Fachhochschule Dortmund

Student Engineering Project

Case study of the creation of an editor interacting with planar linkages using machine learning

Kai Lawrence

February 22, 2021

Contents

1 Introduction **1**

2 First prototype **2**

2.1 The Fully Convolutional Network 2

2.2 Constraint detection 4

2.2.1 Preparing nodes on images 5

2.2.2 Connecting nodes using constraints 5

2.2.3 Cropping images to get the training data 6

2.2.4 Training of the constraint detection model 7

2.2.5 Combining node and constraint detection 8

2.3 Conversion to the web context 9

2.4 Implementation into mec2 11

2.4.1 mec2 11

2.4.2 g2 14

2.4.3 canvasInteractor 15

2.4.4 Putting it together 15

3 App implementation **17**

3.1 Creation of a Progressive Web App 17

3.2 Structure of the application 17

3.2.1 Left drawer 18

3.2.2 Right drawer 19

3.2.3 The canvas 22

3.3 State manipulation 24

3.4 State of the `mec2` model 26

3.4.1 MecModelSlice 26

3.4.2 Updating the mecElement 27

3.4.3 Implement bidirectionality 29

3.4.4 Undo and redo 30

4 Third party integrations **31**

4.1 WPF and WinUI 32

4.2 Interface to the PWA 32

4.3 Interface of the WPF 33

4.4 Predictions in C++ 34

4.5 Implementation in WinUI 36
4.6 Webserver implementation 37

5 Conclusion **38**

1 Introduction

This project builds upon work previously done to learn the fundamentals of machine learning [Law20]. The target of the project was to introduce machine learning in to the field of kinematics. This is should be used mainly in the web environment. Because of this inference model to classify handwritten mechanical symbols is created which can be converted to be used.

In this project this idea is taken a step further to detect whole mechanisms by determining the location of possibly multiple symbols in pictures of various size[1].

After localization is done, the next step is to detect the connections between nodes. For this the images between nodes have to be cropped and processed to be able to get a reasonable prediction. A new inference model is trained to classify these processed images to their respective class of connection[2].

The resulting coordinates and labels are converted into `mec2` nodes and constraints as JSON-string. This conversion makes it possible to convert the gathered information and make them processable for physic engines like `mec2`. The creation of these JSON-strings via this method is then tested in various environments.

A Progressive Web App (PWA)[3] is created to nest this new functionality into an application. For this the inference models are transformed to work in JavaScript to be able to make predictions from within other environments.

The `mec2` HTML element is embedded into this app to be able to apply detected nodes and constraints to a mechanical linkage. This PWA is able to edit the mechanical linkage itself and implements the `mec2` API to offer all features the `mec2` HTML element does.

The PWA is then used as frontend for different implementations using the Windows Presentation Foundation (WPF) [Mic21a] and the Windows User Interface Library (WinUI) [Mic21b]. These applications explore the possibility of linking the model to other programming languages, like C++, to improve performance. Other ideas like communication between the PWA and a webserver to get better performance are also explored.

[1]The data used to train the symbol classifier assumes symbols which are centered and fill the whole picture

[2]A connection between two nodes can be either limiting the rotational or lateral movement

[3]See `https://aka.klawr.de/sep#1`.

2 First prototype

A prototype for the inference model 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. Constraints limit the `nodes` in their degree of freedom, resulting in a mechanical linkage if they are arranged accordingly.

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

2.1 The Fully Convolutional Network

In this section the trained inference model [Law20] is improved to provide not only the class of the detected symbol, but the location in an image of arbitrary size.

The respective model can be loaded using `Keras` [Cho19]. `Keras`' `model.load_model` loads the inference model and by issuing the `summary` method of the resulting object 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
_________________________________________________________________
```

A naive 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[4]. A Jupyter notebook testing the performance of this method can be viewed at

[4] $h + 1 - 32 * w + 1 - 32$, 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#2.

To test this amount of images is often not necessary and can be reduced by increasing the stride of the respective scanning process. This tradeoff reduces 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) [LSD14] all dense layers are replaced by convolutional layers. The input layer is converted to a two dimensional layer which allows for input images of arbitrary size. Because the input of the original model is only defined by the kernel size (32x32), 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[5].

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 of the usage of this code can be found at https://aka.klawr.de/sep#3, where the intricate differences between both approaches are examined.

A measure of the time has shown that the approach using an FCN is roughly ten

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

times faster than by taking crops and predicting them individually. The `model.summary` results in the output given as Listing 3:

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

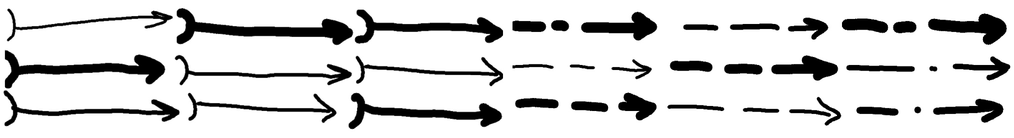
2.2 Constraint detection

To classify constraints an inference model is trained. To create such a model suitable training data has to be generated first.

The target is to be able to distinguish between two types of constraints: Constraints to prevent nodes to change the range between each other, which allows only for rotational movement. It is represented by an arrow with an arc at the origin. The other type of constraint prevents motion that would change the angle between two nodes, which only allows translational movement between two nodes. It is represented by an arrow with two gaps on the shaft and no arc at the origin. Examples for the generated data can be seen in figure 1.

Furthermore, the data is split for constraints of different lengths. This allows for a better selection of constraints when two nodes are to be connected when generating the training data lateron.

Data generation for constraints is similar to data generation in [Law20]. The drawing canvas has a width of fifty pixels and the range can vary between 15–249, 250–349, or



(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 a record when labeling the data dynamically.

350–449 pixels; depending on the selected range.

For each type approximately 1200 images were drawn, 400 for each range, resulting in 2400 images[6]. These images can be used in conjuction of existing images of nodes to create pseudo-realistic images of hand-drawn mechanisms.

2.2.1 Preparing nodes on images

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

Metadata generated during the generation process is saved into a format that is readable by `pandas` [pan21]. `pandas` is an open-source data analysis and manipulation tool which helps to work and to visualize data.

Each generated image begins with a black background of size 360x360. Between 3 and 10 nodes are randomly placed on the image[7]. It is prescribed that each node has to have a minimum distance of 60 to every other node in the image to improve the quality of the training data. 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#6`. Using this method 100.000 images are generated.

2.2.2 Connecting nodes using constraints

The next step is to randomly connect pairs of nodes inside of the image using constraints. The number of constraints is considered be random. It is important to note that this approach does not result in working mechanisms most of the time. But as it turns out

[6]The raw data can be downloaded at `https://aka.klawr.de/sep#4`.

[7]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. It can be accessed at `https://aka.klawr.de/sep#5`.

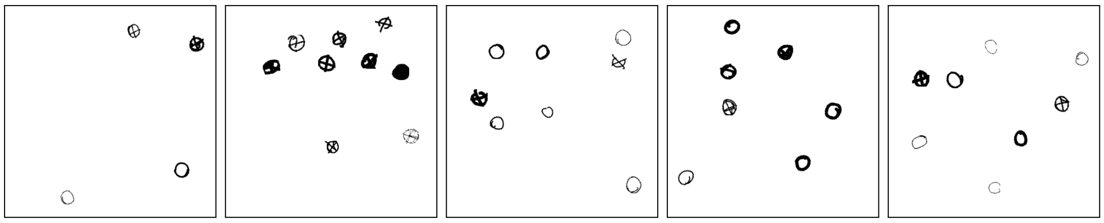


Figure 2: Randomly placed nodes on a blank image. Please note that the colors of these samples are inverted.

Table 1: Relationship of the 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

this is not important, because the model will be classifying individual elements and not the mechanism as a whole.

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 many constraints as there are nodes in the image. The difference of the 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[8].

2.2.3 Cropping images to get the training data

At last, these images are cropped. After cropping and reshaping the training data the images can be inserted into a training process of a `Keras` model. The notebook

[8]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.

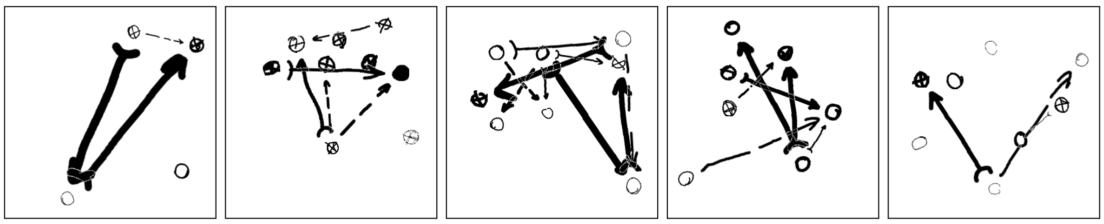


Figure 3: Samples of images generated by connecting randomly selected pairs of nodes using the prepared constraint data. Please note that the colors of these samples are inverted.

demonstrating this operation can be viewed at `https://aka.klawr.de/sep#7`. 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. 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, which is about each 30th time in this case.

The resulting dataset contains 119.189 images, which are subsequently transformed into a `tensorflow record`[9].

The next task is to train the inference model for constraint detection.

2.2.4 Training of the constraint detection model

In this prototype, the design of the constraint detection model is a copy of the results of the previously trained symbol detector. The input size is adapted to fit the cropped images, 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 real impact on the result, but is just another way of defining models[10].

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
_________________________________________________________________
```

[9]The data can be downloaded from `https://aka.klawr.de/sep#8`.
[10]See `https://aka.klawr.de/sep#9`.

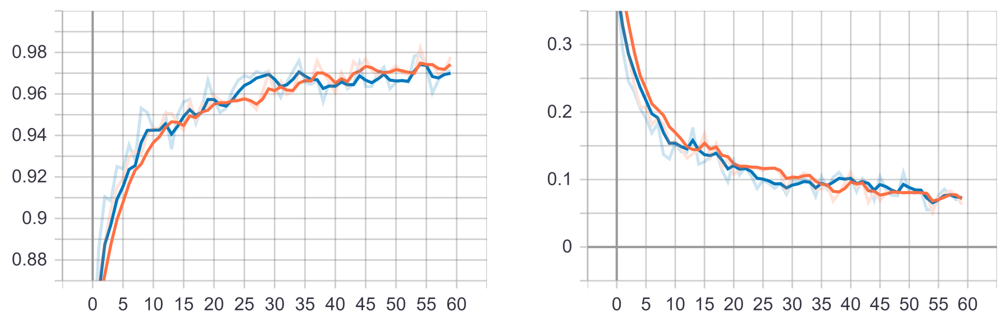


Figure 4: The training accuracy and loss of the crop detector. These graphs were created using TensorBoard, which is used as the callback during training.

```
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 training data as the arguments. `TensorBoard` [Goo21a] is used to log the accuracy and the loss of the model during training. The generated graphs can be seen in figure 4.

2.2.5 Combining node and constraint detection

To be able to predict constraints in production, the generation of crops has to be implemented as an intermediate step between the 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 reshaped to represent 96 by 96 images. Previously the nodes are placed on the image using (ran-

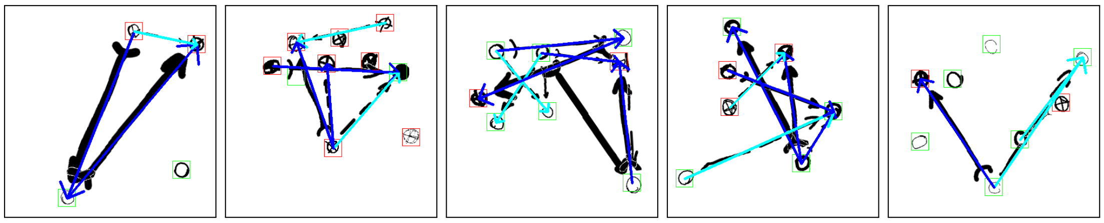


Figure 5: The second step of the data generation processed created images with nodes and constraints that resemble nodes and constraints placed randomly into a canvas. These predictions are made on the images from figure 3. There is only one falsely predicted node in the second image from the left. The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

domly) 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 that are not detected are not taken into account when creating the data for the constraint detector. Falsely predicted nodes on the other hand can not be connected by a constraint, slowing the process down significantly since the number of crops grows exponentially to the number of nodes.

A pipeline of this operation can be reviewed at `https://aka.klawr.de/sep#10`.

2.3 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 JavaScript and the pipeline has to be adjusted accordingly. For this `TensorFlow` has an implementation in JavaScript [Goo21b].

For the conversion of the models, the `tensorflowjs_converter` is used. This program takes the model file as input and outputs a description of the model in `JSON` format. The trained weights and biases are contained in `group1-shard1of1.bin`[11].

To include the weights and biases into an usable 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.

[11]Instructions on how to import a Keras model into Tensorflow.js can be viewed at `https://aka.klawr.de/sep#11`.

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 the listing more concise. The full file can be reviewed at `https://aka.klawr.de/sep#12`.

For the conversion each model is assigned to a class with three members. `model` contains the content of the JSON file and `bin` the base64-string without any further modifications. Furthermore, objects created by this class contain a `load` function. `load` returns the respective model by parsing the JSON into an object. The `weightData` of the inference model is added by decoding the content of `bin`. The inference model can be loaded by issuing `tf.loadLayersModel(new Crop())`, provided by the `Tensorflow.js` library.

At the moment it is not possible to convert a model into a Tensorflow model after it is processed in any form. Therefore the previously discussed Fully Convolutional Network can not be loaded directly. A function has to be written which takes the unmodified node detector as input and converts it into an FCN. For this a generalized form of the conversion processed has been implemented in JavaScript.

Functions loading the inference 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 returned value of this function is immediately used as input for the `toFullyConv` function.

2. `constraintDetector` — This function is similar to `nodeDetector`, 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` — A function which uses an image and previously determined nodes to create crops. These crops are equivalent to those generated in Chapter 2.2.3.
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 used by external processes. It combines all other functions by taking an image as input and returning the respective predictions of nodes and constraints.

By converting all the necessary steps into a JavaScript pipeline, the `deepmech` object may be implemented into other libraries or applications. For demonstration purposes, it is implemented using the Node.js [Joy21] runtime environment, by using `ijavascript` and the `conda` package. A Jupyter Notebook is created to show the results. It can be viewed at `https://aka.klawr.de/sep#13`.

2.4 Implementation into mec2

2.4.1 mec2

`mec2` is a physics simulation engine written in JavaScript [Gös21b][Gös19b][Gös19a]. It is designed to easily create planar mechanical linkages. The models can be rendered on a canvas HTML element using the `g2` graphics library[Gös21a]. `mec2` models provide respective native functionality for `g2`.

`mec2` models are defined using the JSON-format, as shown in Listing 6. Therein `nodes` are described by a unique `id`, their location in respect to the bottom left of the view using the properties `x` and `y` and if they are considered a `base`[12]. Constraints have an `id` too. Furthermore `p1` and `p2` have to be defined, which refer to nodes by their `id`. The `len` and `ori` property have a `type` which, for the scope of this project, can be either

[12]Bases are simulating the ground and are treated as being unmovable

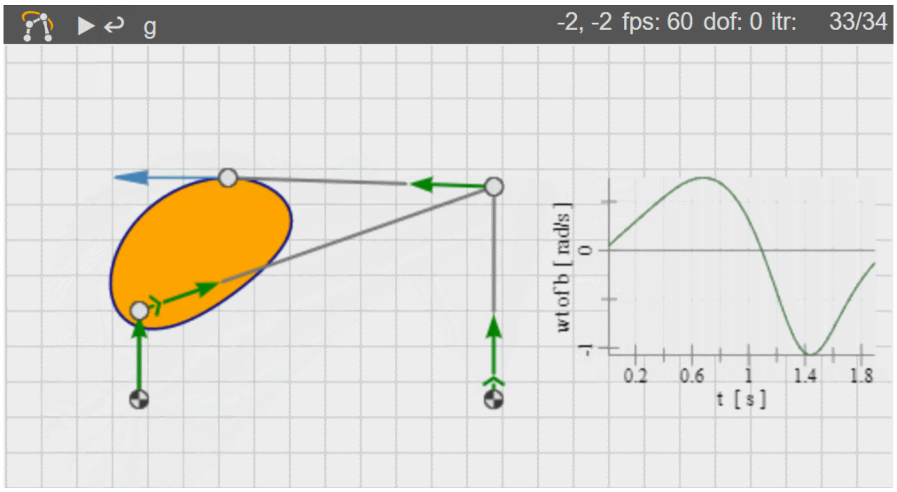


Figure 6: The mechanism defined in Listing 6. It is rendered using the mec2 custom HTML element. Defined are five nodes and 4 constraints. The chart and trace views are generated after one revolution of the leftmost constraint.

`const` or `free`, defining the constrained behaviour of the respective nodes `p1` and `p2` either lateral or rotational (or both).

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`s functionality is divided into multiple plugins; some of which are optional, but some are necessary to simulate functioning mechanisms.

For this prototype only necessary modules have to be used:

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 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 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 model's 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[13] exist to extend the functionality of mec2, but they do not concern the implementation of deepmech yet and are not further discussed here.

Additionally, mec2 provides a custom HTML element, which allows for an easy implementation into web pages[14]. 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 built-in function JSON.parse().

[13]The other modules being mec2.load, mec2.drive, mec2.view, mec2.msg.en and mec2.shape.
[14]For more information about custom HTML elements see https://aka.klawr.de/sep#14.

2.4.2 g2

For rendering models g2 is used. g2 is a JavaScript library that 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)
```

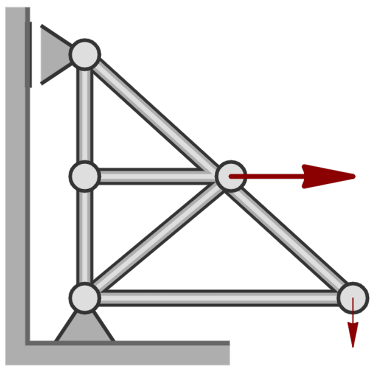


Figure 7: Image generated by Listing 7.

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[15], but at this point only the modules relevant to the case will be examined more closely.

g2.ext provides the 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.

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

2.4.3 canvasInteractor

The canvasInteractor.js is a JavaScript object and acts as an event manager for the HTML canvas element. It is used to centralize all interactions and animation requests; e.g. the canvasInteractor raises one requestAnimationFrame. Other components may register their events to the canvasInteractor instead of raising the respective events on their own. 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. To create a new instance of the canvasInteractor the create function has to be issued. It applies the prototype of the canvasInteractor object onto a given variable. The created objects get all event listeners necessary to provide the full capability for interactions.

2.4.4 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 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. The activation of the button initiates a 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 });
```

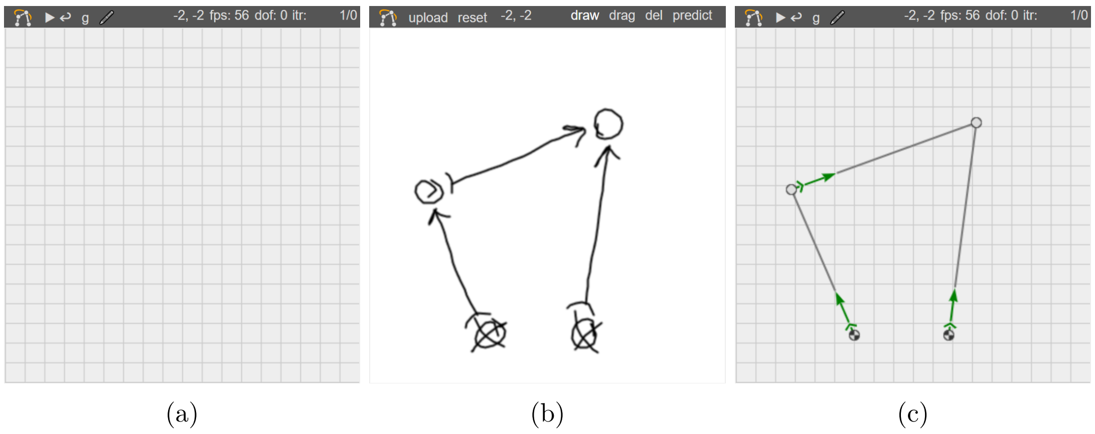

g
-2, -2 fps: 56 dof: 0 itr: 1/0
upload reset -2, -2
draw drag del predict
g
-2, -2 fps: 56 dof: 0 itr: 1/0
(a)
(b)
(c)

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 four-bar. Please note that the actual image is inverted, but that does not make any real difference. The third image shows the predicted mechanism.

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[16].

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.

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. Therefore all elements which are not part of `element._model.nodes` or `element._model.constraints` are filtered first; e.g. the chart which can be seen in figure 6 is removed.

The `ply_placeholder` is responsible for rendering user-generated drawing. The default behavior of the `canvasInteractor` when pressing the pointer-button in combination with movement of the pointer is to drag the viewport. When an element is selected by the `g2.selector` the selected element is dragged instead. When the drawing mode is active this behavior has to be replaced. If the `pointerdown` is issued while in drawing mode, an array is created which is then filled by the coordinates of the pointer on each `tick`[17]. This approach creates one `ply` element for each consecutive line, which can be

[16] Please note that figure 8b contains an inverted image for aesthetic reasons.
[17] A `tick` is issued by the `canvasInteractor` on each `requestAnimationFrame`.

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 2.3. The resulting prediction is the used to update 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, simulated and interacted with.

3 App implementation

The prototype suffices as proof of concept. However, the model is not implemented in a very usable environment. The prototypical implementation in Chapter 2 applies the model and its implementation into each instance of `mec-2` and patches a canvas on top of the element to be able to draw. This is not optimal, because this approach is neither flexible nor extensible. So instead of using just the `mec2` HTML-element, a web application is built around it. This web application may be used as a progressive web app (PWA) to be accessed via a browser or even embedded into other third-party frameworks.

3.1 Creation of a Progressive Web App

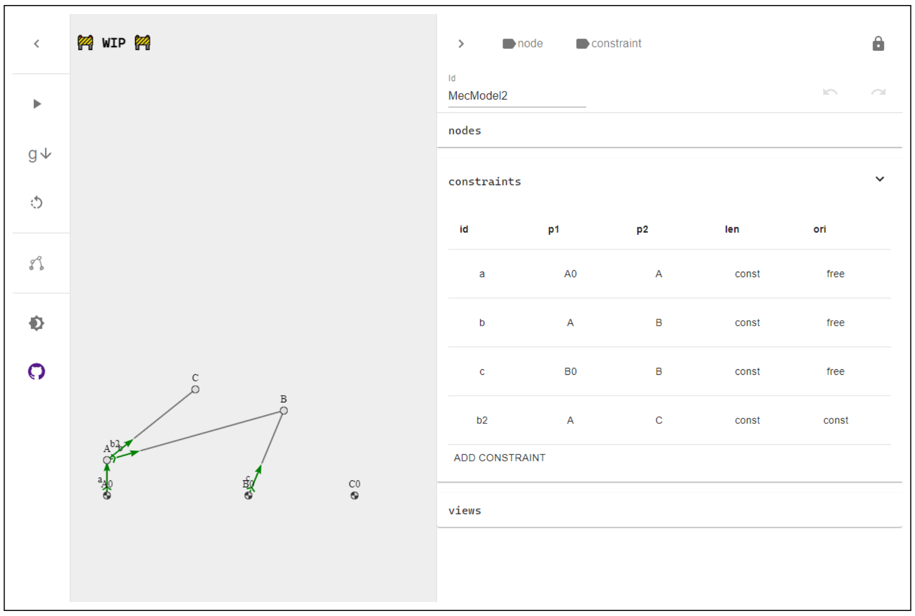
The goal of the application is to provide a user interface to work with `mec2` interactively. The application should be designed to be extensible, thus allowing other means of interactions with `mec2`; for example using hand-drawings.

Components used to create the user interface are mostly written using `React` [Fac21]. `React` is a JavaScript framework, which implements components similar to custom HTML elements. It furthermore allows standard HTML and JavaScript besides it. This is a requirement the original `mec2` HTML-Component should be used without any modifications. By using the `materialUI`[Mat20] a consistent and modern layout is created.

3.2 Structure of the application

The most prominent part of the application is occupied by the `mec2` Custom HTML element. It is stretched across the whole viewport. On both sides are `Material-UI Drawer` elements, enabling the user to interact with different parts of the application.

The structure of the user interface aims to be simple and straight forward. To define individual components `Reacts` `jsx` syntax is used. `jsx` is a syntax very similar to `HTML`



Id
MecModel2

nodes

constraints

id	p1	p2	len	ori
a	A0	A	const	free
b	A	B	const	free
c	B0	B	const	free
b2	A	C	const	const

ADD CONSTRAINT

views

Figure 9: Both drawers are expanded. On the left the `mec2` controls can be seen, on the right a summary of elements of the model are listed. The tab for `constraints` is expanded where some properties are shown

but with the ability to implement JavaScript into the elements[18].

3.2.1 Left drawer

The left drawer contains the components `MecControl`, `DeepmechControl` and other Button elements. `MecControl` is a custom `React` component to replace the default `mec2` controls by issuing its API.

Listing 9: Definition of the `MecControl` component.

```
export default function MecControl({ mecReset, className }) {
    const dispatch = useDispatch();
    const mec = useSelector(mecSelect);

    return <List className={className}>
        <ListButton onClick={() =>
            dispatch(mecAction.toggleRun())} tooltip="Run/Pause mechanism">
            {mec.pausing ? <PlayArrow /> : <Pause />}
        </ListButton>
        <ListButton onClick={() =>
            dispatch(mecAction.toggleGravity())} tooltip="Toggle gravity">
            g {mec.gravity ? <Clear /> : <ArrowDownward />}
        </ListButton>
```

[18]By using `jsx` it is now necessary to transpile the project using `Babel.js` [Bab21].

```
        <ListButton onClick={mecReset} tooltip="Reset">
            <RotateLeft />
        </ListButton>
    </List>
}
```

Listing 9 shows the function which returns the component replacing the default mec2 controls. It contains the definition of the variables dispatch and mec. Selector and useDispatch are used for state management of the user interface and are examined in Chapter 3.3. The return value of a component is defined in jsx syntax. List is a Material-UI component. The list items inside are implemented using ListItems. To reuse Material-UI components multiple times without calling them every time a custom ListButton component is created. The ListButton is a combination of the ListItem, Tooltip and IconButton components and can be reviewed at https://aka.klawr.de/sep#15.

The left drawer also contains the DeepmechControl. It is responsible for handling the draw mode. If the draw mode is disabled, this component only shows a button to activate it. If the drawing mode is enabled the left drawer does not show the MecControl anymore. Instead, the DeepmechControl shows four different modes interacting with the drawing canvas: drawing, panning and deletion of line strokes and a camera mode, which is not implemented yet. Details of the different modes and how they are implemented are explained in Chapter 3.3.

The two bottom-most buttons are responsible for toggling the dark-mode[19] and to navigate to the project page: https://github.com/klawr/deepmech.

3.2.2 Right drawer

The drawer on the right side intends to show all relevant information of the mec2 model. On the top, there are two buttons labeled nodes and constraints each with a Label icon. Those are used to toggle the labels of nodes and constraints respectively. The Lock-button is used to keep the drawer open when interacting with the model. The right drawer is locked by default if the side is wide enough to show the content; i.e. wider than 1200 pixels.

The last component in the RightDrawer is the MecProperties component. The MecProperties component contains a collection of other custom components. At the time of writing, four properties of a mec2 model have designated components. The first component is the Id component. It is showing the id property of the mec2 model and

[19]The app starts in dark mode if the executing browser is in dark-mode, and vice versa.

updates it on changes.

Currently, three of the modules provided by mec2 are implemented into the app: Nodes, Constraints and Views. They are all structured in a very similar way to have a somewhat unified interface for the user. To achieve this the return value of these components is very similar and can be seen in Listing 10.

Listing 10: Return value of a custom mec2 component.

```
return <Accordion>
    <AccordionSummary> {name} </AccordionSummary>
    <AccordionDetails>
        <Grid container direction="row">
            <MultiSelect options={head} updateOptions={updateHead} />
            <MecTable
                SanitizedCell={SanitizedCell}
                head={Object.entries(head).filter(h => h[1]).map(h => h[0])}
                list={mecElement._model[name]} />
        </Grid>
    </AccordionDetails>
</Accordion>
```

The Accordion in this listing is a Material-UI component[20]. This component accepts the AccordionSummary and AccordionDetails, which are defined inside the Accordion. In Listing 10 the AccordionSummary encapsulates the name property of the respective component, which is the name of the component in lower case; e.g. const name = 'nodes' for the Nodes component. AccordionDetails contains elements that are shown when the Accordion is expanded.

The AccordionDetails component shows details of the respective property of the mec2 model. The Grid in the AccordionDetails in 10 contains two custom components. MultiSelect is a dropdown menu which shows a list of the available properties of the module; e.g. for nodes these properties are id, x, y, base. These properties are provided by the variable head, which is given in the definition of the component. This enables the possibility to show and modify other properties later on, without the necessity to show them from the start[21]. The head variable is a list of properties which is further discussed in Chapter 3.4.

The MecTable component is responsible for rendering the table where the properties are shown. MecTable returns a predefined layout of a regular Table element of Material-UI, which can be reviewed at https://aka.klawr.de/sep#17. The TableHead in MecTable contains a mapping of the different properties which are provided by head to show the properties which are shown by each column. The rows of the table are filled

[20] https://aka.klawr.de/sep#16

[21] To show all possible properties directly would also result in a overcrowded user interface.

using yet another custom component: `SanitizedCell`. The `SanitizedCell` component is injected into the `MecTable`, wheras the definition of the component itself is provided by the encompassing component. It provides information on how the individual properties are to be rendered. The respective return value of the `SanitizedCell` function can be seen in Listing 11.

Listing 11: Return statement of the `Node` components `SanitizedCell` function.

```
return <ContextMenu key={idx}>
    {select()}
    <MenuItem onClick={removeNode}>
        {`Remove node ${elm['id']}`}
    </MenuItem>
```

`select` returns the respective function based on a switch statement, which can be seen in Listing 12.

Listing 12: `select` function in `SanitizedCell` of the `Node` component.

```
function select() {
    switch (property) {
        case 'base':
            const [checked, changeChecked] = React.useState(!!elm[property]);
            return <Checkbox checked={checked} onChange={(e) => {
                    changeChecked(e.target.checked);
                    update(e.target.checked);}} />
        case 'x':
        case 'y':
            return <UpdateText
                title={property}
                value={Math.round(elm[property])}
                onSubmit={v => update(+v)} />
        case 'id': /*...*/
        default: return <div>{elm[property]}</div>
    }
}
```

`SanitizedCell` is called with an argument object containing `property`, `idx` and `elm`. The `property` corresponds to the current value of `head`, which is mapped in the `MecTable` to fill the `TableRow`[22]. Based on the `property` value the `select` function returns the component designed for the specific property. If the property is equal to `'base'` the return value is `Material-UIs` `Checkbox`. For textual inputs the custom component `UpdateText` is created. This component handles updates reliably to be translated to the `mec2` model and is updated when the `mec2` model is modified. It is therefore used by number value like `'x'` and `'y'` or textual values like `'id'`.

[22]Yet another property of `Table` which is responsible for the design of the table-rows.

Another custom component, which is used inside a `SanitizedCell` component is `RadioSelect`. This custom component contains a selection of items and identifies one as the selected item; e.g. constraints inside a `mec2` model reference two nodes, which are selected via their respective `id` property. This component can list all nodes by their `id` property and select one respectively.

3.2.3 The canvas

As mentioned above, the viewport is occupied by the `mec2` Custom HTML element. The prototype uses the same canvas the model is rendered in to to draw the inputs for inference. This approach is cumbersome because the corresponding event handlers had to be activated and deactivated manually, which is hard to maintain and is not very flexible or extensible.

In unison with the rest of the application, this canvas element is modeled into a custom component. The `DeepmechCanvas` component does not return a `React` component, but a HTML canvas, as demonstrated in Listing 13.

Listing 13: Return of the `DeepmechCanvas` component.

```
return <canvas
    id="{id}" className={classes.drawCanvas}
    width={globalThis.innerWidth} height={globalThis.innerHeight}
    ref={canvasRef} />
```

The `id` property is used to refer to this canvas from other components. The `DeepmechControl` component uses this `id` to be able to get access to the canvas-HTML element and forwards it to the `predict` function.

`width` and `height` are taken from the `globalThis`[23] property.

The `DeepmechCanvas` component has a `canvasRef` variable which is initially set to `const canvasRef = React.useRef(null)`. Every time the `DeepmechCanvas` component is rendered the `canvasRef` is updated by the `ref` property of the returned HTML element. See `https://aka.klawr.de/sep#19` for more information about `React.useRef`.

Because the reference is created after initialization, `React`s `useEffect`[24] has to be used. `DeepmechCanvas`' `useEffect` is shown in Listing 14.

Listing 14: The `useEffect` function in the `DeepmechCanvas` component.

```
React.useEffect(() => {
    const [nl, cl] = [mec.nodeLabels, mec.constraintLabels];
    dispatch(mecAction.toggleNodelabels(false));
```

[23]`https://aka.klawr.de/sep#18`
[24]`https://aka.klawr.de/sep#20`

```
    dispatch(mecAction.toggleConstraintlabels(false));
    dispatch(UiAction.right(false));
    const ctx = canvasRef.current.getContext('2d');
    return handleInteractor(ctx, deepmech.mode, placeholder, () => {
        dispatch(mecAction.toggleNodelabels(nl));
        dispatch(mecAction.toggleConstraintlabels(cl));
    });
}, [deepmech.mode]);
```

The second argument (`[deepmech.model]`) is an array that defines properties on which changes are monitored. Every time a change is registered, the function given as the first argument is issued.

The first argument given to this function is also called after the component is rendered the first time. This behavior is necessary because the canvas has to be initialized for the context of the canvas element to be set via `canvasRef.current.getContext('2d')`. The `dispatch` calls set certain values which will be further examined in Chapter 3.3. Returned is a `handleInteractor` function.

The `handleInteractor` function handles the interaction of the user with the canvas. It creates a new instance on the `canvasInteractor`[25] for the given `ctx`. As with `mec2`, the global `canvasInteractor` is used here.

For the selection of individual g2-elements a `selector` variable is. This is done via `g2.selector(interactor.evt)`. The respective event listeners are defined for five different events as described in Listing 15.

Listing 15: Definition of event listeners on the `canvasInteractor`.

```
const o = { tick, pointerdown, pointerup, drag, click: pointerup }
Object.entries(o).forEach(e => interactor.on(...e));
```

The event listeners have functions to handle events based on the current `mode` which is inserted by the function call[26].

For example the `pointerdown` function, as described in Listing 16 contains a switch-statement handling events when the `mode` is set to `'draw'` or `'delete'`.

Listing 16: `pointerdown` function in `handleInteractor`.

```
function pointerdown(e) {
    switch (mode) {
        case 'draw':
            // Set ply and add to command queue
            ply = {
                pts: [{ x: e.x, y: e.y }],
```

[25]The `canvasInteractor` is discussed in Chapter 2.4.3

[26]Every time the mode changes the function is called again, as described in the array which is given as the second argument to the `React.useEffect` function.

```
                lw: '2', ls: '#fff', lc: 'round', lj: 'round',
                get sh() { return this.state & g2.OVER ? plyShadow : false; }
            };
            placeholder.ply.ply(ply);
            break;
        case 'delete':
            // Filter selected node from commands array
            placeholder.ply.commands = placeholder.ply.commands.filter(
                cmd => cmd.a !== selector.selection);
            selector.evt.hit = false; // selector gets confused
            selector.selection = false; // overwrite selection
            break;
    }
}
```

The `ply` variable is a placeholder filled by the various event handlers to represent the currently drawn `g2.ply` element. The `handleInteractor` function returns a function shown in Listing 17.

Listing 17: Return value of `handleInteractor`.

```
return () => {
    Object.entries(o).forEach(e => interactor.remove(...e));
    canvasInteractor.remove(interactor);

    fn();
}
```

This function is called by `useEffect` when the respective `DeepmechCanvas` is deleted[27]. It reverts the changes made to the `canvasInteractor` and also calls `fn` which is a function given as argument. `fn` is injected because it needs access to the scope in which the `dispatch` and `mecAction` resides, as seen in Listing 14.

3.3 State manipulation

To track changes inside components `React-Redux`[Abr21] is used. This approach is different to `mec2`'s because changes are not made on the object itself, but a new object is created on each change which replaces the old one.

An upside of this approach is that properties do not have to be tracked. A change will always render the whole application using the new state without the necessity of event listeners on every variable and property which may change. To dispatch changes the `dispatch` function is issued[28]. Those functions update a store that holds the state of the application. The store is defined by different slices. The `DeepmechSlice` is shown in Listing 18.

[27]Which is also done if it is rerendered.
[28]As already seen in Listings 9 and 14

Listing 18: Definition of the DeepmechSlice.

```
const slice = createSlice({
    name: 'Deepmech',
    initialState: {
        active: false,
        mode: 'draw',
        canvas: undefined,
        extern: {
            initiated: false,
            prediction: false,
            canvas: false,
            serverport: 0,
        }
    },
    reducers: {
        changeMode: (state, action) => {
            state.mode = action.payload;
        },
        updateCanvas: (state, action) => {
            state.canvas = action.payload;
        },
    },
});
export const deepmechAction = slice.actions;
export const deepmechSelect = state => state.Deepmech;
export default slice.reducer;
```

createSlice is imported from the reduxjs toolkit. Every slice contains three properties: A name for identification, an initialState which defines the properties that may change including their initial state and reducers.

reducers contain functions that take two arguments. The first argument is the state itself. The second argument is an action which contains information about the respective change in a payload property.

Every slice definition has three export statements. They are used to import the actions, select the slice, and to implement this slice into the store respectively[29].

To dispatch an action, the returned function of React-Redux' useDispatch is used. This returned function is commonly saved in a dispatch variable.

For example to update the mode of DeepmechSlice the DeepmechControl component issues dispatch(deepmechAction.changeMode(val)) when changed. Where the val is set to the respective string of the mode; e.g. "draw".

Another upside of this approach is that the corresponding properties do not have to be injected through the whole application. The parent of DeepmechControl is the LeftDrawer component which resides in the App component. The mode is used by the

[29]The store is defined in the store.js which definition can be seen here https://aka.klawr.de/sep#21.

DeepmechCanvas component which is used in the App component. Without a global state, the mode would have to reside in the App component and has to be injected through all children to be accessible where necessary. By using React-Redux' store, this is avoided by importing the necessary properties where they are needed, agnostic to the architecture of the application.

3.4 State of the mec2 model

The store contains three reducers. These are objects defined in their respective slices.

Deepmech is already discussed in Chapter 3.3. It handles the state of the canvas.

UI contains information about the state of the drawers (expanded or not), whether the application is in deepmech mode or not, if the dark-mode is activated or not, and about all properties of the mecElement which are shown in the right drawer.

The third reducer is the MecModel. It holds information about the state of the mec-model and induces functionality to change properties of the mecElement.

3.4.1 MecModelSlice

The mecElement references the global mec2 HTML Element. The _model property refers to the mec2 model rendered to the canvas of mecElement.

The initial state of the MecModel property is shown in Listing 19.

Listing 19: initialState property of MecModel.

```
initialState: {
    queue: [],
    selected: 0,
    id: ref._model.id,
    pausing: ref.pausing,
    gravity: ref.gravity,
    darkmode: window.matchMedia ?
        window.matchMedia('(prefers-color-scheme: dark)').matches ?
            true : false : false,
    nodeLabels: true,
    constraintLabels: true,
    grid: false,
}
```

Most of the properties handled by this state mirror the properties of the mecElement. They are changed using actions, which update the element and the corresponding property of the state accordingly. The implementation of respective actions is relatively straightforward. To change the value of gravity, the respective action toggleGravity is implemented as shown in Listing 20.

Listing 20: toggleGravity property in MecModels reducer

```
toggleGravity: (state) => {
    ref.gravity = !state.gravity;
    state.gravity = ref.gravity;
}
```

Where ref is a reference to mecElement.

The queue in Listing 19 is an array of objects, where each object implements an interface of four properties:

1. list — The list refers to the property in mecElement._model; e.g. list="nodes" refers to mecElement._model.nodes.
2. idx — Refers to the index of the respective element in said list. This value can also be either "add" or "remove", indicating that an element was added or removed to the respective list.
3. value — This property is an object that contains all the changes made to the element; e.g. {x: 20} to changes the x-coordinate of a node to 20.
4. previous — This object saves the value of properties before they are changed. It is used to revert actions.

These objects are given to MecModels add reducer, which appends it to the queue. By doing so, the selected value will be increased by one. The selected value is used as an index. This index selects the value of the element in the queue that is to be considered next. This is trivial if all that is done is adding values because selected will always correspond to the last entry in list, but this approach makes it possible to undo actions, which is discussed in Chapter 3.4.4.

3.4.2 Updating the mecElement

mecElement is not serializable by React-Redux, because the prototype is changed by design. Therefore another approach has to be used to propagate changes between the user interface and the mecElement.

The App.js contains the handleMecModelUpdate function. This function is handed to the store.subscribe function. By subscribing to the store this function is called every time the store receives an update[30].

[30]Which is every time a useDispatch function is issued.

The first thing the handleMecModelUpdate function checks is if the MecModel state has changed. If this is not the case the dispatch call was not issued to a MecModel reducer and the function returns immediately.

Otherwise the mecModel.selected is compared to a counter variable. The counter is holding the value of the mecModel.selected when handleMecModelUpdate was last issued. By being able to differentiate between the mecModel.selected value increased or decreased after the last call, the object in the mecModel.queue is using the value or the previous property. If the selected value went up is stored in the up variable. The usefulness of this will be examined in Chapter 3.4.4.

The list element that is currently selected will be used as action variable.

If the idx property of this object is of type "number" the action is interpreted to change the value of an existing element. The respective code is shown in Listing 21.

Listing 21: Updating an element in one of mecElement arrays.

```
Object.entries(up ? action.value : action.previous).forEach(e => {
    ref._model[action.list][action.idx][e[0]] = e[1];
});
```

In this piece of code every value inside ref._model that is referenced by action.value or action.previous[31] is replacing the current value of the element which resides in the respective list at the respective idx.

If the idx property is "add" or "remove" the approach is not as generalizable.

Listing 22: Adding or removing an element in mecElement.

```
const add = (up && action.idx === 'add') || (!up && action.idx === 'remove');
if (action.list === 'nodes' ||
    action.list === 'constraints' ||
    action.lists === 'views') {
    const element = { ...action.value };
    if (add) {
        if (ref._model[action.list].find(e => e.id === element.id)) return
        ref._model.plugIns[action.list].extend(element);
        ref._model.add(action.list, element);
        element.init(ref._model);
    }
    else {
        const o = ref._model[action.list].find(e =>
            e.id === element.id);
        if (o) o.remove;
    }
    ref._model.draw(mecElement._g);
}
```

[31]Which is selected by either going up or down the list.

At first, it has to be determined if the current action is to add, or to remove an element. This is necessary, because if `action.idx === 'add'`, but the current value of `up` is false[32], then the actual action to pursue is to remove an element.

The excerpt in Listing 22 shows the current implementation to add an element to `mecElement`[33]. If the `id` is already taken by an element of the same `list` the function is aborted to avoid further problems with ambiguities. Three functions have to be called to register an element to the `mec2` model. The `extend` function[34] is used to apply the respective prorotype to the `element`[35]. The next step is to add the element to the respective `list` in the `mecElement._model`. The third function initializes the element[36].

To remove an element the `remove` function on the element is called, which is provided by the given prototype.

Before the model can be rendered, another function has to be issued first. Because the `value` and `previous` properties of `actions` have to be serializable, properties that refer to other elements inside the model are addressed by their respective `id` and not by the referencing object which is used by `mec2` itself. The element replaces those referencing properties during initialization. If a constraint is changing its `p1` or `p2` property[37] this does not happen and subsequently results in an error. Because of this, the respective references have to be reassigned by a function given through the prototype: `reassignRefs`.

3.4.3 Implement bidirectionality

To ensure that if the user changes the location of nodes in the `mec2` HTML element by dragging, the corresponding change is also made to the respective value in the `Nodes` component. Herein dragging is defined by having a pointerdown event while the pointer is hovering over a node, then moving the pointer before a pointerup event is registered. The selected node follows the pointer respectively. To achieve this certain listeners are implemented.

The functionality to drag nodes is mirrored by the `Nodes` component by adding certain functions to the `mecElement._interactor`. For the respective `pointerdown`, `drag` and `pointerup` are added to the `Nodes` component.

[32]So the selection went a step backwards in the list
[33]The `mecElement` is referenced by `ref`.
[34]The `extend` function is expected to be provided by every `mec2` plugin.
[35]This makes the object also unserializable by `React-Redux`.
[36]For more informations about the inner workings of `mec2` visit `https://aka.klawr.de/sep#22`.
[37]The `p1` and `p2` properties of a constraint refer to the respectives nodes that are used as first and second anchor point.

Listing 23: Adding functions to mecElement._interactor to work bidirectional.

```
let previous;
let value;
let selection;
function deepmechNodeDown(e) {
    selection = mecElement._selector.selection;
    previous = { x: selection.x, y: selection.y };
}
function deepmechNodeDrag(e) {
    if (selection && selection.drag) {
        value = { x: selection.x, y: selection.y };
    }
}
function deepmechNodeUp() {
    if (!value) {
        previous = undefined;
        return;
    }
    dispatch(mecAction.add({
        list: name, idx: mecElement._model[name].indexOf(selection),
        value: { ...value }, previous: { ...previous }
    }));
}
const o = {
    pointerdown: deepmechNodeDown,
    drag: deepmechNodeDrag,
    pointerup: deepmechNodeUp
}
Object.entries(o).forEach(e => mecElement._interactor.on(...e));
```

The previous, value, selection variables are used to keep references consistend through the different events. The deepmechNodeDown function is used to determine the previous variable, which is later used to set the respective property for the action. The deepmechNodeDrag function updates the value property if necessary. The deepmechNodeUp function is used to dispatch the action. This is similar to the behavior expected when a node is edited through the user interface.

The whole operation is wrapped in a React.useEffect and returns a function which remove these function from the mecElement._interactor again. This ensures that these events are only present for the current Nodes component as already seen in Listing 17 for the DeepmechCanvas component.

3.4.4 Undo and redo

The approach of using an array containing objects with respective changes as value property and their value before the change as previous has the advantage of making movement in the list of actions possible. This movement is either executed by adding an action, which extends the list as discussed in previous chapters, or by two actions on the `MecModelSlice` that are named `undo` and `redo`. The `undo` action just decreases the value of `selected` by one[38]. The `redo` action increases the value of `selected` respectively[39].

This action also results in `handleMecModelUpdate` being triggered. As discussed in Chapter 3.4.2 the `up` in `handleMecModelUpdate` variable is determined by whether the value of `selected` went up or down since the last call of `handleMecModelUpdate`. When `undo` is triggered, this value went down, so `up` is set to false. Therefore the update described in Listing 21 will use the `action.previous` to update the `mecElement._model`. This will result in an undoing of the last submitted action and can be repeated as often as the queue of actions is long.

Likewise, the redo is an imitation of the action being issued the first time. The only difference is, that the value of `selected` does not correspond to the length of the queue.

If the current `selected` value does not correspond to the length of the queue[40], any change that is made by a new action will cut the queue at that point and then adding the newly added action to the queue. This ensures that after adding a new action to the queue it is also the latest change in the queue, but resulting in the possibility of going back to respective changes. This corresponds to the behavior of other programs that offer this kind of functionality.

4 Third party integrations

The web application has a lot of advantages. First and foremost it is mostly operating system agnostic since it is available in every modern internet browser and available as a progressive web app.

A major disadvantage is that it is not as performant as other environments. Especially the cropping of images for the preparation to the second model is taking a long time. This part of the process is avoidable, by moving the inference process to a better performing environment[41].

Other applications can take on tasks by registering themself to the web app. If an app is registered is captured in the `extern` property in the state of the `deepmech` slice, as shown in Listing 18. The `extern` property has four properties[42]. To register a third-party

[38]If `selected` is greater than 0.
[39]If `selected` is smaller than the lenght of the queue.
[40]So the current state of the `mecElement._model` does not represent the newest state.
[41]The browser implementation can be optimized of course, but local implementations should outperform browser implementations.
[42]Which is subject to change if the application is growing.

application, it is necessary to call the register action. The action.payload depends on the respective third-party application. At the moment three different third-party approaches are implemented.

4.1 WPF and WinUI

The Windows Presentation Foundation is a subsystem that can create user interfaces for windows applications. The application is implemented using C# and XAML. It offers a WebView2[43] control which embeds the web application into a native windows application.

The application is created by the template WPF Application of the Visual Studio Community Edition 2019 with the official .NET desktop development workload installed. This application template already impements everything necessary to compile an application. The MainWindow.xaml will be the main page of the application. The only window present in this application is the WebView2 component[44]

The WebView2 component itself is controlled by the DeepmechHandler property which is a DeepmechWebView object. To register the DeepmechHandler a Register function is defined, which can be seen in Listing 24.

Listing 24: DeepmechWebView Register function

```
private string WebviewPlaceholder(string message)
    => "globalThis.webviewEventListenerPlaceholder(" + message + ")";

public void Register(canvas, prediction)
{
    WebView.ExecuteScriptAsync(WebviewPlaceholder(
        "{register:{canvas: false, prediction: true}}"));
}
```

The WebviewPlaceholder is used every time a message is send to the PWA, hence it is implemented as function to wrap the message. The WebView is a property of the DeepmechWebView object and passed as argument constructor to be able to access the WebView2 component from within the class.

4.2 Interface to the PWA

The PWA has respective code in the App component to handle calls via an event listener. This will be only activated if the webview property of the chrome object is defined, as can

[43]There is a WebView component, too, but it uses the now deprecated Microsoft Edge rendering engine. The WebView2 component uses the Chromium engine.
[44]The content of the main window can be reviewed at https://aka.klawr.de/sep#23.

be seen in Listing 25. The `WebView2` component creates the `chrome.webview` property in the embedded web application.

Listing 25: PWA event-handler for the chrome webview

```
if (globalThis.chrome?.webview) {
  dispatch(deepmechAction.initiate());

  globalThis.webviewEventListenerPlaceholder = (o) => {
    if (o.register) {
      dispatch(
        deepmechAction.register({
          canvas: o.register.canvas,
          prediction: o.register.prediction,
        })
      );
    }
    if (o.prediction) {
      dispatch(deepmechAction.updateModel(o.prediction));
    }
  };
}
```

This event-handler calls the `updateModel` function if the `prediction` property is set on the transfered object.

The prediction is called from within the `predict` action of the `DeepmechSlice`. If the `extern.predict` property is set, the `predict` command will not issue a prediction in the web application itself, but will try to send messages to the first detected third-party implementation[45]. The message sent consists of two properties: `image` is a base64 representation of the canvas content, generated by `canvas.toDataURL().replace(/^data:image.+;base64,/,"")`. `nodes` contains the already existing nodes, with some processing to remove the cartesian modifier and remove the artificial 16 pixel offset created by the `updateNodes`[46]. The object is stringified by `JSON.stringify` and sent via the `postMessage` function of the `chrome.webview` object.

4.3 Interface of the WPF

The `WebView2` has an event-handler implemented to process messages that are sent via `chrome.webview.postMessage`. This event handler is defined as `ProcessWebMessage` and can be used to implement a listener for messages sent via the website embdedded in `WebView2`. The implementation of this event handler in the WPF can be seen in Listing 26.

[45] There should be at most one active third-party implementation, otherwise this acts as a priority list.
[46] See `https://aka.klawr.de/sep#24`.

Listing 26: Event handler of the WPF-application

```
public void ProcessWebMessage(
    object sender, CoreWebView2WebMessageReceivedEventArgs e)
{
    if (e.Source != WebView.Source.ToString()) //sender.Source.ToString()
    {
        return;
    }

    var message = JsonSerializer.Deserialize<DeepmechMessage>
        (e.TryGetWebMessageAsString());

    if (message == null) return;
    if (message.ready)
    {
        Register(canvas: false, prediction: true);
    }
    if (message.image != null)
    {
        Predict(Deepmech_cxx.Predict(
            Deepmech_ctx, message.image, message.nodes));
    }
}
```

Here it can be seen that `Register` from Listing 24 is called when the PWA sends a message declaring it being ready for third-party applications to register. The PWA sends this message by calling `deepmechAction.initate` which can be seen in Listing 25.

If the `message` contains an `image` property[47] the `Predict` function is called. The `Predict` function is very similar to the `Register` function, with the difference of having a `prediction` property in the message, instead of `register`. The content of the `prediction` property is defined by the return value of `Deepmech_cxx`'s `Predict` function.

4.4 Predictions in C++

There are implementations of TensorFlow in .NET[48], but for the purpose of this implementation I decided to use an implementation in C++[49]. Unfortunately, at the time of writing, the official TensorFlow API can not be used under the Windows operating System[50], which is mandatory for `WPF` and `WinUI` applications. As a substitution for this the `frugally-deep` library is used [Her21].

The created library exposes three functions:

[47] Which is the case when a prediction is requested

[48] Examples being `https://aka.klawr.de/sep#25` and `https://aka.klawr.de/sep#26`

[49] The respective code can be reviewed at `https://aka.klawr.de/sep#27`.

[50] The discussion regarding the issue can be read here `https://aka.klawr.de/sep#28`.

1. create_deepmech_ctx expects two c-style strings. The PWA uses tensorflow.js to load the model, as described in Chapter 2.3. For the implementation using frugally-deep a similar conversion is used, which is described at https://aka.klawr.de/sep#29. After the conversion into a usable JSON file the contents can be passed to create_deepmech_model as c-style strings.

2. predict expects a pointer to a Deepmech_ctx instance, an image and its dimensions and a c-style string containing the known nodes in JSON format. The first parameter can be obtained by calling create_deepmech_ctx. The image is an integer array containing normalized values between 0 and 255 for the gray-scale information. Subsequentially width multiplied by height have to be equal to the length of this integer array. These variables are used by OpenCV to load the image.

3. The application has no way of knowing if the return value of the predict function is longer needed. To compensate deepmech_cxx_free can be called to free the occupied memory.

4. destroy_deepmech_ctx has the same function as deepmech_cxx_free, but refers to the Deepmech_ctx object itself, which is called when an DeepmechWebView is disposed.

The compiled deepmech_cxx.dll is placed into the respective directory for the functions to be usable by the WPF-application. The WPf-application has a static class dedicated to communicate with this library using imports as can be seen in Listing 27.

Listing 27: DllImports in the static Deepmech_cxx class

```
[DllImport("deepmech_cxx")]
public static extern IntPtr create_deepmech_ctx(
    [MarshalAs(UnmanagedType.LPStr)] string symbolModel,
    [MarshalAs(UnmanagedType.LPStr)] string cropModel);

[DllImport("deepmech_cxx")]
private extern static IntPtr predict(
    IntPtr deepmech_ctx, byte[] imageData, uint width, uint height,
    [MarshalAs(UnmanagedType.LPStr)] string nodes);

[DllImport("deepmech_cxx")]
private extern static void deepmech_cxx_free(IntPtr str);

[DllImport("deepmech_cxx")]
public extern static void destroy_deepmech_ctx(IntPtr ctx);
```

These functions are called exclusively by the DeepmechWebView object; e.g. it creates the Deepmech_ctx on construction. The respective JSON-strings containing the inference models are imported and read from the `Properties.Resources` namespace, used by WPF-applications to import external resources.

When the `Deepmech_cxx.Predict` function is called as in Listing 26 the pointer to the `Deepmech_ctx`, the image and known nodes are submitted.

The base64 string is converted to an image using the `SixLabors.ImageSharp` library [Lab21], as seen in Listing 28.

Listing 28: Deepmech_cxx overload for image conversion.

```
public static string Predict(
    IntPtr deepmech_ctx, string base64, string nodes = null)
    => Predict(deepmech_ctx,
               Image.Load<A8>(Convert.FromBase64String(base64)),
               nodes);
```

The overloaded `Predict` function in `Deepmech_cxx` calls the imported `predict` function and returns the resulting c-style string. The return value is already in JSON-format and converted to a C# string by `Marshal.PtrToStringUTF8(resultNativeString)`. This is subsequentially returned and then sent to the PWA, as seen in Listing 26.

The known nodes are included in the inference, by transferring the known nodes as an integer array through this pipeline. This is done to use them when creating the crops for the constraint detection.

The C++ implementation itself is a rewrite of the Python implementation similar to the conversion in Chapter 2.3. It can be reviewed at `https://aka.klawr.de/sep#30`.

4.5 Implementation in WinUI

The WPF-application is working as a functional prototype, but unfortunately the referenced `Microsoft.Toolkit.Wpf.UI.Controls`[51] extension is no longer supported in `.NET 5`[52]. Because of this an older version of the extension is used already[53].

Fortunately Microsoft offers other frameworks to publish applications. The development of `WinUI 3` applications is very similar to WPF[54].

With the exception of some namespaces the source code of the WinUI implementation is the same, which can be seen here `https://aka.klawr.de/sep#33`

[51] See the repository here: `https://aka.klawr.de/sep#31`.

[52] The respective discussion on the issue can be reviewed here `https://aka.klawr.de/sep#32`.

[53] The changes occured while working on the project.

[54] I am sure more sophisticated developers knowing the intricate differences may disagree, but as most of the code can be reused they seem to be similar from my perspective

4.6 Webserver implementation

Another way of getting better performance for the inference is to redirect the process to a webserver. To register a local webserver, the `register` action of the `DeepmechSlice` reducer in the PWA can be issued. It sets the `serverport` and the `prediction` properties of the `state.extern.prediction`.

When a prediction is issued a HTTP request is sent to the webserver. The request `message` contains the same properties as the messages discussed in Chapter 4.2. The respective fetch request and subsequent response handling can be seen in Listing 29.

Listing 29: Communication of PWA via HTTP.

```
fetch("http://localhost:" + port, {
    method: "POST",
    body: JSON.stringify(message),
    headers: new Headers({
        "Content-Type": "application/json",
    }),
}).then((r) => {
    const reader = r.body.getReader();
    reader.read().then(({ done, value }) => {
        updateModel(JSON.parse(String.fromCharCode(...value)));
    });
});
```

To register the webserver a new button is implemented to the `LeftDrawer` component, which will dispatch the respective `register` action.

The prediction functionality of the webserver is using the already existing Python implementation, introduced in Chapter 2.2.5 [55]. The predict function is slightly modified to be able to handle the base64-encoded images provided by the PWA. The function also expects a JSON-string containing information about the known nodes to be able to respect those in the constraint detection.

The webserver is implemented as a `DeepmechPredictionServer` class, which inherits from the `BaseHTTPRequestHandler` imported from the `http.server` package. This class provides functions like `do_OPTIONS`, `do_GET` and `do_PUSH`, corresponding to the standard HTTP requests `OPTIONS`, `GET` and `PUSH`. `do_Get` is used to show some text about this server to a user which is using the browser to get information about the server. `do_Options` and `do_Post` provide the necessary headers to allow cross origin communcation[56].

The server converts the results of the prediction into JSON-format and returns the response to the PWA, which updates the model accordingly as seen in Listing 29.

[55]The corresponding script can be seen at `https://aka.klawr.de/sep#34`.
[56]For more information about `CORS` see `https://aka.klawr.de/sep#35`.

Table 2: Time before and after calling the `predict` action in `DeepmechSlice`. Two predictions are made, because some implementations need extra time for initialization; e.g. JavaScript implementation compiles the code just in time, increasing performance in subsequent runs.

Implementation:	tf.js (PWA)	frugally-deep (WPF)	tf (Python-Webserver)
First prediction:	1895ms	18ms	53ms
Second prediction:	345ms	18ms	22ms

5 Conclusion

The process of predicting constraints using the proposed approach needs a lot more processing than the detection of symbols. Where symbols are still limited to be 32x32 sized images the span of crops can not be limited without losing their intended use. Furthermore constraints are not just a classification problem, but the direction of the constraint has to be identified, too, which doubles the effort.

For inference other approaches exist; e.g. `YOLO` [BWL20], but their exploration would be outside of the scope of this project.

The implementation of `mec2` into an application which is able to run on various platforms that can render web content and can be extended was investigated thoroughly during this project. The system agnostic frontend using the `mec2` HTML element extended by `React` are flexible enough to implement backends fitting for various needs different platforms may have. In this project the implementation into desktop frameworks via `WPF` and `WinUI` are examined. The possibility to outsource the inference to a webserver is also introduced, providing yet another way of distributing the wordloads. Table **??** shows that delegating the inference to a different backend seems to be worth it, considering other tasks as video processing may be desired in the future, making performance a bigger issue.

The overhead created by implementations into other environments is little, if the PWA can be implemented using web-view components. These are available in most modern frameworks. This also makes it possible to bring this functionality to mobile applications; e.g. using the Qt-Framework of Windows Xamarin, but as this project focusses on desktop applications this is not examined further.

The results of this case study are promising, as extending the PWA itself and implementing it seems to be feasible without an overhead to big. The PWA can be accessed via `https://deepmech.klawr.de`, the other implementations are open source and licensed unter the MIT license, so they can be built without a permission.

List of Figures

1 Examples of constraint detector training data 5
2 Randomly placed nodes on blank image 6
3 Randomly connected nodes using constraint data 7
4 TensorBoard output for the crop detector 8
5 Predictions of the first prototype on the generated data 9
6 Example of the mec2 HTML element 12
7 Image generated by Listing 7. 14
8 deepmech prototype. 16
9 Screenshot of the web application 18

Listings

1 Summary of Symbol Classifier. 2
2 Transformation of the Symbol Classifier into a FCN. 3
3 Summary of Symbol Classifier transformed into a FCN. 4
4 Summary of Constraint Detector. 7
5 JavaScript classes for Tensorflow.js. 9
6 Example for a mechanism defined in the syntax proposed by `mec2`. 12
7 Example code of a truss defined with g2. 14
8 Command queue for the drawing context in the prototype. 15
9 Definition of the `MecControl` component. 18
10 Return value of a custom `mec2` component. 20
11 Return statement of the `Node` components `SanitizedCell` function. . . . 21
12 `select` function in `SanitizedCell` of the `Node` component. 21
13 Return of the `DeepmechCanvas` component. 22
14 The `useEffect` function in the `DeepmechCanvas` component. 22
15 Definition of event listeners on the `canvasInteractor`. 23
16 `pointerdown` function in `handleInteractor`. 23
17 Return value of `handleInteractor`. 24
18 Definition of the `DeepmechSlice`. 25
19 `initialState` property of `MecModel`. 26
20 `toggleGravity` property in `MecModel`s `reducer` 27
21 Updating an element in one of `mecElement` arrays. 28
22 Adding or removing an element in `mecElement`. 28

23 Adding functions to `mecElement._interactor` to work bidirectional. . . . 29
24 DeepmechWebView Register function 32
25 PWA event-handler for the chrome webview 33
26 Event handler of the WPF-application 34
27 DllImports in the static Deepmech_cxx class 35
28 Deepmech_cxx overload for image conversion. 36
29 Communication of PWA via HTTP. 37

Link Index - `https://aka.klawr.de/sep`

#1: `https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps` 1
#2: `https://github.com/klawr/deepmech/blob/master/reports/sep/notebooks/2.1.1-scanning_an_image.ipynb` 3
#3: `https://github.com/klawr/deepmech/blob/master/reports/sep/notebooks/2.1.2-fcn.ipynb` 3
#6: `https://github.com/klawr/deepmech/blob/master/reports/sep/notebooks/2.2.1-constraint_data_gen.ipynb` 5
#4: `https://drive.google.com/file/d/1Xff-qFCLYnNqsBIPKw7WiMMz3qa8VIfy/view` 5
#5: `https://drive.google.com/file/d/18uzwfqM4fhfdAQDBd5dvT6k-a-pUWvHR` 5
#7: `https://github.com/klawr/deepmech/blob/master/reports/sep/notebooks/2.2.2-connecting_nodes.ipynb` 7
#8: `https://drive.google.com/file/d/1GkEI71ogtMRSQKdO3OJ1DVNmVJrSVZRN/view` 7
#9: `https://keras.io/guides/functional_api/` 7
#10: `https://github.com/klawr/deepmech/blob/master/reports/sep/notebooks/2.2.5-combining_node_constraints.ipynb` 9
#11: `https://www.tensorflow.org/js/tutorials/conversion/import_keras` 9
#12: `https://raw.githubusercontent.com/klawr/deepmech/master/reports/sep/code/crop.js` 10
#13: `https://github.com/klawr/deepmech/blob/master/reports/sep/notebooks/2.3.1-nodejs_implementation.ipynb` 11
#14: `https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_custom_elements` 13
#15: `https://github.com/klawr/deepmech/blob/gh-pages/src/Components/Utils/ListButton.js` 19
#17: `https://material-ui.com/components/tables/` 20
#16: `https://material-ui.com/components/accordion/` 20
#19: `https://reactjs.org/docs/hooks-reference.html#useref` 22
#18: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/globalThis` 22
#20: `https://reactjs.org/docs/hooks-reference.html#useeffect` 22
#21: `https://github.com/klawr/deepmech/blob/gh-pages/src/Features/store.js` 25
#22: `https://goessner.github.io/mec2/` 29
#23: `https://github.com/klawr/deepmech/blob/master/src/wpf/deepmech/MainWindow.xaml` 32
#24: `https://github.com/klawr/deepmech/blob/gh-pages/src/deepmech/deepmech.js#L277` 33
#25: `https://github.com/SciSharp/TensorFlow.NET` 34
#26: `https://github.com/migueldeicaza/TensorFlowSharp` 34

#27: https://github.com/klawr/deepmech/blob/master/src/cxx/src/deepmech.cpp 34
#28: https://github.com/tensorflow/tensorflow/issues/41904 34
#29: https://github.com/Dobiasd/frugally-deep#usage 35
#30: https://github.com/klawr/deepmech/blob/master/src/cxx/src/deepmech.cpp 36
#33: https://github.com/klawr/deepmech/tree/master/src/winui 36
#31: https://github.com/windows-toolkit/WindowsCommunityToolkit 36
#32: https://github.com/windows-toolkit/Microsoft.Toolkit.Win32/issues/296 36
#34: https://github.com/klawr/deepmech/blob/master/src/webserver/predict.py 37
#35: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS 37

References

[Abr21] Dan Abramov. *React-Redux.* Jan. 8, 2021. URL: https://react-redux.js.org/.

[Bab21] Babel. *babeljs.* Feb. 21, 2021. URL: https://babeljs.io/.

[BWL20] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. “YOLOv4: Optimal Speed and Accuracy of Object Detection”. In: (Apr. 23, 2020). arXiv: 2004.10934 [cs.CV].

[Cho19] François Chollet. *Keras.* 2019. URL: https://keras.io/ (visited on 10/09/2019).

[Fac21] Facebook. *React.js.* Feb. 21, 2021. URL: https://reactjs.org/.

[Goo21a] Google. *TensorBoard.* Feb. 20, 2021. URL: https://www.tensorflow.org/tensorboard/.

[Goo21b] Google. *Tensorflow.js.* Feb. 20, 2021. URL: https://www.tensorflow.org/js.

[Gös19a] Stefan Gössner. “Ebene Mechanismenmodelle als Partikelsysteme - ein neuer Ansatz”. In: *13. Kolloquium-Getriebetechnik - Tagungsband.* Ed. by Burkhard Corves, Philippe Wenger, and Mathias Hüsing. Cham: Springer International Publishing, 2019, pp. 169–180. ISBN: 978-3-8325-4979-4.

[Gös19b] Stefan Gössner. “Fundamentals for Web-Based Analysis and Simulation of Planar Mechanisms”. In: *EuCoMeS 2018.* Ed. by Burkhard Corves, Philippe Wenger, and Mathias Hüsing. Cham: Springer International Publishing, 2019, pp. 215–222. ISBN: 978-3-319-98020-1.

[Gös21a] Stefan Gössner. *g2.* Feb. 21, 2021. URL: https://github.com/goessner/g2.

[Gös21b] Stefan Gössner. *mec2*. Feb. 21, 2021. URL: https://github.com/goessner/mec2.

[Her21] Tobias Hermann. *frugally-deep*. Feb. 21, 2021. URL: https://github.com/Dobiasd/frugally-deep.

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

[Joy21] Joyent. *Node.js*. Feb. 21, 2021. URL: https://nodejs.org/.

[Lab21] Six Labors. *ImageSharp*. Feb. 21, 2021. URL: https://sixlabors.com/products/imagesharp/.

[Law20] Kai Lawrence. *Student Research Project*. Research rep. 2020. URL: https://github.com/klawr/deepmech/tree/master/reports/srp.

[LSD14] Jonathan Long, Evan Shelhamer, and Trevor Darrell. "Fully Convolutional Networks for Semantic Segmentation". In: (Nov. 14, 2014). arXiv: 1411.4038 [cs.CV].

[Mat20] Material-UI. *Material-UI*. Dec. 15, 2020. URL: https://material-ui.com/.

[Mic21a] Microsoft. *Windows Presentation Foundation*. Feb. 19, 2021. URL: https://docs.microsoft.com/en-us/dotnet/desktop/wpf/?view=netframeworkdesktop-4.8.

[Mic21b] Microsoft. *Windows UI Library*. Feb. 19, 2021. URL: https://docs.microsoft.com/en-us/windows/apps/winui/winui3/.

[pan21] pandas. *pandas*. Feb. 20, 2021. URL: https://pandas.pydata.org/.