CAD via Projection and Lifting (V2)
For Maple 16 / 17
Matthew England (University of Bath)
8th June 2013

```
> restart:
```

This file is an introduction to the Maple package "ProjectionCAD". This worksheet refers to the second release of the code.
The code uses work in the RegularChains library and outputs CADs in similar formats. Hence it is useful (but not necessary) to load RegularChains.

```
> with(RegularChains):
```

The code works with Maple 16 and Maple 17. Earlier versions have not been tested. We read in the code.

```
> read("ProjectionCAD.mpl"):
  with(ProjectionCAD);
```

"This is V2.6 of the ProjectionCAD module from 24th May 2013, designed for use in Maple 16 / 17"

[*CADFull*, *CADGenerateStack*, *CADLifting*, *CADProjection*, *ECCAD*, *ECCADFormulations*, *ECCADHeuristic*, *ECCADProjFactors*, *ECCADProjOp*, *NumCellsInCAD*, *NumCellsInPiecewiseCAD*, *TTICAD*, *TTICADFormulations*, *TTICADHeuristic*, *TTICADProjFactors*, *TTICADProjOp*, *TTICADQFFFormulations*, *TTICADQFFHeuristic*, *TTICADResCAD*, *TTICADResCADSet*, *VariableOrderingHeuristic*, *VariableOrderings*, *ndrr*, *sotd*] **(1)**

The code in this package is an implementation of algorithms for constructing various types of Cylindrical Algebraic Decomposition (CAD) via projection and lifting. This includes CADs that are: sign-invariant, order-invariant, sign-invariant over an equational constraint and truth-table invariant. There are also tools for considering the different formulations for these algorithms and heuristics for making these choices.

We note that there is already a command in Maple to produce CADs:
RegularChains:-SemiAlgebraicSetTools:-CylindricalAlgebraicDecompose
However, this uses a different approach (triangular decomposition) and currently (as of Maple 17) only offers sign-invariant CADs.

We introduce the main commands of our module in the sections below. Note that details for an individual command can be obtained using **Describe**:

```
> Describe(NumCellsInPiecewiseCAD);
```

```
# NumCellsInPiecewiseCAD: Calculate the number of cells in a
MapleCAD given in
# the piecewise output format.
# Input: A CAD in piecewise format. This could be output from
either this
# module or the Regular Chains implementation.
# Output: The number of cells in the CAD.
NumCellsInPiecewiseCAD( pwcad::piecewise, $ ) :: posint
```

CONTENTS:

We start by introducing the code relevant to producing sign-invariant CADs.

1. Sign-invariant CADs

Given polynomials and a variable ordering **CADFull** will construct a sign-invariant CAD using either Collins or McCallum projection (default is McCallum) and the corresponding lifting algorithm.

For example:

```
> f:=y^3+y^2+y*x^2-1;
```

$$f := y\,x^2 + y^3 + y^2 - 1 \tag{1.1}$$

```
> CADFull( {f}, [y,x], method=Collins): nops(%);
  CADFull( {f}, [y,x], method=McCallum): nops(%);
```

$$15$$
$$3 \tag{1.2}$$

The projection and lifting steps may be performed separately if desired. The projection factors may be calculated using:

```
> psetC:=CADProjection( {f}, [y,x], method=Collins );
  psetM:=CADProjection( {f}, [y,x], method=McCallum );
```

$$psetC := \{x^2 + 1,\ 3\,x^2 - 1,\ 4\,x^4 - 5\,x^2 + 23,\ y^3 + y\,x^2 + y^2 - 1\}$$
$$psetM := \{x^2 + 1,\ 4\,x^4 - 5\,x^2 + 23,\ y^3 + y\,x^2 + y^2 - 1\} \tag{1.3}$$

Then we can lift with respect to these using:

```
> CADLifting( psetC, [y,x], method=Collins): nops(%);
```

$$15 \tag{1.4}$$

Note that if you were to lift a set of projections polynomials from one algorithm with the other then the output is no longer guaranteed.

2. CAD output formats

There are currently four style of output format for CADs: ***list***, ***listwithrep***, ***rootof*** and ***piecewise***. The default is list, which gives a list of cells, each containing an index and a samplepoint. Note that this is the same format as RegularChains:-SemiAlgebraicSetTools:-CylindricalAlgebraicDecompose with output=list.

```
> f:=y^3+y^2+y*x^2-1: cadl:=CADFull( {f}, [y,x], method=McCallum,
  output=list ); nops(%);
```

$$cadl := \left[[[1,1],[regular\_chain,[[0,0],[-1,-1]]]],\left[[1,2],\left[regular\_chain,\left[[0,0],\left[\frac{3}{4},1\right]\right]\right]\right],[[1,3],[regular\_chain,[[0,0],[2,2]]]]\right]$$

3 **(2.1)**

The choice output=rootof will give a list of cells, each represented by conditions on the variables. Note that this is the same format as RegularChains:-SemiAlgebraicSetTools:-CylindricalAlgebraicDecompose with output=rootof.

```
> cadr:=CADFull( {f}, [y,x], method=McCallum, output=rootof );
  nops(%);
```

$$cadr := \left[\left[x=x, y<RootOf\left(\_Z^3+\_Z x^2+\_Z^2-1, index=real_1\right)\right],\left[x=x, y=RootOf\left(\_Z^3+\_Z x^2+\_Z^2-1, index=real_1\right)\right],\left[x=x, RootOf\left(\_Z^3+\_Z x^2+\_Z^2-1, index=real_1\right)<y\right]\right]$$

3 **(2.2)**

The choice output=listwithrep will give a list of cells which contain all the information from the previous two formats.

```
> cadlwr:=CADFull( {f}, [y,x], method=McCallum, output=
  listwithrep ); nops(%);
```

$$cadlwr := \left[\left[[1,1],\left[x=x, y<RootOf\left(\_Z^3+\_Z x^2+\_Z^2-1, index=real_1\right)\right]\right],[regular\_chain,[[0,0],[-1,-1]]]\right],\left[[1,2],\left[x=x, y=RootOf\left(\_Z^3+\_Z x^2+\_Z^2-1, index=real_1\right)\right],\left[regular\_chain,\left[[0,0],\left[\frac{3}{4},1\right]\right]\right]\right],\left[[1,3],\left[x=x, RootOf\left(\_Z^3+\_Z x^2+\_Z^2-1, index=real_1\right)<y\right],[regular\_chain,[[0,0],[2,2]]]\right]\right]$$

3 **(2.3)**

The choice output=piecewise will show the CAD arranged cylindrically using Maple's piecewise construction.

```
> cadp:=CADFull( {f}, [y,x], method=McCallum, output=piecewise );
```

(2.4

$$cadp := \begin{cases} [regular\_chain, [[0,0],[-1,-1]]] & y < RootOf\left(\_Z^3 + \_Z x^2 + \_Z^2 - 1, index = real_1\right) \\ \left[regular\_chain, \left[[0,0], \left[\frac{3}{4}, 1\right]\right]\right] & y = RootOf\left(\_Z^3 + \_Z x^2 + \_Z^2 - 1, index = real_1\right) \\ [regular\_chain, [[0,0],[2,2]]] & RootOf\left(\_Z^3 + \_Z x^2 + \_Z^2 - 1, index = real_1\right) < y \end{cases} \quad (2.4$$

This format is comparable with RegularChains:-SemiAlgebraicSetTools:-CylindricalAlgebraicDecompose with output=piecewise, however, here radicals are used instead of RootOf constructions for easier examples.

```
> CADFull( {x^2+y^2-1}, [y,x], method=McCallum, output=piecewise
  );
```

$$\begin{cases} [regular\_chain, [[-2,-2],[0,0]]] & x < -1 \\ \begin{cases} [regular\_chain, [[-1,-1],[-1,-1]]] & y < 0 \\ [regular\_chain, [[-1,-1],[0,0]]] & y = 0 \\ [regular\_chain, [[-1,-1],[1,1]]] & 0 < y \end{cases} & x = -1 \\ \begin{cases} [regular\_chain, [[0,0],[-2,-2]]] & y < -\sqrt{-x^2+1} \\ [regular\_chain, [[0,0],[-1,-1]]] & y = -\sqrt{-x^2+1} \\ [regular\_chain, [[0,0],[0,0]]] & \text{And}\left(-\sqrt{-x^2+1} < y, y < \sqrt{-x^2+1}\right) \\ [regular\_chain, [[0,0],[1,1]]] & y = \sqrt{-x^2+1} \\ [regular\_chain, [[0,0],[2,2]]] & \sqrt{-x^2+1} < y \end{cases} & \text{And}(-1 < x, x < \\ \begin{cases} [regular\_chain, [[1,1],[-1,-1]]] & y < 0 \\ [regular\_chain, [[1,1],[0,0]]] & y = 0 \\ [regular\_chain, [[1,1],[1,1]]] & 0 < y \end{cases} & x = 1 \\ [regular\_chain, [[2,2],[0,0]]] & 1 < x \end{cases}$$

Note that we cannot measure the number of cells in a piecewise cad using the nops functions. Use the function ***CADNumCellsInPiecewise*** instead

```
> nops(cadp); NumCellsInPiecewiseCAD(cadp);
```

$$6$$
$$3 \quad (2.6)$$

3. Examing failure (userinfo levels)

Collins' algorithm will always succeed (given sufficient time and memory).
McCallum's algorithm can fail when the input polynomials are not well oriented.

McCallum's algorithm has been adapted here to maximise success by guaranteeing only a sign-invariant CAD (rather than order-invariant) as a final output.
Hence to give an example of failure we need at least 5-dimensions .
(This is because 2d is always OI, 3d can only fail on dim zero cell for which we have McCallum's delineating polynomial method and 4d can still be guaranteed sign-invariant.)
A trivial example of failure:

```
> f:=a*e+b*d+c*e+d+e;
```

$$f := a\,e + b\,d + c\,e + d + e \tag{3.1}$$

```
> CADFull( {f}, [a,b,c,d,e], method=McCallum  ): nops(%);
Warning, there is nullification on cell [1, 2, 2] which has
dim>0. The nullified polynomial was discarded. The outputted
CAD may not be sign-invariant.
Warning, there is nullification on cell [2, 2, 1] which has
dim>0. The nullified polynomial was discarded. The outputted
CAD may not be sign-invariant.
Warning, there is nullification on cell [3, 4, 2] which has
dim>0. The nullified polynomial was discarded. The outputted
CAD may not be sign-invariant.
```

$$241 \tag{3.2}$$

In this case the algorithm continued, but the warning implies that the output may be meaningless.
We can change what happens in this situation with an optional argument.
Using failure=err will generate an error message as opposed to the default warning. Using failure= giveFAIL will return the boolean value FAIL.

```
> CADFull( {f}, [a,b,c,d,e], method=McCallum, failure=err );
Error, (in PCAD_ProjCADLift) there is nullification on cell
[1, 2, 2] which has dim>0. The outputted CAD cannot be
guaranteed sign-invariant.
```

```
> CADFull( {f}, [a,b,c,d,e], method=McCallum, failure=giveFAIL );
```

$$\mathit{FAIL} \tag{3.3}$$

Note that we can get a guaranteed correct output by using Collins:

```
> CADFull( {f}, [a,b,c,d,e], method=Collins  ): nops(%);
```

$$241 \tag{3.4}$$

In this case the number of cells in Collins guaranteed SI CAD was the same as in the "failed" McCallum cad.

UserInfo
We can get more information of the algorithm CADFull as it runs by changing the userinfo level of ProjectionCAD.
Level 1 is the default and gives no information other than errors and warnings.
Level 2 will also report on the progress of the algorithm as it runs and give details on reasons for failure.
Level 3 will also list the all the cells where nullification occurs and what happened because of it.
There is also Level 4 which is rarely used. It sometimes prints the polynomials in question.

Comparing user info levels

```
> infolevel[ProjectionCAD]:=1:
> CADFull( {f}, [a,b,c,d,e], method=McCallum  ):
Warning, there is nullification on cell [1, 2, 2] which has
dim>0. The nullified polynomial was discarded. The outputted
CAD may not be sign-invariant.
Warning, there is nullification on cell [2, 2, 1] which has
dim>0. The nullified polynomial was discarded. The outputted
CAD may not be sign-invariant.
Warning, there is nullification on cell [3, 4, 2] which has
dim>0. The nullified polynomial was discarded. The outputted
CAD may not be sign-invariant.
> infolevel[ProjectionCAD]:=2:
> CADFull( {f}, [a,b,c,d,e], method=McCallum  ):
CADFull: produced set of 6 projection factors using the
McCallum algorithm.
PCAD_ProjCADLift: produced CAD of [e] -space with 3 cells
PCAD_ProjCADLift: produced CAD of [d e] -space with 13 cells
PCAD_ProjCADLift: produced CAD of [c d e] -space with 33
cells
PCAD_ProjCADLift: the polynomial b*d+c*e+d+e is nullified by
[e = -1 d = 0 c = -1] which is a cell of dim>0.
Warning, there is nullification on cell [1, 2, 2] which has
dim>0. The nullified polynomial was discarded. The outputted
CAD may not be sign-invariant.
PCAD_ProjCADLift: the polynomial b*d+c*e+d+e is nullified by
[e = 0 d = 0 c = 0] which is a cell of dim>0.
Warning, there is nullification on cell [2, 2, 1] which has
dim>0. The nullified polynomial was discarded. The outputted
CAD may not be sign-invariant.
PCAD_ProjCADLift: the polynomial b*d+c*e+d+e is nullified by
[e = 1 d = 0 c = -1] which is a cell of dim>0.
Warning, there is nullification on cell [3, 4, 2] which has
dim>0. The nullified polynomial was discarded. The outputted
CAD may not be sign-invariant.
PCAD_ProjCADLift: produced CAD of [b c d e] -space with 85
cells
PCAD_ProjCADLift: produced CAD of [a b c d e] -space with
241 cells
> infolevel[ProjectionCAD]:=3:
> CADFull( {f}, [a,b,c,d,e], method=McCallum  ):
CADFull: produced set of 6 projection factors using the
McCallum algorithm.
PCAD_ProjCADLift: produced CAD of [e] -space with 3 cells
PCAD_ProjCADLift: produced CAD of [d e] -space with 13 cells
PCAD_ProjCADLift: a projection polynomial was nullified on
cell [2 2]
PCAD_ProjCADLift: the nullified polynomial was discarded
since cell [2 2] is zero-dim and the minimal delineating
polynomial is a constant.
PCAD_ProjCADLift: produced CAD of [c d e] -space with 33
cells
PCAD_ProjCADLift: a projection polynomial was nullified on
cell [1 2 2]
PCAD_ProjCADLift: the polynomial b*d+c*e+d+e is nullified by
```

```
[e = -1 d = 0 c = -1] which is a cell of dim>0.
Warning, there is nullification on cell [1, 2, 2] which has
dim>0. The nullified polynomial was discarded. The outputted
CAD may not be sign-invariant.
PCAD_ProjCADLift: a projection polynomial was nullified on
cell [2 2 1]
PCAD_ProjCADLift: the polynomial b*d+c*e+d+e is nullified by
[e = 0 d = 0 c = 0] which is a cell of dim>0.
Warning, there is nullification on cell [2, 2, 1] which has
dim>0. The nullified polynomial was discarded. The outputted
CAD may not be sign-invariant.
PCAD_ProjCADLift: a projection polynomial was nullified on
cell [3 4 2]
PCAD_ProjCADLift: the polynomial b*d+c*e+d+e is nullified by
[e = 1 d = 0 c = -1] which is a cell of dim>0.
Warning, there is nullification on cell [3, 4, 2] which has
dim>0. The nullified polynomial was discarded. The outputted
CAD may not be sign-invariant.
PCAD_ProjCADLift: produced CAD of [b c d e] -space with 85
cells
PCAD_ProjCADLift: a projection polynomial was nullified on
cell [2 1 1 2]
PCAD_ProjCADLift: the nullified polynomial was discarded
since this is the final lift and only sign invariance is
required.
PCAD_ProjCADLift: a projection polynomial was nullified on
cell [2 2 1 1]
PCAD_ProjCADLift: the nullified polynomial was discarded
since this is the final lift and only sign invariance is
required.
PCAD_ProjCADLift: a projection polynomial was nullified on
cell [2 3 1 2]
PCAD_ProjCADLift: the nullified polynomial was discarded
since this is the final lift and only sign invariance is
required.
PCAD_ProjCADLift: produced CAD of [a b c d e] -space with
241 cells
```

Note that userinfo statements can significantly increase the time taken by algorithms and so shouldn't be used as default.

```
> infolevel[ProjectionCAD]:=1:
```

Note that similar userinfo statements are available for the algorithms ECCAD and TTICAD discussed later.

4. Order invariant CADs

As discussed above, McCallum's CADW algorithm has been modified to maximise success - this is the default.
To run McCallum's original algorithm, guaranteeing order-invariance at the final step, (or failure), then the optional argument finalCAD=OI must be given.
Consider for example:

```
> f:=x^2-z*y;
```

$$f := x^2 - z\,y \tag{4.1}$$

```
> infolevel[CADFull]:=3:
> cad1:=CADFull( {f}, [z,y,x], method=McCallum, output=
  listwithrep  ): nops(%);
CADFull: produced set of 3 projection factors using the
McCallum algorithm.
PCAD_ProjCADLift: produced CAD of [x] -space with 3 cells
PCAD_ProjCADLift: produced CAD of [y x] -space with 9 cells
PCAD_ProjCADLift: a projection polynomial was nullified on
cell [2 2]
PCAD_ProjCADLift: the nullified polynomial was discarded since
this is the final lift and only sign invariance is required.
PCAD_ProjCADLift: produced CAD of [z y x] -space with 21 cells
```

$$21 \tag{4.2}$$

```
> cad2:=CADFull( {f}, [z,y,x], method=McCallum, finalCAD=OI,
  output=listwithrep  ): nops(%);
CADFull: produced set of 3 projection factors using the
McCallum algorithm.
PCAD_ProjCADLift: produced CAD of [x] -space with 3 cells
PCAD_ProjCADLift: produced CAD of [y x] -space with 9 cells
PCAD_ProjCADLift: a projection polynomial was nullified on
cell [2 2]
PCAD_ProjCADLift: cell [2 2] is zero-dim so a delineating
polynomial was used.
PCAD_ProjCADLift: produced CAD of [z y x] -space with 23 cells
```

$$23 \tag{4.3}$$

The first cad only has 21 cells, the second 23. Look at the difference by considering the indices

```
> convert( map(X->op(1,X), cad2),set) minus convert(map(X->op(1,
  X),cad1),set);
```

$$\{[2, 2, 2], [2, 2, 3]\} \tag{4.4}$$

```
> select(X->op(1,X)=[2,2,1], cad2);
  select(X->op(1,X)=[2,2,2], cad2);
  select(X->op(1,X)=[2,2,3], cad2);
```

$$[[[2, 2, 1], [x = 0, y = 0, z < 0], [regular\_chain, [[0, 0], [0, 0], [-1, -1]]]]]$$
$$[[[2, 2, 2], [x = 0, y = 0, z = 0], [regular\_chain, [[0, 0], [0, 0], [0, 0]]]]]$$
$$[[[2, 2, 3], [x = 0, y = 0, 0 < z], [regular\_chain, [[0, 0], [0, 0], [1, 1]]]]] \tag{4.5}$$

```
> select(X->op(1,X)=[2,2,1], cad1);
```

$$[[[2, 2, 1], [x = 0, y = 0, z = z], [regular\_chain, [[0, 0], [0, 0], [0, 0]]]]] \tag{4.6}$$

The difference occurs when x=0 and y=0. The first cad has just one cell here, with z free. The second splits into three cells, with z<0,z=0 and z>0.
This is clearly needed for OI, since the order of f at zero is 2 while the order elsewhere is 1 or 0.

By considering the user info statements we see why the outputs differ. In both cases there is a nullification at the final lift.
The first time this was ignored since it was the final lift and it was assumed only a sign-invariant cad is required.
The second time this nullification was examined. Since the cell in question was zero-dimensional we could use McCallum's delineating polynomial method to guarantee order invariance.

```
> infolevel[CADFull]:=1:
```

5. Generate Stack and returning induced CADs

Part of the lifting algorithm involves building a stack over a cell such that a set of polynomials are delineable.
To do this we use the internal RegularChains algorithms, ***RegularChains:-TRD_generate_stack***.
The polynomials passed to this algorithm should separate above the cell, hence to ensure correctness, we must first process our polynomials so that this is true.
The user algorithm, ***CADGenerateStack*** first error checks, then does this pre-processing before finally passing to the TRD algorithm.
Consider for example:

```
> f:=z^2+y^2+x^2-1;
  fullcad:=CADFull([f], [z,y,x], method=McCallum): nops(%);
```

$$f := x^2 + y^2 + z^2 - 1$$

$$25 \quad \textbf{(5.1)}$$

We can force CADFull to break halfway and return the cad of over (x,y)-space using the optional argument retcad=i

```
> xycad:=CADFull([f], [z,y,x], method=McCallum, retcad=2): nops
  (%);
```

$$13 \quad \textbf{(5.2)}$$

Consider a cell of this cad.

```
> cell:=xycad[3];
```

$$cell := [[2, 2], [regular\_chain, [[-1, -1], [0, 0]]]] \quad \textbf{(5.3)}$$

If we generate the stack over this cell wrt to f, we get a list of new cells.

```
> CADGenerateStack( cell, [f], [z,y,x] );
```

$$[[[2, 2, 1], [regular\_chain, [[-1, -1], [0, 0], [-1, -1]]]], [[2, 2, 2], [regular\_chain, [[-1, -1], [0, 0], [0, 0]]]], [[2, 2, 3], [regular\_chain, [[-1, -1], [0, 0], [1, 1]]]]] \quad \textbf{(5.4)}$$

Note that each cell has an index which starts with the index of the previous cell.
If we were to do this to all cells and put them together we get the full cad:

```
> cad:=[]:
  for i from 1 to 13 do
    cad:=[op(cad), op(CADGenerateStack(xycad[i],[f],[z,y,x]))]:
  od: nops(cad);
```

$$25 \quad \textbf{(5.5)}$$

The two cads can be shown to be identical under inspection

We now consider producing CADs invariant with respect to different conditions.

6. CADs with equational constraint

We can build CADs which are sign-invariant with respect to an equational constraint. For example, consider

```
> f:=x^2+y^2-1:
  g:=(x-1/2)*(y-1/2):
```

and the problem f=0, g>0. We can use ECCAD to build the CAD, the first argument is a designated equational constraint and the second a list of any other constraints.

```
> ECCAD( [f, [g]], [y,x]): nops(%);
```

$$43 \tag{6.1}$$

Note that this CAD has less cells than a full sign-invariant CAD.

```
> CADFull( [f,g], [y,x], method=McCallum): nops(%);
```

$$61 \tag{6.2}$$

The algorithm follows McCallum's approach, using his reduced projection operator for the first projection. Hence there is no method choice here.

The algorithm only makes use of one equational constraint. If a problem has more than one then a choice must be made for which to designate.

For example, consider the previous problem with the extra constraint h=0 where

```
> h:=x*y-1:
```

Then we can can obtain a list of the different formulations using ECCADFormulations on a list of equational constraints and a list of non-equational constraints.

```
> Forms:=ECCADFormulations( [[f,h], [g]] );
```

$$Forms := \left[\left[y^2+x^2-1, \left[xy-1, xy-\frac{1}{2}x-\frac{1}{2}y+\frac{1}{4}\right]\right], \left[xy-1, \left[y^2+x^2-1, xy-\frac{1}{2}x-\frac{1}{2}y+\frac{1}{4}\right]\right]\right] \tag{6.3}$$

In this example, a smaller CAD is obtained by designating h:

```
> ECCAD( Forms[1], [y,x]): nops(%);
  ECCAD( Forms[2], [y,x]): nops(%);
```

$$43$$
$$19 \tag{6.4}$$

We can attempt to predict this using heuristics which examine the projection factors.

The projection factors can be obtained using ECCADProjFactors, which takes the same input as ECCAD.

NOTE: The projection polynomials are all those used in Projection. However, unlike CADFull, they are not ALL used in Lifting. In particular the non-equational constraints are not usually used in the final lift.

For more information on this see the section of projection / lifting polynomials below.

```
> PF1:=ECCADProjFactors( Forms[1], [y,x]);
  PF2:=ECCADProjFactors( Forms[2], [y,x]);
```

$$PF1 := \left\{x-1, x+1, x-\frac{1}{2}, y-\frac{1}{2}, x^2-\frac{3}{4}, yx-1, x^4-x^2+1, y^2+x^2-1\right\}$$
$$PF2 := \left\{x, x-2, x-\frac{1}{2}, y-\frac{1}{2}, yx-1, x^4-x^2+1, y^2+x^2-1\right\} \tag{6.5}$$

We use the measures sotd (sum of total degree of all factors) and ndrr (number of distinct real roots of univariate factors) to estimate CAD complexity.

```
> [sotd(PF1), sotd(PF2)];
```

$$[18, 16] \tag{6.6}$$

```
> [ndrr(PF1,x), ndrr(PF2,x)];
```

$$[5, 3] \tag{6.7}$$

We see that both predict designating h gives a less complicated CAD.

These steps can be performed by the command ECCADHeuristic (taking the input of ECCADFormulations and the variable ordering)

```
> ECCADHeuristic( [[f,h], [g]], [y,x] );
```

$$\left[y\,x-1, \left[y^2+x^2-1, \left(x-\frac{1}{2}\right)\left(y-\frac{1}{2}\right)\right]\right] \tag{6.8}$$

By default this picks the formulation with lowest ndrr, breaking ties with sotd (NS). However, the reverse (SN), using one only (S, N), or a weighted average of the relative heuristics (W) are also available and specified by heuristic.

```
> ECCADHeuristic( [[f,h], [g]], [y,x], heuristic=S );
```

$$\left[y\,x-1, \left[y^2+x^2-1, \left(x-\frac{1}{2}\right)\left(y-\frac{1}{2}\right)\right]\right] \tag{6.9}$$

```
> ECCADHeuristic( [[f,h], [g]], [y,x], heuristic=W );
```

$$\left[y\,x-1, \left[y^2+x^2-1, \left(x-\frac{1}{2}\right)\left(y-\frac{1}{2}\right)\right]\right] \tag{6.10}$$

See the TTICAD section for more details on heuristics commands.

7. Truth table invariant CAD

The command TTICAD produces a CAD which is truth table invariant for a list of QFFs (quantifier free formulas). Each QFF is represented by an equational constraint and a list of other constraints. For example, consider.

```
> f1:=x^2+y^2-1:
  g1:=x*y-1/4:
  f2:=(x-4)^2+(y-1)^2-1:
  g2:=(x-4)*(y-1)-1/4:
```

and the formula PHI = phi[1] ∨ phi[2]
where
phi[1] = { f1=0, g1<0 }
phi[2] = { f2=0, g2<0 }.

We treat phi[2] and phi[2] as different QFFs and so use the input

```
> PHI:=[ [f1,[g1]], [f2,[g2]] ]:
```

Then we find the TTICAD with respect to a given variable ordering

```
> TTICAD( PHI, [y,x] ): nops(%);
```

$$105 \tag{7.1}$$

Compare this to a sign-invariant CAD:

```
> CADFull( [f1,f2,g1,g2], [y,x], method=McCallum): nops(%);
```

$$317 \tag{7.2}$$

or the CAD that could be constructed with an implicit equational constraint

```
> ECCAD( [f1*f2, [f1,g1,f2,g2]], [y,x] ): nops(%);
```

$$145 \tag{7.3}$$

We can use the retcad option to see the difference in the induced 1d cads

```
> CADFull( [f1,f2,g1,g2], [y,x], method=McCallum, retcad=1): nops
  (%);
  ECCAD( [f1*f2, [f1,g1,f2,g2]], [y,x], retcad=1 ): nops(%);
```

```
TTICAD( PHI, [y,x], retcad=1 ): nops(%);
```

$$41$$
$$33$$
$$25 \qquad (7.4)$$

We see that a small difference in cells (41-25=16) is magnified to a larger difference (317-105-212) after lifting.

We can see the set of projection factors produced using the command TTICADProjFactors on TTICAD input. Note that these are the polynomials used in the projection but they may not all be used for lifting (see Section 10).

```
> P:=TTICADProjFactors( PHI, [y,x]);
```

$$P := \left\{x-5,\ x-3,\ x-1,\ x+1,\ yx-\frac{1}{4},\ x^2-4x+\frac{285}{68},\ y^2+x^2-1,\ x^4-x^2+\frac{1}{16},\ yx\right. \qquad (7.5)$$
$$\left. -x-4y+\frac{15}{4},\ y^2+x^2-2y-8x+16,\ x^4-16x^3+95x^2-248x+\frac{3841}{16}\right\}$$

TTICAD offers the same output formats as a sign-invariant CAD, discussed above.

```
> TTICAD( [ [x+1,[x,x^2]], [x-1,[x+2*x]] ], [x], output=
  piecewise); NumCellsInPiecewiseCAD(%);
```

$$\begin{cases} [regular\_chain, [[-2, -2]]] & x < -1 \\ [regular\_chain, [[-1, -1]]] & x = -1 \\ \left[regular\_chain, \left[\left[-\frac{1}{2}, -\frac{1}{2}\right]\right]\right] & \mathrm{And}(-1 < x, x < 1) \\ [regular\_chain, [[0, 2]]] & x = 1 \\ [regular\_chain, [[3, 3]]] & 1 < x \end{cases}$$
$$5 \qquad (7.6)$$

When input is given containing only one variable then a sign-invariant CAD is created for the equational constraints only, as in the example above.

Of course, the variable ordering is important:

```
> TTICAD( PHI, [x,y] ): nops(%);
```

$$153 \qquad (7.7)$$

We note that the TTICAD theory can extend to a sequence of clauses in which not all have an equational constraint.

For example, consider the problem above but with f1<0 instead of f1=0.

The implicit equational constraint approach cannot be used but TTICAD can, still giving a reduction compared to a full cad:

```
> PHI2:=[ [[],[f1,g1]], [f2,[g2]] ]:
  TTICAD( PHI2, [y,x] ): nops(%);
```

$$183 \qquad (7.8)$$

TTICAD will offer a benefit over a full cad unless there are no equational constraints at all

```
> PHI3:=[ [[],[f1,g1]], [[],[f2,g2]] ]:
  TTICAD( PHI, [y,x] ): nops(%);
```

$$105 \qquad (7.9)$$

Finally, we note that if there is more than one equational constraint then the TTICAD algorithm will choose which to use based on the heuristics discussed below.

8. The ResCAD algorithm for TTICAD

We can also attempt to create a TTICAD using the ResCAD approach. This is a conceptually simpler algorithm but not as widely applicable.
There is no practical reason to use this approach here, since TTICAD is superior. It is included to compare for experimentation and to compare with other systems.

The ResCAD set can be calculated using:

```
> rcs:=TTICADResCADSet( PHI, [y,x] );
```

$$rcs := \{16\,x^4 - 16\,x^2 + 1,\ y^2 + x^2 - 1,\ 16\,x^4 - 256\,x^3 + 1520\,x^2 - 3968\,x + 3841,\ y^2 + x^2 - 2\,y - 8\,x + 16\} \quad (8.1)$$

Providing no EC is nullified, a full CAD of this set using McCallum projection will be a TTICAD

```
> CADFull( rcs, [y,x], method=McCallum, output=list): nops(%);
```

$$105 \quad (8.2)$$

This approach is combined into one command:

```
> TTICADResCAD( PHI, [y,x] ): nops(%);
```

Warning, The output from TTICADResCAD is only guaranteed truth table invariant if no equational constraint is nullified by a point in [x]

$$105 \quad (8.3)$$

We gives some examples where the ResCAD approach can fail

Example 1 - An equational constraint is reducible:

Consider

```
> f:=x*y: g:=x+y+1: PHI:=[ [f,[g]] ];
```

$$PHI := [[y\,x, [x + y + 1]]] \quad (8.4)$$

```
> rescad:=TTICADResCAD( PHI, [y,x], output=piecewise ):
  NumCellsInPiecewiseCAD(%);
```

Warning, The output from TTICADResCAD is only guaranteed truth table invariant if no equational constraint is nullified by a point in [x]

$$15 \quad (8.5)$$

Note there is a cell [x=0, y<0] over which f is zero but g changes sign. We can avoid this error by using the full TTICAD algorithm

```
> TTICAD( PHI, [y,x], output=piecewise): NumCellsInPiecewiseCAD
  (%);
```

$$17 \quad (8.6)$$

The TTICAD splits this cell, but this is still smaller than the sign-invariant CAD

```
> CADFull( [f,g], [y,x], output=piecewise, method=McCallum):
  NumCellsInPiecewiseCAD(%);
```

$$23 \quad (8.7)$$

Example 2 - An equational constraint is nullified:

Consider

```
> f:=z*y-x^2: g:=x+y+z-1: PHI:=[ [f,[g]] ];
```

$$PHI := [[z\,y - x^2, [x + y + z - 1]]] \quad (8.8)$$

```
> rescad:=TTICADResCAD( PHI, [z,y,x], output=piecewise ):
  NumCellsInPiecewiseCAD(%);
```

```
Warning, The output from TTICADResCAD is only guaranteed truth
table invariant if no equational constraint is nullified by a
point in [y, x]
```

91 **(8.9)**

Note there is a cell [x=0, y=0, z free] over which f is zero but g changes sign!

We can avoid this error by using the full TTICAD algorithm, which splits the cell but is still much smaller than a sign-invariant CAD

```
> TTICAD( PHI, [z,y,x], output=piecewise): NumCellsInPiecewiseCAD
  (%);
  CADFull( [f,g], [z,y,x], output=piecewise, method=McCallum):
  NumCellsInPiecewiseCAD(%);
```

93

147 **(8.10)**

Example 3 - An equational constraint has main variable of a lower level than the other constraints

Consider

```
> f:=x: g:=x+y+1: PHI:=[ [f,[g]] ];
```

$$PHI := [[x, [y + x + 1]]] \quad \textbf{(8.11)}$$

```
> rescad:=TTICADResCAD( PHI, [y,x], output=piecewise ):
  NumCellsInPiecewiseCAD(%);
Warning, The output from TTICADResCAD is only guaranteed truth
table invariant if no equational constraint is nullified by a
point in [x]
Warning, An equational constraint will certainly be nullified
since x does not contain the main variable y.
Warning, there are no projection polynomials in [y, x]
```

3 **(8.12)**

Such cases are simple to recognize so an extra warning is provided

Note there is a cell [x=0, y free] over which f is zero but g changes sign. Once again, the TTICAD algorithm splits this

```
> TTICAD( [ [f,[g]] ], [y,x], output=piecewise):
  NumCellsInPiecewiseCAD(%);
  CADFull( [f,g], [y,x], output=piecewise, method=McCallum):
  NumCellsInPiecewiseCAD(%);
```

5

9 **(8.13)**

9. TTICAD Formulations and Heuristics

Consider

```
> f11:=y^2+x^2-1: f12:=x^3+y^3-1: g1:=y*x-1/4:
  f21:=(x-4)^2+(y-1)^2-1: f22:=(x-4)^3+(y-1)^3-1: g2:=(x-4)*(y-1)
  -1/4:
```

and the problem [f11=0 /\ f12=0 /\ g1<0] \/ [f21=0 /\ f22=0 /\ g2<0].

We can solve this with a TTICAD

```
> PHI1:=[ [f11,[f12,g1]], [f21,[f22,g2]] ]:
  TTICAD(PHI1, [y,x] ): nops(%);
```

$$125 \tag{9.1}$$

However there are different possible formulations for the TTICAD leading to CADs of different sizes, for example

```
> PHI2:=[ [f12,[f11,g1]], [f22,[f21,g2]] ]:
  TTICAD(PHI2, [y,x] ): nops(%);
```

$$85 \tag{9.2}$$

We can obtain a list of different formulations using TTICADFormulations on a list of lists, each containing two lists - one of equational constraints and one of other constraints.

```
> Forms:=TTICADFormulations( [ [[f11,f12],[g1]], [[f21,f22],[g2]]
  ] ): nops(%);
```

$$16 \tag{9.3}$$

There are 16 formulations because it there is not just the choice of equational constraints but also the choice of splitting QFFs. E.g.

```
> Forms[3];
```

$$\left[\left[y^2+x^2-1, \left[y\,x-\frac{1}{4}\right]\right], \left[y^3+x^3-1, [\,]\right], \left[y^2+x^2-2\,y-8\,x+16, \left[y^3+x^3-3\,y^2-12\,x^2+3\,y+48\,x-66, y\,x-x-4\,y+\frac{15}{4}\right]\right]\right] \tag{9.4}$$

Again, we can predict CAD complexity by running the measures on the projection factors

```
> TTICADProjFactors( PHI1, [y,x] ): sotd(%), ndrr(%,x);
  TTICADProjFactors( PHI2, [y,x] ): sotd(%), ndrr(%,x);
```

$$65,\ 14$$
$$101,\ 8 \tag{9.5}$$

In this case the ndrr measure predicted correctly but the sotd measure did not. Emphasizing that these are just heuristics.

We can use the heuristic command to pick a formulation. Note that different heuristics may pick different formulations.

```
> TTICADHeuristic( [ [[f11,f12],[g1]], [[f21,f22],[g2]]], [y,x],
  heuristic=N): is(%=Forms[6]);
```

$$\mathit{true} \tag{9.6}$$

```
> TTICADHeuristic( [ [[f11,f12],[g1]], [[f21,f22],[g2]]], [y,x],
  heuristic=S): is(%=Forms[1]);
```

$$\mathit{true} \tag{9.7}$$

We use the default of N first then S to split. This also picks the sixth option, (which was PHI2 above).

```
> is(Forms[6]=expand(PHI2));
```

$$\mathit{true} \tag{9.8}$$

We demonstrate the weighted heuristic. This considers the ranking by each heuristic, by default giving equal ratio to each. This can be varied with the Wratio argument which gives the ration ndrr:sotd (by default 1:1)

```
> TTICADHeuristic( [ [[f11,f12],[g1]], [[f21,f22],[g2]]], [y,x],
  heuristic=W, Wratio=[1,1]): is(%=Forms[2]);
```

$$true \tag{9.9}$$

The number of TTICAD formulations can be large so we offer two approaches to reducing the heuristic time.

```
> st:=time(): TTICADHeuristic( [ [[f11,f12],[g1]], [[f21,f22],
  [g2]]], [y,x]): et:=time()-st;
```

$$et := 0.785 \tag{9.10}$$

1: Ignore the possibility of splitting QFFs. Although this can help, it usually doesn't

```
> TTICADFormulations( [ [[f11,f12],[g1]], [[f21,f22],[g2]]],
  splitting=false ): nops(%);
```

$$4 \tag{9.11}$$

```
> st:=time(): TTICADHeuristic( [ [[f11,f12],[g1]], [[f21,f22],
  [g2]]], [y,x], splitting=false ): et:=time()-st;
```

$$et := 0.063 \tag{9.12}$$

2: Work on each QFF one by one - the so called modular approach. This will consider less possible formulations, and ignores the effect of the QFFs interacting.

```
> st:=time(): TTICADHeuristic( [ [[f11,f12],[g1]], [[f21,f22],
  [g2]]], [y,x], modular=true): et:=time()-st;
```

$$et := 0.032 \tag{9.13}$$

Note that we can also consider QFF formulations and heuristics individually ourselves

```
> TTICADQFFFormulations( [[f11,f12],[g1]] ): nops(%);
```

$$4 \tag{9.14}$$

```
> TTICADQFFHeuristic( [[f11,f12],[g1]], [y,x] );
```

$$\left[\left[y^3+x^3-1, \left[y^2+x^2-1, y\,x-\frac{1}{4}\right]\right]\right] \tag{9.15}$$

Note that we can give the input to TTICAD directly and it will use the default heuristic on each QFF to choose the formulation.

```
> TTICAD( [ [[f11,f12],[g1]], [[f21,f22],[g2]]], [y,x] ): nops(%)
  ;
```

$$85 \tag{9.16}$$

10. Projection polynomials vs lifting polynomials

When building a full sign-invariant CAD the set of projection polynomials is constructed during projection, and then used during lifting so that all the polynomials are sign-invariant on each cell. As noted above, this is not the case for ECCAD / TTICAD. Theorems on the reduced projection operators used for these algorithms allow us to (usually) ignore the non-equational constraints when lifting. Hence, while these polynomials are part of the projection set, they are not (usually) part of the lifting set.

The projection polynomials for these algorithms can be calculated independently using the commands **ECCADProjFactors** and **TTICADProjFactors.**
The lifting polynomials will vary from stack to stack. For all except the final lift the projection polynomials of that level will be used, as normal.
For the final lift it is usually only the equational constraints which are used. However, if an equational constraint is nullified on a zero dim cell, then the lifting set is extended for that stack only to include the corresponding non-equational constraints.
For information on the process change the TTICAD info level to 3..

```
> infolevel[TTICAD]:=3:
> f:=x*y: g:=x+y+1: PHI:=[ [f,[g]] ];
  TTICAD(PHI, [y,x]): nops(%);
```

$$PHI := [[y\,x, [x+y+1]]]$$

```
TTI_TTIGenerateStack: the equational constraint factor y*x  is
nullified on the cell  [[4] [regular_chain [[0 0]]]]
TTI_TTIGenerateStack: the cell is zero-dimensional so we can
continue by expanding the lifting set on this cell.
```

$$17 \qquad \textbf{(10.1)}$$

The same process is used for ECCAD:

```
> ECCAD([f,[g]], [y,x] ): nops(%);
TTI_TTIGenerateStack: the equational constraint factor y*x  is
nullified on the cell  [[4] [regular_chain [[0 0]]]]
TTI_TTIGenerateStack: the cell is zero-dimensional so we can
continue by expanding the lifting set on this cell.
```

$$17 \qquad \textbf{(10.2)}$$

```
> infolevel[TTICAD]:=1:
```

Modifying the lifting set in this way for ECCAD follows from the original theorems but does not appear to have been implemented before. For example, in QEPCAD the full projection set is used for lifting.
Consider the following example:

```
> f1:=x^2+y^2-1: g1:=x*y-1/4:
  f2:=(x-4)^2+(y-1)^2-1: g2:=(x-4)*(y-1)-1/4:
> ECCAD( [f1*f2, [f1,f2,g1,g2]], [y,x] ): nops(%);
```

$$145 \qquad \textbf{(10.3)}$$

Using QEPCAD with f1*f2 declared an EC produces 249 cells.
The reason is that g1 and g2 are included in the lifting set for QEPCAD, but never here.
We can simulate the QEPCAD approach with the optional command LiftAll

```
> ECCAD( [f1*f2, [f1,f2,g1,g2]], [y,x], LiftAll=true ): nops(%);
```

$$249 \qquad \textbf{(10.4)}$$

This can also be used with TTICADs.

```
> PHI:=[ [f1,[g1]], [f2,[g2]] ]:
  TTICAD( PHI, [y,x] ): nops(%);
  TTICAD( PHI, [y,x], LiftAll=true ): nops(%);
```

105

185 **(10.5)**

Note that there is no theoretical reason to use LiftAll. It is only included here to make comparison with QEPCAD easier.

We consider the effect of variable ordering

11. Choosing a Variable ordering

The variable ordering can make a difference to size and computation time of a CAD.

```
> f:=(x-1)*(y^2+1)-1:
  CADFull( {f}, [y,x], method=Collins): nops(%);
  CADFull( {f}, [x,y], method=Collins): nops(%);
```

$$11$$

$$3 \qquad (11.1)$$

The VariableOrderings command returns a list of the possible variable orderings for a list of variables

```
> VariableOrderings( [y,x] );
```

$$[[y, x], [x, y]] \qquad (11.2)$$

For many applications there are restrictions on variable orderings. For example, in quantifier elimination all quantified variables must be projected first, and two adjacent quantified variables can only be switched in the orderings if they have the same quantifier. The VariableOrderings command can consider such situations when given a list of lists of variables (representing variable blocks within which different orderings are permitted.

```
> VariableOrderings( [x,y,z,w] ): nops(%);
  VariableOrderings( [[x,y], [z,w]] ); nops(%);
```

$$24$$

$$[[x, y, z, w], [y, x, z, w], [x, y, w, z], [y, x, w, z]]$$

$$4 \qquad (11.3)$$

```
> VariableOrderings( [w,x,y,z,a,b] ): nops(%);
  VariableOrderings( [[w],[x,y,z], [a,b]] ); nops(%);
```

$$720$$

$$[[w, x, y, z, a, b], [w, x, z, y, a, b], [w, y, x, z, a, b], [w, y, z, x, a, b], [w, z, x, y, a, b], [w, z, y, x, a, b], [w, x, y, z, b, a], [w, x, z, y, b, a], [w, y, x, z, b, a], [w, y, z, x, b, a], [w, z, x, y, b, a], [w, z, y, x, b, a]]$$

$$12 \qquad (11.4)$$

We can use the ndrr and sotd measures applied to projection polynomials to pick the best variable ordering for a problem.
This is encoded in VariableOrderingHeuristic. The first entry should be a list of variables (or list of lists as above). The second should be the input to the CAD algorithm. The output is then the suggested ordering.

```
> VariableOrderingHeuristic( [y,x], {f} );
```

$$[x, y] \qquad (11.5)$$

The algorithm assumes a sign-invariant CAD (i.e. using CADFull) but this can be specified, (along with the method if using CADFull).

```
> VariableOrderingHeuristic( [y,x], {f}, algorithm=CADFull,
  method=McCallum );
```

$$[x, y] \qquad (11.6)$$

Note the usual heuristic options

```
> VariableOrderingHeuristic( [y,x], {f}, algorithm=CADFull,
  heuristic=S );
Warning, There are 2 formulations which heuristic S cannot
```

differentiate. Only the first has been returned. To display them all run with option SeeAll=true.

$$[y, x] \tag{11.7}$$

```
> VariableOrderingHeuristic( [y,x], {f}, algorithm=CADFull,
  heuristic=N );
```

$$[x, y] \tag{11.8}$$

We can use this command to choose the best variable ordering for ECCAD and TTICAD as well. In general she second argument should be input suitable for the respective algorithm.

```
> f:=x^2+y^2-1: g:=x*(y-1/2):
  ECCAD( [f, [g]], [y,x]): nops(%);
  ECCAD( [f, [g]], [x,y]): nops(%);
```

$$43$$
$$23 \tag{11.9}$$

```
> VariableOrderingHeuristic( [y,x], [f, [g]], algorithm=ECCAD );
```

$$[x, y] \tag{11.10}$$

```
> f1:=x^2+y^2-1: g1:=x*y-1/4:
  f2:=x^2-y^2-1: g2:=(x-4)*(y-1)-1/4:
  PHI:=[ [f1,[g1]], [f2,[g2]] ]:
  TTICAD( PHI, [y,x] ): nops(%);
  TTICAD( PHI, [x,y] ): nops(%);
```

$$101$$
$$175 \tag{11.11}$$

12. Greedy algorithms for variable ordering choices

We have implemented greedy algorithms for choosing variable orderings. This is where each variable is chosen in turn based on the projection polynomials calculated thus far.
For example, the quartic problem: x is quantified so comes first regardless. Then there are 6 variable orderings

```
> ord:=[p,q,r]: CADFull( [x^4+p*x^2+q*x+r], [x,op(ord)], method=
  McCallum ): ord,nops(%);
  ord:=[p,r,q]: CADFull( [x^4+p*x^2+q*x+r], [x,op(ord)], method=
  McCallum ): ord,nops(%);
  ord:=[q,p,r]: CADFull( [x^4+p*x^2+q*x+r], [x,op(ord)], method=
  McCallum ): ord,nops(%);
  ord:=[q,r,p]: CADFull( [x^4+p*x^2+q*x+r], [x,op(ord)], method=
  McCallum ): ord,nops(%);
  ord:=[r,p,q]: CADFull( [x^4+p*x^2+q*x+r], [x,op(ord)], method=
  McCallum ): ord,nops(%);
  ord:=[r,q,p]: CADFull( [x^4+p*x^2+q*x+r], [x,op(ord)], method=
  McCallum ): ord,nops(%);
```

$$[p, q, r], 333$$
$$[p, r, q], 333$$
$$[q, p, r], 277$$
$$[q, r, p], 321$$
$$[r, p, q], 177$$

(12.1)

$$[r, q, p], 213 \tag{12.1}$$

So the best ordering in terms of cell count is [x,r,p,q].

The standard heuristic find this:

```
> st:=time():
  VariableOrderingHeuristic( [[x],[p,q,r]], [x^4+p*x^2+q*x+r],
  heuristic=NS, algorithm=CADFull);
  et:=time()-st;
```

$$[x, r, p, q]$$
$$et := 0.038 \tag{12.2}$$

If we use the greedy algorithm we also find this, but quicker.

```
> st:=time():
  VariableOrderingHeuristic( [[x],[p,q,r]], [x^4+p*x^2+q*x+r],
  heuristic=NS, algorithm=CADFull, greedy=true);
  et:=time()-st;
```

$$[x, r, p, q]$$
$$et := 0.009 \tag{12.3}$$

But note that it was a lot quicker. For larger problems the speed up is more important.

Greedy algorithms are also present for picking the variable orderings for ECCAD:

```
> f:=x^2+y^2+z^2-1: g1:=x*(y-1/2): g2:=z*(y+1/2):
> ord:=[x,y,z]: ECCAD( [f, [g1,g2]], ord): ord,nops(%);
  ord:=[x,z,y]: ECCAD( [f, [g1,g2]], ord): ord,nops(%);
  ord:=[y,x,z]: ECCAD( [f, [g1,g2]], ord): ord,nops(%);
  ord:=[y,z,x]: ECCAD( [f, [g1,g2]], ord): ord,nops(%);
  ord:=[z,x,y]: ECCAD( [f, [g1,g2]], ord): ord,nops(%);
  ord:=[z,y,x]: ECCAD( [f, [g1,g2]], ord): ord,nops(%);
```

$$[x, y, z], 187$$
$$[x, z, y], 131$$
$$[y, x, z], 237$$
$$[y, z, x], 237$$
$$[z, x, y], 131$$
$$[z, y, x], 187 \tag{12.4}$$

The normal heuristic picks the best two

```
> st:=time():
  VariableOrderingHeuristic( [x,y,z], [f,[g1,g2]], heuristic=NS,
  algorithm=ECCAD, SeeAll=true);
  et:=time()-st;
```

Warning, There are 2 formulations which heuristic NS cannot differentiate.

$$[[x, z, y], [z, x, y]]$$
$$et := 0.022 \tag{12.5}$$

The greedy method picks the joint best, and quicker.

```
> st:=time():
  VariableOrderingHeuristic( [x,y,z], [f,[g1,g2]], heuristic=NS,
  algorithm=ECCAD, greedy=true);
  et:=time()-st;
```

$$[x, z, y]$$
$$et := 0.008 \tag{12.6}$$

We can also use greedy algorithms for variable blocks:

```
> st:=time():
  VariableOrderingHeuristic( [[x],[y,z]], [f,[g1,g2]], heuristic=
  NS, algorithm=ECCAD, SeeAll=true);
  et:=time()-st;
  st:=time():
  VariableOrderingHeuristic( [[x],[y,z]], [f,[g1,g2]], heuristic=
  NS, algorithm=ECCAD, greedy=true);
  et:=time()-st;
```

$$[x, z, y]$$
$$et := 0.008$$
$$[x, z, y]$$
$$et := 0.005 \qquad \textbf{(12.7)}$$

```
> st:=time():
  VariableOrderingHeuristic( [[x,y],[z]], [f,[g1,g2]], heuristic=
  NS, algorithm=ECCAD, SeeAll=true);
  et:=time()-st;
  st:=time():
  VariableOrderingHeuristic( [[x,y],[z]], [f,[g1,g2]], heuristic=
  NS, algorithm=ECCAD, greedy=true);
  et:=time()-st;
```

$$[x, y, z]$$
$$et := 0.008$$
$$[x, y, z]$$
$$et := 0.005 \qquad \textbf{(12.8)}$$

Finally we consider greedy algorithms for picking the variable ordering for TTICAD:

```
> f1:=x^2+y^2+z^2-1: g1:=x*y*z-1/4:
  f2:=x^2-y^2-z^2-1: g2:=(x-4)*(y-1)-1/4+z:
  PHI:=[ [f1,[g1]], [f2,[g2]] ]:
> ord:=[x,y,z]: TTICAD( PHI, ord): ord,nops(%);
  ord:=[x,z,y]: TTICAD( PHI, ord): ord,nops(%);
  ord:=[y,x,z]: TTICAD( PHI, ord): ord,nops(%);
  ord:=[y,z,x]: TTICAD( PHI, ord): ord,nops(%);
  ord:=[z,x,y]: TTICAD( PHI, ord): ord,nops(%);
  ord:=[z,y,x]: TTICAD( PHI, ord): ord,nops(%);
```

$$[x, y, z], 1497$$
$$[x, z, y], 889$$
$$[y, x, z], 1135$$
$$[y, z, x], 351$$
$$[z, x, y], 985$$
$$[z, y, x], 463 \qquad \textbf{(12.9)}$$

Normal heuristic picks the best one

```
> st:=time():
  VariableOrderingHeuristic( [x,y,z], PHI, heuristic=NS,
  algorithm=TTICAD, SeeAll=true);
  et:=time()-st;
```

$$[y, z, x]$$
$$et := 1.832 \qquad \textbf{(12.10)}$$

```
> st:=time():
  VariableOrderingHeuristic( [x,y,z], PHI, heuristic=NS,
  algorithm=TTICAD, greedy=true);
```

```
et:=time()-st;
```

$$[z, y, x]$$
$$et := 0.028 \qquad \textbf{(12.11)}$$

Greedy picks second best, but much quicker.

And finally with blocks:

```
> st:=time():
  VariableOrderingHeuristic( [[x],[y,z]], PHI, heuristic=NS,
  algorithm=TTICAD, SeeAll=true);
  et:=time()-st;
  st:=time():
  VariableOrderingHeuristic( [[x],[y,z]], PHI, heuristic=NS,
  algorithm=TTICAD, greedy=true);
  et:=time()-st;
```

$$[x, z, y]$$
$$et := 0.486$$
$$[x, z, y]$$
$$et := 0.030 \qquad \textbf{(12.12)}$$

Both pick correct, greedy much faster

```
> st:=time():
  VariableOrderingHeuristic( [[y,x],[z]], PHI, heuristic=NS,
  algorithm=TTICAD, SeeAll=true);
  et:=time()-st;
  st:=time():
  VariableOrderingHeuristic( [[y,x],[z]], PHI, heuristic=NS,
  algorithm=TTICAD, greedy=true);
  et:=time()-st;
```

$$[y, x, z]$$
$$et := 0.843$$
$$[y, x, z]$$
$$et := 0.032 \qquad \textbf{(12.13)}$$

Both pick correct, greedy much faster