Contents

1 Introduction

Each section describes a `Package` inside *NCAlgebra*.

Packages are automatically loaded unless otherwise noted.

2 Changes in Version 5.0

1. Completely rewritten core handling of noncommutative expressions.
2. Commands `Substitute`, `SubstituteSymmetric`, etc, have been replaced by the much more reliable commands in the new package NCReplace.
3. Modified behavior of `CommuteEverything` (see important notes in CommuteEverything).
4. Improvements and consolidation of NC calculus in the package NCDiff.
5. Added a complete set of linear algebra solvers in the new package MatrixDecomposition and their noncommutative versions in the new package NCMatrixDecomposition.
6. New algorithms for representing and operating with NC polynomials (NCPolynomial) and NC linear polynomials (NCSylvester).
7. General improvements on the Semidefinite Programming package NCSDP.

3 NonCommutativeMultiply

NonCommutativeMultiply is the main package that provides noncommutative functionality to Mathematica's native `NonCommutativeMultiply` bound to the operator `**`.

Members are:

- aj
- co
- Id
- inv
- tp
- rt
- CommutativeQ
- NonCommutativeQ
- SetCommutative
- SetNonCommutative
- Commutative
- CommuteEverything
- BeginCommuteEverything
- EndCommuteEverything
- ExpandNonCommutativeMultiply

3.1 aj

`aj[expr]` is the adjoint of expression `expr`. It is a conjugate linear involution.

See also: tp, co.

3.2 co

`co[expr]` is the conjugate of expression `expr`. It is a linear involution.

See also: aj.

3.3 Id

`Id` is noncommutative multiplicative identity. Actually Id is now set equal `1`.

3.4 inv

`inv[expr]` is the 2-sided inverse of expression `expr`.

3.5 rt

`rt[expr]` is the root of expression `expr`.

3.6 tp

`tp[expr]` is the tranpose of expression `expr`. It is a linear involution.

See also: aj, co.

3.7 CommutativeQ

`CommutativeQ[expr]` is *True* if expression `expr` is commutative (the default), and *False* if `expr` is noncommutative.

See also: SetCommutative, SetNonCommutative.

3.8 NonCommutativeQ

`NonCommutativeQ[expr]` is equal to `Not[CommutativeQ[expr]]`.

See also: CommutativeQ.

3.9 SetCommutative

`SetCommutative[a,b,c,...]` sets all the *Symbols* `a`, `b`, `c`, ... to be commutative.

See also: SetNonCommutative, CommutativeQ, NonCommutativeQ.

3.10 SetNonCommutative

`SetNonCommutative[a,b,c,...]` sets all the *Symbols* `a`, `b`, `c`, ... to be noncommutative.

See also: SetCommutative, CommutativeQ, NonCommutativeQ.

3.11 Commutative

`Commutative[symbol]` is commutative even if `symbol` is noncommutative.

See also: CommuteEverything, CommutativeQ, SetCommutative, SetNonCommutative.

3.12 CommuteEverything

`CommuteEverything[expr]` is an alias for BeginCommuteEverything.

See also: BeginCommuteEverything, Commutative.

3.13 BeginCommuteEverything

`BeginCommuteEverything[expr]` sets all symbols appearing in `expr` as commutative so that the resulting expression contains only commutative products or inverses. It issues messages warning about which symbols have been affected.

`EndCommuteEverything[]` restores the symbols noncommutative behaviour.

`BeginCommuteEverything` answers the question *what does it sound like?*

See also: EndCommuteEverything, Commutative.

3.14 EndCommuteEverything

`EndCommuteEverything[expr]` restores noncommutative behaviour to symbols affected by `BeginCommuteEverything`.

See also: BeginCommuteEverything, Commutative.

3.15 ExpandNonCommutativeMultiply

`ExpandNonCommutativeMultiply[expr]` expands out `**`s in `expr`.

For example

`ExpandNonCommutativeMultiply[a**(b+c)]`

returns

`a**b+a**c.`

Its aliases are `NCE`, and `NCExpand`.

4 NCCollect

Members are:

- NCCollect
- NCCollectSelfAdjoint
- NCCollectSymmetric
- NCStrongCollect
- NCStrongCollectSelfAdjoint
- NCStrongCollectSymmetric
- NCCompose
- NCDecompose
- NCTermsOfDegree

4.1 NCCollect

`NCCollect[expr,vars]` collects terms of nc expression `expr` according to the elements of `vars` and attempts to combine them. It is weaker than NCStrongCollect in that only same order terms are collected togther. It basically is `NCCompose[NCStrongCollect[NCDecompose]]`.

If `expr` is a rational nc expression then degree correspond to the degree of the polynomial obtained using NCRationalToNCPolynomial.

`NCCollect` also works with nc expressions instead of *Symbols* in vars. In this case nc expressions are replaced by new variables and `NCCollect` is called using the resulting expression and the newly created *Symbols*.

This command internally converts nc expressions into the special `NCPolynomial` format.

See also: NCStrongCollect, NCCollectSymmetric, NCCollectSelfAdjoint, NCStrongCollectSymmetric, NCStrongCollectSelfAdjoint, NCRationalToNCPolynomial.

4.2 NCCollectSelfAdjoint

`NCCollectSelfAdjoint[expr,vars]` allows one to collect terms of nc expression `expr` on the variables `vars` and their adjoints without writing out the adjoints.

This command internally converts nc expressions into the special `NCPolynomial` format.

See also: NCCollect, NCStrongCollect, NCCollectSymmetric, NCStrongCollect-Symmetric, NCStrongCollectSelfAdjoint.

4.3 NCCollectSymmetric

`NCCollectSymmetric[expr,vars]` allows one to collect terms of nc expression `expr` on the variables `vars` and their transposes without writing out the transposes.

This command internally converts nc expressions into the special `NCPolynomial` format.

See also: NCCollect, NCStrongCollect, NCCollectSelfAdjoint, NCStrongCollect-Symmetric, NCStrongCollectSelfAdjoint.

4.4 NCStrongCollect

`NCStrongCollect[expr,vars]` collects terms of expression `expr` according to the elements of `vars` and attempts to combine by association.

In the noncommutative case the Taylor expansion and so the collect function is not uniquely specified. The function `NCStrongCollect` often collects too much and while correct it may be stronger than you want.

For example, a symbol `x` will factor out of terms where it appears both linearly and quadratically thus mixing orders.

This command internally converts nc expressions into the special `NCPolynomial` format.

See also: NCCollect, NCCollectSymmetric, NCCollectSelfAdjoint, NCStrongCollectSymmetric, NCStrongCollectSelfAdjoint.

4.5 NCStrongCollectSelfAdjoint

`NCStrongCollectSymmetric[expr,vars]` allows one to collect terms of nc expression `expr` on the variables `vars` and their transposes without writing out the transposes.

This command internally converts nc expressions into the special `NCPolynomial` format.

See also: NCCollect, NCStrongCollect, NCCollectSymmetric, NCCollectSelfAdjoint, NCStrongCollectSymmetric.

4.6 NCStrongCollectSymmetric

`NCStrongCollectSymmetric[expr,vars]` allows one to collect terms of nc expression `expr` on the variables `vars` and their transposes without writing out the transposes.

This command internally converts nc expressions into the special `NCPolynomial` format.

See also: NCCollect, NCStrongCollect, NCCollectSymmetric, NCCollectSelfAdjoint, NCStrongCollectSelfAdjoint.

4.7 NCCompose

`NCCompose[dec]` will reassemble the terms in `dec` which were decomposed by `NCDecompose`.

`NCCompose[dec, degree]` will reassemble only the terms of degree `degree`.

The expression `NCCompose[NCDecompose[p,vars]]` will reproduce the polynomial `p`.

The expression `NCCompose[NCDecompose[p,vars], degree]` will reproduce only the terms of degree `degree`.

This command internally converts nc expressions into the special `NCPolynomial` format.

See also: NCDecompose, NCPDecompose.

4.8 NCDecompose

`NCDecompose[p,vars]` gives an association of elements of the nc polynomial `p` in variables `vars` in which elements of the same order are collected together.

`NCDecompose[p]` treats all nc letters in `p` as variables.

This command internally converts nc expressions into the special `NCPolynomial` format.

Internally `NCDecompose` uses `NCPDecompose`.

See also: NCCompose, NCPDecompose.

4.9 NCTermsOfDegree

`NCTermsOfDegree[expr,vars,indices]` returns an expression such that each term has the right number of factors of the variables in `vars`.

For example,

```
NCTermsOfDegree[x**y**x + x**w,{x,y},{2,1}]
```

returns `x**y**x` and

```
NCTermsOfDegree[x**y**x + x**w,{x,y},{1,0}]
```

return `x**w`. It returns 0 otherwise.

This command internally converts nc expressions into the special `NCPolynomial` format.

See also: NCDecompose, NCPDecompose.

5 NCSimplifyRational

NCSimplifyRational is a package with function that simplifies noncommutative expressions and certain functions of their inverses.

`NCSimplifyRational` simplifies rational noncommutative expressions by repeatedly applying a set of reduction rules to the expression. `NCSimplifyRationalSinglePass` does only a single pass.

Rational expressions of the form

```
inv[A + terms]
```

are first normalized to

inv[1 + terms/A]/A

using `NCNormalizeInverse`.

For each `inv` found in expression, a custom set of rules is constructed based on its associated nc Groebner basis.

For example, if

```
inv[mon1 + ... + K lead]
```

where `lead` is the leading monomial with the highest degree then the following rules are generated:

Original	Transformed
inv[mon1 + … + K lead] lead	(1 - inv[mon1 + … + K lead] (mon1 + …))/K
lead inv[mon1 + … + K lead]	(1 - (mon1 + …) inv[mon1 + … + K lead])/K

Finally the following pattern based rules are applied:

Original	Transformed
inv[a] inv[1 + K a b]	inv[a] - K b inv[1 + K a b]
inv[a] inv[1 + K a]	inv[a] - K inv[1 + K a]
inv[1 + K a b] inv[b]	inv[b] - K inv[1 + K a b] a
inv[1 + K a] inv[a]	inv[a] - K inv[1 + K a]
inv[1 + K a b] a	a inv[1 + K b a]

`NCPreSimplifyRational` only applies pattern based rules from the second table above. In addition, the following two rules are applied:

Original	Transformed
inv[1 + K a b] a b	(1 - inv[1 + K a b])/K
inv[1 + K a] a	(1 - inv[1 + K a])/K
a b inv[1 + K a b]	(1 - inv[1 + K a b])/K
a inv[1 + K a]	(1 - inv[1 + K a])/K

Rules in `NCSimplifyRational` and `NCPreSimplifyRational` are applied repeatedly.

Rules in `NCSimplifyRationalSinglePass` and `NCPreSimplifyRationalSinglePass` are applied only once.

The particular ordering of monomials used by `NCSimplifyRational` is the one implied by the `NCPolynomial` format. This ordering is a variant of the deg-lex ordering where the lexical ordering is Mathematica's natural ordering.

Members are:

- NCNormalizeInverse
- NCSimplifyRational
- NCSimplifyRationalSinglePass
- NCPreSimplifyRational
- NCPreSimplifyRationalSinglePass

5.1 NCNormalizeInverse

`NCNormalizeInverse[expr]` transforms all rational nc expressions of the form `inv[K + b]` into `inv[1 + (1/K) b]/K` if `A` is commutative.

See also: NCSimplifyRational, NCSimplifyRationalSinglePass.

5.2 NCSimplifyRational

`NCSimplifyRational[expr]` repeatedly applies `NCSimplifyRationalSinglePass` in an attempt to simplify the rational nc expression `expr`.

See also: NCNormalizeInverse, NCSimplifyRationalSinglePass.

5.3 NCSimplifyRationalSinglePass

`NCSimplifyRationalSinglePass[expr]` applies a series of custom rules only once in an attempt to simplify the rational nc expression `expr`.

See also: NCNormalizeInverse, NCSimplifyRational.

5.4 NCPreSimplifyRational

`NCPreSimplifyRational[expr]` repeatedly applies `NCPreSimplifyRationalSinglePass` in an attempt to simplify the rational nc expression `expr`.

See also: NCNormalizeInverse, NCPreSimplifyRationalSinglePass.

5.5 NCPreSimplifyRationalSinglePass

`NCPreSimplifyRationalSinglePass[expr]` applies a series of custom rules only once in an attempt to simplify the rational nc expression `expr`.

See also: NCNormalizeInverse, NCPreSimplifyRational.

6 NCDiff

NCDiff is a package containing several functions that are used in noncommutative differention of functions and polynomials.

Members are:

- NCDirectionalD
- NCGrad
- NCHessian

Members being deprecated:

- DirectionalD

6.1 NCDirectionalD

`NCDirectionalD[expr, {var1, h1}, ...]` takes the directional derivative of expression `expr` with respect to variables `var1`, `var2`, ... successively in the directions `h1`, `h2`, ...

For example, if:

```
expr = a**inv[1+x]**b + x**c**x
```

then

```
NCDirectionalD[expr, {x,h}]
```

returns

```
h**c**x + x**c**h - a**inv[1+x]**h**inv[1+x]**b
```

See also: NCGrad, NCHessian.

6.2 NCGrad

`NCGrad[expr, var1, ...]` gives the nc gradient of the expression `expr` with respect to variables `var1`, `var2`, ... If there is more than one variable then `NCGrad` returns the gradient in a list.

The transpose of the gradient of the nc expression `expr` is the derivative with respect to the direction `h` of the trace of the directional derivative of `expr` in the direction `h`.

For example, if:

```
expr = x**a**x**b + x**c**x**d
```

then its directional derivative in the direction `h` is

```
NCDirectionalD[expr, {x,h}]
```

which returns

```
h**a**x**b + x**a**h**b + h**c**x**d + x**c**h**d
```

and

```
NCGrad[expr, x]
```

returns the nc gradient

```
a**x**b + b**x**a + c**x**d + d**x**c
```

For example, if:

```
expr = x**a**x**b + x**c**y**d
```

is a function on variables `x` and `y` then

```
NCGrad[expr, x, y]
```

returns the nc gradient list

```
{a**x**b + b**x**a + c**y**d, d**x**c}
```

IMPORTANT: The expression returned by NCGrad is the transpose or the adjoint of the standard gradient. This is done so that no assumption on the symbols are needed. The calculated expression is correct even if symbols are self-adjoint or symmetric.

See also: NCDirectionalD.

6.3 NCHessian

`NCHessian[expr, {var1, h1}, ...]` takes the second directional derivative of nc expression `expr` with respect to variables `var1`, `var2`, ... successively in the directions `h1`, `h2`, ...

For example, if:

```
expr = y**inv[x]**y + x**a**x
```

then

```
NCHessian[expr, {x,h}, {y,s}]
```

returns

```
2 h**a**h + 2 s**inv[x]**s - 2 s**inv[x]**h**inv[x]**y -
2 y**inv[x]**h**inv[x]**s + 2 y**inv[x]**h**inv[x]**h**inv[x]**y
```

See also: NCDiretionalD, NCGrad.

6.4 DirectionalD

`DirectionalD[expr,var,h]` takes the directional derivative of nc expression `expr` with respect to the single variable `var` in direction `h`.

DEPRECATION NOTICE: This syntax is limited to one variable and is being deprecated in favor of the more general syntax in NCDirectionalD.

See also: NCDirectionalD

7 NCReplace

NCReplace is a package containing several functions that are useful in making replacements in noncommutative expressions. It offers replacements to Mathematica's `Replace`, `ReplaceAll`, `ReplaceRepeated`, and `ReplaceList` functions.

Commands in this package replace the old `Substitute` and `Transform` family of command which are been deprecated. The new commands are much more reliable and work faster than the old commands. From the beginning, substitution was always problematic and certain patterns would be missed. We reassure that the call expression that are returned are mathematically correct but some opportunities for substitution may have been missed.

Members are:

- NCReplace
- NCReplaceAll
- NCReplaceList
- NCReplaceRepeated
- NCMakeRuleSymmetric
- NCMakeRuleSelfAdjoint

7.1 NCReplace

`NCReplace[expr,rules]` applies a rule or list of rules `rules` in an attempt to transform the entire nc expression `expr`.

`NCReplace[expr,rules,levelspec]` applies `rules` to parts of `expr` specified by `levelspec`.

See also: NCReplaceAll, NCReplaceList, NCReplaceRepeated.

7.2 NCReplaceAll

`NCReplaceAll[expr,rules]` applies a rule or list of rules `rules` in an attempt to transform each part of the nc expression `expr`.

See also: NCReplace, NCReplaceList, NCReplaceRepeated.

7.3 NCReplaceList

`NCReplace[expr,rules]` attempts to transform the entire nc expression `expr` by applying a rule or list of rules `rules` in all possible ways, and returns a list of the results obtained.

`ReplaceList[expr,rules,n]` gives a list of at most `n` results.

See also: NCReplace, NCReplaceAll, NCReplaceRepeated.

7.4 NCReplaceRepeated

`NCReplaceRepeated[expr,rules]` repeatedly performs replacements using rule or list of rules `rules` until `expr` no longer changes.

See also: NCReplace, NCReplaceAll, NCReplaceList.

7.5 NCMakeRuleSymmetric

`NCMakeRuleSymmetric[rules]` add rules to transform the transpose of the left-hand side of `rules` into the transpose of the right-hand side of `rules`.

See also: NCMakeRuleSelfAdjoint, NCReplace, NCReplaceAll, NCReplaceList, NCReplaceRepeated.

7.6 NCMakeRuleSelfAdjoint

`NCMakeRuleSelfAdjoint[rules]` add rules to transform the adjoint of the left-hand side of `rules` into the adjoint of the right-hand side of `rules`.

See also: NCMakeRuleSymmetric, NCReplace, NCReplaceAll, NCReplaceList, NCReplaceRepeated.

8 NCSymmetric

Members are:

- NCSymmetricQ
- NCSymmetricTest

8.1 NCSymmetricQ

`NCSymmetricQ[expr]` returns *True* if `expr` is symmetric, i.e. if `tp[exp] == exp`.

`NCSymmetricQ` attempts to detect symmetric variables using `NCSymmetricTest`.

See also: NCSelfAdjointQ, NCSymmetricTest.

8.2 NCSymmetricTest

`NCSymmetricTest[expr]` attempts to establish symmetry of `expr` by assuming symmetry of its variables. `NCSymmetricTest[exp,options]` uses `options`.

`NCSymmetricTest` returns a list of two elements:

- the first element is *True* or *False* if it succeeded to prove `expr` symmetric.
- the second element is a list of the variables that were made symmetric.

The following options can be given:

- `SymmetricVariables`: list of variables that should be considered symmetric; use `All` to make all variables symmetric;
- `ExcludeVariables`: list of variables that should not be considered symmetric; use `All` to exclude all variables.

See also: NCSymmetricQ, NCNCSelfAdjointTest.

9 NCSelfAdjoint

Members are:

- NCSelfAdjointQ
- NCSelfAdjointTest

9.1 NCSelfAdjointQ

`NCSelfAdjointQ[expr]` returns true if `expr` is self-adjoint, i.e. if `aj[exp] == exp`.

See also: NCSymmetricQ, NCSelfAdjointTest.

9.2 NCSelfAdjointTest

`NCSelfAdjointTest[expr]` attempts to establish whether `expr` is self-adjoint by assuming that some of its variables are self-adjoint or symmetric. `NCSelfAdjointTest[expr,options]` uses `options`.

`NCSelfAdjointTest` returns a list of three elements:

- the first element is *True* or *False* if it succeded to prove `expr` self-adjoint.
- the second element is a list of variables that were made self-adjoint.
- the third element is a list of variables that were made symmetric.

The following options can be given:

- `SelfAdjointVariables`: list of variables that should be considered self-adjoint; use `All` to make all variables self-adjoint;
- `SymmetricVariables`: list of variables that should be considered symmetric; use `All` to make all variables symmetric;
- `ExcludeVariables`: list of variables that should not be considered symmetric; use `All` to exclude all variables.

See also: NCSelfAdjointQ.

10 NCOutput

NCOutput is a package that can be used to beautify the display of noncommutative expressions. NCOutput does not alter the internal representation of NC expressions, just the way they are displayed on the screen.

Members are:

- NCSetOutput
- NCOutputFunction

10.1 NCOutputFunction

`NCOutputFunction[exp]` returns a formatted version of the expression `expr` which will be displayed to the screen.

See also: NCSetOutput.

10.2 NCSetOutput

`NCSetOutput[options]` controls the display of expressions in a special format without affecting the internal representation of the expression.

The following `options` can be given:

- `Dot`: If *True* `x**y` is displayed as `x.y`;
- `tp`: If *True* `tp[x]` is displayed as `x` with a superscript 'T';
- `inv`: If *True* `inv[x]` is displayed as `x` with a superscript '-1';
- `aj`: If *True* `aj[x]` is displayed as `x` with a superscript '*';
- `rt`: If *True* `rt[x]` is displayed as `x` with a superscript '1/2';
- `Array`: If *True* matrices are displayed using `MatrixForm`;
- `All`: Set all available options to *True* or *False*.

See also: NCOutputFunciton.

11 NCMatMult

Members are:

- tpMat
- ajMat
- coMat

- MatMult
- NCInverse
- NCMatrixExpand

11.1 tpMat

`tpMat[mat]` gives the transpose of matrix `mat` using `tp`.

See also: ajMat, coMat, MatMult.

11.2 ajMat

`ajMat[mat]` gives the adjoint transpose of matrix `mat` using `aj` instead of `ConjugateTranspose`.

See also: tpMat, coMat, MatMult.

11.3 coMat

`coMat[mat]` gives the conjugate of matrix `mat` using `co` instead of `Conjugate`.

See also: tpMat, ajMat, MatMult.

11.4 MatMult

`MatMult[mat1, mat2, ...]` gives the matrix multiplication of `mat1`, `mat2`, ... using `NonCommutativeMultiply` rather than `Times`.

See also: tpMat, ajMat, coMat.

11.4.1 Notes

The experienced matrix analyst should always remember that the Mathematica convention for handling vectors is tricky.

- `{{1,2,4}}` is a 1x3 *matrix* or a *row vector*;
- `{{1},{2},{4}}` is a 3x1 *matrix* or a *column vector*;
- `{1,2,4}` is a *vector* but **not** a *matrix*. Indeed whether it is a row or column vector depends on the context. We advise not to use *vectors*.

11.5 NCInverse

`NCInverse[mat]` gives the nc inverse of the square matrix `mat`. `NCInverse` uses partial pivoting to find a nonzero pivot.

`NCInverse` is primarily used symbolically. Usually the elements of the inverse matrix are huge expressions. We recommend using `NCSimplifyRational` to improve the results.

See also: tpMat, ajMat, coMat.

11.6 NCMatrixExpand

`NCMatrixExpand[expr]` expands `inv` and `**` of matrices appearing in nc expression `expr`. It effectively substitutes `inv` for `NCInverse` and `**` by `MatMult`.

See also: NCInverse, MatMult.

12 NCPolynomial

This package contains functionality to convert an nc polynomial expression into an expanded efficient representation for an nc polynomial which can have commutative or noncommutative coefficients.

For example the polynomial

```
exp = a**x**b - 2 x**y**c**x + a**c
```

in variables `x` and `y` can be converted into an NCPolynomial using

```
p = NCToNCPolynomial[exp, {x,y}]
```

which returns

```
p = NCPolynomial[a**c, <|{x}->{{1,a,b}},{x**y,x}->{{2,1,c,1}}|>, {x,y}]
```

Members are:

- NCPolynomial
- NCToNCPolynomial
- NCPolynomialToNC
- NCRationalToNCPolynomial
- NCPCoefficients
- NCPTermsOfDegree
- NCPTermsOfTotalDegree
- NCPTermsToNC
- NCPSort
- NCPDecompose
- NCPDegree

- NCPMonomialDegree
- NCPCompatibleQ
- NCPSameVariablesQ
- NCPMatrixQ
- NCPLinearQ
- NCPQuadraticQ
- NCPNormalize

12.1 NCPolynomial

`NCPolynomial[indep,rules,vars]` is an expanded efficient representation for an nc polynomial in `vars` which can have commutative or noncommutative coefficients.

The nc expression `indep` collects all terms that are independent of the letters in `vars`.

The *Association* `rules` stores terms in the following format:

```
{mon1, ..., monN} -> {scalar, term1, ..., termN+1}
```

where:

- `mon1, ..., monN`: are nc monomials in vars;
- `scalar`: contains all commutative coefficients; and
- `term1, ..., termN+1`: are nc expressions on letters other than the ones in vars which are typically the noncommutative coefficients of the polynomial.

`vars` is a list of *Symbols*.

For example the polynomial

```
a**x**b - 2 x**y**c**x + a**c
```

in variables `x` and `y` is stored as:

```
NCPolynomial[a**c, <|{x}->{{1,a,b}},{x**y,x}->{{2,1,c,1}}|>, {x,y}]
```

NCPolynomial specific functions are prefixed with NCP, e.g. NCPDegree.

See also: `NCToNCPolynomial`, `NCPolynomialToNC`, `NCTermsToNC`.

12.2 NCToNCPolynomial

`NCToNCPolynomial[p, vars]` generates a representation of the noncommutative polynomial `p` in `vars` which can have commutative or noncommutative coefficients.

`NCToNCPolynomial[p]` generates an `NCPolynomial` in all nc variables appearing in `p`.

Example:

```
exp = a**x**b - 2 x**y**c**x + a**c
p = NCToNCPolynomial[exp, {x,y}]
```

returns

```
NCPolynomial[a**c, <|{x}->{{1,a,b}},{x**y,x}->{{2,1,c,1}}|>, {x,y}]
```

See also: `NCPolynomial`, `NCPolynomialToNC`.

12.3 NCPolynomialToNC

`NCPolynomialToNC[p]` converts the NCPolynomial `p` back into a regular nc polynomial.

See also: `NCPolynomial`, `NCToNCPolynomial`.

12.4 NCRationalToNCPolynomial

`NCRationalToNCPolynomial[r, vars]` generates a representation of the noncommutative rational expression `r` in `vars` which can have commutative or noncommutative coefficients.

`NCRationalToNCPolynomial[r]` generates an `NCPolynomial` in all nc variables appearing in `r`.

`NCRationalToNCPolynomial` creates one variable for each `inv` expression in `vars` appearing in the rational expression `r`. It returns a list of three elements:

- the first element is the `NCPolynomial`;
- the second element is the list of new variables created to replace `invs`;
- the third element is a list of rules that can be used to recover the original rational expression.

For example:

```
exp = a**inv[x]**y**b - 2 x**y**c**x + a**c
{p,rvars,rules} = NCRationalToNCPolynomial[exp, {x,y}]
```

returns

```
p = NCPolynomial[a**c, <|{rat1**y}->{{1,a,b}},{x**y,x}->{{2,1,c,1}}|>, {x,y,rat1}]
rvars = {rat1}
rules = {rat1->inv[x]}
```

See also: `NCToNCPolynomial`, `NCPolynomialToNC`.

12.5 NCPCoefficients

`NCPCoefficients[p, m]` gives all coefficients of the NCPolynomial `p` in the monomial `m`.

For example:

```
exp = a ** x ** b - 2 x ** y ** c ** x + a ** c + d ** x
p = NCToNCPolynomial[exp, {x, y}]
NCPCoefficients[p, {x}]
```

returns

```
{{1, d, 1}, {1, a, b}}
```

and

```
NCPCoefficients[p, {x ** y, x}]
```

returns

```
{{-2, 1, c, 1}}
```

See also: `NCPTermsToNC`.

12.6 NCPTermsOfDegree

`NCPTermsOfDegree[p,deg]` gives all terms of the NCPolynomial `p` of degree `deg`.

The degree `deg` is a list with the degree of each symbol.

For example:

```
p = NCPolynomial[0, <|{x,y}->{{2,a,b,c}},
                       {x,x}->{{1,a,b,c}},
                       {x**x}->{{-1,a,b}}|>, {x,y}]
NCPTermsOfDegree[p, {1,1}]
```

returns

```
<|{x,y}->{{2,a,b,c}}|>
```

and

```
NCPTermsOfDegree[p, {2,0}]
```

returns

```
<|{x,x}->{{1,a,b,c}}, {x**x}->{{-1,a,b}}|>
```

See also: `NCPTermsOfTotalDegree`,`NCPTermsToNC`.

12.7 NCPTermsOfTotalDegree

`NCPTermsOfDegree[p,deg]` gives all terms of the NCPolynomial `p` of total degree `deg`.

The degree `deg` is the total degree.

For example:

```
p = NCPolynomial[0, <|{x,y}->{{2,a,b,c}},
                      {x,x}->{{1,a,b,c}},
                      {x**x}->{{-1,a,b}}|>, {x,y}]
NCPTermsOfDegree[p, 2]
```

returns

```
<|{x,y}->{{2,a,b,c}},{x,x}->{{1,a,b,c}},{x**x}->{{-1,a,b}}|>
```

See also: `NCPTermsOfDegree`,`NCPTermsToNC`.

12.8 NCPTermsToNC

`NCPTermsToNC` gives a nc expression corresponding to terms produced by `NCPTermsOfDegree` or `NCTermsOfTotalDegree`.

For example:

```
terms = <|{x,x}->{{1,a,b,c}}, {x**x}->{{-1,a,b}}|>
NCPTermsToNC[terms]
```

returns

```
a**x**b**c-a**x**b
```

See also: `NCPTermsOfDegree`,`NCPTermsOfTotalDegree`.

12.9 NCPPlus

`NCPPlus[p1,p2,...]` gives the sum of the nc polynomials `p1`,`p2`,... .

12.10 NCPSort

`NCPSort[p]` gives a list of elements of the NCPolynomial `p` in which monomials are sorted first according to their degree then by Mathematica's implicit ordering.

For example

```
NCPSort[NCPolynomial[c + x**x - 2 y, {x,y}]]
```

will produce the list

```
{c, -2 y, x**x}
```

See also: NCPDecompose, NCDecompose, NCCompose.

12.11 NCPDecompose

`NCPDecompose[p]` gives an association of elements of the NCPolynomial `p` in which elements of the same order are collected together.

For example

```
NCPDecompose[NCPolynomial[a**x**b+c+d**x**e+a**x**e**x**b+a**x**y, {x,y}]]
```

will produce the Association

```
<|{1,0}->a**x**b + d**x**e, {1,1}->a**x**y, {2,0}->a**x**e**x**b, {0,0}->c|>
```

See also: NCPSort, NCDecompose, NCCompose.

12.12 NCPDegree

`NCPDegree[p]` gives the degree of the NCPolynomial `p`.

See also: `NCPMonomialDegree`.

12.13 NCPMonomialDegree

`NCPDegree[p]` gives the degree of each monomial in the NCPolynomial `p`.

See also: `NCDegree`.

12.14 NCPLinearQ

`NCPLinearQ[p]` gives True if the NCPolynomial `p` is linear.

See also: `NCPQuadraticQ`.

12.15 NCPQuadraticQ

`NCPQuadraticQ[p]` gives True if the NCPolynomial `p` is quadratic.

See also: `NCPLinearQ`.

12.16 NCPCompatibleQ

`NCPCompatibleQ[p1,p2,...]` returns *True* if the polynomials `p1`,`p2`,... have the same variables and dimensions.

See also: NCPSameVariablesQ, NCPMatrixQ.

12.17 NCPSameVariablesQ

`NCPSameVariablesQ[p1,p2,...]` returns *True* if the polynomials `p1`,`p2`,... have the same variables.

See also: NCPCompatibleQ, NCPMatrixQ.

12.18 NCPMatrixQ

`NCMatrixQ[p]` returns *True* if the polynomial `p` is a matrix polynomial.

See also: NCPCompatibleQ.

12.19 NCPNormalize

`NCPNormalizes[p]` gives a normalized version of NCPolynomial p where all factors that have free commutative products are collectd in the scalar.

This function is intended to be used mostly by developers.

See also: `NCPolynomial`

13 NCSylvester

NCSylvester is a package that provides functionality to handle linear polynomials.

Members are:

- NCSylvester
- NCSylvesterToNCPolynomial

13.1 NCSylvester

`NCSylvester[p]` gives an expanded representation for the linear `NCPolynomial` `p`.

`NCSylvester` returns a list with two elements:

- the first is a the independent term;
- the second is an association where each key is one of the variables and each value is a list with three elements:
- the first element is a list of left NC symbols;
- the second element is a list of right NC symbols;
- the third element is a numeric `SparseArray`.

Example:

```
p = NCToNCPolynomial[2 + a**x**b + c**x**d + y, {x,y}];
{p0,sylv} = NCSylvester[p,x]
```

produces

```
p0 = 2
sylv = <|x->{{a,c},{b,d},SparseArray[{{1,0},{0,1}}]},
         y->{{1},{1},SparseArray[{{1}}]}|>
```

See also: NCSylvesterToNCPolynomial, NCPolynomial.

13.2 NCSylvesterToNCPolynomial

`NCSylvesterToNCPolynomial[rep]` takes the list `rep` produced by `NCSylvester` and converts it back to an `NCPolynomial`.

`NCSylvesterToNCPolynomial[rep,options]` uses `options`.

The following `options` can be given: * `Collect` (*True*): controls whether the coefficients of the resulting NCPolynomial are collected to produce the minimal possible number of terms.

See also: NCSylvester, NCPolynomial.

14 NCQuadratic

NCQuadratic is a package that provides functionality to handle quadratic polynomials.

Members are:

- NCQuadraticMakeSymmetric
- NCMatrixOfQuadratic
- NCQuadratic
- NCQuadraticToNCPolynomial

14.1 NCQuadratic

`NCQuadratic[p]` gives an expanded representation for the quadratic `NCPolynomial` `p`.

`NCQuadratic` returns a list with four elements:

- the first element is the independent term;
- the second represents the linear part as in `NCSylvester`;
- the third element is a list of left NC symbols;
- the fourth element is a numeric `SparseArray`;
- the fifth element is a list of right NC symbols.

Example:

```
exp = d + x + x**x + x**a**x + x**e**x + x**b**y**d + d**y**c**y**d;
vars = {x,y};
p = NCToNCPolynomial[exp, vars];
{p0,sylv,left,middle,right} = NCQuadratic[p];
```

produces

```
p0 = d
sylv = <|x->{{1},{1},SparseArray[{{1}}]}, y->{{},{},{}}|>
left =  {x,d**y}
middle = SparseArray[{{1+a+e,b},{0,c}}]
right = {x,y**d}
```

See also: NCSylvester,NCQuadraticToNCPolynomial,NCPolynomial.

14.2 NCQuadraticMakeSymmetric

`NCQuadraticMakeSymmetric[{p0, sylv, left, middle, right}]` takes the output of `NCQuadratic` and produces, if possible, an equivalent symmetric representation in which `Map[tp, left] = right` and `middle` is a symmetric matrix.

See also: NCQuadratic.

14.3 NCMatrixOfQuadratic

`NCMatrixOfQuadratic[p, vars]` gives a factorization of the symmetric quadratic function `p` in noncommutative variables `vars` and their transposes.

`NCMatrixOfQuadratic` checks for symmetry and automatically sets variables to be symmetric if possible.

Internally it uses NCQuadratic and NCQuadraticMakeSymmetric.

See also: NCQuadratic, NCQuadraticMakeSymmetric.

14.4 NCQuadraticToNCPolynomial

`NCQuadraticToNCPolynomial[rep]` takes the list `rep` produced by `NCQuadratic` and converts it back to an `NCPolynomial`.

`NCQuadraticToNCPolynomial[rep,options]` uses options.

The following options can be given:

- `Collect` (*True*): controls whether the coefficients of the resulting `NCPolynomial` are collected to produce the minimal possible number of terms.

See also: NCQuadratic, NCPolynomial.

15 NCConvexity

NCConvexity is a package that provides functionality to determine whether a rational or polynomial noncommutative function is convex.

Members are:

- NCConvexityRegion

15.1 NCConvexityRegion

`NCConvexityRegion` is a function which can be used to determine whether a noncommutative function is convex or not.

See also: NCMatrixOfQuadratics.

16 NCRealization

The package **NCRealization** implements an algorithm due to N. Slinglend for producing minimal realizations of nc rational functions in many nc variables. See "Toward Making LMIs Automatically".

It actually computes formulas similar to those used in the paper "Noncommutative Convexity Arises From Linear Matrix Inequalities" by J William Helton, Scott A. McCullough, and Victor Vinnikov. In particular, there are functions for calculating (symmetric) minimal descriptor realizations of nc (symmetric) rational functions, and determinantal representations of polynomials.

Members are:

- Drivers:
- NCDescriptorRealization
- NCMatrixDescriptorRealization
- NCMinimalDescriptorRealization
- NCSymmetrizeMinimalDescriptorRealization
- NCSymmetricDescriptorRealization
- NCSymmetricDeterminantalRepresentationDirect
- NCDeterminantalRepresentationReciprocal
- NCSymmetricDeterminantalRepresentationReciprocal
- Auxiliary:
- RJRTDecomposition
- NonCommutativeLift
- PinnedQ
- PinningSpace
- TestDescriptorRealization
- Other (Mauricio think should be private)
- BlockDiagonalMatrix
- CGBMatrixToBigCGB
- CGBToPencil
- FloatingPointPrecision
- MatMultFromLeft
- MatMultFromRight
- NCFindPencil
- NCFormControllabilityColumns
- NCFormLettersFromPencil
- NCLinearPart
- NCLinearQ
- NCListToPencil
- NCMakeMonic
- NCNonLinearPart

- NCPencilToList
- ReturnWordList
- SignatureOfAffineTerm
- UseFloatingPoint

16.1 NCDescriptorRealization

`NCDescriptorRealization[RationalExpression,UnknownVariables]` returns a list of 3 matrices `{C,G,B}` such that $CG^{-1}B$ is the given `RationalExpression`. i.e. `MatMult[C,NCInverse[G],B] === RationalExpression`.

`C` and `B` do not contain any `UnknownsVariables` and `G` has linear entries in the `UnknownVariables`.

16.2 NCDeterminantalRepresentationReciprocal

`NCDeterminantalRepresentationReciprocal[Polynomial,Unknowns]` returns a linear pencil matrix whose determinant equals `Constant * CommuteEverything[Polynomial]`. This uses the reciprocal algorithm: find a minimal descriptor realization of `inv[Polynomial]`, so `Polynomial` must be nonzero at the origin.

16.3 NCMatrixDescriptorRealization

`NCMatrixDescriptorRealization[RationalMatrix,UnknownVariables]` is similar to `NCDescriptorRealization` except it takes a *Matrix* with rational function entries and returns a matrix of lists of the vectors/matrix `{C,G,B}`. A different `{C,G,B}` for each entry.

17 NCMinimalDescriptorRealization

`NCMinimalDescriptorRealization[RationalFunction,UnknownVariables]` returns `{C,G,B}` where `MatMult[C,NCInverse[G],B] == RationalFunction`, `G` is linear in the `UnknownVariables`, and the realization is minimal (may be pinned).

17.1 NCSymmetricDescriptorRealization

NCSymmetricDescriptorRealization[RationalSymmetricFunction, Unknowns] combines two steps: NCSymmetrizeMinimalDescriptorRealization[NCMinimalDescriptorRealiz Unknowns]].

17.2 NCSymmetricDeterminantalRepresentationDirect

NCSymmetricDeterminantalRepresentationDirect[SymmetricPolynomial,Unknowns] returns a linear pencil matrix whose determinant equals Constant * CommuteEverything[SymmetricPolynomial]. This uses the direct algorithm: Find a realization of 1 - NCSymmetricPolynomial,...

17.3 NCSymmetricDeterminantalRepresentationReciprocal

NCSymmetricDeterminantalRepresentationReciprocal[SymmetricPolynomial,Unknowns] returns a linear pencil matrix whose determinant equals Constant * CommuteEverything[NCSymmetricPolynomial]. This uses the reciprocal algorithm: find a symmetric minimal descriptor realization of inv[NCSymmetricPolynomial], so NCSymmetricPolynomial must be nonzero at the origin.

17.4 NCSymmetrizeMinimalDescriptorRealization

NCSymmetrizeMinimalDescriptorRealization[{C,G,B},Unknowns] symmetrizes the minimal realization {C,G,B} (such as output from NCMinimalRealization) and outputs {Ctilda,Gtilda} corresponding to the realization {Ctilda, Gtilda,Transpose[Ctilda]}.

WARNING: May produces errors if the realization doesn't correspond to a symmetric rational function.

17.5 BlockDiagonalMatrix

BlockDiagonalMatrix[ListOfMatrices] returns the block diagonal matrix with the matrices in ListOfMatrix on the diagonal. Each matrix in ListOfMatrices can be arbitrary size. i.e. the output matrix doesn't have to be square.

17.6 CGBMatrixToBigCGB

`CGBMatrixToBigCGB[MatrixOfCGB]` returns a list of 3 matrices `{C, G, B}` such that `NCMatMult[C, NCInverse[G], B]` is the original matrix that the `MatrixOfCGB` was derived from.

17.7 CGBToPencil

`CGBToPencil[CGB]` takes the list of 3 matrices returned by `NCDescriptorRealization` and returns a matrix with linear entries which has a Schur Complement equivalent to the rational expression that the CGB realization represents.

17.8 MatMultFromLeft

`MatMultFromLeft[A,B,C,...]` is the default of `MatMult`. If you want the matrix multiplications to start on the left. This is most efficient, for example, if the first matrix is a vector (1-by-n) and the rest are square matrices (n-by-n).

17.9 MatMultFromRight

`MatMultFromRight[A,B,C,...]`. It's often more efficient to perform multiplication of several matrices starting from the right. For example, if the last matrix is a vector (n-by-1) and the rest are square matrices (n-by-n).

17.10 NCFindPencil

`NCFindPencil[Expression,Unknowns]` returns a matrix with linear entries in the `Unknowns` (Linear Pencil) such that a Schur Complement of the matrix is the original Expression. Expression can be a rational function or a matrix with rational function entries.

17.11 NCFormControllabilityColumns

`NCFormControllabilityColumns[A_List,B_,opts___]`. Given the realization `MatMult[C, NCInverse[I-A], B]`, this returns a matrix such that the columns of its transpose span the controllability space.

With optional argument `ReturnWordList->False`, the output is `{Matrix,ListOfWords}` where `ListOfWords` is a list of the words used to make the spaning vectors. i.e. The output `ListOfWords == {{},{1},{3,1}}` would correspond to the vectors `{B, A[[1]].B, A[[3]].A[[1]].B}`

Optional argument `Verbose->True`, prints information as it's working.

17.12 NCFormLettersFromPencil

`NCFormLettersFromPencil[A_List,B_]`. Given a realization `C.A^(-1).B`, where `A = A0 + A1*x1 + A2*x2 +...+An*xn`, this returns the list `{ A0^(-1).A1, A0^(-1).A2,...,A0^(-1).An,A0^(-1).B}`. These are the letters that are used when finding the controllability and observability spaces.

17.13 NCLinearPart

`NCLinearPart[RationalExpression,UnknownVariables]` returns the part of `RationalExpression` that is linear in (a list of) `UnknownVariables`.

`RationalExpression` is NOT expanded, so in effect what gets returned is a sum of monomial terms each of which is linear. `NCLinearPart[(inv[x] + A) ** x, {x}] returns (inv[x] + A) ** x` which is actually linear (`NCLinearQ == True`). But, `NCLinearPart[(x + inv[x]) ** x, {x}]` returns `0` since `(x + inv[x]) ** x` is not ENTIRELY linear. `NCLinearPart + NCNonLinearPart == RationalExpression`.

17.14 NCLinearQ

`NCLinearQ[RationalExpression, UnknownVariables]` returns `True` if `RationalExpression` is linear in (a list of) `UnknownVariables`, `False` otherwise. `NCLinearQ` expands expressions using `NCExpand` first, then determines linearity, so `(inv[x]+A)**x` is actually linear in `x`.

17.15 NCListToPencil

`NCListToPencil[ListOfMatrices,Unknowns]` creates a linear pencil.

For example, `NCListToPencil[{A0,A1,A2},{1,x,y}]` is `A0 + A1**x + A2**y`.

17.16 NCMakeMonic

`NCMakeMonic[{CC_,Pencil_,BB_},Unknowns_]` returns a descriptor realization `{C2,Pencil2,B2}` that is monic. For this to be possible, the realization must represent a rational function that's not zero at the origin.

17.17 NCNonLinearPart

`NCNonLinearPart[RationalExpression,UnknownVariables]` returns the part of `RationalExpression` that is not linear in (a list of) `UnknownVariables`. `RationalExpression` is NOT expanded, SO in effect what gets returned is a sum of monomial terms each of which is not linear (`NCLinearQ = False`). `NCNonLinearPart[(inv[x] + A) ** x, {x}]` returns 0 since `(inv[x] + A) ** x` is actually linear. `NCNonLinearPart[ y + (x + inv[x]) ** x, {x,y}]` returns `(x + inv[x]) ** x` since `(x + inv[x]) ** x` is nonlinear as a whole (but `y` isn't). `NCLinearPart + NCNonLinearPart == RationalExpression`.

17.18 NCPencilToList

`NCPencilToList[Pencil,Unknowns]` takes a matrix `Pencil` (linear in the `Unknowns`) and returns a list of matrices `{A0,A1,A2,...}` such that `Pencil == A0 + A1*Unknowns[[1]] + A2*Unkowns[[2]] + ...`

17.19 NCRealization

`NCRealization`...

17.20 NonCommutativeLift

`NonCommutativeLift[Rational]` returns a noncommutative symmetric lift of `Rational`.

17.21 PinnedQ

`PinnedQ[Pencil_,Unknowns_]` is True or False.

17.22 PinningSpace

`PinningSpace[Pencil_,Unknowns_]` returns a matrix whose columns span the pinning space of `Pencil`. Generally, either an empty matrix or a d-by-1 matrix (vector).

17.23 ReturnWordList

`ReturnWordList`

17.24 RJRTDecomposition

`RJRTDecomposition[SymmetricMatrix_,opts___]`. Returns `{R,J}` such that `SymmetricMatrix == R.J.Transpose[R]` and `J` is a signature matrix. Returns the answer in floating point unless the optional argument `UseFloatingPoint->False` is used. Floating point is necessary except for small examples because eigenvectors and eigenvalues are calculated in the algorithm.

17.25 SignatureOfAffineTerm

`SignatureOfAffineTerm[Pencil,Unknowns]` returns a list of the number of positive, negative and zero eigenvalues in the affine part of `Pencil`.

17.26 TestDescriptorRealization

`TestDescriptorRealization[Rat,{C,G,B},Unknowns]` checks if `Rat == C.G^(-1).B` by substituting random 2-by-2 matrices in for the unknowns. `TestDescriptorRealization[Rat,{C,G,B},Unknowns,NumberOfTests]` can be used to specify the `NumberOfTests`, the default being 5.

17.27 UseFloatingPoint

`UseFloatingPoint`

18 NCUtil

NCUtil is a package with a collection of utilities used throughout NCAlgebra.

Members are:

- NCConsistentQ
- NCGrabFunctions
- NCGrabSymbols

18.1 NCConsistentQ

`NCConsistentQ[expr]` returns *True* is `expr` contains no commutative products or inverses involving noncommutative variables.

18.2 NCGrabFunctions

`NCGragFunctions[expr,f]` returns a list with all fragments of `expr` containing the function `f`.

For example:

```
NCGrabFunctions[inv[x] + y**inv[1+inv[1+x**y]], inv]
```

returns

```
{inv[1+inv[1+x**y]], inv[1+x**y], inv[x]}
```

See also: NCGrabSymbols.

18.3 NCGrabSymbols

`NCGragSymbols[expr]` returns a list with all *Symbols* appearing in `expr`.

`NCGragSymbols[expr,f]` returns a list with all *Symbols* appearing in `expr` as the single argument of function `f`.

For example:

```
NCGrabSymbols[inv[x] + y**inv[1+inv[1+x**y]]]
```

returns `{x,y}` and

```
NCGrabSymbols[inv[x] + y**inv[1+inv[1+x**y]], inv]
```

returns `{inv[x]}`.

See also: NCGrabFunctions.

19 MatrixDecompositions

MatrixDecompositions is a package that implements various linear algebra algorithms, such as *LU Decomposition* with *partial* and *complete pivoting*, and *LDL Decomposition*. The algorithms have been written with correctness and easy of customization rather efficiency as the main goals. They were originally developed to serve as the noncommutative linear algebra algorithms for NCAlgebra.

Members are:

- Decompositions
  - LUDecompositionWithPartialPivoting
  - LUDecompositionWithCompletePivoting
  - LDLDecomposition
- Solvers

- LowerTriangularSolve
  - UpperTriangularSolve
  - LUInverse
- Utilities
  - GetLUMatrices
  - GetLDUMatrices
  - LUPartialPivoting
  - LUCompletePivoting

19.1 LUDecompositionWithPartialPivoting

`LUDecompositionWithPartialPivoting[m]` generates a representation of the LU decomposition of the rectangular matrix `m`.

`LUDecompositionWithPartialPivoting[m, options]` uses `options`.

`LUDecompositionWithPartialPivoting` returns a list of two elements:

- the first element is a combination of upper- and lower-triangular matrices;
- the second element is a vector specifying rows used for pivoting.

`LUDecompositionWithPartialPivoting` is similar in functionality with the built-in `LUDecomposition`. It implements a *partial pivoting* strategy in which the sorting can be configured using the options listed below. It also applies to general rectangular matrices as well as square matrices.

The following `options` can be given:

- `ZeroTest` (`PossibleZeroQ`): function used to decide if a pivot is zero;
- `Divide` (`Divide`): function used to divide a vector by an entry;
- `Dot` (`Dot`): function used to multiply vectors and matrices;
- `Pivoting` (`LUPartialPivoting`): function used to sort rows for pivoting;

See also: LUDecomposition, GetLUMatrices, LUPartialPivoting, LUDecompositionWithCompletePivoting.

19.2 LUDecompositionWithCompletePivoting

`LUDecompositionWithCompletePivoting[m]` generates a representation of the LU decomposition of the rectangular matrix `m`.

`LUDecompositionWithCompletePivoting[m, options]` uses `options`.

`LUDecompositionWithCompletePivoting` returns a list of four elements:

- the first element is a combination of upper- and lower-triangular matrices;
- the second element is a vector specifying rows used for pivoting.
- the third element is a vector specifying columns used for pivoting.
- the fourth element is the rank of the matrix.

`LUDecompositionWithCompletePivoting` implements a *complete pivoting* strategy in which the sorting can be configured using the options listed below. It also applies to general rectangular matrices as well as square matrices.

The following `options` can be given:

- `ZeroTest` (`PossibleZeroQ`): function used to decide if a pivot is zero;
- `Divide` (`Divide`): function used to divide a vector by an entry;
- `Dot` (`Dot`): function used to multiply vectors and matrices;
- `Pivoting` (`LUCompletePivoting`): function used to sort rows for pivoting;

See also: LUDecomposition, GetLUMatrices, LUCompletePivoting, LUDecompositionWithPartialPivoting.

19.3 LDLDecomposition

19.4 GetLUMatrices

19.5 GetLDUMatrices

19.6 UpperTriangularSolve

19.7 LowerTriangularSolve

19.8 LUInverse

19.9 LUPartialPivoting

19.10 LUCompletePivoting

20 NCSDP

NCSDP is a package that allows the symbolic manipulation and numeric solution of semidefinite programs.

Problems consist of symbolic noncommutative expressions representing inequalities and a list of rules for data replacement. For example the semidefinite program:

$$\begin{aligned} \min_{Y} \quad & < I, Y > \\ \text{s.t.} \quad & AY + YA^T + I \preceq 0 \\ & Y \succeq 0 \end{aligned}$$

can be solved by defining the noncommutative expressions

```
<< NCSDP`
SNC[a, y];
obj = {-1};
ineqs = {a ** y + y ** tp[a] + 1, -y};
```

The inequalities are stored in the list `ineqs` in the form of noncommutative linear polyonomials in the variable `y` and the objective function constains the symbolic coefficients of the inner product, in this case `-1`. The reason for the negative signs in the objective as well as in the second inequality is that semidefinite programs are expected to be cast in the following *canonical form*:

$$\begin{aligned} \max_{y} \quad & < b, y > \\ \text{s.t.} \quad & f(y) \preceq 0 \end{aligned}$$

or, equivalently:

$$\begin{aligned} \max_{y} \quad & < b, y > \\ \text{s.t.} \quad & f(y) + s = 0, \quad s \succeq 0 \end{aligned}$$

Semidefinite programs can be visualized using `NCSDPForm` as in:

```
vars = {y};
NCSDPForm[ineqs, vars, obj]
```

In order to obtaining a numerical solution to an instance of the above semidefinite program one must provide a list of rules for data substitution. For example:

```
A = {{0, 1}, {-1, -2}};
data = {a -> A};
```

Equipped with a list of rules one can invoke `NCSDP` to produce an instance of `SDPSylvester`:

```
<< SDPSylvester`
{abc, rules} = NCSDP[F, vars, obj, data];
```

It is the resulting `abc` and `rules` objects that are used for calculating the numerical solution using `SDPSolve`:

```
{Y, X, S, flags} = SDPSolve[abc, rules];
```

The variables `Y` and `S` are the *primal* solutions and `X` is the *dual* solution.

An explicit symbolic dual problem can be calculated easily using `NCSDPDual`:

```
{dIneqs, dVars, dObj} = NCSDPDual[ineqs, vars, obj];
```

The corresponding dual program is expressed in the *canonical form*:

$$\begin{aligned} \max_{x} \quad & < c, x > \\ \text{s.t.} \quad & f^*(x) + b = 0, \quad x \succeq 0 \end{aligned}$$

In the case of the above problem the dual program is

$$
\begin{aligned}
\max_{X_1,X_2} \quad & < I, X_1 > \\
\text{s.t.} \quad & A^T X_1 + X_1 A - X_2 - I = 0 \\
& X_1 \succeq 0, \\
& X_2 \succeq 0
\end{aligned}
$$

Dual semidefinite programs can be visualized using `NCSDPDualForm` as in:

```
NCSDPDualForm[dIneqs, dVars, dObj]
```

Members are:

- NCSDP
- NCSDPForm
- NCSDPDual
- NCSDPDualForm

20.1 NCSDP

`NCSDP[inequalities,vars,obj,data]` converts the list of NC polynomials and NC matrices of polynomials `inequalities` that are linear in the unknowns listed in `vars` into the semidefinite program with linear objective `obj`. The semidefinite program (SDP) should be given in the following canonical form:

```
max  <obj, vars>  s.t.  inequalities <= 0.
```

`NCSDP` uses the user supplied rules in `data` to set up the problem data.

`NCSDP[constraints,vars,data]` converts problem into a feasibility semidefinite program.

See also: NCSDPForm, NCSDPDual.

20.2 NCSDPForm

`NCSDPForm[[inequalities,vars,obj]` prints out a pretty formatted version of the SDP expressed by the list of NC polynomials and NC matrices of polynomials `inequalities` that are linear in the unknowns listed in `vars`.

See also: NCSDP, NCSDPDualForm.

20.3 NCSDPDual

`{dInequalities, dVars, dObj} = NCSDPDual[inequalities,vars,obj]` calculates the symbolic dual of the SDP expressed by the list of NC polynomials and NC matrices of polynomials `inequalities` that are linear in the unknowns listed in `vars` with linear objective `obj` into a dual semidefinite in the following canonical form:

```
max <dObj, dVars>  s.t.  dInequalities == 0,   dVars >= 0.
```

See also: NCSDPDualForm, NCSDP.

20.4 NCSDPDualForm

`NCSDPForm[[dInequalities,dVars,dObj]` prints out a pretty formatted version of the dual SDP expressed by the list of NC polynomials and NC matrices of polynomials `dInequalities` that are linear in the unknowns listed in `dVars` with linear objective `dObj`.

See also: NCSDPDual, NCSDPForm.

21 SDP

The package **SDP** provides a crude and highly inefficient way to define and solve semidefinite programs in standard form, that is vectorized. You do not need to load `NCAlgebra` if you just want to use the semidefinite program solver. But you still need to load `NC` as in:

```
<< NC`
<< SDP`
```

Semidefinite programs are optimization problems of the form:

$$\begin{aligned} \min_{y,S} \quad & b^T y \\ \text{s.t.} \quad & Ay + c = S \\ & S \succeq 0 \end{aligned}$$

where S is a symmetric positive semidefinite matrix.

For convenience, problems can be stated as:

$$\begin{aligned} \min_{y} \quad & \text{obj}(y), \\ \text{s.t.} \quad & \text{ineqs}(y) >= 0 \end{aligned}$$

where $obj(y)$ and $ineqs(y)$ are affine functions of the vector variable y.

Here is a simple example:

```
ineqs = {y0 - 2, {{y1, y0}, {y0, 1}}, {{y2, y1}, {y1, 1}}};
obj = y2;
y = {y0, y1, y2};
```

The list of constraints in `ineqs` are to be interpreted as:

$$y_0 - 2 \geq 0,$$
$$\begin{bmatrix} y_1 & y_0 \\ y_0 & 1 \end{bmatrix} \succeq 0,$$
$$\begin{bmatrix} y_2 & y_1 \\ y_1 & 1 \end{bmatrix} \succeq 0.$$

The function `SDPMatrices` convert the above symbolic problem into numerical data that can be used to solve an SDP.

```
abc = SDPMatrices[by, ineqs, y]
```

All required data, that is A, b, and c, is stored in the variable `abc` as Mathematica's sparse matrices. Their contents can be revealed using the Mathematica command `Normal`.

```
Normal[abc]
```

The resulting SDP is solved using `SDPSolve`:

```
{Y, X, S, flags} = SDPSolve[abc];
```

The variables `Y` and `S` are the *primal* solutions and `X` is the *dual* solution. Detailed information on the computed solution is found in the variable `flags`.

The package **SDP** is built so as to be easily overloaded with more efficient or more structure functions. See for example SDPFlat and SDPSylvester.

Members are:

- SDPMatrices
- SDPSolve
- SDPEval
- SDPInner

The following members are not supposed to be called directly by users:

- SDPCheckDimensions
- SDPScale
- SDPFunctions
- SDPPrimalEval
- SDPDualEval
- SDPSylvesterEval
- SDPSylvesterDiagonalEval

21.1 SDPMatrices

21.2 SDPSolve

21.3 SDPEval

21.4 SDPInner

21.5 SDPCheckDimensions

21.6 SDPDualEval

21.7 SDPFunctions

21.8 SDPPrimalEval

21.9 SDPScale

21.10 SDPSylvesterDiagonalEval

21.11 SDPSylvesterEval