Research Report AI–1989–02

A Numerical Equation Solver in Prolog

Michael A. Covington

Artificial Intelligence Programs
The University of Georgia
Athens, Georgia 30602 U.S.A.

A Numerical Equation Solver in Prolog

Michael A. Covington
Artificial Intelligence Programs
The University of Georgia
Athens, Georgia 30602

March 1989

Abstract

The Prolog inference engine can be extended to solve for unknowns in arithmetic equations such as X−1=1/X or X=cos(X), whether or not the equations have analytic solutions. This is done by standard numerical methods, but two features of Prolog make the implementation easy: the ability to treat expressions as data and the ability of the program to extend itself at run time.

1 The problem

The Prolog inference engine can solve for any unknown in symbolic queries, but not in arithmetic queries. For example, given the fact

```
        father(michael,sharon).
```

one can ask

```
        ?- father(michael,X).
```

and get the answer sharon, or ask

```
        ?- father(X,sharon).
```

and get the answer `michael`. This interchangeability of unknowns extends to complex symbolic manipulations (e.g., `append` can be used to split a list as well as to concatenate lists), but not to arithmetic.

Prolog handles arithmetic the way Fortran did thirty years ago: the unknown can only be a single variable on the left of the operator `is`, and everything on the right must be known. Thus

```
?- X is 2 + 2.
```

gets the answer `4`, but

```
?- X is 1 + 1 / X.
```

fails or raises an error condition.

The excuse for this restriction is that Prolog cannot search the set of real numbers the way it searches the symbols in a knowledge base. As far as exhaustive search goes, this is true. However, mathematicians have been using *heuristic* searches to solve equations since the days of Isaac Newton. The procedures given here implement one such method, making it possible to have dialogues with the computer such as:

```
?- solve( X = 1 + 1 / X ).
X = 1.618034

?- solve( X = cos(X) ).
X = 0.739085
```

and so on.

2 The solution

Prolog is an ideal language for solving equations for two reasons: equations can be treated as data, and the program can modify itself. A procedure can accept expressions as parameters, then evaluate them or even create procedures to evaluate them. In Pascal or C, by contrast, there is no simple way to introduce a wholly new equation into the program at run time.

The program given here solves equations by the *secant method*, which is one of the simplest numerical methods, though not the most robust. A different method can easily be substituted once the framework of the program is in place. Standard (Edinburgh-compatible) Prolog is required; Turbo Prolog programs cannot modify themselves in the necessary way.

To solve the equation

$$\mathit{Left} = \mathit{Right}$$

the secant method uses the function

$$\mathit{Dif(x)} = \mathit{Left} - \mathit{Right}$$

where *Left* and *Right* are expressions that contain x. The problem is then to search for an x such that *Dif(x)* $= 0$.

The search is begun by taking two guesses at x and comparing the values of *Dif(x)* for each. One of them will normally be closer to zero than the other. From this information the computer can tell whether to move toward higher or lower guesses. In fact, by assuming that *Dif(x)* increases or decreases linearly with x, the computer can estimate how far to move. (This is why it's called the secant method — given two guesses, the third guess is formed by extending a secant line on the graph of the function.)

Success is not guaranteed — the two *Dif* values could be equal, or the estimate of how far to move could be misleading — but the procedure usually converges on a solution in just a few iterations. Listing 1 shows the algorithm in pseudocode form.

3 Finding the unknown

Expressing this in Prolog boils down to two things: setting up the problem, and then performing the computation. The setting up is done by `solve`, which calls `free_in`, `define_dif`, and `solve_for` (Listing 2).

The first step is to identify the unknown — that is, to pick out the free variable in the equation to be solved. This is done by procedure `free_in`, which finds the free (uninstantiated) variables in any Prolog term. This is more general than what we need, but it's always useful to build a general-purpose tool.

If the term contains a free variable, there are two possibilities: either the term *is* the variable, in which case the search is over, or else the term has a variable somewhere in its argument structure. `free_in` has a clause for each of these cases.

The second case requires that the term be decomposed into a list. The built-in predicate "univ" (=..) does this. For example,

```
a(b,c(d),e) =.. [a,b,c(d),e].

2+3+X =.. ['+',2,3+X]
```

Even lists can be split this way, because any list is really a two-argument structure with the dot (`'.'`) as its principal functor. That is, the list `[a,b,c]` is really `a.(b.(c.[]))`, though not all Prologs allow you to write it that way. So,

```
[a,b,c] =.. ['.',a,[b,c]].
```

Thus a list is not a special case; it can be treated just like any other complex term.

In `free_in`, the statement

```
        Term =.. [_,Arg|Args].
```

discards the functor, which can't be a variable anyway, and obtains two things: the first argument, `Arg`, and the list of subsequent arguments, `Args`. It is then straightforward to search for variables in both `Arg` and `Args`. Further, because `Arg` can be a single variable, the first clause has a chance to terminate the recursion; and if it isn't, whatever is the first element of `Args` on this pass will be `Arg` on the next recursive pass and will get examined then.

There is, however, a special case to rule out. The term `[[]]` decomposes into `['.',[],[]]`, givng `Arg = []` and `Args = [[]]`, which would lead to an endless loop. For this reason, `free_in` explicitly tests for this term and rejects it.

4 Defining a procedure

The next step is to define a procedure to compute the *Dif* function. Recall that the argument of solve is an equation in the form `Left=Right`. Clearly, the *Dif* function is obtained by evaluating `Left-Right`. But how is this done?

There are two possibilities. One possibility would be to pass along the expression `Left-Right` and evaluate it whenever needed. This is easily done, because

```
        X is Y.
```

will evaluate whatever expression `Y` is instantiated to.

But the other, faster, possibility is to define a procedure to do the evaluating. That's what `define_dif` does; it creates a procedure such as

```
Dif(X,Dif) :- Dif is X - cos(X).
```

using whatever expressions the user originally supplied. The result is a procedure called `dif` that accepts a value of `X` and returns the corresponding value of the *Dif* function. In ALS Prolog, this `dif` procedure is compiled into threaded code when `assert` places it into the program; it runs just as fast as if it had been supplied by the original programmer. Other Prologs run it interpretively.

What connects the variable `X` to the expression `Left-Right` in which it is supposed to occur? This question addresses the heart of Prolog's variable scoping system. It isn't enough simply that it is called `X`; like-named variables in Prolog are not the same unless they occur in the same rule, fact, or goal.

That's why so many goals in this program have both `X` and `Left=Right` (or `Left-Right`) as arguments. Initially, `free_in` takes `Left` and `Right` and finds a variable in them. This variable may have any name, but it is unified with the variable `X` in `solve`. This same `X` is then passed, along with `Left` and `Right`, to `define_dif`, which uses it in creating the `dif` procedure. Thereafter, the `X` in dif is guaranteed to be a variable that occurs in `Left-Right`, also in `dif`, regardless of what the user originally called it.

5 Solving the equation

The last step is to implement the secant method (Listing 3). The pseudocode in Listing 1 undergoes several changes when translated into Prolog.

First, Prolog has no loop constructs, so recursion is used instead. The loop is replaced by the procedure `solve_aux`, which calls itself. Because the recursive call is the last step of a procedure with no untried alternatives, the compiler converts it back into a loop in machine language, but conceptually, the programmer thinks in terms of recursion.

Second, in Prolog there is no way to change the value of an instantiated variable. This means, for example, that there is no Prolog counterpart of

```
Guess1 := Guess2
```

when Guess1 already has a value. Instead, the proper Prolog technique is to pass a new value in the same argument position on the next recursive call. Thus the procedure that begins with

```
solve_aux(...Guess1,Dif1,Guess2...) :- ...
```

ends with the recursive call

```
... solve_aux(...Guess2,Dif2,Guess3...).
```

Third, there are minor rearrangements to avoid computing *Dif(x)* more than once with the same value of X. These include the variable `Dif2` and the passing of `Dif1` as an argument from the previous recursive pass.

6 Limits and possibilities

This program is intended as a demonstration of the integration of numerical methods into Prolog, not as a demonstration of numerical methods per se.

The secant method is simple, but far from perfect. It has trouble with some equations. For example, if two successive *Dif* values happen to be exactly the same distance from zero, then `solve_aux` will try to divide by zero. This simply fails in ALS Prolog but may cause an error message in other Prologs. This problem shows up with the equation

```
X^2 - 3*X = 0
```

which has *Dif* $=-2$ for both of the first two guesses (1 and 2). With a case very close to this, such as `X^2 - 3.01*X = 0`, we find that although the method should work in principle, in practice the next guess is a long way from the correct solution, and the guesses run wildly out of the range of representable numbers. And with some equations, the guesses will bounce back and forth between two values, not getting better with successive iterations [1].

More robust numerical methods can easily be substituted into the same program framework. The ability to solve for more than one unknown is desirable; this could be treated as a multi-variable minimization problem where the goal is to minimize *abs(Dif(x))* [2]. It is possible to solve systems of nonlinear equations by reducing them to systems of linear equations, which can then be solved by conventional methods.

The program was written in ALS Prolog and has been tested in Quintus Prolog. However, other Prologs may require minor modifications. For example, Arity Prolog 4.0 requires spaces before certain opening parentheses (e.g., `2 + (3+4) + 5` rather than `2+(3+4)+5`). And it is a general limitation of real-number arithmetic that a negative number cannot be raised to a non-integer power (i.e., `4^2.5` is all right but `(-4)^2.5` is not). Some Prologs assume all exponents are non-integer.

References

[1] Hamming, Richard W., *Introduction to Applied Numerical Analysis* (New York: McGraw-Hill, 1971), especially pp. 33–55.

[2] Press, William H.; Flannery, Brian P.; Teukolsky, Saul A.; and Vetterling, William T., *Numerical Recipes: The Art of Scientific Computing* (Cambridge: Cambridge University Press, 1986), especially pp. 240–334.

Listing 1. The secant method algorithm in pseudocode form.

```
To solve **Left = Right:**

   function Dif(X) = Left − Right
      where X occurs in Left and/or Right;

   procedure Solve;
   begin
      Guess1 := 1;
      Guess2 := 2;
      repeat
         Slope := (Dif(Guess2)-Dif(Guess1))/(Guess2-Guess1);
         Guess1 := Guess2;
         Guess2 := Guess2 - (Dif(Guess2)/Slope)
      until Guess2 is sufficiently close to Guess1;
      result is Guess2
   end.
```

Listing 2. Procedures to set up the problem.

```
% solve(Left=Right)
%
% On entry, Left=Right is an arithmetic
% equation containing an uninstantiated
% variable.
%
% On exit, that variable is instantiated
% to an approximate numeric solution.
%
% The syntax of Left and Right is the same
% as for expressions for the 'is' predicate.

solve(Left=Right) :-
  free_in(Left=Right,X),
  !,/* accept only one solution of free_in */
  define_dif(X,Left=Right),
  solve_for(X).


% free_in(Term,Variable)
%
% Variable occurs in Term and is uninstantiated.

free_in(X,X) :-         % An atomic term
  var(X).

free_in(Term,X) :-      % A complex term
  Term \== [[]],
  Term =.. [_,Arg|Args],
  (free_in(Arg,X) ; free_in(Args,X)).


% define_dif(X,Left=Right)
%
```

```
% Defines a predicate to compute Left-Right
% for the specified equation, given X.

define_dif(X,Left=Right) :-
  abolish(dif,2),
  assert((dif(X,Dif) :- Dif is Left-Right)).
```

Listing 3. Procedures to implement the secant method.

```
% solve_for(Variable)
%
% Sets up arguments and calls solve_aux (below).

solve_for(Variable) :-
  dif(1,Dif1),
  solve_aux(Variable,1,Dif1,2,1).


% solve_aux(Variable,Guess1,Dif1,Guess2,Iteration)
%
% Uses the secant method to find a value of
% Variable that will make the 'dif' procedure
% return a value very close to zero.
%
% Arguments are:
%  Variable  -- Will contain result.
%  Guess1    -- Previous estimated value.
%  Dif1      -- What 'dif' gave with Guess1.
%  Guess2    -- A better estimate.
%  Iteration -- Count of tries taken.


solve_aux(cannot_solve,_,_,_,100) :-
  !,
  write('[Gave up at 100th iteration]'),nl,
  fail.

solve_aux(Guess2,Guess1,_,Guess2,_) :-
  close_enough(Guess1,Guess2),
  !,
  write('[Found a satisfactory solution]'),nl.

solve_aux(Variable,Guess1,Dif1,Guess2,Iteration) :-
```

```
  write([Guess2]),nl,
  dif(Guess2,Dif2),
  Slope is (Dif2-Dif1) / (Guess2-Guess1),
  Guess3 is Guess2 - (Dif2/Slope),
  NewIteration is Iteration + 1,
  solve_aux(Variable,Guess2,Dif2,Guess3,NewIteration).


% close_enough(X,Y)
%
% True if X and Y are the same number to
% within a factor of 0.0001.
%

close_enough(X,Y) :-
  Quot is X / Y,
  Quot > 0.9999,
  Quot < 1.0001.
```