cap Documentation

Release dev

Damien Lebrun-Grandie

Sep 19, 2017

Contents

Contents:

CHAPTER 1

Getting started

Overview

Cap is a library for modeling energy storage devices. Its core is implemented in C++ but Python wrappers are also available.

Cap provides:

1. energy storage device models
2. electrochemical measurement techniques

Guidelines for installation are *available*. Note that it is **not** necessary to build Cap from source to use it. For instructions on how to use Cap without installing it, refer to the following *section*.

Alternative to the full install procedure

All you need is a working installation of Docker. Follow the Docker Engine installation guide for details on how to install it on your machine. It is supported on Linux, Cloud, Windows, and OS X.

The following command starts a Docker container with a Jupyter Notebook server listening for HTTP connections on port 8888. It mounts the present working directory, `$PWD`, into the container at `/notebooks`, which is set as the Jupyter Notebook startup folder. It has pycap already installed on it and comes with a few notebooks as example.

```
$ docker run --rm -it \
      -p 8888:8888 \
      -v $PWD:/notebooks \
      dalg24/cap
```

Open your web browser and follow `http://localhost:8888`.

CHAPTER 2

Installation

This section provide guidelines for installing Cap from source.

Note that it is **not** necessary to build Cap from source to use it. Refer to the *Docker* section for instructions on how to pull the latest image of Cap.

Third-party libraries

Packages	Dependency	Version
MPI	Required	
Python	Optional	
Boost	Required	1.59.0
deal.II with p4est/Trilinos	Optional	8.5.0

Cap and its dependencies may be built using spack. You would need to install the following packages:

```
$ spack install boost +graph +icu +mpi +python
$ spack install trilinos ~hypre ~mumps +boost \
    ^boost+graph+icu+mpi+python
$ spack install dealii~arpack~gsl~oce~petsc+trilinos+mpi \
    ^trilinos~hypre~mumps+boost ^boost+graph+icu+mpi+python
$ spack install py-mpi4py
$ spack install py-matplotlib
$ spack install py-h5py
```

Before buiding Cap, you would then need to load the following modules: dealii, boost, mpi, cmake, python, py-mpi4py, py-matplotlib, py-parsing, py-numpy, and py-h5py.

Message Passing Interface (MPI)

Cap should be working with any of the MPI implementations. It has only been tested with Open MPI, MPICH, and Intel MPI.

Boost

Boost version 1.59.0 or later is required. Boost can be downloaded from here. Make sure to install **all** the libraries. Do not forget to add the `using mpi ;` directive to your `project-config.jam` file before building.

deal.II

The open source finite element library deal.II is optional. It is only required to work with energy storage devices of type `SuperCapacitor`. Version 8.4.0 or later compiled with C++14/MPI/Boost/p4est/Trilinos support is required. The development sources can be found here. Please refer to the deal.II documentation to see how to install p4est and Trilinos.

Install Cap from source

Get the source:

```
$ git clone https://github.com/ORNL-CEES/Cap.git && cd Cap
```

Create a `configure_cap.sh` script such as:

```
#!/usr/bin/env bash

EXTRA_ARGS=$@

cmake \
    -D CMAKE_INSTALL_PREFIX=<your/install/prefix/here> \
    -D BOOST_DIR=<path/to/boost> \
    -D ENABLE_DEAL_II=ON \
    -D DEAL_II_DIR=<path/to/dealii> \
    $EXTRA_ARGS \
    ..
```

Configure, build and install:

```
$ mkdir build && cd build
$ vi configure_cap.sh
$ chmod +x configure_cap.sh
$ ./configure_cap.sh
$ make -j<N> && make install
```

Run the tests:

```
$ ctest -j<N>
```

Enable the Python wrappers

To build the Python wrappers Cap must be configured with an extra flag `-D ENABLE_PYTHON=ON`. It is recommended to use Python 3.X but Cap has been successfully built with Python 2.X in the past.

```
$ ../configure_cap.sh -D ENABLE_PYTHON=ON
$ make install
```

Prepend the Python install directory to your `PYTHONPATH` environment variable in order to import the pycap module from your Python interpreter.

```
$ export PYTHONPATH=<cap/install/prefix>/lib/pythonX.Y/site-packages:${PYTHONPATH}
```

`X.Y` stands for the version of Python that was used to build Cap, for example 2.7 or 3.5.

Launch Python and try:

```
>>> import pycap
>>> help(pycap)
```

Note that a number of Python packages are required to use pycap: numpy, matplotlib, mpi4py, and h5py.

Build this documentation

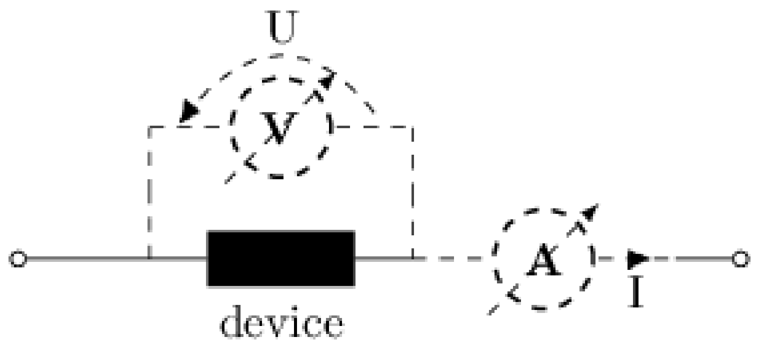
Run the configuration script with the extra flag:

```
$ ../configure_cap.sh -D ENABLE_DOCUMENTATION=ON
```

Open the file `index.html` in the directory `docs/html`.

CHAPTER 3

Energy storage devices



Only the electrical current I and voltage U of the device are measurable. Several operating conditions are possibles. One may want to impose:

- The voltage U across the device.
- The electrical current I that flows through it.
- The load $R = U/I$ the device is subject to.
- The power $P = UI$.

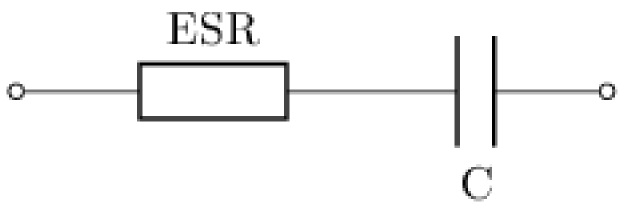
The class `pycap.EnergyStorageDevice` is an abstract representation for an energy storage device. It can evolve in time at various operating conditions and return the voltage drop across itself and the electrical current that flows through it.

The rest of this section describes the energy storage devices that are available in Cap, namely:

- Equivalent circuits
- Supercapacitors

Equivalent circuits

Series RC



A resistor and a capacitor are connected in series (denoted ESR and C in the figure above).

```
type                    SeriesRC
series_resistance         5.0e-3 ; [ohm]
capacitance               3.0    ; [fahrad]
```

Above is the database to build a 3 F capacitor in series with a 50 mΩ resistance.

$$U = U_C + RI$$

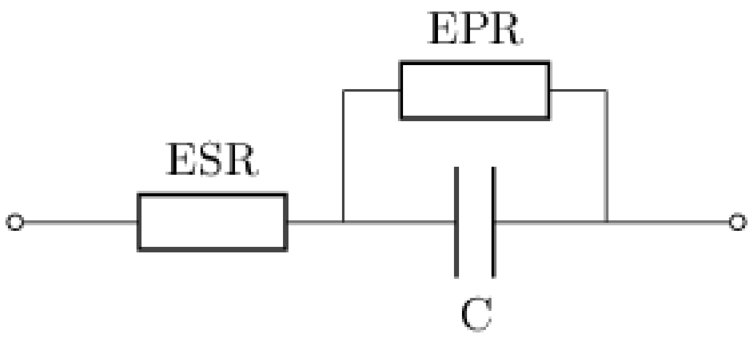
$$I = C\frac{dU_C}{dt}$$

U_C stands for the voltage across the capacitor. Its capacitance, C, represents its ability to store electric charge. The equivalent series resistance, R, add a real component to the impedance of the circuit:

$$Z = \frac{1}{jC\omega} + R$$

As the frequency goes to infinity, the capacitive impedance approaches zero and R becomes significant.

Parallel RC



An extra resistance is placed in parallel of the capacitor. It can be instantiated by the following database.

```
type                     ParallelRC
parallel_resistance          2.5e+6 ; [ohm]
series_resistance           50.0e-3 ; [ohm]
capacitance                  3.0    ; [fahrad]
```

`type` has been changed from `SeriesRC` to `ParallelRC`. A 2.5 MΩ leakage resistance is specified.

$$U = U_C + RI$$

$$I = C\frac{dU_C}{dt} + \frac{U_C}{R_L}$$

R_L corresponds to the "leakage" resistance in parallel with the capacitor. Low values of R_L imply high leakage currents which means the capacitor is not able to hold is charge. The circuit complex impedance is given by:

$$Z = \frac{R_L}{1 + jR_L C\omega} + R$$

Supercapacitors

`type` is set to `SuperCapacitor`. `dim` is used to select two- or three-dimensional simulations.

```
device {
    type SuperCapacitor
    dim 2
    geometry {
        [...]
    }
    material_properties {
        [...]
    }
}
```

Geometry

```
geometry {
    type supercapacitor

    anode_collector_thickness    5.0e-4 ; [centimeter]
    anode_electrode_thickness   50.0e-4 ; [centimeter]
    separator_thickness         25.0e-4 ; [centimeter]
    cathode_electrode_thickness 50.0e-4 ; [centimeter]
    cathode_collector_thickness  5.0e-4 ; [centimeter]
    geometric_area              25.0e-2 ; [square centimeter]
}
```

The thickness of each layer in the sandwich (anode collector, anode electrode, separator, cathode electrode, cathode current collector) can be adjusted independently from one another. The specified cross-sectional area applies to the whole stack.

Governing equations

collector	electrode	separator
$i_1 = -\sigma \nabla \Phi_1$ $\nabla \cdot i_1 = 0$	$i_1 = -\sigma \nabla \Phi_1$ $i_2 = -\kappa \nabla \Phi_2$ $-\nabla \cdot i_1 = \nabla \cdot i_2 = a i_n$	$i_2 = -\kappa \nabla \Phi_2$ $\nabla \cdot i_2 = 0$

collector-electrode interface	electrode-separator interface
$0 = -\kappa \left.\frac{\partial \Phi_2}{\partial n}\right\|_e$ $-\sigma \left.\frac{\partial \Phi_1}{\partial n}\right\|_c = -\sigma \left.\frac{\partial \Phi_1}{\partial n}\right\|_e$	$-\kappa \left.\frac{\partial \Phi_2}{\partial n}\right\|_e = -\kappa \left.\frac{\partial \Phi_2}{\partial n}\right\|_s$ $-\sigma \left.\frac{\partial \Phi_1}{\partial n}\right\|_e = 0$

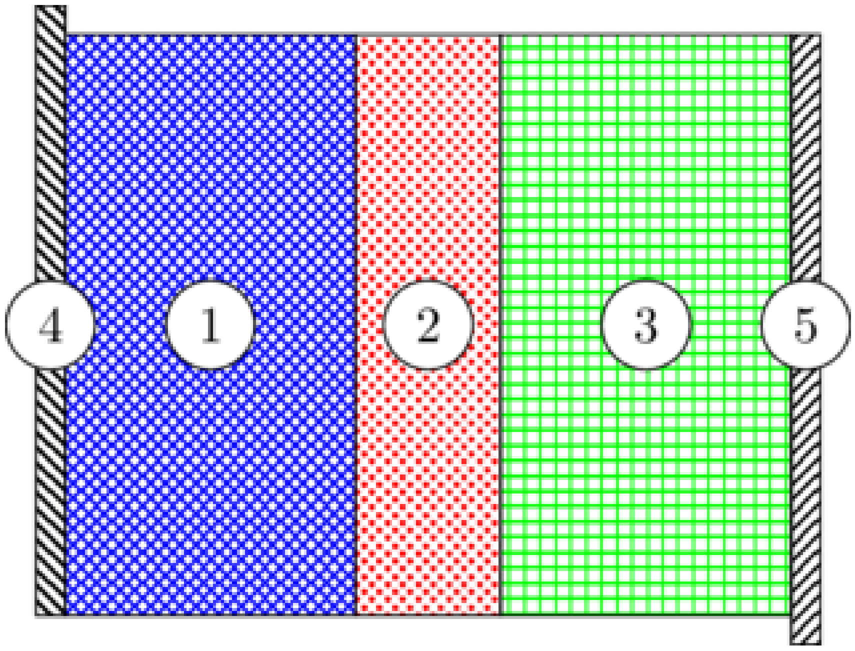


Fig. 3.1: Schematic representation of the supercapacitor conventional sandwich-like configuration. 1: anode electrode, 2: separator, 3: cathode electrode, 4: anode collector, 5: cathode collector.

boundary collector tab
$\Phi_1 = U$ or $-\sigma\frac{\partial\Phi_1}{\partial n} = I/S$ or $-\sigma\frac{\partial\Phi_1}{\partial n}\Phi_1 = P/S$ or $-\sigma\frac{\partial\Phi_1}{\partial n}RS = \Phi_1$

Ignoring the influence of the electrolyte concentration, the current density in the matrix and solution phases can be expressed by Ohm's law as

$$i_1 = -\sigma\nabla\Phi_1$$
$$i_2 = -\kappa\nabla\Phi_2$$

i and Φ represent current density and potential; subscript indices 1 and 2 denote respectively the solid and the liquid phases. σ and κ are the matrix and solution phase conductivities.

The total current density is given by $i = i_1 + i_2$. Conservation of charge dictates that

$$-\nabla \cdot i_1 = \nabla \cdot i_2 = a i_n$$

where a is the interfacial area per unit volume and the current transferred from the matrix phase to the electrolyte i_n is the sum of the double-layer the faradaic currents

$$i_n = C\frac{\partial}{\partial t}\left(\Phi_1 - \Phi_2\right) + i_0\left(e^{\frac{\alpha_a F}{RT}\eta} - e^{-\frac{\alpha_c F}{RT}\eta}\right)$$

C is the double-layer capacitance. i_0 is the exchange current density, α_a and α_c the anodic and cathodic charge transfer coefficients, respectively. F, R, and T stand for Faraday's constant, the universal gas constant and temperature. η is the overpotential relative to the equilibrium potential U_{eq}

$$\eta = \Phi_1 - \Phi_2 - U_{eq}$$

Material properties

```
material_properties {
    anode {
        type            porous_electrode
        matrix_phase    electrode_material
        solution_phase electrolyte
    }
    cathode {
        type            porous_electrode
        matrix_phase    electrode_material
        solution_phase electrolyte
    }
    separator {
        type            permeable_membrane
        matrix_phase    separator_material
        solution_phase electrolyte
    }
    collector {
        type            current_collector
        metal_foil      collector_material
    }

    separator_material {
        void_volume_fraction            0.6        ;
        tortuosity_factor               1.29       ;
        pores_characteristic_dimension  1.5e-7     ; [centimeter]
        pores_geometry_factor           2.0        ;
        mass_density                    3.2        ; [gram per cubic centimeter]
        heat_capacity                   1.2528e3   ; [joule per kilogram kelvin]
        thermal_conductivity            0.0019e2   ; [watt per meter kelvin]
    }
    electrode_material {
        differential_capacitance        3.134      ; [microfarad per square␣
↪centimeter]
        exchange_current_density        7.463e-10  ; [ampere per square centimeter]
        void_volume_fraction            0.67       ;
        tortuosity_factor               2.3        ;
        pores_characteristic_dimension  1.5e-7     ; [centimeter]
        pores_geometry_factor           2.0        ;
        mass_density                    2.3        ; [gram per cubic centimeter]
        electrical_resistivity          1.92       ; [ohm centimeter]
        heat_capacity                   0.93e3     ; [joule per kilogram kelvin]
        thermal_conductivity            0.0011e2   ; [watt per meter kelvin]
    }
    collector_material {
        mass_density                    2.7        ; [gram per cubic centimeter]
        electrical_resistivity         28.2e-7     ; [ohm centimeter]
        heat_capacity                   2.7e3      ; [joule per kilogram kelvin]
        thermal_conductivity          237.0        ; [watt per meter kelvin]
    }
    electrolyte {
        mass_density                    1.2        ; [gram per cubic centimeter]
        electrical_resistivity          1.49e3     ; [ohm centimeter]
        heat_capacity                   0.0        ; [joule per kilogram kelvin]
        thermal_conductivity            0.0        ; [watt per meter kelvin]
    }
}
```

The specific surface area per unit volume a is estimated using

$$a = \frac{(1+\zeta)\varepsilon}{r}$$

where ζ is the pore's geometry factor ($\zeta = 2$ for spheres, 1 for cylinders, and 0 for slabs) and r is the pore's characteristic dimension. [M. W. Verbrugge and B. J. Koch, J. Electrochem. Soc., 150, A374 2003]

The solution electrical conductivity κ incorporates the effect of porosity and tortuosity

$$\kappa = \frac{\kappa_\infty \varepsilon}{\Gamma}$$

where κ_∞ is the liquid phase (free solution) conductivity, ε is the void volume fraction, and κ is the tortuosity factor.

The solid phase conductivity is also corrected for porosity (and tortuosity???)

$$\sigma = \sigma_\infty (1 - \varepsilon)$$

Batteries

NOT IMPLEMENTED

CHAPTER 4

Electrochemical techniques

Cyclic charge discharge

Cyclic Charge-Discharge is a common technique used to test the performance and cycle-life of energy storage devices. Most often, the charge and discharge are conducted at constant current until a set voltage is reached.

The following implements 4 cycles of a repetitive loop through several steps:

1. constant current charge at 0.5 A until voltage reaches a 2.1 V limit
2. potentiostatic hold until the current falls below 1 mA for a maximum duration time of 3 min
3. rest at open circuit potential for 2 s
4. constant load discharge at $3.33\ \Omega$ to 0.7 V
5. rest at open circuit potential for 5 s

```
from pycap import PropertyTree,CyclicChargeDischarge,EnergyStorageDevice

# setup the experiment
ptree=PropertyTree()
ptree.put_string('start_with','charge')
ptree.put_int    ('cycles',4)
ptree.put_double('time_step',0.01)

ptree.put_string('charge_mode','constant_current')
ptree.put_double('charge_current',0.5)
ptree.put_string('charge_stop_at_1','voltage_greater_than')
ptree.put_double('charge_voltage_limit',2.1)
ptree.put_bool   ('charge_voltage_finish',True)
ptree.put_double('charge_voltage_finish_max_time',180)
ptree.put_double('charge_voltage_finish_current_limit',1e-3)
ptree.put_double('charge_rest_time',2)

ptree.put_string('discharge_mode','constant_load')
ptree.put_double('discharge_load',3.33)
ptree.put_string('discharge_stop_at_1','voltage_less_than')
ptree.put_double('discharge_voltage_limit',0.7)
ptree.put_double('discharge_rest_time',5)

ccd=CyclicChargeDischarge(ptree)
```

The property tree is populated interactively here but it can parse directly an input file. Please refer to other examples.

The CCD experiment can be started with a `charge` or a `discharge` step. The length of the test is defined by the cycle number and the loop end criteria.

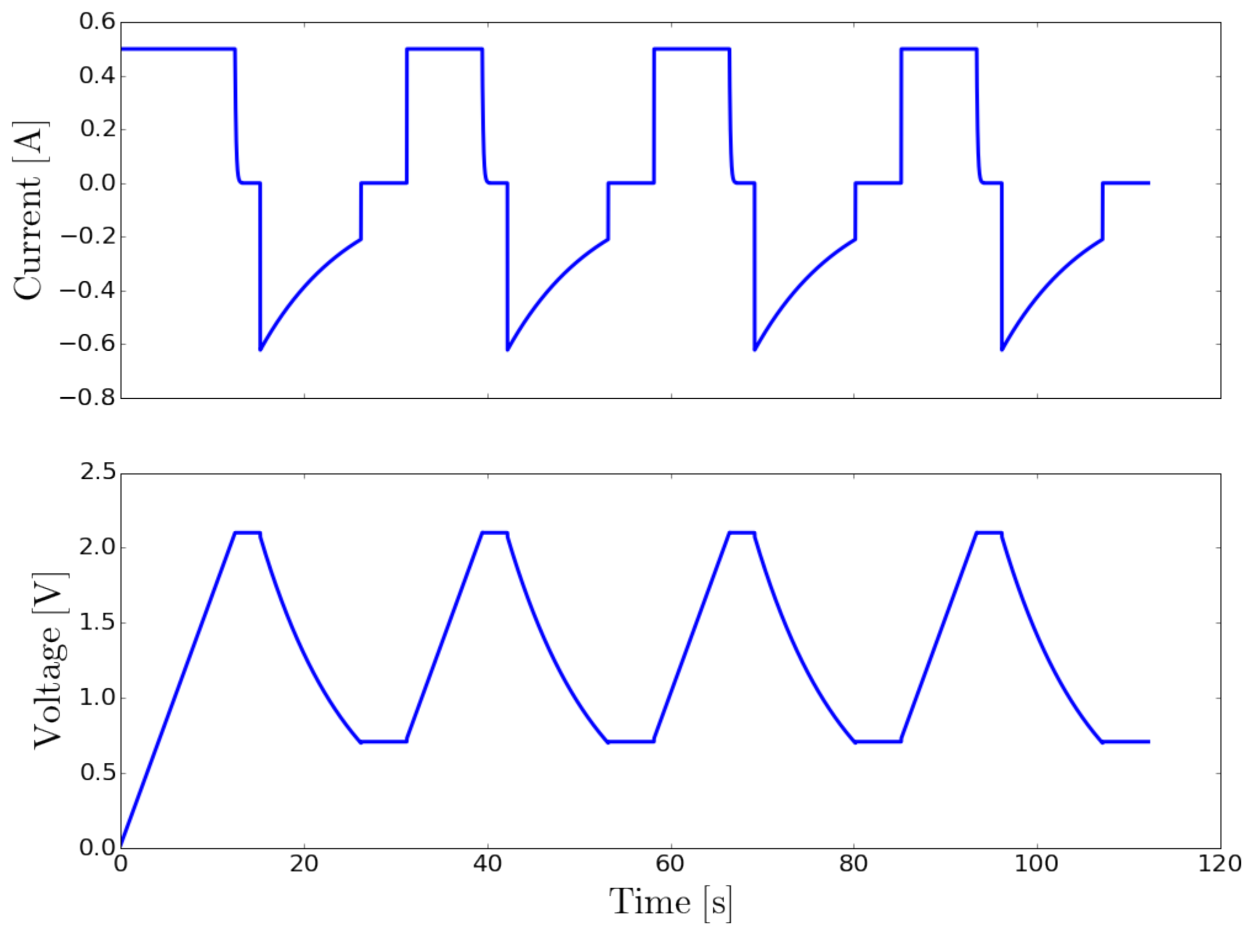
The charge mode can be `constant_current`, `constant_voltage`, or `constant_power`. Two end criteria can be selected although only one is required. **Note that they are no safeguards and poor end criteria will produce infinite loops!** If `voltage_finish` is enabled (default value is `False`), the charge step proceeds to a potentiostatic step that ends after reaching the specified time `voltage_finish_max_time` or when the current falls between the limiting value `voltage_finish_current_limit` (absolute value). The voltage finish step makes little sense in case of a constant voltage charge and therefore is not allowed. The charge ends with an optional rest time period before proceeding with the next step.

The discharge process can be perfomed in four different modes: `contant_current`, `contant_voltage`, `constant_power`, or `constant_load`. End criteria must be chosen carfully here as well.

Let's build an energy storage device, here a simple series RC circuit, with a $40\ \mathrm{m}\Omega$ resistor and a $3\ \mathrm{F}$ capacitor, and run the experiment.

```
# build an energy storage device
ptree=PropertyTree()
ptree.put_string('type','SeriesRC')
ptree.put_double('series_resistance',40e-3)
ptree.put_double('capacitance',3)
device=EnergyStorageDevice(ptree)

from pycap import initialize_data,plot_data

# run the experiment and visualize the measured data
data=initialize_data()
steps=ccd.run(device,data)

print "%d steps"%steps

%matplotlib inline
plot_data(data)
```

11213 time steps ($\Delta t = 0.01\ \mathrm{s}$) are required to complete the CCD experiment. Below are plotted the measured current and voltage data versus time.



Cyclic voltammetry

Cyclic Voltammetry (CV) is a widely-used electrochemical technique to investigate energy storage devices. It consists in measuring the current while varying linearly the voltage back and forth over a given range.

The voltage sweep applied to the device creates a current given by

$$I = C\frac{dU}{dt}$$

where I is the current in ampere and $\frac{dU}{dt}$ is the scan rate of the voltage ramp.

The voltage scan rates for testing energy storage devices are usually between $0.1\ \mathrm{mV/s}$ and $1\ \mathrm{V/s}$. Scan rates at the lower end of this range allow slow processes to occur; fast scans often show lower capacitance than slower scans and may produce large currents on high-value capacitors.

```
from pycap import PropertyTree, CyclicVoltammetry

# setup the experiment
ptree = PropertyTree()
ptree.put_double('initial_voltage', 0) # volt
ptree.put_double('final_voltage', 0) # volt
ptree.put_double('scan_limit_1', 2.4) # volt
ptree.put_double('scan_limit_2', -0.5) # volt
ptree.put_double('scan_rate', 100e-3) # volt per second
ptree.put_double('step_size', 5e-3) # volt
ptree.put_int('cycles', 2)
cv = CyclicVoltammetry(ptree)
```

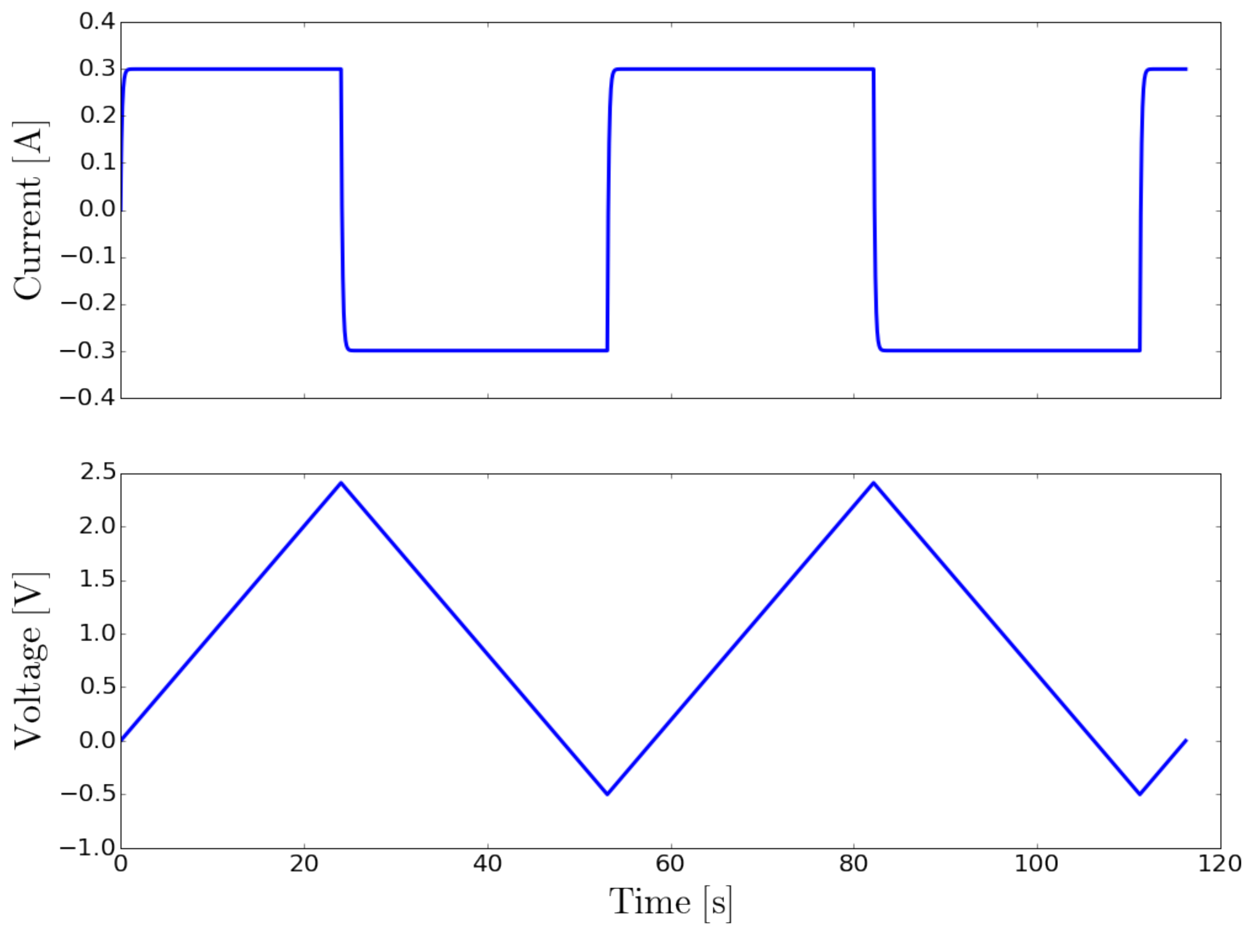
Four parameters define the CV sweep range: The scan starts at `initial_voltage`, ramps to `scan_limit_1`, reverses and goes to `scan_limit_2`. Additional cycles start and end at `scan_limit_2`. The scan ends at `final_voltage`. Here, the sweep range is [2.4 V, -0.5 V]. It both starts and finishes at 0 V.

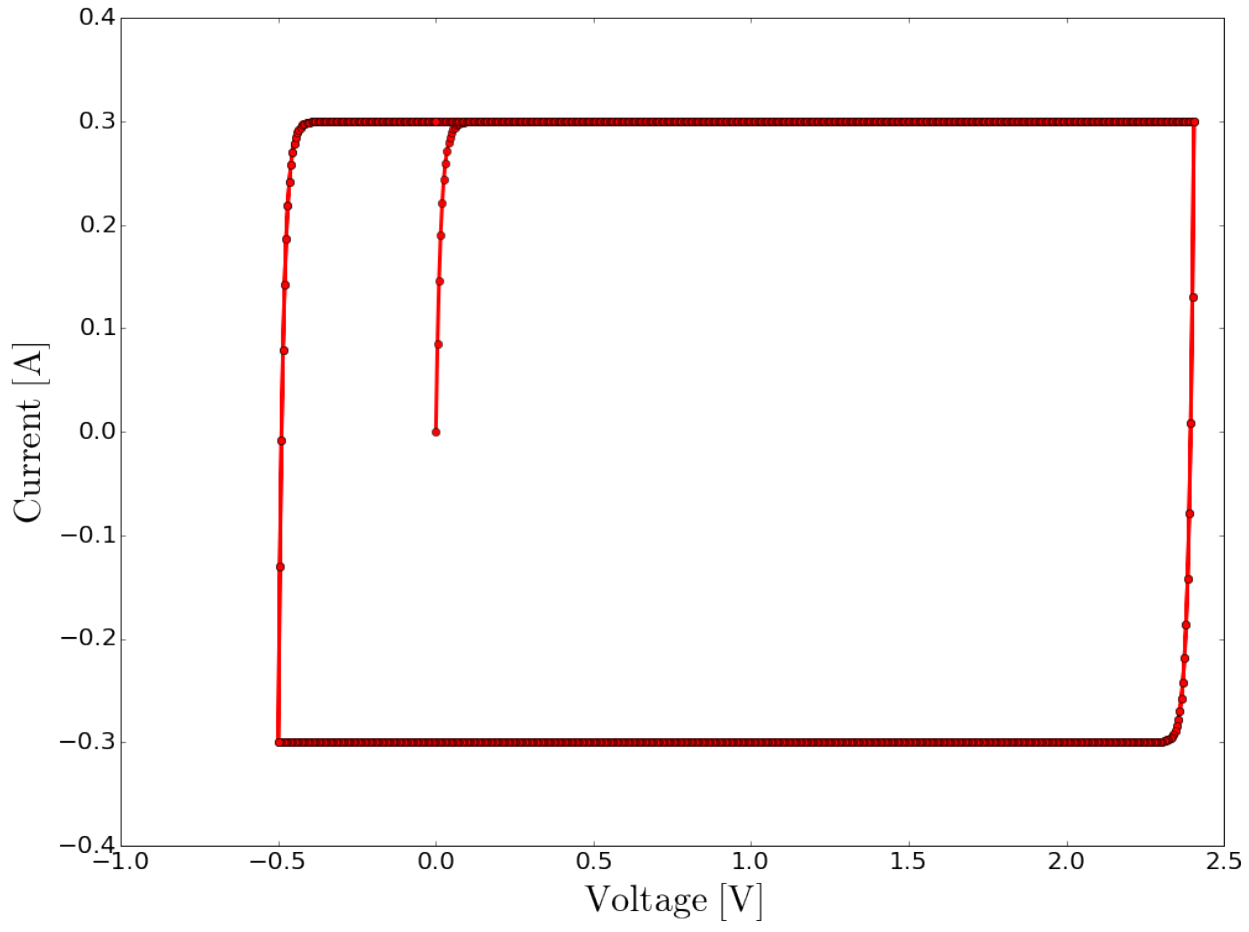
The rate of voltage change over time $\frac{dU}{dt}$ is specified using `scan_rate` which is here set to $100\ \mathrm{mV/s}$. The linear ramp is imposed in increments of $5\ \mathrm{mV}$. The number of sweep is controlled by `cycles`.

Here we run the experiment with a 3 F capacitor in series with a $50\ \mathrm{m\Omega}$ resistor.

```
# build an energy storage device
ptree=PropertyTree()
ptree.put_string('type','SeriesRC')
ptree.put_double('capacitance',3)
ptree.put_double('series_resistance',50e-3)
device=EnergyStorageDevice(ptree)

from pycap import initialize_data,report_data,plot_data
from pycap import plot_cyclic_voltammogram

# run the experiment and visualize the measured data
data=initialize_data()
steps=cv.run(device,data)

print "%d steps"%steps

%matplotlib inline
plot_data(data)
plot_cyclic_voltammogram(data)
```

```
2320 steps
```


Current [A]
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4
Voltage [V]
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
0
20
40
60
80
100
120
Time [s]



On the CV plot (current on the y-axis and voltage on the x-axis), we read

$$I = C\frac{dU}{dt} = 300\,\mathrm{mA}$$

as expected for a 3 F capacitor. For an ideal capacitor (i.e. no equivalent series resistance), the plot would be a perfect rectangle. The resistor causes the slow rise in the current at the scan's start and rounds two corners of the rectangle. The time constant $\tau = RC$ controls rounding of corners.

Electrochemical impedance spectroscopy

Electrochemical Impedance Spectroscopy (EIS) is a powerful experimental method for characterizing electrochemical systems. This technique measures the complex impedance of the device over a range of frequencies.

A sinusoidal excitation signal (potential or current) is applied:

$$E = E_0 + \sum_k E_k \sin(\omega_k t + \varphi_k)$$

That signal consists in the superposition of AC sine waves with amplitude E_k, angular frequency $\omega_k = 2\pi k f$, and phase shift ϕ_k. E_0 is the DC component.

```
; `eis.info` file

frequency_upper_limit  1e+3 ; hertz
```

```
frequency_lower_limit  1e-2 ; hertz
steps_per_decade          6

cycles                    2
ignore_cycles             1
steps_per_cycle         128

harmonics                 1
dc_voltage                0 ; volt
amplitudes             5e-3 ; volt
phases                    0 ; degree
```

In the input data above:

- `frequency_upper_limit`, `frequency_lower_limit`, and `steps_per_decade` define the frequency range and the resolution on a log scale (for the fundamental frequency). Frequencies are scanned from the upper limit to the lower one.
- Electric current and potential signals are sampled at regular time interval and `steps_per_cycle` controls the size of that interval. `ignore_cycles` allows to truncate the data in the Fourier analysis. It is best when `(cycles - ignore_cycles) * steps_per_cycle` is a power of two (most efficient in the discrete Fourier transform) but this does not have to be so.
- `harmonics` allows to select what harmonics k of the fundamental frequency f to excite. `amplitudes` and `phases` are used to specify E_k and φ_k, respectively. They may be given as arrays and must have the same size. This multi-sine feature is **experimental** though. In principle, exciting simultaneously multiple frequencies reduces the computational cost associated with a full spectrum acquisition, but in practice, it is hard to maintain the quality of the data measurement without increasing the number of steps.

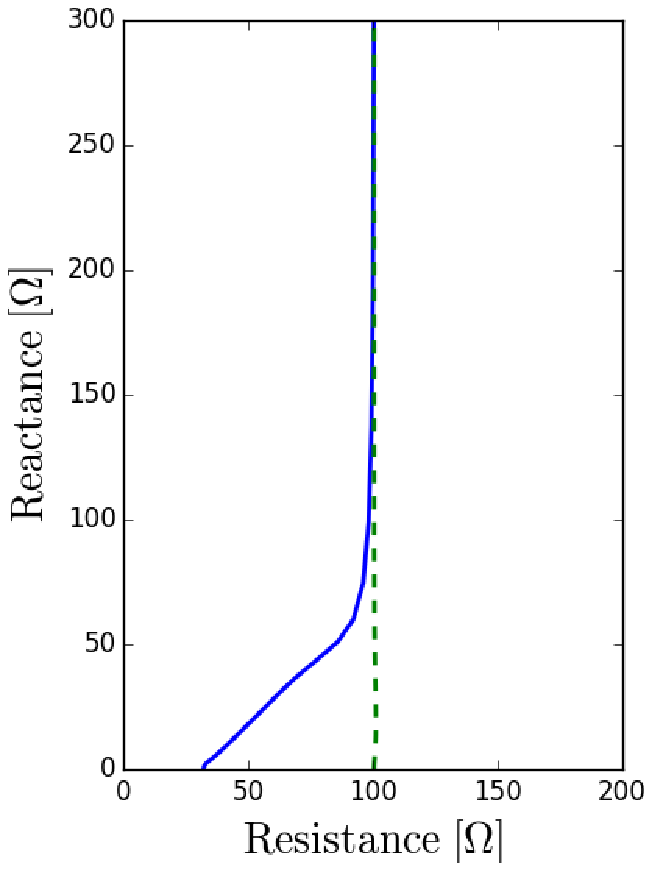
Below is an example of EIS measurement using Cap:

```python
from pycap import PropertyTree, ElectrochemicalImpedanceSpectroscopy,\
                  NyquistPlot

# setup the experiment
ptree = PropertyTree()
ptree.parse.info('eis.info')
eis = ElectrochemicalImpedanceSpectroscopy(ptree)

# build an energy storage device and run the EIS measurement
eis.run(device)

# visualize the impedance spectrum
nyquist = NyquistPlot(filename='nyquist.png')
nyquist.update(eis)
```

On the Nyquist plot above, the solid blue line shows the impedance of a supercapacitor on the complex plane with the typical 45 degrees slope for the higher frequencies. The vertical dashed green line corresponds to an equivalent RC circuit.

Ragone plot

Conceptually, the y-axis describes how much energy is available and the the x-axis shows how quickly that energy can be delivered.


300
250
200
150
100
50
0
0
50
100
150
200
Reactance [Ω]
Resistance [Ω]

CHAPTER 5

Examples

Constant current charge constant voltage discharge

This is based on [Journal of The Electrochemical Society, 152 (5) D79-D87 (2005)] by M. Verbrugge and P. Liu. It illustrates how easy it is to specify complex operating conditions for energy storage devices in `pycap`.

```python
from pycap import EnergyStorageDevice,PropertyTree
from pycap import initialize_data,report_data,plot_data
from matplotlib import pyplot
%matplotlib inline
```

The supercapacitor is initially fully discharged. It is charged to 1.7 V at a constant current of 100 A. Subsequently, a constant 1.4 V is applied for 5 s and the supercapacitor is allowed to rest at open circuit potential for 3 min. This sequence is repeated for a series of charge potentials at 0.1 V increments from 1.8 to 2.4 V. The routine defined below, `run_verbrugge_experiment`, implements that experiment and records measurements for the time, current and voltage.

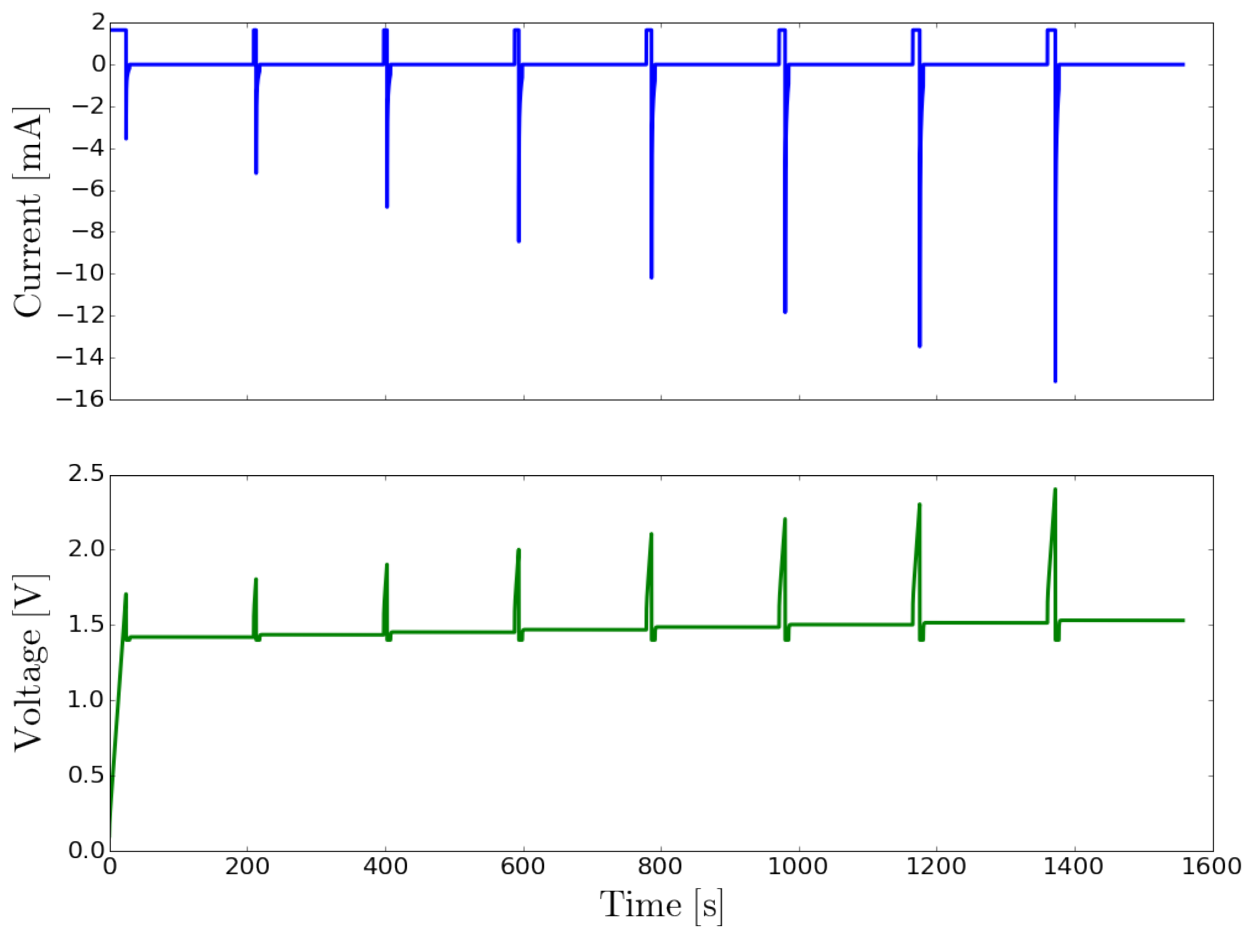
```python
def run_verbrugge_experiment(device):
    charge_current=1.65e-3 # ampere
    discharge_voltage=1.4 # volt
    discharge_time=5.0 # second
    rest_time=180.0 # second
    time_step=0.1 # second
    time=0.0
    data=initialize_data()
    for charge_voltage in [1.7,1.8,1.9,2.0,2.1,2.2,2.3,2.4]:
        # constant current charge
        while device.get_voltage()<charge_voltage:
            time+=time_step
            device.evolve_one_time_step_constant_current(time_step,charge_current)
            report_data(data,time,device)
        # constant voltage discharge
        tick=time
        while time-tick<discharge_time:
            time+=time_step
            device.evolve_one_time_step_constant_voltage(time_step,discharge_voltage)
            report_data(data,time,device)
        # rest at open circuit
        tick=time
        while time-tick<rest_time:
            time+=time_step
            device.evolve_one_time_step_constant_current(time_step,0.0)
            report_data(data,time,device)
    return data
```

Make an energy storage device (here a supercapacitor) and run the experiment.

```
input_database=PropertyTree()
input_database.parse_xml('super_capacitor.xml')
# no faradaic processes
input_database.put_double('device.material_properties.electrode_material.exchange_
↪current_density',0.0)
device=EnergyStorageDevice(input_database.get_child('device'))
# run experiment
data=run_verbrugge_experiment(device)
```

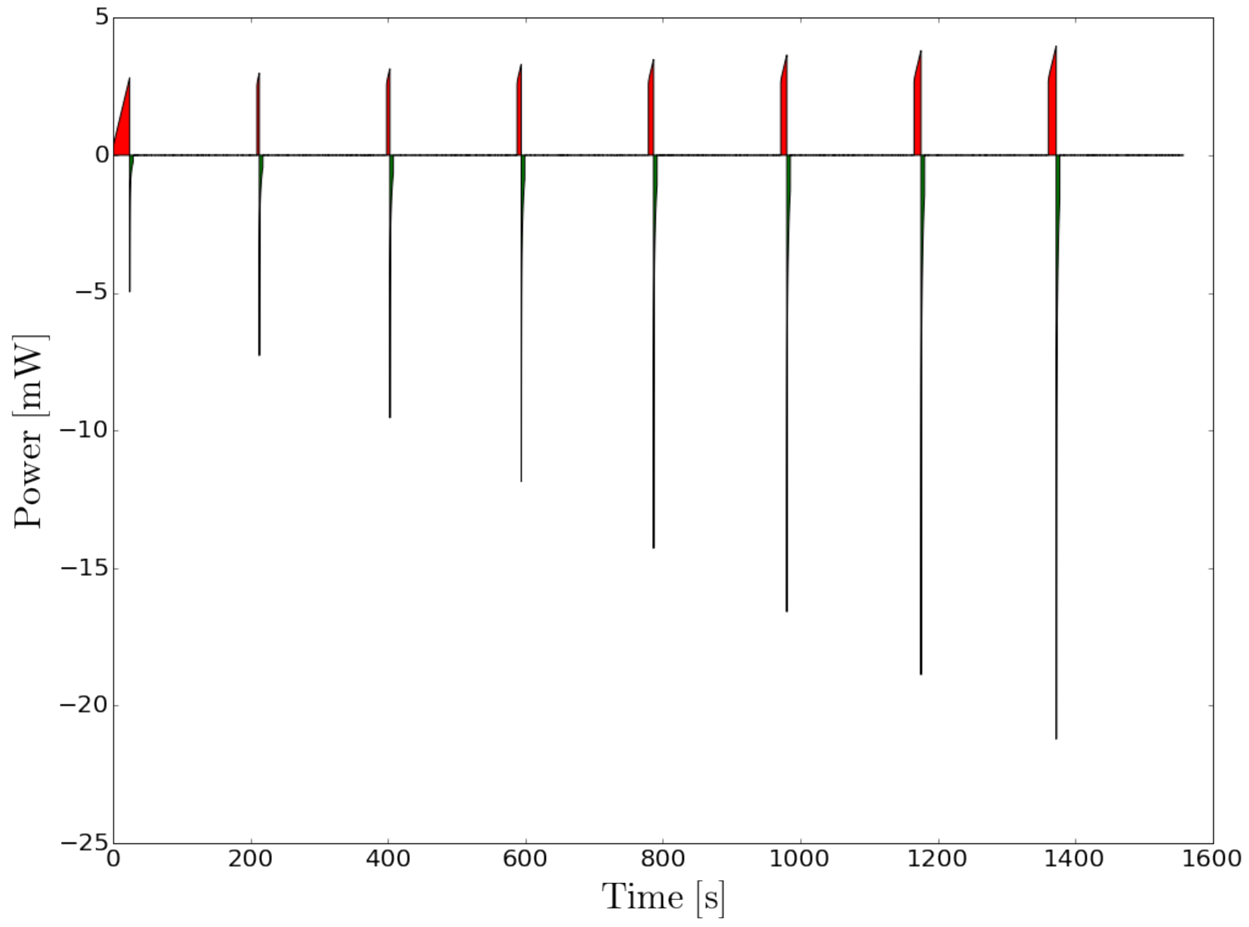
Postprocess the results.

```
time=data['time']
current=data['current']
voltage=data['voltage']
label_fontsize=30
tick_fontsize=20
labelx=-0.05
labely=0.5
plot_linewidth=3
f,axarr=pyplot.subplots(2,sharex=True,figsize=(16,12))
axarr[0].plot(time,1e+3*current,'b-',lw=plot_linewidth)
axarr[0].set_ylabel(r'$\mathrm{Current\ [mA]}$',fontsize=label_fontsize)
axarr[0].get_yaxis().set_tick_params(labelsize=tick_fontsize)
axarr[0].yaxis.set_label_coords(labelx,labely)
axarr[1].plot(time,voltage,'g-',lw=plot_linewidth)
axarr[1].set_ylabel(r'$\mathrm{Voltage\ [V]}$',fontsize=label_fontsize)
axarr[1].set_xlabel(r'$\mathrm{Time\ [s]}$',fontsize=label_fontsize)
axarr[1].get_yaxis().set_tick_params(labelsize=tick_fontsize)
axarr[1].get_xaxis().set_tick_params(labelsize=tick_fontsize)
axarr[1].yaxis.set_label_coords(labelx,labely)
pyplot.show()
```



Plot the power versus time. The red surface area represents the energy used to charge the supercapacitor and the green on the power pulses is the energy recovered.

```
power=current*voltage
pyplot.figure(figsize=(16,12))
pyplot.fill_between(time,1e+3*power,0,where=power>0,facecolor='r')
pyplot.fill_between(time,1e+3*power,0,where=power<0,facecolor='g')
pyplot.xlabel(r'$\mathrm{Time\ [s]}$',fontsize=label_fontsize)
pyplot.ylabel(r'$\mathrm{Power\ [mW]}$',fontsize=label_fontsize)
pyplot.gca().get_xaxis().set_tick_params(labelsize=tick_fontsize)
pyplot.gca().get_yaxis().set_tick_params(labelsize=tick_fontsize)
pyplot.show()
```


Power [mW]
Time [s]
5
0
−5
−10
−15
−20
−25
0
200
400
600
800
1000
1200
1400
1600

CHAPTER 6

Frequently Asked Questions

CHAPTER 7

Troubleshooting

Cap is under active development. To find out what version of Cap you are using, you may use:

```
>>> print("pycap Branch: {0} Commit: {1}"\
... .format(pycap.__git_branch__, pycap.__git_commit_hash__))
```

CHAPTER 8

Acknowledgements

Cap development has been supported by Department of Energy (DOE) DiaMonD project.

CHAPTER 9

Appendix: Weak Formulation

Strong formulation

- In the collector: $i_1 = -\sigma\nabla\Phi_1 \; \nabla\cdot i_1 = 0$
- In the electrode: $i_1 = -\sigma\nabla\Phi_1 \; i_2 = -\kappa\nabla\Phi_2 \; \nabla\cdot i_1 = \nabla\cdot i_2 = ai_n$
- In the seperator: $i_2 = -\kappa\nabla\Phi_2 \; \nabla\cdot i_2 = 0$

Weak formulation

- In the collector:

$$-\int_{\Omega_c} dr\phi_{1,i}\sigma\Delta\Phi_{1,j}\phi_{1,j} = -\int_{\partial\Omega_c} dr\phi_{1,i}\sigma\nabla\Phi_{1,j}\phi_{1,j} + \int_{\Omega_c} dr\nabla\phi_{1,i}\sigma\nabla\Phi_{1,j}\phi_{1,j}$$

- In the electrode:

$$\begin{pmatrix} \int_{\Omega_e} dr\phi_{1,i}(-\sigma)\Delta\Phi_{1,j}\phi_{1,j} + aC\frac{\partial\Phi_{1,j}\phi_{1,j}}{\partial t} & -\int_{\Omega_e} dr\phi_{1,i}aC\frac{\partial\Phi_{2,j}\phi_{2,j}}{\partial t} \\ -\int_{\Omega_e} dr\phi_{2,i}ac\frac{\partial\phi_{1,j}}{\partial t} & \int_{\Omega_2} dr\phi_{2,i}(-\kappa)\Delta\Phi_{2,j}\phi_{2,j} - \phi_{2,i}aC\frac{\partial\Phi_{2,j}\phi_{2,j}}{\partial t} \end{pmatrix} = \Bigg(-\int_{\partial\Omega_e} dr\phi_{1,i}\; sigma\nabla\Phi_{1,j}\phi_{1,j} + \quad -\int_{\Omega_e} dr$$

- In the seperator:

$$-\int_{\Omega_s} dr\phi_{2,i}\kappa\Delta\Phi_{2,j}\phi_{2,j} = -\int_{\partial\Omega_s} dr\phi_{2,i}\kappa\nabla\Phi_{2,j}\phi_{2,j} + \int_{\Omega_s} dr\nabla\phi_{2,i}\sigma\nabla\Phi_{2,j}\phi_{2,j}`$$

CHAPTER 10

Indices and tables

- genindex
- modindex
- search