Appendix C

Reference Source Code

C.1 Kriging Variance

The implementation of kriging variance computation (equation 8.1) used in this thesis, in the R statistical computing language, using the geoR library is:

```
# This definition of kriging variance taken from
# equation (1) of Delmelle \textit{et al.} (2009), and
# is compatable with equation 4.18 in Spatial Statistics by
# Ripley. It appears to be a constant shift off the kriging
# variance computed by the geoR krige.conv method
krige.var <- function(dcoords, loci, kc){
  cs <- cov.spatial(obj=loccoords(coords=dcoords, locations=loci),
          cov.model=kc$cov.model, cov.pars=kc$cov.pars,
          kappa=kc$cov.kappa)
  vcinv <- varcov.spatial(coords=dcoords, cov.model=kc$cov.model,
             cov.pars=kc$cov.pars, kappa=kc$kappa, nugget=kc$nugget,
             inv=TRUE)$inverse
  sigmak <- NULL;
  sigmasq <- kc$cov.pars[1]
  for(i in seq(1,nrow(loci))){
    v <- sigmasq - t(cs[,i]) %*% vcinv %*% t(t(cs[,i]))
    sigmak <- rbind(sigmak,v)
  }
  rm(vcinv, cs)
  return(sigmak)
}
```

The for-loop in this function involves the bulk of computation, but can be parallelized like so:

```
# argument 4 is a 'cluster' made with a command like makeForkCluster(N)
krige.var.par <- function(dcoords, loci, kc, c1){
  cs <- cov.spatial(obj=loccoords(coords=dcoords, locations=loci),
                    cov.model=kc$cov.model, cov.pars=kc$cov.pars,
                    kappa=kc$cov.kappa)
  vcinv <- varcov.spatial(coords=dcoords, cov.model=kc$cov.model,
                          cov.pars=kc$cov.pars, kappa=kc$kappa,
                          nugget=kc$nugget, inv=TRUE)$inverse
  sigmasq <- kc$cov.pars[1]
  sigmak <- parLapply(c1, seq(1,nrow(loci)),
                      function(i){ sigmasq - t(cs[,i]) %*%
                                   vcinv %*% t(t(cs[,i])) })
  rm(vcinv, cs)
  return(as.numeric(sigmak))
}
```

C.2 Path Loss Prediction

The following code provides a ruby class that models a “path”, and provides implementations (or wrappers for those with outside/reference implenentations) of the path loss models studied in chapter 3. As it is defined here, a path must be at least two points (transmitter and receiver), although the terrain models will need the path to include a number of intermediary points and their elevations. In order to conserve space only those models with substantial complexity have been included.

```
class Path

  ############### HELPER FUNCTIONS #######################

  # the angle, in degrees between this site and a given site, in the zenith
  # i.e., the angle of the LOS path from the perspective of the transmitter
  # if you want instead, the angle between the street and the line of sight path
  # (i.e.,the angle from the receiver's perspective), then this is just the
  # ascension negated (they are alternate internal angles, which by definition
  # are congruent)
  def ascension(x,i)
    h1 = x.z
    h2 = i.z
    dh = h1 - h2
    d = distance(x,i)
    -(rad_to_deg(atan(dh/d)))
  end

  # path LOS bit-vector calculation as in splat.cpp:PlotPath()
  #
  # returns a path-sized array where the i^th element is true if there's
  # no obstruction (i.e., los) and false otherwise. the 0^th element
  # is nil since it is meaningless (los from transmitter to itself)
  #
  # splat.cpp does some cos() comparison voodoo I don't understand.
  # here I'm just using plain-old right-triangle trigonometry.
  # which is probably slow, but probably correct.
  #
  # path is an array of Site objects. the first is the transmitter.
  # fresnel is the fraction of the fresnel zone that can be obscured before
  #          we deem the path as non-los. nil means don't bother thinking about
  #          fresnel (same as fresnel = 0.5 AFAICT)
  # frequency in Mhz
  def los(f,fresnel)
    tx = self.tx
    agl_tx = tx.z               # in meters
    ret = Array.new(self.length+1,nil)
    w = freq_to_wavelength(f/1000)      # in meters

    return ret if tx.ele.nil?

    (1..self.length-1).each{ |i|

      rx = self[i]

      next if rx.ele.nil?

      d = distance(rx,tx)*1000          # in meters
      agl_rx = rx.ele+rx.h              # in meters
```

```
    x = agl_tx - agl_rx                 # negative if rx is above tx
    alpha = acos(x/sqrt(x**2 + d**2)) # angle from rx to tx in radians

    los = true

    if i > 1
      # check to see if any point between this rx and the tx is in the way
      (1..i-1).each{ |j|
        int = self[j]
        next if int.ele.nil?
        di = distance(int,tx)*1000       # in meters
        agl_int = int.ele+int.h          # in meters
        xi = agl_tx - agl_int            # in meters

        unless fresnel.nil?
          # a whole bunch of calculations...
          d_a = distance(int,rx)*1000                    # meters
          d_b = distance(int,tx)*1000                    # meters
          # length of line from rx to tx
          r_ab = sqrt((d_a+d_b)**2 + (agl_tx-agl_rx)**2)
          # angle from rx to tx in radians (always pos)
          angle_ab = acos((d_a+d_b) / r_ab)
          # angle should be neg if rx is higher than tx
          angle_ab = angle_ab * -1 if agl_tx < agl_rx
          # radius of fresnel lens at int
          r_f = sqrt((1*w*d_a*d_b) / (d_a+d_b))
          # distance from rx to los point above/below int
          r_los = d_a/cos(angle_ab)
          # "width" of fresnel lens from receiver's view
          angle_f = acos(r_los/sqrt(r_f**2 + r_los**2))
          # length of line from rx to int
          r_ai = sqrt((agl_int-agl_rx)**2 + d_a**2)
          # angle from rx to int (always pos)
          angle_ai = acos(d_a/r_ai)
          # angle should be neg if rx is higher than int
          angle_ai = angle_ai * -1 if agl_int < agl_rx

          # proceed in calculating fraction
          f = nil
          # zone isn't obscured at all
          if angle_ai < (angle_ab - angle_f)
            f = 0.0
          # zone is totally obscured
          elsif angle_ai > (angle_ab + angle_f)
            f = 1.0
          # if the zone is partially obscured...
          else
            # if we're exactly at the LOS center line
            if angle_ai == angle_ab
              f = 0.5
            elsif angle_ai > angle_ab
              # if we're below the LOS center line
              f = 0.5 - ((angle_ai-angle_ab) / (2*angle_f))
            else
              # if we're above the LOS center line
              f = 0.5 + ((angle_ab-angle_ai) / (2*angle_f))
            end
            # need to invert if we're working with neg angles
            f = 1 - f if angle_ab < 0
          end

          los = false if f > fresnel
        else
          # angle from rx to int in radians
          gamma = acos(xi/sqrt(xi**2 + di**2))
          if gamma >= alpha
            los = false
```

```
                break
              end
            end
          }
        end
        ret[i] = los
      }
      return ret
    end

    # Hata-based prop models assume that:
    # (a) the tx is higher than the rx
    # (b) rx is in [1,10]
    # (c) tx is in [30,10]
    #
    # Here, we subtract off the minimum so
    # that the heights are relative.
    #
    # Then, we swap them if the rx was higher (since
    # loss is proportional, we'll just pretend
    # we're transmitting from the rx to tx and
    # calculate the loss, or something).
    #
    # We hard-code the rx to 1.0 (and adjust tx as such)
    #
    # Finally, we crudely lower or raise the resulting
    # tx to make sure it's in the right range.
    def hata_fix_heights(h1,h2)
      m = [h1,h2].min
      h1 -= m
      h2 -= m
      h2,h1 = [h1,h2].sort
      h2 = 1.0
      h1 -= 1.0
      h1 = [h1,200.0].min
      h1 = [h1,30.0].max
      return h1,h2
    end

    ##### Path Loss Model Functions - All of these return an array of Path Loss
    ##### Components (up to 4 of them) which are assumed to be attenuation in dB
    ##### (i.e. positive means it is loss and negative means it is gain. Aside
    ##### from model specific parameters, they should all "work" the same. They
    ##### also all log warning messages to @warn and it's expected you use the
    ##### warnage function (above) to clear out this array after each is used.

    ############### BASIC MODELS ########################

    # Simplified Egli Model
    #
    # Egli, John J. (Oct. 1957). "Radio Propagation above 40 MC over Irregular
    # Terrain". Proceedings of the IRE (IEEE) 45 (10): 1383 1391. ISSN 0096-8390.
    #
    # Simplified version due to:
    #
    # Deslisle G.Y., Lefevre J., Lecours M., and Chouinard, J. Propagation loss
    # prediction: a comparative study with application to the mobile radio channel.
    # IEEE Trans. Veh. Tech. 1985. 26. 4. p. 295-308.
    #
    # This version presented in:
    #
    # Les Barclay. Propagation of Radiowaves. IEE. 2003. p. 209
    #
    # f is in MHz
    def egli(f)
      hb = self.tx.h # m
      hm = self.rx.h # m
```

```
    @warn.push "Frequency #{f} is out of the Egli Model's coverage" if f > 3000
      or f < 30
    @warn.push "Mobile receiver height lies at model discontinuity" if hm == 10
    d = distance(self.tx, self.rx) # Km
    lm = 76.3 - ((hm < 10) ? 10.0*log10(hm) : 20.0*log10(hm))
    l = 40.0*log10(d) + 20.0*log10(f) - 20.0*log10(hb)
    lfs = freespace(f).sum
    [(l < lfs) ? lfs : l] # use freespace if our prediction is less than it
  end

  # Walfish-Ikegami model
  #
  # Ikegami proposed calculating the diffraction over each building in a path.
  # The Walfish-Ikegami model assumes a regular grid of rectangular buildings,
  # but otherwise makes the same computations.
  #
  # From: Les Barclay. Propagation of Radiowaves. IEE. 2003. p. 197
  #
  # (and numerous others)
  #
  # f is in MHz
  # los is boolean (line-of-sight)
  # h1 is in m
  # h2 is in m
  # hb is the nominal height of building roofs in m
  # b is the nominal building separation in m
  # w is the nominal street width in m
  # phi is the angle of incident wave with respect to street in degrees
  # city_size can be :medium or :large
  #
  # Default parameters provided on p. 152 of Barclay
  #
  # For 800 to 2000 MHz
  def walfish(f,los,hb,b=20.0,w=10.0,phi=90,city_size=:medium)
    h1 = self.tx.h
    h2 = self.rx.h
    d = distance(self.tx, self.rx)
    model_name = "Walfish-Ikegami"
    @warn.push "#{f} MHz is outside the #{model_name} model's coverage"
      if f > 2000.0 or f < 800.0
    @warn.push "#{d} Km is further than the #{model_name} model can support"
      if d > 5 or d < 0.02
    if los
      return [freespace(f).sum,6*log10(d*50)]
    else
      dhb = h1 - b
      dhm = hb - h2

      # First, calculate the Roof-to-Street diffraction and scatter loss, rts
      # ori is the orientation loss
      ori = nil
      if phi >= 0.0 and phi <= 35.0
        ori = -10.0 + 0.354*phi
      elsif phi >= 25.0 and phi <= 55.0
        ori = 2.5 + 0.075*(phi-35)
      elsif phi >= 55.0 and phi <= 90.0
        ori = 4.0 - 0.144*(phi-55)
      end
      rts = -16.9 - 10*log10(w) + 10*log10(f) + 20*log10(dhm.abs) + ori

      # Then, calculate the Multiscreen Diffraction loss, msd
      # bsh is the shadowing gain that occurs when the base-station is higher
      # than the rooftops in the msd calculation
      bsh = (dhb <= 0) ? 0 : -18.0*log10(1 + dhb.abs)
      ka = 54.0
      if dhb <= 0 and d >= 0.5
        ka += 0.8*dhb.abs
```

```
    elsif dhb <= 0 and d < 0.5
      ka += 0.8*dhb.abs*(d/0.5)
    end
    # kd is the distance factor in msd calculation
    kd = 18.0
    kd += 17*(dhb.abs/h1) if dhb <= 0.0
    # kf is the frequency factor in msd calculation
    kf = -4.0
    kf += (city_size == :large) ? 1.5*(f/925.0 - 1) : 0.7*(f/925.0 - 1)
    msd = bsh + ka + kd*log10(d) + kf*log10(f) - 9*log10(b)

    # Finally, return the calculated path loss if it seems legal
    if rts + msd >= 0.0
      return [freespace(f).sum, rts, msd]
    else
      return [freespace(f).sum]
    end
  end
end

# COST-Hata/COST-231 model/Extended Hata Model
#
# http://www.iucaf.org/sschool/procs/propag.pdf
# http://en.wikipedia.org/wiki/COST_Hata_model
#
# f is in MHz
# city_size can be :medium, :large
# d is in km
# h1 & h2 are in m
#
# For 1500 to 2000 MHz
def cost_hata(f, city_size=:medium)
  h1 = self.tx.h
  h2 = self.rx.h
  d = distance(self.tx, self.rx)

  model_name = "Cost-231/Cost-Hata"
  @warn.push "#{f} MHz is outside the #{model_name} model's coverage"
    if f > 2000.0 or f < 1500.0
  @warn.push "#{d} Km is further than the #{model_name} model can support"
    if d > 20 or d < 1
  @warn.push "#{h1} m (tx-height) is too high or low for the model"
    if h1 < 30 or h1 > 200
  @warn.push "#{h2} m (tx-height) is too high or low for the model"
    if h2 < 1 or h2 > 10

  h1,h2 = hata_fix_heights(h1,h2)
  # crudely "round" down or up the freq
  f = [f,1500.0].max
  f = [f,2000.0].min

  a = (1.1*log10(f) - 0.7)*h2 - (1.56*log10(f) - 0.8)
  c = (city_size == :large) ? 3.0 : 0.0
  [46.33 + 33.9*log10(f) - 13.82*log10(h2) - a + (44.9
    - 6.55*log10(h2))*log10(d) + c]
end

# Hata-Okumura Model
#
# http://w3.antd.nist.gov/wctg/manet/calcmodels_dstlr.pdf
# http://w3.antd.nist.gov/cgi-bin/req_propcalc_tar.pl
# http://en.wikipedia.org/wiki/Hata_Model_for_Urban_Areas
#
# f is in Mhz
# h1 & h2 are in m
# d is in km
# city size can be :open, :suburban, :medium, :large
```

```
#
# For 150-1500 MHz
def hata(f, city_size=:medium, suppress_warnings=false, dont_fix_heights=false)
  h1 = self.tx.h
  h2 = self.rx.h
  d = distance(self.tx, self.rx)
  unless suppress_warnings
    model_name = "Hata-Okumura"
    @warn.push "#{f} MHz is outside the #{model_name} model's coverage"
      if f > 1500.0 or f < 150.0
    @warn.push "#{d} Km is further than the #{model_name} model can support"
      if d > 10 or d < 1
    @warn.push "#{h1} m (tx-height) is too high or low for the model"
      if h1 < 30 or h1 > 200
    @warn.push "#{h2} m (tx-height) is too high or low for the model"
      if h2 < 1 or h2 > 20
  end

  h1,h2 = hata_fix_heights(h1,h2) unless dont_fix_heights
  # crudely "round" down or up the freq
  f = [f,150.0].max
  f = [f,2000.0].min

  a = (city_size == :large) ? 3.2*log10(pow(11.75*h2,2.0)) - 4.97 :
    (1.1*log10(f) - 0.7)*h2 - (1.56*log10(f) - 0.8)
  k = 0.0
  if city_size == :suburban
    k = 2*pow(log10(f/28.0),2.0) + 5.4
  elsif city_size == :open
    k = 4.78*pow(log10(f),2.0) - 18.33*log10(f) + 40.94
  end
  [69.55 + 26.16*log10(f) - 13.82*log10(h1) - a +
    (44.9-6.55*log10(h1))*log10(d) - k]
end

# ITU-R/CCIR Model
#
# building percent should be in [0,100]
# f is in Mhz
# h1 & h2 are in m
# d is in km
def itu_r(f, building_percent=20.0)
  h1 = self.tx.h
  h2 = self.rx.h
  d = distance(self.tx, self.rx)
  model_name = "ITU-R/CCIR"
  @warn.push "#{f} MHz is outside the #{model_name} model's coverage"
    if f > 2000.0 or f < 1500.0
  @warn.push "#{d} Km is further than the #{model_name} model can support"
    if d > 10 or d < 1
  @warn.push "#{h1} m (tx-height) is too high or low for the model"
    if h1 < 30 or h1 > 200
  @warn.push "#{h2} m (tx-height) is too high or low for the model"
    if h2 < 1 or h2 > 10

  h1,h2 = hata_fix_heights(h1,h2)
  # crudely "round" down or up the freq
  f = [f,1500.0].max
  f = [f,2000.0].min

  a = (1.1*log10(f) - 0.7)*h2 - (1.56*log10(f) - 0.8)
  b = (building_percent == 0.0) ? 0.0 : 30 - 25*log10(building_percent)
  [69.55 + 26.16*log10(f) - 13.82*log10(h1) - a +
    (44.9 - 6.55*log10(h1))*log10(d) - b]
end

# Hata-Davidson Model
```

```
#
# http://w3.antd.nist.gov/wctg/manet/calcmodels_r1.pdf
#
# f is in MHz
# city_size is same as for hata
# this is just hata with some corrections for long links mostly
def hata_davidson(f, city_size=:medium)
  h1 = self.tx.h
  h2 = self.rx.h
  d = distance(self.tx, self.rx) # in km

  model_name = "Hata-Davidson"
  @warn.push "#{d} Km is further than the #{model_name} model can support"
    if d > 300 or d < 1
  @warn.push "#{h1} m (tx-height) is too high or low for the model"
    if h1 < 30.0 || h1 > 2500.0
  @warn.push "#{f} MHz is outside the #{model_name} model's coverage"
    if f > 1500.0 or f < 150.0
  @warn.push "#{h2} m (rx-height) is too high or low for the model"
    if h2 < 1 or h2 > 20

  a = (d >= 20) ? 0.62137*(d - 20.0)*(0.5 + 0.15*log10(h1/121.92)) : 0.0
  s1 = (d >= 64.38) ? 0.174*(d - 64.38) : 0.0
  s2 = (h1 > 300) ? 0.00784*log10(9.98/d).abs*(h1 - 300.0) : 0.0
  s3 = (f/250.0)*log10(1500.0/f)
  s4 = (d > 64.38) ? (0.112*log10(1500.0/f)*(d - 64.38)) : 0.0

  hata(f, city_size, true, false) + [a - s1 - s2 - s3 - s4]
end

# Green-Obaidat model.
#
# From: "An Accurate Line of Sight Propagation Performance Model for Ad-hoc
# 802.11 Wireless (WLAN) Devices". 2002.
#
# Is basically freespace pathloss with a correction for antenna heights...
#
# f is in MHz
# d is in Km
# h1 & h2 are in m
def green_obaidat(f)
  h1 = self.tx.h
  h2 = self.rx.h
  d = distance(self.tx, self.rx)
  [40.0*log10(d),20.0*log10(f),-20.0*log10(h1*h2)]
end

# Flat Edge Model
#
# From: S.R. Saunders and F.R. Bonar. Explicit Multiple building diffraction
# attenuation function for mobile radio wave propagation. Electron. Lett. 1991.
# 27 (14). p. 1276-1277.
#
# This version, with some approximations from:
# Les Barclay. Propagation of Radiowaves. IEE. 2003. p. 197
#
# n is number of buildings between tx and rx
# h0 is nominal height of a building
# w is the distance between buildings (or maybe the width of a building,
# it doesn't really matter)
#
# All distances are in meters unless otherwise specified
def flat_edge(f,n=5,h0=20,w=10)
  lr = 0.25 # the refraction loss
  hm = self.rx.h
  @warn.push "Receiver height (#{hm}) is above assumed building height (#{h0})"
    if hm > h0
```

```
    # angle between ground and tx->rx LOS path
    phi = deg_to_rad(ascension(self.rx, self.tx).abs)
    lf = freespace(f).sum
    wl = freq_to_wavelength(f/1000.0)
    c1 = 3.29
    c2 = 9.90
    c3 = 0.77
    c4 = 0.26
    t = (phi*sqrt((PI*w)/wl)).abs
    ln = 0
    if t < 0 and t >= -1 and n >= 1 and n <= 100
      # This is an approximate fit due to Barclay which he claims is
      # accurate to less than +/- 1.5dB for 1<=n<=100 and -1<=t<0.
      ln = -(c1 + c2*log10(t) - (c3 + c4*log10(n)))
    else
      # These fresnel approximations due to Saunders. Antennas and Propagation
      # for Wireless Communication Systems. Appendix B.3.
      fres_fu = Proc.new{ |u| (1.0 + 0.926*u)/(2.0 + 1.792*u + 3.104*(u**2)) }
      fres_gu = Proc.new{ |u| 1.0/(2.0 + 4.142*u + 3.492*(u**2) + 6.670*(u**3)) }
      fres_cu = Proc.new{ |u| 0.5 + fres_fu.call(u)*sin((PI/2)*(u**2)) -
        fres_gu.call(u)*cos((PI/2)*(u**2)) }
      fres_su = Proc.new{ |u| 0.5 - fres_fu.call(u)*cos((PI/2)*(u**2)) -
        fres_gu.call(u)*sin((PI/2)*(u**2)) }
      fs = Proc.new{ |jx| (exp(Complex(0,-jx**2))/(sqrt(Complex(0,2))))*
                          ((fres_su.call(jx.real*sqrt(2.0/PI)) + 0.5) +
                          Complex(0,fres_cu.call(jx.real*sqrt(2.0/PI)) + 0.5)) }
      lnt = Proc.new{ |n,t| n == 0 ? 1.0 : (1.0/n)*(0..n-1).inject(0.0){ |sum,m|
        sum + lnt.call(m,t)*fs.call(Complex(0,-t)*sqrt(n-m)) } }
      ln = lnt.call(n,t).abs
    end
    # This equation is a simple knife-edge diffraction loss due to the Ikegami model
    le = 0
    le = 10.0*log10(f) + 10.0*log10(sin(deg_to_rad(phi))) + 20.0*log10(h0-hm) -
      10.0*log10(w) - 10.0*log10(1.0 + 3.0/(lr**2)) - 5.8 if hm < h0 and
        phi != 0
    [ln,lf,le]
  end

  # Walfisch-Bertoni Model
  #
  # Much like the Flat Edge model, but assumes "many" buildings.
  #
  # From: Walfisch J. and Bertoni H.L.. A theoretical model of UHF propagation
  # in urban environments. IEEE Trans. Ant. Prop. 1988. 36. (12) p. 1788-1796
  #
  # h0 is nominal height of a building (m)
  # w is the distance between buildings (m)
  def walfish_bertoni(f,h0=20,w=10)
    d = distance(self.tx,self.rx)
    hb = self.tx.h
    hm = self.rx.h
    @warn.push "Receiver height (#{hm}) is above assumed building height (#{h0})"
      if hm > h0
    la = (hm > h0) ? 0.0 : 5.0*log10((w/2) + (h0-hm)**2) - 9.0*log10(w) +
      20.0*log10(atan((2.0*(h0-hm))/w))
    lex = 57.1 + log10(f) + 18.0*log10(d) - ((hb-h0 > 0) ? 18.0*log10(hb-h0) :
      0.0) - 18.0*log10(1.0 - d**2/(17.0*(hb-h0)))
    [freespace(f),lex,la]
  end

  # Riback-Medbo Model
  #
  # From: M. Riback, J. Medbo, J.E. Berg, F. Harrysson, and H. Asplund. Carrier
  # Frequency Effects on Path Loss. 20006.
  #
  # Attempts to provide a correction for using a given model from one frequency
  # domain to predict PL values at a different frequency.
```

```
#
# f is the frequency WE are modeling in MHz
# f0 is the frequency the model we want to use was based on in MHz
# lf0 is the PL predicted by this model at the f0 frequency
def riback_medbo(f,f0,lf0)
  # fitted constants
  a = 0.09
  b = 256*(10**6)
  c = 1.8
  k = a*(atan(f0/b - c) - atan(f/b - c)) # correction factor
  return [lf0,20.0*log10(f/f0),-k*(lf0 - freespace(f0,2.0).sum)]
end

# Building-Transmission Model
#
# From: Y.L.C. de Jong, M. H. J. L. Koelen, and M. H. A. J. Herben. A
# Building-Transmission Model for Improved Propagation Prediction
# in Urban Microcells. IEEE Transactions on Vehicular Technology. Vol 53.
# No. 2. March, 2004.
#
# Predicts average loss due to transmitting "through" buildings
# This is for 1.9 GHz and must be used in combination with some other path-loss
# or ray-tracing model
def building_transmission(config_filename="config.yaml")
  datasource = "buildings"
  conductivity = 0.0
  permitivity = 5.0
  alpha = 2.1 # average attenuation (in dB) per meter inside building
  sum = 0.0
  intersections(config_filename,datasource){ |din|
    # assuming 90-degree angles of incidence
    theta0 = Math::PI/2.0 # angle relative to building surface going in
    theta1 = Math::PI/2.0 # angle relative to building surface going out
    # eq. 5
    r0 = (sin(theta0)-sqrt(permitivity-cos(theta0)**2)) /
      (sin(theta0)+sqrt(permitivity-cos(theta0)**2))
    r1 = (sin(theta1)-sqrt(permitivity-cos(theta1)**2)) /
      (sin(theta1)+sqrt(permitivity-cos(theta1)**2))
    # eq. 7
    t0 = sqrt(1 - r0.abs**2)
    t1 = sqrt(1 - r1.abs**2)
    # eq. 10
    sum += alpha*din - 20.0*log10(t0) -  20.0*log10(t1)
  }
  return [sum]
end

# Gas Attenuation Model
#
# Computes additional attenuation due to transmission through water vapor
# within oxygen. Note that this is for sea-level and that the ITU
# recommendation is to not bother for f < 10 GHz.
#
# For the sort of distances and frequencies we're working with, this is an
# attenuation of like 0.01 dB. Not really worth considering...
#
# From ITU-R P.676.
#
# ITU-R P.836 gives information on water vapor density.
#
# ITU-R P.452-13 gives some description fo this too
#
# http://www.mike-willis.com/Tutorial/PF5.htm
#
# p is water vapour concentration in grams per cubic meter
# P.452 suggests that you can use p = 7.5 + 2.5*omega
# where omega is the fraction of the total path over water
```

```
  def gas_attenuation(f,p=7.5)
    d = distance(self.tx, self.rx) # in Km
    f = f/1000.0 # in GHz
    a_w = (0.0050 + 0.0021*p + 3.6/((f - 22.2)**2  + 8.5) + 10.6/
      ((f - 183.3)**2 + 9.0) + 8.9/((f - 325.4)**2 + 26.3))*(f**2)*p*0.001
    a_o2 = (7.19*0.01 + 6.09/(f**2 + 0.277) + 4.81/
      ((f - 57.0)**2 + 1.5))*(f**2)*0.01
    [a_w*d, a_o2*d]
  end

  # Standford University Interim Model (SUI)
  #
  # Note that this is a less-complex precurser to the Erceg-Greenstrein Model
  #
  # From:
  #
  # Abhayawardhana \textit{et al.} Comparison of Empirircal Propagation Path
  # Loss Models for Fixed Wireless Access Systems.
  #
  # and
  #
  # Erceg \textit{et al.} Channel Models for Fixed Wireless Applications. Tech.
  # Report. IEEE 802.16 Broadband Wireless Access Working Group. January 2001.
  #
  # f is in MHz
  # terrain type can be :a, :b, or :c
  #   from the paper: The maximum path loss category is hilly terrain with
  #   moderate-to-heavy tree densities (Category A). The minimum path
  #   loss category is mostly flat terrain with light tree densities
  #   (Category C). Intermediate path loss condition is captured in
  #   Category B.
  def sui(f, terrain_type=:a, vary=false)
    hb = self.tx.h
    hr = self.rx.h
    d0 = 100.0 # m
    d = distance(self.tx, self.rx)*1000.0 # m
    wl = freq_to_wavelength(f/1000.0)
    biga = 20.0*log10((4.0*PI*d0)/wl)
    a = {:a => 4.65, :b => 4.0, :c => 3.6}[terrain_type]
    b = {:a => 0.0075, :b => 0.0065, :c => 0.005}[terrain_type]
    c = {:a => 12.6, :b => 17.1, :c => 20.0}[terrain_type]
    xf = 6.0*log10(f/2000.0)
    gamma = a - b*hb + c/hb
    xh = (terrain_type == :c) ? -20.0*log10(hr/2000.0) : -10.8*log10(hr/2000.0)
    s = vary ? rlognorm(0.0, runif(8.2,10.6)) : 0.0
    [biga, 10.0*gamma*log10(d/d0), xf, xh, s]
  end

  # ECC-33 Model
  #
  # From:
  #
  # Abhayawardhana \textit{et al.} Comparison of Empirircal Propagation Path
  # Loss Models for Fixed Wireless Access Systems.
  #
  # f is in MHz
  # city_size can be large or medium
  def ecc33(f, city_size=:large)
    f = f/1000.0 # GHz
    hb = self.tx.h
    hr = self.rx.h
    d = distance(self.tx, self.rx) # km
    afs = 92.4 + 20.0*log10(d) + 20.0*log10(f)
    abm = 20.41 + 9.83*log10(d) + 7.894*log10(f) + 9.56*(log10(f))**2
    gb = log10(hb/200.0)*(13.958 + 5.8*(log10(d))**2)
    gr = (city_size == :medium) ? 0.0 : (42.57 + 13.7*log10(f))*(log10(hr) - 0.585)
    [afs, abm, -gb, -gr]
```

```
  end

  # Edwards-Durkin Model
  #
  # From:
  #
  # G. Y. Delisle, J. P. Lefevre, M. Lecours, and J.Y. Choinard. Propagation
  # Loss Prediction: A Comparative Study with Application to the Mobile Radio
  # Channel. IEEE Trans on Vehicular Technology. Vol. VT-34. No. 2. May, 1985.
  #
  # f is carrier in MHz
  # use_terrain decided whether we should compute diffraction over the path
  def edwards_durkin(f,use_terrain=false,delta_h=15.0)
    r = distance(self.tx,self.rx) # km
    hb = self.tx.h
    hm = self.rx.h
    k1 = 32.45 # for isotropic ant.; use 28.85 for half-wave dipoles
    k2 = 118.7 # for isotropic ant.; use 115.1 for half-wave dipoles

    # lf is a lower bound which we won't use here because Delisle says lp + ld is a better
    # fit to data in practice
    #lf = k1 + 20.0*log10(f) + 20.0*log10(r) # classical freespace loss

    lp = k2 - 20.0*log10(hm) - 20.0*log10(hb) + 40.0*log10(r) # plane earth loss
    ld = use_terrain ? terrain_diffraction_estimate(f,delta_h).sum : 0.0

    return [lp,ld]
  end

  # Blomquist-Ladell
  #
  # From:
  #
  # G. Y. Delisle, J. P. Lefevre, M. Lecours, and J.Y. Choinard. Propagation
  # Loss Prediction: A Comparative Study with Application to the Mobile Radio
  # Channel. IEEE Trans on Vehicular Technology. Vol. VT-34. No. 2. May, 1985.
  #
  # f is carrier in MHz
  # use_terrain decided whether we should compute diffraction over the path
  def blomquist_ladell(f,use_terrain=false,delta_h=15.0)
    r = distance(self.rx,self.tx) # km
    d = 1000.0*r # m
    hb = self.tx.h
    hm = self.rx.h
    eb = em = 10.0 # permitivity
    wl = freq_to_wavelength(f/1000.0)
    lf = 32.45 + 20.0*log10(f) + 20.0*log10(r)
    k = 4.0/3.0 # earth radius factor
    a = 6.371*(10**6) # earth radius in m
    x = ((2.0*PI/wl)**(1.0/3.0))*((k*a)**(-2.0/3.0))*d
    y = (x < 0.53) ? -2.8*x : 6.7 + 10.0*log10(x) - 10.2*x
    fb = 10.0*log10( (((4.0*PI*hb**2)/(wl*d)) + ((wl*(eb**2))/(PI*d*(eb-1)))) *
                     (((4.0*PI*hm**2)/(wl*d)) + ((wl*(em**2))/(PI*d*(em-1)))) ) + y
    ld = use_terrain ? terrain_diffraction_estimate(f,delta_h).sum : 0.0
    lt = 0.0
    if fb <= 0
      [lf,sqrt(fb**2 + ld**2)]
    elsif fb > 0 and fb <= ld.abs
      [lf,sqrt(fb**2 - ld**2)]
    elsif fb > 0 and fb > ld.abs
      [lf,-sqrt(fb**2 - ld**2)]
    end
  end

  # Alsebrook Parsons Model
  #
  # From:
```

```
#
# G. Y. Delisle, J. P. Lefevre, M. Lecours, and J.Y. Choinard. Propagation
# Loss Prediction: A Comparative Study with Application to the Mobile Radio
# Channel. IEEE Trans on Vehicular Technology. Vol. VT-34. No. 2. May, 1985.
#
# f is carrier in MHz
# use_terrain decided whether we should compute diffraction over the path
# delta_h is a terrain roughness parameter passed to
#   terrain_diffraction_estimate() if required
# h0 is the average height of buildings in m
# d2 is the average width of streets in m
def allsebrook_parsons(f,use_terrain=false,delta_h=15.0,h0=5.0,d2=20.0)
  r = distance(self.rx,self.tx) # km
  d = 1000.0*r # m
  hb = self.tx.h
  hm = self.rx.h
  eb = em = 10.0 # permitivity
  wl = freq_to_wavelength(f/1000.0)
  lf = 32.45 + 20.0*log10(f) + 20.0*log10(r)

  k = 4.0/3.0 # earth radius factor
  a = 6.371*(10**6) # earth radius in m
  x = ((2.0*PI/wl)**(1.0/3.0))*((k*a)**(-2.0/3.0))*d
  y = (x < 0.53) ? -2.8*x : 6.7 + 10.0*log10(x) - 10.2*x
  fb = 10.0*log10( (((4.0*PI*hb**2)/(wl*d)) + ((wl*(eb**2))/(PI*d*(eb-1)))) *
                   (((4.0*PI*hm**2)/(wl*d)) + ((wl*(em**2))/(PI*d*(em-1)))) ) + y

  ld = use_terrain ? terrain_diffraction_estimate(f,delta_h).sum : 0.0

  gamma = (f > 200) ? 13.0 : 0.0
  lb = (h0 > hm) ? 20.0*log10((h0-hm)/(548.0*sqrt(d2*0.01*f))) + 16.0 : 0.0

  [lf,sqrt(fb**2+ld**2),lb,gamma]
end

# Rural Hata/Medeisis-Hata model
#
# From:
#
# A. Medeisis and A. Kajackas. On the Use of the Universal Okumura-Hata
# Propagation Prediction Model in Rural Areas. IEEE Vehicular Technology
# Conference Proceedings. 2000-Spring. Tokyo. 1815-1818.
#
# env can be :rural or :urban
#
def rural_hata(f,config,env=:rural,dont_fix_heights=false)
  hms = self.rx.h # m
  hbs = self.tx.h # m
  hbs,hms = hata_fix_heights(hbs,hms) unless dont_fix_heights
  r = distance(self.tx,self.rx) # km
  closest_f = nil
  [160,450,900].each{ |f2|
    closest_f = f2 if closest_f.nil? or (f-f2).abs < (closest_f-f).abs
  }
  e0 = { :rural => { 160 => 40.0, 450 => 40.0, 900 => 35.0 },
         :urban => { 160 => 40.0, 450 => 50.0, 900 => 60.0 } }
  gamma = { :rural => { 160 => 1.25, 450 => 1.30, 900 => 1.00 },
            :urban => { 160 => 1.20, 450 => 1.20, 900 => 1.25 } }

  my_e0 = e0[env][closest_f]
  my_gamma = gamma[env][closest_f]

  # it's not clear if we should be using f or closest_f in these calcs. I think f is
  # more better even though it conflicts with the f we use to select fitted params.


  a = Proc.new { |hms,f| (1.1*log(f) - 0.7)*hms - (1.56*log(f) - 0.8) }
```

```
    esys = -6.16*log(f) + 13.82*log(hbs) + a.call(hms,f)
    gamma_sys = -my_gamma*(44.9 - 6.55*log(hbs))
    loss_dBuVm = my_e0 + esys + gamma_sys*log(r)
    # uhh... dBuV/m...
    # http://www.microvolt.com/table.html
    # http://www.softwright.com/faq/engineering/FIELD%20INTENSITY%20UNITS.html
    #
    g_rx = ant_gain(self.rx, self.tx, config)
    loss_dBm = loss_dBuVm + g_rx - 20.0*log(f) - 77.0
    [-loss_dBm]
  end

  # Oda Model
  #
  # From:
  #
  # Yasuhiro Oda and Koichi Tsunekawa. Advanced LOS Path Loss Model in Microwave
  # Mobile Communications. 10th International Conference on Antennas and
  # Propagation. 1997.
  #
  # A very minor correction to 2-ray path loss. Pretty dumb.
  #
  # h0 is average the height of street-level scatters such as traffic and signs
  #   and mailboxes and whatnot
  # s is the probability of collision per unit of distance (err...)
  def oda(f,h0=1.0,s=0.5)
    hb = self.tx.h
    hm = self.rx.h
    r = distance(self.rx, self.tx) # m
    rrm = sqrt(r**2 + ((hb-h0)+(hm-h0))**2) # distance along relfected path in m
    rt = sqrt(r**2 + (hb-hm)**2) # distance along LOS path in m
    pr = exp(-s*r)
    wl = freq_to_wavelength(f/1000.0) # wavelength
    k = (2.0*PI)/wl # wave number

    # This is from the paper, but it doesn't seem to work...
    #a = Complex.new(0.0,-k*rt)
    #b = Complex.new(0.0,-k*rrm)
    #bigr = -1.0 # assumed reflection coefficient is exactly out of phase
    #[10.0*log(pr*(wl/(4.0*PI))*((exp(a)/rt) + bigr*(exp(b)/rrm)).abs)]

    # This is based on Saunders2007 and assumes R = -1.
    # The paper gives no guidance, so this is probably fine.
    hb -= h0
    hm -= h0
    [10.0*log(pr*2.0*((wl/(4.0*PI*r))**2)*(1.0 - cos((k*2.0*hm*hb)/r)))]
  end

  # deSouza-Lins
  #
  # From:
  #
  # R. S. deSouza and R. D. Lins. A New Propagation Model for 2.4 GHz Wireless LAN.
  # APCC 2008.
  #
  # An explicitly data-fitted model which includes relative humidity. Probably
  # doesn't work for anything longer than about 120m since that is as far away
  # as they got from the AP in measurement.
  def desouza_lins(f,rh=50.0)
    d = distance(self.tx, self.rx)*1000.0 # m
    b0 = [38.88,37.67].mean
    b1 = [25.849,15.402].mean
    b2 = [0.099,0.155].mean
    b3 = [7.508,11.56].mean
    [b0,b1*log(d),b2*d,b3*log(rh)]
  end

```

```
################ TERRAIN MODELS ########################

# ITU Terrain Model
#
# From:
#
# J.S. Seybold. Introduction to RF Propagation. p. 144-145
#
# Propagation data and prediction methods required for the design of
# terrestrial line-of-sight systems. Recommendation ITU-R P.530-11.
#
def itu_terrain(f,path)
  d = distance(self.tx,self.rx)*1000 # in m
  wavelength = freq_to_wavelength(f/1000)
  agl_rx = rx.ele+rx.h                  # in m
  agl_tx = tx.ele+tx.h                  # in m
  los_slope = (agl_tx-agl_rx)/d         # negative if tx is above rx
  max_obstruction = -1.0
  los_max_obstruction = nil
  fres_max_obstruction = nil

  (1..self.length-2).each{ |i|
    int = self[i]
    next if rx.ele.nil? or tx.ele.nil? or int.ele.nil?
    di = distance(self.tx,int)*1000      # in m
    agl_int = int.ele+int.h                # in m
    los_elev = di*los_slope + agl_tx
    fresnel_radius = sqrt((wavelength*di*(d-di))/(di+(d-di)))
    fresnel_lower = los_elev - fresnel_radius
    if agl_int > fresnel_lower and agl_int > max_obstruction
      max_obstruction = agl_int
      los_max_obstruction = los_elev
      fres_max_obstruction = fresnel_radius
    end
  }
  a = 0.0
  if max_obstruction >= 0
    h = los_max_obstruction - max_obstruction
    a = (-20.0*h)/fres_max_obstruction + 10.0
    # negative and small losses are not realistic according to Seybold
    a = 0.0 if a <= 6
  end
  return [freespace(f,2.0).sum,a]
end

# From: ITU-R P.452
#
# p is the percentile not-to-exceed, so 50 means this is the median value.
# 100 would be a worst-case and 0 a best-case.
#
# delta_n is radio refractivity of the earth. some values:
#   35 - boulder average
#   50 - hamilton average
#   40 - portland average, boulder worst
#   45 - portland worst
#   60 - hamilton worst
# n0 is the sea level surface refractivity. some values:
#   300 - boulder
#   320 - portland
#   340 - hamilton
# omega is the fraction of the total path over water
def itu_r_452(f,config,p=50.0,delta_n=40.0,n0=320.0,omega=0.0)
  d = distance(self.tx,self.rx)   # in km
  k50 = 157.0/(157.0 - delta_n)
  kbeta = 3.0
  ae = 6371.0*k50
  abeta = 6371.0*kbeta
```

```
hts = self.tx.z # height above mean sea level (m)
hrs = self.rx.z # m
phi = self.center_latitude
lambda1 = freq_to_wavelength(f/1000)
ha = @path.collect{ |int| int.h + int.ele }.mean # mean height

### PATH CLASSIFICATION CALC (Appendex 2. Sect. 4 and 5.1.1 − 5.1.5) ###

theta_td = (hrs−hts)/d − (1000*d)/(2.0*ae)
theta_max = 0.0
theta_t = 0.0     # transmitter antenna horizon elevation angle (mrad)
theta_r = 0.0     # receiver antenna horizon elevation angle (mrad)
dlt = 0.0
ilr = 0
dlr = 0.0
ilt = 0

(1..self.length −2).each{ |i| # loop over path omitting tx and rx
  int = self[i] # intermediary point
  hi = int.z # m
  di = distance(self.tx,int) # km

  theta_i = (hi−hts)/di − (1000*di)/(2.0*ae)
  theta_j = (hi−hrs)/(d−di) − (1000*(d−di))/(2.0*ae)

  if theta_max.nil? or theta_i > theta_max
    theta_max = theta_i
    theta_t = theta_i
    dlt = di
    ilt = i
  end
  if theta_r.nil? or theta_j > theta_r
    theta_r = theta_j
    dlr = d − di
    ilr = i
  end
}

theta = (1000*d)/ae + theta_t + theta_r    # angular distance in mrad

# true if this is a trans−horizon path (unlikely for short distances)
transhorizon = theta_max > theta_td

### DIFFRACTION CALCULATIONS (Sect. 4.2) ###

# IMPORTANT ASSUMPTION:
# assume the entire path is over land...
# this doesn't mean small bodies of water.
# This means, like the ocean and stuff.
dtm = dlm = dct = dcr = tau = 0

mu1 = (10**(−dtm/(16−6.6*tau)) + (10**−(0.496+0.354*tau))**5)**0.2
mu1 = 1.0 if mu1 > 1.0
mu4 = (phi <= 70) ? 10**((−0.953+0.0176*phi.abs)*log10(mu1)) :
  10**(0.3*log10(mu1))

# point of incidence of anomolous propagation (%) for the path center location
beta0 = (phi <= 70) ? (10**(−0.015*phi.abs + 1.67))*mu1*mu4 : 4.17*mu1*mu4

# interpolation factor for path angular distance
fj = 1.0 − 0.5*(1.0 + tanh(3.0*0.8*((theta −0.3)/0.3)))
# interpolation factor for great circle path distance
fk = 1.0 − 0.5*(1.0 + tanh(3.0*0.5*((d−20)/20)))

# water vapor density and gaseous attenuation
p = 7.5 + 2.5*omega
ag = gas_attenuation(f,p).sum
```

```
    esp = 2.6*(1.0 - exp(-0.1*(dlt+dlr)))*log(p/50)
    esbeta = 2.6*(1.0 - exp(-0.1*(dlt+dlr)))*log(beta0/50)

    # approximate inverse cumulative normal distribution (Appendix 3)
    inv_cum_norm = Proc.new{ |x|
      c = [2.515516698,0.802853,0.010328]
      d = [1.432788,0.189269,0.001308]
      t = sqrt(-2.0*log(x))
      xi = ((c[2]*t + c[1])*t + c[0])/(((d[2]*t + d[1])*t + d[0])*t + 1.0)
      xi - t
    }

    # approximate knife-edge diffraction loss eq. 13
    ke_diff_loss = Proc.new{ |v|
      v < -0.78 ? 0.0 : 6.9 + 20.0*log10(sqrt((v-0.1)**2 + 1) + v - 0.1)
    }

    fi = 0
    if p == 50.0
      fi = 0
    elsif p > 50.0 and p < beta0
      fi = inv_cum_norm.call(p/100)/inv_cum_norm.call(beta0/100)
    elsif beta0 >= p
      fi = 1
    end

    # basic transmission loss due to free-space propagation and attenuation by
    # atmospheric gasses (Sect. 4.1)
    lbfsg = 92.5 + 20.0*log(f/1000) + 20.0*log(d) + ag

    # correction for overall path slope
    xi_m = cos(atan(0.01*(hrs-hts)/d))
    # principle edge diffraction parameter
    vm50 = 0.0
    im50 = 0
    him50 = 0.0
    dim50 = 0.0
    (1..@path.length-2).each{ |i|
      int = @path[i]
      di = distance(int, self.tx)
      his = int.z # height above mean sea level
      hi = his + 1000*(di*(d-di))/(2.0*ae) - (hts*(d-di)+hrs*di)/d
      val = xi_m*hi*sqrt((0.02*d)/(lambda1*di*(d-di)))
      if vm50.nil? or val > vm50
        vm50 = val
        im50 = i
        him50 = his
        dim50 = di
      end
    }
    lm50 = ke_diff_loss.call(vm50)

    ld50 = 0.0
    ldbeta = 0.0
    lt50 = 0.0
    lr50 = 0.0
    ltbeta = 0.0
    lrbeta = 0.0

    # only calculate ld50 and ldbeta if lm50 is nonzero
    if lm50 != 0.0
      # only calculate lt50 if there is a transmitter-side secondary edge
      if im50 <= 1
        xi_t = cos(atan(0.01*(him50-hts)/dim50))
        vt50 = nil
        it50 = nil
```

```
  hit50 = nil
  dit50 = nil
  (1..im50-1).each{ |i|
    int = @path[i]
    di = distance(int, self.tx)
    his = int.z
    hi = his + 1000*(di*(dim50-di))/(2.0*ae) -
      (hts*(dim50-di)+him50*di)/dim50
    val = xi_t*hi*sqrt((0.02*dim50)/(lambda1*di*(dim50-di)))
    if vt50.nil? or val > vt50
      vt50 = val
      it50 = i
      hit50 = his
      dit50 = di
    end
  }
  lt50 = (im50 >= 2) ? ke_diff_loss.call(vt50) : 0.0

  if lt50 != 0.0
    hitbeta = hit50 + 1000*(dit50*(dim50-dit50))/(2.0*abeta) -
      (hts*(dim50-dit50)+him50*dit50)/dim50
    vtbeta = xi_t*hitbeta*sqrt((0.02*dim50)/(lambda1*dit50*(dim50-dit50)))
    ltbeta = ke_diff_loss.call(vtbeta)
  end
end

# only calculate lr50 if there is a receiver-side secondary egde
if im50 < @path.length-2
  xi_r = cos(atan(0.01*(hrs-him50)/(d-dim50)))
  vr50 = nil
  ir50 = nil
  hir50 = nil
  dir50 = nil
  (im50+1..@path.length-2).each{ |i|
    int = @path[i]
    di = distance(int, self.tx)
    his = int.z
    hi = his + 1000*((di-dim50)*(d-di))/(2.0*ae) - (him50*(d-di) +
      hrs*(di-dim50))/(d-dim50)
    val = xi_r*hi*sqrt((0.02*(d-dim50))/(lambda1*(di-dim50)*(d-di)))
    if vr50.nil? or val > vr50
      vr50 = @path.inject(0.0){ |r, int|
        hi = int.h + int.ele
        dis = distance(int, self.tx)
        r += (hi-ha)*(dis-(d/2.0)) }
      ir50 = i
      hir50 = his
      dir50 = di
    end
  }
  lr50 = (im50 < @path.length-2) ? ke_diff_loss.call(vr50) : 0.0
  if lr50 != 0.0
    hirbeta = hir50+1000*((dir50-dim50)*(d-dir50))/(2.0*abeta) -
      (him50*(d-dir50)+hrs*(dir50-dim50))/(d-dim50)
    vrbeta = xi_r*hirbeta*sqrt((0.02*(d-dim50))/
      (lambda1*(dir50-dim50)*(d-dir50)))
    lrbeta = ke_diff_loss.call(vrbeta)
  end
end

# finally calculate ld50 from lt50 and lr50 and lm50
ld50 = lm50 + (1-exp(-lm50/6.0))*(lt50 + lr50 + 10.0 + 0.04*d)

# then the beta stuff...
himbeta = him50 + 1000*(dim50*(d-dim50))/(2.0*ae) -
  (hts*(d-dim50)+hrs*dim50)/d
vmbeta = xi_m*himbeta*sqrt((0.02*d)/(lambda1*dim50*(d-dim50)))
```

```
    lmbeta = ke_diff_loss.call(vmbeta)

    ldbeta = lmbeta + (1.0 - exp(-lmbeta/6.0))*(ltbeta + lrbeta + 10.0 +0.04*d)
  end

  ### TROPOSPHERIC SCATTER CALCULATIONS (Sect. 4.3)

  gt = ant_gain(self.tx, self.rx, config)
  gr = ant_gain(self.rx, self.tx, config)

  # aperture to medium coupling loss
  lc = 0.051*exp(0.055*(gt+gr))
  # frequency dependent loss
  lf = 25.0*log10(f) - 2.5*log10(f/2)**2
  # basic transmission loss due to troposcatter
  lbs = 190.0 + lf + 20.0*log10(d) + 0.573*theta - 0.15*n0 + lc + ag -
    10.1*(-log10(p/50.0))**0.7

  ### DUCTING/LAYER-REFLECTION CALCULATIONS (Sect. 4.2)

  theta_t1 = (theta_t <= dlt) ? theta_t : 0.1*dlt
  theta_r1 = (theta_r <= dlr) ? theta_r : 0.1*dlr

  theta1 = (1000*d)/ae + theta_t1 + theta_r1

  theta_t2 = theta_t - 0.1*dlt
  theta_r2 = theta_r - 0.1*dlr

  # over sea surface duct coupling corrections
  act = 0.0
  if omega >= 0.75 and dct <= dlt and dct <= 5.0
    act = -3.0*exp(-0.25*dct**2)*(1.0 + tanh(0.07*(50.0-hts)))
  end
  acr = 0.0
  if omega >= 0.75 and dcr <= dlr and dcr <= 5.0
    acr = -3.0*exp(-0.25*dcr**2)*(1.0 + tanh(0.07*(50.0-hrs)))
  end

  ### SMOOTH EARTH MODEL CALCULATIONS (Appendix 2. Sect.5.1.6) ###

  # slope of the smooth-earg surface relative to sea level
  # IMPORTANT ASSUMPTION: assume sample points are equally spaced.
  # there are other ways of calculating m if they are not
  mnum = 0.0
  mdem = 0.0
  (0..@path.length-1).each{ |i|
    hi = @path[i].z
    di = distance(@path[i], self.tx)
    mnum += (hi-ha)*(di-d/2.0)
    #puts "ha = #{ha}, di = #{di}, d = #{d}"
    mdem += (di-d/2.0)**2
  }
  m = mnum/mdem

  #puts "m = #{mnum}/#{mdem} = #{m}"

  hst = ha - m*d/2.0
  hsr = hst + m*d

  recalc_m = false
  if hst > self.tx.z
    hst = self.tx.z
    recalc_m = true
  end
  if hsr > self.rx.z
    hsr = self.rx.z
    recalc_m = true
```

```
    end
    m = (hsr-hst)/m if recalc_m

    # terrain roughness parameter
    hm = 0.0
    (ilt..ilr).each{ |i|
      dis = distance(@path[i],self.tx)
      hi = @path[i].z
      val = hi - (hst + m*dis)
      #puts "val = #{hi} - (#{hst} + #{m}*#{dis}"
      hm = val if hm.nil? or val > hm
    } unless ilr < ilt
    #puts "hm = #{hm}"

    epsilon = 3.5 # not used
    alpha = 0 # because tau is zero based on IMPORTANT ASSUMPTION above
    mu2 = 1.0 # because alpha is zero
    mu3 = (hm <= 10) ? 1.0 : exp(-4.6*(10**-5)*(hm-10)*(43+6*([d-dlt-dlr,40.0].min)))
    beta = beta0*mu2*mu3
    #puts "beta = #{beta0}*#{mu2}*#{mu3} = #{beta}"
    gamma = (1.076/((2.0058-log10(beta))**1.012))*
      exp(-(9.51-4.8*log10(beta)+0.198*(log(beta)**2))*(10**-6)*(d**1.13))
    ap = -12.0 + (1.2 + 0.037*d)*log10(p/beta) + 12.0*(p/beta)**gamma
    gamma_d = 0.0005*ae*(f**(1.0/3.0))

    ad = gamma_d*theta1 + ap

    # site shielding losses
    ast = 0.0
    asr = 0.0
    if theta_t2 > 0
      ast = 20.0*log10(1.0 + 0.361*theta_t2*sqrt(f*dlt)) +
        0.264*theta_t2*(f**(1.0/3.0))
    end
    if theta_r2 > 0
      asr = 20.0*log10(1.0 + 0.361*theta_r2*sqrt(f*dlr)) +
        0.264*theta_r2*(f**(1.0/3.0))
    end

    # total fixed coupling losses (except for local clutter losses) between the
    # antennas and the anomolous propagation structure within the atmosphere
    af = 102.45 + 20.0*log10(f) + (dlt+dlr > 0.0 ? 20.0*log10(dlt+dlr) : 0.0) +
      ast + asr + act + acr

    # basic transmission loss occuring during periods of anomalous propagation
    # (ducting and layer reflection)
    lba = af + ad + ag

    ### ADDITIONAL CLUTTER LOSSES (Sect. 4.5)

    # Note, it's note clear if these should be calculated over the total path
    # or just near the ends Also, if we calculate total path clutter for both
    # the receiver and transmitter, some double counting occurs. What I'm going
    # to do here is call aht the additional loss from the clutter on the tx side
    # of the path and ahr the addition loss from the receiver side of the path.
    # Each will be capped at 20dB as specified. If there's supposed to be a gap
    # in between, I'm not sure what it should be (i.e. how far away something
    # can be and still be considered "local clutter") maybe for microcell
    # networks, it's all relevant...

    # Note also that this will count more clutter for more sample points,
    # which is maybe wrong. Really need to know what "percentage" of the path is
    # "local" clutter.  For now, we'll be conservative and count everything

    aht = 0.0
    ahr = 0.0
    (0..@path.length-1).each{ |i|
```

```
      int = @path[i]
      next if int.clutter.nil?
      middle_i = (@path.length/2).floor
      tx_side = (i <= middle_i)
      h = tx_side ? self.tx.h : self.rx.h
      d = tx_side ? distance(self.tx,@path[i]) : distance(self.rx,@path[i])
      h_clutter, d_clutter = int.clutter
      val = 10.25*exp(-d_clutter)*(1.0-tanh(6.0*((h/h_clutter)-0.625))) - 0.33
      tx_side ? aht += val : ahr += val
    }
    aht = [aht,20.0].min
    ahr = [ahr,20.0].min

    ### OVERALL PREDICTION (Sect. 4.6)

    # diffraction loss not to exceed p%
    ldp = ld50 + fi*(ldbeta - ld50)

    # median basic transmission loss associated with diffraction
    lbd50 = lbfsg + ld50

    # basic transmission loss not to exceed for time
    # percentage p% due to LOS propagation
    lb0p = lbfsg + esp

    # basic transmission loss not exceeded for the time percentage
    # beta0% due to LOS propagation
    lb0beta = lbfsg + esbeta

    # basic transmission loss associated with diffraction not exceed p% of time
    lbd = lb0p + ldp

    # notational minimum basic transmission loss for LOS propagation
    # and over-sea subpath diffraction
    lminb0p = (p < beta0) ? lb0p + (1-omega)*ldp : lbd50 +
      (lb0beta + (1-omega)*ldp - lbd50)*fi

    # notational minimum basic transmission loss
    lminbap = 2.5*log(exp(lba/2.5) + exp(lb0p/2.5))

    # notational basic transmission loss
    lbda = (lminbap > lbd) ? lbd : lminbap + (lbd-lminbap)*fk

    # modified basic transmission loss
    lbam = lbda + (lminb0p - lbda)*fj

    # final basic transmission loss not exceeded p% of the time
    [-5.0*log10(10.0**(-0.2*lbs)+10**(-0.2*lbam)),aht,ahr]
  end

  # Generic Statistical Estimation of Terrain Diffraction Loss
  #
  # From:
  #
  # G. Y. Delisle, J. P. Lefevre, M. Lecours, and J.Y. Choinard. Propagation
  # Loss Prediction: A Comparative Study with Application to the Mobile Radio
  # Channel. IEEE Trans on Vehicular Technology. Vol. VT-34. No. 2. May, 1985.
  #
  # f is carrier in MHz
  # delta_h is a terrain roughness parameter which might be somewhere in the
  #   neighborhood of 15.0 for open terrain, 200ish for hilly terrain, and
  #   400ish for rugged terrain
  def terrain_diffraction_estimate(f,delta_h=15.0)
    r = distance(self.tx,self.rx)
    hb = self.tx.h
    hm = self.rx.h

```

```
  # effective heights in m
  heb = hb # I'm not sure how this differs from heights
  # Wikipedia seems to imply they are the same:
  # http://en.wikipedia.org/wiki/Effective_height
  hem = hm

  # horizon distances in m
  dlsb = sqrt(17.0*heb)
  dlsm = sqrt(17.0*hem)

  a = Proc.new { |v|
    (v > 2.4) ? 12.953 + 20.0*log10(v) : 6.02 + 9.11*v - 1.27*(v**2)
  }
  dhr = Proc.new { |dh,r|
    dh*(1.0 - 0.8*exp(-0.02*r))
  }

  dlb = dlsb*exp(-0.07*sqrt(delta_h/[5.0,heb].max))
  dlm = dlsm*exp(-0.07*sqrt(delta_h/[5.0,hem].max))

  dl = dlb + dlm
  dls = dlsb + dlsm

  theta_eb = (0.0005/dlsb)*(1.3*((dlsb/dlb)-1.0)*delta_h - 4.0*heb)
  theta_em = (0.0005/dlsm)*(1.3*((dlsm/dlm)-1.0)*delta_h - 4.0*hem)

  d1prime = dl + 0.5*((72165000.0/f)**(1.0/3.0))
  d1 = (d1prime <= dls) ? dls : d1prime
  d2 = d1 + ((72165000.0/f)**(1.0/3.0))

  theta1 = [theta_eb+theta_em,-dl/8495.0].max + d1/8495.0
  theta2 = [theta_eb+theta_em,-dl/8495.0].max + d2/8495.0

  vb1 = 1.2915*theta1*sqrt(f*dlb*(d1-dl)/(d1-dlm))
  vb2 = 1.2915*theta2*sqrt(f*dlb*(d2-dl)/(d1-dlm))
  vm1 = 1.2915*theta1*sqrt(f*dlm*(d1-dl)/(d1-dlb))
  vm2 = 1.2915*theta2*sqrt(f*dlm*(d2-dl)/(d1-dlb))

  ak1 = a.call(vb1) + a.call(vm1)
  ak2 = a.call(vb1) + a.call(vm2)

  md = (ak2 - ak1)/(d2 - d1)

  sigma = 0.78*dhr.call(delta_h,dls)*exp(-0.5*(dhr.call(delta_h,dls)**(1.0/4.0)))
  af0prime = 5.0*log10(1.0 + 0.0001*hm*hb*f*sigma)
  af0 = [af0prime,15.0].min
  a0 = af0 + ak2 - md*d2

  ld = md*r + a0
  return [ld]
end

################ STOCHASTIC MODELS ######################

# The Directional Gain Reduction Factor from:
#
# Greenstein and Erceg. "Gain Reductions Due to Scatter on Wireless
# Paths with Directional Antennas". IEEE Comms. Letters. 1999.
#
# A correction for multipath effects at the receiver due to the receiver
# using a directional antenna.
#
# If vary is false, the median case is given.
#
# For 1.9 GHz

def gain_reduction_factor(f,winter=true,vary=false)
```

```
    h2 = self.rx.h
    beamwidth = rx.beamwidth
    model_name = "Directional Gain Reduction Factor"
    @warn.push "Receiver height (#{h2}m) is outside the model's coverage"
      if h2 > 10 or h2 < 3
    @warn.push "Beamwidth (#{beamwidth} degrees) is outside the model's coverage"
      if beamwidth < 17 or beamwidth > 65
    @warn.push "Frequency (#{f} MHz) is outside the #{model_name} model's coverage"
      if f != 1900.0

    return [0.0] if beamwidth == 360

    i = (winter) ? 1.0 : -1.0
    mu = -(0.53 + 0.1*i)*log(beamwidth/360.0) + (0.50 + 0.04*i)*(log(beamwidth/360.0)**2.0)
    sigma = -(0.93 + 0.02*i)*log(beamwidth/360.0)
    return [vary ? rnorm(mu,sigma) : mu]
  end

  # EDAM "directivity" model from:
  #
  # Eric Anderson, Gary Yee, Caleb Phillips, Douglas Sicker, and Dirk Grunwald.
  # The Impact of Directional Antenna Models on Simulation Accuracy. 7th
  # International Symposium on Modeling and Optimization in Mobile, Ad Hoc,
  # and Wireless Networks (WiOpt 2009). Seoul, Korea. June 23 - 27, 2009.
  #
  # If vary is false, the median case is given.
  #
  # For 2.4GHz
  def edam(f, config, environment=:open_outdoor, vary=false)
    @warn.push "Frequency #{f} MHz is out of range for EDAM's coverage"
      if f > 2500.0 or f < 2400.0

    # setup ranges
    kgain = nil
    soff = nil
    sss = nil
    case environment
      when :open_outdoor
        kgain = [0.01,0.04]
        soff = [1.326,2.675]
        sss = [2.68,3.75]
      when :urban_outdoor
        kgain = [0.15,0.19]
        soff = [2.244,3.023]
        sss = [2.46,2.75]
      when :los_indoor
        kgain = [0.25,0.38]
        soff = [2.837,5.242]
        sss = [2.9,5.28]
      when :nlos_indoor
        kgain = [0.67,0.70]
        soff = [3.17,3.566]
        sss = [3.67,6.69]
    end

    # select uniformly at random from within range
    kgain = vary ? runif(kgain[0],kgain[1]) : kgain.mean
    soff = vary ? runif(soff[0],soff[1]) : soff.mean
    sss = vary ? runif(sss[0],sss[1]) : sss.mean

    f_src = ant_gain(self.tx, self.rx, config)
    f_dst = ant_gain(self.rx, self.tx, config)

    g_src = (f_src*kgain + (vary ? rnorm(0.0,soff) : 0.0))
    g_dst = (f_dst*kgain + (vary ? rnorm(0.0,soff) : 0.0))

    epsilon = (vary) ? rnorm(0.0,sss) : 0.0
```

```

    return [g_src, g_dst, epsilon]
  end


  # Herring Air-to-Ground Model
  #
  # From: Keith Herring, Jack Holloway, David Staelin. "Path-Loss Characteristics
  # of Urban Wireless Channels". IEEE Trans. On Antennas and Propogation. 2009
  #
  # This is a stochastic measurement-based predictor for 2.4GHz
  def herring_atg(f, vary=false)
    [freespace(f, 2.0).sum, (vary ? rnorm(30, 8.3) : 30.0)]
  end

  # Herring Ground-to-Ground Model
  #
  # Assumes a single corner between two radios at street level.
  #
  # This is a stochastic measurement-based predictor for 2.4GHz
  def herring_gtg(f, vary=false)
    alpha = vary ? runif(2.0, 5.0) : [2.0, 5.0].mean
    ahat = alpha + (vary ? rnorm(0.0, 0.22) : 0.0)
    b = vary ? rnorm(40.0, 5.5) : 40.0
    [freespace(f, ahat).sum, b]
  end

  # TM-90 Model
  #
  # From:
  #
  # William Daniel and Harry Wong. Propagation in Suburban Areas at Distances
  # less than Ten Miles. FCC Technical Report. FCC/OET TM 91-1. January 25, 1991.
  #
  def tm90(f, eirp, building_penetration=false)
    dkm = distance(self.tx, self.rx)
    d = dkm*3280.84 # feet
    h1 = self.tx.h*3.28 # feet
    h2 = self.tx.h*3.28 # feet
    b = building_penetration ? -5.75 + 4.5*log(f) : 0.0
    bigf = 141.4 + 20.0*log10(h1*h2) - 40.0*log(d) + b
    # Now attempt to convert this value, which is in dBuV/m to dB
    # I'm using here, the same equations that SPLAT! uses, but
    # I'm not sure where they came from...
    erp = eirp - 2.14
    p = 10**(erp/10)/1000.0
    ldb = 10*log10(p/1000.0) + 139.4 + 20*log10(f) - bigf
    [10*log10(p/1000.0), 139.4, 20*log10(f), -bigf]
  end

  # IMT-2000 Pedestrian Environment Model
  #
  # From: Vikay J. Garg. Wireless Communications and Networking. Elsevier. 2007. p. 73.
  #
  # This is an attempt at worst-case path loss for urban environments, which
  # assumes transmitters are outdoors and receivers are indoors. Hence, it
  # assumes a outdoor-indoor penetration loss (of 18 dB), a shadowing loss (of 10 dB)
  # and a PL exponent of 4.
  #
  # If vary is false, median case is given
  def imt2000_pedestrian(f, indoor_receivers=false, vary=false)
    penetration_loss = indoor_receivers ? 18.0 : 0.0
    shadowing_loss = vary ? rlognorm(0.0, 10.0) : 0.0
    d = distance(self.tx, self.rx) # in Km
    [40.0*log10(d), 30.0*log10(f), shadowing_loss+penetration_loss, 21]
  end

```

```
# Erceg-Greenstein Model
#
# From: V. Erceg, L. Greenstein, S. Tjandra, S. Parkoff, A. Gupta, B. Kulic,
# A. Julius, and R. Bianchi. An Empirically Based Path Loss Model for Wireless
# Channels in Suburban Environments. Journal on Selected Areas in Communications.
# Vol. 17 No. 7. July, 1999.
#
# For 1.9 GHz
#
# terrain_category can be:
#   :A - Hilly/Moderate to Heavy Tree Density
#   :B - Hilly/Light Tree Density or Flat/Moderate-to-Heavy Tree Density
#   :C - Flat/Light Tree Density
#
# f is the frequency in MHz
# if vary is false, median case is given
def erceg_greenstein(f, terrain_category=:C, vary=false)
  # variables
  d = distance(self.tx, self.rx)*1000.0 # m
  d0 = 100.0 # reference distance in m
  # PL in dB at reference dist for this freq
  biga = freespace(f,2.0,d0/1000.0).sum
  hb = self.rx.h

  # static model params
  a =           {:A => 4.6,    :B => 4.0,    :C => 3.6}
  b =           {:A => 0.0075, :B => 0.0065, :C => 0.0050}
  c =           {:A => 12.6,   :B => 17.1,   :C => 20.0}
  sigma_gamma = {:A => 0.57,   :B => 0.75,   :C => 0.59}
  mu_sigma =    {:A => 10.6,   :B => 9.6,    :C => 8.2}
  sigma_sigma = {:A => 2.3,    :B => 3.0,    :C => 1.6}

  # pick the right params for the terrain
  a = a[terrain_category]
  b = b[terrain_category]
  c = c[terrain_category]
  sigma_sigma = sigma_sigma[terrain_category]
  sigma_gamma = sigma_gamma[terrain_category]
  mu_sigma = mu_sigma[terrain_category]

  # three zero-mean unit standard-deviation gaussian random vars
  # x*sigma_gamma is truncated at +/- 1.5
  # y and z are truncated at +/- 2.0
  # in order to avoid impossible values (however unlikely)
  x = [[(vary ? rnorm(0.0,1.0) : 0.0)*sigma_gamma,1.5].min,-1.5].max
  # truncate these two to make sure
  y = [[(vary ? rnorm(0.0,1.0) : 0.0),2.0].min,-2.0].max
  z = [[(vary ? rnorm(0.0,1.0) : 0.0),2.0].min,-2.0].max

  return [biga,10*(a - b*hb + c/hb)*log10(d/d0),10.0*x*log10(d/d0) +
    y*mu_sigma + y*z*sigma_sigma]
end

# Barclay-Okumura Fading
#
# Frequency-dependent fading based on data from Okumura and several
# other publications, included in:
#
# Les Barclay. Propagation of Radiowaves. IEE. 2003. p. 209
#
# Environment can be either :urban or :suburban
# if vary is false, returns median case which is always zero
def okumura_fc(f, environment=:urban, vary=false)
  a = environment == :urban ? 5.2 : 6.6
  sigma = 0.65*log10(f)**2 - 1.3*log10(f) + a
  [vary ? rnorm(0.0,sigma) : 0.0]
end
```

```
end
```

C.3 Effective Signal to Noise Ratio

Following is an implementation, in R, of the Effective SNR calculation used in this thesis. Some of the functions were derived from the Matlab implementation of Halperin *et al.* in [90]. To conserve space some of the simpler supporting functions have been excluded.

```
# 2.1-98 in Proakis
Q <- function(x){
  0.5*erfc(x/sqrt(2))
}

Qinv <- function(y){
  sqrt(2)*erfcinv(2*y)
}

# Marcum Q from 2.1-122 in Proakis
Q1 <- function(a,b,kmax=100){
  s <- 0
  for(k in seq(0,kmax)){
    s <- s + ((a/b)^k)*besselI(a*b,k)
  }
  exp(-(a^2 + b^2)/2)*s
}

# ps is probability of symbol error, which
# is mod dependent

# 5.2-57 in Proakis
ps.bpsk <- function(snr){
  Q(sqrt(2*snr))
}

ps.bpsk.inv <- function(ber){
  (Qinv(ber)^2)/2
}

# 5.2-59 in Proakis
ps.qpsk <- function(snr){
  2*Q(sqrt(2*snr))*(1 - 0.5*Q(sqrt(2*snr)))
}

# inverse solution via the quadratic equation...
ps.qpsk.inv <- function(ber){
  a <- (Qinv(1-sqrt(1-ber))^2)/2
  b <- (Qinv(1+sqrt(1-ber))^2)/2
  if(is.finite(a) & is.finite(b)){
    c(a,b)
  }else if(is.finite(a)){
    c(a)
  }else if(is.finite(b)){
    c(b)
  }else{
    NA
  }
```

```
}

# A simpler version due to Daniel Halperin <dhalperi@cs.washington.edu>
# linux-80211n-csitool-supplementary/matlab/qpsk_berinv.m
ps.qpsk.inv.dh <- function(ber){
  Qinv(ber)^2
}

# 5.2-61 in Proakis
ps.mpsk <- function(snr,m){
  2*Q(sqrt(2*snr)*sin(pi/m))
}

ps.mpsk.inv <- function(ber,m){
  0.5*(Qinv(ber/2)/sin(pi/m))^2
}

# 5.2-78 in Proakis
ps.sqmqam <- function(snr,m){
  2*(1 - (1/sqrt(m)))*Q(sqrt((3/(m-1))*snr))
}

ps.sqmqam.inv <- function(ber,m){
  (Qinv(ber/(2*(1-(1/sqrt(m))))))*(m-1))/3
}

# 5.2-79 in Proakis
ps.mqam <- function(snr,m){
  1 - (1 - ps.sqmqam(snr, sqrt(m)))^2
}

ps.mqam.inv <- function(ber,m){
  ps.sqmqam.inv(1 - sqrt(1 - ber), sqrt(m))
}

# These four via Daniel Halperin <dhalperi@cs.washington.edu>
# linux-80211n-csitool-supplementary/matlab/...
ps.16qam.inv <- function(ber){
  Qinv(ber*4/3)^2 * 5
}
ps.64qam.inv <- function(ber){
  Qinv(12/7*ber)^2 * 21
}
ps.16qam <- function(snr){
  3/4 * Q(sqrt(snr/5))
}
ps.64qam <- function(snr){
  7/12 * Q(sqrt(snr/21))
}
ps.qpsk.dh <- function(snr){
 Q(sqrt(snr))
}

bits.per.sym <- function(mod){
  if(mod == "qpsk") 2
  else if(mod == "dbpsk") 1
  else if(mod == "dqpsk") 2
  else if(mod == "bpsk") 1
  else if(mod == "qam16") 4
  else if(mod == "qam64") 6
}

# 5.2-70 in Proakis
pb.dqpsk <- function(snr){
  a <- sqrt(2*snr*(1 - sqrt(1/2)))
  b <- sqrt(2*snr*(1 + sqrt(1/2)))
  Q1(a,b) - (1/2)*besselI(a*b,0)*exp((-1/2)*(a^2 + b^2))
```

```
}

# 5.2-69 in Proakis
pb.dbpsk <- function(snr){
  (1/2)*exp(-snr)
}

# NOTE: I've "turned on" David Halperin's
#       alternative versions of several functions below
#       his versions deviate from Proakis and are
#       simpler (probably approximations), but are easier
#       to compute, invert, and are comparable with the
#       Effective SNR paper.

# pb is probabilility of bit error: (1/j)*ps where
# j is the number of bits per symbol (which is mod dep)
# 5.2-62 in Proakis
pb <- function(snr,mod){
  j <- bits.per.sym(mod)
  if(mod == "bpsk") (1/j)*ps.bpsk(snr)
  else if(mod == "qpsk") (1/j)*ps.qpsk.dh(snr)
#  else if(mod == "qpsk") (1/j)*ps.qpsk(snr)
  else if(mod == "qam16") (1/j)*ps.16qam(snr)
  else if(mod == "qam64") (1/j)*ps.64qam(snr)
#  else if(mod == "qam16") (1/j)*ps.mqam(snr,16)
#  else if(mod == "qam64") (1/j)*ps.mqam(snr,64)
  else if(mod == "dbpsk") pb.dbpsk(snr)
  else if(mod == "dqpsk") pb.dqpsk(snr)
}

pb.inv <- function(ber,mod){
  j <- bits.per.sym(mod)
  if(mod == "bpsk") ps.bpsk.inv(ber*j)
  else if(mod == "qpsk") ps.qpsk.inv.dh(ber*j)
# else if(mod == "qpsk") ps.qpsk.inv(ber*j)
  else if(mod == "qam16") ps.16qam.inv(ber*j)
  else if(mod == "qam64") ps.64qam.inv(ber*j)
# else if(mod == "qam16") ps.mqam.inv(ber*j,16)
# else if(mod == "qam64") ps.mqam.inv(ber*j,64)

}

# From: http://msenux.redwoods.edu/math/R/StandardNormal.php
stand.norm <- function(x){
  1/sqrt(2*pi)*exp(-x^2/2)
}

# From Pursley \textit{et al.} Properties and Performance of the IEEE 802.11b
# Complementary-Code-Key Signal Sets. IEEE Trans on Comms. Feb. 2009.
pu.cck <- function(snr,k,l2=8){
  n <- k/2
  i <- c()
  # fake vectorization
  b = sqrt(2*snr)
  if(length(b) > 1){
    for(bprime in b){
      # eq. 18
      integrand <- function(x){ ((2*stand.norm(x+bprime)-1)^(n-1)) *
        (exp((-x^2)/2)/sqrt(2*pi)) }
      i <- append(i,integrate(integrand,lower=-bprime,upper=Inf)$value)
    }
  }else{
    integrand <- function(x){ ((2*stand.norm(x+b)-1)^(n-1)) *
      (exp((-x^2)/2)/sqrt(2*pi)) }
    i <- integrate(integrand,lower=-b,upper=Inf)$value
  }
  pe2 <- 1 - i
```

```

  # eq. 20
  1 - (1 - pe2)^12
}

# pu is probability of uncorrectable symbol error
# cr is coding rate, k is number of subcarriers
# k is 1 for non OFDM and usually 52 for 802.11-style OFDM
pu.mod <- function(snr,mod,cr,k){
  if(mod == "cck16") pu.cck(snr,16)
  else if(mod == "cck256") pu.cck(snr,256)
  else{
    j <- bits.per.sym(mod)
    m <- k*j # number of total bits
    t <- 0   # number of correctable bits
    if(cr == (1/2)) t <- 4
    else if(cr == (2/3)) t <- 2
    else if(cr == (3/4)) t <- 2
    else if(cr == 1) t <- 0

    if(consider.coding && (t > 0)) choose(m,t+1)*(pb(snr,mod)^(t+1))
    else pb(snr,mod)
  }
}

pu.mod.inv <- function(ber,mod,cr,k){
  j <- bits.per.sym(mod)
  m <- k*j # number of total bits
  t <- 0   # number of correctable bits
  if(cr == (1/2)) t <- 4
  else if(cr == (2/3)) t <- 2
  else if(cr == (3/4)) t <- 2
  else if(cr == 1) t <- 0

  if(consider.coding && (t > 0)) pb.inv((ber/choose(m,t+1))^(1/(t+1)),mod)
  else pb.inv(ber,mod)
}

pu <- function(snr,rate){
  ofdm.k <- 52 # 48 + 4 pilots
  cr <- NULL   # coding rate
  k <- NULL    # number of subcarriers
  mod <- NULL

  # values from 802.11 spec table 17-3
  if(rate == 1){
    mod <- "dbpsk"
    k <- 1
    cr <- 1
  }else if(rate == 2){
    mod <- "dqpsk"
    cr <- 1
    k <- 1
  }else if(rate == 6){
    mod <- "bpsk"
    k <- ofdm.k
    cr <- 1/2
  }else if(rate == 9){
    mod <- "bpsk"
    cr <- 3/4
    k <- ofdm.k
  }else if(rate == 12){
    mod <- "qpsk"
    cr <- 1/2
    k <- ofdm.k
  }else if(rate == 18){
    mod <- "qpsk"
```

```
    cr <- 3/4
    k <- ofdm.k
  }else if(rate == 24){
    mod <- "qam16"
    cr <- 1/2
    k <- ofdm.k
  }else if(rate == 36){
    mod <- "qam16"
    cr <- 3/4
    k <- ofdm.k
  }else if(rate == 48){
    mod <- "qam64"
    cr <- 2/3
    k <- ofdm.k
  }else if(rate == 54){
    mod <- "qam64"
    cr <- 3/4
    k <- ofdm.k
  }else if(rate == 11){
    mod <- "cck256"
    cr <- 1
    k <- 1
  }else if(rate == 5){
    mod <- "cck16"
    cr <-1
    k <- 1
  }
  pu.mod(snr,mod,cr,k)
}

pu.inv <- function(ber,rate){
  ofdm.k <- 52 # 48 + 4 pilots
  cr <- NULL    # coding rate
  k <- NULL     # number of subcarriers
  mod <- NULL

  # values from 802.11 spec table 17-3
  if(rate == 6){
    mod <- "bpsk"
    k <- ofdm.k
    cr <- 1/2
  }else if(rate == 9){
    mod <- "bpsk"
    cr <- 3/4
    k <- ofdm.k
  }else if(rate == 12){
    mod <- "qpsk"
    cr <- 1/2
    k <- ofdm.k
  }else if(rate == 18){
    mod <- "qpsk"
    cr <- 3/4
    k <- ofdm.k
  }else if(rate == 24){
    mod <- "qam16"
    cr <- 1/2
    k <- ofdm.k
  }else if(rate == 36){
    mod <- "qam16"
    cr <- 3/4
    k <- ofdm.k
  }else if(rate == 48){
    mod <- "qam64"
    cr <- 2/3
    k <- ofdm.k
  }else if(rate == 54){
    mod <- "qam64"
```

```
    cr <- 3/4
    k <- ofdm.k
  }
  pu.mod.inv(snr,mod,cr,k)
}

erate <- function(snr,rate){
  rate*(1 - pu(snr,rate))
}

# Receiver minimum input sensitivity from the 802.11 spec Table 17-13
rmis <- function(r){
  if(r == 1) -85
  else if(r == 2) -84
  else if(r == 6) -82
  else if(r == 12) -80
  else if(r == 24) -77
  else if(r == 36) -73
  else if(r == 48) -69
  else if(r == 54) -68
  else 0
}
```

C.4 Spatial Simulated Annealing

The following R code performs spatial simulated annealing. It uses the code listed in C.1 and the roughness function that follows next. It assumes there is a list of candidate sample locations named “candidates” from which measurements locations are selected.

```
# simulated annealing
n <- 50
tmax <- 2000
dcoords.new <- NULL
num.children <- 12
parallelize <- TRUE

# first argument is a period-separated list of indices into the
# candidates dataframe
e <- commandArgs(TRUE)[1]
e <- as.numeric(unlist(strsplit(e, "\\.")))
de <- candidates[e,]
runid <- commandArgs(TRUE)[2]
e_before <- e

t <- tmax
if(parallelize){
  c1 <- makeForkCluster(num.children)
  kv <- krige.var.par(rbind(dcoords,de[,c("x","y")]),loci,kc,c1)
}else{
  kv <- krige.var(rbind(dcoords,de[,c("x","y")]),loci,kc)
}
vmap <- flipud(matrix(kv,nrow=height,ncol=width,byrow=TRUE))
fitness <- wpe(rmap,vmap)
fitness2 <- mean(sqrt(vmap))
first.fitness <- fitness
first.fitness2 <- fitness2
rm(kv,vmap)

```

```
linear.cooling = FALSE

log <- NULL

while(t > 0){
  e2 <- e[sample(seq(1,n),n-1)] # n - 1 sized sample of indices
  de2 <- candidates[e2,]
  while(length(e2) < n){
    p <- sample(seq(1,nrow(candidates)),1)
    if(any(e2 == p)) next
    e2 <- append(e2,p)
    de2 <- rbind(de2,candidates[p,])
  }
  if(parallelize){
    kv <- krige.var.par(rbind(dcoords,de2[,c("x","y")]),loci,kc,c1)
  }else{
    kv <- krige.var(rbind(dcoords,de2[,c("x","y")]),loci,kc)
  }
  vmap <- flipud(matrix(kv,nrow=height,ncol=width,byrow=TRUE))
  new.fitness <- wpe(rmap,vmap)
  new.fitness2 <- mean(sqrt(vmap))
  rm(vmap,kv)

  replaced <- TRUE
  deltaf <- new.fitness-fitness
  p <- NA
  if(deltaf < 0){
    fitness <- new.fitness
    fitness2 <- new.fitness2
    e <- e2
    de <- de2
  }else{
    if(linear.cooling){
      p <- t/tmax
    }else{
      temp <- t/tmax
      # scale up deltaf by 10^2 to get a more meaningful cooling curve
      p <- exp(-100.0*deltaf/temp)
    }
    print(paste(t,"worse :(",fitness,fitness2,deltaf,p))
    if(runif(1) <= p){
      print("accepted badness")
      fitness <- new.fitness
      fitness2 <- new.fitness2
      e <- e2
      de <- de2
    }else{
      replaced <- FALSE
    }
  }
  t <- t - 1
  log <- rbind(log,data.frame(t=t,replaced=replaced,p=p,fitness=fitness,
               fitness2=fitness2,deltaf=deltaf))
}

if(parallelize){
  stopCluster(c1)
}

wpe.gain <- first.fitness-fitness
akv.gain <- first.fitness2-fitness2

etime <- as.numeric(Sys.time())

print(cat("FITNESS",first.fitness,fitness,first.fitness2,fitness2,etime,""))
print(cat("SAMPLE",e,""))
```

```
save(n, first.fitness, first.fitness2, dcoords, wpe.gain, akv.gain, fitness,
      fitness2, etime, e_before, e, log, tmax, candidates,
      file=paste(sep="", "sa_slave_", runid, "_", etime, ".RData"))

print(cat("DONE", ""))
```

The following code computes the roughness map given a map (as a matrix). A function to compute the WPE using this roughness map and the kriging variance map are also provided.

```
# calculate roughness
roughness <- function(map, height, width, nr=1, pix.per.m=0.2, beta=1.5, alpha=1.0){
  height <- nrow(map)
  width <- ncol(map)
  ret <- map
  neigh <- expand.grid(seq(-nr, nr), seq(-nr, nr))
  dsum <- 0.0

  for(k in nrow(neigh)){
    x <- neigh[k,1]
    y <- neigh[k,2]
    if(x == 0 && y == 0) next;
    d <- sqrt(x^2 + y^2)/pix.per.m
    dsum <- dsum + d
  }
  for(i in seq(1, height)){
    for(j in seq(1, width)){
      s <- 0.0
      v <- map[i, j]
      for(k in nrow(neigh)){
        x <- neigh[k,1]
        y <- neigh[k,2]
        xi <- x + j
        yi <- y + i
        if(x == 0 && y == 0) next;
        if(xi < 1 || yi < 1 || xi > width || yi > height) next;
        d <- sqrt(x^2 + y^2)/pix.per.m
        v2 <- map[yi, xi]
        s <- s + (d^(-beta) * (v2 - v)^2)/dsum
      }
      ret[i, j] <- s
    }
  }
  ret <- (ret/max(ret))^alpha
  return(ret)
}

wpe <- function(rmap, vmap){
  # note this is not a matrix mult (%*%) so will just multiple rmap[i,i]*vmap[i,i]
  mean(rmap*vmap)
}
```

C.5 Variogram Fitting and Kriging

A simplified (but still largely complete) version of the variogram fitting and kriging code, utilizing the geoR library, is provided below.

```r
library(geoR) # for all the kriging stuff
library(lattice) # xyplot and friends
library(dichromat) # for ramp()
library(matlab) # for flipud()

guess.range <- function(v){
  lastval <- 0
  thisi <- 1
  for(i in 1:length(v$v)){
    thisval <- v$v[i]
    if(thisval < lastval) break;
    thisi <- i
    lastval <- thisval
  }
  v$u[i]
}

krige.per.ap <- function(fname, subtitle, fnsubtitle, lag, ap, nug.tol, d, metric, freq,
                         noiseval, max.dist.m, na.value, p.tx, data.combined=FALSE){

  minval = min(d$sig, na.rm=TRUE)
  maxval = max(d$sig, na.rm=TRUE)
  valrange = maxval-minval

  d2 <- d
  d2$non <- FALSE
  d2[is.na(d2$sig),"non"] <- TRUE
  d2[is.na(d2$sig),"sig"] <- na.value # d2 has NA replaced with na.value

  # convert signal to total PL (if possible)
  if(metric == "snr" || metric == "esnr6" || metric == "esnr54"){
    # SNR = P_tx - (N + PL) = P_tx - N - PL
    # t.f. PL = P - N - SNR
    d2$sig <- p.tx - noiseval - d2$sig
    print(paste("NA Value in PL = ",p.tx - noiseval - na.value,
      " Versus minimum PL observed = ",min(d2$sig)))
    metric <- "pl"
  }else if(metric == "rss"){
    d2$sig <- p.tx - d2$sig
    metric <- "pl"
  }else if(metric == "tput"){
    d2$sig <- (d2$sig-minval)/valrange
  }

  sigma2 <- NA
  if(metric == "pl" && !all(is.na(d$dist))){
    # Fit friis' PL to data
    m2 <- lm(sig ~ log10(dist),data=d2)

    print("Model Fitting Summary")
    print(summary(m2))

    slope2 <- m2$coefficients[2]
    intercept2 <- m2$coefficients[1]
    alpha2 <- slope2/10
    epsilon2 <- intercept2 - 20*log10(freq) - 32.45

    sigma2 <- round(summary(m2)$sigma,3)

    # sig2 is PL reduced by frii's trivial PL
    d2$sig2 <- d2$sig - friis(d2$dist,freq,alpha2,epsilon2)
  }else{
    # Don't know how to remove trend for other metrics, so just
    # do nothing...
    d2$sig2 <- d2$sig
  }

```

```
n <- nrow(d2)

# d1 has null measurements excluded
d1 <- d2[!d2$non,]

d1coords <- data.frame(x=d1$east, y=d1$north)
d2coords <- data.frame(x=d2$east, y=d2$north)

# if we are combing across several APs, co-located points must be jittered
if(data.combined){
  print("jittering duplicated coordinates by up to 20 wavelengths")
  # jitter up to 20 wavelengths
  maxjitter <- 2*freq.to.wavelength(freq/1000.0)*20.0
  # is it bad that these are being independently jittered?
  d2coords <- jitter2d(d2coords,max=maxjitter)
  d1coords <- jitter2d(d1coords,max=maxjitter)
}

eastrng <- range(d2coords$x)
northrng <- range(d2coords$y)

# representative example of detrended, truncated, and with null measurements
# given, although this may not be the best performing model for all scenarios

# compute empirical variogram
v2.detrend.trunc <- variog(coords=d2coords, data=d2$sig2,
  nugget.tolerance=nug.tol, option="bin",max.dist=max.dist.m)

# perform fitting
range.ini <- guess.range(v2.detrend.trunc)
nug.ini <- v2.detrend.trunc$v[1]
sill.ini <- max(v2.detrend.trunc$v)-nug.ini
v2.detrend.trunc.fit.gauss <- variofit(v2.detrend.trunc, cov.model="gaussian",
  ini.cov.pars=c(sill.ini, range.ini), nugget=nug.ini, fix.nugget=TRUE)
v2.detrend.trunc.fit.cubic <- variofit(v2.detrend.trunc, cov.model="cubic",
  ini.cov.pars=c(sill.ini, range.ini), nugget=nug.ini, fix.nugget=TRUE)

n.sample.max <- 50   # max points to validate
n.sample.frac <- 0.2 # fraction of points to validate
n.folds <- 10
n.sample <- min(c(n.sample.max, ceil(n.sample.frac*length(d2$sig2))))

# try both cubic and gaussian fits and keep whichever is better
best.model <- NA
best.model.truncated <- TRUE
best.model.rmse <- NA
best.model.name <- NA
best.model.non <- NA

v <- do.validate(d2,d2coords,v2.detrend.fit.gauss,n.sample,n.folds)
best.model <- v2.detrend.fit.gauss
best.model.truncated <- FALSE
best.model.rmse <- mean(v$rmse)
best.model.name <- "Gaussian w/ Null"
best.model.non <- TRUE
fitstats <- rbind(fitstats, data.frame(m=v2.detrend.fit.gauss$cov.model,
  ssq=v2.detrend.fit.gauss$value, sigmasq=v2.detrend.fit.gauss$cov.pars[1],
  phi=v2.detrend.fit.gauss$cov.pars[2], kappa=v2.detrend.fit.gauss$kappa,
  tausq=v2.detrend.fit.gauss$nugget, ap=ap, wneg=TRUE, truncated=FALSE,
  lag=lag, n=n, xv.rmse.mean=mean(v$rmse), xv.rmse.std=std(v$rmse),
  mq90=mean(v$q90), mq75=mean(v$q75), mq100=mean(v$q100),
  xv.mskv.mean=mean(v$mskv), xv.rmse.std=std(v$mskv), sigma1=sigma1,
  sigma2=sigma2, sigma3=sigma3))

v <- do.validate(d2,d2coords,v2.detrend.fit.cubic,n.sample,n.folds)
if(mean(v$rmse) < best.model.rmse){
  best.model <- v2.detrend.fit.gauss
```

```
      best.model.truncated <- FALSE
      best.model.rmse <- mean(v$rmse)
      best.model.name <- "Cubic w/ Null"
      best.model.non <- TRUE
    }
    fitstats <- rbind(fitstats, data.frame(m=v2.detrend.fit.cubic$cov.model,
      ssq=v2.detrend.fit.cubic$value, sigmasq=v2.detrend.fit.cubic$cov.pars[1],
      phi=v2.detrend.fit.cubic$cov.pars[2], kappa=v2.detrend.fit.cubic$kappa,
      tausq=v2.detrend.fit.cubic$nugget, ap=ap, wneg=TRUE, truncated=FALSE, lag=lag,
      n=n, xv.rmse.mean=mean(v$rmse), xv.rmse.std=std(v$rmse), mq90=mean(v$q90),
      mq75=mean(v$q75), mq100=mean(v$q100), xv.mskv.mean=mean(v$mskv),
      xv.rmse.std=std(v$mskv), sigma1=sigma1, sigma2=sigma2, sigma3=sigma3))



    width <- round(diff(range(d2coords$x))*pix.per.meter)
    height <- round(diff(range(d2coords$y))*pix.per.meter)
    loci2 <- expand.grid(seq(min(d2coords$x),max(d2coords$x),length.out=width),
      seq(min(d2coords$y),max(d2coords$y),length.out=height))

    dk <- d2
    dkcoords <- d2coords

    if(!best.model.non && !best.model.truncated){
      dk <- d1
      dkcoords <- d1coords
    }else if(!best.model.non && best.model.truncated){
      dk <- d1
      dkcoords <- d1coords
    }
    tryCatch(do.krige(best.model, dk, dkcoords, loci2, "best", best.model.name),
             error=function(err){ print(paste("ERROR kriging: ", err)); return(0) })
    print(fitstats)
    list(fitstats=fitstats, eastrng=eastrng, northrng=northrng, width=width, height=height)
}

do.validate <- function(d2, d2coords, model, n.sample, n.folds){
    valdata <- NULL
    for(i in seq(1,n.folds)){
      tryCatch(x <- xvalid(coords=d2coords, data=d2$sig2, model=model,
        locations.xvalid=sample(seq(1,length(d2$sig2)),n.sample)),
        error=function(err){ print(paste("ERROR xvalid: ", err)); return(0) })
      if(length(x) > 1){
        x.rmse <- sqrt(mean((x$error)^2))
        x.mskv <- sqrt(mean(x$krige.var))
        q <- quantile(abs(x$error), probs=c(0.75,0.9,1.0))
        valdata <- rbind(valdata, data.frame(n=n.sample, f=i, rmse=x.rmse,
          mskv=x.mskv, q75=q[1], q90=q[2], q100=q[3]))
      }
    }
    valdata
}

do.krige <- function(model, d, dcoords, loci, name, prettyname, local=FALSE,
    n.local=8, universal=FALSE){

    kc <- krige.control(type.krige="ok", obj.model=model)
    k <- krige.conv(coords=dcoords, data=d$sig2, locations=loci, krige=kc)

    write.table(flipud(matrix(k$predict, nrow=height, ncol=width, byrow=TRUE)),
                file=paste(sep="/", fig.dir, paste(sep="_", ap, fnsubtitle,
                eastrng[1], eastrng[2], northrng[1],
                northrng[2], pix.per.meter, name, "detrend_map.csv")))

    ape <- d$apeast[1] # all rows should be the same
    apn <- d$apnorth[1] # ...
    if(metric == "pl" && !all(is.na(d$dist))){
```

```
    for(i in seq(1,length(loci[,1]))){
      e <- loci[i,1]
      n <- loci[i,2]
      # distance between grid point and AP in km
      dist <- sqrt((ape-e)^2 + (apn-n)^2)/1000

      # convert back to signal strength
      k$predict[i] <- p.tx - (k$predict[i] + friis(dist,freq,alpha2,epsilon2))
    }
  }

  print(paste("saving map to file",paste(sep="/",fig.dir,paste(sep="_",ap,
              fnsubtitle,eastrng[1],eastrng[2],northrng[1],
              northrng[2],pix.per.meter,name,"map.csv"))))
  write.table(flipud(matrix(k$predict,nrow=height,ncol=width,byrow=TRUE)),
              file=paste(sep="/",fig.dir,paste(sep="_",ap,fnsubtitle,eastrng[1],
              eastrng[2],northrng[1],
              northrng[2],pix.per.meter,name,"map.csv")))
  write.table(flipud(matrix(k$krige.var,nrow=height,ncol=width,byrow=TRUE)),
             file=paste(sep="/",fig.dir,paste(sep="_",ap,fnsubtitle,
             eastrng[1],eastrng[2],northrng[1],northrng[2],pix.per.meter,name,
             "var_map.csv")))


  # make sure everything gets cleaned up
  k <- NULL
  gc(verbose=TRUE)
}
```

C.6 Anritsu National Instruments Interface

The following C code implements a network-based communication interface to an Anritsu MS2712B portable spectrum analyzer. It was used to partially automate data collection for the experiments described in section 6.1 and 8.

```
#include "stdlib.h"
#include "stdio.h"
#include "unistd.h"
#include "string.h"
#include "time.h"
#include "visa.h"

#define NO_ERROR 0
#define USAGE_ERROR 1
#define VISA_ERROR 2

#define BUFFER_SIZE 512

#define SIGNAL_STANDARD 9
#define CHANNEL_BANDWIDTH 3

#define DEVICE_TIMEOUT 30

#define SWITCH_TO_WIMAX 0
#define ENABLE_GPS 1

void usage(){
  fprintf(stderr,"Usage: ./measure <IP Address> <channel1,channel2,channel3> \
```

```
<num_measurements>\n");
    exit(USAGE_ERROR);
}

int do_read_write(ViSession instr, const char *cmd){
    ViStatus status;
    ViUInt32 retCount;
    ViChar vbuffer[BUFFER_SIZE];
    char cbuffer[BUFFER_SIZE];

    sprintf(vbuffer,"%s",cmd);
    if((status = viWrite(instr,(unsigned char *)&vbuffer[0],strlen(vbuffer),
      &retCount)) < VI_SUCCESS){

      viStatusDesc(instr,status,vbuffer);
      fprintf(stderr,"VISA Write Error: %s\nCommand Was: %s",vbuffer,cmd);
      return VISA_ERROR;
    }
    if((status = viRead(instr,(unsigned char *)vbuffer,BUFFER_SIZE,&retCount)) <
      VI_SUCCESS){

      viStatusDesc(instr,status,vbuffer);
      fprintf(stderr,"VISA Read Error: %s\nCommand Was: %s",vbuffer,cmd);
      return VISA_ERROR;
    }
    strncpy(cbuffer,vbuffer,retCount);
    cbuffer[retCount] = 0; // null terminate
    printf("%d: %s\n",(int)retCount,cbuffer);
    return NO_ERROR;
}

int do_write(ViSession instr, const char *cmd, int post_sleep){
    ViStatus status;
    ViUInt32 retCount;
    ViChar vbuffer[BUFFER_SIZE];
    char cbuffer[BUFFER_SIZE];
    sprintf(vbuffer,"%s",cmd);
    if((status = viWrite(instr,(unsigned char *)&vbuffer[0],strlen(vbuffer),
                         &retCount)) < VI_SUCCESS){

      viStatusDesc(instr,status,vbuffer);
      fprintf(stderr,"VISA Write Error: %s\nCommand Was: %s",vbuffer,cmd);
      return VISA_ERROR;
    }
    sleep(post_sleep);
    return NO_ERROR;
}

// http://www.ni.com/pdf/manuals/370132c.pdf

int main(int argc, char* argv[]){
    ViStatus status;
    ViSession defaultRM, instr;
    ViUInt32 retCount;
    char cbuffer[BUFFER_SIZE];
    char tbuffer[BUFFER_SIZE];
    char *addr, *tok, *channels;
    time_t rawtime;
    struct tm *now;
    int chan, num_measurements;

    if(argc < 4) usage();

    addr = argv[1];
    channels = argv[2];
    num_measurements = atoi(argv[3]);
    status = viOpenDefaultRM(&defaultRM);
```

```
  if(status < VI_SUCCESS){
    fprintf(stderr,"Can't initialize VISA\n");
    return VISA_ERROR;
  }
  sprintf(cbuffer,"TCPIP0::%s::INSTR",addr);
  status = viOpen(defaultRM,cbuffer,VI_NULL,VI_NULL,&instr);
  status = viSetAttribute(instr,VI_ATTR_TMO_VALUE,DEVICE_TIMEOUT*1000);

  do_read_write(instr,"*IDN?\n");

  if(SWITCH_TO_WIMAX) do_write(instr,":INSTrument:SELect \"WIMAX_E\"\n",30);
  if(ENABLE_GPS) do_write(instr,":SENSe:GPS ON\n",5);

  tok = strtok(channels,",");
  while(tok != NULL){
    chan = atoi(tok);
    printf("Channel %d\n",chan);
    printf("Setting Standard (%d), Channel (%d), and Bandwidth (%d)\n",
      SIGNAL_STANDARD,chan,CHANNEL_BANDWIDTH);
    sprintf(cbuffer,":SENSe:FREQuency:SIGSTANDARD %d\n",SIGNAL_STANDARD);
    do_write(instr,(const char *)cbuffer,2);
    sprintf(cbuffer,":SENSE:FREQUENCY:SIGSTANDARD:CHANNEL %d\n",chan);
    do_write(instr,(const char *)cbuffer,2);
    sprintf(cbuffer,":SENSe:BANDwidth %d\n",CHANNEL_BANDWIDTH);
    do_write(instr,(const char *)cbuffer,2);
    for(int i = 0; i < num_measurements; i++){
      fflush(stdout);
      time(&rawtime);
      now = localtime(&rawtime);
      strftime(tbuffer,BUFFER_SIZE,"%Y%m%d%H%M%S",now);
      printf("Doing measurements %d of %d @ %s\n",i+1,num_measurements,tbuffer);
      if(ENABLE_GPS) do_read_write(instr,":FETCh:GPS?");
      printf(" => Configuration\n");
      sprintf(cbuffer,":MMEMory:STORe:STATe 0,\"con%s\"\n",tbuffer);
      do_write(instr,(const char *)cbuffer,5);
      printf(" => Summary\n");
      do_write(instr,":CONFigure:DEMod SUMMary\n",10);
      sprintf(cbuffer,":MMEMory:STORe:TRACe 0,\"sum%s\"\n",tbuffer);
      do_write(instr,(const char *)cbuffer,5);
      printf(" => Spectrum Flatness\n");
      do_write(instr,":CONFigure:DEMod SFL\n",10);
      sprintf(cbuffer,":MMEMory:STORe:TRACe 0,\"sfl%s\"\n",tbuffer);
      do_write(instr,(const char *)cbuffer,5);
      printf(" => Constellation Plot\n");
      do_write(instr,":CONFigure:DEMod CONSTln\n",10);
      sprintf(cbuffer,":MMEMory:STORe:TRACe 0,\"cns%s\"\n",tbuffer);
      do_write(instr,(const char *)cbuffer,5);
      if(ENABLE_GPS) do_read_write(instr,":FETCh:GPS?");
    }
    fflush(stdout);
    tok = strtok(NULL,",");
  }

  status = viClose(instr);
  status = viClose(defaultRM);

  return NO_ERROR;
}
```