Procédure de minimisation
(adaptée à l'ajustement de courbe par minimisation d'un chi2)

introduction

• De nombreux laboratoires de recherche en physique utilisent le logiciel de minimisation MINUIT pour ajuster des modèles théoriques sur des données expérimentales.

Aux environs de 1970, au fur et à mesure de ses développements successifs, le logiciel MINUIT s'était toutefois déjà progressivement transformé en "usine à gaz" : un logiciel hyper complet, mais d'utilisation très lourde pour effectuer des actions basiques. La situation a encore abominablement empiré depuis (l'usine à gaz est devenue carrément "intergalactique"... les fous d'informatique peuvent le vérifier facilement avec une petite recherche sur internet : rien que pour en comprendre les rouages élémentaires, il faut se plonger dedans pendant trois jours).

Dès le début des complications, Berthon et Portes avaient choisi de simplifier la tâche des utilisateurs "ordinaires" en en programmant (en fortran) une version très basique : MINCON ; beaucoup plus simple à mettre en oeuvre, mais possédant de nombreuses fonctionnalités dont une prise en compte efficace des calculs d'incertitudes.

• Dans les années 1980, faute de logiciels équivalents sur Macintosh, j'en avais reprogrammé une version en pascal ; encore plus simplifiée, mais j'y avais joint une interface rudimentaire avec un interpréteur de formules et un traceur de graphiques (MINGRAPH ; dont le code est sur mon site).

L'évolution des ordinateurs et de leurs systèmes d'exploitation nécessite toutefois une mise à niveau incessante des logiciels. Je n'ai hélas que très peu de temps pour assurer cela. Or, bien que plusieurs logiciels récents disposent de fonctionnalités semblables, aucun (ni même Maple ou Mathematica, semble-t-il) ne gère efficacement les calculs d'incertitudes et des corrélations. L'enseignement de ces dernières s'est d'ailleurs un peu perdu, à tel point qu'il semble que de nombreux enseignants eux mêmes n'en maîtrisent plus très bien certains aspects.

• Je tente ici de mettre au point une version Maple de la procédure de minimisation MINIMI.

J M Laffaille - avril 2014

```
> restart
> # pour faciliter les mises à jour (et éviter d'encombrer les fichiers d'utilisation), on choisir
      d'inclure Minimi dans une bibliothèque
  with(LibraryTools)
[ActivationModule, AddFromDirectory, Author, Browse, BuildFromDirectory,                    (1)
   ConvertVersion, Create, Delete, FindLibrary, PrefixMatch, Priority, Save,
   ShowContents, Timestamp, UpdateFromDirectory, WriteMode]
> savelibname := "MinimiLib_42.mla"
                          savelibname := "MinimiLib_42.mla"                                 (2)
> # remarque générale : pour forcer Maple à transmettre par référence certains arguments
      qu'on souhaite variables, on les définit ici comme élément d'un vecteur à une seule
      composante
```

fonc

```
> # prédéfini pour la compilation (modèle parabolique), mais doit etre redéfini selon le
      modèle de l'utilisateur
  fonc := proc(NPar :: integer, p :: Array(1 ..30, datatype = float), x :: float) :: float;
  local
    f :: float;
  description "définit la fonction théorique ajustée sur les données expérimentales";

  # les paramètres utilisés dans l'expression ajustée sont ici notés p[i]
  # des noms plus explicites sont définis dans le programme appelant minimi, mais non
      utilisés ici
  # (ceci n'a rien d'obligatoire, fonc n'est utilisé que par FCN qui peut etre écrit autrement)

  # le nombre maximum de paramètres est fixé à 30, ceci est imposé dans minimi
  # le nombre de paramètres effectivement actifs est NPar, les suivants sont ignorés
  # (NPar n'est transmis ici que pour vérification éventuelle)

  # l'abscisse est notée x

    f := p[1]·x^2 + p[2]·x + p[3]; # ici on ajuste une parabole
    return(f);
      # non nécessaire dans Maple (il retourne par défaut la dernière quantité calculée)
  end proc:
> Save('fonc')
```

FCN

```
> # prédéfini pour la compilation (cas d'un chi2), mais peut etre redéfini par l'utilisateur
  FCN := proc(NPar :: integer, xp :: Array(1 ..30, datatype = float)) :: float;
  global NPts, xPts, yPts, dxPts, dyPts;

      # De façon générale, minimi ne sait pas a priori de quoi peut dépendre FCN en plus
      des paramètres ;
   # ces quantités, si elles existent, ne sont donc pas transmises en arguments
   # mais comme des variables globales du programme utilisateur de minimi.
  local
    f :: float,
    ii :: integer,
    ddd :: float;
  description "définit la fonction minimisée (généralement un chi2)";
    # le nom FCN est réservé par minimi

    f := 0;
    # reste à définir la fonction fonc(NPar, params, abscisse) qui décrit le modèle ajusté
   for ii from 1 to NPts do
    # ici on ajoute quadratiquement l'incertitude de l'ordonnée des points expérimentaux...

      # ...et la propagation sur l'ordonnée théorique de l'incertitude de l'abscisse des points
      expérimentaux
    # (en supposant que les abscisses et les ordonnées sont des mesures indépendantes)
```

```
    ddd := dyPts[ii]^2
        + ( 1/2 (fonc(NPar, xp, xPts[ii] + dxPts[ii]) - fonc(NPar, xp, xPts[ii]
        - dxPts[ii])) )^2;
    f := f + (yPts[ii] - fonc(NPar, xp, xPts[ii]))^2 / ddd;
    end do;
    return(f);
      # non nécessaire dans Maple (il retourne par défaut la dernière quantité calculée)
  end proc:
> Save('FCN')
```

SorInt

```
> # impression de valeurs intermédiaires lors de la minimisation
  SorInt := proc(NSt :: integer, NPar :: integer, xp :: Array(1..30, datatype = float), xpN
        :: Array(1..30, datatype = string), AMin1 :: float) :: string;
  local ii :: integer, Tlist :: string;
  description "liste des valeurs intermédiaires des paramètres";
    Tlist := cat("\r\rParamètres pour le pas numéro : ", NSt, "  (Min = ", convert(AMin1,
      string), " )");
    for ii from 1 to NPar do
      Tlist := cat(Tlist , "\r", xpN[ii], " : ", convert(xp[ii], string));
    end do;
    return(Tlist);
  end proc:
> Save('SorInt')
```

descente

```
> # suite de pas pour rechercher un minimum dans une direction
  descente := proc(NPar :: integer, xp :: Array(1..30, datatype = float), jg :: integer, yy
        :: Array(1..30, datatype = float), DirIn :: Array(1..31, datatype = float), dd
        :: Array(1..31, datatype = float), xs :: Array(1..31, 1..31, datatype = float), AMin
        :: Array(1..1, datatype = float), NFCN :: Array(1..1, datatype = integer), NPFN
        :: Array(1..1, datatype = integer), Yk :: Array(1..2, datatype = float), aa :: Array(1
      ..1, datatype = float))
  local
    al :: float, be :: float,
    NStepQ :: integer,
    NS1 :: boolean, NS2 :: boolean, NS3 :: boolean,
    ii :: integer,
    NStepD :: integer;
  description "recherche le minimum dans une direction";
    al := 3.0 : # al et be sont utilisées par plusieurs procédures
    be := -0.4 :
    NStepQ := 5 : # NStepQ est utilisée par plusieurs procédures
    Yk[1] := 1.;
```

```
    NS1 := false;
    NS2 := false;
    NS3 := false;
    NStepD := 0;
    dd[jg] := 0.;
    while (not NS2) do
      for ii from 1 to NPar do
        yy[ii] := xp[ii] + DirIn[jg]·xs[jg, ii];
      end do;
      aa[1] := FCN(NPar, yy);
      NFCN[1] := NFCN[1] + 1;
      Yk[2] := AMin[1] − aa[1];
      if Yk[2] < 0 then
        DirIn[jg] := be·DirIn[jg];
        NStepD := NStepD + 1;
        if (NS1 or NS3)then
          NS2 := true;
        elif (NStepD ≥ NStepQ) then
          Yk[1] := Yk[2];
          NS3 := true;
        end if;
      elif((Yk[2] = 0)and(Yk[1] = 0))then
        NS2 := true;
      else
        AMin[1] := aa[1];
        Yk[1] := Yk[2];
        NS3 := false;
        NStepD := 0;
        NPFN[1] := NFCN[1];
        for ii from 1 to NPar do
          xp[ii] := yy[ii];
        end do;
        dd[jg] := dd[jg] + DirIn[jg];
        DirIn[jg] := al·DirIn[jg];
        NS1 := true;
      end if;
    end do;
  end proc:
> Save('descente')
```

parabole

```
> # pour estimer un minimum local par approximation parabolique à partir de trois points
  parabole := proc(NPar :: integer, xp :: Array(1 ..30, datatype = float), jg :: integer, mieux
      :: boolean, yy :: Array(1 ..30, datatype = float), DirIn :: Array(1 ..31, datatype
      = float), dd :: Array(1 ..31, datatype = float), xs :: Array(1 ..31, 1 ..31, datatype
      = float), AMin :: Array(1 ..1, datatype = float), NFCN :: Array(1 ..1, datatype
      = integer), NPFN :: Array(1 ..1, datatype = integer), Yk :: Array(1 ..2, datatype
      = float), aa :: Array(1 ..1, datatype = float))
  local
    al :: float, be :: float,
```

```
    sss :: float, stj :: float,
    ii :: integer;
  description "estime le minimum par approximation parabolique";
    al := 3.0 : # al et be sont utilisées par plusieurs procédures
    be := -0.4 :
    if mieux then
      sss := (al·al·Yk[1] + Yk[2]) / (al·Yk[1] - Yk[2]);
      stj := 0.5·DirIn[jg]·sss / (al·be);
    else
      sss := (be·be·Yk[1] - Yk[2]) / (be·Yk[1] - Yk[2]);
      stj := 0.5·DirIn[jg]·sss / (be·be);
    end if;
    if stj ≠ 0 then
      for ii from 1 to NPar do
        yy[ii] := xp[ii] + stj·xs[jg, ii];
      end do;
      aa[1] := FCN(NPar, yy);
      NFCN[1] := NFCN[1] + 1;
      if aa[1] < AMin[1] then
        AMin[1] := aa[1];
        NPFN[1] := NFCN[1];
        for ii from 1 to NPar do
          xp[ii] := yy[ii];
        end do;
        dd[jg] := dd[jg] + stj;
      end if;
    end if;
  end proc:
> Save('parabole')
```

errors

```
> # pour calculer une estimation des incertitudes sur les paramètres
  errors := proc(NPar :: integer, xp :: Array(1..30, datatype = float), ig :: integer, NStepC
      :: Array(1..1, datatype = integer), yy :: Array(1..30, datatype = float), invSig2
      :: Array(1..31, datatype = float), pp :: Array(1..31, datatype = float), xs :: Array(1
      ..31, 1..31, datatype = float), AMin :: Array(1..1, datatype = float), GoToFroid
      :: Array(1..1, datatype = boolean)) :: boolean;
  local
    al :: float, be :: float,
    ii, jj, kk :: integer,
    GoToChaud :: boolean,
    ff :: float, ee :: array(1..2, datatype = float);
  description "estimation des incertitudes";
    al := 3.0 : # al et be sont utilisées par plusieurs procédures
    be := -0.4 :
    GoToChaud := false; # sert au redémarrage à chaud
```

```
    for kk from 1 to 2 do # on teste de chaque coté pour estimer  ff = AMin + p2sur2sigma2
      for jj from 1 to NPar do
        yy[jj] := xp[jj] + pp[ig]·xs[ig,jj]·(-1)^kk;
      # on change de coté par le signe ici plutot que celui de pp
      end do;
      ff := FCN(NPar, yy);
      if ff > AMin[1] then # le test semble correct du premier coté
        ee[kk] := ff;
      elif ff < AMin[1] then
      # on a trouvé mieux (par hasard...) ; on recentre et on repart à froid
        for ii from 1 to NPar do
          xp[ii] := yy[ii];
        end do;
        AMin[1] := ff;
        pp[ig] := al·pp[ig];
      # on augmente le pas car il était probablement petit (voisinage du minimum)
        GoToFroid[1] := true; # pour redémarrage à froid
      elif ff = AMin[1] then
      # c'est plat, ou bien le pas est trop petit (le minimum parabolique n'est pas précis)...
        NStepC[1] := NStepC[1] + 1;
        pp[ig] := pp[ig]·(al^((NStepC[1] + 1.)/2.)); # ...on va essayer plus loin
        GoToChaud := true; # il faudra tester à nouveau SVP (redémarrage à chaud)
      else
      # indéterminé ? serait on allé trop loin ?... (normalement on devrait ne jamais passer
      ici)
        pp[ig] := be·pp[ig];
        GoToChaud := true; # il faudra tester à nouveau SVP (redémarrage à chaud)
      end if;
      if (GoToFroid[1] or GoToChaud) then
        break;
      # for kk (s'il faut recentrer ou si c'est du premier coté, inutile de chercher du second car
      il faut recommencer)
      end if;
    end do; # for kk
    if not(GoToFroid[1] or GoToChaud) then
      # terminaison normale (sinon il faudra recommencer)
      invSig2[ig] := (ee[1] + ee[2] - 2·AMin[1])/(pp[ig]·pp[ig]);
      # on estime l'inverse de sigma2 (et non sigma2) car cette estimation peut etre nulle, voir
      négative
    end if;
    return(GoToChaud);
  end proc:
> Save('errors')
```

voisinage

```
> # étude du voisinage pour calculer une estimation des incertitudes sur les paramètres
  voisinage := proc(NPar :: integer, xp :: Array(1..30, datatype = float), NP :: integer, yy
      :: Array(1..30, datatype = float), invSig2 :: Array(1..31, datatype = float), pp
```

```
        :: Array(1..31, datatype = float), xs :: Array(1..31, 1..31, datatype = float), AMin
        :: Array(1..1, datatype = float)) :: boolean;
  local
    ig :: integer, # pour transmettre le numéro du paramètre à la sous-procédure
    NStepC :: Array(1..1, datatype = integer),
    GoToFroid :: Array(1..1, datatype = boolean),
    GoToChaud :: boolean;
  description "calcul des incertitudes";
    NStepC := Array(1..1, datatype = integer, fill = 0) :
      #initialisation des variables globales
    GoToFroid := Array(1..1, datatype = boolean, fill = false) :
      # sert au redémarrage à froid

    for ig from 1 to NP do
      NStepC[1] := 0; # déjà fait pour ig=1 mais à remettre pour chaque ig
      GoToChaud := true; # sert au redémarrage à chaud
      while (GoToChaud or not (invSig2[ig] > 0)) do
        GoToChaud := errors(NPar, xp, ig, NStepC, yy, invSig2, pp, xs, AMin,
       GoToFroid);
        if GoToFroid[1] then
          break; # inutile de continuer ici en cas de redémarrage à froid
        end if;
      end do;
      if not GoToFroid[1] then # on passe en cas de redémarrage à froid
        pp[ig] := 1/sqrt(invSig2[ig]) ; # pp est une estimation de sigma
      end if;
    end do; # for ig
    return (GoToFroid[1]);
  end proc:
> Save('voisinage')
```

incertitudes

```
> # pour calculer une estimation des incertitudes sur les paramètres
  incertitudes := proc(NPar :: integer, xp :: Array(1..30, datatype = float), wt :: Array(1
        ..30, datatype = float), NP :: integer, yy :: Array(1..30, datatype = float), invSig2
        :: Array(1..31, datatype = float), pp :: Array(1..31, datatype = float), xs :: Array(1
        ..31, 1..31, datatype = float), eet :: Array(1..31, 1..31, datatype = float), AMin
        :: Array(1..1, datatype = float), ErrorsOK :: Array(1..1, datatype = boolean))
        :: string;
  local
    ii :: integer, jj :: integer, kk :: integer, ll :: integer, mm :: integer,
    GoToFroid1 :: boolean, # GoToFroid[1]
    NESt :: integer, NEStMx :: integer,
    ppt :: Array(1..31, datatype = float),
    suffisant :: boolean,
    NElist :: string,
    compar :: float,
    Ncompar :: integer;
  description "calcul des incertitudes";
```

```
invSig2[1] := invSig2[NP + 1]; # calcul d'incertitudes sans dérivées (on fait ce qu
  'on peut)
for ii from 1 to NP do
  if invSig2[ii] > 0.00000001 then
  # on prend une marge de sécurité par rapport à 0 pour ne pas surestimer pp (en
  évitant d'etre trop restrictif)
    pp[ii] := 1.00 / sqrt(invSig2[ii]); # pp correspond à sigma
  else
    pp[ii] := 0.01;
  # on initialise en fonction des données accessibles... (normalement on ne passe pas ici
  si le minimum est bien trouvé)
  end if;
  compar := 1.0;
  # il semble prudent de comparer l'estimation aux suggestions wt(kk) de l'utilisateur
  Ncompar := 0;
  for kk from 1 to NPar do
    if (wt[kk] ≠ 0) and (xs[ii, kk] ≠ 0) then
      compar := compar·abs( pp[ii]·xs[ii, kk] / wt[kk] ); # négatifs possibles
      Ncompar := Ncompar + 1;
    end if;
  end do;
  compar := compar^(1. / Ncompar); # on calcule le coefficient de comparaison moyen
  compar := compar^(1 - 4 / (6 + abs(ln(compar)))),
  #modérateur si compar est très différent de 1
  pp[ii] := pp[ii] / compar; # proposition rectifiée
end do;

NEStMx := 5;
for jj from 1 to NEStMx do # on traite le calcul plusieurs fois pour améliorer
  for ii from 1 to NP do
    ppt[ii] := pp[ii]; # on note où on en était, afin de tester s'il est utile de recommencer
  end do;

  GoToFroid1 := true; # pour redémarrage à froid
  while GoToFroid1 do
    GoToFroid1 := voisinage(NPar, xp, NP, yy, invSig2, pp, xs, AMin);
  end do;

  NESt := jj;
  suffisant := true;
  for ii from 1 to NP do
    if (pp[ii] < ppt[ii]·0.95) or (pp[ii] > ppt[ii]·1.05) then
      suffisant := false;
      break;
    end if;
  end do;
```

```
    if suffisant then
      NElist := cat("\rCalcul des incertitudes en ", NESt, " étapes");
      break;
    elif NESt = NEStMx then
      NElist := cat("\rApproximation des incertitudes en ", NESt,
     " étapes (NEStMx atteint)");
    end if;
  end do; # fin de répétition for jj (deux fois suffisent souvent :
     le processus converge rapidement)

  for ll from 1 to NPar do
    for mm from ll to NPar do
      eet[ll, mm] := 0;
      for ii from 1 to NP do
        eet[ll, mm] := eet[ll, mm] + xs[ii, ll]·xs[ii, mm] / invSig2[ii];
      end do;
      eet[ll, mm] := 2·eet[ll, mm]; #  eet = sigma2  pour les paramètres réels
      eet[mm, ll] := eet[ll, mm];
    end do;
  end do; # calculs d'incertitudes sans dérivées
  for ii from 1 to NPar do
    pp[ii] := 0;
    if eet[ii, ii] > 0 then
      pp[ii] := sqrt(eet[ii, ii]);
    end if;
  end do;
  ErrorsOK[1] := true;
  return(NElist);
end proc:
> Save('incertitudes')
```

minimi

```
> minimi := proc(NPar :: integer, xp :: Array(1 ..30, datatype = float), xpN :: Array(1 ..30,
       datatype = string), wtt :: Array(1 ..30, datatype = float), step :: float, epsi :: float,
      iimp :: integer, wErrors :: boolean) :: string;
  local
    al :: float, be :: float,
    NStepQ :: integer,
    yy :: Array(1 ..30, datatype = float),
    wt :: Array(1 ..30, datatype = float),
    DirIn :: Array(1 ..31, datatype = float),
    dd :: Array(1 ..31, datatype = float),
    invSig2 :: Array(1 ..31, datatype = float),
    pp :: Array(1 ..31, datatype = float),
    xs :: Array(1 ..31, 1 ..31, datatype = float),
    eet :: Array(1 ..31, 1 ..31, datatype = float),
    AMin :: Array(1 ..1, datatype = float),
      # pour passer des arguments variables il faut les placer dans une boite...
    NFCN :: Array(1 ..1, datatype = integer),
```

```
    NPFN :: Array(1..1, datatype = integer),
    Yk :: Array(1..2, datatype = float),
    aa :: Array(1..1, datatype = float),
    ErrorsOK :: Array(1..1, datatype = boolean),
    jg :: integer, # pour transmettre le numéro du paramètre aux sous-procédures
    NP :: integer,
    NEq :: integer,
    avr :: float, av :: float, ast :: float, ame :: float, dp :: float, AM :: float,
    NSt :: integer, NP1 :: integer, JM :: integer,
    ii :: integer, jj :: integer, kk :: integer,
    ExitMin :: boolean,
    xn :: float, dir :: float,
    impr :: integer,
    liste :: string, Elist :: string, NElist :: string;
  description "recherche le minimum d'une fonction (généralement un chi2)";
    al := 3.0 : # al et be sont utilisées par plusieurs procédures
    be := -0.4 :
    NStepQ := 5 : # NStepQ est utilisée par plusieurs procédures
    yy := Array(1..30, datatype = float, fill = 0.) : #initialisation des variables globales
    wt := Array(1..30, datatype = float, fill = 0.) :
    DirIn := Array(1..31, datatype = float, fill = 0.) :
    dd := Array(1..31, datatype = float, fill = 0.) :
    invSig2 := Array(1..31, datatype = float, fill = 0.) :
    pp := Array(1..31, datatype = float, fill = 0.) :
    xs := Array(1..31, 1..31, datatype = float, fill = 0.) :
    eet := Array(1..31, 1..31, datatype = float, fill = 0.) :
    AMin := Array(1..1, datatype = float, fill = 1000.) :
    NFCN := Array(1..1, datatype = integer, fill = 0) :
    NPFN := Array(1..1, datatype = integer, fill = 0) :
    Yk := Array(1..2, datatype = float, fill = 0.) :
    aa := Array(1..1, datatype = float, fill = 0.) :
    ErrorsOK := Array(1..1, datatype = boolean, fill = false) :
    liste := "IMI    (minimisation sans dérivées)    MINIMI" :
      # le début sert au formatage final
    liste := cat(liste , "\rNombre de paramètres : ", NPar) :

    for ii from 1 to NPar do
      wt[ii] := wtt[ii];
     DirIn[ii] := wt[ii];
    end do;
    NEq := 0; # Initialisation, premier appel de FCN
    AMin[1] := FCN(NPar, xp);
    NFCN[1] := 1;
    NPFN[1] := NFCN[1];
    aa[1] := AMin[1];
    NSt := 0;
    avr := AMin[1];
    NP := 0; # nombre des paramètres effectivement ajustés
    for ii from 1 to NPar do
      if (wt[ii] ≠ 0) then
        NP := NP + 1;
        for jj from 1 to NPar do
```

```
      xs[NP, jj] := 0.;
    end do;
    invSig2[NP] := 0.;
    xs[NP, ii] := 1.;
    DirIn[NP] := DirIn[ii]·step;
  end if;
end do;

liste := cat(liste , "\rNombre de paramètres effectifs : ", NP ) :
liste := cat(liste , "\rTaille des pas : ", convert(step, string) ) :
liste := cat(liste , "\rPrécision : ", convert(epsi, string) ) :
if wErrors then
  liste := cat(liste , "\rAnalyse des incertitudes pour un chi2") :
end if;
liste := cat(liste , "\r\rValeurs initiales    [ Pas relatif ]") :
for ii from 1 to NPar do
  liste := cat(liste , "\r", xpN[ii], " : ", convert(xp[ii], string) ) :
  liste := cat(liste, "    [ D", xpN[ii], " : ", convert(wt[ii], string), " ]") :
end do;
liste := cat(liste, "\r\rPremier calcul de la quantité minimisée :  Min = ",
   convert(AMin[1], string) ) :

NP1 := NP + 1;
   # pour éviter de le recalculer trop souvent (une fois terminé le calcul de NP)
if iimp = 0 then
  impr := NP1; # on peut faire comme on veut, mais c'est généralement bien ainsi
else
  impr := iimp;
end if;
invSig2[NP1] := 0;
ExitMin := false; # Début du voyage, on y est pour un bout de temps...
Elist := ""; # pour indiquer le diagnostic de fin
while not ExitMin do
  ################################################## minimisation
  for ii from 1 to NPar do
    xs[NP1, ii] := xs[1, ii];
  end do;
  DirIn[NP1] := DirIn[1];
  AM := 0;
  JM := 0;
  for jg from 1 to NP1 do
  ##################################### série de NP1 steps, au travail !
    descente(NPar, xp, jg, yy, DirIn, dd, xs, AMin, NFCN, NPFN, Yk, aa);
   # AMin contient un paramètre variable, transmis par référence
    if (Yk[1] < 0 and Yk[2] < 0) then
      parabole(NPar, xp, jg, false, yy, DirIn, dd, xs, AMin, NFCN, NPFN, Yk, aa);
    elif not (Yk[1] = 0 and Y2k[2] = 0) then
      parabole(NPar, xp, jg, true, yy, DirIn, dd, xs, AMin, NFCN, NPFN, Yk, aa);
    end if;
    NSt := NSt + 1;
    av := avr - AMin[1];
    avr := AMin[1];
```

```
  if jg < 2 then # pas d'amélioration possible au premier essai
    ast := AMin[1];
  elif av > AM then
    AM := av;
    JM := jg;
  end if;
  if (NSt mod impr = 0)then # valeurs intermédiaires
    liste := cat(liste, SorInt(NSt, NPar, xp, xpN, AMin[1]) );
# AMin[1] est transmis par valeur
  end if;
end do;
################################################## la série est terminée, on agite
les mains...
ame := AMin[1] - ast;
if ame ≥ 0 then
  ExitMin := true; # déclenche la fin du while
  Elist := "\rLe minimum n'a pas été amélioré à la dernière étape";
  break; # provoque la sortie directe du while
end if;
if epsi + ame < 0 then
  NEq := -1;
end if;
NEq := NEq + 1;
if NEq ≥ NStepQ then
  ExitMin := true; # déclenche la fin du while
  Elist := cat("\rLe minimum n'a pas été amélioré de ", convert(epsi, string),
 " après chacune des ", NStepQ, " dernières étapes");
  break; # provoque la sortie directe du while
end if;
for kk from 1 to NPar do
  pp[kk] := 0.;
  for ii from 2 to NP1 do
    pp[kk] := pp[kk] + dd[ii]·xs[ii, kk];
  end do;
end do;
dp := 1.;
for ii from 1 to NPar do
  yy[ii] := xp[ii] + pp[ii];
end do;
aa[1] := FCN(NPar, yy);
NFCN[1] := NFCN[1] + 1;
while aa[1] < AMin[1] do
  AMin[1] := aa[1];
  dp := 2·dp;
  for ii from 1 to NPar do
    xp[ii] := yy[ii];
    pp[ii] := 2·pp[ii];
    yy[ii] := xp[ii] + pp[ii];
  end do;
  aa[1] := FCN(NPar, yy);
  NFCN[1] := NFCN[1] + 1;
end do;
```

```
  dp := (aa[1] + ast - 2·AMin[1]) / (2·dp·dp);
  if dp ≤ AM then
    if JM ≤ NP then
      for ii from JM to NP do
        invSig2[ii] := invSig2[ii + 1];
        DirIn[ii] := DirIn[ii + 1];
        for jj from 1 to NPar do
          xs[ii, jj] := xs[ii + 1, jj];
        end do;
      end do;
    end if;
    xn := 0;
    for kk from 1 to NPar do
      xs[1, kk] := pp[kk];
      xn := xn + xs[1, kk]·xs[1, kk];
    end do;
    dir := sqrt(xn);
    DirIn[1] := dir;
    invSig2[NP1] := 2·dp;
    for kk from 1 to NPar do
      xs[1, kk] := xs[1, kk] / dir;
    end do;
  end if;
end do;
  ################################################## while - minimisation
  (retour au début et on recommence...)

ErrorsOK[1] := false;
if (wErrors and (NSt ≥ NPar·NP)) then
  NElist := incertitudes(NPar, xp, wt, NP, yy, invSig2, pp, xs, eet, AMin, ErrorsOK);
end if;

# encore quelques impressions avant d'aller se coucher (la journée a été dure)
liste := cat(liste, "\r\rLa minimisation est terminée") :
liste := cat(liste, Elist) :
liste := cat(liste, "\r\rLa plus faible valeur est :  Min = ", convert(AMin[1], string)) :
liste := cat(liste, "  ( pour l'entrée : ", NPFN[1], " )") :

if not ErrorsOK[1] then
  if wErrors then
    liste := cat(liste,
   "\r\rLe nombre de pas est insuffisant pour calculer les incertitudes") :
  end if;
  liste := cat(liste, SorInt(NSt, NPar, xp, xpN, AMin[1])) :
  # impression des paramètres sans incertitudes
else
  liste := cat(liste, NElist) :
  liste := cat(liste , "\r\rParamètres    [ Déviations standard ]") :
  for ii from 1 to NPar do
    liste := cat(liste , "\r", xpN[ii], " : ", convert(xp[ii], string)) :
```

```
        liste := cat(liste, "     [ D", xpN[ii], " : ", convert(pp[ii], string), " ]") :
        for jj from 1 to ii − 1 do
          if pp[ii]·pp[jj] ≠ 0 then # corrélation
            liste := cat(liste, "\rCov[", xpN[ii], ",") :
            liste := cat(liste, xpN[jj], "] : ") :
            liste := cat(liste, convert(eet[ii, jj], string), "  ;  Cor[") :
            liste := cat(liste, xpN[ii], ",", xpN[jj]) :
            liste := cat(liste, "] : ", convert(eet[ii, jj] / (pp[ii]·pp[jj]), string)) :
          end if;
        end do;
      end do;
    end if;

    liste := cat(liste, "\r\rStatistique de la minimisation :  nombre d'entrées = ", NFCN[1]) :
    liste := cat(liste, "  ;  nombre de pas = ", NSt) :
    liste := cat(MIN, liste) : # formatage final
    # on peut ajouter un appel à FCN avec une option d'impression finale (non utilisé ici)
    # c'est fini, bonsoir...
    return (liste);
  end proc:
> Save('minimi')
>
```