VPRCS Document

View and Navigate Canvas

- **Pan (Move around the Canvas)**
  - **Action**: Navigate the canvas.
  - **How-to**: Right-click, then drag in the desired direction.
- **Zoom In/Out**
  - **Action**: Adjust the canvas magnification.
  - **How-to**:
    - **Mouse**: Scroll forward to zoom in, and backward to zoom out.
    - **Tool Panel**: Click the zoom tool in the bottom-left control panel.
- **Fit to Screen**
  - **Action**: Display the entire canvas within the visible workspace.
  - **How-to**: Click the fit-to-screen tool in the bottom-left control panel.
- **Center the Selection**
  - **Action**: Position selected nodes at the workspace's center.
  - **How-to**: Press `F`.

Node Operations

- **Add Node**
  - **Context Menu**
    - **Action**: Insert a new node using the context menu.
    - **How-to**: Right-click on the canvas background to open the menu, then select the desired node type from the menu items to add.
  - **Port-to-Background**
    - **Action**: Extend from an existing port, add a new node, and connect them.
    - **How-to**: Left-click and drag from a node's port, and release on the canvas background. When the menu appears, select the desired node type. The new node's corresponding port will automatically connect to the initial node's port from which you dragged.

- **Comment Node**
    - **Add Comment**
        - **Action**: Add a comment on the canvas.
        - **How-to**: Press `C`.
    - **Comment on Selected Node(s)**
        - **Action**: Add a comment for a selected node or nodes.
        - **How-to**: Select the node or nodes you wish to comment on, then press `C`.
- **Select Node(s)**
    - **Single Node Selection**
        - **Action**: Select an individual node.
        - **How-to**: Left-click the node or press `Enter`.
    - **Multiple Node Selection**
        - **Action**: Select several nodes.
        - **How-to**: Hold `Shift`, then left-click on each desired node.
    - **Marquee Selection**
        - **Action**: Select nodes within a defined area.
        - **How-to**: Left-click and drag to create a selection box around the nodes.
    - **Select All**
        - **Action**: Choose all nodes on the canvas.
        - **How-to**: Press `Ctrl` + `A`.
    - **Deselect All**
        - **Action**: Clear all node selections.
        - **How-to**: Click on the background or press `Escape`.
- **Move Node(s)**
    - **Drag to Move**
        - **Action**: Reposition a node manually.
        - **How-to**: Select the node, then left-click and drag it to the desired location.
    - **Arrow Key Movement**
        - **Action**: Nudge a node in specific directions.
        - **How-to**: Select the node and use the arrow keys (`↑`, `↓`, `←`, `→`) to move it up, down, left, or right.
- **Connect Nodes**
    - **Action**: Create a link between two nodes.
    - **How-to**: Left-click on a port of one node and drag the connection to a port on another node, then release.
- **Copy Node**
    - **Action**: Create a duplicate reference of a selected node for pasting.
    - **How-to**:
        - **Context Menu**: Right-click the node and select "Copy".
        - **Shortcut**: Press `Ctrl` + `C`.

- **Paste Node**
  - **Action**: Place the copied node onto the canvas.
  - **How-to**:
    - **Context Menu**: Right-click on the canvas background and select "Paste".
    - **Shortcut**: Press `Ctrl` + `V`.
- **Duplicate Node**
  - **Action**: Instantly create an identical copy of a selected node on the canvas.
  - **How-to**:
    - **Context Menu**: Right-click the node and select "Duplicate".
    - **Shortcut**: Press `Ctrl` + `D.
- **Delete Node**
  - **Action**: Remove a selected node from the canvas.
  - **How-to**:
    - **Context Menu**: Right-click the node and select "Delete".
    - **Shortcut**: Press the `Delete` key.
- **Delete Elements**
  - **Node Deletion**
    - **Action**: Remove a selected node from the canvas.
    - **How-to**: Select the node and press the `Delete` key.
  - **Edge Deletion**
    - **Action**: Remove a connection or edge between two nodes.
    - **How-to**: Hold `Ctrl`, then double-left-click on the edge you wish to delete.

Function

- **New Function via Context Menu**
  - **Action**: Create a new function on the canvas.
  - **How-to**: Right-click on the canvas background to open the context menu, then select the "Create Function" menu item.
- **Call Function via Context Menu**
  - **Action**: Invoke a previously created function on the canvas.
  - **How-to**: Right-click on the canvas background to open the context menu. Navigate to the "Functions" category and select the desired function to call it.

Variable

- **New Variable via Context Menu**
  - **Action**: Create a new variable on the canvas.
  - **How-to**: Right-click on the canvas background to open the context menu. Then, select the "Create Variable" menu item.
- **Get/Set Variable via Context Menu**

    - **Action**: Access or modify a variable on the canvas.
    - **How-to**: Right-click on the canvas background to open the context menu. Navigate to the "Variable" category and select the desired variable. From there, choose either "Set" to modify the variable or "Get" to access its value.

Auto Arrange

Employ automated layout adjustments for optimal node alignment and view fitting.

- **Using the Auto-Arrange Tool**
    - **Action**: Automatically organize the nodes on the canvas for a clearer view.
    - **How-to**: Click on the "Auto-Arrange" tool located in the bottom left panel.
- **Auto-Arrange via Context Menu**
    - **Action**: Arrange nodes on the canvas via the context menu.
    - **How-to**: Right-click on the canvas background to open the context menu and select "Auto Arrange"

JSON-based specification for Node Library

We've established a JSON-based configuration specifically crafted to define a node library within our node-based programming interface. The configuration encompasses several pivotal components:

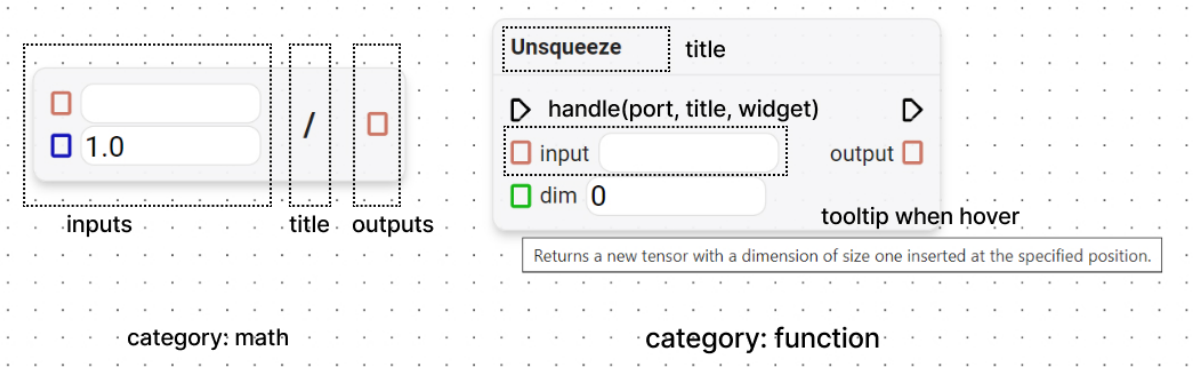
- **description**: Provides a succinct overview of the library's purpose or functionality.
- **enable**: A boolean value dictating the activation status of the node library.
- **nodes**: An object where:
    - The **key** represents the type of the node.
    - The **value** is a node object that comprises:
        - **type**: Designates the kind of node.
        - **category**: Categorized either as 'math' or 'function'.
        - **title**: A brief descriptor or name for the node.
        - **tooltip**: Offers a more detailed explanation or hint about the node's operation.
        - **externalImports**: Lists import dependencies necessary for executing the node's functions.
        - **sourceCode**: Utilizes the Mustache template system for rendering. The template includes placeholders for:
            - **inputs**: A list of input values specific to the node.
            - **outputs**: A list of output variable names.
            - **indent**: To maintain proper indentation, consistent with Python's code formatting, each new line is appropriately indented.

          If the **category** is 'function', the execution sequence of subsequent nodes is considered. This is typically appended to the end of the source code like `(\n{{{outputs.0}}})`.
        - **inputs** and **outputs**: Objects detailing the interactions of the node. Within these:
            - Each **key** is **unique** and corresponds to a handle object.

        - The handle object incorporates:
            - **dataType**: Specifies the kind of data the handle deals with.
            - **defaultValue**: The initial or default value, if any.
            - **tooltip**: A brief hint or description.
            - **title**: The handle's label or name.
            - **showWidget**: Dictates the visibility of a related UI widget.
            - **showTitle**: Governs the display of the handle's title.
- **types**: An object detailing the unique data types used within this library. For each type:
    - The **key** signifies the unique data type identifier.
    - The **value** provides a comprehensive type definition that might include:
        - **defaultValue**: A default value, if applicable.
        - **shownInColor**: The designated display color for the data type.
        - **widget**: An optional UI component associated with the data type. Possible values include: `IntegerInput`, `NumberInput`, `TextInput`, `BooleanInput`, and `EnumSelect`.
        - **options**: Additional configurable settings for the data type.

With this structured approach, our goal is to foster a clear, transparent, and adaptable interface for node-based programming.

Example:



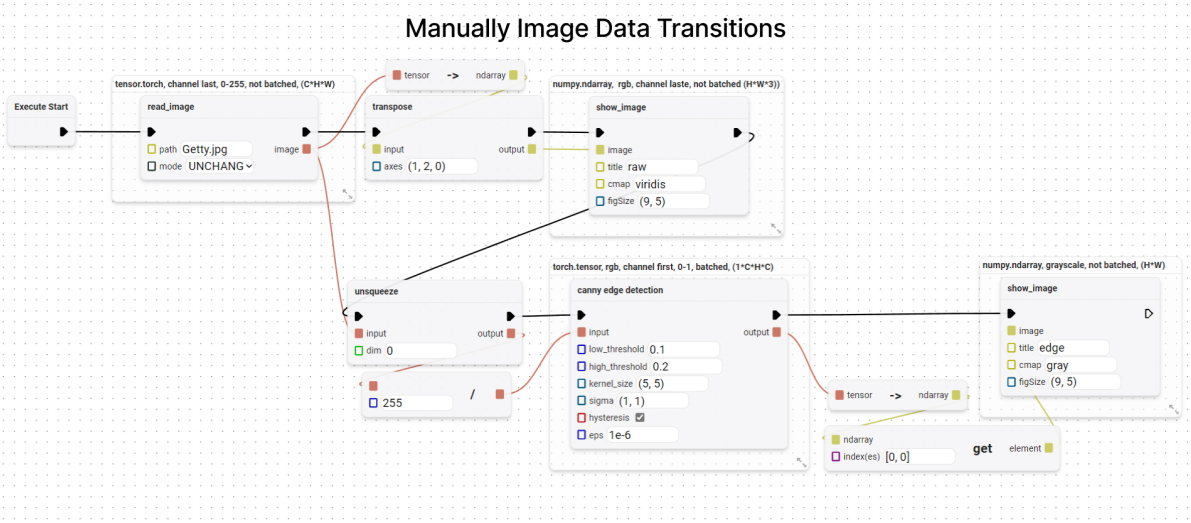

```
{
  "description": "The torch tensor type definition",
  "enable": true,
  "nodes": {
    "tensor / float": {
      "type": "tensor / float",
      "category": "math",
      "title": "/",
      "tooltip": "Tensor Divide by a scalar",
      "inputs": {
        "tensor": {
          "title": "tensor",
          "dataType": "torch.tensor"
        },
        "float": {
          "title": "float",
          "dataType": "float",
```

```
          "defaultValue": "1.0"
        }
      },
      "outputs": {
        "output": {
          "title": "tensor",
          "dataType": "torch.tensor"
        }
      },
      "sourceCode": "{{indent}}{{{outputs.0}}} = {{{inputs.0}}}/{{{inputs.1}}}",
    },
    "unsqueeze": {
      "type": "unsqueeze",
      "category": "function",
      "title": "unsqueeze",
      "tooltip": "Returns a new tensor with a dimension of size one inserted at
the specified position.",
      "inputs": {
        "execIn": {
          "title": "execIn",
          "dataType": "exec",
          "showWidget": false,
          "showTitle": false
        },
        "input": {
          "title": "input",
          "dataType": "torch.tensor"
        },
        "dim": {
          "title": "dim",
          "dataType": "integer",
          "defaultValue": "0"
        }
      },
      "outputs": {
        "execOut": {
          "title": "execOut",
          "dataType": "exec",
          "showWidget": false,
          "showTitle": false
        },
        "output": {
          "title": "output",
          "dataType": "torch.tensor"
        }
      },
      "externalImports": "import torch",
      "sourceCode": "{{indent}}{{{outputs.1}}} = torch.unsqueeze({{{inputs.1}}},
{{{inputs.2}}})\n{{{outputs.0}}}",
    }
  },
  "types": {
    "tensor": {
      "default": "",
      "widget": "TextInput",
      "shownInColor": "hsl(10, 50%, 60%)"
```

```
    }
  },
}
```

Automatic Image Data Transitions via a Configurable State Machine

Different software libraries and tools (such as TensorFlow, PyTorch, and OpenCV) have their distinct ways of representing and working with images. By having a unified image representation, you allow seamless interaction between these tools. We've developed a unified, abstract representation for image data, paired with a state machine model. This system facilitates automatic data type conversion based on predefined transition rules, streamlining the process and eliminating the need for cumbersome manual conversions.



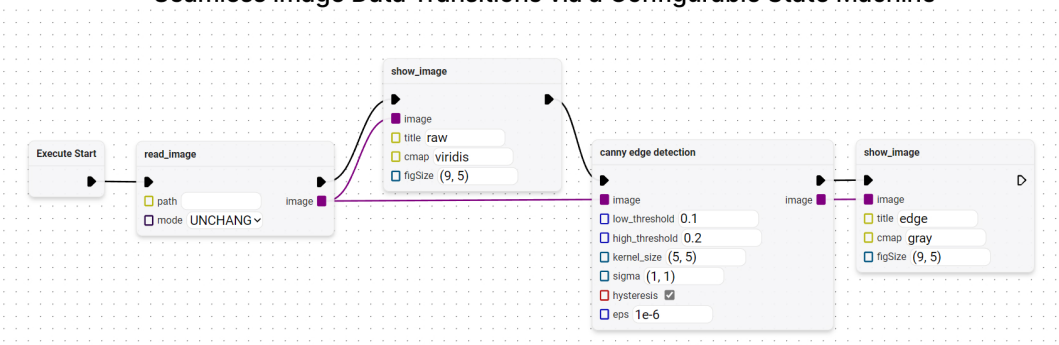


Image Representation

```
interface ImageMetadata {
  colorSpace: 'rgb' | 'gbr' | 'grayscale';
  channelOrder: 'none' | 'first' | 'last';
  isBatched: boolean;
  intensityRange: '0-255' | '0-1';
  device: 'cpu' | 'gpu';
}

export interface ISourceImage {
  dataType: string;
  value: any;
  metadata: ImageMetadata;
}
```

```
export interface ITargetImage {
  dataType: string;
  value: any;
  metadata: ImageMetadata[];
}

export type IImage = ISourceImage | ITargetImage;
```

How-to

In the output handle, define the default value for that specific handle and then return the image representation in the source code. Here are some examples:

```
{
    "image": {
          "title": "image",
          "dataType": "image",
          "defaultValue": {
            "dataType": "torch.tensor"
          },
    }
}
```

```
{
"sourceCode": "{{indent}}{{{outputs.1}}} = io.read_image({{{inputs.1}}},
{{{inputs.2}}})\n{{indent}}{{{outputs.1}}} = {'value': {{{outputs.1}}},
'dataType': 'torch.tensor', 'metadata': {'colorSpace': 'rgb', 'channelOrder':
'first', 'isBatched': False, 'intensityRange': '0-255', 'device':
'cpu'}}\n{{{outputs.0}}}"
...
}
```

In the output handle, define the default value for that specific handle.

```
{
    "image": {
      "title": "image",
      "dataType": "image",
      "defaultValue": {
        "dataType": "numpy.ndarray",
        "metadata": [
          {
            "colorSpace": "rgb",
            "isBatched": false,
            "channelOrder": "last",
            "intensityRange": "0-255"
          },
          {
            "colorSpace": "grayscale",
            "isBatched": false,
            "channelOrder": "none",
            "intensityRange": "0-255"
```

```
          }
        ]
      },
 }
```

The conversion rule can be configured using a JSON file. For instance, here's how you can define a conversion from `torch.tensor` to `numpy.ndarray`:

```
{
  "imageTypeConversion": {
    "torch.tensor": {
      "numpy.ndarray": {
        "function_definition": "def tensor2ndarray(src_image):\n  import copy\n
numpy_image = copy.deepcopy(src_image['value'].cpu().numpy())\n  return {\n
'dataType': 'numpy.ndarray',\n       'value': numpy_image,\n       'metadata':
src_image['metadata']\n  }",
        "function_name": "tensor2ndarray"
      }
    }
  }
}
```

For more conversion rules please check [Automatic_Image_Data_Transitions_via_a_Configurable_State_Machine.ipynb]