

The MuPAD solver

Part II

MuPAD seminar, 17.05.2001
Stefan Wehmeier



Last week ...

... we talked about the solver for equations. Today, our subject will be the other cases:

- systems
- inequalities
- preimages of sets



The system solver

Called if:

- the first argument to `solve` is a set, list, table, or array, or
- the second argument is a set or list, or
- the second argument is missing.

If the second argument is missing, the system is solved for all variables in it; they are determined by `solveLib::indets`.

Returns:

a condition on the variables equivalent to the system, or a set of lists of substitutions



```
>> solve({x^2 + 1 = y, y^2 - 1 = x}, {x, y})
```

```
{[x = 0, y = 1], [x = y2 - 1, y =
```

```
RootOf(X3 - X2 + X - 2, X)]}]
```

```
>> solve(cos(x) = 1)
```

```
x in { 2*PI*X35 | X35 in Z_ }
```

To do:

Introduce a unique output format.



Subroutines of the system solver

The system solver splits into subroutines for the following cases:

- Systems that contain inequalities. Normally, these cannot be solved.
- Linear systems. They are solved using Gaussian elimination.
- Polynomial systems.
- Systems with only one equation. This case is passed to the equation solver.

If no method applies, an unevaluated call to `solve` is returned.



ToDo's in the system solver

Still a lot is to be added:

To do:

Replace square roots by new identifiers, and add the minimal polynomials to the system.

To do:

Non-algebraic triangular systems.

To do:

Correct handling of parametrized equations using piecewise.



Solving algebraic systems

Let a system $\{f_1 = 0, \dots, f_n = 0\}$ in the variables x_1, \dots, x_k be given.

The solver proceeds as follows:

- Compute a Groebner basis of the ideal I generated by the f_i .
- Factor: try to write the variety of I as a union of varieties (described as zero-sets of polynomial ideals); if this succeeds, find the zeroes of the ideals separately
- Determine a *triangular set*, that is, polynomials g_1, \dots, g_k with g_k only depending on x_k , g_{k-1} only depending on x_{k-1} and x_k , etc.
- Solve g_k for x_k , substitute the solutions into g_{k-1} , solve this for x_{k-1} , etc.



NOTE THAT:

This is known to work only for zero-dimensional systems (systems with finitely many solutions). In other cases, the solver produces nonsense “solutions” at the moment !

To do:

I'm just working on

- parametrized systems
- systems of dimension ≥ 1



Example

Consider the system

$$\begin{aligned}2xy^2 - xy - x + y^2 - y &= 0 \\x^2y + x^2 + xy^2 + 1 &= 0\end{aligned}$$

```
>> sys:=map([2*x*y^2-x*y-x+y^2-y, x^2+x^2*y+x*y^2+1], poly, [x, y])
```

```
      2          2  
[poly(2 x y  - x y - x + y  - y, [x, y]),
```

```
      2      2      2  
poly(x  y + x  + x y  + 1, [x, y])]
```



To solve this, we first compute a Groebner basis:

```
>> bas:= groebner::gbasis(sys, LexOrder)
```

```
poly(2 x2 + x4 - 2 y3 + 4 y3 + y2 - 3 y + 1, [x, y]),
```

```
poly(x y4 - x3 - 4 y3 + 6 y3 + 7 y2 - 5 y - 4, [x, y]),
```

```
poly(2 y5 - 2 y4 - 5 y3 + y2 + 3 y + 1, [x, y])]
```

Remark:

It is known that Groebner bases with respect to lexicographical order are difficult to compute. Therefore, MuPAD proceeds as follows: it first computes a Groebner basis with respect to a degree ordering (DegInvLexOrder) and applies a Groebner basis computation w.r.t. LexOrder to the result.



To do:

We should use walk methods, but we would need weighted degree orderings in the kernel to do that.

The last polynomial in bas can be factored:

```
>> factor(op(bas, 3))  
  
poly(y - 1, [x, y]) poly(y + 1, [x, y])  
  
      3      2  
poly(2 y  - 2 y  - 3 y - 1, [x, y])
```

Hence, we are in the following situation: we want to solve $f_1 = 0, f_2 = 0, g_1 g_2 g_3 = 0$. We may write the set of solutions as union of the solutions to $f_1 = 0, f_2 = 0, g_i = 0, 1 \leq i \leq 3$. For each of these systems, we start again and compute a new Groebner basis. This is done by `groebner::factor`:



```

>> fac:= groebner::factor(bas)

{[poly(x + 1, [x, y]), poly(y + 1, [x, y])],

  [poly(2 x2 + x + 1, [x, y]), poly(y - 1, [x, y])],

  [poly(x - 2 y2 + 3 y + 2, [x, y]),

  poly(2 y3 - 2 y2 - 3 y - 1, [x, y])]}

```

Each of these Groebner bases happens to be in triangular form.



For each list, **MuPAD** now solves the last polynomial in the list, substitutes each solution for y in the first polynomial, and solves the first polynomial.

```
>> solve(sys, [x, y])
```

$$\{[x = -1, y = -1], [x = 2y^2 - 3y - 2,$$

$$y = \text{RootOf}(2X^3 - 2X^2 - 3X - 1, X)],$$

$$[x = -\frac{1}{4}i\sqrt{7} - \frac{1}{4}, y = 1],$$

$$[x = \frac{1}{4}i\sqrt{7} - \frac{1}{4}, y = 1]\}$$


The inequality solver

Called if:

the first argument to `solve` is an inequality.

Returns:

an expression equivalent to the input if one of the arguments is a list or set; otherwise, the set of all solutions. (This is exactly the same distinction as between solving equations and systems of equations.)

NOTE THAT:

The variable to solve for and all parameters are implicitly supposed to be real.



Remark:

At the moment, (systems of) inequalities in more than one unknown cannot be solved.

To do:

In order to know e.g. where some polynomial $f(x, y)$ takes on positive values, we would need to determine the connected components of $\{(x, y) \in \mathbb{R}^2; f(x, y) \neq 0\}$, and test one point in each component. There is some literature about polynomial inequalities (“cylindrical algebraic decomposition”), would require some work.



The algorithm for inequalities

Suppose that $f(x) < 0$ is to be solved. (Other types of inequalities are handled the same way.)

The basic algorithm is as follows: **MuPAD** determines the set of all x for which $f(x)$ is real, and intersect that set with the set obtained as follows:

- Solutions to $f(x) = 0$, and discontinuities of f are called critical points; determine these, add $-\infty$ and ∞ .
- Order the critical points. The order may depend on free parameters if there are such in f ; then each possible case has to be handled separately.
- Between each two critical points, choose a point at random and check whether the inequality is satisfied there; if yes, then it is satisfied in the whole interval between the critical points.



NOTE THAT:

This part of the solver has still many bugs, in particular for parametrized inequalities.

```
>> solve(a*x^2 > 1, x)
      /
piecewise| {} if a = 0, ]1/a^(1/2), infinity[ union
      |
      \
      ]-infinity, -1/a^(1/2)[ if undefined,
      .....

```

Bugs are sometimes due to problems in recursive solve calls on the conditions or property problems.



An example for solving an inequality

We want to solve $a * x > 1$ for x .

Given our implicit assumption, the left hand side takes on real values for all $x \in \mathbb{R}$, and there are no discontinuities:

```
>> assume({a,x}, Type::Real):  
>> solve(a*x in R_, x)  
  
                                     R_  
>> discont(a*x, x)  
  
                                     {}
```



The corresponding equation has the solutions

```
>> solve(a*x = 1, x)
```

```
      / { 1 } \
piecewise| { - } if a <> 0, {} if a = 0 |
      \ { a } /
```

In the case $a = 0$, the inequality $0 > 1$ has no solution; otherwise, we test the points $x = 1/a - 1$ and obtain inequalities where only the identifier a appears; eventually, we get

```
>> solve(a*x > 1, x)
```

```
piecewise([-infinity, 1/a[ if a < 0 and not 0 < a,
```

```
  ]1/a, infinity[ if not a < 0 and 0 < a,
```

```
  {} if not a < 0 and not 0 < a and a <> 0, {} if a = 0)
```



To do:

The simplifier is being improved; one day we will find out that $a < 0$ and not $0 < a$ can be simplified to $a < 0$.



Another example showing a few of the problems

We want to solve $x^2 > y^2$ for x .

The corresponding equation has two solutions $-y$ and y , expressed in a complicated way by **MuPAD**:

```
>> solve(x^2=y^2, x)
```

```
      2 1/2      2 1/2  
{(y ) , - (y ) }
```



Luckily, the sort method is happy with this, finding out that no case analysis is necessary and $-(y^2)^{1/2} \leq (y^2)^{1/2}$ for all y .

```
>> solvelib::conditionalSort([op(%)])  
  
      2 1/2    2 1/2  
[- (y )    , (y )    ]
```



Hence we insert three points for x (one smaller than the left solution, one larger than the right solution, and one between the two solutions). This gives us inequalities depending only on y ; solving these gives a case analysis

```
>> solve(x^2 > y^2, x)

piecewise([y, infinity[ union ]-infinity, -y[ if 0 <= y,
  {} if 0 <= y and not -1/2 < y,
]-infinity, y[ union ]-y, infinity[ if y < 0,
  {} if y < 0 and not y < 1/2)
```

where, unfortunately, some conditions are not recognized as impossible.



The preImage function

Called if:

solve is called with a logical expression $f(x)$ in S , or directly as `solveLib::preImage(f(x), x, S)`.

Returns:

the set of all x for which $f(x) \in S$.

This is just a call to the preImage slot of the domain of S , i.e., a case analysis on the type of S .

To do:

The important case $S =$ the set of real numbers deserves more careful treatment than it is given now.



The handling of options

The function `solveLib::getoptions(opt1,...,optn)` returns a table of all options that are valid if the default options are overridden by `opt1` through `optn`.

For example, if no defaults are overridden,

```
>> solveLib::getoptions()

      table(
        "DontRewriteBySystem" = FALSE,
        "IgnoreSpecialCases" = FALSE,
        "Domain" = Expr,
        "BackSubstitution" = TRUE,
        "PrincipalValue" = FALSE,
        "Multiple" = FALSE,
        "MaxDegree" = 2
      )
```



To do:

For every function environment f , we could do the following:

- $f::\text{defaultOptions}$ is a table containing the default options to f ;
- $f::\text{getOptions}$ is a function that, applied to the optional arguments of f , returns a table where the default options are replaced by the options actually passed to f
- $\text{setDefaultOptions}(f, \dots)$ is a function that modifies the default options attached to f
- $f::\text{originalDefaultOptions}$ could be a table with the original default options for f , used to undo the user's changes if desired



The meaning of the options

Most options have only minor influence.

`DontRewriteBySystem` do not reduce solving an equation to solving an algebraic system

`IgnoreSpecialCases` a heuristic introduced to avoid piecewise defined solutions

`Domain=d` solve over a domain different from \mathbb{C} . This is simply done by solving over \mathbb{C} and intersecting with the given domain.

`BackSubstitution=FALSE` for systems: the solution for one variable may depend on another variable to solve for

`PrincipalValue` obtain only one solution



Multiple determine multiplicity of roots (works for polynomial equations only)

MaxDegree= n use explicit formulas for solving polynomials up to degree n

To do:

Write specialised methods for the most important domains d .



Weaknesses and subjects to discuss

The following points have been made:

- The solver has bugs. (True.)
- The solver produces complicated output that nobody can understand. (True, depending on the difficulty of the input.)
- It is difficult to use the results for further computations. (This may be true, but what would you like to do e.g. if

$$\{1/2 + k\pi; k \in \mathbb{Z}\} \cup \{k\pi; k \in \mathbb{Z}\}$$

is the mathematically correct set of solutions to your equation?)

- The solver is slow. (True for many cases. It would be possible to speed up by skipping some heuristics, at the price of getting a solution in fewer cases.)



- The solver is “too exact”. It should make sensible assumptions on the free parameters. (This made me introduce the option `IgnoreSpecialCases`. What do you mean by sensible?)
- The user should be able to proceed step by step, see `slib/apply`. (Would be possible to implement.)
- `solve` should be essentially inert, like `RootOf`, with the unevaluated call representing a set that can be manipulated using function attributes. Reason: `solve` is expensive, but evaluation should *a/ways* be cheap. (Interesting idea.)
- ... and many more suggestions I don't recall now.

Feel free to discuss whatever you want!

