Unbalanced OPF Toolkit Documentation

Release 0.1

Alvaro Estandia

Jan 03, 2020

Contents:

The Unbalanced OPF Toolkit (UOT) is a general code base to formulate and solve unbalanced optimal power flow (OPF) problems written in *MATLAB*. UOT formulates the optimization problems using YALMIP [8] allowing to use a multitude of solvers. UOT supports GridLAB-D [9] models of the power distribution network.

CHAPTER 1

Citing Unbalanced OPF Toolkit

If you use Unbalanced OPF Toolkit for a scientific publication, please cite [10].

Note: The documentation is a work in progress. If you would benefit from documentation in a specific set of files please create an issue so that they get higher priority.

1.1 Getting Started

The following sections will help you set up the environment and install UOT.

1.1.1 Prerequisites

YALMIP

The code relies on YALMIP to formulate and solve optimization problems. Before running the code make sure that YALMIP works and is able to solve linear (LPs) and semi-definite programs (SDPs) by running:

```
yalmiptest
```

and verifying that the LP and SDP tests pass.

Solvers

Small problems can be solved using free solvers like GLPK and SeDuMi. However, large-scale problems benefit from a commercial solver. In our experience, Gurobi offers the best results, followed by MOSEK for LPs. For SDPs, MOSEK is the best choice. Both Gurobi and MOSEK offer free licenses for academia. See YALMIP's solver page for more information.

At a minimum, you should install Sedumi which is free and solves LPs and SDPs.

GridLAB-D

Small networks can be defined directly in code (see GetLoadCaseIEEE_13_NoRegs_Manual.m). However, defining large models is time consuming and error prone Hence, we support GridLAB-D models of power distribution networks.

To enable support of GridLAB-D models, you need to install ASL's GridLAB-D fork. See installation instructions there.

Environment

You need to create a file `src/main/aaSetEnvironment.m` with environment specific configuration. See `src/main/aaSetEnvironmentTemplate.m` for an example file.

1.1.2 Installation

Clone the unbalanced-opf-toolkit repository.

If you want to use UOT from outside the folder where it resides, you need to add this folder to MATLAB's path with the command:

```
addpath('path to unbalanced-opf-toolkit')
```

For example:

```
addpath('~/Repos/unbalanced-opf-toolkit')
```

1.2 Tutorial

This tutorial will show you how to solve power flow and optimal power flow problems. In addtion, a more advanced part will show you how to implement a new power flow surrogate from scratch.

1.2.1 Solve power flow

Preliminaries

First, set MATLAB path to `src/demo`, then run `aaSetupPath.m`.

Solve power flow problem

We solve a power flow problem using UOT's solver.

Generated from aaUseCase_SolvePF.m

```
clear variables
aaSetupPath

% If true, use GridLAB-D to get the power network model
use_gridlab = false;
```

Initialize network and loads

Create a LoadCase object which includes the network and loads. Also define the voltage at the PCC bus.

```
% Get load case and voltage at PCC
if use_gridlab
    model_importer = GetModelImporterIEEE_13_NoRegs();
    model_importer.Initialize();

    load_case = model_importer.load_case_prerot;

    u_pcc_array = model_importer.u_pcc_array;
    t_pcc_array = model_importer.t_pcc_array;
else
    load_case = GetLoadCaseIEEE_13_NoRegs_Manual();

    u_pcc_array = [1, 1, 1];
    t_pcc_array = deg2rad([0, -120, 120]);
end
```

Solve power flow

Solve power flow with UOT's solver.

```
[U_array,T_array,p_pcc_array,q_pcc_array] = load_case.SolvePowerFlow(u_pcc_array,t_
↪pcc_array);

U_array
```

```
U_array =

    1.0000    1.0000    1.0000
    0.9227    1.0017    0.8995
    0.9227    1.0017    0.8995
    0.9227    1.0017    0.8995
    0.9155    1.0041    0.8972
    0.9209       NaN    0.8974
       NaN       NaN    0.8953
    0.9557    0.9910    0.9438
       NaN    0.9817    0.9421
       NaN    0.9800    0.9402
    0.9524    0.9891    0.9410
    0.9267    0.9697    0.9206
    0.9157       NaN       NaN
```

1.2.2 Solve optimal power flow

Preliminaries

First, set MATLAB path to `src/demo`, then run `aaSetupPath.m`.

Solve optimal power flow problem

We solve an optimal power flow problem using UOT.

Generated from aaUseCase_SolveOPF.m

```
clear variables
aaSetupPath

% If true, use GridLAB-D to get the power network model
use_gridlab = false;
```

Initialize OPF problem

We use the power flow surrogate BFM SDP from Gan2014

```
pf_surrogate_spec = uot.PowerFlowSurrogateSpec_Gan2014_SDP();
opf_problem = GetExampleOPFproblem(pf_surrogate_spec,use_gridlab);

% Select solver
opf_problem.sdpsettings = sdpsettings('solver','sedumi');
```

Solve OPF problem

Solve optimization problem

```
[objective_value,solver_time,diagnostics] = opf_problem.Solve();

% Estimate voltage from result of optimization problem
[U_array,T_array]= opf_problem.GetVoltageEstimate();
```

```
SeDuMi 1.32 by AdvOL, 2005-2008 and Jos F. Sturm, 1998-2003.
Alg = 2: xz-corrector, theta = 0.250, beta = 0.500
Put 234 free variables in a quadratic cone
eqs m = 340, order n = 207, dim = 1557, blocks = 15
nnz(A) = 3135 + 0, nnz(ADA) = 115600, nnz(L) = 57970
 it :     b*y       gap    delta  rate   t/tP*  t/tD*   feas cg cg  prec
  0 :            9.44E-01 0.000
  1 :   1.18E+01 5.67E-01 0.000 0.6007 0.9000 0.9000   1.68  1  1  6.8E+00
  2 :  -1.63E+00 2.65E-01 0.000 0.4675 0.9000 0.9000   2.49  1  1  3.2E+00
  3 :   4.61E-01 1.26E-01 0.000 0.4744 0.9000 0.9000   1.32  1  1  1.2E+00
  4 :   8.07E-01 4.93E-02 0.000 0.3919 0.9000 0.9000   1.97  1  1  3.1E-01
  5 :   9.13E-01 2.17E-02 0.000 0.4404 0.9000 0.9000   1.94  1  1  9.6E-02
  6 :   9.86E-01 7.47E-03 0.000 0.3439 0.9000 0.9000   1.42  1  1  2.7E-02
  7 :   1.02E+00 2.37E-03 0.000 0.3180 0.9000 0.9000   1.20  1  1  7.9E-03
  8 :   1.04E+00 4.71E-04 0.000 0.1985 0.9000 0.9000   1.07  1  1  1.5E-03
  9 :   1.04E+00 9.96E-05 0.000 0.2112 0.9000 0.9000   1.02  1  1  3.2E-04
 10 :   1.04E+00 2.18E-05 0.000 0.2191 0.9000 0.9000   1.01  1  1  6.9E-05
 11 :   1.04E+00 4.69E-06 0.000 0.2149 0.9000 0.9000   1.02  1  1  1.5E-05
 12 :   1.04E+00 9.78E-07 0.000 0.2086 0.9000 0.9000   1.02  1  1  3.0E-06
 13 :   1.04E+00 1.98E-07 0.000 0.2028 0.9000 0.9000   1.01  1  1  6.1E-07
 14 :   1.04E+00 3.93E-08 0.000 0.1984 0.9000 0.9000   1.00  2  2  1.2E-07
 15 :   1.04E+00 7.64E-09 0.000 0.1943 0.9000 0.9000   1.00  2  2  2.3E-08
 16 :   1.04E+00 1.49E-09 0.000 0.1949 0.9000 0.9000   1.00  2  2  4.6E-09
```

(continues on next page)

(continued from previous page)

```
 17 :   1.04E+00 3.12E-10 0.000 0.2096 0.9000 0.9000   1.00  2  2  9.6E-10

iter seconds digits       c*x               b*y
 17      0.6   8.8  1.0417424945e+00  1.0417424930e+00
|Ax-b| =   1.9e-09, [Ay-c]_+ =   3.0E-10, |x|=  1.1e+01, |y|=  1.3e+01

Detailed timing (sec)
   Pre          IPM          Post
8.772E-02    5.089E-01    2.397E-02
Max-norms: ||b||=1, ||c|| = 5,
Cholesky |add|=0, |skip| = 0, ||L.L|| = 41661.5.
```

Verify if the solution is exact

Compare the indirect variables (voltages at non-PCC buses and power injection at PCC) from the optimization problem and the power flow solution. If they match, then the solution is exact.

```
% Solve power flow using the controllable loads computed in the optimization
[U_array_ref,T_array_ref,p_pcc_array_ref,q_pcc_array_ref] = opf_problem.
↪SolvePFwithControllableLoadValues();

% Convert polar representation to complex
V_array = uot.PolarToComplex(U_array,T_array);
V_array_ref = uot.PolarToComplex(U_array_ref,T_array_ref);

% Compare voltages at non-PCC buses
V_array_error = abs(V_array - V_array_ref);
V_array_error_max = max(V_array_error(:))

% Compare power injection at PCC
[p_pcc_array,q_pcc_array] = opf_problem.EvaluatePowerInjectionFromPCCload();
s_pcc_array = p_pcc_array + 1i*q_pcc_array;
s_pcc_array_ref = p_pcc_array_ref + 1i*q_pcc_array_ref;

s_pcc_array_ref_error = abs(s_pcc_array - s_pcc_array_ref);

s_pcc_array_ref_error_max = max(s_pcc_array_ref_error(:))
```

```
V_array_error_max =

   6.1412e-10


s_pcc_array_ref_error_max =

   1.9129e-09
```

1.2.3 Implement a new power flow surrogate

This tutorial shows how to implement a new power flow surrogate based on [4].

The purpose of a power flow surrogate is to relate direct variables (voltage at PCC and power injections at PQ buses) to indirect variables (power injection at PCC and voltage at PQ buses). Concretely, the power flow surrogate needs to implement the following constraints:

- Voltage magnitude limits
- Power injection at PCC

In addition, the surrogate needs to specify the voltage at PCC. At first, this might seem surprising since the voltage at PCC is a direct variable. However, recall that different surrogates represent voltage in different ways. For example:

- *uot.PowerFlowSurrogate_Gan2014_SDP* uses one hermitian matrix per bus
- *uot.PowerFlowSurrogate_Bolognani2015_LP* uses one real vector for voltage magnitude and another for voltage phase

This means, the implementation of the voltage at PCC constraint is different in each case. Consequently, UOT leaves the implementation of the constraint to each surrogate.

The opposite happens for the power injection at the PCC: it is an indirect variable but the constraints on it are implemented by `uot.ControllableLoad` and not by the surrogate. The reason is that the PCC load is always represented in the same way: one real vector for active power injection and another for reactive power injection. Since the constraints always take the same form, it makes sense to implement them only once.

One important remark is in order: the PCC load is implemented in *uot.OPFproblem* but the power flow surrogate *must* couple it to the rest of the problem through a constraint. This is typically done through:

```
[P_inj_array,Q_inj_array] = obj.opf_problem.ComputeNodalPowerInjection();
```

For an example, see `uot.PowerflowSurrogate_Bolognani2015_LP.GetConstraintArray`.

Power flow surrogate specification

The first step is creating `PowerFlowSurrogateSpec_Bernstein2017_LP_1` which is a derived class of *uot.AbstractPowerFlowSurrogateSpec*. A good starting point is making a copy of *uot.SpecTemplate* and changing the parent class to *uot.AbstractPowerFlowSurrogateSpec*. By convention, we put the class in a directory named @PowerFlowSurrogateSpec_Bernstein2017_LP_1. This is not strictly necessary but makes it easier to see if a file is a class, function or script. For now, we can leave the sample documentation as it is.

Looking at *uot.AbstractPowerFlowSurrogateSpec*, we learn that the purpose of `PowerFlowSurrogateSpec_Bernstein2017_LP_1` is simply telling *uot.OPFproblem* how to instantiate the power flow surrogate. For this purpose, we need to create a concrete implementation of `uot.AbstractPowerFlowSurrogateSpec.Create`. For inspiration, we can look how this is done in *uot.PowerFlowSurrogateSpec_Bolognani2015_LP*.

At the end, we have:

```
classdef PowerFlowSurrogateSpec_Bernstein2017_LP_1 < uot.AbstractPowerFlowSurrogateSpec
    % Class to specify xx
    %
    % Synopsis::
    %
    %   obj = uot.SpecTemplate(bus,varargin)
    %
    % Description:
    %   This class is intended as a starting point to develop classes that
    %   derive from ``uot.Spec``.
    %
    %
    %   We can also itemize
    %
    %     - item 1
```

(continues on next page)

(continued from previous page)

```
    %     - item 2
    %
    %
    % Arguments:
    %   bus (char): Bus name
    %
    % Keyword Arguments:
    %   'param1': Something.
    %
    % Returns:
    %   obj: Instance of `uot.SpecTemplate`.
    %
    % Note:
    %  Possibly add a note here
    %
    % Example::
    %
    %   object = uot.ObjectTemplate(uot.SpecTemplate())
    %
    % See Also:
    %   `uot.ObjectTemplate`
    %
    % Todo:
    %
    %   - Write documentation
    %
    % .. Line with 80 characters for reference ####################################

    methods
        function obj = PowerFlowSurrogateSpec_Bernstein2017_LP_1()
        end
    end

    % Implemented abstract methods from uot.AbstractPowerFlowSurrogateSpec
    methods
        function pf_surrogate = Create(obj,varargin)
            pf_surrogate = PowerFlowSurrogate_Bernstein2017_LP_1(obj,varargin{:});
        end
    end
end
```

Note: We add the suffix *_1* to distinguish this first implementation from more evolved ones we will develop in the course of the tutorial.

Note: The classes created in this tutorial are not prefixed by *uot*, since they are not in the +uot directory which builds the module. The production implementation (which is preceded by uot) are optimized for performance and include further functionality which is outside the scope of this tutorial

Barebones power flow surrogate

The next step is creating a barebones power flow surrogate class which is derived of *uot.AbstractPowerFlowSurrogate*.

Looking at *uot.AbstractPowerFlowSurrogate*, we see that we need to implement some abstract methods:

- `AssignBaseCaseSolution()`
- `ComputeVoltageEstimate()`
- `GetConstraintArray()`
- `SolveApproxPowerFlow()` (static)

We now create the class `PowerFlowSurrogateSpec_Bernstein2017_LP_1` with these methods as stubs. A good starting point is making a copy of *uot.ObjectTemplate* and changing the parent class to *uot.AbstractPowerFlowSurrogate*. For now, we can leave the sample documentation as it is.

We implement the constructor following the example of *uot.PowerFlowSurrogate_Bolognani2015_LP*. At the end, we have the barebones implementation:

```
classdef PowerFlowSurrogate_Bernstein2017_LP_1 < uot.AbstractPowerFlowSurrogate
    % Template class for uot.Object.
    %
    % Synopsis::
    %
    %   obj = uot.ObjectTemplate(spec,'param1',val1)
    %
    % Description:
    %   This class is intended as a starting point to develop classes that
    %   derive from ``uot.Object``.
    %
    %
    %   We can also itemize
    %
    %     - item 1
    %     - item 2
    %
    % Arguments:
    %   spec (|uot.SpecTemplate|): Object specification
    %
    % Keyword Arguments:
    %   'param1' (:class:`uot.PCCloadSpec<+uot.@PCCloadSpec.PCCloadSpec>`): Parameter
    %
    %
    % Note:
    %  Possibly add a note here
    %
    % Example::
    %
    %   object = uot.ObjectTemplate(uot.SpecTemplate())
    %
    % See Also:
    %   |uot.SpecTemplate|
    %
    % Todo:
    %
    %   - Write documentation
    %

    % .. Line with 80 characters for reference ####################################

    methods
        function obj = PowerFlowSurrogate_Bernstein2017_LP_1(spec,opf_problem)
```

(continues on next page)

(continued from previous page)

```
            validateattributes(spec,{'PowerFlowSurrogateSpec_Bernstein2017_LP_1'},{
↪'scalar'},mfilename,'spec',1);

            decision_variables = [];
            obj@uot.AbstractPowerFlowSurrogate(spec,opf_problem,decision_variables)
        end

        constraint_array = GetConstraintArray(obj) % Abstract method from uot.
↪AbstractPowerFlowSurrogate
    end

    methods (Static)
        [U_array,T_array,p_pcc_array,q_pcc_array,opf_problem] =␣
↪SolveApproxPowerFlow(load_case,u_pcc_array,t_pcc_array)  % Abstract method from uot.
↪AbstractPowerFlowSurrogate class
    end

    methods (Access = {?uot.AbstractPowerFlowSurrogate,?uot.OPFproblem})
        [U_array,T_array,p_pcc_array,q_pcc_array] = AssignBaseCaseSolution(obj) %␣
↪Abstract method from uot.AbstractPowerFlowSurrogate
        [U_array,T_array] = ComputeVoltageEstimate(obj) % Abstract method from uot.
↪AbstractPowerFlowSurrogate
    end
end
```

Initialize OPF problem with PowerFlowSurrogateSpec_Bernstein2017_LP_1

Here, we instantiate OPF problem with PowerFlowSurrogateSpec_Bernstein2017_LP_1 to verify that out barebones implementation is working as expected.

Generated from aaInstantiateBarebones.m

```
clear variables

aaSetupPath

[opf_spec, load_case] = CreateSampleOPFspec();

% Replace pf_surrogate_spec with PowerFlowSurrogateSpec_Bernstein2017_LP_1
opf_spec.pf_surrogate_spec = PowerFlowSurrogateSpec_Bernstein2017_LP_1();

opf_problem = uot.OPFproblem(opf_spec,load_case);
```

Note: By convention, we use the prefix *aa* in scripts to distinguish them from functions

Solving approximate power flow

Implementing power flow surrogates is tricky and error-prone. Hence, testing is an integral part of the process. One sensible way of doing this is trying to replicate the results in the paper.

The paper presents a *continuation analysis* where they approximately solve power flow for a series of scaled loads using the proposed linear formulation. Then, they compare the results with the actual power flow solution and compute the error.

We will try to replicate the results in Figures 3 and 5 of the paper. Figure 3 shows an error of less than 0.2% in voltage whereas Figure 5 shows an error of less than 1.5% for the power at the PCC.

In order to replicate the paper's results, we need to solve *power flow* and not *optimal power flow*. It is very common that power flow surrogates offer a way of approximately solving power flow algebraically. In fact, Bernstein's paper is not about OPF at all. The method `uot.AbstractPowerFlowSurrogate.SolveApproxPowerFlowAlt` exists precisely for this use case. It can be overridden by a power flow surrogate to offer a way of solving power flow.

Note: The Alt in *SolveApproxPowerFlowAlt* is there because `uot.AbstractPowerFlowSurrogate.SolveApproxPowerFlow` also exists. However, *SolveApproxPowerFlow* uses a different method that requires solving an optimization problem. The advantage of the optimization-based method is that it works even for surrogates where power flow cannot be solved in an algebraic fashion (e.g., *uot.PowerFlowSurrogate_Gan2014_SDP*)

For this purpose, we implement `PowerFlowSurrogate_Bernstein2017_LP_2.SolveApproxPowerFlowAlt()`:

```
function [U_array,T_array, p_pcc_array, q_pcc_array,extra_data] = SolveApproxPowerFlowAlt(load_case,u_pcc_array,t_pcc_array,varargin)
% This function is static
% [...] = SolveApproxPowerFlowAlt(load_case,u_pcc_array,t_pcc_array) linearization at the flat voltage solution
% [...] = SolveApproxPowerFlowAlt(load_case,u_pcc_array,t_pcc_array,U_ast,T_ast) linearization at U_ast,T_ast

% Check inputs following how it's done in uot.PowerFlowSurrogate_Bolognani2015_LP.SolveApproxPowerFlowAlt
validateattributes(load_case,{'uot.LoadCasePy'},{'scalar'},mfilename,'load_case',1);

network = load_case.network;

n_time_step = load_case.spec.n_time_step;
n_phase = network.n_phase;

validateattributes(u_pcc_array,{'double'},{'size',[n_time_step,n_phase]},mfilename,'u_pcc_array',2);
validateattributes(t_pcc_array,{'double'},{'size',[n_time_step,n_phase]},mfilename,'t_pcc_array',3);

n_varargin = numel(varargin);

switch n_varargin
    case 0
        % Linearize at flat voltage
        linearization_point = uot.enum.CommonLinearizationPoints.FlatVoltage;
        [U_ast,T_ast] = linearization_point.GetVoltageAtLinearizationPoint(load_case,u_pcc_array,t_pcc_array);

    case 2
        U_ast = varargin{1};
        T_ast = varargin{2};

        validate_phase_consistency_h = @(x) uot.AssertPhaseConsistency(x,network.bus_has_phase);

        uot.ValidateAttributes(U_ast,{'double'},{'size',[nan,nan,1],validate_phase_consistency_h},mfilename,'U_ast',4);
```

(continues on next page)

(continued from previous page)

```
        uot.ValidateAttributes(T_ast,{'double'},{'size',[nan,nan,1],validate_phase_consistency_h},mfilename,'T_ast',5);
    otherwise
        error('Invalid number of arguments.')
end

% Compute x_y
% Loads are negative power injections
S_inj_array = -load_case.S_y;
P_inj_array = real(S_inj_array);
Q_inj_array = imag(S_inj_array);

% x_y is the phase-consistent stack (see glossary) excluding the pcc (bus 1)
x_y = [uot.StackPhaseConsistent(P_inj_array(2:end,:,:),network.bus_has_phase(2:end,:,:));uot.StackPhaseConsistent(Q_inj_array(2:end,:,:),network.bus_has_phase(2:end,:,:))];

V_ast = uot.PolarToComplex(U_ast,T_ast);

% Remove pcc
V_ast_nopcc_stack = uot.StackPhaseConsistent(V_ast(2:end,:,:),network.bus_has_phase(2:end,:,:));

[P_ast,Q_ast] = network.ComputePowerInjectionFromVoltage(U_ast,T_ast);
x_y_ast = [uot.StackPhaseConsistent(P_ast(2:end,:,:),network.bus_has_phase(2:end,:,:));uot.StackPhaseConsistent(Q_ast(2:end,:,:),network.bus_has_phase(2:end,:,:))];

[~,~,w] = network.ComputeNoLoadVoltage(u_pcc_array,t_pcc_array);

% Compute voltage according to eq. 5a
Ybus_NN_inv = inv(network.Ybus_NN);
M_y_pre = Ybus_NN_inv*inv(diag(conj(V_ast_nopcc_stack)));

M_y = [M_y_pre,-1i*M_y_pre];

a = w;

V_nopcc_array_stack = M_y*x_y + a;

% Add pcc voltage
v_pcc_array = uot.PolarToComplex(u_pcc_array,t_pcc_array);

V_array_stack = [v_pcc_array.';V_nopcc_array_stack];

V_array = uot.UnstackPhaseConsistent(V_array_stack,network.bus_has_phase);

[U_array,T_array] = uot.ComplexToPolar(V_array);

% Compute pcc power injection according to eq. 5c and 13
s_pcc_array = zeros(n_time_step,network.n_phase);

for i = 1:n_time_step
    v_pcc_i = v_pcc_array(i,:).';
    G_y = diag(v_pcc_i)*conj(network.Ybus_SN)*conj(M_y);

    c = diag(v_pcc_i)*(conj(network.Ybus_SS)*conj(v_pcc_i) + conj(network.Ybus_SN)*conj(a(:,i)));
```

(continues on next page)

(continued from previous page)

```
    s_pcc_array(i,:) = G_y*x_y(:,i) + c;
end

p_pcc_array = real(s_pcc_array);
q_pcc_array = imag(s_pcc_array);

% In addition to the results from standard power flow, the paper
% presents 2 approaches to approximate the magnitude of the voltages in the
% PQ nodes: using eq. 9 and eq. 12.
% We include them here so that we can test them.

% Approach from eq. 9
V_ast_nopcc_stack_abs = abs(V_ast_nopcc_stack);
K_y_eq9 =  inv(diag(V_ast_nopcc_stack_abs))*real(diag(conj(V_ast_nopcc_stack))*M_y);
b_eq9 = V_ast_nopcc_stack_abs - K_y_eq9*x_y_ast;

U_array_stack_pre_eq9 = K_y_eq9*x_y + b_eq9;

U_array_stack_eq9 = [u_pcc_array.';U_array_stack_pre_eq9];

U_array_eq9 = uot.UnstackPhaseConsistent(U_array_stack_eq9,network.bus_has_phase);

% Approach from eq. 12
U_array_eq12 = zeros(network.n_bus,network.n_phase,n_time_step);

for i = 1:n_time_step
    w_i = w(:,i);
    W = diag(w_i);

    K_y_eq12 = abs(W)*real(inv(W)*M_y);

    b_eq12 = abs(w_i);

    U_stack_pre_eq12 = K_y_eq12*x_y(:,i) + b_eq12;

    U_stack_eq12 = [u_pcc_array(i,:).';U_stack_pre_eq12];

    U_array_eq12(:,:,i) = uot.UnstackPhaseConsistent(U_stack_eq12,network.bus_has_
↪phase);
end

extra_data.U_array_eq9 = U_array_eq9;
extra_data.U_array_eq12 = U_array_eq12;
end
```

Replicate the paper's results to verify correctness

This file replicates the results in Figures 3 and 5 of [4].

Generated from aaReplicatePaperResults.m

```
clear variables
aaSetupPath
```

Initialize OPF problem with PowerFlowSurrogate_Bernstein2017_LP

```
[opf_spec, load_case] = CreateSampleOPFspec();

% Replace pf_surrogate_spec with PowerFlowSurrogateSpec_Bernstein2017_LP_2
opf_spec.pf_surrogate_spec = PowerFlowSurrogateSpec_Bernstein2017_LP_2();

opf_problem = uot.OPFproblem(opf_spec,load_case);
```

Solve approximate power flow to see if everything is working

```
linearization_point = uot.enum.CommonLinearizationPoints.FlatVoltage;

u_pcc_array = opf_problem.u_pcc_array;
t_pcc_array = opf_problem.t_pcc_array;

[U_ast,T_ast] = linearization_point.GetVoltageAtLinearizationPoint(load_case,u_pcc_
↪array,t_pcc_array);
[U_array,T_array, p_pcc_array, q_pcc_array,extra_data] = PowerFlowSurrogate_
↪Bernstein2017_LP_2.SolveApproxPowerFlowAlt(load_case,u_pcc_array,t_pcc_array,U_ast,
↪T_ast);
```

Set-up for continuation analysis

Now, we will try to replicate the paper's results. In the continuation analysis, the authors used a range of factors kappa in [-1,2]

```
kappa_vec = -1:0.1:2;

% We reorder kappa_vec so that 1 is the first element.
% The reason for this is explained below.
kappa_vec = [kappa_vec(kappa_vec == 1), kappa_vec(kappa_vec ~= 1)];

% Then we use kappa_vec to create a load_case which is scaled
% according to kappa_vec. This works because load_case contains loads for
% a single time-step (as in the IEEE specification).
load_case_continuation = load_case.*kappa_vec;

% The result is a load case with loads for n_time_step
% time steps. Here, they do not represent actual time steps, but rather
% the different loads for our continuation analysis. We take advantage
% of the capability of representing time-varying loads to make the code
% more concise.
n_time_step = load_case_continuation.n_time_step
```

```
n_time_step =

    31
```

Choose a linearization point

Now we need to choose a linearization point for solving the approximate power flow. One common choice is the power flow solution at the first time of the base case. Here, the base case should be understood in the context of OPF problems: the case in the absence of controllable loads. However, since we are not working with controllable loads yet, this is simply the first time step of the load case. Here, it should become clear why we reordered kappa_vec above so that 1 is the first element: so that we use it as our linearization point.

```
linearization_point_continuation = uot.enum.CommonLinearizationPoints.
↪PFbaseCaseFirstTimeStep;
```

Solve approximate power flow using the linear formulation

Extend u_pcc_array and t_pcc_array to have one entry per time step

```
u_pcc_array_continuation = repmat(u_pcc_array,n_time_step,1);
t_pcc_array_continuation = repmat(t_pcc_array,n_time_step,1);

[U_ast_continuation,T_ast_continuation] = linearization_point_continuation.
↪GetVoltageAtLinearizationPoint(load_case_continuation,u_pcc_array_continuation,t_
↪pcc_array_continuation);
[U_array_continuation,T_array_continuation, p_pcc_array_continuation, q_pcc_array_
↪continuation,extra_data_continuation] = PowerFlowSurrogate_Bernstein2017_LP_2.
↪SolveApproxPowerFlowAlt(load_case_continuation,u_pcc_array_continuation,t_pcc_array_
↪continuation,U_ast_continuation,T_ast_continuation);

V_array_continuation = uot.PolarToComplex(U_array_continuation,T_array_continuation);
s_pcc_continuation = p_pcc_array_continuation + 1i*q_pcc_array_continuation;
```

Get reference values from solving exact power flow

```
[U_array_ref,T_array_ref, p_pcc_array_ref, q_pcc_array_ref] = load_case_continuation.
↪SolvePowerFlow(u_pcc_array_continuation,t_pcc_array_continuation);

V_array_ref = uot.PolarToComplex(U_array_ref,T_array_ref);
s_pcc_ref = p_pcc_array_ref + 1i*q_pcc_array_ref;
```

Compute error

```
v_error = zeros(n_time_step,1);
u_error_eq9 = zeros(n_time_step,1);
u_error_eq12 = zeros(n_time_step,1);
s_pcc_error_pre = zeros(n_time_step,1);

network = load_case_continuation.network;

for i = 1:n_time_step
    % Stack to get a vector without nans
    V_continuation_stack = uot.StackPhaseConsistent(V_array_continuation(:,:,i),
↪network.bus_has_phase);
    V_ref_stack = uot.StackPhaseConsistent(V_array_ref(:,:,i),network.bus_has_phase);
```

(continues on next page)

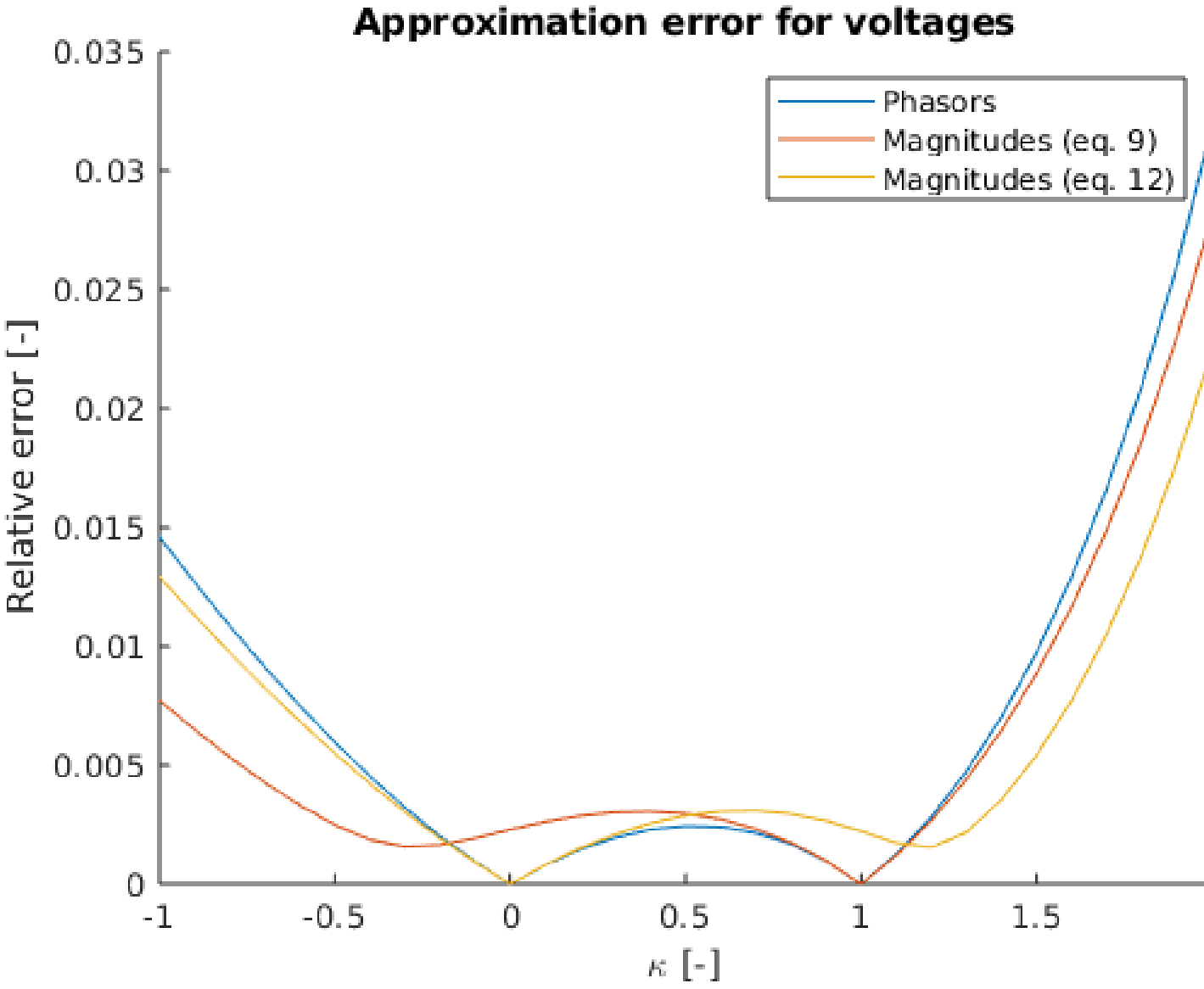
(continued from previous page)

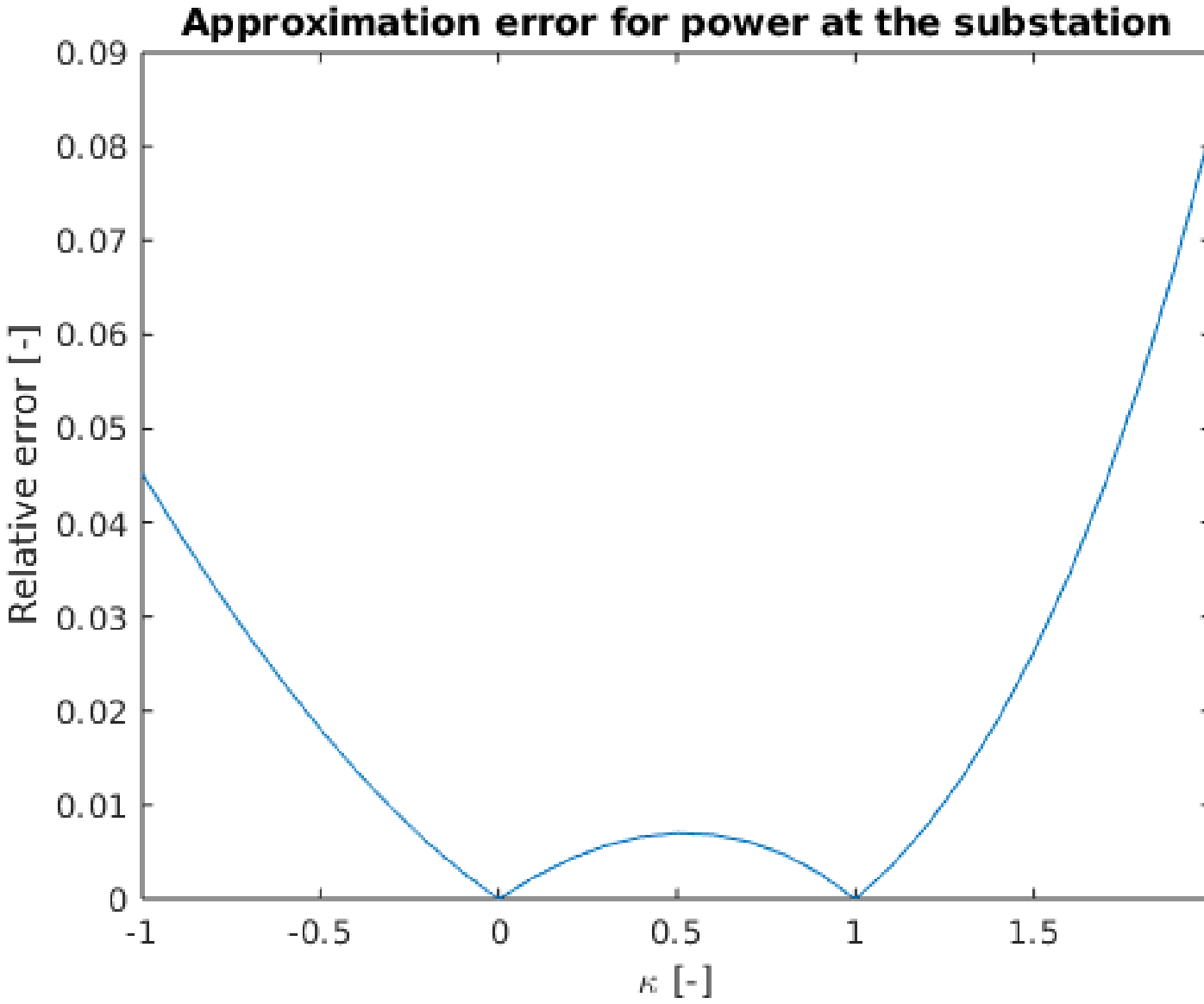
```
    U_continuation_eq9_stack = uot.StackPhaseConsistent(extra_data_continuation.U_
↪array_eq9(:,:,i),network.bus_has_phase);
    U_continuation_eq12_stack = uot.StackPhaseConsistent(extra_data_continuation.U_
↪array_eq12(:,:,i),network.bus_has_phase);

    U_ref_stack = uot.StackPhaseConsistent(U_array_ref(:,:,i),network.bus_has_phase);

    v_error(i) = norm(V_continuation_stack - V_ref_stack,2)/norm(V_ref_stack,2);
    u_error_eq9(i) = norm(U_continuation_eq9_stack - U_ref_stack,2)/norm(U_ref_stack,
↪2);
    u_error_eq12(i) = norm(U_continuation_eq12_stack - U_ref_stack,2)/norm(U_ref_
↪stack,2);

    s_pcc_error_pre(i) = norm(s_pcc_continuation(i,:) - s_pcc_ref(i,:));
end

% Compute apparent power by summing across phases and taking absolute value.
s_pcc_mag_ref = abs(sum(s_pcc_ref,2));

s_pcc_error = s_pcc_error_pre/max(s_pcc_mag_ref);
```

Create plots

Sort kappa_vec to get a nice plot

```
[kappa_vec_sorted,kappa_vec_sorted_ind] = sort(kappa_vec);

v_error_sorted = v_error(kappa_vec_sorted_ind);
u_error_eq9_sorted = u_error_eq9(kappa_vec_sorted_ind);
u_error_eq12_sorted = u_error_eq12(kappa_vec_sorted_ind);

s_pcc_error_sorted = s_pcc_error(kappa_vec_sorted_ind);

% Compute error in voltage approximation
figure
hold on
plot(kappa_vec_sorted,v_error_sorted);
plot(kappa_vec_sorted,u_error_eq9_sorted);
plot(kappa_vec_sorted,u_error_eq12_sorted);
legend('Phasors','Magnitudes (eq. 9)','Magnitudes (eq. 12)')
title('Approximation error for voltages')
xlabel('\kappa [-]')
ylabel('Relative error [-]')

% Compute error in PCC power
figure
plot(kappa_vec_sorted,s_pcc_error_sorted);
title('Approximation error for power at the substation')
xlabel('\kappa [-]')
ylabel('Relative error [-]')
```


Approximation error for voltages
Phasors
Magnitudes (eq. 9)
Magnitudes (eq. 12)
Relative error [-]
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
-1
-0.5
0
0.5
1
1.5
κ [-]



Creating an initial test case

We managed to replicate the results of the paper giving us trust in the correctness of our implementation. The next step is to make a functional test case out of this experiment. The starting point for this is `SampleTest()`.

As before, we can take inspiration from other power flow surrogates, for example, `PowerFlowSurrogate_Bolognani2015_LPtest()` for inspiration. This file has two test cases (i.e., functions starting with Test): `TestAgainstPaper()` and `TestPowerFlowSurrogate_Bolognani2015_LP()`. `TestAgainstPaper()` compares the results of the UOT implementation with code published by the paper's authors. This test is rooted on `uot.PowerFlowSurrogate_Bolognani2015_LP.SolveApproxPowerFlowAlt`.

`PowerFlowSurrogate_Bolognani2015_LP()` then verifies that the result of solving an OPF problem are consistent with `uot.PowerFlowSurrogate_Bolognani2015_LP.SolveApproxPowerFlowAlt`. Furthermore, it verifies that constraints of the OPF problem are satisfied.

Now, we create a test case analogous to *TestAgainstPaper* based on our previous results:

```
function tests = PowerFlowSurrogate_Bernstein2017_LP_2test
% Verifies that PowerFlowSurrogate_Bernstein2017_LP_2:
% - PowerFlowSurrogate_Bernstein2017_LP.SolveApproximatePowerFlowAlt can
%   replicate the results in the original paper.

% This enables us to run the test directly instead of only through runtests
call_stack = dbstack;

% Call stack has only one element if function was called directly
if ~any(contains({call_stack.name},'runtests'))
    this_file_name = mfilename();
    runtests(this_file_name)
end

tests = functiontests(localfunctions);
end

function setupOnce(test_case)
aaSetupPath

test_case.TestData.abs_tol_equality = 5e-6;
end

function TestAgainstPaper(test_case)
load_case_zip = GetLoadCaseIEEE_13_NoRegs_Manual();
% Reference voltage from spec (we do not model the regulator)
u_pcc = [1.0625, 1.0500, 1.0687];
t_pcc = deg2rad([0, -120, 120]);

% Convert all loads to constant power wye-connected
load_case_pre = load_case_zip.ConvertToLoadCasePy();

% The paper uses kappa \in [-1,2]. However, we focus here on a narrower
% range where the approximate power flow is closest to the exact one.
kappa_vec = -0.2:0.1:1.2;

% Reorder kappa_vec so that kappa_vec(1) is 1.
% This, will allow us to linearize at the unscaled load
kappa_vec = [kappa_vec(kappa_vec == 1), kappa_vec(kappa_vec ~= 1)];

load_case = load_case_pre.*kappa_vec;

% Extend u_pcc_array and t_pcc_array to have one entry per time step
n_time_step = load_case.n_time_step;
u_pcc_array = repmat(u_pcc,n_time_step,1);
t_pcc_array = repmat(t_pcc,n_time_step,1);

linearization_point = uot.enum.CommonLinearizationPoints.PFbaseCaseFirstTimeStep;
[U_ast,T_ast] = linearization_point.GetVoltageAtLinearizationPoint(load_case,u_pcc_
↪array,t_pcc_array);

[U_array,T_array, p_pcc_array, q_pcc_array,extra_data] = PowerFlowSurrogate_
↪Bernstein2017_LP_3.SolveApproxPowerFlowAlt(load_case,u_pcc_array,t_pcc_array,U_ast,
↪T_ast);

V_array = uot.PolarToComplex(U_array,T_array);
```

(continues on next page)

(continued from previous page)

```
s_pcc = p_pcc_array + 1i*q_pcc_array;

% Get reference values from solving exact power flow
[U_array_ref,T_array_ref, p_pcc_array_ref, q_pcc_array_ref] = load_case.
↪SolvePowerFlow(u_pcc_array,t_pcc_array);

V_array_ref = uot.PolarToComplex(U_array_ref,T_array_ref);
s_pcc_ref = p_pcc_array_ref + 1i*q_pcc_array_ref;

% We select the initial tolerance based on aaReplicatePaperResults and
% we tune them so that the tests just pass.
v_tol = 6e-3;

% For simplicity, we compare elementwise instead of using the paper's relative
% error metric. This will also issue a more informative error if the test fails.
% We use absolute tolerances because voltage magnitude is close to 1
verifyEqual(test_case,V_array,V_array_ref,'AbsTol',v_tol)
verifyEqual(test_case,extra_data.U_array_eq9,U_array_ref,'AbsTol',v_tol)
verifyEqual(test_case,extra_data.U_array_eq12,U_array_ref,'AbsTol',v_tol)

% We compare s_pcc using relative error because it is not necessarily close to 1
% Note that the tolerance is much larger than the 0.02 error we saw in the plot
% of aaReplicatePaperResults. This is because here we normalize with the actual
% load and not the maximal one as done in the paper. Thus the errors for the
% small loads (small kappa) are much larger than before.
s_pcc_tol = 0.15;
verifyEqual(test_case,s_pcc,s_pcc_ref,'RelTol',s_pcc_tol)
end
```

Implementing the power flow surrogate

Finally, we get to the implementation of the actual surrogate. For this purpose, we need to implement the following methods:

- `GetConstraintArrayHelper()`
- `DefineDecisionVariables()`

Let's get started with `GetConstraintArrayHelper()`, which defines the constraints we need to implement. From above, we know these are:

- Voltage at PCC
- Voltage magnitude limits
- Power injection at PCC

Again, we can use `uot.PowerFlowSurrogate_Bolognani2015_LP.GetConstraintArrayHelper` for guidance. Furthermore, we can use much of the code that we wrote for `PowerFlowSurrogate_Bernstein2017_LP_2.SolveApproxPowerFlowAlt()`. In order to reduce code duplication we will be moving some of the code there into new methods. We just need to keep in mind that these methods must be static so that `PowerFlowSurrogate_Bernstein2017_LP_3.SolveApproxPowerFlowAlt()` (which is static) can call them.

```
function constraint_array = GetConstraintArray(obj)
% |protected| Creates the array of constraints that result from applying the power␣
↪flow surrogate to the |opf| problem
```

(continues on next page)

(continued from previous page)

```
%
% Synopsis::
%
%   constraint_array = pf_surrogate.GetConstraintArray()
%
% Description:
%   The power flow surrogate implements the following constraints:
%   - Voltage magnitude limits
%   - Power injection at pcc
%   - Voltage at |pcc|
%
% Returns:
%
%   - **constraint_array** (constraint) - Array of constraints

% .. Line with 80 characters for reference ####################################

load_case = obj.opf_problem.load_case;
network = load_case.network;
n_time_step = obj.opf_problem.n_time_step;

% This power injection includes both controllable and uncontrollable loads
[P_inj_array,Q_inj_array] = obj.opf_problem.ComputeNodalPowerInjection();

% At this point, we need to define decision variables over which we will optimize.
% Most power flow surrogates need, at the very least, decision variables for voltage.
% Similar to uot.PowerFlowSurrogate_Bolognani2015_LP we use U_array_stack.
% However, we do not create a  T_array_stack because we do not need to keep
% track of phase information. This merits some explanation: in principle
% we could keep track of phase using eq. 5a. However, there is little
% use in doing so since phase does not figure in any of the relevant constraints.
U_array_stack = obj.decision_variables.U_array_stack;

% First, we implement the constraint on voltage at the pcc.
% By convention, bus 1 is the pcc
n_phase_pcc = network.n_phase_in_bus(1);

u_pcc_array = obj.opf_problem.u_pcc_array;
t_pcc_array = obj.opf_problem.t_pcc_array;

u_pcc_constraint = U_array_stack(1:n_phase_pcc,:) == u_pcc_array.';
% Tagging the constraints makes debugging much easier
u_pcc_constraint = uot.TagConstraintIfNonEmpty(u_pcc_constraint,'u_pcc_constraint');

% Second, we need to specify constraints on voltage magnitude.
% The paper presents two ways of doing this: with eq. 9 or eq. 12.
% We choose eq. 9 because in Figures 3 and 7, it shows better performance
% near kappa = 1 which is the nominal operating point.
% Our implementation of SolveApproxPowerFlowAlt shows us the way to go

x_y = [uot.StackPhaseConsistent(P_inj_array(2:end,:,:),network.bus_has_phase(2:end,:,
↪:));uot.StackPhaseConsistent(Q_inj_array(2:end,:,:),network.bus_has_phase(2:end,:,
↪:))];

% We create the method GetLinearizationXy to avoid code duplication with
% SolveApproxPowerFlowAlt. The method has to be static so that we can call it
% from SolveApproxPowerFlowAlt which is static.
```

(continues on next page)

(continued from previous page)

```
%
% We add linearization_point as a public property. We add it to the power flow surrogate
% and not to the spec to respect the principle of separating the opf_problem from
% the way we solve it.
[U_ast,T_ast] = obj.GetLinearizationVoltage();
[x_y_ast,V_ast_nopcc_stack] = PowerFlowSurrogate_Bernstein2017_LP_3.GetLinearizationXy(load_case,U_ast,T_ast);

% Again, we create ComputeMyMatrix and ComputeVoltageMagnitudeWithEq9 to avoid code duplication
M_y = PowerFlowSurrogate_Bernstein2017_LP_3.ComputeMyMatrix(network,V_ast_nopcc_stack);
U_array_eq9 = PowerFlowSurrogate_Bernstein2017_LP_3.ComputeVoltageMagnitudeWithEq9(network,u_pcc_array,x_y,x_y_ast,V_ast_nopcc_stack,M_y);

U_array_stack_eq9 = uot.StackPhaseConsistent(U_array_eq9,network.bus_has_phase);

% Now we set our decision variables U_array_stack to match U_array_stack_eq9.
% We exclude the pcc because we defined it above.
voltage_magnitude_def_constraint = U_array_stack((n_phase_pcc + 1):end,:) == U_array_stack_eq9((n_phase_pcc + 1):end,:);
voltage_magnitude_def_constraint = uot.TagConstraintIfNonEmpty(voltage_magnitude_def_constraint,'voltage_magnitude_def_constraint');

% Now we can define the constraint on voltage magnitude as done in
% uot.PowerFlowSurrogate_Bolognani2015_LP

% Voltage magnitude bound
% Non-enforced u_min constraint is given as 0. Here we convert it to -inf so that
% CreateBoxConstraint ignores the constraint
voltage_magnitude_spec = obj.opf_problem.spec.voltage_magnitude_spec;

if voltage_magnitude_spec.u_min == 0
    u_min = -inf;
else
    u_min = voltage_magnitude_spec.u_min;
end

u_max = voltage_magnitude_spec.u_max;

% Note that voltage magnitude bounds does not apply to pcc (which are the
% first n_phase_pcc elements in U_array_stack)
U_box_constraint = uot.CreateBoxConstraint(U_array_stack((n_phase_pcc+1):end,:),u_min,u_max,'U_box_constraint');

% Finally, we need to couple the pcc load to the rest of the problem. We do this
% through eqs. 5c and 13
v_pcc_array = uot.PolarToComplex(u_pcc_array,t_pcc_array);
[~,~,w] = network.ComputeNoLoadVoltage(u_pcc_array,t_pcc_array);

% We need to be careful when pre-allocating arrays for sdpvars
% s_pcc_array will be an sdpvar because x_y is an sdpvar. The key issue
% is that by doing
% A = zeros(2,2);
% A(1,1) = sdpvar(1,1)
```

(continues on next page)

(continued from previous page)

```
% The sdpvar appears as a double nan in A which is useless.
% For more information see https://yalmip.github.io/naninmodel/
%
% One way of doing this is creating a cell array and then concatenating the
% contents.
s_pcc_cell = cell(n_time_step,1);

for i = 1:n_time_step
    v_pcc_i = v_pcc_array(i,:).';
    G_y = diag(v_pcc_i)*conj(network.Ybus_SN)*conj(M_y);

    c = diag(v_pcc_i)*(conj(network.Ybus_SS)*conj(v_pcc_i) + conj(network.Ybus_
↪SN)*conj(w(:,i)));

    % Transpose to keep covension of phases being along dimension 2
    s_pcc_cell{i} = (G_y*x_y(:,i) + c).';
end

s_pcc_array = vertcat(s_pcc_cell{:});

p_pcc_array = real(s_pcc_array);
q_pcc_array = imag(s_pcc_array);

% Recall that bus 1 is the pcc. Further, we permute the dimensions
% so that they match (p_pcc_array has time along dimension 1 and P_inj_array
% has time along dimension 3).
p_pcc_array_constraint = p_pcc_array == uot.PermuteDims1and3(P_inj_array(1,:,:));
p_pcc_array_constraint = uot.TagConstraintIfNonEmpty(p_pcc_array_constraint,'p_pcc_
↪array_constraint');

q_pcc_array_constraint = q_pcc_array == uot.PermuteDims1and3(Q_inj_array(1,:,:));
q_pcc_array_constraint = uot.TagConstraintIfNonEmpty(q_pcc_array_constraint,'q_pcc_
↪array_constraint');

constraint_array = [
    u_pcc_constraint;
    voltage_magnitude_def_constraint;
    U_box_constraint;
    p_pcc_array_constraint;
    q_pcc_array_constraint
];
end
```

Trying out the power flow surrogate

We try solving an OPF problem with our brand new surrogate.

Generated from aaSolveOPF.m

```
clear variables
aaSetupPath
```

Initialize OPF problem

```
pf_surrogate_spec = PowerFlowSurrogateSpec_Bernstein2017_LP_3();
use_gridlab = false;
opf_problem = GetExampleOPFproblem(pf_surrogate_spec,use_gridlab);

% Select solver
opf_problem.sdpsettings = sdpsettings('solver','sedumi');
```

Solve OPF problem

The problem is infeasible (see error message at the bottom) which suggests that there might a bug in our implementation. As we mentioned at the beginning, implementing power flow surrogates is an error prone process.

```
[objective_value,solver_time,diagnostics] = opf_problem.Solve();
```

```
SeDuMi 1.32 by AdvOL, 2005-2008 and Jos F. Sturm, 1998-2003.
Alg = 2: xz-corrector, theta = 0.250, beta = 0.500
Put 38 free variables in a quadratic cone
eqs m = 55, order n = 91, dim = 129, blocks = 3
nnz(A) = 691 + 0, nnz(ADA) = 3025, nnz(L) = 1540
 it :     b*y       gap    delta  rate   t/tP*  t/tD*   feas cg cg  prec
  0 :            1.04E+00 0.000
  1 :   4.42E+00 3.77E-01 0.000 0.3644 0.9000 0.9000  -0.29  1  1  2.7E+00
  2 :   1.97E+00 1.63E-01 0.000 0.4313 0.9000 0.9000   2.66  1  1  6.7E-01
  3 :   1.34E+00 6.54E-02 0.000 0.4016 0.9000 0.9000   2.97  1  1  1.3E-01
  4 :   1.11E+00 2.11E-02 0.000 0.3232 0.9000 0.9000   1.39  1  1  4.4E-02
  5 :   1.03E+00 7.22E-03 0.000 0.3416 0.9000 0.9000   0.09  1  1  5.6E-02
  6 :   3.64E-01 5.78E-04 0.000 0.0801 0.9900 0.9900  -0.75  1  1  3.3E-02
  7 :   8.22E-01 1.29E-05 0.000 0.0224 0.9900 0.9900  -1.00  1  1  2.7E-02
  8 :   9.02E-02 1.37E-10 0.000 0.0000 1.0000 1.0000  -1.00  1  1  3.0E-02
  9 :   9.02E-02 3.12E-14 0.000 0.0002 0.9999 0.9999  -1.00  1  1  3.5E-02

Dual infeasible, primal improving direction found.
iter seconds  |Ax|    [Ay]_+     |x|       |y|
  9      0.1   1.9e-10   2.5e-11   4.5e+02   4.1e-11

Detailed timing (sec)
   Pre          IPM          Post
1.640E-02    7.147E-02    6.809E-03
Max-norms: ||b||=1, ||c|| = 5,
Cholesky |add|=0, |skip| = 0, ||L.L|| = 1.22198.
Warning: Solver experienced issues: Infeasible problem (SeDuMi-1.3)
```

Preparation for debugging the power flow surrogate

One good approach for debugging problems is to set the decision variables to the base case solution (i.e., when all controllable loads are zero). In most OPF problems this is a feasible solution, albeit a non-optimal one. This debugging process has been so helpful in the past that there is an abstract method just for this purpose: `AssignBaseCaseSolution()`.

We now implement `PowerFlowSurrogate_Bernstein2017_LP_3.AssignBaseCaseSolution()`. Once again, we use `uot.PowerFlowSurrogate_Bolognani2015_LP.AssignBaseCaseSolution` for inspiration.

```
function [U_array,T_array,p_pcc_array,q_pcc_array] = AssignBaseCaseSolution(obj)
% |protected| Assigns the :term:`base case` solution to the decision variables in the surrogate. Returns approximate power flow solution that is consistent with these values.
%
% Synopsis::
%
%   [U_array,T_array,p_pcc_array,q_pcc_array] = pf_surrogate.AssignBaseCaseSolution()
%
% Returns:
%
%   - **U_array** (double) - Phase consistent array (n_bus,n_phase,n_timestep) with voltage magnitudes
%   - **T_array** (double) - Empty array
%   - **p_pcc_array** (double) - Array (n_timestep,n_phase_pcc) with active power injection at the |pcc|
%   - **q_pcc_array** (double) - Array (n_timestep,n_phase_pcc) with reactive power injection at the |pcc|
%
% Note:
%  T_array is empty because this surrogate does not keep track of voltage angles.

% .. Line with 80 characters for reference ####################################

% This method assigns the base case solution (i.e., where all controllable
% loads are zero) to the decision variables in the surrogate.

u_pcc_array = obj.opf_problem.u_pcc_array;
t_pcc_array = obj.opf_problem.t_pcc_array;

% SolveApproxPowerFlowAlt operates in the same way as our optimization.
% Hence, since we are not considering controllable loads, the results from
% using it on load_case tell us the values that our decision variables
% should take in the base case solution.
load_case = obj.opf_problem.load_case;

% There are two important things to keep in mind:
% 1) In GetConstraintArray, we set decision_variables.U_array_stack equal to
% U_array_eq9. Whereas U_array in SolveApproxPowerFlowAlt comes from eq. 5a.
% The reason for this is that the constraint on minimal voltage magnitude is convex
% when using the formulation in eq. 9 but non convex when using the one in eq. 5a.
% We just need to be consistent.
%
% 2) we are not keeping track of phase. Hence, we ignore T_array
T_array = [];

[U_ast,T_ast] = obj.GetLinearizationVoltage();
[~,~, p_pcc_array, q_pcc_array,extra_data] = PowerFlowSurrogate_Bernstein2017_LP_3.SolveApproxPowerFlowAlt(load_case,u_pcc_array,t_pcc_array,U_ast,T_ast);

U_array = extra_data.U_array_eq9;

network = load_case.network;
U_array_stack = uot.StackPhaseConsistent(U_array,network.bus_has_phase);

% We use YALMIP's assign method to give the decision variables its values
assign(obj.decision_variables.U_array_stack,U_array_stack);
end
```

Debugging the power flow surrogate

We use `PowerFlowSurrogate_Bernstein2017_LP_3.AssignBaseCaseSolution()` to debug the power flow surrogate.

Generated from aaDebugPowerFlowSurrogate.m

```
clear variables
aaSetupPath
```

Initialize OPF problem

```
pf_surrogate_spec = PowerFlowSurrogateSpec_Bernstein2017_LP_3();
use_gridlab = false;
opf_problem = GetExampleOPFproblem(pf_surrogate_spec,use_gridlab);

% Select solver
opf_problem.sdpsettings = sdpsettings('solver','sedumi');
```

Try to solve

Problem is infeasible as we expected (see error message at the bottom)

```
[objective_value,solver_time,diagnostics] = opf_problem.Solve();
```

```
SeDuMi 1.32 by AdvOL, 2005-2008 and Jos F. Sturm, 1998-2003.
Alg = 2: xz-corrector, theta = 0.250, beta = 0.500
Put 38 free variables in a quadratic cone
eqs m = 55, order n = 91, dim = 129, blocks = 3
nnz(A) = 691 + 0, nnz(ADA) = 3025, nnz(L) = 1540
 it :     b*y       gap    delta  rate   t/tP*  t/tD*   feas cg cg  prec
  0 :            1.04E+00 0.000
  1 :   4.42E+00 3.77E-01 0.000 0.3644 0.9000 0.9000  -0.29  1  1  2.7E+00
  2 :   1.97E+00 1.63E-01 0.000 0.4313 0.9000 0.9000   2.66  1  1  6.7E-01
  3 :   1.34E+00 6.54E-02 0.000 0.4016 0.9000 0.9000   2.97  1  1  1.3E-01
  4 :   1.11E+00 2.11E-02 0.000 0.3232 0.9000 0.9000   1.39  1  1  4.4E-02
  5 :   1.03E+00 7.22E-03 0.000 0.3416 0.9000 0.9000   0.09  1  1  5.6E-02
  6 :   3.64E-01 5.78E-04 0.000 0.0801 0.9900 0.9900  -0.75  1  1  3.3E-02
  7 :   8.22E-01 1.29E-05 0.000 0.0224 0.9900 0.9900  -1.00  1  1  2.7E-02
  8 :   9.02E-02 1.37E-10 0.000 0.0000 1.0000 1.0000  -1.00  1  1  3.0E-02
  9 :   9.02E-02 3.12E-14 0.000 0.0002 0.9999 0.9999  -1.00  1  1  3.5E-02

Dual infeasible, primal improving direction found.
iter seconds  |Ax|    [Ay]_+     |x|       |y|
  9      0.0   1.9e-10   2.5e-11   4.5e+02   4.1e-11

Detailed timing (sec)
   Pre          IPM          Post
9.706E-03    1.948E-02    1.841E-03
Max-norms: ||b||=1, ||c|| = 5,
Cholesky |add|=0, |skip| = 0, ||L.L|| = 1.22198.
Warning: Solver experienced issues: Infeasible problem (SeDuMi-1.3)
```

Assign the base case solution

```
[U_array,T_array,p_pcc_array,q_pcc_array] = opf_problem.AssignBaseCaseSolution();
```

Check constraint satisfaction

The base case solution should be feasible. We can verify this by seeing if the constraints are fulfilled using YALMIP's check method (https://yalmip.github.io/command/check/). From this page, we see that "a solution is feasible if all residuals related to inequalities are non-negative."

```
constraint_array = opf_problem.GetConstraintArray();
check(constraint_array)
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|    ID|              Constraint|   Primal residual|                                              ␣
↪           Tag|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|    #1|   Elementwise inequality|                 0|   charger_611_12: p_box_
↪constraint lower bound|
|    #2|   Elementwise inequality|                 0|   charger_611_12: q_box_
↪constraint lower bound|
|    #3|   Elementwise inequality|                 0|   charger_611_12: q_box_
↪constraint upper bound|
|    #4|   Elementwise inequality|                 0|    charger_632_2: p_box_
↪constraint lower bound|
|    #5|   Elementwise inequality|                 0|    charger_632_2: q_box_
↪constraint lower bound|
|    #6|   Elementwise inequality|                 0|    charger_632_2: q_box_
↪constraint upper bound|
|    #7|   Elementwise inequality|                 0|   charger_652_13: p_box_
↪constraint lower bound|
|    #8|   Elementwise inequality|                 0|   charger_652_13: q_box_
↪constraint lower bound|
|    #9|   Elementwise inequality|                 0|   charger_652_13: q_box_
↪constraint upper bound|
|   #10|   Elementwise inequality|                 0|    charger_675_9: p_box_
↪constraint lower bound|
|   #11|   Elementwise inequality|                 0|    charger_675_9: q_box_
↪constraint lower bound|
|   #12|   Elementwise inequality|                 0|    charger_675_9: q_box_
↪constraint upper bound|
|   #13|   Elementwise inequality|            1.0102|       swing_load: p_box_
↪constraint lower bound|
|   #14|   Elementwise inequality|            1.0316|            s_sum_mag_max_
↪constraint swing_load|
|   #15|      Equality constraint|                 0|                          u_
↪pcc_constraint|
|   #16|      Equality constraint|       -1.1102e-16|          voltage_magnitude_
↪def_constraint|
|   #17|   Elementwise inequality|          -0.04039|                    U_box_
↪constraint lower bound|
|   #18|   Elementwise inequality|           0.04794|                    U_box_
↪constraint upper bound|
|   #19|      Equality constraint|                 0|                        p_pcc_
↪array_constraint|
```

(continues on next page)

(continued from previous page)

```
|   #20|      Equality constraint|                    0|                           q_pcc_
↪array_constraint|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Examine U_array

We see that "U_box_constraint lower bound" is clearly infeasible. Clearly, the constraint is violated since we set a minimal magnitude of 0.95 in GetExampleOPFproblem.

```
U_array
```

```
U_array =

    1.0625    1.0500    1.0687
    0.9328    0.9998    0.9136
    0.9328    0.9998    0.9136
    0.9328    0.9998    0.9136
    0.9266    1.0021    0.9117
    0.9308       NaN    0.9116
       NaN       NaN    0.9096
    0.9624    0.9904    0.9518
       NaN    0.9812    0.9499
       NaN    0.9795    0.9479
    0.9594    0.9885    0.9492
    0.9357    0.9698    0.9305
    0.9251       NaN       NaN
```

Examine U_array_ref

We now examine what U_array is if we solve power flow for the base case exactly. We see that all voltages are above 0.95. In fact, some are even above 1.05 which is the maximal allowed voltage magnitude in our OPF problem. This suggests that the issue is that the power flow surrogate is not very accurate in this case.

```
[U_array_ref,T_array_ref,p_pcc_array_ref,q_pcc_array_ref] = opf_problem.
↪SolvePFbaseCase();
U_array_ref
```

```
U_array_ref =

    1.0625    1.0500    1.0687
    0.9899    1.0532    0.9774
    0.9899    1.0532    0.9774
    0.9899    1.0532    0.9774
    0.9836    1.0554    0.9755
    0.9879       NaN    0.9754
       NaN       NaN    0.9734
    1.0210    1.0422    1.0173
       NaN    1.0332    1.0154
       NaN    1.0316    1.0134
    1.0180    1.0403    1.0147
    0.9940    1.0220    0.9959
    0.9821       NaN       NaN
```

Change linearization point

One way of increasing the accuracy is by bringing the linearization point closer to the operating conditions. Hence, we now linearize at the load in the first time step of the base case.

```
opf_problem.pf_surrogate.linearization_point = uot.enum.CommonLinearizationPoints.
↪PFbaseCaseFirstTimeStep;
```

Assign the new base case solution

We assign the base case solution again and examine U_array_2 It now matches U_array_ref exactly. This is not surprising since we are linearizing at precisely that operating point.

```
[U_array_2,T_array_2,p_pcc_array_2,q_pcc_array_2] = opf_problem.
↪AssignBaseCaseSolution();
U_array_2
```

```
U_array_2 =

    1.0625    1.0500    1.0687
    0.9899    1.0532    0.9774
    0.9899    1.0532    0.9774
    0.9899    1.0532    0.9774
    0.9836    1.0554    0.9755
    0.9879       NaN    0.9754
       NaN       NaN    0.9734
    1.0210    1.0422    1.0173
       NaN    1.0332    1.0154
       NaN    1.0316    1.0134
    1.0180    1.0403    1.0147
    0.9940    1.0220    0.9959
    0.9821       NaN       NaN
```

Check constraint satisfaction

We now see that U_box_constraint lower bound is feasible (i.e, has non-negative residual). We also notice that U_box_constraint upper bound is now infeasible. However, this is not a problem since we know that the exact solution has a few voltages above the upper limit of 1.05.

```
constraint_array = opf_problem.GetConstraintArray();
check(constraint_array)
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|     ID|                 Constraint|    Primal residual|                                                          ␣
↪             Tag|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|     #1|    Elementwise inequality|                   0|    charger_611_12: p_box_
↪constraint lower bound|
|     #2|    Elementwise inequality|                   0|    charger_611_12: q_box_
↪constraint lower bound|
|     #3|    Elementwise inequality|                   0|    charger_611_12: q_box_
↪constraint upper bound|
|     #4|    Elementwise inequality|                   0|     charger_632_2: p_box_
↪constraint lower bound|
```

(continues on next page)

(continued from previous page)

```
|    #5|   Elementwise inequality|                  0|    charger_632_2: q_box_
↪constraint lower bound|
|    #6|   Elementwise inequality|                  0|    charger_632_2: q_box_
↪constraint upper bound|
|    #7|   Elementwise inequality|                  0|   charger_652_13: p_box_
↪constraint lower bound|
|    #8|   Elementwise inequality|                  0|   charger_652_13: q_box_
↪constraint lower bound|
|    #9|   Elementwise inequality|                  0|   charger_652_13: q_box_
↪constraint upper bound|
|   #10|   Elementwise inequality|                  0|    charger_675_9: p_box_
↪constraint lower bound|
|   #11|   Elementwise inequality|                  0|    charger_675_9: q_box_
↪constraint lower bound|
|   #12|   Elementwise inequality|                  0|    charger_675_9: q_box_
↪constraint upper bound|
|   #13|   Elementwise inequality|            0.95731|       swing_load: p_box_
↪constraint lower bound|
|   #14|   Elementwise inequality|             1.0265|            s_sum_mag_max_
↪constraint swing_load|
|   #15|      Equality constraint|                  0|                          u_
↪pcc_constraint|
|   #16|      Equality constraint|                  0|        voltage_magnitude_
↪def_constraint|
|   #17|   Elementwise inequality|            0.02337|                  U_box_
↪constraint lower bound|
|   #18|   Elementwise inequality|         -0.0053805|                  U_box_
↪constraint upper bound|
|   #19|      Equality constraint|                  0|                    p_pcc_
↪array_constraint|
|   #20|      Equality constraint|                  0|                    q_pcc_
↪array_constraint|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Try to solve again

It works!

```
[objective_value,solver_time,diagnostics] = opf_problem.Solve();
```

```
SeDuMi 1.32 by AdvOL, 2005-2008 and Jos F. Sturm, 1998-2003.
Alg = 2: xz-corrector, theta = 0.250, beta = 0.500
Put 38 free variables in a quadratic cone
eqs m = 55, order n = 91, dim = 129, blocks = 3
nnz(A) = 691 + 0, nnz(ADA) = 3025, nnz(L) = 1540
 it :     b*y       gap    delta  rate   t/tP*  t/tD*   feas cg cg  prec
  0 :            1.04E+00 0.000
  1 :   4.56E+00 3.80E-01 0.000 0.3665 0.9000 0.9000  -0.30  1  1  2.7E+00
  2 :   2.04E+00 1.63E-01 0.000 0.4287 0.9000 0.9000   2.64  1  1  6.6E-01
  3 :   1.42E+00 7.16E-02 0.000 0.4399 0.9000 0.9000   3.36  1  1  1.1E-01
  4 :   1.14E+00 2.02E-02 0.000 0.2825 0.9000 0.9000   2.31  1  1  1.8E-02
  5 :   1.11E+00 7.07E-03 0.000 0.3500 0.9000 0.9000   1.32  1  1  5.7E-03
  6 :   1.11E+00 3.67E-03 0.000 0.5187 0.9000 0.9000   1.09  1  1  2.9E-03
  7 :   1.10E+00 8.78E-04 0.000 0.2393 0.9000 0.9000   1.06  1  1  6.8E-04
```

(continues on next page)

(continued from previous page)

```
  8 :   1.10E+00 7.15E-05 0.000 0.0814 0.9900 0.9900   1.01  1  1  5.6E-05
  9 :   1.10E+00 2.20E-06 0.000 0.0307 0.9900 0.9900   1.00  1  1  1.7E-06
 10 :   1.10E+00 6.71E-08 0.000 0.0305 0.9900 0.9900   1.00  1  1  5.3E-08
 11 :   1.10E+00 4.43E-09 0.000 0.0661 0.9900 0.9900   1.00  1  1  3.5E-09
 12 :   1.10E+00 8.84E-10 0.000 0.1996 0.9000 0.9000   1.00  1  2  7.0E-10

iter seconds digits       c*x               b*y
 12      0.0   8.7  1.0964634360e+00  1.0964634338e+00
|Ax-b| =   7.8e-10, [Ay-c]_+ =   6.9E-10, |x|=  3.6e+00, |y|=  8.2e+00

Detailed timing (sec)
   Pre          IPM          Post
7.222E-03    2.418E-02    1.891E-03
Max-norms: ||b||=1, ||c|| = 5,
Cholesky |add|=0, |skip| = 0, ||L.L|| = 6.80562.
```

Improving the test case

When we examined `PowerFlowSurrogate_Bolognani2015_LPtest()` in *Creating an initial test case* we saw that it included two test cases: `TestAgainstPaper()` and `TestPowerFlowSurrogate_Bolognani2015_LP()`. Earlier, we implemented the equivalent to `TestAgainstPaper()` for the power flow surrogate we are developing. Now, we implement the equivalent to `TestPowerFlowSurrogate_Bolognani2015_LP()`.

In the case of `PowerFlowSurrogate_Bolognani2015_LPtest()`,`TestPowerFlowSurrogate_Bolognani2015_LP(` calls a helper function `SolutionFulfillsLinearPowerFlowAndConstraints()` which first solves the OPF problem and then creates a load case which includes the values of the controllable loads in the optimal solution. Finally, it calls `uot.PowerFlowSurrogate_Bolognani2015_LP.SolveApproxPowerFlowAlt` using this load case.

The test `TestAgainstPaper()` gives us confidence that `uot.PowerFlowSurrogate_Bolognani2015_LP.SolveApproxPowerFlowAlt` is correct. In light of this, `PowerFlowSurrogate_Bolognani2015_LPtest()` then verifies that the decision variables in the optimization take values that are consistent with `uot.PowerFlowSurrogate_Bolognani2015_LP.SolveApproxPowerFlowAlt`. Then, it calls *uot.OPFproblem.AssertConstraintSatisfaction* which verifies that the decision variables in the OPF problem fulfill the specified constraints. This ensures that we did not forget to implement any constraints in the powerflow surrogate.

Note: By using a power flow surrogate we make the OPF problem tractable. However, we pay a price for this: we lose exact knowledge of the indirect variables (see [11] for a more detailed explanation). This means that having the decision variables in the OPF problem fulfill the specified constraints (which is what we verify in the test), does not guarantee that the constraints will be fulfilled if we solve the power flow equations using the values for the controllable loads. This difference becomes clear with the following code for a generic OPF problem

```
[objective_value,solver_time,diagnostics] = opf_problem.Solve();

% These are the values from the decision variables in the optimization problem
[U_array,T_array] = opf_problem.GetVoltageEstimate();
[p_pcc_array,q_pcc_array] = opf_problem.EvaluatePowerInjectionFromPCCload();

% These are the result of solving the power flow equations with the
% values of the controllable loads computed in the optimization
[U_array_pf,T_array_pf,p_pcc_array_pf,q_pcc_array_pf] = opf_problem.
↪SolvePFwithControllableLoadValues();
```

(continues on next page)

(continued from previous page)

```
% In general, U_array will not match U_array_pf, T_array will not match T_array_pf
% and so on.
```

Now, we go back to implementing our test case following the example of `PowerFlowSurrogate_Bolognani2015_LPtest()`.

```
function tests = PowerFlowSurrogate_Bernstein2017_LP_3test
% Verifies that |PowerFlowSurrogate_Bernstein2017_LP_3| works as expected
%
% Description:
%   This test verifies that:
%
%       - :meth:`PowerFlowSurrogate_Bernstein2017_LP_3.SolveApproxPowerFlowAlt` can replicate the results in :cite:`Bernstein2017`.
%       - The solution from the |opf| problem are consistent with those from :meth:`PowerFlowSurrogate_Bernstein2017_LP_3.SolveApproxPowerFlowAlt`
%       - The solution from the |opf| problem fulfills constraints

% This enables us to run the test directly instead of only through runtests
call_stack = dbstack;

% Call stack has only one element if function was called directly
if ~any(contains({call_stack.name},'runtests'))
    this_file_name = mfilename();
    runtests(this_file_name)
end

tests = functiontests(localfunctions);
end

function setupOnce(test_case)
aaSetupPath

test_case.TestData.abs_tol_equality = 5e-6;
end

function TestPowerFlowSurrogate_Bolognani2015_LP(test_case)
pf_surrogate_spec = PowerFlowSurrogateSpec_Bernstein2017_LP_3();
use_gridlab = false;
opf_problem = GetExampleOPFproblem(pf_surrogate_spec,use_gridlab);

% Select solver
opf_problem.sdpsettings = sdpsettings('solver','sedumi');

opf_problem.pf_surrogate.linearization_point = uot.enum.CommonLinearizationPoints.PFbaseCaseFirstTimeStep;
SolutionFulfillsLinearPowerFlowAndConstraints(test_case,opf_problem);
end

function SolutionFulfillsLinearPowerFlowAndConstraints(test_case,opf_problem)
opf_problem.ValidateSpec();

[objective_value,solver_time,diagnostics] = opf_problem.Solve();

```

(continues on next page)

(continued from previous page)

```
% Verify that optimization results match those from approximate power flow
% Since, PowerFlowSurrogate_Bernstein2017_LP_3 does not estimate phases,
% we ignore T_array
U_array = opf_problem.GetVoltageEstimate();

[p_pcc_array,q_pcc_array] = opf_problem.EvaluatePowerInjectionFromPCCload();

load_case = opf_problem.CreateLoadCaseIncludingControllableLoadValues();
u_pcc_array = opf_problem.u_pcc_array;
t_pcc_array = opf_problem.t_pcc_array;

[U_ast,T_ast] = opf_problem.pf_surrogate.GetLinearizationVoltage();
[~,~,p_pcc_array_ref,q_pcc_array_ref,extra_data] = PowerFlowSurrogate_Bernstein2017_
↪LP_3.SolveApproxPowerFlowAlt(load_case,u_pcc_array,t_pcc_array,U_ast,T_ast);

% Recall that our implementation uses eq 9 to estimate voltage. Hence,
% we need to be consistent.
U_array_ref = extra_data.U_array_eq9;

verifyEqual(test_case,U_array,U_array_ref,'AbsTol',test_case.TestData.abs_tol_
↪equality)

s_pcc_array = p_pcc_array + 1i*q_pcc_array;
s_pcc_array_ref = p_pcc_array_ref + 1i*q_pcc_array_ref;

verifyEqual(test_case,s_pcc_array,s_pcc_array_ref,'AbsTol',test_case.TestData.abs_tol_
↪equality)

% Verify constraint satisfaction
opf_problem.AssertConstraintSatisfaction();
end

function TestAgainstPaper(test_case)
load_case_zip = GetLoadCaseIEEE_13_NoRegs_Manual();
% Reference voltage from spec (we do not model the regulator)
u_pcc = [1.0625, 1.0500, 1.0687];
t_pcc = deg2rad([0, -120, 120]);

% Convert all loads to constant power wye-connected
load_case_pre = load_case_zip.ConvertToLoadCasePy();

% The paper uses kappa \in [-1,2]. However, we focus here on a narrower
% range where the approximate power flow is closest to the exact one.
kappa_vec = -0.2:0.1:1.2;

% Reorder kappa_vec so that kappa_vec(1) is 1.
% This, will allow us to linearize at the unscaled load
kappa_vec = [kappa_vec(kappa_vec == 1), kappa_vec(kappa_vec ~= 1)];

load_case = load_case_pre.*kappa_vec;

% Extend u_pcc_array and t_pcc_array to have one entry per time step
n_time_step = load_case.n_time_step;
u_pcc_array = repmat(u_pcc,n_time_step,1);
t_pcc_array = repmat(t_pcc,n_time_step,1);

linearization_point = uot.enum.CommonLinearizationPoints.PFbaseCaseFirstTimeStep;
```

(continues on next page)

(continued from previous page)

```
[U_ast,T_ast] = linearization_point.GetVoltageAtLinearizationPoint(load_case,u_pcc_
↪array,t_pcc_array);

[U_array,T_array, p_pcc_array, q_pcc_array,extra_data] = PowerFlowSurrogate_
↪Bernstein2017_LP_3.SolveApproxPowerFlowAlt(load_case,u_pcc_array,t_pcc_array,U_ast,
↪T_ast);

V_array = uot.PolarToComplex(U_array,T_array);
s_pcc = p_pcc_array + 1i*q_pcc_array;

% Get reference values from solving exact power flow
[U_array_ref,T_array_ref, p_pcc_array_ref, q_pcc_array_ref] = load_case.
↪SolvePowerFlow(u_pcc_array,t_pcc_array);

V_array_ref = uot.PolarToComplex(U_array_ref,T_array_ref);
s_pcc_ref = p_pcc_array_ref + 1i*q_pcc_array_ref;

% We select the initial tolerance based on aaReplicatePaperResults and
% we tune them so that the tests just pass.
v_tol = 6e-3;

% For simplicity, we compare elementwise instead of using the paper's relative
% error metric. This will also issue a more informative error if the test fails.
% We use absolute tolerances because voltage magnitude is close to 1
verifyEqual(test_case,V_array,V_array_ref,'AbsTol',v_tol)
verifyEqual(test_case,extra_data.U_array_eq9,U_array_ref,'AbsTol',v_tol)
verifyEqual(test_case,extra_data.U_array_eq12,U_array_ref,'AbsTol',v_tol)

% We compare s_pcc using relative error because it is not necessarily close to 1
% Note that the tolerance is much larger than the 0.02 error we saw in the plot
% of aaReplicatePaperResults. This is because here we normalize with the actual
% load and not the maximal one as done in the paper. Thus the errors for the
% small loads (small kappa) are much larger than before.
s_pcc_tol = 0.15;
verifyEqual(test_case,s_pcc,s_pcc_ref,'RelTol',s_pcc_tol)
end
```

Wrapping up the implementation

We have implemented almost all the abstract methods mentioned in *Barebones power flow surrogate*. We are only missing `PowerFlowSurrogate_Bernstein2017_LP_3.SolveApproxPowerFlow()`. Looking at `uot.PowerFlowSurrogate_Bolognani2015_LP.SolveApproxPowerFlow`, we can see that the method is really simple. It only tells `uot.AbstractPowerFlowSurrogate.SolveApproxPowerFlowHelper` what power flow spec to use. We go ahead and implement it.

Documentation

Documentation is key to make the code understandable for new users or for yourself after a few months have passed. As always, writing self-documenting code is a great starting point. However, it is typically very useful to have a header describing the code's behavior. The Unbalanced OPF Toolkit uses Sphynx and the MATLAB domain for documentation.

The documentation of code is based on headers which are writing using a particular format, see *Templates*. Then, the files must be added to an `rst` file as we will do here.

The reStructuredText markup and resulting output are below.

reStructuredText markup

```
.. Add substitutions for this tutorial
.. |PowerFlowSurrogateSpec_Bernstein2017_LP_3| replace::␣
↪:class:`PowerFlowSurrogateSpec_Bernstein2017_LP_3<demo.PowerFlowSurrogateTutorial.
↪@PowerFlowSurrogateSpec_Bernstein2017_LP_3.PowerFlowSurrogateSpec_Bernstein2017_LP_
↪3>`
.. |PowerFlowSurrogate_Bernstein2017_LP_3| replace:: :class:`PowerFlowSurrogate_
↪Bernstein2017_LP_3<demo.PowerFlowSurrogateTutorial.@PowerFlowSurrogate_
↪Bernstein2017_LP_3.PowerFlowSurrogate_Bernstein2017_LP_3>`


|PowerFlowSurrogateSpec_Bernstein2017_LP_3|
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.. automodule:: demo.PowerFlowSurrogateTutorial.@PowerFlowSurrogateSpec_Bernstein2017_
↪LP_3

.. autoclass:: PowerFlowSurrogateSpec_Bernstein2017_LP_3
        :members:
        :show-inheritance:


|PowerFlowSurrogate_Bernstein2017_LP_3|
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.. automodule:: demo.PowerFlowSurrogateTutorial.@PowerFlowSurrogate_Bernstein2017_LP_3

.. autoclass:: PowerFlowSurrogate_Bernstein2017_LP_3
        :members:
        :show-inheritance:

.. autofunction:: GetLinearizationVoltage
.. autofunction:: SolveApproxPowerFlow
.. autofunction:: SolveApproxPowerFlowAlt

Protected
"""""""""""
.. autofunction:: AssignBaseCaseSolution
.. autofunction:: ComputeVoltageEstimate
.. autofunction:: GetConstraintArray

Private
"""""""""""
.. autofunction:: ComputeMyMatrix
.. autofunction:: ComputeVoltageMagnitudeWithEq9
.. autofunction:: DefineDecisionVariables
.. autofunction:: GetLinearizationXy


PowerFlowSurrogate_Bernstein2017_LP_3test
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
.. autofunction:: demo.PowerFlowSurrogateTutorial.PowerFlowSurrogate_Bernstein2017_LP_
↪3test
```

PowerFlowSurrogateSpec_Bernstein2017_LP_3

class PowerFlowSurrogateSpec_Bernstein2017_LP_3

Bases: main.+uot.@AbstractPowerFlowSurrogateSpec.uot.AbstractPowerFlowSurrogateSpec

Class to specify the power flow surrogate presented in [4]

Synopsis:

```
spec = PowerFlowSurrogateSpec_Bernstein2017_LP_3()
```

Note: We add the suffix _3 to distinguish this implementation from the others that were developed in the course of the tutorial.

See also:

PowerFlowSurrogate_Bernstein2017_LP_3

PowerFlowSurrogate_Bernstein2017_LP_3

class PowerFlowSurrogate_Bernstein2017_LP_3(*spec*, *opf_problem*)

Bases: main.+uot.@AbstractPowerFlowSurrogate.uot.AbstractPowerFlowSurrogate

Implements the power flow surrogate presented in [4]

Synopsis:

```
obj = uot.PowerFlowSurrogate_Bernstein2017_LP_3(spec,opf_problem)
```

Description: This class implements the Bernstein2017 power flow surrogate presented in [4]. It uses an explicit linear formulation to approximate the power flow equation. The formulation is based on the fixed-point solution to the power flow equation.

This implementation does not keep track of phase since it is not necessary for implementing the required constraints.

Parameters

- **spec** (*PowerFlowSurrogateSpec_Bernstein2017_LP_3*) – Object specification
- **opf_problem** (*uot.OPFproblem*) – OPF problem where the power flow surrogate will be used

Note: The paper considers the posibility of delta-connected loads. However, we do not use them here since *uot.OPFproblem* does not support them.

Note: We add the suffix _3 to distinguish this implementation from the others that were developed in the course of the tutorial.

Example:

```
% This class is should be instantiated via its specification
spec = PowerFlowSurrogateSpec_Bernstein2017_LP_3();
pf_surrogate = spec.Create(opf_problem);
```

See also:

PowerFlowSurrogateSpec_Bernstein2017_LP_3

linearization_point = 'uot.enum.CommonLinearizationPoints.FlatVoltage'
Linearization point (uot.enum.CommonLinearizationPoints)

GetLinearizationVoltage(*obj*)
Returns the linearization voltage given by the choice of linearization_point

Synopsis:

```
[U_ast,T_ast] = pf_surrogate.GetLinearizationVoltage
```

Returns

- **U_ast** (double) - *Phase-consistent array* (n_bus,n_phase) with voltage magnitudes at the linearization point
- **T_ast** (double) - *Phase-consistent array* (n_bus,n_phase) with voltage angles at the linearization point

SolveApproxPowerFlow(*load_case*, *u_pcc_array*, *t_pcc_array*)
(static) Solves the power flow equations approximately using a power flow surrogate

Synopsis:

```
[U_array,T_array,p_pcc_array,q_pcc_array,opf_problem] = PowerFlowSurrogate_
↪Bernstein2017_LP_3.SolveApproxPowerFlow(load_case,u_pcc_array,t_pcc_array)
```

Description: Tells uot.AbstractPowerFlowSurrogate.SolveApproxPowerFlowHelper which power flow surrogate to use for approximately solving the power flow equations

Parameters

- **load_case** (uot.LoadCasePy) – Load case for which power flow will be solved
- **u_pcc_array** (*double*) – Array(n_phase,n_time_step) of voltage magnitudes at PCC
- **t_pcc_array** (*double*) – Array(n_phase,n_time_step) of voltage angles at PCC

Returns

- **U_array** (double) - *Phase-consistent array* (n_bus,n_phase,n_timestep) with voltage magnitudes
- **T_array** (double) - *Phase-consistent array* (n_bus,n_phase,n_timestep) with voltage angles
- **p_pcc_array** (double) - Array (n_timestep,n_phase_pcc) with active power injection at the PCC
- **q_pcc_array** (double) - Array (n_timestep,n_phase_pcc) with reactive power injection at the PCC
- **opf_problem** (*uot.OPFproblem*) - Power flow problem used to approximately solve power flow

See also:

`uot.AbstractPowerFlowSurrogate.SolveApproxPowerFlowHelper`

`SolveApproxPowerFlowAlt` (*load_case*, *u_pcc_array*, *t_pcc_array*, *varargin*)
(static) Approximately solve power flow

Synopsis:

```
[U_array,T_array,p_pcc_array,q_pcc_array,extra_data] = PowerFlowSurrogate_
↪Bernstein2017_LP_3.SolveApproxPowerFlowAlt(load_case,u_pcc_array,t_pcc_array)
[U_array,T_array,p_pcc_array,q_pcc_array,extra_data] = PowerFlowSurrogate_
↪Bernstein2017_LP_3.SolveApproxPowerFlowAlt(load_case,u_pcc_array,t_pcc_array,U_
↪ast,T_ast)
```

Description: Compute an approximate solution to the power flow equation using eqs. 5a and 5c in [4]. Additionally, use eq. 5c together with eq. 9 or 12 to compute another two approximations of voltage magnitude.

If no additional arguments are passed, the solution is computed by linearizing at the flat voltage solution. Alternatively, the linearization can be done at an arbitrary voltage.

Parameters

- **`load_case`** (`uot.LoadCasePy`) – Load case for which power flow will be approximately solved
- **`u_pcc_array`** (*double*) – Array(n_phase,n_time_step) of voltage magnitudes at the PCC
- **`t_pcc_array`** (*double*) – Array(n_phase,n_time_step) of voltage angles at the PCC
- **`U_ast`** (*double*) – *Phase-consistent array* (n_bus,n_phase) with voltage magnitudes for linearization
- **`T_ast`** (*double*) – *Phase-consistent array* (n_bus,n_phase) with voltage angles for linearization

Returns

- **U_array** (double) - *Phase-consistent array* (n_bus,n_phase,n_timestep) with voltage magnitudes
- **T_array** (double) - *Phase-consistent array* (n_bus,n_phase,n_timestep) with voltage angles
- **p_pcc_array** (double) - Array (n_timestep,n_phase_pcc) with active power injection at the PCC
- **q_pcc_array** (double) - Array (n_timestep,n_phase_pcc) with reactive power injection at the PCC
- **extra_data** (struct) - Struct with fields U_array_eq9 and U_array_eq12 containing the other two approximations of voltage magnitude.

See also:

`uot.AbstractPowerFlowSurrogate.SolveApproxPowerFlowHelper`

Protected

`AssignBaseCaseSolution` (*obj*)
(protected) Assigns the *base case* solution to the decision variables in the surrogate. Returns approximate power

flow solution that is consistent with these values.

Synopsis:

```
[U_array,T_array,p_pcc_array,q_pcc_array] = pf_surrogate.AssignBaseCaseSolution()
```

Returns

- **U_array** (double) - Phase consistent array (n_bus,n_phase,n_timestep) with voltage magnitudes
- **T_array** (double) - Empty array
- **p_pcc_array** (double) - Array (n_timestep,n_phase_pcc) with active power injection at the PCC
- **q_pcc_array** (double) - Array (n_timestep,n_phase_pcc) with reactive power injection at the PCC

Note: T_array is empty because this surrogate does not keep track of voltage angles.

ComputeVoltageEstimate (*obj*)
(protected) Computes the estimate for complex voltages given by the power flow surrogate

Synopsis:

```
[U_array,T_array] = pf_surrogate.ComputeVoltageEstimate()
```

Returns

- **U_array** (double) - *Phase-consistent array* (n_bus,n_phase,n_timestep) with voltage magnitudes
- **T_array** (double) - empty array

Note: T_array is empty because this surrogate does not keep track of voltage angles.

GetConstraintArray (*obj*)
(protected) Creates the array of constraints that result from applying the power flow surrogate to the OPF problem

Synopsis:

```
constraint_array = pf_surrogate.GetConstraintArray()
```

Description: The power flow surrogate implements the following constraints: - Voltage magnitude limits - Power injection at pcc - Voltage at PCC

Returns

- **constraint_array** (constraint) - Array of constraints

Private

ComputeMyMatrix (*network*, *V_ast_nopcc_stack*)
(static), (private) Computes the M_y matrix defined below eq. 7 in [4]

Synopsis:

```
M_y = ComputeMyMatrix(network,V_ast_nopcc_stack)
```

Parameters **V_ast_nopcc_stack** (*double*) – *phase-consistent stack* with complex voltages at the linearization point (excluding those for the PCC)

Returns

- **M_y** (double) - M_y matrix

ComputeVoltageMagnitudeWithEq9 (*network*, *u_pcc_array*, *x_y_array*, *x_y_ast*, *V_ast_nopcc_stack*, *M_y*)
(static), (private) Computes the voltage magnitude according to eqs. 5b and 9 in [4]

Synopsis:

```
U_array_eq9 = PowerFlowSurrogate_Bernstein2017_LP_3.
↪ComputeVoltageMagnitudeWithEq9(network,u_pcc_array,x_y,x_y_ast,V_ast_nopcc_
↪stack,M_y)
```

Description:

Parameters

- **network** (*uot.Network_Unbalanced*) – Power network
- **u_pcc_array** (*double*) – Array(n_phase,n_time_step) of voltage magnitudes at PCC
- **x_y_array** (*double*) – Array of power injections (according to definition in paper)
- **x_y_ast** (*double*) – Array of power injections at linearization point
- **V_ast_nopcc_stack** (*double*) – *phase-consistent stack* with complex voltages at the linearization point (excluding those for the PCC)
- **M_y** (*double*) – M_y matrix defined below eq. 7

Returns

- **U_array_eq9** (double) - *phase-consistent array* of voltage magnitudes

DefineDecisionVariables (*opf_problem*)
(static), (private) Defines the decision variables used by this power flow surrogate

Synopsis:

```
decision_variables = PowerFlowSurrogate_Bernstein2017_LP_3.
↪DefineDecisionVariables(opf_problem)
```

Description: Create the sdpvars that will be used as decision variables. Namely, a *phase-consistent stack* of voltage magnitudes

Parameters **opf_problem** (*uot.OPFproblem*) – OPF problem where the power flow surrogate will be used

Returns

- **decision_variables** (struct) - Struct with field U_array_stack which

Note: The method is static so that it can be safely called from the constructor

GetLinearizationXy(*load_case*, *U_ast*, *T_ast*)
(private), (static) Returns the linearization power injections at the linearization point as defined above eq. 5 in [4]

Synopsis:

```
[U_ast,T_ast] = PowerFlowSurrogate_Bernstein2017_LP_3.
↪GetLinearizationVoltage(load_case,U_ast,T_ast)
```

Parameters

- **U_ast** (*double*) – *Phase-consistent array* (n_bus,n_phase) with voltage magnitudes at the linearization point
- **T_ast** (*double*) – *Phase-consistent array* (n_bus,n_phase) with voltage angles at the linearization point

Returns

- **x_y_ast** (double) - Array of power injections at linearization point
- **V_ast_nopcc_stack** (double) - *phase-consistent stack* with complex voltages at the linearization point (excluding those for the PCC)

PowerFlowSurrogate_Bernstein2017_LP_3test

PowerFlowSurrogate_Bernstein2017_LP_3test()
Verifies that *PowerFlowSurrogate_Bernstein2017_LP_3* works as expected

Description: This test verifies that:

- `PowerFlowSurrogate_Bernstein2017_LP_3.SolveApproxPowerFlowAlt()` can replicate the results in [4].
- The solution from the OPF problem are consistent with those from `PowerFlowSurrogate_Bernstein2017_LP_3.SolveApproxPowerFlowAlt()`
- The solution from the OPF problem fulfills constraints

1.3 API

1.3.1 Power networks

Networks

uot.Network_Unbalanced

class Network_Unbalanced(*spec*)
Bases: `uot.AbstractNetwork`

Class to represent unbalanced power networks

Synopsis:

```
network = uot.Network_Unbalanced(spec)
```

Description: This class is used to represent unbalanced power networks

Parameters **spec** (uot.NetworkSpec) – Network specification

---

Note: Using the constructor directly is not recommended. Instead, create the object through the factory method uot.NetworkSpec.Create

---

See also:

uot.AbstractNetwork, uot.NetworkSpec

uot.Network_Prunned

class Network_Prunned(*network_full*, *removed_bus_name_cell*, *cut_link_name_cell*)

Bases: uot.Network_Unbalanced

Class to represent prunned power networks

Description: The process of prunning a network consists in cutting away certain links and replacing them with constant loads. This class is meant to deal with these objects that are represented by a network and a load case.

Synopsis:

```
obj = Network_Prunned(network_full,removed_bus_name_cell,cut_link_name_cell)
```

Parameters

- **network_full** (*uot.AbstractNetwork*) – Full network
- **removed_bus_name_cell** (*cell*) – Cell array with the names of buses to be removed
- **cut_link_name_cell** (*cell*) – Cell array with the names of the links to be cut and replaced with loads

validate = 'true'

Flag to validate AdaptLoadCase against original power flow solution

validate_tol = '1e-8'

Absolute tolerance to validate AdaptLoadCase

uot.Network_Splitphased

class Network_Splitphased(*spec*)

Bases: uot.AbstractNetwork

Class to represent unbalanced power networks with split phases

Synopsis:

```
network = uot.Network_Splitphased(spec)
```

Description: This class is used to represent unbalanced power networks with split phases

Parameters **spec** (uot.NetworkSpec) – Network specification

---

Note: Using the constructor directly is not recommended. Instead, create the object through the factory method uot.NetworkSpec.Create:

```
network = spec.Create()
```

See also:

uot.AbstractNetwork, uot.NetworkSpec

CreateNetworkWithPrunedSecondaries(*obj*)
Create a new network that results from prunning the split-phased secondaries

Synopsis:

```
network_prunned = network.CreateNetworkWithPrunedSecondaries()
```

See also:

uot.Network_Prunned

uot.AbstractNetwork

class AbstractNetwork(*spec*)
Bases: uot.Object

Abstract base class with common functionality for power networks

Description: This class builds the foundation to represent power networks.

Parameters

- **spec** (uot.NetworkSpec) – Network specification
- **bus_data_array** (uot.BusData) – Array describing the buses
- **link_data_array** (uot.LinkData) – Array describing the links

See also:

uot.NetworkSpec, *uot.Network_Unbalanced*, *uot.Network_Splitphased*

U_base_v = None
Graph theoretic variables

Y_shunt_bus_prec = '10'
Rounding precision of Y_shunt_bus to remove numerical noise. See CreateYshuntBusCell.

Ybus = None
Shunts

bus_data_array = None
Link variables

connectivity_graph = None
Ybus variables

link_data_array = None
Variables for per-unit system

ComputeCurrentInjectionFromVoltage(*obj*, *U_array*, *T_array*)
Compute current injection for a given voltage profile

Synopsis:

```
I_inj_array = network.ComputeCurrentInjectionFromVoltage(U_array,T_array)
```

Description: This method implements Ohm´s law $I_{inj} = Y_{bus} * V$ to compute current injection based on voltage.

Parameters

- **U_array** (*double*) – *Phase-consistent array* with voltage magnitudes
- **T_array** (*double*) – *Phase-consistent array* with voltage angles

Returns

- **I_inj_array** (double) - *Phase-consistent array* with complex power injection

ComputeLinkCurrentsAndPowers(*obj*, *U_array*, *T_array*)
Compute currents and power flow through links for a given voltage profile

Synopsis:

```
[I_link_from_array,I_link_to_array,S_link_from_array,S_link_to_array] = network.
↪ComputeCurrentInjectionFromVoltage(U_array,T_array)
```

Description: This method computes the current flowing through the network's links according to Eq. 4 in [2]. The power flows are then computed from these currents.

Parameters

- **U_array** (*double*) – *Phase-consistent array* with voltage mangitudes
- **T_array** (*double*) – *Phase-consistent array* with voltage angles

Returns

- **I_link_from_array** (double) - *Phase-consistent array* with link currents leaving the from bus
- **I_link_to_array** (double) - *Phase-consistent array* with link currents arriving at the to bus
- **S_link_from_array** (double) - *Phase-consistent array* with power flows leaving the from bus
- **S_link_to_array** (double) - *Phase-consistent array* with power flows arriving at the to bus

ComputePowerInjectionFromVoltage(*obj*, *U_array*, *T_array*)
Compute power injection for a given voltage profile

Synopsis:

```
[P_inj_array,Q_inj_array] = network.ComputePowerInjectionFromVoltage(U_array,T_
↪array)
```

Description: This method computes power injection based on the power flow equation.

$$S_{inj} = diag(V) * conj(Y_{bus} * V)$$

Parameters

- **U_array** (*double*) – *Phase-consistent array* with voltage mangitudes
- **T_array** (*double*) – *Phase-consistent array* with voltage angles

Returns

- **P_inj_array** (double) - *Phase-consistent array* with active power injection
- **Q_inj_array** (double) - *Phase-consistent array* with reactive power injection

Buses

uot.BusSpec_Unbalanced

class BusSpec_Unbalanced(*name*, *phase*, *u_nom_v*, *varargin*)

Bases: uot.AbstractBusSpec

Class to specify unbalanced power buses

Synopsis:

```
obj = BusSpec_Unbalanced(name,phase,u_nom_v)
obj = BusSpec_Unbalanced(name,phase,u_nom_v,'bus_type',bus_type)
```

Parameters

- **name** (*char*) – Bus name
- **phase** (*char*) – Bus' phase vector (as *logical phase vector*)
- **u_nom_v** (*double*) – nominal voltage (in volt)

Keyword Arguments 'bus_type' (*uot.enum.BusType*) – Bus type

phase = None

bus phase

uot.BusSpec_Splitphased

class BusSpec_Splitphased(*name*, *parent_phase*, *u_nom_v*)

Bases: uot.AbstractBusSpec

Class to specify power buses with split phases

Synopsis:

```
obj = BusSpec_Splitphased(name,parent_phase,u_nom_v)
```

Parameters

- **name** (*char*) – Bus name
- **parent_phase** (*logical*) – Phase to which the split phase bus is connected (as *logical phase vector*)
- **u_nom_v** (*double*) – nominal voltage (in volt)

parent_phase = None
bus parent phase

Interfaces

uot.AbstractLinkSpec

class AbstractLinkSpec(*name*, *from*, *to*, *Y_from_siemens*, *Y_to_siemens*, *Y_shunt_from_siemens*, *Y_shunt_to_siemens*, *created_from_Y_link*)
Bases: uot.Spec, matlab.mixin.Heterogeneous

AbstractLinkSpec Class with common functionality to specify links of the power network

Y_from_siemens = None
from-side admittance matrix

Y_shunt_from_siemens = None
from-side shunt admittance matrix

Y_shunt_to_siemens = None
to-side shunt admittance matrix

Y_to_siemens = None
to-side admittance matrix

created_from_Y_link = 'false'
flag to denote that LinkSpec was created using Y_line_siemens

from = None
link start bus

name = None
link name

to = None
link end bus

uot.AbstractBusSpec

class AbstractBusSpec(*name*, *u_nom_v*, *varargin*)
Bases: uot.Spec, matlab.mixin.Heterogeneous

AbstractBusSpec Class with common functionality to specify buses of the power network

Synopsis:

```
obj = AbstractBusSpec(name,u_nom_v)
obj = AbstractBusSpec(name,u_nom_v,'bus_type',bus_type)
```

Parameters

- **name** (*char*) – Bus name
- **u_nom_v** (*double*) – Nominal voltage (in volt)

Keyword Arguments 'bus_type' (*uot.enum.BusType*) – Bus type

bus_type = 'uot.enum.BusType.PQ'
bus type

name = None
bus name

u_nom_v = '1'
bus nominal voltage in volt

1.3.2 Loads

Load cases

uot.AbstractLoadCase

class AbstractLoadCase(*spec*, *network*)
Bases: uot.Object

Interface with common functionality for representing loads acting on a power network

Synopsis:

```
obj = uot.AbstractLoadCase(spec,network)
```

Parameters

- **spec** (uot.AbstractLoadCaseSpec) – Load case specification
- **network** (*uot.AbstractNetwork*) – Network on which the loads act

pf_abs_tol = '1e-9'
Absolute tolerance for solving power flow

pf_max_iter = '30'
Maximal number of iterations for solving power flow

SolvePowerFlow(*obj*, *u_pcc_array*, *t_pcc_array*)
Solve power flow for the load case acting on the network.

Synopsis:

```
[U_array,T_array, p_pcc_array, q_pcc_array] = load_case.SolvePowerFlow(u_pcc_
↪array,t_pcc_array)
```

Description: This function uses the Zbus iterative method [1] to solve the power flow problem.

Parameters

- **u_pcc_array** (*double*) – Array with voltage magnitude at the PCC
- **t_pcc_array** (*double*) – Array with phase at the PCC

Returns

- **U_array** (double) - *Phase-consistent array* (n_bus,n_phase,n_timestep) with voltage magnitudes
- **T_array** (double) - *Phase-consistent array* (n_bus,n_phase,n_timestep) with voltage angles
- **p_pcc_array** (double) - Array (n_timestep,n_phase_pcc) with active power injection at the PCC
- **q_pcc_array** (double) - Array (n_timestep,n_phase_pcc) with reactive power injection at the PCC

1.3.3 Optimal power flow problems

Optimal power flow problems are comprised of the following elements

- Network
- Load case
- Power flow surrogate
- Optimization objective
- Controllable loads
- Specification of voltage at PCC
- Constraints on voltage magnitude
- (Optional) additional constraints (e.g., on the PCC load or line currents)

As is common for the Toolkit, there is a class to specify the problem *@OPFspec.OPFspec* and one for the actual problem *@OPFproblem.OPFproblem*.

uot.OPFspec: Specify OPF problems

class OPFspec(*pf_surrogate_spec*, *controllable_load_spec_array*, *objective_spec*, *pcc_voltage_spec*, *voltage_magnitude_spec*, *varargin*)

Bases: uot.ProblemSpec

Class to specify an *uot.OPFproblem*.

Synopsis:

```
spec = uot.OPFspec(pf_surrogate_spec,controllable_load_spec_array,objective_spec,...
    pcc_voltage_spec,voltage_magnitude_spec,'pcc_load_spec',pcc_load_spec)
```

Description: This class incorporates several specifications needed to specify an *uot.OPFproblem*

Parameters

- **pf_surrogate_spec** (*uot.AbstractPowerFlowSurrogateSpec*) – Power flow surrogate specification
- **controllable_load_spec_array** (uot.ControllableLoadSpec) – Array of controllable load specifications
- **objective_spec** (uot.ObjectiveSpec) – Objective specification
- **pcc_voltage_spec** (uot.PCCvoltageSpec) – Voltage at the PCC specification

- **voltage_magnitude_spec** (uot.VoltageMaginitudeSpec) – Specification of constraints on voltage magnitude

Keyword Arguments 'pcc_load_spec' (uot.PCCloadSpec) – Specification of constraints on the PCC load

See also:

uot.OPFproblem

OPFspec(*pf_surrogate_spec*, *controllable_load_spec_array*, *objective_spec*, *pcc_voltage_spec*, *voltage_magnitude_spec*, *varargin*)
Allow no-argument constructor for preallocation

controllable_load_spec_array = None
Array of controllable load specifications (uot.ControllableLoadSpec)

objective_spec = None
Array of controllable load specifications (uot.ControllableLoadSpec)

pcc_load_spec = 'uot.PCCloadSpec()'
Specification of constraints on voltage magnitude (uot.PCCloadSpec)

pcc_voltage_spec = None
Array of controllable load specifications (uot.PCCvoltageSpec)

pf_surrogate_spec = None
Power flow surrogate specification (*uot.AbstractPowerFlowSurrogateSpec*)

voltage_magnitude_spec = None
Specification of constraints on the PCC load (uot.VoltageMaginitudeSpec)

uot.OPFproblem: Solve OPF problems

class OPFproblem(*spec*, *load_case*)
Bases: uot.Problem

Class to represent Optimal Power Flow problems

Synopsis:

```
opf_problem = uot.OPFproblem(spec,load_case)
```

Description: This class helps to formulate and solve unbalanced OPF problems

Parameters

- **spec** (*uot.OPFspec*) – Specification of the OPF problem
- **load_case** (uot.LoadCasePy) – Model of uncontrollable loads

---

Note: The OPF formulation considers only wye-connected constant power loads, represented in an uot.LoadCasePy object. The reason is that more detailed models are typically non-convex [3]. If loads are specified using the ZIP model, delta-to-wye conversions for some loads and approximating constant current and constant impedance loads as constant power ones is necessary. This can be done using uot.LoadCaseZip.ConvertToLoadCasePy.

OPFproblem(*spec*, *load_case*)
Allow no-argument constructor for preallocation

n_time_step = None
This is just for convenience

network = None
This is just for convenience

s_base_va = None
This is just for convenience

t_pcc_array = None
This is just for convenience

u_pcc_array = None
This is just for convenience

AssertConstraintSatisfaction(*obj*)
Verify that specified constraints are fulfilled

Synopsis:

```
opf_problem.AssertConstraintSatisfaction()
```

Description: The specification of the OPF problem *uot.OPFspec* contains several constraints which we want the solution to fulfill. This method verifies that the solution does indeed fulfill these constraints.

Note:

- This method requires solving the optimization problem first
- This method is meant to be used for debugging

AssignBaseCaseSolution(*obj*)
Assign no load solution to OPF problem

Synopsis:

```
[U_array,T_array,p_pcc_array,q_pcc_array] = opf_problem.AssignBaseCaseSolution()
```

Description: This methods assigns zero load to the controllable loads. It then requests the power flow surrogate to assign the no load solution (i.e., only including the uncontrollable loads. Finally, it sets the pcc load to the no load solution.

Returns

- **U_array** (double) - Voltage magnitude array of the no-load solution
- **T_array** (double) - Voltage angle array of the no-load solution
- **p_pcc_array** (double) - Real power of the swing load in the no-load solution
- **q_pcc_array** (double) - Reactive power of the swing load in the no-load solution

Note: This method is meant for debugging infeasible problems. Typically, the no load solution should be feasible. After calling this method, we can use YALMIP's `check()` function to find out what constraints are infeasible.

EvaluatePowerInjectionFromPCCload(*obj*)
Compute the current value of the PCC load

Synopsis:

```
[p_pcc_array_val,q_pcc_array_val] = opf_problem.
↪EvaluatePowerInjectionFromPCCload()
```

Description: Compute the current value of the PCC load (which is an sdpvar) and return it

Returns

- **p_pcc_array_val** (double) - Array(n_time_step,n_phase) with active power due to the PCC load
- **q_pcc_array_val** (double) - Array(n_time_step,n_phase) with reactive power due to the PCC load

See also:

`uot.OPFproblem.GetPowerInjectionFromPCCload`

GetConstraintArray(*obj*)
Assembles constraints for the OPF problem into an array

Synopsis:

```
constraint_array = opf_problem.GetConstraintArray()
```

Description: Requests constraints from controllable loads, PCC load and the used power flow surrogate. Then, puts them all into an array.

Implements `uot.ConstraintProvider.GetConstraintArray`

Returns

- **constraint_array** (constraint) - Array of constraints

AssignControllableLoadsToNoLoad(*obj*)
(private) Assign no load solution to controllable loads

Description: This methods assigns zero load to the controllable loads.

ComputeNodalPowerInjection(*obj*)
(private) Compute power injection at the buses from controllable and uncontrollable loads

Synopsis:

```
val = obj.ComputeNodalPowerInjection()
```

Description: This method sums the power injection from the controllable loads, if there are any, with the power injection from the uncontrollable loads in the load case.

Returns

- **P_inj_array** (sdpvar) - Array(n_bus, n_phase, n_time_step) of active power injection
- **Q_inj_array** (sdpvar) - Array(n_bus, n_phase, n_time_step) of reactive power injection

CreateControllableLoadHashTable(*obj*)

(private) Instantiate controllable loads based on specifications and store them in a hash table

Synopsis:

```
controllable_load_hashtable = obj.CreateControllableLoadHashTable()
```

Description: Instantiate controllable loads based on the specifications in `obj.spec.controllable_load_spec_array`. Store the resulting controllable loads in a hash table.

Returns

- **controllable_load_hashtable** (`containers.Map`) - Hash table of controllable loads

---

Note:

- We use a hash table to store the controllable loads so that we can refer to them by name
- The hash table is implemented as a `containers.Map`

CreateLoadSpecArrayWithControllableLoadValues(*obj*)

(private) Create an array of `uot.LoadPySpec` with the current value of the controllable loads

Synopsis:

```
load_spec_array = obj.CreateLoadSpecArrayWithControllableLoadValues()
```

Description: This method first evaluates the current value of the controllable loads. Then, these values are put into an array of `uot.LoadPySpec`. This is part of the process to solve power flow with the current values of the controllable loads.

Returns

- **load_spec_array** (`uot.LoadPySpec`) - Array(n_controllable_load) with the current value of the controllable loads

See also:

`uot.OPFproblem.SolvePFwithControllableLoadValues`

DefineObjective_LoadCost(*obj*, *objective_spec*)

(private) Define objective of minimizing the cost of controllable loads.

Synopsis:

```
objective = obj.DefineObjective_LoadCost()
```

Description: Compute the cost of the controllable loads according to the specification in `uot.OPFobjectiveSpec_LoadCost`.

Parameters **objective_spec** (uot.OPFobjectiveSpec_LoadCost) – Specification of controllable load cost

Returns

- **objective** (sdpvar) - Controllable load cost

See also:

uot.OPFobjectiveSpec_LoadCost

Todo:

- Explain in uot.OPFobjectiveSpec_LoadCost what the cost entails

---

GetControllableLoadConstraintArray(*obj*)
(private) Collects constraints from controllable loads into an array

Synopsis:

```
controllable_load_constraint_array = obj.GetControllableLoadConstraintArray()
```

Description: Goes through all controllable loads requesting their constraints. Then, assembles all of them in an array.

Returns

- **controllable_load_constraint_array** (constraint) - Array of constraints from controllable loads

See also:

uot.ControllableLoad.GetConstraintArray

Interfaces

uot.ConstraintProvider

class ConstraintProvider(*spec*, *decision_variables*)
Bases: uot.Object

Interface to define building blocks of optimization problems, they are not necessarily solvable.

Synopsis:

```
obj = uot.ConstraintProvider(spec,decision_variables)
```

Parameters

- **spec** (uot.Spec) – Object specification
- **decision_variables** (*struct*) – Struct with decision variables (as sdpvars)

constraint_tol = '1e-6'
Tolerance for AssertConstraintSatisfaction

1.3.4 Power flow surrogates

Fixed-point linear power flow model

uot.PowerFlowSurrogateSpec_Bernstein2017_LP

class PowerFlowSurrogateSpec_Bernstein2017_LP
Bases: uot.AbstractPowerFlowSurrogateSpec

Class to specify the power flow surrogate presented in [4].

Synopsis:

```
spec = PowerFlowSurrogateSpec_Bernstein2017_LP()
```

See also:

uot.PowerFlowSurrogate_Bernstein2017_LP

Create (*varargin*)
Returns an instance of *uot.PowerFlowSurrogate_Bernstein2017_LP*.

uot.PowerFlowSurrogate_Bernstein2017_LP

class PowerFlowSurrogate_Bernstein2017_LP (*spec*, *opf_problem*)
Bases: uot.AbstractPowerFlowSurrogate

Implements the power flow surrogate presented in [4]

Synopsis:

```
obj = uot.PowerFlowSurrogate_Bernstein2017_LP(spec,opf_problem)
```

Description: This power flow surrogate uses an explicit linear formulation to approximate the power flow equation. The formulation is based on the fixed-point solution to the power flow equation.

This implementation does not keep track of phase since it is not necessary for implementing the required constraints.

Parameters

- **spec** (*uot.PowerFlowSurrogateSpec_Bernstein2017_LP*) – Object specification
- **opf_problem** (*uot.OPFproblem*) – OPF problem where the power flow surrogate will be used

---

Note: The paper considers the possibility of delta-connected loads. However, we do not use them here since *uot.OPFproblem* does not support them.

Example:

```
% This class should be instantiated via its specification
spec = uot.PowerFlowSurrogateSpec_Bernstein2017_LP();
pf_surrogate = spec.Create(opf_problem);
```

See also:

uot.PowerFlowSurrogateSpec_Bernstein2017_LP

linearization_point = 'uot.enum.CommonLinearizationPoints.FlatVoltage'
Linearization point (uot.enum.CommonLinearizationPoints)

GetConstraintArray (*obj*)
(protected) Creates the array of constraints that result from applying the power flow surrogate to the OPF problem

Synopsis:

```
constraint_array = pf_surrogate.GetConstraintArray()
```

Description: The power flow surrogate implements at least the following constraints: - Voltage magnitude limits - Power injection at pcc - Voltage at PCC

Returns

- **constraint_array** (constraint) - Array of constraints

SolveApproxPowerFlow (*load_case*, *u_pcc_array*, *t_pcc_array*)
(static) Solves the power flow equations approximately using a power flow surrogate

Synopsis:

```
[U_array,T_array,p_pcc_array,q_pcc_array,opf_problem] = PowerFlowSurrogate_
↪Bernstein2017_LP.SolveApproxPowerFlow(load_case,u_pcc_array,t_pcc_array)
```

Description: Tells uot.AbstractPowerFlowSurrogate.SolveApproxPowerFlowHelper which power flow surrogate to use for approximately solving the power flow equations

Parameters

- **load_case** (uot.LoadCasePy) – Load case for which power flow will be solved
- **u_pcc_array** (*double*) – Array(n_phase,n_time_step) of voltage magnitudes at PCC
- **t_pcc_array** (*double*) – Array(n_phase,n_time_step) of voltage angles at PCC

Returns

- **U_array** (double) - *Phase-consistent array* (n_bus,n_phase,n_timestep) with voltage magnitudes
- **T_array** (double) - *Phase-consistent array* (n_bus,n_phase,n_timestep) with voltage angles
- **p_pcc_array** (double) - Array (n_timestep,n_phase_pcc) with active power injection at the PCC
- **q_pcc_array** (double) - Array (n_timestep,n_phase_pcc) with reactive power injection at the PCC
- **opf_problem** (*uot.OPFproblem*) - Power flow problem used to approximately solve power flow

See also:

uot.AbstractPowerFlowSurrogate.SolveApproxPowerFlowHelper

SolveApproxPowerFlowAlt (*load_case*, *u_pcc_array*, *t_pcc_array*, *varargin*)
(static) Approximately solve power flow

Synopsis:

```
[U_array,T_array,p_pcc_array,q_pcc_array,extra_data] = uot.PowerFlowSurrogate_
↪Bernstein2017_LP.SolveApproxPowerFlowAlt(load_case,u_pcc_array,t_pcc_array)
[U_array,T_array,p_pcc_array,q_pcc_array,extra_data] = uot.PowerFlowSurrogate_
↪Bernstein2017_LP.SolveApproxPowerFlowAlt(load_case,u_pcc_array,t_pcc_array,U_
↪ast,T_ast)
```

Description: Compute an approximate solution to the power flow equation using eqs. 5a and 5c in [4]. Additionally, use eq. 5c together with eq. 9 or 12 to compute another two approximations of voltage magnitude.

If no additional arguments are passed, the solution is computed by linearizing at the flat voltage solution. Alternatively, the linearization can be done at an arbitrary voltage.

Parameters

- **load_case** (uot.LoadCasePy) – Load case for which power flow will be approximately solved
- **u_pcc_array** (*double*) – Array(n_phase,n_time_step) of voltage magnitudes at the PCC
- **t_pcc_array** (*double*) – Array(n_phase,n_time_step) of voltage angles at the PCC
- **U_ast** (*double*) – *Phase-consistent array* (n_bus,n_phase) with voltage magnitudes for linearization
- **T_ast** (*double*) – *Phase-consistent array* (n_bus,n_phase) with voltage angles for linearization

Returns

- **U_array** (double) - *Phase-consistent array* (n_bus,n_phase,n_timestep) with voltage magnitudes
- **T_array** (double) - *Phase-consistent array* (n_bus,n_phase,n_timestep) with voltage angles
- **p_pcc_array** (double) - Array (n_timestep,n_phase_pcc) with active power injection at the PCC
- **q_pcc_array** (double) - Array (n_timestep,n_phase_pcc) with reactive power injection at the PCC
- **extra_data** (struct) - Struct with fields U_array_eq9 and U_array_eq12 containing the other two approximations of voltage magnitude.

See also:

uot.AbstractPowerFlowSurrogate.SolveApproxPowerFlowHelper

Protected

AssignBaseCaseSolution (*obj*)
(protected) Assigns the *base case* solution to the decision variables in the surrogate. Returns approximate power flow solution that is consistent with these values.

Synopsis:

```
[U_array,T_array,p_pcc_array,q_pcc_array] = pf_surrogate.AssignBaseCaseSolution()
```

Returns

- **U_array** (double) - Phase consistent array (n_bus,n_phase,n_timestep) with voltage magnitudes
- **T_array** (double) - Empty array
- **p_pcc_array** (double) - Array (n_timestep,n_phase_pcc) with active power injection at the PCC
- **q_pcc_array** (double) - Array (n_timestep,n_phase_pcc) with reactive power injection at the PCC

Note: T_array is empty because this surrogate does not keep track of voltage angles.

ComputeVoltageEstimate (*obj*)
(protected) Computes the estimate for complex voltages given by the power flow surrogate

Synopsis:

```
[U_array,T_array] = pf_surrogate.ComputeVoltageEstimate()
```

Returns

- **U_array** (double) - *Phase-consistent array* (n_bus,n_phase,n_timestep) with voltage magnitudes
- **T_array** (double) - empty array

Note: T_array is empty because this surrogate does not keep track of voltage angles.

Private

ComputeMyMatrix (*network*, *V_ast_nopcc_stack*)
(static), (private) Computes the M_y matrix defined below eq. 7 in [4]

Synopsis:

```
M_y = ComputeMyMatrix(network,V_ast_nopcc_stack)
```

Parameters **V_ast_nopcc_stack** (*double*) – *phase-consistent stack* with complex voltages at the linearization point (excluding those for the PCC)

Returns

- **M_y** (double) - M_y matrix

ComputeVoltageMagnitudeWithEq9 (*network*, *u_pcc_array*, *x_y_array*, *x_y_ast*, *V_ast_nopcc_stack*, *M_y*)
(static), (private) Computes the voltage magnitude according to eqs. 5b and 9 in [4]

Synopsis:

```
U_array_eq9 = PowerFlowSurrogate_Bernstein2017_LP.
↪ComputeVoltageMagnitudeWithEq9(network,u_pcc_array,x_y,x_y_ast,V_ast_nopcc_
↪stack,M_y)
```

Description:

Parameters

- **network** (*uot.Network_Unbalanced*) – Power network
- **u_pcc_array** (*double*) – Array(n_phase,n_time_step) of voltage magnitudes at PCC
- **x_y_array** (*double*) – Array of power injections (according to definition in paper)
- **x_y_ast** (*double*) – Array of power injections at linearization point
- **V_ast_nopcc_stack** (*double*) – *phase-consistent stack* with complex voltages at the linearization point (excluding those for the PCC)
- **M_y** (*double*) – M_y matrix defined below eq. 7

Returns

- **U_array_eq9** (double) - *phase-consistent array* of voltage magnitudes

DefineDecisionVariables (*opf_problem*)

(static), (private) Defines the decision variables used by this power flow surrogate

Synopsis:

```
decision_variables = PowerFlowSurrogate_Bernstein2017_LP.
↪DefineDecisionVariables(opf_problem)
```

Description: Create the sdpvars that will be used as decision variables. Namely, a *phase-consistent stack* of voltage magnitudes

Parameters **opf_problem** (*uot.OPFproblem*) – OPF problem where the power flow surrogate will be used

Returns

- **decision_variables** (struct) - Struct with field U_array_stack which

---

Note: The method is static so that it can be safely called from the constructor

---

GetLinearizationXy (*load_case*, *U_ast*, *T_ast*)

(private), (static) Returns the linearization power injections at the linearization point as defined above eq. 5 in [4]

Synopsis:

```
[U_ast,T_ast] = PowerFlowSurrogate_Bernstein2017_LP.GetLinearizationVoltage(load_
↪case,U_ast,T_ast)
```

Parameters

- **U_ast** (*double*) – *Phase-consistent array* (n_bus,n_phase) with voltage magnitudes at the linearization point

- **T_ast** (*double*) – *Phase-consistent array* (n_bus,n_phase) with voltage angles at the linearization point

Returns

- **x_y_ast** (double) - Array of power injections at linearization point
- **V_ast_nopcc_stack** (double) - *phase-consistent stack* with complex voltages at the linearization point (excluding those for the PCC)

Linearized power flow manifold model

uot.PowerFlowSurrogateSpec_Bolognani2015_LP

class PowerFlowSurrogateSpec_Bolognani2015_LP

Bases: uot.AbstractPowerFlowSurrogateSpec

Class to specify the power flow surrogate presented in [5].

Synopsis:

```
spec = PowerFlowSurrogateSpec_Bolognani2015_LP()
```

See also:

uot.PowerFlowSurrogate_Bernstein2017_LP

Create (*varargin*)

Returns an instance of *uot.PowerFlowSurrogate_Bolognani2015_LP*.

uot.PowerFlowSurrogate_Bolognani2015_LP

class PowerFlowSurrogate_Bolognani2015_LP (*spec*, *opf_problem*)

Bases: uot.AbstractPowerFlowSurrogate

Implements the power flow surrogate presented in [5]

Synopsis:

```
obj = uot.PowerFlowSurrogate_Bolognani2015_LP(spec,opf_problem)
```

Description: This power flow surrogate uses an implicit linear formulation to approximate the power flow equation. The formulation is based on the linearization of the power flow manifold.

This implementation does not keep track of phase since it is not necessary for implementing the required constraints.

Parameters

- **spec** (*uot.PowerFlowSurrogateSpec_Bolognani2015_LP*) – Object specification
- **opf_problem** (*uot.OPFproblem*) – OPF problem where the power flow surrogate will be used

Example:

```
% This class should be instantiated via its specification
spec = uot.PowerFlowSurrogateSpec_Bolognani2015_LP();
pf_surrogate = spec.Create(opf_problem);
```

See also:

uot.PowerFlowSurrogateSpec_Bolognani2015_LP

Todo:

- Check if sparse_precision is necessary. Maybe we can get rid of it.

Linear branch flow model

uot.PowerFlowSurrogateSpec_Gan2014_LP

class PowerFlowSurrogateSpec_Gan2014_LP

Bases: uot.AbstractPowerFlowSurrogateSpec

Class to specify the linear power flow surrogate presented in [6].

Synopsis:

```
spec = PowerFlowSurrogateSpec_Gan2014_LP()
```

See also:

uot.PowerFlowSurrogate_Gan2014_LP

Create (*varargin*)

Returns an instance of *uot.PowerFlowSurrogate_Gan2014_LP*.

uot.PowerFlowSurrogate_Gan2014_LP

class PowerFlowSurrogate_Gan2014_LP (*spec*, *opf_problem*)

Bases: uot.PowerFlowSurrogate_Gan2014

Implements the linear power flow surrogate presented in [6].

Synopsis:

```
obj = uot.PowerFlowSurrogate_Gan2014_LP(spec,opf_problem)
```

Description: This power flow surrogate is derived from the branch flow model SDP used in *uot.PowerFlowSurrogate_Gan2014_SDP* by assuming fixed ratios of voltages between phases in a bus and fixed line losses.

This implementation does not keep track of phase since it is not necessary for implementing the required constraints.

Parameters

- **spec** (*uot.PowerFlowSurrogateSpec_Gan2014_LP*) – Object specification
- **opf_problem** (*uot.OPFproblem*) – OPF problem where the power flow surrogate will be used

Note: This implementation uses the linearization method from [7]. The linearization used in the original paper [6] is a special case that can be recovered by setting:

```
pf_surrogate.linearization_point = uot.enum.CommonLinearizationPoints.FlatVoltage
```

Example:

```
% This class should be instantiated via its specification
spec = uot.PowerFlowSurrogateSpec_Gan2014_LP();
pf_surrogate = spec.Create(opf_problem);
```

See also:

uot.PowerFlowSurrogateSpec_Gan2014_LP

Convex branch flow model

uot.PowerFlowSurrogateSpec_Gan2014_SDP

class PowerFlowSurrogateSpec_Gan2014_SDP

Bases: uot.AbstractPowerFlowSurrogateSpec

Class to specify the convex (SDP) power flow surrogate presented in [6].

Synopsis:

```
spec = PowerFlowSurrogateSpec_Gan2014_SDP()
```

See also:

uot.PowerFlowSurrogate_Gan2014_SDP

Create(*varargin*)

Returns an instance of *uot.PowerFlowSurrogate_Gan2014_SDP*.

uot.PowerFlowSurrogate_Gan2014_SDP

class PowerFlowSurrogate_Gan2014_SDP(*spec*, *opf_problem*)

Bases: uot.PowerFlowSurrogate_Gan2014

Implements the convex (SDP) power flow surrogate presented in [6].

Synopsis:

```
obj = uot.PowerFlowSurrogate_Gan2014_SDP(spec,opf_problem)
```

Description: This power flow surrogate is a relaxation of the power flow equations for a radial power distribution network. The relaxation consists in removing a non-convex rank-constraint.

Since the surrogate is a relaxation, if the optimal solution happens to fulfill the rank constraint, then the solution is optimal for the original problem as well. If this is not the case, ComputeVoltageEstimate will throw a warning: "Max M eigenvalue ratio = %d is too high, solution is inaccurate.".

Parameters

- **spec** (*uot.PowerFlowSurrogateSpec_Gan2014_SDP*) – Object specification
- **opf_problem** (*uot.OPFproblem*) – OPF problem where the power flow surrogate will be used

Example:

```
% This class should be instantiated via its specification
spec = uot.PowerFlowSurrogateSpec_Gan2014_SDP();
pf_surrogate = spec.Create(opf_problem);
```

See also:

uot.PowerFlowSurrogateSpec_Gan2014_SDP

uot.PowerFlowSurrogate_Gan2014

class PowerFlowSurrogate_Gan2014 (*spec*, *opf_problem*, *decision_variables*)

Bases: uot.AbstractPowerFlowSurrogate

Abstract class with common functionality for *uot.PowerFlowSurrogate_Gan2014_LP* and *uot.PowerFlowSurrogate_Gan2014_SDP*.

See also:

uot.PowerFlowSurrogateSpec_Gan2014_SDP, *uot.PowerFlowSurrogate_Gan2014_LP*

Interface for power flow surrogates

uot.AbstractPowerFlowSurrogateSpec

class AbstractPowerFlowSurrogateSpec

Bases: uot.Spec, uot.Factory, matlab.mixin.Heterogeneous

Abstract class to specify a power flow surrogate

Description: The main purpose of an AbstractPowerFlowSurrogateSpec is telling the *uot.OPFproblem* how to instantiate the powerflow surrogate. This is necessary because there is a chicken and egg problem: to instantiate the surrogate, we need to pass it the OPFproblem. At the same time, to instantiate the OPFproblem we need the power flow surrogate.

uot.AbstractPowerFlowSurrogate

class AbstractPowerFlowSurrogate (*spec*, *opf_problem*, *decision_variables*)

Bases: uot.ConstraintProvider

Interface to implement power flow surrogates

Description: This abstract class specifies the interface to implement power flow surrogates.

Parameters

- **spec** (*uot.AbstractPowerFlowSurrogateSpec*) – Power flow surrogates specification
- **opf_problem** (*uot.OPFproblem*) – OPF problem

- **decision_variables** (*struct*) – Struct with decision variables (as sdpvars)

SolveApproxPowerFlowAlt (*load_case*, *u_pcc_array*, *t_pcc_array*, *varargin*)
(static) Approximately solve power flow

Synopsis:

```
[U_array,T_array,p_pcc_array,q_pcc_array,extra_data] = uot.
↪AbstractPowerFlowSurrogate.SolveApproxPowerFlowAlt(load_case,u_pcc_array,t_pcc_
↪array,varargin)
```

Description: It is very common that power flow surrogates offer a way of approximately solving power flow. This is typically an algebraic approach that does not rely on solving an optimization problem as done in `uot.AbstractPowerFlowSurrogate.SolveApproxPowerFlow` This method can be overriden for this purpose.

Parameters

- **load_case** (`uot.LoadCasePy`) – Load case for which power flow will be approximately solved
- **u_pcc_array** (*double*) – Array(n_phase,n_time_step) of voltage magnitudes at PCC
- **t_pcc_array** (*double*) – Array(n_phase,n_time_step) of voltage angles at PCC
- **varargin** – Additional arguments (implementation dependent)

Returns

- **U_array** (double) - *Phase-consistent array* (n_bus,n_phase,n_timestep) with voltage magnitudes
- **T_array** (double) - *Phase-consistent array* (n_bus,n_phase,n_timestep) with voltage angles
- **p_pcc_array** (double) - Array (n_timestep,n_phase_pcc) with active power injection at the PCC
- **q_pcc_array** (double) - Array (n_timestep,n_phase_pcc) with reactive power injection at the PCC
- **extra_data** (struct) - Struct with extra data (implementation dependent)

See also:

`uot.AbstractPowerFlowSurrogate.SolveApproxPowerFlowHelper`

1.3.5 Tests

The tests are located in the directory `src/test`. They can be run by setting this folder as MATLAB's current folder and executing `aaRunAllTests.m`.

1.3.6 Importing models from Gridlab

Todo: Note that #include uot_data.glm must be after clock since it is overridden

Todo: Add explanation on how to use gld models with UOT

Todo: Explain format and time series format

1.4 Conventions

1.4.1 File naming

- Code files are named in UpperCamelCase with an initial capital
- Files that are meant to be executed directly start with aa. For example, `aaSetEnvironment.m` and `aaUseCase_SolvePF.m`
- Function and class names are written in UpperCamelCase
- Variable names are generally lowercase (with some exceptions where uppercase is justified) and always separated with underscores
- Names for variables with units have the unit at the end. For example, `time_step_s` which is given in seconds. Throughout the code, we use base units (meter instead of kilometer, joule instead of kilo-watt-hour, watt instead of kilo-watt and so on)

1.4.2 Code conventions

Class constructors

We use `nargin > 0` in class constructors to allow for no-argument constructor. This allows MATLAB to preallocate objects.

Class property validation

We validate class properties for type and number of elements whenever possible.

Validating number of elements for abstract classes

In some cases, a property should have the type of an abstract class. For example, `opf_problem` in *uot.AbstractPowerFlowSurrogate* must be of type *uot.OPFproblem* which is abstract. In this case, we cannot set the property validation to:

```
opf_problem(1,1) uot.OPFproblem
```

because, since *uot.OPFproblem* cannot be instantiated directly, `opf_problem` will be empty when the object is created.

Instead, we use:

```
opf_problem(:,:) uot.OPFproblem {uot.NumelMustBeLessThanOrEqual(1,opf_problem)}
```

which allows for `opf_problem` to be empty at creation and constraints it to have at most 1 element.

1.5 Documentation style guide

1.5.1 Sphinx Matlab Domain

The code is documented using Sphinx Matlab Domain. To build the documentation, cd to `docs` and run

```
make html
```

Then open `docs/_build/index.html` with a browser.

1.5.2 Title formatting

Use the following punctuation characters in the section titles:

- * for Chapters
- = for sections ("Heading 1")
- – for subsections ("Heading 2")
- ^ for subsubsections ("Heading 3")
- " for paragraphs ("Heading 4")

1.5.3 Templates

Object template

class ObjectTemplate(*spec*, *varargin*)

Bases: `uot.Object`

Template class for uot.Object.

Synopsis:

```
obj = uot.ObjectTemplate(spec,'param1',val1)
```

Description: This class is intended as a starting point to develop classes that derive from `uot.Object`.

We can also itemize

- item 1
- item 2

Parameters **spec** (*uot.SpecTemplate*) – Object specification

Keyword Arguments **'param1'** (`uot.PCCloadSpec`) – Parameter

Note: Possibly add a note here

Example:

```
object = uot.ObjectTemplate(uot.SpecTemplate())
```

See also:

uot.SpecTemplate

Todo:

- Write documentation

ComputeValue(*arg_1*)
Compute value based on current state

Synopsis:

```
val = object.ComputeValue()
```

Description: Method description

We can also itemize

- item 1
- item 2

Parameters **arg_1** (*double*) – Argument 1

Returns

- **val_1** (double) - Val1
- **val_2** (double) - Val2

Note: Possibly add a note here

Example:

```
val = object.ComputeValue(arg_1);
```

See also:

uot.SpecTemplate

ObjectTemplate(*spec*, *varargin*)
Note: constructor does not have doc (it would repeat what is already in class)

prop_1 = None
Property 1 (double)

prop_2 = None
Property 2 (logical)

Spec template

class SpecTemplate(*bus*, *varargin*)
Bases: uot.Spec

Class to specify xx

Synopsis:

```
obj = uot.SpecTemplate(bus,varargin)
```

Description: This class is intended as a starting point to develop classes that derive from `uot.Spec`.

We can also itemize

- item 1
- item 2

Parameters **bus** (*char*) – Bus name

Keyword Arguments **'param1'** – Something.

Note: Possibly add a note here

Example:

```
object = uot.ObjectTemplate(uot.SpecTemplate())
```

See also:

uot.ObjectTemplate

Todo:

- Write documentation

SpecTemplate(*bus*, *varargin*)

Note: constructor does not have doc (it would repeat what is already in class)

1.6 Troubleshooting common problems

1.6.1 savepath

Sometimes when executing *savepath* in Ubuntu the following warning appears:

```
Warning: Unable to save path to file '/usr/local/MATLAB/R2019b/toolbox/local/pathdef.m
↪'. You can save your path to a different
location by calling SAVEPATH with an input argument that specifies the full path. For␣
↪MATLAB to use that path in future
sessions, save the path to 'pathdef.m' in your MATLAB startup folder.
```

This can be fixed by running:

```
sudo chown $USER /usr/local/MATLAB/R2019b/toolbox/local/pathdef.m
```

1.6.2 Calling GridLAB-D from MATLAB

Sometimes when calling GridLAB-D from MATLAB, the following error appears:

```
ERROR    [INIT] : gldcore/module.c(426): module 'powerflow' load failed - /usr/local/
↪MATLAB/R2017b/sys/os/glnxa64/libstdc++.so.6: version `GLIBCXX_3.4.21' not found␣
↪(required by /home/ubuntu/lib/gridlabd/powerflow.so)
```

To fix this problem, execute the following in a terminal:

```
cd /usr/local/MATLAB/R2019b/sys/os/glnxa64

sudo mv libstdc++.so.6 libstdc++.so.6.old
```

For more information, see here.

1.6.3 Sphinx documentation issues

- Sphinx aliases (e.g., `longtext| replace:: this is a very very long text to include`) do not work when documenting keyword arguments in MATLAB code files

1.7 To-do list

Todo: Note that #include uot_data.glm must be after clock since it is overridden

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/unbalanced-opf-toolkit/checkouts/latest/docs/api/index.rst, line 17.)

Todo: Add explanation on how to use gld models with UOT

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/unbalanced-opf-toolkit/checkouts/latest/docs/api/index.rst, line 18.)

Todo: Explain format and time series format

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/unbalanced-opf-toolkit/checkouts/latest/docs/api/index.rst, line 19.)

Todo:

- Explain in `uot.OPFobjectiveSpec_LoadCost` what the cost entails

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/unbalanced-opf-toolkit/checkouts/latest/src/main/+uot/@OPFproblem:docstring of main.+uot.@OPFproblem.DefineObjective_LoadCost, line 21.)

Todo:

- Check if sparse_precision is necessary. Maybe we can get rid of it.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/unbalanced-opf-toolkit/checkouts/latest/src/main/+uot/@PowerFlowSurrogate_Bolognani2015_LP:docstring of main.+uot.@PowerFlowSurrogate_Bolognani2015_LP.PowerFlowSurrogate_Bolognani2015_LP, line 28.)

Todo:

- Write documentation

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/unbalanced-opf-toolkit/checkouts/latest/src/main/+uot/@ObjectTemplate:docstring of main.+uot.@ObjectTemplate.ObjectTemplate, line 30.)

Todo:

- Write documentation

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/unbalanced-opf-toolkit/checkouts/latest/src/main/+uot/@SpecTemplate:docstring of main.+uot.@SpecTemplate.SpecTemplate, line 31.)

1.8 Glossary

base case Refers to the case when all the controllable loads are set to zero. Typically used in the context of OPF problems.

logical phase vector A definition

phase-consistent stack A definition

phase-consistent array A definition

1.9 References

CHAPTER 2

Indices and tables

- genindex
- search

Bibliography

[1] Mohammadhafez Bazrafshan and Nikolaos Gatsis. Convergence of the Z-Bus Method for Three-Phase Distribution Load-Flow with ZIP Loads. *IEEE Transactions on Power Systems*, 33(1):153–165, 2018. doi:10.1109/TPWRS.2017.2703835.

[2] Mohammadhafez Bazrafshan and Nikolaos Gatsis. Comprehensive Modeling of Three-Phase Distribution Systems via the Bus Admittance Matrix. *IEEE Transactions on Power Systems*, 33(2):2015–2029, 2018. doi:10.1109/TPWRS.2017.2728618.

[3] Joshua Adam Taylor. *Convex Optimization of Power Systems*. Cambridge University Press, Cambridge, UK, 1 edition, feb 2015. doi:10.1017/CBO9781139924672.

[4] Andrey Bernstein and Emiliano Dall'Anese. Linear power-flow models in multiphase distribution networks. In *2017 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*. Torino, Italy, sep 2017. IEEE. doi:10.1109/ISGTEurope.2017.8260205.

[5] Saverio Bolognani and Florian Dorfler. Fast power system analysis via implicit linearization of the power flow manifold. In *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 402–409. Monticello, IL, USA, sep 2015. IEEE. doi:10.1109/ALLERTON.2015.7447032.

[6] Lingwen Gan and Steven H. Low. Convex relaxations and linear approximation for optimal power flow in multiphase radial networks. In *2014 Power Systems Computation Conference*, 1–9. Wroclaw, Poland, aug 2014. IEEE. doi:10.1109/PSCC.2014.7038399.

[7] Michael D. Sankur, Roel Dobbe, Emma Stewart, Duncan S. Callaway, and Daniel B. Arnold. A Linearized Power Flow Model for Optimization in Unbalanced Distribution Systems. *arXiv preprint*, june 2016. arXiv:arXiv:1606.04492v1.

[8] J. Lofberg. YALMIP: a toolbox for modeling and optimization in MATLAB. In *2004 IEEE International Conference on Robotics and Automation*, 284–289. IEEE, 2004. doi:10.1109/CACSD.2004.1393890.

[9] David P. Chassin, Jason C. Fuller, and Ned Djilali. GridLAB-D: An agent-based simulation framework for smart grids. *Journal of Applied Mathematics*, 2014. doi:10.1155/2014/492320.

[10] Alvaro Estandia, Maximilian Schiffer, Federico Rossi, Emre Can Kara, Ram Rajagopal, and Marco Pavone. On the Interaction between Autonomous Mobility on Demand Systems and Power Distribution Networks – An Optimal Power Flow Approach. *arXiv preprint*, may 2019. arXiv:arXiv:1905.00200v1.

[11] Alvaro Estandia. On the interaction between autonomous mobility on demand systems and power distribution networks – an optimal power flow approach. Master's thesis, ETH Zurich, 2018. URL: https://www.research-collection.ethz.ch/handle/20.500.11850/340087.

MATLAB Module Index

d

m

Index

O

P

S

T

U

V

Y