FlowApp — User Manual

`version` 1.0

Vojtěch Tomas

Contents

1 Installation

There are two possible ways to install the application. The first is suitable for local use and development. The second is more suitable when a robust solution is required. Both of the methods require creating Python Virtualenv, installing dependencies, and building the application.

> There is a standrad `makefile` for Linux (command `make`) and Mac (command `make mac`). Please run the installation command for your platform, it should do all the steps of the local developer installation for you. Please pay attention, your input is required from time to time. Other makefile commands can be found in table 1.
>
> 1. navigate to the main `makefile`
> 2. run `make` or `make mac` to build the application
> 3. run `make run` to launch the application
> 4. open a new terminal
> 5. navigate to the main `makefile`
> 6. run `make backend` to launch the computation backend
> 7. in your browser, go to `localhost:9000` and start working

Please make sure you have at least Python 3.6 installed. The application will likely work with any 3.* installation of Python.[1] The following guide is for Linux only; however, the setup on other OS' should be similar.

1.1 Local Developer Installation

Navigate into the local copy of the folder containing the application. The folder should contain the following structure:

```
src
├── flow
│   ├── build.sh
│   └── jsbuild.sh
├── sage
├── makefile
└── requirements.txt
```

Create a Python environment and launch it. You can use Conda or Virtualenv. The use of Virtualenv is recommended. If you do not have Virtualenv installed, run the following command:

```
sudo apt install virtualenv
```

[1] All tested versions were 3.6 or newer.

command	description
`make`	build for Linux
`make mac`	build for Mac
`make run`	run the application
`make backend`	run the backend
`make rebuild`	Linux and Mac rebuild C and JS
`make clean`	delete built files
`make cleanall`	delete all built files, including added static and user media (accounts, etc.)

Table 1: Makefile commands

The following code creates the virtual env:

```
virtualenv --python=python3 env
```

Now activate the environment with the first line of the following code and then install all dependencies. All Python dependencies are listed in the requirements.txt file. You can install them with the second line of code.

```
. ./env/bin/activate                    #activate environment
pip install -r requirements.txt         #install dependencies
```

Now it is time to build the application. The application contains two installation scripts — build.sh and jsbuild.sh. They assemble the JavaScript and C parts of the application.

```
cd ./flow
./jsbuild.sh     #assemble JS files
./build.sh       #build C
```

The installation is done. Now it is necessary to initialize the database.

```
# in the ./flow folder
python manage.py makemigrations
python manage.py migrate
```

Create the administrator of the local server. Please change the credentials according to your needs.

```
#in the ./flow folder
python manage.py createsuperuser
#insert the information requested by the app
```

And finally, launch the application.

```
#in the ./flow folder
#please do not change the port 9000
python manage.py runserver "0.0.0.0:9000"
```

The application should be now accessible from your browser on the address `localhost:9000`.

1.2 More Robust Installation

The alternative more robust installation uses Redis for enabling the web socket communication, Nginx for serving the static files, Guincorn for serving HTTP requests, and Daphne for serving the web socket requests. This setup has been chosen deliberately to distribute the load.

> If you plan to deploy the application and expect a heavy use, please adjust all of the actions below according to your needs. The correct parameters of the setup are highly dependent on your requirements.

The following code suggests how to install individual components:

```
##install stuff from makefile
make #or make mac

##install redis on Linux
##in the ~/Downloads or anywhere you like

wget http://download.redis.io/redis-stable.tar.gz
tar xvzf redis-stable.tar.gz
cd redis-stable
make install

## when somethign goes wrong
#make distclean

## install nginx
sudo apt install nginx
```

It is necessary to locate the Django config file `./flow/flow/settings.py` and swap the code bellow with the code in the comments.

```
CHANNEL_LAYERS={
    "default": {
        "BACKEND": "channels.layers.InMemoryChannelLayer"
     }
}

#CHANNEL_LAYERS = {
#    'default': {
#        'BACKEND': 'channels_redis.core.RedisChannelLayer',
#        'CONFIG': {
#            "hosts": [('127.0.0.1', 6379)],
#        },
#    },
#}
```

Launch Redis, the command will be probably different depending on your distribution.

```
redis-server --daemonize yes
```

To launch Nginx, you need to set up your local config file. An example of used config file usable on Linux is in the `misc` folder — `nginx.conf`

```
sudo nginx -c ./flow/nginx.conf >> runlog.txt 2>&1
```

Launch Gunicorn:

```
# name of the application (*)
NAME="flow"
# django project directory (*)
DJANGODIR="./flow"
# we will communicate using this unix socket (*)
SOCKFILE="/flow/run/gunicorn.sock"
# how many worker processes should Gunicorn spawn (*)
NUM_WORKERS=1
# which settings file should Django use (*)
DJANGO_SETTINGS_MODULE=flow.settings
# WSGI module name (*)
DJANGO_WSGI_MODULE=flow.wsgi

# create the run directory if it doesn't exist
RUNDIR=$(dirname $SOCKFILE)
test -d $RUNDIR || mkdir -p $RUNDIR

# start your Django Unicorn
gunicorn ${DJANGO_WSGI_MODULE}:application --name $NAME --workers
↪  $NUM_WORKERS --bind=unix:$SOCKFILE
```

Launch Daphne:

```
daphne flow.asgi:application --bind 0.0.0.0 --port 9000
↪  --verbosity 1
```

And there you go! Now the app should be up and running. Of course, do not forget to migrate the database and create the admin as was illustrated in the local developer installation.

2 Using the Application

Managing the application involves everything from creating the account to performing extensive computations. The following guide should clarify how to use the app, and enable creating 3D vector field visualizations.

2.1 Launching the Application

The local installation of the application can be easily launched with the command `make run`. If you are running the robust installation of the application, you have to start Redis, Nginx, Gunicorn, and Daphne yourself.

2.2 Launching the Computation Backend

Inside the Python environment run the command `make backend`. The computation backend will start automatically.

2.3 Database management

The database of the running application can be easily managed using the built-in Django administration application. It can be accessed via the `/admin` address. The database contains user accounts, datasets, notebooks, and a queue of running tasks processed by the computation backend. Please use the admin account you have created during the installation of the application to log into the administration.

2.4 Log in

Go to the main screen (usually `localhost:9000`) and log in with your admin account. The other option is to create a new account using the registration form. The registration form is accessible via a link on the main screen.

2.5 Uploading dataset

The dataset to be uploaded has to be in the following format. Currently, the only supported format utilizes the `FITs` format. Further, the structure of the file has to be the following:

```
Filename: example.fits

No.  Type            Dimensions   Format
  0  PrimaryHDU      ()
  1  ImageHDU        (x, y, z)    float32  #x component values
  2  ImageHDU        (x, y, z)    float32  #y component values
  3  ImageHDU        (x, y, z)    float32  #z component values
  4  ImageHDU        (x,)         float32  #x comp. of locations
  5  ImageHDU        (y,)         float32  #y comp. of locations
  6  ImageHDU        (z,)         float32  #z comp. of locations
```

When the contents of the file can be accessed via a single object `dataset` which behaves like an array, it is possible to access, e.g., the third `HDU` (`PrimaryHDU` included) as `dataset[2]`. Given a three-dimensional vector field V, the value at the position `p = (dataset[4][a], dataset[5][b], dataset[6][c])` is equal

status	description
0	the dataset is currently being processed
1	the dataset format is valid
-1	the dataset format is invalid, do not use it

Table 2: Dataset status

to `(dataset[1][*p], dataset[2][*p], dataset[3][*p])`. The positions stored in the arrays 4–6 have to be ascending or descending. The sampling along individual axes does not have to be uniform.

How to upolad a new dataset The dataset can be uploaded via the upload form in the sidebar. Select the local copy of the dataset file and enter the name of the dataset. Wait till the upload completes. When the upload is completed, the dataset is processed, and its validity is checked. When the processing is finished, the dataset is assigned a status. The table 2 lists possible states.

2.6 Managing dataset

Every uploaded dataset can be renamed, downloaded and deleted. Each uploaded dataset is assigned a unique UUID, which is used to identify the dataset during the calculation.

Rename The dataset can be renamed via the rename widget. Click the rename icon, insert a new name and confirm the change. The dataset will be renamed.

Copy UUID Retrieve the UUID of the uploaded dataset by clicking the code icon. On click, the code is copied into your clipboard. You can use the code inside the pipeline editor (notebook).

Download Any uploaded dataset (even invalid) can be downloaded back from the server. The downloading can be initialized by clicking the download button of the corresponding dataset.

Delete Delete the dataset from the server. This action is irreversible.

2.7 Managing notebook

Every notebook contains a description of the visualization pipeline. You can upload an existing notebook or create a new one. In the side panel form, name the uploaded notebook and select the local file, or click the new notebook button and create a blank one. Every notebook has the following controls:

Open Open the notebook. This action will present the editor and terminal UI for the corresponding notebook.

key	action
H	open or close help
D	start delete mode, you can delete nodes and edges
A	open or close menu with available nodes

Table 3: Editor key controls

Rename The notebook can be renamed via the rename widget. Click the rename icon, insert a new name and confirm the change. The notebook will be renamed.

Download Any notebook can be downloaded back from the server. The downloading can be initialized by clicking the download button of the corresponding notebook.

Canvas Open the output model created by the pipeline specified in the notebook. It opens the canvas interface (renderer) and automatically displays the output of the last calculation from the server.

Delete Delete the notebook from the server. This action is irreversible.

2.8 Assembling Pipeline Description

Notebooks allow assembling and storing visualization pipelines description. The visualization pipeline consists of operations such as loading dataset or integrating streamlines. The operations are represented as nodes and the input and output data of individual operations as edges between the different nodes.

Example setup This section presents a short how-to, demonstrating a simple pipeline setup. The goal is to create a simple streamline visualization.

1. First, you need a dataset. Copy the code (UUID) of a valid uploaded dataset you wish to visualize.
2. Open a new notebook (from the main screen, push the *new notebook* button). The controls of the notebook editor can be displayed anytime by pressing the H (as *help*) key. The key controls of the notebook editor are listed in the table 3.
3. Now you need to create a dataset node. This node represents loading the dataset. Press A (as *add*) to display the node menu.
4. Click the *dataset* button (in the data section).
5. Click again to place the dataset node.

6. Select the code input in the dataset node and press ctrl+V (or ⌘+V) to insert the value. The editor should automatically inform you about the sampling and dimensions of the dataset.

7. Select the *c* mode of the dataset. This means the dataset will use an interpolator written purely in C.

8. Add a points node by pressing the A again and clicking the points button. Place the node in the editor below the dataset node.

9. Depending on the dimension of the dataset, imagine a rectangle inside the dataset you would like to use as a seeding space for the streamlines. Now, you need to write down the coordinates of two opposite corners of the rectangle — write the individual coordinates into the inputs $x_{min} \dots z_{max}$. To clarify, the coordinates of the two opposite corners are $(x_{min}, y_{min}, z_{min})$ and $(x_{max}, y_{max}, z_{max})$. As it is a rectangle, a pair of coordinates of both points, e.g., z_{min} and z_{max}, have to be the same.

10. Select a sampling. The sampling along the axis perpendicular to the rectangle should be 1. For a start, use sampling such as $10 \times 10 \times 1$.

    The last two performed steps are telling the computation backend to take space defined by the two endpoints and regularly sample it with the specified sampling.

11. Finally, add a *streamline* node. Set the mode again to *c*, t_0 to 0 and t_{bound} to anywhere around 10. Leave the *appearance* on solid and set the *size* to 0.5.

    This tells the backend to perform the streamline integration using a solver implemented purely in C. The *t* parameters specify the length of the streamline, and the *size* parameter specifies the thickness of the streamline.

12. The light gray half-circle parts of the nodes are called ports. Click the **output** port of the **dataset** node.

13. Click the **input** port of the **streamline** node. A light gray line should appear between the two clicked nodes.

14. Click the **output** port of the **points** node.

15. Click the **input** port of the **streamline** node. A light gray line should appear again.

16. Finally, add a display node and link the streamline node with the new display node. The width parameter of the display node indicates how big the output scene should be. You can leave it at 100.

With the last step, the setup of a simple visualization pipeline is done. Save the notebook by clicking the save button in the top left section of the screen. Now you can leave the editor and open it again; the graph should be still there.

Nodes Each node contains three fields — title, in and out.

title	The title field contains a user-friendly name of the pipeline procedure which is represented by the node.
in	In field contains a list of types which are accepted by the node as an input. All required input types are listed without round brackets. Round brackets denote that the node does not require the selected data type to perform the operation represented by the node. Square brackets denote that the node accepts multiple instances of the given data type. When all of the input types are optional, still, at least one is required.
out	The out field contains a list of types which are the output of the node operation. Usually, it is a single new type or a list of the same types as the input types. The brackets carry the same meaning as in the input field.

Following list presents the available nodes and their parameters.

Dataset Dataset node represents the loading dataset operation. Requires none and produces a dataset.

code	UUID of the dataset on the server
mode	Denotes which interpolator implementation should be used. Two implementations are available - C and SciPy.

Points Marks a cuboid area of interest. Seeding points are constructed in this area. The area is specified by two opposite corners of the box. Creates a grid of points. Requires none and produces points.

$x_{min} \dots z_{max}$	Coordinates of the points defining the area of interest.
$x_{sampling} \dots z_{sampling}$	Sampling of the area of interest along individual axes.

Plane Marks a rectangle area of interest. Seeding points are constructed in this area. The area is specified by two opposite corners of the rectangle. Creates a grid of points. Requires none and produces a plane.

$norm_{axis}$	Defines the normal axis of the rectangle.
$norm_{value}$	Defines the position of the rectangle along the normal axis.

$a_{start} \dots b_{end}$	Opposite corner coordinates of the rectangle.
$a_{sampling}$, $b_{sampling}$	Sampling along the axes of the rectangle.

Glyphs Represents the glyphs construction operation. Requires one or more instances of points and dataset; produces glyphs.

size	Maximum size of the glyphs.
appearance	Denotes whether the glyphs are transparent or solid.

Streamlines Represents the streamline integration operation. The operation uses adaptive *RK45* method. Requires one or more instances of points and dataset; produces streamlines.

mode	Denotes which solver implementation should be used. Two implementations are available - C and SciPy.
t_0	Start time of the integration.
t_{bound}	End time of the integration.
appearance	Denotes whether the streamlines are transparent or solid.
size	Defines the thickness of the streamlines.

Layer Represents the operation of 2D slice construction. Requires plain and dataset; produces a layer.

appearance	Denotes whether the layer is transparent or solid.

Scale Represents a scaling operation. Positions or values can be scaled. Requires at least one instance of glyphs, streamlines, or layers; produces at least one instance of glyphs, streamlines, or layers.

part	Denotes what should be scaled.
$x_{scale} \dots z_{scale}$	Defines the scale of the output data along individual axes.

Translate Represents a translation operation. Positions or values can be translated. Requires at least one instance of glyphs, streamlines, or layers; produces at least one instance of glyphs, streamlines, or layers.

$x_{translate} \dots z_{translate}$	Defines the translation of the output data along individual axes.
part	Denotes what should be translated.

Colormap Defines the colormap applied to selected objects. Requires at least one instance of glyphs, streamlines, or layers; produces at least one instance of glyphs, streamlines, or layers.

$color_0 \dots color_5$	Colors used by the colormap.
sampling	Defines the number of colors used by the colormap.

Glyph Geometry Modifies the geometry of the input glyphs. Requires at least one instance of glyphs; produces at least one instance of glyphs.

geometry	Defines what geometry mode should be used for the glyphs.
size	Maximum size of the glyphs.
sampling	Defines how many sides the glyph geometry has.
appearance	Denotes whether the glyphs are transparent or solid.

Streamline Geometry Modifies the geometry of a streamline segment. Requires at least one instance of streamlines; produces at least one instance of streamlines.

size	Thickness of the streamlines.
sampling	Defines how many sides the streamline segment geometry has.
divisions	Defines how many intermediate subsegments a single segment consists of.
appearance	Denotes whether the streamlines are transparent or solid.

Visual Represents the rendering operation. Requires at least one instance of glyphs, streamlines, or layers; produces none.

$width_{max}$	Defines the maximum width of the scene.

Editor Controls Some additional notes about the editor:

- You can move the editor around by dragging the screen. Zooming with the mouse wheel is also possible.
- You can move the nodes by dragging them around. This action won't break the connections.
- Exit the delete mode by clicking anywhere in the editor.

command	action
`start`	starts the computation
`stop`	stops running computation or removes it from the queue of tasks
`help`	lists available commands

Table 4: Terminal commands

- Sometimes the window with the editor loses focus and does not register a key press. Click inside and the editor window and continue.[2]
- The graph should not contain any cycles.
- Only nodes with the same output and input types can be connected.

2.9 Launching Computation

The computation backend executes the visualization pipeline. There is a terminal in the notebook editor which allows communication with the computation backend. The terminal supports three commands — `start`, `stop`, and `help` — see table 4.

The computation is initialized with the start command. The representation of the graph is serialized and added to the queue of waiting tasks, which are consequently processed by the computation backend. When the computation begins, the user is informed about the execution of each node by the terminal. When the calculation finishes, a new file is generated and made available for download. The new file contains a computation result[3] and can be inserted into the renderer (canvas).

2.10 Opening Computation Result

The output file of the computation has a suffix `.imgflow`. It is possible to open and visualize a downloaded computation result or to open the result of a notebook uploaded and calculated on the server. To open a downloaded result:

1. Navigate to the main screen.
2. Click the canvas component in the sidebar.
3. Upload the downloaded `.imgflow` file and open the visualization.

To open a result computed on the server:

1. Navigate to the main screen.
2. Click the canvas icon of the corresponding notebook.

[2]This happens regularely after any deletion. No fix was discovered to prevent this behavior; it happens automatically when the browser removes an HTML element.

[3]The complete model constatis of various data formats such as streamline trajectories, glyph positions and values, or slice data.

key	action
M	open or close menu
H	display or hide help
↑	move up in the menu
↓	move down in the menu
→	increase highlighted value/change value
←	decrease highlighted value/change value
⌥ + →	increase highlighted value with bigger step/change value
⌥ + ←	decrease highlighted value with bigger step/change value
⇧ + →	select next scene (if multiple scenes are available)
⇧ + ←	select previous scene (if multiple scenes are available)
⇧ + ↑	select previous object (if multiple objects are available)
⇧ + ↓	select next object (if multiple objects are available)
⏎	load the selected scene
P	change perspective
W	rotate the view up
S	rotate the view down
A	rotate the view left
D	rotate the view right
C	capture the current image (in 4K resolution)
V	create image with depth map
F	front view
R	right-side view
T	top view
⇧ + W	move the view up
⇧ + S	move the view down
⇧ + A	move the view left
⇧ + D	move the view right
B	start or stop the animation
Space	pause or play the animation
G	enable color transformations
L	invert the colormap of selected object
U / I J / K	local widget controls

Table 5: Renderer (canvas) key controls

2.11 Downloading Computation Result

The output of the pipeline computation can be downloaded only from the terminal. When the calculation finishes, the terminal presents you with a link, prompting you to download the file. If you wish to download the result of the last computation but the corresponding notebook has been closed and opened again; thus, the terminal has been cleared, you can:

1. Open the relevant notebook.
2. Extract the notebook UUID from the URL notebook address. The UUID is also displayed in the terminal.
3. Insert the UUID into the following address:

   ```
   /media/notebook/<insert your UUID here>/output.imgflow
   ```

4. Paste the address into the browser. The computation result should be downloaded.

2.12 Canvas (Renderer)

The canvas is the UI of the renderer, allowing to visualize the computation outputs. The table 5 contains a list of control keys. The UI of the renderer consists of multiple widgets on the right side of the screen and a single status bar in the bottom left corner of the screen. The type of the currently selected object is always displayed in the first field of the UI and all of the listed controls are related to the selected object. The table 6 lists all properties of the scene objects.

Animation widget The animation widget is displayed if there is any streamline model in the scene. The parameters presented in the widget control the animation of the streamlines.

Color widget The color widget allows for color transformation; the supported methods are gamma correction, map inversion, and change of the colorbar mode. One mode uses the entire specified color range; the second one divides the color range, uses the middle color for zero values, and spreads the color range equally in both directions, preserving the optical distances.

field	meaning	values
type	type of selected object	`glyphs | streamlines | layer`
appearance	appearance of the object	`solid | transparent`
culling	controls whether the culling is applied	`on | off`
size	size of the glyphs or streamlines	`a number`
t_{start}	start time of the displayed streamlines	`a number`
t_{end}	end time of the displayed streamlines	`a number`
mode	color mode of the selected object	`xyz | x | y | z`
brightness	brightness of the selected object	`a number`
visibility	whether object, box and labels are visible	`visible | hidden`
box	whether box and labels are visible	`visible | hidden`
labels	whether labels are visible	`visible | hidden`

Table 6: Properties of the renderer objects

3 Complete Project Structure

Following tree illustrates the overall structure of the project.

```
src
├── flow
│   ├── flow
│   │   └── settings.py ..... Django project settings
│   ├── visual ..... Django app containing the entire project
│   │   ├── modules
│   │   │   ├── editor ..... editor module of the computation backend
│   │   │   └── numeric ..... numeric module of the computation backend
│   │   └── static/visual
│   │       ├── canvas ..... renderer source codes
│   │       └── portal ..... web app source codes
│   ├── misc
│   │   └── benchmarks.py .... computation backend benchmark source codes
│   ├── setup.py ..... Cython compilation script
│   ├── background.py ..... compuation backend main script
│   ├── manage.py ..... Django project main script
│   ├── build.sh ..... computation backend build script
│   └── jsbuild.sh ..... script assembling renderer and portal JS codes
├── sage ..... the directory containing source code of SAGE2 app
├── makefile ..... main project makefile
└── requirements.txt ..... Python requirements
```

4 Proprietary formats

The application uses two proprietary formats. The first is used for storing the graph representation, the second stores the output of the computation backend.

Notebook file `.docflow` The listing 1 describes the structure of the notebook file. The notebook file contains a representation of the pipeline graph. The file utilizes JSON for storing the nodes as objects. Individual parameters are stored as object properties.

```
[
  //instance of a single node:
  {
    "id": //node id
    "in": [/*ids of input nodes*/]
    "out": [/*ids of output nodes*/]
    "position": {
      "x": //x node coordinate
      "y": //y node coordinate
    }
    "data": {
      "structure": {/*node parameters*/}
      "in": {
        //instance of a single input type:
        "intype": {
          "required": bool //the input type is required
          "multipart": bool //multiple insances are accepted
        }
        ...
      }
      "out": {
        //instance of a single output type:
        "outtype": {
          "required": bool //the output type is always produced
          "multipart": bool //multiple insances can be produces
        }
        ...
      }
    }
  },
  //followed by more nodes...
  ...
]
```

Listing 1: Notebook file structure

Image file `.imgflow` The listing 2 describes the structure of the image file produced by the computation backend. The file again utilizes the JSON format. The contained structures represent individual scenes and individual objects.

5 GitHub Repository

The application is still a prototype; if you experience any issues with the application, please report them by creating an issue in the git repository of the application at https://github.com/vojtatom/flow.

```
[
  //contents of a single scene:
  {
    //array of layers conained in the scene:
    "layer": [
      {
        "meta": {
          "appearance": //layer appearance
          "bounds": {
            "dim": //3-component vector of layer dimensions
            "start": //3-component coordinate of the layer origin
          },
          "colormap": {
            "colors": //array of colors
            "sampling": //number of utilized colors
          },
          "geometry": {
            "normal": //x, y or z axis
            "normal_value": //position of the layer along the normal axis
            "samlping": //3-component vector
          }
        }
        "points": //Base64 encoded array of 3-component vectors, represents the space points
        "values": //Base64 encoded array of 3-component vectors, represents the values
      },
      ...
    ],
    //array of streamlines conained in the scene:
    "streamlines": [
      {
        "meta": {
          "appearance": //see above
          "bounds": //see above
          "colormap": //see above
          "divisions": //number of segment divisions
          "sampling": //number of segment geometry sides
          "size": //streamline thickness
          "t0": //start time
          "tbound": //end time
        },
        "points": //Base64 encoded array of 3-component vectors, represents the space points
        "values": //Base64 encoded array of 3-component vectors, represents the values
        "lengths": //Base64 encoded array of integers, number of streamline segments
        "times": //Base64 encoded array of floats, streamline segment timestamps
      },
      ...
    ],
    //array of glyphs conained in the scene:
    "glyphs": [
      {
        "meta": {
          "appearance": //see above
          "bounds": //see above
          "colormap": //see above
          "geometry": //geometry type
          "sampling": //number of glyph geometry sides
          "size": //maximum glyph size
        },
        "points": //Base64 encoded array of 3-component vectors, represents the space points
        "values": //Base64 encoded array of 3-component vectors, represents the values
      },
      ...
    ],
    //scene stats
    "stats": {
      "points": {
        "center": //3-component vector, geometric center of the scene
        "max": //near bottom left corner of the scene
        "min": //far top right corner of the scene
        "scale_factor": //scene scale
      },
      "values": {
        "x": //scene values statistics for x-components
        "xyz": //scene values statistics for entire vectors
        "y": //scene values statistics for y-components
        "z": //scene values statistics for z-components
      }
  },{
      //next scene
      ...
  }
]
```

Listing 2: Image file structure