```
/* interv.
Load files for testing interval stuff
Use with UTIL
Alan Bundy 15.6.81 */

:- compile([
        int,                % My interval stuff
'extras:sportr'             % General portray
]).

:- [
        test,               % Test examples for paper
        probs,              % Current problems
'arith:odds',               % odd stuff from eval
        qik                 % Quick save and restore
].
```

```
/* INT : Finds intervals of terms in PRESS

                                                   Alan Bundy
                                                   Updated: 25 June 81
        Alan Bundy 19.12.79
        Revised version 14.3.80
        Further revised 26.3.81
        Cosmetics by Lawrence 18 June 81
*/

/* EXPORT */

  :- public      vet/2,
                 positive/1,
                 negative/1,
                 non_neg/1,
                 non_pos/1,
                 non_zero/1,
                 acute/1,
                 obtuse/1,
                 non_reflex/1,

                 find_int/2,             % Exported for convenience
                 ~~apply_int/3.~~


/* IMPORT */
/*
                 error/3                 from  UTIL:TRACE

                 memberchk/2             from  UTIL:SETROU

                 number/1                from  LONG
                 eval/1
                 eval/2

                 measure/2               from  notional Mecho database
                 quantity/1
                 angle/3
                 incline/3
                 concavity/2
                 slope/2
                 partition/2
*/


/* MODES */

  :- mode        vet(+,?),
                 positive(+),
                 negative(+),
                 non_neg(+),
                 non_pos(+),
                 non_zero(+),
                 acute(+),
                 obtuse(+),
                 non_reflex(+),

                     gen_combine(+,?),
                     combine(+,+,?),
```

```
                    in(+,+),
                    sub_int(+,+),
                    below(+,+),
                    disjoint(+,+),
                    overlap(+,+),
                    marker_flip(?,?),

                        default_interval(?),
                    find_int(+,?),
                        find_int2(+,-),
                        find_int_args(+,-,-),
                        find_simple_int(+,-),
                            make_assumption_positive(+),

                    int_apply(+,+,-),
                        int_apply_all(+,+,-),
                        all_are_contained(+,+),
                        make_regions(+,+,-),
                            split(+,+,+,-),
                                split1(+,+,-),
                            cartesian_product(+,+,-,?),
                                cart_prod(+,+,+,-,?),
                        find_limits(+,+,+,-),
                            clean_up(+,-),
                            limits(+,+,+,+,?),
                            set_bnds(+,+,+,-),
                                updown_flip(+,+,-),
                                set_bnd(+,+,-),

                    order(+,+,?,?),
                    less_than(+,+),
                    calc(+,+,?),
                        breakup_bnds(+,-,-),
                    comb(+,?),

                    mono(+,?,?),

                    classify(+,-),
                        interval(+,-,-),
                        collect_intervals(+,+,-),
                        quad(+,+,+,?).


/*
    Data structures

        <interval>      has form        i(LMarker,Bottom,Top,RMarker)
        <boundary>      has form        b(N,Marker)

                where:
                        Bottom, Top, N              are   <numbers>
                        LMarker, RMarker, Marker    are   one of {open,closed}

        An interval ranges between Bottom and Top and is open or closed at
        the ends depending on LMarker (for Bottom) and RMarker (for Top).

        A boundary is an end of an interval. There are operations defined
        over these boundaries which are then used to help define the
        operations over intervals. Note that the notion of a boundary does
        NOT involve any specific end of an interval (ie Top/Bottom). They
```

```
        are a generalisation over all such ends.

*/

%% @@@ - marker (top of code)


/****************************************/
/* Use interval information - top level */
/****************************************/

                          % Check that solution is admissible

vet(true,true).

vet(false,false).

vet(A&B,A1&B1) :- vet(A,A1), vet(B,B1).

vet(A#B,A1#B1) :- vet(A,A1), vet(B,B1).

vet(A=B,A=B) :-
        find_int(A,IntA), find_int(B,IntB),
        overlap(IntA,IntB),
        !.

vet(A=B,false).


                          % X is positive, negative, acute, etc.

positive(X) :- find_int(X,i(L,B,T,R)), less_than(b(0,closed),b(B,L)).

negative(X) :- find_int(X,i(L,B,T,R)), less_than(b(T,R),b(0,closed)).

non_neg(X) :- find_int(X,i(L,B,T,R)), less_than(b(0,open),b(B,L)).

non_pos(X) :- find_int(X,i(L,B,T,R)), less_than(b(T,R),b(0,open)).

non_zero(X^N) :- !, non_zero(X).        %ad hoc patch (replaces negative(N))

non_zero(X) :-
        find_int(X,i(L,B,T,R)),
        ( less_than(b(0,closed),b(B,L)) ; less_than(b(T,R),b(0,closed)) ),
        !.

acute(X) :-
        find_int(X,i(L,B,T,R)),
        less_than(b(0,open),b(B,L)),
        less_than(b(T,R),b(90,open)).

obtuse(X) :-
        find_int(X,i(L,B,T,R)),
        less_than(b(90,open),b(B,L)),
        less_than(b(T,R),b(180,open)).

non_reflex(X) :-
        find_int(X,i(L,B,T,R)),
        less_than(b(0,open),b(B,L)),
        less_than(b(T,R),b(180,open)).
```

```
/*****************************************/
/*      Manipulating Intervals           */
/*****************************************/


                        % Combine a list of intervals by sweeping list and
                        %  accumulating the combined intervals.

gen_combine([FirstInt|RestInts],Result)
     :- gen_combine(RestInts,FirstInt,Result).


gen_combine([],Result,Result).

gen_combine([Int|RestInts],Acc,Result)
     :- combine(Int,Acc,NewAcc),
        gen_combine(RestInts,NewAcc,Result).


                        % Combine x and y intervals

combine(i(Lx,Bx,Tx,Rx), i(Ly,By,Ty,Ry), i(L,B,T,R)) :-
        order(b(Tx,Rx),b(Ty,Ry),_,b(T,R)),
        order(b(Bx,Lx),b(By,Ly),b(B,L),_).


                        % Number N is contained in interval

in(N,i(L,B,T,R)) :- !,
        sub_int(i(closed,N,N,closed),i(L,B,T,R)).


                        % x interval is contained in second interval

sub_int(i(Lx,Bx,Tx,Rx),i(L,B,T,R)) :-
        marker_flip(L,L1), marker_flip(R,R1),
        less_than(b(B,L1),b(Bx,Lx)), less_than(b(Tx,Rx),b(T,R1)).

                        % x interval is wholly below y interval

below(i(Lx,Bx,Tx,Rx),i(Ly,By,Ty,Ry)) :-
        less_than(b(Tx,Rx),b(By,Ly)), !.

                        % x and y intervals are disjoint

disjoint(IntX,IntY) :- below(IntX,IntY), !.
disjoint(IntX,IntY) :- below(IntY,IntX), !.

                        % x and y intervals overlap

%% overlap(IntX,IntY) :- not disjoint(IntX,IntY).

overlap(IntX,IntY) :- disjoint(IntX,IntY), !, fail.
overlap(_,_).


                        % open and closed are opposites
```

```
                        %  (this is how to flip them)

marker_flip(open,closed) :- !.
marker_flip(closed,open).



/****************************************/
/* X lies in closed or open interval    */
/****************************************/


                % Worst case default for intervals

default_interval(i(open,neginfinity,infinity,open)).


                % Lets try to do better..

find_int(X,Interval)
     :- find_int2(X,Result),            % guarantee mode (+,-)
        Interval = Result.

                        % Catch variables (shouldn't be there!)

find_int2(V,_)
     :- var(V),
        !,
        error('Interval package given variable: %w',[V],fail).

                        % Base cases
                        %  Numbers have point intervals
                        %  Symbols (atoms) have various special cases

find_int2(X,i(closed,X,X,closed)) :- number(X), !.

find_int2(X,Interval) :- atom(X), !, find_simple_int(X,Interval).


                        % Special case normalisation

                        % Convert ^(-1) to 1/

find_int2(X^(-1), Int) :- !,
        find_int2(1/X, Int).

                        % Deal with exponentials to even power

find_int2(X^N, i(L,B,T,R)) :-
        even(N), !,
        find_int2(abs(X), i(Lx,Bx,Tx,Rx)),
        calc(^,[b(Bx,Lx),b(N,closed)],b(B,L)),
        calc(^,[b(Tx,Rx),b(N,closed)],b(T,R)).

                        % Convert cosecant to sine

find_int2(csc(X), Int) :- !, find_int2(1/sin(X), Int).

                        % Convert secant to cosine
```

```
find_int2(sec(X), Int) :- !, find_int2(1/cos(X), Int).

                        % Convert cotangent to tangent

find_int2(cot(X), Int) :- !, find_int2(1/tan(X), Int).


                        % General case
                        %  Recursively find intervals for arguments and
                        %  then int_apply to sort this out. This will use
                        %  monotonicity of F to calculate interval of Term
                        %  from arguments.

find_int2(Term,Int) :-
        find_int_args(Term,F,IntList),
        int_apply(F,IntList,Int),
        !.


                        % If the general case fails

find_int2(sin(X), i(closed,(-1),1,closed)) :- !.
find_int2(cos(X), i(closed,(-1),1,closed)) :- !.

find_int2(X,Default) :- default_interval(Default).



                        % Find a list of intervals corresponding to the
                        %  arguments of Term. Also return the functor.

find_int_args(Term,Fn,IntList)
     :- functor(Term,Fn,Arity),
        find_int_args(1,Arity,Term,IntList).


find_int_args(N,Max,_,[]) :- N > Max, !.

find_int_args(N,Max,Term,[Int|IntRest])
     :- arg(N,Term,Arg),
        find_int2(Arg,Int),
        N1 is N+1,
        find_int_args(N1,Max,Term,IntRest).



                        % Find the interval for a simple symbol
                        %  This involves looking to see if we know
                        %  anything special about the symbol which will
                        %  help us.
                        % Ad hoc patch for gravity - proper solution means
                        %  allowing equations between quantities and defining
                        %  g as measure(g,32,ft/sec^2).
                        % Otherwise try to classify symbol (if it is an angle)
                        % Otherwise assume all quantities are positive
                        %       (possibly extreme?)
                        % If there is no useful info we must use the default.

find_simple_int(g,i(open,1,infinity,open)) :- !.
```

```
find_simple_int(X,Int) :- classify(X,Int), !.

find_simple_int(M,i(open,0,infinity,open)) :-
        measure(Q,M), quantity(Q),
        !,
        make_assumption_positive(M).

find_simple_int(X,Default) :- default_interval(Default).



                        % Make and remember assumption

make_assumption_positive(X) :- assumed_positive(X), !.

make_assumption_positive(X)
     :- assert( assumed_positive(X) ),
        trace('I assume %t positive.\n',[X],1).



/*************************************************************/
/* Find interval of function from intervals of its arguments */
/*************************************************************/


                                % Simple case

int_apply(F,Region,Int) :-
        mono(F,Is,Mono),
        all_are_contained(Region,Is),
        !,
        find_limits(F,Region,Mono,Int).

                                % Complex Case

int_apply(F,Region,Int) :-
        mono(F,MRegion,Mono),
        make_regions(Region,MRegion,NewRegions),
        int_apply_all(NewRegions,F,IntervalSet),
        !,
        gen_combine(IntervalSet,Int).



                        % int_apply all intervals in a set (list)

int_apply_all([],_,[]).

int_apply_all([Region1|Rest],F,[Int1|IRest])
     :- int_apply(F,Region1,Int1),
        int_apply_all(Rest,F,IRest).



                        % All the argument intervals are sub intervals of
                        %  the corresponding monotonic intervals for the
                        %  function (from mono). (ie maplist sub_int down
                        %  the two "argument" lists).
```

```
all_are_contained([],[]).

all_are_contained([ArgInt|ArgRest],[FInt|FRest])
     :- sub_int(ArgInt,FInt),
        all_are_contained(ArgRest,FRest).



                        % Given the list of actual intervals and the list
                        %  of monotonic intervals for the function build
                        %  a set of similar interval lists, derived from the
                        %  actual interval list, but such that each element
                        %  of each list in the set is wholly inside or outside
                        %  its corresponding monotonic function interval.
                        % This amounts to case splitting the actual interval
                        %  list into a set of intervals for more tractable
                        %  (sub) regions in the nD space.
                        % Implemented by splitting lists to form a list of
                        %  sets and taking the nD cartesian product. Note
                        %  that both split/4 and cartesian_product/4 perform
                        %  order reversals - which cancel each other out.

make_regions(Region,MRegion,NewRegions)
     :- split(Region,MRegion,[],ListOfSets),
        cartesian_product(ListOfSets,[],NewRegions,[]).



                        % Given the list of actual intervals and the list of
                        %  monotonic intervals for the function, we build
                        %  a list of n sets, where n is the arity of the
                        %  function (ie the length of the lists) and where
                        %  each set contains intervals which are wholly inside
                        %  or outside the corresponding monotonic function
                        %  intervals, such that the intervals in each set
                        %  would combine to form the corresponding actual
                        %  interval.
                        % The combining property follows from the way we split
                        %  up the actual intervals.
                        % The sets produced at the moment will only ever have
                        %  number of members m such that: 1 =< m =< 3.
                        %  The following special representations are used for
                        %  these cases:
                        %               singleton(A)
                        %               pair(A,B)
                        %               triple(A,B,C)
                        %  In fact the code will currently never produce sets
                        %  of 3 elements (triples), but I (Lawrence) think
                        %  this is probably a bug so have left the option, and
                        %  this comment, around til we see.
                        % Note that the list of sets built will be in reverse
                        %  order compared with the "argument" lists. This is
                        %  is implemented by an extra accumulator argument
                        %  (should be [] to start with) onto which each Set
                        %  is pushed.

split([],[],Result,Result).

split([ArgInt|ArgRest],[FInt|FRest],Sofar,Result)
     :- split1(ArgInt,FInt,Set),
```

```
        split(ArgRest,FRest,[Set|Sofar],Result).


                                  % Intx wholly within Int

split1(Intx,Int,singleton(Intx)) :-
        sub_int(Intx,Int),
        !.

                                  % Intx and Int overlap with Intx leftmost

split1(i(Lx,Bx,Tx,Rx), i(L,B,T,R), pair(i(L,B,Tx,Rx),i(Lx,Bx,B1,L1)) ) :-
        marker_flip(R,R1), marker_flip(L,L1),
        marker_flip(Lx,Lx1),
        correct(B,B1),
        less_than(b(Tx,Rx),b(T,R1)),
        not less_than(b(Tx,Rx),b(B,L)),
        less_than(b(Bx,Lx1),b(B,L)), !.



                          % Given a list of n sets produce the a set of the
                          %  elements from the nD cartesian product of the sets.
                          %  The incoming sets are represented with special
                          %  functors as there are only a few special cases (see
                          %  split). The resulting product set is represented as
                          %  a list. Each element will itself be a list (of n
                          %  intervals) where the order of this element list will
                          %  be the reverse of the order in which the items
                          %  were found in the original list of sets.
                          % The implementation involves an accumulator for the
                          %  (partial) element being built and uses the
                          %  difference list technique to build the final set
                          %  of elements (repn as a list).

cartesian_product([],Element,[Element|Z],Z).

cartesian_product([First|Rest],PartialElement,ProductSet,Z)
     :- cart_prod(First,Rest,PartialElement,ProductSet,Z).



cart_prod(singleton(A),Rest,PartialElement,PSet,Z)
     :- cartesian_product(Rest,[A|PartialElement],PSet,Z).

cart_prod(pair(A,B),Rest,PartialElement,PSet0,Z)
     :- cartesian_product(Rest,[A|PartialElement],PSet0,PSet1),
        cartesian_product(Rest,[B|PartialElement],PSet1,Z).

cart_prod(triple(A,B,C),Rest,PartialElement,PSet0,Z)
     :- cartesian_product(Rest,[A|PartialElement],PSet0,PSet1),
        cartesian_product(Rest,[B|PartialElement],PSet1,PSet2),
        cartesian_product(Rest,[C|PartialElement],PSet2,Z).



                          % Calculate Bottom and Top of Interval

find_limits(F,Region,Mono,Int) :-
        limits(bottom,F,Region,Mono,b(B,L)),
```

```
        limits(top,F,Region,Mono,b(T,R)),
        clean_up(i(L,B,T,R), Int).


                        % Hack to clear up various funnies

clean_up(i(_,undefined,_,_), Int) :- !, default_interval(Int).
clean_up(i(_,_,undefined,_), Int) :- !, default_interval(Int).
clean_up(i(L,B,0,R), i(L,B,-(0),R)) :- !.
clean_up(Int, Int).

correct(0,-(0)) :- !.
correct(B,B) :- !.


                        % Calculate limit for a particular boundary

limits(TopBot,F,Region,Mono,Boundary)
     :- set_bnds(Mono,TopBot,Region,BoundaryList),
        calc(F,BoundaryList,Boundary).



                        % Form a boundary list from an interval list
                        %  given various details - up+down x top+bottom.

set_bnds([],_,[],[]).

set_bnds([Mono|MRest],TopBot,[Int|IRest],[Bnd|BRest])
     :- updown_flip(TopBot,Mono,NewMono),
        set_bnd(NewMono,Int,Bnd),
        set_bnds(MRest,TopBot,IRest,BRest).


        updown_flip(top,UD,UD).
        updown_flip(bottom,up,down) :- !.
        updown_flip(bottom,down,up).


        set_bnd(up,  i(L,B,T,R), b(T,R)).
        set_bnd(down,i(L,B,T,R), b(B,L)).




/****************************************/
/*      Manipulating Boundaries         */
/****************************************/


                        % Put boundaries in order

                                        % Boundaries are identical
order(Bnd,Bnd,Bnd,Bnd) :- !.
                                        % One of Mis is closed
order(b(N,M1),b(N,M2),b(N,closed),b(N,closed)) :- !.
                                        % Numbers are different, N1 smallest
order(b(N1,M1),b(N2,M2),b(N1,M1),b(N2,M2)) :-
        eval(N1 < N2), !.
                                        % N2 is smallest
```

```
order(b(N1,M1),b(N2,M2),b(N2,M2),b(N1,M1)).



                         % Ordering of boundaries
                         %  (assumes intervals are consecutive)

less_than(b(X,Mx),b(Y,My)) :-
        comb([Mx,My],M),
        less_than_eval(M,X,Y).


less_than_eval(open,X,Y) :- eval( X =< Y ).

less_than_eval(closed,X,Y) :- eval( X < Y ).



                         % Apply Function F to a boundary list
                         %  Do this by combining the boundary markers and
                         %  applying F to the numbers.

calc(F,BoundaryList,b(X,M)) :-
        breakup_bnds(BoundaryList,Markers,Numbers),
        comb(Markers,M),
        Term =.. [F|Numbers],
        eval(Term,X),
        !.


breakup_bnds([],[],[]).

breakup_bnds([b(N,M)|Rest],[M|MRest],[N|NRest])
     :- breakup_bnds(Rest,MRest,NRest).



                         % Combine boundary markers
                         %  Result = open if any of the inputs is open

comb(MarkerList,Result) :- memberchk(open,MarkerList), !, Result = open.

comb(_,closed).



/************************************************/
/* Monotonicity of Functions in each Interval */
/************************************************/

/* unary minus */
mono(-, [i(closed,neginfinity,infinity,closed)], [down]).

/* addition */
mono(+,[i(closed,neginfinity,infinity,closed),
        i(closed,neginfinity,infinity,closed)], [up,up]).

/* binary minus */
mono(-,[i(closed,neginfinity,infinity,closed),
        i(closed,neginfinity,infinity,closed)], [up,down]).
```

```
/* absolute value */
mono(abs,[i(closed,neginfinity,-(0),closed)], [down]).
mono(abs,[i(closed,0,infinity,closed)], [up]).

/* multiplication */
mono(*,[i(closed,0,infinity,closed), i(closed,0,infinity,closed)],
        [up,up]).
mono(*,[i(closed,0,infinity,closed), i(closed,neginfinity,-(0),closed)],
        [down,up]).
mono(*,[i(closed,neginfinity,-(0),closed), i(closed,0,infinity,closed)],
        [up,down]).
mono(*,[i(closed,neginfinity,-(0),closed), i(closed,neginfinity,-(0),closed)],
        [down,down]).



/* division */
mono(/,[i(closed,0,infinity,closed), i(closed,0,infinity,closed)],
        [up,down]).
mono(/,[i(closed,0,infinity,closed), i(closed,neginfinity,-(0),closed)],
        [down,down]).
mono(/,[i(closed,neginfinity,-(0),closed), i(closed,0,infinity,closed)],
        [up,up]).
mono(/,[i(closed,neginfinity,-(0),closed), i(closed,neginfinity,-(0),closed)],
        [down,up]).


/* exponentiation */
mono(^,[i(open,0,infinity,closed),i(closed,0,infinity,closed)],
        [up,up]).
mono(^,[i(open,0,infinity,closed),i(closed,neginfinity,-(0),closed)],
        [down,up]).


/* logarithm */
mono(log,[i(closed,0,infinity,closed),i(closed,0,infinity,closed)],
        [down,up]).

/* sine */
mono(sin,[i(closed,(-90),90,closed)],[up]).
mono(sin,[i(closed,90,270,closed)],[down]).
mono(sin,[i(closed,270,450,closed)],[up]).

/* cosine */
mono(cos,[i(closed,0,180,closed)],[down]).
mono(cos,[i(closed,180,360,closed)],[up]).

/* tangent */
mono(tan,[i(open,(-90),90,open)],[up]).
mono(tan,[i(open,90,270,open)],[up]).
mono(tan,[i(open,270,450,open)],[up]).

/* inverse sine */
mono(arcsin,[i(closed,(-1),1,closed)],[up]).

/* inverse cosine */
mono(arccos,[i(closed,(-1),1,closed)],[down]).

/* inverse tangent */
```

```
mono(arctan,[i(open,neginfinity,infinity,open)],[up]).

/* inverse cosecant */
mono(arccsc,[i(closed,neginfinity,(-1),closed)],[down]).
mono(arccsc,[i(closed,1,infinity,closed)],[down]).

/* inverse secant */
mono(arcsec,[i(closed,neginfinity,(-1),closed)],[up]).
mono(arcsec,[i(closed,1,infinity,closed)],[up]).

/* inverse cotangent */
mono(arccot,[i(closed,neginfinity,-(0),open)],[down]).
mono(arccot,[i(open,0,infinity,closed)],[down]).




/**************************************************/
/*  Calculate Interval of Angle from Curve Type   */
/**************************************************/


                        % We classify a symbol using semantic information
                        %  from the (Mecho) database. Calls which are to
                        %  this database (notionally, Press does not really
                        %  share the same object-level database) are marked
                        %  as such.
                        % This method is only appropriate if the symbol is an
                        %  <angle>, and tries to find the interval of the
                        %  angle using general principles about curve types.

classify(Angle, Int ) :-
        measure(Q, Angle ),                     % database
        angle(Point, Q, Curve ), !,             % database
        interval(angle, Curve, Int ).

classify(Angle, Int ) :-
        measure(Q, Angle ),                     % database
        incline(Curve, Q, Point ), !,           % database
        interval(incline, Curve, Int ).



                        % Find interval from curve shape

                                % For simple curves
interval(AI, Curve, Int ) :-
        concavity(Curve, Conv ),                % database
        slope(Curve, Slope ), !,                % database
        quad(AI, Slope, Conv, Int ).

                                % For complex curves
interval(AI, Curve, Int ) :-
        partition(Curve, Clist ), !,            % database
        collect_intervals(Clist, AI, Rlist),
        gen_combine(Rlist, Int ).



                        % Collect up a list of intervals for all the parts
```

```
                              % of a partitioned curve.

collect_intervals([],_,[]).

collect_intervals([First|Rest],AI,[FirstInt|RestInt])
     :- interval(AI,First,FirstInt),
        collect_intervals(Rest,AI,RestInt).


                              % Information about properties of simple curves
                              %  The interval depends on both the slope and the
                              %  concavity.

quad(angle,left,right,i(closed,0,90,closed)) :- !.
quad(incline,left,right,i(closed,90,180,closed)) :- !.

quad(angle,right,right,i(closed,90,180,closed)) :- !.
quad(incline,right,right,i(closed,180,270,closed)) :- !.

quad(angle,left,left,i(closed,180,270,closed)) :- !.
  'ad(incline,left,left,i(closed,270,360,closed)) :- !.

quad(angle,right,left,i(closed,270,360,closed)) :- !.
quad(incline,right,left,i(closed,0,90,closed)) :- !.

quad(angle,left,stline,i(open,180,270,open)) :- !.
quad(incline,left,stline,i(open,270,360,open)) :- !.

quad(angle,right,stline,i(open,270,360,open)) :- !.
quad(incline,right,stline,i(open,0,90,open)) :- !.

quad(angle,hor,stline,i(closed,270,270,closed)) :- !.
quad(incline,hor,stline,i(closed,0,0,closed)) :- !.

quad(angle,vert,stline,i(closed,180,180,closed)) :- !.
quad(incline,vert,stline,i(closed,270,270,closed)) :- !.


   JOBS TO DO

        write symbolic version for finding max/mins

        use monotonicity in > >= etc Isolation rules
*/
```

```
/* probs.
CURRENT PROBLEMS*/

/* interval and eval problems */

pba(I) :- find_int(m1/(-m2),I).

pbb :- acute(x*2).

non_zero(a).

pbc :- non_zero((73^(1/2)+9)*(1/2)).
```

```
'* TEST.
est Examples for Paper
lan Bundy 15.6.81 */

'* Test run with timings */
un :- checklist(stats, [test1,test2,test3,test4,test5,test6,test7,
                              test8,test9]).

'* Tests */

est1(I) :- int_apply( sin, [i(open,30,90,closed)], I).          % I = (1/2,1]
est2(I) :- int_apply( sin, [i(open,30,150,closed)], I).         % I = [1/2,1]
est3(I) :- int_apply( abs, [i(open,(-1),1,closed)], I).         % I = [0,1]
est4(I) :-            % I = [-oo,-2/3)
        int_apply( /, [i(open,2,3,closed), i(closed,(-3),-(0),closed)], I).
est5(I) :-            % I = [-3,3]
        int_apply( *, [i(closed,(-3),-(0),open), i(closed,(-1),1,closed)], I).
est6(I) :-            % I = [1/2,2]
        int_apply( ^, [i(open,1,2,closed), i(closed,(-1),1,closed)], I).
est7(I) :- find_int( (-m1)*g/(m1+m2) , I).             % I = (-oo,0)
   8(I) :- find_int( (sin(theta)+2)/cos(phi) , I).     % I = (-oo,oo)
   9(I) :- find_int( log(2,sin(theta)), I).            % I = (-oo,0)



uantity(m1q).            measure(m1q,m1).        % m1 is positive
uantity(m2q).            measure(m2q,m2).        % m2 is positive

uantity(thetaq).         measure(thetaq,theta).
uantity(phiq).           measure(phiq,phi).

incline(path3,thetaq,point).    slope(path3,right).     concavity(path3,stline).
        % Hence theta is acute

angle(point,phiq,semi). partition(semi,[path1,path2]).
                         slope(path1,left).             concavity(path1,right).
                         slope(path2,right).            concavity(path2,right).
        % Hence phi is obtuse


/* un problem with statistics*/

stats(Name) :- Problem=..[Name,Arg], statistics(runtime,_),
                call(Problem), !, statistics(runtime,[ _, Time]),
                trace('\n%t took %t milliseconds and produced answer %t\n\n',
                        [Name,Time,Arg], 0).

stats(Name) :- statistics(runtime,[ _, Time]),
  trace('\nSorry I could not prove %t and I spent %t not doing it \n\n',
        [Name, Time], 0).
```

```
es
  ?- run.

est1 took 9 milliseconds and produced answer i(open, (1/2), 1, closed)


est2 took 68 milliseconds and produced answer i(closed, (1/2), 1, closed)


est3 took 38 milliseconds and produced answer i(closed, 0, 1, closed)


est4 took 22 milliseconds and produced answer i(closed, neginfinity, (-2/3), open


est5 took 104 milliseconds and produced answer i(closed, -3, 3, closed)


est6 took 57 milliseconds and produced answer i(closed, (1/2), 2, closed)


est7 took 59 milliseconds and produced answer i(open, neginfinity, - 0, open)


est8 took 123 milliseconds and produced answer i(open, neginfinity, infinity, ope


est9 took 36 milliseconds and produced answer i(open, neginfinity, - 0, open)


es
 ?- core     68096  (38912 lo-seg + 29184 hi-seg)
eap      33792 =  31227 in use +   2565 free
lobal     1175 =     16 in use +   1159 free
local     1024 =     16 in use +   1008 free
rail       511 =      0 in use +    511 free
    0.05 sec. for 2 GCs gaining 1213 words
    0.12 sec. for 20 local shifts and 21 trail shifts
    9.66 sec. runtime
```

```
yes
| ?- run.

test1 took 31 milliseconds and produced answer
X1
  where :
    X1 =  i(open, (1/2), 1, closed)



test2 took 145 milliseconds and produced answer
X1
  where :
    X1 =  i(closed, (1/2), 1, closed)



test3 took 115 milliseconds and produced answer
X1
  where :
    X1 =  i(closed, 0, abs(- 1), open)



test4 took 65 milliseconds and produced answer
X1
  where :
    X1 =  i(closed, neginfinity, (-2/3), open)



test5 took 255 milliseconds and produced answer
X1
  where :
    X1 =  i(closed, -3, 3, closed)



test6 took 175 milliseconds and produced answer
X1
  where :
    X1 =  i(closed, (1/2), 2, closed)



test7 took 211 milliseconds and produced answer
X1
  where :
    X1 =  i(open, neginfinity, - 0, open)



test8 took 383 milliseconds and produced answer
X1
  where :
    X1 =  i(open, neginfinity, infinity, open)
```

Interpretés

```
test9 took 102 milliseconds and produced answer
X1
  where :
    X1 =  i(open, neginfinity, - 0, open)



yes
| ?- core     65536  (36352 lo-seg + 29184 hi-seg)
heap      31232 =  29522 in use +   1710 free
global     1175 =     16 in use +   1159 free
local      1024 =     16 in use +   1008 free
trail       511 =      0 in use +    511 free
    0.03 sec. for 1 GCs gaining 103 words
    0.22 sec. for 25 local shifts and 26 trail shifts
    5.39 sec. runtime
```

DEPARTMENT OF ARTIFICIAL INTELLIGENCE

UNIVERSITY OF EDINBURGH

DAI Working Paper No:86
8 July 1981

Subject: A Generalized Interval Package and its Use for Semantic Checking
Author: Alan Bundy

Abstract

We describe an interval arithmetic package, INT, which generalises previous interval packages by using information about the monotonicity of functions. INT has been used in an algebraic manipulation package, PRESS, to check the conditions of rewrite rules and to vet the solutions to equations.

1. Introduction

In this paper we describe a general interval package; that is, a computer program, called INT, in which arithmetic functions are extended so that they apply, not just to numbers, but to intervals of the real line. If f is an n-ary function then it can be extended to intervals with the definition

$$f(i_1,...,i_n) = \{f(x_1,...,x_n): x_j \in i_j \text{ for all } j\}$$

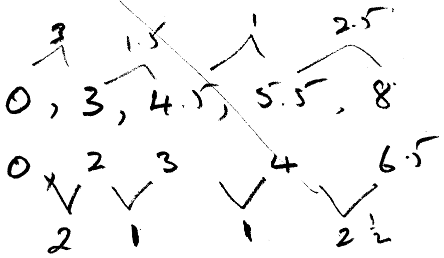
so that

$$[1,2] + [3,5] = [4,7]$$

where $[a,b] = \{x: a \le x \le b\}$ is the closed interval from a to b. Such packages are in common use for providing guaranteed error bounds in arithmetic (see e.g. [Good & London 70, Yohe 79]).

INT was built for a different purpose: namely, for checking the conditions of rewrite rules and vetting the solutions to equations, in an algebraic manipulation program, PRESS, [Bundy and Welham 81]. For instance, the rewrite rule

$$u*v \ge w \quad \Rightarrow \quad u \ge w/v$$



has the condition that v be positive, i.e lie in the open interval $(0,\infty)$. Other typical conditions are that a variable be non zero, an acute angle, etc. If a, m1, m2 and g are all known to be a positive quantities (e.g. because they are known to be an acceleration, mass, mass and acceleration due to gravity, respectively) then we want to be able to vet, and reject, the equation

a = -m1*g/(m1+m2)

as a possible solution for a in terms of m1, m2 and g, since it would imply that a were negative.

The INT package differs from previous packages in the following respects.

- INT can deal with intervals whose boundaries include the infinite numbers, $\pm\infty$.
- INT can deal with intervals with open or closed boundaries. These are denoted by round and square brackets, respectively, e.g. $[0,5) = \{x: 0 \le x < 5\}$
- INT can deal with any function, provided only that information is provided about where that function is monotonically increasing and decreasing. Previous packages have defined interval arithmetic for only a few specific functions, e.g. [Good & London 70] gives definitions only for +, -, * and /. Currently, the INT package can deal with +, -, *, /, exponentiation, logarithms, trigonometric and inverse trigonometric functions, and absolute value.
- INT can deal with intervals which straddle several monotonic regions of a function. In particular, it can deal with i/j when j contains 0.
- INT can use information about a particular constant, which specifies in what interval it lies. This has been used in conjunction with a program for solving Mechanics problems, [Bundy et al 79], to allow semantic information about physical quantities, e.g. that m is positive, ϕ is obtuse, etc, to influence the algebraic manipulation.

2. Monotonicity and Unary Functions

The key idea behind INT is to use the monotonicity of a function to decide which boundaries of its arguments to use to calculate its upper and lower bounds. For instance, sin is monotonically increasing on the interval (30,90], so the lower bound of sin (30,90] should be calculated from (30 and the upper bound from 90], giving the interval (sin 30, sin 90], which equals (1/2,1]. Note that the type of the boundary (open or closed) is inherited along with its value. We will say that sin is **mapped across** the interval (30,90] to produce the interval (sin 30, sin 90] and then the boundaries are **evaluated** to produce the interval (1/2,1].

Now consider the application of sin to (90,150]. sin is monotonically decreasing on this interval, so the lower bound of sin (90,150] should be calculated from 150] and the upper bound from (90, giving the interval [1/2 ,1). Note again that both the type and value of the interval are inherited. We can summarise this by saying that the interval (90,150] is **inverted to the pseudo-interval [150,90).*** sin is mapped across it, to produce [sin 150, sin 90), and the boundaries are evaluated to produce [1/2,1).

sin is simply monotonic on both the intervals (30,90] and (90,150]. To calculate the value of sin on an interval, on which it is not simply monotonic, is more complicated. Consider the interval (30,150]. To calculate sin (30,150] INT first divides the interval into sub-intervals, on each of which sin is simply monotonic. In this case (30,150] would be divided into (30,90] and (90,150]. sin is then applied separately to each sub-interval, to produce (1/2,1] and [1/2,1), and these results are then combined into the interval [1/2,1].

3. Generalized Monotonicity and Non-Unary Functions

To deal with non-unary functions we have to generalize the notion of monotonicity to a tuple valued function. This is because a non-unary function may have different monotonicity behaviour on different arguments. For instance, binary minus is monotonically increasing on its first argument and monotonically decreasing on its second. That is x-y increases as x increases, but decreases as y increases. We represent this by saying that - has monotonicity <up,down> on the region <[-oo,+oo], [-oo,oo]> (i.e. everywhere). Using the same notation, sin has monotonicity <up> on region <[-90,90]>, and monotonicity <down> on region <[90,270]>.

We can formalize this notation as follows:

> **Definition 1:** An **nD region** is an n-tuple of intervals. If every interval is [-oo,+oo] the the region is called the **whole nD space**.

> **Definition 2:** An n-ary function, f, is **monontonically increasing on its jth argument in region** $\langle i_1, \ldots, i_n \rangle$ iff

---

*[150,90) would be the empty interval by the normal definition. However, the phrase 'pseudo-interval' is meant to imply that we will not regard it as a proper interval, but merely as a syntactic device.

$x_j < y_j \rightarrow f(x_1,\ldots,x_j,\ldots,x_n) < f(x_1,\ldots,y_j,\ldots,x_n)$
for all $x_k \in i_k$ and $y_j \in i_j$

Definition 3: A function, f, is **monontonically decreasing on its jth argument in region $\langle i_1,\ldots,i_n \rangle$** iff

$x_j < y_j \rightarrow f(x_1,\ldots,x_j,\ldots,x_n) > f(x_1,\ldots,y_j,\ldots,x_n)$
for all $x_k \in i_k$ and $y_j \in i_j$

Definition 4: An n-ary function is **simply monotonic in region r** iff it is monotonically increasing or monotonically decreasing on each of its arguments in region r. Its **monotonicity** is given by an n-tuple in which the jth element is 'up' or 'down' according as the function is monotonically increasing or decreasing on the jth argument in r.

Armed with this notation, we can now tackle the problem of extending interval arithmetic to non-unary functions. Consider the application of - to the arguments <[3,5], [1,2)>. Since - is monotonically increasing on its first argument and monotonically decreasing on its second we should calculate the lower bound of [3,5] - [1,2) from [3 and 2) to yield (1. Similarly, the upper bound should be calculated from 5] and [1 to yield 4]. This can be summarised by saying that [1,2) is inverted to the pseudo-interval (2,1] and - is then mapped across the pseudo-region <[3,5],(2,1]> to produce (3-2, 5-1]. The rule is that the jth interval is inverted iff the function is monotonically decreasing on the jth argument. Note that the type of a boundary is open unless all of the boundaries it is calculated from are closed, since a function can only attain a boundary iff all its arguments do.

Both + and - are simply monotonic throughout the whole 2D space. * and /, however, are not simply monotonic throughout the space, they have four regions of simple monotonicity. For instance, / has the monotonicities given in figure 3-1.

<up,up> in
<[-oo,-0], [+0,+oo]>

<up,down> in
<[+0,+oo], [+0,+oo]>

<down,up> in
<[-oo,-0], [-oo,-0]>

<down,down> in
<[+0,+oo], [-oo,-0]>

Figure 3-1: The Monotonicities of /

Consider the application of / to <[2,3], [-1,1]>. INT breaks this down into the separate applications of / to <[2,3], [-1,-0)> and <[2,3], [+0,1]>, in which regions / is simply monotonic. Calculating these applications from the monotonicity information given above yields the following process.

To calculate / applied to <[2,3], [-1,-0)>
invert both arguments and map / across them to produce
[3/-0,2/-1]
Evaluate the boundaries to produce
[-oo,-2]

To apply / to <[2,3], [+0,1]>
invert the second argument and map / across the result to produce
[2/1,3/+0]
Evaluate the boundaries to produce
[2,+oo]

The two resulting intervals are then combined into a new interval whose upper bound is their maximum upper bound and whose lower bound is their minimum lower bound. This yields the interval [-oo,+oo]. Note that this interval is too permissive, in the sense that it includes the interval (-2,2), which should be excluded.

4. The Algorithm

After the informal introduction of the preceeding sections we now turn to a formal account of the general, interval-arithmetic algorithm, which we will call **Int-Apply**.

To Int-Apply a function, f, to a region, r, we must consider two cases:

(a) If f is simply monotonic on r then

> Form a pseudo-region r′ from r by replacing the jth element of r by its inverse iff f is monotonically decreasing on its jth argument.
>
> Map f across r′, evaluate the boundaries of the result and return it.

(b) If f is not simply monotonic on r then

> Split r into a sub-region on which f is simply monotonic and the 2^n-1 complementary sub-regions (some of which may be empty).
>
> Call the procedure recursively on each of the 2^n sub-regions which are not empty.
>
> Combine the resulting intervals into one interval.

The above description leaves various sub-procedures undefined, namely the processes of ‘mapping across’, ‘inverting’, ‘evaluating the boundaries’, splitting a region into sub-regions and combining several regions into one. We now proceed to define these processes.

- **Intervals.** Intervals are represented as quadruples, in which the first and last elements are either ‘open’ or ‘closed’, to represent the type of boundary, and the second and third elements are numbers representing the value of the boundaries, e.g. (90,150] is represented by <open,90,150,closed>.

- **Inverting.** The **inverse** of interval <L,B,T,R> is <R,T,B,L>.

- **Mapping Across.** The **mapping of f across**

  <<L1,B1,T1,R1>,...,<Ln,Bn,Tn,Rn>>

  is

  <c({L1,...,Ln}), f(B1,...,Bn), f(T1,...,Tn), c({R1,...,Rn})>

  where c(Set)=open iff open ∈ Set and c(Set)=closed otherwise.

- **Evaluation.** As can be seen from previous sections the normal arithmetic functions must be extended to deal with the infinite numbers -oo and +oo. It is also necessary to distinguish between -0 and +0, since 3/-0 = -oo and 3/+0 = +oo. For evaluating the boundaries of intervals INT uses an arbitrary-precision, rational-number, arithmetic package developed by Richard O'Keefe. In this package numbers are represented by triples of <sign,numerator,denominator>, e.g. <-,2,1>. +oo is represented by <+,1,0> and -0 by <-,0,1>, etc. The standard rational arithmetic operations require only trivial adaptation to return the correct answers for infinite numbers. In indeterminate cases, e.g. 0/0, the answer 'undefined' is returned. Such an answer causes Int-Apply to return the default interval (-oo,+oo).

- **Splitting.** To split region <I1,...,In> into appropriate sub-regions go through the simply monotonic regions of f until one, <M1,...,Mn>, is found with the property that each Ij is a disjoint union of intervals Ij' and Ij", where Ij" is non-empty, and a sub-interval of Mj. Return the 2^n sub-regions <I1*,...,In*>, where Ij* is either Ij' or Ij". Note that f is simply monotonic on sub-region <I1",...,In">. If Ij' is empty then so is any region containing it. If <I1,...,In> cannot be divided into a finite set of simply monotonic sub-regions then this splitting process may not terminate.

- **Combining.** To combine a set of intervals, form a new interval whose lower bound is the minimum of the lower bounds of the set and whose upper bound is the maximum of the set. Note that the minimum of two boundaries with the same value, but different types, is the closed boundary. Similarly with the maximum. The combination of two intervals is the smallest interval containing their union, but it may not be equal to their union, e.g. combining [-oo,-2] and [2,+oo] produces [-oo,+oo].

5. Semantic Checking

he above sections describe an interval arithmetic algorithm. We now explain the applications of this algorithm in the algebraic manipulation package, PRESS, [Bundy and Welham 81].

As describe in the introduction, the applications are twofold: checking the conditions of rewrite rules and vetting the solutions to equations. Both these applications make use of a common sub-procedure, **Find-Int**. Find-Int takes an algebraic term and returns the interval within which it lies. For instance, if a, m1, m2 and g are all positive then Find-Int applied to the term -m1*g/(m1+m2) will return (-oo,0), from which it can be deduced that the term is negative and that a = -m1*g/(m1+m2) is false.

The procedure, Find-Int, works by call by value. Applied to a term

$f(t_1,\ldots,t_n)$, Find-Int is called recursively on each t_j and returns i_j. f is then applied to $<i_1,\ldots,i_n>$ by Int-Apply. If Find-Int is applied to a number, n, then the interval [n,n] is returned, e.g. [2,2]. If Find-Int is applied to a symbolic constant or variable, e.g. m1, then semantic information is used to try to determine the result. If this is not successful then the default interval, (-oo,+oo), is returned.

The semantic information is provided by the MECHO program, [Bundy et al 79], for solving Mechanics problems. Information is provided about two sorts of constant: physical quantities (e.g. masses, accelerations, etc) and angles. All physical quantities are assumed to be positive, that is, to lie in the interval (0,+oo). Provision exists for providing more sophisticated information about each kind of quantity, but this has not been exploited. In the case of angles an attempt is made to infer in which quadrant(s) of the circle the angle lies, for instance, [0,90], [180,360], etc.

ngles are defined in MECHO either as the inclination or the normal to a 2 dimensional curve, where the normal is towards the convex side of the curve and the inclination is 90 degrees greater. For **simple curves**, i.e. monotonic curves with monotonic first derivatives, both the inclination and the normal lie wholly within a single quadrant. Which quadrant this is depends on the sign of the first and second derivatives. We call the first derivative the **slope** and the second derivative the **concavity** of the curve. For instance, if the slope is positive and the concavity is positive then the inclination lies in the quadrant [0,90] and the normal lies in the quadrant [270,360]. This and the other seven cases are illustrated in figure 5-1.

If Find-Int is applied to an angle defined on a non-simple curve then the curve is first broken into simple curves, the above process is applied to each of these and the resulting quadrants are combined, using the combination procedure described in section 4.

6. Results

The INT interval arithmetic package has been implemented in PROLOG ([Pereira et al 79] on a DEC10. It occupies 39k, 36 bit words and the PROLOG system occupies a further 29k. The former figure could probably be reduced substantially by deleting various utility procedures required by PRESS but not by INT. Table 6-1 summarises the results of applying Int-Apply to some typical functions and regions. Table 6-2 summarises the results of applying Find-Int to some typical formulae.

7. Limitations

The INT package uses information about the simply monotonic regions of a function to extend its definition on numbers to one on intervals. In this way

Concavity Slope	Positive	Negative	Zero
Positive	normal [270,360] incline [0,90]	normal [90,180] incline [180,270]	normal (270,360) incline (0,90)
Negative	normal [180,270] incline [270,360]	normal [0,90] incline [90,180]	normal (180,270) incline (270,360)
Zero	impossible	impossible	normal=270 incline=0
Infinity	impossible	impossible	normal=180 incline=270

Figure 5-1: Classification of the Angles of Simple Curves

Interval Calculation	**Answer**	**CPU Time in msecs**	**Is Region Simply Monotonic?**
sin (30,90]	(1/2,1]	13	yes
sin (30,150]	[1/2,1]	64	no
abs (-1,1]	[0,1]	38	no
(2,3] / [-3,-0]	[-oo,-2/3)	18	yes
[-3,-0) * [-1,1]	[-3,3]	120	no
$(1,2]^{[-1,1]}$	[1/2,2]	58	no

Table 6-1: Results of Applying Int-Apply

Formula	Interval it lies in	CPU Time in msecs
$-m1*g/(m1+m2)$	(-oo,-0)	63
$\sin(\theta)+2/\cos(\phi)$	(-oo,oo)	128
$\log_2 \sin(\theta)$	(-oo,-0)	37

where m1, m2 and g are positive numbers, θ is an acute angle and ϕ is an obtuse angle.

Table 6-2: Results of Applying Find-Int

it generalises previous interval packages, which could deal with only a -specified number of functions. However, INT only works well on functions which are well behaved monotonically, i.e. divide the whole space into a finite set of continuous simply monotonic regions. Some of the situations in which INT does not behave ideally are listed below.

- INT cannot deal with the situation where a function must be applied to a region which only divides into an infinite set of simply monotonic sub-regions, because Int-Apply will not terminate. For instance, sin applied to <(-oo,+oo)> should return [-1,1], but <(-oo,+oo)> cannot be divided into a finite set of simply monotonic sub-regions, so Int-Apply will not terminate. This particular case is dealt with by a patch to the INT package to take into account the boundedness of sin, cos, etc.

- Int-Apply is constrained to return a single continuous interval as its result. To return, say, a set of disjoint intervals would alter the whole basis of the algorithm with a consequent loss of simplicity and efficiency. The price to be paid for this is that the result of Int-Apply will sometimes be too inclusive. An example of this was given in section 3 where the result of [2,3]/[-1,1] included the interval (-2,2).* Sometimes the best description of the monotonicity

---

*G. Caplat, in a private communication, has suggested that allowing 'extervals' to be returned by Int-Apply, would account for the majority of counterexamples which arise in practice, for only a small increase in the complexity of the algorithm. An exterval is a set of real numbers obtained by subtracting an interval from the real line. I have not explored this possibility.

of a function involves non-continuous sets of numbers. For instance, x^y has monotonicity <down,up> when x is negative and y is an even rational. Unfortunately, the even rationals cannot be represented as an interval. In fact exponentiation has no simply monotonic sub-regions in <[-oo,0],[-oo,+oo]>. There is a patch in INT to deal with this particular case, but in general such situations are outside its scope.

3. Conclusion

We have seen that the INT interval arithmetic package generalises previous interval packages by the use of the monotonicity of functions. It can deal with more functions than previous packages and can deal correctly with the application of a function to a region which straddles a finite number of simply monotonic regions. It breaks down only when a function has a particularly complex monotonic behaviour.

The INT package has been applied in unusual ways, to check the conditions of rewrite rules and to vet the solutions to equations, as part of an algebraic manipulation package.

References

[Bundy and Welham 81]
Bundy, A. and Welham, B.
Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation.
Artificial Intelligence 16(2), 1981.

[Bundy et al 79]
Bundy, A., Byrd, L., Luger, G., Mellish, C., Milne, R. and Palmer, M.
Solving Mechanics Problems Using Meta-Level Inference.
In *Procs of the sixth.* IJCAI, Tokyo, 1979.
Also available from Edinburgh as DAI Research Paper No. 112.

[Good & London 70]
Good, D.I. and London, R.L.
Computer interval arithmetic: definition and proof of correct implementation.
JACM 17(4):603-612, 1970.

[Pereira et al 79]
Pereira, L.M., Pereira, F.C.N. and Warren, D.H.D.
User's guide to DECsystem-10 PROLOG.
Occasional Paper 15, Dept. of Artificial Intelligence, Edinburgh, 1979.

Yohe 79]

Yohe, J.M.
Software for interval arithmetic: A reasonably portable package.
ACM Trans. Math. Software 5(1):pp50-63, March, 1979.

Master

DEPARTMENT OF ARTIFICIAL INTELLIGENCE

UNIVERSITY OF EDINBURGH

DAI Working Paper No: 86
27 March 1981

Subject: A Generalized Interval Package and its Use for Semantic Checking
Author: Alan Bundy

Abstract

We describe an interval arithmetic package, INT, which generalises previous interval packages by using information about the monotonicity of functions. INT has been used in an algebraic manipulation package, PRESS, to check the conditions of rewrite rules and to vet the solutions to equations.

1. Introduction

In this paper we describe a general interval package; that is, a computer program, called INT, in which arithmetic functions are extended so that they apply, not just to numbers, but to intervals of the real line. If f is an n-ary function then it can be extended to intervals with the definition

$$f(i_1,...,i_n) = \{f(x_1,...,x_n): x_j \in i_j \text{ for all } j\}$$

so that

$$[1,2] + [3,5] = [4,7]$$

where $[a,b] = \{x: a \leq x \leq b\}$ is the closed interval from a to b. Such packages are in common use for providing guaranteed error bounds in arithmetic (see [Good & London 70]).

INT was built for a different purpose: namely, for checking the conditions of rewrite rules and vetting the solutions to equations, in an algebraic manipulation program, PRESS, [Bundy and Welham 81]. For instance, the rewrite rule

$$u*v \geq w \Rightarrow u \geq w/v$$

has the condition that v be positive, i.e lie in the open interval $(0,\infty)$. Other typical conditions are that a variable be non zero, an acute angle, etc. If a, m1, m2 and g are all known to be a positive quantities (e.g. because they are known to be an acceleration, mass, mass and acceleration due to gravity, respectively) then we want to be able to vet, and reject, the equation

$$a = -m1*g/(m1+m2)$$

as a possible solution for a in terms of m1, m2 and g, since it would imply that a were negative.

The INT package differs from previous packages in the following respects.

- INT can deal with intervals whose boundaries include the infinite numbers, $\pm\infty$.

- INT can deal with intervals with open or closed boundaries. These are denoted by round and square brackets, respectively, e.g.

$[0,5) = \{x: 0 \leq x < 5\}$

- INT can deal with any function, provided only that information is provided about where that function is monotonically increasing and decreasing. Previous packages have defined interval arithmetic for only a few specific functions, e.g. [Good & London 70] gives definitions only for +, -, * and /. Currently, the INT package can deal with +, -, *, /, exponentiation, logarithms, trigonometric and inverse trigonometric functions, and absolute value.

- INT can deal with intervals which straddle several monotonic regions of a function. In particular, it can deal with i/j when j contains 0.

- INT can use information about a particular constant, which specifies in what interval it lies. This has been used in conjunction with a program for solving Mechanics problems, [Bundy et al 79], to allow semantic information about physical quantities, e.g. that m is positive, ϕ is obtuse, etc, to influence the algebraic manipulation.

2. Monotonicity and Unary Functions

The key idea behind INT is to use the monotonicity of a function to decide which boundaries of its arguments to use to calculate its upper and lower bounds. For instance, sin is monotonically increasing on the interval (30,90], so the lower bound of sin (30,90] should be calculated from (30 and the upper bound from 90], giving the interval (sin 30, sin 90], which equals (1/2,1]. Note that the *type* of the boundary (open or closed) is inherited along with its value. We will say that sin is mapped across the interval (30,90] to produce the interval (sin 30, sin 90] and then the boundaries are evaluated to produce the interval (1/2,1].

Now consider the application of sin to (90,150]. sin is monotonically decreasing on this interval, so the lower bound of sin (90,150] should be calculated from 150] and the upper bound from (90, giving the interval [1/2,1). Note again that both the type and value of the interval is inherited. We can summarise this by saying that the interval (90,150] is inverted to the pseudo-interval [150,90).* sin is mapped across it, to produce [sin 150, sin 90), and the boundaries are evaluated to produce [1/2,1).

sin is simply monotonic on both the intervals (30,90] and (90,150]. To calculate the value of sin on an interval, on which it is not simply monotonic, is more complicated. Consider the interval (30,150]. To calculate sin (30,150], INT first divides the interval into sub-intervals, on each of which sin is simply monotonic. In this case (30,150] would be divided into (30,90] and (90,150]. sin is then applied separately to each sub-interval, to produce (1/2,1] and [1/2,1), and these results are then combined into the interval [1/2,1].

---

*[150,90) would be the empty interval by the normal definition. However, the phrase 'pseudo-interval' is meant to imply that we will not regard it as a proper interval, but merely as a syntactic device.

3. Generalized Monotonicity and Non-Unary Functions

To deal with non-unary functions we have to generalize the notion of monotonicity to a tuple valued function. This is because a non-unary function may have different monotonicity behaviour on different arguments. For instance, binary minus is monotonically increasing on its first argument and monotonically decreasing on its second. That is x-y increases as x increases, but decreases as y increases. We represent this by saying that - has monotonicity <up,down> on the region $\langle[-\infty,+\infty], [-\infty,\infty]\rangle$ (i.e. everywhere). Using the same notation, sin has monotonicity <up> on region <[-90,90]>, and monotonicity <down> on region <[90,270]>.

We can formalize this notation as follows:

> Definition 1: An nD region is an n-tuple of intervals. If every interval is $[-\infty,+\infty]$ the the region is called the whole nD space.
>
> Definition 2: An n-ary function, f, is monontonically increasing on its jth argument in region $\langle i_1,\ldots,i_n\rangle$ iff
>
> $$x_j < y_j \;\rightarrow\; f(x_1,\ldots,x_j,\ldots,x_n) < f(x_1,\ldots,y_j,\ldots,x_n)$$
> for all $x_k \in i_k$ and $y_j \in i_j$
>
> Definition 3: A function, f, is monontonically decreasing on its jth argument in region $\langle i_1,\ldots,i_n\rangle$ iff
>
> $$x_j < y_j \;\rightarrow\; f(x_1,\ldots,x_j,\ldots,x_n) > f(x_1,\ldots,y_j,\ldots,x_n)$$
> for all $x_k \in i_k$ and $y_j \in i_j$
>
> Definition 4: An n-ary function is simply monotonic in region r iff it is monotonically increasing or monotonically decreasing on each of its arguments in region r. Its monotonicity is given by an n-tuple in which the jth element is 'up' or 'down' according as the function is monotonically increasing or decreasing on the jth argument in r.

Armed with this notation, we can now tackle the problem of extending interval arithmetic to non-unary functions. Consider the application of - to the arguments <[3,5], [1,2)>. Since - is monotonically increasing on its first argument and monotonically decreasing on its second we should calculate the lower bound of [3,5] - [1,2) from [3 and 2) to yield (1. Similarly, the upper bound should be calculated from 5] and [1 to yield 4]. This can be summarised by saying that [1,2) is inverted to the pseudo-interval (2,1] and - is then mapped across the pseudo-region <[3,5],(2,1]> to produce (3-2, 5-1]. The rule is that the jth interval is inverted iff the function is monotonically decreasing on the jth argument. Note that the type of a boundary is open unless all of the boundaries it is calculated from are closed, since a function can only attain a boundary iff all its arguments do.

Both + and - are simply monotonic throughout the whole 2D space. * and /, however, are not simply monotonic throughout the space, they have four regions of simple monotonicity. For instance, / has the ~~following~~ monotonicities given in figure 3-1.

Consider the application of / to <[2,3], [-1,1]>. INT breaks this down into the separate applications of / to <[2,3], [-1,-0)> and <[2,3], [+0,1]>, in which regions / is simply monotonic. Calculating these applications from the monotonicity information given above yields the following process.

To calculate / applied to <[2,3], [-1,-0)>

<up,up> in $<[-\infty,-0], [+0,+\infty]>$	<up,down> in $<[+0,+\infty], [+0,+\infty]>$
<down,up> in $<[-\infty,-0], [-\infty,-0]>$	<down,down> in $<[+0,+\infty], [-\infty,-0]>$

Figure 3-1: The Monotonicities of /

invert both arguments and map / across them to produce
[3/-0,2/-1]
Evaluate the boundaries to produce
$[-\infty,-2]$

To apply / to <[2,3], [+0,1]>
invert the second argument and map / across the result to produce
[2/1,3/+0]
Evaluate the boundaries to produce
$[2,+\infty]$

The two resulting intervals are then combined into a new interval whose upper bound is their maximum upper bound and whose lower bound is their minimum lower bound. This yields the interval $[-\infty,+\infty]$. Note that this interval is too permissive, in the sense that it includes the interval (-2,2), which should be excluded.

4. The Algorithm

After the informal introduction of the preceeding sections we now turn to a formal account of the general, interval-arithmetic algorithm, which we will call Int-Apply.

To Int-Apply a function, f, to a region, r, we must consider two cases:

(a) If f is simply monotonic on r then

Form a pseudo-region r' from r by replacing the jth element of r by its inverse iff f is monotonically decreasing on its jth argument.

Map f across r', evaluate the boundaries of the result and return it.

(b) If f is not simply monotonic on r then

Split r into a sub-region on which f is simply monotonic and the 2^n-1 complementary sub-regions (some of which may be empty).

Call the procedure recursively on each of the 2^n sub-regions which are not empty.

Combine the resulting intervals into one interval.

The above description leaves various sub-procedures undefined, namely the processes of 'mapping across', 'inverting', 'evaluating the boundaries', splitting a region into sub-regions and combining several regions into one. We now proceed to define these processes.

- Intervals. Intervals are represented as quadruples, in which the first and last elements are either 'open' or 'closed', to represent the *type* of boundary, and the second and third elements are numbers representing the *value* of the boundaries, e.g. (90,150] is represented by <open,90,150,closed>.

- Inverting. The inverse of interval <L,B,T,R> is <R,T,B,L>.

- Mapping Across. The mapping of f across

  <<L1,B1,T1,R1>,...,<Ln,Bn,Tn,Rn>>

  is

  <c({L1,...,Ln}), f(B1,...,Bn), f(T1,...,Tn), c({R1,...,Rn})>

  where c(Set)=open iff open $\in$ Set and c(Set)=closed otherwise.

- Evaluation. As can be seen from previous sections the normal arithmetic functions must be extended to deal with the infinite numbers -oo and +oo. It is also necessary to distinguish between -0 and +0, since 3/-0 = -oo and 3/+0 = +oo. For evaluating the boundaries of intervals INT uses an arbitrary-precision, rational-number, arithmetic package developed by Richard O'Keefe. In this package numbers are represented by triples of <sign,numerator,denominator>, e.g. <-,2,1>. +oo is represented by <+,1,0> and -0 by <-,0,1>, etc. The standard rational arithmetic operations require only trivial adaptation to return the correct answers for infinite numbers. In indeterminate cases, e.g. 0/0, the answer 'undefined' is returned. Such an answer causes Int-Apply to return the default interval (-oo,+oo).

- Splitting. To split region <I1,...,In> into appropriate sub-regions go through the simply monotonic regions of f until one, <M1,...Mn>, is found with the property that each Ij is a disjoint union of intervals Ij' and Ij", where Ij" is non-empty, and a sub-interval of Mj. Return the 2^n sub-regions <I1*,...,In*>, where Ij* is either Ij' or Ij". Note that f is simply monotonic on sub-region <I1",...,In">. If Ij' is empty then so is any region containing it. If <I1,...,In> cannot be divided into a finite set of simply monotonic sub-regions then this splitting process may not terminate.

- Combining. To combine a set of intervals, form a new interval whose lower bound is the minimum of the lower bounds of the set and whose upper bound is the maximum of the set. Note that the minimum of two boundaries with the same value, but different types, is the closed boundary. Similarly with the maximum. The combination of two intervals is the smallest interval containing their union, but it may not be equal to their union, e.g. combining [-∞,-2] and [2,+∞] produces [-∞,+∞].

5. Semantic Checking

The above sections describe an interval arithmetic algorithm. We now explain the applications of this algorithm in the algebraic manipulation package, PRESS, [Bundy and Welham 81].

As describe in the introduction, the applications are twofold: checking the conditions of rewrite rules and vetting the solutions to equations. Both these applications make use of a common sub-procedure, Find-Int. Find-Int takes an algebraic term and returns the interval within which it lies. For instance, if a, m1, m2 and g are all positive then Find-Int applied to the term -m1*g/(m1+m2) will return (-∞,0), from which it can be deduced that the term is negative and that a = -m1*g/(m1+m2) is false.

The procedure, Find-Int, works by call by value. Applied to a term $f(t_1,...,t_n)$, Find-Int is called recursively on each t_j and returns i_j. f is then applied to $<i_1,...,i_n>$ by Int-Apply. If Find-Int is applied to a number, n, then the interval [n,n] is returned, e.g. [2,2]. If Find-Int is applied to a symbolic constant or variable, e.g. m1, then semantic information is used to try to determine the result. If this is not successful then the default interval, (-oo,+oo), is returned.

The semantic information is provided by the MECHO program, [Bundy et al 79], for solving Mechanics problems. Information is provided about two sorts of constant: physical quantities (e.g. masses, accelerations, etc) and angles. All physical quantities are assumed to be positive, that is, to lie in the interval (0,+oo). Provision exists for providing more sophisticated information about each kind of quantity, but this has not been exploited. In the case of angles an attempt is made to infer in which quadrant(s) of the circle the angle lies, for instance, [0,90], [180,360], etc.

Angles are defined in MECHO either as the inclination or the normal to a 2 dimensional curve, where the normal is towards the convex side of the curve and the inclination is 90 degrees greater. For simple curves, i.e. monotonic curves with monotonic first derivatives, both the inclination and the normal lie wholly within a single quadrant. Which quadrant this is depends on the sign of the first and second derivatives. We call the first derivative the slope and the second derivative the concavity of the curve. For instance, if the slope is positive and the concavity is positive then the inclination lies in the quadrant [0,90] and the normal lies in the quadrant [270,360]. This and the other seven cases are illustrated in figure 5-1.

If Find-Int is applied to an angle defined on a non-simple curve then the curve is first broken into simple curves, the above process is applied to each of these and the resulting quadrants are combined, using the combination procedure described in section 4.

✓ Results seen

6. Limitations

The INT package uses information about the simply monotonic regions of a function to extend its definition on numbers to one on intervals. In this way it generalises previous interval packages, which could deal with only a pre-specified number of functions. However, INT only works well on functions which are well behaved monotonically, i.e. divide the whole space into a finite set of continuous simply monotonic regions. Some of the situations in which INT does not behave ideally are listed below.

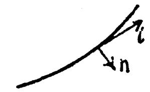
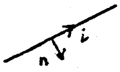
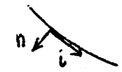
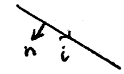
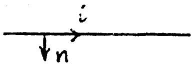
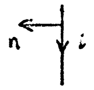
Concavity Slope	Positive	Negative	Zero
Positive	 normal∈[270,360] incline∈[0,90]	 normal∈[90,180] incline∈[180,270]	 normal∈(270,360) incline∈(0,90)
Negative	 normal∈[180,270] incline∈[270,360]	 normal∈[0,90] incline∈[90,180]	 normal∈(180,270) incline∈(270,360)
Zero	impossible	impossible	 normal=270 incline=0
Infinity	impossible	impossible	 normal=180 incline=270

Figure 5-1: Classification of the Angles of Simple Curves

- INT cannot deal with the situation where a function must be applied to a region which only divides into an infinite set of simply monotonic sub-regions, because Int-Apply will not terminate. For instance, sin applied to <(-∞,+∞)> should return [-1,1], but <(-∞,+∞)> cannot be divided into a finite set of simply monotonic sub-regions, so Int-Apply will not terminate. This particular case is dealt with by a patch to the INT package to take into account the boundedness of sin, cos, etc.

- Int-Apply is constrained to return a single continuous interval as its result. To return, say, a set of disjoint intervals would alter the whole basis of the algorithm with a consequent loss of simplicity and efficiency. The price to be paid for this is that the result of Int-Apply will sometimes be too inclusive. An example of this was given in section 3 where the result of [2,3]/[-1,1] included the interval (-2,2).

- Sometimes the best description of the monotonicity of a function involves non-continuous sets of numbers. For instance, x^y has monotonicity <down,up> when x is negative and y is an even rational. Unfortunately, the even rationals cannot be represented as an interval. In fact exponentiation has no simply monotonic sub-regions in <[-∞,0],[-∞,+∞]>. There is a patch in INT to deal with this particular case, but in general such situations are outside its scope.

7. Conclusion

We have seen that the INT interval arithmetic package generalises previous interval packages by the use of the monotonicity of functions. It can deal with more functions than previous packages and can deal correctly with the application of a function to a region which straddles a finite number of simply monotonic regions. It breaks down only when a function has a particularly complex monotonic behaviour.

The INT package has been applied in unusual ways, to check the conditions of rewrite rules and to vet the solutions to equations, as part of an algebraic manipulation package.

References

[Bundy and Welham 81]
Bundy, A. and Welham, B.
Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation.
Artificial Intelligence , in press 1981.

[Bundy et al 79]
Bundy, A., Byrd, L., Luger, G., Mellish, C., Milne, R. and Palmer, M.
Solving Mechanics Problems Using Meta-Level Inference.
In *Procs of the sixth*. IJCAI, Tokyo, 1979.
Also available from Edinburgh as DAI Research Paper No. 112.

[Good & London 70]
Good, D.I. and London, R.L.
Computer interval arithmetic: definition and proof of correct implementation.
JACM 17(4):603-612, 1970.

ALBERT-LUDWIGS-UNIVERSITÄT

INSTITUT FÜR ANGEWANDTE MATHEMATIK
Prof. Dr. K. Nickel

D-7800 FREIBURG I. BR., den May 5, 1981
Hermann-Herder-Strasse 10
Telefon (0761) 203-3062

RECEIVED
11 MAY 1981
DEPT. OF ARTIFICIAL INTELLIGENCE

Dr. Alan Bundy
University of Edinburgh
Department of Artificial Intelligence
Hope Park Square, Meadow Lane

Edinburgh EH8 9NW

Dear Doctor Bundy!

Thank you very much for your letter of April 20 and for your manuscript "A generalized interval package ..." I read it with great interest. We will include it in our "interval library" such that everybody interested in it can look at it. I enclose a list of the papers which are encluded in that library.

Some years ago I worked also in the field of the monotonicity of functions. A paper of mine is enclosed. I found your ideas very stimulating and I am sorry that I did not know them at that time.

I would very much like to see your results published. Probabely a Journal on Computer Science and/or on Artificial Intelligence would be the best. Perhaps you could also add some practical results? Unfortunately I do not know which Journal accepts such a paper. There is still much hesitation in accepting papers from Interval Mathematics.

Sincerely yours

Karl Nickel

Moore RE

Interval Analysis

Prentice Hall

1966.