NAMNIs: Neuromodulation And Multimodal NeuroImaging software

T. Karali[1, 2], F. Padberg[1], V. Kirsch[4], S. Stoecklein[2], P. Falkai[1], D. Keeser[1, 2, 3]

2021

[1]Department of Psychiatry and Psychotherapy, University Hospital LMU, Munich, Germany
[2]Department of Radiology, University Hospital LMU, Munich, Germany
[3]Munich Center for Neurosciences (MCN) – Brain & Mind, Planegg-Martinsried, Germany
[4]Department of Neurology, University Hospital LMU, Munich, Germany

Abstract

The basic requirements in neuroimaging research are changing rapidly due to the growing size of datasets and multimodal approaches with increasingly complex, time-consuming, diverse and fast-moving standards. Neuromodulation And Multimodal NeuroImaging software (NAMNIs) offers a ready-to-use, open-source pipeline for pre- and post-processing of multimodal neuroimaging and neuromodulation data. A strong focus is maintained on reproducibility and support for multi-platform parallelization. Computations are performed in both native volume and surface spaces as well as in MNI standard space. Input and output data of these calculations conform to the international Brain Imaging Data Structure (BIDS) format. The software is intended to enable users to better interpret results using MRI-based modalities and to calculate simulations based on this data that can later be compared with the results of neuromodulation pilot or clinical studies. The simple integration into a high-performance computing (HPC) environment allows the calculation of large datasets or retrospectively combined samples in a feasible period of time.

version `0.3` (status: `prototype`)	contact:
check for updates	`namnis.team@protonmail.com`

1 Introduction

State of research and functionality Multivariate analysis methods of MRI brain imaging have become into efficient tools of applied and basic neuroscience in the last decade. A large variety of tools for the analysis of multimodal MRI neuroimaging has been developed in recent years. Often, these software solutions specialize in one modality, for example structural neuroimaging (e.g. MRtrix3 [1]) or functional connectivity (e.g. fmriprep [2]). In addition, there are generally large toolboxes (e.g. AFNI [3], FSL [4], SPM [5]) that are suitable for analysis and statistical evaluation of a variety of modalities, but often the steps have to be compiled manually by the user. The program code varies in the analysis tools from mainly Python and R in AFNI, bash and Python in FSL and Matlab in SPM.

NAMNIs consists of a processing pipeline for multimodal magnetic resonance imaging (MRI) data analysis using parallel processing. It performs various pre-processing steps with the aforementioned data, calculating relevant metrics such as the number of activated voxels (spatial extent), within regions of interest (ROIs) effects, ROI-to-whole-brain calculations, probabilistic values, motion parameters, connectivity strength values (standardized in z scores), and more. This is implemented in native space, standard MNI space, and surface space for structural T1-, T2-weighted data. In addition, structural T1- and T2-data are used for the simulation of non-invasive brain stimulation. Calculation steps are performed in parallel with different topologies to allow processing of large datasets in reasonable time. It is applicable to HPC and is provided as research software that is free and open-source software (FOSS). Already in the proof-of-concept and prototype phase, the project was published as abstracts of international conferences [6] [7].

Key features Key features of NAMNIs are displayed below.

Implementation	Fully implemented in Python 3 Dependencies are open-source software tools only Local, container, and HPC environments are supported
Multimodality	Apply same atlases with multiple ROIs to all modalities T1 and T2 (VBM) calculations (more available soon) Head mesh and surface reconstruction for neuromodulation
Calculations	Run surface-based and volumetric pre-processing Run post-processing including ICV-corrections Output values for the whole sample size as CSV files
Reproducibility	All steps can be reproduced using a single config file Reusable input and output data format Built-in dependency verification for reproducible calculations

2 Implementation Details

Design choices In order to reduce reproducibility issues across different operating systems, a design choice was made to support only Linux-based operating systems and the 64-bit system architecture. This design choice is important to achieve reproducibility goals using containerization solutions, as explained in subsection 3.2. The latest LTS version of Ubuntu (`20.04`) has been used as the base image for the `Dockerfile` that is distributed with NAMNIs, though it is possible to use NAMNIs with any other 64-bit Linux distribution as well.

In order to offer NAMNIs completely as free and open source software (FOSS), a design choice was made to exclusively use FOSS as dependencies of NAMNIs, regardless of whether these are functionality dependencies or platform dependencies. This provides a key advantage for users and developers who wish to base their platform exclusively on FOSS to achieve maximum transparency as open-science best practice [8]. There are several research software approaches in the literature that are FOSS, but depend on proprietary software packages, making it impossible to use these software packages in a fully open source research environment. It was preferred that NAMNIs has no such limitations and can be used in accordance with best practices in open-science research.

Functionality dependencies NAMNIs code makes extensive use of third-party dependencies to provide core functionality (i.e. functionality dependencies). The most important third-party dependencies of NAMNIs are listed below.

- ciftify [9]
- freesurfer [10]
- FSL [4]
- gmsh [11]
- numpy [12]
- octave [13]
- pandas [14]
- pybids [15]
- SIMNIBS (only in combination with the provided patch file) [16]

Platform dependencies NAMNIs codebase was programmed exclusively in Python 3 [17]. Therefore, Python 3 (version 3.7 or higher) and a Linux-based operating system are the main platform dependencies of NAMNIs. Despite cross-platform support of Python [17], a design choice was made not to support any of the earlier versions of Python in order to take advantage of the new features offered by Python 3.7 such as dataclasses, f-strings, ordered dictionaries by default, and more [17] without having to rely on fallbacks, backports or self-implementations for older versions. This allows a cleaner code-base that is more intuitive and less error-prone. Optional platform dependencies of NAMNIs are listed below.

- Docker [18] for containerized installation
- SLURM [19] for high-performance computing (HPC) parallelization support

Status of development The current status of development is **prototype**. NAMNIs is actively maintained and we intend to continue extensive development in the future as well. The current version is developed and internally verified to the best of our knowledge. However we acknowledge that it is possible that errors or bugs of any kind may exist in the code. **Please take this into consideration if you wish to use the software or publish results generated with the software.** We would appreciate if you contact us regarding problems, bug reports, and any other comments. Please use the e-mail address above if you would like to contact us.

3 Methods and Results

3.1 Reproducibility-optimized Workflow

A combination of raw data and configuration data is defined as a *NAMNIs workflow*. The aim is to enable standardization at all levels of user input to ensure better reusability and reproducibility. This workflow is fully BIDS compatible and utilizes `rawdata/` and `code/` components in the specification [20]. Each NAMNIs workflow produces reusable output that is also BIDS compatible.

Raw data BIDS [20] is the only compatible input raw data format. Any BIDS compatible dataset is valid as raw data for NAMNIs, though a comprehensive example has been made freely available in a recent publication [22].

As of this version, NAMNIs uses only `anat` data, as defined in the BIDS specification version 1.3.0 [20] (structural imaging such as T1, T2, etc.). It is planned to support more data types in future versions, as mentioned in section 5. Using this raw data, a NAMNIs workflow can be configured to perform processing tasks. A default workflow function is pre-defined for each of these tasks, though in a modular design it is possible for software developers to create their own workflow functions, if preferred.

Configuration data A common framework for parsing configuration files provides the capacity for a configuration file to define all relevant variables of a certain run on a given dataset. Configuration input is also standardized using JSON files, an example of which is elaborated in section 4. This input will be further formalized for robustness in the future versions, as mentioned in section 5.

Output data In order to increase reproducibility and enable reusability of output data, it has been constructed in accordance with the BIDS Derivatives specification, as defined in the BIDS specification version 1.3.0-dev[5]. This provides a standardized structure of the output data, which comes useful especially in further automatised analyses of the output data. As of this version, under `derivatives/` subdirectory, the following outputs are generated (`{modality}` noting the name of modality as defined in the BIDS specification version 1.3.0 [20]).

- `namnis_{modality}_intermediate` contains the intermediate files that are only generated in the steps leading to the generation of the actual output files.
- `namnis_{modality}_results` contains binary- (such as `.nii.gz`) and text-format pre-processing outputs.
- `namnis_post` contains all post-processing outputs.

[5] `https://bids-specification.readthedocs.io/en/derivatives/`

- `derivatives` contains all other outputs that are generated by the third-party dependencies and are not BIDS compatible

Since the derivatives branch of the BIDS specification is still in the draft phase (as of version 1.3.0-dev)[6], the exact structure of the output files may change in the future versions, in order to adapt to changing specification and possibly to better follow the recommendations in the specification. Regardless, thanks to the standardization of the derivatives files, individual modules of NAMNIs and other BIDS-compatible software (such as [15]) will continue to work with BIDS Derivatives compatible data.

3.2 Containerization and Run-time Dependency Verification

Containerization Achieving reproducibility has been one of the most significant motivations behind NAMNIs. In order to achieve maximum reproducibility, we implement containerization, as recommended in the current literature [26]. On par with other solutions in the literature [21] that enable creating containers with versioned dependencies, NAMNIs also provides a `Dockerfile`, with which a reproducible environment including all third party dependencies can be built independently by the user. Assuming that same NAMNIs container is deployed across different systems, it is possible to reproduce the same results in this way, apart from the variance caused by different operating system kernels and different hardware. The operating system kernel factor could be eliminated by deploying NAMNIs in a virtual machine, however it would require substantially more effort on the user side to set up, and would only be possible with a performance downgrade. The hardware factor cannot be realistically eliminated; therefore it is recommended that some hardware details about the system(s) used for analyses are mentioned in publications.

Run-time dependency verification Comparing the version number of a software instance to another is insufficient to prove that both instances are identical, given that differences can be found —or even expected— depending on the build process of the software code. In order to address this challenge, NAMNIs features a run-time dependency verification mechanism. This mechanism is based on validating SHA256 hashes of software packages in a way that is agnostic of the package manager used. The concrete steps of the verification process are enumerated below.

1. During the container build process (equivalent to *compile-time*), a `namnis/` folder at the root directory is created (i.e. `/namnis/`). During the installation of each dependency, SHA256 hashes are generated and saved into a plain text files in this directory. The verification method used for each package manager is listed below.

   `apt` First the package is installed using `apt-get`, then the SHA256 hash of the installed package is obtained using `apt-cache`.

   `pip` Given that `pip` offers a built-in mechanism to ensure repeatability[7], this mechanism is used. Simply the SHA256 hash values that have been used for the hash checking mode are copied.

   `opt` For packages that are installed without using a package manager, SHA256 hash of the archive file containing the binary distribution is calculated using `sha256sum`.
2. During run-time, hash values stored in the `namnis.common.dependency` module are compared to the values that have been collected and stored in `/namnis/` directory.
3. If all hash values are identical, a note indicating thereof is added to `dataset_description.json`.

3.3 Parallelization

Two topologies are available for parallelization: simple parallelization using python `multiprocessing` pools [17] and complex parallelization by incorporating a job scheduler (SLURM [19]) in high performance clustered environments.

Job scheduler The main advantage of using a job scheduler is to enable collaboration between different machines, thereby enabling the use of large amounts of computing resources, especially for calculations with large sample sizes. This requires a clustered HPC environment to be available. This parallelization approach is implemented by incorporating two key features of SLURM [19] into NAMNIs framework, namely *job arrays* and *job dependencies*.

[6] `https://bids-specification.readthedocs.io/en/derivatives/`

[7] `https://pip.pypa.io/en/stable/user_guide/#repeatability`

`multiprocessing` pools The simpler parallelization approach (i.e. parallelization at the subject-level based on a fixed `multiprocessing` pool) is particularly useful on single-machine systems.

Even though the simpler parallelization approach does not support all the advanced features of the complex parallelization approach, it was crucial that the calculations could be performed in a similar way regardless of the parallelization approach available. In order to achieve this, core functionality of `namnis.common.starter` module had to be platform-abstracted. The table below elaborates the solution approaches to implementing this core functionality in a platform-abstract manner.

	Support for varying input data	Managing nested parallel job structures
`multiprocessing` pools	`starmap()` method of `multiprocessing.Pool`	`join()` on `Sequence` of `multiprocessing.Pool` objects
Job scheduler	job arrays	job dependencies

The code snippet below demonstrates how the aforementioned features are implemented independently of the parallelization approach.

```
def _start(self, is_distributed: bool, **kwargs):
  """
  Platform abstraction (local or SLURM) of job starter

  :param is_distributed: Whether the code is running using SLURM
  """

  if is_distributed:
    log.debug("Using sbatch")

    # set dependencies if all dependencies are SLURM job IDs
    try:
      dependencies = "afterok:" + ",".join(self.dependency)
      kwargs.update({'dependency': dependencies})

    except TypeError:
      pass

    # set executable path
    executable = [kwargs.pop('script')]

    # set environment variables
    kwargs.update({
      'export': ','.join(f"{k}={v}" for k, v in kwargs['export'].items())})

    job = run(["sbatch"] +
          list(sum(({f'--{k}': (str(v) if v is not None else '') for k, v
               in kwargs.items()}.items()), ())) +
          executable, silent=True,
          return_output=lambda x: re.match(
            r'Submitted batch job (\d+)', x).groups()[0])

    log.info(f"Submitted batch job {job}")
    return job

  else:
    # join `multiprocessing.pool.Pool` dependencies, if any
    try:
      for pool in self.dependency:
        pool.join()
```

```
    except TypeError:
      pass

    with multiprocessing.Pool() as pool:
      pool.starmap(
        _run_single_job,
        [(self.job_script, self.job_data_file, i) for i in range(
          len(self.job_list))]
        if self.job_list != [Ellipsis] else [
          (self.job_script, self.job_data_file)])

      pool.close()
      return pool
```

Note that `self.dependency` is of `Union` type.

```
dependency: Union[None, Sequence[str], Sequence[multiprocessing.Pool]]
```

3.4 Overview of the Pipeline Steps

A non-exhaustive list of the pipeline steps for each modality is summarized in the corresponding subsections.

3.4.1 `imaging` > `anat`

Pre-processing

1. In the first step, `fslreorient2standard` [4] is called to correctly orient all MR MPRAGE imaging data to match the orientation of the MNI152 standard template.
2. The re-oriented MPRAGE images are brain extracted using `bet` [4] with the specific parameters as outlined below (note that these parameters may vary depending on scanner type and MPRAGE scan MR acquisition settings).

   `-R` enable robust brain centre estimation (multiple iterations)

   `-f 0.45` fractional intensity threshold. This parameter might change or be adjusted in the future in order to accomodate any dataset.

   `-g` 0
3. Afterwards, a binary mask is created using `fslmaths` [4].
4. With the help of `fast` (FMRIB's Automated Segmentation Tool) [4], the brain is segmented into the tissue groups CSF (0), gray matter (1), and white matter (2). This fully-automated tool is based on a hidden Markov random field model and an associated expectation-maximization algorithm.
5. Both linear and non-linear registrations (using `flirt`, FMRIB's Linear Image Registration Tool [23] and `fnirt`, FMRIB's Nonlinear Image Registration Tool [24], respectively) are used to register the atlas on the individual MPRAGE images. By default, 12 degrees of freedom (DOF) are used.
6. The transformation/deformation field is inverted.
7. The total volume for the tissue CSF (0), GM (1) and WM (2) is calculated for each subject. The intracranial volume (ICV) is also calculated for all subjects.
8. Individual MPRAGE brain atlases are generated using `applywarp` [4]. The atlases are read from the fourth dimension of the atlas file using `fslval` tool [4]. The native subject T1 space is normalized to the MNI standard space with affine registration and nonlinear registration. The individual parcelling of the brain is performed according to the atlases used. All regions are extracted individually for GM and WM for the volume (in mm^3) and number of voxels; `fslmaths -mas` [4] and `fslstats -V` [4] are used for this step.

As input data for both surface-based and non-surface-based calculations either T1-MPRAGE data (BIDS `suffix: T1w`) or additionally T2 weighted data (FLAIR/T2 SPACE, BIDS `suffix: T2w`) can be used. The use of T2 weighted structural data is optional. An evaluation of the results between MPRAGE and T2-FLAIR datasets can be found in a recent publication [25].

Post-processing As of this version, `anat` post-processing only includes the generation of combined tables with ICV-corrected volumes.

1. First, subject-specific CSV files under `namnis_anat_results` are merged into combined CSV files with multi-indexes.
2. Then, ICV corrected volumes are calculated. Each $volume$ value of each region of each subject is divided by the sum of the total tissue values CSF, GM, WM of each subject. This value is then multiplied by the arithmetic mean of the sum of the total CSF, GM, WM values of the whole sample. With a sample size of n, each ICV-corrected $volume$ is calculated using the following function.

$$f(r,s) = \frac{volume_{\text{r, s}}}{totalCSF_{\text{s}} + totalGM_{\text{s}} + totalWM_{\text{s}}} * \frac{\sum_{i=0}^{n}(totalCSF_{\text{i}} + totalGM_{\text{i}} + totalWM_{\text{i}})}{n}$$

   Where r is the region index and s the subject index. This calculation is carried out analogously for each gm and wm volume from `T1w` and `T2w` data, depending on availability.

3.4.2 `modulation` > `surface`

As of this version, this module consists of using `freesurfer` [10] for the reconstruction of the surface space. Afterwards the reconstructed surfaces are converted and represented into a quality-control view using `ciftify` [9].

3.4.3 `modulation` > `mesh`

Pre-processing As of this version, this module consists of using `octave` [13], `gmsh` [11], and `headreco` [16] to generate a head mesh model. In the future versions, the neuromodulation steps for tDCS and TMS simulations will be included within this module, as mentioned in section 5.

Post-processing Based on the head mesh that has been generated during pre-processing, a ROI analysis using user-provided MNI coordinates is conducted.

1. First, gray matter is extracted from the head mesh.
2. Afterwards, given MNI coordinates are cast onto subject coordinates using `SIMNIBS` [16].
3. Element field of the ROI is determined and added onto gray matter that has been extracted.
4. Weighted mean of gray matter fields are calculated.
5. Finally, a combined CSV file with multi-indexes is generated.

4 Usage Example

Suppose one would like to generate all pre-processing outputs of the referenced example dataset [22] using NAMNIs, and nothing else. After downloading `EX.tar.gz`, it can be extracted using the following command. The following command also creates a `code/` subdirectory in the root directory of the dataset, which can be used to store configuration-related files and the configuration file itself.

```
tar -xf EX.tar.gz && mkdir -p EX/code/
```

In this way, raw data is available to be used with NAMNIs. The next step is to prepare the configuration data, as described in subsection 3.1. It is easy to notice if every top-level object in the configuration file except `globals` were to be collapsed into a single on-off switch, the configuration file would look like the code snippet below.

```
{
    "globals": {},
    "imaging": {
        "anat": {
            "enabled": true
        }
    },
    "modulation": {
        "mesh": {
            "enabled": true
        },
        "surface": {
            "enabled": true
        }
    }
}
```

An actual configuration file is merely an elaboration of the basic structure above, since modules are exactly what constitutes the core of a NAMNIs configuration file. A module is defined as either a modality or a reconstruction procedure. To enhance readability, these are split into corresponding categories `imaging` and `modulation`. It is also easy to notice that all boolean values are preset to `false` by default when parsed, meaning that the only values that had to be changed for this example are those listed above.

Note that there are other parameters that must be set outside of `globals`:

- `fsl_config_file` path in `anat` section. Here, the path for a FSL configuration file [4] that is compatible with the standard template of choice must be provided.
- `post_processing` object in each section. For this example, this will be an object that contains only one boolean `{enabled: false}` value.

At this point, only `globals` section is not set. In accordance with the previous code snippet, represented below is a simplified `globals` section, with all of its subsections as empty objects and boolean values as `false`.

```
{
    "globals": {
        "atlases": {},
        "standard_template_file": {}
    }
}
```

`standard_template_file` must be a valid path to the standard template of choice. Objects in the `atlases` section must contain a path to the atlas file, and a list of regions. The list of regions can be provided either as a JSON list or in a separate plain text file, whose path must be specified in place of the JSON list above. Ideally, this text file should list the name of each region in each line. In the following configuration file example, two atlas definitions (as defined by each of the aforementioned methods for listing regions) are used.

Below the previous code snippets are extended into a complete configuration file.

```
{
    "globals": {
        "atlases": {
            "atlas0": {
                "file": {
                    "__relative_path__": "code/atlas/atlas0.nii.gz"
                },
                "regions": {
                    "__relative_path__": "code/atlas/atlas0.txt"
                }
            },
            "atlas1": {
                "file": {
                    "__relative_path__": "code/atlas/atlas1.nii.gz"
                },
                "regions": [
                    "region0",
                    "region1"
                ]
            }
        },
        "standard_template_file": {
            "__relative_path__": "code/standard_template.nii.gz"
        }
    },
    "imaging": {
        "anat": {
            "enabled": true,
            "fsl_config_file": {
                "__relative_path__": "code/standard_template.cfg"
            },
            "post_processing": {
                "enabled": false
            }
        }
    },
    "modulation": {
        "mesh": {
            "enabled": true,
            "post_processing": {
                "coordinates": {},
                "enabled": false,
                "radius": 0
            }
        },
        "surface": {
            "enabled": true
        }
    }
}
```

Assuming a working NAMNIs installation is present, if the configuration above is stored in a file located under, e.g. `~/EX/code/config_1.json`, starting the NAMNIs workflow is as simple as running the command below.

```
namnis_starter --config ~/EX/code/config_1.json
```

Additional parameters may be provided with the `namnis_starter` command. A complete list of these parameters can be obtained using the following command.

```
namnis_starter --help
```

Once the NAMNIs workflows are completed, the results can be found under `~/EX/derivatives/`.

5 Future Perspectives

NAMNIs is expected to undergo extensive further development, including changes at the technical level of software implementation. It goes beyond the scope of this section to provide an exhaustive list of all future prospects of this project. The following is a summary of some of the key features that we expect to be provide as part of NAMNIs.

- Introduce pipeline steps for `dwi`, `func`, `fmap`, and `beh` modalities, as defined in the BIDS specification as of version 1.3.0 [20].
- Formalize the processing of configuration data using a specific JSON schema.
- Add support for tDCS and rTMS neuromodulation simulation tasks.

6 Conclusion

The reproducibility crisis in scientific papers [27] and constantly growing requirements in computational resources show that reproducible workflows with strong parallelization support will be a prerequisite for any neuroimaging lab in the future. NAMNIs offers a ready-made, cost-free, and open source alternative. It is planned to publish NAMNIs also in the form of a "peer-reviewed" publication.

7 References

[1] Tournier et al. (2019): MRtrix3: A fast, flexible and open software framework for medical image processing and visualisation. NeuroImage, 2019, 116137.

[2] Esteban et al. (2019): fMRIPrep: a robust preprocessing pipeline for functional MRI. Nat Methods 16:111–116.

[3] Cox (1996): AFNI: Software for Analysis and Visualization of Functional Magnetic Resonance Neuroimages. Computers and Biomedical Research 29:162-173.

[4] Jenkinson et al. (2012): FSL. NeuroImage 62:782-790.

[5] Friston (2007): Statistical parametric mapping. Statistical Parametric Mapping. doi:10.1016/b978-012372560-8/50002-4

[6] Karali et al. (2017): LMU Scripts • Ready-Made HPC-Applicable Pipeline for Structural and Functional Data Analyses. 23th Human Brain Mapping Congress Vancouver, Canada.

[7] Karali et al. (2019): NAMNIs: Neuromodulation And Multimodal NeuroImaging scripts. 25th Human Brain Mapping Congress Rome, Italy.

[8] Halchenko et al. (2015): Four aspects to make science open "by design" and not as an afterthought. GigaScience 4:31.

[9] Dickie et al. (2019): edickie/ciftify: Fix to ciftify_meants and new ciftify_dlabel_to_vol script (Version 2.3.3). Zenodo. doi:10.5281/zenodo.3369937

[10] Fischl (2012): FreeSurfer. NeuroImage, Volume 62, Issue 2, Pages 774-781, ISSN 1053-8119. doi:10.1016/j.neuroimage.2012.01.021

[11] Geuzaine et al. (2009): Gmsh: a three-dimensional finite element mesh generator withbuilt-in pre- and post-processing facilities. Int J Numer Meth Engng 2009;0:1–24

[12] van der Walt et al. (2011): The NumPy Array: A Structure for Efficient Numerical Computation, Computing in Science & Engineering, 13, 22-30, doi:10.1109/MCSE.2011.37 (publisher link).

[13] Eaton et al. (2015): GNU Octave version 4.0.0 manual: a high-level interactive language for numerical computations. `https://www.gnu.org/software/octave/doc/interpreter/`

[14] Reback et al. (2019): pandas-dev/pandas: v0.25.3 (Version v0.25.3). Zenodo. doi:10.5281/zenodo.3524604

[15] Tal et al. (2019): PyBIDS: Python tools for BIDS datasets (Version 0.9.4). Zenodo. doi:10.5281/zenodo.3458537

[16] Thielscher et al. (2015): Field modeling for transcranial magnetic stimulation: a useful tool to understand the physiological effects of TMS? IEEE EMBS 2015, Milano, Italy.

[17] van Rossum et al. (1991): Interactively Testing Remote Servers Using the Python Programming Language, CWI Quarterly, Volume 4, Issue 4, Amsterdam, pp 283–303.

[18] Merkel (2014): Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux J

[19] Yoo et al. (2003): SLURM: Simple Linux Utility for Resource Management. Job Scheduling Strategies for Parallel Processing. JSSPP 2003. Lecture Notes in Computer Science, vol 2862.

[20] Gorgolewski et al. (2016): The brain imaging data structure, a format for organizing and describing outputs of neuroimaging experiments, Scientific Data, 3, 160044.

[21] Kaczmarzyk et al. (2020): ReproNim/neurodocker: Version 0.7.0. Zenodo. doi:10.5281/zenodo.3753248

[22] Keeser et al. (2020): Effect of Aerobic Exercise on Cortical Thickness in Patients with Schizophrenia – a Dataset. OSF, 12 Mar. 2020. Web. doi:10.17605/OSF.IO/SFGXK

[23] Jenkinson et al. (2002): Improved Optimisation for the Robust and Accurate Linear Registration and Motion Correction of Brain Images. NeuroImage, 17(2), 825-841.

[24] Andersson et al. (2010): Non-linear registration, aka spatial normalisation. FMRIB technical report TR07JA2.

[25] Beller et al. (2019): T1-MPRAGE and T2-FLAIR segmentation of cortical and subcortical brain regions-an MRI evaluation study. Neuroradiology vol. 61,2: 129-136. doi:10.1007/s00234-018-2121-2

[26] Boettiger (2015): An introduction to Docker for reproducible research. Operating Systems Review, 49, 71-79.

[27] Nature Editors (2012): Must try harder. Nature. 483, 7391, 509–509.