doi: 10.1038/nature05747

```
############ NPMR: Gaussian kernel fitting #################
# y is a vector of single response variable
# x is a matrix of predictors; must be same number of x's and y's (dim(x)[1] =
length(y))
# with just one predictor, x can be a vector
# tolerance is a vector of the kernels' Gaussian SDs; must be same as the number
of predictors (length(tolerance) = dim(x)[2])

# Function setupDist.kernelGauss
# Calculates the distance matrix for each predictor. For a single predictor,
distances are in a NxN matrix, where
# N is the number records (number of data points). The function outer creates
this matrix in one step.
# The complete set of distance matrices (one per predictor) are returned as a 3D
array

setupDist.kernelGauss=function(x)
{
 predictors=dim(x)[2]
 if(is.null(predictors)) predictors=1

 records=dim(x)[1]
 if(predictors==1) records=length(x)

 # Declare blank 3D array
 distmatrix=array(dim=c(predictors,records,records))

 # Use outer to create one distance matrix for each predictor
 if(predictors==1) distmatrix[1,,]=outer(x,x,"-")
 else for(i in 1:predictors) distmatrix[i,,]=outer(x[,i],x[,i],"-")

 return(distmatrix)
}


# Function kernelGauss
# Calculates a Gaussian kernel for a single response variable y, given
tolerances for each Gaussian and a matrix of distances (one for each predictor,
as created
# by the function setupDist.kernelGauss). The weightings from each predictor are
multipled together to produce the total weight, as in McCune's NPMR

kernelGauss=function(y,tolerance,distmatrix,elimself=TRUE)
{
 predictors=length(tolerance)
 records=length(y)

 for(i in 1:predictors)
  {
   wt=dnorm(distmatrix[i,,],mean=0,sd=tolerance[i])    # Gaussian weightings
calculated from distance matrix for one predictor
   if(elimself) diag(wt)=0                             # Diagonal set to zero so
prediction at a point does not use that point

   if(i==1) totalwt=wt
   else totalwt=totalwt*wt                             # This is total weight,
product of weightings from each predictor
  }

 y.By.WtProduct=t(apply(totalwt,1,vectMult,v2=y))      # Uses R function apply
and my function vectMult (at bottom) to multiply
                                                       # response y by each row
of Gaussian weights
```

```
 prediction=rowSums(y.By.WtProduct)/rowSums(totalwt)   # This is eq. 3 in McCune
paper

 return(prediction)
}


# Function llike.NPMR. Likelihood: probability of observing the response's y
given the prediction from the NPMR kernel.
# Three options allowed here: Poisson, negative binomial, and Gaussian.
# Parameters are the tolerances of the kernel; function kernelGauss is
calculated using these parameters and the distance matrix,
# passed as the argument dist. The negative binomial and normal require an
additional parameter, called k here, passed as the final parameter.
# k serves as the clumping parameter of the negative binomial or the standard
deviation of the Gaussian

# Other likelihood functions could be handled.

# current options for argument link are "poisson", "negbinom", "binom", and
"normal"

llike.NPMR=function(param,y,dist,link="poisson")
{
 noparam=length(param)

 if(link=="poisson" | link=="binom") tolerance=param
 else
  {
   tolerance=param[-noparam]
   k=param[noparam]
  }

 if(length(param[param<=0])>0) return(-Inf)
# Avoid negative parameters, which crash likelihood

 pred=kernelGauss(y=y,tolerance=tolerance,distmatrix=dist)
# Calculate kernel given distmatrix and tolerances

 if(link=="binom" & (length(pred[pred>=1])>0 | length(pred[pred<=0])>0))
return(-Inf)

 if(link=="poisson") llike=dpois(y,lambda=pred,log=T)
 else if(link=="binom") llike=dbinom(y,size=1,prob=pred,log=T)  # Calculate log-
likelihood with R's density functions
 else if(link=="negbinom") llike=dnbinom(y,mu=pred,size=k,log=T)  # Calculate
log-likelihood with R's density functions
 else if(link=="normal") llike=dnorm(y,mean=pred,sd=k,log=T)

 totallike=sum(llike)
# Total log-liklihood

 if(counter%%10==1) cat(counter,round(param,2),round(totallike,4),"\n")
# Output current parameters and likelihood to screen
 counter<<-counter+1

 return(sum(totallike))
}


# Function fit.kernel.optim. Finds maximum-likelihood tolerances for the
multiplicative Gaussian kernel, using R's function optim.
```

```
# Poisson model takes only one parameter per predictor; others need the extra
scale parameter.

fit.kernel.optim=function(response,predictors,start,linkfunc="poisson",fitmethod
="Nelder-Mead",maxstep=10000)
{
 assign("counter",1,pos=1)                          # Creating counter here
allows regular output from within likelihood function
 distmatrix=setupDist.kernelGauss(predictors)       # Calculate distance matrix
just once at the start (it never changes)
 maxdist=apply(distmatrix,1,max)

 if(length(start)==1)

fit=optimize(f=llike.NPMR,interval=c(0,maxdist),y=response,dist=distmatrix,link=
linkfunc,maximum=TRUE)
 else

fit=optim(par=start,fn=llike.NPMR,y=response,dist=distmatrix,link=linkfunc,contr
ol=list(fnscale=-1,maxit=maxstep),method=fitmethod)

 if(length(start)==1) bestpar=fit$maximum
 else bestpar=fit$par

 pred=kernelGauss(y=response,tolerance=bestpar,dist=distmatrix)

 return(list(fit=fit,prediction=pred))
}


# Function llike.NPMR.Gibbs is the same likelihood function, as in llike.NPMR,
but it has to be structured differently for use by the
# Gibbs sampler. The sampler tests one parameter at a time, and the parameter
being tested has to be submitted first (this is how
# the function metrop1step works).

# maxtolerance is the maximum allowed tolerance. I don't know whether this
matters, but tolerance for a predictor with no
# impact on the response can drift off to absurdly high number

llike.NPMR.Gibbs=function(test,other,whichtest,y,dist,maxtolerance,link="binom",
k=NULL)
{
 if(whichtest!="k")
  {
   tolerance=numeric(length(other)+1)
   tolerance[whichtest]=test
   tolerance[-whichtest]=other
  }
 else
  {
   k=test
   tolerance=other
  }

 if(length(tolerance[tolerance<=0])>0) return(-Inf)              # All
parameters must be positive
 if(!is.null(k)) if(k<=0) return(-Inf)
 if(length(tolerance[tolerance>maxtolerance])>0) return(-Inf)    # Disallow any
tolerance above maxtolerance

 pred=kernelGauss(y,tolerance,dist)
 if(link=="binom" & length(pred[pred>=1])>0) return(-Inf)

 if(link=="poisson") llike=dpois(y,lambda=pred,log=T)
 else if(link=="binom") llike=dbinom(y,size=1,prob=pred,log=T)  # Calculate log-
likelihood with R's density functions
 else if(link=="negbinom" & !is.null(k)) llike=dnbinom(y,mu=pred,size=k,log=T)
 else if(link=="normal" & !is.null(k)) llike=dnorm(y,mean=pred,sd=k,log=T)

 totallike=sum(llike)
 if(is.na(totallike)) return(-Inf)

 return(totallike)
}


# Function fit.kernel.Gibbs uses Gibbs sampler to generate posterior
distributions of the parameters -- the Gaussian tolerances
# as well as negative binomial clumping parameter or normal SD. The response
vector (y above) is passed as the argument response,
# and the matrix of predictors as the argument predictors. The last 3 parameters
tell the Gibbs sampler how long to run (steps),
# which to use (after burn-in), and how frequently progress should be printed to
the screen (showstep).
# The predicted kernel can be saved every so often to get confidence on the
predictions, by setting savestep (savestep=100 means
# save every 100th step)

# The kernel has to be calculated once for every Gibbs step, for each parameter.
To save the kernel at every step adds one more
# calculation. With 3 predictors, the negative binomial k parameter, and saving
the kernel at each step is 5 calculations. To run
# 4000 Gibbs steps thus requires 20,000 kernel calculation. Each kernel
calculation requires a normal density at every distance,
# which is the square of the number of points in the data.

# fit.kernel.Gibbs starts with the distance matrix. So
fit.kernel.Gibbs.predictors starts with the predictors themselves and calculates
# distance matrix.

fit.kernel.Gibbs.predictors=function(response,predictors,start,linkfunc="binom",
savestep=NULL,steps=4500,burnin=500,showstep=250)
{
 distmatrix=setupDist.kernelGauss(predictors)

fit=fit.kernel.Gibbs(response=response,distmatrix=distmatrix,start=start,linkfun
c=linkfunc,savestep=savestep,steps=steps,burnin=burnin,showstep=showstep)

 return(fit)
}


fit.kernel.Gibbs=function(response,distmatrix,start,linkfunc="binom",savestep=NU
LL,steps=4000,burnin=500,showstep=100)
{
 notoler=dim(distmatrix)[1]
 noparam=length(start)

 tolerance=matrix(nrow=steps,ncol=notoler)
 tolerance[1,]=start[1:notoler]
 accept=rep(0,notoler)

 k=numeric()
 if(linkfunc!="poisson" & linkfunc!="binom") k[1]=start[noparam]
 else k=1

 scale=tolerance[1,]
 kscale=.5
 accept=rep(0,notoler)
 kaccept=0

 maxdist=apply(distmatrix,1,max)

 like=numeric()
 prediction=matrix(nrow=steps,ncol=length(response))

 like[1]=llike.NPMR.Gibbs(tolerance[1,1],tolerance[1,-
1],whichtest=1,y=response,dist=distmatrix,link=linkfunc,k=k[1],maxtolerance=5*ma
xdist)
 cat("step 1: ",round(tolerance[1,],2),round(k[1],3),"--
",round(scale,2),round(kscale,2),"--",round(accept,2),round(kaccept,2),"--
",round(like[1],1),"\n")

 for(i in 2:steps)                     # Loop through the Gibbs sampler
  {
   for(j in 1:notoler)                 # Loop through the tolerance parameters,
updating one at a time
    {
     testparam=tolerance[i-1,]
     if(j>1) testparam[1:(j-1)]=tolerance[i,1:(j-1)]


metropResult=metrop1step(func=llike.NPMR.Gibbs,start.param=testparam[j],scale.pa
ram=scale[j],adjust=1.02,target=0.25,other=testparam[-j],whichtest=j,
                              k=k[i-
1],y=response,dist=distmatrix,link=linkfunc,maxtolerance=5*maxdist)
              # Update each tolerance parameter

     tolerance[i,j]=metropResult[1]
     scale[j]=metropResult[2]
     accept[j]=accept[j]+metropResult[3]    # Keep track of acceptance rate,
which will converge on target=0.25
    }

   if(linkfunc!="poisson" & linkfunc!="binom")
    {
     metropResult=metrop1step(func=llike.NPMR.Gibbs,start.param=k[i-
1],scale.param=kscale,adjust=1.02,target=0.25,other=tolerance[i,],whichtest="k",

y=response,dist=distmatrix,link=linkfunc,maxtolerance=5*maxdist)
              # Update the additional parameter, k (clumping parameter of
negative binomial or normal SD)

     k[i]=metropResult[1]
     kscale=metropResult[2]
     kaccept=kaccept+metropResult[3]
    }
   else k[i]=1

#   browser()
   like[i]=metropResult[4]
   if(i%%showstep==0) cat("step", i,": ",round(tolerance[i,],2),round(k[i],3),"-
-",round(scale,2),round(kscale,2),"--",
                           round(accept,2)/(i-1),round(kaccept,2)/(i-1),"--
",round(like[i],1),"\n")
                                                 # output to screen every
showstep steps
```

```
   if(!is.null(savestep))
     if(i%%savestep==0)
prediction[i,]=kernelGauss(response,tolerance[i,],distmatrix)   # save predicted
kernel every savestep steps
  }

# browser()
 tolerance=as.matrix(tolerance[-(1:burnin),])
 meantoler=colMeans(tolerance)                            # Calculate various
descriptions of the Gibbs chain
 meank=mean(k[-(1:burnin)])

 mediantoler=apply(tolerance,2,median)
 mediank=median(k[-(1:burnin)])

 CItoler=apply(tolerance,2,quantile,prob=c(.025,.975))
 CIk=quantile(k[-(1:burnin)],prob=c(.025,.975))

 best=kernelGauss(response,meantoler,distmatrix)                 # Best-fit
kernel's prediction

 prediction=prediction[-(1:burnin),]
 include=which(!is.na(prediction[,1]))
 prediction=prediction[include,]
 lower=apply(prediction,2,quantile,prob=.025)                    # Prediction
interval at each point
 upper=apply(prediction,2,quantile,prob=.975)

# browser()
 return(list(fulltoler=tolerance,fullk=k[-
(1:burnin)],tolerance=meantoler,CItolerance=CItoler,k=meank,CIk=CIk,
             mediantolerance=mediantoler,mediank=mediank,like=like[-
(1:burnin)],bestpred=best,lowerPred=lower,upperPred=upper,
           predictions=prediction))
}



# Function vectMult calculates product of two vectors, producing a 3rd vector of
same length. Used inside apply in function kernelGauss.
# R's function prod might do this?

vectMult=function(v1,v2) return(v1*v2)


# Metropolis algorithm used in the Gibbs sampler.
# Adjusting the scale parameter to achieve a target acceptance rate is H.
Muller-Landau's trick

# Takes a single metropolis step on a single parameter for any given likelihood
function. The likelihood function
# is passed as the argument func.
# The arguments start.param and scale.param are atomic (single values), as are
adjust and target.
# The ellipses handle all other arguments to the likelihood function. The
function func must accept the test
# parameter as the first argument, plus any additional arguments which come as
the ellipses.

# Note the metropolis rule: if rejected, the old value is returned to be re-
used. The return value
```

```
# includes a one if accepted, zero if rejected, and also the likelihood at the
final parameter value.

# The step size, refered to as scale.param, is adjusted following Helene's rule:
# For every acceptance, scale.param is multiplied
# by adjust, which is a small number > 1 (1.01, 1.02, 1.1 all seem to work). For
every rejection, scale.param
# is multiplied by (1/adjust) raised to a power (AdjExp) that is based on the
target acceptance rate.
# When the target acceptance rate is 0.25, which is recommended for any model
with > 4 parameters,
# AdjExp=3. It's easy to see how this system arrives at an equilibrium
acceptance rate=target.

# The program calling metrop1step has to keep track of the scaling parameter:
submitting it each time
# metrop1step is called, and saving the adjusted value for the next call. Given
many parameters, a
# scale must be stored separately for every one.

# The return value is a vector of 4: first the new parameter value, second the
new scale (step size),
# third a zero or a one to keep track of the acceptance rate, and finally the
likelihood.

#  metrop1step=function(func,start.param,scale.param,adjust,target,...)
#  {
#   origlike=func(start.param,...)
#   newval=rnorm(1,mean=start.param,sd=scale.param)
#   newlike=func(newval,...)
#
#   AdjExp=(1-target)/target
#
#   likeratio=exp(newlike-origlike)
#   if(runif(1)<likeratio)
#    {
#     newscale=scale.param*adjust^AdjExp
#     return(c(newval,newscale,1,newlike))
#    }
#   else
#    {
#     newscale=scale.param*(1/adjust)
#     return(c(start.param,newscale,0,origlike))
#    }
#
#  }



#### Graphing output of NPRM fit ####
# Requires the predictors and observed (=response), the link, and output, which
is what fit.kernel.Gibbs produces.
# If CI==TRUE, confidence intervals are added to the prediction. If
limits==TRUE, intervals for all observed data
# are also graphed.
# Whichgraph indicates which of the predictors is graphed. It must be a number
<= the number of predictors.
# Output is a graph of the observed, along with prediction, as a function of the
one predictor selected.

graph.NPMR=function(predictors,observed,whichgraph,output,link="poisson",xlabel=
"x",CI=TRUE,limits=TRUE)
{
 x=predictors[,whichgraph]

 if(link=="poisson")
  {
   upper=qpois(.975,lambda=output$bestpred)
   lower=qpois(.025,lambda=output$bestpred)
  }
 else if(link=="negbinom")
  {
   upper=qnbinom(.975,mu=output$bestpred,size=output$k)
   lower=qnbinom(.025,mu=output$bestpred,size=output$k)
  }
 else if(link=="normal")
  {
   upper=qnorm(.975,mean=output$bestpred,sd=output$k)
   lower=qnorm(.025,mean=output$bestpred,sd=output$k)
  }

 if(limits) yupper=max(c(upper,observed,output$bestpred))
 else yupper=max(c(observed,output$bestpred))

 plot(x,observed,ylim=c(0,yupper),xlab=xlabel)

 ord=order(x)
 if(limits) lines(x[ord],upper[ord],col="gray",cex=.75,pch=16)
 if(limits) lines(x[ord],lower[ord],col="gray",cex=.75,pch=16)

 points(x,output$bestpred,col="blue",pch=16)
 if(CI) segments(x,output$lowerPred,x,output$upperPred,col="red")
 points(x,observed)

 abline(lm(observed~x))
}
```