The NCAlgebra Suite

J. William Helton   Mauricio C. de Oliveira

with earlier contributions by Bob Miller & Mark Stankus

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 Commands `Transform`, `Substitute`, `SubstituteSymmetric`, etc, have been replaced by the much more reliable commands in the new package NCReplace.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

This chapter provides a gentle introduction to some of the commands available in `NCAlgebra`. Before you can use `NCAlgebra` you first load it with the following commands:

```
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 noncommutative by default and all others are commutative by default. For example:

```
a**b-b**a
```

results in

```
a**b-b**a
```

while

```
A**B-B**A
A**b-b**A
```

both result in `0`.

One of Bill's favorite commands is `CommuteEverything`, which temporarily makes all noncommutative symbols appearing in a given expression to behave as if they were commutative and returns the resulting commutative expression. For example:

```
CommuteEverything[a**b-b**a]
```

results in `0`. The command

```
EndCommuteEverything[]
```

restores the original noncommutative behavior.

One can make any symbol behave as noncommutative using `SetNonCommutative`. For example:

```
SetNonCommutative[A,B]
A**B-B**A
```

results in:

```
A**B-B**A
```

Likewise, symbols can be made commutative using `SetCommutative`. For example:

```
SetNonCommutative[A]
SetCommutative[B]
A**B-B**A
```

results in `0`. `SNC` is an alias for `SetNonCommutative`. So, `SNC` can be typed rather than the longer `SetNonCommutative`:

```
SNC[A];
A**a-a**A
```

results in:

```
-a**A+A**a
```

One can check whether a given symbol is commutative or not using `CommutativeQ` or `NonCommutativeQ`. For example:

```
CommutativeQ[B]
NonCommutativeQ[a]
```

both return `True`.

3.2 Inverses, Transposes and Adjoints

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]`. `inv` operates as expected in most cases.

For example:

```
inv[a]**a
inv[a**b]**a**b
```

both lead to `Id = 1` and

```
a**b**inv[b]
```

results in `a`.

`tp[x]` denotes the transpose of symbol `x` and `aj[x]` denotes the adjoint of symbol `x`. Like `inv`, the properties of transposes and adjoints that everyone uses constantly are built-in. For example:

```
tp[a**b]
```

leads to

```
tp[b]**tp[a]
```

and

```
tp[a+b]
```

returns

```
tp[a]+tp[b]
```

Likewise `tp[tp[a]] == a` and `tp` for anything for which `CommutativeQ` returns `True` is simply the identity. For example `tp[5] == 5`, `tp[2 + 3I] == 2 + 3 I`, and `tp[B] == B`.

Similar properties hold to `aj`. Moreover

```
aj[tp[a]]
tp[aj[a]]
```

return `co[a]` where `co` stands for complex-conjugate.

Version 5.0: transposes (`tp`), adjoints (`aj`) and complex conjugates (`co`) in a notebook environment render as x^T, x^*, and $\bar{x}$.

3.3 Expand and Collect

The command `NCExpand` expands noncommutative products. For example:

```
NCExpand[(a+b)**x]
```

returns

```
a**x+b**x
```

Conversely, one can collect noncommutative terms involving same powers of a symbol using `NCCollect`. For example:

```
NCCollect[a**x+b**x,x]
```

recovers

```
(a+b)**x
```

`NCCollect` groups terms by degree before collecting and accepts more than one variable. For example:

```
expr = a**x+b**x+y**c+y**d+a**x**y+b**x**y
NCCollect[expr, {x}]
```

returns

```
y**c+y**d+(a+b)**x**(1+y)
```

and

```
NCCollect[expr, {x, y}]
```

returns

```
(a+b)**x+y**(c+d)+(a+b)**x**y
```

Note that the last term has degree 2 in `x` and `y` and therefore does not get collected with the first order terms.

The list of variables accepts `tp`, `aj` and `inv`, and

```
NCCollect[tp[x]**a**x+tp[x]**b**x+z,{x,tp[x]}]
```

returns

```
z+tp[x]**(a+b)**x
```

Alternatively one could use

```
NCCollectSymmetric[tp[x]**a**x+tp[x]**b**x+z,{x}]
```

to obtain the same result. A similar command, NCCollectSelfAdjoint, works with self-adjoint variables.

There is also a stronger version of collect called `NCStrongCollect`. `NCStrongCollect` does not group terms by degree. For instance:

```
NCStrongCollect[expr, {x, y}]
```

produces

```
y**(c+d)+(a+b)**x**(1+y)
```

Keep in mind that `NCStrongCollect` often collects *more* than one would normally expect.

3.4 Replace

A key feature of symbolic computation is the ability to perform substitutions. The Mathematica substitute commands, e.g. `ReplaceAll` (`/.`) and `ReplaceRepeated` (`//.`), are not reliable in `NCAlgebra`, so you must use our `NC` versions of these commands. For example:

```
NCReplaceAll[x**a**b,a**b->c]
```

results in

```
x**c
```

and

```
NCReplaceAll[tp[b**a]+b**a,b**a->c]
```

results in

```
c+tp[a]**tp[b]
```

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

```
NCReplaceAll[tp[b**a]+b**a, NCMakeRuleSymmetric[b ** a -> c]]
```

returns

```
c + tp[c]
```

The difference between `NCReplaceAll` and `NCReplaceRepeated` can be understood in the example:

```
NCReplaceAll[a ** b ** b, a ** b -> a]
```

that results in

```
a ** b
```

and

```
NCReplaceRepeated[a ** b ** b, a ** b -> a]
```

that results in

```
a
```

Beside `NCReplaceAll` and `NCReplaceRepeated` we offer `NCReplace` and `NCReplaceList`, which are analogous to the standard `ReplaceAll` (`/.`), `ReplaceRepeated` (`//.`), `Replace` and `ReplaceList`. Note that one rarely uses `NCReplace` and `NCReplaceList`.

Version 5.0: the commands `Substitute` and `Transform` have been deprecated in favor of the above nc versions of `Replace`.

3.5 Polynomials

3.6 Rationals and Simplification

One of the great challenges of noncommutative symbolic algebra is the simplification of rational nc expressions. `NCAlgebra` provides various algorithms that can be used for simplification and general manipulation of nc rationals.

One such function is `NCSimplifyRational`, which attempts to simplify noncommutative rationals using a predefined set of rules. For example:

```
expr = 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
NCSimplifyRational[expr]
```

leads to `1`. Of course the great challenge here is to reveal well known identities that can lead to simplification. For example, the two expressions:

```
expr1 = a**inv[1+b**a]
expr2 = inv[1+a**b]**a
```

and one can use `NCSimplifyRational` to test such equivalence by evaluating

```
NCSimplifyRational[expr1 - expr2]
```

which results in `0` or

```
NCSimplifyRational[expr1**inv[expr2]]
```

which results in `1`. `NCSimplifyRational` works by transforming nc rationals. For example, one can verify that

```
NCSimplifyRational[expr2] == expr1
```

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]
```

?? ADD INTEGRATE AND HESSIAN ??

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}}
```

?? ADD NCInverse, and much more ??

Chapter 4

NonCommutative Expressions

4.1 Expanding

4.2 Rules and Substitution

4.3 Collecting

4.4 Polynomials

4.5 Rationals

4.6 Simplification

Chapter 5

Matrices with NonCommutative Entries

Chapter 6

NonCommutative Gröbner Basis

The package `NCGBX` provides an implementation of a noncommutative Gröbner Basis algorithm. Gröbner Basis are useful in the study of algebraic relations.

In order to load `NCGB` one types:

```
<< NC`
<< NCGBX`
```

or simply

```
<< NCGBX`
```

if `NC` and `NCAlgebra` have already been loaded.

6.1 What is a Gröbner Basis?

Most commutative algebra packages contain commands based on Gröbner Basis and uses of Gröbner Basis. For example, in Mathematica, the `Solve` command puts collections of equations in a *canonical form* which, for simple collections, readily yields a solution. Likewise, the Mathematica `Eliminate` command tries to convert a collection of m polynomial equations

$$\begin{aligned} p_1(x_1, \ldots, x_n) &= 0 \\ p_2(x_1, \ldots, x_n) &= 0 \\ \vdots \quad &\quad \vdots \\ p_m(x_1, \ldots, x_n) &= 0 \end{aligned}$$

in variables $x_1, x_2, \ldots x_n$ to a *triangular* form, that is a new collection of equations like

$$\begin{aligned} q_1(x_1) &= 0 \\ q_2(x_1, x_2) &= 0 \\ q_3(x_1, x_2) &= 0 \\ q_4(x_1, x_2, x_3) &= 0 \\ \vdots \quad &\quad \vdots \\ q_r(x_1, \ldots, x_n) &= 0. \end{aligned}$$

Here the polynomials $\{q_j : 1 \leq j \leq k_2\}$ generate the same *ideal* that the polynomials $\{p_j : 1 \leq j \leq k_1\}$ generate. Therefore, the set of solutions to the collection of polynomial equations $\{p_j = 0 : 1 \leq j \leq k_1\}$ equals the set of solutions to the collection of polynomial equations $\{q_j = 0 : 1 \leq j \leq k_2\}$. This canonical form greatly simplifies the task of solving collections of polynomial equations by facilitating backsolving for x_j in terms of $x_1, \ldots, x_{j-1}$.

Readers who would like to know more about Gröbner Basis may want to read [CLS]. The noncommutatative version of the algorithm implemented by `NCGB` is loosely based on [Mora].

6.2 Solving equations

Before calculating a Gröbner Basis, one must declare which variables will be used during the computation and must declare a *monomial order* which can be done using `SetMonomialOrder` as in:

```
SetMonomialOrder[{a, b, c}, x];
```

The monomial ordering imposes a relationship between the variables which are used to *sort* the monomials in a polynomial. The ordering implied by the above command can be visualized using:

```
PrintMonomialOrder[];
```

which in this case prints:

$$a < b < c \ll x.$$

A user does not need to know theoretical background related to monomials orders. Indeed, as we shall see soon, in many engineering problems, it suffices to know which variables correspond to quantities which are *known* and which variables correspond to quantities which are *unknown*. If one is solving for a variable or desires to prove that a certain quantity is zero, then one would want to view that variable as *unknown*. In the above example, the symbol '$\ll$' separate the *knowns*, a, b, c, from the *unknown*, x. For more details on orderings see Section Orderings.

Our goal is to calculate the Gröbner basis associated with the following relations:

$$a\,x\,a = c, \quad a\,b = 1, \quad b\,a = 1.$$

We shall use the word *relation* to mean a polynomial in noncommuting indeterminates. For example, if an analyst saw the equation $AB = 1$ for matrices A and B, then he might say that A and B satisfy the polynomial equation $a\,b - 1 = 0$. An algebraist would say that $a\,b - 1$ is a relation.

To calculate a Gröbner basis one defines a list of relations

```
rels = {a ** x ** a - c, a ** b - 1, b ** a - 1}
```

and issues the command:

```
gb = NCMakeGB[rels, 10]
```

which should produces an output similar to:

```
* * * * * * * * * * * * * * * *
* * *   NCPolyGroebner    * * *
* * * * * * * * * * * * * * * *
* Monomial order : a < b < c << x
* Reduce and normalize initial basis
> Initial basis could not be reduced
* Computing initial set of obstructions
> MAJOR Iteration 1, 4 polys in the basis, 2 obstructions
> MAJOR Iteration 2, 5 polys in the basis, 2 obstructions
* Cleaning up basis.
```

```
* Found Groebner basis with 3 relations
* * * * * * * * * * * * * * * *
```

The number `10` in the call to `NCMakeGB` is very important because a finite GB may not exist. It instructs `NCMakeGB` to abort after `10` iterations if a GB has not been found at that point.

The result of the above calculation is the list of relations:

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

Our favorite format for displaying lists of relations is `ColumnForm`.

```
ColumnForm[gb]
```

which results in

```
x -> b ** c ** b
a ** b -> 1
b ** a -> 1
```

The *rules* in the output represent the relations in the GB with the left-hand side of the rule being the leading monomial. Replacing `Rule` by `Subtract` recovers the relations but one would then loose the leading monomial as Mathematica alphabetizes the resulting sum.

Someone not familiar with GB's might find it instructive to note this output GB effectively *solves* the input equation

$$a\,x\,a - c = 0$$

under the assumptions that

$$b\,a - 1 = 0, \quad a\,b - 1 = 0,$$

that is $a = b^{-1}$ and produces the expected result in the form of the relation:

$$x = b\,c\,b.$$

6.3 A slightly more challenging example

For a slightly more challenging example consider the same monomial order as before:

```
SetMonomialOrder[{a, b, c}, x];
```

that is

$a < b < c \ll x$

and the relations:

$$\begin{aligned} a\,x - c &= 0, \\ a\,b\,a - a &= 0, \\ b\,a\,b - b &= 0, \end{aligned}$$

from which one can recognize the problem of solving the linear equation $a\,x = c$ in terms of the *pseudo-inverse* $b = a^{\dagger}$. The calculation:

```
gb = NCMakeGB[{a ** x - c, a ** b ** a - a, b ** a ** b - b}, 10];
```

finds the Gröbner basis:

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

In this case the Gröbner basis cannot quite *solve* the equations but it remarkably produces the necessary condition for existence of solutions:

$$0 = a\,b\,c - c = a\,a^{\dagger}c - c$$

that can be interpreted as c being in the range-space of a.

6.4 Simplifying polynomial expresions

Our goal now is to verify if it is possible to *simplify* the following expression:

$$b\,b\,a\,a - a\,a\,b\,b + a\,b\,a$$

if we know that

$$a\,b\,a = b$$

using Gröbner basis. With that in mind we set the order:

```
SetMonomialOrder[a,b];
```

and calculate the GB associated with the constraint:

```
rels = {a ** b ** a - b};
rules = NCMakeGB[rels, 10];
```

which produces the output

```
* * * * * * * * * * * * * * * *
* * *   NCPolyGroebner    * * *
* * * * * * * * * * * * * * * *
* Monomial order : a << b
* Reduce and normalize initial basis
> Initial basis could not be reduced
* Computing initial set of obstructions
> MAJOR Iteration 1, 2 polys in the basis, 1 obstructions
* Cleaning up basis.
* Found Groebner basis with 2 relations
* * * * * * * * * * * * * * * *
```

and the associated GB

```
a ** b ** a -> b
b ** b ** a -> a ** b ** b
```

The GB revealed another relationship that must hold true if $a\,b\,a = b$. One can use this relationship to simplify the original expression using `NCReplaceRepeated` as in

```
expr = b ** b ** a ** a - a ** a ** b ** b + a ** b ** a
simp = NCReplaceRepeated[expr, rules]
```

which results in

```
simp = b
```

6.5 Simplifying rational expresions

It is often desirable to simplify expressions involving inverses of noncommutative expressions. One challenge is to recognize identities implied by the existence of certain inverses. For example, that the expression

$$x(1-x)^{-1} - (1-x)^{-1}x$$

is equivalent to 0. One can use a nc Gröbner basis for that task. Consider for instance the order

$$x \ll (1-x)^{-1}$$

implied by the command:

```
SetMonomialOrder[x, inv[1-x]]
```

This ordering encodes the following precise idea of what we mean by *simple* versus *complicated*: it formally corresponds to specifying that x is simpler than $(1-x)^{-1}$, which might sits well with one's intuition.

Not consider the following command:

```
rules = NCMakeGB[{}, 3]
```

which produces the output

```
* * * * * * * * * * * * * * * *
* * *   NCPolyGroebner    * * *
* * * * * * * * * * * * * * * *
* Monomial order : x <<  inv[x] << inv[1 - x]
* Reduce and normalize initial basis
> Initial basis could not be reduced
* Computing initial set of obstructions
> MAJOR Iteration 1, 6 polys in the basis, 6 obstructions
* Cleaning up basis.
* Found Groebner basis with 6 relations
* * * * * * * * * * * * * * * *
```

and results in the rules:

```
x ** inv[1 - x] -> -1 + inv[1 - x],
inv[1-x] ** x -> -1 + inv[1-x],
```

As in the previous example, the GB revealed new relationships that must hold true if $1-x$ is invertible, and one can use this relationship to *simplify* the original expression using `NCReplaceRepeated` as in:

```
NCReplaceRepeated[x ** inv[1 - x] - inv[1 - x] ** x, rules]
```

The above command results in `0`, as one would hope.

For a more challenging example consider the identity:

$$\left(1-x-y(1-x)^{-1}y\right)^{-1} = \frac{1}{2}(1-x-y)^{-1} + \frac{1}{2}(1-x+y)^{-1}$$

One can verify that the rule based command NCSimplifyRational fails to simplify the expression:

```
expr = inv[1 - x - y ** inv[1 - x] ** y] - 1/2 (inv[1 - x + y] + inv[1 - x - y])
NCSimplifyRational[expr]
```

We set the monomial order and calculate the Gröbner basis

```
SetMonomialOrder[x, y, inv[1-x], inv[1-x+y], inv[1-x-y], inv[1-x-y**inv[1-x]**y]];
rules = NCMakeGB[{}, 3];
```

based on the rational involved in the original expression. The result is the nc GB:

```
inv[1-x-y**inv[1-x]**y] -> (1/2)inv[1-x-y]+(1/2)inv[1-x+y]
x**inv[1-x] -> -1+inv[1-x]
y**inv[1-x+y] -> 1-inv[1-x+y]+x**inv[1-x+y]
y**inv[1-x-y] -> -1+inv[1-x-y]-x**inv[1-x-y]
inv[1-x]**x -> -1+inv[1-x]
```

```
inv[1-x+y]**y -> 1-inv[1-x+y]+inv[1-x+y]**x
inv[1-x-y]**y -> -1+inv[1-x-y]-inv[1-x-y]**x
inv[1-x+y]**x**inv[1-x-y] -> -(1/2)inv[1-x-y]-(1/2)inv[1-x+y]+inv[1-x+y]**inv[1-x-y]
inv[1-x-y]**x**inv[1-x+y] -> -(1/2)inv[1-x-y]-(1/2)inv[1-x+y]+inv[1-x-y]**inv[1-x+y]
```

which succesfully simplifyes the original expression using:

```
expr = inv[1 - x - y ** inv[1 - x] ** y] - 1/2 (inv[1 - x + y] + inv[1 - x - y])
NCReplaceRepeated[expr, rules] // NCExpand
```

resulting in `0`.

6.6 Simplification with `NCGBSimplifyRational`

The simplification process described above is automated in the function NCGBSimplifyRational and calls to

```
expr = x ** inv[1 - x] - inv[1 - x] ** x
NCGBSimplifyRational[expr]
```

or

```
expr = inv[1 - x - y ** inv[1 - x] ** y] - 1/2 (inv[1 - x + y] + inv[1 - x - y])
NCGBSimplifyRational[expr]
```

both result in `0`.

?? DO WE WANT TO SUPPORT pinv, linv and rinv? ??

6.7 Ordering on variables and monomials

As seen above, one needs to declare a *monomial order* before making a Gröbner Basis. There are various monomial orders which can be used when computing Gröbner Basis. The most common are *lexicographic* and *graded lexicographic* orders. We consider also *multi-graded lexicographic* orders.

Lexicographic and multi-graded lexicographic orders are examples of elimination orderings. An elimination ordering is an ordering which is used for solving for some of the variables in terms of others.

We now discuss each of these types of orders.

6.7.1 Lex Order: the simplest elimination order

To impose lexicographic order, say $a \ll b \ll x \ll y$ on a, b, x and y, one types

```
SetMonomialOrder[a,b,x,y];
```

This order is useful for attempting to solve for y in terms of a, b and x, since the highest priority of the GB algorithm is to produce polynomials which do not contain y. If producing high order polynomials is a consequence of this fanaticism so be it. Unlike graded orders, lex orders pay little attention to the degree of terms. Likewise its second highest priority is to eliminate x.

Once this order is set, one can use all of the commands in the preceeding section in exactly the same form.

We now give a simple example how one can solve for y given that a,b,x and y satisfy the equations:

$$\begin{aligned} -b\,x + x\,y\,a + x\,b\,a\,a &= 0 \\ x\,a - 1 &= 0 \\ a\,x - 1 &= 0 \end{aligned}$$

The command

```
NCMakeGB[{-b**x+x**y**a+x**b**a**a, x**a-1, a**x-1},4]
```

produces the Gröbner basis:

```
y -> -b**a + a**b**x**x
a**x -> 1
x**a -> 1
```

after one iteration.

Now, we change the order to

```
SetMonomialOrder[y,x,b,a];
```

and run the same `NCMakeGB` as above:

```
NCMakeGB[{-b**x+x**y**a+x**b**a**a, x**a-1, a**x-1},4]
```

which, this time, results in

```
x**a -> 1
a**x -> 1
x**b**a -> -x**y+b**x**x
b**a**a -> -y**a+a**b**x
x**b**b**a -> -x**b**y-x**y**b**x**x+b**x**x**b**x**x
b**x**x**x -> x**b+x**y**x
b**a**b**a -> -y**y-b**a**y-y**b**a+a**b**x**b**x**x
a**b**x**x -> y+b**a
b**a**b**b**a -> -y**b**y-b**a**b**y-y**b**b**a-y**y**b**x**x-
                 b**a**y**b**x**x+a**b**x**b**x**x**b**x**x
```

which is not a Gröbner basis since the algorithm was interrupted at 4 iterations. Note the presence of the rule

```
a**b**x**x -> y+b**a
```

which shows that the order is not set up to solve for y in terms of the other variables in the sense that y is not on the left hand side of this rule (but a human could easily solve for y using this rule). Also the algorithm created a number of other relations which involved y.

6.7.2 Graded lex ordering: a non-elimination order

To impose graded lexicographic order, say $a < b < x < y$ on a, b, x and y, one types

```
SetMonomialOrder[{a,b,x,y}];
```

This ordering puts high degree monomials high in the order. Thus it tries to decrease the total degree of expressions. A call to

```
NCMakeGB[{-b**x+x**y**a+x**b**a**a, x**a-1, a**x-1},4]
```

now produces

```
a**x -> 1
x**a -> 1
b**a**a -> -y**a+a**b**x
x**b**a -> -x**y+b**x**x
a**b**x**x -> y+b**a
b**x**x**x -> x**b+x**y**x
a**b**x**b**x**x -> y**y+b**a**y+y**b**a+b**a**b**a
b**x**x**b**x**x -> x**b**y+x**b**b**a+x**y**b**x**x
```

```
a**b**x**b**x**b**x**x -> y**y**y+b**a**y**y+y**b**a**y+y**y**b**a+
                          b**a**b**a**y+b**a**y**b**a+y**b**a**b**a+
                          b**a**b**a**b**a
b**x**x**b**x**b**x**x -> x**b**y**y+x**b**b**a**y+x**b**y**b**a+
                          x**b**b**a**b**a+x**y**b**x**b**x**x
```

which again fails to be a Gröbner basis and does not eliminate y. Instead, it tries to decrease the total degree of expressions involving a, b, x, and y.

6.7.3 Multigraded lex ordering : a variety of elimination orders

There are other useful monomial orders which one can use other than graded lex and lex. Another type of order is what we call multigraded lex and is a mixture of graded lex and lex order. To impose multi-graded lexicographic order, say $a < b < x \ll y$ on a, b, x and y, one types

```
SetMonomialOrder[{a,b,x},y];
```

which separates y from the remaining variables. This time, a call to

```
NCMakeGB[{-b**x+x**y**a+x**b**a**a, x**a-1, a**x-1},4]
```

yields once again

```
y -> -b**a+a**b**x**x
a**x -> 1
x**a -> 1
```

which not only eliminates y but is also Gröbner basis, calculated after one iteration.

For an intuitive idea of why multigraded lex is helpful, we think of a, b, and x as corresponding to variables in some engineering problem which represent quantities which are *known* and y to be *unknown*. The fact that a, b and x are in the top level indicates that we are very interested in solving for y in terms of a, b, and x, but are not willing to solve for, say x, in terms of expressions involving y.

This situation is so common that we provide the commands `SetKnowns` and `SetUnknowns`. The above ordering would be obtained after setting

```
SetKnowns[a,b,x];
SetUnknowns[y];
```

6.8 A complete example: the partially prescribed matrix inverse problem

This is a type of problem known as a *matrix completion problem*. This particular one was suggested by Hugo Woerdeman. We are grateful to him for discussions.

Problem: *Given matrices a, b, c, and d, we wish to determine under what conditions there exists matrices x, y, z, and w such that the block matrices*

$$\begin{bmatrix} a & x \\ y & b \end{bmatrix} \qquad \begin{bmatrix} w & c \\ d & z \end{bmatrix}$$

are inverses of each other. Also, we wish to find formulas for x, y, z, and w.

This problem was solved in a paper by W.W. Barrett, C.R. Johnson, M. E. Lundquist and H. Woerderman [BJLW] where they showed it splits into several cases depending upon which of a, b, c and d are invertible. In our example, we assume that a, b, c and d are invertible and discover the result which they obtain in this case.

First we set the matrices a, b, c, and d and their inverses as *knowns* and x, y, w, and z as unknowns:

```
SetKnowns[a, inv[a], b, inv[b], c, inv[c], d, inv[d]];
SetUnknowns[{z}, {x, y, w}];
```

Note that the graded ordedring of the unknowns means that we care more about solving for x, y and w than for z.

Then we define the relations we are interested in, which are obtained after multiplying the two block matrices on both sides and equating to identity

```
A = {{a, x}, {y, b}}
B = {{w, c}, {d, z}}

rels = {
  MatMult[A, B] - IdentityMatrix[2],
  MatMult[B, A] - IdentityMatrix[2]
} // Flatten
```

We use `Flatten` to reduce the matrix relations to a simple list of relations. The resulting relations in this case are:

```
rel = {-1+a**w+x**d, a**c+x**z, b**d+y**w, -1+b**z+y**c,
       -1+c**y+w**a, c**b+w**x, d**a+z**y, -1+d**x+z**b}
```

After running

```
NCMakeGB[rels, 8]
```

we obtain the Gröbner basis:

```
x -> inv[d]-inv[d]**z**b
y -> inv[c]-b**z**inv[c]
w -> inv[a]**inv[d]**z**b**d
z**b**z -> z+d**a**c
c**b**z**inv[c]**inv[a] -> inv[a]**inv[d]**z**b**d
inv[c]**inv[a]**inv[d]**z**b -> b**z**inv[c]**inv[a]**inv[d]
inv[d]**z**b**d**a -> a**c**b**z**inv[c]
z**b**d**a**c -> d**a**c**b**z
z**inv[c]**inv[a]**inv[d]**inv[b] -> inv[b]**inv[c]**inv[a]**inv[d]**z
z**inv[c]**inv[a]**inv[d]**z -> inv[b]+inv[b]**inv[c]**inv[a]**inv[d]**z
d**a**c**b**z**inv[c] -> z**b**d**a
```

after seven iterations. The first four relations

$$
\begin{aligned}
x &= d^{-1} - d^{-1}\, z\, b \\
y &= c^{-1} - b\, z\, c^{-1} \\
w &= a^{-1}\, d^{-1}\, z\, b\, d \\
z\, b\, z &= z + d\, a\, c
\end{aligned}
$$

are the solutions we are looking for, which states that one can find x, y, z, and w such that the matrices above are inverses of each other if and only if $z\, b\, z = z + d\, a\, c$. The first three relations gives formulas for x, y and w in terms of z.

A variety of scenarios can be quickly investigated under different assumptions. For example, say that c is not invertible. Is it still possible to solve the problem? One solution is obtained with the ordering implied by

```
SetKnowns[a, inv[a], b, inv[b], c, d, inv[d]];
SetUnknowns[{y}, {z, w, x}];
```

In this case

```
NCMakeGB[rels, 8]
```

produces the Gröbner basis:

```
z -> inv[b]-inv[b]**y**c
w -> inv[a]-c**y**inv[a]
x -> a**c**y**inv[a]**inv[d]
y**c**y -> y+b**d**a
c**y**inv[a]**inv[d]**inv[b] -> inv[a]**inv[d]**inv[b]**y**c
d**a**c**y**inv[a] -> inv[b]**y**c**b**d
inv[d]**inv[b]**y**c**b -> a**c**y**inv[a]**inv[d]
y**c**b**d**a -> b**d**a**c**y
y**inv[a]**inv[d]**inv[b]**y**c -> 1+y**inv[a]**inv[d]**inv[b]
```

after five iterations. Once again, the first four relations

$$
\begin{aligned}
z &= b^{-1} - b^{-1}\, y\, c \\
w &= a^{-1} - c\, y\, a^{-1} \\
x &= a\, c\, y\, a^{-1}\, d^{-1} \\
y\, c\, y &= y + b\, d\, a
\end{aligned}
$$

provide formulas, this time for z, w, and z in terms of y satisfying $y\,c\,y = y + b\,d\,a$. Note that these formulas do not involve c^{-1} since c is no longer assumed invertible.

Chapter 7

Semidefinite Programming

7.1 Semidefinite Programs in Matrix Variables

The package NCSDP allows the symbolic manipulation and numeric solution of semidefinite programs.

The package must be loaded using:

```
<< NCSDP`
```

Semidefinite programs 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

```
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]
```

The above commands produce a formatted output similar to the ones shown above.

In order to obtaining a numerical solution for 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 the above list of rules representing a problem instance one can load `SDPSylvester` and use `NCSDP` to create a problem instance as follows:

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

The resulting `abc` and `rules` objects are used for calculating the numerical solution using `SDPSolve`. The command:

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

produces an output like the folowing:

```
Problem data:
* Dimensions (total):
  - Variables             = 4
  - Inequalities          = 2
* Dimensions (detail):
  - Variables             = {{2,2}}
  - Inequalities          = {2,2}
Method:
* Method                  = PredictorCorrector
* Search direction        = NT
Precision:
* Gap tolerance           = 1.*10^(-9)
* Feasibility tolerance   = 1.*10^(-6)
* Rationalize iterates    = False
Other options:
* Debug level             = 0

 K     <B, Y>         mu  theta/tau      alpha     |X S|2    |X S|oo  |A* X-B|   |A Y+S-C|
-------------------------------------------------------------------------------------------
 1  1.638e+00  1.846e-01  2.371e-01  8.299e-01  1.135e+00  9.968e-01  9.868e-16  2.662e-16
 2  1.950e+00  1.971e-02  2.014e-02  8.990e-01  1.512e+00  9.138e-01  2.218e-15  2.937e-16
 3  1.995e+00  1.976e-03  1.980e-03  8.998e-01  1.487e+00  9.091e-01  1.926e-15  3.119e-16
 4  2.000e+00  9.826e-07  9.826e-07  9.995e-01  1.485e+00  9.047e-01  8.581e-15  2.312e-16
 5  2.000e+00  4.913e-10  4.913e-10  9.995e-01  1.485e+00  9.047e-01  1.174e-14  4.786e-16
-------------------------------------------------------------------------------------------
* Primal solution is not strictly feasible but is within tolerance
(0 <= max eig(A* Y - C) = 8.06666*10^-10 < 1.*10^-6 )
* Dual solution is within tolerance
(|| A X - B || = 1.96528*10^-9 < 1.*10^-6)
* Feasibility radius = 0.999998
(should be less than 1 when feasible)
```

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

A symbolic dual problem can be calculated easily using `NCSDPDual`:

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

The dual program for the example problem above is:

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

In the case of the above problem the dual program is

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

which can be visualized using `NCSDPDualForm` using:

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

7.2 Semidefinite Programs in Vector Variables

The package SDP provides a crude and not very efficient 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 and y is a vector of decision variables.

A user can input the problem data, the triplet (A, b, c), or use the following convenient methods for producing data in the proper format.

For example, problems can be stated as:

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

where $f(y)$ and $G(y)$ are affine functions of the vector of variables y.

Here is a simple example:

```
y = {y0, y1, y2};
f = y2;
G = {y0 - 2, {{y1, y0}, {y0, 1}}, {{y2, y1}, {y1, 1}}};
```

The list of constraints in `G` is 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[f, G, 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.

Chapter 8

Pretty Output with Mathematica Notebooks and TeX

`NCAlgebra` comes with several utilities for facilitating formatting of expression in notebooks or using LaTeX.

8.1 Pretty Output

On a Mathematica notebook session the package NCOutput can be used to control how nc expressions are displayed. `NCOutput` does not alter the internal representation of nc expressions, just the way they are displayed on the screen.

The function NCSetOutput can be used to set the display options. For example:

```
NCSetOutput[tp -> False, inv -> True];
```

makes the expression

```
expr = inv[tp[a] + b]
```

be displayed as

$\texttt{(tp[a] + b)}^{-1}$

Conversely

```
NCSetOutput[tp -> True, inv -> False];
```

makes `expr` be displayed as

$\texttt{inv[a}^{\texttt{T}}\texttt{ + b]}$

The default settings are

```
NCSetOutput[tp -> True, inv -> True];
```

which makes `expr` be displayed as

$\texttt{(a}^{\texttt{T}}\texttt{ + b)}^{-1}$

The complete set of options and their default values are:

- `NonCommutativeMultiply` (`False`): If `True` `x**y` is displayed as '`x` • `y`';
- `tp` (`True`): If `True` `tp[x]` is displayed as '$\texttt{x}^{\texttt{T}}$';
- `inv` (`True`): If `True` `inv[x]` is displayed as '$\texttt{x}^{-1}$';
- `aj` (`True`): If `True` `aj[x]` is displayed as '$\texttt{x}^{*}$';

- `co` (`True`): If `True` `co[x]` is displayed as ‘$\bar{\mathtt{x}}$’;
- `rt` (`True`): If `True` `rt[x]` is displayed as ‘$\mathtt{x}^{1/2}$’.

The special symbol `All` can be used to set all options to `True` or `False`, as in

```
NCSetOutput[All -> True];
```

8.2 Using NCTeX

You can load NCTeX using the following command

```
<< NC`
<< NCTeX`
```

`NCTeX` does not need `NCAlgebra` to work. You may want to use it even when not using NCAlgebra. It uses `NCRun`, which is a replacement for Mathematica's Run command to run `pdflatex`, `latex`, `divps`, etc.

WARNING: Mathematica does not come with LaTeX, dvips, etc. The package `NCTeX` does not install these programs but rather assumes that they have been previously installed and are available at the user's standard shell. Use the `Verbose` options to troubleshoot installation problems.

With `NCTeX` loaded you simply type `NCTeX[expr]` and your expression will be converted to a PDF image after being processed by `LaTeX`.

For example:

```
expr = 1 + Sin[x + (y - z)/Sqrt[2]];
NCTeX[expr]
```

produces

$1 + \sin\left(x + \frac{y-z}{\sqrt{2}}\right)$

If `NCAlgebra` is not loaded then `NCTeX` uses the built in `TeXForm` to produce the LaTeX expressions. If `NCAlgebra` is loaded, `NCTeXForm` is used. See NCTeXForm for details.

Here is another example:

```
expr = {{1 + Sin[x + (y - z)/2 Sqrt[2]], x/y}, {z, n Sqrt[5]}};
NCTeX[expr]
```

that produces

$$\begin{pmatrix} \sin\left(x + \frac{y-z}{\sqrt{2}}\right) + 1 & \frac{x}{y} \\ z & \sqrt{5}n \end{pmatrix}$$

In some cases Mathematica will have difficulty displaying certain PDF files. When this happens `NCTeX` will span a PDF viewer so that you can look at the formula. If your PDF viewer does not pop up automatically you can force it by passing the following option to `NCTeX`:

```
expr = {{1 + Sin[x + (y - z)/2 Sqrt[2]], x/y}, {z, n Sqrt[5]}};
NCTeX[exp, DisplayPDF -> True]
```

Here is another example were the current version of Mathematica fails to import the PDF:

```
expr = Table[x^i y^(-j) , {i, 0, 10}, {j, 0, 30}];
NCTeX[expr, DisplayPDF -> True]
```

You can also suppress Mathematica from importing the PDF altogether as well. This and other options are covered in detail in the next section.

8.2.1 NCTeX Options

The following command:

```
expr = {{1 + Sin[x + (y - z)/2 Sqrt[2]], x/y}, {z, n Sqrt[5]}};
NCTeX[exp, DisplayPDF -> True, ImportPDF -> False]
```

uses `DisplayPDF -> True` to ensure that the PDF viewer is called and `ImportPDF -> False` to prevent Mathematica from displaying the formula inline. In other words, it displays the formula in the PDF viewer without trying to import the PDF into Mathematica. The default values for these options when using the Mathematica notebook interface are:

1. `DisplayPDF` (`False`)
2. `ImportPDF` (`True`)

When `NCTeX` is invoked using the command line interpreter version of Mathematica the defaults are:

1. `DisplayPDF` (`False`)
2. `ImportPDF` (`True`)

Other useful options and their default options are:

1. `Verbose` (`False`),
2. `BreakEquations` (`True`)
3. `TeXProcessor` (`NCTeXForm`)

Set `BreakEquations -> True` to use the LaTeX package `beqn` to produce nice displays of long equations. Try the following example:

```
expr = Series[Exp[x], {x, 0, 20}]
NCTeX[expr]
```

Use `TexProcessor` to select your own `TeX` converter. If `NCAlgebra` is loaded then `NCTeXForm` is the default. Otherwise Mathematica's `TeXForm` is used.

If `Verbose -> True` you can see a detailed display of what is going on behing the scenes. This is very useful for debugging. For example, try:

```
expr = BesselJ[2, x]
NCTeX[exp, Verbose -> True]
```

to produce an output similar to the following one:

```
* NCTeX - LaTeX processor for NCAlgebra - Version 0.1
> Creating temporary file '/tmp/mNCTeX.tex'...
> Processing '/tmp/mNCTeX.tex'...
> Running 'latex -output-directory=/tmp/  /tmp/mNCTeX 1> "/tmp/mNCRun.out" 2> "/tmp/mNCRun.err"'...
> Running 'dvips -o /tmp/mNCTeX.ps -E /tmp/mNCTeX 1> "/tmp/mNCRun.out" 2> "/tmp/mNCRun.err"'...
> Running 'epstopdf  /tmp/mNCTeX.ps 1> "/tmp/mNCRun.out" 2> "/tmp/mNCRun.err"'...
> Importing pdf file '/tmp/mNCTeX.pdf'...
```

Locate the files with extension .err as indicated by the verbose run of NCTeX to diagnose errors.

The remaining options:

1. `PDFViewer` (`"open"`),
2. `LaTeXCommand` (`"latex"`)
3. `PDFLaTeXCommand` (`Null`)
4. `DVIPSCommand` (`"dvips"`)
5. `PS2PDFCommand` (`"epstopdf"`)

let you specify the names and, when appropriate, the path, of the corresponding programs to be used by `NCTeX`. Alternatively, you can also directly implement custom versions of

```
NCRunDVIPS
NCRunLaTeX
NCRunPDFLaTeX
NCRunPDFViewer
NCRunPS2PDF
```

Those commands are invoked using `NCRun`. Look at the documentation for the package NCRun for more details.

8.3 Using NCTeXForm

`NCTeXForm` is a replacement for Mathematica's `TeXForm` that can handle noncommutative expressions. It works just as `TeXForm`. `NCTeXForm` is automatically loaded with `NCAlgebra` and becomes the default processor for `NCTeX`.

Here is an example:

```
SetNonCommutative[a, b, c, x, y];
exp = a ** x ** tp[b] - inv[c ** inv[a + b ** c] ** tp[y] + d]
NCTeXForm[exp]
```

produces

```
a.x.{b}^T-{\left(d+c.{\left(a+b.c\right)}^{-1}.{y}^T\right)}^{-1}
```

Note that the LaTeX output contains special code so that the expression looks neat on the screen. You can see the result using `NCTeX` to convert the expression to PDF. Try

```
SetOptions[NCTeX, TeXProcessor -> NCTeXForm];
NCTeX[exp]
```

to produce

$a.x.{b}^T-{\left(d+c.{\left(a+b.c\right)}^{-1}.{y}^T\right)}^{-1}$

`NCTeX` represents noncommutative products with a dot (`.`) in order to distinguish it from its commutative cousin. We can see the difference in an expression that has both commutative and noncommutative products:

```
exp = 2 a ** b - 3 c ** d
NCTeX[exp]
```

produces

$2\,(a.b) - 3(c.d)$

NCTeXForm handles lists and matrices as well. Here is a list:

```
exp = {x, tp[x], x + y, x + tp[y], x + inv[y], x ** x}
NCTeX[exp]
```

and its output:

$\{x, x^T, x+y, x+y^T, x+y^{-1}, x.x\}$

and here is a matrix example:

```
exp = {{x, y}, {y, z}}
NCTeX[exp]
```

and its output:

$$\begin{bmatrix} x & y \\ y & z \end{bmatrix}$$

Here are some more examples:

```
exp = {{1 + Sin[x + (y - z)/2 Sqrt[2]], x/y}, {z, n Sqrt[5]}}
NCTeX[exp]
```

produces

$$\begin{bmatrix} 1+\sin\left(x+\frac{1}{\sqrt{2}}\left(y-z\right)\right) & xy^{-1} \\ z & \sqrt{5}n \end{bmatrix}$$

```
exp = {inv[x + y], inv[x + inv[y]]}
NCTeX[exp]
```

produces:

$$\{(x+y)^{-1}, \left(x+y^{-1}\right)^{-1}\}$$

```
exp = {Sin[x], x y, Sin[x] y, Sin[x + y], Cos[gamma],
       Sin[alpha] tp[x] ** (y - tp[y]), (x + tp[x]) (y ** z), -tp[y], 1/2,
       Sqrt[2] x ** y}
NCTeX[exp]
```

produces:

$$\{\sin x, xy, y\sin x, \sin\left(x+y\right), \cos\gamma, \left(x^T.\left(y-y^T\right)\right)\sin\alpha, yz\left(x+x^T\right), -y^T, \frac{1}{2}, \sqrt{2}\left(x.y\right)\}$$

```
exp = inv[x + tp[inv[y]]]
NCTeX[exp]
```

produces:

$$\left(x+y^{T^{-1}}\right)^{-1}$$

`NCTeXForm` does not know as many functions as `TeXForm`. In some cases `TeXForm` will produce better results. Compare:

```
exp = BesselJ[2, x]
NCTeX[exp, TeXProcessor -> NCTeXForm]
```

output:

$\mathrm{BesselJ}\left(2, x\right)$

with

```
NCTeX[exp, TeXProcessor -> TeXForm]
```

output:

$J_2(x)$

It should be easy to customize `NCTeXForm` though. Just overload `NCTeXForm`. In this example:

```
NCTeXForm[BesselJ[x_, y_]] := Format[BesselJ[x, y], TeXForm]
```

makes

```
NCTeX[exp, TeXProcessor -> NCTeXForm]
```

produce

$J_2(x)$

Part II

Reference Manual

Chapter 9

Introduction

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

Packages are automatically loaded unless otherwise noted.

Chapter 10

Packages for manipulating NC expressions

10.1 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

10.1.1 aj

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

See also: tp, co.

10.1.2 co

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

See also: aj.

10.1.3 Id

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

10.1.4 inv

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

10.1.5 rt

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

10.1.6 tp

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

See also: aj, co.

10.1.7 CommutativeQ

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

See also: SetCommutative, SetNonCommutative.

10.1.8 NonCommutativeQ

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

See also: CommutativeQ.

10.1.9 SetCommutative

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

See also: SetNonCommutative, CommutativeQ, NonCommutativeQ.

10.1.10 SetNonCommutative

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

See also: SetCommutative, CommutativeQ, NonCommutativeQ.

10.1.11 Commutative

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

See also: CommuteEverything, CommutativeQ, SetCommutative, SetNonCommutative.

10.1.12 CommuteEverything

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

See also: BeginCommuteEverything, Commutative.

10.1.13 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.

10.1.14 EndCommuteEverything

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

See also: BeginCommuteEverything, Commutative.

10.1.15 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`.

10.2 NCCollect

Members are:

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

10.2.1 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.

10.2.1.1 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, NCStrongCollectSelfAdjoint, NCRationalToNCPolynomial.

10.2.2 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.

10.2.3 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.

10.2.4 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.

10.2.5 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.

10.2.6 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.

10.2.7 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.

10.2.8 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.

10.2.9 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.

10.3 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]
inv[A inv[a] + B b] inv[a]	(1/A) inv[1 + (B/A) a b]
inv[a] inv[A inv[a] + K b]	(1/A) inv[1 + (B/A) 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

Original	Transformed
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

10.3.1 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.

10.3.2 NCSimplifyRational

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

See also: NCNormalizeInverse, NCSimplifyRationalSinglePass.

10.3.3 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.

10.3.4 NCPreSimplifyRational

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

See also: NCNormalizeInverse, NCPreSimplifyRationalSinglePass.

10.3.5 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.

10.4 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

10.4.1 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.

10.4.2 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.

10.4.3 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.

10.4.4 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.

10.4.5 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.

10.5 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

10.5.1 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.

10.5.2 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.

10.5.3 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.

10.5.4 NCReplaceRepeated

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

See also: NCReplace, NCReplaceAll, NCReplaceList.

10.5.5 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.

10.5.6 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.

10.6 NCSelfAdjoint

Members are:

- NCSymmetricQ
- NCSymmetricTest
- NCSymmetricPart
- NCSelfAdjointQ
- NCSelfAdjointTest

10.6.1 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.

10.6.2 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.

10.6.3 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.

10.6.4 NCSelfAdjointQ

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

See also: NCSymmetricQ, NCSelfAdjointTest.

10.6.5 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 11

Packages for manipulating NC block matrices

11.1 NCMatMult

Members are:

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

11.1.1 tpMat

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

See also: ajMat, coMat, MatMult.

11.1.2 ajMat

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

See also: tpMat, coMat, MatMult.

11.1.3 coMat

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

See also: tpMat, ajMat, MatMult.

11.1.4 MatMult

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

See also: tpMat, ajMat, coMat.

11.1.4.1 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*.

11.1.5 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.

11.1.6 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.

11.2 NCMatrixDecompositions

Members are:

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

11.2.1 NCLDLDecomposition

11.2.2 NCLeftDivide

11.2.3 NCLowerTriangularSolve

11.2.4 NCLUCompletePivoting

11.2.5 NCLUDecompositionWithCompletePivoting

11.2.6 NCLUDecompositionWithPartialPivoting

11.2.7 NCLUInverse

11.2.8 NCLUPartialPivoting

11.2.9 NCMatrixDecompositions

11.2.10 NCRightDivide

11.2.11 NCUpperTriangularSolve

11.3 MatrixDecompositions: linear algebra templates

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

11.3.1 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.

11.3.2 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.

11.3.3 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.

11.3.4 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.

11.3.5 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.

11.3.6 LUInverse

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

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

See also: LUDecompositionWithPartialPivoting.

11.3.7 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.

11.3.8 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.

11.3.9 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.

11.3.10 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.

11.3.11 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 12

Packages for pretty output, testing, and utilities

12.1 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

12.1.1 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:

- `NonCommutativeMultiply` (`False`): If `True` `x**y` is displayed as '`x • y`';
- `tp` (`True`): If `True` `tp[x]` is displayed as '$\mathtt{x}^{\mathtt{T}}$';
- `inv` (`True`): If `True` `inv[x]` is displayed as '$\mathtt{x}^{-1}$';
- `aj` (`True`): If `True` `aj[x]` is displayed as '$\mathtt{x}^{*}$';
- `co` (`True`): If `True` `co[x]` is displayed as '$\bar{\mathtt{x}}$';
- `rt` (`True`): If `True` `rt[x]` is displayed as '$\mathtt{x}^{1/2}$';
- `All`: Set all available options to `True` or `False`.

See also: NCTex, NCTexForm.

12.2 NCTeX

Members are:

- NCTeX
- NCRunDVIPS
- NCRunLaTeX
- NCRunPDFLaTeX
- NCRunPDFViewer

- NCRunPS2PDF

12.2.1 NCTeX

`NCTeX[expr]` typesets the LaTeX version of `expr` produced with TeXForm or NCTeXForm using LaTeX.

12.2.2 NCRunDVIPS

`NCRunDVIPS[file]` run dvips on `file`. Produces a ps output.

12.2.3 NCRunLaTeX

`NCRunLaTeX[file]` typesets the LaTeX `file` with latex. Produces a dvi output.

12.2.4 NCRunPDFLaTeX

`NCRunLaTeX[file]` typesets the LaTeX `file` with pdflatex. Produces a pdf output.

12.2.5 NCRunPDFViewer

`NCRunPDFViewer[file]` display pdf `file`.

12.2.6 NCRunPS2PDF

`NCRunPS2PDF[file]` run pd2pdf on `file`. Produces a pdf output.

12.3 NCTeXForm

Members are:

- NCTeXForm
- NCTeXFormSetStarStar

12.3.1 NCTeXForm

`NCTeXForm[expr]` prints a LaTeX version of `expr`.

The format is compatible with AMS-LaTeX.

Should work better than the Mathematica `TeXForm` :)

12.3.2 NCTeXFormSetStarStar

`NCTeXFormSetStarStar[string]` replaces the standard '**' for `string` in noncommutative multiplications.

For example:

`NCTeXFormSetStarStar["."]`

uses a dot (`.`) to replace `NonCommutativeMultiply`(`**`).

See also: NCTeXFormSetStar.

12.3.3 NCTeXFormSetStar

`NCTeXFormSetStar[string]` replaces the standard '*' for `string` in noncommutative multiplications.

For example:

`NCTeXFormSetStar[" "]`

uses a space (`') to replace`Times(*').

NCTeXFormSetStarStar.

12.4 NCRun

Members are:

- NCRun

12.4.1 NCRun

12.5 NCTest

Members are:

- NCTest
- NCTestRun
- NCTestSummarize

12.5.1 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

12.5.2 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

12.5.3 NCTestSummarize

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

See also: NCTestRun

Chapter 13

Data structures for fast calculations

NCPoly

13.1 Efficient storage of NC polynomials with rational coefficients

Members are:

- Constructors
  - NCPoly
  - NCPolyMonomial
  - NCPolyConstant
- Access and utilities
  - NCPolyMonomialQ
  - NCPolyDegree
  - NCPolyNumberOfVariables
  - NCPolyCoefficient
  - NCPolyGetCoefficients
  - NCPolyGetDigits
  - NCPolyGetIntegers
  - NCPolyLeadingMonomial
  - NCPolyLeadingTerm
  - NCPolyOrderType
  - NCPolyToRule
- Formatting
  - NCPolyDisplay
  - NCPolyDisplayOrder
- Arithmetic
  - NCPolyDivideDigits
  - NCPolyDivideLeading
  - NCPolyFullReduce
  - NCPolyNormalize
  - NCPolyProduct
  - NCPolyQuotientExpand
  - NCPolyReduce
  - NCPolySum
- State space realization
  - NCPolyHankelMatrix

    - NCPolyRealization (#NCPolyRealization)
- Auxiliary functions
    - NCFromDigits
    - NCIntegerDigits
    - NCDigitsToIndex
    - NCPadAndMatch

13.2 Ways to represent NC polynomials

13.2.1 NCPoly

`NCPoly[coeff, monomials, vars]` constructs a noncommutative polynomial object in variables `vars` where the monomials have coefficient `coeff`.

Monomials are specified in terms of the symbols in the list `vars` as in NCPolyMonomial.

For example:

```
vars = {x,y,z};
poly = NCPoly[{-1, 2}, {{x,y,x}, {z}}, vars];
```

constructs an object associated with the noncommutative polynomial $2z - xyx$ in variables `x`, `y` and `z`.

The internal representation varies with the implementation but it is so that the terms are sorted according to a degree-lexicographic order in `vars`. In the above example, `x < y < z`.

The construction:

```
vars = {{x},{y,z}};
poly = NCPoly[{-1, 2}, {{x,y,x}, {z}}, vars];
```

represents the same polyomial in a graded degree-lexicographic order in `vars`, in this example, `x << y < z`.

See also: NCPolyMonomial, NCIntegerDigits, NCFromDigits.

13.2.2 NCPolyMonomial

`NCPolyMonomial[monomial, vars]` constructs a noncommutative monomial object in variables `vars`.

Monic monomials are specified in terms of the symbols in the list `vars`, for example:

```
vars = {x,y,z};
mon = NCPolyMonomial[{x,y,x},vars];
```

returns an `NCPoly` object encoding the monomial xyx in noncommutative variables `x`,`y`, and `z`. The actual representation of `mon` varies with the implementation.

Monomials can also be specified implicitly using indices, for example:

```
mon = NCPolyMonomial[{0,1,0}, 3];
```

also returns an `NCPoly` object encoding the monomial xyx in noncommutative variables `x`,`y`, and `z`.

If graded ordering is supported then

```
vars = {{x},{y,z}};
mon = NCPolyMonomial[{x,y,x},vars];
```

or

```
mon = NCPolyMonomial[{0,1,0}, {1,2}];
```

construct the same monomial xyx in noncommutative variables `x`,`y`, and `z` this time using a graded order in which `x << y < z`.

There is also an alternative syntax for `NCPolyMonomial` that allows users to input the monomial along with a coefficient using rules and the output of NCFromDigits. For example:

```
mon = NCPolyMonomial[{3, 3} -> -2, 3];
```

or

```
mon = NCPolyMonomial[NCFromDigits[{0,1,0}, 3] -> -2, 3];
```

represent the monomial $-2xyx$ with has coefficient `-2`.

See also: NCPoly, NCIntegerDigits, NCFromDigits.

13.2.3 NCPolyConstant

`NCPolyConstant[value, vars]` constructs a noncommutative monomial object in variables `vars` representing the constant `value`.

For example:

```
NCPolyConstant[3, {x, y, z}]
```

constructs an object associated with the constant `3` in variables `x`, `y` and `z`.

See also: NCPoly, NCPolyMonomial.

13.3 Access and utlity functions

13.3.1 NCPolyMonomialQ

`NCPolyMonomialQ[poly]` returns `True` if `poly` is a `NCPoly` monomial.

See also: NCPoly, NCPolyMonomial.

13.3.2 NCPolyDegree

`NCPolyDegree[poly]` returns the degree of the nc polynomial `poly`.

13.3.3 NCPolyNumberOfVariables

`NCPolyNumberOfVariables[poly]` returns the number of variables of the nc polynomial `poly`.

13.3.4 NCPolyCoefficient

`NCPolyCoefficient[poly, mon]` returns the coefficient of the monomial `mon` in the nc polynomial `poly`.

For example, in:

```
coeff = {1, 2, 3, -1, -2, -3, 1/2};
mon = {{}, {x}, {z}, {x, y}, {x, y, x, x}, {z, x}, {z, z, z, z}};
vars = {x,y,z};
poly = NCPoly[coeff, mon, vars];
```

```
c = NCPolyCoefficient[poly, NCPolyMonomial[{x,y},vars]];
```

returns

```
c = -1
```

See also: NCPoly, NCPolyMonomial.

13.3.5 NCPolyGetCoefficients

`NCPolyGetCoefficients[poly]` returns a list with the coefficients of the monomials in the nc polynomial `poly`.

For example:

```
vars = {x,y,z};
poly = NCPoly[{-1, 2}, {{x,y,x}, {z}}, vars];
coeffs = NCPolyGetCoefficients[poly];
```

returns

```
coeffs = {2,-1}
```

The coefficients are returned according to the current graded degree-lexicographic ordering, in this example `x < y < z`.

See also: NCPolyGetDigits, NCPolyCoefficient, NCPoly.

13.3.6 NCPolyGetDigits

`NCPolyGetDigits[poly]` returns a list with the digits that encode the monomials in the nc polynomial `poly` as produced by NCIntegerDigits.

For example:

```
vars = {x,y,z};
poly = NCPoly[{-1, 2}, {{x,y,x}, {z}}, vars];
digits = NCPolyGetDigits[poly];
```

returns

```
digits = {{2}, {0,1,0}}
```

The digits are returned according to the current ordering, in this example `x < y < z`.

See also: NCPolyGetCoefficients, NCPoly.

13.3.7 NCPolyGetIntegers

`NCPolyGetIntegers[poly]` returns a list with the digits that encode the monomials in the nc polynomial `poly` as produced by NCFromDigits.

For example:

```
vars = {x,y,z};
poly = NCPoly[{-1, 2}, {{x,y,x}, {z}}, vars];
digits = NCPolyGetIntegers[poly];
```

returns

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

The digits are returned according to the current ordering, in this example `x < y < z`.

See also: NCPolyGetCoefficients, NCPoly.

13.3.8 NCPolyLeadingMonomial

`NCPolyLeadingMonomial[poly]` returns an `NCPoly` representing the leading term of the nc polynomial `poly`.

For example:

```
vars = {x,y,z};
poly = NCPoly[{-1, 2}, {{x,y,x}, {z}}, vars];
lead = NCPolyLeadingMonomial[poly];
```

returns an `NCPoly` representing the monomial xyx. The leading monomial is computed according to the current ordering, in this example `x < y < z`. The actual representation of `lead` varies with the implementation.

See also: NCPolyLeadingTerm, NCPolyMonomial, NCPoly.

13.3.9 NCPolyLeadingTerm

`NCPolyLeadingTerm[poly]` returns a rule associated with the leading term of the nc polynomial `poly` as understood by NCPolyMonomial.

For example:

```
vars = {x,y,z};
poly = NCPoly[{-1, 2}, {{x,y,x}, {z}}, vars];
lead = NCPolyLeadingTerm[poly];
```

returns

```
lead = {3,3} -> -1
```

representing the monomial $-xyx$. The leading monomial is computed according to the current ordering, in this example `x < y < z`.

See also: NCPolyLeadingMonomial, NCPolyMonomial, NCPoly.

13.3.10 NCPolyOrderType

`NCPolyOrderType[poly]` returns the type of monomial order in which the nc polynomial `poly` is stored. Order can be `NCPolyGradedDegLex` or `NCPolyDegLex`.

See also: NCPoly,

13.3.11 NCPolyToRule

`NCPolyToRule[poly]` returns a `Rule` associated with polynomial `poly`. If `poly = lead + rest`, where `lead` is the leading term in the current order, then `NCPolyToRule[poly]` returns the rule `lead -> -rest` where the coefficient of the leading term has been normalized to `1`.

For example:

```
vars = {x, y, z};
poly = NCPoly[{-1, 2, 3}, {{x, y, x}, {z}, {x, y}}, vars];
rule = NCPolyToRule[poly]
```

returns the rule `lead -> rest` where `lead` represents is the nc monomial xyx and `rest` is the nc polynomial $2z + 3xy$

See also: NCPolyLeadingTerm, NCPolyLeadingMonomial, NCPoly.

13.4 Formating functions

13.4.1 NCPolyDisplay

`NCPolyDisplay[poly]` prints the noncommutative polynomial `poly`.

`NCPolyDisplay[poly, vars]` uses the symbols in the list `vars`.

13.4.2 NCPolyDisplayOrder

`NCPolyDisplayOrder[vars]` prints the order implied by the list of variables `vars`.

13.5 Arithmetic functions

13.5.1 NCPolyDivideDigits

`NCPolyDivideDigits[F,G]` returns the result of the division of the leading digits lf and lg.

13.5.2 NCPolyDivideLeading

`NCPolyDivideLeading[lF,lG,base]` returns the result of the division of the leading Rules lf and lg as returned by NCGetLeadingTerm.

13.5.3 NCPolyFullReduce

`NCPolyFullReduce[f,g]` applies NCPolyReduce successively until the remainder does not change. See also NCPolyReduce and NCPolyQuotientExpand.

13.5.4 NCPolyNormalize

`NCPolyNormalize[poly]` makes the coefficient of the leading term of `p` to unit. It also works when `poly` is a list.

13.5.5 NCPolyProduct

`NCPolyProduct[f,g]` returns a NCPoly that is the product of the NCPoly's f and g.

13.5.6 NCPolyQuotientExpand

`NCPolyQuotientExpand[q,g]` returns a NCPoly that is the left-right product of the quotient as returned by NCPolyReduce by the NCPoly g. It also works when g is a list.

13.5.7 NCPolyReduce

13.5.8 NCPolySum

`NCPolySum[f,g]` returns a NCPoly that is the sum of the NCPoly's f and g.

13.6 State space realization functions

13.6.1 NCPolyHankelMatrix

`NCPolyHankelMatrix[poly]` produces the nc *Hankel matrix* associated with the polynomial `poly` and also their shifts per variable.

For example:

```
vars = {{x, y}};
poly = NCPoly[{1, -1}, {{x, y}, {y, x}}, vars];
{H, Hx, Hy} = NCPolyHankelMatrix[poly]
```

results in the matrices

```
H =  {{  0,  0,  0,  1, -1 },
      {  0,  0,  1,  0,  0 },
      {  0, -1,  0,  0,  0 },
      {  1,  0,  0,  0,  0 },
      { -1,  0,  0,  0,  0 }}
Hx = {{  0,  0,  1,  0,  0 },
      {  0,  0,  0,  0,  0 },
      { -1,  0,  0,  0,  0 },
      {  0,  0,  0,  0,  0 },
      {  0,  0,  0,  0,  0 }}
Hy = {{  0, -1,  0,  0,  0 },
      {  1,  0,  0,  0,  0 },
      {  0,  0,  0,  0,  0 },
      {  0,  0,  0,  0,  0 },
      {  0,  0,  0,  0,  0 }}
```

which are the Hankel matrices associated with the commutator $xy - yx$.

See also: NCPolyRealization, NCIntegerToIndex.

13.6.2 NCPolyRealization

`NCPolyRealization[poly]` calculate a minimal descriptor realization for the polynomial `poly`.

`NCPolyRealization` uses `NCPolyHankelMatrix` and the resulting realization is compatible with the format used by `NCRational`.

For example:

```
vars = {{x, y}};
poly = NCPoly[{1, -1}, {{x, y}, {y, x}}, vars];
{{a0,ax,ay},b,c,d} = NCPolyRealization[poly]
```

produces a list of matrices `{a0,ax,ay}`, a column vector `b` and a row vector `c`, and a scalar `d` such that $c.inv[a0 + ax\, x + ay\, y].b + d = xy - yx$.

See also: NCPolyHankelMatrix, NCRational.

13.7 Auxiliary functions

13.7.1 NCFromDigits

`NCFromDigits[list, b]` constructs a representation of a monomial in `b` encoded by the elements of `list` where the digits are in base `b`.

`NCFromDigits[{list1,list2}, b]` applies `NCFromDigits` to each `list1`, `list2`, ...

List of integers are used to codify monomials. For example the list `{0,1}` represents a monomial xy and the list `{1,0}` represents the monomial yx. The call

```
NCFromDigits[{0,0,0,1}, 2]
```

returns

```
{4,1}
```

in which `4` is the degree of the monomial $xxxy$ and `1` is `0001` in base `2`. Likewise

```
NCFromDigits[{0,2,1,1}, 3]
```

returns

```
{4,22}
```

in which `4` is the degree of the monomial $xzyy$ and `22` is `0211` in base `3`.

If `b` is a list, then degree is also a list with the partial degrees of each letters appearing in the monomial. For example:

```
NCFromDigits[{0,2,1,1}, {1,2}]
```

returns

```
{3, 1, 22}
```

in which `3` is the partial degree of the monomial $xzyy$ with respect to letters `y` and `z`, `1` is the partial degree with respect to letter `x` and `22` is `0211` in base `3 = 1 + 2`.

This construction is used to represent graded degree-lexicographic orderings.

See also: NCIntergerDigits.

13.7.2 NCIntegerDigits

`NCIntegerDigits[n,b]` is the inverse of the `NCFromDigits`.

`NCIntegerDigits[{list1,list2}, b]` applies `NCIntegerDigits` to each `list1`, `list2`, ...

For example:

```
NCIntegerDigits[{4,1}, 2]
```

returns

```
{0,0,0,1}
```

in which `4` is the degree of the monomial `x**x**x**y` and `1` is `0001` in base `2`. Likewise

```
NCIntegerDigits[{4,22}, 3]
```

returns

```
{0,2,1,1}
```

in which `4` is the degree of the monomial `x**z**y**y` and `22` is `0211` in base `3`.

If `b` is a list, then degree is also a list with the partial degrees of each letters appearing in the monomial. For example:

```
NCIntegerDigits[{3, 1, 22}, {1,2}]
```

returns

```
{0,2,1,1}
```

in which `3` is the partial degree of the monomial `x**z**y**y` with respect to letters `y` and `z`, `1` is the partial degree with respect to letter `x` and `22` is `0211` in base `3 = 1 + 2`.

See also: NCFromDigits.

13.7.3 NCDigitsToIndex

`NCDigitsToIndex[digits, b]` returns the index that the monomial represented by `digits` in the base `b` would occupy in the standard monomial basis.

`NCDigitsToIndex[{digit1,digits2}, b]` applies `NCDigitsToIndex` to each `digit1`, `digit2`, ...

`NCDigitsToIndex` returns the same index for graded or simple basis.

For example:

```
digits = {0, 1};
NCDigitsToIndex[digits, 2]
NCDigitsToIndex[digits, {2}]
NCDigitsToIndex[digits, {1, 1}]
```

all return

```
5
```

which is the index of the monomial xy in the standard monomial basis of polynomials in x and y. Likewise

```
digits = {{}, {1}, {0, 1}, {0, 2, 1, 1}};
NCDigitsToIndex[digits, 2]
```

returns

```
{1,3, 5,27}
```

See also: NCFromDigits, NCIntergerDigits.

13.7.4 NCPadAndMatch

When list `a` is longer than list `b`, `NCPadAndMatch[a,b]` returns the minimum number of elements from list a that should be added to the left and right of list `b` so that `a = l b r`. When list `b` is longer than list `a`, return the opposite match.

`NCPadAndMatch` returns all possible matches with the minimum number of elements.

NCPolynomial

13.8 Efficient storage of NC polynomials with nc coefficients

This package contains functionality to convert an nc polynomial expression into an expanded efficient representation that 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

13.9 Ways to represent NC polynomials

13.9.1 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.

13.9.2 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.

13.9.3 NCPolynomialToNC

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

See also: NCPolynomial, NCToNCPolynomial.

13.9.4 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`.

13.10 Grouping terms by degree

13.10.1 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`.

13.10.2 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`.

13.10.3 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`.

13.11 Utilities

13.11.1 NCPDegree

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

See also: `NCPMonomialDegree`.

13.11.2 NCPMonomialDegree

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

See also: `NCDegree`.

13.11.3 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`.

13.11.4 NCPLinearQ

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

See also: `NCPQuadraticQ`.

13.11.5 NCPQuadraticQ

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

See also: `NCPLinearQ`.

13.11.6 NCPCompatibleQ

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

See also: NCPSameVariablesQ, NCPMatrixQ.

13.11.7 NCPSameVariablesQ

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

See also: NCPCompatibleQ, NCPMatrixQ.

13.11.8 NCPMatrixQ

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

See also: NCPCompatibleQ.

13.11.9 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`

13.12 Operations on NC polynomials

13.12.1 NCPPlus

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

13.12.2 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.

13.12.3 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.

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

NCUtil

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

Members are:

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

17.1 NCConsistentQ

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

17.2 NCGrabFunctions

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

`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.

17.3 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.

17.4 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.

17.5 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`

17.6 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`.

17.7 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.

17.8 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 18

NCSDP

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

Members are:

- NCSDP
- NCSDPForm
- NCSDPDual
- NCSDPDualForm

18.1 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.

18.2 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.

18.3 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.

18.4 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 19

SDP

SDP is a package that provides algorithms for the numeric solution of semidefinite programs.

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

19.1 SDPMatrices

19.2 SDPSolve

19.3 SDPEval

19.4 SDPInner

19.5 SDPCheckDimensions

19.6 SDPDualEval

19.7 SDPFunctions

19.8 SDPPrimalEval

19.9 SDPScale

19.10 SDPSylvesterDiagonalEval

19.11 SDPSylvesterEval

Chapter 20

NCGBX

Members are:

- NCToNCPoly
- NCPolyToNC
- NCRuleToPoly
- SetMonomialOrder
- SetKnowns
- SetUnknowns
- ClearMonomialOrder
- GetMonomialOrder
- PrintMonomialOrder
- NCMakeGB
- NCReduce

20.1 NCToNCPoly

`NCToNCPoly[expr, var]` constructs a noncommutative polynomial object in variables `var` from the nc expression `expr`.

For example

```
NCToNCPoly[x**y - 2 y**z, {x, y, z}]
```

constructs an object associated with the noncommutative polynomial $xy - 2yz$ in variables `x`, `y` and `z`. The internal representation is so that the terms are sorted according to a degree-lexicographic order in `vars`. In the above example, $x < y < z$.

20.2 NCPolyToNC

`NCPolyToNC[poly, vars]` constructs an nc expression from the noncommutative polynomial object `poly` in variables `vars`. Monomials are specified in terms of the symbols in the list `var`.

For example

```
poly = NCToNCPoly[x**y - 2 y**z, {x, y, z}];
expr = NCPolyToNC[poly, {x, y, z}];
```

returns

```
expr = x**y - 2 y**z
```

See also: NCPolyToNC, NCPoly.

20.3 NCRuleToPoly

20.4 SetMonomialOrder

`SetMonomialOrder[var1, var2, ...]` sets the current monomial order.

For example

```
SetMonomialOrder[a,b,c]
```

sets the lex order $a \ll b \ll c$.

If one uses a list of variables rather than a single variable as one of the arguments, then multigraded lex order is used. For example

```
SetMonomialOrder[{a,b,c}]
```

sets the graded lex order $a < b < c$.

Another example:

```
SetMonomialOrder[{{a, b}, {c}}]
```

or

```
SetMonomialOrder[{a, b}, c]
```

set the multigraded lex order $a < b \ll c$.

Finally

```
SetMonomialOrder[{a,b}, {c}, {d}]
```

or

```
SetMonomialOrder[{a,b}, c, d]
```

is equivalent to the following two commands

```
SetKnowns[a,b]
SetUnknowns[c,d]
```

There is also an older syntax which is still supported:

```
SetMonomialOrder[{a, b, c}, n]
```

sets the order of monomials to be $a < b < c$ and assigns them grading level `n`.

```
SetMonomialOrder[{a, b, c}, 1]
```

is equivalent to `SetMonomialOrder[{a, b, c}]`. When using this older syntax the user is responsible for calling ClearMonomialOrder to make sure that the current order is empty before starting.

See also: ClearMonomialOrder, GetMonomialOrder, PrintMonomialOrder, SetKnowns, SetUnknowns.

20.5 SetKnowns

`SetKnowns[var1, var2, ...]` records the variables `var1`, `var2`, ... to be corresponding to known quantities.

`SetUnknowns` and `Setknowns` prescribe a monomial order with the knowns at the the bottom and the unknowns at the top.

For example

```
SetKnowns[a,b]
SetUnknowns[c,d]
```

is equivalent to

```
SetMonomialOrder[{a,b}, {c}, {d}]
```

which corresponds to the order $a < b \ll c \ll d$ and

```
SetKnowns[a,b]
SetUnknowns[{c,d}]
```

is equivalent to

```
SetMonomialOrder[{a,b}, {c, d}]
```

which corresponds to the order $a < b \ll c < d$.

Note that `SetKnowns` flattens grading so that

```
SetKnowns[a,b]
```

and

```
SetKnowns[{a},{b}]
```

result both in the order $a < b$.

Successive calls to `SetUnknowns` and `SetKnowns` overwrite the previous knowns and unknowns. For example

```
SetKnowns[a,b]
SetUnknowns[c,d]
SetKnowns[c,d]
SetUnknowns[a,b]
```

results in an ordering $c < d \ll a \ll b$.

See also: SetUnknowns, SetMonomialOrder.

20.6 SetUnknowns

`SetUnknowns[var1, var2, ...]` records the variables `var1`, `var2`, ... to be corresponding to unknown quantities.

`SetUnknowns` and `SetKnowns` prescribe a monomial order with the knowns at the the bottom and the unknowns at the top.

For example

```
SetKnowns[a,b]
SetUnknowns[c,d]
```

is equivalent to

```
SetMonomialOrder[{a,b}, {c}, {d}]
```

which corresponds to the order $a < b \ll c \ll d$ and

```
SetKnowns[a,b]
SetUnknowns[{c,d}]
```

is equivalent to

```
SetMonomialOrder[{a,b}, {c, d}]
```

which corresponds to the order $a < b \ll c < d$.

Note that `SetKnowns` flattens grading so that

```
SetKnowns[a,b]
```

and

```
SetKnowns[{a},{b}]
```

result both in the order $a < b$.

Successive calls to `SetUnknowns` and `SetKnowns` overwrite the previous knowns and unknowns. For example

```
SetKnowns[a,b]
SetUnknowns[c,d]
SetKnowns[c,d]
SetUnknowns[a,b]
```

results in an ordering $c < d \ll a \ll b$.

See also: SetKnowns, SetMonomialOrder.

20.7 ClearMonomialOrder

`ClearMonomialOrder[]` clear the current monomial ordering.

It is only necessary to use `ClearMonomialOrder` if using the indexed version of `SetMonomialOrder`.

See also: SetKnowns, SetUnknowns, SetMonomialOrder, ClearMonomialOrder, PrintMonomialOrder.

20.8 GetMonomialOrder

`GetMonomialOrder[]` returns the current monomial ordering in the form of a list.

For example

```
SetMonomialOrder[{a,b}, {c}, {d}]
order = GetMonomialOrder[]
```

returns

```
order = {{a,b},{c},{d}}
```

See also: SetKnowns, SetUnknowns, SetMonomialOrder, ClearMonomialOrder, PrintMonomialOrder.

20.9 PrintMonomialOrder

`PrintMonomialOrder[]` prints the current monomial ordering.

For example

```
SetMonomialOrder[{a,b}, {c}, {d}]
PrintMonomialOrder[]
```

print $a < b \ll c \ll d$.

See also: SetKnowns, SetUnknowns, SetMonomialOrder, ClearMonomialOrder, PrintMonomialOrder.

20.10 NCMakeGB

`NCMakeGB[{poly1, poly2, ...}, k]` attempts to produces a nc Gröbner Basis (GB) associated with the list of nc polynomials `{poly1, poly2, ...}`. The GB algorithm proceeds through *at most* `k` iterations until a Gröbner basis is found for the given list of polynomials with respect to the order imposed by SetMonomialOrder.

If `NCMakeGB` terminates before finding a GB the message `NCMakeGB::Interrupted` is issued.

The output of `NCMakeGB` is a list of rules with left side of the rule being the *leading* monomial of the polynomials in the GB.

For example:

```
SetMonomialOrder[x];
gb = NCMakeGB[{x^2 - 1, x^3 - 1}, 20]
```

returns

```
gb = {x -> 1}
```

that corresponds to the polynomial $x - 1$, which is the nc Gröbner basis for the ideal generated by $x^2 - 1$ and $x^3 - 1$.

`NCMakeGB[{poly1, poly2, ...}, k, options]` uses `options`.

The following `options` can be given:

- `SimplifyObstructions` (`True`): control whether obstructions are simplified before being added to the list of active obstructions;
- `SortObstructions` (`False`): control whether obstructions are sorted before being processed;
- `SortBasis` (`False`): control whether initial basis is sorted before initiating algorithm;
- `VerboseLevel` (`1`): control level of verbosity from `0` (no messages) to `5` (very verbose);
- `PrintBasis` (`False`): if `True` prints current basis at each major iteration;
- `PrintObstructions` (`False`): if `True` prints current list of obstructions at each major iteration;
- `PrintSPolynomials` (`False`): if `True` prints every S-polynomial formed at each minor iteration.

`NCMakeGB` makes use of the algorithm `NCPolyGroebner` implemented in NCPolyGroeber.

See also: ClearMonomialOrder, GetMonomialOrder, PrintMonomialOrder, SetKnowns, SetUnknowns, NCPolyGroebner.

20.11 NCReduce

`NCAutomaticOrder[ aMonomialOrder, aListOfPolynomials ]`

This command assists the user in specifying a monomial order. It inserts all of the indeterminants found in $aListOfPolynomials$ into the monomial order. If x is an indeterminant found in $aMonomialOrder$ then any indeterminant whose symbolic representation is a function of x will appear next to x. For example, NCAutomaticOrder[{{a},{b}},{ a**Inv[a]**tp[a] + tp[b]}] would set the order to be $a < tp[a] < Inv[a] \ll b < tp[b]$.} {A list of indeterminants which specifies the general order. A list of polynomials which will make up the input to the Gröbner basis command.} {If tp[Inv[a]] is found after Inv[a] NCAutomaticOrder[ ] would generate the order $a < tp[Inv[a]] < Inv[a]$. If the variable is self-adjoint (the input contains the relation $ tp[Inv[a]] == Inv[a]$) we would have the rule, $Inv[a] \to tp[Inv[a]]$, when the user would probably prefer $tp[Inv[a]] \to Inv[a]$.}

Chapter 21

NCPolyGroebner

Members are:

- NCPolyGroebner

21.1 NCPolyGroebner

`NCPolyGroebner[G]` computes the noncommutative Groebner basis of the list of `NCPoly` polynomials `G`.

`NCPolyGroebner[G, options]` uses `options`.

The following `options` can be given:

- `SimplifyObstructions` (`True`) whether to simplify obstructions before constructions S-polynomials;
- `SortObstructions` (`False`) whether to sort obstructions using Mora's SUGAR ranking;
- `SortBasis` (`False`) whether to sort basis before starting algorithm;
- `Labels` (`{}`) list of labels to use in verbose printing;
- `VerboseLevel` (`1`): function used to decide if a pivot is zero;
- `PrintBasis` (`False`): function used to divide a vector by an entry;
- `PrintObstructions` (`False`);
- `PrintSPolynomials` (`False`);

The algorithm is based on T. Mora, "An introduction to commuative and noncommutative Groebner Bases," *Theoretical Computer Science*, v. 134, pp. 131-173, 2000.

See also: NCPoly.

Part III

Work in Progress

Chapter 22

Introduction

Chapters in this part describe experimental packages which are still under development.

Chapter 23

NCRational

This package contains functionality to convert an nc rational expression into a descriptor representation.

For example the rational

```
exp = 1 + inv[1 + x]
```

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:

- NCRational
- NCToNCRational
- NCRationalToNC
- NCRationalToCanonical
- CanonicalToNCRational
- NCROrder
- NCRLinearQ
- NCRStrictlyProperQ
- NCRPlus
- NCRTimes
- NCRTranspose
- NCRInverse
- NCRControllableSubspace
- NCRControllableRealization
- NCRObservableRealization
- NCRMinimalRealization

23.1 State-space realizations for NC rationals

23.1.1 NCRational

NCRational::usage

23.1.2 NCToNCRational

NCToNCRational::usage

23.1.3 NCRationalToNC

NCRationalToNC::usage

23.1.4 NCRationalToCanonical

NCRationalToCanonical::usage

23.1.5 CanonicalToNCRational

CanonicalToNCRational::usage

23.2 Utilities

23.2.1 NCROrder

NCROrder::usage

23.2.2 NCRLinearQ

NCRLinearQ::usage

23.2.3 NCRStrictlyProperQ

NCRStrictlyProperQ::usage

23.3 Operations on NC rationals

23.3.1 NCRPlus

NCRPlus::usage

23.3.2 NCRTimes

NCRTimes::usage

23.3.3 NCRTranspose

NCRTranspose::usage

23.3.4 NCRInverse

NCRInverse::usage

23.4 Minimal realizations

23.4.1 NCRControllableRealization

NCRControllableRealization::usage

23.4.2 NCRControllableSubspace

NCRControllableSubspace::usage

23.4.3 NCRObservableRealization

NCRObservableRealization::usage

23.4.4 NCRMinimalRealization

NCRMinimalRealization::usage

Chapter 24

NCRealization

WARNING: OBSOLETE PACKAGE WILL BE REPLACED BY NCRational

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

24.1 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`.

24.2 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.

24.3 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.

24.4 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).

24.5 NCSymmetricDescriptorRealization

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

24.6 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,. . .

24.7 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.

24.8 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.

24.9 NonCommutativeLift

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

24.10 SignatureOfAffineTerm

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

24.11 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.

24.12 PinnedQ

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

24.13 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).