R code for calculating Orientation Distribution Functions (ODFs) and example runs and plots

Alan E. Berger December 30, 2023

in https://github.com/AlanBerger/R-code-for-calculating-orientation-distribution-functions-for-rodlike-particles

Introduction

ODF is the abbreviation for ***orientation distribution function***

Below, we first give R code for calculating ODFs. For use as in

Judith Herzfeld, Alan E. Berger and John W. Wingate, A highly convergent algorithm for computing the orientation distribution functions of rodlike particles, Macromolecules 1984, v17, 1718-1723, https://pubs.acs.org/doi/pdf/10.1021/ma00139a014 **[HBW 1984]** Note web links in pdf files as displayed in GitHub do not work, but will work if you download the pdf file to your computer.

This R code should not need to be modified; one should be able to do desired runs just by varying the arguments in the function calls, including the function W(gamma) in the particle interaction term in the free energy function.

Then, example runs of this code are given, including plots. Several cases done in [HBW 1984] (via Fortran code) are repeated here (for the "hard core reference system" ($W(\gamma) = \sin(\gamma)$), giving results consistent with those in Tables I and II in [HBW 1984].

Results are also given displaying an ODF that has a shape different from the planar and axial solutions in [HBW 1984], arising from a free energy function other than the hard core reference system. Example cases that do not satisfy the conditions for the free energy (of the continuous, i.e., not discretized, free energy) to decrease at each iterative step (unless one was at a fixed point of the iteration) are shown to have an increase in the calculated (discretized) free energy at some steps (a proof that the iteration in [HBW 1984] decreases the free energy at each step (for the continuous case) under natural conditions on the function W in the free energy is given in: Alan E. Berger, manuscript in preparation).

If one wishes to just look at the computational results, scroll down to the plots and their Figure Legends below.

Note also:

R. F. Kayser Jr. and H. J. Raveche', Bifurcation in Onsager's model of the isotropic-nematic transition, Physical Review A 1978, v17, 2067-2072, https://journals.aps.org/pra/abstract/10.1103/PhysRevA.17.2067 **[KR 1978]**

A. E. Berger, Nonlinear Analysis, Analysis of a constrained minimization problem modeling the orientation distribution of rod-like particles, Theory, Methods & Applications 1987, v11, 719-731, https://www.sciencedirect.com/science/article/abs/pii/0362546X87900381 **[B 1987]**

The following section of code provides (and "compiles") the R function **compute_approx_ODF** which does some preliminary calculations and returns the R function **calculate_ODF**. The results of the preliminary calculations are available to the calculate_ODF function, which implements the iterative method in [HBW 1984] for the choices of the arguments of compute_approx_ODF and calculate_ODF.

The arguments of compute_approx_ODF are (some of these employ names corresponding to the notation in [HBW 1984], see also the detailed comments within these two functions):

N.theta.intervals the number of equally spaced θ intervals in [0, π]

N.phi.intervals the number of equally spaced ϕ intervals in [0, 2π] used for the numerical integration over ϕ in the particle interaction term in the free energy (see equation (22b) in [HBW 1984] where here G will be W)

W the function of γ in the particle interaction term in the free energy function F

The arguments of calculate_ODF are:

initial.f a vector of initial values at the grid points within $(0, \pi)$ at which ODF values are calculated

B the constant coefficient of the particle interaction term in the free energy

tau.conv the convergence criterion constant

max.num.iter the maximum number of iteration steps allowed

min.iterations the number of iteration steps to be carried out before doing the convergence test at each step

Note that once compute_approx_ODF has been called, calculate_ODF can be called multiple times with different values for one or more of its arguments. If one wants to do calculations with different values for one or more of N.theta.intervals, N.phi.intervals, W, then one needs to do another run of compute_approx_ODF with the desired values of its arguments.

The iterations carried out here determine a vector f of approximate values for an ODF at the θ grid points interior to $(0, \pi)$ specified by N.theta.intervals (f is intended to approximate an ODF that is a local minimum of the free energy). Integrations are carried out using trapezoidal numerical integration. Since here there is a $\sin(\theta)$ factor in the integrals over θ, the value of the ODF at the points $\theta = 0$ and $\theta = \pi$ has no effect. Iterations for the values of f are thus carried out only at the θ grid points interior to $(0, \pi)$. Values for the ODF at 0 and π for use in plots are obtained from f by extrapolation.

The quantities returned by a run of calculate_ODF are in an R ***list*** which is called **results** in the R code below (examples of obtaining desired values from **results** are given below). **results** contains values for (notation in [HBW 1984] is used here) :

f.for.plotting The vector of values V of the calculated ODF at the grid points in the interior of $(0, \pi)$ with extrapolated values at $\theta = 0$ and $\theta = \pi$ appended at the left and right ends of V, respectively

free.energy.for.each.iteration The vector containing the calculated free energy value at each iteration

vector.of.values A vector containing values of interest (see below)

names.for.vector.of.values The names of the entries of vector.of.values

The vector.of.values contains the following entries (values for the following):

B The positive constant in the free energy function

N.theta.intervals The number of θ intervals used for integration over θ in $[0, \pi]$

f0 The extrapolated value for f at $\theta = 0$

L The estimated value of the Lipschitz constant for the iterative mapping around the value of the ODF returned by calculate_ODF (see [HBW 1984])

pi/4 - <<W(gamma)>> See the notation in [HBW 1984] for the definition of this value (for the ODF returned by calculate_ODF)

<ln(f(theta))> See the notation in [HBW 1984] for the definition of this value

<P2(cos(theta))> See the notation in [HBW 1984] for the definition of this value

<P4(cos(theta))> See the notation in [HBW 1984] for the definition of this value

free.energy The value of the free energy for the ODF returned by calculate_ODF

num_iterations The number of iterations carried out by calculate_ODF (if this equals max.num.iter then a warning is printed)

num_iter_F_did_NOT_decrease The number of iterations where the free energy did not decrease from one iteration of the ODF to the next

If one wants a copy of R code in this file, one should copy it from the Rmd file (which is a plain text file). To save, as a text file, a copy of an Rmd file that is in a repository on GitHub, one can: Open the Rmd file in GitHub by clicking on its name in the list of files at the left side of the screen when the desired repository is opened. Then (for a Windows computer) right click on the "raw" button (toward the upper right of the window displaying the Rmd file) and then choose save link as, or in general, click on the symbol with a downward pointing arrow that represents "download", that is to the right of the raw button.

In an Rmd file, a line with three left single quotes, then a single space, then {r} at the beginning of the line starts a section of R code that is executed when an Rmd (R markdown) file is run through R's ***knitr***. A section of R code is ended by a line containing three left single quotes at the beginning of the line. R code that is to be displayed as code but not to be executed starts with three left single quotes at the beginning of its own separate line, and ends with a line containing three left single quotes at the beginning of the line. These beginning and ending lines do not appear in the processed file.

One can examine accuracy by comparing the values of quantities of interest resulting from using a succession of doubling the number of intervals used for the integrations over ϕ in [0, 2π] and, particularly, θ in [0, π], and by decreasing the convergence criterion setting tau.conv

To view sample runs and their output and plots and commentary, skip past the following section of R code to the section titled Sample runs of compute_approx_ODF and calculate_ODF

*********The R code for compute_approx_ODF and calculate_ODF follows**********

```
## ***  START OF **The R code for compute_approx_ODF and calculate_ODF *****

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

compute_approx_ODF <- function(N.theta.intervals, N.phi.intervals, W) {
# December 8, 2023  Alan E. Berger
print("run of   compute_approx_ODF   December 8, 2023  version")

# Use the iterative method indicated by the calculus of variations, discretized
# as in J. Herzfeld, A. E. Berger, J. W. Wingate, Macromolecules 1984, v17,
# 1718-1723, [HBW 1984], to calculate approximations to Orientation Distribution
# Functions

# abbreviation used below: ODF is orientation distribution function

# The code below is generally using the notation and setup in the
# Judith Herzfeld, Alan E. Berger and John W. Wingate paper,
# except, for simplicity, doing calculations on [0, pi] rather than taking
# advantage of symmetry about pi/2 (current computing power relative
# to that in 1984 means that for these calculations one can favor simpler code
# over running speed). Also, rather than use Richardson extrapolation as done in
# [HBW 1984] to obtain increased accuracy for quantities of interest, here the
# intent is to simply use larger numbers of grid points to obtain desired accuracy.

# Use the  lexical scoping  and  function closure  properties of the
# R programming language to do once various setup calculations, and then have the
# results available. And define and return the function   calculate_ODF   that for
# a given value of the constant B in the free energy function F, and initial
# vector of values initial.f at the  grid points {Pj = j * pi / N.theta.intervals},
```

```
# j = 1, 2, ..., N.theta.intervals - 1),
# calculates an approximate ODF f (a vector of values {fj} approximating the values
# of an ODF at the grid points {Pj}).
# f generally should approximate the values of a local minimum of F at
# the grid points {Pj}. Convergence of the iteration is tested as in [HBW 1984],
# using as the convergence bound   tau.conv   (an argument of the
# function  calculate_ODF  defined below).

# Note also: R. F. Kayser Jr. and H. J. Raveche', Physical Review A 1978, v17,
# 2067-2072 [KR 1978],
# and A. E. Berger, Nonlinear Analysis, Theory, Methods & Applications 1987,
# v11, 719-731 [B 1987].

# For the hard core reference system, the function W in the free energy function
# (W is a function whose argument is denoted by gamma) equals sin(gamma)
# gamma will be equal
#     acos(sin(theta1)*sin(theta2)*cos(phi) + cos(theta1)*cos(theta2))
# (see for example [KR 1978], [B 1987])

# Trapezoidal numerical integration is used to approximate the value of
# integrals, and the presence of sin(theta) factors in the free energy function
# results in the values of f at the points corresponding to theta = 0 and
# theta = pi having no effect on the value of the discretized free energy.
# Note f (the vector of values defined at the grid points Pj) has a natural
# extension to a function defined for all theta in [0, pi]: see equation (2.8)
# in [B 1987]; and for the hard core reference system, W(gamma) equal sin(gamma),
# by Theorem 1.2 or Lemma 3.1 in [B 1987] any ODF that is a
# local minimum of the actual (not discretized) free energy function F will
# have (at least) 2 continuous derivatives
# on [0, pi] and its first derivative will be 0 at theta = 0, pi/2 and pi

# Note for the free energy functions being considered here, local minima will be
# symmetric about pi/2 so one could reduce calculations to be over [0, pi/2]
# For conceptual simplicity in the programming we choose not to do that here.
# In a number of places, we choose simpler code over faster running code.

# An approximation for the value f(0) is obtained as the value at theta = 0 of
# the quadratic function that has the values f1, f2, f3 at the grid points P1,
# P2, P3, respectively (as done in [HBW 1984]). For the free energy
# functions F considered here, any ODF that is a local minimum of F satisfies
# f(pi - theta) equals f(theta). Hence we take f(0) as an approximation to the
# value of f(pi). While values of an approximate (discretized) ODF at
# theta = 0 and theta = pi do not matter for calculating integrals involving
# a sin(theta) factor via the trapezoidal formula, these values are used in
# doing plots.


#23456789 123456789 123456789 123456789 123456789 123456789 123456789 1234567890
# try to keep to max of 80 characters per line


# Calculate various vectors and the matrix for the discretized version of the
# integral operator in the free energy. These only need to be calculated once,
```

```
# and the setup here ensures that the values are available and preserved between
# calls to   calculate_ODF

delta.theta <- pi / N.theta.intervals  # length of theta subintervals

# The Pj points (located in the vector theta.vec) are interior to [0, pi])
# These are the theta locations for the values in discrete ODF vectors.
# The sin(theta) factor in theta integrals results in values at theta
# equal 0 and theta equal pi having no effect on trapezoidal approximation
# values

num.theta.points <- N.theta.intervals - 1
theta.indices <- 1:num.theta.points
theta.vec <- theta.indices * delta.theta  # theta values for numerical
#                                           integration

# Calculating values of the kernel of the integral operator in F
# requires integrals over [0, 2*pi] which are calculated using
# trapezoidal numerical integration. These integrals do involve values
# at phi = 0 and phi = 2*pi since there is no sin(phi) factor in the
# integrals over phi

delta.phi <- 2*pi / N.phi.intervals  # length of phi subintervals

# phi example   4 phi intervals   N.phi.intervals equals 4

# phi points     c(0,  1,    2,              3,              4) * delta.phi
#              2*pi equals N.phi.intervals * delta.phi
# phi indices      1   2     3               4               5
#
# point location   0   1*delta.phi ... (N.phi.intervals - 1)*delta.phi
#                                                N.phi.intervals*delta.phi

num.phi.points <- N.phi.intervals + 1
phi.indices <- 1:num.phi.points  # phi.indices[2:(num.phi.points - 1)] are
#           the phi indices for phi values in the interior of [0, 2*pi]
#           the interior phi points have trapezoidal weight 1
#           the phi points at 0 and 2*pi have trapezoidal weight 1/2
phi.vec <- (phi.indices - 1) * delta.phi  # locations of the phi points

sin.theta.vec <- sin(theta.vec)
cos.theta.vec <- cos(theta.vec)

# sin.phi.vec <- sin(phi.vec)  # not needed
cos.phi.vec <- cos(phi.vec)

# for theta1 defined as theta.vec[i] and
# theta2 defined as theta.vec[j], i and j running
# from 1 through num.theta.points, we want to have the matrix of values K[i,j]
# defined by
# (1/(2*pi) integral from phi = 0 to 2*pi of W(gamma)
# where gamma = acos(sin(theta1)*sin(theta2)*cos(phi) + cos(theta1)*cos(theta2))
# This integral will be calculated using trapezoidal numerical integration
```

```
# over phi

K <- matrix(0, nrow = num.theta.points, ncol = num.theta.points) # initialize K

# K will be symmetric, but it will be conceptually simpler just to calculate all
# of the entries (this is only done once for a given run of B values).
# Having the full matrix K, and K being symmetric, means we can use column j
# of K for row j of K (and column entries have consecutive storage locations).

for (j in theta.indices) {
   for (i in theta.indices) {

      acos.arg.vec <- sin.theta.vec[i] * sin.theta.vec[j] * cos.phi.vec +
                           cos.theta.vec[i] * cos.theta.vec[j]

#          need to check that each entry of acos.arg.vec is in [0,1]
#          this could fail to be the case due to finite precision
#          for example, in one case got a couple values v equal to
#          1.0000000000000002
#          which would result in acos(v) giving not a number (NaN)

      wm1 <- which(acos.arg.vec < -1)
      if(length(wm1) > 0) acos.arg.vec[wm1] <-  -1

      w1 <- which(acos.arg.vec > 1)
      if(length(w1) > 0)  acos.arg.vec[w1] <-   1

      trapezoid.values <- W(acos(acos.arg.vec))  # W needs to handle vectors
#            (return a vector of values when called with a vector of values)

      trap.sum <- (trapezoid.values[1] + trapezoid.values[num.phi.points]) / 2 +
                        sum(trapezoid.values[2:(num.phi.points - 1)])

      K[i, j] <- trap.sum * delta.phi / (2 * pi)
#             the denominator two * pi is in the definition of K
   }
}

# Now have finished the preliminary calculations,
# next define the function  calculate_ODF  which calculates
# an ODF, given various required input values.
# The quantities that were calculated above are available
# to calculate_ODF since compute_approx_ODF returns
# calculate_ODF in the form of a function closure
# (this is a property of the R programming language)


#23456789 123456789 123456789 123456789 123456789 123456789 123456789 1234567890
##############################################################################


calculate_ODF <- function(initial.f, B, tau.conv, max.num.iter, min.iterations) {
# December 8, 2023   Alan E. Berger
```

```
print("run of   calculate_ODF    December 8, 2023  version")


# calculate what should be an approximation f to a local minimum of the
# free energy F with parameter value B, given the vector of initial
# values initial.f (corresponding to values of an ODF at the Pj points).
# The vector f gives the (approximate) values at the Pj points.
#
# tau.conv is the convergence bound stopping condition as in [HBW 1984]
# max.num.iter is the maximum number of iterations allowed - a warning message
# will be printed if the iteration didn't converge (didn't satisfy
# the stopping condition within max.num.iter iterations)
# min.iterations    require having done  min.iterations  iterations before
#                   start testing for convergence

# use the iterations suggested by the calculus of variations - see [HBW 1984]

# W(gamma) is the function used in calculating the matrix K which is the kernel
# of the discrete integral operator in the discrete form of F (K is obtained in
# compute_approx_ODF  above).
# For the hard core reference system, W(gamma) = sin(gamma)

# gamma will be equal
#     acos(sin(theta1)*sin(theta2)*cos(phi) + cos(theta1)*cos(theta2))

# use the various calculated values preserved in the
# function closure created by  compute_approx_ODF

# will check on whether each iteration decreases
# the (discretized / numerically calculated) free energy.

# When W satisfies appropriate conditions, it has been proven (AEB) that
# each iteration, in the case of continuous ODFs,
# will decrease the free energy
# unless one was at a fixed point of the iteration
# (manuscript giving the proof is in preparation).


# Return a list of desired quantities including the discrete converged
# ODF f (so can plot it if desired) and various values associated with it,
# including f(0) (calculated using the quadratic polynomial
# passing through f[1], f[2], f[3]),
# an estimate of the local Lipschitz constant L,
# and calculated values for  pi/4 - <<W(gamma)>>  (see [HBW 1984] and the
# calculations below for how this and similar quantities are defined);
# <ln(f(theta))>, <P2(cos(theta))>, <P4(cos(theta))> where P2 and P4 here
# are the second and fourth Legendre polynomial.
# Put the scalar quantities into a vector that will be returned in the list.

# Also return the number of iterations it took to converge, and the number
# of iterative steps for which the calculated free energy failed to decrease

# first convert initial.f to be a (discrete) ODF that is positive
```

```
# (set any values <= 0 to be slightly above 0, and then scale (normalize) so
# the integral over theta in [0, pi] of  (1/2) * f(theta) * sin(theta) = 1
# as calculated using trapezoidal numerical integration)

#23456789 123456789 123456789 123456789 123456789 123456789 123456789 1234567890

fn <- pmax(initial.f, 0.0001) # fn is now > 0
# having the initial fn > 0 avoids any issue with ln(f) since by the nature of
# the iteration, all successive iterates will also be > 0

theta.integral.of.fn <- sum(fn * sin.theta.vec) * delta.theta / 2
# this is trapezoidal integration for
# (1/2) integral over theta in [0, pi] of fn * sin(theta)

fn <- fn / theta.integral.of.fn  #  fn is now normalized
# so (1/2) integral over theta in [0, pi] of fn(theta) sin(theta) equals 1
# (as calculated by the trapezoidal formula)


######### do the first iteration "by hand" since need three
######### successive approximate f values
######### to estimate the approximation error - see [HBW 1984]

# use the iteration suggested by the calculus of variations - see [HBW 1984]

# Also, calculate the value of the free energy F for fn

free.energy.for.each.iteration <- numeric(0)
# successively append each calculated F, use for
# plotting F vs. interation number

fnp1 <- numeric(num.theta.points)
# initialize "container" vector for the next iterate f^{n+1}

vector.for.free.energy <- numeric(num.theta.points)  # container for vector
#                           used for calculating F(fn) and for getting fnp1


for (i in theta.indices) {  # calculate each entry of fnp1

#  use trapezoidal formula to approximate
#  integral over theta2 in [0, pi] of
#                                      K(theta1, theta2) fn(theta2) sin(theta2)
   Ki <- K[, i]  # since K[i,j] is symmetric, this equals the vector
#                  equal to the ith row of K
#                  (K[i, ] whose entries corr. to values of theta2)
   trapez.sum <- sum(Ki * fn * sin.theta.vec)
   value.of.integral.over.theta2 <- trapez.sum * delta.theta
   value.for.free.energy <-  B * value.of.integral.over.theta2 / 2
   vector.for.free.energy[i] <- value.for.free.energy  # for getting F for fn
   fnp1[i] <- exp(-value.for.free.energy)  # fnp1 is not yet normalized
}
```

```
# Deal with the limits of finite precision in computations:
# if the argument v of the exponential function is too large
# (this is above 709 and below 710 on my computer), exp(v) will, in R,
# return Inf (a reserved word in R).
# This would lead to error messages from downstream calculations.
# fnp1 is equal exp(-vector.for.free.energy); note the minus sign
   maxv <- max(abs(vector.for.free.energy))
   if(maxv > 700) cat("an argument of exp will be ", maxv, "  at iteration ", 1)
   if(maxv > 700) {
      stop("have situation of abs of argument of exp > 700; ending execution")
   }


#  now normalize fnp1
   theta.integral.of.fnp1 <- sum(fnp1 * sin.theta.vec) * delta.theta / 2
   fnp1 <- fnp1 / theta.integral.of.fnp1  #  fnp1 is now normalized

#  finish calculation of integral operator K contribution to the discrete
#  free energy (for fn)
   integral.operator.K.part.of.free.energy <-
        sum(fn * vector.for.free.energy * sin.theta.vec) * delta.theta / 4

#  calculate the contribution to the discrete free energy from
#  the natural log term (for fn)

   log.part.for.F <- sum(fn * log(fn) * sin.theta.vec) * delta.theta / 2

   free.energy.for.fn <-
               log.part.for.F + integral.operator.K.part.of.free.energy
   free.energy.for.each.iteration  <-
               c(free.energy.for.each.iteration, free.energy.for.fn)

# end of first iteration (done individually)


# Now can do successive iterations, estimating the Lipschitz constant L for the
# iteration and the approximation error (relative to max(fn)) as in [HBW 1984].
# Use tau.conv as the convergence criterion on the approximation error.

num_iter_F_did.NOT.decrease <- 0

for (iter in 2:max.num.iter) {
   iteration.num <- iter

   fnm1 <- fn  # fnm1 is the variable name for f^{n-1}
#                when iter is 2, fn is from the calculations above
   free.energy.for.fnm1 <- free.energy.for.fn
#       when iter is 2, free.energy.for.fn   is from the calculations above
   fn   <- fnp1  # when iter is 2, fnp1 is from the calculations above

#  calculate next iteration value fnp1 using fn, and then use fnm1, fn, fnp1
#  to estimate the error between fnp1 and what the iteration
#  supposedly converges to
```

```
# calculate the free energy for fn (while calculating fnp1) and test if
# it is < the free energy for fnm1


#  from previous calculations, fnp1 is an existing vector of the correct
#  length so can use it as a container vector for the next iteration values

#23456789 123456789 123456789 123456789 123456789 123456789 123456789 1234567890

   for (i in theta.indices) {  # calculate each entry of fnp1

#  use trapezoidal formula to approximate
#  integral over theta2 in [0, pi] of
#                                      K(theta1, theta2) fn(theta2) sin(theta2)
   Ki <- K[, i]  # since K[i,j] is symmetric, this equals the vector
#                  equal to the ith row of K
#                  (K[i, ] whose entries corr. to values of theta2)
   trapez.sum <- sum(Ki * fn * sin.theta.vec)
   value.of.integral.over.theta2 <- trapez.sum * delta.theta
   value.for.free.energy <-  B * value.of.integral.over.theta2 / 2
   vector.for.free.energy[i] <- value.for.free.energy  # for fn
   fnp1[i] <- exp(-value.for.free.energy)  # not yet normalized

   }   # end of loop to calculate each entry of fnp1

# Deal with the limits of finite precision in computations:
# if the argument v of the exponential function is too large
# (this is above 709 and below 710 on my computer), exp(v) will, in R,
# return Inf (a reserved word in R).
# This would lead to error messages from downstream calculations.
# fnp1 is equal exp(-vector.for.free.energy); note the minus sign
   maxv <- max(abs(vector.for.free.energy))
   if(maxv > 700) cat("an argument of exp will be ", maxv, "  at iteration ",
                                 iteration.num)
   if(maxv > 700) {
      stop("have situation of abs of argument of exp > 700; ending execution")
   }


#  now normalize fnp1
   theta.integral.of.fnp1 <- sum(fnp1 * sin.theta.vec) * delta.theta / 2
   fnp1 <- fnp1 / theta.integral.of.fnp1  #  fnp1 is now normalized

#  finish calculation of integral operator K contribution to the discrete
#  free energy (for fn)
   integral.operator.K.part.of.free.energy <-
        sum(fn * vector.for.free.energy * sin.theta.vec) * delta.theta / 4

#  calculate the contribution to the discrete free energy from
#  the natural log term (for fn)

   log.part.for.F <- sum(fn * log(fn) * sin.theta.vec) * delta.theta / 2
```

```
   free.energy.for.fn <-
               log.part.for.F + integral.operator.K.part.of.free.energy
   free.energy.for.each.iteration  <-
               c(free.energy.for.each.iteration, free.energy.for.fn)

# don't flag if test for increase in free energy is close to
# rounding error
# flag only the first 12 non-decreases in F
   if (free.energy.for.fn >= free.energy.for.fnm1 + 1.0e-12) {
      if(num_iter_F_did.NOT.decrease <= 11) {
        cat("iter = ", iter,
            "  F did not decrease   only printing for first 12 instances", "\n")
      }
            num_iter_F_did.NOT.decrease <- num_iter_F_did.NOT.decrease + 1
   }

############################  test for convergence; if converged then
#                             break   out of the iteration loop

############  estimate the approximation error as done in [HBW 1984]

# do min.iterations iterations before start testing for convergence

   L <- max(abs(fnp1 - fn)) / max(abs(fn - fnm1))
   err.est.denom <- abs(1 - L) + 1.0e-4  # protect against L = 1

   estimated.error <- (L / err.est.denom) * max(abs(fnp1 - fn)) / max(abs(fnp1))

   if ((estimated.error <= tau.conv) && (iter > min.iterations)) break

}  # end of iteration loop

#23456789 123456789 123456789 123456789 123456789 123456789 123456789 1234567890
############  have finished iterations

if (iteration.num == max.num.iter) print("WARNING iteration did not converge")

################################ calculate desired quantities
################################ and return them in a list

f <- fnp1
f0 <- 3 * f[1] - 3 * f[2] + f[3]  # quadratic extrapolation
fpi <- f0  # by symmetry of ODFs that minimize F, f(pi) = f(0)

f.for.plotting <- c(f0, fnp1, fpi)
# length is:   num.theta.points + 2 = (N.theta.intervals - 1) + 2

# P2(x)  the 2nd Legendre polynomial is  (3*x^2 - 1) / 2
# P4(x)  the 4th Legendre polynomial is  (35*x^4 - 30*x^2 + 3) / 8

pi.d4.minus.ave.W.gamma <- pi/4 -
                          integral.operator.K.part.of.free.energy * 2 / B
# note this is for the previous iterate, not the final iterate fnp1,
```

```
# but should be quite close enough if the iteration converged

ave.ln.f <- log.part.for.F   # also for previous iterate

ave.P2.cos.theta <- 0.5 * sum((3*cos.theta.vec^2 - 1) * fnp1 * sin.theta.vec) *
                          delta.theta / 2

ave.P4.cos.theta <- 0.125 * sum((35*cos.theta.vec^4 - 30*cos.theta.vec^2  + 3) *
                          fnp1 * sin.theta.vec) * delta.theta / 2

vector.of.values <- c(B, N.theta.intervals, f0, L, pi.d4.minus.ave.W.gamma,
                      ave.ln.f, ave.P2.cos.theta, ave.P4.cos.theta,
                      free.energy.for.fn,
                      iteration.num, num_iter_F_did.NOT.decrease )

# use notation in [HBW 1984]
names.for.vector.of.values <- c("B", "num_theta_intervals", "f(0)", "L",
         "pi/4 - <<W(gamma)>>", "<ln(f(theta))>", "<P2(cos(theta))>",
         "<P4(cos(theta))>", "free.energy",
         "num_iterations", "num_iter_F_did_NOT_decrease")

list.to.return <- list(f.for.plotting = f.for.plotting,
          free.energy.for.each.iteration = free.energy.for.each.iteration,
          vector.of.values = vector.of.values,
          names.for.vector.of.values = names.for.vector.of.values)

return(list.to.return)

}  # end of  calculate_ODF   function

return(calculate_ODF)  # calculate_ODF is defined when call compute_approx_ODF

}  # end of  compute_approx_ODF

## ***  END OF **The R code for compute_approx_ODF and calculate_ODF *****

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

Sample runs of compute_approx_ODF and calculate_ODF **********

```
# If you were starting a new R session to run this code on your computer you would do:

rm(list = ls())  # clear out any "left over" variables (R objects) (usually a good idea)
##### note rm in R is not the Unix rm that permanently removes files

dirstr <- "C:/berger/orientationdistr/ODF_R_fcns"  # replace this location by the location
#               on your computer of the file containing the R code for compute_approx_ODF

filename <- "compute_approx_ODF_8Dec2023.R"
# the name of the file on my computer containing the code for compute_approx_ODF

full.file.name <- paste(dirstr, "/", filename, sep = "")  # the full path
```

```
source(full.file.name)  # "compile" the program
##### when this Rmd file is run through knitr,
##### the R code for compute_approx_ODF is "compiled"
##### by the previous section of R code in this Rmd file
```

Sample runs for the hard core reference system $W(\gamma) = \sin(\gamma)$

Run a case (choices of B and initial.f) that will produce an **axial** solution

```
# give appropriate values for the arguments of the   compute_approx_ODF   function

N.theta.intervals <- 256  # values of the approximate ODF that this code
#                                                 calculates correspond to equally spaced
#                                                 theta points inside (0, pi)
N.phi.intervals <- 2048

W <- function(gamma) sin(gamma)
# this is W for the hard core reference system

###############   run  calculate_ODF             ######################

calculate_ODF <- compute_approx_ODF(N.theta.intervals, N.phi.intervals, W)
```

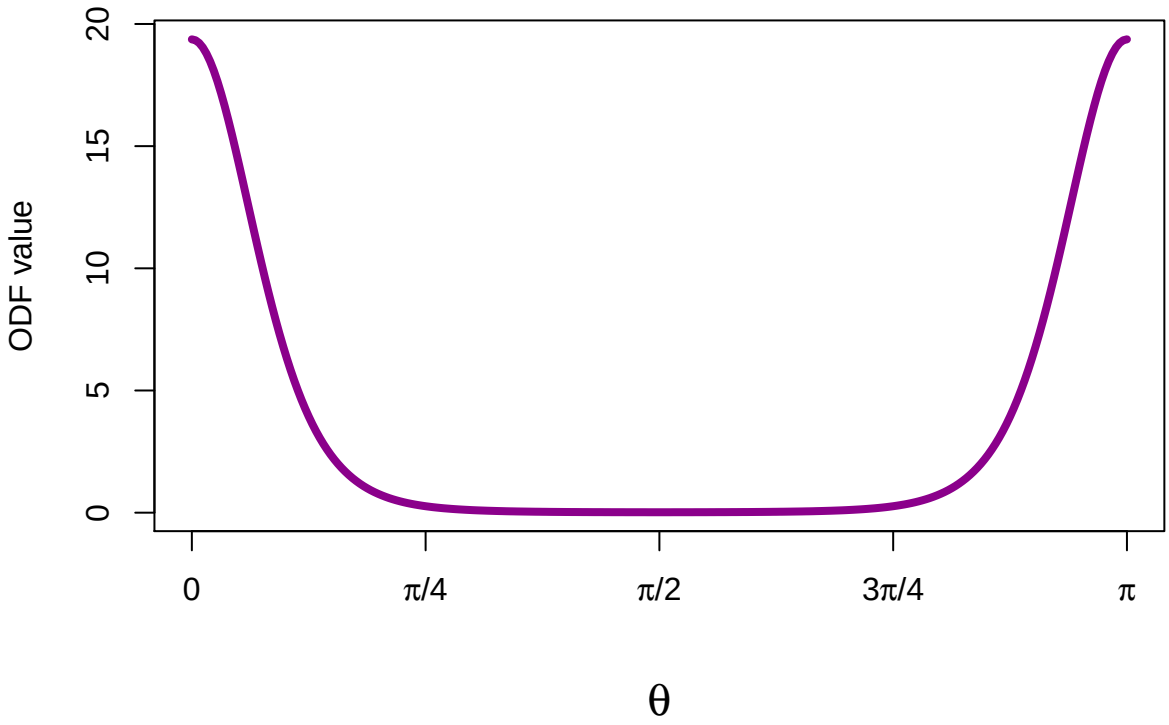
```
## [1] "run of   compute_approx_ODF   December 8, 2023  version"
```

```
# compute_approx_ODF
# calculates quantites that stay fixed for given values of N.theta.intervals,
# N.phi.intervals,
# and the given function W,   (W(gamma) is sin(gamma) for the hard core reference system),
# and returns the function   calculate_ODF   that carries out the iteration to obtain
# an ODF, here a vector of values (at the theta points inside (0, pi)
# determined by  N.theta.intervals) that should approximate a local minimum
# of the free energy function F

# compute_approx_ODF   takes about 20 seconds to run on my relatively old Windows 10
# computer when W(gamma) is sin(gamma), longer for more complicated W


# Now can call   calculate_ODF   with, in particular, various values of the parameter B
# (1 value of B for each run of  calculate_ODF).
# calculate_ODF  will obtain a discrete ODF
# (meaning a vector f of values at the theta grid points)
# where the discrete ODF returned by   calculate_ODF  should approximate a continuous ODF
# that is a local minimum of the free energy, along with associated values of interest.
# calculate_ODF takes very little time to complete a run

###############    specify values for the arguments of    calculate_ODF   for a sample run

B <- 10.8  # the "main" varying parameter in the free energy function F
tau.conv <- 1.0e-5  # the bound for the convergence criterion
max.num.iter <- 200  # maximum number of iterations allowed
min.iterations <- 10  # require having done  min.iterations  iterations before
```

```
#                                          start testing for convergence

# supply an initial value, called initial.f, for the discrete ODF; first determine
# the theta grid points inside [0, pi] at which the discrete ODF will be calculated

delta.theta <- pi / N.theta.intervals  # length of theta subintervals
num.theta.points <- N.theta.intervals - 1
theta.indices <- 1:num.theta.points
theta.vec <- theta.indices * delta.theta
# theta points for values of the calculated discrete ODF

# specify an initial ODF, here an "axial" function (i.e., roughly U shaped on [0, pi])

initial.f <- cos(theta.vec) * cos(theta.vec)  # a convenient U shaped function on [0, pi]

# initial.f will be normalized to be a positive ODF inside calculate_ODF

############   run   calculate_ODF   for the input values that were defined above ########

results <- calculate_ODF(initial.f, B, tau.conv, max.num.iter, min.iterations)
```

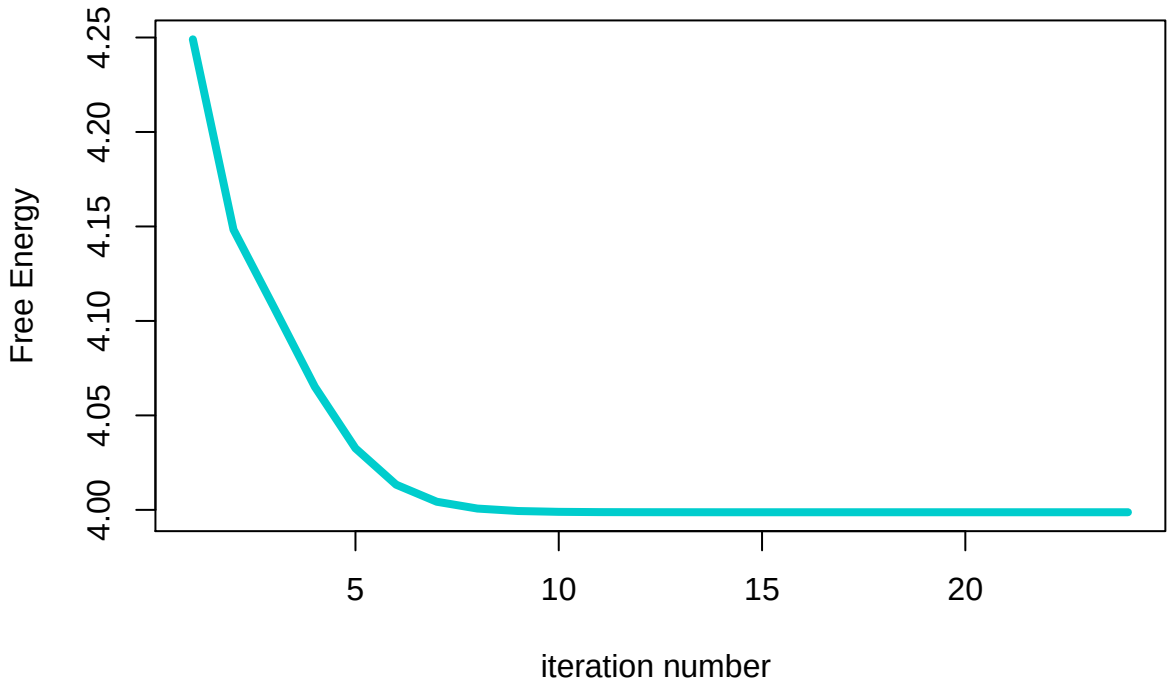
```
## [1] "run of   calculate_ODF    December 8, 2023  version"
```

```
# The results R list returned by calculate_ODF contains:
# 1. the vector of values for the discrete ODF (at the theta grid points theta.vec)
#      obtained by the iteration carried out by calculate_ODF,
#      with extrapolated values for theta = 0 and
#      theta = pi appended at the left and right, respectively
#      (for being able to do plots on [0, pi])
# 2. the value of the calculated free energy at all but the last iterative step
#      (all the values of interest have essentially converged by then
#      if the iteration converged)
# 3. a vector of values of interest (as in Tables I and II in [HBW 1984])
# 4. a vector of the names of these values (using the notation in [HBW 1984])

f.for.plotting  <- results[[1]]  # plot if desired
theta.points.for.plot <- c(0, theta.vec, pi)

line.color <- "darkmagenta"
plot(theta.points.for.plot, f.for.plotting, lwd = 4, ,
       type = "l", col = line.color, xaxt = "n", xlab = ' ', ylab = 'ODF value')
# request tic marks and labels for the x-axis:
axis(1, at = c(0, pi/4,  pi/2, 3*pi/4,  pi),
        labels = c("0", expression(paste(pi,"/4")), expression(paste(pi,"/2")),
                        expression(paste("3",pi,"/4")),  expression(pi)))
title(main = paste("Fig. 1.  axial ODF   B = ", B, "   W(g) = sin(g) " ), cex.main = 1.2)
title(main = NULL, sub = expression(theta), cex.sub = 1.4)
```




```
# Fig. 1.  Legend: The initial ODF is an axial (U shaped ) discrete ODF with peaks
# at theta = 0 and theta = pi. Values of interest displayed below after the free energy
# plot are consistent with those for B = 10.8 in Table I of [HBW 1984].

# plot the calculated free energy as a function of iteration number
free.energy.for.each.iteration <- results[[2]]

line.color <- "cyan3"
plot(free.energy.for.each.iteration, lwd = 4, ,
       type = "l", col = line.color, xlab = 'iteration number ', ylab = 'Free Energy')
title(main = paste("Fig. 2.  free energy F    for an axial ODF  B = ", B, "   W(g) = sin(g) " ),
cex.main = 1.2)
```

Fig. 2. free energy F for an axial ODF B = 10.8 W(g) = sin(g)




```
# Fig. 2.  Legend: The initial ODF is an axial (U shaped ) discrete ODF with peaks
# at theta = 0 and theta = pi. The free energy is seen to monotonically decrease
# as confirmed by num_iter_F_did_NOT_decrease being 0 (see the display below).

#### display values of interest

vector.of.values <- results[[3]]

names.for.vector.of.values <- results[[4]]
# explanation / notation of what these values are is given at the
# beginning of this Rmd file

# display these values using a data frame
df.for.values <- data.frame(names.for.vector.of.values,  vector.of.values,
                                                    stringsAsFactors = FALSE)
colnames(df.for.values) <- c("variable name", "value")
df.for.values
```

```
##                 variable name       value
## 1                           B  10.8000000
## 2        num_theta_intervals 256.0000000
## 3                        f(0)  19.3708823
## 4                           L   0.5569568
## 5        pi/4 - <<W(gamma)>>   0.3497262
## 6            <ln(f(theta))>   1.6461194
## 7          <P2(cos(theta))>   0.8002117
```

```
## 8             <P4(cos(theta))>   0.5230953
## 9                  free.energy   3.9987481
## 10               num_iterations  24.0000000
## 11 num_iter_F_did_NOT_decrease   0.0000000
```

specify an initial ODF that is a "planar" function with a peak at theta = pi / 2

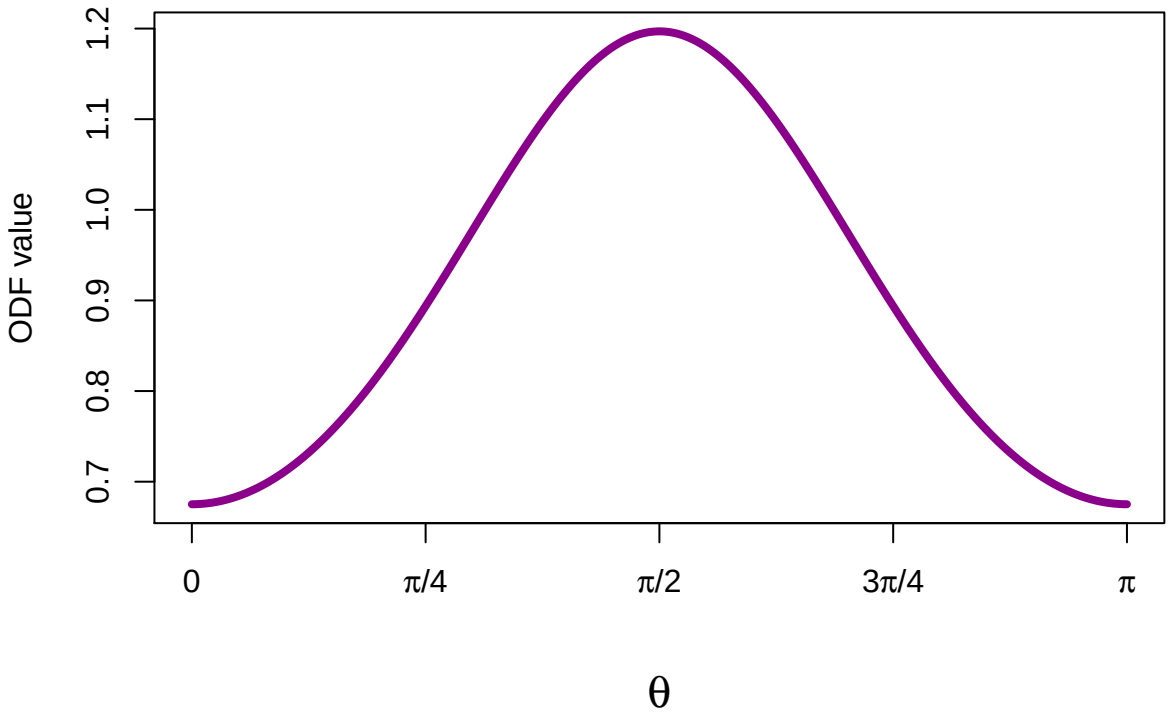
Leave the rest of the parameters as above.

```
# specify an initial ODF that is a "planar" function with a peak at theta = pi / 2
initial.f <- sin(theta.vec) * sin(theta.vec)

###########   run   calculate_ODF   for the input values that were defined above ########

results <- calculate_ODF(initial.f, B, tau.conv, max.num.iter, min.iterations)
```

```
## [1] "run of   calculate_ODF    December 8, 2023  version"
```

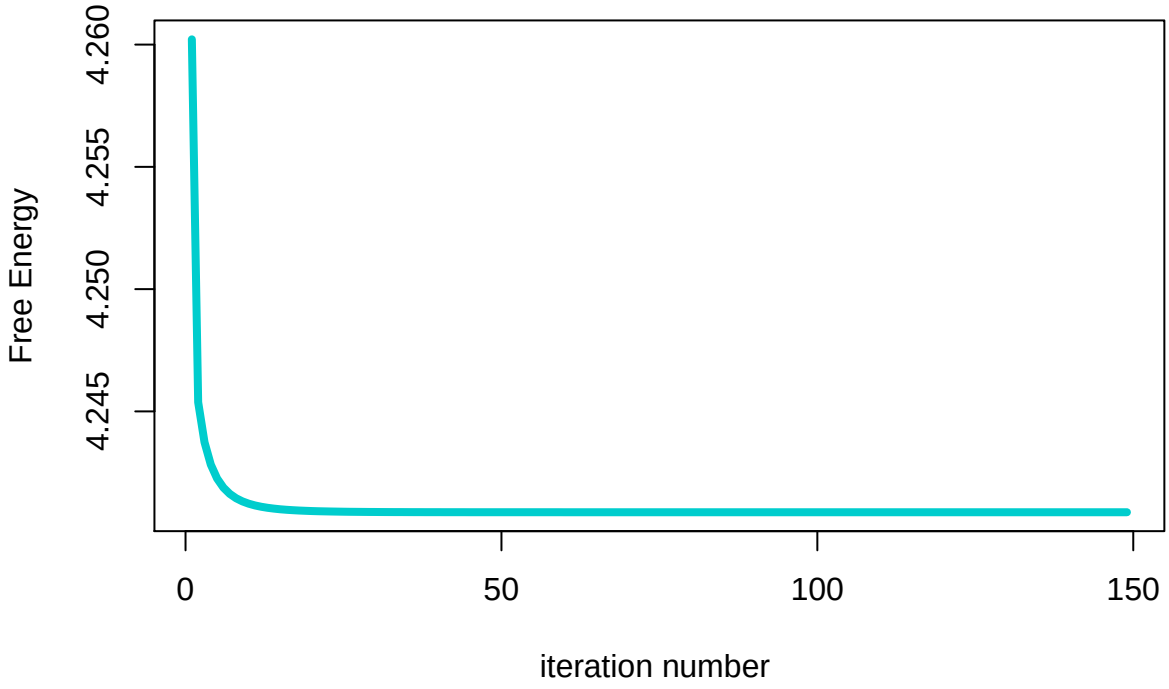
```
f.for.plotting  <- results[[1]]  # plot if desired

line.color <- "darkmagenta"
plot(theta.points.for.plot, f.for.plotting, lwd = 4, ,
       type = "l", col = line.color, xaxt = "n", xlab = ' ', ylab = 'ODF value')
# request tic marks and labels for the x-axis:
axis(1, at = c(0, pi/4,  pi/2, 3*pi/4,  pi),
        labels = c("0", expression(paste(pi,"/4")), expression(paste(pi,"/2")),
                        expression(paste("3",pi,"/4")),  expression(pi)))
title(main = paste("Fig. 3.  planar ODF    B = ", B, "   W(g) = sin(g) " ), cex.main = 1.2)
title(main = NULL, sub = expression(theta), cex.sub = 1.4)
```

Fig. 3. planar ODF B = 10.8 W(g) = sin(g)




```
# Fig. 3.  Legend: The initial ODF is a planar (peak at pi / 2) discrete ODF
# Values of interest displayed below after the free energy
# plot are consistent with those for B = 10.8 in Table II of [HBW 1984].

# plot the calculated free energy as a function of iteration number
free.energy.for.each.iteration <- results[[2]]

line.color <- "cyan3"
plot(free.energy.for.each.iteration, lwd = 4, ,
       type = "l", col = line.color, xlab = 'iteration number ', ylab = 'Free Energy')
title(main = paste("Fig. 4.  free energy F    for a planar ODF  B = ", B, "   W(g) = sin(g) " ),
cex.main = 1.2)
```

Fig. 4. free energy F for a planar ODF B = 10.8 W(g) = sin(g)




```
# Fig. 4.  Legend: The initial ODF is a planar (peak at pi / 2) discrete ODF
# The free energy is seen to monotonically decrease
# as confirmed by num_iter_F_did_NOT_decrease being 0 (see the display below).

#### display values of interest

vector.of.values <- results[[3]]
names.for.vector.of.values <- results[[4]]
# explanation / notation of what these values are is given at the
# beginning of this Rmd file

# display these values using a data frame
df.for.values <- data.frame(names.for.vector.of.values,  vector.of.values,
                                                    stringsAsFactors = FALSE)
colnames(df.for.values) <- c("variable name", "value")
df.for.values
```

```
##                 variable name         value
## 1                           B  10.800000000
## 2        num_theta_intervals 256.000000000
## 3                        f(0)   0.675105861
## 4                           L   0.937785055
## 5        pi/4 - <<W(gamma)>>   0.002575473
## 6             <ln(f(theta))>   0.013633173
## 7          <P2(cos(theta))>  -0.072403273
## 8          <P4(cos(theta))>   0.004422443
```

```
## 9                  free.energy   4.240875700
## 10              num_iterations 149.000000000
## 11 num_iter_F_did_NOT_decrease   0.000000000
```

example of writing out to a file results from several runs as a tab delimited text file

```
# do multiple runs with an axial initial.f and "collect them" for output
B.values <- c(8.9, 9.0, 9.1, 9.2, 9.5, 9.7, 10.8, 11.9, 12.9, 14.0)

# have already run, above,
# calculate_ODF <- compute_approx_ODF(N.theta.intervals, N.phi.intervals, W)
# with N.theta.intervals equal 256,  N.phi.intervals equal 2048,
# and W(gamma) equal sin(gamma)

tau.conv <-  1.0e-5
max.num.iter <-  300
min.iterations <- 10
axial.initial.f <- cos(theta.vec) * cos(theta.vec)  # a convenient U shaped function on [0, pi]

matrix.of.results <- numeric(0)  # initialize "container" for the results

for (Bv in B.values) {
   results <- calculate_ODF(axial.initial.f, Bv, tau.conv, max.num.iter, min.iterations)
   vector.of.values <- results[[3]]
   matrix.of.results <- rbind(matrix.of.results, vector.of.values)  # row bind
}
```

```
## [1] "run of   calculate_ODF    December 8, 2023  version"
## [1] "run of   calculate_ODF    December 8, 2023  version"
## [1] "run of   calculate_ODF    December 8, 2023  version"
## [1] "run of   calculate_ODF    December 8, 2023  version"
## [1] "run of   calculate_ODF    December 8, 2023  version"
## [1] "run of   calculate_ODF    December 8, 2023  version"
## [1] "run of   calculate_ODF    December 8, 2023  version"
## [1] "run of   calculate_ODF    December 8, 2023  version"
## [1] "run of   calculate_ODF    December 8, 2023  version"
## [1] "run of   calculate_ODF    December 8, 2023  version"
```

```
names.for.vector.of.values <- results[[4]]
df.for.output <- data.frame(matrix.of.results)  # one row for each B
colnames(df.for.output) <- names.for.vector.of.values

outp.file.name <- "results.for.several.B.axial.ODFs.tab"
# this will be a tab delimited text file (view it with Excel)
dirstr <- "C:/berger/orientationdistr/ODF_R_fcns"  # replace this location by the location
#               on your computer where you want the file written
full.output.file.name <- paste(dirstr, "/", outp.file.name, sep = "")

# remove the ## (commenting out) of the call to write.table below to write out the file
# as a tab ("\t") delimited text file on your computer after choosing
```

```
# an appropriate location for   dirstr
##write.table(df.for.output, file = full.output.file.name,
##            append = FALSE, quote = FALSE, sep = "\t",
##            row.names = FALSE, col.names = TRUE)
```

Run W(gamma) = sin(gamma) - 8 * P6(cos(gamma)) to display a different shape ODF

Notation: P2(x) is the 2nd Legendre polynomial which is $(3x^2 - 1)/2$
Notation: P4(x) is the 4th Legendre polynomial which is $(35x^4 - 30x^2 + 3)/8$
Notation: P6(x) is the 6th Legendre polynomial which is $(231x^6 - 315x^4 + 105x^2 - 5)/16$

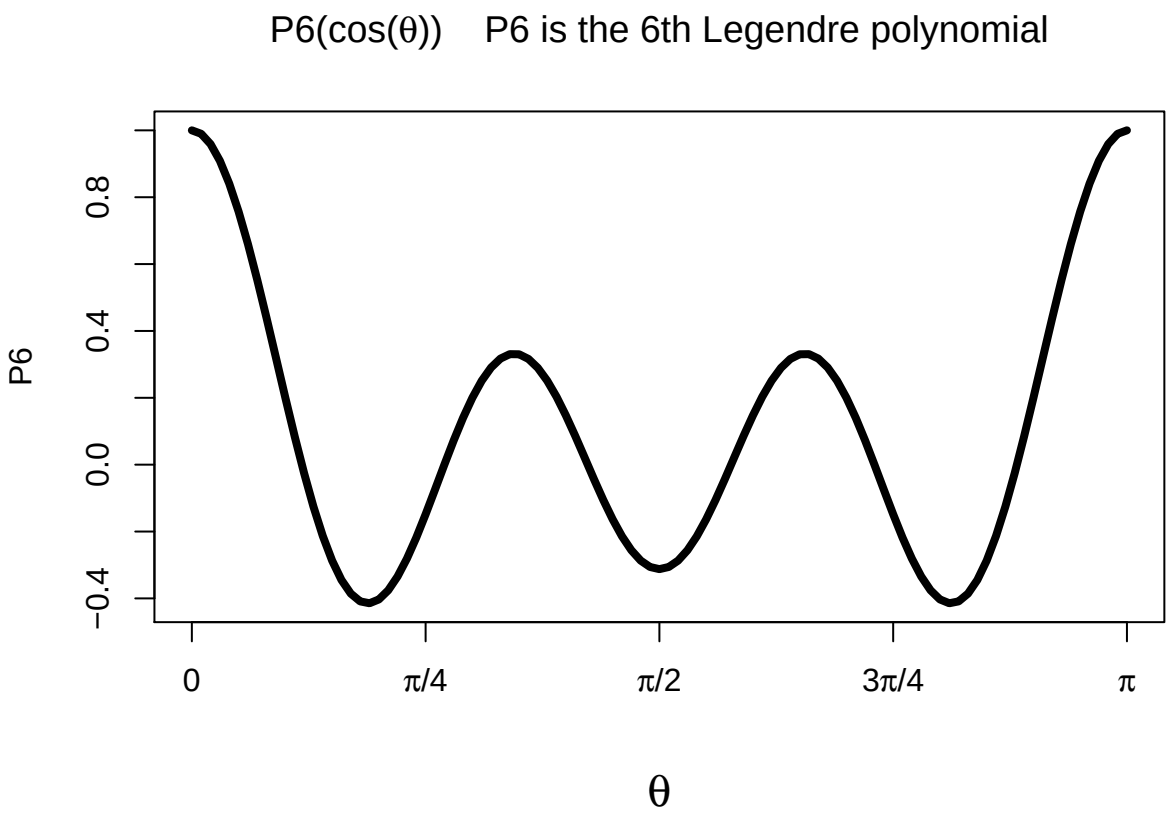
The coefficient (-8)of P6 in W above has the correct sign to have each iteration decrease the free energy. Use the planar shaped initial condition.

Note the coefficient w_6 of P6(cos(gamma)) in the expansion of sin(gamma) in terms of the Legendre polynomials P_{2m}(cos(gamma)) with m = 0,1,2,3, ...
is -$65\pi/2^{12}$ which equals -0.04985438 to that many digits.

The coefficients w_0, w_2, and w_4 of P0(cos(gamma)), P2(cos(gamma)), P4(cos(gamma)), in the expansion of sin(gamma) are, respectively, $\pi/4$, -$5\pi/32$, -$9\pi/256$

By symmetry considerations, since $\sin(\pi - \gamma) = \sin(\gamma)$, the coefficients w_1, w_3, w_5, ... of the odd index Legendre polynomials in the expansion of sin(gamma) are all 0. Also, all the coefficients for the positive even index Legendre polynomials in the expansion of sin(gamma) are negative (so less than or equal 0) [KR 1978], [HBW 1984]. These conditions are sufficient to quarantee that the iteration (for continuous ODFs) decreases the free energy at each step (unless the iteration didn't change the ODF). These conditions are in fact sufficient for general continuous functions W for which $W(\pi - \gamma) = W(\gamma)$, which is a natural condition when considering cylindrically symmetric ODFs, i.e., independent of ϕ, assuming the sum of the even coefficients w_{2m} for W is absolutely convergent (AEB manuscript in preparation).

```
# First, here is a plot of P6(cos(theta))

# plot P6(cos(theta))
  P6 <- function(x) (231*x^6 - 315*x^4 + 105*x^2 - 5) / 16
  dtheta <- pi / 100
  indices <- 0:100
  thetavec <- dtheta*indices
  P6vec <- P6(cos(thetavec))

line.color <- "black"
plot(thetavec, P6vec, lwd = 4, ,
        type = "l", col = line.color, xaxt = "n", xlab = ' ', ylab = 'P6')
# request tic marks and labels for the x-axis:
axis(1, at = c(0, pi/4,  pi/2, 3*pi/4,  pi),
         labels = c("0", expression(paste(pi,"/4")), expression(paste(pi,"/2")),
                         expression(paste("3",pi,"/4")),  expression(pi)))
title(main = expression(paste("P6(cos(", theta, "))    P6 is the 6th Legendre polynomial")),
cex.main = 1.2)
title(main = NULL, sub = expression(theta), cex.sub = 1.4)
```




```
W <- function(gamma) {
   x <- cos(gamma)
   return(sin(gamma) - 8 * (231*x^6 - 315*x^4 + 105*x^2 - 5) / 16)
}

initial.f <- sin(theta.vec) * sin(theta.vec)  # planar shaped

B <- 12  # use a larger B to display an ODF with a different shape

# leave the rest of the parameters as above.

# since have a new W, need to do a new run of  compute_approx_ODF
calculate_ODF <- compute_approx_ODF(N.theta.intervals, N.phi.intervals, W)
```

```
## [1] "run of   compute_approx_ODF   December 8, 2023  version"
```

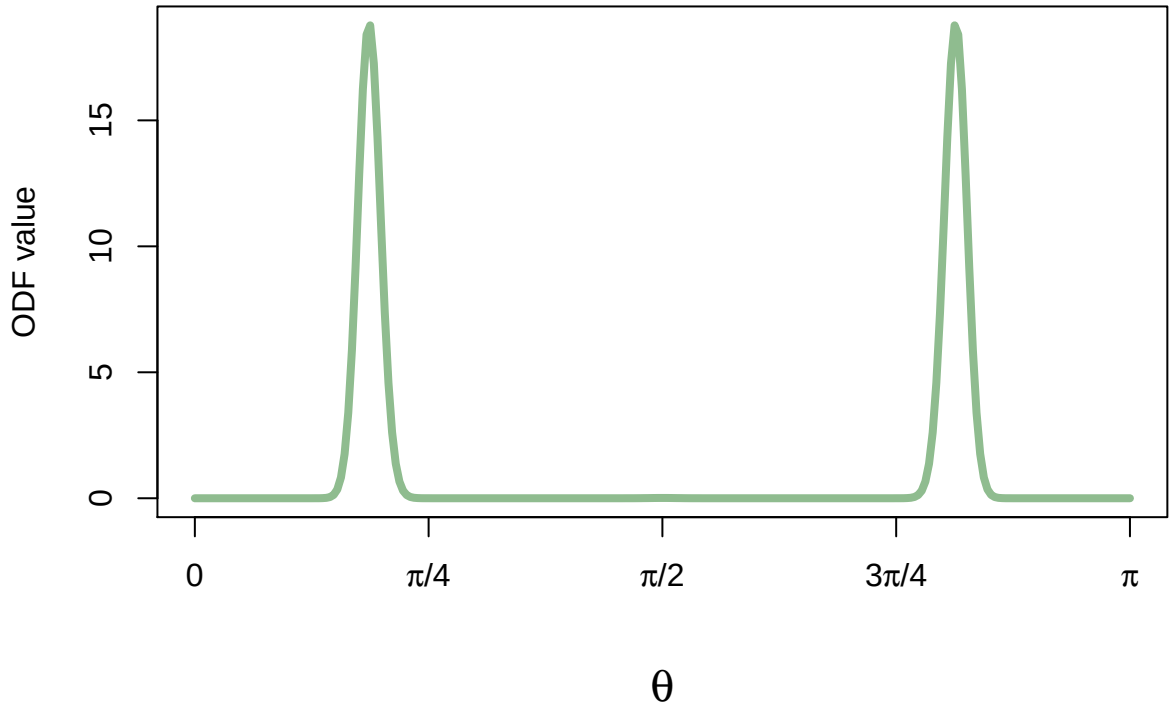
```
###########   run   calculate_ODF   for the input values that were defined above ########

results <- calculate_ODF(initial.f, B, tau.conv, max.num.iter, min.iterations)
```

```
## [1] "run of   calculate_ODF    December 8, 2023  version"
```

```
f.for.plotting  <- results[[1]]  # plot if desired
```

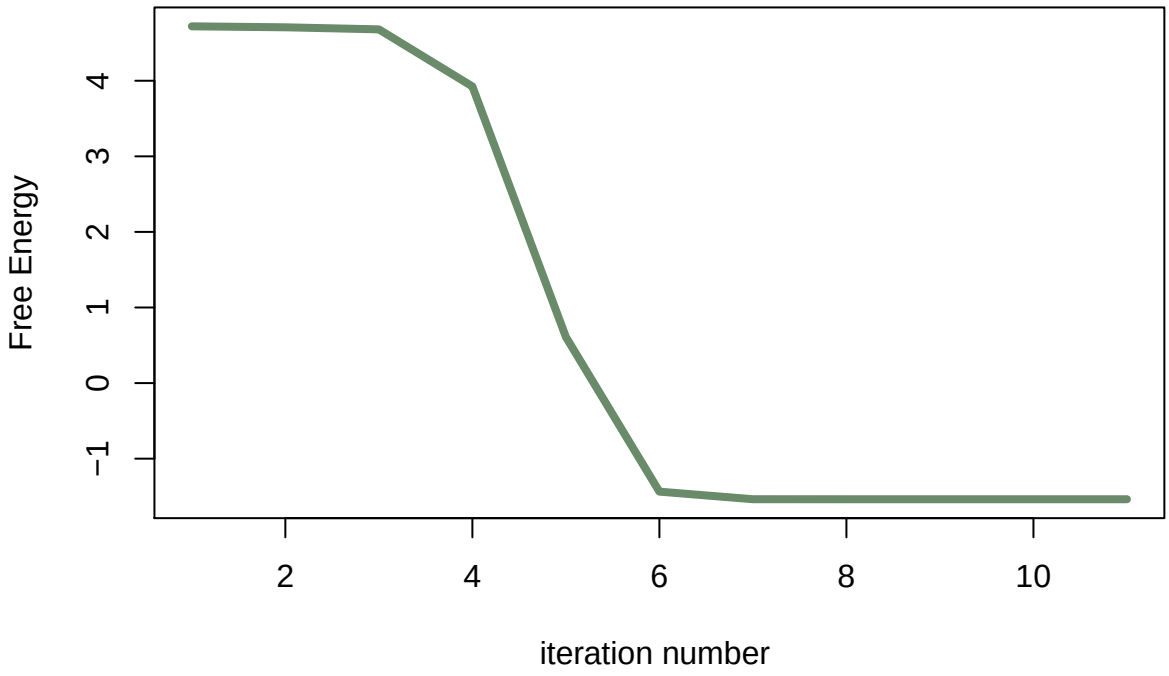
```
line.color <- "darkseagreen"
plot(theta.points.for.plot, f.for.plotting, lwd = 4, ,
       type = "l", col = line.color, xaxt = "n", xlab = ' ', ylab = 'ODF value')
# request tic marks and labels for the x-axis:
axis(1, at = c(0, pi/4,  pi/2, 3*pi/4,  pi),
        labels = c("0", expression(paste(pi,"/4")), expression(paste(pi,"/2")),
                        expression(paste("3",pi,"/4")),  expression(pi)))
title(main = paste("Fig. 5.  different shape ODF    B = ", B, "   W(g) = sin(g) - 8*P6(cos(g))" ),
cex.main = 1.1)
title(main = NULL, sub = expression(theta), cex.sub = 1.4)
```

Fig. 5. different shape ODF B = 12 W(g) = sin(g) − 8*P6(cos(g))




```
# Fig. 5.  Legend: The initial ODF is a planar shaped discrete ODF
# The free energy decreased at each iteration (see plot and results below)

# plot the calculated free energy as a function of iteration number
free.energy.for.each.iteration <- results[[2]]

line.color <- "darkseagreen4"
plot(free.energy.for.each.iteration, lwd = 4, ,
       type = "l", col = line.color, xlab = 'iteration number ', ylab = 'Free Energy')
title(main = paste("Fig. 6.  free energy F    for Fig. 5. run    B = ", B), cex.main = 1.2)
```

Fig. 6. free energy F for Fig. 5. run B = 12




```
# Fig. 6.  Legend:  The free energy is seen to monotonically decrease
# as confirmed by num_iter_F_did_NOT_decrease being 0 (see the display below).

#### display values of interest

vector.of.values <- results[[3]]
names.for.vector.of.values <- results[[4]]
# explanation / notation of what these values are is given at the
# beginning of this Rmd file

# display these values using a data frame
df.for.values <- data.frame(names.for.vector.of.values,  vector.of.values,
                                                    stringsAsFactors = FALSE)
colnames(df.for.values) <- c("variable name", "value")
df.for.values
```

```
##                  variable name         value
## 1                            B  1.200000e+01
## 2          num_theta_intervals  2.560000e+02
## 3                         f(0)  1.009017e-22
## 4                            L  3.543127e-02
## 5          pi/4 - <<W(gamma)>>  1.445551e+00
## 6               <ln(f(theta))>  2.424155e+00
## 7             <P2(cos(theta))>  5.363273e-01
## 8             <P4(cos(theta))> -1.206562e-01
## 9                  free.energy -1.536760e+00
```

```
## 10              num_iterations  1.100000e+01
## 11 num_iter_F_did_NOT_decrease  0.000000e+00
```

Run W(gamma) = -sin(gamma)

For this W the signs of all the coefficients of the positive even index Legendre polynomials in the expansion of W have the wrong sign. The ODF and the calculated free energy oscillates from iterate to iterate.

```
##################     Run W(gamma) = -sin(gamma)   ########

N.theta.intervals <- 256
N.phi.intervals <- 2048

W <- function(gamma) {-sin(gamma)}

# since have a new W, need to do a new run of  compute_approx_ODF
calculate_ODF <- compute_approx_ODF(N.theta.intervals, N.phi.intervals, W)
```

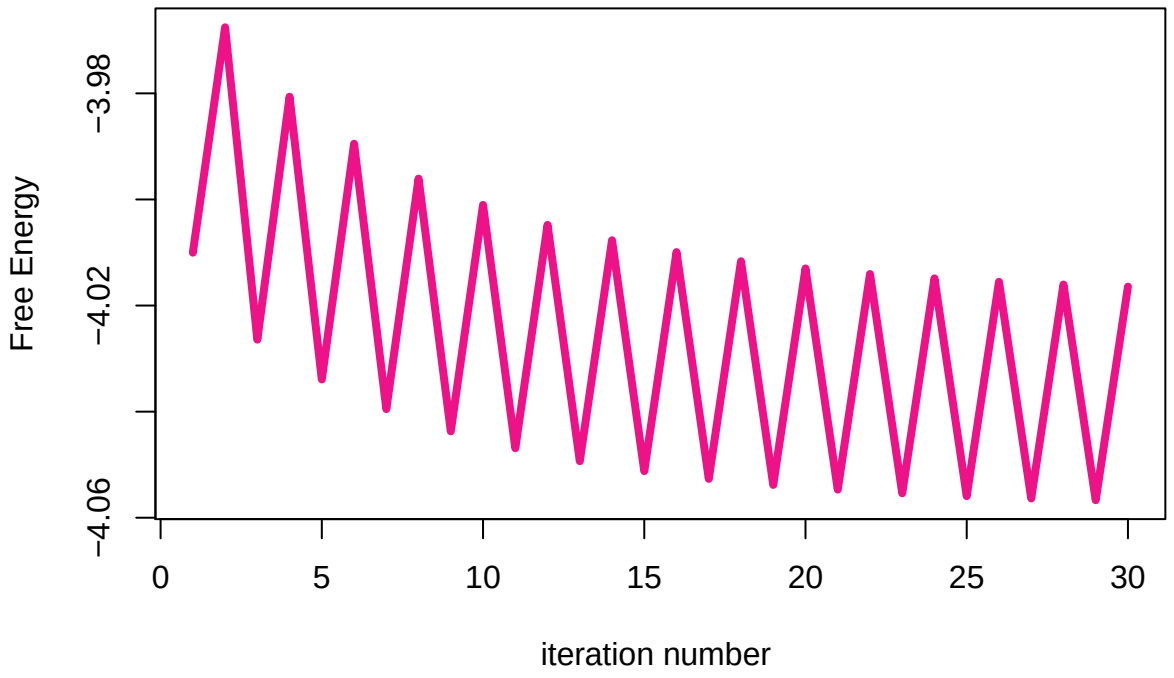
```
## [1] "run of   compute_approx_ODF   December 8, 2023  version"
```

```
B <- 10.8
initial.f <- sin(theta.vec) * sin(theta.vec)  # planar shaped

tau.conv <- 1.0e-5  # the bound for the convergence criterion
max.num.iter <- 30  # maximum number of iterations allowed
min.iterations <- 10  # require having done  min.iterations  iterations before
#                                      start testing for convergence

results <- calculate_ODF(initial.f, B, tau.conv, max.num.iter, min.iterations)
```

```
## [1] "run of   calculate_ODF    December 8, 2023  version"
## iter =  2   F did not decrease   only printing for first 12 instances
## iter =  4   F did not decrease   only printing for first 12 instances
## iter =  6   F did not decrease   only printing for first 12 instances
## iter =  8   F did not decrease   only printing for first 12 instances
## iter =  10   F did not decrease   only printing for first 12 instances
## iter =  12   F did not decrease   only printing for first 12 instances
## iter =  14   F did not decrease   only printing for first 12 instances
## iter =  16   F did not decrease   only printing for first 12 instances
## iter =  18   F did not decrease   only printing for first 12 instances
## iter =  20   F did not decrease   only printing for first 12 instances
## iter =  22   F did not decrease   only printing for first 12 instances
## iter =  24   F did not decrease   only printing for first 12 instances
## [1] "WARNING iteration did not converge"
```

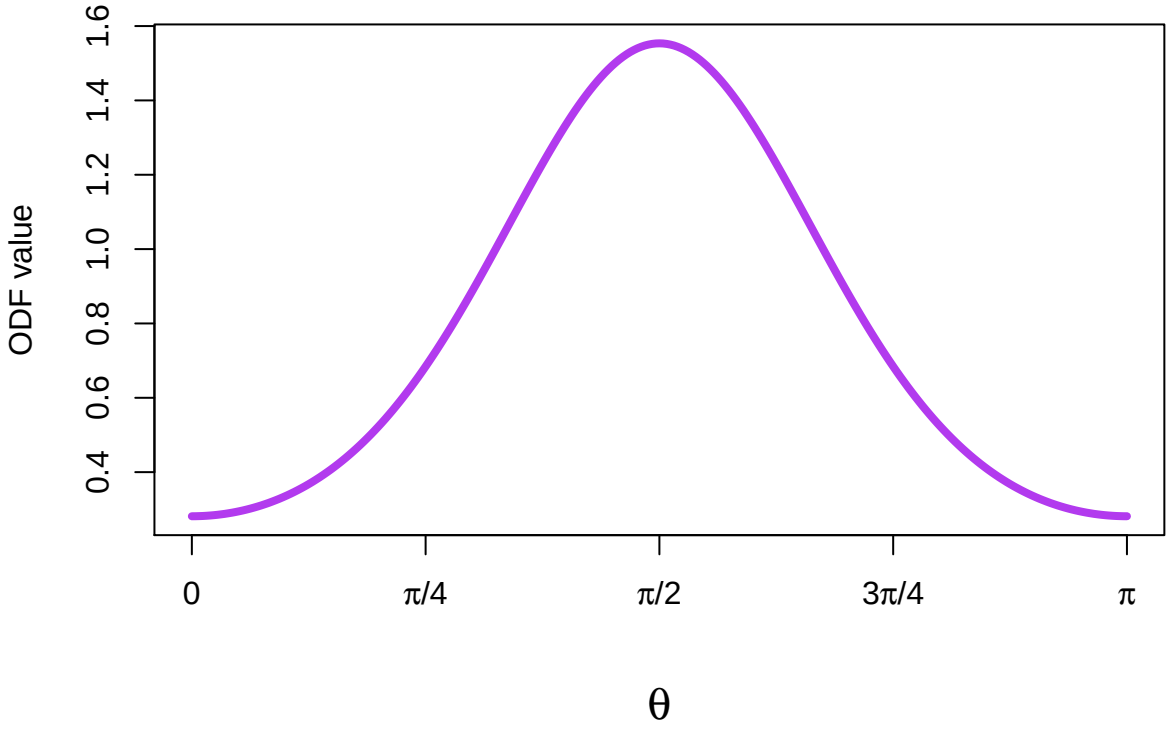
```
# plot the calculated free energy as a function of iteration number
free.energy.for.each.iteration <- results[[2]]

line.color <- "deeppink2"
plot(free.energy.for.each.iteration, lwd = 4, ,
        type = "l", col = line.color, xlab = 'iteration number ', ylab = 'Free Energy')
```

```
title(main = paste("Fig. 7.  free energy F   W(g) = -sin(g) so coeffs have wrong sign" ),
cex.main = 1.2)
```




```
# Fig. 7.  Legend: All the positive even integer coefficients of this W in its expansion
# in terms of Legendre polynomials Pm(cos(gamma)) are positive (have the wrong
# sign as far the assumptions for the proof that the iteration decreases the free energy
# (for the case of continuous ODFs)).
# The free energy is seen to oscillate, and the iteration does not converge.
# The oscillation was observed to continue similarly out to 200 iterations
# (run not shown here).

##############   plot the ODF iterate after the 30 iterations

f.for.plotting  <- results[[1]]  # plot if desired
theta.points.for.plot <- c(0, theta.vec, pi)  # theta.vec has been defined above

line.color <- "darkorchid2"
plot(theta.points.for.plot, f.for.plotting, lwd = 4, ,
       type = "l", col = line.color, xaxt = "n", xlab = ' ', ylab = 'ODF value')
# request tic marks and labels for the x-axis:
axis(1, at = c(0, pi/4,  pi/2, 3*pi/4,  pi),
        labels = c("0", expression(paste(pi,"/4")), expression(paste(pi,"/2")),
                        expression(paste("3",pi,"/4")),  expression(pi)))
title(main = paste("Fig. 8.  ODF after 30 iterations  W(g) = -sin(g)" ), cex.main = 1.2)
title(main = NULL, sub = expression(theta), cex.sub = 1.4)
```

Fig. 8. ODF after 30 iterations W(g) = −sin(g)




```
# Fig. 8.  Legend: The ODF after 30 iterations with W(g) = -sin(g)

# print out vector.of.values for this ODF (the ODF after 30 iterations)

vector.of.values <- results[[3]]
names.for.vector.of.values <- results[[4]]
# explanation / notation of what these values are is given at the
# beginning of this Rmd file

# display these values using a data frame
df.for.values <- data.frame(names.for.vector.of.values,  vector.of.values,
                                                    stringsAsFactors = FALSE)
colnames(df.for.values) <- c("variable name", "value")
df.for.values
```

```
##                  variable name        value
## 1                            B  10.80000000
## 2          num_theta_intervals 256.00000000
## 3                         f(0)   0.28134362
## 4                            L   0.99982822
## 5          pi/4 - <<W(gamma)>>   1.54860561
## 6               <ln(f(theta))>   0.10488420
## 7             <P2(cos(theta))>  -0.18335045
## 8             <P4(cos(theta))>   0.02512136
## 9                 free.energy  -4.01643600
## 10              num_iterations  30.00000000
```

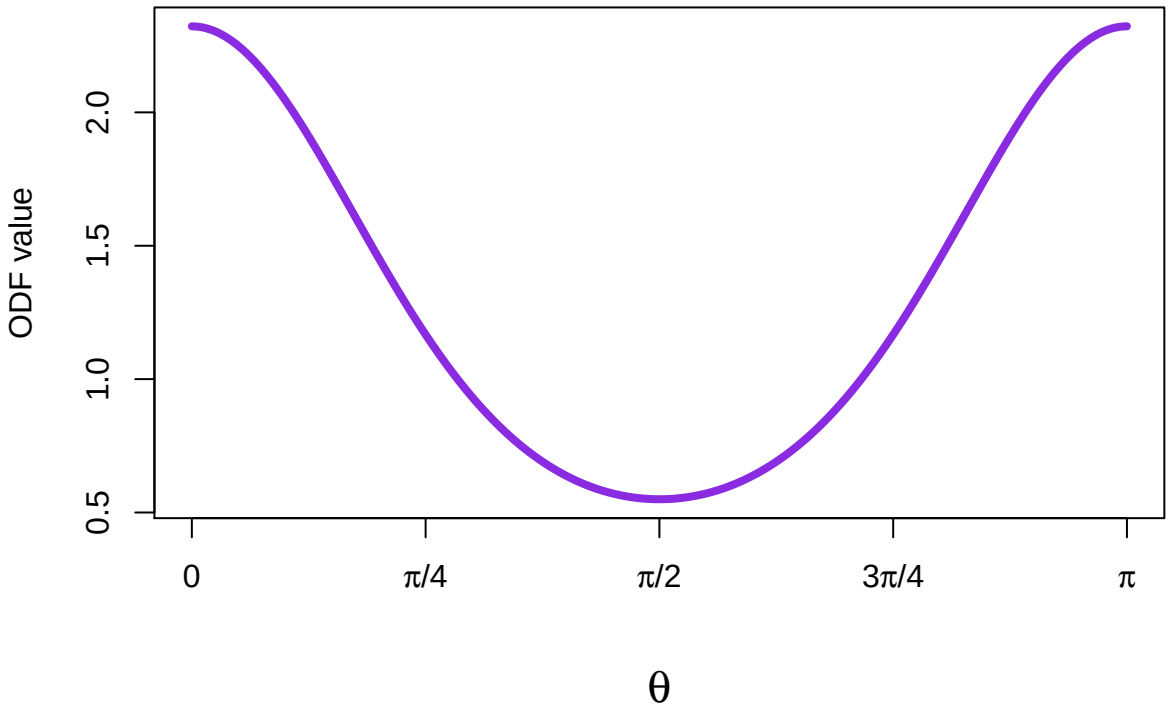
```
## 11 num_iter_F_did_NOT_decrease  15.00000000
```

```
################################  now display the ODF after 31 iterations

max.num.iter <- 31  # rest of arguments left the same

results <- calculate_ODF(initial.f, B, tau.conv, max.num.iter, min.iterations)
```

```
## [1] "run of   calculate_ODF    December 8, 2023  version"
## iter =  2   F did not decrease   only printing for first 12 instances
## iter =  4   F did not decrease   only printing for first 12 instances
## iter =  6   F did not decrease   only printing for first 12 instances
## iter =  8   F did not decrease   only printing for first 12 instances
## iter =  10   F did not decrease   only printing for first 12 instances
## iter =  12   F did not decrease   only printing for first 12 instances
## iter =  14   F did not decrease   only printing for first 12 instances
## iter =  16   F did not decrease   only printing for first 12 instances
## iter =  18   F did not decrease   only printing for first 12 instances
## iter =  20   F did not decrease   only printing for first 12 instances
## iter =  22   F did not decrease   only printing for first 12 instances
## iter =  24   F did not decrease   only printing for first 12 instances
## [1] "WARNING iteration did not converge"
```

```
#############   plot the ODF iterate after the 31 iterations

f.for.plotting  <- results[[1]]  # plot if desired
theta.points.for.plot <- c(0, theta.vec, pi)  # theta.vec has been defined above

line.color <- "blueviolet"
plot(theta.points.for.plot, f.for.plotting, lwd = 4, ,
       type = "l", col = line.color, xaxt = "n", xlab = ' ', ylab = 'ODF value')
# request tic marks and labels for the x-axis:
axis(1, at = c(0, pi/4,  pi/2, 3*pi/4,  pi),
        labels = c("0", expression(paste(pi,"/4")), expression(paste(pi,"/2")),
                        expression(paste("3",pi,"/4")),  expression(pi)))
title(main = paste("Fig. 9.  ODF after 31 iterations  W(g) = -sin(g)" ), cex.main = 1.2)
title(main = NULL, sub = expression(theta), cex.sub = 1.4)
```

Fig. 9. ODF after 31 iterations W(g) = −sin(g)




```
# Fig. 9.  Legend: The ODF after 31 iterations with W(g) = -sin(g); one sees that
# the form of the ODF oscillates between planar and axial

# print out vector.of.values for this ODF (the ODF after 31 iterations)

vector.of.values <- results[[3]]
names.for.vector.of.values <- results[[4]]
# explanation / notation of what these values are is given at the
# beginning of this Rmd file

# display these values using a data frame
df.for.values <- data.frame(names.for.vector.of.values,  vector.of.values,
                                                    stringsAsFactors = FALSE)
colnames(df.for.values) <- c("variable name", "value")
df.for.values
```

```
##                  variable name        value
## 1                            B  10.80000000
## 2          num_theta_intervals 256.00000000
## 3                         f(0)   2.32245010
## 4                            L   0.99945037
## 5          pi/4 - <<W(gamma)>>   1.55422443
## 6               <ln(f(theta))>   0.09470514
## 7             <P2(cos(theta))>   0.21210983
## 8             <P4(cos(theta))>   0.02595917
## 9                  free.energy  -4.05695669
```

```
## 10              num_iterations  31.00000000
## 11 num_iter_F_did_NOT_decrease  15.00000000
```

The iteration can still converge if a coefficient is positive

If one of the positive even index coefficients w_{2m} in the expansion of W(gamma) in terms of even index polynomials $P_{2m}(\cos\,\theta)$ is positive, the iteration ***might*** still converge. (For the W functions under consideration, $W(\pi - \gamma) = W(\gamma)$, so the odd index coefficients will all be 0.)

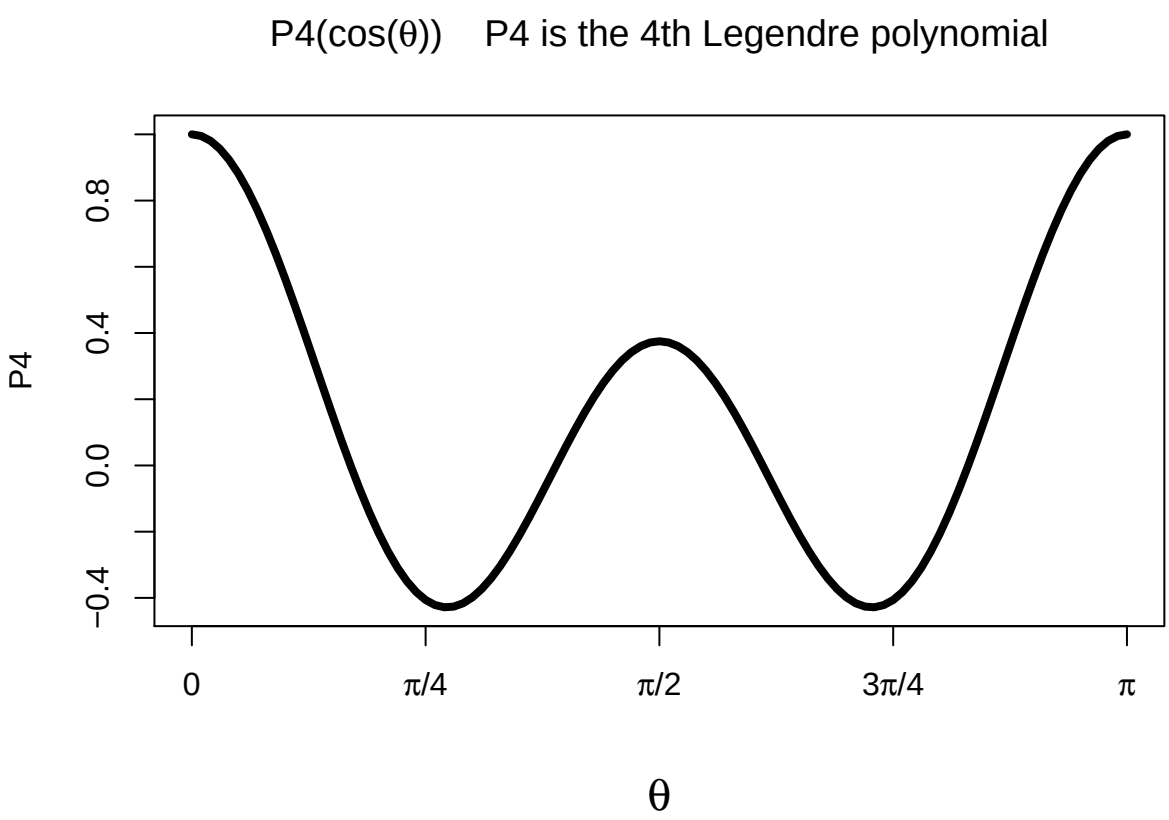
** Example: W(gamma) = sin(gamma) + 0.5 * P4(cos(gamma))**

P4(x) is the 4th Legendre polynomial which is $(35x^4 - 30x^2 + 3)/8$

As noted above, the coefficient of P4(cos(gamma)) in the expansion of sin(gamma) is -$9\pi/256$ which is -0.1104466 so the coefficient of P4 in the expansion of the function W immediately above is positive but not by a whole lot.

It turns out with this W, with both the axial type and planar type initial conditions, and the other parameters as above, the iteration converges respectively to an axial and a planar form ODF and the free energy decreases at each step, as displayed in the next section of R code.

```
##################     Run W(gamma) = sin(gamma)  + 0.5 * P4(cos(gamma)   ########

P4 <- function(x)  (35*x^4 - 30*x^2 + 3) / 8

# First, here is a plot of P4(cos(theta))

# plot P4(cos(theta))
  dtheta <- pi / 100
  indices <- 0:100
  thetavec <- dtheta*indices
  P4vec <- P4(cos(thetavec))

line.color <- "black"
plot(thetavec, P4vec, lwd = 4, ,
       type = "l", col = line.color, xaxt = "n", xlab = ' ', ylab = 'P4')
# request tic marks and labels for the x-axis:
axis(1, at = c(0, pi/4,  pi/2, 3*pi/4,  pi),
        labels = c("0", expression(paste(pi,"/4")), expression(paste(pi,"/2")),
                        expression(paste("3",pi,"/4")),  expression(pi)))
title(main = expression(paste("P4(cos(", theta, "))    P4 is the 4th Legendre polynomial")),
cex.main = 1.2)
title(main = NULL, sub = expression(theta), cex.sub = 1.4)
```




```
N.theta.intervals <- 256
N.phi.intervals <- 2048

W <- function(gamma) {
   x <- cos(gamma)
   return(sin(gamma) +  0.5 * P4(x))
}

# since have a new W, need to do a new run of  compute_approx_ODF
calculate_ODF <- compute_approx_ODF(N.theta.intervals, N.phi.intervals, W)
```

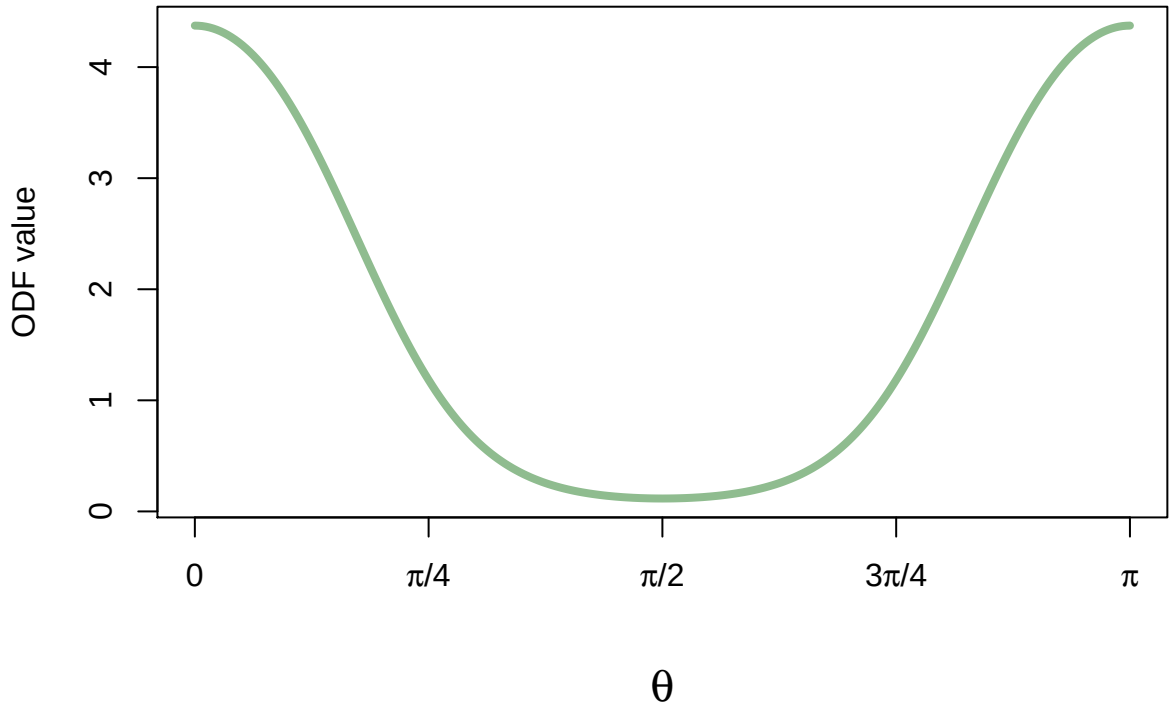
```
## [1] "run of   compute_approx_ODF   December 8, 2023  version"
```

```
B <- 10.8
# have defined theta.vec previously
initial.f <- cos(theta.vec) * cos(theta.vec)  # axial shaped

tau.conv <- 1.0e-5  # the bound for the convergence criterion
max.num.iter <- 200  # maximum number of iterations allowed
min.iterations <- 10  # require having done  min.iterations  iterations before
#                                        start testing for convergence

results <- calculate_ODF(initial.f, B, tau.conv, max.num.iter, min.iterations)
```

```
## [1] "run of   calculate_ODF    December 8, 2023  version"
```

```
f.for.plotting  <- results[[1]]  # plot if desired

line.color <- "darkseagreen"
plot(theta.points.for.plot, f.for.plotting, lwd = 4, ,
        type = "l", col = line.color, xaxt = "n", xlab = ' ', ylab = 'ODF value')
# request tic marks and labels for the x-axis:
axis(1, at = c(0, pi/4,  pi/2, 3*pi/4,  pi),
         labels = c("0", expression(paste(pi,"/4")), expression(paste(pi,"/2")),
                         expression(paste("3",pi,"/4")),  expression(pi)))
title(main = paste("Fig. 10.  W(g) = sin(g) + 0.5 * P4(cos(g)),  axial shaped initial f" ),
cex.main = 1.1)
title(main = NULL, sub = expression(theta), cex.sub = 1.4)
```

Fig. 10. W(g) = sin(g) + 0.5 * P4(cos(g)), axial shaped initial f




```
# Fig. 10.  Legend: The coefficient of P4(cos(gamma)) for this W is positive but
# still have convergence.  All the positive even index coefficients for W being less
# than or equal 0 is sufficient to prove that the iteration always decreases the free
# energy (for continuous ODFs), but is not necessarily required.
# Here the initial ODF is an axial shaped discrete ODF.
# The free energy decreased at each iteration (num_iter_F_did_NOT_decrease = 0,
# see the results below)

vector.of.values <- results[[3]]
names.for.vector.of.values <- results[[4]]

# display these values using a data frame
```

```
df.for.values <- data.frame(names.for.vector.of.values,  vector.of.values,
                                                    stringsAsFactors = FALSE)
colnames(df.for.values) <- c("variable name", "value")
df.for.values
```

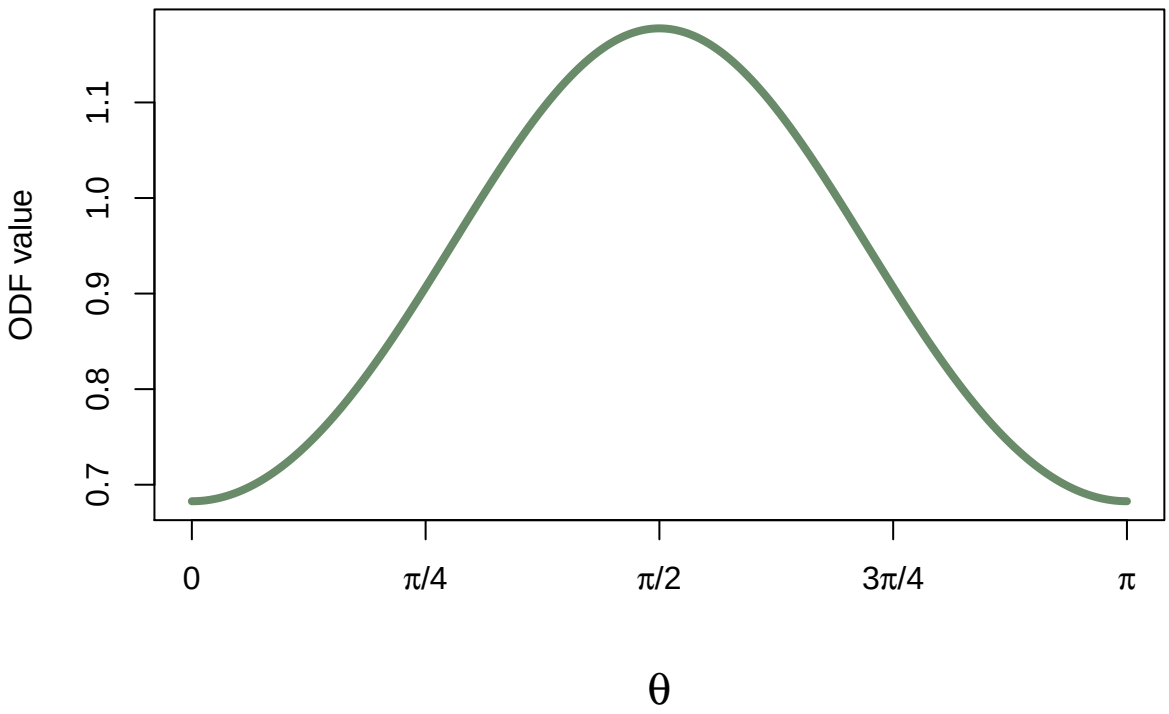
```
##                   variable name       value
## 1                             B  10.8000000
## 2          num_theta_intervals 256.0000000
## 3                         f(0)   4.3734097
## 4                            L   0.1565842
## 5          pi/4 - <<W(gamma)>>   0.1144541
## 6               <ln(f(theta))>   0.5637745
## 7             <P2(cos(theta))>   0.4927489
## 8             <P4(cos(theta))>   0.1101210
## 9                  free.energy   4.1868726
## 10              num_iterations  16.0000000
## 11 num_iter_F_did_NOT_decrease   0.0000000
```

```
###############  now run with a planar shaped initial condition

initial.f <- sin(theta.vec) * sin(theta.vec)  # planar shaped
max.num.iter <- 500  # this run will converge but requires more iterations

results <- calculate_ODF(initial.f, B, tau.conv, max.num.iter, min.iterations)
```

```
## [1] "run of   calculate_ODF    December 8, 2023  version"
```

```
f.for.plotting  <- results[[1]]  # plot if desired

line.color <- "darkseagreen4"
plot(theta.points.for.plot, f.for.plotting, lwd = 4, ,
        type = "l", col = line.color, xaxt = "n", xlab = ' ', ylab = 'ODF value')
# request tic marks and labels for the x-axis:
axis(1, at = c(0, pi/4,  pi/2, 3*pi/4,  pi),
         labels = c("0", expression(paste(pi,"/4")), expression(paste(pi,"/2")),
                          expression(paste("3",pi,"/4")),  expression(pi)))
title(main = paste("Fig. 11.  W(g) = sin(g) + 0.5 * P4(cos(g)),  planar shaped initial f" ),
cex.main = 1.1)
title(main = NULL, sub = expression(theta), cex.sub = 1.4)
```

Fig. 11. W(g) = sin(g) + 0.5 * P4(cos(g)), planar shaped initial f




```
# Fig. 11.  Legend: The coefficient of P4(cos(gamma)) for this W is positive but
# still have convergence.  All the positive even index coefficients for W being less
# than or equal 0 is sufficient to prove that the iteration always decreases the free
# energy  (for continuous ODFs),  but is not necessarily required.
# Here the initial ODF is a planar shaped discrete ODF.
# The free energy decreased at each iteration (num_iter_F_did_NOT_decrease = 0,
# see the results below)

vector.of.values <- results[[3]]
names.for.vector.of.values <- results[[4]]

# display these values using a data frame
df.for.values <- data.frame(names.for.vector.of.values,  vector.of.values,
                                                    stringsAsFactors = FALSE)
colnames(df.for.values) <- c("variable name", "value")
df.for.values
```

```
##                 variable name         value
## 1                           B  10.800000000
## 2        num_theta_intervals 256.000000000
## 3                        f(0)   0.682702542
## 4                           L   0.934201366
## 5        pi/4 - <<W(gamma)>>   0.002251166
## 6             <ln(f(theta))>   0.011910519
## 7          <P2(cos(theta))>  -0.067753403
## 8          <P4(cos(theta))>   0.002395321
```

```
## 9                   free.energy   4.240904305
## 10              num_iterations 141.000000000
## 11 num_iter_F_did_NOT_decrease   0.000000000
```

When the coefficient of P4(cos(gamma)) is a little larger, the behavior changes

With W(gamma) equal sin(gamma) + 1.0 * P4(cos(gamma)), free energy decrease at each iteration and convergence depends on the initial condition

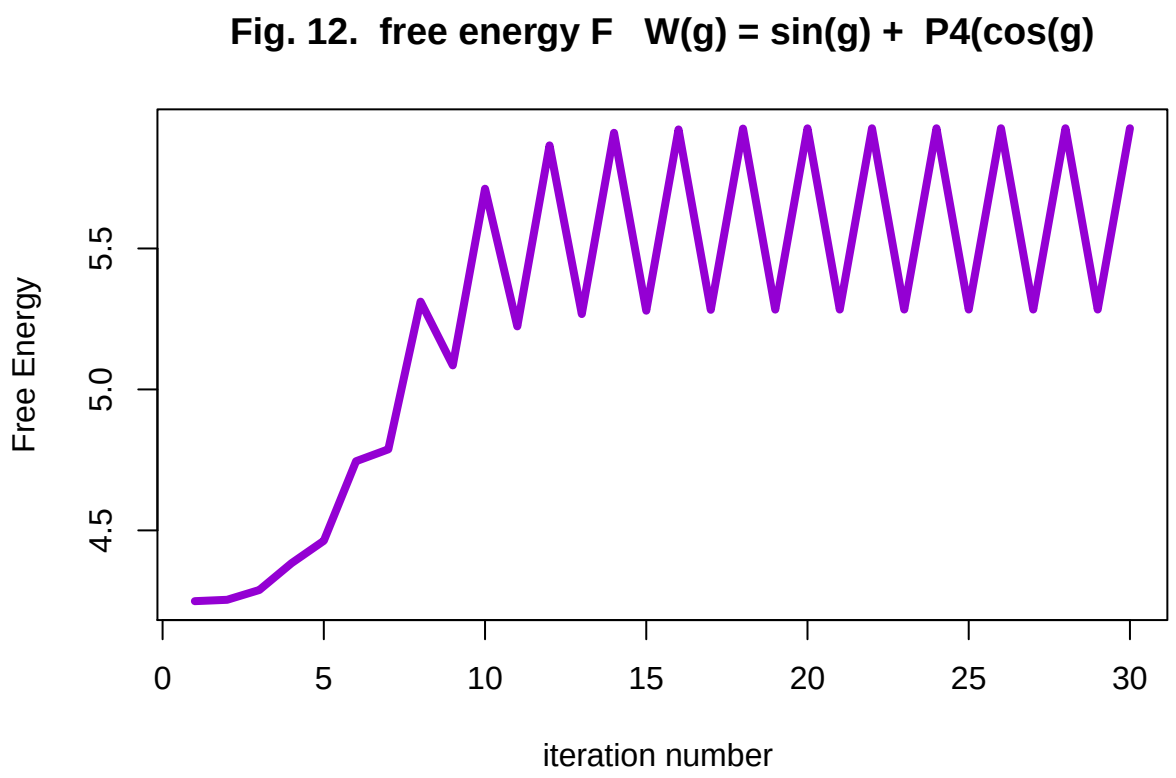
```
#########  With W(gamma) equal  sin(gamma) +  1.0 * P4(cos(gamma)))
#########  the behavior of the iteration depends on the initial condition

P4 <- function(x)  (35*x^4 - 30*x^2 + 3) / 8

N.theta.intervals <- 256
N.phi.intervals <- 2048

W <- function(gamma) {
   x <- cos(gamma)
   return(sin(gamma) +  1.0 * P4(x))
}

# since have a new W, need to do a new run of  compute_approx_ODF
calculate_ODF <- compute_approx_ODF(N.theta.intervals, N.phi.intervals, W)
```

```
## [1] "run of   compute_approx_ODF   December 8, 2023  version"
```

```
B <- 10.8
# have defined theta.vec previously
initial.f <- cos(theta.vec) * cos(theta.vec)  # axial shaped

tau.conv <- 1.0e-5  # the bound for the convergence criterion
max.num.iter <- 30  # maximum number of iterations allowed
min.iterations <- 10  # require having done  min.iterations  iterations before
#                                       start testing for convergence

results <- calculate_ODF(initial.f, B, tau.conv, max.num.iter, min.iterations)
```

```
## [1] "run of   calculate_ODF    December 8, 2023  version"
## iter =  2   F did not decrease   only printing for first 12 instances
## iter =  3   F did not decrease   only printing for first 12 instances
## iter =  4   F did not decrease   only printing for first 12 instances
## iter =  5   F did not decrease   only printing for first 12 instances
## iter =  6   F did not decrease   only printing for first 12 instances
## iter =  7   F did not decrease   only printing for first 12 instances
## iter =  8   F did not decrease   only printing for first 12 instances
## iter =  10   F did not decrease   only printing for first 12 instances
## iter =  12   F did not decrease   only printing for first 12 instances
## iter =  14   F did not decrease   only printing for first 12 instances
## iter =  16   F did not decrease   only printing for first 12 instances
## iter =  18   F did not decrease   only printing for first 12 instances
## [1] "WARNING iteration did not converge"
```

```
# plot the calculated free energy as a function of iteration number
free.energy.for.each.iteration <- results[[2]]

line.color <- "darkviolet"
plot(free.energy.for.each.iteration, lwd = 4, ,
       type = "l", col = line.color, xlab = 'iteration number ', ylab = 'Free Energy')
title(main = "Fig. 12.  free energy F   W(g) = sin(g) +  P4(cos(g)", cex.main = 1.2)
```




```
# Fig. 12.  Legend: The coefficient of P4(cos(gamma)) for this W is a little larger
# than the previous case.  Here the initial ODF is an axial shaped discrete ODF.
# The free energy increased at some iterations and the iteration failed to converge
# (even when run out to 2000 iterations).


vector.of.values <- results[[3]]
names.for.vector.of.values <- results[[4]]

# display these values using a data frame
df.for.values <- data.frame(names.for.vector.of.values,  vector.of.values,
                                                    stringsAsFactors = FALSE)
colnames(df.for.values) <- c("variable name", "value")
df.for.values
```

```
##                 variable name        value
## 1                           B  10.80000000
```

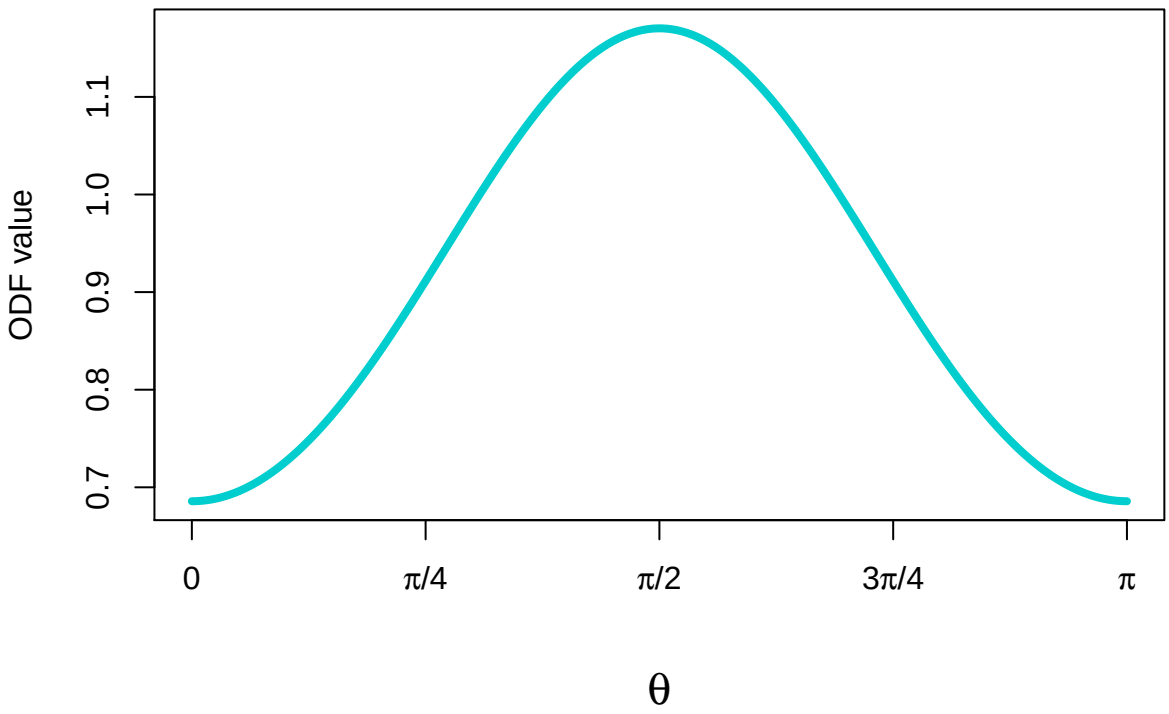
```
## 2          num_theta_intervals 256.00000000
## 3                        f(0)   0.03788411
## 4                           L   1.00000000
## 5          pi/4 - <<W(gamma)>>  -0.05196719
## 6               <ln(f(theta))>   1.40459712
## 7             <P2(cos(theta))>   0.26834577
## 8             <P4(cos(theta))>  -0.29911363
## 9                  free.energy   5.92637006
## 10              num_iterations  30.00000000
## 11 num_iter_F_did_NOT_decrease  18.00000000
```

```
###############  now run with a planar shaped initial condition

initial.f <- sin(theta.vec) * sin(theta.vec)  # planar shaped
max.num.iter <- 500  # this run will converge but requires more iterations

results <- calculate_ODF(initial.f, B, tau.conv, max.num.iter, min.iterations)
```

```
## [1] "run of   calculate_ODF    December 8, 2023  version"
```

```
f.for.plotting  <- results[[1]]  # plot if desired

line.color <- "cyan3"
plot(theta.points.for.plot, f.for.plotting, lwd = 4, ,
        type = "l", col = line.color, xaxt = "n", xlab = ' ', ylab = 'ODF value')
# request tic marks and labels for the x-axis:
axis(1, at = c(0, pi/4,  pi/2, 3*pi/4,  pi),
         labels = c("0", expression(paste(pi,"/4")), expression(paste(pi,"/2")),
                         expression(paste("3",pi,"/4")),  expression(pi)))
title(main = "Fig. 13.  W(g) = sin(g) + P4(cos(g)),  planar shaped initial f", cex.main = 1.1)
title(main = NULL, sub = expression(theta), cex.sub = 1.4)
```

Fig. 13. W(g) = sin(g) + P4(cos(g)), planar shaped initial f




```
# Fig. 13.  Legend: With a planar shaped initial f, the iteration converged!
# And the free energy decreased at each iteration!
#  (num_iter_F_did_NOT_decrease = 0, see the results below).
# Actually, this behavior (convergence, and decreasing free energy at
# each iteration, occurring with some initial conditions and not with
# others) when one or more of the positive even index coefficients in the
# expansion of W(gamma) in terms of Legendre polynomials is positive
# is not surprising (one can view the iteration as a dynamical system
# with the iteration a function mapping a member f of the set of ODFs
# to another member of the set of ODFs, and for many dynamical systems,
# the qualitative behavior of the sequence of iterates will depend on
# the region the initial value belongs to).
# When the coefficient of P4 was increased to 2, so
# W(g) = sin(g) + 2 * P4(cos(g)), then the iteration did not converge
# and there were increases in the free energy, for both the axial and planar
# shaped initial f (calculations not displayed here).

vector.of.values <- results[[3]]
names.for.vector.of.values <- results[[4]]

# display these values using a data frame
df.for.values <- data.frame(names.for.vector.of.values,  vector.of.values,
                                                    stringsAsFactors = FALSE)
colnames(df.for.values) <- c("variable name", "value")
df.for.values
```

```
##                   variable name         value
## 1                             B  10.800000000
## 2          num_theta_intervals 256.000000000
## 3                         f(0)   0.685718690
## 4                            L   0.969005263
## 5          pi/4 - <<W(gamma)>>   0.002134344
## 6               <ln(f(theta))>   0.011290295
## 7             <P2(cos(theta))>  -0.065976444
## 8             <P4(cos(theta))>   0.001641648
## 9                  free.energy   4.240914921
## 10              num_iterations 375.000000000
## 11 num_iter_F_did_NOT_decrease   0.000000000
```

= =