

Exercises in Algebraic Solving of Minimal Problems

Henrik Stewenius

August 22, 2005

Abstract

Here Macaulay 2 will be used to build minimal case solvers.

1 Checking software

The following should be installed on your system in order to follow all parts of this material:

- Macaulay 2 (M2). If you do not have this you get it from

`\verb http://www.math.uiuc.edu/Macaulay2/`

It is a good idea to set up M2 so that it runs inside emacs, there are instructions in the help-files.

- Matlab or other numerical tool, preferably installed with some symbolig toolkit.

2 Warming up

Entering Code

Open a new file *test.m2* in *emacs*. To make it easier to read the code *font-lock-mode* is suggested. Type into the file

```
KK = ZZ / 30097;  
R = KK[x,y];  
eq = {x^2+1,x*y};  
I = ideal eq;  
dim I , degree I
```

In order to execute these lines we split the window (*CTRL-x 2*), jump to the other half (*CTRL-x o*) and press F12 to start M2. Then we jump back and place the cursor on the first row. On the first row we press F11 and this row is copied to the lower window-half and executed. We continue to press F11 until all our lines are executed.

The last line gives the dimension and degree of the variety, that is, a measure of the number of solutions.

Enter a simple Problem

We will now try to compute the number of intersection points of two circles. The first circle is centered in X1 and the second in X2.

```
KK = ZZ / 30097  
R = KK[x,y ]  
X1 = random(R^2,R^1)  
X2 = random(R^2,R^1)  
Y = matrix {{x},{y}}
```

```

d1 = random(R^1,R^1)
d2 = random(R^1,R^1)
dist1 = Y-X1
dist2 = Y-X2
eq = { transpose(dist1)*dist1 - d1^2,  transpose(dist2)*dist2 - d2^2 }
I = ideal eq;
dim I , degree I

```

It is not very surprising to find that there are two intersection points. As all programming languages there are many ways to write the same thing

```

KK = ZZ / 30097
R = KK[x,y ]
X = random(R^2,R^2)
Y = matrix {{x},{y}}
d = random(R^1,R^2)
eq = apply( 2, i->(a:= Y-X_{i}; transpose a*a - d_{i}^2))
I = ideal eq
dim I , degree I

```

Here we used the *apply* operator which takes two arguments, the first controlling over which numbers to apply the function which is the second argument. Now try to find the number of intersection points between three spheres!

gbTrace

If we add the line execute the line

```
gbTrace(3);
```

the verbosity of the calculations will increase. Do this and rerun the code. Try also to set

```
gbTrace(100);
```

This command will be used a lot later.

Building a solver

Consider a planar linear kinematic robot:

```

KK = ZZ / 30097;
R = KK[a,b,x,y ];
fixpoints = random( R^2,R^3);
arms = random( R^2,R^3);
armlengths = random(R^1,R^3);
ROT = matrix {{a,b},{-b,a}};
X = matrix{{x},{y}};
eq = fold( (a,b)->a||b, apply( 3, i->( a:= X + ROT*arms_{i} - fixpoints_{i} ; transpose a*a
rot = a^2+b^2-1;
I = ideal eq + ideal rot;
dim I , degree I

```

there are now 6 solutions. We would like to build a numerical solver to this problem and start by looking at the output from *gbTrace(100)*, there we see that we have to add some S-polynomials. Therefore we add

```
eq2 = (eq || a*eq || b*eq || x*eq || y*eq) % rot
```

it is now possible to write this on matrix form and eliminate (if some of my components have been loaded).

```
load "gj4.m2";
eq3 = elim(eq2);
```

we test again a look at the gbTrace

```
I = ideal eq3 + ideal rot;
gbTrace 100;
dim I , degree I
```

and see that some more polynomials have to be added. In order to avoid adding fourth order polynomials we find the second order parts of $eq3$.

```
tt = eq3^{8..14};
eq4 = elim( (tt || a*tt||b*tt||x*tt||y*tt) % rot ) ;
```

by looking at the output again we see that there are no longer any need for generating more S-polynomials.

Reading M2 output

```
i14 : dim I , degree I
--- computing pair ( . . a2) ----
.0 gb 0 = a2<0>+b2<0>-14366ax<0>-11232bx<0>+7031x2<0>-...
-13475y<0>-5605<0>
--- computing pair ( . . a2) ----
.1insert (1 0 a x a2x)
gb 1 = ax<0>-6400bx<0>-1022x2<0>-6400ay<0>-by<0>-...
--- computing pair ( . . a2) ----
.2insert (2 1 a b abx)
gb 2 = bx<0>+12850x2<0>+ay<0>+12850y2<0>+5794a<0>-...
--- computing pair ( . . a2) ----
.3insert (3 2 b x bx2)
insert (3 1 a x ax2)
gb 3 = x2<0>+y2<0>-2413a<0>-5410b<0>+7113x<0>-...
--- computing pair (3 2 b x bx2) ----
.7insert (4 2 x a abx)
insert (4 0 a b a2b)
gb 4 = ab<0>-5860b2<0>-13681ay<0>-4143by<0>-...
--- computing pair (3 1 a x ax2) ----
.10insert (5 2 x b b2x)
insert (5 4 a b ab2)
gb 5 = b2<0>+4227ay<0>-13292by<0>-10867xy<0>+...
--- computing pair (5 2 x b b2x) ----
```

- **computing pair** means that M2 has computed a new S-polynomial.

```
--- computing pair ( . . a2) ----
```

is a polynomial read from input. While

```
--- computing pair (3 2 b x bx2) ----
```

means the S-polynomial from gb elements 3 and 2 with multiples b and x , the leading term of the S-polynomial before reduction is bx^2 .

- **insert** says that elements are inserted into the list of S-polynomials that will be computed.
- **gb i** = means that an S-polynomial did not reduce to 0 and is inserted into the basis.

Reading the raw output is quite tiring for large problems and it is a good idea to pipe it through some script

```
rm temp
cat myscript.m2 | M2 > temp
cat temp | perl -p -e 's/gb/###gb;/s/comput/###comput;/s/<0>.*//'
          | egrep "###"
          | perl -p -e 's/###//'
          | egrep -v marked
```

possibly also adding

```
egrep gb -B 1
```

to reduce the data further. When your elimination steps are correct you will only get *computing pair* of the doubledot type.

Porting to matlab

The first step is to get the equations in, I find that the most convenient method to this is to use the symbolic toolbox as much as possible.

After this it is mainly a question about patience and looking at matrices in M2 to see which elements are zero.

There are some very good commands in matlab:

```
mcoeff =inline(['maple(''coeff'', '...
               'maple(''coeff'', '...
               'f, ''a'',n(1)), ''b'',n(2))'], 'f', 'n');
a = sym('a');
b = sym('b');
eq = {7*a*b+ 7*a, a+b};
MONS = [1 1 0 ; 1 0 1];
for j=1:2
    for i=1:3
        M(j,i) = eval(mcoeff( eq{j}, MONS(:,i)));
    end
end
```

will read the system matrix of the above equations. This is quite slow but very convenient when experimenting and building code.

It is also a very good idea to have some code that builds the monomial order vector. If you insert the right elements into the vector it can be sorted to GrevLex by

```
[dummy,I]=sortrows( [sum(MONS); -MONS(end:-1:1,:) ]' );
MONS = MONS(:,I(end:-1:1));
```

Assume now that we have a *gb* sitting in a matrix *M* and a monomial vector *MONS* and we want to compute an action matrix

```
NN=[];
inner_points = %%Define here which points that are in the polytope
               %%that is, not divisible by LT(I)
outer_points = %%Those in the MONS vector that are not in.
               %% that is, LT(I), zero columns should NOT be in this list.
for i=1:nbr_solutions
    p = MONS(:,inner_points(i));
    pnew = p+[0;1;0]; %%Possible to select other action matrices here.
    test = ismember(pnew', MONS(:,inner_points)', 'rows');
    loc = find(ismember( MONS(:,inner_points)',pnew', 'rows'));
    if test
        fprintf('NN(%i,%i)=1;\n', i ,loc );
    end
end
```

```

    NN(i,loc)=1;
else
    test = ismember(pnew', MONS(:,outer_points)', 'rows');
    loc = find(ismember( MONS(:,outer_points)', pnew', 'rows'));
    if test
        fprintf('NN(%i,.)=-M(%i,inner_points);\n', i ,loc );
        NN(i,:) = -M(loc,inner_points);
    else
        error('not found, that is bad.');
```

end

```

end
end
```

the last part of the code then usually looks something like:

```

[V,D] = eig( NN );
SOLS= V(end-3:end,i) /V(end,i);
```