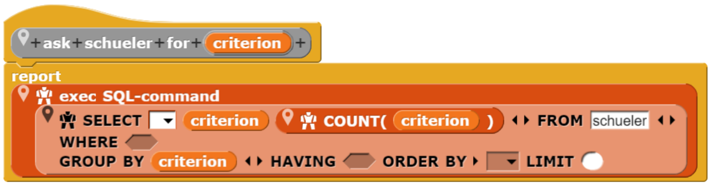
```
+results+of+ (name) +
connect
choose database no. (2)
import SQL-data from
  exec SQL-command
    SELECT [kursthema] [punkte] FROM [schueler] [hatkurs] [kurse] WHERE
      ( ( [schueler.ID_Nummer] = [hatkurs.ID_Nummer] ) AND
        ( ( [hatkurs.kursnummer] = [kurse.kursnummer] ) AND
          ( [schueler.Name] LIKE (join ["] (name) ["]) ) ) )
    GROUP BY HAVING < > ORDER BY [punkte] [DESC] LIMIT ( )
to myData
report (myData)

ask [SQLSprite] for (results of [ ]) of [SQLSprite] with inputs [Meyer]
```



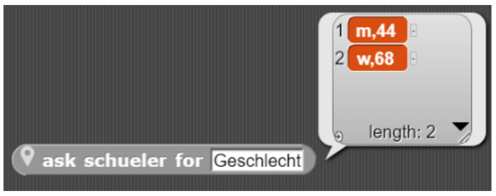
```
results of [Meyer]
```

107	A	B
1	Dramatische Spannung ir	15
2	Praesentation des Projek	15
3	Erziehung und Gesellsch	14
4	Einstudierung einer Soap	14
5	Das Thema stand zum Ze	14
6	Probleme der Demokratie	14
7	Angewandete und numm	14
8	Erziehung und Muendigk	14
9	Grundstrukturen dramatis	14
10	Angewandete und numm	14

Example: For statistical purposes, the ***schueler*** table should be searchable according to different criteria.



```
+ask+schueler+for+ (criterion) +
report
  exec SQL-command
    SELECT [ ] (criterion) (COUNT( (criterion) )) FROM [schueler]
    WHERE < >
    GROUP BY (criterion) HAVING < > ORDER BY [ ] LIMIT ( )
```

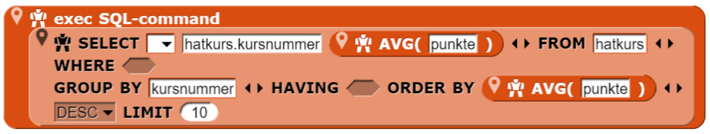


```
ask schueler for [Geschlecht]
```

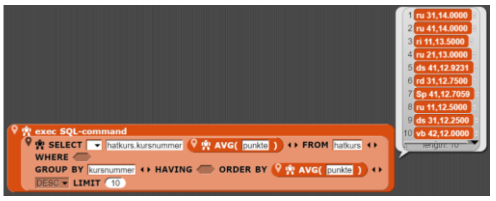
1	m,44
2	w,68

length: 2

Example: The 10 courses with the "*best*" results should be sorted in descending order.



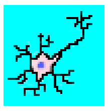
```
exec SQL-command
  SELECT [ ] [hatkurs.kursnummer] (AVG( [punkte] )) FROM [hatkurs]
  WHERE < >
  GROUP BY [kursnummer] HAVING < > ORDER BY (AVG( [punkte] ))
  [DESC] LIMIT (10)
```



1	ru 31,14.0000
2	ru 41,14.0000
3	ri 11,13.5000
4	ru 21,13.0000
5	ds 41,12.9231
6	rd 31,12.7500
7	Sp 41,12.7059
8	ru 11,12.5000
9	ds 31,12.2500
10	vb 42,12.0000

length: 10

4.9 The NeuralNetSprite

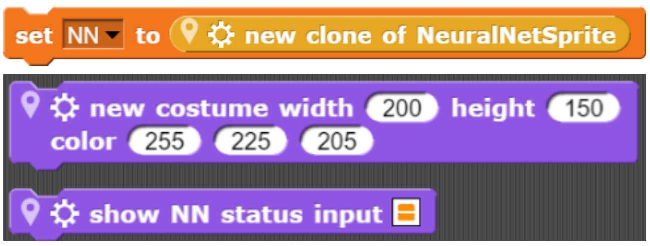


"*Deep*" neural networks are shaping the discussion on current "*artificial intelligence*". These are usually "*fully connected*" networks consisting of several perceptron layers. "*Fully connected*" means that all neurons of one layer are connected to all of the next layer. Each connection is assigned a *weight*, from which its influence on the connected perceptron is derived - but you'd better read this elsewhere.

Let's have a look at a net of three layers that receives the pixels of a current 20 M-pixel-photo as input, thus 2x10^7 pixels. The input layer consists of 3x2x10^7 MB numerical values between 0 and 255 (if we leave out the transparency byte). To the next layer, there are then (6x10^7)^2=3.6x10^{15} connections - and then twice more. In total, 3x3.6x10^{15}, i.e. about 10^{16} weights, would have to be determined - a completely utopian task for "*normal*" computers. So, we will have to restrict ourselves to somewhat smaller neural networks.

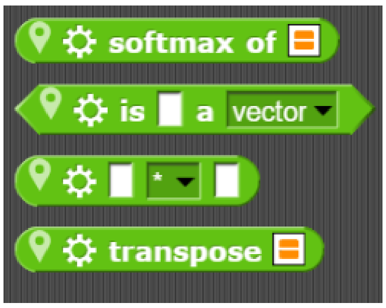
One way to train perceptron networks is to present them input vectors and the desired output right away. The net then calculates the output, which is the result of the existing, initially randomly selected weights, and determines the difference from the given result. Starting from the last result layer, it corrects the weights "*going backwards*" so that its output "*somewhat better*" matches the given result. The procedure is called ***backpropagation***. You should also inform yourself about this elsewhere. The trained net results from many such corrections. So, "*learning*" means to adapt the parameters (the weights) by means of many examples. With the help of these parameters, the net determines an output vector from the input vector: it calculates a function value.

Our *NeuralNetSprite* can simulate and train such perceptron networks. Clones of the net can be created for this purpose - just like with the other *ML.Sprites*. For a *NeuralNetSprite* you can create costumes as usual and display the current state of the net on them. We have already described the new blocks in the *Control* and *Looks* palette.



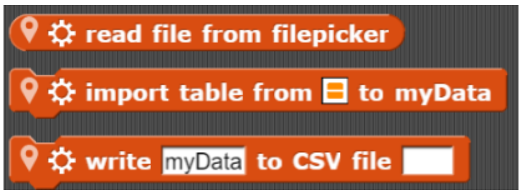
```
set NN to new clone of NeuralNetSprite
new costume width 200 height 150 color 255 225 205
show NN status input
```

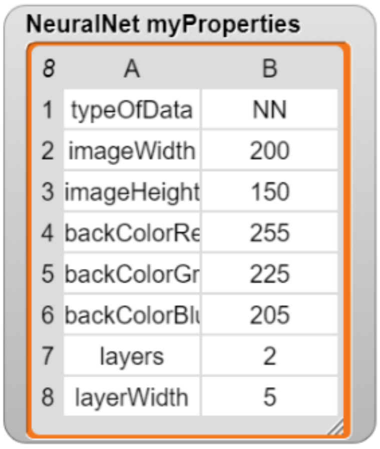
Our weights together form a ***tensor*** with ***m*** layers consisting of ***nxn*** matrices. *NeuralNetsSprites* should therefore master linear algebra. To avoid having to ask the *MathSprite* every time, the *NeuralNetSprite* knows the most important operations itself. They can be found in the *Operators* palette. The only new feature is the ***Softmax*** function, which can be used to scale input vectors, for example. You should also inform yourself about this.



```
softmax of
is  a vector
 * 
transpose
```

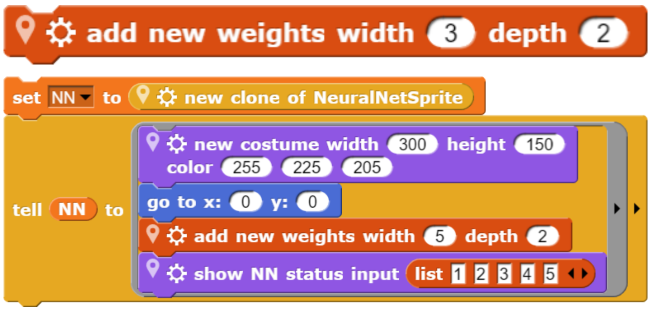
The actual *NeuralNet* blocks are therefore found in the *Variables* palette. On the one hand, the few properties of the sprite are managed there, on the other hand, the weight tensor can be loaded as usual and saved again as a CSV file.




NeuralNet myProperties

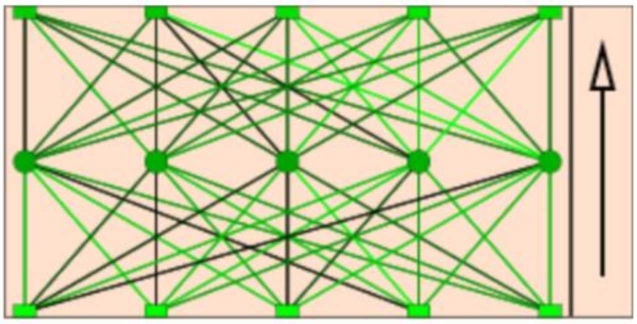
8	A	B
1	typeOfData	NN
2	imageWidth	200
3	imageHeight	150
4	backColorRe	255
5	backColorGr	225
6	backColorBl	205
7	layers	2
8	layerWidth	5

The dimensioning and initial assignment of the network is done in the block ***add new weights***. With these blocks we can create and display a new neural net of any size. In this case, it has the ***width*** 5 and the ***depth*** 2. The result of the calculations with the input vector to be specified is displayed.

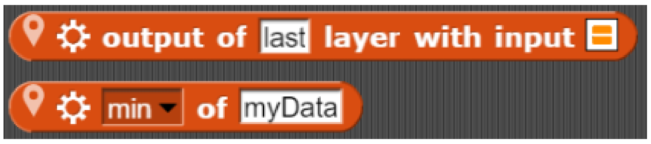

```
add new weights width 3 depth 2

set NN to new clone of NeuralNetSprite
tell NN to
    new costume width 300 height 150
    color 255 225 205
    go to x: 0 y: 0
    add new weights width 5 depth 2
    show NN status input list 1 2 3 4 5
```

As the display of the many numbers would be rather confusing and also hardly informative, the connecting lines (the *edges*) are color-coded according to the values of the corresponding weights: from full green for large positive values to black for small amounts to red negative weights. As at the beginning only positive numbers are generated by random generator, new net is predominantly green.

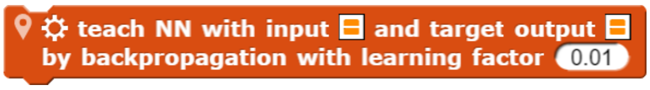
The *nodes* of the net are color-coded like the edges. Below are the elements of the input vector as small rectangles. The inner layers form colored circles and the last layer is displayed as output layer again rectangular. The direction of the calculation from bottom to top is shown by the arrow on the far right. But since sprites can be easily rotated, the direction can be displayed differently.



Often you need the results of the last or even an inner layer of the network. These can be calculated using the ***output of...*** block for a given input. Since the color coding does not necessarily show the largest or smallest element clearly, it can be determined using the ***of ...*** block.



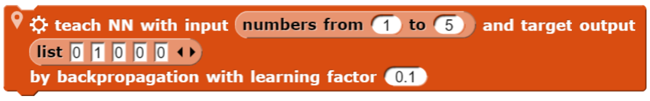
Now we have to train the net. In block ***teach NN ...*** this is done by backpropagation with a learning factor to be specified. At the beginning this factor may be a little bit bigger, but then it can be reduced.



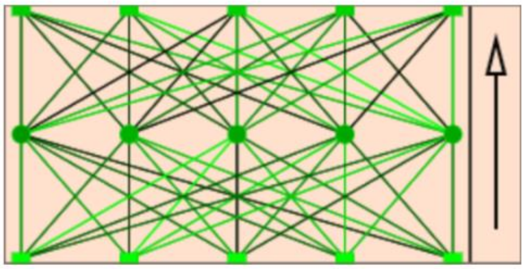
```
teach NN with input ☐ and target output ☐
by backpropagation with learning factor (0.01)
```

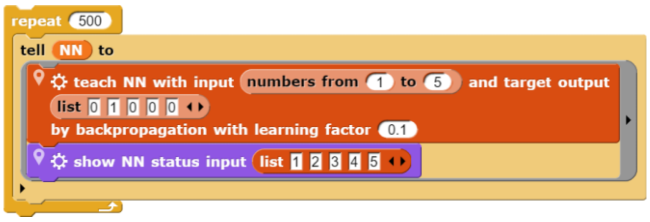
Example: A neural network should learn to calculate the output vector ***<0,1,0,0,0>*** from the input vector ***<1,2,3,4,5>***. If it did this successfully, the second output rectangle should appear light green, the rest darker.

So we create a new neural net as shown above and repeatedly apply the ***teach*** block. The state of the net is shown after each 20 repetitions.



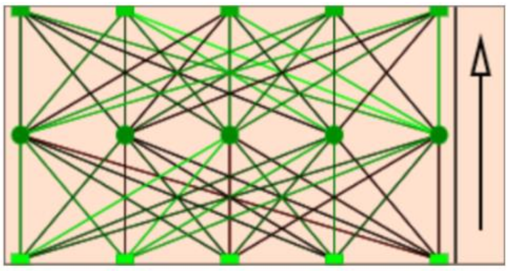
```
teach NN with input (numbers from (1) to (5)) and target output
(list 0 1 0 0 0)
by backpropagation with learning factor (0.1)
```

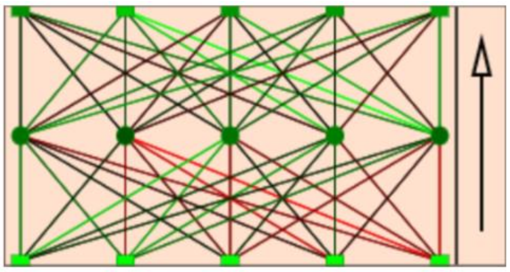


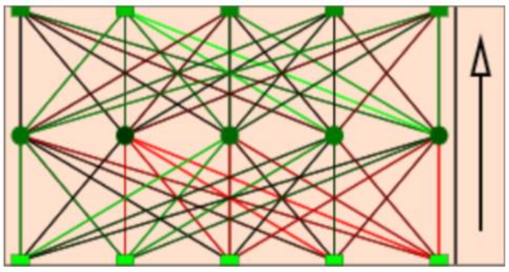


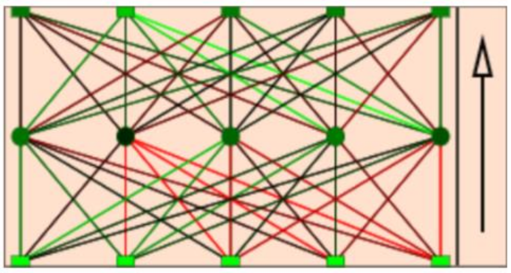
```
repeat (500)
  tell (NN) to
    teach NN with input (numbers from (1) to (5)) and target output
    (list 0 1 0 0 0)
    by backpropagation with learning factor (0.1)
    show NN status input (list 1 2 3 4 5)
```

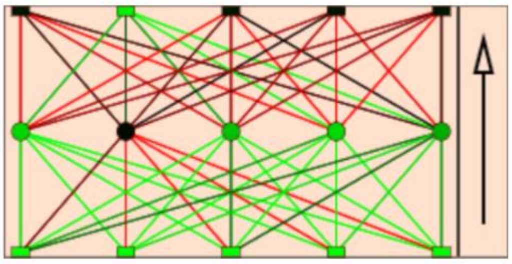
And after about 200 more passes the pattern has clearly formed. The net maps the input vector to <0.1.0.0.0> reliably. You can see quite clearly that the first layer configures the inner layer in such a way that it can in turn strengthen (green) or inhibit (red) the output neurons with the correct weighting.











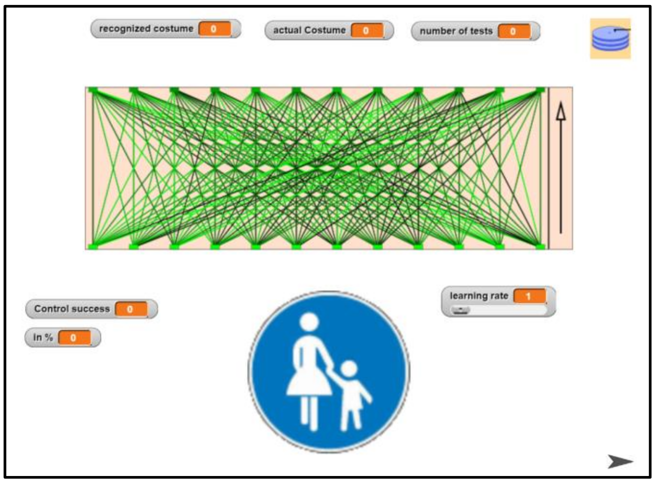
Of course, the net should not only recognize one pattern correctly. In training, it is given various other input vectors in random order with the desired outputs, with the weights being slightly modified so that the strongest output neuron has the correct place. Even such a small network has enough parameters for this. Examples of applications are given in the next section.

<u>Example</u>: Traffic sign recognition

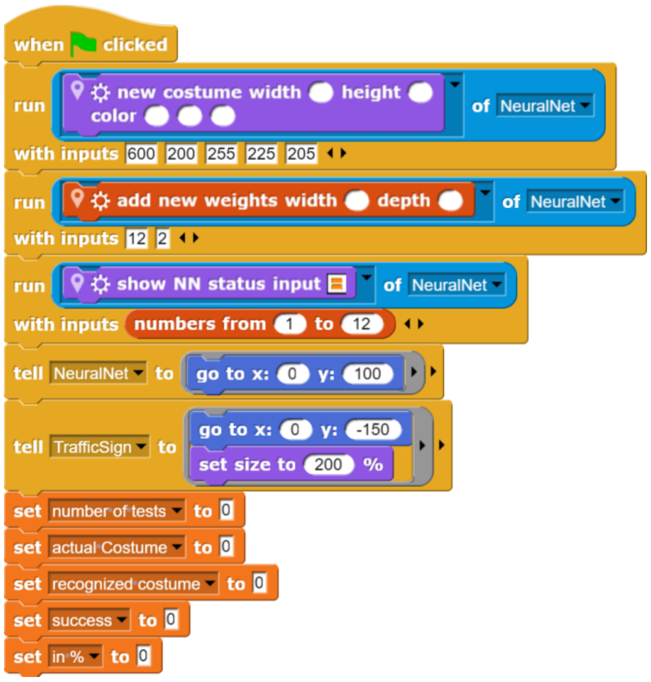
We want to train a neural network (NN) to recognize 12 different traffic signs. For this purpose, we search for images of these traffic signs in the internet and reduce them to the format 100 x 100 pixels. You can display them on the screen, but the 10000 pixels are of course much too much for an NN.

To bring the amount of data within tolerable limits, we reduce the pixels to a 2x2 format by ***mean-pooling***, i.e. we form the mean values of the color pixels in the four quadrants of the image. From the 30000 values of the traffic sign image we derive 12.

Since only the *DataSprite* can handle the pooling operation, we import it along with the *NeuralNetSprite* to the project with the *TrafficSignSprite*. As a fourth sprite we add a *ControlSprite* that should control everything.

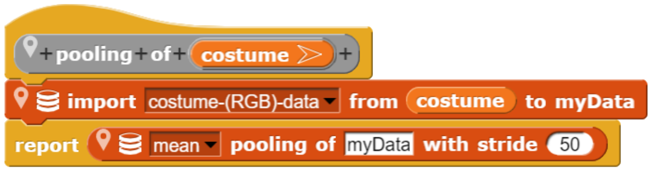


For the start we give the NN-Sprite a new costume. Since we need to specify several parameters for size and color, we use the run block. Then we create the weights for a new (here: 12x2) net in NN. This net is drawn with a (still senseless) input. Next we send the NN to a well-chosen place in the upper middle and do the same with the traffic sign down there. Finally, we set some variables to 0. We need them later.



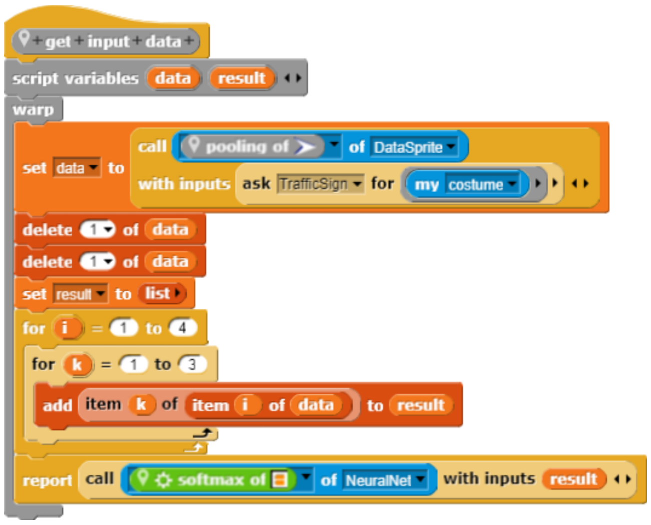
```
when [flag] clicked
run (new costume width ( ) height ( ) color ( ) ( ) ( ) of NeuralNet) with inputs 600 200 255 225 205
run (add new weights width ( ) depth ( ) of NeuralNet) with inputs 12 2
run (show NN status input [ ] of NeuralNet) with inputs (numbers from 1 to 12)
tell NeuralNet to (go to x: 0 y: 100)
tell TrafficSign to (go to x: 0 y: -150
                     set size to 200 %)
set number of tests to 0
set actual Costume to 0
set recognized costume to 0
set success to 0
set in % to 0
```

We only need the *DataSprite* for calculation. To use the ***pooling*** operation, the sprite must import the image data. Then it can convert it and return the result. We summarize everything in a new block ***pooling of*** *<costume>*, which we call from *Control*.



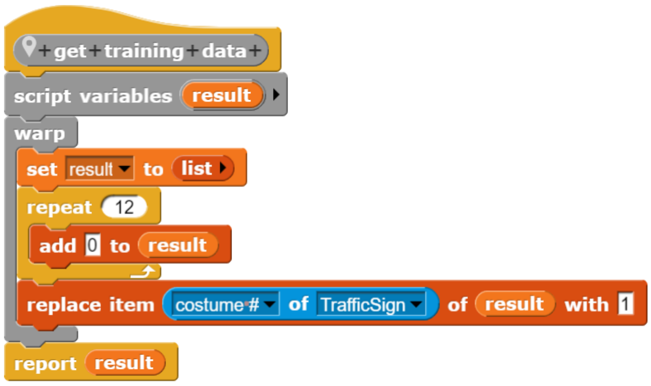
```
+pooling+of+ costume +
import costume-(RGB)-data from costume to myData
report (mean pooling of myData with stride 50)
```

The color values of the reduced image are combined into an input vector using the new block of *Control*. The first two values of the ***pooling*** result specify the new image dimensions and are deleted. The vector from ***get input data*** thus contains the desired 12 values. These are modeled using the *NeuralNetSprite's* ***Softmax*** function to eliminate unfavorable input values.



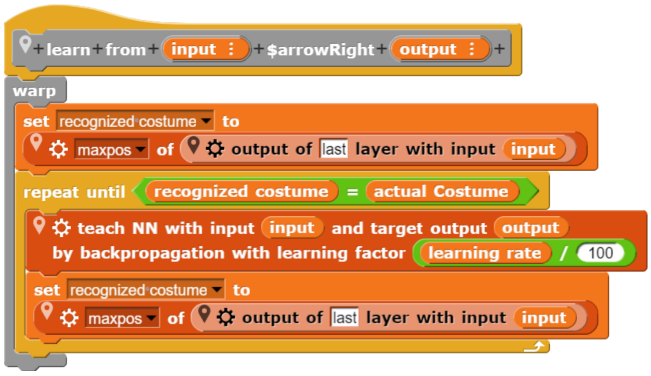
```
+get+input+data+
script variables data result ◂▸
warp
    set data to call (pooling of >) of DataSprite
        with inputs ask TrafficSign for (my costume) ▸ ◂▸
    delete 1 of data
    delete 1 of data
    set result to list ▸
    for i = 1 to 4
        for k = 1 to 3
            add item k of item i of data to result
report call (softmax of ☰) of NeuralNet with inputs result ◂▸
```

Accordingly, the training vector in ***get training data*** can be determined with the output values of the NN you are looking for. In our case all values should be 0 except the one corresponding to the costume number of the traffic sign.



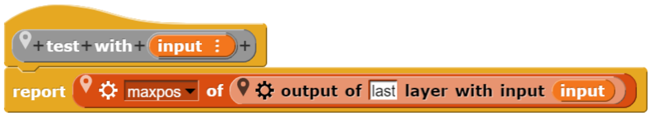
```
+get+training+data+
script variables result ▸
warp
    set result to list ▸
    repeat 12
        add 0 to result
    replace item (costume# of TrafficSign) of result with 1
report result
```

The NN gets two new methods ***learn from...*** and ***test with...*** During learning, the position of the place with the highest output value of the NN is determined and compared with the current costume number of the traffic sign. If these values do not match, learning continues.



```
+learn+from+ input ⋮ +$arrowRight+ output ⋮ +
warp
    set recognized costume to
        maxpos of output of last layer with input input
    repeat until recognized costume = actual Costume
        teach NN with input input and target output output
            by backpropagation with learning factor learning rate / 100
        set recognized costume to
            maxpos of output of last layer with input input
```

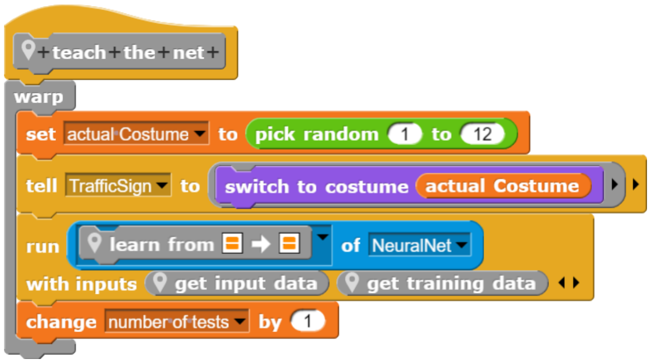
For testing, the same operation is performed only once.



```
+test+with+ input ⋮ +
report maxpos of output of last layer with input input
```

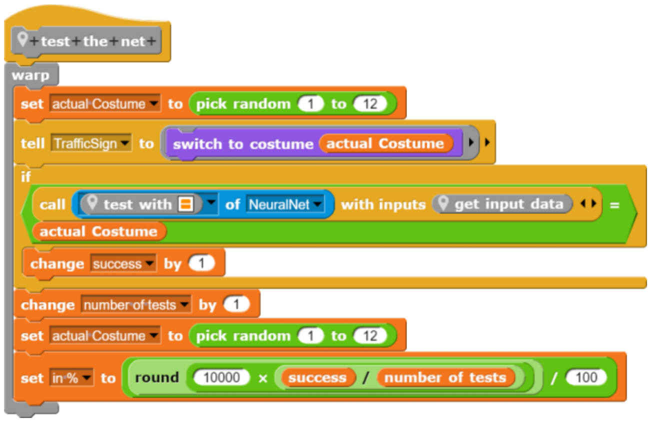
Now we have everything together to make *Control* work properly.

A teaching process consists of determining a random costume number with the corresponding costume change. Afterwards the learning process of the NN is started with new input and target data. The passes are counted.



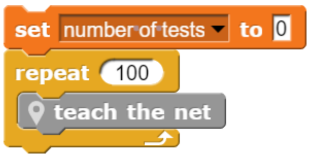
```
+teach+the+net+
warp
    set actual Costume to pick random 1 to 12
    tell TrafficSign to (switch to costume actual Costume) ▸
    run (learn from ☰ ➔ ☰) of NeuralNet
        with inputs (get input data) (get training data) ◂▸
    change number of tests by 1
```

The procedure for testing is similar: The costume is changed, and it is checked whether NN calculates the correct costume number. If this is the case, everyone is happy. Next, the percentage of correct attempts is determined.

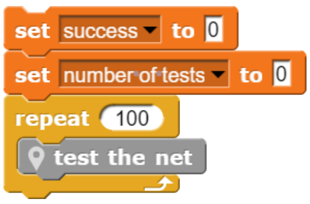


```
+test+the+net+
warp
  set actual Costume to pick random 1 to 12
  tell TrafficSign to switch to costume actual Costume
  if call test with ▤ of NeuralNet with inputs get input data = actual Costume
    change success by 1
  change number of tests by 1
  set actual Costume to pick random 1 to 12
  set in % to round 10000 × success / number of tests / 100
```

Several learning and test runs can be easily triggered.

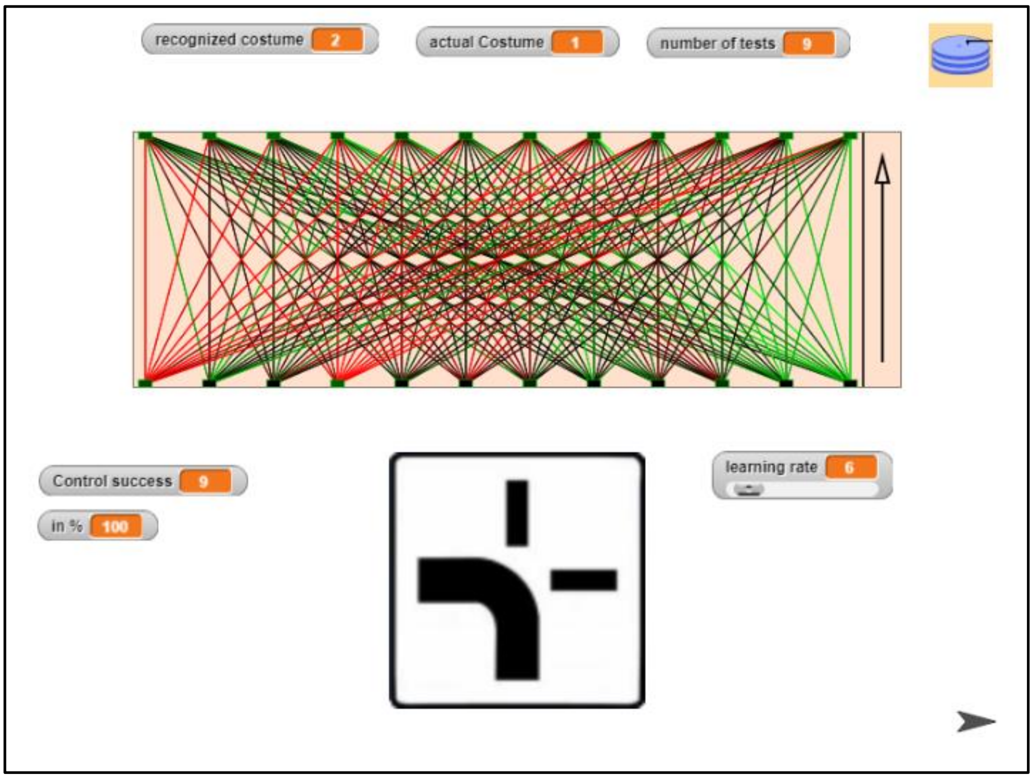


```
set number of tests to 0
repeat 100
  teach the net
```



```
set success to 0
set number of tests to 0
repeat 100
  test the net
```

After about 50 training runs with a higher learning rate and another 50 with a low one for fine tuning, we reach detection rates of 100%.



Tasks:

1. Train a single-layer net with different learning rates and numbers of learning passes. Determine the recognition rate in percentage terms.
2. Plot the results from 1. graphically using a *PlotSprite*.
3. Experiment with multi-layer NNs. Will the results be better?
4. Increase the length of the input vector by changing the pooling. Do the results get better?
5. Increase the number of recognizable labels by allowing more than one 1 in the output.

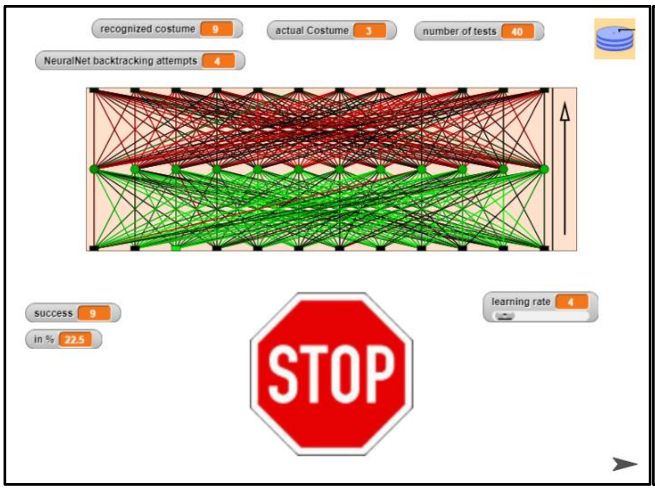
5 Applications with ML.Sprites

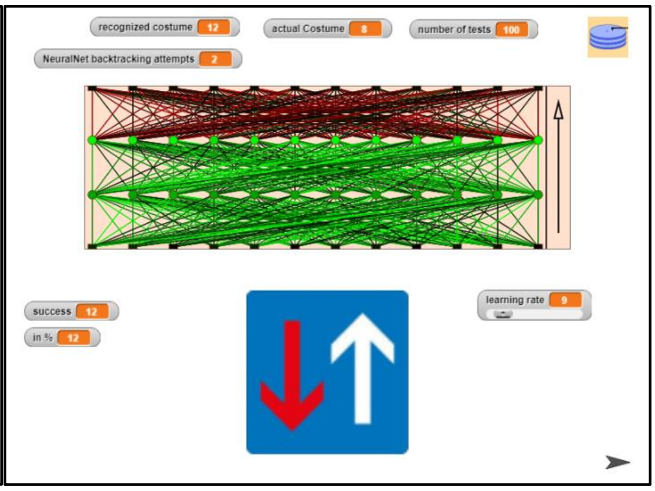
5.1 Under- and Overfitting

Machine learning uses training data to adjust the parameters of a function so that other values are predicted well - if everything works out. So, you build a forecasting instrument, a kind of "telescope" for data.

One might think that the more customizable parameters a function contains, the better such a function is. But this is not so. On the one hand, (1.) more parameters also require more training data and training runs, i.e. more learning time; on the other hand, (2.) an "inappropriate" number of parameters can also prevent "good" solutions. For both we now give an example.

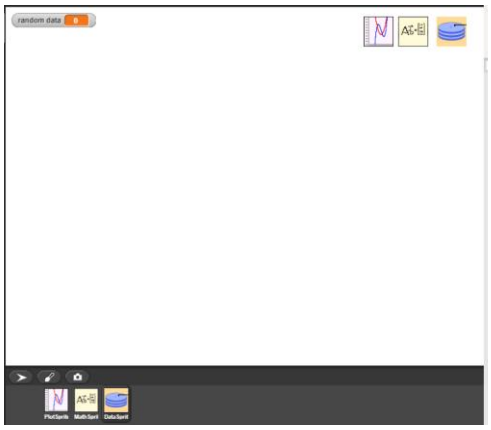
1: In the neural network for traffic sign recognition we have achieved very good results with one layer. If we increase the number of layers and leave the number of training runs the same, the recognition rate will deteriorate drastically.





2: If the training data are well reproduced by the function, this does not mean that this is also true for other data. It depends a lot on the type of function that is generated. As application we choose the example *polynomial interpolation*.

The task is to use training data to adjust the coefficients of a polynomial so that OTHER data are predicted as well as possible.

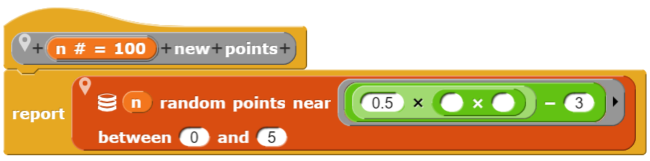


To do this, we have to generate data that can be used to calculate an interpolation polynomial. The functionalities for this are divided into three prototypes: the *PlotSprite* for plotting the graphs, the *MathSprite* for the polynomial interpolation and the *DataSprite* for generating random data that scatters around a given function.

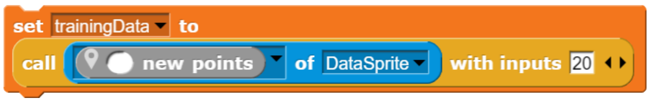
So, we create a new project, enlarge the stage to 800x600 pixels and load the three prototypes. Then we create a variable ***random data***. Now, the arrangement looks like the one on the right.

As "workhorse" we choose the *PlotSprite*. If necessary, we import required functionality from other sprites.

First of all, we ask the *DataSprite* to generate 20 random data scattered around the parabola $0{,}5 * x^2 - 3$. For this we write a function ***<n> new points*** in the *DataSprite*. We call this function from the *PlotSprite*.

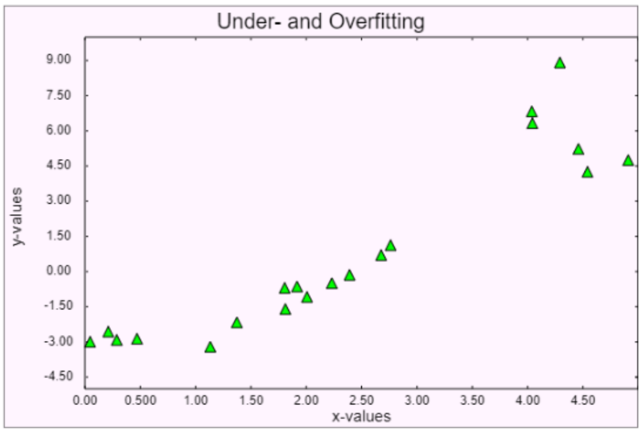


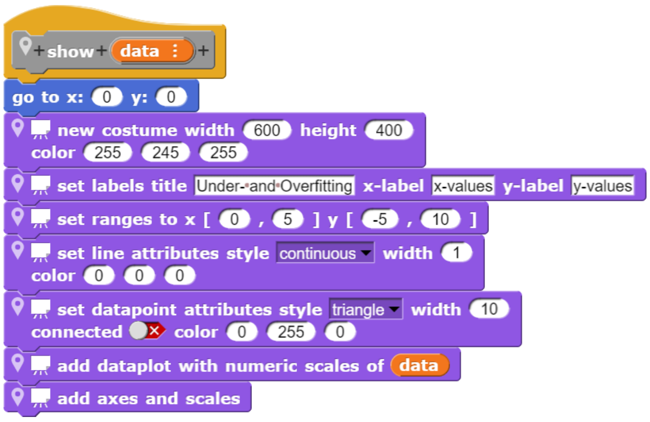
```
+ n # = 100 + new + points +
report ( n random points near ( ( 0.5 × ( ) × ( ) ) − 3 ) between 0 and 5 )
```



```
set trainingData to
call ( ( ) new points ) of DataSprite with inputs 20
```

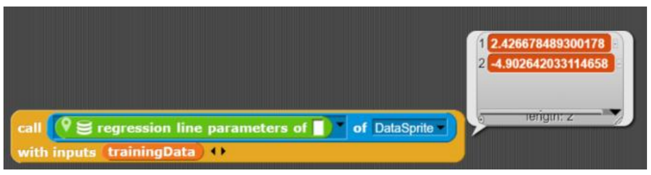
This is enough to display the data. We write a block ***show <data>***, which makes the necessary settings and does that.





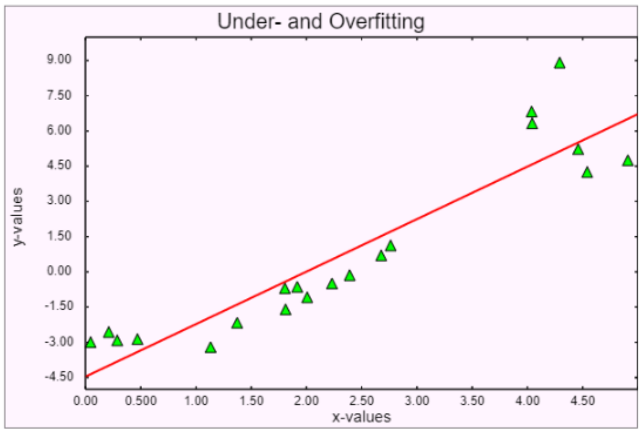
```
+ show + data : +
go to x: 0 y: 0
new costume width 600 height 400 color 255 245 255
set labels title Under-and-Overfitting x-label x-values y-label y-values
set ranges to x [ 0 , 5 ] y [ -5 , 10 ]
set line attributes style continuous width 1 color 0 0 0
set datapoint attributes style triangle width 10 connected ✗ color 0 255 0
add dataplot with numeric scales of data
add axes and scales
```

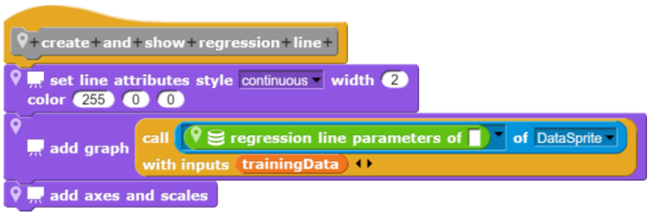
We first try the interpolation with a regression line. The *DataSprite* can calculate the required parameters.



```
call ( regression line parameters of [ ] ) of DataSprite
with inputs trainingData
```

```
1 2.426678489300178
2 -4.902642033114658
length: 2
```



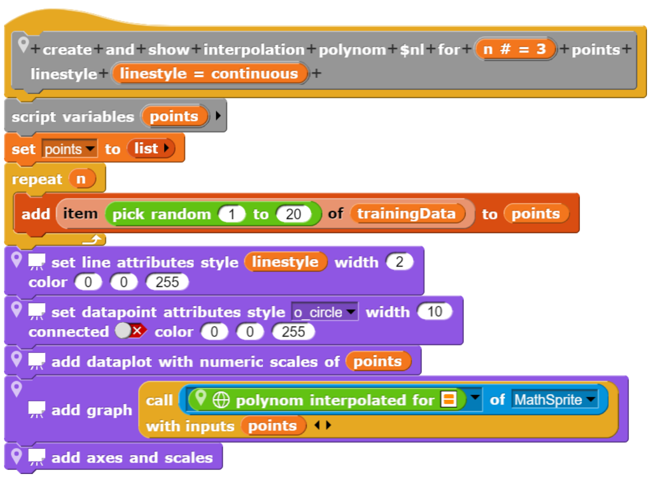


```
+ create + and + show + regression + line +
set line attributes style continuous width 2 color 255 0 0
add graph ( call ( regression line parameters of [ ] ) of DataSprite
            with inputs trainingData )
add axes and scales
```

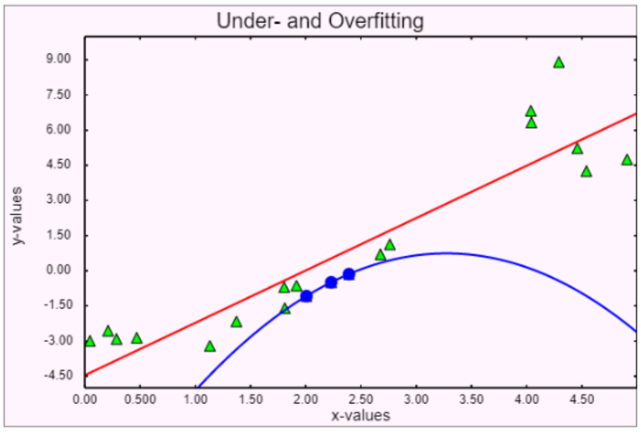
This looks actually quite nice, but on the sides it doesn't really fit.

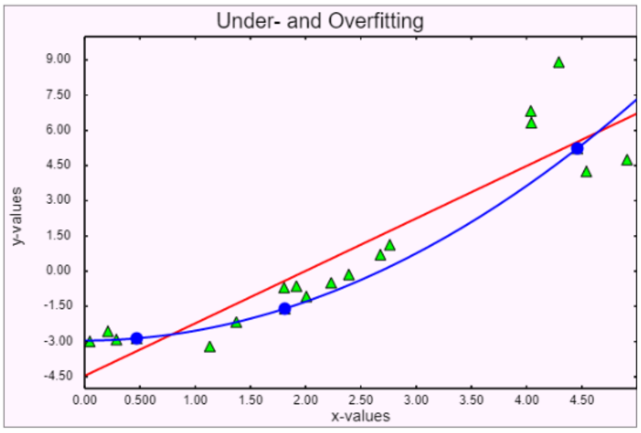
So, we try a polynomial interpolation.

First of all, we select three random pairs from the training data, determine the interpolation polynomial from it and draw it. Because we want to experiment further, we generalize the solution to a polynomial by ***n*** points. We hope that everything goes well with the selection! The results depend on which points were hit (blue). Enclosed a bad and a quite good result.

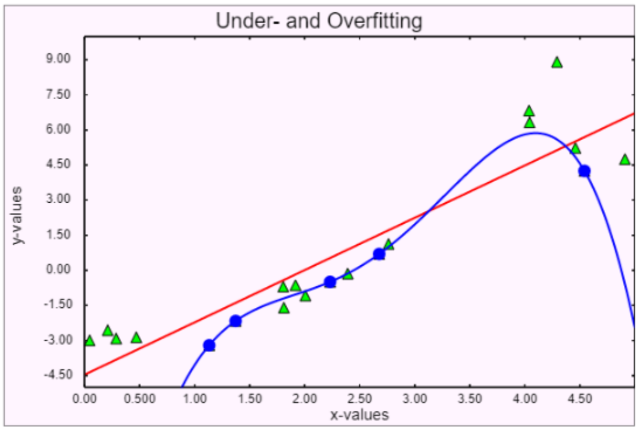


```
+create+and+show+interpolation+polynom+$nl+for+ (n # = 3) +points+
linestyle+ (linestyle = continuous) +
script variables (points)
set points to (list)
repeat (n)
    add (item (pick random 1 to 20) of (trainingData)) to (points)
set line attributes style (linestyle) width 2
color 0 0 255
set datapoint attributes style o_circle width 10
connected ✗ color 0 0 255
add dataplot with numeric scales of (points)
add graph (call (polynom interpolated for ▭) of MathSprite
with inputs (points))
add axes and scales
```

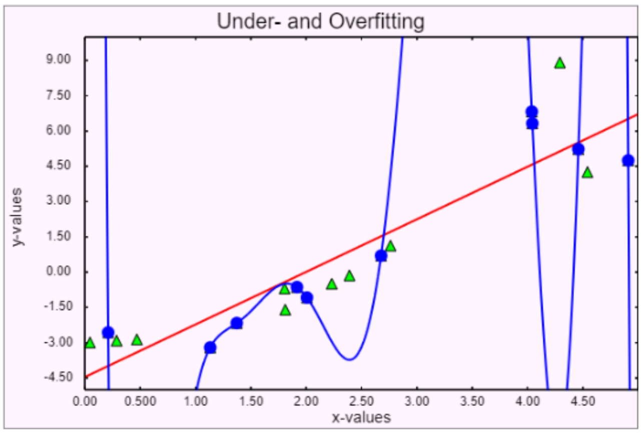


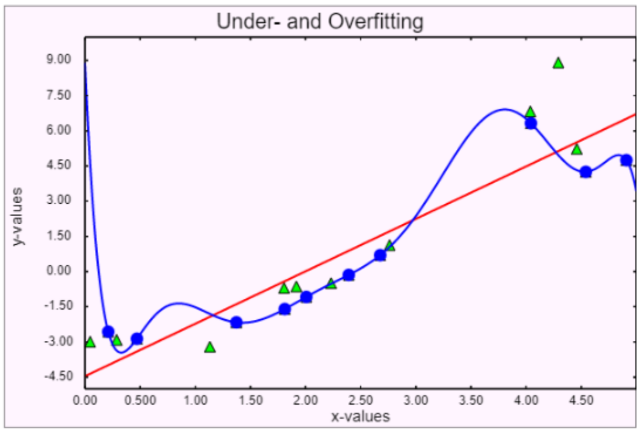


Now we're getting brave! Instead of three points we vote for 5. After all, we want to do a good job! That works out great in the middle, and then – oops!

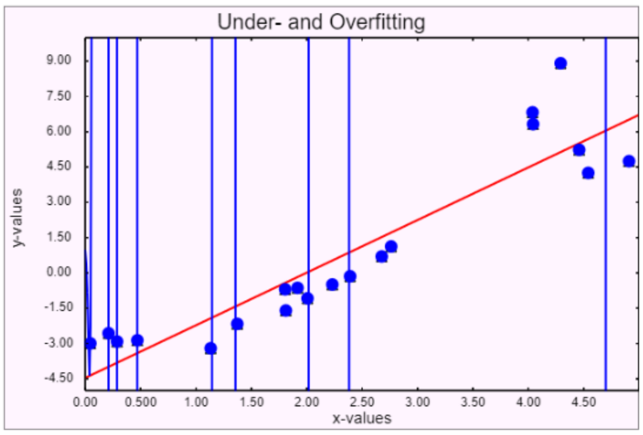


Maybe we just need to take more points. Let's try it with 10. The polynomials run through more points, but at the edges they "run away" - and in between mostly.





Well, then with all points!



You can see that with an increasing degree of the polynomial there is more training data directly on the graph, but in between only nonsensical values are "predicted" by the wild oscillations of the polynomial.

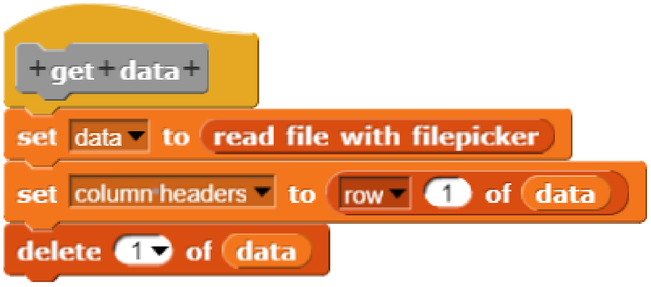
The quality of what is learned therefore depends very much on how we deal with deviations. We have to decide which inaccuracies can be tolerated in detail so that the forecast becomes reliable overall. If the degree of the polynomial is too small, we speak of *underfitting*, if it is too high, of *overfitting*.

Tasks:

1. Discuss different ways to determine a "good" degree of the interpolation polynomial (i.e. its highest power).
2. Formulate your results so precisely that they can be realized as scripts.
3. Test the scripts on different data sets.

5.2 New York Citibike Tripdata [NYcitibike]

Even in New York, cycling has become "hip" and borrowing data can be loaded as CSV files. We do so and load the almost 600,000 data records from June 2013 into a table. We split the column headings to get a pure data table. Of course, this is done in a *DataSprite*. Since we also want to create graphics, we load the *PlotSprite* right away.

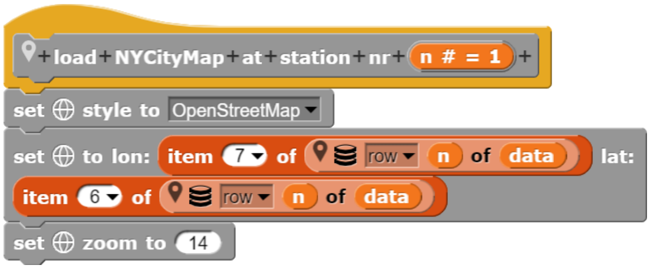


```
+get+data+
set data to read file with filepicker
set column headers to row 1 of data
delete 1 of data
```

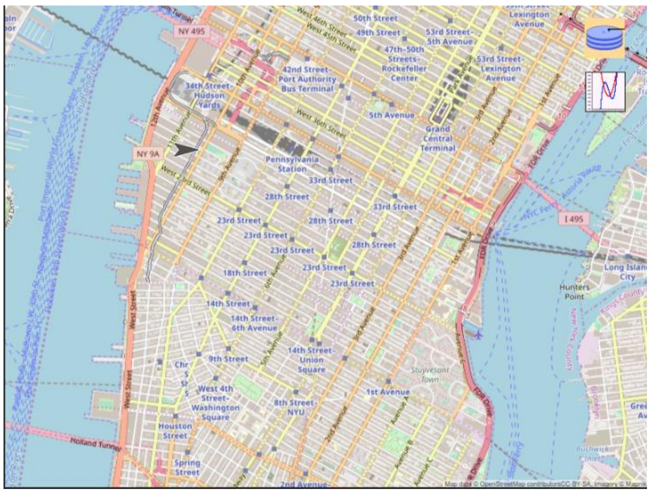
What did we actually find there?

The data legend provides the interpretation for the data: *Trip Duration (seconds), Start Time and Date, Stop Time and Date, Start Station Name, End Station Name, Station ID, Station Lat/Long, Bike ID, User Type (Customer = 24-hour pass or 3-day pass user; Subscriber = Annual Member), Gender (Zero=unknown; 1=male; 2=female), Year of Birth*

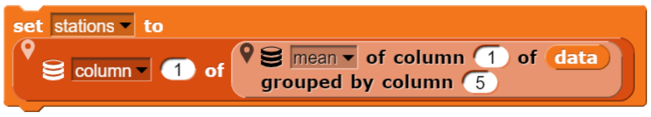
Since the geographical longitude and latitude of the rental stations are given, it is a good idea to use the *Word Map Library* from *Snap!*. We write a small block, which shows the surroundings of a rental station as a map.



```
+load+NYCityMap+at+station+nr+ n # = 1 +
set style to OpenStreetMap
set to lon: item 7 of row n of data lat:
    item 6 of row n of data
set zoom to 14
```



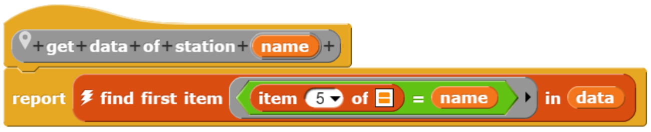
Let's see where you can rent bicycles. For the overview we extract the rental stations from the complete list, e.g. by grouping them according to the name of the starting station (column 5) and selecting only this column as the result.



```
set stations to
    column 1 of mean of column 1 of data
                grouped by column 5
```

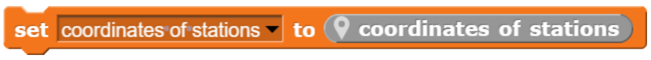
We get 337 stations after all.

Now we collect the data of a station …

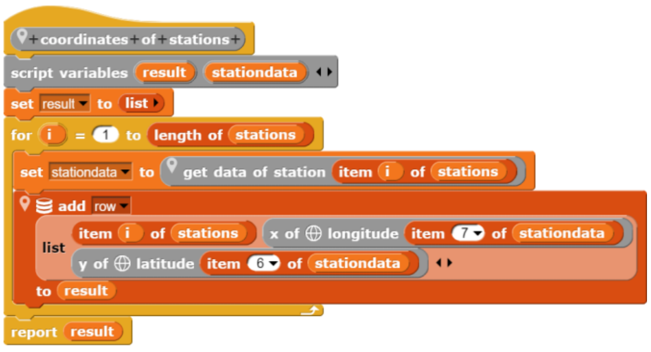


```
+get+data+of+station+ name +
report find first item ( item 5 of ▭ = name ) in data
```

… and build the coordinate list of the stations.

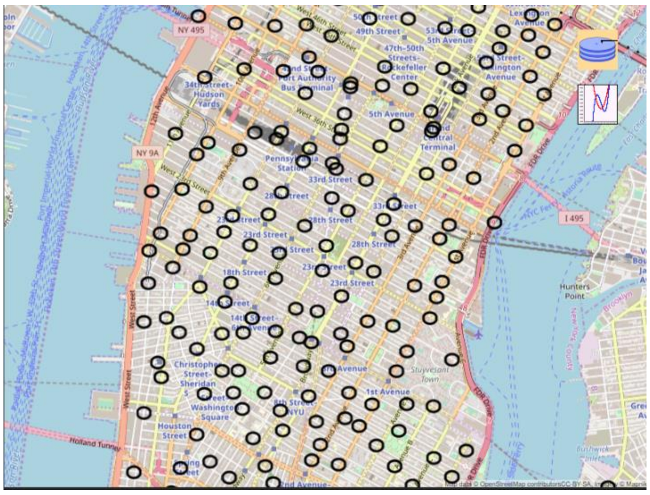


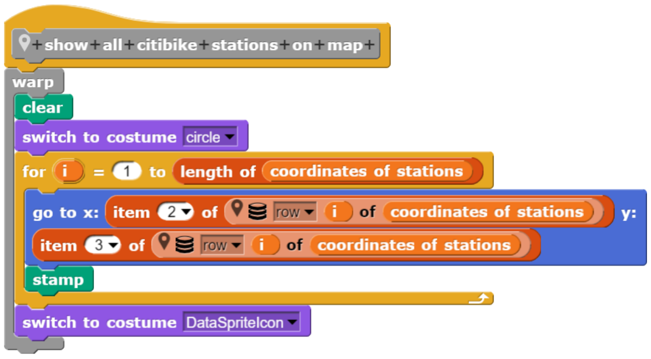
```
set coordinates of stations to coordinates of stations
```



```
+coordinates+of+stations+
script variables result stationdata
set result to list
for i = 1 to length of stations
    set stationdata to get data of station item i of stations
    add row
        list item i of stations
             x of longitude item 7 of stationdata
             y of latitude item 6 of stationdata
        to result
report result
```

With these data we can send the sprite to the individual positions, where we leave circles with the *stamp*-block.

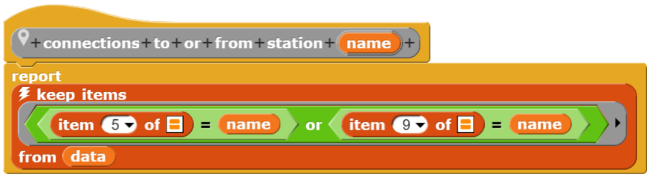




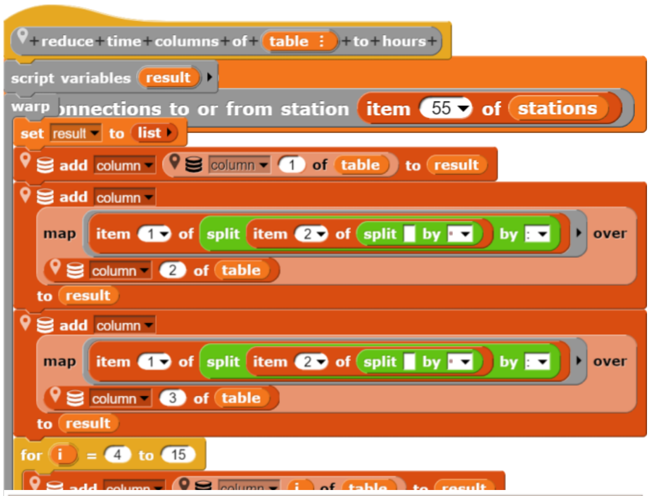
```
show all citibike stations on map
warp
  clear
  switch to costume circle
  for i = 1 to length of coordinates of stations
    go to x: item 2 of row i of coordinates of stations y: item 3 of row i of coordinates of stations
    stamp
  switch to costume DataSpriteIcon
```

At least in Midtown Manhattan, we don't have to worry about finding a rental station!

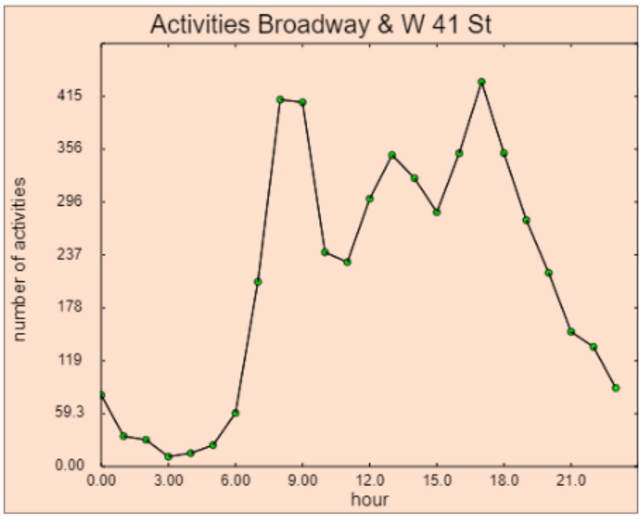
Now we want to have a closer look at the rental station Broadway - corner 41 Street (No. 55). To do this, we look for all records from the list that start or end at this station. That's 5005 events that day. Times are entered in this list together with the (same) date. We can throw this out (*split* with " ") and reduce it to the hour (*split* with ":"). We then have a numerical scale with the unit "hour". Now we can see what's going on at the station during the individual hours of the day. And we can graphically display this as usual using the *PlotSprite*.

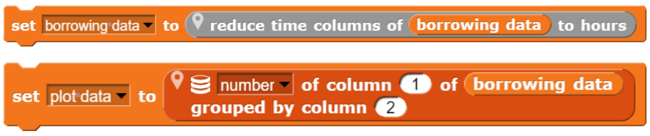


```
connections to or from station name
report keep items (item 5 of ( ) = name) or (item 9 of ( ) = name) from data
```

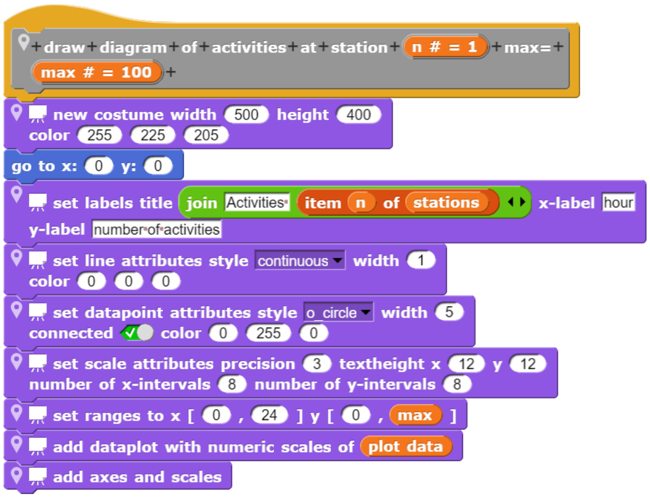


```
reduce time columns of table : to hours
script variables result
warp connections to or from station item 55 of stations
  set result to list
  add column column 1 of table to result
  add column
    map item 1 of split item 2 of split ( ) by [ ] by [ ] over
      column 2 of table
    to result
  add column
    map item 1 of split item 2 of split ( ) by [ ] by [ ] over
      column 3 of table
    to result
  for i = 4 to 15
    add column column i of table to result
```

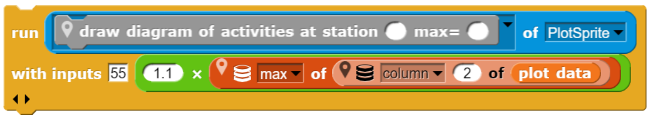




```
set borrowing data to reduce time columns of borrowing data to hours
set plot data to number of column 1 of borrowing data grouped by column 2
```



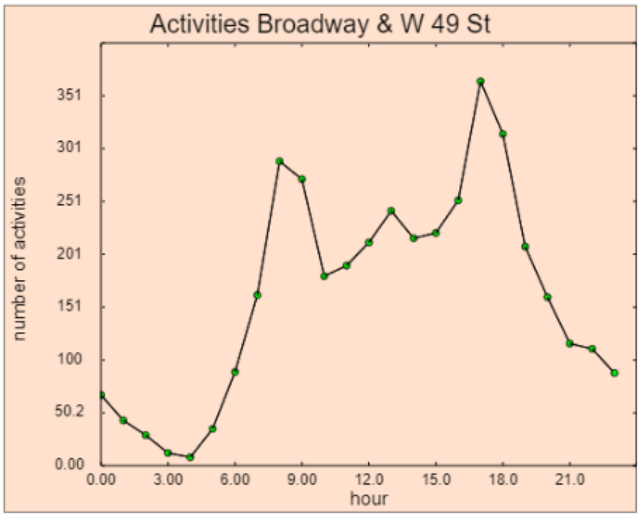
```
draw diagram of activities at station n # = 1 max= max # = 100
new costume width 500 height 400 color 255 225 205
go to x: 0 y: 0
set labels title join Activities item n of stations x-label hour y-label number of activities
set line attributes style continuous width 1 color 0 0 0
set datapoint attributes style o circle width 5 connected ✓ color 0 255 0
set scale attributes precision 3 textheight x 12 y 12 number of x-intervals 8 number of y-intervals 8
set ranges to x [ 0 , 24 ] y [ 0 , max ]
add dataplot with numeric scales of plot data
add axes and scales
```



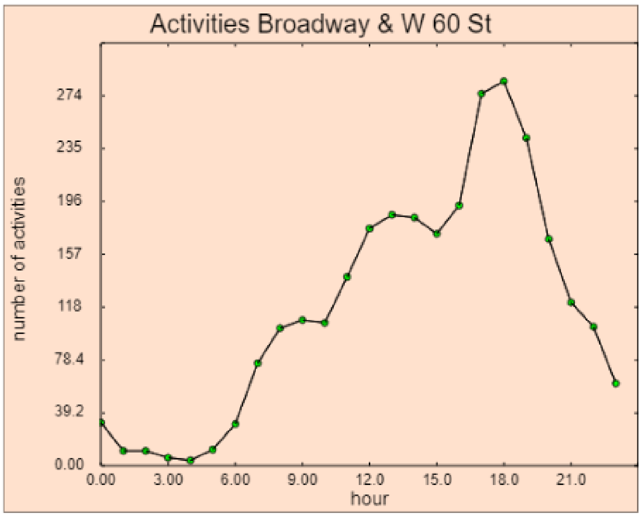
```
run (draw diagram of activities at station ( ) max= ( )) of PlotSprite
with inputs 55 (1.1 × max of column 2 of plot data)
```

A few streets down the road, it looks similar.

Is that a general pattern?



Well, at Central Park the people get up later and the tourists are not there yet. But the museums always close at the same time.



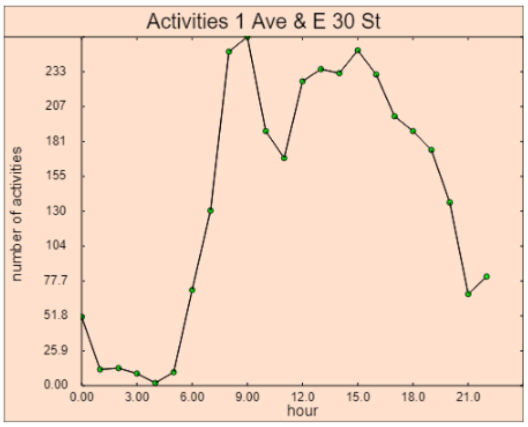
What can our programs learn from these data?

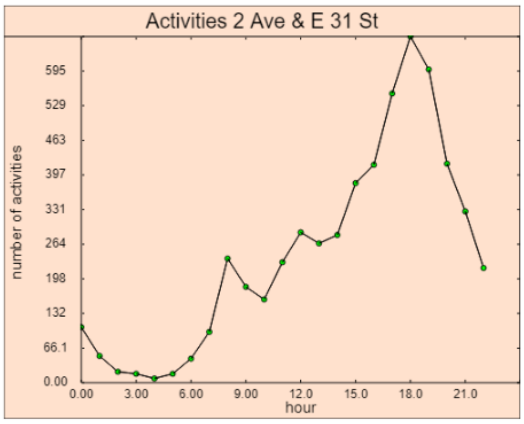
- *We could, for example, predict from the usual departures and arrivals as well as from the actual stock whether enough bicycles will be returned in time at a station or whether it would be better to transport some of them there.*
- *We could determine which rechargeable batteries are needed for eBikes from the average path lengths.*
- *We could determine whether women or men would rather borrow the bikes at a certain time of day and then make sure that the offer is right. We could do the appropriate thing for the age of the borrower.*
- *We could determine the borrowing data per bike and predict when repairs will be due. We could also do this, for example, depending on the location of the stands.*
- *We could try to generalize distributions from some stations in such a way that forecasts for others can be derived from them. So, when the museums close at Central Park, the program can "learn" from the old data in which districts the bikes will presumably be delivered and warn if there are not enough free slots available.*

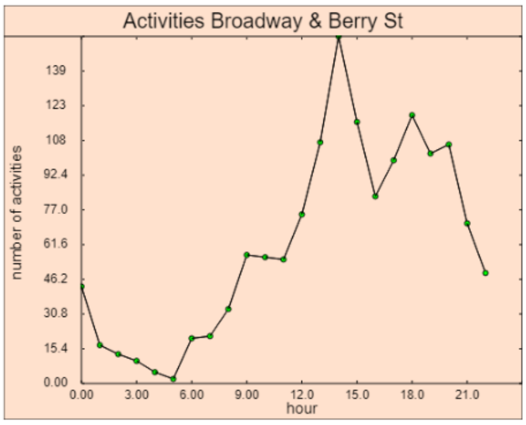
etc.

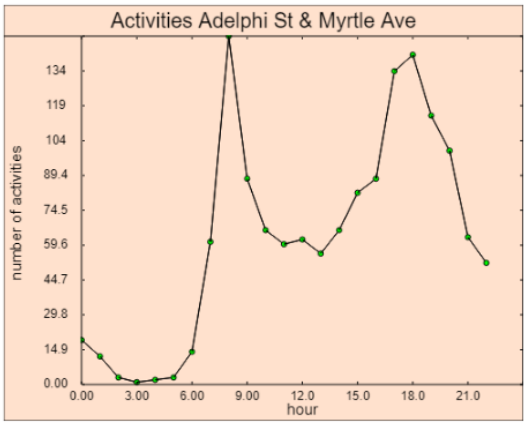
Tasks:

1. Break down the activities of the stations according to arrivals and departures.
2. Write a forecast function that warns if there is a risk of a lack of bikes at a station in the next few hours.
3. For certain stations, display the connections to the most selected delivery stations graphically on the map using direct lines. Select the thickness of the lines according to the number of borrowing operations and the colors depending on the station. Are clusters formed?
4. Find out with the help of correlations-block whether there are correlations in rental behavior (e.g. with regard to times of day, location, ...) with gender, age, status of borrowers. You may have to replace the data with numeric data beforehand - similar to the times. Discuss possible consequences.
5. For a small section of Midtown (where everything is beautifully right-angled), find the coordinates of the street corners. Then develop a router that shows the shortest route to the nearest Citibike station.
6. The rental numbers depending on the time of day show quite a difference in different areas of Manhattan. Systematically examine similarities and differences and try to explain the results.



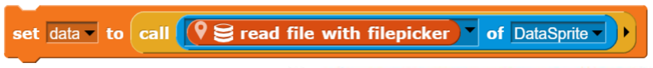




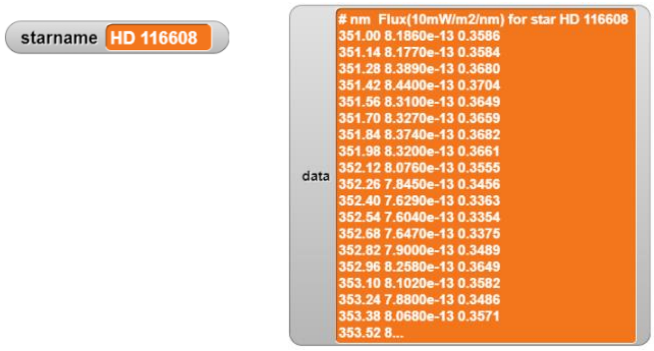


5.3 Star spectra [UniGOE]

Stars shine in different colors because they have different temperatures. In addition, the spectra differ in their absorption lines. We want to investigate this in more detail. We use a *PlotSprite* and a *DataSprite* as an assistant, e.g. for loading and preparing the data. Inside of this we start to work.



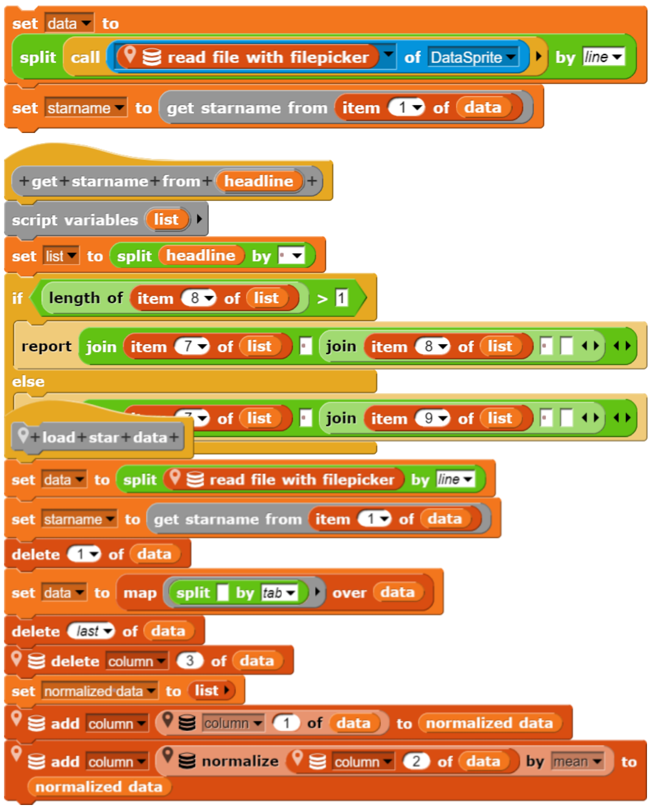
```
set data to call (read file with filepicker) of DataSprite ▸
```



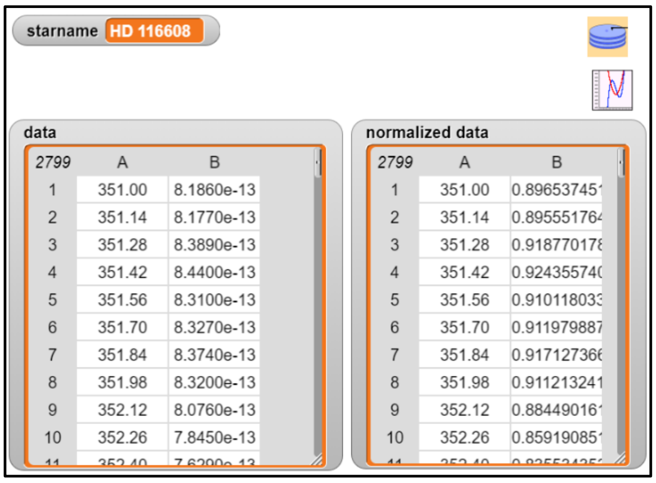
```
starname HD 116608

data
# nm  Flux(10mW/m2/nm) for star HD 116608
351.00 8.1860e-13 0.3586
351.14 8.1770e-13 0.3584
351.28 8.3890e-13 0.3680
351.42 8.4400e-13 0.3704
351.56 8.3100e-13 0.3649
351.70 8.3270e-13 0.3659
351.84 8.3740e-13 0.3682
351.98 8.3200e-13 0.3661
352.12 8.0760e-13 0.3555
352.26 7.8450e-13 0.3456
352.40 7.6290e-13 0.3363
352.54 7.6040e-13 0.3354
352.68 7.6470e-13 0.3375
352.82 7.9000e-13 0.3489
352.96 8.2580e-13 0.3649
353.10 8.1020e-13 0.3582
353.24 7.8800e-13 0.3486
353.38 8.0680e-13 0.3571
353.52 8...
```

We get some star spectra (source: [UniGOE]) and save them as a text file. We read in such a file. In the first line we see the star name after the column captions. We isolate it and store it in the variable ***starname***.

We know the star's name now. If you search the Internet for it, you will find a wealth of information about it. For repeating the loading process with other data, we encapsulate it in a separate block. After its execution, the actual star data are available as a table. What's unpleasant about this is the very different order of magnitude of the data in the two columns. We therefore normalize the second column using the mean value and save the result as ***normalized data***.



```
set data to
  split call (read file with filepicker) of DataSprite ▸ by line
set starname to get starname from item 1 of data

+get+starname+from+ headline +
script variables list ▸
set list to split headline by
if length of item 8 of list > 1
  report join item 7 of list   join item 8 of list   ◂▸ ◂▸
else
  ... 7 of list   join item 9 of list   ◂▸ ◂▸

+load+star+data+
set data to split (read file with filepicker) by line
set starname to get starname from item 1 of data
delete 1 of data
set data to map (split  by tab) ▸ over data
delete last of data
delete column 3 of data
set normalized data to list ▸
add column (column 1 of data) to normalized data
add column (normalize (column 2 of data) by mean) to
  normalized data
```



starname HD 116608

data

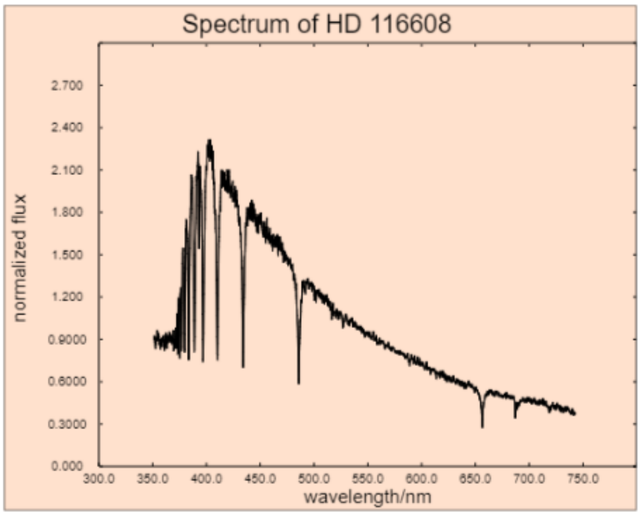
2799	A	B
1	351.00	8.1860e-13
2	351.14	8.1770e-13
3	351.28	8.3890e-13
4	351.42	8.4400e-13
5	351.56	8.3100e-13
6	351.70	8.3270e-13
7	351.84	8.3740e-13
8	351.98	8.3200e-13
9	352.12	8.0760e-13
10	352.26	7.8450e-13
11	352.40	7.6290e-13

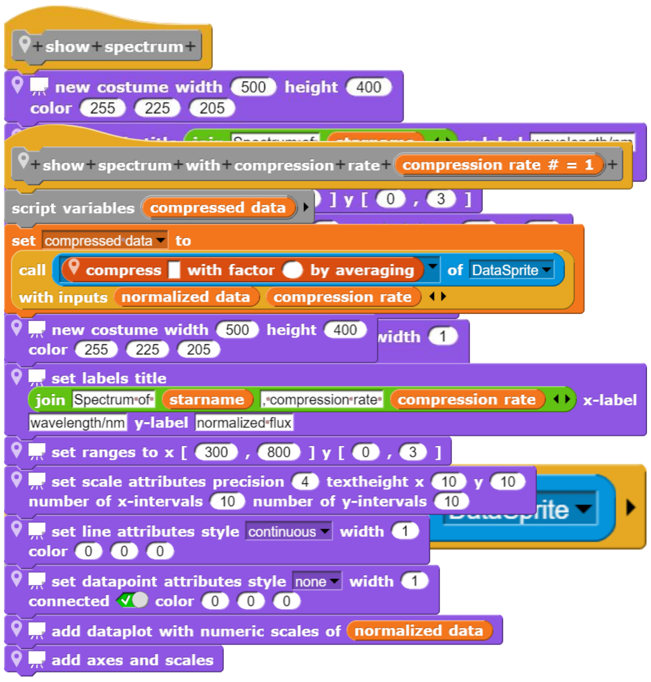
normalized data

2799	A	B
1	351.00	0.89653745
2	351.14	0.89555176
3	351.28	0.91877017
4	351.42	0.92435574
5	351.56	0.91011803
6	351.70	0.91197988
7	351.84	0.91712736
8	351.98	0.91121324
9	352.12	0.88449016
10	352.26	0.85919085
11	352.40	0.83552425

With this the *DataSprite* has done its duty for now. We change to the *PlotSprite*.

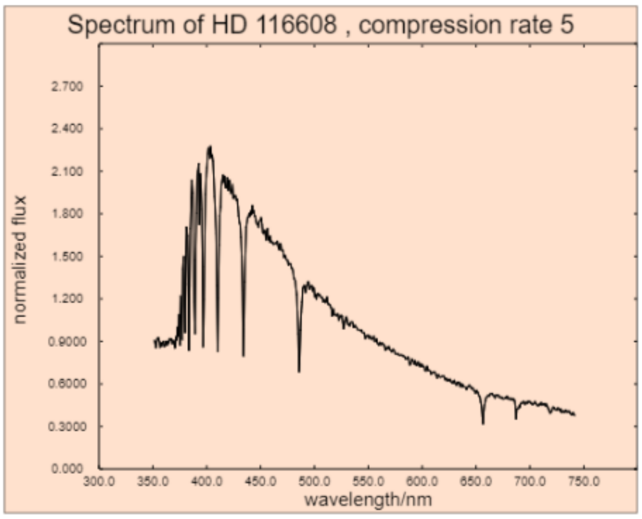
With the normalized data you can quickly create a chart in *PlotSprite*.





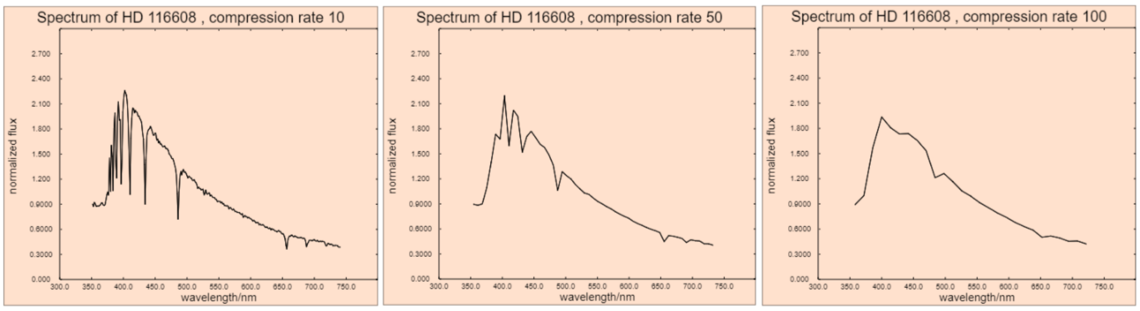
```
+show+spectrum+
new costume width (500) height (400)
color (255) (225) (205)

+show+spectrum+with+compression+rate+ (compression rate # = 1) +
script variables (compressed data)
set [compressed data] to
    call (compress [] with factor () by averaging) of [DataSprite]
    with inputs (normalized data) (compression rate)
new costume width (500) height (400)
color (255) (225) (205)
set labels title
    (join [Spectrum of ] (starname) [, compression rate ] (compression rate)) x-label
    [wavelength/nm] y-label [normalized flux]
set ranges to x [ (300) , (800) ] y [ (0) , (3) ]
set scale attributes precision (4) textheight x (10) y (10)
number of x-intervals (10) number of y-intervals (10)
set line attributes style [continuous] width (1)
color (0) (0) (0)
set datapoint attributes style [none] width (1)
connected ✓ color (0) (0) (0)
add dataplot with numeric scales of (normalized data)
add axes and scales
```

The sloping course with some prominent absorption lines is clearly visible. But is it necessary to have all spectral data for this realization? Maybe it is enough to reduce the amount of data by averaging. We introduce a compression factor ***compression rate*** and complete the script before the diagram is created.

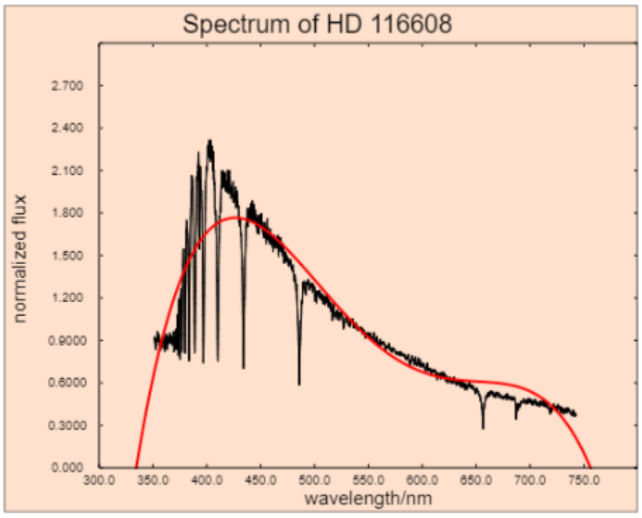


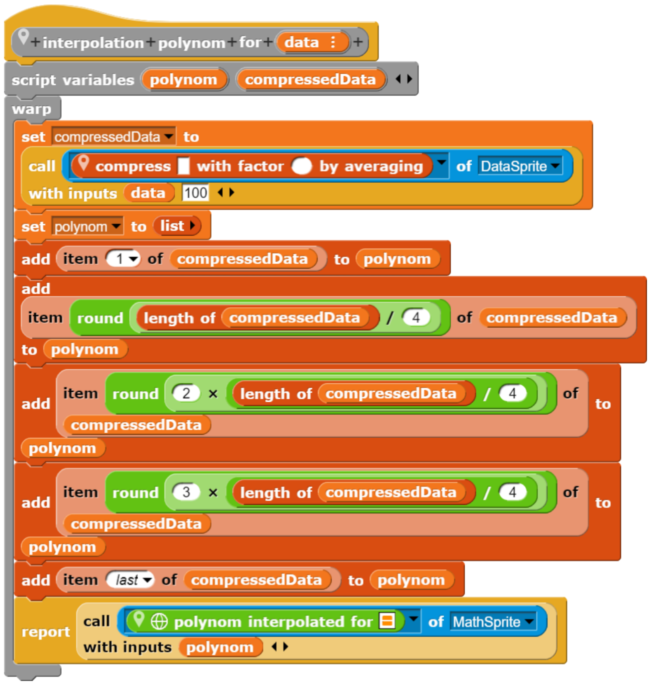
The factor 5 does not change much. So, let's keep trying.

One can see that the temperature-dependent course of the spectrum is hardly changed. Only the absorption lines are lost.



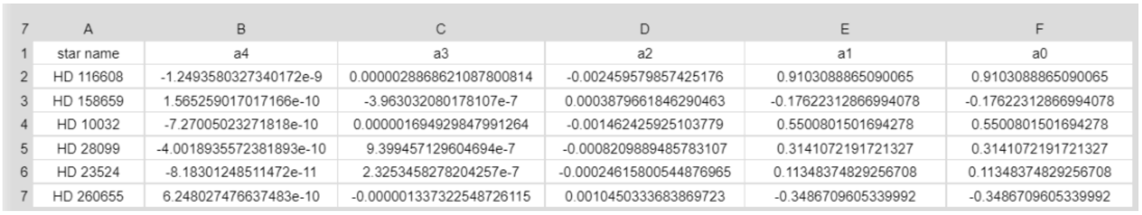
Thus, the type of spectrum should be describable by an interpolation polynomial, e.g. 4th degree. We load the *MathSprite* additionally and try it like this:



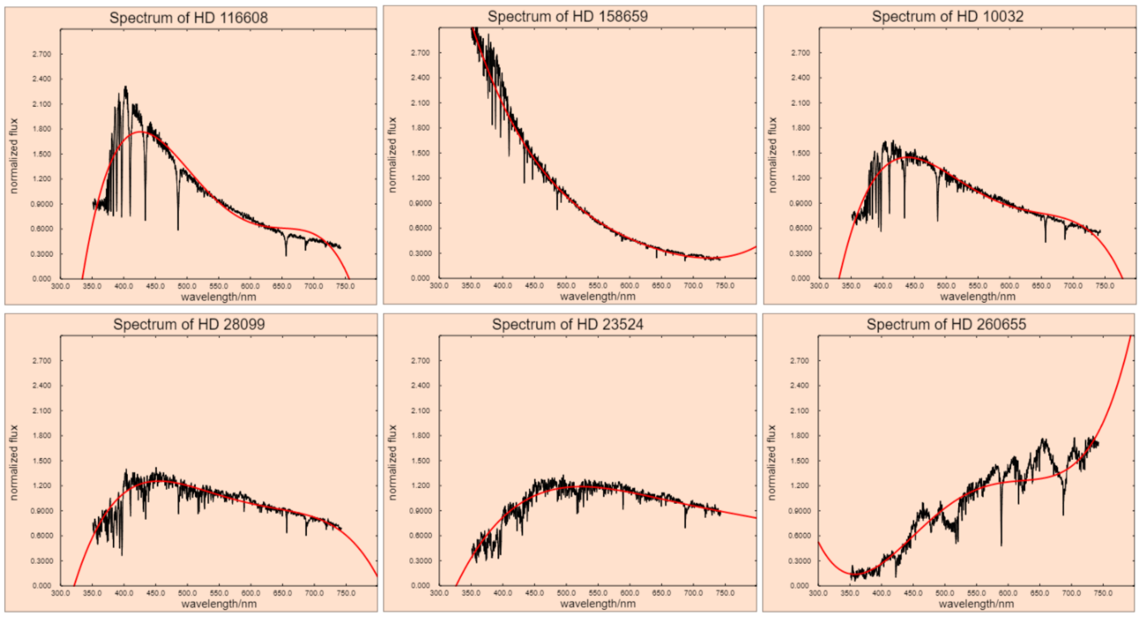


```
+interpolation+polynom+for+ data +
script variables polynom compressedData
warp
    set compressedData to
        call (compress [] with factor () by averaging) of DataSprite
        with inputs data 100
    set polynom to list
    add item 1 of compressedData to polynom
    add
        item round (length of compressedData / 4) of compressedData
    to polynom
    add item round (2 × length of compressedData / 4) of compressedData to polynom
    add item round (3 × length of compressedData / 4) of compressedData to polynom
    add item last of compressedData to polynom
    report
        call (polynom interpolated for []) of MathSprite
        with inputs polynom
```

So, this works great! If we log the polynomial parameters during the test, we can easily distinguish the star types by means of the parameter ranges.



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	0.000001694929847991264	-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



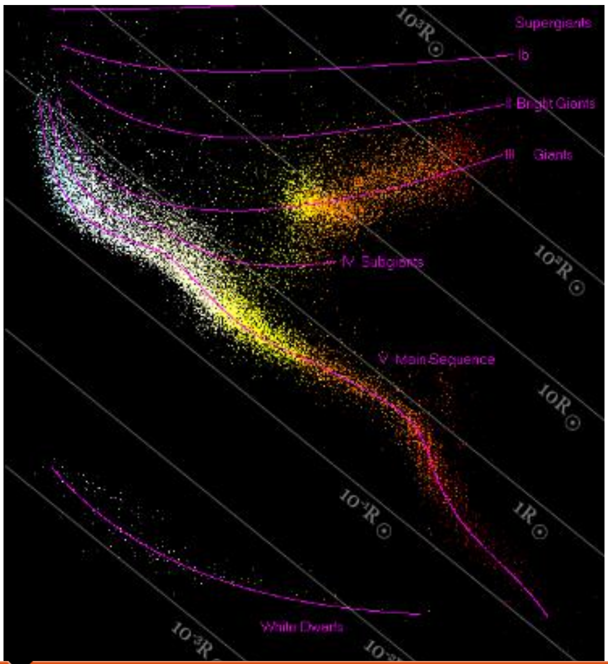
If you feed the polynomial coefficients to a neural network, it quickly learns to roughly assign a diagram to a star type. The program can therefore "learn" which parameter intervals belong to which star classes based on the old data. If you enter the data of a new star, it determines the coefficients of the polynomial and then makes a well-founded prognosis about what kind of star it could be.

<u>Tasks</u>:

1. Set up an interpolation polynomial of the lowest possible degree for the uncompressed spectrum data. Which points should be selected for this? Are there any differences between these polynomials and the results of the method shown above?
2. Develop a script that assigns an unknown spectrum to one of the previously occurring types.
3. Develop a method to examine the most prominent absorption lines more closely. Enlarge them for stars of the same class and try to determine differences "automatically". Discuss your ideas before realization.

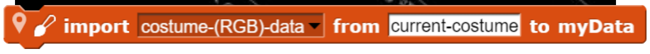
5.4 Classification of stars according to the kNN method

In the Hertzsprung-Russel diagram (see Wikipedia) the luminosity of stars is plotted above their star class. The result is a kind of line from top left to bottom right, the "main sequence". On this line stars like the sun are mostly located. Right above the main row we find the red giants, left below the main row the white dwarfs. That's enough for first. (Picture source: [HR])



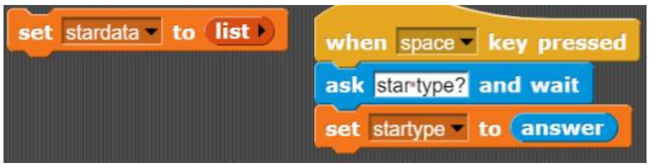
We want to classify new stars in this diagram using the k-next neighbor (kNN) method: As training data we generate a list of stars with their coordinates (simply as image coordinates in the diagram) and their type. If we want to classify a new star, we determine its position in the diagram and look for the nearest k (e.g. k=5) neighbors. Then we determine the most frequently appearing star type in this list. We assign it to the new star.

First of all, we need a picture of the Hertzsprung-Russel diagram ([HR]). We import it into Snap! as costume of an *ImageSprite* and generate the required data from it.

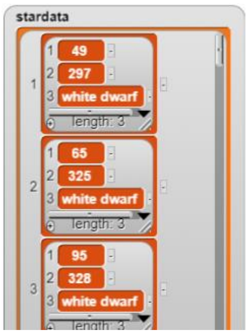


```
import costume-(RGB)-data from current-costume to myData
```

We generate the training data by specifying a star type and then clicking on some points in the diagram that correspond to this type.

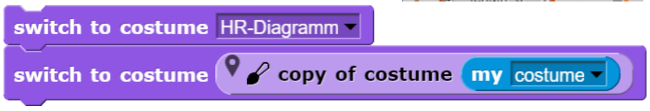


```
set stardata to list

when space key pressed
ask star type? and wait
set startype to answer
```



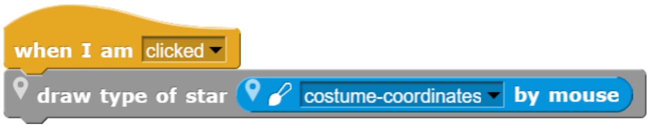
```
stardata
1  1 49
   2 297
   3 white dwarf
   length: 3
2  1 65
   2 325
   3 white dwarf
   length: 3
3  1 95
   2 328
   3 white dwarf
   length: 3
```

Since we want to draw on the image, we work with a copy of the HR diagram so as not to alter the original.



```
switch to costume HR-Diagramm
switch to costume copy of costume my costume
```

Then we can classify new stars by clicking on them (here) and labeling them.

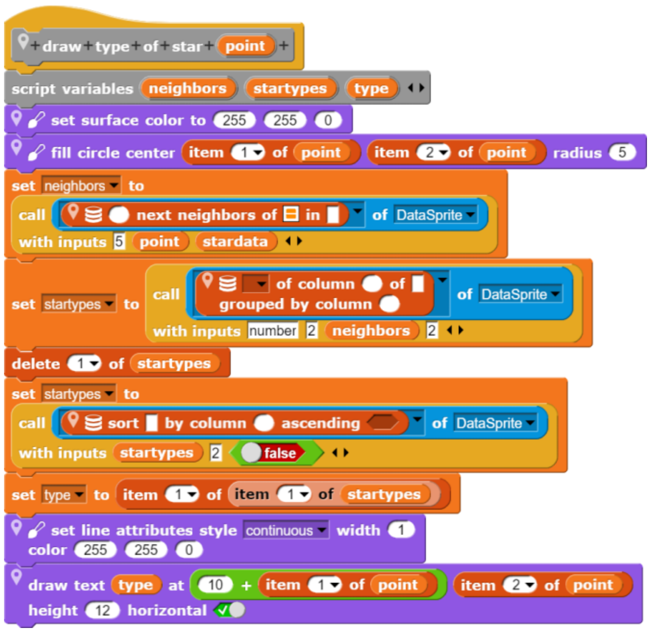


```
when I am clicked
draw type of star costume-coordinates by mouse
```

We set some properties for the representation ...

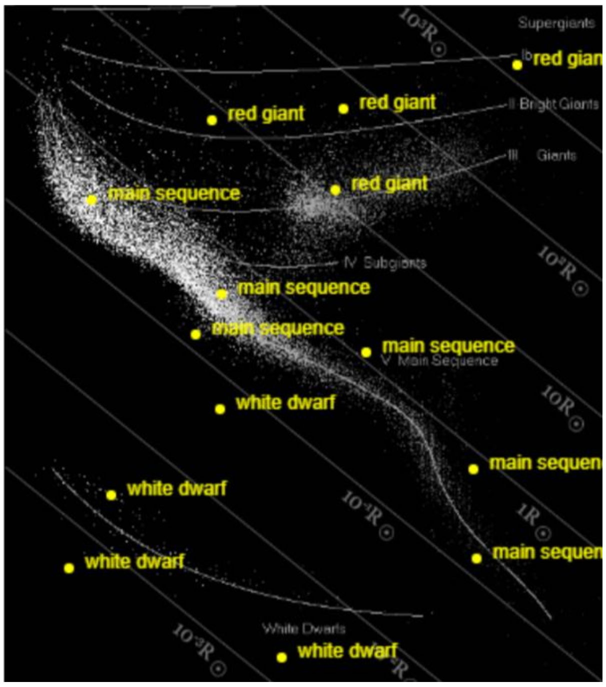
...and draw a circle at the location of the star.

Then we determine the five nearest neighbors and the number of occurrences of their type. In the result we delete the headings and sort the list in descending order. The type of the new star is then the first element in the first line. We write this next to the star.



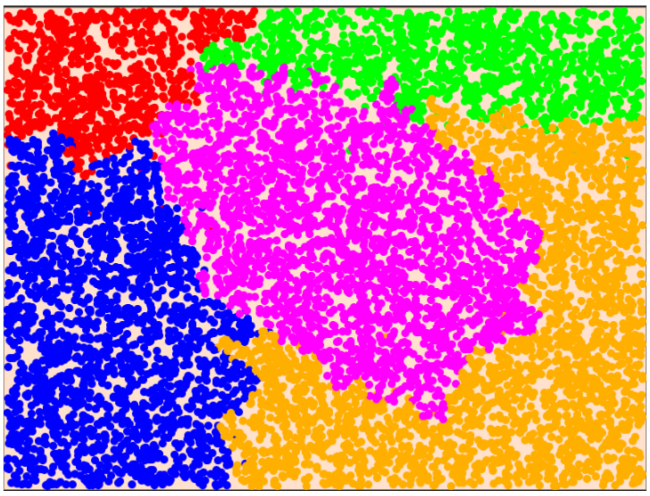
```
+draw+type+of+star+ (point) +
script variables (neighbors) (startypes) (type)
set surface color to 255 255 0
fill circle center item 1 of point   item 2 of point   radius 5
set neighbors to
  call ( next neighbors of [] in [] ) of DataSprite
  with inputs 5 point stardata
set startypes to
  call ( [] of column () of []
         grouped by column () ) of DataSprite
  with inputs number 2 neighbors 2
delete 1 of startypes
set startypes to
  call ( sort [] by column () ascending <> ) of DataSprite
  with inputs startypes 2 false
set type to item 1 of item 1 of startypes
set line attributes style continuous width 1
  color 255 255 0
draw text type at (10 + item 1 of point) item 2 of point
  height 12 horizontal ✓
```

The result:



Tasks:

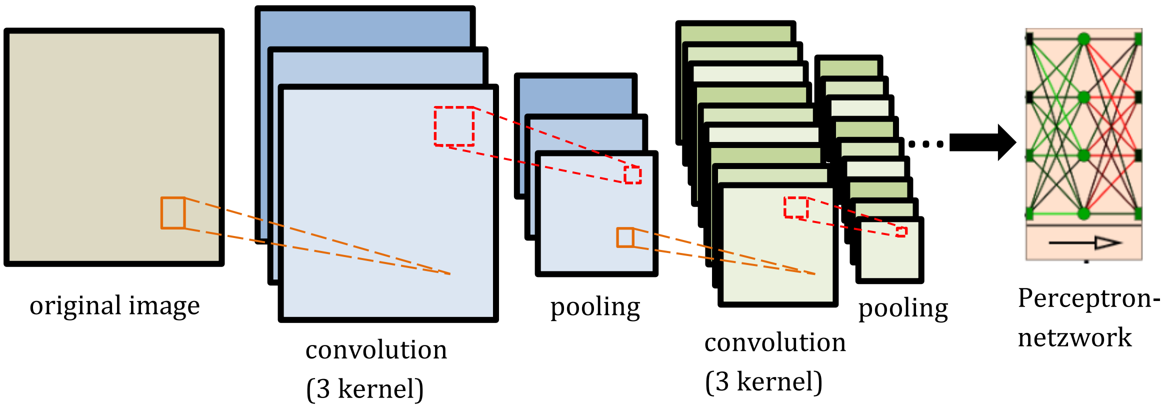
1. Add the newly classified stars to the sample list so that they are included in further classifications.
2. Draw differently colored dots at the correct places on the sprite for the different types of stars instead of labeling them.
3. Run the process for randomly selected points. Is the pattern always the same? Do completely different or similar patterns emerge? What does it depend on?



5.5 Character recognition with a convolutional network

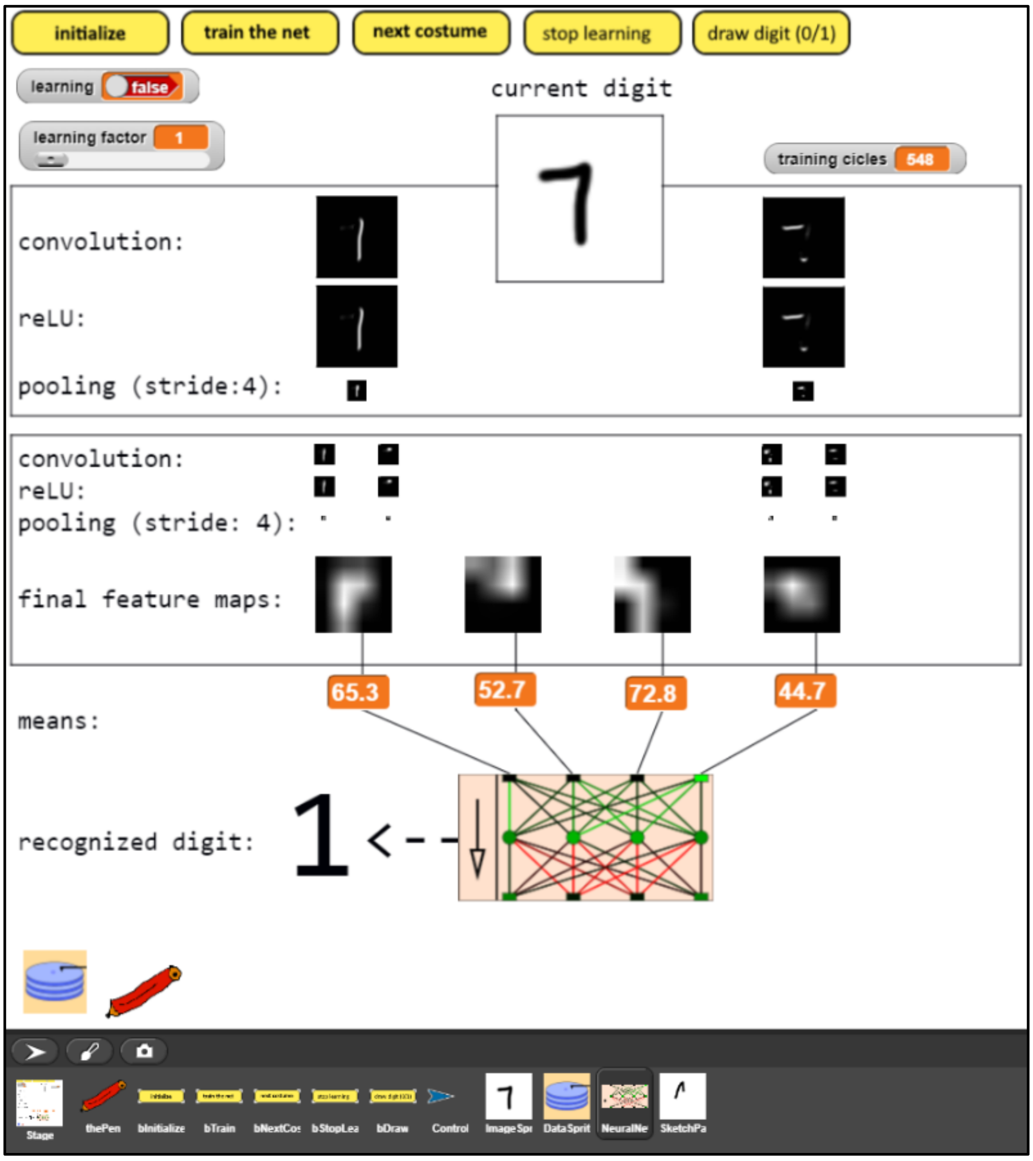
The immense number of parameters in fully connected perceptron networks and the resulting need for huge amounts of training data has led to other network variants to drastically reduce this number. One of these is the *Convolutional Neural Network* (CNNs), where the amount of input data for the perceptron network is reduced. This type of network is used very successfully in image and speech recognition, for example.

CNNs reduce the amount of data by first applying several *kernels* in a multi-stage process, which filter out certain properties of e.g. an image (edges, oval surfaces, ...) and thus lead to several *feature maps*, which usually have the same size as the original. This first increases the amount of data. Afterwards, a non-linear activation function (*reLU*) is usually applied to the feature maps, followed by a *pooling operation* that reduces the amount of data again. This is usually called *Max-Pooling*, where the maximum value is determined from a section of the data. If you do this with a "window" that is moved across the feature map with a certain *stride*, each pooling step creates a value of the next, reduced feature map.



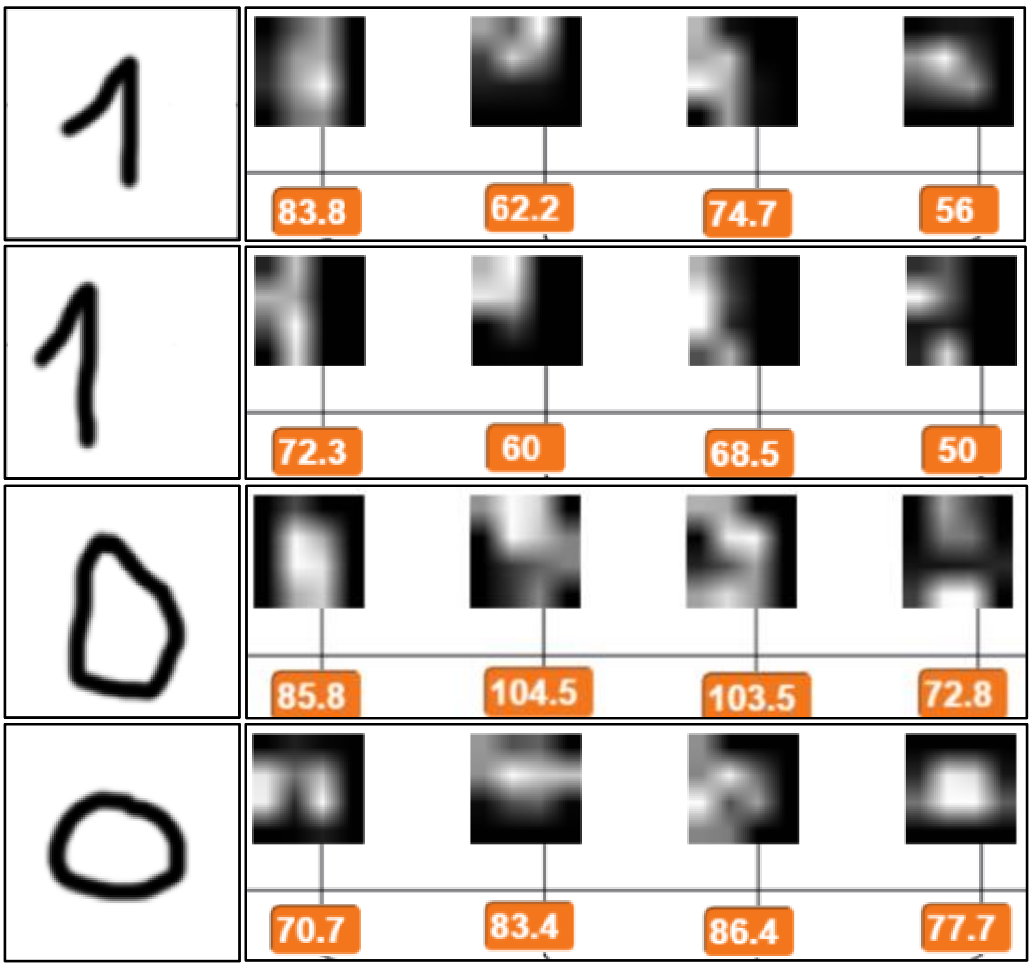
As an example, let's take a kernel that filters vertical lines: it colors a point white if there is a second pixel next to the point, otherwise black. In the "folded" image we can see vertical lines of the original as bright spots. If it is not so important where exactly these lines are, we do not lose too much information during the pooling. A white point in a feature map after various pooling processes means then: "*In this area there was a vertical line somewhere*". Using such data from several feature maps, it can be deduced, for example, that there was also a horizontal line, i.e. a corner. If we had searched for "pink" areas and "oval" shapes, the chance of identifying faces would not be so bad.

We now want to build a model for such a CNN that can distinguish the handwritten digits *zero* and *one*. For this we use a *DataSprite* for auxiliary operations, an *ImageSprite* for the actual image and - of course - a *NeuralNetSprite* for the perceptron network at the end of the chain. Another, "normal" sprite called *Control* is supposed to control the operations. To make the model easier to use, we add some buttons and a pen to make the interface clearer. In the screenshot, the image to be analyzed is located at the top of the box, the neural network shows its result at the bottom. In between, the different intermediate layers are run through and displayed from top to bottom. As an addition, the model contains the possibility to draw your own numbers.



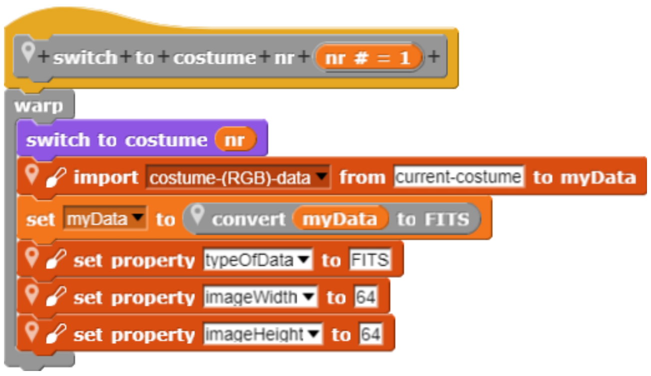
Our CNN is trained with 10 digits of 64x64 pixels each for the zeros and ones. Afterwards it should "recognize" these and other handwritten ones. Actually, we would have to train several kernels of our CNN especially for this task. Instead, we take only two known kernels for the recognition of vertical and horizontal lines, because by limiting the number of kernels to two, everything can be displayed on the screen and the results can even be interpreted halfway. (The recognition rate suffers severely from this!) So, we train only the perceptron network with four input values.

In the image above, after two stages of reduction, four feature maps of 16x16 pixels each are left over, each of which has undergone the *Convolution → reLU → Max-Pooling* operations twice: on the far left with the kernel for vertical lines, then with both kernels in different order, and finally twice with the kernel for horizontal lines. The numbers below indicate the average brightness value measured over the entire image. If we apply this to different digits, the possibility to measure differences between zeros and ones becomes apparent despite the very simple procedure.



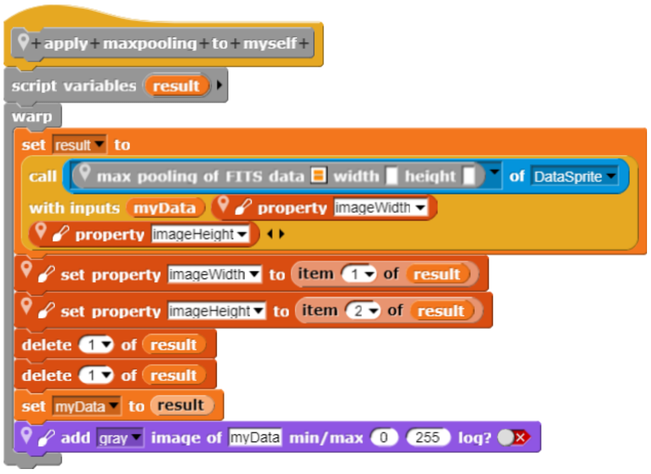
Let us look at the functionalities of the individual objects:

The *ImageSprite* should import the data of a new costume as gray values into its data area. This is very simple.

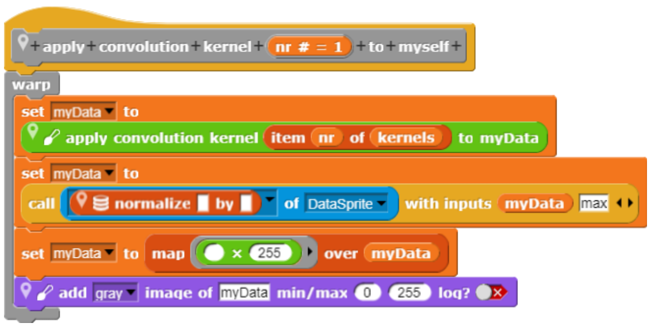


```
+switch+to+costume+nr+ (nr # = 1) +
warp
  switch to costume (nr)
  import costume-(RGB)-data from current-costume to myData
  set myData to convert (myData) to FITS
  set property typeOfData to FITS
  set property imageWidth to 64
  set property imageHeight to 64
```

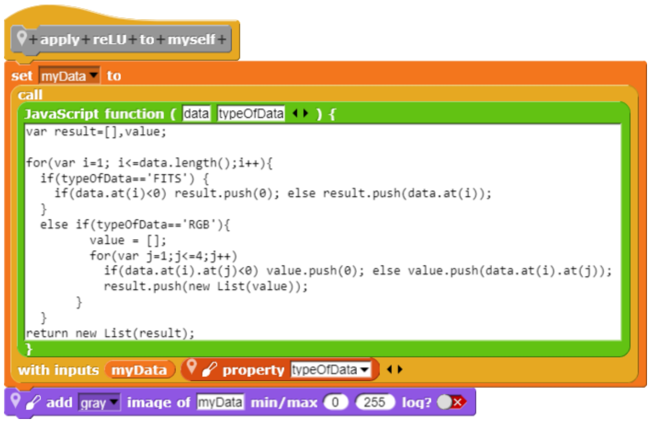
Furthermore, the sprite and its clones should be able to perform the three operations of a CNN. With some help of the *DataSprite* this is also possible.



```
+apply+maxpooling+to+myself+
script variables (result)
warp
  set result to
    call (max pooling of FITS data width height) of DataSprite
    with inputs (myData) (property imageWidth)
    (property imageHeight)
  set property imageWidth to (item 1 of result)
  set property imageHeight to (item 2 of result)
  delete 1 of result
  delete 1 of result
  set myData to result
  add gray image of myData min/max 0 255 log?
```

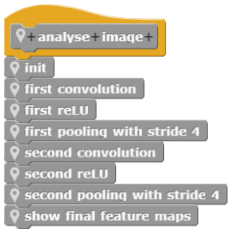


```
+apply+convolution+kernel+(nr # = 1)+to+myself+
warp
  set myData to
    apply convolution kernel (item nr of kernels) to myData
  set myData to
    call (normalize by) of DataSprite with inputs myData max
  set myData to map (( ) x 255) over myData
  add gray image of myData min/max 0 255 log?
```



```
+apply+reLU+to+myself+
set myData to
call
JavaScript function ( data typeOfData ) {
var result=[],value;

for(var i=1; i<=data.length();i++){
  if(typeOfData=='FITS') {
    if(data.at(i)<0) result.push(0); else result.push(data.at(i));
  }
  else if(typeOfData=='RGB'){
          value = [];
          for(var j=1;j<=4;j++)
            if(data.at(i).at(j)<0) value.push(0); else value.push(data.at(i).at(j));
            result.push(new List(value));
        }
  }
return new List(result);
}
with inputs myData (property typeOfData)
add gray image of myData min/max 0 255 log?
```

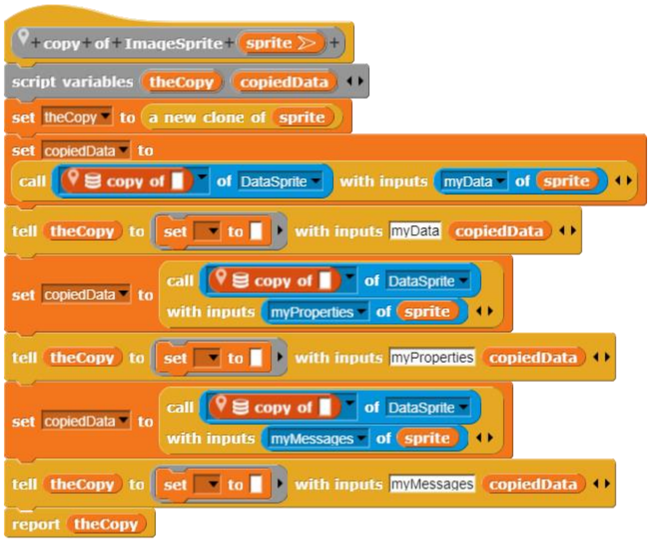
The *ImageSprite* does not have to master more for our purposes.



```
+analyse+image+
init
first convolution
first reLU
first pooling with stride 4
second convolution
second reLU
second pooling with stride 4
show final feature maps
```

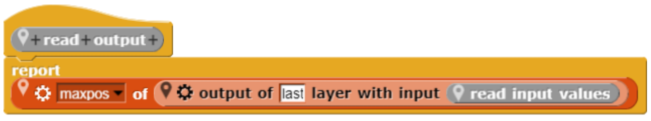
The *Control* sprite has to ask the *ImageSprite* to change the costume and analyze it afterwards. In doing so, it strictly adheres to the specifications for CNNs. The ***init*** method only takes care of drawing the lines on stage. The other methods work with two layers of CNN, ***first layer*** and ***second layer***, each containing the versions of the characters that appear on stage. So that they do not interfere with each other, copies of the *ImageSprite* are used, not clones. The *DataSprite* again helps with copying.

After the required copies have been made, *Control* asks them to perform the relevant CNN operation. Finally, the now rather small (4x4 pixel) clones of the last layer are displayed as "*final feature maps*", greatly enlarged. These are used to train the neural network.

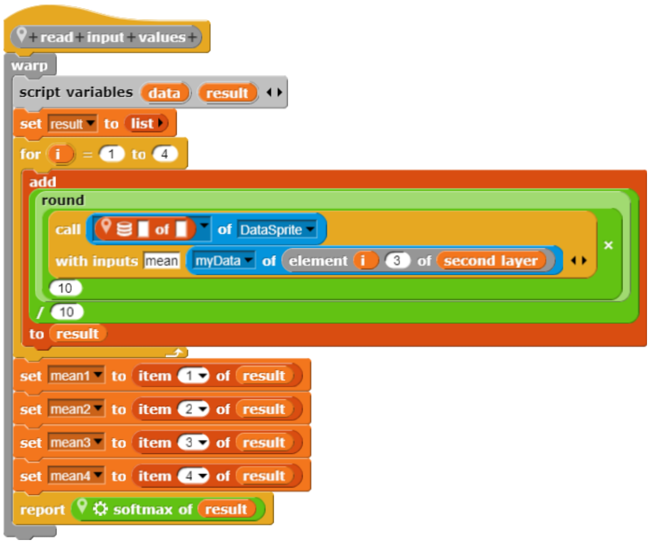


```
+copy+of+ImageSprite+(sprite)+
script variables (theCopy) (copiedData)
set theCopy to (a new clone of sprite)
set copiedData to
  call (copy of) of DataSprite with inputs (myData of sprite)
tell theCopy to (set to) with inputs myData copiedData
set copiedData to
  call (copy of) of DataSprite
  with inputs (myProperties of sprite)
tell theCopy to (set to) with inputs myProperties copiedData
set copiedData to
  call (copy of) of DataSprite
  with inputs (myMessages of sprite)
tell theCopy to (set to) with inputs myMessages copiedData
report theCopy
```

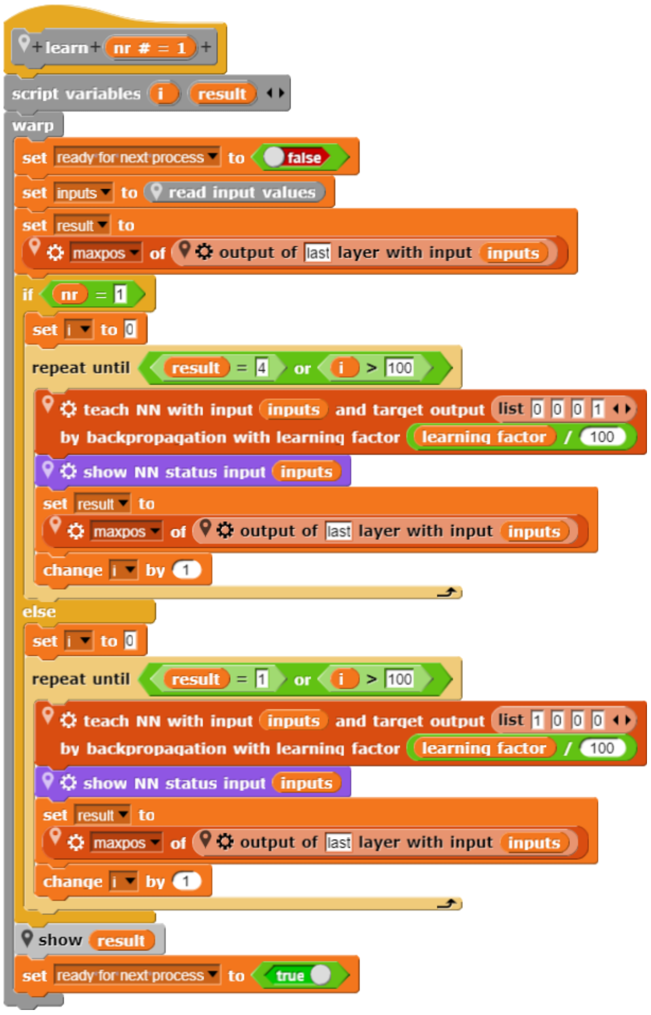
The neural network in the form of a *NeuralNetSprite* should produce the largest output at output 1 when there are zeros and at output 4 when there are ones. This is of course completely arbitrary. The current output value is determined by the function ***read output***. With its components the net can be trained, if we succeed in determining the mean values from the last layer of ***second layer***. We model these aptly with the ***softmax*** function.



```
+read+output+
report ( maxpos of ( output of last layer with input ( read input values ) ) )
```

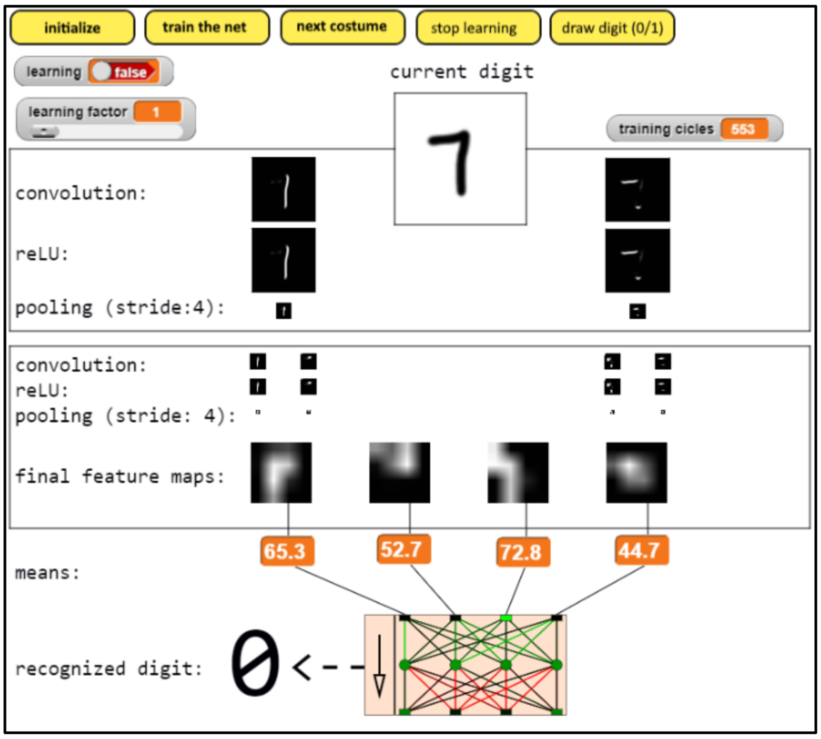


```
+read+input+values+
warp
  script variables data result
  set result to list
  for i = 1 to 4
    add round ( ( call ( [ ] of [ ] ) of DataSprite with inputs mean myData of element i 3 of second layer ) × 10 ) / 10 to result
  set mean1 to item 1 of result
  set mean2 to item 2 of result
  set mean3 to item 3 of result
  set mean4 to item 4 of result
  report ( softmax of result )
```



```
+learn+ nr # = 1 +
script variables i result
warp
  set ready for next process to false
  set inputs to read input values
  set result to ( maxpos of ( output of last layer with input inputs ) )
  if nr = 1
    set i to 0
    repeat until ( result = 4 ) or ( i > 100 )
      teach NN with input inputs and target output list 0 0 0 1
        by backpropagation with learning factor ( learning factor / 100 )
      show NN status input inputs
      set result to ( maxpos of ( output of last layer with input inputs ) )
      change i by 1
  else
    set i to 0
    repeat until ( result = 1 ) or ( i > 100 )
      teach NN with input inputs and target output list 1 0 0 0
        by backpropagation with learning factor ( learning factor / 100 )
      show NN status input inputs
      set result to ( maxpos of ( output of last layer with input inputs ) )
      change i by 1
  show result
  set ready for next process to true
```

And - has the net learned something?



Well - there is still room for improvement!

Latest hints

Machine learning consists to a large extent of preparing data - whether it is tabular data or images. The actual learning processes of the machines then consist of the parameter adjustments that result from the data. Since both can be visualized well, there is a broad field for beginners with many transitions to the area of "computer science and society".

Examples for the application of the operations of the *ML.SpriteLibrary*, especially the convolution with help of a kernel, can be found abundantly in [DBV].

List of examples

References and sources

[Albon]	Albon, Chris: Machine Learning Kochbuch, O'Reilly, 2919
[Baumann]	Baumann, Rüdeger: Didaktik der Informatik, Klett, 1990
[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
[Goodfellow]	Goodfellow, I.; Bengio, Y.; Courville, A.: Deep Learning, MIT Press, 2016
[Grus]	Grus, Joel: Einführung in Data Science, O'Reilly, 2016
[HOU]	Hands-On Universe: handsonuniverse.org/
[HR]	https://studylibde.com/doc/2985884/hertzsprung-russell--und-farb-hel-ligkeits
[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
[Minsky]	Minsky, Marvin: Computation: Finite and Infinite Machines, Prentice-Hall, Englewood Cliffs, New York, 1967
[Modrow1]	Modrow, Eckart: Technische Informatik mit Delphi, emu-online, 2004
[Modrow2]	Modrow, Eckart: Zur Didaktik des Informatikunterrichts – Band 2, Dümmler, 1992
[Rojas]	Rojas, Rau´l: Neural Networks - A Systematic Introduction, Springer Berlin, 1996
[SAP]	www.sap.com/germany/products/leonardo/machine-learning.html
[SchulAstro]	www.schul-astronomie.de
[SQL]	Modrow, Eckart: Informatik mit Snap!, http://ddi-mod.uni-goettingen.de/InformatikMitSnap.pdf
[SZ]	Süddeutsche Zeitung: 10. April 2019 www.sueddeutsche.de/wissen/schwarzes-loch-bild-1.4404130
[UniGOE]	Institut für Astrophysik, Universitaet Goettingen