Supplementary material for the paper: Simulating cyclic voltammetry at rough electrodes by the digital-simulation-deconvolution-convolution algorithm

In general we require a strategy for the simulation of the diffusive mass transfer towards a rough electrode structure. For this purpose, a modified version of the digital-simulation-deconvolution-convolution (DSDC) algorithm was developed (cf. main paper). The main steps are described below.

1) Digital simulation and the Douglas-Gunn algorithm

The first step of the algorithm is based on solving of the three-dimansional diffusion at the rough interface under investigation and to compute the cumulated flux-response towards this particular structure. In order to reduce the computational effort, a modified version of the Douglas-Gunn algorithm (an alternating-direction-implicit, i.e. ADI-method) was employed and optimized for the given task. This was done by introducing an exponentially expanding spatial grid inside of the electrolyte layer in front of the rough surface. Since seting up the related (pythonic) scripts may not be trivial for some users, a detailed guide for the installation of the respective scripts is given below.

2) Deconvolution of the mass-transfer functions

Once the cumulated flux is obtaied from successfully by completing step 1), the mass-transfer function needs to be calculated. For this purpose, a modified deconvolution step which is based on a numerical Laplace transformation, a Laplace-domain integration and a numerical inverse Laplace transformation using the Gaver-Stehfest inversion formula was developed. Scripts for performing these simulations are much more straightforward than for step 1).

3) Convolution of cyclic voltammetry

Once the mass-transfer functions are generated, cyclic voltammetry can be simulated by classical convolutive modeling.

Technical Documentation for the digital simulation step

Software

A set of software is expected to be installed:

- VS Code
- Git
- Developed under Python 3.10
    - Since most of the packages are vanialla and strongly maintained packages, other python and package versions should work as well
    - Simulations have been thoroughly tested for python 3.12

Installation

- Obtain the code as ZIP-file (from Journal) *or* via Git
    - Either: Download the ZIP Folder from the supplementary information of the publication
        - unzip the folder and enter it
        - rename the files example_planar_analyze._bat, example_planar_analyze._sh, example_planar_run._bat and example_planar_run._sh to example_planar_analyze.bat, example_planar_analyze.sh, example_planar_run.bat and example_planar_run.sh, respectively (i.e. rremove the underscore)
        - start a console (CMD) window in the folder of this `README.md` file (right click, 'open in terminal')
            - Note: Make sure you are NOT in the PowerShell (PS on the start of the command line), if you ended up there, simply type `cmd` and press `Enter` to reach the usual windows commandline
    - Or: Download the repository from GitHub
        - navigate to a folder where you want to download the repository to
        - start a console (CMD) window in the folder (right click, 'open in terminal')
        - enter the following command:
            - `git clone https://github.com/Polarographica/DSDC_algorithm_rough_electrodes`
            - enter the folder by
            - `cd DSDC_algorithm_rough_electrodes`
- Create the virtual environment
    - Linux: `python3 -m venv venv`
    - Windows: `python -m venv venv`
        - Note: if you are running a parallelsetup with WSL (Windows Subsystem Linux) you might want to call the environments differently. For example for windows `python -m venv winenv`
- Activate the environment
    - Linux: `source venv/bin/activate`
    - Windows: `venv\Scripts\activate`
        - Note: backslashes in Windows are essential for this command to work
    - on the left hand side of the command line one should see `(venv)`
- Install the packages
    - Linux: `pip3 install -r requirements.txt`
    - Windows: `pip install -r requirements.txt`

Planar stack example: Go for DS-D-C

- Don't forget to activate your environment

    - Linux: `source venv/bin/activate`
    - Windows: `venv\Scripts\activate`

- DS: Run the example script to simulate the DS-step for a planar stack by typing the following command in the console:

    - Linux: `./example_planar_run.sh`

    - Windows: `example_planar_run.bat`

- D+C: This will I) reconstrct the flux from the results of the DS-step, II) D: deconvolute the mass-transfer function and III) C: simulate the CV-profile with the default parameters. For this purpose, run the example script to analyze the planar stack by typing the following command in the console:

    - Linux: `./example_planar_analyze.sh`
    - Windows: `example_planar_analyze.bat`

- Results can be reviewed under `data/output/runs/DGA_DGA_001_Planar_stack_8x8_example_planar_99.99/plots/*`

How to build/use customized electrode stacks

Users can also run simulations on customized electrode stacks by creating a folder under `data/electrode/structures/MY_ELECTRODE/` (Please refer to the .tif guide below).

Similar to the planar example, one needs to execute the respective commands - one for the simulation and one for the analysis.

Simulation: The basic command is `python scripts/main.py --electrodestructure MY_ELECTRODE --runnameaddition MY_TAG --n_power_of_two NUMBER_OF_EXPONENTIAL_SPATIAL_INCREASES --debug True`

Analysis: The basic command is `python scripts/main_analysis.py --electrodestructure MY_ELECTRODE --runnameaddition MY_TAG`

.tif-guite for generating custom electrode structures

Customized electrode stacks may be generated by using very fundamental image processing software. Even Paint will do the job and is chosen as example. For generating an electrode/electrolyte stack, white color stands for electrode material and black color for electrolyte. Therefore, for simulating a planar electrode, simply draw one single purely white image (of e.g. 8x8 pixel), name it 000.tif and store it in a data folder. Subsequently, generate at least ten "electrolyte images" with the same size as the "electrode image". These should be labelled as 001.tif to 010.tif and stored in the very same folder. These particular "just electrolyte images" are required to provide a small "equal incremetation" in the DS-step before the spatial grid is allowed to expand.

Generating the dataset of a rough electrode (for example with one single elevation) is done by setting image 001.tif to be black and white - so partially electrode and partially electrolyte. The ten "just electrolyte images" are then labelled as 002.tif to 012.tif, respectively and stored in the same folder.

Explanation of the parameters

The example scripts (`example_planar_[run,analyze].[bat/sh]`) already give some insight on how different electrode stacks can get simulated and analyzed. In addition to that, the scripts allow for more arguments to adjust the simulation/analysis. To get an extensive list the user can run `python scripts/[main.py,main_analysis.py] --help`.

Below follows an explanation of the various arguments you can pass to the main scripts.

Mandatory

- `--electrodestructure`: Foldername of the electrode structure to simulate under `data/electrode_structures/`
- `--runnameaddition`: A string tag to identify the run in `data/output/runs/` if the same electrode structure is simulated multiple times. For example, if the similarity ratios differ between 99.9 and 99.99
- `--n_power_of_two`: The number of refinement steps used for the simulation. Note that this will tremendously scale the stack size and therefore the demand on computational power. If the stacks bottom area is 8 by 8 voxels (such as the planar example stack), then `--n_power_of_two` = 3 runs the 8x8xdim(z) stack first, followed by an iteration of 16x16x2*dim(z), and finally a run of 32x32x4*dim(z) voxels.

The `continuerun` Parameter

Simulations for higher "upvoxeling"-steps take significantly more time. Sometimes one might want to simulate a large number of electrodes but doesn't need the high resolutions to get an indication of the first results. Or one wants to outsource a higher resolution to a stronger machine. For this reason, the authors included the option to further refine previous simulations, with the `--continuerun True` argument.

Assume, one has simulated the planar stack until `--n_power_of_two` = 3 (with the command `python scripts/main.py --electrodestructure "Planar_stack_8x8" --runnameaddition "example_planar_99.99" --stop_ratio_slope 99.99 --stop_ratio_similarity 99.99 --n_power_of_two 3`) and one judges the results to be interesting enough to go for a higher resolution (say `--n_power_of_two` = 5) it would be wasteful to to resimulate the powers 1-3. Then one can simply extend the command above with the continuerun tag and increase the number of powers: `python scripts/main.py --electrodestructure "Planar_stack_8x8" --runnameaddition "example_planar_99.99" --stop_ratio_slope 99.99 --stop_ratio_similarity 99.99 --n_power_of_two 5 --continuerun True`

Now, once the stack was calculated until a x-y size of e.g. 128 voxels, a personal computer is potentially running out of RAM or the CPU is too weak to further increase the voxelsize to 256 or even 512. In this case, one can simply transfer the whole results folder from `data/output/runs/DGA_Planar_stack_8x8_example_planar_99.99/` to the same directory on a different (yet more powerful) machine and and further refine it there with the command: `python scripts/main.py --electrodestructure "Planar_stack_8x8" --runnameaddition "example_planar_99.99" --stop_ratio_slope 99.99 --stop_ratio_similarity 99.99 --n_power_of_two 7 --continuerun True`

Optional Parameters

- `--delta_x`: spatial increment of the simulation inside of the rough structure
- `--diffusion_coefficient`: diffusion coefficient of the electrochemically active species
- `--time_max`: Maximum time to be simulated
- `--z_grid_expander`: Defines how the z grid will expand
- `--lambda_min`: The minimum lambda value
- `--lambda_max`: The maximum lambda value
- `--stop_ratio_slope`: Similarity ratio of the doulbe-logarithmic slope of the flux to stop the simulation
- `--stop_ratio_similarity`: The ratio of the similarity to stop the simulation

- `--slope_compare_min_time`: The minimum time to compare the slope
- `--np_dtype`: Precision of the simulation (e.g. float32, float64, float64 is recommended)
- `--jit_parallel`: Whether to use parallelization with numba
- `--jit_nopython`: Whether to use nopython with numba

Explanation of the code structure

As above mentioned, the digital-simulation(DS)-deconvolution(D)-convolution(C) algorithm consists of three sucessive steps (DS,D,C), which have been grouped into two categories:

`Digital Simulation`, in the folder (`scripts/DigitalSimulation/`) the reader can find the classes used for this particular step. The heart of this part is the adaptive Douglas-Gunn algorithm (`scripts/DigitalSimulation/DouglasGunnAdaptive.py`), which can simulate the diffusion of rough electrodes with an arbitrary discretized spatial grid. To easily interface with this script, there is a `scripts/main.py` file which provides an argument parser such that the important parameters can be easily configured.

`Analysis`, in the respective folder (`scripts/Analysis/`), the reader can find the three steps (`FluxReconstruction`, `DeconvolutionStep`, `ConvolutionStep`) to perform the analysis. Note, that the `FluxReconstruction` step has been added as part of the analysis. This has to do with the fact that the Digital simulation, has no access to the information of how many additional spatial grid increases will follow. Hence, it is assumed that the Digital Simulation only performs the calculation for each spatial grid setting while storing the raw data. Thus, before one can enter the DeconvolutionStep, it is required to first unify the distinct spatial grid expansions. Subsequently to the flux reconstruction, the `DeconvolutionStep` and `ConvolutionStep`, can then get executed.

Explanation of the three steps

Digital Simulation

The `scripts/DigitalSimulation/DouglasGunnAdaptive.py` script handles all the functions to perform the `DS` step. Under the hood, the `DouglasGunnAdaptive.py` utilizes various factories to utilize numba/jit optimizations. Generally, the interface given by `main.py` should satisfy the needs of most users.

Analysis

`scripts/Analysis/Anlaysis.py` provides an interface to run the whole analysis pipeline with the standard parameter settings. If one wants to customize the analysis part. The three classes can be called separately, to adjust certain parameters. Due to the large variety and many standard settings, the choice has been made to not expose them via the `main_analysis.py` file.

Flux Reconstruction

The `scripts/Analysis/FluxReconstruction.py` class converts the different refinement steps into a single flux and time array for further analysis. We don't expect that frequent adjustments would need to be made here.

Deconvolution

The `scripts/Analysis/DeconvolutionStep.py` class creates the masstransfer function. This is done by loading the cleaned flux and times arrays, performing the Laplacetransformation, the Laplace-domain processing and finally the numerical inversion of the Laplacetransformation. We don't expect that frequent adjustments are needed here.

Convolution

The `scripts/Analysis/ConvolutionStep.py` class finally calculates the cyclic voltammetry (CV) response (which we are actually interested in). It requires the mass-transfer function computed in the previous step.

Note, that some default assumptions regarding the CV-parameters which are provided in the `initalize_cv_params` method of the script have been made. However, the class allows for the initalization of a different dictionary for the parameters to calculate the CV response.

An example has been give in a separate script `CV_with_customparameters.py`, where users can simply import the class and insatatiate it directly with the `plot_path` i.e. the path to the plots folder of the electrode structure you want to analyse, and `cv_dict`.

Before starting a CV simulation, ensure that the mass-transfer function has been calculated previously - in other words: check that the files `M_t_times.npy` and `M_t_values.npy` are in the specified path (i.e. run the analysis once).