The NCAlgebra Suite

John W. Helton   Mauricio C. de Oliveira

September, 2016

Contents

Part I

User Guide

Chapter 1

Changes in Version 5.0

1. Completely rewritten core handling of noncommutative expressions with significant speed gains.
2. Commands `Transform`, `Substitute`, `SubstituteSymmetric`, etc, have been replaced by the much more reliable commands in the new package NCReplace.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.
8. New algorithms for simplification of noncommutative rationals (NCSimplifyRational).

Chapter 2

Introduction

This *User Guide* attempts to document the many improvements introduced in `NCAlgebra` Version 5.0. Please be patient, as we move to incorporate the many recent changes into this document.

See Reference Manual for a detailed description of the available commands.

2.1 Running NCAlgebra

In *Mathematica* (notebook or text interface), type

```
<< NC`
```

If this step fails, your installation has problems (check out installation instructions on the main page). If your installation is succesful you will see a message like:

```
You are using the version of NCAlgebra which is found in:
   /your_home_directory/NC.
You can now use "<< NCAlgebra`" to load NCAlgebra or "<< NCGB`" to load NCGB.
```

Just type

```
<< NCAlgebra`
```

to load `NCAlgebra`, or

```
<< NCGB`
```

to load `NCAlgebra` *and* `NCGB`.

2.2 Now what?

Basic documentation is found in the project wiki:

https://github.com/NCAlgebra/NC/wiki

Extensive documentation is found in the directory `DOCUMENTATION`.

You may want to try some of the several demo files in the directory `DEMOS` after installing `NCAlgebra`.

You can also run some tests to see if things are working fine.

2.3 Testing

Type

```
<< NCTEST
```

to test NCAlgebra. Type

```
<< NCGBTEST
```

to test NCGB.

We recommend that you restart the kernel before and after running tests. Each test takes a few minutes to run.

Chapter 3

Most Basic Commands

First you must load in NCAlgebra with the following command

```
In[1]:= <<NC`
In[2]:= <<NCAlgebra`
```

3.1 To Commute Or Not To Commute?

In `NCAlgebra`, the operator `**` denotes *noncommutative multiplication*.

At present, single-letter lower case variables are non-commutative by default and all others are commutative by default.

We consider non-commutative lower case variables in the following examples:

```
In[3]:= a**b-b**a
Out[3]= a**b-b**a
In[4]:= A**B-B**A
Out[4]= 0
In[5]:= A**b-b**A
Out[5]= 0
```

`CommuteEverything` temporarily makes all noncommutative symbols appearing in a given expression to behave as if they were commutative and returns the resulting commutative expression:

```
In[6]:= CommuteEverything[a**b-b**a]
Out[6]= 0
In[7]:= EndCommuteEverything[]
In[8]:= a**b-b**a
Out[8]= a**b-b**a
```

`EndCommuteEverything` restores the original noncommutative behavior.

`SetNonCommutative` makes symbols behave permanently as noncommutative:

```
In[9]:= SetNonCommutative[A,B]
In[10]:= A**B-B**A
Out[10]= A**B-B**A
In[11]:= SetNonCommutative[A]; SetCommutative[B];
In[12]:= A**B-B**A
Out[12]= 0
```

`SNC` is an alias for `SetNonCommutative`. So, `SNC` can be typed rather than the longer `SetNonCommutative`.

```
In[13]:= SNC[A];
In[14]:= A**a-a**A
Out[14]= -a**A+A**a
```

`SetCommutative` makes symbols permanently behave as commutative:

```
In[15]:= SetCommutative[v];
In[16]:= v**b
Out[16]= b v
```

3.2 Transposes and Adjoints

`tp[x]` denotes the transpose of symbol `x`

`aj[x]` denotes the adjoint of symbol `x`

The properties of transposes and adjoints that everyone uses constantly are built-in:

```
In[17]:= tp[a**b]
Out[17]= tp[b]**tp[a]
In[18]:= tp[5]
Out[18]= 5
In[19]:= tp[2+3I]   (* I is the imaginary unit *)
Out[19]= 2+3 I
In[20]:= tp[a]
Out[20]= tp[a]
In[21]:= tp[a+b]
Out[21]= tp[a]+tp[b]
In[22]:= tp[6x]
Out[22]= 6 tp[x]
In[23]:= tp[tp[a]]
Out[23]= a
In[24]:= aj[5]
Out[24]= 5
In[25]:= aj[2+3I]
Out[25]= 2-3 I
In[26]:= aj[a]
Out[26]= aj[a]
In[27]:= aj[a+b]
Out[27]= aj[a]+aj[b]
In[28]:= aj[6x]
Out[28]= 6 aj[x]
In[29]:= aj[aj[a]]
Out[29]= a
```

3.3 Inverses

The multiplicative identity is denoted `Id` in the program. At the present time, `Id` is set to 1.

A symbol `a` may have an inverse, which will be denoted by `inv[a]`.

```
In[30]:= Id
Out[30]= 1
```

```
In[31]:= inv[a**b]
Out[31]= inv[a**b]
In[32]:= inv[a]**a
Out[32]= 1
In[33]:= a**inv[a]
Out[33]= 1
In[34]:= a**b**inv[b]
Out[34]= a
```

3.4 Expand and Collect

One can collect noncommutative terms involving same powers of a symbol using `NCCollect`. `NCExpand` expand noncommutative products.

```
In[35]:= NCExpand[(a+b)**x]
Out[35]= a**x+b**x
In[36]:= NCCollect[a**x+b**x,x]
Out[36]= (a+b)**x
In[37]:= NCCollect[tp[x]**a**x+tp[x]**b**x+z,{x,tp[x]}]
Out[37]= z+tp[x]**(a+b)**x
```

3.5 Replace

The Mathematica substitute commands, e.g. `Replace`, `ReplaceAll` (`/.`) and `ReplaceRepeated` (`//.`), are not reliable in `NCAlgebra`, so you must use our `NC` versions of these commands:

```
In[38]:= NCReplace[x**a**b,a**b->c]
Out[38]= x**a**b
In[39]:= NCReplaceAll[tp[b**a]+b**a,b**a->p]
Out[39]= p+tp[a]**tp[b]
```

USe NCMakeRuleSymmetric and NCMakeRuleSelfAdjoint to automatically create symmetric and self adjoint versions of your rules:

```
In[40]:= NCReplaceAll[tp[a**b]+w+a**b,a**b->c]
Out[40]= c+w+tp[b]**tp[a]
In[41]:= NCReplaceAll[tp[a**b]+w+a**b,NCMakeRuleSymmetric[a**b->c]]
Out[41]= c+w+tp[c]
```

3.6 Rationals and Simplification

`NCSimplifyRational` attempts to simplify noncommutative rationals.

```
In[42]:= f1=1+inv[d]**c**inv[S-a]**b-inv[d]**c**inv[S-a+b**inv[d]**c]**b\
         -inv[d]**c**inv[S-a+b**inv[d]**c]**b**inv[d]**c**inv[S-a]**b
Out[42]= 1+inv[d]**c**inv[-a+S]**b-inv[d]**c**inv[-a+S+b**inv[d]**c]**b\
         -inv[d]**c**inv[-a+S+b**inv[d]**c]**b**inv[d]**c**inv[-a+S]**b
In[43]:= NCSimplifyRational[f1]
Out[43]= 1
In[44]:= f2=2inv[1+2a]**a;
In[45]:= NCSimplifyRational[f2]
Out[45]= 1-inv[1+2 a]
```

NCSR is the alias for NCSimplifyRational.

```
In[46]:= f3=a**inv[1-a];
In[47]:= NCSR[f3]
Out[47]= -1+inv[1-a]
In[48]:= f4=inv[1-b**a]**inv[a];
In[49]:= NCSR[f4]
Out[49]= inv[a]+b**inv[1-b**a]
```

3.7 Calculus

One can calculate directional derivatives with DirectionalD and noncommutative gradients with NCGrad.

```
In[50]:= DirectionalD[x**x,x,h]
Out[50]= h**x+x**h
In[51]:= NCGrad[tp[x]**x+tp[x]**A**x+m**x,x]
Out[51]= m+tp[x]**A+tp[x]**tp[A]+2 tp[x]
```

3.8 Matrices

NCAlgebra has many algorithms that handle matrices with noncommutative entries.

```
In[52]:= m1={{a,b},{c,d}}
Out[52]= {{a,b},{c,d}}
In[53]:= m2={{d,2},{e,3}}
Out[53]= {{d,2},{e,3}}
In[54]:= MatMult[m1,m2]
Out[54]= {{a**d+b**e,2 a+3 b},{c**d+d**e,2 c+3 d}}
```

Chapter 4

Things you can do with NCAlgebra and NCGB

In this page you will find some things that you can do with `NCAlgebra` and `NCGB`.

4.1 Noncommutative Inequalities

Is a given noncommutative function *convex*? You type in a function of noncommutative variables; the command `NCConvexityRegion[Function, ListOfVariables]` tells you where the (symbolic) `Function` is *convex* in the Variables. This corresponds to papers of *Camino, Helton and Skelton*.

4.2 Linear Systems and Control

`NCAlgebra` integrates with *Mathematica*'s control toolbox (version 8.0 and above) to work on noncommutative block systems, just as a human would do...

Look for NCControl.nb in the `NC/DEMOS` subdirectory.

4.3 Semidefinite Programming

`NCAlgebra` now comes with a numerical solver that can compute the solution to semidefinite programs, aka linear matrix inequalities.

Look for demos in the `NC/NCSDP/DEMOS` subdirectory.

You can also find examples of systems and control linear matrix inequalities problems being manipulated and numerically solved by NCAlgebra on the UCSD course webpage.

Look for the .nb files, starting with the file sat5.nb at Lecture 8.

4.4 NonCommutative Groebner Bases

`NCGB` Computes NonCommutative Groebner Bases and has extensive sorting and display features and algorithms for automatically discarding *redundant* polynomials, as well as *kludgy* methods for suggesting changes of variables (which work better than one would expect).

`NCGB` runs in conjunction with `NCAlgebra`.

4.5 Groups

You can compute a complete list of rewrite rules for Groups using `NCGB`. See demos at http://math.ucsd.edu/~ncalg.

4.6 NCGBX

`NCGBX` is a 100% Mathematica version of our NC Groebner Basis Algorithm and does not require C/C++ code compilation.

Look for demos in the `NC/NCPoly/DEMOS` subdirectory of the most current distributions.

IMPORTANT: Do not load NCGB and NCGBX simultaneously.

Part II

Reference Manual

Chapter 5

Introduction

Each following chapter describes a `Package` inside *NCAlgebra*.

Packages are automatically loaded unless otherwise noted.

Chapter 6

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

6.1 aj

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

See also: tp, co.

6.2 co

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

See also: aj.

6.3 Id

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

6.4 inv

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

6.5 rt

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

6.6 tp

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

See also: aj, co.

6.7 CommutativeQ

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

See also: SetCommutative, SetNonCommutative.

6.8 NonCommutativeQ

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

See also: CommutativeQ.

6.9 SetCommutative

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

See also: SetNonCommutative, CommutativeQ, NonCommutativeQ.

6.10 SetNonCommutative

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

See also: SetCommutative, CommutativeQ, NonCommutativeQ.

6.11 Commutative

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

See also: CommuteEverything, CommutativeQ, SetCommutative, SetNonCommutative.

6.12 CommuteEverything

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

See also: BeginCommuteEverything, Commutative.

6.13 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.

6.14 EndCommuteEverything

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

See also: BeginCommuteEverything, Commutative.

6.15 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`.

Chapter 7

NCCollect

Members are:

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

7.1 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.

7.1.1 Notes

While `NCCollect[expr, vars]` always returns mathematically correct expressions, it may not collect `vars` from as many terms as one might think it should.

See also: NCStrongCollect, NCCollectSymmetric, NCCollectSelfAdjoint, NCStrongCollectSymmetric, NC-StrongCollectSelfAdjoint, NCRationalToNCPolynomial.

7.2 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, NCStrongCollectSymmetric, NCStrongCollectSelfAdjoint.

7.3 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, NCStrongCollectSymmetric, NCStrongCollectSelfAdjoint.

7.4 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.

7.5 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.

7.6 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.

7.7 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.

7.8 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.

7.9 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.

Chapter 8

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

8.1 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.

8.2 NCSimplifyRational

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

See also: NCNormalizeInverse, NCSimplifyRationalSinglePass.

8.3 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.

8.4 NCPreSimplifyRational

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

See also: NCNormalizeInverse, NCPreSimplifyRationalSinglePass.

8.5 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.

Chapter 9

NCDiff

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

Members are:

- NCDirectionalD
- NCGrad
- NCHessian
- NCIntegrate

Members being deprecated:

- DirectionalD

9.1 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`

In the case of more than one variables `NCDirectionalD[expr, {x,h}, {y,k}]` takes the directional derivative of `expr` with respect to `x` in the direction `h` and with respect to `y` in the direction `k`.

See also: NCGrad, NCHessian.

9.2 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.

9.3 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
```

In the case of more than one variables `NCHessian[expr, {x,h}, {y,k}]` takes the second directional derivative of `expr` with respect to `x` in the direction `h` and with respect to `y` in the direction `k`.

See also: NCDiretionalD, NCGrad.

9.4 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.

9.5 NCIntegrate

`NCIntegrate[expr,{var1,h1},...]` attempts to calculate the nc antiderivative of nc expression `expr` with respect to the single variable `var` in direction `h`.

For example:

```
NCIntegrate[x**h+h**x, {x,h}]
```

returns

```
x**x
```

See also: NCDirectionalD.

Chapter 10

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

10.1 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.

10.2 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.

10.3 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.

10.4 NCReplaceRepeated

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

See also: NCReplace, NCReplaceAll, NCReplaceList.

10.5 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.

10.6 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.

Chapter 11

NCSelfAdjoint

Members are:

- NCSymmetricQ
- NCSymmetricTest
- NCSymmetricPart
- NCSelfAdjointQ
- NCSelfAdjointTest

11.1 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.

11.2 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;
- `Strict`: treats as non-symmetric any variable that appears inside `tp`.

See also: NCSymmetricQ, NCNCSelfAdjointTest.

11.3 NCSymmetricPart

`NCSymmetricPart[expr]` returns the *symmetric part* of `expr`.

`NCSymmetricPart[exp,options]` uses `options`.

`NCSymmetricPart[expr]` returns a list of two elements:

- the first element is the *symmetric part* of `expr`;
- the second element is a list of the variables that were made symmetric.

`NCSymmetricPart[expr]` returns `{$Failed, {}}` if `expr` is not symmetric.

For example:

```
{answer, symVars} = NCSymmetricPart[a ** x + x ** tp[a] + 1];
```

returns

```
answer = 2 a ** x + 1
symVars = {x}
```

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.
- `Strict`: treats as non-symmetric any variable that appears inside `tp`.

See also: NCSymmetricTest.

11.4 NCSelfAdjointQ

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

See also: NCSymmetricQ, NCSelfAdjointTest.

11.5 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.
- `Strict`: treats as non-self-adjoint any variable that appears inside `aj`.

See also: NCSelfAdjointQ.

Chapter 12

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

12.1 NCOutputFunction

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

See also: NCSetOutput.

12.2 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.

Chapter 13

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

13.1 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`.

13.2 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`.

13.3 NCPolynomialToNC

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

See also: `NCPolynomial`, `NCToNCPolynomial`.

13.4 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`.

13.5 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`.

13.6 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`.

13.7 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`.

13.8 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`.

13.9 NCPPlus

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

13.10 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.

13.11 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.

13.12 NCPDegree

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

See also: `NCPMonomialDegree`.

13.13 NCPMonomialDegree

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

See also: `NCDegree`.

13.14 NCPLinearQ

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

See also: `NCPQuadraticQ`.

13.15 NCPQuadraticQ

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

See also: `NCPLinearQ`.

13.16 NCPCompatibleQ

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

See also: NCPSameVariablesQ, NCPMatrixQ.

13.17 NCPSameVariablesQ

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

See also: NCPCompatibleQ, NCPMatrixQ.

13.18 NCPMatrixQ

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

See also: NCPCompatibleQ.

13.19 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`

Chapter 14

NCSylvester

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

Members are:

- NCPolynomialToNCSylvester
- NCSylvesterToNCPolynomial

14.1 NCPolynomialToNCSylvester

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

`NCPolynomialToNCSylvester` 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} = NCPolynomialToNCSylvester[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.

14.2 NCSylvesterToNCPolynomial

`NCSylvesterToNCPolynomial[rep]` takes the list `rep` produced by `NCPolynomialToNCSylvester` 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: NCPolynomialToNCSylvester, NCPolynomial.

Chapter 15

NCQuadratic

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

Members are:

- NCQuadraticMakeSymmetric
- NCMatrixOfQuadratic
- NCQuadratic
- NCQuadraticToNCPolynomial

15.1 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.

15.2 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.

15.3 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.

It returns a list of three elements:

- the first is the left border row vector;
- the second is the middle matrix;
- the third is the right border column vector.

For example:

```
expr = x**y**x + z**x**x**z;
{left,middle,right}=NCMatrixOfQuadratics[expr, {x}];
```

returns:

```
left={x, z**x}
middle=SparseArray[{{y,0},{0,1}}]
right={x,x**z}
```

The answer from `NCMatrixOfQuadratics` always satisfies `p = MatMult[left,middle,right]`.

See also: NCQuadratic, NCQuadraticMakeSymmetric.

15.4 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.

Chapter 16

NCConvexity

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

Members are:

- NCIndependent
- NCConvexityRegion

16.1 NCIndependent

`NCIndependent[list]` attempts to determine whether the nc entries of `list` are independent.

Entries of `NCIndependent` can be nc polynomials or nc rationals.

For example:

```
NCIndependent[{x,y,z}]
```

return *True* while

```
NCIndependent[{x,0,z}]
NCIndependent[{x,y,x}]
NCIndependent[{x,y,x+y}]
NCIndependent[{x,y,A x + B y}]
NCIndependent[{inv[1+x]**inv[x], inv[x], inv[1+x]}]
```

all return *False*.

See also: NCConvexity.

16.2 NCConvexityRegion

`NCConvexityRegion[expr,vars]` is a function which can be used to determine whether the nc rational `expr` is convex in `vars` or not.

For example:

```
d = NCConvexityRegion[x**x**x, {x}];
```

returns

```
d = {2 x, -2 inv[x]}
```

from which we conclude that `x**x**x` is not convex in `x` because $x \succ 0$ and $-x^{-1} \succ 0$ cannot simultaneously hold.

`NCConvexityRegion` works by factoring the `NCHessian`, essentially calling:

```
hes = NCHessian[expr, {x, h}];
```

then

```
{lt, mq, rt} = NCMatrixOfQuadratic[hes, {h}]
```

to decompose the Hessian into a product of a left row vector, `lt`, times a middle matrix, `mq`, times a right column vector, `rt`. The middle matrix, `mq`, is factored using the `NCLDLDecomposition`:

```
{ldl, p, s, rank} = NCLDLDecomposition[mq];
{lf, d, rt} = GetLDUMatrices[ldl, s];
```

from which the output of NCConvexityRegion is the a list with the block-diagonal entries of the matrix `d`.

See also: NCHessian, NCMatrixOfQuadratic, NCLDLDecomposition.

Chapter 17

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
  - NCDeterminantalRepresentationReciprocal
  - NCSymmetrizeMinimalDescriptorRealization
  - NCSymmetricDescriptorRealization
  - NCSymmetricDeterminantalRepresentationDirect
  - NCSymmetricDeterminantalRepresentationReciprocal
  - NonCommutativeLift
- Auxiliary:
  - PinnedQ
  - PinningSpace
  - TestDescriptorRealization
  - SignatureOfAffineTerm

17.1 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`.

17.2 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.

17.3 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.

17.4 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).

17.5 NCSymmetricDescriptorRealization

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

17.6 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,...

17.7 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.

17.8 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.

17.9 NonCommutativeLift

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

17.10 SignatureOfAffineTerm

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

17.11 TestDescriptorRealization

`TestDescriptorRealization[Rat,{C,G,B},Unknowns]` checks if `Rat` equals $CG^{-1}B$ 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.

17.12 PinnedQ

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

17.13 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).

Chapter 18

NCMatMult

Members are:

- tpMat
- ajMat
- coMat
- MatMult
- NCInverse
- NCMatrixExpand

18.1 tpMat

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

See also: ajMat, coMat, MatMult.

18.2 ajMat

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

See also: tpMat, coMat, MatMult.

18.3 coMat

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

See also: tpMat, ajMat, MatMult.

18.4 MatMult

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

See also: tpMat, ajMat, coMat.

18.4.1 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*.

18.5 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.

18.6 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.

Chapter 19

NCMatrixDecompositions

Members are:

- Decompositions
  - NCLUDecompositionWithPartialPivoting
  - NCLUDecompositionWithCompletePivoting
  - NCLDLDecomposition
- Solvers
  - NCLowerTriangularSolve
  - NCUpperTriangularSolve
  - NCLUInverse
- Utilities
  - NCLUCompletePivoting
  - NCLUPartialPivoting
  - NCLeftDivide
  - NCRightDivide

19.1 NCLDLDecomposition

19.2 NCLeftDivide

19.3 NCLowerTriangularSolve

19.4 NCLUCompletePivoting

19.5 NCLUDecompositionWithCompletePivoting

19.6 NCLUDecompositionWithPartialPivoting

19.7 NCLUInverse

19.8 NCLUPartialPivoting

19.9 NCMatrixDecompositions

19.10 NCRightDivide

19.11 NCUpperTriangularSolve

Chapter 20

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 than efficiency as the main goals. They were originally developed to serve as the core of the noncommutative linear algebra algorithms for NCAlgebra. See NCMatrixDecompositions.

Members are:

- Decompositions
  - LUDecompositionWithPartialPivoting
  - LUDecompositionWithCompletePivoting
  - LDLDecomposition
- Solvers
  - LowerTriangularSolve
  - UpperTriangularSolve
  - LUInverse
- Utilities
  - GetLUMatrices
  - GetLDUMatrices
  - GetDiagonal
  - LUPartialPivoting
  - LUCompletePivoting
  - LUNoPartialPivoting
  - LUNoCompletePivoting

20.1 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 triangular factors are recovered using GetLUMatrices.

The following `options` can be given:

- `ZeroTest` (`PossibleZeroQ`): function used to decide if a pivot is zero;
- `RightDivide` (`RightDivide`): 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;
- `SuppressPivoting` (`False`): whether to perform pivoting or not.

See also: LUDecompositionWithPartialPivoting, LUDecompositionWithCompletePivoting, GetLUMatrices, LUPartialPivoting.

20.2 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 triangular factors are recovered using GetLUMatrices.

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.

20.3 LDLDecomposition

`LDLDecomposition[m]` generates a representation of the LDL decomposition of the symmetric or self-adjoint matrix `m`.

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

`LDLDecomposition` 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 and columns used for pivoting;
- the thir element is a vector specifying the size of the diagonal blocks; it can be 1 or 2;
- the fourth element is the rank of the matrix.

`LUDecompositionWithCompletePivoting` implements a *Bunch-Parlett pivoting* strategy in which the sorting can be configured using the options listed below. It applies only to square symmetric or self-adjoint matrices.

The triangular factors are recovered using GetLDUMatrices.

The following `options` can be given:

- `ZeroTest` (`PossibleZeroQ`): function used to decide if a pivot is zero;
- `RightDivide` (`RightDivide`): function used to divide a vector by an entry on the right;
- `LeftDivide` (`LeftDivide`): function used to divide a vector by an entry on the left;
- `Dot` (`Dot`): function used to multiply vectors and matrices;
- `CompletePivoting` (`LUCompletePivoting`): function used to sort rows for complete pivoting;
- `PartialPivoting` (`LUPartialPivoting`): function used to sort matrices for complete pivoting;
- `Inverse` (`Inverse`): function used to invert 2x2 diagonal blocks;
- `SelfAdjointQ` (`SelfAdjointMatrixQ`): function to test if matrix is self-adjoint;
- `SuppressPivoting` (`False`): whether to perform pivoting or not.

See also: LUDecompositionWithPartialPivoting, LUDecompositionWithCompletePivoting, GetLUMatrices, LUCompletePivoting, LUPartialPivoting.

20.4 UpperTriangularSolve

`UpperTriangularSolve[u, b]` solves the upper-triangular system of equations $ux = b$ using back-substitution.

For example:

```
x = UpperTriangularSolve[u, b];
```

returns the solution `x`.

See also: LUDecompositionWithPartialPivoting, LUDecompositionWithCompletePivoting, LDLDecomposition.

20.5 LowerTriangularSolve

`LowerTriangularSolve[l, b]` solves the lower-triangular system of equations $lx = b$ using forward-substitution.

For example:

```
x = LowerTriangularSolve[l, b];
```

returns the solution `x`.

See also: LUDecompositionWithPartialPivoting, LUDecompositionWithCompletePivoting, LDLDecomposition.

20.6 LUInverse

`LUInverse[a]` calculates the inverse of matrix `a`.

`LUInverse` uses the LuDecompositionWithPartialPivoting and the triangular solvers LowerTriangularSolve and UpperTriangularSolve.

See also: LUDecompositionWithPartialPivoting.

20.7 GetLUMatrices

`GetLUMatrices[m]` extracts lower- and upper-triangular blocks produced by `LDUDecompositionWithPartialPivoting` and `LDUDecompositionWithCompletePivoting`.

For example:

```
{lu, p} = LUDecompositionWithPartialPivoting[A];
{l, u} = GetLUMatrices[lu];
```

returns the lower-triangular factor `l` and upper-triangular factor `u`.

See also: LUDecompositionWithPartialPivoting, LUDecompositionWithCompletePivoting.

20.8 GetLDUMatrices

`GetLDUMatrices[m,s]` extracts lower-, upper-triangular and diagonal blocks produced by `LDLDecomposition`.

For example:

```
{ldl, p, s, rank} = LDLDecomposition[A];
{l,d,u} = GetLDUMatrices[ldl,s];
```

returns the lower-triangular factor `l`, the upper-triangular factor `u`, and the block-diagonal factor `d`.

See also: LDLDecomposition.

20.9 GetDiagonal

`GetDiagonal[m]` extracts the diagonal entries of matrix `m`.

`GetDiagonal[m, s]` extracts the block-diagonal entries of matrix `m` with block size `s`.

For example:

```
d = GetDiagonal[{{1,-1,0},{-1,2,0},{0,0,3}}];
```

returns

```
d = {1,2,3}
```

and

```
d = GetDiagonal[{{1,-1,0},{-1,2,0},{0,0,3}}, {2,1}];
```

returns

```
d = {{{1,-1},{-1,2}},3}
```

See also: LDLDecomposition.

20.10 LUPartialPivoting

`LUPartialPivoting[v]` returns the index of the element with largest absolute value in the vector `v`. If `v` is a matrix, it returns the index of the element with largest absolute value in the first column.

`LUPartialPivoting[v, f]` sorts with respect to the function `f` instead of the absolute value.

See also: LUDecompositionWithPartialPivoting, LUCompletePivoting.

20.11 LUCompletePivoting

`LUCompletePivoting[m]` returns the row and column index of the element with largest absolute value in the matrix `m`.

`LUCompletePivoting[v, f]` sorts with respect to the function `f` instead of the absolute value.

See also: LUDecompositionWithCompletePivoting, LUPartialPivoting.

Chapter 21

NCUtil

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

Members are:

- NCConsistentQ
- NCGrabFunctions
- NCGrabSymbols
- NCGrabIndeterminants
- NCConsolidateList
- NCLeafCount
- NCReplaceData
- NCToExpression

21.1 NCConsistentQ

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

21.2 NCGrabFunctions

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

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

For example:

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

returns

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

and

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

returns

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

See also: NCGrabSymbols.

21.3 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.

21.4 NCGrabIndeterminants

`NCGragIndeterminants[expr]` returns a list with first level symbols and nc expressions involved in sums and nc products in `expr`.

For example:

```
NCGrabIndeterminants[y - inv[x] + tp[y]**inv[1+inv[1+tp[x]**y]]]
```

returns

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

See also: NCGrabFunctions, NCGrabSymbols.

21.5 NCConsolidateList

`NCConsolidateList[list]` produces two lists:

- The first list contains a version of `list` where repeated entries have been suppressed;
- The second list contains the indices of the elements in the first list that recover the original `list`.

For example:

```
{list,index} = NCConsolidateList[{z,t,s,f,d,f,z}];
```

results in:

```
list = {z,t,s,f,d};
index = {1,2,3,4,5,4,1};
```

See also: `Union`

21.6 NCLeafCount

`NCLeafCount[expr]` returns an number associated with the complexity of an expression:

- If `PossibleZeroQ[expr] == True` then `NCLeafCount[expr]` is `-Infinity`;
- If `NumberQ[expr]] == True` then `NCLeafCount[expr]` is `Abs[expr]`;

- Otherwise NCLeafCount[expr] is -LeafCount[expr];

NCLeafCount is Listable.

See also: LeafCount.

21.7 NCReplaceData

NCReplaceData[expr, rules] applies rules to expr and convert resulting expression to standard Mathematica, for example replacing ** by ..

NCReplaceData does not attempt to resize entries in expressions involving matrices. Use NCToExpression for that.

See also: NCToExpression.

21.8 NCToExpression

NCToExpression[expr, rules] applies rules to expr and convert resulting expression to standard Mathematica.

NCToExpression attempts to resize entries in expressions involving matrices.

See also: NCReplaceData.

Chapter 22

NCTest

Members are:

- NCTest
- NCTestRun
- NCTestSummarize

22.1 NCTest

`NCTest[expr,answer]` asserts whether `expr` is equal to `answer`. The result of the test is collected when `NCTest` is run from `NCTestRun`.

See also: #NCTestRun, #NCTestSummarize

22.2 NCTestRun

`NCTest[list]` runs the test files listed in `list` after appending the '.NCTest' suffix and return the results.

For example:

```
results = NCTestRun[{"NCCollect", "NCSylvester"}]
```

will run the test files "NCCollec.NCTest" and "NCSylvester.NCTest" and return the results in `results`.

See also: #NCTest, #NCTestSummarize

22.3 NCTestSummarize

`NCTestSummarize[results]` will print a summary of the results in `results` as produced by `NCTestRun`.

See also: #NCTestRun

Chapter 23

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

23.1 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.

23.2 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.

23.3 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.

23.4 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.

Chapter 24

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:

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

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

24.1 SDPMatrices

24.2 SDPSolve

24.3 SDPEval

24.4 SDPInner

24.5 SDPCheckDimensions

24.6 SDPDualEval

24.7 SDPFunctions

24.8 SDPPrimalEval

24.9 SDPScale

24.10 SDPSylvesterDiagonalEval

24.11 SDPSylvesterEval