Fachhochschule Dortmund

Student Engineering Project

Kai Lawrence

August 4, 2020

Contents

1 First prototype

To create a first prototype capable of detecting hand-drawn mechanism, the capabilities necessary are have to be considered first. The application has to be able to detect and to localize `nodes` and to detect `constraints`, which are connecting pairs of `nodes`. Subsequently the gathered information has to be transformed into a usable format for further processing.

At first the detection and localization of `nodes` is examined.

1.1 The Fully Convolutional Network

The topic of a previous work was the recognition of hand-drawn mechanical symbols [Law20]. Building on this, the trained model is improved to provide not only the class, but the location of the classification in an image of arbitrary size.

The respective model can be loaded using `Keras'`, `model.load_model` and by issuing the `summary` method we get listing 1.

Listing 1: Summary of Symbol Classifier.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 16)        272
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 32)        8224
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 32)          0
_________________________________________________________________
flatten (Flatten)            (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 3)                 6147
=================================================================
Total params: 14,643
Trainable params: 14,643
Non-trainable params: 0
_________________________________________________________________
```

One way to localize images in a bigger image is to scan each sector of an image and predicting the generated crops. Using a 360x360 sized image, a stride of one in each direction, the number of images which are used in the prediction process are 108.241 images[1]. A Jupyter notebook testing the performance of this method can be viewed at

[1] $h + 1 - 32 * w + 1 - 32$, where h and w are the height and width of the input image and the size of the kernel used to predict is 32

`https://aka.klawr.de/sep#1`.

To test this amount of images is often not necessary and can be reduced by increasing the stride of the respective scanning process. This would reduce the accuracy of the localization, but would increase the speed.

Another approach is to use the properties of convolutional layer to restructure the model and thereby making the whole procedure much more efficient.

By transforming a model into a Fully Convolutional Network (FCN) all dense layers are replaced by convolutional layers, and the input layer is also a two dimensional layer but allows for an input image of arbitrary size. Because the input of the original model is only defined by the kernel size, it is agnostic to the size of a previous layer. Listing 2 transforms the `old_model` by appending an `tf.keras.Input` layer without any specified size and replacing the output layer by a `tf.keras.Conv2D` layer[2].

Listing 2: Transformation of the Symbol Classifier into a FCN.

```
inputs = tf.keras.Input(shape=(None, None, 1))

hidden = old_model.layers[0](inputs)

for layer in old_model.layers[1:4]:
    hidden = layer(hidden)

# Get the input dimensions of the flattened layer:
f_dim = old_model.layers[4].input_shape
# And use it to convert the next dense layer:
dense = old_model.layers[5]
out_dim = dense.get_weights()[1].shape[0]
W, b = dense.get_weights()
new_W = W.reshape((f_dim[1], f_dim[2], f_dim[3], out_dim))
outputs = tf.keras.layers.Conv2D(out_dim,
                           (f_dim[1], f_dim[2]),
                           name = dense.name,
                           strides = (1, 1),
                           activation = dense.activation,
                           padding = 'valid',
                           weights = [new_W, b])(hidden)

model = tf.keras.Model(inputs = inputs, outputs = outputs)

model.summary()
```

An example for the usage of this code can be found at `https://aka.klawr.de/sep#2` , where the intricate differences between both approaches are discussed.

The FCN approach is roughly ten times faster than taking crops and predicting them individually. The `model.summary` results in the output given as listing 3:

[2]And thus removing the `tf.keras.flatten` layer.

Listing 3: Summary of Symbol Classifier transformed into a FCN.

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, None, None, 1)]   0

_________________________________________________________________
conv2d (Conv2D)              multiple                  272
_________________________________________________________________
max_pooling2d (MaxPooling2D) multiple                  0
_________________________________________________________________
conv2d_1 (Conv2D)            multiple                  8224
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 multiple                  0
_________________________________________________________________
dense (Conv2D)               (None, None, None, 3)     6147
=================================================================
Total params: 14,643
Trainable params: 14,643
Non-trainable params: 0
_________________________________________________________________
```

Granted this model is satisfactory for the moment, the problem of detecting constraints can be addressed.

1.2 Constraint detection

There was already a model providing the capability of detecting symbols, to prior work. To create a model for detecting constraints the same procedure has to be followed, which means that training data has to be generated first.

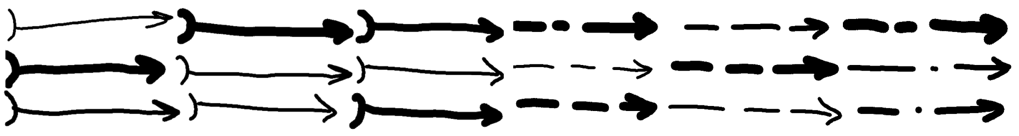
It is planned to support two different types of constraints, which indicate whether two nodes are either able to rotate around one another or to be able to move in a translational fashion. Beside different types of constraints data for three different ranges are generated for a better selection of constraint data when two nodes are to be connected.

Data generation for constraints is similar to data generation in [Law20]. The drawing canvas has a width of fifty and the range can vary between 150-249, 250-349 or 350-449; depending on the selected range.

1.2.1 Preparing nodes on images

Since constraints can occur at all possible angles, the data must be generated in such a way that it resembles the reality the most. Because of this, multiple steps have to be taken to generate comprehensible data.

Metadata generated during the generation process is saved into a format which is



(a) Rotational Constraints (b) Translational Constraints

Figure 1: Some examples of the data created with the described method. Only these two classes were created for the first tests. The images are created horizontally, so their initial angle can be assumed to be zero to keep record when labeling the data dynamically.

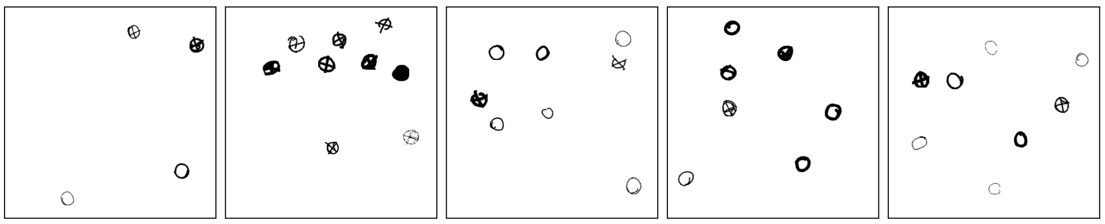


Figure 2: Randomly placed nodes on blank image. Please note that the colors of these samples are inverted.

readable by `pandas`. `pandas` is a open source data analysis and manipulation tool which helps to work and to visualize data.

In the first step 100.000 images are generated. Each image begins with a black background of size 360x360. Between 3 and 10 nodes are randomly placed on the image[3]. Each node has to have a minimum distance of 60 to every other node in the image. Examples for the resulting images are shown in figure 2. The respective notebook containing the code for this process can be viewed at `https://aka.klawr.de/sep#3`.

1.2.2 Connecting nodes using constraints

The next step is to randomly connect pairs of nodes inside of the image. For this the prepared constraints are used, whereas the number of constraints should be random, too.

On an image with n nodes the number of possible connections is $\frac{n\times(n-1)}{2}$. As a reasonable heuristic to keep the number of constraints in check $s = 1 - \frac{2}{n-1}$ is used; where s is the chance of the connection being skipped and n is the number of nodes.

Table 1 describes the expected number of constraints per image. This approach tries to generate as much constraints as there are nodes in the image. The difference of the

[3]In an intermediate step the node data from previous work is modified to consist of white symbols on black background, instead of random grayscale images.

Table 1: Relation of number of nodes to the resulting number of constraints.

Number of nodes:	3	4	5	6	7	8	9
Possible connections:	3	6	10	15	21	28	36
Chance to skip node pair:	0	$\frac{1}{3}$	$\frac{1}{2}$	$\frac{3}{5}$	$\frac{2}{3}$	$\frac{5}{7}$	$\frac{3}{4}$
Expected number of constraints:	3	4	5	6	7	8	9
Mean of constraints in the dataset:	2.62	3.48	4.35	5.21	6.10	6.91	7.78

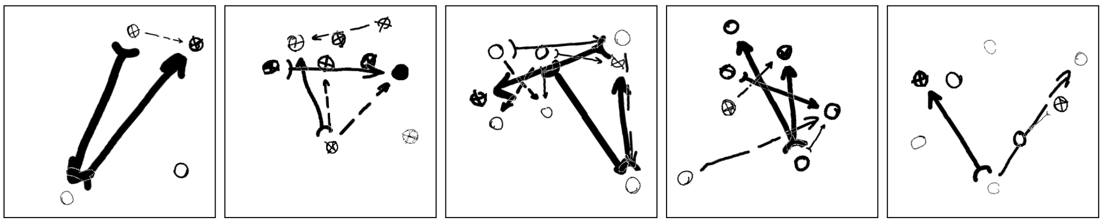


Figure 3: Samples of images generated by connecting randomly selected pairs of nodes using the prepared constraint data. Please note that the colors of these samples are inverted.

expected number of constraints and the mean of constraints per number of nodes in table 1 occurs most likely due to constraints shorter than 50 or longer than 350 being skipped, too. The mean of the number of constraints in relation to the number of nodes is calculated by issuing `[df[df.nodes.str.len() == i].constraints.str.len().mean() for i in range(3,10)]`, where `df` is the respective `pandas.DataFrame`.

The images which are generated in this step are shown in figure 3[4].

1.2.3 Cropping images to get the training data

At last these images are cropped. By cropping and reshaping the models the images can be fed into the training process of a Keras model. The notebook doing this operation can be viewed at `https://aka.klawr.de/sep#4`. As the number of possible connections is $34 = \frac{\sum_{i=3}^{9} i(i-1)}{(10-3)}$, the expected number of generated crop-images is 3.400.000. It is also important to check for reversed constraints between node pairs to classify them as a non connection, to be able to correctly predict the direction of the constraint, too. As 100.000 images are already a good size to work with, another measure was taken to keep the number of expected crops in check. For this crops are only kept $\frac{1}{m(m-1)}$ of the time,

[4]In the initial dataset for constraints 3 different types were made for different length-intervals. After initial testing the shorter constraints did not meet the visual expectations. Thus these images are generated with images from a range of 350-449 only.

which is about each 30th time in this case.

The resulting dataset contains 119.189 images, which are subsequently transformed into a `tensorflow record`.

With the data prepared the next step is to train the constraint detecting model.

1.2.4 Training of the constraint detection model

In this first prototype the design of the constraint detection model is a copy of the results of the previous trained symbol detector. The input size is adapted to fit for the crops, which have a size of 96 by 96 instead of the 32 by 32 images used for the symbol detector. The model is created using the functional API of Keras, which has no functional implications, but is just another way of defining models.

The output of `model.summary()` can be viewed in listing 4.

Listing 4: Summary of Constraint Detector.

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 96, 96, 1)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 96, 96, 16)        272
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 48, 48, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 48, 48, 32)        8224
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 24, 24, 32)        0
_________________________________________________________________
flatten (Flatten)            (None, 18432)             0
_________________________________________________________________
dense (Dense)                (None, 3)                 55299
=================================================================
Total params: 63,795
Trainable params: 63,795
Non-trainable params: 0
_________________________________________________________________
```

The training is done by decoding the record which was created in earlier steps and then splitting the data into 80.000 images for training, 20.000 images for validation and the remaining 19.189 images are used for testing the model.

The training is then initiated by using the `model.fit` method, passing the data as argument. `TensorBoard` is used to log the accuracy and the loss of the model during training. The respective graphs can be seen in figure 4.

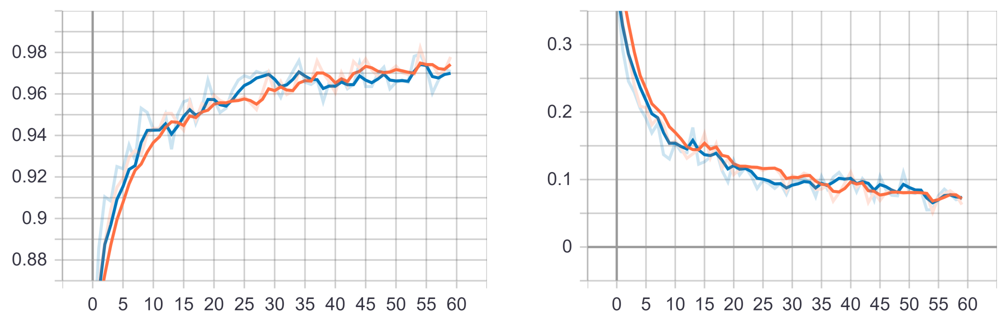


Figure 4: The training accuracy and loss of the crop detector. These graphs were created using TensorBoard, which is used as callback during training.

1.2.5 Combining node and constraint detection

To be able to predict constraints in production the generation of crops has to be implemented as a intermediate step between node and constraint detection. The images must be cropped according to the generation of the training data, which means the coordinates of the nodes have to be detected first, and the area between them is resized to 96 by 96 images. Previously the nodes are placed on the image using (randomly) given coordinates. This process is now revered, getting the coordinates using the Fully Connected Network on the training data and processing the data.

The constraint detection relies heavily on the performance of the node detection, because nodes which are not detected are not taken into account when creating the data for the constraint detector.

Falsely predicted nodes are most likely not going to result in a pair of nodes connected by constraints, since the resulting crop would most likely not look like a constraint, but they are slowing the process down significantly, since the number of crops grows exponentially to the number of nodes.

A pipeline of this process can be reviewed at `https://aka.klawr.de#5`.

1.3 Conversion to the web context

So far all the code is written in Python. To make the data pipeline available in a web based context, the models have to be converted into a format readable by a JavaScript and the pipeline has to be adjusted accordingly.

For the conversion of the models the library `Tensorflow.js` is used. Tensorflow.js takes the model file as input and outputs a description of the model in `JSON` format and

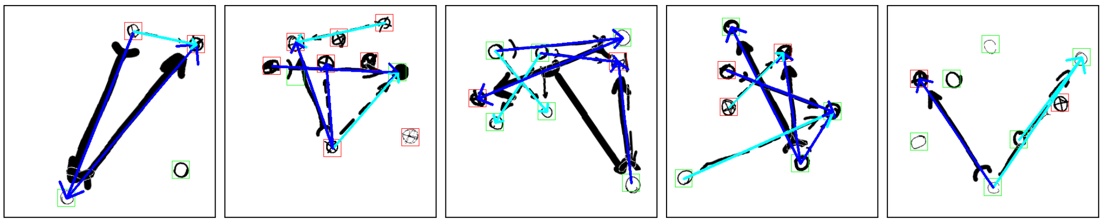


Figure 5: The second step of the data generation processed created images with nodes and constraints that resemble nodes and constraints placed randomly into a canvas. These predictions are made on the images from figure 3. There is only one falsely predicted node in the second image from the left. The constraints are way less accurate, which is not surprising considering the composition of these images.

the trained weights and biases are contained in a `group1-shard1of1.bin` file.[5].

To include the weights and binaries into a importable JavaScript file it is converted into a base64 formatted string. For this the `base64` [Jos18] module contained in `GNU Coreutils` is used. The resulting string is copied into a JavaScript file, containing one class, as can be seen in listing 5

Listing 5: Class definition of one of the models providing necessary functions to be used by `tf.loadLayersModel` to return a usable model in JavaScript.

```
class ConstraintModel {
    model=`{/* JSON with information about the model */ }`;
    bin=`/* base64 string with information about weights and biases */`;
    async load() {
        function base64Decode(base64) {
            const binaryString = window.atob(base64);
            const bytes = new Uint8Array(binaryString.length);
            for (let i = 0; i < binaryString.length; ++i) {
                bytes[i] = binaryString.charCodeAt(i);
            }
            return bytes.buffer;
        }

        const a = JSON.parse(this.model);
        a.weightSpecs = a.weightsManifest[0].weights;
        a.weightData = base64Decode(this.bin);
        return a;
    }
}
```

The contents of some properties are commented out, to keep this listing concise. The full file can be reviewed at `https://aka.klawr.de/sep#7`.

[5]Instructions on how to import a Keras model into Tensorflow.js can be viewed at `https://aka.klawr.de/sep#6`.

For the conversion each model is assigned to one class, which has three properties. `model` contains the content of the JSON and `bin` the base64 string without any further modifications. Furthermore objects created by this class contain a `load` function, which returns the respective model by parsing the JSON into an object, where the `weightData` of the model is added by decoding the content of `bin`.

A model can then be created by issuing `tf.loadLayersModel(new Crop())`[6].

At the moment it is not possible to convert a model into a Tensorflow model after it is processed in any form. Therefore a function has to be written which takes the initially used node detector as input and converts it into a FCN. For this a generalized form of the conversion processed has been written, which can be reviewed at `https://aka.klawr.de/sep#8`.

The models are implemented into the JavaScript pipeline by embedding all necessary functions into an object, called `deepmech`. This object has 6 properties, which are used to use the previously defined classes to create models and predict images using them.

1. `nodeDetector` - an immediately invoked function execution is used to call the `tf.loadLayersModel` function provided by Tensorflow.js by submitting a `new models.NodeModel()`. The return of this function is immediately used as input for the `toFullyConv` function.

2. `constraintDetector` - a function similar to the `nodeDetector` property, but without the necessity to convert the model into a FCN.

3. `detectNodes` - this function takes an image and the previously defined nodeDetector as input and returns the detected nodes with some preprocessing steps in between.

4. `getCrops` - another function which uses an image and previously determined nodes to create crops. These crops are equivalent to those generated in chapter 1.2.3.

5. `detectConstraints` - The generated crops are used in conjunction with the model returned by `constraintDetector` to predict constraints.

6. `predict` - This function is intended to be actually used by external processes. It combines all other functions by taking an image as input and returning the respective predictions of nodes and constraints in an image.

[6] It is actually a promise of the respective model object, so this has to be loaded using an `async` function.

By converting all the necessary steps into a JavaScript pipeline, the `deepmech` object may be implemented into a library to be used in any combination. For demonstration purposes it is implemented into Node.js and by using `ijavascript` and the `conda` package of Node.js integrated into a Jupyter Notebook, which can be viewed at `https://aka.klawr.de/sep#9`.

1.4 Implementation into mec2

1.4.1 mec2

mec2 is a 2D physics simulation engine written in JavaScript. It is designed to easily create 2D mechanisms for rapid sketching and rendering the resulting models in a 2D canvas using g2.

Mechanisms are defined by using JSON formatting, as shown in listing 6.

Listing 6: Example for a mechanism defined in the syntax proposed by `mec2`.

```
{
    "nodes": [
        { "id": "A0", "x": 75, "y": 50, "base": true },
        { "id": "A", "x": 75, "y": 100 },
        { "id": "B", "x": 275, "y": 170 },
        { "id": "B0", "x": 275, "y": 50, "base": true },
        { "id": "C", "x": 125, "y": 175 } ],
    "constraints": [
        {   "id": "a", "p1": "A0", "p2": "A", "len": { "type":"const" },
            "ori": { "type": "drive", "Dt": 2, "Dw": 6.28 } },
        {   "id": "b", "p1": "A", "p2": "B", "len": { "type":"const" } },
        {   "id": "c", "p1": "B0", "p2": "B", "len": { "type":"const" } },
        {   "id": "d", "p1": "B", "p2": "C", "len": { "type":"const" },
            "ori": { "ref": "b", "type": "const" } } ],
    "views": [
        {   "show": "pos", "of": "C", "as": "trace", "Dt":2.1,
            "mode":"preview", "fill":"orange" },
        {   "show": "vel", "of": "C", "as": "vector" },
        {   "as": "chart", "x": 340, "y": 75, "Dt": 1.9,
            "show": "wt", "of": "b" } ]
}
```

mec2 is divided into many sub modules; some of which are optional, but some are necessary to simulate functioning mechanisms.

For the first prototype only necessary modules have to be used; namely:

1. `mec2.core`, which defines the central JavaScript object other modules are built upon. The core `mec` object defines central properties, which make it easy to change certain parameters for the whole simulation. Tolerances are defined for exit conditions inside the iterative calculations of the simulation. Central settings

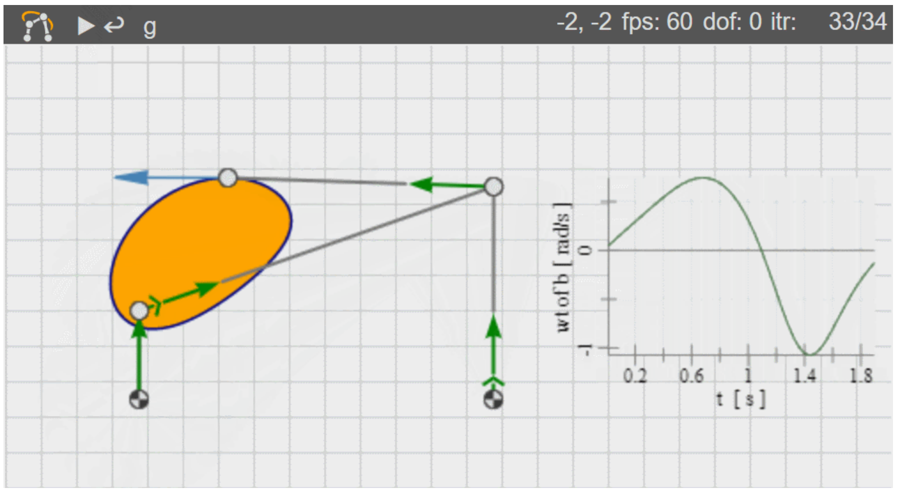


Figure 6: The mechanism defined in listing 6. It is rendered using the mec2 custom HTML element. Defined are five nodes and 4 constraints. The chart and trace views are generated after one revolution of the leftmost constraint.

for rendering, such as colors for individual parts, color modes for readability in light and dark environments are defined here. To show and hide certain parts of the mechanism can be controlled here. For each of these properties default values are set.

2. `mec2.model` adds certain functionality to the `core` object. This and other `mec2` modules add properties to the prototype of the `mec2` core object. `models extend` function sets the prototype of a JavaScript object to be the prototype of `mec2`'s `core` Object. This approach allows for an easy extensibility of objects by adding modules to the core object.

   `mec2.model` also serves as a hub for all other modules to be implemented in. By delegating certain functionality it suffices to issue e.g. the models `init` function to call `init` on all other objects handled by other modules as well.

3. `mec2.node` implements one of the two essential elements of mechanisms (the other being constraints). Nodes are to be seen as particles, which can have a degree of freedom of 2. They are implemented with a default mass of 1kg.

   Nodes do not interact with anything but the environment, so no collision is implemented. They are only restricted in certain movements by constraints.

4. `mec2.constraints` are the only thing able to restrict nodes in certain directions. Usually a constraint is used to reduce the degree of freedom of a node by one, but

it is also possible to take all two degrees of freedom of a node, or none.

At the time of writing five other modules[7] exist to extend the functionality of `mec2`, but they do not concern the implementation of `deepmech` and are not further discussed here.

Additionally `mec2` provides a custom HTML element, which allows for an easy implementation into web pages. The object defining the model is given as JSON inside the `innerHTML` of the respective custom HTML to define the input, which is respectively parsed using the standard built-in function `JSON.parse()`. The custom HTML element adds other functionality besides the appropriate implementation of `mec2` with all modules.

1.4.2 g2

For rendering models `g2` is used. `g2` is a JavaScript library which constructs an array of commands which can then be handed over into a drawing context via a dedicated `exe` method. The default handler of `g2` issues commands to a canvas handler, which in turn uses the standard built-in canvas API to draw images on an HTML canvas element.

Listing 7: Example code of a truss defined with g2.

```
const A = { x: 40, y: 30 },
    B = { x: 150, y: 30 }, C = { x: 40, y: 80 },
    D = { x: 100, y: 80 }, E = { x: 40, y:130 };

g2().view({ cartesian: true })
    .link2({ pts: [ A, B, E, A, D, C ]})
    .nodfix({ ...A, scl: 1.5 })
    .nodflt({ ...E, scl: 1.5, w: -Math.PI / 2 })
    .nod({ ...B, scl: 1.5 })
    .nod({ ...C, scl: 1.5 })
    .nod({ ...D, scl: 1.5 })
    .vec({
        x1: D.x, y1: D.y, x2: D.x + 50,
        y2: D.y, ls:'darkred', lw :2 })
    .vec({
        x1: B.x, y1: B.y, y2: B.y - 20,
        x2: B.x, ls:'darkred', lw :0.5 })
    .ground({ pts: [
        { y: E.y + 20, x: E.x - 23 },
        { y: A.y - 18, x: A.x - 23 },
        { y: A.y - 18, x: D.x }]})
    .exe(ctx)
```

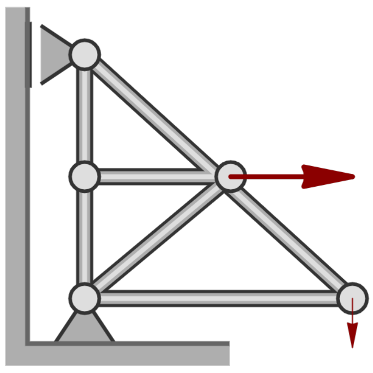


Figure 7: Image generated by listing 7.

[7]The other modules being `mec2.load`, `mec2.drive`, `mec2.view`, `mec2.msg.en` and `mec2.shape`

Likewise mec2, g2 is structured in a modular manner, but the g2.core module suffices for basic drawings. At the time of writing six other modules exist for g2[8], but at this point only the modules relevant to the case will be examined more closely.

g2.ext provides necessary functionality for other modules to use, such as positioning capability for labels. Modules like g2.chart provide only one new (but extensive) command, which can be subsequently used by other libraries, like mec2. g2.mec provides definitions to draw shapes and symbols that are often used in engineering, as can be seen in figure 7. g2.selector is an extension used in interactive environments to be able to select certain geometries and interact with them. For this purpose g2.selector enables the introduction of a new context which can be used to find elements whose coordinates match those of a corresponding mouse pointer.

1.4.3 canvasInteractor

The canvasInteractor.js is an interaction manager for the HTML canvas element. It is used to centralize all interactions and animation requests. canvasInteractor manages one requestAnimationFrame, which minimizes overhead as if each component would issue a new request. Additionally this approach parallelizes all requests, because they are all handled in one function call.

New instances can be added to the canvasInteractor object by calling its add function, but to create a new interactor the create function has to be issued, which applies the prototype of the canvasInteractor object onto the variable. The created objects get all event listeners necessary to provide full capability for interactions.

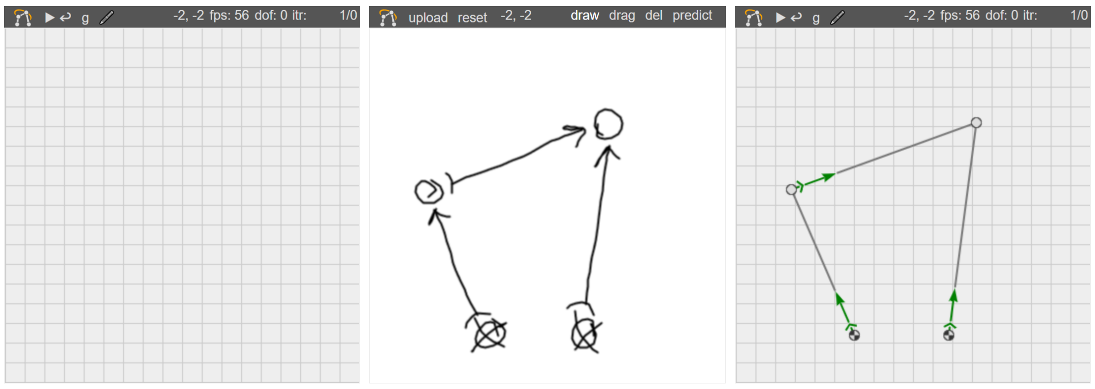
1.4.4 Putting it together

The next step is to implement deepmech into this set of libraries. At first a script is written, which adds the necessary functionality into all mec2 HTML elements. This is done by document.getElementsByTagName('mec-2') and then iterating over the returned array, to ensure the extensions are applied everywhere.

The target of the written script is to create the possibility inside the mec2 HTML element to to create images, predict nodes and constraints and return the respective mec2 representation.

For the activation of the additions described here a button has been added to the navigation bar. The respective button in the form of a pencil can bee seen in figure 8a.

[8]Namely g2.ext, g2.lib, g2.io, g2.mec, g2.chart, g2.selector and g2.editor.



(a) (b) (c)

Figure 8: The first image depicts `mec2` in its default state. The only thing visible due to the additions is the pencil symbol in the navigation bar. The second image depicts a drawing of a fourbar. Please note that the actual image is inverted, but that does not make any real difference. The third image shows the predicted mechanism.

The activation of the button initiates few changes to the navigation bar, which changes the context the user interacts with, which can be seen in the navigation bar in figure 8b.

Changes in the canvas are managed by only one g2 command queue, which inherits multiple other command queues for each different case, as can be seen in listing 8.

Listing 8: Command queue for the drawing context in the prototype.

```
const command_queue = g2().rec({
        x: () => -view.x / view.scl,
        y: () => -view.y / view.scl,
        b: () => element.width / view.scl + 1,
        h: () => element.height / view.scl + 1,
        fs: '#000',
        isSolid: false
    })
    .view(view)
    .use({ grp: () => img_placeholder })
    .use({ grp: () => mec_placeholder })
    .use({ grp: () => ply_placeholder });
```

Herein `view` is taken from `element._interactor.view` of the respectve `mec2` `element`.

The background of the drawing canvas is colored black (`#000`) by issuing the `rec` command of g2 for the active viewport[9].

The `img_placeholder` is used to draw images loaded by issuing `uploadImage`, which loads an image and saves the respective image inside this command queue.

[9]Please note that figure 8b contains an inverted image for contrast reasons.

The `mec_placeholder` is used to be able to show already determined nodes and constraints of the mechanism inside the drawing context. This makes it possible to extend existing mechanisms. A `mec2` model is usually defined by more than just nodes and costraints, so all elements which are not part of codeelement._model.nodes or `element._model.constraints` are filtered first; e.g. the chart which can be seen in figure 6.

The `ply_placeholder` is responsible for rendering user generated drawing. The default behavior of the `canvasInteractor` when pressing the pointerbutton in conjugation of movement of the pointer is to drag the viewport, or, when an element is selected by the `g2.selector`, to drag an element. When the drawing mode is active this behavior has to be replaced. When the draw mode is active and `pointerdown` is issued an array is created which is then filled by the coordinates of the pointer on each `tick`[10]. This approach creates one `ply` element for each consecutive line, which can be dragged or deleted by changing the respective mode using the `g2.selector`.

By starting a prediction the content of the canvas is fed into the trained models, as discussed in chapter 1.3. The resulting prediction updates the `mec2` model. This is similar to the generation of bounding boxes, but instead of drawing them onto the canvas to show the results the respective coordinates are added to the `mec2` model. The assembled `mec2` model can than be rendered and interacted with.

References

[Jos18] Simon Josefsson. *base64*. 2018. URL: `https://www.gnu.org/software/coreutils/manual/html_node/base64-invocation.html` (visited on 04/01/2020).

[Law20] Kai Lawrence. *Student Research Project*. Research rep. 2020.

[10]A `tick` is issued by the `canvasInteractor` on each requestAnimationFrame