User manual for Bernaise

Gaute Linga

May 30, 2018

Bernaise (Binary ElectRohydrodyNAmIc Solver) is a flexible high-level finite element solver of two-phase electrohydrodynamic flow in complex geometries. *Bernaise* is implemented in the Python interface to FEniCS, which effectively utilizes MPI and domain decomposition. The software should therefore suitable for large-scale/high-performance computing.

In this document, we demonstrate briefly how to install *Bernaise*, and how new solvers and problem set-ups can be implemented and added to the Bernaise framework by experienced Python users. For the physical (and industrial) motivation, the underlying equations and a description of the solution schemes, we refer to the paper [2].

1 Prerequisites

To work with *Bernaise*, a basic familiarity with Python programming is needed. Further, the user should be familiar with the finite element method (FEM) and how to solve partial differential equations (PDEs), in particular using FEM and FEniCS through the Python interface. Otherwise, we refer readers to the tutorial by Langtangen and Logg [1]. Experience with the *Oasis* flow solver [4], which targets high-level/high-performance numerical solution of the Navier–Stokes equations, is also an advantage, since *Bernaise* is inspired both in implementation and use of the latter. You can find the *Oasis* git repository on `github.com/mikaem/Oasis`.

On the software side, a working installation of Python 2.7 is needed, with the FEniCS/Dolfin package installed. We refer to the FEniCS project's webpage `fenicsproject.org` for download and installation instructions. Further, several other packages are required to benefit from the full functionality of Bernaise:

- `dolfin` and `mshr`: fundamental components (these are usually parts of the FEniCS install).
- `numpy`: for calculations (necessary).
- `scipy`: for analyzing/visualising data.
- `simplejson`: for parsing the parameters.
- `meshpy`: for generating periodic meshes.
- `matplotlib`: for plotting/visualising data.
- `argparse`: for some utility functionality.

- `h5py` (parallel install): for accessing the output data.
- `mpi4py`: for MPI.
- `pytest`: for testing.
- `scikit-image`: for utilities (mesh creation from images).

2 Installation instructions

You can install *Bernaise* by cloning the Git repository (recommended) or by downloading a packaged version. Packaged versions will be sought to be launched shortly after new stable versions of FEniCS. At the time of writing, version 2017.2.0 is the latest stable FEniCS version, which is compatible with version 1.0 of *Bernaise*.

2.1 Installation via Git

To install *Bernaise* by cloning the GitHub repository.

```
>> git clone https://github.com/gautelinga/BERNAISE.git
>> cd BERNAISE
```

To switch to, e.g., version 1.0 of *Bernaise*, you can type:

```
>> git checkout v1.0
```

which gives you the (as of writing) latest "stable" realease.

2.2 Installation of the packaged version

To install *Bernaise* from a packaged file, you can navigate to `github.com/gautelinga/Bernaise/releases`. In a Unix terminal, you can then install by performing the following commands:

```
>> wget https://github.com/gautelinga/Bernaise/archive/v1.0.tar.gz
>> tar -xvf v1.0.tar.gz
>> cd Bernaise-v1.0
```

2.3 Installing the dependencies

The usually fastest way to install up-to-date dependencies is via `pip`:

```
>> pip install numpy scipy simplejson meshpy matplotlib \
>> argparse mpi4py pytest skimage
```

Guides to install `h5py` in *parallel* can be found several places on the web; one way is the following:

```
>> sudo apt-get install libhdf5-openmpi-10 libhdf5-openmpi-dev hdf5-tools
>> git clone https://github.com/h5py/h5py.git
>> cd h5py
>> git checkout 2.6.0 #(or a newer version)
>> export CC=mpicc.openmpi
>> python setup.py configure --mpi --hdf5=/usr/lib/x86_64-linux-gnu/hdf5/openmpi/
>> python setup.py build
>> sudo python setup.py install
```

If you do not want to install systemwide, the last `sudo` can be skipped.

3 Code structure

Bernaise is designed as a Python package, and its main executable script for running simulations is `sauce.py`. For postprocessing, the main executable script is `postprocessing.py`. The base level of the directory structure is shown in Fig. 1.

```
BERNAISE
├─ README.md
├─ sauce.py
├─ postprocess.py
├─ common
│   └─ ...
├─ problems
│   └─ ...
├─ solvers
│   └─ ...
├─ utilities
│   └─ ...
├─ tests
│   └─ ...
├─ data
│   └─ ...
├─ meshes
│   └─ ...
├─ scripts
│   └─ ...
└─ ...
```

Figure 1: First level of the directory structure of Bernaise.

3.1 `sauce.py`

The task of the main module `sauce.py` is initializing the necessary variables to run a simulation, importing routines from the specified `problem` and `solver`, to iterate the solver in time, and to output and store data at appropriate times. In particular, various `hooks` are called at various places in the code, where users can perform actions of choice within the code.

A simulation is typically run from a terminal, pointing to the *Bernaise* directory, using the command

```
>> python sauce.py problem=charged_droplet
```

where `charged_droplet` may be exchanged with another problem script of choice. The main script `sauce.py` fetches a `problem`, from the folder `problems` (see below), and connects it with the `solver`, fetched from the folder `solvers` (see below). It sets up the finite element problem with all the given parameters, initializes the finite element fields with the specified initial state, and solves it with the specified boundary condition at each time step, until the specified (physical) simulation time `T` is exceeded.

All default parameters in a problem by specifying an additional keyword from the command line; for example, the simulation time can be set to 1000 by running the command:

```
>> python sauce.py problem=charged_droplet T=1000
```

After every given interval of steps, specified by the parameter `checkpoint_interval`, a checkpoint is stored, including all fields, and all problem parameters at the time of writing to file. The checkpoint can be loaded, and the simulation can be continued, by running the command:

```
>> python sauce.py problem=charged_droplet \
   restart_folder=results_charged_droplet/1/Checkpoint/
```

where the `restart_folder` points to an appropriate checkpoint folder. Here, the problem parameters stored within the checkpoint have precedence over the default parameters given in the `problem` script. Further, any parameters specified by command line keywords have precedence over the checkpoint parameters.

3.2 `problems`

The `problems` submodule (in the folder `problems`) contains the various problems that can be run in *Bernaise*. An incomplete list of problems that are currently implemented is shown in Fig. 2.

```
problems
├─ __init__.py
├─ charged_droplet.py
├─ charged_droplets.py
├─ charhed_droplets_3D.py
├─ taylorgreen.py
├─ snoevsen.py
├─ charged_droplets_3D.py
├─ barbell_capilar.py
├─ dielectric.py
├─ dolphin.py
├─ electrowetting.py
├─ hourglass.py
├─ intrusion_bulk.py
├─ porous.py
├─ simple.py
├─ simple_3D.py
├─ single_cell.py
├─ single_reaction.py
├─ single_taylorgreen.py
└─ ...
```

Figure 2: The directory structure of the `problems` submodule.

Code that is shared between various solvers, in particular default values of the `hooks`, are placed in the top level `__init__.py` script. For example, a list of "base elements" is defined:

```python
# Default base elements
# Format: name : (family, degree, is_vector)
base_elements = dict(u=["Lagrange", 2, True],
                     p=["Lagrange", 1, False],
                     phi=["Lagrange", 1, False],
                     g=["Lagrange", 1, False],
                     c=["Lagrange", 1, False],
                     V=["Lagrange", 1, False],
                     p0=["Real", 0, False],
                     c0=["Real", 0, False],
                     V0=["Real", 0, False])
```

This `dict` defines which finite elements are automatically created by the `sauce.py` script (and can be overruled in the particular `problem`).

A list of default `parameters` is also defined:

```
# Set default parameters
parameters = dict(
    folder="results",  # default folder to store results in
    info_intv=10,
    use_iterative_solvers=False,
    use_pressure_stabilization=False,
    dump_subdomains=False,
    V_lagrange=False,
    p_lagrange=False,
    base_elements=base_elements,
    c_cutoff=0.,
    q_rhs=dict(),
    EC_scheme="NL2",
    grav_dir=[1., 0],
    pf_mobility_coeff=1.,
    grav_const=0.,
    surface_tension=0.,
    interface_thickness=0.,
    reactions=[],
    density_per_concentration=None,
    viscosity_per_concentration=None,
    testing=False,
    tstep=0
)
```

We will not go into detail on the meaning of all these entries; they should be explained in the source code or in [2]. These can also be overruled in a `problem`; in particular, the function `problem` in any problem script is required to return a `dict`. We shall look at a concrete example in Sec. 5.

Further, the following functions can be overruled:

- `constrained_domain`: Returns e.g. periodic domain.
- `initialize`: Initialize solution, i.e., the initial conditions.
- `create_bcs`: Returns a `dict` of Dirichlet boundary conditions.
- `start_hook`: Called just before entering the time loop.
- `tstep_hook`: Called in the beginning of timestep loop.
- `end_hook`: Called just before program ends.
- `import_problem_hook`: Called after importing problem.

- `rhs_source`: for adding source terms, e.g., for validation purposes.
- `pf_mobility`: default phase field mobility function (see [2]).

Note that all of these functions (with the exception of `pf_mobility`) can import *any parameter returned by the* `problem` *function*, which will be exemplified in Sec. 5.

3.3 `solvers`

Similarly to the `problems` submodule, there are several solvers implemented in the `solvers` module. In particular, a solver can be defined in the `dict` returned by the `problem` function in a problem, or it can be set from the command line by running:

```
>> python sauce.py problem=charged_dropet solver=basicnewton
```

if, by chance, we wanted to use the `basicnewton` scheme instead of `basic`. A list of implemented solver is shown Fig. 3.

```
solvers
├─ __init__.py
├─ basic.py
├─ basicnewton.py
├─ fracstep.py
├─ stable_single.py
├─ stable_single_fracstep.py
└─ ...
```

Figure 3: The directory structure of the `solvers` submodule.

The solvers that start with the name `stable_single` only support single-phase flow, and have been documented in Ref. [3].

Code that is shared across the different solvers is defined in the top level file `__init__.py`. In common with its counterpart in the `problems` submodule, `__init__.py` defines a set of placeholder functions that should be overloaded in the specific `solver` instance.

- `get_subproblems`: Returns `dict` of subproblems as defined by the solver.
- `setup`: Sets up all equations that should be solved. Returns `dict` of solvers.
- `solve`: Solves equations at each timestep.
- `update`: Update work arrays at the end of timestep.

We will examplify these for the `basic` solver in Sec. 4.

3.4 `utilities`

Bernaise comes with a set of utility scripts that are located in the `utilities` folder. This subdirectory is shown Fig. 4.

```
utilities
 ├─ extract_polygons.py
 ├─ generate_mesh.py
 ├─ get_info.py
 ├─ TimeSeries.py
 └─ ...
```

Figure 4: The directory structure of the `utilities` submodule.

Here, `generate_mesh.py` is somewhat similar to `sauce.py` in that it pulls in a required mesh generation script from the folder `utilities/mesh_scripts`. For example, an hourglass mesh can be generated by navigating into the `utilities` folder and running the command:

```
>> python generate_mesh.py mesh=hourglass
```

An hourglass mesh will then be created in the `meshes` folder (see Fig. 1. An incomplete list of meshes that can be generated, and thus are stored in the latter folder, is the following:

- `barbell_capilar`
- `extended_polygon.py`
- `hourglass.py`
- `periodic_porous.py`
- `snoevsen.py`
- `straight_capilar.py`

`TimeSeries.py` is fundamental for the post-processing procedures, which will be covered in Sec. 3.5.

3.5 `postprocessing.py`

4 The `basic` solver

Now we briefly explain the implementation of the `basic` solver.

4.1 get_subproblems

The overloaded `get_subproblems` function in `basic` consists of:

```python
def get_subproblems(base_elements, solutes, p_lagrange,
                    enable_NS, enable_PF, enable_EC,
                    **namespace):
    subproblems = dict()
    if enable_NS:
        subproblems["NS"] = [dict(name="u", element="u"),
                             dict(name="p", element="p")]
        if p_lagrange:
            subproblems["NS"].append(dict(name="p0", element="p0"))
    if enable_PF:
        subproblems["PF"] = [dict(name="phi", element="phi"),
                             dict(name="g", element="g")]
    if enable_EC:
        subproblems["EC"] = ([dict(name=solute[0], element="c")
                              for solute in solutes]
                             + [dict(name="V", element="V")])
    return subproblems
```

First, note that any parameter that has been defined in the namespace, can inter into the list of arguments. Here, `base_elements` were defined in the `problem` that has called the `basic` solver, and the entries of the subproblems `dict` are the subproblems that the solver splits the full problem into, here `NS` (Navier–Stokes), `PF` (phase field) and `EC` (electrochemistry). Within each entry, a list of `dicts` that make up mixed finite elements is specified. The `name` key gives the name of the field and `element` points to the element whose key is the value. The `solutes` parameter is an array of arrays which give (1) name, (2) valency, (3) diffusivity in phase 1, (4) diffusivity in phase 2, (5) solubility energy in phase 1, and (6) solubility energy in phase 2. See Sec. 5 for an example. The boolean parameter `p_lagrange` states whether a Lagrange multiplier should be used to fix the pressure gauge (this enters as an additional degree of freedom in the subproblem). The parameters `enable_NS`, `enable_PF`, and `enable_EC` are boolean values that determine whether the various subproblems should be enabled.

5 A test problem

References

[1] H. P. Langtangen and A. Logg. *Solving PDEs in Python.* Springer, 2017. ISBN 978-3-319-52461-0. doi: 10.1007/978-3-319-52462-7.

[2] G. Linga, A. Bolet, and J. Mathiesen. Bernaise: A flexible framework for simulating two-phase electrohydrodynamic flows in complex domains. Submitted, 2018.

[3] G. Linga, A. Bolet, and J. Mathiesen. Transient electrohydrodynamic flow

with concentration dependent fluid properties: modelling and energy-stable schemes. Submitted, 2018.

[4] M. Mortensen and K. Valen-Sendstad. Oasis: A high-level/high-performance open source Navier–Stokes solver. *Comput. Phys. Commun.*, 188:177–188, 2015. doi: 10.1016/j.cpc.2014.10.026.