```
# fichier créé avec python 3.9.1

##### AVERTISSEMENT #####
# dans python, la protection des types est encore très primitive...
# je décline toute responsabilité associée à ce défaut regrettable !
#########################

# • 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 python de la procédure de
# minimisation MINIMI (malgré quelques réticences dues aux manques de rigueur
# de ce langage dans ses versions actuelles).
# J M Laffaille - septembre 2023

##################################################################
#############

import numpy as np

# remarque générale : pour forcer Python à transmettre les valeurs de certains
# arguments qu'on souhaite variables, on considère ici les valeurs modifiées
# comme faisant partie des quantités retournées en sortie, charge ensuite à
# l'utilisateur de les réaffecter aux variables correspondantes
```

```
#########################################################################
#############

import myFCN as mf # bidon pour la compilation (pour montrer la structure)

#########################################################################
#############
# impression des valeurs intermédiaires lors de la minimisation

def SorInt(NSt: int, NPar: int, xp, xpN, AMin: float) -> str:
    # xp devrait être un array(30) float et xpN un array(30) str

    # liste des valeurs intermédiaires des paramètres

    Tlist = "\n\nParamètres pour le pas numéro : " + str(NSt)
    Tlist = Tlist + " (Min = " + str(AMin) + ")"
    for ii in range(NPar): # l'indexation commence à 0
        Tlist = Tlist + "\n" + xpN[ii] + " : " + str(xp[ii])
    return Tlist

#########################################################################
#############
# suite de pas pour rechercher un minimum dans une direction

def descente(NPar: int, jg: int, xs,
             # les autres doivent être modifiables :
             xp, yy, DirIn, dd, AMin: float,
             NFCN: int, NPFN: int, Yk, aa: float):
    # xs devrait être un array(31, 31) float
    ### on met à la fin tous les arguments qui doivent être modifiables ###
    # xp et yy devraient être des array(30) float
    # DirIn et dd devraient être des array(31) float
    # Yk devrait être un array(2) float

    # recherche le minimum dans une direction

    al = 3.0 # al et be sont utilisés par plusieurs procédures
    be = -0.4
    NStepQ = 5 # NStepQ est utilisé par plusieurs procédures

    Yk[0] = 1. # l'indexation commence à 0
    NS1 = False
    NS2 = False
    NS3 = False
    NStepD = 0
    dd[jg] = 0. # l'indexation commence à 0
    while (not NS2):
        for ii in range(NPar): # l'indexation commence à 0
            yy[ii] = xp[ii] + DirIn[jg] * xs[jg, ii]
        aa = mf.FCN(NPar, yy, mf.autres)
```

```
        NFCN = NFCN + 1
        Yk[1] = AMin - aa # l'indexation commence à 0
        if (Yk[1] < 0):
            DirIn[jg] = be * DirIn[jg]
            NStepD = NStepD + 1
            if (NS1 or NS3):
                NS2 = True
            elif (NStepD >= NStepQ):
                Yk[0] = Yk[1] # l'indexation commence à 0
                NS3 = True
        elif ((Yk[1] == 0) and (Yk[0] == 0)):
            NS2 = True
        else:
            AMin = aa
            Yk[0] = Yk[1] # l'indexation commence à 0
            NS3 = False
            NStepD = 0
            NPFN = NFCN
            for ii in range(NPar):
                xp[ii] = yy[ii]
            dd[jg] = dd[jg] + DirIn[jg]
            DirIn[jg] = al * DirIn[jg]
            NS1 = True

    # en sortie il faudra affecter les valeurs retournées aux arguments entrés
    return xp, yy, DirIn, dd, AMin, NFCN, NPFN, Yk, aa

##########################################################################
#############
# pour estimer un minimum local par approximation parabolique avec trois points

def parabole(NPar: int, jg: int, mieux: bool, DirIn, xs, Yk,
             # les autres doivent être modifiables
             xp, yy, dd, AMin: float, NFCN: int, NPFN: int, aa: float):
    # DirIn devrait être un array(31) float
    # xs devrait être un array(31, 31) float
    # Yk devrait être un array(2) float
    ### on met à la fin tous les arguments qui doivent être modifiables ###
    # xp et yy devraient être des array(30) float
    # dd devrait être un array(31) float

    # estime le minimum par approximation parabolique

    al = 3.0 # al et be sont utilisés par plusieurs procédures
    be = -0.4

    if mieux:
        sss = (al*al*Yk[0] + Yk[1]) / (al*Yk[0] - Yk[1])
        stj = (0.5*DirIn[jg]*sss) / (al*be)
    else:
        sss = (be*be*Yk[0] - Yk[1]) / (be*Yk[0] - Yk[1])
```

```
        stj = (0.5*DirIn[jg]*sss) / (be*be)
    if (stj != 0):
        for ii in range(NPar):
            yy[ii] = xp[ii] + stj * xs[jg, ii]
        aa = mf.FCN(NPar, yy, mf.autres)
        NFCN = NFCN + 1
        if (aa < AMin):
            AMin = aa
            NPFN = NFCN
            for ii in range(NPar):
                xp[ii] = yy[ii]
            dd[jg] = dd[jg] + stj

    # en sortie il faudra affecter les valeurs retournées aux arguments entrés
    return xp, yy, dd, AMin, NFCN, NPFN, aa

########################################################################
#############
# pour calculer une estimation des incertitudes sur les paramètres

def errors(NPar: int, ig: int, xs,
           # les autres doivent être modifiables
           xp, NStepC: int, yy, invSig2, pp, AMin: float, GoToFroid: bool):
    # xs devrait être un array(31, 31) float
    ### on met à la fin tous les arguments qui doivent être modifiables ###
    # xp et yy devraient être des array(30) float
    # invSig2 et pp devraient être des array(31) float

    # estimation des incertitudes

    al = 3.0 # al et be sont utilisés par plusieurs procédures
    be = -0.4

    ee = [np.nan, np.nan] # il faut initialiser pour dire que c'est un vecteur

    GoToChaud = False # sert au redémarrage à chaud
    for kk in range(2): # test de chaque côté pour estimer ff=Amin+p2sur2sigma2
        for jj in range(NPar):
            yy[jj] = xp[jj] + pp[ig] * xs[ig, jj] * (-1)**(kk+1)
            # on procède selon kk plutôt que de changer le signe de pp
        ff = mf.FCN(NPar, yy, mf.autres)
        if (ff > AMin):
            ee[kk] = ff
        elif (ff < AMin): # on a trouvé mieux (par hasard...)
            for ii in range(NPar): # on recentre et on repart à froid
                xp[ii] = yy[ii]
            AMin = ff
            pp[ig] = al * pp[ig] # on augmente le pas...
                # ...car il était probablement petit (voisinage du minimum)
            GoToFroid = True # pour redémarrage à froid
        elif (ff == AMin): # c'est plat, ou bien le pas est trop petit...
```

```
            NStpC = NStepC + 1 # ...(le minimum parabolique n'est pas précis)
            pp[ig] = pp[ig] * (al**((NStepC+1)/2)) # on va essayer plus loin
            GoToChaud = True # il faudra retester SVP (redémarrage à chaud)
         else:
            # indéterminé ? serait-t-on allé trop loin ?...
            # (normalement on devrait ne jamais passer ici)
            pp[ig] = be * pp[ig]
            GoToChaud = True # il faudra retester SVP (redémarrage à chaud)
         if (GoToFroid or GoToChaud):
            break # for kk (s'il faut recentrer ou si c'est du premier côté...
               # ... inutile de chercher du second car il faut recommencer)
      if not (GoToFroid or GoToChaud):
         # terminaison normale (sinon il faudra recommencer)
         invSig2[ig] = (ee[0]+ee[1]-2*AMin)/(pp[ig]*pp[ig])
         # on estime l'inverse de sigma2 (et non sigma2)...
         # ...car cette estimation peut être nulle, voir négative

      # en sortie il faudra affecter les valeurs retournées aux arguments entrés
      # on y ajoute GoToChaud dont on a besoin ensuite
      return xp, NStepC, yy, invSig2, pp, AMin, GoToFroid, GoToChaud

#####################################################################
#############
# étude du voisinage pour estimer les incertitudes sur les paramètres

def voisinage(NPar: int, NP: int, xs,
              # les autres doivent être modifiables
              xp, yy, invSig2, pp, AMin: float):
      # xs devrait être un array(31, 31) float
      ### on met à la fin tous les arguments qui doivent être modifiables ###
      # xp et yy devraient être des array(30) float
      # invSig2 et pp devraient être des array(31) float

      # calcul des incertitudes

      GoToFroid = False # sert au redémarrage à froid

      for ig in range(NP): # ig sert à transmettre le numéro du paramètre
         NStepC = 0
         GoToChaud = True # sert au redémarrage à chaud
         while (GoToChaud or not (invSig2[ig] > 0)):
            xp,NStepC,yy,invSig2,pp,AMin,GoToFroid,GoToChaud=errors(NPar,ig,xs,
                        # les autres sont modifiables
                        xp, NStepC, yy, invSig2, pp, AMin, GoToFroid)
            if GoToFroid:
               break # inutile de continuer ici si redémarrage à froid
         if not GoToFroid: # on passe en cas de redémarrage à froid
            pp[ig] = 1/np.sqrt(invSig2[ig]) # pp est une estimation de sigma

      # en sortie il faudra affecter les valeurs retournées aux arguments entrés
      # on y ajoute GoToFroid dont on a besoin ensuite
```

```
    return xp,yy,invSig2,pp,AMin,GoToFroid

##################################################################
#############
# pour calculer une estimation des incertitudes sur les paramètres

def incertitudes(NPar: int, wt, NP: int, xs,
            # les autres doivent être modifiables
            xp, yy, invSig2, pp, eet, AMin: float, ErrorsOK: bool):
    # wt devrait être un array(30) float
    # xs devrait être un array(31, 31) float
    ### on met à la fin tous les arguments qui doivent être modifiables ###
    # xp et yy devraient être des array(30) float
    # invSig2 et pp devraient être des array(31) float
    # eet devrait être un array(31, 31) float

    ppt = np.zeros(31) # initialisation

    # calcul des incertitudes

    invSig2[0] = invSig2[NP] # l'indexation commence à 0
    # calcul d'incertitudes sans dérivées (on fait ce qu'on peut)
    for ii in range(NP):
        if (invSig2[ii] > 0.00000001):
            # on prend une marge de sécurité par rapport à 0...
            # ...pour ne pas surestimer pp (en évitant d'être trop restrictif)
            pp[ii] = 1.0/np.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)
        compar = 1.0 # il semble prudent de comparer l'estimation...
                    # ...aux suggestions wt[kk] de l'utilisateur
        Ncompar = 0
        for kk in range(NPar):
            if ((wt[kk] != 0) and (xs[ii, kk] != 0)):
                compar=compar*abs(pp[ii]*xs[ii,kk]/wt[kk]) # négatifs possibles
                Ncompar = Ncompar + 1
        compar = compar ** (1/Ncompar) # coefficient de comparaison moyen
        compar = compar ** (1 - 4/(6+abs(np.log(compar))))
            # modérateur si compar est très différent de 1
        pp[ii] = pp[ii] / compar # proposition rectifiée

    NEStMx = 5
    for jj in range(NEStMx): # on calcule plusieurs fois pour améliorer
        for ii in range(NP):
            ppt[ii] = pp[ii]
        GoToFroid1 = True # pour redémarrage à froid
        while GoToFroid1:
            xp,yy,invSig2,pp,AMin,GoToFroid1 = voisinage(NPar,NP,xs,
                                    # les autres sont modifiables
                                        xp,yy,invSig2,pp,AMin)
```

```
        NESt = jj + 1 # l'indexation commence à 0, les estimations à 1
        suffisant = True
        for ii in range(NP):
            if (pp[ii] < ppt[ii]*0.95) or (pp[ii] > ppt[ii]*1.05):
                suffisant = False
                break
        if suffisant:
            NElist = "\nCalcul des incertitudes en " + str(NESt) + " étapes"
            break
        elif (NESt == NEStMx):
            NElist = "\nApproximation des incertitudes en " + str(NESt)
            NElist = NElist + " étapes (NEStMx atteint)"
        # fin de la répétition for jj
        # (deux fois suffisent souvent : le processus converge rapidement)

    for ll in range(NPar):
        for mm in range(ll, NPar):
            eet[ll, mm] = 0
            for ii in range(NP):
                eet[ll, mm] = eet[ll, mm] + xs[ii, ll]*xs[ii, mm]/invSig2[ii]
            eet[ll, mm] = 2 * eet[ll, mm] # eet=sigma2 pour les paramètres réels
            eet[mm, ll] = eet[ll, mm]
    # fin des calculs d'incertitudes sans dérivées

    for ii in range(NPar):
        pp[ii] = 0
        if (eet[ii, ii] > 0):
            pp[ii] = np.sqrt(eet[ii, ii])
    ErrorsOK = True

    # en sortie il faudra affecter les valeurs retournées aux arguments entrés
    # on y ajoute NElist dont on a besoin ensuite
    return xp, yy, invSig2, pp, eet, AMin, ErrorsOK, NElist

##########################################################################
#############
# minimiseur

def minimi(NPar: int, xp, xpN, wtt,
           step: float, epsi: float, iimp: int, wErrors: bool):
    # xp devrait être un array(30) float
    # xpN devrait être un array(30) str
    # wtt devrait être un array(30) float

    # recherche le minimum d'une fonction (généralement un chi2)

    al = 3.0 # al et be sont utilisés par plusieurs procédures
    be = -0.4
    NStepQ = 5 # NStepQ est utilisée par plusieurs procédures
    yy = np.zeros(30) # initialisation des variables utilisées globalement
    wt = np.zeros(30)
```

```
DirIn = np.zeros(31)
dd = np.zeros(31)
invSig2 = np.zeros(31)
pp = np.zeros(31)
xs = np.zeros((31,31))
eet = np.zeros((31,31))
AMin = 1000.
NFCN = 0
NPFN = 0
Yk = np.zeros(2)
aa = 0.
ErrorsOK = False
liste = "MINIMI   (minimisation sans dérivées)   MINIMI"
                # le début sert au formatage final
liste = liste + "\nNombre de paramètres : " + str(NPar)

for ii in range(NPar):
    wt[ii] = wtt[ii]
    DirIn[ii] = wt[ii]
NEq = 0 # Initialisation, premier appel de FCN
AMin = mf.FCN(NPar, xp, mf.autres)
NFCN = 1
NPFN = NFCN
aa = AMin
NSt = 0
avr = AMin
NP = 0 # nombre des paramètres effectivement ajustés
for ii in range(NPar):
    if (wt[ii] != 0):
        NP = NP + 1
        for jj in range(NPar):
            xs[NP-1, jj] = 0. # l'indexation commence à 0
        invSig2[NP-1] = 0. # NP-1 est l'index qui représente NP
        xs[NP-1, ii] = 1.
        DirIn[NP-1] = DirIn[ii] * step

liste = liste + "\nNombre de paramètres effectifs : " + str(NP)
liste = liste + "\nTaille des pas : " + str(step)
liste = liste + "\nPrécision : " + str(epsi)
if wErrors:
    liste = liste + "\nAnalyse des incertitudes pour un chi2"
liste = liste + "\n\nValeurs initiales   [ Pas relatif ]"
for ii in range(NPar):
    liste = liste + "\n" + xpN[ii] + " : " + str(xp[ii])
    liste = liste + "   [ D" + xpN[ii] + " : " + str(wt[ii]) + " ]"
liste = liste + "\n\nPremier calcul de la quantité minimisée :"
liste = liste + "  Min = " + str(AMin)

NP1 = NP + 1 # pour éviter de le recalculer trop souvent...
             # ...(une fois terminé le calcul de NP)
if (iimp == 0):
```

```
        impr = NP1 # on peut faire comme on veut, mais c'est généralement bien
    else:
        impr = iimp
    invSig2[NP1-1] = 0 # l'index qui représente NP1 est NP1-1
    ExitMin = False # Début du voyage, on y est pour un bout de temps...
    Elist = "" # pour indiquer le diagnostic de fin
    while not ExitMin: ###################################### minimisation
####
        for ii in range(NPar):
            xs[NP1-1, ii] = xs[0, ii] # l'indexation commence à 0
        DirIn[NP1-1] = DirIn[0] # l'index qui représente NP1 est NP1-1
        AM = 0.
        JM = 0
        for jg in range(NP1): ############### série de NP1 steps, au travail !
            xp,yy,DirIn,dd,AMin,NFCN,NPFN,Yk,aa = descente(NPar,jg,xs,
                                    # les autres sont modifiables
                                xp,yy,DirIn,dd,AMin,NFCN,NPFN,Yk,aa)
            if ((Yk[0] < 0) and (Yk[1] < 0)):
                xp,yy,dd,AMin,NFCN,NPFN,aa = parabole(NPar,jg,False,DirIn,xs,Yk,
                                # les autres doivent être modifiables
                                        xp,yy,dd,AMin,NFCN,NPFN,aa)
            elif not ((Yk[0] == 0) and (Yk[1] == 0)):
                xp,yy,dd,AMin,NFCN,NPFN,aa = parabole(NPar,jg,True,DirIn,xs,Yk,
                                # les autres doivent être modifiables
                                        xp,yy,dd,AMin,NFCN,NPFN,aa)
            NSt = NSt + 1
            av = avr - AMin
            avr = AMin
            if (jg < 2): # pas d'amélioration possible au premier essai
                ast = AMin
            elif (av > AM):
                AM = av
                JM = jg
            if ((NSt % impr) == 0): # valeurs intermédiaires
                liste = liste + SorInt(NSt, NPar, xp, xpN, AMin)
        ######################### la série est terminée, on agite les mains...
        ame = AMin - ast
        if (ame >= 0):
            ExitMin = True # déclenche la fin du while
            Elist = "\nLe minimum n'a pas été amélioré à la dernière étape"
            break # provoque la sortie directe du while
        if ((epsi + ame) < 0):
            NEq = -1 # sinon repartir de 0 et sauter la ligne suivante
        NEq = NEq + 1
        if (NEq >= NStepQ):
            ExitMin = True # déclenche la fin du while
            Elist = "\nLe minimum n'a pas été amélioré de " + str(epsi)
            Elist = Elist + " après chacune des " + str(NStepQ)
            Elist = Elist + " dernières étapes"
            break # provoque la sortie directe du while
        for kk in range(NPar):
```

```
        pp[kk] = 0
        for ii in range(1, NP1): # l'indexation commence à 0
            pp[kk] = pp[kk] + dd[ii] * xs[ii, kk]
    dp = 1.0
    for ii in range(NPar):
        yy[ii] = xp[ii] + pp[ii]
    aa = mf.FCN(NPar, yy, mf.autres)
    NFCN = NFCN + 1
    while (aa < AMin):
        AMin = aa
        dp = 2 * dp
        for ii in range(NPar):
            xp[ii] = yy[ii]
            pp[ii] = 2 * pp[ii]
            yy[ii] = xp[ii] + pp[ii]
        aa = mf.FCN(NPar, yy, mf.autres)
        NFCN = NFCN + 1
    dp = (aa + ast - 2 * AMin)/(2 * dp * dp)
    if (dp <= AM):
        if (JM <= NP-1):
            for ii in range(JM, NP): # l'indexation commence à 0...
                # JM est une valeur de jg dans range(NP1)
                # donc JM peut être NP=NP1-1, mais il représente alors NP1
                # on veut s'arrêter au JM=NP-1 qui représente NP (inclus)
                invSig2[ii] = invSig2[ii+1]
                DirIn[ii] = DirIn[ii+1]
                for jj in range(NPar):
                    xs[ii, jj] = xs[ii+1, jj]
        xn = 0.0
        for kk in range(NPar):
            xs[0, kk] = pp[kk] # l'indexation commence à 0
            xn = xn + xs[0, kk] * xs[0, kk]
        dir = np.sqrt(xn)
        DirIn[0] = dir # l'indexation commence à 0
        invSig2[NP1-1] = 2 * dp # l'indexation commence à 0
        for kk in range(NPar):
            xs[0, kk] = xs[0, kk]/dir # l'indexation commence à 0
################## while - minimisation (retour au début et on recommence...)

ErrorsOK = False
if (wErrors and (NSt >= NPar*NP)):
    xp,yy,invSig2,pp,eet,AMin,ErrorsOK,NElist = incertitudes(NPar,wt,NP,xs,
                                # les autres doivent être modifiables
                                xp,yy,invSig2,pp,eet,AMin,ErrorsOK)

# encore quelques impressions avant d'aller se coucher (la journée fut dure)
liste = liste + "\n\nLa minimisation est terminée"
liste = liste + Elist
liste = liste + "\n\nLa plus faible valeur est : Min = " + str(AMin)
liste = liste + " ( pour l'entrée : " + str(NPFN) + " )"
```

```
    if not ErrorsOK:
        if wErrors:
            liste = liste + "\n\nLe nombre de pas est insuffisant"
            liste = liste + " pour calculer les incertitudes"
        liste = liste + Sorint(NSt, NPar, xp, xpN, AMin)
                # impression des paramètres sans incertitudes
    else:
        liste = liste + NElist
        liste = liste + "\n\nParamètres  [ Déviations standard ]"
        for ii in range(NPar):
            liste = liste + "\n" + xpN[ii] + " : " + str(xp[ii])
            liste = liste +"   [ D" + xpN[ii] + " : " + str(pp[ii]) + " ]"
            for jj in range(ii): # on s'arrête à ii-1 et ii est un index
                if (pp[ii] * pp[jj] != 0): # corrélation
                    liste = liste + "\nCov[" + xpN[ii] + ","
                    liste = liste + xpN[jj] + "] : "
                    liste = liste + str(eet[ii,jj]) + " ; Cor["
                    liste = liste + xpN[ii] + "," + xpN[jj]
                    liste = liste + "] : " + str(eet[ii,jj]/(pp[ii]*pp[jj]))

    liste = liste + "\n\nStatistique de la minimisation : "
    liste = liste + "nombre d'entrées = " + str(NFCN)
    liste = liste + " ; nombre de pas = " + str(NSt)

    # on peut ajouter un appel à FCN avec une option d'impression finale...
    # ...(non utiilsé ici)
    # c'est fini, bonsoir...

    return liste

##########################################################################
#############
```