```
classdef PhasorArray  < matlab.mixin.indexing.RedefinesParen &
matlab.mixin.indexing.RedefinesBrace

    properties
        Phasor3D %A 3D array reprensenting the harmonics of PhasorArray. Can
be a 3D Array of double, of sim or sdpvar
    end
    methods
        function obj = PhasorArray(varargin,varg)
            %Contructor for PhasorArray
            % Synthax :
            %   pA = PhasorArray(A) where A is 3D array of
            %       double/sym/sdpvar, with size(A,3) is odd.
            %   pA = PhasorArray(A, "z_pospart",true) where A is 3D array
            %       of double/sym/sdp, with A(:,:,1) represent the zero
phasor,
            %       A(:,:,k) is the k+1 phasor. Negative phasor are deduced
as
            %       conjugate of provided phasors.
            %   pA = PhasorArray(A0,Ap,"z_pospart",true) where A0 is a
            %       matrix and Ap a 3D Array of compatible size, A0 contain
the
            %       0 phasor, Ap the positive phasor. Negative phasor are
            %       deduced by conjugaison.
            %   pA = PhasorArray(A,"reduce",true) , delete symmetrically
extremal slice
            %       in A with value 0.
            %
            %Expect a 3D array of double, sim or sdp var
            % optionnal argument include
            %   -"reduce" (true/false) to automatically delete extremal 0
            %       phasor
            %   -"z_pospart" for zero positive part, in case only 0 and
            %       positive harmonics are given, and negative harmonics are
            %       deduce by taking conjugate of positive harmonics (for
real
            %       valued periodic matrices)
            %
            %   optionnal acceptable input
            %   PhasorArray(n,m) is the same as PhasorArray(zeros(n,m))
            %   PhasorArray(X0,Xpos,"z_pospart",true) is an alternate way
            %   to provide 0 pos part phasor
            arguments (Repeating)
                varargin
            end
            arguments
                varg.reduce=false
                varg.z_pospart=false
            end
            if nargin == 0
                obj = PhasorArray(1);
                return
```

```
            end

            if varg.z_pospart %si on a indiqué fournir un 0-phaosr et les
phasor positif
                if numel(varargin)==1
                    varP=varargin{1};
                    varP0=varP(:,:,1);
                    varPpos=varP(:,:,2:end);
                    obj = PosPart2PhasorArray(varP0,varPpos);
                else
                    assert(numel(varargin)==2,'If 0 and PosPhasor are
provided, only 2 argument must be provided')
                    obj = PosPart2PhasorArray(varargin{:});
                end
            elseif numel(varargin)>1 %SINON, si on a fourni plusieurs
argument scalaire, c'est qu'on fournit une taille pour un phasor array nul

assert(and(isscalar(varargin{1}),isscalar(varargin{2})),"error")
                obj.Phasor3D=zeros(varargin{:});
                varg.reduce =0;
            elseif isa(varargin{1},'PhasorArray') %si le premier arg est un
phasor array, tous les arguments suivant sont ignorés
                obj.Phasor3D = pvalue(varargin{1});
            elseif isa(varargin{1},'sparsePhasorArray') %si le premier arg
est un sparse phasor array, tous les arguments suivant sont ignorés
                obj = varargin{1}.toPhasorArray();
                return
            else
                assert(mod(size(varargin{1},3),2)==1,'Dim3 of 3D array
cannot be even')
                obj.Phasor3D = varargin{1};
            end
            if varg.reduce
                if isa(obj.Phasor3D,'double')
                    obj.Phasor3D=ReduceArray(obj.Phasor3D);
                end
            end

        end
        function info = info(o1, tol)
        % INFO Retrieve and evaluate properties of a PhasorArray object
        %   info = INFO(o1, tol) computes multiple characteristics of the
        %   PhasorArray, including harmonic information, real and imaginary
energy
        %   distribution, and shape properties. It returns a structured
output
        %   containing these metrics.
        %
        %   INPUTS:
        %       o1  - The PhasorArray object to analyze.
        %       tol - (Optional) A tolerance used to determine if the object
        %             is considered real. Defaults to machine precision if
not
        %             specified.
```

```
        %
        %   OUTPUT:
        %       info - A struct containing:
        %          - isReal             : Logical flag indicating if the
array
        %                                 is real within the specified
tolerance.
        %          - maxImagCoeff       : Maximum magnitude of the imaginary
        %                                 coefficients.
        %          - minImagCoeff       : Minimum magnitude of the imaginary
        %                                 coefficients.
        %          - maxRealCoeff       : Maximum magnitude of the real
        %                                 coefficients.
        %          - minRealCoeff       : Minimum magnitude of the real
        %                                 coefficients.
        %          - iscomplex          : Logical flag indicating if the
array
        %                                 has non-zero imaginary content.
        %          - realEnergy_EW      : Element-wise real energy
distribution.
        %          - imagEnergy_EW      : Element-wise imaginary energy
distribution.
        %          - realEnergy         : Total real energy.
        %          - imagEnergy         : Total imaginary energy.
        %          - Energy_EW          : Element-wise total energy
distribution.
        %          - Energy             : Total energy.
        %          - realRelativeEnergy : Fraction of total energy carried
by real
        %                                 parts.
        %          - imagRelativeEnergy : Fraction of total energy carried by
        %                                 imaginary parts.
        %          - imagRelativeEnergy_EW : Element-wise relative imaginary
energy.
        %          - realRelativeEnergy_EW : Element-wise relative real
energy.
        %          - h                  : Number of harmonics in the array.
        %          - size               : Size of the PhasorArray.
        %          - isSymmetric        : Logical flag for symmetry check.
        %          - isHermitian        : Logical flag for Hermitian check.
        %          - isSquare           : Logical flag for square dimension
check.
        %          - isScalar           : Logical flag for scalar dimension
check.
        %          - isVector           : Logical flag for vector dimension
check.
        %
        %   NOTE:
        %       If the array contains small imaginary components but is
flagged
        %       as real, proceed with caution if the imaginary energy is non-
zero.
            arguments
                o1
```

```
            tol  = []
        end
        if nargin == 1
            warning("tol is not specified, default to eps, real
evaluation of the PhasorArray may be affected")
        end
        %display information about the PhasorArray
        disp("PhasorArray of size "+num2str(size(o1))+" with
"+num2str(o1.h)+" harmonics")
        [r,~] = isreal(o1,tol);
        info.isReal = r;
        info.maxImagCoeff = max(abs(value(mimag(o1))),[],'all');
        info.minImagCoeff = min(abs(value(mimag(o1))),[],'all');
        info.maxRealCoeff = max(abs(value(mreal(o1))),[],'all');
        info.minRealCoeff = min(abs(value(mreal(o1))),[],'all');

        info.iscomplex = ~r;

        ro1 = mreal(o1);
        info.realEnergy_EW = sum(ro1.value.*conj(ro1.value),3);
        info.imagEnergy_EW =
sum(value(mimag(o1)).*conj(value(mimag(o1))),3);
        info.imagEnergy = sum(info.imagEnergy_EW,'all');
        info.realEnergy = sum(info.realEnergy_EW,'all');
        info.Energy_EW = info.realEnergy_EW + info.imagEnergy_EW;
        info.Energy = sum(info.Energy_EW,'all');
        info.realRelativeEnergy = info.realEnergy/info.Energy;
        info.imagRelativeEnergy = info.imagEnergy/info.Energy;
        info.imagRelativeEnergy_EW = info.imagEnergy_EW./info.Energy_EW;
        info.realRelativeEnergy_EW = info.realEnergy_EW./info.Energy_EW;



        info.h = o1.h;
        info.size = size(o1);
        info.isSymmetric = issymmetric(o1);
        info.isHermitian =ishermitian(o1);
        info.isSquare = issquare(o1);
        info.isScalar = isscalar(o1);
        info.isVector = isvector(o1);

        if info.imagEnergy>0 && info.isReal
            disp("PhasorArray is real within machine precision, but has
non zero imaginary energy, proceed with caution")
        end

    end
    function [Eew,E] = realEnergy(o1)
    %REALENERGY Compute the total real energy of the PhasorArray.
    %   [Eew,E] = REALENERGY(o1) returns element-wise real energy Eew
    %   and total real energy E.
        Eew = sum(o1.mreal.value.*conj(o1.mreal.value),3);
        E = sum(Eew,'all');
    end
```

```
        function [Eew,E] = imagEnergy(o1)
            %IMAGENERGY Compute the total imaginary energy of the
PhasorArray.
            %   [Eew,E] = IMAGENERGY(o1) returns element-wise imaginary
energy Eew
            %   and total imaginary energy E.
            Eew = sum(o1.mimag.value.*conj(o1.mimag.value),3);
            E = sum(Eew,'all');
        end

        function [Eew,E] = energy(o1)
            %ENERGY Compute the total energy of the PhasorArray.
            %   [Eew,E] = ENERGY(o1) returns element-wise energy Eew
            %   and total energy E.
            Eew = sum(o1.value.*conj(o1.value),3);
            E = sum(Eew,'all');
        end


        function [Eew,E] = pageEnergy(o1,normalized,cumulative,plotVar)
            %PAGEENERGY Computes and optionally plots the phasor energy in a
PhasorArray object.
            %
            %   [Eew, E] = pageEnergy(o1, normalized, cumulative, plotVar)
            %
            %   This function calculates the energy contributed by each
phasor in the
            %   PhasorArray object "o1", storing the element-wise energy in
"Eew" and
            %   the total energy in "E". When the "normalized" flag is set
to true, both
            %   outputs are normalized by their respective sums. The
"cumulative"
            %   argument specifies whether the energies should be
accumulated in forward
            %   ("cumulative") or reverse ("reverse") order, or left
unmodified ("none").
            %   The "plotVar" argument determines whether to visualize the
results in
            %   linear scale ("linear"), logarithmic scale ("log"), or skip
plotting
            %   ("none").
            %
            %   Inputs:
            %       o1          - PhasorArray object containing phasors
            %       normalized  - Logical flag to normalize outputs
(default: false)
            %       cumulative  - String indicating cumulative summation
mode:
            %                     'cumulative', 'reverse', or 'none'
(default: 'none')
            %       plotVar     - String for plotting options: 'none',
'linear', or
```

```
            %                       'log' (default: 'none')
            %
            %   Outputs:
            %       Eew         - Element-wise phasor energy (matrix)
            %       E           - Total energy of each phasor (vector)
            %
            %   Example Usage:
            %       % Compute and plot phasor energies in linear scale:
            %       [Eew, E] = pageEnergy(o1, false, 'none', 'linear');
            arguments
                o1
                normalized = false
                cumulative {mustBeMember(cumulative,
{'cumulative','reverse','none'})} = 'none'
                plotVar {mustBeMember(plotVar,{'none','linear','log'})} =
"none"
            end
            for hi = o1.h:-1:0
                oi = o1.extract(hi);
                [Eew(:,:,hi+1),E(hi+1)] = oi.energy;
            end
            if normalized
                [EewTotal,Etotal] = o1.energy;
                Eew = Eew./EewTotal;
                E = E./Etotal;
                E(isnan(E))=0;
                Eew(isnan(Eew))=0;
            end
            switch cumulative
                case "cumulative"
                    Eew = cumsum(Eew,3);
                    E = cumsum(E);
                case "reverse"

                    Eew = flip(Eew,3);
                    E = flip(E);
                    Eew = cumsum(Eew,3);
                    E = cumsum(E);
                    Eew = flip(Eew,3);
                    E = flip(E);
                case "none"
                otherwise
                    error("cumulative must be 'cumulative', 'reverse' or
'none'")
            end
            if strcmp(plotVar,"linear") || strcmp(plotVar,"log")
                if normalized
                    prefix = "Normalized ";
                else
                    prefix = "";
                end
                switch cumulative
                    case "cumulative"
                    plot(0:o1.h,E(end)-E)
```

```
                        title("Cumulative " +prefix+"energy of each phasor
(from 0 to harmonic order), 1-Energy")
                    case "reverse"
                    plot(0:o1.h,E)
                        title("Cumulative " +prefix+"energy of each phasor
in reverse order (from end to harmonic order)")
                    case "none"

                    plot(0:o1.h,E)
                        title(prefix+" Energy of each phasor")
                end
                xlabel("Harmonic order")
                ylabel("Energy")
                if strcmp(plotVar,"log")
                    set(gca,'YScale','log')
                end
            end
        end



        function r = plus(o1,o2)
            % PLUS Overloads the plus (+) operator for the PhasorArray class.
            %    PLUS(A,B) returns a new PhasorArray representing the
element-wise
            %    sum of A(t) and B(t). The second argument can be a scalar
or a
            %    scalar PhasorArray, which is broadcast to match the
dimensions of A.
            if isscalar(o2)
                o2=ones(size(o1,1),size(o1,2))*o2;
            end
            r = PhasorArray(PhasorArrayAdd(o1,o2));
        end
        function r = pageplus(o1,o2)
            %PAGEPLUS(A,B) : A+B = A(t)+B(t)
            %restricted to phasor of same third dimension
            %shouldn't be used directly, use PLUS instead
            %
            %See also PLUS
            r=o1.value+o2.value;
        end
        function r = minus(o1,o2)
            %MINUS overloading for PhasorArray
            %MINUS(A,B) : A-B = A(t)-B(t)
            %MINUS can accept scalar or scalar PhasorArray as second argument
            r = o1 +(-o2);
        end
        function r = uminus(o1)
            %UMINUS overloading for PhasorArray
            %UMINUS(A) : -A = -A(t)
            r = PhasorArray(-pvalue(o1));
        end
```

```
        function r = uplus(o1)
            %operator overloading for consistency, does nothing
            r = o1;
        end
        function r = pagetimes(o1,o2)
            %Realise le produit terme à terme de phaseurs de deux PhasorArray
            d=PhasorUnif(o1,o2);
            r = PhasorArray(pvalue(d{1}).*pvalue(d{2}));
        end
        function r = times(o1,o2)
        %TIMES Element-wise multiplication of two PhasorArray objects.
        %   R = TIMES(O1,O2) returns a PhasorArray containing the element-
wise
        %   multiplication of O1 and O2. Both inputs must be PhasorArray
objects
        %   (or unify to the same dimension).
            d=PhasorUnif(o1,o2);
            o1=d{1};
            o2=d{2};
            for ii = 1:size(o1,1)
            for jj = 1:size(o1,2)
                if ii==1 && jj==1
                rr=PhasorArrayTimes(o1(ii,jj,:),o2(ii,jj,:),reduce=false);
                r=zeros(size(o1,1),size(o1,2),size(rr,3));
                r(ii,jj,:)=rr;
                else

r(ii,jj,:)=PhasorArrayTimes(o1(ii,jj,:),o2(ii,jj,:),reduce=false);
                end
            end
            end
            r = PhasorArray(r);
        end
        function r = mtimes(o1,o2)
            %MTIMES Overloads the matrix multiplication operator for
PhasorArray.
            %   R = MTIMES(O1,O2) performs the time-domain product of the two
            %   PhasorArray objects O1 and O2 (convolution of the 3D arrays
along the third dimension), returning a PhasorArray result.
            r = PhasorArray(PhasorArrayTimes(o1,o2));
        end
        function r = rdivide(o1,o2,varargin)
        % RDIVIDE Overloads the right array division operator for the
PhasorArray class.
        %   RDIVIDE(A,B) returns a PhasorArray that represents A./B term by
term.
            for ii = 1:size(o2,1)
            for jj = 1:size(o2,2)
                d2(ii,jj,:)=PhasorInv(o2(ii,jj,:),varargin{:});
            end
            end
            d2=PhasorArray(d2);
            d=PhasorUnif(o1,d2);
            r = PhasorArray(d{1}.*d{2});
```

```
        end

        function r = ldivide(o1,o2,varargin)
        % LDIVIDE Overloads the left array division operator for the
PhasorArray class.
        %   LDIVIDE(A,B) returns a PhasorArray that represents A.\B term by
term.
            for ii = 1:size(o1,1)
            for jj = 1:size(o1,2)
                d1(ii,jj,:)=PhasorInv(o1(ii,jj,:),varargin{:});
            end
            end
            d1=PhasorArray(d1);
            d=PhasorUnif(d1,o2);
            r = PhasorArray(d{1}.*d{2});
        end
        function r = pagerdivide(o1,o2)
            %Realise la r division terme à terme des Phaseurs de 2
            %PhasorArray
            d=PhasorUnif(o1,o2);
            r = PhasorArray(d{1}./d{2});
        end
        function r = pageldivide(o1,o2)
            %Realise la l division terme à terme des Phaseurs de 2
            %PhasorArray
            d=PhasorUnif(o1,o2);
            r = PhasorArray(d{1}.\d{2});
        end
        function r = mrdivide(o1,o2,varargin)
            %MRDIVIDE Overloads the right matrix division operator for the
PhasorArray class.
            %   MRDIVIDE(A,B) returns a PhasorArray that represents A(t)/
B(t), which is
            %   equivalent to A * inv(B) in the time domain.
            %   This function performs the time-domain matrix division of
the two
            %   PhasorArray objects O1 and O2.
            d2=PhasorInv(o2,varargin{:});
            r = PhasorArray(PhasorArrayTimes(o1,d2));
        end
        function r = mldivide(o1,o2,varargin)
            %MLDIVIDE Overloads the left matrix division operator for the
PhasorArray class.
            %   MLDIVIDE(A,B) returns a PhasorArray that represents A(t)
\B(t), which is
            %   equivalent to inv(A) * B in the time domain.
            %   This function performs the time-domain matrix division of
the two
            %   PhasorArray objects O1 and O2.
            d1=PhasorInv(o1,varargin{:});
            r = PhasorArray(PhasorArrayTimes(d1,o2));
        end

        function r = pagepower(o1,m)
```

```
            % pagepower Element-wise exponentiation of phasors.
            %
            %   r = pagepower(o1, m) computes `o1 .^ m` phasor-wise,
applying exponentiation
            %   to each phasor coefficient individually.
            %
            %   Inputs:
            %   - o1: A PhasorArray.
            %   - m : A scalar or a PhasorArray of matching size.
            %
            %   Behavior:
            %   - If `m` is a scalar, applies `.^m` to each individual
phasor of `o1`.
            %   - If `m` is a PhasorArray, exponentiation is applied to
matching phasor coefficients.
            %   - Each frequency component of `o1` is exponentiated
separately.
            %
            %   Notes:
            %   - This is **not** equivalent to exponentiating the time-
domain matrix \( A(t) \).
            %   - Rarely useful in control or signal processing applications.
            %
            %   See also: power, mpower
            if isa(m,'PhasorArray')
                d=PhasorUnif(o1,m);
                m=d{2};
                r = PhasorArray((pvalue(d{1})).^(pvalue(m)));
            else
                r = PhasorArray(pvalue(o1).^m);
            end
        end
        function r = power(o1,m)
            % POWER Element-wise power of a periodic matrix A(t).
            %
            %   r = power(o1, m) computes the element-wise power of each
matrix entry of a periodic matrix \( A(t) \),
            %   represented as a PhasorArray. This is equivalent to
computing \( A(t)^{m} \) for each individual element of the matrix
            %   at each time instance in the time domain.
            %
            %   Inputs:
            %   - o1: A PhasorArray representing the periodic matrix \( A(t)
\).
            %   - m: The exponent (scalar value) to apply element-wise.
            %
            %   Output:
            %   - r: A PhasorArray representing the matrix \( A(t)^{m} \)
computed element-wise.
            %
            %   Behavior:
            %   - The function applies the power operation **element-wise**
for each entry in the time-domain representation of \( A(t) \).
            %   - This corresponds to raising each individual element \
```

```
( A_{ij}(t) \) of the matrix to the power \( m \).
            %
            %   See also: PhasorPow, mpower, logm, expm
            r=o1;
            % element wise power
            for ii=1:size(o1,1)
                for jj=1:size(o1,2)
                    r{ii,jj}=o1{ii,jj}^m;
                end
            end
        end
        function r = mpower(o1,m)
            % MPOWER Matrix power of a periodic matrix A(t), computed for
integer exponents.
            %
            %   r = mpower(o1, m) computes the matrix power \( A(t)^m \) for
a periodic matrix \( A(t) \),
            %   represented as a PhasorArray. This operation is only valid
for **integer exponents** and is done in the time domain.
            %   It computes the repeated matrix multiplication of \( A(t) \)
with itself \( m \) times.
            %
            %   Inputs:
            %   - o1: A PhasorArray representing the periodic matrix \( A(t)
\).
            %   - m: A positive or negative integer exponent representing
the power.
            %
            %   Output:
            %   - r: A PhasorArray representing \( A(t)^m \), the matrix
raised to the \( m \)-th power.
            %
            %   Behavior:
            %   - For positive integer \( m \), it multiplies \( A(t) \)
with itself \( m \) times.
            %   - For negative integer \( m \), it computes the inverse of \
( A(t) \) raised to the power \( |m| \) and then multiplies.
            %   - For non integer \(m \), it calls PhasorPow, which perform
time domain matrix power and then IFFT.
            %
            %   Notes:
            %   - This function applies matrix exponentiation, unlike
`power`, which operates element-wise.
            %
            %   See also: PhasorPow, power, logm, expm
            if mod(m,1)==0 && m>1
                prec=pvalue(o1);
                for ii=2:m
                    prec=PhasorArrayTimes(prec,o1);
                end
            elseif mod(m,1)==0 && m<0
                d1=PhasorInv(o1);
                prec=d1;
                for ii=-2:-1:m
```

```
                prec=PhasorArrayTimes(prec,d1);
            end
        else
            prec=PhasorPow(o1,m);
        end
        r=PhasorArray(prec);
    end

    function r = oplus(o1,o2)
        %OPLUS Kronecker sum of two PhasorArray objects.
        %   R = OPLUS(O1,O2) returns a PhasorArray representing the
Kronecker
        %   sum of O1 and O2.
        %
        %   The kronecker sum A(t) oplus B(t) is usually defined as
        %          A(t) otimes I + I otimes B(t)
        %   where I is the identity matrix of the same size as A(t) and
B(t)
        %   and otimes is the usual kronecker product.
        %
        %   see also KRON, TROPLUS, CTROPLUS
        r=PhasorArray(PhasorArrayOplus(o1,o2));
    end
    function r = troplus(o1,o2)
        %TROPLUS Kronecker sum of the transpose of two PhasorArray
objects.
        %   R = TROPLUS(O1,O2) returns a PhasorArray representing the
Kronecker
        %   sum of the transpose of O1 and O2.
        %
        %   see also KRON, OPLUS, CTROPLUS
        r=PhasorArray(PhasorArrayOplus(pagetranspose(pvalue(o1)),o2));
    end
    function r = ctroplus(o1,o2)
        %CTROPLUS Kronecker sum of the conjugate transpose of two
PhasorArray objects.
        %   R = CTROPLUS(O1,O2) returns a PhasorArray representing the
Kronecker
        %   sum of the conjugate transpose of O1 and O2.
        %
        %   see also KRON, OPLUS, TROPLUS
        r=PhasorArray(PhasorArrayOplus(pagectranspose(pvalue(o1)),o2));
    end
    function r = kron(o1,o2)
        %KRON Kronecker product of two PhasorArray objects.
        %   R = KRON(O1,O2) returns a PhasorArray representing the
Kronecker
        %   product of O1 and O2.
        %
        %   see also OPLUS, TROPLUS, CTROPLUS
        r=PhasorArray(PhasorArrayKron(o1,o2));
    end
    function r = retro(o1)
        %RETRO Reverse the time axis of the PhasorArray.
```

```
            %   B = RETRO(A) returns a PhasorArray B such that B(t) = A(-t).
            %   This function flips the PhasorArray along the third
dimension,
            %   effectively reversing the time axis.
            r=PhasorArray(flip((pvalue(o1)),3));
        end
        function r = trretro(o1)
            %TRRETRO Reverse the time axis and transpose the PhasorArray.
            %   B = TRRETRO(A) returns a PhasorArray B such that B(t) = A(-
t).'
            %   This function flips the PhasorArray along the third
dimension,
            %   effectively reversing the time axis and transposing each
page.
            r=PhasorArray(flip(pagetranspose(pvalue(o1)),3));
        end
        function r = ctrretro(o1)
            %CTRRETRO Reverse the time axis and conjugate transpose the
PhasorArray.
            %   B = CTRRETRO(A) returns a PhasorArray B such that B(t) = A(-
t)'.
            %   This function flips the PhasorArray along the third
dimension,
            %   effectively reversing the time axis and conjugate
transposing each page.
            r=PhasorArray(flip(pagectranspose(pvalue(o1)),3));
        end

        function out = extract(o1,index,symmetric)
            %EXTRACT Extract specified phasors from a PhasorArray.
            %   OUT = EXTRACT(O1, INDEX) returns a PhasorArray containing
only the
            %   phasors specified by INDEX. All other phasors are set to
zero.
            %
            %   OUT = EXTRACT(O1, INDEX, SYMMETRIC) allows for symmetric
extraction.
            %   If INDEX contains only positive integers and SYMMETRIC is
true, the
            %   corresponding negative phasors are also included. If INDEX
contains
            %   negative integers, SYMMETRIC is forced to false.
            %
            %   INPUTS:
            %       O1        - The PhasorArray object to extract from.
            %       INDEX     - A logical array matching the third dimension
of the
            %                   PhasorArray or a vector of positive and
negative integers.
            %       SYMMETRIC - (Optional) A logical flag indicating whether
to include
            %                   symmetric phasors. Defaults to true.
            %
            %   OUTPUT:
```

```
            %       OUT       - A PhasorArray containing only the specified
phasors.
            %
            %   EXAMPLES:
            %       A = PhasorArray.random(3, 3, 5);
            %       B = A.extract([1, 3, 5]);
            %       C = A.extract([1, 3], false);
            %       D = A.extract([1, 3], true);
            %
            %   See also trunc, reduce, neglect
            arguments
            o1
            index
            symmetric=true
            end
            if islogical(index)
            if numel(index)~=size(o1,3)
                error('logical array size must match third dim of
phasorArray')
            end
            index=find(index);
            %find complement of index
            indexc=setdiff(1:size(o1,3),index);

            out = o1;
            out(:,:,indexc)=0;
            else
            if all(index>=0) && symmetric
                index=[index -index];
                maxIndex = max(max(index),o1.h);
                indexc=setdiff(-maxIndex:maxIndex,index);
                out = o1;
                out{:,:,indexc}=0;
            else
                out = o1;
                maxIndex = max(max(abs(index)),o1.h);
                indexc=setdiff(-maxIndex:maxIndex,index);
                out{:,:,indexc}=0;
            end
            end
        end

        function r = trunc(o1,m)
            %TRUNC Truncate the PhasorArray to a specified number of phasors.
            %   R = TRUNC(O1, M) returns a new PhasorArray that is truncated
to
            %   contain only M phasors. If M is not specified, the function
will
            %   use a default value.
            %
            %   INPUTS:
            %       o1 - The original PhasorArray object to be truncated.
            %       m  - (Optional) The number of phasors to retain in the
truncated
```

```
            %            PhasorArray. If not provided, a default value is
used.
            %
            %   OUTPUT:
            %       r  - The truncated PhasorArray object containing only M
phasors.
            %
            %   EXAMPLE:
            %       A = PhasorArray.random(3, 3, 5);
            %       B = A.trunc(3);
            %       % B now contains only 3 phasors from the original
PhasorArray A.
            %
            %   See also REDUCE, NEGLECT

            arguments
            o1
            m=[]
            end
            r=PhasorArray(ReduceArray(o1,m));
        end
        function r = reduce(o1,htrunc,varg)
            %REDUCE Truncate or filter the PhasorArray based on harmonic
order or magnitude thresholds.
            %
            %   This function reduces a PhasorArray by either:
            %   - Truncating harmonics beyond a given order (`htrunc`).
            %   - Filtering harmonics based on their magnitude
(`reduceMethod` and `reduceThreshold`).
            %   - Optionally controlling how the 0th harmonic is handled
(`exclude0Phasor`).
            %   - Applying a hard or soft thresholding strategy
(`hardThresholdPhasors`).
            %
            %   Syntax:
            %   r = REDUCE(o1)
            %       Reduces `o1` using the default relative thresholding
method,
            %       preserving all harmonics unless their magnitudes are
below 1e-15.
            %
            %   r = REDUCE(o1, htrunc)
            %       Truncates `o1` to harmonics of order ≤ `htrunc`.
            %
            %   r = REDUCE(o1, htrunc, 'reduceMethod', method,
'reduceThreshold', threshold,
            %                'exclude0Phasor', exclude,
'hardThresholdPhasors', hardThreshold)
            %       Applies a combination of truncation and magnitude
filtering based on additional parameters.
            %
            %   Input Arguments:
            %   - o1 (PhasorArray) : The PhasorArray object to be reduced.
            %   - htrunc (integer, optional) : The maximum harmonic order to
```

```
retain. Default is empty ([]),
            %       meaning harmonics are only removed based on filtering
criteria.
            %
            %   Name-Value Pair Arguments:
            %   - 'reduceMethod' (char, optional) : Defines the reduction
approach.
            %       - 'absolute' : Removes phasors whose magnitude is below
the **absolute** threshold.
            %       - 'relative' (default) : Removes phasors whose magnitude
is below a **relative** threshold.
            %
            %   - 'reduceThreshold' (double, optional) : The threshold value
for filtering phasors.
            %       - If 'absolute', removes phasors with magnitude <
`reduceThreshold`.
            %       - If 'relative', removes phasors with magnitude <
`reduceThreshold * max(magnitude)`.
            %       - Default: 1e-15.
            %
            %   - 'exclude0Phasor' (logical, optional) :
            %       - true (default) : The 0th harmonic (DC component) is
always considered in max magnitude evaluation.
            %       - false : The 0th harmonic is ignored when computing the
relative maximum magnitude.
            %                 This affects thresholding in 'relative' mode.
            %
            %   - 'hardThresholdPhasors' (logical, optional) :
            %       - false (default) : `o1` is truncated to order `m`,
meaning **only harmonics beyond order m**
            %         are removed when all phasors of order > `m` fall below
the threshold.
            %         Lower-order phasors (< `m`) are **never** removed,
even if below threshold.
            %       - true : Any phasor below the threshold is explicitly
**set to zero**. Then, trailing zero phasors
            %         (i.e., those bearing no information) are removed to
optimize storage.
            %
            %   Output:
            %   - r (PhasorArray) : The reduced PhasorArray after applying
truncation or filtering.
            %
            %   Example:
            %   % Truncate PhasorArray to the 5th harmonic
            %   r = reduce(o1, 5);
            %
            %   % Apply absolute thresholding without truncating harmonics
            %   r = reduce(o1, [], 'reduceMethod', 'absolute',
'reduceThreshold', 1e-10, 'exclude0Phasor', false);
            %
            %   % Apply soft thresholding with truncation logic
            %   r = reduce(o1, [], 'hardThresholdPhasors', false,
'reduceMethod', 'relative', 'reduceThreshold', 1e-12);
```

```
            %
            %   % Apply hard thresholding where all small phasors are
explicitly set to zero
            %   r = reduce(o1, [], 'hardThresholdPhasors', true,
'reduceMethod', 'relative', 'reduceThreshold', 1e-12);
            %
            %   See also: ReduceArray
            arguments
                o1
                htrunc=[]
                varg.reduceMethod char {mustBeMember(varg.reduceMethod,
{'absolute','relative'})} = 'relative'
                varg.reduceThreshold {mustBeNumeric,mustBeReal} = 1e-15
                varg.exclude0Phasor (1,1) logical = true
                varg.hardThresholdPhasors=false
            end

r=PhasorArray(ReduceArray(o1,htrunc,reduceMethod=varg.reduceMethod,reduceThre
shold=varg.reduceThreshold,exclude0Phasor=varg.exclude0Phasor,hardThresholdPh
asors=varg.hardThresholdPhasors));
        end
        function r = neglect(o1, reduceThreshold, varg)
            %NEGLECT Set to zero all phasors below a given threshold in a
PhasorArray.
            %
            %   This function filters out low-magnitude phasors in a
PhasorArray `o1`,
            %   setting them to zero based on a specified threshold and
reduction method.
            %
            %   Syntax:
            %   r = NEGLECT(o1, reduceThreshold)
            %       Sets to zero all phasors in `o1` with magnitude below
`reduceThreshold`,
            %       using the default 'relative' thresholding method.
            %
            %   r = NEGLECT(o1, reduceThreshold, 'reduceMethod', method,
'exclude0Phasor', exclude, 'h', h)
            %       Applies phasor filtering with additional options.
            %
            %   Input Arguments:
            %   - o1 (PhasorArray) : The PhasorArray object whose small
phasors should be neglected.
            %   - reduceThreshold (double) : The magnitude threshold below
which phasors are set to zero.
            %
            %   Name-Value Pair Arguments:
            %   - 'reduceMethod' (char, optional) : Defines how the
threshold is applied.
            %       - 'absolute' : A phasor is set to zero if its magnitude
is below `reduceThreshold`.
            %       - 'relative' (default) : A phasor is set to zero if its
magnitude is below
            %         `reduceThreshold * max(magnitude)` (where the
```

```
reference maximum can exclude the 0th phasor).
            %
            %   - 'exclude0Phasor' (logical, optional) :
            %       - false (default) : The 0th phasor (DC component) **is
included** when computing the reference magnitude.
            %       - true : The 0th phasor is **ignored** when computing
the reference magnitude for relative thresholding.
            %
            %   - 'h' (integer, optional) :
            %       - If provided, reduces the PhasorArray to at most `h`
harmonics after applying the threshold.
            %       - Similar to `reduce(h)`, but performed after zeroing
small phasors.
            %
            %   Output:
            %   - r (PhasorArray) : The filtered PhasorArray, with phasors
below the threshold set to zero.
            %
            %   Example:
            %   % Set to zero all phasors with magnitude below 1e-15 using
default relative method
            %   r = neglect(o1, 1e-15);
            %
            %   % Apply absolute thresholding (removes all phasors with
magnitude < 1e-15)
            %   r = neglect(o1, 1e-15, 'reduceMethod', 'absolute');
            %
            %   % Use relative thresholding, ignoring the 0th phasor when
computing the max reference
            %   r = neglect(o1, 1e-15, 'reduceMethod', 'relative',
'exclude0Phasor', true);
            %
            %   % Apply thresholding and then truncate to at most 5 harmonics
            %   r = neglect(o1, 1e-12, 'h', 5);
            %
            %   See also: REDUCE, TRUNC, ReduceArray
            arguments
                o1
                reduceThreshold {mustBeNumeric,mustBeReal} = 1e-15
                varg.reduceMethod {mustBeMember(varg.reduceMethod,
{'absolute','relative'})}  = 'relative'
                varg.exclude0Phasor (1,1) logical = false
                varg.h=[]
            end
            val=o1.value;
            h=o1.h;
            switch  varg.reduceMethod
                case 'absolute'
                    val_rel=val;
                otherwise
                    if varg.exclude0Phasor
                        ref=max(abs(val(:,:,[1:h , (h+2):(2*h+1)])),[],3);
%maximum harmonic on each coeef, excepting the phasor 0.
                    else
```

```
                        ref=max(abs(val),[],3); %maximum harmonic on each
coeef, excepting the phasor 0.
                    end
                    val_rel=abs(val./ref);
            end
            val(abs(val_rel)<reduceThreshold)=0;
            r = PhasorArray(val);
            r =
r.reduce(varg.h,"reduceMethod",varg.reduceMethod,"reduceThreshold",reduceThre
shold,"exclude0Phasor",varg.exclude0Phasor);
        end


        function r = flip(o1,dim)
            %FLIP the PhasorArray along the dim dimension
            %   - dim : dimension to flip
            %   - r = flip(o1,dim) flip the PhasorArray along the dim
dimension
            %
            %   See also FLIPLR, FLIPUD
            %
            %   Inputs:
            %       o1  - PhasorArray object to be flipped
            %       dim - Dimension along which to flip the PhasorArray
            %
            %   Outputs:
            %       r   - Resulting PhasorArray after flipping
            %
            %   Example:
            %       pa = PhasorArray(rand(3,3,3));
            %       pa_flipped = flip(pa, 2);
            %
            %   Note:
            %       If dim is 3, a warning is issued indicating that the flip
            %       along the third dimension produces M(-t).
            if dim==3
                warning("flip along third dimension, produces M(-t)")
            end
            r=PhasorArray(flip(o1.value,dim));
        end
        function r = fliplr(o1)
            %FLIPLR Flip the PhasorArray left to right
            %   r = FLIPLR(o1) flips the PhasorArray o1 in the left-right
direction.
            %   The function returns a new PhasorArray r with the elements
flipped.
            %
            %   Example:
            %       pa = PhasorArray(rand(3,3,3));
            %       pa_flipped = fliplr(pa);
            %
            %   See also FLIP, FLIPUD

            r=PhasorArray(fliplr(o1.value));
```

```
        end
        function r = flipud(o1)
            %FLIPUD Flip the PhasorArray up to down
            %   r = FLIPUD(o1) flips the PhasorArray o1 in the up-down
direction.
            %   The function returns a new PhasorArray r with the elements
flipped.
            %
            %   Example:
            %       pa = PhasorArray(rand(3,3,3));
            %       pa_flipped = flipud(pa);
            %
            %   See also FLIP, FLIPLR

            r=PhasorArray(flipud(o1.value));
        end

        function r = rot90(o1,varargin)
            %ROT90 Rotate the PhasorArray by 90 degrees
            %   r = ROT90(o1) rotates the PhasorArray object o1 by 90 degrees
            %   counterclockwise and returns the result in r.
            %
            %   r = ROT90(o1, k) rotates the PhasorArray object o1 by 90
degrees
            %   counterclockwise k times.
            %
            %   Input:
            %       o1 - PhasorArray object to be rotated
            %       k  - (Optional) Number of times to rotate by 90 degrees
            %
            %   Output:
            %       r  - PhasorArray object after rotation
            %
            %   see also FLIP, FLIPLR, FLIPUD
            r=PhasorArray(rot90(o1.value,varargin{:}));
        end


        function r = horzcat(o1,varargin)
            %HORZCAT Concatenate PhasorArray objects horizontally
            %
            %   R = HORZCAT(O1, VARARGIN) concatenates the PhasorArray
object O1
            %   with additional PhasorArray objects provided in VARARGIN,
            %   horizontally, if their vertical dimensions are compatible.
            %
            %   Input arguments:
            %   O1 - The first PhasorArray object.
            %   VARARGIN - Additional PhasorArray objects to concatenate
with O1.
            %
            %   Output arguments:
            %   R - The resulting PhasorArray object after horizontal
concatenation.
```

```
            %
            %   If only one input argument is provided, the function returns
O1.
            %   If the third dimension sizes of O1 and the current object in
VARARGIN
            %   are not equal, the function uses PhasorUnif to unify them
before
            %   concatenation.
            %
            %   Example:
            %       r = horzcat(o1, o2, o3);
            %
            %   See also: VERTCAT, PHASORUNIF, PVALUE
            if nargin==1
                r=o1;
                return
            end
            for ii=1:numel(varargin)
                o2=varargin{ii};
                if size(o1,3) ~= size(o2,3)
                    d=PhasorUnif(o1,o2);
                    r = PhasorArray([pvalue(d{1}) pvalue(d{2})]);
                else
                    u1=pvalue(o1);
                    u2=pvalue(o2);
                    r = PhasorArray([u1 , u2]);
                end
                o1=r;
            end
        end
        function r = vertcat(varargin)
            %VERTCAT Concatenate PhasorArray objects vertically
            %   R = VERTCAT(O1, O2, ...) concatenates the PhasorArray object
O1
            %   with additional PhasorArray objects provided in VARARGIN,
            %   vertically, if their horizontal dimensions are compatible.
            %
            %   Input arguments:
            %   O1 - The first PhasorArray object.
            %   VARARGIN - Additional PhasorArray objects to concatenate
with O1.
            %
            %   Output arguments:
            %   R - The resulting PhasorArray object after vertical
concatenation.
            %
            %   If only one input argument is provided, the function returns
O1.
            %   If the third dimension sizes of O1 and the current object in
VARARGIN
            %   are not equal, the function uses PhasorUnif to unify them
before
            %   concatenation.
            %
```

```
            %   Example:
            %       r = vertcat(o1, o2, o3);
            %
            %   See also: HORZCAT, PHASORUNIF, PVALUE

            if nargin==1
            r=varargin{1};
            return
            end

            if nargin>=2
            r=cat(1,varargin{:});
            return
            else
            o1=varargin{1};
            o2=varargin{2};
            end
            if size(o1,3)~=size(o2,3)
            d=PhasorUnif(o1,o2);
            r = PhasorArray([pvalue(d{1}) ; pvalue(d{2})]);
            else
            u1=pvalue(o1);
            u2=pvalue(o2);
            r = PhasorArray([u1 ; u2]);
            end

        end
        function r = blkdiag(o1)
            %BLKDIAG of phasorarray
            %  - r = blkdiag(o1,o2,...) : block diagonal of phasorArray
            %
            %   See also HORZCAT, VERTCAT
            %
            %   This function creates a block diagonal matrix from the input
            %   phasor arrays. It takes multiple phasor arrays as input and
            %   constructs a block diagonal matrix where each block
corresponds
            %   to one of the input phasor arrays.
            %
            %   Input:
            %       o1 - Repeating argument list of phasor arrays
            %
            %   Output:
            %       r - Resulting PhasorArray object containing the block
            %           diagonal matrix
            %
            %   Example:
            %       pa1 = PhasorArray(rand(3,3,2));
            %       pa2 = PhasorArray(rand(2,2,2));
            %       result = blkdiag(pa1, pa2);
            %
            %   This example creates a block diagonal matrix from two phasor
            %   arrays pa1 and pa2.
            arguments (Repeating)
```

```
            o1
        end
        b1=PhasorUnif(o1{:});
        h=size(b1{1},3);
        out=cell(h,1);
        for hi=1:h
            argi=cell(numel(b1),1);
            for b1i=1:numel(b1)
                bb1i=pvalue(b1{b1i});
                argi{b1i}=bb1i(:,:,hi);
            end
            out{hi}=blkdiag(argi{:});
        end
        out=cat(3,out{:});
        r=PhasorArray(out);
    end
    function r = repmat(o1,M,N)
        %REPMAT Replicate and tile a PhasorArray.
        %   r = REPMAT(o1,M,N) replicates and tiles the PhasorArray o1 M
times along the first dimension and N times along the second dimension.
        %   r = REPMAT(o1,M) replicates and tiles the PhasorArray o1 M
times along the first and second dimensions.
        %   r = REPMAT(o1,M,N,L) replicates and tiles the PhasorArray o1
M times along the first dimension, N times along the second dimension, and L
times along the third dimension.
        %
        %   Inputs:
        %       o1 - The PhasorArray object to be replicated.
        %       M  - Number of times to replicate along the first
dimension.
        %       N  - (Optional) Number of times to replicate along the
second dimension.
        %
        %   Outputs:
        %       r  - The resulting PhasorArray after replication.
        %
        %   Example:
        %       pa = PhasorArray(rand(3,3,3));
        %       pa_replicated = repmat(pa, 2, 3);
        %
        %   See also REPMAT, RESHAPE

        arguments
        o1
        M
        end
        arguments (Repeating)
        N
        end
        switch nargin
        case 1
            r = o1;
        case 2
            switch numel(M)
```

```
                case 1
                    N=M;
                    L=1;
                    M=[M N L];
                case 2
                    N=M(2);
                    M=M(1);
                    L=1;
                    M=[M N L];
                case 3
                    warning("You shouldn't specify a third dimension for
repmat with phasorarray, proceed with caution")
                otherwise
                    warning("You shouldn't specify more than 2 dimensions
for repmat with phasorarray, proceed with caution")
                end
                r = PhasorArray(repmat(o1.value,M));
                return
            case 3
                assert(numel(M)==1)
                N=N{1};
                assert(numel(N)==1)
                L=1;
                r = PhasorArray(repmat(o1.value,M,N,L));
                return
            otherwise
                warning("You shouldn't specify more than 2 dimensions for
repmat with phasorarray, proceed with caution")
                r = PhasorArray(repmat(o1.value,M,N{:}));
                return
            end

        end
        function r = reshape(o1,varargin)
            % reshape Reshape a PhasorArray while preserving the phasor
dimension.
            %
            %   r = reshape(o1, M, N)     - Reshapes `o1` into an MxN
PhasorArray,
            %                               preserving the third (phasor)
dimension.
            %   r = reshape(o1, M, N, L)  - Reshapes `o1` into an MxNxL
PhasorArray.
            %
            %   Notes:
            %   - If only `M, N` are provided, `L` is automatically set to
`size(o1,3)`.
            %   - The third dimension (phasor order) should remain unchanged
in most cases.
            %
            %   See also: repmat, reshape
            if numel(varargin)==2
                varargin{3}=o1.size(3);
            end
```

```
            r=PhasorArray(reshape(pvalue(o1),varargin{:}));
        end
        function r = permute(o1,varargin)
            % permute Reorder dimensions of a PhasorArray.
            %
            %   r = permute(o1, order)   - Rearranges the dimensions of `o1`
according to `order`.
            %   r = permute(o1)          - Swaps the first and second
dimensions (acts as transpose).
            %
            %   Notes:
            %   - By default, swaps dimensions `[1 2]`, equivalent to a
transpose.
            %   - The third dimension (phasor order) is **preserved** unless
explicitly modified.
            %   - Usually applied to **matrix-like** PhasorArrays, not 3D
arrays.
            %
            %   See also: transpose, permute
            if nargin==1
                varargin{1}=[2 1];
            end
            if numel(varargin{1})==2
                varargin{1}=[varargin{1} 3];
            end
            r=PhasorArray(permute(pvalue(o1),varargin{:}));
        end
        function r = sub(o1,n1,n2)
            % sub Extract specific elements from a PhasorArray based on
indices.
            %
            %   r = sub(o1, n1, n2) extracts the phasor array component
            %   `Phasor3D(n1, n2, :)`, equivalent to `A{n1, n2}`.
            %
            %   Inputs:
            %       o1  - The PhasorArray object to extract from.
            %       n1  - Row indices or logical mask (can be `:` for all
rows).
            %       n2  - (Optional) Column indices (default: 1 if o1 is a
column vector).
            %
            %   Output:
            %       r   - Extracted PhasorArray elements with `reduce=false`
to
            %             preserve the structure.
            %
            %   Behavior:
            %   - If `n1` is a logical array, it is converted to indices.
            %   - If `o1` is a column vector (`size(o1,2)==1`), `n2`
defaults to 1.
            %   - If `n1` and `n2` are omitted, the function reshapes `r` to
match
            %     `size(n1,1) × size(n1,2)`.
            %
```

```
            %   See also: PhasorArray, reduce, logical indexing
            if nargin<3 %cas ou on a en apparence que une seule entrée de
coordonnée
                % cas 1 c'est une matrice d'entier
                % cas 2 c'est une matrice de logical, et il faut verif
                % qu'elle fait la bonne taille

                if ~strcmp(n1,':')%on a une matrice en input
                    if islogical(n1)
                        if numelt(o1)==numel(n1)
                            n1=find(logical(n1));
                        else
                            error('logical array size %d x %d must match
first two dim of phasorArray %d x
%d',size(n1,1),size(n1,2),size(o1,1),size(o1,1))
                        end
                    end
                end

                if size(o1,2)==1 % c'est déjà un phasorArray colonne, normal
de n'avoir eu qu'un indice
                    n2=1;
                else
                    o1=vect(o1); % on le rend colonne, ce sera plus simple à
manipuler
                    n2=1;
                end
            end
            r=PhasorArray(o1(n1,n2,:),reduce=false);
            if nargin == 1
            r = reshape(r,size(n1,1),size(n1,2));
            end
        end

        function r = vect(o1)
           %VECT transform phasorArray matrix to column vector (col operator)
            %   This function takes a PhasorArray object and transforms it
into
            %   a column vector by stacking the columns of the input matrix.
            %
            %   Syntax:
            %       r = vect(o1)
            %
            %   Input:
            %       o1 - PhasorArray object to be transformed.
            %
            %   Output:
            %       r  - Column vector (PhasorArray) obtained by stacking the
            %            columns of the input matrix.
            %
            %   Example:
            %       % Assuming o1 is a PhasorArray object
            %       r = vect(o1);
            %
```

```
            %   See also: reshape, pvalue
            r1=reshape(pvalue(o1),[] ,1 ,size(o1,3));
            r=PhasorArray(r1,reduce=false);
        end
        function r = pad(o1,delta_h)
            %PAD the PhasorArray with delta_h phasor
            %   - r = pad(o1,delta_h) : pad the PhasorArray o1 with delta_h
0 phasor
            %   - r = pad(o1,[delta_h1 delta_h2 delta_h3]) : pad the
PhasorArray o1 with delta_h1 0 phasor along the first dimension, delta_h2 0
phasor along the second dimension and delta_h3 0 phasor along the third
dimension
            %
            %pad (add zeros) to phasor array
            %pad(A, h) add h phasor to the phasor array
            %pad(A, [h1 h2 h3]) pad A in each direction
            r=PhasorArrayPad(o1,delta_h);
        end

        function r = ctranspose(o1)
            %CTRANSPOSE overloading for PhasorArray
            %CTRANSPOSE(A) : A' = A(t)'
            %transpose avec '
            r = mctranspose(o1);
        end
        function r = transpose(o1)
            %TRANSPOSE overloading for PhasorArray
            %TRANSPOSE(A) : A.' = A(t).'
            %transpose avec .'
            r = mtranspose(o1);
        end
        function r = pagectranspose(o1)
            %PAGECTRANSPOSE overloading for PhasorArray
            %PAGECTRANSPOSE(A) apply ctranspose to each page of A resulting
in A(-t)'
            r = PhasorArray(pagectranspose(pvalue(o1)));
        end
        function r = pagetranspose(o1)
            %PAGETRANSPOSE overloading for PhasorArray
            %PAGETRANSPOSE(A) apply transpose to each page of A resulting in
A(t).'
            % transpose appliqué à chacun des phaseurs
            r = PhasorArray(pagetranspose(pvalue(o1)));
        end
        function r = mtranspose(o1)
            % transpose appliqué à la matrice temporelle
            r=pagetranspose(o1);
        end
        function r = mctranspose(o1)
            % ctranspose appliqué à la matrice temporelle
            r=PhasorArray(flip(pagectranspose(o1.value),3));
        end

        function r = pmax(o1,o2,varg)
```

```
            % pmax Compute elementwise or phasorwise max between two
PhasorArray objects.
            %
            %   r = pmax(o1, o2, 'method', method) computes the maximum
between two
            %   PhasorArray objects `o1` and `o2` according to the specified
method.
            %
            %   Inputs:
            %       o1, o2  - PhasorArray objects to compare.
            %       method   - (Optional) Specifies the max computation
method:
            %                  * 'elementwise' (default): Max computed based
on norm.
            %                  * 'phasorwise': Max computed independently
for each phasor.
            %
            %   Output:
            %       r       - The resulting PhasorArray after applying the
max operation.
            %
            %   Notes:
            %   - In 'elementwise' mode, each phasor's norm is computed
first, and the
            %     element-wise max is applied to select the dominant phasor.
            %   - In 'phasorwise' mode, each corresponding phasor component
is compared directly.
            %   - Both inputs are unified in size using `PhasorUnif` before
processing.
            %
            %   See also: PhasorUnif, pvalue, norm
            arguments
                o1
                o2
                varg.method char {mustBeMember(varg.method,
{'elementwise','phasorwise'})} = 'elementwise'
            end
            if strcmp(varg.method,'elementwise')
                d=PhasorUnif(o1,o2);
                o1 = d{1};
                o2 = d{2};

                % norm of each phasor (along the third dim)
                n1=sum(o1.value .* conj(o1.value),3);
                n2=sum(o2.value .* conj(o2.value),3);
                %max of the norm
                n=max(n1,n2);


                %elementwise max
                r=PhasorArray(pvalue(o1).*(n1==n) + pvalue(o2).*(n1~=n));
            else
                d=PhasorUnif(o1,o2);
                r = PhasorArray(max(pvalue(d{1}),pvalue(d{2})));
```

```
            end
        end


        function varargout=phas(o1,h)
            % phas Extract the phasor of order `h` from a PhasorArray.
            %
            %   phas(o1, h) returns the phasor of order `h` from the
PhasorArray `o1`.
            %
            %   Inputs:
            %       o1 - The PhasorArray object.
            %       h  - The order of the phasor to extract.
            %            - If `h` is a scalar, returns a single phasor slice.
            %            - If `h` is a vector, returns multiple slices at
the specified orders.
            %
            %   Outputs:
            %       - A 3D array corresponding to the phasor(s) of order `h`.
            %
            %   Behavior:
            %   - If `abs(h) <= o1.h`, extracts the corresponding phasor.
            %   - If `h` is **out of range**, pads `o1` to match the highest
requested order.
            %   - If `h` is **beyond padding limits**, returns a zero matrix.
            %
            %   See also: pad, reduce, PhasorArray
            if isscalar(h)
                if abs(h)<=o1.h
                    varargout{1}=o1(:,:,(end+1)/2+h);
                else
                    varargout{1}=zeros(size(o1,[1 2]));
                end
            else
                if max(abs(h))>=o1.h
                    o1=o1.pad( max(abs(h))-o1.h);
                end
                varargout{1}=o1(:,:,(end+1)/2+h);
            end

        end
        function obj =phasAssign(obj,h,varargin)
            % phasAssign Assign values to specific phasor components in a
PhasorArray.
            %
            %   obj = phasAssign(obj, h, values) returns a copy of the
PhasorArray
            %   with the phasor at index h replaced by the provided values.
            %
            %   Inputs:
            %       obj    - The PhasorArray object to modify.
            %       h      - The index (or indices) of the phasor(s) to be
assigned.
            %       values - The new values to be assigned to the specified
```

```
phasors.
            %
            %   Outputs:
            %       obj    - A modified PhasorArray with updated phasor
values.
            %
            %   Notes:
            %   - If h exceeds the current order of the PhasorArray, it is
automatically expanded.
            %   - This method does not modify the original object but
returns a modified copy.
            %   - The provided values must be compatible in size with the
existing array dimensions.
            %
            %   See also: phas, sub, reduce, trunc
            obj{:,:,h}=varargin{:};
        end


        function r = lt(o1,o2)
            r= (pvalue(o1)<pvalue(o2));
        end
        function r = gt(o1,o2)
            r= (pvalue(o1)>pvalue(o2));
        end
        function r = le(o1,o2)
            r= (pvalue(o1)<=pvalue(o2));
        end
        function r = ge(o1,o2)
            r= (pvalue(o1)>=pvalue(o2));
        end
        function r = ne(o1,o2)
            r= (pvalue(o1)~=pvalue(o2));
        end
        function r = eq(o1,o2)
            r= (pvalue(o1)==pvalue(o2));
        end
        function r = and(o1,o2)
            r= (pvalue(o1) & pvalue(o2));
        end
        function r = or(o1,o2)
            r= (pvalue(o1) | pvalue(o2));
        end
        function r = not(o1)
            r= not(pvalue(o1));
        end
        function r = double(o1)
            %DOUBLE Convert PhasorArray to double precision.
            %   r = DOUBLE(o1) converts the PhasorArray object o1 to a
double precision array.
            r= double(pvalue(o1));
        end
```

```
        function o2 = repeat(o1,m)
            %REPEAT Repeat the PhasorArray object over its period by a
factor m
            %
            %   o2 = repeat(o1, m) repeats the PhasorArray object o1 over its
            %   period by a factor m. If m is negative, the signal is
reversed
            %   in time, effectively dividing the period by m.
            %
            %   Inputs:
            %       o1 - The PhasorArray object to be repeated.
            %       m  - An integer factor by which to repeat the period.
            %            Default value is 2. If m is negative, the signal is
            %            reversed in time.
            %
            %   Outputs:
            %       o2 - The resulting PhasorArray object after repeating the
            %            period by a factor m.
            %
            %   Example:
            %       o1 = PhasorArray(...); % Create a PhasorArray object
            %       o2 = repeat(o1, 3);    % Repeat the period of o1 by a
factor of 3
            %
            %   See also: PhasorArray.zeros, retro
            arguments
                o1
                m {mustBeInteger(m)} = 2
            end
            h=o1.h;
            h_new=h*abs(m);
            o2 = PhasorArray.zeros(size(o1,1),size(o1,2),2*h_new+1);
            o2(:,:,1:abs(m):end)=o1.value;
            if m<0
                o2=retro(o2);
            end

        end


        function r = antiD(o1,T)
            %ANTI-D Compute the anti-derivative (primitive) of non-zero
phasors of a PhasorArray.
            %   r = ANTI-D(o1, T) computes the anti-derivative of the
PhasorArray o1
            %   with respect to the period T. The function ignores the 0-th
phasor to
            %   produce a periodic output.
            %
            %   INPUTS:
            %       o1 - The PhasorArray object to be anti-differentiated.
            %       T  - (Optional) The period of the PhasorArray. Defaults
to 2*pi.
            %
```

```
            %   OUTPUT:
            %       r  - The resulting PhasorArray after computing the anti-
derivative.
            %
            %   EXAMPLE:
            %       A = PhasorArray.random(3, 3, 5);
            %       B = A.antiD(2*pi);
            %
            %   NOTE:
            %       The function divides each non-zero phasor by (1i * k * 2
* pi / T),
            %       where k is the phasor index. The 0-th phasor is set to
zero.
            arguments
            o1
            T=2*pi
            end
            o1=o1.value;
            [~,~,n3]=size(o1);
            for ii=1:n3
            k=-(n3-1)/2+ii-1;
            if k~=0
            o1(:,:,ii)=o1(:,:,ii)/(1i*k*2*pi/T);
            else
                o1(:,:,ii)=0;
            end
            end
            r=PhasorArray(o1);
        end

        function r = d(o1,T)
            % d - Derive phasor array with respect to a given period
            %
            % Syntax: r = d(o1, T)
            %
            % Inputs:
            %    o1 - The input phasor array object
            %    T  - The period with respect to which the phasor array is
derived (default is 2*pi)
            %
            % Outputs:
            %    r  - The resulting phasor array after derivation
            %
            % Example:
            %    phasorArray = PhasorArray(someValues);
            %    result = phasorArray.d(phasorArray, 2*pi);
            arguments
                o1
                T=2*pi
            end
            %[~,~,n3]=size(o1);
            try
                K=permute((-o1.h):(o1.h),[1 3 2])*1i*2*pi/T;
                oo=bsxfun(@times,o1.value,K);
```

```
                r=PhasorArray(oo);
            catch e
                e
                % We need to manually perform the operation intended for
each "slice" of P.
                for idx = 1:size(o1, 3)
                    o1(:,:,idx) = o1(:,:,idx) * K(idx);
                end
                r = o1;
            end
        end
        function r = PhaseShift(o1,angle)
            % PhaseShift Apply a phase shift to a PhasorArray.
            %
            %   r = PhaseShift(o1, angle) shifts the phase of the
PhasorArray `o1`
            %   by `angle` radians.
            %
            %   Inputs:
            %       o1    - The PhasorArray object.
            %       angle - The phase shift value(s). It can be:
            %               - A scalar (applies to all elements of `o1`).
            %               - A row vector (broadcasts across `o1` columns
if `o1` is a column).
            %               - A column vector (broadcasts across `o1` rows
if `o1` is a row).
            %               - A matrix (applies element-wise if `o1` is
scalar).
            %
            %   Output:
            %       r - The phase-shifted PhasorArray.
            %
            %   Behavior:
            %   - If `o1` is a scalar, `angle` can be a **matrix**, applying
element-wise shifts.
            %   - If `o1` is a **row vector**, `angle` must be a column
vector (broadcasted).
            %   - If `o1` is a **column vector**, `angle` must be a row
vector (broadcasted).
            %   - If `o1` is a **matrix**, `angle` must be scalar.
            %
            %   Errors:
            %   - If `o1` and `angle` are both row or column vectors, the
function raises an error.
            %   - If `o1` is a matrix, `angle` must be a scalar.
            %
            %   See also: dephase, PhasorArray
            if numel(angle)>1
                if isrow(o1) && iscolumn(angle)
                    n1=numel(angle);
                    %n2=o1.size(2);
                    r=repmat(o1,n1,1);
                    for angli=1:numel(angle)
                        angle(angli);
```

```
                        r{angli,:}=o1.PhaseShift(angle(angli));
                    end
                    return
                elseif iscolumn(o1) && isrow(angle)
                    %n1=o1.size(1);
                    n2=numel(angle);
                    r=repmat(o1,1,n2);
                    for angli=1:numel(angle)
                        r{:,angli}=o1.PhaseShift(angle(angli));
                    end
                    return
                elseif isscalar(o1) && ismatrix(angle)
                    [n1,n2]=size(angle,[1,2]);

                    r=repmat(o1,n1,n2);
                    for angli=1:size(angle,1)
                        for anglj=1:size(angle,2)
                            r{angli,anglj}=o1.PhaseShift(angle(angli,anglj));
                        end
                    end
                    return
                elseif iscolumn(o1) && iscolumn(angle)
                    error('if o1 is a column, angle must be row and vice
versa')
                elseif isrow(o1) && isrow(angle)
                    error('if o1 is a row, angle must be colmun and vice
versa')
                else
                    error('angle must be a scalar input for matricial
PhasorArray')
                end
            else
                r=dephase(o1,angle);
            end
        end

        function [oInv, oInvt, norm_err, norm_ref] = inv(o1, varargin)
            %INV Compute the phasor representation of the pointwise inverse
of A(t).
            %
            %   This function computes the **pointwise inverse** of the time-
domain realization
            %   of the periodic matrix A(t) and then reconstructs its phasor
representation.
            %
            %   The inversion process follows these steps:
            %   1. Compute the **time-domain representation** A(t) using an
**IFFT**.
            %   2. Perform **pointwise inversion** A⁻¹(t).
            %   3. Reconstruct phasors by applying an **FFT** to A⁻¹(t).
            %
            %   Syntax:
            %   r = INV(o1)
            %       Computes the phasors of the pointwise inverse of `o1`
```

```
using default parameters.
            %
            %   [r, At] = INV(o1)
            %       Returns the time-domain realization A⁻¹(t) alongside the
phasors.
            %
            %   [r, At, norm_err, norm_ref] = INV(o1)
            %       Also returns reconstruction error metrics.
            %
            %   Input Arguments:
            %   - o1 (PhasorArray) : The PhasorArray object representing
A(t).
            %
            %   Output Arguments:
            %   - oInv (PhasorArray) : The phasor representation of A⁻¹(t).
            %   - oInvt (array, optional) : The time-domain realization of A⁻¹
(t).
            %   - norm_err (double, optional) : Reconstruction error ||
Ainv_ph - Ainv_t||_F.
            %   - norm_ref (double, optional) : Reference norm ||Ainv_t||_F.
            %
            %   Example:
            %   % Compute the phasors of the inverse of a given PhasorArray A
            %   r = inv(A);
            %
            %   % Compute both the phasors and time-domain representation
            %   [r, At] = inv(A);
            %
            %   % Compute full outputs including reconstruction errors
            %   [r, At, err, ref] = inv(A);
            %
            %   See also: DET, REDUCE.

            % Call PhasorInv with only the required number of outputs
            if nargout == 1
                Ainvph = PhasorInv(o1, varargin{:});
                oInv = PhasorArray(Ainvph);
            elseif nargout == 2
                [Ainvph, oInvt] = PhasorInv(o1, varargin{:});
                oInv = PhasorArray(Ainvph);
            else
                [Ainvph, oInvt, norm_err, norm_ref] = PhasorInv(o1,
varargin{:});
                oInv = PhasorArray(Ainvph);
            end
        end
        function [PhDet,det_t] = det(o1,varg)
            %DET Compute the pointwise determinant of A(t) and reconstruct
its phasors.
            %
            %   This function computes the determinant of the **time-domain
realization** A(t),
            %   then reconstructs its phasor representation via Fourier
Transform.
```

```
            %
            %   The determinant is computed **pointwise at each time step**,
meaning:
            %   - If A(t) is periodic, det(A(t)) is computed over time.
            %   - The final result represents the phasor decomposition of
det(A(t)).
            %
            %   Syntax:
            %   [PhDet, det_t] = DET(o1)
            %       Computes the phasors of det(A(t)) using default settings.
            %
            %   [PhDet, det_t] = DET(o1, 'nT', nT, 'T', T, 'm', m,
            %                            'plot', plotFlag, 'autoTrunc',
autoTrunc)
            %       Computes det(A(t)) with additional control over
truncation and plotting.
            %
            %   Input Arguments:
            %   - o1 (PhasorArray) : The PhasorArray object whose
determinant is computed.
            %
            %   Name-Value Pair Arguments:
            %   - 'nT' (integer, optional) : Number of periods used in the
time-domain evaluation. Default: 1.
            %   - 'T' (double, optional) : The period used for simulation.
Default: 1.
            %   - 'm' (integer, optional) :
            %       - Power of two controlling time-domain discretization.
            %       - Can be set to [] for automatic selection based on the
number of phasors.
            %   - 'plot' (logical, optional) : If true, plots det(A(t))
after computation. Default: false.
            %   - 'autoTrunc' (logical, optional) :
            %       - true : Uses the derivative of phasors to
**automatically detect**
            %         the significant number of phasors.
            %       - false (default) : Uses a fixed threshold-based
reduction method.
            %
            %   - If 'autoTrunc' is false, the following options apply:
            %       - 'reduceThreshold' (double, optional) : The threshold
for reducing phasors. Default: 1e-20.
            %       - 'reduceMethod' (char, optional) : Reduction strategy.
            %           - 'absolute' : Remove phasors with magnitude <
reduceThreshold.
            %           - 'relative' (default) : Remove phasors with
magnitude < max(magnitude) * reduceThreshold.
            %
            %   Output Arguments:
            %   - PhDet (PhasorArray) : The phasor representation of
det(A(t)).
            %   - det_t (array) : The time-domain realization of det(A(t)).
            %
            %   Algorithm:
```

```
            %   1. Compute A(t) in the time domain using an **IFFT**.
            %   2. Perform **pointwise determinant computation** det(A(t)).
            %   3. Reconstruct phasors by applying an **FFT** to det(A(t)).
            %
            %   Example:
            %   % Compute the phasors of det(A(t)) using default settings
            %   [PhDet, det_t] = det(A);
            %
            %   % Compute det(A(t)) over 2 periods with auto truncation
            %   [PhDet, det_t] = det(A, 'nT', 2, 'autoTrunc', true);
            %
            %   % Compute det(A(t)) with manual truncation and thresholding
            %   [PhDet, det_t] = det(A, 'reduceThreshold', 1e-15,
'reduceMethod', 'absolute');
            %
            %   See also: INV, REDUCE, FFT, IFFT.
            arguments
            o1
            varg.nT=1
            varg.T=1
            varg.m=[]
            varg.plot=false
            varg.reduceThreshold = 1e-20
            varg.reduceMethod = 'relative'
            varg.autoTrunc = false
            end


            C=namedargs2cell(varg);
            [PhDet,det_t] =  PhasorDet(o1,C{:});
            PhDet=PhasorArray(PhDet);

        end

        function r = diag(o1,K)
            %DIAG Extract or construct a diagonal PhasorArray.
            %
            %   This function operates similarly to MATLAB's `diag`:
            %   - If `o1` is a **vector**, it constructs a **diagonal
PhasorArray**.
            %   - If `o1` is a **square matrix**, it extracts its diagonal
as a **vector**.
            %
            %   Syntax:
            %   r = DIAG(A)
            %       Constructs a diagonal PhasorArray from a vector `A`.
            %
            %   r = DIAG(A, K)
            %       Constructs a diagonal PhasorArray, with `K` as the
diagonal offset.
            %
            %   r = DIAG(A)
            %       Extracts the diagonal as a PhasorArray vector when `A`
is square.
```

```
            %
            %   Input Arguments:
            %   - o1 (PhasorArray) : Input PhasorArray, either a vector or a
square matrix.
            %   - K (integer, optional) : Offset for the diagonal placement.
Default: 0.
            %
            %   Output:
            %   - r (PhasorArray) :
            %       - If `o1` is a vector, `r` is a **diagonal PhasorArray**.
            %       - If `o1` is square, `r` is the **extracted diagonal**.
            %
            %   Example:
            %   % Create a diagonal PhasorArray from a vector
            %   A = PhasorArray(rand(5,1));
            %   R = diag(A);
            %
            %   % Extract the diagonal from a square PhasorArray
            %   B = PhasorArray(rand(4,4));
            %   d = diag(B);
            %
            %   % Construct a diagonal PhasorArray with an offset
            %   C = diag(A, 1);
            %
            %   See also: TRACE.
            arguments
                o1
                K=0
            end
            if isscalar(o1) && K==0
                r = o1;
                return
            end

            if isvector(o1)
                if ~iscolumn(o1)
                    o1=o1.';
                end
                if isa(o1.value,"sdpvar")
                    r = PhasorArray(diag(o1));
                    return
                end

                if isa(o1.value,"ndsdpvar") || isa(o1.value,"sym")
                    for ii = 1:size(o1,3)
                        r{ii}  = PhasorArray(diag(o1(:,:,ii)));
                    end
                    r=cat(3,r{:});
                    return
                end

                r = PhasorArray.zeros(numelt(o1)+abs(K));
```

```
                I = logical(diag(ones(numelt(o1),1),K));
                r{I}=r{I}+o1;
                return
            end
            assert(issquare(o1),"diag function : PhasorArray must represent
a square matrix");
            n=size(o1,1);
            I=(1:n)*(n+1)-n;
            r=o1{I};
        end
        function r = trace(o1,type)
            %TRACE Compute the phasor representation of the trace of A(t).
            %
            %   This function computes the **trace** of a PhasorArray A,
which is the sum
            %   of its diagonal elements. It supports two modes:
            %   - 'phasor' (default) : Returns the **full** phasor
representation of the trace.
            %   - '0'                : Returns only the **DC (0th order)**
component of the trace.
            %
            %   Syntax:
            %   r = TRACE(A)
            %       Computes the phasor representation of the trace of `A`.
            %
            %   r = TRACE(A, 'phasor')
            %       Same as above; returns the full phasor trace.
            %
            %   r = TRACE(A, '0')
            %       Returns only the DC component of the trace.
            %
            %   Input Arguments:
            %   - o1 (PhasorArray) : A **square PhasorArray**.
            %   - type (char, optional) :
            %       - 'phasor' (default) : Returns the **full phasor
representation** of trace(A).
            %       - '0' : Returns only the **DC component** (0th phasor)
of trace(A).
            %
            %   Output:
            %   - r (PhasorArray or numeric) :
            %       - If 'phasor', `r` is a **PhasorArray** containing the
full phasor trace.
            %       - If '0', `r` is a **scalar numeric value** (the DC
component of trace(A)).
            %
            %   Example:
            %   % Compute the full phasor trace of a PhasorArray
            %   A = PhasorArray(rand(3,3,11));
            %   tr_phasor = trace(A);
            %
            %   % Compute only the DC component of the trace
            %   tr_DC = trace(A, '0');
            %
```

```
            %   See also: DIAG.
            arguments
                o1
                type char {mustBeMember(type,{'phasor','0'})} = 'phasor'
            end
            r=sum(diag(o1));
            if strcmp(type,'0')
                %return the DC value of the trace
                r=r{:,:,0};
            end
        end

        function r = numelt(o1)
            %NUMELT Compute the number of elements in the first two
dimensions of A.
            %
            %   This function returns `size(A,1) * size(A,2)`, which
represents
            %   the number of matrix elements per time instant.
            %
            %   Syntax:
            %   r = NUMELT(A)
            %       Computes the total number of elements in A(t).
            %
            %   Input:
            %   - o1 (PhasorArray) : A PhasorArray object.
            %
            %   Output:
            %   - r (integer) : Number of elements in `A(t)`.
            %
            %   Example:
            %   % Compute the number of elements in a 4x5 PhasorArray
            %   A = PhasorArray(rand(4,5,10));
            %   e = numelt(A);  % Returns 4*5 = 20.
            r=size(o1,1)*size(o1,2);
        end
        function r = h(o1)
            %H Compute the maximal phasor order stored in A.
            %
            %   This function extracts the **highest phasor order** present
in a PhasorArray.
            %
            %   Syntax:
            %   r = H(A)
            %       Computes the highest stored harmonic order.
            %
            %   Input:
            %   - o1 (PhasorArray) : A PhasorArray object.
            %
            %   Output:
            %   - r (integer) : The highest stored harmonic order in A.
            %
            %   Example:
            %   % Get the maximum harmonic order in a PhasorArray
```

```
            %   A = PhasorArray(rand(4,4,11));
            %   max_h = h(A);  % Returns (11-1)/2 = 5.
            %
            %   See also: DIM.
            r=(size(o1,3)-1)/2;
        end
        function r = dim(o1)
            %DIM Compute the number of elements in A(t).
            %
            %   This function returns `size(A,1) * size(A,2)`, which
represents
            %   the number of matrix elements per time instant.
            %
            %   Syntax:
            %   r = DIM(A)
            %       Computes the total number of elements in A(t).
            %
            %   Input:
            %   - o1 (PhasorArray) : A PhasorArray object.
            %
            %   Output:
            %   - r (integer) : Number of elements in `A(t)`.
            %
            %   Example:
            %   % Compute the number of elements in a 4x5 PhasorArray
            %   A = PhasorArray(rand(4,5,10));
            %   e = dim(A);  % Returns 4*5 = 20.
            %
            %   See also: NUMELT.
            r=size(o1,1)*size(o1,2);
        end


        function r = BT(o1,m)
            %BT Construct a Block Toeplitz matrix from a PhasorArray.
            %
            %   This function builds a **Block Toeplitz matrix** of order
`m` from the given
            %   PhasorArray `A`. A **Block Toeplitz matrix** is a **block-
structured matrix**
            %   where **entire blocks** (submatrices) are repeated along the
diagonals.
            %
            %   Specifically, the matrix `BT(A, m)` is given by:
            %       B = sum_{k=-2m}^{2m} ( I^k ⊗ A_k ),
            %   where:
            %   - `I^k` is the **diagonal selector matrix**, which has ones
on its `k`-th diagonal.
            %   - `⊗` denotes the **Kronecker product**.
            %   - `A_k` are the Fourier components (harmonics) of `A`.
            %
            %   This matrix represents the **finite-dimensional truncation**
of an
            %   **infinite-dimensional Block Toeplitz operator** acting on
```

```
ℓ², where the truncation
            %   is performed at order `m`.
            %
            %   **Resulting Dimensions:**
            %   If `A` is an `N×N` PhasorArray, then `BT(A, m)` is a
`(2m+1)N × (2m+1)N` matrix.
            %
            %   Syntax:
            %   r = BT(A, m)
            %       Constructs a Block Toeplitz matrix of order `m`.
            %
            %   Input Arguments:
            %   - o1 (PhasorArray) : The input PhasorArray.
            %   - m (integer, optional) : The order of the Block Toeplitz
matrix.
            %
            %   Output:
            %   - r ((2m+1)N × (2m+1)N matrix) : The Block Toeplitz matrix.
            %
            %   Example:
            %   % Generate a Block Toeplitz matrix from a PhasorArray
            %   A = PhasorArray(rand(4,4,11));
            %   BT_A = BT(A, 5);
            %
            %   See also: TB, spTB.
            arguments
                o1
                m=[]
            end
            r=array2BToepliz(o1,2*m);
        end
        function r = TB(o1,m)
            %TB Construct a Toeplitz Block matrix from a PhasorArray.
            %
            %   This function constructs a **Toeplitz Block (TB) matrix** of
order `m`, which is an alternative
            %   representation of `A` where **each block** of the matrix is
a Toeplitz structure.
            %
            %   Specifically, `TB(A, m)` is given by:
            %       B = sum_{k=-2m}^{2m} ( A_k ⊗ I^k ),
            %   where:
            %   - `I^k` is the **diagonal selector matrix**, which has ones
on its `k`-th diagonal.
            %   - `⊗` denotes the **Kronecker product**.
            %   - `A_k` are the Fourier components (harmonics) of `A`.
            %
            %   Unlike **Block Toeplitz (BT)**, where **blocks are repeated
along diagonals**, here **each block itself**
            %   follows a **Toeplitz structure**. Both `BT(A, m)` and `TB(A,
m)` are equivalent in the **harmonic space**,
            %   up to a basis transformation.
            %
            %   This matrix represents the **finite-dimensional truncation**
```

```
of an
            %   **infinite-dimensional Toeplitz Block operator** acting on
ℓ², with truncation at order `m`.
            %
            %   **Resulting Dimensions:**
            %   If `A` is an `N×N` PhasorArray, then `TB(A, m)` is a
`(2m+1)N × (2m+1)N` matrix.
            %
            %   Syntax:
            %   r = TB(A, m)
            %       Constructs a Toeplitz Block matrix of order `m`.
            %
            %   Input Arguments:
            %   - o1 (PhasorArray) : The input PhasorArray.
            %   - m (integer, optional) : The order of the Toeplitz Block
matrix.
            %
            %   Output:
            %   - r ((2m+1)N × (2m+1)N matrix) : The Toeplitz Block matrix.
            %
            %   Example:
            %   % Generate a Toeplitz Block matrix from a PhasorArray
            %   A = PhasorArray(rand(4,4,11));
            %   TB_A = TB(A, 5);
            %
            %   See also: BT, spTB.
            arguments
                o1
                m=[]
            end
            r=array2TBlocks(o1,2*m);
        end
        function r = spTB(o1,m)
            %SPTB Construct a sparse Toeplitz Block representation of A.
            %
            %   This function generates a **sparse** Toeplitz Block
representation
            %   of order `m` for faster computations. The structure follows:
            %
            %       B = sum_{k=-2m}^{2m} ( A_k ⊗ I^k ),
            %
            %   where:
            %   - `I^k` is the **diagonal selector matrix**, which has ones
on its `k`-th diagonal.
            %   - `⊗` denotes the **Kronecker product**.
            %   - `A_k` are the Fourier components (harmonics) of `A`.
            %
            %   This represents the **truncation of an infinite-dimensional
Toeplitz Block operator on ℓ²**.
            %
            %   **Resulting Dimensions:**
            %   If `A` is an `N×N` PhasorArray, then `spTB(A, m)` is a
`(2m+1)N × (2m+1)N` **sparse matrix**.
            %
```

```
            %   Syntax:
            %   r = SPTB(A, m)
            %       Computes a sparse Toeplitz Block representation of order
`m`.
            %
            %   Input Arguments:
            %   - o1 (PhasorArray) : The input PhasorArray.
            %   - m (integer, optional) : The order of the Toeplitz Block.
            %
            %   Output:
            %   - r ((2m+1)N × (2m+1)N sparse matrix) : The sparse Toeplitz
Block matrix.
            %
            %   Example:
            %   % Generate a sparse Toeplitz Block matrix
            %   A = PhasorArray(rand(4,4,11));
            %   spTB_A = spTB(A, 5);
            %
            %   See also: TB, BT.
            arguments
                o1
                m=[]
            end
            r=sparray2TBlocks(o1,2*m);
        end
        function r = spBT(o1,m)
            %WIP%%%%%%%%%%
            arguments
                o1
                m=[]
            end
            warning("NOT IMPLEMENTED YET")
            %             r=sparray2BToeplitz(o1,2*m);
            r=sparse(array2BToepliz(o1,2*m));
        end
        function r = FvTB(o1,h)
            %FVTB Compute the Fourier representation of the vectorized form
of A(t).
            %
            %   This function first **vectorizes** the time-dependent matrix
A(t) by stacking
            %   all its columns into a **single-column vector** a(t) =
vect(A(t)), and then
            %   applies the **TF_TB** transformation to obtain its Fourier
representation.
            %
            %   **Alternative Computation:**
            %   - In MATLAB notation, vectorizing A(t) is equivalent to
**A{:}**.
            %   - Thus, this function is **equivalent to applying TF_TB to
A{:}**, i.e.:
            %       FvTB(A, h) = TF_TB(A{:}, h).
            %
            %   **Procedure:**
```

```
            %   1. Compute the column-wise vectorization:
            %       - If A(t) is an `N×M` matrix, then vect(A(t)) is a
column vector of size `NM × 1`.
            %       - This stacking is done **column by column** (following
MATLAB's column-major order).
            %   2. Compute the Fourier series coefficients **up to order h**:
            %       - Instead of applying `TF_TB` directly to A(t), we apply
it to a(t) = vect(A(t)).
            %       - The result is the Fourier representation of
`vect(A(t))`, which is a vectorized form
            %         of `TF_TB(A, h)`.
            %
            %   **Key Property:**
            %   If `y(t) = A(t)x(t)`, then:
            %       FvTB(y) = FvTB(A) · FvTB(x),
            %   where `FvTB(A, h)` is the **Fourier vectorized
representation** of A.
            %
            %   **Dimension of the Output:**
            %   - If `A` is an `N×M` matrix, then `FvTB(A, h)` is a
`((2h+1)NM × 1)` vector.
            %
            %   Syntax:
            %   r = FVTB(A, h)
            %       Computes the Fourier representation of `vect(A)` up to
order `h`.
            %
            %   Input Arguments:
            %   - o1 (PhasorArray) : The input PhasorArray representing A(t).
            %   - h (integer) : The highest harmonic order to retain in the
Fourier series.
            %
            %   Output:
            %   - r ((2h+1)NM × 1 vector) : The Fourier representation of
vect(A).
            %
            %   Example:
            %   % Compute Fourier-vectorized representation of A(t)
            %   A = PhasorArray(rand(4,4,11));
            %   FvTB_A = FvTB(A, 5);
            %
            %   % Compute using TF_TB on the vectorized form
            %   FvTB_A_alt = TF_TB(A{:}, 5); % Equivalent computation
            %
            %   See also: TF_TB, vect, trunc, pad.
            if size(o1,3)>2*h+1
                o1=trunc(o1,h);
            else
                o1=pad(o1,h-o1.h);
            end
            d1=reshape(o1,[],1,size(o1,3));

            r=pvalue(permute(d1,[3 1 2]));
            r=r(:);
```

```
        end

        function [HpJ,JHm,Hp,Hm] = TBHankel(o1,m)
            %TBHANKEL Compute Toeplitz Block Hankel matrices of order m.
            %
            %   This function computes the **Toeplitz Block Hankel
matrices** (H) and the
            %   associated **J-Hankel matrices** (H_J) of the given matrix
`A(t)` represented
            %   by the PhasorArray `o1`.
            %
            %   **Definition:**
            %   - A **Hankel matrix** is symmetric with entries reflected
across its antidiagonals.
            %   - A **J-Hankel matrix** is defined by pre- and post-
multiplying the Hankel matrix
            %     with the flipping operator `J_m`, where `J_m` flips the
diagonals of a matrix.
            %
            %   **Key Property:**
            %   - For a periodic matrix function A(t), the Toeplitz Block
Hankel decomposition
            %     complements the Toeplitz Block transform (TB).
            %
            %   **Truncation:**
            %   - The matrix is truncated to order `2m` harmonics before
performing the computation.
            %
            %   **Output Details:**
            %   - `HpJ`: Positive J-Hankel matrix for `A(t)`.
            %   - `JHm`: Negative J-Hankel matrix for `A(t)`.
            %   - `Hp`: Positive Hankel block matrix for `A(t)`.
            %   - `Hm`: Negative Hankel block matrix for `A(t)`.
            %
            %   **Dimensions:**
            %   - If `A` is an `N×N` matrix, then all outputs have
dimensions `(2m+1)N × (2m+1)N`.
            %
            %   Syntax:
            %   [HpJ, JHm, Hp, Hm] = TBHANKEL(A, m)
            %       Computes the Toeplitz Block Hankel and J-Hankel matrices
of order `m`.
            %
            %   Input Arguments:
            %   - o1 (PhasorArray) : The input PhasorArray representing A(t).
            %   - m (integer) : The truncation order for the harmonics.
            %
            %   Output:
            %   - HpJ ((2m+1)N × (2m+1)N matrix) : Positive J-Hankel matrix.
            %   - JHm ((2m+1)N × (2m+1)N matrix) : Negative J-Hankel matrix.
            %   - Hp ((2m+1)N × (2m+1)N matrix) : Positive Hankel block
matrix.
            %   - Hm ((2m+1)N × (2m+1)N matrix) : Negative Hankel block
matrix.
```

```
            %
            %   Example:
            %   % Compute Hankel matrices for a given A(t)
            %   A = PhasorArray(rand(4,4,11));
            %   [HpJ, JHm, Hp, Hm] = TBHankel(A, 5);
            %
            %   See also: spTBHANKEL, BTHANKEL.
            [HpJ,JHm,Hp,Hm] = Array2TBHankel(o1,2*m);
        end
        function [HpJ,JHm,Hp,Hm] = spTBHankel(o1,m)
            %SPTBHANKEL Compute sparse Toeplitz Block Hankel matrices of
order m.
            %
            %   This function computes the **sparse Toeplitz Block Hankel
matrices** (H) and
            %   their associated **J-Hankel matrices** (H_J) for the given
PhasorArray `o1`.
            %
            %   **Sparse Computation:**
            %   - The matrices are stored and computed in **sparse format**
for efficiency,
            %     especially for high-dimensional problems or large
truncation orders.
            %
            %   **Output Details:** (same as `TBHANKEL`)
            %   - `HpJ`: Positive sparse J-Hankel matrix.
            %   - `JHm`: Negative sparse J-Hankel matrix.
            %   - `Hp`: Positive sparse Hankel block matrix.
            %   - `Hm`: Negative sparse Hankel block matrix.
            %
            %   **Syntax:**
            %   [HpJ, JHm, Hp, Hm] = SPTBHANKEL(A, m)
            %       Computes the sparse Toeplitz Block Hankel and J-Hankel
matrices of order `m`.
            %
            %   See also: TBHANKEL, BTHANKEL.
            [HpJ,JHm,Hp,Hm] = spArray2TBHankel(o1,2*m);
        end
        function [HpJ,JHm,Hp,Hm] = BTHankel(o1,m)
            %BTHANKEL Compute Block Toeplitz Hankel matrices of order m.
            %
            %   This function computes the **Block Toeplitz Hankel
matrices** (H) and the
            %   associated **J-Hankel matrices** (H_J) for the given
PhasorArray `o1`.
            %
            %   **Difference with TBHankel:**
            %   - While `TBHankel` corresponds to **Toeplitz Block (TB)**
structures, `BTHankel`
            %     corresponds to **Block Toeplitz (BT)** structures.
            %
            %   **Output Details:** (same as `TBHANKEL`)
            %   - `HpJ`: Positive J-Hankel matrix.
            %   - `JHm`: Negative J-Hankel matrix.
```

```
            %   - `Hp`: Positive Hankel block matrix.
            %   - `Hm`: Negative Hankel block matrix.
            %
            %   **Syntax:**
            %   [HpJ, JHm, Hp, Hm] = BTHANKEL(A, m)
            %       Computes the Block Toeplitz Hankel and J-Hankel matrices
of order `m`.
            %
            %   See also: SPBTHANKEL, TBHANKEL.
            [HpJ,JHm,Hp,Hm] = Array2BTHankel(o1,2*m);
        end
        function [HpJ,JHm,Hp,Hm] = spBTHankel(o1,m)
            %SPBTHANKEL Compute sparse Block Toeplitz Hankel matrices of
order m.
            %
            %   This function computes the **sparse Block Toeplitz Hankel
matrices** (H) and
            %   the associated **J-Hankel matrices** (H_J) for the given
PhasorArray `o1`.
            %
            %   **Sparse Computation:** (same as `spTBHANKEL`)
            %   - The matrices are stored and computed in **sparse format**
for efficiency.
            %
            %   **Syntax:**
            %   [HpJ, JHm, Hp, Hm] = SPBTHANKEL(A, m)
            %       Computes the sparse Block Toeplitz Hankel and J-Hankel
matrices of order `m`.
            %
            %   See also: BTHANKEL, TBHANKEL.
            [HpJ,JHm,Hp,Hm] = spArray2BTHankel(o1,2*m);
        end

        function AB_TB = TBmtimes(o1,o2,h)
            % TBMTIMES Compute the Toeplitz Block (TB) matrix of the product
A(t)B(t).
            %
            %   TBMTIMES(o1, o2, h) computes the Toeplitz Block matrix of
order `h`
            %   that represents the product of two periodic matrix functions
A(t) and B(t).
            %
            %   Inputs:
            %     o1  - (PhasorArray) The first input PhasorArray
representing A(t).
            %     o2  - (PhasorArray) The second input PhasorArray
representing B(t).
            %     h   - (integer) The truncation order for the harmonics.
            %
            %   Outputs:
            %     AB_TB - ((2h+1)N × (2h+1)N matrix) The Toeplitz Block
matrix for A(t)B(t).
            %
            %   Behavior:
```

```
            %     - Uses the Hankel matrices and TB representation of A(t)
and B(t).
            %     - The resulting TB matrix satisfies:
            %         T_tb(A * B) = T_tb(A) * T_tb(B) + H_p(A) * H_m(B) +
JH_m(A) * JH_p(B).
            %
            %   Output Dimensions:
            %     - If A and B are `N×N` matrices and the truncation order
is `h`, then:
            %         - `AB_TB` is a `(2h+1)N × (2h+1)N` matrix.
            %
            %   Example Usage:
            %     % Compute TB matrix of the product A(t)B(t)
            %     A = PhasorArray(rand(4,4,11));
            %     B = PhasorArray(rand(4,4,11));
            %     AB_TB = TBmtimes(A, B, 5);
            %
            %   See also: spTBmtimes, TB, TBHankel.
            [~,JHm,Hp,~] = TBHankel(o1,h);
            [HpJ2,~,~,Hm2] = TBHankel(o2,h);
            o1TB=o1.TB(h);
            o2TB=o2.TB(h);
            AB_TB=o1TB*o2TB+Hp*Hm2+JHm*HpJ2;
        end
        function AB_TB = spTBmtimes(o1,o2,h)
            % SPTBMTIMES Compute the sparse Toeplitz Block (TB) matrix of
the product A(t)B(t).
            %
            %   SPTBMTIMES(o1, o2, h) computes the sparse Toeplitz Block
matrix of order `h`
            %   that represents the product of two periodic matrix functions
A(t) and B(t).
            %
            %   Inputs:
            %     o1  - (PhasorArray) The first input PhasorArray
representing A(t).
            %     o2  - (PhasorArray) The second input PhasorArray
representing B(t).
            %     h   - (integer) The truncation order for the harmonics.
            %
            %   Outputs:
            %     AB_TB - ((2h+1)N × (2h+1)N sparse matrix) Sparse TB matrix
for A(t)B(t).
            %
            %   Behavior:
            %     - Uses sparse versions of Hankel matrices and TB
representation of A(t) and B(t).
            %     - The result satisfies:
            %         T_tb(A * B) = T_tb(A) * T_tb(B) + H_p(A) * H_m(B) +
JH_m(A) * JH_p(B).
            %
            %   Output Dimensions:
            %     - If A and B are `N×N` matrices and the truncation order
is `h`, then:
```

```
            %         - `AB_TB` is a `(2h+1)N × (2h+1)N` sparse matrix.
            %
            %   Example Usage:
            %     % Compute sparse TB matrix of the product A(t)B(t)
            %     A = PhasorArray(rand(4,4,11));
            %     B = PhasorArray(rand(4,4,11));
            %     AB_TB = spTBmtimes(A, B, 5);
            %
            %   See also: TBmtimes, spTB, spTBHankel.
            [~,JHm,Hp,~] = spTBHankel(o1,h);
            [HpJ2,~,~,Hm2] = spTBHankel(o2,h);
            o1TB=o1.spTB(h);
            o2TB=o2.spTB(h);
            AB_TB=o1TB*o2TB+Hp*Hm2+JHm*HpJ2;
        end

        function r= TF_TB(o1,m)
            %TF_TB Compute the Fourier representation of A in a form
compatible with TB(A, m).
            %
            %   This function computes the **Fourier representation** of
A(t) up to order `m`,
            %   but instead of forming a **square Toeplitz Block matrix**
(TB(A, m)), it **stacks
            %   the phasors of each element of A into a structured
vectorized form**.
            %
            %   **Definition:**
            %   - Let `a(t)` be a scalar function with Fourier coefficients
`(a_k)_{k∈ℤ}`.
            %   - Then, `TF_TB(a, m) = [a_(-m); a_(-m+1); ... a_0; a_1; ...
a_m]`.
            %   - If `A(t)` is an `N×M` matrix, its Fourier representation
is given by:
            %       TF_TB(A, m) = [TF_TB(A_11), TF_TB(A_12); TF_TB(A_21),
TF_TB(A_22)].
            %
            %   **Key Property (TB Compatibility):**
            %   If `y(t) = A(t)x(t)`, then:
            %       F_TB(y) = TB(A, m) · F_TB(x),
            %   where `TB(A, m)` is the **Toeplitz Block representation** of
`A` (see `TB` method).
            %
            %   **Dimension of the Output:**
            %   - If `A` is an `N×M` matrix, then `TF_TB(A, m)` is a
`((2m+1)N) × M` matrix.
            %
            %   Syntax:
            %   r = TF_TB(A, m)
            %       Computes the Fourier representation of `A` up to order
`m`, compatible with TB(A, m).
            %
            %   Input Arguments:
            %   - o1 (PhasorArray) : The input PhasorArray representing A(t).
```

```
            %   - m (integer, optional) : The highest harmonic order to
retain. Default: `A.h`.
            %
            %   Output:
            %   - r ((2m+1)N × M matrix) : The Fourier representation of A
in a form compatible with TB(A, m).
            %
            %   Example:
            %   % Compute Fourier representation of A in TB form
            %   A = PhasorArray(rand(4,4,11));
            %   TF_TB_A = TF_TB(A, 5);
            %
            %   See also: TF_BT, TB.
            arguments
                o1
                m=(size(o1,3)-1)/2;
            end
            nho=(size(o1,3)-1)/2;
            if nho<m
                toto=value(o1.pad(m-nho));
            elseif nho>m
                toto=pvalue(trunc(o1,m));
            else
                toto=pvalue(o1);
            end
            titi=permute(toto,[3 1 2]);
            r=reshape(titi,[],size(o1,2),1);
        end
        function r= TF_BT(o1,m)
            %TF_BT Compute the Fourier representation of A in a form
compatible with BT(A, m).
            %
            %   This function computes the **Fourier representation** of
A(t) up to order `m`,
            %   but instead of forming a **square Block Toeplitz matrix**
(BT(A, m)), it **stacks
            %   the phasors of each element of A into a structured
vectorized form**.
            %
            %   **Definition:**
            %   - Let `a(t)` be a scalar function with Fourier coefficients
`(a_k)_{k∈ℤ}`.
            %   - Then, `TF_BT(a, m) = [a_(-m); a_(-m+1); ... a_0; a_1; ...
a_m]`.
            %   - If `A(t)` is an `N×M` matrix, its Fourier representation
is given by:
            %       TF_BT(A, m) = [TF_BT(A_11), TF_BT(A_12); TF_BT(A_21),
TF_BT(A_22)].
            %
            %   **Key Property (BT Compatibility):**
            %   If `y(t) = A(t)x(t)`, then:
            %       F_BT(y) = BT(A, m) · F_BT(x),
            %   where `BT(A, m)` is the **Block Toeplitz representation** of
`A` (see `BT` method).
```

```
            %
            %   **Dimension of the Output:**
            %   - If `A` is an `N×M` matrix, then `TF_BT(A, m)` is an `N ×
(2m+1)M` matrix.
            %
            %   Syntax:
            %   r = TF_BT(A, m)
            %       Computes the Fourier representation of `A` up to order
`m`, compatible with BT(A, m).
            %
            %   Input Arguments:
            %   - o1 (PhasorArray) : The input PhasorArray representing A(t).
            %   - m (integer, optional) : The highest harmonic order to
retain. Default: `A.h`.
            %
            %   Output:
            %   - r (N × (2m+1)M matrix) : The Fourier representation of A
in a form compatible with BT(A, m).
            %
            %   Example:
            %   % Compute Fourier representation of A in BT form
            %   A = PhasorArray(rand(4,4,11));
            %   TF_BT_A = TF_BT(A, 5);
            %
            %   See also: TF_TB, BT.
            arguments
                o1
                m=(size(o1,3)-1)/2;
            end
            nho=(size(o1,3)-1)/2;
            if nho<m
                toto=padarray(pvalue(o1),[0 0 m-nho]);
            elseif nho>m
                toto=pvalue(trunc(o1,m));
            else
                toto=pvalue(o1);
            end
            titi=permute(toto,[1 3 2]);
            r=reshape(titi,[],size(o1,2),1);
        end

        function r= HmqNEig(o1,h,T,bandlimit)
            % HMQNEIG Compute the eigenvalues of TB(o1, h) - NTB(o1, h, T).
%
%   HMQNEIG(o1, h, T, bandlimit) computes the eigenvalues of the difference
between
%   the Toeplitz Block matrix `TB(o1, h)` and the non-periodic Toeplitz
approximation
%   `NTB(o1, h, T)`, capturing spectral properties of `o1`.
%
%   Inputs:
%     o1        - (PhasorArray) The input PhasorArray representing a
periodic matrix function.
%     h         - (integer, optional) The truncation order for the harmonics.
```

```
%                   - Default: `2*o1.h` (twice the highest harmonic stored
in `o1`).
%     T         - (double) The period of the PhasorArray.
%     bandlimit - (char) Specifies band-limiting of eigenvalues:
%                   - 'none' (default): No band-limiting.
%                   - 'fundamental': Retains eigenvalues where |imag(r)| <
π / |T|.
%
%   Outputs:
%     r - (vector) The eigenvalues of the matrix `TB(o1, h) - NTB(o1, h, T)`.
%
%   Behavior:
%     - Computes eigenvalues of the difference matrix `Hmq = TB(o1, h) -
NTB(o1, h, T)`.
%     - Band-limiting removes eigenvalues outside the specified range.
%
%   Example Usage:
%     % Compute eigenvalues with fundamental band-limiting
%     A = PhasorArray(rand(4,4,11));
%     r = HmqNEig(A, 10, 2, 'fundamental');
%
%     % Compute eigenvalues without band-limiting
%     r = HmqNEig(A, [], 1, 'none');
%
%   See also: TB, NTB, eig.
            arguments
                o1
                h=20
                T=1
                bandlimit {mustBeMember(bandlimit,
{'none','fundamental'})}='none'
            end
            if isempty(h)
                h=2*o1.h;
            end
            r1=o1.TB(h)-NTB(o1.value,h,T);
            r=eig(r1);
            switch bandlimit
                case 'fundamental'
                    r=r(abs(imag(r))<pi/abs(T));
                case 'none'
                otherwise
                    r=r(abs(imag(r))<=bandlimit);

            end
        end
        function r= HmqEig(o1,h)
            % HMQEIG Compute the eigenvalues of the Toeplitz Block (TB)
matrix of A(t).
            %
            %   HMQEIG(o1, h) computes the eigenvalues of the Toeplitz Block
(TB) matrix
            %   representation of the periodic matrix function `A(t)`,
capturing its spectral properties.
```

```
            %
            %   Inputs:
            %     o1  - (PhasorArray) The input PhasorArray representing a
periodic matrix function.
            %     h   - (integer, optional) The truncation order for the
harmonics.
            %           - Default: `o1.h` (the highest harmonic stored in
`o1`).
            %
            %   Outputs:
            %     r   - (vector) The eigenvalues of the Toeplitz Block
matrix `TB(o1, h)`.
            %
            %   Behavior:
            %     - The matrix is truncated to order `h` to form `TB(o1, h)`.
            %     - If `h` is not specified, the function uses the highest
harmonic available in `o1`.
            %
            %   Output Dimensions:
            %     - If `A(t)` is an `N×N` matrix and the truncation order is
`h`, then:
            %         - `TB(o1, h)` is a `(2h+1)N × (2h+1)N` matrix.
            %         - `r` is a vector of eigenvalues of length `(2h+1)N`.
            %
            %   Example Usage:
            %     % Compute eigenvalues of TB(A, h) with a specific
truncation order
            %     A = PhasorArray(rand(4,4,11));
            %     r = HmqEig(A, 10);
            %
            %     % Compute eigenvalues using the default harmonic order
            %     r = HmqEig(A);
            %
            %   See also: TB, eig.
            arguments
                o1
                h=o1.h
            end
            r1=o1.TB(h);
            r=eig(r1);
        end


        function Tout = stem(o1,varopt)
            % STEM Generate a stem plot for one or more `PhasorArray`
objects.
            %
            %   STEM(o1, varopt) generates a stem plot to visualize the
phasors of one or more
            %   `PhasorArray` objects. Each input object is plotted with a
distinct marker style,
            %   and the function integrates seamlessly with existing
figures, axes, or tiled layouts.
            %
```

```
            %   Inputs:
            %     o1          - (Repeating, `PhasorArray`) One or more
`PhasorArray` objects to be plotted.
            %
            %   Name-Value Pair Arguments:
            %     'scale'     - (char) Y-axis scale. Options:
            %                     - 'log' (default): Logarithmic scale.
            %                     - 'linear': Linear scale.
            %     'explosed'  - (logical) Determines subplot arrangement:
            %                     - true (default): Each matrix component is
plotted in its own subplot.
            %                     - false: All components are combined in a
single plot.
            %     'display'   - (char) Specifies which part of the phasor to
display:
            %                     - 'real': Real part.
            %                     - 'imag': Imaginary part.
            %                     - 'both': Both real and imaginary parts.
            %                     - 'abs' (default): Magnitude of the
phasors.
            %     'marker'    - (cell array of chars) Marker styles for each
`PhasorArray`.
            %                     - Default:
{"o","*","x","square","diamond","^","v",">","<"}.
            %     'side'      - (char) Determines which harmonics are
displayed:
            %                     - 'both': Includes both positive and
negative harmonics.
            %                     - 'oneSided' (default): Displays only non-
negative harmonics.
            %     'parent'    - (graphics handle) Parent figure or axes for
the plot.
            %                     - Default: gcf (current figure).
            %
            %   Outputs:
            %     Tout        - (tiledlayout object) Handle to the tiled
layout used for the plot.
            %
            %   Behavior:
            %     - Multiple `PhasorArray` Objects: Each object in `o1` is
plotted separately using unique markers.
            %       If more objects are provided than marker styles, markers
are cycled.
            %     - Real vs. Complex Phasors: Automatically adjusts `side`
to 'both' if any input contains complex values.
            %
            %   Example Usage:
            %     % Plot absolute values of multiple PhasorArray objects
with linear scale
            %     A1 = PhasorArray(rand(3,3,11));
            %     A2 = PhasorArray(rand(3,3,11));
            %     Tout = stem(A1, A2, 'scale', 'linear', 'display', 'abs');
            %
            %     % Plot real part of a PhasorArray using custom markers
```

```
            %     A = PhasorArray(rand(4,4,11));
            %     stem(A, 'display', 'real', 'marker', {"o", "^"});
            %
            %     % Combine all components of multiple PhasorArray objects
in one plot
            %     stem(A1, A2, 'explosed', false, 'side', 'both');
            %
            %   See also: stemPhasor, tiledlayout.
            arguments (Repeating)
                o1
            end
            arguments
                varopt.scale {mustBeMember(varopt.scale,
{'log','linear'})}='log'
                varopt.explosed = true
                varopt.display {mustBeMember(varopt.display,
{'real','imag','both','abs'})} = 'abs'
                varopt.marker
={"o","*","x","square","diamond","^","v",">","<"};
                varopt.side {mustBeMember(varopt.side,{'both','oneSided'})}
= 'oneSided'
                varopt.parent = gcf
            end
            %check if all phasorArray in the cell o1 are real using cellfun
and isreal method
            if ~all(cellfun(@(x) isreal(x),o1))
                varopt.side='both';
            end

            if ~isa(varopt.marker,"cell")
                varopt.marker= { varopt.marker};
            end
            varhold=ishold;
            T =
stemPhasor(o1{1},scale=varopt.scale,hold=varhold,explosed=varopt.explosed,dis
play=varopt.display,marker=varopt.marker{1},side=varopt.side,parent=varopt.pa
rent);
            n=numel(o1);
            nmarker=numel(varopt.marker);
            for n_iter=2:n
                oi=o1{n_iter};
                ni=mod(n_iter,nmarker);
                if mod(ni,nmarker)==0
                    ni=nmarker;
                end

stemPhasor(oi,scale=varopt.scale,hold=true,explosed=varopt.explosed,marker=va
ropt.marker{ni},display=varopt.display,parent=T);
            end
            hold off
            if varhold
                hold on
            end
```

```
            if nargout>0
                Tout = T;
            end
        end
        function barsurf(o1,thres,hdel,varg)
            % BARSURF Generate a 3D bar surface plot of the phasors of a
`PhasorArray`.
            %
            %   BARSURF(o1, thres, hdel, varg) produces a 3D bar surface
plot representing
            %   the phasors of `o1`. Phasors below a specified threshold are
truncated, except
            %   for the first `hdel` harmonics that meet the condition,
which are retained as a margin.
            %
            %   Inputs:
            %     o1     - (PhasorArray) The PhasorArray object to be
plotted.
            %     thres  - (double, optional) The relative threshold for
truncating phasors.
            %                - Default: `1e-6`.
            %     hdel   - (integer, optional) The number of harmonics to
retain as a margin.
            %                - Default: `3`.
            %     varg   - (struct, optional) Name-value pair arguments:
            %                - 'scale' (char): Scale of the plot.
            %                    - 'log' (default): Logarithmic scale.
            %                    - 'linear': Linear scale.
            %
            %   Behavior:
            %     - **Thresholding**: Phasors with absolute values below
`thres * maxPhasor`
            %       are truncated, except for the first `hdel` harmonics.
            %     - **Logarithmic Scaling**: By default, the `log` scale is
used for better
            %       visualization of magnitude variations.
            %     - **Matrix Reshaping**: The function reshapes the phasors
into a suitable
            %       format for `bar3` visualization.
            %
            %   Example Usage:
            %     % Generate a bar plot with default threshold and margin
            %     barsurf(o1);
            %
            %     % Use a custom threshold and linear scaling
            %     barsurf(o1, 1e-6, 3, 'scale', 'linear');
            %
            %   See also: ReduceArray, bar3.
            arguments
                o1
                thres=1e-6;
                hdel=3
                varg.scale {mustBeMember(varg.scale,{'log','linear'})} ='log'
            end
```

```
            if isa(o1.value,'ndsdpvar')
                o1=value(value(o1));
                boolnan=isnan(o1);
                nnz(boolnan);
                if nnz(boolnan)>0
                    warning("Some sdpvar value are NaN")
                    o1(boolnan)=0;
                    o1=ReduceArray(o1);
                end
            end
            [nx,nz]=size(o1,[1 2]);
            nh=(size(o1,3)-1)/2;

[~,refM,hresM]=ReduceArray(o1,reduceMethod="relative",reduceThreshold=thres,e
xclude0Phasor=false);
            %refM contient en val absolue le plus grand phasor de chaque
            %composante de o1.
            minM_signif=min(abs(refM),[],'all')*thres;

            hdel=min(nh-hresM,hdel);
            epsM=10^(floor(log10(minM_signif)))*10^-(1.5);

            reshM=abs(reshape(ReduceArray(o1,hresM+hdel),nx*nz,[]));
            barsurf(reshM(:,((end+1)/2):end).',epsM,"yticklabel",
(0:hresM+hdel)','scale',varg.scale)
            %     barsurf(reshM(:,hres+1:end),min(ref,
[],'all')*thres,"xticklabel",(0:hres)','scale','log')
            xlabel("States")
            ylabel("Harmonics")
            title('Phasor of M')
            xlim([0 nx*nz+1])

        end
        function [r,t]=plot(o1,T,t,arg)
            % PLOT Evaluate and plot a T-periodic `PhasorArray`.
            %
            %   PLOT(o1, T, t, arg) computes and plots the time-domain
representation of a
            %   `PhasorArray`, assuming it is `T`-periodic.
            %
            %   This method evaluates the time-domain representation of a
`PhasorArray` and
            %   plots it over a specified period `T`. It provides flexible
options for
            %   visualization, including real/imaginary decomposition,
subplot arrangements,
            %   and axis linking.
            %
            %   Inputs:
            %     o1  - (PhasorArray) The `PhasorArray` object to be
evaluated and plotted.
            %     T   - (double, optional) Period of the `PhasorArray`.
            %               - Default: `1`.
            %     t   - (vector or scalar, optional) Time instants for
```

```
evaluation.
            %               - If `t = []`: A default time grid is generated
as `0:dt:T-dt`, where `dt = T/(20*h)`, with `h` the highest harmonic.
            %               - If `t = [tmin tmax]`: Uses `t = tmin:dt:tmax`
with `dt` computed as above.
            %               - If `t` is a vector: Evaluates `A(t)` at the
specified values.
            %               - If `t` is a scalar: Evaluates `A(t)` at that
single instant.
            %     arg - (struct, optional) Name-value pair arguments:
            %               - 'plot' (logical): Display the plot (default:
`true`).
            %               - 'explosed' (logical): Plot each matrix
component in a separate subplot (default: `true`).
            %               - 'hold' (logical): Hold the current plot
(default: `false`).
            %               - 'DispImag' (logical): Display the imaginary
part of the matrix (default: `false`).
            %               - 'DispReal' (logical): Display the real part of
the matrix (default: `true`).
            %               - 'ZeroCentered' (logical): Center the Y-axis
around zero (default: `false`).
            %               - 'title' (string): Custom title for the figure
(default: `[]`).
            %               - 'linetype' (string): Line style for the plot
(default: `'-'`).
            %               - 'GlobalYLim' (logical): Apply the same Y-limits
across subplots (default: `false`).
            %               - 'linkaxes' (string): Link axes of subplots
(`'x'`, `'y'`, `'xy'`, etc.) (default: `'x'`).
            %               - 'forceReal' (logical): Assume `o1` is real-
valued and simplify computation (default: `false`).
            %
            %   Outputs:
            %     r - (m × n × length(t) array) Evaluated time-domain
representation of `o1`.
            %     t - (vector) Time instants at which `o1` is evaluated.
            %
            %   Behavior:
            %     - **Evaluation**: Computes `A(t)` using
`PhasorArray2time`, which performs an inverse Fourier summation.
            %     - **Automatic Time Grid**: If `t = []`, it generates a
default time grid based on `T` and harmonics.
            %     - **Real-Valued Constraints**: If `o1` is real-valued,
`forceReal` is automatically set to `true`.
            %     - **Plot Customization**:
            %         - Supports separate subplots for each matrix element
(`explosed = true`).
            %         - Can plot only the real or imaginary part
(`DispReal`, `DispImag`).
            %         - Allows axis linking and uniform Y-axis scaling
across subplots.
            %
            %   Example Usage:
```

```
            %     % Evaluate and plot a PhasorArray over one period
            %     A = PhasorArray(rand(3,3,11));
            %     plot(A, 2*pi, []);
            %
            %     % Evaluate A(t) on a custom time range
            %     t = linspace(0, 2*pi, 100);
            %     plot(A, 2*pi, t, 'plot', true, 'explosed', false);
            %
            %     % Plot only the imaginary part with a different linestyle
            %     plot(A, 2*pi, [], 'DispImag', true, 'DispReal', false,
'linetype', '--');
            %
            %   See also: PhasorArray2time.
            arguments
                o1
                T=1
                t=[]
                arg.plot logical =true
                arg.explosed logical =true
                arg.hold logical =false
                arg.DispImag logical =false
                arg.DispReal logical =true
                arg.ZeroCentered logical =false
                arg.title=[]
                arg.linetype='-'
                arg.GlobalYLim logical =false
                arg.linkaxes='x'
                arg.forceReal = false
            end
            if ishold
                arg.hold = true;
            end
            if isreal(o1)
                arg.forceReal = true;
            end
            [rr,tt]=PhasorArray2time(o1,T,t,plot=arg.plot,
DispImag=arg.DispImag, ...

DispReal=arg.DispReal,explosed=arg.explosed,hold=arg.hold,ZeroCentered=arg.Ze
roCentered, ...

title=arg.title,linetype=arg.linetype,GlobalYLim=arg.GlobalYLim,linkaxes=arg.
linkaxes,forceReal=arg.forceReal);

            if nargout>0
                r=rr;
            end
            if nargout>1
                t=tt;
            end
        end
        function r=plot3D(o1,T,t,arg)
            %PLOT3D Produce a 3D plot where x-axis is the real part, y-axis
is the imaginary part, and z-axis is time
```

```
            %
            %   r = PLOT3D(o1, T, t, arg) produces a 3D plot where the x-
axis is the real part of A(t),
            %   the y-axis is the imaginary part of A(t), and the z-axis is
time.
            %   For each element of A, a subplot is created.
            %
            %   Input Arguments:
            %   o1 - The PhasorArray object to be plotted.
            %   T  - The period of the PhasorArray. Default is 1.
            %   t  - Time instants at which to evaluate the PhasorArray. Can
be:
            %       - [] (empty), then t takes the value [0 T].
            %       - [tmin tmax], then t=tmin:dt:tmax with automatic
discretization.
            %       - A vector on which A(t) is evaluated.
            %       - A single scalar, then t takes the value [0 t].
            %   arg - (Optional) Name-value pair arguments:
            %       'ZeroCentered' - Logical flag to normalize x and y axes
around zero. Default is false.
            %       'title' - String to display a custom title to the
figure. Default is [].
            %       'GlobalYLim' - Logical flag to enforce same Y and X
limits on the axes. Default is false.
            %       'linkaxes' - String to link zoom on the x, y, z axes.
Default is 'x'.
            %
            %   Output Arguments:
            %   r - The evaluated PhasorArray at the specified time instants.
            %
            %   Example:
            %   r = plot3D(o1, 2*pi, []);
            %   r = plot3D(o1, 2*pi, [0 2*pi], 'ZeroCentered', true,
'title', '3D Plot');
            %
            %   See also: PhasorArray2time
            arguments
                o1
                T=1
                t=[0 T]
                arg.ZeroCentered=false
                arg.title=[]
                arg.GlobalYLim=false
                arg.linkaxes='x'
            end
            if numel(t)==1
                t=sort([0 t]);
            end
            rr=PhasorArray2time(o1,T,t,plot=true,plot3D=true,
DispImag=false,
DispReal=true,explosed=true,ZeroCentered=arg.ZeroCentered,title=arg.title,Glo
balYLim=arg.GlobalYLim,linkaxes=arg.linkaxes);

            if nargout
```

```
            r=rr;
        end

    end

    function [r,t]=sim(o1,T,t,arg)
        % SIM Evaluate the time-domain representation of a `PhasorArray`.
        %
        %   SIM(o1, T, t, arg) computes the time-domain representation
of a `PhasorArray`,
        %   assuming it is `T`-periodic.
        %
        %   SIM is a **convenience function** for evaluating `A(t)`,
leveraging `PhasorArray2time`.
        %   It includes an option to enforce real-valued computation
when `A(t)` is known to be real.
        %
        %   Inputs:
        %     o1  - (PhasorArray) The `PhasorArray` object representing
a periodic matrix.
        %     T   - (double, optional) The period of `A(t)`.
        %               - Default: `1`.
        %     t   - (vector or scalar, optional) Time instants for
evaluation.
        %               - If `t = []`: Uses a default time grid `0:dt:T-
dt`, where `dt = T/(20*h)`, and `h` is the highest harmonic.
        %               - If `t = [tmin tmax]`: Uses `t = tmin:dt:tmax`
with automatic step size.
        %               - If `t` is a vector: Evaluates `A(t)` at the
specified values.
        %               - If `t` is a scalar: Evaluates `A(t)` at a
single instant.
        %     arg - (struct, optional) Name-value pair arguments:
        %               - 'isRealValued' (logical): Enforce real-valued
computation (default: `false`).
        %
        %   Outputs:
        %     r - (m × n × length(t) array) Evaluated time-domain
representation of `o1`.
        %     t - (vector) Time instants at which `o1` was evaluated.
        %
        %   Behavior:
        %     - If `o1` is **real-valued**, `isRealValued` is
automatically set to `true`.
        %     - Calls `PhasorArray2time` with a structured argument list
for evaluation.
        %
        %   Example Usage:
        %     % Evaluate A(t) over one period
        %     A = PhasorArray(rand(3,3,11));
        %     [r, t] = sim(A, 2*pi, []);
        %
        %     % Force real-valued computation
        %     [r, t] = sim(A, 2*pi, [], 'isRealValued', true);
```

```
            %
            %   See also: PhasorArray2time, plot.
            arguments
                o1
                T=1
                t=[]
                arg.isRealValued = false;
            end


            if isreal(o1)
                arg.isRealValued = true;
            end

            argo=struct;
            argo.plot=false;
            argo.explosed=true;
            argo.hold=false;
            argo.DispImag=false;
            argo.DispReal=true;
            argo.ZeroCentered=false;
            argo.title=[];
            argo.forceReal = arg.isRealValued;
            C=namedargs2cell(argo);
            [r,t]=PhasorArray2time(o1,T,t,C{:});
        end
        function r=evalTime(o1,T,t)
            % EVALTIME Evaluate a `PhasorArray` in the time domain for a
given period `T`.
            %
            %   EVALTIME(o1, T, t) computes the time-domain representation
of a `PhasorArray`,
            %   assuming it is `T`-periodic. The function returns `A(t)`,
evaluated at specified
            %   time instants `t`, without generating any plots.
            %
            %   Inputs:
            %     o1  - (PhasorArray) The `PhasorArray` object to be
evaluated.
            %     T   - (double, optional) The period of the `PhasorArray`.
            %               - Default: `1`.
            %     t   - (vector or scalar, optional) Time instants for
evaluation.
            %               - If `t = []`: Uses a default grid `0:dt:T-dt`,
with `dt = T/(20*h)`, where `h` is the highest harmonic.
            %               - If `t = [tmin tmax]`: Uses `t = tmin:dt:tmax`
with automatic step size.
            %               - If `t` is a vector: Evaluates `A(t)` at the
specified values.
            %               - If `t` is a scalar: Evaluates `A(t)` at a
single instant.
            %
            %   Outputs:
            %     r - (m × n × length(t) array) Evaluated time-domain
```

```
representation of `o1`.
            %
            %   Behavior:
            %     - Uses `plot(o1, T, t, ...)` internally but **without
plotting** (`'plot', false`).
            %     - Automatically generates a time grid if `t = []`.
            %     - Designed for numerical evaluation without visualization.
            %
            %   Example Usage:
            %     % Evaluate A(t) over one period
            %     A = PhasorArray(rand(3,3,11));
            %     r = evalTime(A, 2*pi, []);
            %
            %     % Evaluate A(t) on a custom time range
            %     t = linspace(0, 2*pi, 100);
            %     r = evalTime(A, 2*pi, t);
            %
            %   See also: plot, PhasorArray2time.
            arguments
                o1
                T=1
                t=[]
            end
            argo=struct;
            argo.plot=false;
            argo.explosed=true;
            argo.hold=false;
            argo.DispImag=false;
            argo.DispReal=true;
            argo.ZeroCentered=false;
            argo.title=[];
            C=namedargs2cell(argo);
            r=plot(o1,T,t,C{:});
        end
        function [y,t,dy]=initial(o1,x0,T,tfinal,varg)
            % INITIAL Compute the system response to an initial state in a
periodic state-space model.
            %
            %   INITIAL(o1, x0, T, tfinal, varg) simulates the system
response for:
            %       dx/dt = A(t)x
            %   where `A(t)` is a `T`-periodic state-space matrix. This
function calls `lsim` to compute the response.
            %
            %   Syntax:
            %     [y, t] = INITIAL(A, x0, T, tfinal)
            %     [y, t, dy] = INITIAL(A, x0, T, tfinal, Name, Value)
            %
            %   Inputs:
            %     o1     - (PhasorArray) The time-varying system matrix
`A(t)`, stored as a **3D phasor array**.
            %     x0     - (vector, optional) Initial state `x(0)`.
            %                - Default: `ones(size(o1,1),1)`.
            %     T      - (double, optional) Period of the system. Default:
```

```
`1`.
            %     tfinal - (scalar or vector, optional) Final simulation
time.
            %                - Default: `10*T`.
            %                - If scalar: Simulates from `t=0` to `t=tfinal`.
            %                - If `[tmin tmax]`: Simulates from `tmin` to
`tmax`.
            %                - If vector: Uses provided time grid.
            %
            %   Name-Value Pair Arguments:
            %     'opts'             - (struct) Options for the ODE solver
(`odeset`). Default: `[]`.
            %     'plot'             - (logical) Plot the state trajectory.
Default: `true`.
            %     'solver'           - (char) ODE solver method. Default:
`'adaptative'`. Options:
            %                               - `'adaptative'`, `'forward-
euler'`, `'RK4'`
            %     'FSprecpow'        - (integer) Power of 2 for frequency
sampling. Default: `8`.
            %     'checkReal'        - (logical) Force real-valued output
(only if system is **guaranteed** real). Default: `false`.
            %     'isRealValued'     - (logical) Force real-valued
computation. Default: `false`.
            %
            %   Outputs:
            %     y  - (matrix) State trajectory of the system (`size(y,1) =
size(o1,1)`).
            %     t  - (vector) Time instants at which `y(t)` is evaluated.
            %     dy - (matrix, optional) Derivative of the state trajectory
(only if `nargout > 2`).
            %
            %   Behavior:
            %     - This function is a **wrapper for `lsim`**, automatically
setting `U(t) = 0`.
            %     - If `isRealValued` is **not provided**, it is
automatically detected from `o1`.
            %     - If `tfinal = 0`, the function **automatically sets**
`tfinal = 10*T`.
            %
            %   Example Usage:
            %     % Simulate free response over one period
            %     A = PhasorArray(rand(3,3,11));
            %     [y, t] = initial(A, ones(3,1), 2*pi, 2*pi);
            %
            %     % Simulate with a fixed-step RK4 method
            %     [y, t] = initial(A, ones(3,1), 2*pi, 2*pi, 'solver',
'RK4');
            %
            %   See also: LSIM, HMQ_SIM.
            arguments
                o1
                x0=ones(size(o1,1),1)
                T=1
```

```
                tfinal=0
                varg.opts=[] %odeset option
                varg.plot=true
                varg.solver='adaptative'
                varg.FSprecpow=8
                varg.checkReal=0
                varg.isRealValued logical = false
            end
            if isreal(o1)
                varg.isRealValued = true;
            end
            vvarg = namedargs2cell(varg);
            if nargout == 3
                [y,t,dy]=lsim(o1,tfinal,x0,T,vvarg{:});
            else
                [y,t]=lsim(o1,tfinal,x0,T,vvarg{:});
            end

        end
        function [y,t,dy]=lsim(o1,tfinal,x0,T,Uph,varg)
            % LSIM Simulate the response of a time-periodic linear system.
            %
            %   LSIM(o1, tfinal, x0, T, Uph, varg) simulates the system:
            %       dx/dt = A(t)x + U(t)
            %   where `A(t)` and `U(t)` are `T`-periodic matrices,
represented as phasor arrays.
            %
            %   The function supports **adaptive and fixed-step solvers**,
allows **real-valued computation**, and
            %   provides **ODE solver customization** via `odeset` options.
            %
            %   Syntax:
            %     [y, t] = LSIM(A, tfinal, x0, T)
            %     [y, t, dy] = LSIM(A, tfinal, x0, T, U, Name, Value)
            %
            %   Inputs:
            %     o1     - (PhasorArray) The periodic system matrix `A(t)`,
stored as a **3D phasor array**.
            %     tfinal - (scalar or vector, optional) Final simulation
time.
            %                - Default: `10*T`
            %                - If scalar: Simulates from `t=0` to `t=tfinal`.
            %                - If `[tmin tmax]`: Simulates from `tmin` to
`tmax`.
            %                - If vector: Uses provided time grid.
            %     x0     - (vector, optional) Initial condition `x(0)`.
            %                - Default: `ones(size(o1,1),1)`.
            %     T      - (double, optional) The period of `A(t)`. Default:
`1`.
            %     Uph    - (PhasorArray or matrix, optional) The time-
varying input matrix `U(t)`.
            %                - Default: `[]` (zero input).
            %
            %   Name-Value Pair Arguments:
```

```
            %     'opts'            - (struct) Options for the ODE solver
(`odeset`). Default: `[]`.
            %     'plot'            - (logical) Plot the state trajectory.
Default: `true`.
            %     'solver'          - (char) ODE solver method. Default:
`'adaptative'`. Options:
            %                             - `'adaptative'`, `'forward-
euler'`, `'RK4'`
            %     'FSprecpow'       - (integer) Power of 2 for frequency
sampling. Default: `8`.
            %     'checkReal'       - (logical) Force real-valued output
(only if system is **guaranteed** real). Default: `false`.
            %     'isRealValued'    - (logical) Force real-valued
computation. Default: `false`.
            %
            %   Outputs:
            %     y  - (matrix) State trajectory of the system (`size(y,1) =
size(o1,1)`).
            %     t  - (vector) Time instants at which `y(t)` is evaluated.
            %     dy - (matrix, optional) Derivative of the state trajectory
(only if `nargout > 2`).
            %
            %   Behavior:
            %     - If `isRealValued` is **not provided**, it is
automatically detected from `o1` and `Uph`.
            %     - **Default solver:** `ode15s` (adaptive).
            %     - If `tfinal = 0`, the function **automatically sets**
`tfinal = 10*T`.
            %     - If `Uph` is empty, the system is simulated as
**homogeneous** (`dx/dt = A(t)x`).
            %     - The highest resolved harmonic `h` determines the
**integration step size**.
            %
            %   Example Usage:
            %     % Simulate system response over one period
            %     A = PhasorArray(rand(3,3,11));
            %     [y, t] = lsim(A, 2*pi, ones(3,1), 2*pi);
            %
            %     % Simulate with a time-varying input U(t)
            %     U = PhasorArray(rand(3,1,11));
            %     [y, t] = lsim(A, 2*pi, ones(3,1), 2*pi, U);
            %
            %     % Use a fixed-step RK4 method
            %     [y, t] = lsim(A, 2*pi, ones(3,1), 2*pi, [], 'solver',
'RK4');
            %
            %   See also: HMQ_SIM, ode15s.
            arguments
                o1
                tfinal=0
                x0=ones(size(o1,1),1)
                T=1
                Uph=[]
                varg.opts=[] %odeset option
```

```
                varg.plot=true
                varg.solver='adaptative'
                varg.FSprecpow=8
                varg.checkReal=0
                varg.isRealValued logical = false
            end
            if isempty(x0)
                x0=ones(size(o1,1),1);
            end
            if isreal(o1)
                varg.isRealValued = true;
            end

            C=namedargs2cell(varg);
            C{1}="odeOpts";

            %asking derivative trigger more computation from hmq_sim,
            %procede with care
            if nargout>2
                [y,t,dy] = hmq_sim(o1,tfinal,x0,T,Uph,C{:});
            else %sinon
                [y,t]    = hmq_sim(o1,tfinal,x0,T,Uph,C{:});
            end

        end

        function r=evalp(o1,angle,arg)
            % EVALP Evaluate a periodic matrix at a given phase angle.
            %
            %   EVALP(o1, angle, arg) computes the value of a periodic
matrix `A` at a
            %   specified phase angle `θ` instead of time. This function is
useful for
            %   evaluating periodic signals in the frequency domain.
            %
            %   Syntax:
            %     r = EVALP(A, angle)
            %     r = EVALP(A, angle, Name, Value)
            %
            %   Inputs:
            %     o1    - (PhasorArray) The periodic matrix represented as a
**phasor array**.
            %     angle - (double, vector) The phase angle(s) at which to
evaluate `A(θ)`.
            %
            %   Name-Value Pair Arguments:
            %     'forceReal'    - (logical) If `true`, forces real-valued
output. Default: `false`.
            %     'checkReal'    - (logical) If `true`, checks whether the
result is real-valued. Default: `false`.
            %     'checkRealTol' - (double) Tolerance for checking real-
valued results. Default: `1e-8`.
            %
            %   Outputs:
```

```
            %     r - (matrix) Evaluated periodic matrix at the given
**phase angle** `θ`.
            %
            %   Behavior:
            %     - This function evaluates `A(θ)`, where `θ` represents the
**phase angle** of the periodic function.
            %     - The function internally calls `PhasorArray2time` with `T
= 2π`, assuming **one full cycle** of the periodic function.
            %     - If `o1` is detected as real-valued, `forceReal` is
automatically set to `true`.
            %
            %   Example Usage:
            %     % Evaluate a phasor array at a phase angle of π/4
            %     result = evalp(A, pi/4);
            %
            %     % Force real-valued output
            %     result = evalp(A, pi/4, 'forceReal', true);
            %
            %   See also: PHASORARRAY2TIME, EVALTIME.
            arguments
                o1
                angle
                arg.forceReal logical    = false
                arg.checkReal logical    = false
                arg.checkRealTol         = 1e-8
            end

            if isreal(o1)
                arg.forceReal = true;
            end

            varg = namedargs2cell(arg);
            %evaluate periodic matrix A for an angle argument instead of
            %time
            r=PhasorArray2time(o1,2*pi,angle,varg{:});
        end


        function r = mreal(o1)
            %MREAL Compute the real part of a PhasorArray in the time domain.
            %   r = MREAL(o1) returns the real part of the PhasorArray o1 in
the time domain.
            %   The real part R(t) is such that A(t) = R(t) + i*I(t), where
I(t) is the imaginary part.
            %   Both R(t) and I(t) are real-valued.
            %
            %   INPUT:
            %       o1 - The PhasorArray object.
            %
            %   OUTPUT:
            %       r  - The real part of the PhasorArray in the time domain.
            %
            %   Example:
            %       A = PhasorArray.random(3, 3, 5);
```

```
            %       R = mreal(A);
            %
            %   See also: mimag, mconj
            dval=pvalue(o1);
            r = real(dval + flip(dval,3))*(1/2) + 1i * imag(dval -
flip(dval,3))*(1/2);
            r = PhasorArray(r);
        end

        function r = mimag(o1)
            %MIMAG Compute the imaginary part of a PhasorArray in the time
domain.
            %   r = MIMAG(o1) returns the imaginary part of the PhasorArray
o1 in the time domain.
            %   The imaginary part I(t) is such that A(t) = R(t) + i*I(t),
where R(t) is the real part.
            %   Both R(t) and I(t) are real-valued.
            %
            %   INPUT:
            %       o1 - The PhasorArray object.
            %
            %   OUTPUT:
            %       r  - The imaginary part of the PhasorArray in the time
domain.
            %
            %   Example:
            %       A = PhasorArray.random(3, 3, 5);
            %       I = mimag(A);
            %
            %   See also: mreal, mconj
            dval=pvalue(o1);
            r = real(dval - flip(dval,3))/2/1i + 1i * imag(dval +
flip(dval,3))/2/1i;
            r = PhasorArray(r);
        end

        function r = mconj(o1)
            %MCONJ Compute the complex conjugate of a PhasorArray in the
time domain.
            %   r = MCONJ(o1) returns the complex conjugate of the
PhasorArray o1 in the time domain.
            %   The complex conjugate is computed as R(t) - i*I(t), where
R(t) is the real part and I(t) is the imaginary part.
            %
            %   INPUT:
            %       o1 - The PhasorArray object.
            %
            %   OUTPUT:
            %       r  - The complex conjugate of the PhasorArray in the
time domain.
            %
            %   Example:
            %       A = PhasorArray.random(3, 3, 5);
            %       C = mconj(A);
```

```
    %
    %   See also: mreal, mimag
    r = mreal(o1)-1i*mimag(o1);
end

function o1=confirm_reality(o1)
    %  depreciated
    %
    % see also MREAL.

    pos_part=o1{:,:,1:o1.h};
    neg_part=o1{:,:,-1:-1:(-o1.h)};
    z_part=o1.phas(0);

    o1{:,:,1:o1.h}=(pos_part+conj(neg_part))/2;
    o1{:,:,0}=real(z_part);
    o1{:,:,-1:-1:(-o1.h)}=conj(pos_part+conj(neg_part))/2;
end

function r = pageconj(o1)
    %phasor conjugate
    r=(conj(pvalue(o1)));
end
function r = pagereal(o1)
    %phasor real part
    r=(real(pvalue(o1)));
end
function r = pageimag(o1)
    %phasor imag part
    r=(imag(pvalue(o1)));
end
function r = pageabs(o1)
    %phasor absolute value
    r=(abs(pvalue(o1)));
end


function r = isvector(o1)
    % ISVECTOR True if A(t) is a vector (row or column).
    r=isvector(o1.phas(0));
end
function r = isscalar(o1)
    % ISSCALAR True if A(t) is a scalar.
    r=isscalar(o1.phas(0));
end
function r = isrow(o1)
    % ISROW True if A(t) is a row vector.
    r=isrow(o1.phas(0));
end
function r = iscolumn(o1)
    % ISCOLUMN True if A(t) is a column vector.
    r=iscolumn(o1.phas(0));
end
function r = ismatrix(o1)
```

```
            % ISMATRIX True if A(t) is a matrix.
            r=ismatrix(o1.phas(0));
        end
        function r = isnumeric(o1)
            % ISNUMERIC True if A(t) contains numeric values.
            r=isnumeric(o1.value);
        end
        function r = islogical(o1)
            % ISLOGICAL True if A(t) contains logical values.
            r=islogical(o1.value);
        end
        function r = isempty(o1)
            % ISEMPTY True if A(t) is empty.
            r=isempty(o1.value);
        end
        function r = issquare(o1)
            % ISSQUARE True if A(t) is a square matrix.
            r = (size(o1,1)==size(o1,2));
        end

        function [r,R] = issymmetric(o1,arg)
            % ISSYMMETRIC Check if the PhasorArray object is symmetric.
            %
            %   [r, R] = ISSYMMETRIC(o1, arg) checks if the PhasorArray
object o1
            %   is symmetric based on the specified arguments.
            %
            %   Input arguments:
            %       o1 - The PhasorArray object to be checked.
            %       arg - A structure with the following fields:
            %           skewOption - A string specifying the type of
symmetry to check.
            %                        It can be either 'nonskew' or 'skew'.
Default is 'nonskew'.
            %           tol - A tolerance value for the symmetry check.
Default is 0.
            %
            %   Output arguments:
            %       r - A logical value indicating if the entire PhasorArray
object is symmetric.
            %       R - A logical array indicating the symmetry of each
slice of the PhasorArray object.
            %
            %   Example:
            %       [r, R] = issymmetric(o1, 'skewOption', 'nonskew', 'tol',
1e-6);
            %
            %   See also: arrayfun, norm, nnz
            arguments
                o1
                arg.skewOption {mustBeMember(arg.skewOption,
{'nonskew','skew'})} = 'nonskew'
                arg.tol = 0
            end
```

```
            if arg.tol == 0
                R = arrayfun(@(ii) issymmetric(o1(:,:,ii),arg.skewOption),1:
(2*o1.h+1));
            else
                tol=abs(arg.tol);
                switch arg.skewOption
                    case 'nonskew'
                        R = arrayfun(@(ii) norm(o1(:,:,ii)-
o1(:,:,ii).',"inf")/norm(o1(:,:,ii),"inf")<tol,1:(2*o1.h+1));
                    otherwise
                        R = arrayfun(@(ii) norm(o1(:,:,ii)
+o1(:,:,ii).',"inf")/norm(o1(:,:,ii),"inf")<tol,1:(2*o1.h+1));
                end
            end
            r = ((nnz(R))==(2*o1.h+1));
        end

        function [r,R] = ishermitian(o1,arg)
            % ISHERMITIAN Check if the PhasorArray object is Hermitian.
            %
            %   [r, R] = ISHERMITIAN(o1, arg) checks if the PhasorArray
object o1
            %   is Hermitian. The function returns a logical scalar r
indicating
            %   if all slices of the PhasorArray are Hermitian, and a
logical array R
            %   indicating if each individual slice is Hermitian.
            %
            %   Input arguments:
            %       o1 - PhasorArray object to be checked.
            %       arg - Structure with the following fields:
            %           skewOption - (optional) Specifies whether to check
for
            %                        'nonskew' (default) or 'skew'
Hermitian.
            %                        Must be one of {'nonskew', 'skew'}.
            %           tol - (optional) Tolerance for numerical comparison.
Default is 0.
            %
            %   Output arguments:
            %       r - Logical scalar indicating if all slices of the
PhasorArray
            %           are Hermitian.
            %       R - Logical array indicating if each individual slice of
the
            %           PhasorArray is Hermitian.
            arguments
                o1
                arg.skewOption {mustBeMember(arg.skewOption,
{'nonskew','skew'})} = 'nonskew'
                arg.tol = 0
            end
            if arg.tol == 0
                R = arrayfun(@(ii) ishermitian(o1(:,:,ii),arg.skewOption),1:
```

```
(2*o1.h+1));
            else
                tol=abs(arg.tol);
                switch arg.skewOption
                    case 'nonskew'
                        R = arrayfun(@(ii) norm(o1(:,:,ii)-
o1(:,:,ii)',"inf")/norm(o1(:,:,ii),"inf")<tol,1:(2*o1.h+1));
                    otherwise
                        R = arrayfun(@(ii) norm(o1(:,:,ii)
+o1(:,:,ii)',"inf")/norm(o1(:,:,ii),"inf")<tol,1:(2*o1.h+1));
                end
            end
            r = ((nnz(R))==(2*o1.h+1));
        end
        function [r,R] = isreal(o1,tol)
            % ISREAL Check if the imaginary part of the time realization of
            % PhasorArray is negligible compared to a given tolerance.
            %
            % Syntax:
            %   [r, R] = isreal(o1)
            %   [r, R] = isreal(o1, tol)
            %
            % Description:
            %   This function checks if the imaginary part of the time
realization
            %   of a PhasorArray object is negligible compared to a
specified tolerance.
            %   It returns a boolean value 'r' indicating whether the
imaginary part
            %   is negligible, and a logical array 'R' indicating the same
for each
            %   element of the PhasorArray.
            %
            % Inputs:
            %   o1  - PhasorArray object to be checked.
            %   tol - (Optional) Tolerance value for comparison. If not
provided,
            %         the function uses the default machine epsilon.
            %
            % Outputs:
            %   r   - Boolean value (true/false) indicating if the imaginary
part
            %         is negligible for the entire PhasorArray.
            %   R   - Logical array indicating if the imaginary part is
negligible
            %         for each element of the PhasorArray.
            %
            % Example:
            %   [r, R] = isreal(o1)
            %   [r, R] = isreal(o1, 1e-6)
            %
            % See also:
            %   mreal, ismembertol, abs
            arguments
```

```
                o1
                tol=[]
            end
            o1_r = mreal(o1);
            %if o1.value is not sym, sdpvar nor ndsdpvar
            if ~(isa(o1.value,"sym") || isa(o1.value,"ndsdpvar") ||
isa(o1.value,"sdpvar"))
                if isempty(tol)
                    r1 = ismembertol(real(o1.value),real(o1_r.value));
                    r2 = ismembertol(imag(o1.value),imag(o1_r.value));
                else
                    r1 = ismembertol(real(o1.value),real(o1_r.value),tol);
                    r2 = ismembertol(imag(o1.value),imag(o1_r.value),tol);
                end
            else %in this case ismembertol doesn't accept ym and sdpvr
input, use manuel difference instead
                if isempty(tol)
                    r1 = abs(real(o1.value)-real(o1_r.value))<eps;
                    r2 = abs(imag(o1.value)-imag(o1_r.value))<eps;
                else
                    r1 = abs(real(o1.value)-real(o1_r.value))<tol;
                    r2 = abs(imag(o1.value)-imag(o1_r.value))<tol;
                end
            end
                R = zeros(size(o1.value),'logical');
                R((r1 + r2)==2)= true;
                r = all(R,'all');
        end


        function tol = tolReal(o1,tolstart,tolTol)
            % tolReal - Determines the highest tolerance for which the input
is real.
            %
            % Syntax: tol = tolReal(o1, tolstart, tolTol)
            %
            % Inputs:
            %    o1 - The input object to be checked for reality.
            %    tolstart - (Optional) The starting tolerance value for the
binary search. Default is 100.
            %    tolTol - (Optional) The tolerance for the binary search
convergence. Default is 1e-20.
            %
            % Outputs:
            %    tol - The highest tolerance for which the input is real
through function isreal(o1, tol).
            %
            % Example:
            %    tol = tolReal(o1, 100, 1e-20);
            %
            % See also: isreal
            arguments
                o1
                tolstart=100
```

```
            tolTol=1e-20;
        end
        % Initialize variables
        low = 0;
        high = tolstart;

        % Binary search for minimum tolerance
        while (high - low) > tolTol
            mid = (low + high) / 2;
            if isreal(o1, mid)
                high = mid;
            else
                low = mid;
            end
        end

        % Return the minimum tolerance
        tol = high;
    end

    function [r,R] = isimag(o1,tol)
        % ISIMAG Check if the real part of the time realization of a
PhasorArray is negligible.
        %
        %   [r, R] = ISIMAG(o1, tol) checks whether the real part of the
time-domain
        %   realization of the PhasorArray `o1` is negligible relative
to a given tolerance.
        %
        %   Inputs:
        %       o1  - (PhasorArray) The PhasorArray object to be checked.
        %       tol - (double, optional) Tolerance threshold for
comparison.
        %             - Default: `[]`, meaning it uses machine precision.
        %
        %   Outputs:
        %       r - (logical) True if the entire PhasorArray is purely
imaginary.
        %       R - (logical array) Element-wise result indicating if
each entry is purely imaginary.
        %
        %   Example:
        %       [r, R] = isimag(o1, 1e-6);
        %
        %   See also: mimag, isreal, abs
        arguments
            o1
            tol=[]
        end
        o1_i = mimag(o1);
        %if o1.value is not sym, sdpvar nor ndsdpvar
        if ~(isa(o1.value,"sym") || isa(o1.value,"ndsdpvar") ||
isa(o1.value,"sdpvar"))
            if isempty(tol)
```

```
                    r1 = ismembertol(real(o1.value),real(o1_i.value));
                    r2 = ismembertol(imag(o1.value),imag(o1_i.value));
                else
                    r1 = ismembertol(real(o1.value),real(o1_i.value),tol);
                    r2 = ismembertol(imag(o1.value),imag(o1_i.value),tol);
                end
            else %in this case ismembertol doesn't accept ym and sdpvr
input, use manuel difference instead
                if isempty(tol)
                    r1 = abs(real(o1.value)-real(o1_i.value))<eps;
                    r2 = abs(imag(o1.value)-imag(o1_i.value))<eps;
                else
                    r1 = abs(real(o1.value)-real(o1_i.value))<tol;
                    r2 = abs(imag(o1.value)-imag(o1_i.value))<tol;
                end
            end
                R = zeros(size(o1.value),'logical');
                R((r1 + r2)==2)= true;
                r = all(R,'all');

        end
        function [r,R] = iscomplex(o1,tol)
            % ISCOMPLEX Check if the PhasorArray contains significant
imaginary components.
            %
            %   [r, R] = ISCOMPLEX(o1, tol) determines whether the
PhasorArray `o1`
            %   contains significant imaginary components relative to a
given tolerance.
            %
            %   Inputs:
            %       o1  - (PhasorArray) The PhasorArray object to be checked.
            %       tol - (double, optional) Tolerance threshold for
comparison.
            %             - Default: `eps` (machine precision).
            %
            %   Outputs:
            %       r - (logical) True if the entire PhasorArray has
significant imaginary components.
            %       R - (logical array) Element-wise result indicating if
each entry is complex.
            %
            %   Example:
            %       [r, R] = iscomplex(o1, 1e-6);
            %
            %   See also: isreal, isimag, abs
            arguments
                o1
                tol=eps
            end
            [r,R]=isreal(o1,tol);
            r=~r;
            R=~R;
        end
```

```
        function [r,R] = iszero(o1,tol)
            % ISZERO Check if the PhasorArray is numerically zero.
            %
            %   [r, R] = ISZERO(o1, tol) determines whether the PhasorArray
`o1`
            %   is numerically zero, based on an absolute tolerance.
            %
            %   Inputs:
            %       o1  - (PhasorArray) The PhasorArray object to be checked.
            %       tol - (double, optional) Tolerance threshold for
comparison.
            %             - Default: `0`, meaning exact zero comparison.
            %
            %   Outputs:
            %       r - (logical) True if all entries in the PhasorArray are
within the tolerance of zero.
            %       R - (logical array) Element-wise result indicating if
each entry is within tolerance.
            %
            %   Example:
            %       [r, R] = iszero(o1, 1e-6);
            %
            %   See also: pagenorm, isreal, iscomplex
            arguments
                o1
                tol =0
            end
            R = pagenorm(o1.value,"inf")<=tol;
            r = ((nnz(R))==(2*o1.h+1));
        end


        function r = ImagRealForm(o1)
            % IMAGREALFORM Convert PhasorArray to Imaginary-Real form.
            %
            %   r = IMAGREALFORM(o1) converts the PhasorArray `o1` into the
            %   Imaginary-Real representation, ensuring:
            %       - **Negative harmonics (`-k`) store the imaginary part
(left side).**
            %       - **Positive harmonics (`+k`) store the real part (right
side).**
            %       - The order `k` and `-k` remain **symmetric** around the
DC component.
            %
            %   This format assumes that `o1` represents a **real-valued
periodic matrix**,
            %   meaning `X_-k = conj(X_k)`.
            %
            %   Input:
            %       o1 - (PhasorArray) The input PhasorArray object.
            %
            %   Output:
            %       r - (array) The Imaginary-Real form representation:
            %           - Dimensions: `(m × n × (2h+1))`
```

```
            %           - Order `-k` maps to `(:,:,h-k+1)` (imaginary part,
negated).
            %           - Order `+k` maps to `(:,:,h+k+1)` (real part).
            %
            %   Example:
            %       r = ImagRealForm(o1);
            %
            %   See also: RealImagForm, SinCosForm
            h=o1.h;
            o1=o1.value;
            r=cat(3,-imag(o1(:,:,1:h)),real(o1(:,:,(h+1):end)));
        end
        function r = RealImagForm(o1)
            % REALIMAGFORM Convert PhasorArray to Real-Imaginary form.
            %
            %   r = REALIMAGFORM(o1) converts the PhasorArray `o1` into the
            %   Real-Imaginary representation, ensuring:
            %       - **Negative harmonics (`-k`) store the real part (left
side).**
            %       - **Positive harmonics (`+k`) store the imaginary part
(right side).**
            %       - The order `k` and `-k` remain **symmetric** around the
DC component.
            %
            %   This format assumes that `o1` represents a **real-valued
periodic matrix**,
            %   meaning `X_-k = conj(X_k)`.
            %
            %   Input:
            %       o1 - (PhasorArray) The input PhasorArray object.
            %
            %   Output:
            %       r - (array) The Real-Imaginary form representation:
            %           - Dimensions: `(m × n × (2h+1))`
            %           - Order `-k` maps to `(:,:,h-k+1)` (real part).
            %           - Order `+k` maps to `(:,:,h+k+1)` (imaginary part).
            %
            %   Example:
            %       r = RealImagForm(o1);
            %
            %   See also: ImagRealForm, SinCosForm
            r = flip(ImagRealForm(o1),3);
        end
        function r = SinCosForm(o1,isReal,realTol)
            % SINCOSFORM Convert PhasorArray to Sine-Cosine form.
            %
            %   r = SINCOSFORM(o1, isReal, realTol) converts the PhasorArray
`o1`
            %   into the **Sine-Cosine representation**, ensuring:
            %       - **Negative harmonics (`-k`) store sine coefficients
(left side).**
            %       - **Positive harmonics (`+k`) store cosine coefficients
(right side).**
            %       - The order `k` and `-k` remain **symmetric** around the
```

```
DC component.
            %
            %   Inputs:
            %       o1      - (PhasorArray) The input PhasorArray object.
            %       isReal  - (logical, optional) If `true`, forces real-
valued output.
            %                 Default: `isreal(o1)`.
            %       realTol - (double, optional) Tolerance for enforcing
real values.
            %                 Default: `1e-14`.
            %
            %   Output:
            %       r - (array) The Sine-Cosine form representation:
            %           - Dimensions: `(m × n × (2h+1))`
            %           - Order `-k` maps to `(:,:,h-k+1)` (sine
coefficient, negated).
            %           - Order `+k` maps to `(:,:,h+k+1)` (cosine
coefficient).
            %
            %   Example:
            %       r = SinCosForm(o1, true, 1e-14);
            %
            %   See also: CosSinForm, ImagRealForm
            arguments
                o1
                isReal logical = isreal(o1)
                realTol = 1e-14
            end
            %normal sin / cos form
            h=o1.h;
            o1=o1.value;
            r=cat(3,1i*(flip(o1(:,:,h+2:end),3)-o1(:,:,1:h)),o1(:,:,h+1),
(o1(:,:,h+2:end)+flip(o1(:,:,1:h),3)));
            if isReal
                er = real(r-real(r));
                if norm(double(er),'fro') > realTol
                    error("SinCosForm : Imaginary part of matrix %d is
superior to tolerance %d, switch isReal to false or adjust
tolerance",norm(double(er),'fro'),realTol)
                end
                r=real(r);
            end
        end
        function r = AngleAmpForm(o1)
            % ANGLEAMPFORM Convert PhasorArray to Angle-Amplitude form.
            %
            %   r = ANGLEAMPFORM(o1) converts the PhasorArray `o1` into
            %   the **Angle-Amplitude representation**, ensuring:
            %       - **Negative harmonics (`-k`) store phase angles (left
side).**
            %       - **Positive harmonics (`+k`) store amplitudes (right
side).**
            %       - The order `k` and `-k` remain **symmetric** around the
DC component.
```

```
            %
            %   This format assumes `o1` represents a **real-valued**
periodic matrix.
            %
            %   Input:
            %       o1 - (PhasorArray) The input PhasorArray object.
            %
            %   Output:
            %       r - (array) The Angle-Amplitude form representation:
            %           - Dimensions: `(m × n × (2h+1))`
            %           - Order `-k` maps to `(:,:,h-k+1)` (phase angle).
            %           - Order `+k` maps to `(:,:,h+k+1)` (amplitude).
            %
            %   Example:
            %       r = AngleAmpForm(o1);
            %
            %   See also: SinCosForm, ImagRealForm
            h=o1.h;

r=cat(3,angle(flip(o1.phas(1:h))),real(o1.phas(0)),abs(o1.phas(1:h)));
        end
        function r = CosSinForm(o1)
            % COSSINFORM Convert PhasorArray to Cosine-Sine form.
            %
            %   r = COSSINFORM(o1) converts the PhasorArray `o1` into
            %   the **Cosine-Sine representation**, ensuring:
            %       - **Negative harmonics (`-k`) store cosine coefficients
(left side).**
            %       - **Positive harmonics (`+k`) store sine coefficients
(right side).**
            %
            %   Input:
            %       o1 - (PhasorArray) The input PhasorArray object.
            %
            %   Output:
            %       r - (array) The Cosine-Sine form representation.
            %
            %   Example:
            %       r = CosSinForm(o1);
            %
            %   See also: SinCosForm, ImagRealForm
            r = flip(SinCosForm(o1),3);
        end

        function r = squeeval(o1)% SQUEEVAL Evaluate and squeeze PhasorArray.
            %
            %   r = SQUEEVAL(o1) evaluates the PhasorArray `o1` and applies
`squeeze`
            %   to remove singleton dimensions. This is useful for cases
where `o1`
            %   represents a scalar-valued PhasorArray.
            %
            %   Input:
            %       o1 - (PhasorArray) The input PhasorArray object.
```

```
            %
            %   Output:
            %       r - (array) The squeezed numerical evaluation of
`o1.value`.
            %
            %   Example:
            %       r = squeeval(o1);
            %
            %   See also: value, squeeze
            r=squeeze(value(o1));
        end

        function r = expandBase(o1,m)
            % EXPANDBASE Insert zeroed phasors to change the frequency base.
            %
            %   r = EXPANDBASE(o1, m) modifies the PhasorArray `o1` by
inserting `m-1`
            %   zero-valued phasors between each existing phasor,
effectively changing
            %   the frequency base from `w0` to `w1 = w0 / m`.
            %
            %   This transformation results in a PhasorArray with an
increased harmonic
            %   resolution, suitable for frequency resampling or down-
sampling applications.
            %
            %   Input:
            %       o1 - (PhasorArray) The input PhasorArray.
            %       m  - (integer) The expansion factor (must be ≥ 1).
            %
            %   Output:
            %       r  - (PhasorArray) The modified PhasorArray with
inserted zero phasors.
            %
            %   Behavior:
            %       - The harmonics are redistributed such that phasors
appear only at
            %         indices `k*m`, while the in-between phasors are zeroed.
            %       - The resulting PhasorArray corresponds to the same
periodic function,
            %         but expressed in a lower fundamental frequency `w1 =
w0/m`.
            %
            %   Example:
            %       A_new = expandBase(A, 3); % Adjusts A for frequency w0/3
            %
            %   See also: squishBase, pad
            h = o1.h;
            nx = size(o1,1);
            ny = size(o1,2);

            r = zeros(nx,ny,2*m*h+1);
            r(:,:,1:m:end) = o1.value;
            r = PhasorArray(r);
```

```
        end

        function r = squishBase(o1,m)
            % SQUISHBASE Remove phasors to change the frequency base.
            %
            %   r = SQUISHBASE(o1, m) modifies the PhasorArray `o1` by
retaining only
            %   every `m`-th phasor, effectively changing the frequency base
from `w0`
            %   to `w1 = w0 * m`.
            %
            %   This transformation results in a reduced harmonic
resolution, increasing
            %   the fundamental frequency but potentially losing information
in the process.
            %
            %   Input:
            %       o1 - (PhasorArray) The input PhasorArray.
            %       m  - (integer) The reduction factor (must be ≥ 1).
            %
            %   Output:
            %       r  - (PhasorArray) The modified PhasorArray with
retained harmonics.
            %
            %   Behavior:
            %       - Only harmonics at indices `k*m` are retained, while
others are discarded.
            %       - If `o1.h` is not a multiple of `m`, it is first padded
with zeros.
            %       - A warning is issued if nonzero harmonics are removed,
as this can
            %         lead to information loss and an inaccurate
representation.
            %
            %   Example:
            %       A_new = squishBase(A, 3); % Adjusts A for frequency w0 *
3
            %
            %   See also: expandBase, pad
            h = o1.h;
            nx = size(o1,1);
            ny = size(o1,2);

            %ensure o1.h is a multiple of m, otherwise pad o1 with zeros
            if mod(h,m)~=0
                o1 = pad(o1,m-mod(h,m));
                h = o1.h;
            end

            r = o1(:,:,1:m:end);

            r = PhasorArray(r);

            %evaluate the norm of deleted phasors in original phasorArray
```

```
            deleted = o1(:,:,setdiff(1:(2*h+1),1:m:(2*h+1)));
            normDeleted = norm(deleted(:),'fro');
            if normDeleted > 1e-10
                warning("squishBase : deleted phasors have a norm of %d, the
new phasorArray may not accurately represent the original
signal",normDeleted)
            end
        end

    end

    methods (Access=protected)
        function varargout = parenReference(obj,indexOp)
            obj.Phasor3D = obj.Phasor3D.(indexOp(1));
            if isscalar(indexOp)
                varargout{1} = pvalue(obj);
                return;
            end
            % Syntax for forwarding indexing operations
            [varargout{1:nargout}] = obj.(indexOp(2:end));
        end

        function obj = parenAssign(obj,indexOp,varargin)
            if isscalar(indexOp)
                assert(nargin==3);
                rhs = varargin{1};
                if isa(rhs,'PhasorArray')
                    obj.Phasor3D.(indexOp) = rhs.value;
                else
                    obj.Phasor3D.(indexOp) = rhs;
                end
                return;
            end
            [obj.(indexOp(2:end))] = varargin{:};
        end

        function n = parenListLength(obj,indexOp,ctx)
            if numel(indexOp) <= 2
                n = 1;
                return;
            end
            containedObj = obj.(indexOp(1:2));
            n = listLength(containedObj,indexOp(3:end),ctx);
        end

        function obj = parenDelete(obj,indexOp)
            obj.Phasor3D.(indexOp) = [];
        end

        function  varargout=braceReference(obj,indexOp)
            if numel(indexOp(1).Indices)==3
                argou=obj.sub(indexOp(1).Indices{1:2});
                varargout{1}=argou.phas(indexOp(1).Indices{3});
            else
```

```
            [varargout{1:nargout}]=obj.sub(indexOp(1).Indices{:});
            if ~isscalar(indexOp)
                for ii=1:numel(varargout)
                    varargout{ii}=varargout{ii}.(indexOp(2:end));
                end
            end
        end

        %             indexOp
        %             isscalar(indexOp)
        %             indexOp(1)
        %             indexOp.Type=Paren
        %             a=indexOp

        %             contobj=obj.Phasor3D
        %             contobj=contobj(:,:,indexOp(1))
        % %             obj.Phasor3D = obj.Phasor3D.(:,:,indexOp(1));
        %             if isscalar(indexOp)
        %                 varargout{1} = contobj;
        %                 return;
        %             end
        %             % Syntax for forwarding indexing operations
        % %             [varargout{1:nargout}] = obj.(indexOp(2:end));
        % %             objv=obj.value;
        %             [varargout{1:nargout}] = objv(:,:,indexOp);
    end

    function n = braceListLength(obj,indexOp,indexContext)
        if numel(indexOp) <= 2
            n = 1;
            return;
        end
        n=1;
    end

    function obj = braceAssign(obj,indexOp,varargin)

        h1=size(obj,3);
        h2=size(varargin{:},3);
        m1=size(obj,1);
        m2=size(obj,2);

        if numel(indexOp.Indices)==3
            n1=indexOp.Indices{1};
            n2=indexOp.Indices{2};

            n3=indexOp.Indices{3};
            if ischar(n3) %on a reçu l'argument ':', donc forcement un
phasorArray

            else
                if max(abs(n3))>obj.h
                    obj=obj.pad(max(abs(n3))-obj.h);
                end
```

```
                    n3=n3+1+obj.h;
                end
                obj(n1,n2,n3)=varargin{:};
                return
            end

            if h1<h2
                obj=obj.pad((h2-h1)/2);
            else
                varargin{:}=PhasorArray(PhasorArrayPad(varargin{:},(h1-h2)/
2));
            end

            if isscalar(indexOp)
                switch numel(indexOp.Indices)
                    case 1
                        m=indexOp.Indices{1};
                        if  numel(m)==obj.numelt %numelt renvoit la dim de
la matrice tempo ie n1 \times n2, donc ce serait un logical array en entrée
                            if any((m ~= 0) & (m ~= 1)) %on verifie que
c'est un logical array
                                error('Array indices must be positive
integers or logical values.')
                            end

                            n1=find(indexOp.Indices{1});
                        else
                            n1=indexOp.Indices{1};
                        end
                        n2=1;
                        % obj
                        obj=obj{:};
                    case 2
                        n1=indexOp.Indices{1};
                        n2=indexOp.Indices{2};
                end
                if isscalar(varargin{:})
                    varargin{:}=varargin{:}*ones(numel(n1),numel(n2));
                end

                %                 obj(n1,n2,:) = vect(varargin{:});
                obj(n1,n2,:) = (varargin{:});
                obj=reshape(obj,m1,m2,[]);
                return;
            end
            %             [braceReference(obj,indexOp)] = varargin{:};
        end

    end

    methods (Access=public)

        function ind = end(obj,k,n)
            s1 = size(obj,1);
```

```
            s2 = size(obj,2);
            s3 = size(obj,3);
            sz = [s1 s2 s3];
            if n == 1
                ind = s1*s2;
                return
            end
            ind = sz(k);
        end

        function out = value(obj)
            out = obj.Phasor3D;
        end

        function out = sum(obj,dim)
            %SUM Compute the sum along a given dimension.
            %
            %   out = SUM(obj, dim) returns the sum along the specified
dimension.
            %   If dim is 1 or 2, the output remains a PhasorArray.
            arguments
                obj
                dim=1
            end
            out = sum(obj.Phasor3D,dim);
            if dim==1 || dim==2
                out=PhasorArray(out);
            end
        end

        function out = cat(dim,varargin)
            %CAT Concatenate multiple PhasorArrays along a given dimension.
            %
            %   out = CAT(dim, A1, A2, ...) concatenates multiple
PhasorArrays or
            %   numeric arrays along the specified dimension.
            numCatArrays = nargin-1;
            if dim==1 || dim==2
                varargin2=PhasorUnif(varargin{:});
            else
                varargin2=varargin;
            end

            newArgs = cell(numCatArrays,1);
            for ix = 1:numCatArrays
                if isa(varargin2{ix},'PhasorArray')
                    newArgs{ix} = varargin2{ix}.Phasor3D;
                else
                    newArgs{ix} = varargin2{ix};
                end
            end
            out = PhasorArray(cat(dim,newArgs{:}));
        end
```

```
        function varargout = size(obj,varargin)
            [varargout{1:nargout}] = size(obj.Phasor3D,varargin{:});
        end

        function r = sdpval(o1)
            %SDPVAL Extract the numerical value of a PhasorArray whose value
is an SDPVAR.
            %
            %    r = sdpval(o1) extract the numerical value of o1 a
phasorArray with NDSDPVAR 3DArray.
            %    Basically perform PhasorArray(value(value(o1)))
            if  isa(o1.value,'ndsdpvar')||isa(o1.value,'sdpvar')||
isa(o1.value,'sym')
                o1=o1.value;
            end
            r=PhasorArray(value(o1));
        end

        [K,P,res] = place(A,B,poles,varg) ;

        out = expm(A,varg)

        out = logm(A,varg)
    end

    methods
        function r = Value(obj)
            r=obj.value;
        end
    end

    methods (Static, Access=public)
        function out = time2Phasor(At,nT,t,varg)
            %TIME2PHASOR Convert a time-dependent matrix into a PhasorArray
representation.
            %
            %   out = TIME2PHASOR(At, nT, t, varg) converts a 3D array
representing a
            %   real, time-dependent matrix (with time stored along the 3rd
dimension)
            %   into a PhasorArray by computing its Fourier coefficients.
            %
            %   Assumptions:
            %     - The sampling is uniform: Ts = t(2) - t(1).
            %     - The final time matches a full number of periods: t(end)
= nT*T - dt.
            %
            %   Inputs:
            %     At       - (3D array) Time-dependent matrix, stored as
MxNxtime.
            %     nT       - (integer, default: 1) Number of periods
captured in `t`.
            %     t        - (vector, optional) Time vector. If empty, it is
inferred.
```

```
            %     varg     - (Optional) Name-value pair arguments:
            %         'truncIndex' (integer, default: Inf)  - Maximum
harmonic order for truncation.
            %         'real'       (logical, default: true) - Enforce
conjugate symmetry (A_k = conj(A_-k)).
            %         'timeDim'    (integer, default: 3)    - Dimension
along which time varies.
            %
            %   Outputs:
            %     out - (PhasorArray) The computed PhasorArray
representation of `At`.
            %
            %   Example:
            %     t = linspace(0, 2*pi, 100);
            %     At = cat(3, sin(t), cos(t));
            %     Ph = time2Phasor(At, 1, t, 'truncIndex', 5);
            %
            %   See also: PhasorArray, TimeArray2Phasors
            %
            arguments
            At
            nT=1
            t=[]
            varg.truncIndex=Inf
            varg.real=true
            varg.timeDim=3
            end
            out =
PhasorArray(TimeArray2Phasors(At,nT,t,'truncIndex',varg.truncIndex,'isReal',v
arg.real,'timeDim',varg.timeDim));
        end

        function [phasorArr,feval] = funcToPhasorArray(func, T, n,varg)
            %FUNCTOPHASORARRAY Convert a time function into a PhasorArray.
            %
            %   [phasorArr, feval] = FUNCTOPHASORARRAY(func, T, n, varg)
evaluates a
            %   time-dependent function over a uniform time grid and
converts it into
            %   a PhasorArray using a Fourier transform.
            %
            %   Inputs:
            %     func - (function_handle) Time-dependent function, f(t),
returning an MxN matrix.
            %     T    - (scalar, default: 1) Period of the function.
            %     n    - (integer, default: 4) Number of frequency bins used
(2^n time steps).
            %     varg - (Optional) Name-value pair arguments:
            %         'reduce'     (logical, default: true)  - Reduce the
output PhasorArray.
            %         'reduceTol'  (double, default: 1e-15) - Threshold for
reducing small harmonics.
            %
            %   Outputs:
```

```
            %     phasorArr - (PhasorArray) The computed PhasorArray
representation of `func(t)`.
            %     feval     - (struct) Structure containing function
evaluations for debugging:
            %         - feval.T  : Period used.
            %         - feval.n  : Number of frequency bins.
            %         - feval.dt : Time step.
            %         - feval.func : Function handle used.
            %         - feval.At  : Evaluated matrix over time.
            %         - feval.t   : Time vector used.
            %
            %   Example:
            %     func = @(t) [sin(t); cos(t)];
            %     T = 2*pi;
            %     n = 5;
            %     [Ph, fData] = funcToPhasorArray(func, T, n);
            %
            %   See also: time2Phasor, PhasorArray
            %
            arguments
                func (1,1) {mustBeA(func, 'function_handle')} = @(t)
ones(3,3)
                T (1,1) {mustBeNumeric, mustBePositive} = 1
                n (1,1) {mustBeNumeric, mustBePositive, mustBeInteger} = 4
                varg.reduce = true;
                varg.reduceTol = 1e-15;
            end

            % Calculate dt
            dt = T / (2^n);

            % Create a time vector
            t = 0:dt:T-dt;

            % Evaluate the function over the time vector
            At = arrayfun(func, t, 'UniformOutput', false);
            At = cat(3, At{:});

            AtT = func( T);

            %evaluate the jump between the last and first value
            if norm(At(:,:,end)-At(:,:,1),'fro')>1e-10
                warning('The function has discontinuities, or a steep
derivative at the end, or has a jump between the last and first value, the
result may be incorrect. Jump value is %d (froebenius norm of f(T-dt)-
f(0))',norm(At(:,:,end)-At(:,:,1),'fro'));
            end

            if norm(At(:,:,1)-AtT,'fro')>1e-10
                warning('A(T) is different from A(0), the result may be
incorrect. Jump value is %d (froebenius norm of f(T)-f(0)), resulting
phasorArray is a periodic function with jump at time T',norm(At(:,:,1)-
AtT,'fro'));
            end
```

```
            % Convert the time-domain function values to a PhasorArray
            phasorArr = PhasorArray.time2Phasor(At);
            if varg.reduce
                phasorArr = reduce(phasorArr,
"reduceThreshold",varg.reduceTol,"reduceMethod","relative","exclude0Phasor",f
alse,"hardThresholdPhasors",true);

            end
            if nargout>1
                feval=struct;
                feval.T=T;
                feval.n=n;
                feval.dt=dt;
                feval.func=func;
                feval.At=At;
                feval.t=t;
                feval.plot = @(n,alone) At2plot(n,alone);
            end



            function At2plot(n,alone)
                %PLOTFUNC Plot time-domain representation of function
evaluations.
                %
                %   plotFunc(numPeriods, mode) plots the function
evaluations over `numPeriods`
                %   periods. The `mode` argument can be 'alone' (plot time-
domain only) or 'both'
                %   (overlay with the PhasorArray reconstruction).
                %
                %   Example:
                %     feval.plot(4, 'both');
                %
                if nargin==0
                    n=4;
                    alone="alone";
                end

                if nargin==1
                    if (ischar(n) || isstring(n))
                        alone=n;
                        n=4;


                    elseif nargin==1 && isnumeric(n)
                        alone="alone";
                    end
                end

                %assert if alone is "alone" or "both"
                assert(strcmp(alone,"alone")||strcmp(alone,"both"),"alone
must be 'alone' or 'both'");
```

```
                dt=T/(100);
                t = 0:dt:n*T;
                At = arrayfun(func, t, 'UniformOutput', false);
                At = cat(3, At{:});

                %plot each A(i,j,t) as a function of time
                for i=1:size(At,1)
                    for j=1:size(At,2)
                        subplot(size(At,1),size(At,2),(i-1)*size(At,2)+j);
                        plot(t,squeeze(At(i,j,:)));
                        title(sprintf('A_{%d,%d}(t)',i,j));
                    end
                end
                if strcmp(alone,"both")
                    hold on;
                    phasorArr.plot(T,[0 n*T]);
                    hold off;
                end
            end
        end


        function out = cqt2ScalarPhasor(cqtobj,varg)
            %CQT2SCALARPHASOR Convert a CQT object to a ScalarPhasorArray
representation.
            %
            %   out = CQT2SCALARPHASOR(cqtobj, varg) extracts the symbolic
Toeplitz
            %   representation of a compact quasi-Toeplitz (CQT) object and
maps it
            %   to a ScalarPhasorArray.
            %
            %   Inputs:
            %     cqtobj - (CQT object) The input compact quasi-Toeplitz
object.
            %     varg   - (Optional) Name-value pair arguments:
            %         'isReal' (logical, default: false) - If true, enforces
real-valued output
            %                                                  by ensuring
conjugate symmetry in
            %                                                  the phasor
representation.
            %
            %   Outputs:
            %     out - (ScalarPhasorArray) The extracted and converted
phasor representation.
            %
            %   Example:
            %     cqtA = cqt(rand(10), rand(10)); % Example CQT matrix
            %     phasorA = cqt2ScalarPhasor(cqtA, 'isReal', true);
            %
            %   See also: ScalarPhasorArray, cqt
            %
            arguments
```

```
                cqtobj %object of class cqt
                varg.isReal = false %enforce real output (ie conjugate
phasore pos/neg)
            end
            [nsTA,psTA]=symbol(cqtobj);
            n=min(numel(nsTA),numel(psTA));
            nsTA=nsTA(1:n);
            psTA=psTA(1:n);
            out=ScalarPhasorArray([flip(nsTA(2:end)) psTA]);
            if varg.isReal
                out = mreal(out);
            end
        end


        function obj = empty(varargin)
            %EMPTY Create an empty PhasorArray of specified size.
            %   See also: zeros, ones, eye
            obj = PhasorArray(double.empty(varargin{:}));
        end
        function obj = zeros(varargin)
            %ZEROS Create a zero-filled PhasorArray of specified size.
            %   See also: ones, eye, empty
            obj = PhasorArray(zeros(varargin{:}));
        end
        function obj = ones(varargin)
            %ONES Create a PhasorArray filled with ones.
            %   See also: zeros, eye, empty
            obj = PhasorArray(ones(varargin{:}));
        end
        function obj = eye(varargin)
            %EYE Create a PhasorArray with identity matrices at all
harmonics.
            %   EYE(n) returns an `n × n × 1` PhasorArray filled with
identity matrices.
            %   EYE(n, m) returns an `n × m × 1` PhasorArray filled with
identity matrices.
            %   EYE(n, m, h) returns an `n × m × (2h+1)` PhasorArray where
each slice along the 3rd dimension is an identity matrix.
            %   Unlike a standard identity matrix, this function replicates
the identity structure across all harmonic indices.
            %   See also: zeros, ones, empty
            if nargin==1
                varargin{2}=varargin{1};
                varargin{3}=1;
            elseif nargin==2
                varargin{3}=1;
            elseif nargin==3

            else
            end
            u=repmat(eye(varargin{1:2}),[1 1 2*varargin{3}+1]);
            obj = PhasorArray(u);
        end
```

```
        function obj = randomPhasorArrayWithPole(nx,poles,T,varg)
            % RANDOMPHASORARRAYWITHPOLE Generate a PhasorArray with
prescribed poles.
            %
            %   OBJ = RANDOMPHASORARRAYWITHPOLE(NX, POLES, T, VARG)
constructs a
            %   time-periodic matrix `A(t)` with the specified eigenvalues
(`poles`).
            %   A first random matrix `A` is generated, and a B*G term is
computed to
            %   ensure the desired eigenvalues by performing pole placement.
            %   The resulting PhasorArray `OBJ` is the difference `A-BK`,
where `K` is
            %   the obtained feedback gain.
            %
            %   Inputs:
            %     NX    - (integer) The size of the square matrix.
            %     POLES - (vector) Desired eigenvalues for `A(t)`, must have
length NX.
            %     T     - (scalar, optional) Period of the matrix. Default
is `2*pi`.
            %     VARG  - (optional) Name-value pair arguments:
            %               - 'h'      (integer) Harmonic order (default:
`5`).
            %               - 'BG'     (PhasorArray) Custom B*G term
(default: random).
            %               - 'isReal' (logical) Force a real-valued matrix
(default: `true`).
            %
            %   Outputs:
            %     OBJ - (PhasorArray) Generated PhasorArray with specified
poles.
            %
            %   See also: Sylv_harmonique, random
            arguments
                nx
                poles
                T  = 2*pi
                varg.h = []
                varg.BG = []
                varg.isReal = true
            end
            if isempty(varg.h) && isempty(varg.BG)
                varg.h=5;
                A  = PhasorArray.random(nx,nx,varg.h);
                BG = PhasorArray.random(nx,nx,varg.h);
            elseif isempty(varg.BG)
                A  = PhasorArray.random(nx,nx,varg.h);
                BG = PhasorArray.random(nx,nx,varg.h);
            elseif isempty(varg.h)
                varg.h = varg.BG.h;
                A  = PhasorArray.random(nx,nx,varg.h);
            else
                A  = PhasorArray.random(nx,nx,varg.h);
```

```
            end
            assert(numel(poles)==nx)
            La = PhasorArray(diag(poles));

            %Solve the appropriate Sylvester equation
            P = PhasorArray(Sylv_harmonique(-A,La,BG,4*varg.h,2*pi/T));
            %Compute K
            K = BG/P;
            %compute the new A with appropriate eigen values
            obj = A-K;
        end

        function obj = random(nx,ny,h,arg)
            % RANDOM Generate a random PhasorArray with optional structure
constraints.
            %
            %   OBJ = RANDOM(NX, NY, H, ARG) generates a random 3D
PhasorArray representing
            %   a time-periodic matrix with harmonics and structural
constraints.
            %
            %   Inputs:
            %     NX  - (integer) Number of rows.
            %     NY  - (integer) Number of columns.
            %     H   - (integer) Number of harmonics.
            %     ARG - (optional) Name-value pair arguments:
            %             - 'time_structure' (char) Time-domain structure:
            %                   'real', 'symmetric', 'antisymmetric', 'sdp',
'hermitian',
            %                   'cmplx', 'retroHermitian', etc. (default:
'real').
            %             - 'hurwitzeig'  (vector) Predefined Hurwitz
eigenvalues (default: `[]`).
            %             - 'T'           (scalar) Time period (default:
`1`).
            %             - 'Q'           (matrix) Optional quadratic matrix
(default: `[]`).
            %             - 'average_power_decay' (scalar) Power decay rate
of harmonics (default: `2`).
            %
            %   Outputs:
            %     OBJ - (PhasorArray) Randomly generated PhasorArray.
            %
            %   See also: rand_phasor
            arguments
                nx
                ny
                h
                arg.time_structure {mustBeMember(arg.time_structure,
{'real','symetric','antisymetric','sdp','hurwitz','Q-
spec','hermitian','cmplx','retroHermitian'})} ='real'
                arg.hurwitzeig=[]
                arg.T=1
                arg.Q=[]
```

```
            arg.average_power_decay=2;
        end
        if nargin ==1
            switch numel(nx)
                case 1
                case 2
                    ny = nx(2);
                    nx = nx(1);
                    h = 0;
                case 3
                    ny = nx(2);
                    h  = nx(3);
                    nx = nx(1);
            end
        end
        arg.output ='PhasorArray'; %or PhasorArray

        if ~isempty(arg.hurwitzeig) && strcmp(arg.time_structure,'real')
            arg.time_structure = 'hurwitz';
        end

        C=namedargs2cell(arg);
        obj = rand_phasor(nx,ny,h,C{:});
    end
    function A = sym(nx,ny,h,name)
        % SYM Construct a symbolic PhasorArray with structured harmonic
components.
        %
        %   A = SYM(NX, NY, H, NAME) creates an NX-by-NY PhasorArray
where each
        %   element is a symbolic expression representing harmonic
components.
        %
        %   Inputs:
        %     NX   - (integer) Number of rows (default: 1).
        %     NY   - (integer) Number of columns (default: NX).
        %     H    - (integer) Number of harmonics (default: 0).
        %     NAME - (string or cell) Base name for symbolic variables
(default: "a").
        %
        %   Outputs:
        %     A - (PhasorArray) Symbolic PhasorArray with structured
harmonics.
        %
        %   Notes:
        %     - Each entry in A has a symbolic expression for phasors:
        %       `A_0`, `A_plus_k`, `A_minus_k` for k > 0.
        %     - If NAME is a cell array, it must match the size of NX ×
NY.
        %     - For scalar NX × NY, NAME is used as the base for all
symbols.
        %
        %   See also: sym, ScalarPhasorArray, PhasorArray.
        arguments
```

```
                nx=1
                ny=nx
                h = 0
                name ="a"
            end
            if (nargin == 1) || (nargin==2 && (ischar(ny) || isstring(ny) ||
iscell(ny)))
                if ischar(ny) || isstring(ny) || iscell(ny)
                    name = ny;
                end
                switch numel(nx)
                    case 1
                        ny = nx;
                        h  = 0;
                    case 2
                        ny = nx(2);
                        nx = nx(1);
                        h  = 0;
                    case 3
                        ny = nx(2);
                        h  = nx(3);
                        nx = nx(1);
                    otherwise
                        error('cannot specify a dimension input of
length >3')
                end
            end
            if max(ny,nx)==1
                name={name};
            end
            if iscell(name)
                assert(numel(name)==nx*ny);
                A=PhasorArray.zeros(nx,ny);
                A.Phasor3D = sym(A.Phasor3D);
                for ii =  1:numel(name)
                    clear ap am a0 a
                    name_i=name{ii};
                    ap = sym(name_i+"_plus_",[1 h]);
                    am = sym(name_i+"_minus_",[1 h] );
                    a0 = sym(name_i+"_0");

                    a = cat(2,flip(am), a0, ap);
                    A{ii} = ScalarPhasorArray(a);
                end

            else
                ap = sym(name+"__%d__%d_plus_%d",[nx ny h]);
                am = sym(name+"__%d__%d_minus_%d",[nx ny h] );
                a0 = sym(name+"__%d__%d_0",[nx ny]);

                a = cat(3,flip(am,3), a0, ap);

                A= PhasorArray(a);
            end
```

```
        end
        function P = ndsdpvar(n1,n2,h,varg)
            % NDSPDPVAR Construct an `sdpvar`-based PhasorArray of specified
size and structure.
            %
            %   P = NDSPDPVAR(N1, N2, H, <name-value arguments>) generates
an SDP variable-based
            %   PhasorArray suitable for optimization in YALMIP.
            %
            %   Inputs:
            %     N1 - (integer) First dimension size.
            %     N2 - (integer, optional) Second dimension size (default:
N1).
            %     H  - (integer, optional) Number of harmonics (default: 0).
            %
            %   Name-Value Arguments:
            %     'PhasorType' (char) - Defines the structure of the phasor
(default: 'symmetric').
            %                           Options: 'symmetric', 'full',
'diagonal', etc.
            %                           See YALMIP documentation for the
complete list.
            %     'real' (logical)    - If true, ensures conjugate symmetry
for real-valued signals (default: true).
            %
            %   Outputs:
            %     P - (PhasorArray) PhasorArray containing `sdpvar` elements.
            %
            %   Notes:
            %     - The `PhasorType` argument defines **phasor structure**,
not time-domain structure.
            %     - Example: A **Hermitian** phasor structure enforces:
            %          conj(A_(ij)(t)) = A_(ji)(-t)
            %       However, for A(t) to be Hermitian in the **time
domain**, phasors must satisfy:
            %          A_k = ctrans(A_-k)
            %     - If `real=true`, ensures the phasor array represents a
real-valued periodic matrix.
            %     - If `h>0`, higher-order harmonics are created as
additional `sdpvar` variables.
            %
            %   Example:
            %     P = PhasorArray.ndsdpvar(4,4,5, 'PhasorType', 'symmetric',
'real', true);
            %       -> Produces a real-valued P(t), with 5 harmonics (size
11 along the third dimension),
            %          and enforces symmetry (i.e., P_ij(t) = P_ji(t)).
            %
            %     A = PhasorArray.ndsdpvar(4,4,5, 'PhasorType', 'full',
'real', true);
            %       -> Produces a real-valued A(t) with no additional
structure constraints.
            %
            %   See also: sdpvar, PhasorArray, PosPart2PhasorArray.
```

```
            arguments
                n1
                n2=n1
                h=0
                varg.PhasorType='symmetric'
                varg.real=true
            end
            if nargin ==1
                switch numel(n1)
                    case 1
                    case 2
                        n2 = n1(2);
                        n1 = n1(1);
                        h = 0;
                    case 3
                        n2 = n1(2);
                        h = n1(3);
                        n1 = n1(1);
                end
            end
            if n1~=n2
                if ismember(varg.PhasorType,{'symmetric'})
                    warning('non square matrix, PhasorType switched to
"full"')
                    varg.PhasorType='full';
                end
            end
            if varg.real
                P1=(ndsdpvar(n1,n2,1,varg.PhasorType,'real'));
                if h>0
                    P2=(ndsdpvar(n1,n2,h,varg.PhasorType,'complex'));
                    P = PosPart2PhasorArray(P1,P2);
                else
                    P=PhasorArray(P1);
                end
            else

P=PhasorArray(ndsdpvar(n1,n2,2*h+1,varg.PhasorType,'complex'));
            end
        end
    end
end


ans =

  PhasorArray with properties:

    Phasor3D: 1
```

Published with MATLAB® R2023b