```
> with(DirectSearch);
                              [GlobalSearch, Search]                                    (1)
> interface(verboseproc = 2);
                                        1                                               (2)
> op(Search);
proc(OBJ::{algebraic, procedure}, constr::(expects({set(relation), list(relation)})) :=        (3)
    NULL, {assume::(identical(positive, negative, nonpositive, nonnegative)) := NULL,
    checkexit::posint := 2, checksolution::nonnegint := 0, evaluationlimit::posint := 10000,
    initialpoint::{set(equation), Array(equation), Array(realcons), Vector(equation),
    Vector(realcons), list(equation), list(realcons)} := NULL, maximize::truefalse := false,
    searchpath::name := NULL, step::positive := 1.0, tolerances::{positive, [positive],
    [positive, positive]} := [0.000001000000000, 0.000001000000000], usewarning::
    truefalse := true, variables::(list(name)) := NULL})
    local n, i, i1, j, k, Nmax, Niter, Obj, f, s, ind, indconj, params, ineq, IneqPnames, IneqTrue,
    eq, EqPnames, EvalEq, eqtrue, UseHfloat, xtry, xtryOld, xmin, xmin0, xmin1, xmax, fmin0,
    fmin1, fmax, objmin0, objmin1, eqmin0, eqmin1, xOld, u, utry, bound, EvalFun,
    FeasiblePoint, DirectionalMove, feasiblestep, ftry, fOld, objtry, eqtry, fmin, objmin, eqmin,
    diff2, xq, xl, qroots, qmin, r, Nleft, Nright, xleft, xright, fleft, fright, step0, stepav, stepshift,
    stepleft, shorten, kbad, kgood, knonfeasiblestep, parabolic, linear, simple, complexmessage,
    goodstep, tol, exit, Nexit, leftcons, rightcons, fleftOld, frightOld, mincheck, kmincheck,
    Spath, Searchpath, isquadratic, fquadratic, R, opt;
    description "numeric optimization, local minimum (maximum) of a function searching";
    IneqTrue := proc(X::Vector)
        option hfloat;
        local n, i, pvalues, ineq1, p;
        n := nops(IneqPnames);
        if 0 < n then
            pvalues := Vector(n, datatype = anything);
            for i to n do
                p := evaln(IneqPnames[i]);
                if UseHfloat then
                    try
                        pvalues[i] := evalhf(p(seq(j, j = X)))
                    catch:
                        pvalues[i] := evalf(p(seq(j, j = X)))
                    end try
                else
                    pvalues[i] := evalf(p(seq(j, j = X)))
                end if
            end do;
            ineq1 := evalf(subs([seq(IneqPnames[i] = pvalues[i], i = 1 ..n)], ineq))
        else
```

```
            ineq1 := ineq
        end if;
        if 0 < nops(ineq1) then
            ineq1 := evalf(subs([seq(params[i] = X[i], i = 1 ..nops(params))], ineq1))
        else
            return true
        end if;
        ineq1 := `~`[is](ineq1);
        if 0 < nops(remove(has, ineq1, true)) then ineq1 := false else ineq1 := true end if;
        ineq1
    end proc;
    EvalEq := proc(X::Vector)
        option hfloat;
        local n, i, pvalues, eq1, p;
        n := nops(EqPnames);
        if 0 < n then
            pvalues := Vector(n, datatype = anything);
            for i to n do
                p := evaln(EqPnames[i]);
                if UseHfloat then
                    try
                        pvalues[i] := evalhf(p(seq(j, j = X)))
                    catch:
                        pvalues[i] := evalf(p(seq(j, j = X)))
                    end try
                else
                    pvalues[i] := evalf(p(seq(j, j = X)))
                end if
            end do;
            eq1 := evalf(subs([seq(EqPnames[i] = pvalues[i], i = 1 ..n)], eq))
        else
            eq1 := eq
        end if;
        if 0 < nops(eq1) then
            eq1 := evalf(subs([seq(params[i] = X[i], i = 1 ..nops(params))], eq1))
        else
            return HFloat(0.)
        end if;
        eq1 := evalf(add((lhs(eq1[i]) - rhs(eq1[i]))^2, i = 1 ..nops(eq1)));
        if is(Im(eq1) = 0) then eq1 := abs(eq1) end if;
        eq1
    end proc;
```

```
diff2 := proc(x0, x1, x2, f0, f1, f2)
    option hfloat;
    if x2 = x1 or x1 = x0 then return HFloat(0.) end if;
    evalf( ( f2 − f1 ) / (x2 − x1) − ( f1 − f0) / (x1 − x0) )
end proc;
qroots := proc(x0, x1, x2, f0, f1, f2)
    option hfloat;
    local a, b, c, r;
    if f0 <= HFloat(0.) or f1 <= HFloat(0.) or f2 <= HFloat(0.) then return FAIL
    end if;
    a := −abs(evalf(x0 * ( f1 − f2) + x1 * ( f2 − f0) + x2 * ( f0 − f1)));
    b := evalf(x0^2 * ( f2 − f1) + x1^2 * ( f0 − f2) + x2^2 * ( f1 − f0));
    c := evalf(x0^2 * ( f1 * x2 − f2 * x1) + x1^2 * ( f2 * x0 − f0 * x2) + x2^2 * (f0
    * x1 − f1 * x0));
    if a = HFloat(0.) and b <> HFloat(0.) then
        return Vector(2, [evalf( −c / b), evalf( −c / b)], datatype = float)
    end if;
    if a <> HFloat(0.) then
        return Vector(2, [evalf(1 / 2 * ( −b + sqrt(abs(b^2 − 4 * a * c))) / a), evalf(1
        / 2 * ( −b − sqrt(abs(b^2 − 4 * a * c))) / a)], datatype = float)
    end if;
    FAIL
end proc;
xl := proc(x0, x1, x2, x3, f0, f1, f2, f3)
    option hfloat;
    local a1, b1, a2, b2;
    if x0 = x1 or x2 = x3 then return FAIL end if;
    a1 := evalf( ( f0 − f1) / (x0 − x1));
    a2 := evalf( ( f2 − f3) / (x2 − x3));
    if a1 = a2 then return FAIL end if;
    b1 := evalf( ( f1 * x0 − x1 * f0) / (x0 − x1));
    b2 := evalf((x2 * f3 − x3 * f2) / (x2 − x3));
    evalf((b2 − b1) / (a1 − a2))
end proc;
xq := proc(x0, x1, x2, f0, f1, f2)
    option hfloat;
    local d;
    d := evalf(HFloat(2.) * ( f2 * (x0 − x1) + f1 * (x2 − x0) + f0 * (x1 − x2)));
    if d <> HFloat(0.) then
        d := evalf((x0^2 * ( f2 − f1) + x1^2 * ( f0 − f2) + x2^2 * ( f1 − f0)) / d)
    else
```

```
        return FAIL
    end if;
    d
end proc;
qmin := proc(X0::Vector, X1::Vector, X2::Vector, f0, f1, f2)
    option hfloat;
    local x, f, x0, x1, x2, d01, d02, d12, dmax, d, t, u;
    if not (type(f0, realcons) and type(f1, realcons) and type(f2, realcons)) then
        fquadratic := infinity; return FAIL
    end if;
    d01 := LinearAlgebra:-VectorNorm(X1 - X0, 2, conjugate = false);
    d02 := LinearAlgebra:-VectorNorm(X2 - X0, 2, conjugate = false);
    d12 := LinearAlgebra:-VectorNorm(X2 - X1, 2, conjugate = false);
    dmax := max(d01, d02, d12);
    x0 := HFloat(0.);
    if evalf(min(d01, d02, d12)) <= HFloat(0.) then
        fquadratic := infinity; return FAIL
    end if;
    if dmax = d12 then
        x := [X1, X0, X2]; f := [f1, f0, f2]; x1 := d01; x2 := d12
    elif dmax = d01 then
        x := [X0, X2, X1]; f := [f0, f2, f1]; x1 := d02; x2 := d01
    else
        x := [X0, X1, X2]; f := [f0, f1, f2]; x1 := d01; x2 := d02
    end if;
    if evalf((f[3] - f[2])/(x2 - x1) - (f[2] - f[1])/(x1 - x0)) <= 0 then
        fquadratic := infinity; return FAIL
    end if;
    d := evalf(HFloat(2.) * (f[3]* (x0 - x1) + f[2]* (x2 - x0) + f[1]* (x1 - x2)));
    if d <> HFloat(0.) then
        d := evalf((x0^2 * (f[3] - f[2]) + x1^2 * (f[1] - f[3]) + x2^2 * (f[2] - f[1
        ]))/d)
    else
        fquadratic := infinity; return FAIL
    end if;
    u := LinearAlgebra:-Normalize(x[2] - x[1], 2, conjugate = false);
    if checkexit = 1 then
        x0 := CurveFitting:-PolynomialInterpolation([x0, x1, x2], f, t);
        t := d;
        fquadratic := evalf(eval(x0))
    end if;
```

```
        evalf(x[1] + d*u)
    end proc;
    EvalFun := proc(X::Vector)
        option hfloat;
        Nmax := Nmax + 1;
        if UseHfloat then
            try
                objtry := evalhf(Obj(seq(j, j = X)))
            catch: objtry := evalf(Obj(seq(j, j = X))) end try
        else
            objtry := evalf(Obj(seq(j, j = X)))
        end if;
        if maximize then objtry := -objtry end if;
        if eqtrue then
            eqtry := EvalEq(X); ftry := objtry + r*eqtry
        else
            ftry := objtry; eqtry := HFloat(0.)
        end if;
        if Searchpath and is(Im(ftry) = 0) then
            Spath := ArrayTools:-Concatenate(2, Spath, X)
        end if;
        ftry
    end proc;
    FeasiblePoint := proc({reduction::truefalse := true})
        option hfloat;
        local i, steptry, shortsteptry, realx;
        bound := false;
        if evaluationlimit <= Nmax then return false end if;
        if IneqTrue(xtry) then
            realx := is(Im(EvalFun(xtry)) = 0);
            if realx then return true end if;
            if complexmessage = false and usewarning then
                complexmessage := true;
                WARNING("complex value encountered; trying to find a feasible point")
            end if;
            if evaluationlimit <= Nmax then return false end if
        end if;
        bound := true;
        if reduction = false then return false end if;
        if n = 1 then
            for i to 50 do
```

```
    if i <= 6 then
        shortsteptry := HFloat(1.10000000000000008)
    elif i <= 8 then
        shortsteptry := HFloat(1.19999999999999994)
    elif i <= 10 then
        shortsteptry := HFloat(1.50000000000000000)
    elif i <= 16 then
        shortsteptry := HFloat(2.)
    elif i <= 20 then
        shortsteptry := HFloat(5.)
    elif i <= 40 then
        shortsteptry := HFloat(10.)
    elif i <= 60 then
        shortsteptry := HFloat(100.)
    else
        shortsteptry := HFloat(1000.)
    end if;
    steptry := (xtry[1] − xmin[1]) / shortsteptry;
    xtryOld[1] := xtry[1];
    xtry[1] := xmin[1] + steptry;
    if abs(xtry[1] − xmin[1]) <= HFloat(0.) then return false end if;
    if IneqTrue(xtry) then
        realx := is(Im(EvalFun(xtry)) = 0);
        if realx then return true end if;
        if complexmessage = false and usewarning then
            complexmessage := true;
            WARNING("complex value encountered; trying to find a feasible
            point")
        end if
    end if;
    if evaluationlimit <= Nmax then return false end if
end do;
if tol[1] < abs(steptry) then
    xtryOld[1] := xtry[1];
    steptry := sign(steptry) * tol[1] * HFloat(0.900000000000000022);
    xtry[1] := xmin[1] + steptry;
    if abs(xtry[1] − xmin[1]) <= HFloat(0.) then return false end if;
    if IneqTrue(xtry) then
        realx := is(Im(EvalFun(xtry)) = 0);
        if realx then return true end if;
        if complexmessage = false and usewarning then
```

```
                complexmessage := true;
                WARNING("complex value encountered; trying to find a feasible
                point")
            end if
        end if
    end if;
    return false
else
    if evaluationlimit <= Nmax then return false end if;
    for i to 50 do
        if i <= 6 then
            shortsteptry := HFloat(1.10000000000000008)
        elif i <= 8 then
            shortsteptry := HFloat(1.19999999999999994)
        elif i <= 10 then
            shortsteptry := HFloat(1.50000000000000000)
        elif i <= 15 then
            shortsteptry := HFloat(2.)
        elif i <= 20 then
            shortsteptry := HFloat(5.)
        elif i <= 10 then
            shortsteptry := HFloat(10.)
        elif i <= 20 then
            shortsteptry := HFloat(100.)
        else
            shortsteptry := HFloat(1000.)
        end if;
        steptry := LinearAlgebra:-VectorNorm(xtry − xmin1, 2, conjugate = false)
        / shortsteptry;
        xtryOld := LinearAlgebra:-Copy(xtry);
        xtry := xmin1 + steptry * utry;
        if LinearAlgebra:-VectorNorm(xtry − xmin1, 2, conjugate = false)
        <= HFloat(0.) then
            return false
        end if;
        if IneqTrue(xtry) then
            realx := is(Im(EvalFun(xtry)) = 0);
            if realx then return true end if;
            if complexmessage = false and usewarning then
                complexmessage := true;
                WARNING("complex value encountered; trying to find a feasible
```

```
                    point")
                end if
            end if;
            if evaluationlimit <= Nmax then return false end if
        end do;
        if tol[1] < steptry then
            xtryOld := LinearAlgebra:-Copy(xtry);
            steptry := tol[1]*HFloat(0.900000000000000022);
            xtry := xmin1 + steptry*utry;
            if LinearAlgebra:-VectorNorm(xtry - xmin1, 2, conjugate = false)
             <= HFloat(0.) then
                return false
            end if;
            if IneqTrue(xtry) then
                realx := is(Im(EvalFun(xtry)) = 0);
                if realx then return true end if;
                if complexmessage = false and usewarning then
                    complexmessage := true;
                    WARNING("complex value encountered; trying to find a feasible
                    point")
                end if
            end if
        end if;
        return false
    end if
end proc;
DirectionalMove := proc(STEP)
    option hfloat;
    local shorten, goodstep, step1, s, ind0, xmin0, fmin0;
    ind0 := ind;
    step1 := STEP;
    goodstep := true;
    bound := false;
    isquadratic := false;
    xmin0, fmin0 := LinearAlgebra:-Copy(xtry), ftry;
    xmin1, fmin1, objmin1 := LinearAlgebra:-Copy(xtry), ftry, objtry;
    while goodstep and not bound and Nmax < evaluationlimit do
        xtry := xmin1 + step1*utry;
        if Nleft <= 5 then
            shorten := HFloat(2.)
        elif Nleft <= 10 then
            shorten := HFloat(5.)
```

```
elif Nleft <= 20 then
    shorten := HFloat(10.)
elif Nleft <= 30 then
    shorten := HFloat(100.)
else
    shorten := HFloat(1000.)
end if;
s := 0;
if Nleft = 2 and ind0 = 0 then
    ind0 := 1;
    s := LinearAlgebra:-Copy(xleft[ ( ) ..( ), 2 ]), LinearAlgebra:-Copy(xleft[ ( )
    ..( ), 1 ]), LinearAlgebra:-Copy(xmin1), fleft[2], fleft[1], fmin1;
    s := qmin(s);
    if s <> FAIL then xtry := LinearAlgebra:-Copy(s) end if
end if;
if s <> FAIL and FeasiblePoint( ) then
    if ftry < fmin1 then
        goodstep := true;
        if 0 < LinearAlgebra:-VectorNorm(xtry − xmin1, 2, conjugate = false)
        then
            if ind0 <> 1 or ind0 = 1 and LinearAlgebra:-VectorNorm(xleft[ ( )
            ..( ), 1 ] − xmin1, 2, conjugate = false) < LinearAlgebra:-
            VectorNorm(xleft[ ( ) ..( ), 1 ] − xtry, 2, conjugate = false) then
                Nleft := Nleft + 1;
                if 1 < Nleft then
                    xleft[ ( ) ..( ), 2 ] :=
                    LinearAlgebra:-Copy(xleft[ ( ) ..( ), 1 ]);
                    fleft[2] := fleft[1]
                end if;
                xleft[ ( ) ..( ), 1 ] := LinearAlgebra:-Copy(xmin1);
                fleft[1] := fmin1
            elif ind0 = 1 and LinearAlgebra:-VectorNorm(xleft[ ( ) ..( ), 1
            ] − xtry, 2, conjugate = false) < LinearAlgebra:-VectorNorm(xleft[ ( )
            ..( ), 1 ] − xmin1, 2, conjugate = false) then
                Nright := Nright + 1;
                goodstep := false;
                if 1 < Nright then
                    xright[ ( ) ..( ), 2 ] :=
                    LinearAlgebra:-Copy(xright[ ( ) ..( ), 1 ]);
                    fright[2] := fright[1]
                end if;
                xright[ ( ) ..( ), 1 ] := LinearAlgebra:-Copy(xmin1);
```

```
                    fright[1] := fmin1
                end if
            end if;
            xmin1, fmin1, objmin1 := LinearAlgebra:-Copy(xtry), ftry, objtry;
            if bound then return true else step1 := shorten * step1 end if
        else
            bound := false;
            if ind0 <> 1 then goodstep := false end if;
            if 0 < LinearAlgebra:-VectorNorm(xtry - xmin1, 2, conjugate = false)
            then
                if ind0 <> 1 or ind0 = 1 and LinearAlgebra:-VectorNorm(xleft[ ( )
                ..( ), 1] - xmin1, 2, conjugate = false) < LinearAlgebra:-
                VectorNorm(xleft[ ( ) ..( ), 1] - xtry, 2, conjugate = false) then
                    Nright := Nright + 1;
                    goodstep := false;
                    if 1 < Nright then
                        xright[ ( ) ..( ), 2] :=
                        LinearAlgebra:-Copy(xright[ ( ) ..( ), 1]);
                        fright[2] := fright[1]
                    end if;
                    xright[ ( ) ..( ), 1] := LinearAlgebra:-Copy(xtry);
                    fright[1] := ftry
                elif ind0 = 1 and LinearAlgebra:-VectorNorm(xleft[ ( ) ..( ), 1
                ] - xtry, 2, conjugate = false) < LinearAlgebra:-VectorNorm(xleft[ ( )
                ..( ), 1] - xmin1, 2, conjugate = false) then
                    Nleft := Nleft + 1;
                    if 1 < Nleft then
                        xleft[ ( ) ..( ), 2] :=
                        LinearAlgebra:-Copy(xleft[ ( ) ..( ), 1]);
                        fleft[2] := fleft[1]
                    end if;
                    xleft[ ( ) ..( ), 1] := LinearAlgebra:-Copy(xtry);
                    fleft[1] := ftry
                end if
            end if
        end if
    else
        goodstep := false
    end if;
    if s = FAIL then goodstep := true end if;
    if ind0 = 1 then ind0 := 2 end if;
    if is(fmin1 = -infinity) then exit := true; break end if
```

```
end do;
if 0 < Nleft and 0 < Nright then goodstep := false else goodstep := true end if;
step1 := STEP;
bound := false;
if ind <> 0 then ind0 := 2 end if;
while goodstep and not bound and Nmax < evaluationlimit do
    xtry := xmin1 - step1 * utry;
    if Nright <= 5 then
        shorten := HFloat(2.)
    elif Nright <= 10 then
        shorten := HFloat(5.)
    elif Nright <= 20 then
        shorten := HFloat(10.)
    elif Nright <= 30 then
        shorten := HFloat(100.)
    else
        shorten := HFloat(1000.)
    end if;
    s := 0;
    if Nright = 2 and ind0 = 0 then
        ind0 := 1;
        s := LinearAlgebra:-Copy(xright[ ( ) ..( ), 2 ]), LinearAlgebra:-Copy(xright[ ( )
        ..( ), 1 ]), LinearAlgebra:-Copy(xmin1), fright[2], fright[1], fmin1;
        s := qmin(s);
        if s <> FAIL then xtry := LinearAlgebra:-Copy(s) end if
    end if;
    if s <> FAIL and FeasiblePoint( ) then
        if ftry < fmin1 then
            goodstep := true;
            if 0 < LinearAlgebra:-VectorNorm(xtry - xmin1, 2, conjugate = false)
            then
                if ind0 <> 1 or ind0 = 1 and LinearAlgebra:-VectorNorm(xright[ ( )
                ..( ), 1 ] - xmin1, 2, conjugate = false) < LinearAlgebra:-
                VectorNorm(xright[ ( ) ..( ), 1 ] - xtry, 2, conjugate = false) then
                    Nright := Nright + 1;
                    if 1 < Nright then
                        xright[ ( ) ..( ), 2 ] :=
                        LinearAlgebra:-Copy(xright[ ( ) ..( ), 1 ]);
                        fright[2] := fright[1]
                    end if;
                    xright[ ( ) ..( ), 1 ] := LinearAlgebra:-Copy(xmin1);
```

```
                fright[1] := fmin1
            elif ind0 = 1 and LinearAlgebra:-VectorNorm(xright[ ( ) .. ( ), 1
            ] - xtry, 2, conjugate = false) < LinearAlgebra:-VectorNorm(xright
            [ ( ) .. ( ), 1] - xmin1, 2, conjugate = false) then
                Nleft := Nleft + 1;
                goodstep := false;
                if 1 < Nleft then
                    xleft[ ( ) .. ( ), 2] :=
                    LinearAlgebra:-Copy(xleft[ ( ) .. ( ), 1]);
                    fleft[2] := fleft[1]
                end if;
                xleft[ ( ) .. ( ), 1] := LinearAlgebra:-Copy(xmin1);
                fleft[1] := fmin1
            end if
        end if;
        xmin1, fmin1, objmin1 := LinearAlgebra:-Copy(xtry), ftry, objtry;
        if bound then return true else step1 := shorten * step1 end if
    else
        bound := false;
        if ind0 <> 1 then goodstep := false end if;
        if 0 < LinearAlgebra:-VectorNorm(xtry - xmin1, 2, conjugate = false)
        then
            if ind0 <> 1 or ind0 = 1 and LinearAlgebra:-VectorNorm(xright[ ( )
            .. ( ), 1] - xmin1, 2, conjugate = false) < LinearAlgebra:-
            VectorNorm(xright[ ( ) .. ( ), 1] - xtry, 2, conjugate = false) then
                Nleft := Nleft + 1;
                goodstep := false;
                if 1 < Nleft then
                    xleft[ ( ) .. ( ), 2] :=
                    LinearAlgebra:-Copy(xleft[ ( ) .. ( ), 1]);
                    fleft[2] := fleft[1]
                end if;
                xleft[ ( ) .. ( ), 1] := LinearAlgebra:-Copy(xtry);
                fleft[1] := ftry
            elif ind0 = 1 and LinearAlgebra:-VectorNorm(xright[ ( ) .. ( ), 1
            ] - xtry, 2, conjugate = false) < LinearAlgebra:-VectorNorm(xright
            [ ( ) .. ( ), 1] - xmin1, 2, conjugate = false) then
                Nright := Nright + 1;
                if 1 < Nright then
                    xright[ ( ) .. ( ), 2] :=
                    LinearAlgebra:-Copy(xright[ ( ) .. ( ), 1]);
                    fright[2] := fright[1]
```

```
                    end if;
                    xright[ ( ) ..( ), 1 ] := LinearAlgebra:-Copy(xtry);
                    fright[1 ] := ftry
                end if
            end if
        end if
    else
        goodstep := false
    end if;
    if s = FAIL then goodstep := true end if;
    if ind0 = 1 then ind0 := 2 end if;
    if is( fmin1 = − infinity) then exit := true; break end if
end do;
if 0 < Nleft and (LinearAlgebra:-VectorNorm(xmin1 − xleft[ ( ) ..( ), 1], 2, conjugate
 = false) <= HFloat(0.) or fleft[1 ] < fmin1) then
    Nleft := 0
end if;
if 0 < Nright and (LinearAlgebra:-VectorNorm(xmin1 − xright[ ( ) ..( ), 1], 2,
conjugate = false) <= HFloat(0.) or fright[1 ] < fmin1) then
    Nright := 0
end if;
if Nleft = 0 and Nright = 0 then return false end if;
if 0 < Nleft and 0 < Nright then
    s := LinearAlgebra:-Copy(xleft[ ( ) ..( ), 1 ]), LinearAlgebra:-Copy(xmin1),
    LinearAlgebra:-Copy(xright[ ( ) ..( ), 1 ]), fleft[1 ], fmin1, fright[1 ];
    s := qmin(s);
    if s <> FAIL then xtry := LinearAlgebra:-Copy(s) end if;
    if s <> FAIL and FeasiblePoint( ) then
        if checkexit = 1 and abs( ftry − fquadratic) <= tol[2 ] and LinearAlgebra:-
        VectorNorm(xmin0 − xtry, 2, conjugate = false) <= tol[1 ] then
            isquadratic := true
        end if;
        if ftry < fmin1 then
            if 0 < LinearAlgebra:-VectorNorm(xtry − xmin1, 2, conjugate = false)
            then
                if LinearAlgebra:-VectorNorm(xleft[ ( ) ..( ), 1 ] − xmin1, 2, conjugate
                 = false) < LinearAlgebra:-VectorNorm(xleft[ ( ) ..( ), 1 ] − xtry, 2,
                conjugate = false) then
                    Nleft := Nleft + 1;
                    if 1 < Nleft then
                        xleft[ ( ) ..( ), 2 ] :=
```

```
                    LinearAlgebra:-Copy(xleft[ ( ) ..( ), 1 ]);
                    fleft[2] := fleft[1]
                end if;
                xleft[ ( ) ..( ), 1 ] := LinearAlgebra:-Copy(xmin1);
                fleft[1] := fmin1
            elif LinearAlgebra:-VectorNorm(xleft[ ( ) ..( ), 1 ] - xtry, 2, conjugate
             = false) < LinearAlgebra:-VectorNorm(xleft[ ( ) ..( ), 1 ] - xmin1, 2,
            conjugate = false) then
                Nright := Nright + 1;
                if 1 < Nright then
                    xright[ ( ) ..( ), 2 ] :=
                    LinearAlgebra:-Copy(xright[ ( ) ..( ), 1 ]);
                    fright[2] := fright[1]
                end if;
                xright[ ( ) ..( ), 1 ] := LinearAlgebra:-Copy(xmin1);
                fright[1] := fmin1
            end if
        end if;
        xmin1, fmin1, objmin1 := LinearAlgebra:-Copy(xtry), ftry, objtry
    else
        if 0 < LinearAlgebra:-VectorNorm(xtry - xmin1, 2, conjugate = false)
        then
            if LinearAlgebra:-VectorNorm(xleft[ ( ) ..( ), 1 ] - xmin1, 2, conjugate
             = false) < LinearAlgebra:-VectorNorm(xleft[ ( ) ..( ), 1 ] - xtry, 2,
            conjugate = false) then
                Nright := Nright + 1;
                goodstep := false;
                if 1 < Nright then
                    xright[ ( ) ..( ), 2 ] :=
                    LinearAlgebra:-Copy(xright[ ( ) ..( ), 1 ]);
                    fright[2] := fright[1]
                end if;
                xright[ ( ) ..( ), 1 ] := LinearAlgebra:-Copy(xtry);
                fright[1] := ftry
            elif LinearAlgebra:-VectorNorm(xleft[ ( ) ..( ), 1 ] - xtry, 2, conjugate
             = false) < LinearAlgebra:-VectorNorm(xleft[ ( ) ..( ), 1 ] - xmin1, 2,
            conjugate = false) then
                Nleft := Nleft + 1;
                if 1 < Nleft then
                    xleft[ ( ) ..( ), 2 ] :=
                    LinearAlgebra:-Copy(xleft[ ( ) ..( ), 1 ]);
                    fleft[2] := fleft[1]
```

```
                    end if;
                    xleft[ ( ) ..( ), 1 ] := LinearAlgebra:-Copy(xtry);
                    fleft[1] := ftry
                end if
            end if
        end if
    end if
  end if;
  true
end proc;
if UseHardwareFloats = true or UseHardwareFloats = deduced and Digits
 <= evalhf (Digits) then
  UseHfloat := true; Digits := round(evalhf (Digits))
else
  UseHfloat := false
end if;
R := Statistics:-RandomVariable(Laplace(0., 1.0));
Nmax := 0;
eqmin := 0;
step0 := step;
Nleft := 0;
Nright := 0;
fquadratic := infinity;
bound := false;
complexmessage := false;
Searchpath := false;
isquadratic := infinity;
tol := tolerances;
if type(tol, positive) then tol := [tol] end if;
if nops(tol) = 2 then
  tol := Statistics:-EvaluateToFloat(convert(tol, 'Vector'))
else
  tol := Statistics:-EvaluateToFloat(`<,>`(tol[1], tol[1]))
end if;
s := evalf ( − log10(min(tol[1], tol[2])));
if Digits < s then
  s := ceil(s);
  if usewarning then
    WARNING("tolerance %1 is too small for Digits %2; increasing Digits to %3. Note:
    software floats are slower than hardware floats", min(tol[1], tol[2]), Digits, s)
  end if;
  Digits := s
```

```
end if;
Obj := eval(OBJ);
if variables <> NULL then params := variables end if;
if not type(Obj, 'procedure') then
    Obj := evalf(Obj);
    if variables = NULL then
        params := [op(indets(Obj, name))];
        if has(Obj, 'Int') or has(Obj, 'int') then
            s := [op(indets(Obj, 'equation'))];
            if 0 < nops(s) then
                s := [seq(lhs(s[i]), i = 1 ..nops(s))];
                s := select(type, s, 'name');
                if 0 < nops(s) then params := remove(has, params, {op(s)}) end if
            end if
        end if
    end if;
    Obj := unapply(Obj, op(params))
else
    if variables = NULL then params := [op(1, eval(Obj))] end if
end if;
n := nops(params);
if n = 1 then r := max(10.^8, 0.3/tol[2]) else r := max(10.^5, 0.1/tol[2]) end if;
if n = 0 then error "number of problem variables are equal to 0" end if;
if constr <> NULL then
    ineq := [op(evalf(select(type, constr, {'`<`', '`<=`', '`<>`'})))];
    eq := [op(evalf(select(type, constr, '`=`')))];
    IneqPnames := [op(indets(ineq, 'procedure'))];
    EqPnames := [op(indets(eq, 'procedure'))];
    for i in [op(IneqPnames), op(EqPnames)] do
        if nops([op(1, eval(i))]) <> n then
            error
            "number of parameters in procedure %1 must coincides with number of
            variables %2", uneval(op(1, evaln(op(i)))), n
        end if
    end do
else
    ineq := []; IneqPnames := []; eq := []; EqPnames := []
end if;
if 0 < nops(eq) then eqtrue := true else eqtrue := false end if;
if variables = NULL and not type(OBJ, 'procedure') and 1 < n then
    if 0 < nops(IneqPnames) + nops(EqPnames) then
```

```
            error "please specify option variables with list of problem variables names"
        end if
    end if;
    if assume <> NULL then
        if assume = 'positive' then
            ineq := [seq(0. < i, i = params), op(ineq)]
        elif assume = 'negative' then
            ineq := [seq(i < 0., i = params), op(ineq)]
        elif assume = 'nonpositive' then
            ineq := [seq(i <= 0., i = params), op(ineq)]
        elif assume = 'nonnegative' then
            ineq := [seq(0. <= i, i = params), op(ineq)]
        end if
    end if;
    if initialpoint = NULL then
        xtry := Vector[column](n, fill = 0.9, datatype = float)
    else
        if type(initialpoint[1], 'equation') then
            xtry := convert(subs(convert(initialpoint, list), params), 'Vector[column]')
        else
            if variables = NULL and not type(OBJ, 'procedure') and 1 < n then
                if type(initialpoint, {'Array(realcons)', 'Vector(realcons)', 'list(realcons)'})
                then
                    error
                    "please specify option variables with list of problem variable names or
                    provide initial point as a list or set of equations varname=value"
                end if
            end if;
            xtry := convert(initialpoint, 'Vector[column]')
        end if;
        if UseHfloat then xtry := Statistics:-EvaluateToFloat(xtry) else xtry := evalf(xtry)
        end if;
        if Statistics:-Count(xtry) <> n then
            error "number of initial point values must coincides with number of variables", n
        end if;
        xtry := convert(xtry, 'Vector[column]')
    end if;
    if IneqTrue(xtry) = false then
        xmin := [seq(params[i] = xtry[i], i = 1 ..n)];
        if usewarning then
            WARNING("initial point %1 does not satisfy the inequality constraints; trying to
```

```
        find a feasible initial point", xmin)
    end if;
    for i in [30, 70, 90, 100, 150, 200, 300, 400] do
        if i = 30 then
            s := 0.01
        elif i = 70 then
            s := 0.1
        elif i = 90 then
            s := 1.00001
        elif i = 100 then
            s := 10.
        elif i = 150 then
            s := 100.
        elif i = 200 then
            s := 1000.
        elif i = 300 then
            s := 10000.
        else
            s := 1.00000·10^5
        end if;
        xmin := Matrix(n, i, datatype = float);
        ind := FAIL;
        for j to n do xmin[j] := Statistics:-Sample(Laplace(xtry[j], s), i) end do;
        for j to i do
            if IneqTrue(xmin[ ( ) ..( ), j]) then
                xtry := LinearAlgebra:-Copy(xmin[ ( ) ..( ), j]); ind := true; break
            end if
        end do;
        if ind = true then break end if
    end do;
    if ind = FAIL then
        error "cannot find feasible initial point; please, specify a new one"
    else
        xmin := [seq(params[i] = xtry[i], i = 1 ..n)];
        if usewarning then WARNING("the new feasible initial point is %1", xmin) end if
    end if
end if;
s := EvalEq(xtry);
if is(Im(s) = 0) <> true then
    xmin := [seq(params[i] = xtry[i], i = 1 ..n)];
    if usewarning then
```

```
            WARNING("unfeasible value %1 is encountered in equality constraints for initial
            point %2;trying to find a feasible initial point", s, xmin)
        end if;
        for i in [30, 70, 90, 100, 150, 200, 300, 400] do
            if i = 30 then
                s := 0.01
            elif i = 70 then
                s := 0.1
            elif i = 90 then
                s := 1.00001
            elif i = 100 then
                s := 10.
            elif i = 150 then
                s := 100.
            elif i = 200 then
                s := 1000.
            elif i = 300 then
                s := 10000.
            else
                s := 1.00000·10^5
            end if;
            xmin := Matrix(n, i, datatype = float);
            ind := FAIL;
            for j to n do xmin[j] := Statistics:-Sample(Laplace(xtry[j], s), i) end do;
            for j to i do
                if IneqTrue(xmin[( ) ..( ), j]) and is(Im(EvalEq(xmin[( ) ..( ), j])) = 0)
                then
                    xtry := LinearAlgebra:-Copy(xmin[( ) ..( ), j]); ind := true; break
                end if
            end do;
            if ind = true then break end if
        end do;
        if ind = FAIL then
            error "cannot find feasible initial point; please, specify a new one"
        else
            xmin := [seq(params[i] = xtry[i], i = 1 ..n)];
            if usewarning then WARNING("the new feasible initial point is %1", xmin) end if
        end if
    end if;
    s := EvalFun(xtry);
    if is(Im(s) = 0) <> true then
```

```
        xmin := [seq(params[i] = xtry[i], i = 1 ..n)];
        if usewarning then
            WARNING("objective function returns unfeasible value %1 for initial point %2;
            trying to find a feasible initial point", s, xmin)
        end if;
        for i in [80, 100, 200, 300] do
            if i = 80 then
                s := 1.00001
            elif i = 100 then
                s := 10.
            elif i = 200 then
                s := 10000.
            else
                s := 1.00000·10^5
            end if;
            xmin := Matrix(n, i, datatype = float);
            ind := FAIL;
            for j to n do xmin[j] := Statistics:-Sample(Laplace(xtry[j], s), i) end do;
            for j to i do
                if evaluationlimit <= Nmax then break end if;
                if IneqTrue(xmin[( ) ..( ), j]) and is(Im(EvalEq(xmin[( ) ..( ), j])) = 0)
                and is(Im(EvalFun(xmin[( ) ..( ), j])) = 0) then
                    xtry := LinearAlgebra:-Copy(xmin[( ) ..( ), j]); ind := true; break
                end if
            end do;
            if ind = true then break end if
        end do;
        if ind = FAIL then
            error "cannot find feasible initial point; please, specify a new one"
        else
            xmin := [seq(params[i] = xtry[i], i = 1 ..n)];
            if usewarning then WARNING("the new feasible initial point is %1", xmin) end if
        end if
    end if;
    if searchpath <> NULL then
        Searchpath := true;
        Spath := Matrix(n, 1, datatype = float);
        Spath[( ) ..( ), 1] := LinearAlgebra:-Copy(xtry)
    end if;
    if n = 1 then
        xleft := Vector(2, datatype = float);
```

```
xright := Vector(2, datatype = float);
fleft := Vector(2, datatype = float);
fright := Vector(2, datatype = float);
xOld := Vector(1, datatype = float);
xmin, fmin, objmin := LinearAlgebra:-Copy(xtry), ftry, objtry;
goodstep := true;
exit := false;
ind := 0;
while goodstep and Nmax < evaluationlimit do
    xtry := `~`[`+`](xmin, $, step0);
    if 0 < Nleft then
        xOld[1] := xleft[1]; fOld := fleft[1]
    else
        xOld[1] := xmin[1]; fOld := fmin
    end if;
    if Nleft <= 10 then
        shorten := 2.
    elif Nleft <= 30 then
        shorten := 10.
    elif Nleft <= 40 then
        shorten := 100.
    else
        shorten := 1000.
    end if;
    s := 0;
    if Nleft = 2 and ind = 0 then
        ind := 1;
        s := xleft[2], xleft[1], xmin[1], fleft[2], fleft[1], fmin;
        if 0 < diff2(s) then
            s := xq(s); if s <> FAIL then xtry[1] := s end if
        else
            s := qroots(s); if s <> FAIL then xtry[1] := max(s[1], s[2]) end if
        end if
    end if;
    if s <> FAIL and FeasiblePoint( ) then
        if ftry < fmin then
            goodstep := true;
            if xmin[1] < xtry[1] then
                Nleft := Nleft + 1;
                if 1 < Nleft then xleft[2] := xleft[1]; fleft[2] := fleft[1] end if;
                xleft[1] := xmin[1];
```

```
          fleft[1] := fmin
        elif ind = 1 and xtry[1] < xmin[1] then
          Nright := Nright + 1;
          goodstep := false;
          if 1 < Nright then xright[2] := xright[1]; fright[2] := fright[1]
          end if;
          xright[1] := xmin[1];
          fright[1] := fmin
        end if;
        if bound then
          step0 := abs(xtryOld[1] - xmin[1]) * 0.3819660113
        else
          step0 := shorten * step0
        end if;
        xmin, fmin, objmin := LinearAlgebra:-Copy(xtry), ftry, objtry
      else
        bound := false;
        if ind <> 1 then goodstep := false end if;
        if xmin[1] < xtry[1] then
          Nright := Nright + 1;
          goodstep := false;
          if 1 < Nright then xright[2] := xright[1]; fright[2] := fright[1]
          end if;
          xright[1] := xtry[1];
          fright[1] := ftry
        elif ind = 1 and xtry[1] < xmin[1] then
          Nleft := Nleft + 1;
          if 1 < Nleft then xleft[2] := xleft[1]; fleft[2] := fleft[1] end if;
          xleft[1] := xtry[1];
          fleft[1] := ftry
        end if
      end if
    else
      goodstep := false
    end if;
    if ind = 1 and bound or s = FAIL then goodstep := true end if;
    s := max(2, 25 * checkexit - 50);
    if ind <> 1 and bound and step0 * 3. * s <= tol[1] and abs( fOld - fmin) * s
     <= tol[2] then
      break
    end if;
    if ind = 1 then ind := 2 end if;
```

```
        if is(fmin = -infinity) then exit := true; break end if
    end do;
    if Nright = 0 and Nmax < evaluationlimit then
        if 0 < Nleft then step0 := min(tol[1], (xmin[1] - xleft[1])/2.) else
            step0 := tol[1]
        end if;
        if step0 <= 0. then step0 := tol[1] end if;
        s := 2.*step0/max(2, checkexit);
        if 0 < s then step0 := s end if;
        xtry := `~`[`-`](xmin, $, step0);
        if FeasiblePoint( ) then
            if ftry < fmin then
                if xtry[1] < xmin[1] then
                    Nright := Nright + 1;
                    if 1 < Nright then xright[2] := xright[1]; fright[2] := fright[1]
                    end if;
                    xright[1] := xmin[1];
                    fright[1] := fmin
                end if;
                xmin, fmin, objmin := LinearAlgebra:-Copy(xtry), ftry, objtry
            else
                exit := true
            end if
        else
            exit := true
        end if
    end if;
    if 0 < Nleft and 0 < Nright then goodstep := false else goodstep := true end if;
    step0 := step;
    ind := 0;
    while goodstep and Nmax < evaluationlimit do
        xtry := `~`[`-`](xmin, $, step0);
        if 0 < Nright then
            xOld[1] := xright[1]; fOld := fright[1]
        else
            xOld[1] := xmin[1]; fOld := fmin
        end if;
        if Nright <= 10 then
            shorten := 2.
        elif Nright <= 30 then
            shorten := 10.
```

```
    elif Nright <= 40 then
        shorten := 100.
    else
        shorten := 1000.
    end if;
    s := 0;
    if Nright = 2 and ind = 0 then
        ind := 1;
        s := xmin[1], xright[1], xright[2], fmin, fright[1], fright[2];
        if 0 < diff2(s) then
            s := xq(s); if s <> FAIL then xtry[1] := s end if
        else
            s := qroots(s); if s <> FAIL then xtry[1] := min(s[1], s[2]) end if
        end if
    end if;
    if s <> FAIL and FeasiblePoint( ) then
        if ftry < fmin then
            goodstep := true;
            if xtry[1] < xmin[1] then
                Nright := Nright + 1;
                if 1 < Nright then xright[2] := xright[1]; fright[2] := fright[1]
                end if;
                xright[1] := xmin[1];
                fright[1] := fmin
            elif ind = 1 and xmin[1] < xtry[1] then
                Nleft := Nleft + 1;
                goodstep := false;
                if 1 < Nleft then xleft[2] := xleft[1]; fleft[2] := fleft[1] end if;
                xleft[1] := xmin[1];
                fleft[1] := fmin
            end if;
            if bound then
                step0 := abs(xtryOld[1] - xmin[1]) * 0.3819660113
            else
                step0 := shorten * step0
            end if;
            xmin, fmin, objmin := LinearAlgebra:-Copy(xtry), ftry, objtry
        else
            bound := false;
            if ind <> 1 then goodstep := false end if;
            if xtry[1] < xmin[1] then
                Nleft := Nleft + 1;
```

```
                    goodstep := false;
                    if 1 < Nleft then xleft[2] := xleft[1]; fleft[2] := fleft[1] end if;
                    xleft[1] := xtry[1];
                    fleft[1] := ftry
                elif ind = 1 and xmin[1] < xtry[1] then
                    Nright := Nright + 1;
                    if 1 < Nright then xright[2] := xright[1]; fright[2] := fright[1]
                    end if;
                    xright[1] := xtry[1];
                    fright[1] := ftry
                end if
            end if
        else
            goodstep := false
        end if;
        if ind = 1 and bound or s = FAIL then goodstep := true end if;
        s := max(2, 25 * checkexit - 50);
        if ind <> 1 and bound and step0 * 3. * s <= tol[1] and abs(fOld - fmin) * s
         <= tol[2] then
            break
        end if;
        if ind = 1 then ind := 2 end if;
        if is(fmin = -infinity) then exit := true; break end if
    end do;
    if Nleft = 0 and Nmax < evaluationlimit then
        if 0 < Nright then
            step0 := min(tol[1], (xright[1] - xmin[1]) / 2.)
        else
            step0 := tol[1]
        end if;
        if step0 <= 0. then step0 := tol[1] end if;
        s := 2. * step0 / max(2, checkexit);
        if 0 < s then step0 := s end if;
        xtry := `~`[`+`](xmin, $, step0);
        if FeasiblePoint( ) then
            if ftry < fmin then
                if xmin[1] < xtry[1] then
                    Nleft := Nleft + 1;
                    if 1 < Nleft then xleft[2] := xleft[1]; fleft[2] := fleft[1] end if;
                    xleft[1] := xmin[1];
                    fleft[1] := fmin
```

```
                end if;
                xmin, fmin, objmin := LinearAlgebra:-Copy(xtry), ftry, objtry
            else
                exit := true
            end if
        else
            exit := true
        end if
    end if;
    if 0 < min(Nleft, Nright) then exit := false else exit := true end if;
    if 0 < Nleft and 0 < Nright and max(fleft[1], fright[1]) <= fmin then exit := true
    end if;
    Nexit := 0;
    shorten := 0.15;
    kbad := 0;
    kgood := 0;
    knonfeasiblestep := 0;
    leftcons := 0;
    rightcons := 0;
    fleftOld := 0.;
    frightOld := 0.;
    j := 0;
    fOld := max(fleft[1], fright[1]);
    kmincheck := 0;
    stepleft := true;
    parabolic := true;
    linear := false;
    simple := false;
    mincheck := false;
    while not exit and Nmax < evaluationlimit do
        feasiblestep := true;
        simple := true;
        ind := 0;
        s := 0;
        if parabolic then
            parabolic := false;
            linear := true;
            simple := false;
            ind := 1;
            s := xleft[1], xmin[1], xright[1], fleft[1], fmin, fright[1];
            if 0 < diff2(s) then
                s := xq(s); if s<>FAIL then xtry[1] := s else feasiblestep := false end if
```

```
        end if
    elif linear and 1 < Nleft and 1 < Nright and (j = 0 or j = 1 and xright[1
    ] - xleft[1] <= 4. * tol[1]) then
        parabolic := false;
        linear := false;
        simple := false;
        ind := 1;
        j := j + 1;
        s := xleft[2], xleft[1], xright[1], xright[2], fleft[2], fleft[1], fright[1], fright[2];
        s := xl(s);
        if s <> FAIL and s < xright[1] and xleft[1] < s then
            xtry[1] := s
        else
            feasiblestep := false
        end if
    else
        if mincheck and kmincheck < 2 then
            kmincheck := kmincheck + 1;
            if xright[1] - xmin[1] < xmin[1] - xleft[1] then
                xtry[1] := xmin[1] - tol[1]/2.; stepleft := true
            else
                xtry[1] := xmin[1] + tol[1]/2.; stepleft := false
            end if
        elif 0 < kgood then
            if stepleft then
                xtry[1] := xmin[1] - (xmin[1] - xleft[1]) * shorten; stepleft := true
            else
                xtry[1] := xmin[1] + (xright[1] - xmin[1]) * shorten; stepleft :=
                false
            end if
        else
            if xright[1] - xmin[1] < xmin[1] - xleft[1] then
                xtry[1] := xmin[1] - (xmin[1] - xleft[1]) * shorten; stepleft := true
            else
                xtry[1] := xmin[1] + (xright[1] - xmin[1]) * shorten; stepleft :=
                false
            end if
        end if
    end if;
    if ind = 1 and s <> FAIL and tol[1]/shorten < xright[1] - xleft[1] then
        if abs(xtry[1] - xmin[1]) <= tol[1] then mincheck := true end if
```

```
end if;
if xtry[1] = xmin[1] or xtry[1] <= xleft[1] or xright[1] <= xtry[1] then
    feasiblestep := false; shorten := 0.5; kgood := 0; kbad := 0
end if;
if s <> FAIL and feasiblestep and FeasiblePoint( ) then
    if ftry < fmin then
        goodstep := true;
        if xtry[1] < xmin[1] then
            Nright := Nright + 1;
            if 1 < Nright then xright[2] := xright[1]; fright[2] := fright[1]
            end if;
            xright[1] := xmin[1];
            fright[1] := fmin
        elif xmin[1] < xtry[1] then
            Nleft := Nleft + 1;
            if 1 < Nleft then xleft[2] := xleft[1]; fleft[2] := fleft[1] end if;
            xleft[1] := xmin[1];
            fleft[1] := fmin
        end if;
        xmin, fmin, objmin := LinearAlgebra:-Copy(xtry), ftry, objtry
    else
        goodstep := false;
        if xtry[1] < xmin[1] then
            Nleft := Nleft + 1;
            if 1 < Nleft then xleft[2] := xleft[1]; fleft[2] := fleft[1] end if;
            xleft[1] := xtry[1];
            fleft[1] := ftry
        elif xmin[1] < xtry[1] then
            Nright := Nright + 1;
            if 1 < Nright then xright[2] := xright[1]; fright[2] := fright[1]
            end if;
            xright[1] := xtry[1];
            fright[1] := ftry
        end if;
        if xright[1] - xleft[1] <= tol[1] then
            if Nleft = 1 then
                if
                fleftOld <= fleft[1] - fmin then
                    leftcons := leftcons + 1
                else
                    leftcons := 0
```

```
                end if;
                fleftOld := fleft[1] - fmin
            elif 1 < Nleft then
                if (fleft[2] - fleft[1]) / tol[2] <= fleft[1] - fmin then
                    leftcons := leftcons + 1
                else
                    leftcons := 0
                end if
            end if;
            if Nright = 1 then
                if
                frightOld <= fright[1] - fmin then
                    rightcons := rightcons + 1
                else
                    rightcons := 0
                end if;
                frightOld := fright[1] - fmin
            elif 1 < Nright then
                if (fright[2] - fright[1]) / tol[2] <= fright[1] - fmin then
                    rightcons := rightcons + 1
                else
                    rightcons := 0
                end if
            end if
        end if
    end if
else
    feasiblestep := false
end if;
if feasiblestep and simple then
    if goodstep then
        kbad := 0;
        kgood := kgood + 1;
        if kgood = 1 then
            shorten := 0.15
        elif kgood = 2 then
            shorten := 0.3
        elif kgood = 3 then
            shorten := 0.6
        elif kgood = 4 then
            shorten := 0.9
        elif kgood = 5 then
```

```
            shorten := 0.99
        else
            shorten := 0.999
        end if
    else
        parabolic := true;
        linear := false;
        simple := false;
        kgood := 0;
        if 0 < kgood then
            shorten := 0.15; kbad := 0
        else
            kbad := kbad + 1;
            if kbad < 3 then
                shorten := 0.15
            elif kbad < 5 then
                shorten := 0.015
            elif kbad < 7 then
                shorten := 0.0015
            else
                shorten := 0.001
            end if
        end if
    end if
end if;
if feasiblestep = false then
    knonfeasiblestep := knonfeasiblestep + 1
else
    knonfeasiblestep := 0
end if;
if ind = 0 then
    if 10 < leftcons then
        fOld := fright[1]
    elif 10 < rightcons then
        fOld := fleft[1]
    else
        fOld := max(fleft[1], fright[1])
    end if;
    if (xright[1] - xleft[1]) * 2. <= tol[1] and (fOld - fmin) * 2. <= tol[2]
    then
        Nexit := Nexit + 1
    else
```

```
                Nexit := 0
            end if
        end if;
        if 0 < Nleft and 0 < Nright and max( fleft[1], fright[1]) <= fmin
        then
            exit := true
        end if;
        if checkexit <= Nexit or 8 <= knonfeasiblestep or xright[1] - xleft[1] <= 0. or
        max( fleft[1], fright[1]) <= fmin or is( fmin = -infinity) then
            exit := true
        end if
    end do
else
    xleft := Matrix(n, 2, datatype = float);
    xright := Matrix(n, 2, datatype = float);
    fleft := Vector(2, datatype = float);
    fright := Vector(2, datatype = float);
    xmax := Vector(n, datatype = float);
    fmax := Vector(n, datatype = float);
    indconj := Vector(n, datatype = integer);
    u := Matrix(n, n, datatype = float);
    for i to n do u[i, i] := 1. end do;
    xmin, fmin, objmin := LinearAlgebra:-Copy(xtry), ftry, objtry;
    xmin1 := LinearAlgebra:-Copy(xmin);
    fOld := fmin;
    exit := false;
    ind := 0;
    step0 := step;
    stepshift := evalf(step0 * 0.62);
    if stepshift <= 0. then stepshift := step0 end if;
    for i to n do
        xtry := xmin1 + step0 * u[( ) ..( ), i];
        utry := LinearAlgebra:-Copy(u[( ) ..( ), i]);
        if FeasiblePoint( ) and 0 < LinearAlgebra:-VectorNorm(xtry - xmin1, 2,
        conjugate = false) then
            fOld := max( fOld, ftry);
            u[i, i] := ( ftry - fmin) / LinearAlgebra:-VectorNorm(xtry - xmin1, 2,
            conjugate = false);
            if ftry < fmin then
                xmin, fmin, objmin := LinearAlgebra:-Copy(xtry), ftry, objtry;
                xmin1 := LinearAlgebra:-Copy(xmin)
            end if;
```

```
            if evaluationlimit <= Nmax then exit := true; break end if
        else
            if evaluationlimit <= Nmax then exit := true; break end if;
            xtry := xmin1 − step0 * u[ ( ) ..( ), i];
            utry := LinearAlgebra:-Copy( − u[ ( ) ..( ), i]);
            if FeasiblePoint( ) and 0 < LinearAlgebra:-VectorNorm(xtry − xmin1, 2,
            conjugate = false) then
                fOld := max( fOld, ftry);
                u[i, i] := − ( ftry − fmin) / LinearAlgebra:-VectorNorm(xtry − xmin1, 2,
                conjugate = false);
                if ftry < fmin then
                    xmin, fmin, objmin := LinearAlgebra:-Copy(xtry), ftry, objtry;
                    xmin1 := LinearAlgebra:-Copy(xmin)
                end if
            else
                u[i, i] := 0.
            end if
        end if;
        if is( fmin = − infinity) or evaluationlimit <= Nmax then exit := true; break
        end if
    end do;
    for i from 2 to n do u[ ( ) ..( ), 1] := u[ ( ) ..( ), 1] + u[ ( ) ..( ), i] end do;
    if ArrayTools:-IsZero(u[ ( ) ..( ), 1]) then for i to n do u[i, 1] := 1. end do end if;
    for i to n do
        u[ ( ) ..( ), i] := LinearAlgebra:-Normalize(u[ ( ) ..( ), i], 2, conjugate = false)
    end do;
    Nleft, Nright, ind := 0, 0, 0;
    xtry, ftry, objtry := LinearAlgebra:-Copy(xmin), fmin, objmin;
    xmin1 := LinearAlgebra:-Copy(xmin);
    utry := LinearAlgebra:-Copy(u[ ( ) ..( ), 1]);
    if not DirectionalMove(step0) then
        xmin1 := LinearAlgebra:-Copy(xmin);
        s := FAIL;
        for i to 1000 do
            if evaluationlimit <= Nmax then exit := true; break end if;
            xtry := Statistics:-Sample(R, n);
            xtry := xmin + xtry^`%T`;
            if not IneqTrue(xtry) then next end if;
            utry := LinearAlgebra:-Normalize(xmin − xtry, 2, conjugate = false);
            if not FeasiblePoint( ) or LinearAlgebra:-VectorNorm(xtry − xmin1, 2,
            conjugate = false) <= 0 then
```

```
                next
            end if;
            s := true;
            break
        end do;
        if s = FAIL and not exit and usewarning then
            WARNING("cannot find feasible initial direction of search; please, specify a
            new initial point, decrease step or repeat search")
        end if;
        if s = FAIL then exit := true end if;
        if s = true and not exit and Nmax < evaluationlimit then
            if ftry < fmin then
                Nleft, Nright, ind := 1, 0, 0;
                xleft[ ( ) ..( ), 1 ] := LinearAlgebra:-Copy(xmin);
                fleft[1] := fmin;
                utry := LinearAlgebra:-Normalize(xtry − xmin, 2, conjugate = false);
                xmin, fmin, objmin := LinearAlgebra:-Copy(xtry), ftry, objtry;
                xmin1 := LinearAlgebra:-Copy(xmin)
            else
                Nleft, Nright, ind := 1, 0, 0;
                xleft[ ( ) ..( ), 1 ] := LinearAlgebra:-Copy(xtry);
                fleft[1] := ftry;
                fOld := max( fOld, ftry);
                utry := LinearAlgebra:-Normalize(xmin − xtry, 2, conjugate = false);
                xtry, ftry, objtry := LinearAlgebra:-Copy(xmin), fmin, objmin;
                xmin1 := LinearAlgebra:-Copy(xmin)
            end if;
            DirectionalMove(step0);
            u[ ( ) ..( ), 1 ] := LinearAlgebra:-Copy(utry);
            if 0 < Nright then fOld := max( fOld, fright[1]) end if
        end if
    end if;
    for i from 2 to n do u[i, i] := 1. end do;
    indconj[1] := 2;
    if fmin1 < fmin then
        xmin, fmin, objmin := LinearAlgebra:-Copy(xmin1), fmin1, objmin1
    end if;
    if fOld <= fmin then exit := true end if;
    xmin0, fmin0, objmin0 := LinearAlgebra:-Copy(xmin), fmin, objmin;
    if not exit and Nmax < evaluationlimit then
        for i from 2 to n do
            xmin0, fmin0, objmin0 := LinearAlgebra:-Copy(xmin), fmin, objmin;
```

```
s := LinearAlgebra:-GramSchmidt([seq(u[( ) .. ( ), j], j = 1 ..i)], conjugate
 = false);
k := nops(s);
if k < i then
    for i1 to 100 do
        s := [op(s[1 ..k]), seq(Statistics:-Sample(R, n)^`%T`, j = 1 ..i - k)];
        s := LinearAlgebra:-GramSchmidt(s, conjugate = false, normalized);
        k := nops(s);
        if k = i then break end if
    end do
end if;
utry := LinearAlgebra:-Normalize(s[k], 2, conjugate = false);
xmin1 := LinearAlgebra:-Copy(xmin);
xtry := xmin + stepshift * utry;
s := FAIL;
k := reduction = false;
for i1 to 600 do
    if evaluationlimit <= Nmax then exit := true; break end if;
    if FeasiblePoint(k) and 0 < LinearAlgebra:-VectorNorm(xtry - xmin1,
    2, conjugate = false) then
        if ftry < fmin then
            xmin, fmin, objmin := LinearAlgebra:-Copy(xtry), ftry, objtry;
            xmin1 := LinearAlgebra:-Copy(xmin)
        end if;
        if evaluationlimit <= Nmax then exit := true; break end if;
        s := true;
        break
    else
        if evaluationlimit <= Nmax then exit := true; break end if;
        xtry := xmin - stepshift * utry;
        if FeasiblePoint(k) and 0 < LinearAlgebra:-
        VectorNorm(xtry - xmin1, 2, conjugate = false) then
            if ftry < fmin then
                xmin, fmin, objmin := LinearAlgebra:-Copy(xtry), ftry, objtry;
                xmin1 := LinearAlgebra:-Copy(xmin)
            end if;
            s := true;
            break
        else
            utry := Statistics:-Sample(R, n);
            utry := LinearAlgebra:-Normalize(utry^`%T`, 2, conjugate
             = false);
```

```
            xtry := xmin + stepshift * utry
        end if
    end if;
    if 500 < i1 then k := reduction = true end if
end do;
if s = FAIL and not exit and usewarning then
    WARNING("cannot find %1 feasible initial directions of search; please,
    specify a new initial point, decrease step or repeat search", n)
end if;
if s = FAIL then exit := true; break end if;
xmin1, fmin1, objmin1 := LinearAlgebra:-Copy(xtry), ftry, objtry;
for j to i − 1 do
    Nleft, Nright, ind := 0, 0, indconj[j];
    xtry, ftry, objtry := LinearAlgebra:-Copy(xmin1), fmin1, objmin1;
    utry := LinearAlgebra:-Copy(u[ ( ) ..( ), j]);
    DirectionalMove(step0);
    if fmin1 < fmin then
        xmin, fmin, objmin := LinearAlgebra:-Copy(xmin1), fmin1, objmin1
    end if;
    indconj[j] := 2
end do;
Nleft, Nright, ind := 1, 0, indconj[i];
if fmin0 < fmin1 then
    xleft[ ( ) ..( ), 1] := LinearAlgebra:-Copy(xmin1);
    fleft[1] := fmin1;
    xtry, ftry, objtry := LinearAlgebra:-Copy(xmin0), fmin0, objmin0;
    u[ ( ) ..( ), i] := LinearAlgebra:-Normalize(xmin0 − xmin1, 2, conjugate
     = false)
else
    xleft[ ( ) ..( ), 1] := LinearAlgebra:-Copy(xmin0);
    fleft[1] := fmin0;
    xtry, ftry, objtry := LinearAlgebra:-Copy(xmin1), fmin1, objmin1;
    u[ ( ) ..( ), i] := LinearAlgebra:-Normalize(xmin1 − xmin0, 2, conjugate
     = false)
end if;
if ArrayTools:-IsZero(u[ ( ) ..( ), i]) then u[i, i] := 1. end if;
utry := LinearAlgebra:-Copy(u[ ( ) ..( ), i]);
DirectionalMove(step0);
if fmin1 < fmin then
    xmin, fmin, objmin := LinearAlgebra:-Copy(xmin1), fmin1, objmin1
end if;
indconj[i] := 2
```

```
        end do
    end if;
    Nexit := 0;
    fOld := fmin;
    isquadratic := true;
    s := LinearAlgebra:-VectorNorm(xmin0 − xmin, 2, conjugate = false);
    step0 := evalf(s * 0.38);
    stepav := step0;
    Niter := 1;
    if step0 <= 0. then step0 := evalf(0.1 * step) end if;
    if step0 <= 0. then step0 := tol[1] end if;
    stepshift := evalf(step0 * 0.62);
    if stepshift <= 0. then stepshift := step0 end if;
    while not exit and Nmax < evaluationlimit do
        xmin0, fmin0, objmin0 := LinearAlgebra:-Copy(xmin), fmin, objmin;
        s := LinearAlgebra:-GramSchmidt([seq(u[( )..( ), n − j + 1], j = 1..n)],
        conjugate = false);
        k := nops(s);
        if k < n then
            for i1 to n do
                utry := Statistics:-Sample(R, n);
                u[( )..( ), i1] := LinearAlgebra:-Normalize(utry^`%T`, 2, conjugate
                 = false)
            end do
        end if;
        utry := LinearAlgebra:-Normalize(s[k], 2, conjugate = false);
        xmin1 := LinearAlgebra:-Copy(xmin);
        xtry := xmin + stepshift * utry;
        s := FAIL;
        if evaluationlimit <= Nmax then exit := true; break end if;
        if FeasiblePoint(reduction = false) and 0 < LinearAlgebra:-
        VectorNorm(xtry − xmin1, 2, conjugate = false) then
            if ftry < fmin then
                xmin, fmin, objmin := LinearAlgebra:-Copy(xtry), ftry, objtry;
                xmin1 := LinearAlgebra:-Copy(xmin)
            end if;
            if evaluationlimit <= Nmax then exit := true; break end if;
            s := true
        else
            if evaluationlimit <= Nmax then exit := true; break end if;
            xtry := xmin − stepshift * utry;
            if FeasiblePoint(reduction = false) and 0 < LinearAlgebra:-
```

```
        VectorNorm(xtry − xmin1, 2, conjugate = false) then
            if ftry < fmin then
                xmin, fmin, objmin := LinearAlgebra:-Copy(xtry), ftry, objtry;
                xmin1 := LinearAlgebra:-Copy(xmin)
            end if;
            s := true
        else
            xtry := xmin + stepshift * utry;
            if FeasiblePoint( ) and 0 < LinearAlgebra:-VectorNorm(xtry − xmin1,
            2, conjugate = false) then
                if ftry < fmin then
                    xmin, fmin, objmin := LinearAlgebra:-Copy(xtry), ftry, objtry;
                    xmin1 := LinearAlgebra:-Copy(xmin)
                end if;
                s := true
            elif Nmax < evaluationlimit then
                xtry := xmin − stepshift * utry;
                if FeasiblePoint( ) and 0 < LinearAlgebra:-
                VectorNorm(xtry − xmin1, 2, conjugate = false) then
                    if ftry < fmin then
                        xmin, fmin, objmin := LinearAlgebra:-Copy(xtry), ftry, objtry;
                        xmin1 := LinearAlgebra:-Copy(xmin)
                    end if;
                    s := true
                end if
            end if
        end if
    end if;
    if s = FAIL then
        xtry, ftry, objtry := LinearAlgebra:-Copy(xmin), fmin, objmin
    else
        u := ArrayTools:-CircularShift(u, 0, −1)
    end if;
    xmin1, fmin1, objmin1 := LinearAlgebra:-Copy(xtry), ftry, objtry;
    for i to n − 1 do
        Nleft, Nright, ind := 0, 0, 2;
        xtry, ftry, objtry := LinearAlgebra:-Copy(xmin1), fmin1, objmin1;
        utry := LinearAlgebra:-Copy(u[( )..( ), i]);
        DirectionalMove(3.0 * step0);
        if fmin1 < fmin then
            xmin, fmin, objmin := LinearAlgebra:-Copy(xmin1), fmin1, objmin1
        end if
```

```
end do;
Nleft, Nright, ind := 1, 0, 2;
if fmin0 < fmin1 then
    xleft[ ( ) ..( ), 1] := LinearAlgebra:-Copy(xmin1);
    fleft[1] := fmin1;
    xtry, ftry, objtry := LinearAlgebra:-Copy(xmin0), fmin0, objmin0;
    u[ ( ) ..( ), n] := LinearAlgebra:-Normalize(xmin0 − xmin1, 2, conjugate
     = false)
else
    xleft[ ( ) ..( ), 1] := LinearAlgebra:-Copy(xmin0);
    fleft[1] := fmin0;
    xtry, ftry, objtry := LinearAlgebra:-Copy(xmin1), fmin1, objmin1;
    u[ ( ) ..( ), n] := LinearAlgebra:-Normalize(xmin1 − xmin0, 2, conjugate
     = false)
end if;
if ArrayTools:-IsZero(u[ ( ) ..( ), n]) then
    utry := Statistics:-Sample(R, n);
    u[ ( ) ..( ), n] := LinearAlgebra:-Normalize(utry^`%T`, 2, conjugate = false)
end if;
utry := LinearAlgebra:-Copy(u[ ( ) ..( ), n]);
DirectionalMove(step0);
if fmin1 < fmin then
    xmin, fmin, objmin := LinearAlgebra:-Copy(xmin1), fmin1, objmin1
end if;
if eqtrue and r < 1./tol[2] then
    r := 10.*r; fmin := EvalFun(xmin); objmin, eqmin := objtry, eqtry
end if;
s := LinearAlgebra:-VectorNorm(xmin0 − xmin, 2, conjugate = false);
step0 := evalf(0.3*s + step0*0.091);
if step0 <= 0. then step0 := s end if;
if step0 <= 0. then step0 := evalf(0.38*tol[1]) end if;
if step0 <= 0. then step0 := tol[1] end if;
stepshift := evalf(step0*0.62);
if stepshift <= 0. then stepshift := step0 end if;
stepav := stepav + step0;
Niter := Niter + 1;
if checkexit = 1 and isquadratic and s <= tol[1] and abs(fmin0 − fmin) <= tol
[2] then
    if tol[1] < step0 then
        step0 := tol[1];
        stepshift := evalf(step0*0.62);
        if stepshift <= 0. then stepshift := step0 end if
```

```
        end if
      end if;
      isquadratic := true;
      if step0 <= 1.1 * tol[1] and fOld - fmin <= tol[2] then
        Nexit := Nexit + 1
      else
        Nexit := 0; fOld := fmin
      end if;
      if checkexit <= Nexit or is( fmin = -infinity) then exit := true end if
    end do
  end if;
  if searchpath <> NULL then searchpath := Spath end if;
  if maximize then objmin := -objmin end if;
  if 0 < checksolution then
    stepav := stepav / Niter;
    step0 := stepav;
    opt := [_options['checkexit'], _options['maximize'], _options['evaluationlimit'],
    _options['tolerances']];
    if assume <> NULL then opt := [op(opt), _options['assume']] end if;
    if variables <> NULL then
      opt := [op(opt), _options['variables']]
    else
      opt := [op(opt), lhs(_options['variables']) = params]
    end if;
    while Nmax < evaluationlimit do
      i1 := [seq(xmin[i] - 10. * step0 ..xmin[i] + 10. * step0, i = 1 ..n)];
      s := DirectSearch:-GlobalSearch(OBJ, constr, op(opt), lhs(_options['step'])
       = step0, pointrange = i1, number = checksolution, totalevaluations = 'k');
      Nmax := Nmax + k;
      stepav := LinearAlgebra:-VectorNorm(s[1, 2] - xmin, 2, conjugate = false);
      step0 := stepav;
      if step0 <= 0. then step0 := tol[1] end if;
      if maximize then
        if objmin < s[1, 1] then
          if abs(s[1, 1] - objmin) <= tol[2] and stepav <= tol[1] then
            xmin, objmin := LinearAlgebra:-Copy(s[1, 2]), s[1, 1]; break
          else
            xmin, objmin := LinearAlgebra:-Copy(s[1, 2]), s[1, 1]
          end if
        else
          break
        end if
```

```
                else
                    if s[1, 1] < objmin then
                        if abs(s[1, 1] - objmin) <= tol[2] and stepav <= tol[1] then
                            xmin, objmin := LinearAlgebra:-Copy(s[1, 2]), s[1, 1]; break
                        else
                            xmin, objmin := LinearAlgebra:-Copy(s[1, 2]), s[1, 1]
                        end if
                    else
                        break
                    end if
                end if
            end do
        end if;
        if evaluationlimit <= Nmax and usewarning then
            WARNING("limiting number of function evaluations (%1) is reached; set initial point
            equal to extremum point obtained, increase evaluationlimit option and continue
            search", Nmax)
        end if;
        if type(OBJ, 'procedure') or variables <> NULL then
            s := [Re(objmin), xmin, Nmax]
        else
            s := [Re(objmin), [seq(params[i] = xmin[i], i = 1 ..n)], Nmax]
        end if;
        s
    end proc
> ?DirectSearch
> op(GlobalSearch);

proc(OBJ::{algebraic, procedure}, constr::(expects({set(relation), list(relation)})) :=                    (4)
    NULL, {assume::(identical(positive, negative, nonpositive, nonnegative)) := NULL,
    checkexit::posint := 2, checksolution::nonnegint := 0, distance::positive := 0.01,
    evaluationlimit::posint := 2000, initialpoints::{Matrix, list(Array), list(Vector),
    list(list)} := NULL, maximize::truefalse := false, number::posint := 100, pointrange::
    {list(range), list(name = range)} := NULL, solutions::posint := NULL, step::positive :=
    0.005, tolerances::{positive, [positive], [positive, positive]} := [0.000001000000000,
    0.000001000000000], totalevaluations::name := NULL, variables::(list(name)) := NULL})
    option hfloat;
    local n, i, i1, j, k, Nmax, N, N0, Nsol, Obj, params, pointrange1, f, s, s1, R, x0, xcenter, x1,
    UseHfloat, tol, xleft, xright, opt, sol, err, thesame, extrem, extremrank;
    description "numeric optimization, global minimum (maximum) of a function searching";
    if UseHardwareFloats = true or UseHardwareFloats = deduced and Digits
     <= evalhf(Digits) then
```

```
        UseHfloat := true; Digits := round(evalhf(Digits))
    else
        UseHfloat := false
    end if;
    pointrange1 := pointrange;
    tol := tolerances;
    if type(tol, positive) then tol := [tol] end if;
    if nops(tol) = 2 then
        tol := Statistics:-EvaluateToFloat(convert(tol, 'Vector'))
    else
        tol := Statistics:-EvaluateToFloat(`<,>`(tol[1], tol[1]))
    end if;
    s := evalf(-log10(min(tol[1], tol[2])));
    if Digits < s then
        s := ceil(s);
        WARNING("tolerance %1 is too small for Digits %2; increasing Digits to %3. Note:
        software floats are slower than hardware floats", min(tol[1], tol[2]), Digits, s);
        Digits := s
    end if;
    Obj := evalf(eval(OBJ));
    if variables <> NULL then
        params := variables
    elif variables = NULL and type(Obj, 'procedure') then
        params := [op(1, eval(Obj))]
    else
        params := [op(indets(Obj, name))];
        if has(Obj, 'Int') or has(Obj, 'int') then
            s := [op(indets(Obj, 'equation'))];
            if 0 < nops(s) then
                s := [seq(lhs(s[i]), i = 1 ..nops(s))];
                s := select(type, s, 'name');
                if 0 < nops(s) then params := remove(has, params, {op(s)}) end if
            end if
        end if
    end if;
    n := nops(params);
    Nmax := 0;
    if n = 0 then error "number of problem variables are equal to 0" end if;
    if variables = NULL and not type(OBJ, 'procedure') and 1 < n and initialpoints
    <> NULL then
        if type(initialpoints[1], {'Array(realcons)', 'Matrix(realcons)', 'Vector(realcons)',
        'list(realcons)'}) then
```

```
            error
            "please specify option variables with list of problem variable names or provide
            initial points as a list or set of equations varname=value"
        end if
    end if;
    opt := [usewarning = false, _options['checkexit'], _options['checksolution'], _options[
    'maximize'], _options['step'], _options['evaluationlimit'], _options['tolerances']];
    if assume <> NULL then opt := [op(opt), _options['assume']] end if;
    if variables <> NULL then
        opt := [op(opt), _options['variables']]
    else
        opt := [op(opt), lhs(_options['variables']) = params]
    end if;
    xcenter := Vector(n, datatype = float);
    if initialpoints = NULL and pointrange = NULL then
        s1 := [seq(HFloat(0.00899999999999999932), i = 1 ..n)];
        err := true;
        try
            s := DirectSearch:-Search(Obj, constr, op(opt), initialpoint = s1, step
             = 200);
            err := false
        catch:
        end try;
        if err = false then
            sol := [s]; xcenter := LinearAlgebra:-Copy(s[2]); Nmax := Nmax + s[3]
        else
            sol := [ ]
        end if;
        N0 := 2
    else
        sol := [ ]; N0 := 1
    end if;
    if initialpoints = NULL then
        N := number;
        if pointrange <> NULL and nops(pointrange) <> n then
            error "number of point ranges %1 must coincides with number of variables %2",
            nops(pointrange), n
        end if;
        xleft := Vector(n);
        xright := Vector(n);
        if pointrange = NULL then
            for i to n do
```

```
            xleft[i] := xcenter[i] − HFloat(100.); xright[i] := xcenter[i] + HFloat(100.)
        end do
    else
        if type(pointrange1[1], 'equation') then
            pointrange1 := [ ];
            for i to n do
                if has(pointrange, params[i]) then
                    pointrange1 := [op(pointrange1), op(select(has, pointrange, params[i
                    ]))]
                else
                    error "do not find variable %1 in pointrange list", params[i]
                end if
            end do;
            for i to n do
                xleft[i] := evalf(lhs(rhs(pointrange1[i]))); xright[i] :=
                evalf(rhs(rhs(pointrange1[i])))
            end do
        else
            if variables = NULL and not type(OBJ, 'procedure') and 1 < n then
                error
                "please specify option variables with list of problem variable names or
                provide pointrange option as a list of equations varname=range"
            end if;
            for i to n do
                xleft[i] := evalf(lhs(pointrange1[i])); xright[i] := evalf(rhs(pointrange1
                [i]))
            end do
        end if
    end if;
    x1 := Matrix(n, N, datatype = float);
    for i to n do x1[i] := Statistics:-Sample(Uniform(xleft[i], xright[i]), N) end do;
    if type(OBJ, 'procedure') or variables <> NULL then
        x0 := LinearAlgebra:-Copy(x1)
    else
        x0 := [ ]; for i to N do
            x0 := [op(x0), [seq(params[j] = x1[j, i], j = 1 ..n)]]
        end do
    end if
else
    x0 := initialpoints;
    if type(x0, 'Matrix') then
        N := Statistics:-Count(x0[1, ( ) ..( )])
```

```
    else
        N := Statistics:-Count(x0)
    end if
end if;
for i from N0 to N do
    if type(x0, 'Matrix') then s1 := x0[( )..( ), i] else s1 := x0[i] end if;
    err := true;
    try
        s := DirectSearch:-Search(Obj, constr, op(opt), initialpoint = s1); err := false
    catch:
    end try;
    if err = false then
        Nmax := Nmax + s[3];
        if nops(sol) = 0 then
            sol := [s]
        else
            thesame := false;
            for j to nops(sol) do
                if LinearAlgebra:-VectorNorm(s[2] - sol[j][2], 2, conjugate = false)
                 <= distance then
                    thesame := true;
                    if maximize then
                        if sol[j][1] < s[1] then
                            sol := [op(sol[1 ..j - 1]), s, op(sol[j + 1 ..nops(sol)])]
                        end if
                    else
                        if s[1] < sol[j][1] then
                            sol := [op(sol[1 ..j - 1]), s, op(sol[j + 1 ..nops(sol)])]
                        end if
                    end if;
                    break
                end if
            end do;
            if thesame = false then sol := [op(sol), s] end if
        end if
    end if
end do;
if nops(sol) = 0 then
    sol := [DirectSearch:-Search(OBJ, constr, op(opt), initialpoint = s1)];
    Nmax := Nmax + sol[1][3]
end if;
for i to nops(sol) do
```

```
    if sol[i] = 0 then next end if;
    for j to nops(sol) do
        if i = j or sol[j] = 0 then next end if;
        if LinearAlgebra:-VectorNorm(sol[i][2] − sol[j][2], 2, conjugate = false)
         <= distance then
            if maximize then
                if sol[j][1] < sol[i][1] then
                    sol := [op(sol[1 ..j − 1]), 0, op(sol[j + 1 ..nops(sol)])]
                else
                    sol := [op(sol[1 ..i − 1]), 0, op(sol[i + 1 ..nops(sol)])]
                end if
            else
                if sol[i][1] < sol[j][1] then
                    sol := [op(sol[1 ..j − 1]), 0, op(sol[j + 1 ..nops(sol)])]
                else
                    sol := [op(sol[1 ..i − 1]), 0, op(sol[i + 1 ..nops(sol)])]
                end if
            end if;
            break
        end if
    end do
end do;
sol := remove(is, sol, 0);
Nsol := nops(sol);
extrem := Array(1 ..Nsol, datatype = float);
for i to Nsol do extrem[i] := sol[i][1] end do;
sol := convert(sol, 'Array', datatype = anything);
if maximize then
    extremrank := Statistics:-Rank(extrem, order = descending);
    sol := Statistics:-OrderByRank(sol, extremrank, order = ascending)
else
    extremrank := Statistics:-Rank(extrem, order = ascending);
    sol := Statistics:-OrderByRank(sol, extremrank, order = ascending)
end if;
if solutions <> NULL and solutions < Nsol then Nsol := solutions end if;
if not (type(OBJ, 'procedure') or variables <> NULL) then
    for i to Nsol do
        sol[i, 2] := [seq(params[j] = sol[i, 2][j], j = 1 ..n)]
    end do
end if;
if totalevaluations <> NULL then totalevaluations := Nmax end if;
sol[1 ..Nsol]
```

```
end proc
>
```