Kernels and BO in Julia

David Bindel

2025-06-19

Table of contents

Introduction

Given that there are several of us working on kernel methods and Bayesian optimization at the moment, it seems worth giving a bare-bones treatment of the computational pieces. We will make some effort to be at least a little efficient and to show off how to use the Julia language effectively, but will not worry too much about all the details, nor will we try to make this as efficient as at all possible.

The basic moving pieces are:

- Choosing an initial sample
- Approximating a function with a kernel
- Choosing kernel hyperparameters
- Gradients of posterior means and variances
- Some standard acquisition functions
- Optimization of acquisition functions
- Additional numerical tricks

Initial sample

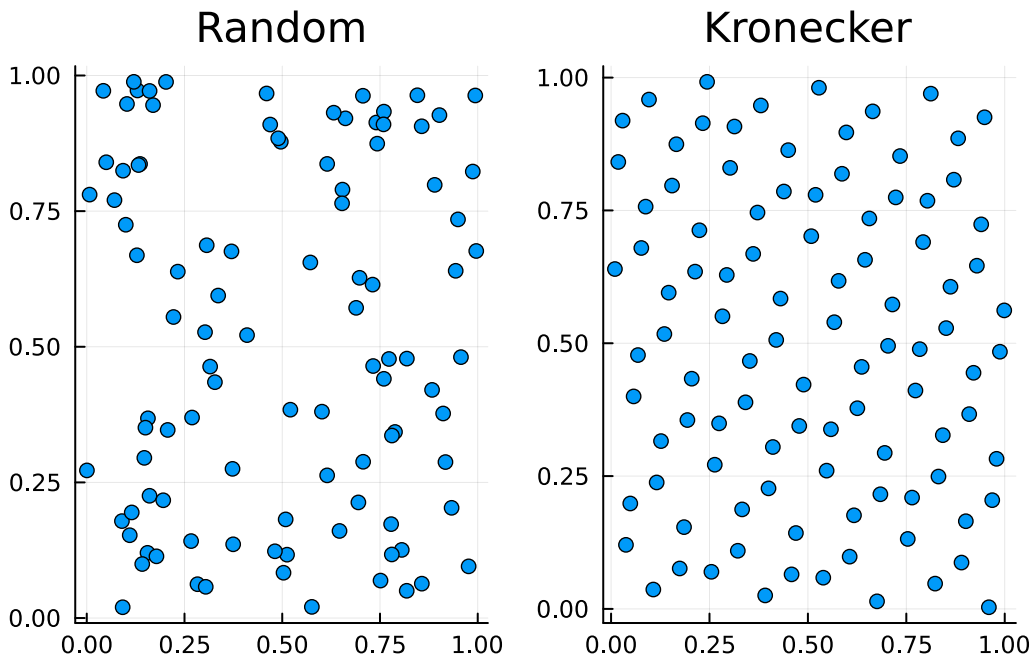
In general, we are interested in functions on a hypercube in $[0,1]^d$, though we can deal with more general regions. For many of us, our first inclination is either random sampling or sampling on a grid. However, choosing independent random samples tends to lead to "clumps" of points in some parts of the domain, and so is not terribly efficient; and sampling on a grid tends to get expensive as d grows. Therefore, we usually default to either statistical experimental design methods like Latin hypercubes, or we use a *low-discrepancy* sequence, usually generated by something that looks like a low-quality random number generator.

There is a nice blog post on low discrepancy sequences that gives some of the relevant background and recommends a fairly effective sampler based on Kronecker sequences. We provide some Julia code for this below.

```
"""
    kronecker_quasirand(d, N, start=0)

Return an `d`-by-`N` array of `N` quasi-random samples in [0, 1]^d
generated by an additive quasi-random low-discrepancy sample sequence.
"""
function kronecker_quasirand(d, N, start=0)

    # Compute the recommended constants ("generalized golden ratio")
    ϕ = 1.0+1.0/d
    for k = 1:10
        gϕ = ϕ^(d+1)-ϕ-1
        dgϕ= (d+1)*ϕ^d-1
        ϕ -= gϕ/dgϕ
    end
    αs = [mod(1.0/ϕ^j, 1.0) for j=1:d]

    # Compute the quasi-random sequence
    Z = zeros(d, N)
    for j = 1:N
        for i=1:d
            Z[i,j] = mod(0.5 + (start+j)*αs[i], 1.0)
        end
    end

    Z
end
```

```
kronecker_quasirand
```

A 2D plot shows the difference between using random samples and the low-discrepancy sequence.



Finite difference checks

A finite-difference tester is a useful thing to have.

```
function check_approx(xref, x; rtol=1e-6, atol=0.0)
    abserr = norm(xref-x)
    refnorm = norm(xref)
    if abserr > rtol*refnorm + atol
        error("Check failed ($x v $xref): $abserr > $rtol * $refnorm + $atol")
    end
    abserr/refnorm
end

diff_fd(f, x; h=1e-6) = (f(x+h)-f(x-h))/(2h)

check_fd(df_ref, f, x; h=1e-6, rtol=1e-6, atol=0.0) =
    check_approx(df_ref, diff_fd(f, x, h=h), rtol=rtol, atol=atol)
```

```
check_fd (generic function with 1 method)
```

Kernel functions

A kernel function is a symmetric positive definite function $k : \Omega \times \Omega \to \mathbb{R}$. Here positive definiteness means that for any set of distinct points $x_1, \ldots, x_n$ in Ω, the matrix with entries $k(x_i, x_j)$ is positive definite. We usually denote the ordered list of points x_i as X, and write K_{XX} for the kernel matrix, i.e. the matrix with entries $k(x_i, x_j)$. When the set of points is obvious from context, we will abuse notation slightly and write K rather than K_{XX} for the kernel matrix.

Common kernels

We are generally interested in kernel functions that are stationary (invariant under translation) and isotropic (invariant under rotation). In these cases, we can write the kernel as

$$k(x, y) = \phi(\|x - y\|)$$

where ϕ is sometimes called a *radial basis function*. We often define parametric families of kernels by combining a standard radial basis function with a length scale parameter ℓ, i.e.

$$k(x, y) = \phi(\|x - y\|/\ell)$$

In terms of the scaled distance $s = \|x - y\|/\ell$, some common positive definite radial basis functions include:

Name	$\phi(s)$	Smoothness
Squared exponential	$\exp(-s^2/2)$	C^∞
Matern 1/2	$\exp(-s)$	C^0
Matern 3/2	$(1 + \sqrt{3}s)\exp(-\sqrt{3}s)$	C^1
Matern 5/2	$(1 + \sqrt{5}s + 5s^2/3)\exp(-\sqrt{5}s)$	C^2
Inv quadratic	$(1 + s^2)^{-1}$	C^∞
Inv multiquadric	$(1 + s^2)^{-1/2}$	C^∞
Rational quadratic	$(1 + s^2)^{-\alpha}$	C^∞

When someone writes about "the" radial basis function, they are usually referring to the squared exponential function. We will frequently default to the squared exponential function in our examples. When we go beyond the squared exponential function, we will most often use the Matern 5/2, inverse quadratic, or inverse multiquadric kernel, since we usually want more smoothness than the Matern 1/2 and Matern 3/2 kernels provide.

In code, these functions are

```
ϕ_SE(s) = exp(-s^2/2)
ϕ_M1(s) = exp(-s)

ϕ_M3(s) = let t = sqrt(3)*s
    (1+t)*exp(-t)
end

ϕ_M5(s) = let t = sqrt(5)*s
    (1+t*(1+t/3))*exp(-t)
end

ϕ_IQ(s) = 1/(1+s^2)
ϕ_IM(s) = 1/sqrt(1+s^2)
ϕ_RQ(s; α=1.0) = (1+s^2)^-α
```

```
ϕ_RQ (generic function with 1 method)
```

For later work, we will need functions for $\phi(s)$, $\phi'(s)/s$, $\phi'(s)$, and $\phi''(s)$. It is helpful to pack these all together in a single function definition.

```
function Dϕ_SE(s)
    ϕ = exp(-s^2/2)
    dϕ_div = -ϕ
    dϕ = dϕ_div*s
    Hϕ = (-1+s^2)*ϕ
    ϕ, dϕ_div, dϕ, Hϕ
end

function Dϕ_M1(s)
    ϕ = exp(-s)
    ϕ, -ϕ/s, -ϕ, ϕ
end

function Dϕ_M3(s)
    t = √3*s
    ψ = exp(-t)
    ϕ = (1+t)*ψ
    dϕ_div = -3*ψ
    dϕ = dϕ_div*s
    Hϕ = 3*(t-1)*ψ
    ϕ, dϕ_div, dϕ, Hϕ
```

```
end

function Dϕ_M5(s)
    t = √5*s
    ψ = exp(-t)
    ϕ = (1+t*(1+t/3))*ψ
    dϕ_div = -5/3*(1+t)*ψ
    dϕ = dϕ_div*s
    Hϕ = -5/3*(1+t*(1-t))*ψ
    ϕ, dϕ_div, dϕ, Hϕ
end

function Dϕ_IQ(s)
    ϕ = 1/(1+s^2)
    dϕ_div = -2*ϕ^2
    dϕ = dϕ_div*s
    Hϕ = 2*ϕ^2*(4*ϕ*s^2-1)
    ϕ, dϕ_div, dϕ, Hϕ
end

function Dϕ_IM(s)
    ϕ = 1/sqrt(1+s^2)
    dϕ_div = -ϕ^3
    dϕ = dϕ_div*s
    Hϕ = ϕ^3*(3*s^2*ϕ^2-1)
    ϕ, dϕ_div, dϕ, Hϕ
end

function Dϕ_RQ(s; α=1.0)
    ϕ = (1+s^2)^-α
    dϕ_div = -α*(1+s^2)^-(α+1)*2
    dϕ = dϕ_div*s
    Hϕ = dϕ_div + α*(α+1)*(1+s^2)^-(α+2)*4*s^2
    ϕ, dϕ_div, dϕ, Hϕ
end
```

```
Dϕ_RQ (generic function with 1 method)
```

It makes sense to have a finite difference check for each of these:

```
function fd_check_Dϕ(name, Dϕ, s, h=1e-6, tol=1e-8; verbose=false, kwargs... )
    ϕ,  dϕ_div,  dϕ,  Hϕ  = Dϕ(s; kwargs... )
    ϕp, dϕp_div, dϕp, Hϕp = Dϕ(s+h; kwargs... )
    ϕm, dϕm_div, dϕm, Hϕm = Dϕ(s-h; kwargs... )
    relerr1 = abs( (dϕ_div*s-dϕ)/dϕ )
    relerr2 = abs( (dϕ-(ϕp-ϕm)/(2h))/dϕ )
    relerr3 = abs( (Hϕ-(dϕp-dϕm)/(2h))/Hϕ )
    if verbose
        println("Check $name:\t$relerr1\t$relerr2\t$relerr3")
    elseif max(relerr1, relerr2, relerr3) > tol
        error("Check $name:\t$relerr1\t$relerr2\t$relerr3")
    end
end

let s = 0.89
    fd_check_Dϕ("SE", Dϕ_SE, s)
    fd_check_Dϕ("M1", Dϕ_M1, s)
    fd_check_Dϕ("M3", Dϕ_M3, s)
    fd_check_Dϕ("M5", Dϕ_M5, s)
    fd_check_Dϕ("IQ", Dϕ_IQ, s)
    fd_check_Dϕ("IM", Dϕ_IM, s)
    fd_check_Dϕ("RQ", Dϕ_RQ, s; α=0.75)
end
```

A brief aside: Having coded and tested the various derivative functions, we make the checker quiet unless some relative error is too big. During development, I usually would use the checker in verbose mode.

Distance functions

There are several ways to compute Euclidean distance functions in Julia. The most obvious ones (e.g. `norm(x-y)`) involve materializing an intermediate vector. Since we will be doing this a lot, we will write a loopy version that runs somewhat faster.

```
function dist2(x :: AbstractVector{T}, y :: AbstractVector{T}) where {T}
    s = zero(T)
    for k = 1:length(x)
        dk = x[k]-y[k]
        s += dk*dk
    end
    s
```

```
end

dist(x :: AbstractVector{T}, y :: AbstractVector{T}) where {T} =
    sqrt(dist2(x,y))
```

```
dist (generic function with 1 method)
```

Kernel contexts

We define a *kernel context* as "all the stuff you need to work with a kernel." This includes the type of the kernel, the dimension of the space, and any hyperparameters.

```
abstract type KernelContext end
(ctx :: KernelContext)(args ... ) = kernel(ctx, args ... )
```

All of the kernels we work with are based on radial basis functions, and have the form

$$k(x, y) = \phi(\|x - y\|/\ell)$$

where ℓ is a length scale parameter. There may be other hyperparameters as well. We therefore define an `RBFKernelContext` subtype that includes the dimension of the space as a type parameter, and define a getter function to extract that information.

```
"""
For an RBFKernelContext{d}, we should define

ϕ(ctx, s) = RBF evaluation at s
Dϕ(ctx, s) = RBF derivatives at s
nhypers(ctx) = Number of tuneable hyperparameters
getθ!(θ, ctx) = Extract the hyperparameters into a vector
updateθ(ctx, θ) = Create a new context with updated hyperparameters

We also have the predefined ndims function to extract {d}.
"""
abstract type RBFKernelContext{d} <: KernelContext end

ndims(::RBFKernelContext{d}) where {d} = d

function getθ(ctx :: KernelContext)
    θ = zeros(nhypers(ctx))
    getθ!(θ, ctx)
    θ
end
```

```
getθ (generic function with 1 method)
```

Simple RBF kernels

In most cases, the only hyperparameter we will track in the type is the length scale hyperparameter. For these subtypes of `RBFKernelContext`, we have a fairly boilerplate structure and method definition that we encode in a macro.

```
macro rbf_simple_kernel(T, ϕ_rbf, Dϕ_rbf)
    T, ϕ_rbf, Dϕ_rbf= esc(T), esc(ϕ_rbf), esc(Dϕ_rbf)
    quote
        struct $T{d} <: $(esc(:RBFKernelContext)){d}
            ℓ :: Float64
        end
        $(esc(:ϕ))(::$T, s) = $ϕ_rbf(s)
        $(esc(:Dϕ))(::$T, s) = $Dϕ_rbf(s)
        $(esc(:nhypers))(::$T) = 1
        $(esc(:getθ!))(θ, ctx :: $T) = θ[1]=ctx.ℓ
        $(esc(:updateθ))(ctx :: $T{d}, θ) where {d} = $T{d}(θ[1])
    end
end

@rbf_simple_kernel(KernelSE, ϕ_SE, Dϕ_SE)
@rbf_simple_kernel(KernelM1, ϕ_M1, Dϕ_M1)
@rbf_simple_kernel(KernelM3, ϕ_M3, Dϕ_M3)
@rbf_simple_kernel(KernelM5, ϕ_M5, Dϕ_M5)
@rbf_simple_kernel(KernelIQ, ϕ_IQ, Dϕ_IQ)
@rbf_simple_kernel(KernelIM, ϕ_IM, Dϕ_IM)
```

```
updateθ (generic function with 7 methods)
```

Kernel operations

One of the reasons for defining a kernel context is that it allows us to have a generic high-performance interface for kernel operations. The most fundammental operation, of course, is evaluating the kernel on a pair of points.

```
kernel(ctx :: RBFKernelContext, x :: AbstractVector, y :: AbstractVector) =
    ϕ(ctx, dist(x, y)/ctx.ℓ)
```

```
kernel (generic function with 5 methods)
```

The interface for computing derivatives will involve two functions: one for computing the gradient with respect to the hypers, the other for computing the Hessian. In the case of radial basis functions where the only intrinsic hyperparameter is the length scale, we have

$$\nabla_\theta k(x,y) = \left[-\phi'(s)s/\ell\right]$$
$$H_\theta k(x,y) = \left[(\phi''(s)s + 2\phi'(s))s/\ell^2\right]$$

This gives us the following generic code:

```
function gθ_kernel!(g :: AbstractVector, ctx :: RBFKernelContext,
                    x :: AbstractVector, y :: AbstractVector, c=1.0)
    ℓ = ctx.ℓ
    s = dist(x,y)/ℓ
    _, _, dϕ, _ = Dϕ(ctx, s)
    g[1] -= c*dϕ*s/ℓ
    g
end

function Hθ_kernel!(H :: AbstractMatrix, ctx :: RBFKernelContext,
                    x :: AbstractVector, y :: AbstractVector, c=1.0)
    ℓ = ctx.ℓ
    s = dist(x,y)/ℓ
    _, _, dϕ, Hϕ = Dϕ(ctx, s)
    H[1,1] += c*(Hϕ*s + 2*dϕ)*s/ℓ^2
    H
end
```

```
Hθ_kernel! (generic function with 4 methods)
```

For spatial derivatives of kernels based on radial basis functions, it is useful to write $r = x - y$ and $\rho = \|r\|$, and to write the derivative formulae in terms of r and ρ. Using $f_{,i}$ to denote the ith partial derivative of a function f, we have first derivatives

$$[\phi(\rho)]_{,i} = \phi'(\rho)\rho_{,i}$$

and second derivatives

$$[\phi(\rho)]_{,ij} = \phi''(\rho)\rho_{,i}\rho_{,j} + \phi'(\rho)\rho_{,ij}$$

The derivatives of $\rho = \|r\|$ are given by

$$\rho_{,i} = \rho^{-1} r_i = u_i$$

and

$$\rho_{,ij} = \rho^{-1}\left(\delta_{ij} - u_i u_j\right)$$

where $u = r/\rho$ is the unit length vector in the r direction. Putting these calculations together, we have

$$\nabla\phi = \phi'(\rho)u$$

and

$$H\phi = \frac{\phi'(\rho)}{\rho} I + \left(\phi''(\rho) - \frac{\phi'(\rho)}{\rho}\right) uu^T$$

This gives us the following generic code.

```
function gx_kernel!(g :: AbstractVector, ctx :: RBFKernelContext,
                    x :: AbstractVector, y :: AbstractVector, c=1.0)
    ℓ = ctx.ℓ
    d = ndims(ctx)
    ρ = dist(x,y)
    s = ρ/ℓ
    _, _, dϕ, _ = Dϕ(ctx, s)
    if ρ ≠ 0.0
        dϕ /= ctx.ℓ
        C = c*dϕ/ρ
        for i = 1:d
            g[i] += C*(x[i]-y[i])
        end
    end
    g
end

function Hx_kernel!(H :: AbstractMatrix, ctx :: RBFKernelContext,
                    x :: AbstractVector, y :: AbstractVector, c=1.0)
    ℓ = ctx.ℓ
    d = ndims(ctx)
    ρ = dist(x,y)
    s = ρ/ℓ
    _, dϕ_div, _, Hϕ = Dϕ(ctx, s)
    Hϕ /= ℓ^2
    dϕ_div /= ℓ^2
    for j = 1:d
        H[j,j] += c*dϕ_div
    end
    if ρ ≠ 0.0
        C = c*(Hϕ-dϕ_div)/ρ^2
        for j = 1:d
```

```
            xj, yj = x[j], y[j]
            for i = 1:d
                xi, yi = x[i], y[i]
                H[i,j] += C*(xj-yj)*(xi-yi)
            end
        end
    end
    H
end
```

```
Hx_kernel! (generic function with 2 methods)
```

RQ kernel

The rational quadratic case is a little more complicated, with two adjustable hyperparameters (the length scale ℓ and the exponent α).

```
struct KernelRQ{d} <: RBFKernelContext{d}
    ℓ :: Float64
    α :: Float64
end

ϕ(ctx :: KernelRQ, s) = ϕ_RQ(s; α=ctx.α)
Dϕ(ctx :: KernelRQ, s) = Dϕ_RQ(s; α=ctx.α)

nhypers(:: KernelRQ) = 2
function getθ!(θ, ctx :: KernelRQ)
    θ[1] = ℓ
    θ[2] = α
end
updateθ(ctx :: KernelRQ{d}, θ) where {d} = KernelRQ{d}(θ[1], θ[2])
```

```
updateθ (generic function with 7 methods)
```

Here we also want to compute the gradients and Hessians with respect to both ℓ and α.

```
function gθ_kernel!(g :: AbstractVector, ctx :: KernelRQ,
                    x :: AbstractVector, y :: AbstractVector, c=1.0)
    ℓ, α = ctx.ℓ, ctx.α
    s  = dist(x,y)/ℓ
```

```
    s2 = s^2
    z  = 1.0 + s2
    ϕ     = z^-α
    gℓ    = 2*α*ϕ/z * s2/ℓ
    gα    = -ϕ * log(z)
    g[1] += c*gℓ
    g[2] += c*gα
    g
end

function Hθ_kernel!(H :: AbstractMatrix, ctx :: KernelRQ,
                    x :: AbstractVector, y :: AbstractVector, c=1.0)
    ℓ, α = ctx.ℓ, ctx.α
    s  = dist(x,y)/ℓ
    s2 = s^2
    z  = 1.0 + s2
    logz  = log(z)
    ϕ     = z^-α
    Hℓℓ   = 2*ϕ/z*α*s^2/ℓ*( 2*(α+1)/z*s^2/ℓ - 3/ℓ )
    Hℓα   = 2*ϕ/z*(1-α*logz) * s2/ℓ
    Hαα   = ϕ * logz^2
    H[1,1] += c*Hℓℓ
    H[1,2] += c*Hℓα
    H[2,1] += c*Hℓα
    H[2,2] += c*Hαα
    H
end
```

```
Hθ_kernel! (generic function with 4 methods)
```

Convenience functions

While it is a little more efficient to use a mutating function to compute kernel gradients and Hessians, it is also convenient to have a version available to allocate an outupt vector or matrix. We note that these convenience functions do not need to be specialized for the rational quadratic case.

```
gθ_kernel(ctx :: RBFKernelContext,
          x :: AbstractVector, y :: AbstractVector) =
              gθ_kernel!(zeros(nhypers(ctx)), ctx, x, y)
```

```
Hθ_kernel(ctx :: RBFKernelContext,
          x :: AbstractVector, y :: AbstractVector) =
              Hθ_kernel!(zeros(nhypers(ctx), nhypers(ctx)), ctx, x, y)

gx_kernel(ctx :: RBFKernelContext{d},
          x :: AbstractVector, y :: AbstractVector) where {d} =
              gx_kernel!(zeros(d), ctx, x, y)

Hx_kernel(ctx :: RBFKernelContext{d},
          x :: AbstractVector, y :: AbstractVector) where {d} =
              Hx_kernel!(zeros(d,d), ctx, x, y)
```

```
Hx_kernel (generic function with 1 method)
```

Testing

```
let
    x, y = [0.1; 0.2], [0.8; 0.8]
    ℓ = 0.2+rand()
    check_fd(gθ_kernel(KernelSE{2}(ℓ),x,y)[1],
             s→kernel(KernelSE{2}(ℓ+s),x,y), 0.0),
    check_fd(Hθ_kernel(KernelSE{2}(ℓ),x,y)[1,1],
             s→gθ_kernel(KernelSE{2}(ℓ+s),x,y)[1], 0.0)
end
```

```
(2.8878643367018276e-10, 8.838308084024689e-11)
```

```
let
    x, y = [0.1; 0.2], [0.8; 0.8]
    ℓ, α = 0.2+rand(), rand()
    k(ℓ,α) = KernelRQ{2}(ℓ,α)
    check_fd(gθ_kernel(k(ℓ,α),x,y)[1], s→kernel(k(ℓ+s,α),x,y), 0.0),
    check_fd(gθ_kernel(k(ℓ,α),x,y)[2], s→kernel(k(ℓ,α+s),x,y), 0.0),
    check_fd(Hθ_kernel(k(ℓ,α),x,y)[:,1], s→gθ_kernel(k(ℓ+s,α),x,y), 0.0),
    check_fd(Hθ_kernel(k(ℓ,α),x,y)[:,2], s→gθ_kernel(k(ℓ,α+s),x,y), 0.0)
end
```

```
(8.774194212527598e-11, 2.1483242039500427e-11, 4.8188745838275046e-11, 1.789525057716482e-
11)
```

```
let
    x, y, dx = [0.1; 0.2], [0.8; 0.8], rand(2)
    ctx = KernelM5{2}(0.5)
    check_fd(gx_kernel(ctx,x,y)'*dx, s→kernel(ctx,x+s*dx,y), 0.0),
    check_fd(Hx_kernel(ctx,x,y)*dx, s→gx_kernel(ctx,x+s*dx,y), 0.0)
end
```

```
(4.926374404343749e-11, 2.7534853846083826e-11)
```

Kernel matrices

Kernel evaluation organization

Now we would like code to compute kernel matrices K_{XY} (and vectors of kernel evaluations k_{Xz}). Julia allows us to do this concisely using *comprehensions*:

```
kernel0(k :: KernelContext, X :: AbstractMatrix, Y :: AbstractMatrix) =
    [k(x,y) for x in eachcol(X), y in eachcol(Y)]
kernel0(k :: KernelContext, X :: AbstractMatrix, z :: AbstractVector) =
    [k(x,z) for x in eachcol(X)]
```

```
kernel0 (generic function with 2 methods)
```

We use the *splatting* operation to pass the keyword arguments through to the invocation of the kernel functions. This allows us to put together a set of named hyperparameters.

There are two minor drawbacks to forming kernel matrices this way. First, the computation allocates a new output matrix or vector each time it is invoked; we would like to be able to write into existing storage, if storage has already been allocated. Second, we want to exploit symmetry when computing K_{XX}. Hence, we will put together a few helper functions for these tasks:

```
function kernel!(KXX :: AbstractMatrix, k :: KernelContext,
                 X :: AbstractMatrix)
    for j = 1:size(X,2)
        xj = @view X[:,j]
        KXX[j,j] = k(xj, xj)
        for i = 1:j-1
            xi = @view X[:,i]
            kij = k(xi, xj)
```

```
            KXX[i,j] = kij
            KXX[j,i] = kij
        end
    end
    KXX
end

function kernel!(KXz :: AbstractVector, k :: KernelContext,
                 X :: AbstractMatrix, z :: AbstractVector)
    for i = 1:size(X,2)
        xi = @view X[:,i]
        KXz[i] = k(xi, z)
    end
    KXz
end

function kernel!(KXY :: AbstractMatrix, k :: KernelContext,
                 X :: AbstractMatrix, Y :: AbstractMatrix)
    for j = 1:size(Y,2)
        yj = @view Y[:,j]
        for i = 1:size(X,2)
            xi = @view X[:,i]
            KXY[i,j] = k(xi, yj)
        end
    end
    KXY
end

kernel(k :: KernelContext, X :: AbstractMatrix) =
    kernel!(zeros(size(X,2), size(X,2)), k, X)

kernel(k :: KernelContext, X :: AbstractMatrix, z :: AbstractVector) =
    kernel!(zeros(size(X,2)), k, X, z)

kernel(k :: KernelContext, X :: AbstractMatrix, Y :: AbstractMatrix) =
    kernel!(zeros(size(X,2), size(Y,2)), k, X, Y)
```

```
kernel (generic function with 5 methods)
```

Sometimes, we want to incorporate a shift in the kernel matrix construction as well.

```
function kernel!(KXX :: AbstractMatrix, ctx :: KernelContext,
                 X :: AbstractMatrix, η :: Real)
    for j = 1:size(X,2)
        xj = @view X[:,j]
        KXX[j,j] = kernel(ctx, xj, xj) + η
        for i = 1:j-1
            xi = @view X[:,i]
            kij = kernel(ctx, xi, xj)
            KXX[i,j] = kij
            KXX[j,i] = kij
        end
    end
    KXX
end

kernel(ctx :: KernelContext, X :: AbstractMatrix, s :: Real) =
    kernel!(zeros(size(X,2), size(X,2)), ctx, X, s)
```

```
kernel (generic function with 5 methods)
```

It's always useful to sanity check that these computations are done correctly, or at least that there is internal consistency between the versions:

```
let
    Zk = kronecker_quasirand(2,10)
    k = KernelSE{2}(1.0)

    # Comprehension-based eval
    KXX1 = kernel0(k, Zk, Zk)
    KXz1 = kernel0(k, Zk, Zk[:,1])

    # Dispatch through call mechanism
    KXX2 = k(Zk)
    KXX3 = k(Zk, Zk)
    KXz2 = k(Zk, Zk[:,1])

    norm(KXX1-KXX2)/norm(KXX1),
    norm(KXX1-KXX3)/norm(KXX1),
    norm(KXz1-KXX1[:,1])/norm(KXX1[:,1]),
    norm(KXz2-KXX1[:,1])/norm(KXX1[:,1])
end
```

```
(0.0, 0.0, 0.0, 0.0)
```

We note that apart from allocations in the initial compilation, every version of the kernel matrix and vector evaluations does a minimal amount of memory allocation: two allocations for the versions that create outputs, zero allocations for the versions that are provided with storage.

```
let
    Zk = kronecker_quasirand(2,10)
    k = KernelSE{2}(1.0)
    Ktemp = zeros(10,10)
    Kvtemp = zeros(10)
    Zk1 = Zk[:,1]
    KXX1 = @time kernel0(k, Zk, Zk)
    KXz1 = @time kernel0(k, Zk, Zk1)
    KXX2 = @time kernel!(Ktemp, k, Zk)
    KXX3 = @time kernel!(Ktemp, k, Zk, Zk)
    KXz2 = @time kernel!(Kvtemp, k, Zk, Zk1)
    nothing
end
```

```
  0.000002 seconds (2 allocations: 944 bytes)
  0.000001 seconds (2 allocations: 144 bytes)
  0.000001 seconds
  0.000002 seconds
  0.000000 seconds
```

We will also later want the first derivatives with respect to hyperparameters packed into a set of n-by-n matrices:

```
function dθ_kernel!(δKs :: AbstractArray, ctx :: KernelContext,
                    X :: AbstractMatrix)
    n, n, d = size(δKs)
    for j = 1:n
        xj = @view X[:,j]
        δKjj = @view δKs[j,j,:]
        gθ_kernel!(δKjj, ctx, xj, xj)
        for i = j+1:n
            xi = @view X[:,i]
            δKij = @view δKs[i,j,:]
            δKji = @view δKs[j,i,:]
```

```
            gθ_kernel!(δKij, ctx, xi, xj)
            δKji[:] .= δKij
        end
    end
    δKs
end

dθ_kernel(ctx :: KernelContext, X :: AbstractMatrix) =
    dθ_kernel!(zeros(size(X,2), size(X,2), nhypers(ctx)), ctx, X)
```

```
dθ_kernel (generic function with 1 method)
```

Cholesky and whitening

In the GP setting, the kernel defines a covariance. We say f is distributed as a GP with mean $\mu(x)$ and covariance kernel $k(x, x')$ to mean that the random vector f_X with entries $f(x_i)$ is distributed as a multivariate normal with mean μ_X and covariance matrix K_{XX}. That is,

$$p(f_X = \mu_X + y) = \frac{1}{\sqrt{\det(2\pi K_{XX})}} \exp\left(-\frac{1}{2} y^T K_{XX}^{-1} y\right).$$

We can always subtract off the mean function in order to get a zero-mean random variable (and then add the mean back later if we wish). In the interest of keeping notation simple, we will do this for the moment.

Factoring the kernel matrix is useful for both theory and computation. For example, we note that if $K_{XX} = LL^T$ is a Cholesky factorization, then

$$p(f_X = y) \propto \exp\left(-\frac{1}{2}(L^{-1}y)^T (L^{-1}y)\right).$$

Hence, $Z = L^{-1} f_X$ is distributed as a *standard* normal random variable:

$$p(L^{-1} f_X = z) \propto \exp\left(-\frac{1}{2} z^T z\right).$$

That is, a triangular solve with L is what is known as a "whitening transformation," mapping a random vector with correlated entries to a random vector with independent standard normal entries. Conversely, if we want to sample from our distribution for f_X, we can compute the samples as $f_X = LZ$ where Z has i.i.d. standard normal entries.

Now suppose we partition $X = \begin{bmatrix} X_1 & X_2 \end{bmatrix}$ where data is known at the X_1 points and unknown at the X_2 points. We write the kernel matrix and its Cholesky factorization in block form as:

$$\begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{bmatrix},$$

and observe that

$$\begin{aligned} L_{11}L_{11}^T &= K_{11} \\ L_{21} &= K_{21}L_{11}^{-T} \\ L_{22}L_{22}^T &= S \coloneqq K_{22} - L_{21}L_{21}^T = K_{22} - K_{21}K_{11}^{-1}K_{12} \end{aligned}$$

Using the Cholesky factorization as a whitening transformation, we have

$$\begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix},$$

which we can solve by forward substitution to obtain

$$\begin{aligned} z_1 &= L_{11}^{-1}y_1 \\ z_2 &= L_{22}^{-1}(y_2 - L_{21}z_1). \end{aligned}$$

From here, we rewrite the prior distribution as

$$\begin{aligned} p(f_{X_1} = y_1, f_{X_2} = y_2) &\propto \exp\left(-\frac{1}{2}\left\|\begin{bmatrix} L_{11}^{-1}y_1 \\ L_{22}^{-1}(y_2 - L_{21}L_{11}^{-1}y_1) \end{bmatrix}\right\|^2\right) \\ &= \exp\left(-\frac{1}{2}y_1^T K_{11}^{-1}y_1 - \frac{1}{2}(y_2 - L_{21}z_1)^T S^{-1}(y_2 - L_{21}z_1)\right) \end{aligned}$$

Note that $L_{21}z_1 = L_{21}L_{11}^{-1}y_1 = K_{21}K_{11}^{-1}y_1$. Therefore, the posterior conditioned on $f_{X_1} = y_1$ is

$$\begin{aligned} p(f_{X_2} = y_2 | f_{X_1} = y_1) &= \frac{p(f_{X_1} = y_1, f_{X_2} = y_2)}{p(f_{X_1} = y_1)} \\ &\propto \exp\left(-\frac{1}{2}(y_2 - K_{21}K_{11}^{-1}y_1)^T S^{-1}(y_2 - K_{21}K_{11}^{-1}y_1)\right). \end{aligned}$$

That is, the Schur complement S serves as the posterior variance and $K_{21}K_{11}^{-1}y_1 = L_{21}L_{11}^{-1}y_1$ is the posterior mean. We usually write the posterior mean in terms of a weight vector c derived from interpolating the y_1 points:

$$\mathbb{E}[f_{X_2} | f_{X_1} = y_1] = K_{21}c, \quad K_{11}c = y_1.$$

Note that in the pointwise posterior case (i.e. where X_1 consists of only one point), the pointwise posterior standard deviation is

$$l_{22} = \sqrt{k_{22} - \|L^{-1}k_{12}\|^2}.$$

Cholesky in Julia

A `Cholesky` object in Julia represents a Cholesky factorization. The main operations it supports are extracting the triangular factor (via `U` or `L` for the upper or lower triangular versions) and solving a linear system with the full matrix. The `Cholesky` object itself is really a view on a matrix of storage that can be separately allocated. If we call `cholesky`, new storage is allocated to contain the factor, and the original matrix is left untouched. The mutating `cholesky!` overwrites the original storage.

We are frequently going to want the Cholesky factorization of the kernel matrix much more than the kernel matrix itself. We will therefore write some convenience functions for this.

```
kernel_cholesky!(KXX :: AbstractMatrix, ctx :: KernelContext,
                 X :: AbstractMatrix) =
    cholesky!(kernel!(KXX, ctx, X))

kernel_cholesky(ctx :: KernelContext, X :: AbstractMatrix) =
    cholesky!(kernel(ctx, X))

kernel_cholesky!(KXX :: AbstractMatrix, ctx :: KernelContext,
                 X :: AbstractMatrix, s :: Real) =
    cholesky!(kernel!(KXX, ctx, X, s))

kernel_cholesky(ctx :: KernelContext, X :: AbstractMatrix, s :: Real) =
    cholesky!(kernel(ctx, X, s))
```

```
kernel_cholesky (generic function with 2 methods)
```

It is also useful to have a convenience function for evaluating a function at a set of sample points in order to compute weights:

```
sample_eval(f, X :: AbstractMatrix) = [f(x) for x in eachcol(X)]
sample_eval2(f, X :: AbstractMatrix) = [f(x...) for x in eachcol(X)]
```

```
sample_eval2 (generic function with 1 method)
```

Because we will frequently run examples with a low-discrepancy sampling sequence, we put together something for that as well

```
function test_setup2d(f, n)
    Zk = kronecker_quasirand(2,n)
    y = sample_eval2(f, Zk)
    Zk, y
end

test_setup2d(f) = test_setup2d(f,10)
```

```
test_setup2d (generic function with 2 methods)
```

We are now set to evaluate the posterior mean and standard deviation for a GP at a new point. We will use one auxiliary vector here (this could be passed in if we wanted). Note that once we have computed the mean field, we no longer really need k_{Xz}, only the piece of the Cholesky factor ($r_{Xz} = L_{XX}^{-1} k_{Xz}$). Therefore we will use the overwriting version of the triangular solver.

```
function eval_GP(KC :: Cholesky, ctx :: KernelContext, X :: AbstractMatrix,
                 c :: AbstractVector, z :: AbstractVector)
    kXz = kernel(ctx, X, z)
    μz  = dot(kXz, c)
    rXz = ldiv!(KC.L, kXz)
    σz  = sqrt(kernel(ctx,z,z)-rXz'*rXz)
    μz, σz
end
```

```
eval_GP (generic function with 1 method)
```

As usual, a demonstration and test case is a good idea.

```
let
    # Set up sample points and test function
    testf(x,y) = x^2 + y
    Zk, y = test_setup2d(testf)
    ctx = KernelSE{2}(1.0)

    # Form kernel Cholesky and weights
    KC = kernel_cholesky(ctx, Zk)
    c = KC\y

    # Evaluate true function and GP at a test point
```

```
    z = [0.456; 0.456]
    fz = testf(z ... )
    μz, σz = eval_GP(KC, ctx, Zk, c, z)

    # Compare GP to true function
    zscore = (fz-μz)/σz
    println("""
        True value:       $fz
        Posterior mean:   $μz
        Posterior stddev: $σz
        z-score:          $zscore
        """)
end
```

```
True value:       0.6639360000000001
Posterior mean:   0.6738680868304441
Posterior stddev: 0.008980490037452743
z-score:          -1.105962680101271
```

Extending Cholesky

It is sometimes useful to be able to extend an existing Cholesky factorization stored in a submatrix without allocating any new storage; Julia makes this fairly easy. The two things worth noting are that

- The default in Julia is that the Cholesky factor is stored in the upper triangle.
- The BLAS symmetric rank-k update routine (`syrk`) is in-place and is generally optimized better than would be `K22 -= R12'*R12`.

```
function extend_cholesky!(Kstorage :: AbstractMatrix, n, m)

    # Construct views for active part of the storage
    R   = @view Kstorage[1:m, 1:m]
    R11 = @view Kstorage[1:n, 1:n]
    K12 = @view Kstorage[1:n, n+1:m]
    K22 = @view Kstorage[n+1:m, n+1:m]

    ldiv!(UpperTriangular(R11)', K12)         # R12 = R11'\K12
    BLAS.syrk!('U', 'T', -1.0, K12, 1.0, K22) # S = K22-R12'*R12
    cholesky!(Symmetric(K22))                 # R22 = chol(S)
```

```
    # Return extended cholesky
    Cholesky(UpperTriangular(R))
end
```

```
extend_cholesky! (generic function with 1 method)
```

As an example of the extension, let's consider computing the predictive standard deviation at a new point given previous points.

```
let
    Zk = kronecker_quasirand(2,10)
    ctx = KernelSE{2}(1.0)
    z = [0.456; 0.456]
    w = randn(10)

    # Pre-allocate storage, form K, and Cholesky factor in place.
    Kstorage = zeros(11,11)
    K11 = @view Kstorage[1:10,1:10]
    KC = kernel_cholesky!(K11, ctx, Zk)

    # Add a column of K to the matrix
    K12 = @view Kstorage[1:10,11]
    kernel!(K12, ctx, Zk, z)
    Kstorage[11,11] = kernel(ctx, z, z)

    # Extend the Cholesky factorization
    Kfac = extend_cholesky!(Kstorage, 10, 11)

    # Compare to the earlier approach
    c = zeros(10)
    μz, σz = eval_GP(KC, ctx, Zk, c, z)

    # Relative error in computed stddev
    abs( (σz-Kstorage[11,11])/σz )
end
```

```
0.0
```

The nugget

We often add a small "noise variance" term (also called a "nugget term") to the kernel matrix. The name "noise variance" comes from the probabilistic interpretation that the observed function values are contaminated by some amount of mean zero Gaussian noise (generally assumed to be iid). The phrase "nugget" comes from the historical use of Gaussian processing in geospatial statistics for predicting where to find mineral deposits – noise in that case corresponding to nuggets of minerals that might show up in a particular sample. Either, we solve the kernel system

$$\tilde{K}c = y, \quad \tilde{K} = K + \eta I.$$

The numerical reason for including this term is because kernel matrices tend to become ill-conditioned as we add observations – and the smoother the kernel, the more ill-conditioned the problem. This has an immediate downstream impact on the numerical stability of essentially all the remaining tasks for the problem. There are various clever ways that people consider to side-step this ill conditioning, but for the moment we will stick with a noise variance term.

What is an appropriate vaule for η? If the kernel family is appropriate for the smoothness of the function being modeled, then it may be sensible to choose η to be as small as we can manage without running into numerical difficulties. A reasonable rule of thumb is to choose η around $\sqrt{\epsilon_{\text{mach}}}$ (i.e. about 10^{-8} in double precision). This will be our default behavior.

On the other hand, if the function is close to something smooth but has some non-smooth behavior (or high-frequency oscillations), then it may be appropriate to try to model the non-smooth or high-frequency piece as noise, and use a larger value of η. There is usually a stationary point for the likelihood function that corresponds to the modeling assumption that the observations are almost all noise; we would prefer to avoid that, so we also want η to not be too big. Hence, if the noise variance is treated as a tunable hyperparameter, we usually work with $\log \eta$ rather than with η, and tune subject to upper and lower bounds on $\log \eta$.

GPP context

As with kernels, it is helpful to define a context object that includes the fundamental data needed for working with a GP posterior (GPP). This includes the kernel information, data points, kernel matrix, function values, weights, and some scratch space useful for computing the posterior variance and its derivatives. We will typically allocate some extra space so that we can change the number of points without reallocating all our storage.

```
struct GPPContext{T <: KernelContext}
    ctx :: T
    η :: Float64
    Xstore  :: Matrix{Float64}
```

```
    Kstore  :: Matrix{Float64}
    cstore  :: Vector{Float64}
    ystore  :: Vector{Float64}
    scratch :: Matrix{Float64}
    n :: Integer
end

getX(gp :: GPPContext) = view(gp.Xstore,:,1:gp.n)
getc(gp :: GPPContext) = view(gp.cstore,1:gp.n)
gety(gp :: GPPContext) = view(gp.ystore,1:gp.n)
getK(gp :: GPPContext) = view(gp.Kstore,1:gp.n,1:gp.n)
getKC(gp :: GPPContext) = Cholesky(UpperTriangular(getK(gp)))
capacity(gp :: GPPContext) = length(gp.ystore)
getXrest(gp :: GPPContext) = view(gp.Xstore,:,gp.n+1:capacity(gp))
getyrest(gp :: GPPContext) = view(gp.ystore,gp.n+1:capacity(gp))
getXrest(gp :: GPPContext, m) = view(gp.Xstore,:,gp.n+1:gp.n+m)
getyrest(gp :: GPPContext, m) = view(gp.ystore,gp.n+1:gp.n+m)

function GPPContext(ctx :: KernelContext, η :: Float64, capacity)
    d = ndims(ctx)
    Xstore  = zeros(d, capacity)
    Kstore  = zeros(capacity, capacity)
    cstore  = zeros(capacity)
    ystore  = zeros(capacity)
    scratch = zeros(capacity,max(d+1,3))
    GPPContext(ctx, η, Xstore, Kstore, cstore, ystore, scratch, 0)
end
```

```
GPPContext
```

For internal use, we want to be able to regularly refactor the current kernel matrix and resolve the coefficient problem.

```
refactor!(gp :: GPPContext) = kernel_cholesky!(getK(gp), gp.ctx, getX(gp), gp.η)
resolve!(gp :: GPPContext) = ldiv!(getKC(gp), copyto!(getc(gp), gety(gp)))
```

```
resolve! (generic function with 1 method)
```

The basic operations we need are to add or remove data points, evaluate the predictive mean and variance (and gradients), and update the kernel hyperparameters.

Adding points

We start with adding data points, which is maybe the most complicated operation (since it involves extending a Cholesky factorization).

```
function add_points!(gp :: GPPContext, m)
    n = gp.n + m
    if gp.n > capacity(gp)
        error("Added points exceed GPPContext capacity")
    end

    # Create new object (same storage)
    gpnew = GPPContext(gp.ctx, gp.η, gp.Xstore, gp.Kstore,
                       gp.cstore, gp.ystore, gp.scratch, n)

    # Refactor (if start from 0) or extend Cholesky (if partly done)
    if gp.n == 0
        refactor!(gpnew)
    else
        X1, X2 = getX(gp), getXrest(gp,m)
        R11 = getK(gp)
        K12 = view(gp.Kstore,1:gp.n,gp.n+1:n)
        K22 = view(gp.Kstore,gp.n+1:n,gp.n+1:n)
        kernel!(K12, gp.ctx, X1, X2)
        kernel!(K22, gp.ctx, X2, gp.η)
        ldiv!(UpperTriangular(R11)', K12)         # R12 = R11'\K12
        BLAS.syrk!('U', 'T', -1.0, K12, 1.0, K22) # S = K22-R12'*R12
        cholesky!(Symmetric(K22))                 # R22 = chol(S)
    end

    # Update c
    resolve!(gpnew)

    gpnew
end

function add_points!(gp :: GPPContext,
                     X :: AbstractMatrix, y :: AbstractVector)
    m = length(y)
    if size(X,2) ≠ m
        error("Inconsistent number of points and number of values")
    end
    copy!(getXrest(gp,m), X)
```

```
    copy!(getyrest(gp,m), y)
    add_points!(gp, m)
end

function add_point!(gp :: GPPContext, x :: AbstractVector, y :: Float64)
    add_points!(gp, reshape(x, length(x), 1), [y])
end

function GPPContext(ctx :: KernelContext, η :: Float64,
                    X :: Matrix{Float64}, y :: Vector{Float64})
    d, n = size(X)
    if d ≠ ndims(ctx)
        error("Mismatch in dimensions of X and kernel")
    end
    gp = GPPContext(ctx, η, n)
    copy!(gp.Xstore, X)
    copy!(gp.ystore, y)
    add_points!(gp, n)
end
```

```
GPPContext
```

Removing points

Removing points is rather simpler.

```
function remove_points!(gp :: GPPContext, m)
    if m > gp.n
        error("Cannot remove $m > $(gp.n) points")
    end
    gpnew = GPPContext(gp.ctx, gp.η, gp.Xstore, gp.Kstore,
                       gp.cstore, gp.ystore, gp.scratch, gp.n-m)
    resolve!(gpnew)
    gpnew
end
```

```
remove_points! (generic function with 1 method)
```

Changing kernels

Changing the kernel is also simple, though it involves a complete refactorization.

```
function change_kernel_nofactor!(gp :: GPPContext, ctx :: KernelContext, η :: Float64)
    GPPContext(ctx, η, gp.Xstore, gp.Kstore,
               gp.cstore, gp.ystore, gp.scratch, gp.n)
end

function change_kernel!(gp :: GPPContext, ctx :: KernelContext, η :: Float64)
    gpnew = change_kernel_nofactor!(gp, ctx, η)
    refactor!(gpnew)
    resolve!(gpnew)
    gpnew
end
```

```
change_kernel! (generic function with 1 method)
```

Predictive mean

And now we compute the predictive mean and its derivatives.

```
function mean(gp :: GPPContext, z :: AbstractVector)
    ctx, X, c = gp.ctx, getX(gp), getc(gp)
    d, n = size(X)
    sz = 0.0
    for j = 1:n
        xj = @view X[:,j]
        sz += c[j]*kernel(ctx, z, xj)
    end
    sz
end

function gx_mean!(gsz :: AbstractVector, gp :: GPPContext, z :: AbstractVector)
    ctx, X, c = gp.ctx, getX(gp), getc(gp)
    d, n = size(X)
    for j = 1:n
        xj = @view X[:,j]
        gx_kernel!(gsz, ctx, z, xj, c[j])
    end
    gsz
```

```
end

function gx_mean(gp :: GPPContext, z :: AbstractVector)
    d = ndims(gp.ctx)
    gx_mean!(zeros(d), gp, z)
end

function Hx_mean!(Hsz :: AbstractMatrix, gp :: GPPContext, z :: AbstractVector)
    ctx, X, c = gp.ctx, getX(gp), getc(gp)
    d, n = size(X)
    for j = 1:n
        xj = @view X[:,j]
        Hx_kernel!(Hsz, ctx, z, xj, c[j])
    end
    Hsz
end

function Hx_mean(gp :: GPPContext, z :: AbstractVector)
    d = ndims(gp.ctx)
    Hx_mean!(zeros(d,d), gp, z)
end
```

```
Hx_mean (generic function with 1 method)
```

Straightforward or not, we will stick with the habit of including a finite difference check.

Predictive variance

The predictive variance is a little less straighforward than the predictive mean, but only a little. The formula is

$$v(z) = k(z,z) - k_{zX} K_{XX}^{-1} k_{Xz},$$

where we will assume $k(z,z)$ is constant. Differentiating once with respect to z gives

$$\nabla v(x) = -2 \nabla k_{zX} (K_{XX}^{-1} k_{Xz}),$$

and differentiating a second time gives

$$Hv(x) = -2 \sum_j Hk(z, x_j)(K_{XX}^{-1} k_{Xz})_j - 2(\nabla k_{zX}) K_{XX}^{-1} (\nabla k_{zX})^T.$$

```
function var(gp :: GPPContext, z :: AbstractVector)
    kXz = view(gp.scratch,1:gp.n,1)
    kernel!(kXz, gp.ctx, getX(gp), z)
    L = getKC(gp).L
    v = ldiv!(L, kXz)
    kernel(gp.ctx,z,z) - v'*v
end

function gx_var!(g :: AbstractVector, gp :: GPPContext, z :: AbstractVector)
    X, KC, ctx = getX(gp), getKC(gp), gp.ctx
    d, n = size(X)
    kXz  = view(gp.scratch,1:n,1)
    gkXz = view(gp.scratch,1:n,2:d+1)
    gkXz[:] .= 0.0
    for j = 1:n
        xj = @view X[:,j]
        kXz[j] = kernel(ctx, z, xj)
        gx_kernel!(view(gkXz,j,:), ctx, z, xj)
    end
    w = ldiv!(KC,kXz)
    mul!(g, gkXz', w, -2.0, 0.0)
end

function gx_var(gp :: GPPContext, z :: AbstractVector)
    d = ndims(gp.ctx)
    gx_var!(zeros(d), gp, z)
end

function Hx_var!(H :: AbstractMatrix, gp :: GPPContext, z :: AbstractVector)
    X, KC, ctx = getX(gp), getKC(gp), gp.ctx
    d, n = size(X)
    kXz  = view(gp.scratch,1:n,1)
    gkXz = view(gp.scratch,1:n,2:d+1)
    gkXz[:] .= 0.0
    for j = 1:n
        xj = @view X[:,j]
        kXz[j] = kernel(ctx, z, xj)
        gx_kernel!(view(gkXz,j,:), ctx, z, xj)
    end
    w = ldiv!(KC,kXz)
    invL_gkXz = ldiv!(KC.L, gkXz)
    H[:] .= 0.0
```

```
    for j = 1:n
        xj = @view X[:,j]
        Hx_kernel!(H, ctx, z, xj, w[j])
    end
    mul!(H, invL_gkXz', invL_gkXz, -2.0, -2.0)
end

function Hx_var(gp :: GPPContext, z :: AbstractVector)
    d = ndims(gp.ctx)
    Hx_var!(zeros(d,d), gp, z)
end
```

```
Hx_var (generic function with 1 method)
```

Testing

And, as usual, we have some tests. First, we check adding and removing points and changing the kernel.

```
let
    testf(x,y) = x^2+y
    Zk, y = test_setup2d(testf)
    ctx = KernelSE{2}(1.0)
    ctx2 = KernelSE{2}(0.8)

    gp1 = GPPContext(ctx, 0.0, Zk, y)
    gpt = GPPContext(ctx, 0.0, 10)
    gpt = add_points!(gpt, Zk, y)
    check_approx(getc(gpt), getc(gp1))

    gp2 = GPPContext(ctx2, 0.0, Zk, y)
    gpt = change_kernel!(gpt, ctx2, 0.0)
    check_approx(getc(gpt), getc(gp2))
    gpt = change_kernel!(gpt, ctx, 0.0)

    gpt = remove_points!(gpt, 2)
    gp3 = GPPContext(ctx, 0.0, Zk[:,1:end-2], y[1:end-2])
    check_approx(getc(gpt), getc(gp3))

    gpt = add_points!(gpt, Zk[:,end-1:end], y[end-1:end])
```

```
    check_approx(getc(gpt), getc(gp1))
end
```

```
8.170362242404771e-13
```

And now we check consistency of the predictive mean and variance and their derivatives.

```
let
    Zk, y = test_setup2d((x,y) → x^2 + cos(3*y))
    z = [0.47; 0.47]
    dz = randn(2)

    gp = GPPContext(KernelSE{2}(0.5), 1e-8, Zk, y)
    max(check_fd(gx_mean(gp, z)'*dz, s→mean(gp, z+s*dz), 0.0),
        check_fd(Hx_mean(gp, z)*dz, s→gx_mean(gp, z+s*dz), 0.0),
        check_fd(gx_var(gp, z)'*dz, s→var(gp, z+s*dz), 0.0),
        check_fd(Hx_var(gp, z)*dz, s→gx_var(gp, z+s*dz), 0.0))
end
```

```
1.9577095443284557e-9
```

Now we check that the predictive mean and variance agree with our earlier calculations without the GPP context object.

```
let
    testf(x,y) = x^2+y
    Zk, y = test_setup2d(testf)
    ctx = KernelSE{2}(1.0)
    gp = GPPContext(ctx, 0.0, Zk, y)

    z = [0.456; 0.456]
    fz = testf(z ... )
    μz, σz = mean(gp, z), sqrt(var(gp,z))
    zscore = (fz-μz)/σz
    println("""
        True value:       $fz
        Posterior mean:   $μz
        Posterior stddev: $σz
        z-score:          $zscore
        """)
end
```

```
True value:       0.6639360000000001
Posterior mean:   0.6738680868304527
Posterior stddev: 0.008980490037452743
z-score:          -1.105962680102235
```

Hyperparameter tuning

Kernel hyperparameters are things like the diagonal variance, length scale, and noise variance. When we don't have a good prior guess for their values (which is usually the case), we need to find some way to choose them automatically. We generally denote the vector of hyperparameters as θ. In the interest of minimizing visual clutter, we will mostly write the kernel matrix as K (rather than K_{XX}) in this section.

A standard approach is to compute the maximum (marginal) likelihood estimator; equivalently, we minimize the negative log likelihood (NLL)

$$\phi(\theta) = \frac{1}{2}\log\det K + \frac{1}{2}y^T K^{-1} y + \frac{n}{2}\log(2\pi).$$

The first term (the log determinant) penalizes model complexity; the second term captures data fidelity; and the third term is simply a dimension-dependent normalizing constant. Of these, the first term is generally the most tricky to work with numerically.

As a starting point, we note that after computing the Cholesky factorization of K and solving $c = K^{-1}y$, evaluating the NLL can be done in $O(n)$ time, using the fact that

$$\frac{1}{2}\log\det K = \log\det R = \sum_j \log r_{jj}.$$

```
function nll(KC :: Cholesky, c :: AbstractVector, y :: AbstractVector)
    n = length(c)
    ϕ = (dot(c,y) + n*log(2π))/2
    for k = 1:n
        ϕ += log(KC.U[k,k])
    end
    ϕ
end

nll(gp :: GPPContext) = nll(getKC(gp), getc(gp), gety(gp))
```

```
nll (generic function with 2 methods)
```

Differentiating the NLL

We first briefly recall how variational notation works. For a given function f, the symbol δf (read "variation of f") represents a generic directional derivative with respect to some underlying parameter. If f depends on x, for example, we would write $\delta f = f'(x)\,\delta x$. For second variations, we would usually use Δ, e.g.

$$\Delta\delta f = f''(x)\,\delta x\,\Delta x + f'(x)\Delta\delta x$$

The advantage of this notation is that it sweeps under the rug some of the book-keeping of tracking what parameter we differentiate with respect to.

Differentiating through the inverse

To differentiate the NLL, we need to be able to differentiate inverses and log-determinants. We begin with inverses. Applying implicit differentiation to the equation $A^{-1}A = I$ gives us

$$\delta[A^{-1}]\,A + A^{-1}\,\delta A = 0,$$

which we can rearrange to

$$\delta[A^{-1}] = -A^{-1}(\delta A)A^{-1}.$$

The second derivative is

$$\Delta\delta[A^{-1}] = A^{-1}(\Delta A)A^{-1}(\delta A)A^{-1} + A^{-1}(\delta A)A^{-1}(\Delta A)A^{-1} - A^{-1}(\Delta\delta A)A^{-1}$$

It is a useful habit to check derivative computations with finite differences, and we will follow that habit here.

```
let
    A0, δA, ΔA, ΔδA = randn(10,10), rand(10,10), rand(10,10), rand(10,10)

    δinv(A,δA) = -A\δA/A
    invAδA, invAΔA, invAΔδA = A0\δA, A0\ΔA, A0\ΔδA
    ΔδinvA = (invAδA*invAΔA + invAΔA*invAδA - invAΔδA)/A0

    check_fd(δinv(A0,δA), s→inv(A0+s*δA), 0),
    check_fd(ΔδinvA, s→δinv(A0+s*ΔA, δA+s*ΔδA), 0)
end
```

```
(6.169202946187787e-10, 8.094958949879665e-11)
```

Differentiating the log determinant

For the case of the log determinant, it is helpful to decompose a generic square matrix F as

$$F = L + D + U$$

where L, D, and U are the strictly lower triangular, diagonal, and strictly upper triangular parts of F, respectively. Then note that

$$(I + \epsilon F) = (I + \epsilon L)(I + \epsilon(D + U)) + O(\epsilon^2),$$

and therefore

$$\begin{aligned}
\det(I + \epsilon F) &= \det(I + \epsilon(D + U)) + O(\epsilon^2) \\
&= \prod_i (1 + \epsilon d_i) + O(\epsilon^2) \\
&= 1 + \epsilon \sum_i d_i + O(\epsilon^2) \\
&= 1 + \epsilon \operatorname{tr}(F) + O(\epsilon^2).
\end{aligned}$$

Hence the derivative of $\det(A)$ about $A = I$ is $\operatorname{tr}(A)$.

Now consider

$$\det(A + \epsilon(\delta A)) = \det(A) \det(I + \epsilon A^{-1} \delta A) = \det(A) + \epsilon \det(A) \operatorname{tr}(A^{-1} \delta A) + O(\epsilon^2).$$

This gives us that in general,

$$\delta[\det(A)] = \det(A) \operatorname{tr}(A^{-1} \delta A),$$

and hence

$$\delta[\log \det(A)] = \frac{\delta[\det(A)]}{\det(A)} = \operatorname{tr}(A^{-1} \delta A).$$

We can also write this as

$$\delta[\log \det(A)] = \langle A^{-T}, \delta A \rangle_F,$$

i.e. A^{-T} is the gradient of $\log \det(A)$.

The second derivative is

$$\Delta\delta[\log \det(A)] = \operatorname{tr}(A^{-1} \Delta \delta A) - \operatorname{tr}(A^{-1} \Delta A \, A^{-1} \delta A).$$

Again, a finite difference check is a useful thing. We do need to be a little careful here in order to make sure that the log determinant is well defined at A and in a near neighborhood.

```
let
    V = randn(10,10)
    A = V*Diagonal(1.0.+rand(10))/V
    δA, ΔA, ΔδA = randn(10,10), randn(10,10), randn(10,10)

    δlogdet(A, δA) = tr(A\δA)
    Δδlogdet(A, δA, ΔA, ΔδA) = tr(A\ΔδA)-tr((A\ΔA)*(A\δA))

    check_approx(δlogdet(A, δA), dot(inv(A'), δA)),
    check_fd(δlogdet(A,δA), s→log(det(A+s*δA)), 0),
    check_fd(Δδlogdet(A,δA,ΔA,ΔδA), s→δlogdet(A+s*ΔA,δA+s*ΔδA), 0)
end
```

```
(1.0925968186896108e-14, 3.1199710890195106e-9, 7.119051413015065e-11)
```

Putting it together

Putting together the results of the previous section, we have

$$\delta\phi = \frac{1}{2}\operatorname{tr}(K^{-1}\,\delta K) - \frac{1}{2}c^T(\delta K)c$$

where $c = K^{-1}y$. For the second derivative, we have

$$\begin{aligned}\Delta\delta\phi =& \frac{1}{2}\operatorname{tr}(K^{-1}\Delta\delta K) - \frac{1}{2}\operatorname{tr}(K^{-1}\Delta K\, K^{-1}\delta K) \\ & - \frac{1}{2}c^T(\Delta\delta K)c + c^T(\Delta K)K^{-1}(\delta K)c.\end{aligned}$$

Now consider the case of n data points and d tunable hyperparameters. In general, we can assume that n is significantly larger than d; if this is not the case, we probably need more data or fewer hyperparameters! Cholesky factorization of the kernel matrix in order to compute a mean field or the negative log likelihood takes $O(n^3)$ time. How long does it take to compute gradients and Hessians with respect to the hyperparameters?

Just computing the matrix of derivatives of the kernel with respect to a hyperparameter will generally take $O(n^2)$ time; so, with a few exceptions, we do not expect to be able to compute any derivative term in less than $O(n^2)$. But how much more than $O(n^2)$ time might we need?

Fast gradients

The tricky piece in computing gradients is the derivative of the log determinant term. If we are willing to form K^{-1} explicitly, we can write

$$\delta\phi = \left\langle \delta K, \frac{1}{2}\left(K^{-1} - cc^T\right)\right\rangle_F.$$

Computing this way costs an additional fixed $O(n^3)$ cost to form K^{-1} explicitly, followed by $O(n^2)$ time to compute each of the derivatives, for an overall cost of $O(n^3 + dn^2)$. The danger here is that K^{-1} will generally have some very large entries with alternating sign, and so the inner product here is somewhat numerically sensitive. Of course, this is associated with ill-conditioning of K, which can be a problem for other methods of computation as well.

If we compute the gradient this way, we do not ever need to materialize the full δK matrices. We can also take advantage of symmetry.

```
function gθ_nll!(g :: AbstractVector, gp :: GPPContext,
                 invK :: AbstractMatrix)
    ctx, X, c = gp.ctx, getX(gp), getc(gp)
    d, n = size(X)
    for j = 1:n
        xj = @view X[:,j]
        cj = c[j]
        gθ_kernel!(g, ctx, xj, xj, (invK[j,j]-cj*cj)/2)
        for i = j+1:n
            xi = @view X[:,i]
            ci = c[i]
            gθ_kernel!(g, ctx, xi, xj, (invK[i,j]-ci*cj))
        end
    end
    g
end

gθ_nll(gp :: GPPContext, invK) =
    let nθ = nhypers(gp.ctx)
        gθ_nll!(zeros(nθ), gp, invK)
    end
```

```
gθ_nll (generic function with 1 method)
```

We treat the hyperparameter associated with the noise variance as a special case. If $z = \log\eta$ is the log-scale version of the noise variance, we also want to compute the extended gradient with the derivative with respect to z at the end.

```
function gθz_nll!(g, gp :: GPPContext, invK)
    ctx, X, c, s = gp.ctx, getX(gp), getc(gp), gp.η
    gθ_nll!(g, gp, invK)
    g[nhypers(ctx)+1] = (tr(invK) - c'*c)*s/2
    g
end

gθz_nll(gp :: GPPContext, invK) =
    gθz_nll!(zeros(nhypers(gp.ctx)+1), gp, invK)

gθz_nll(gp :: GPPContext) = gθz_nll(gp, getKC(gp)\I)
```

```
gθz_nll (generic function with 2 methods)
```

Per usual, we also do a finite difference check.

```
let
    Zk, y = test_setup2d((x,y) → x^2 + y)
    s, ℓ = 1e-4, 1.0
    z=log(s)

    gp_SE_nll(ℓ,z) = nll(GPPContext(KernelSE{2}(ℓ), exp(z), Zk, y))
    g = gθz_nll(GPPContext(KernelSE{2}(ℓ), s, Zk, y))

    check_fd(g[1], ℓ→gp_SE_nll(ℓ,z), ℓ),
    check_fd(g[2], z→gp_SE_nll(ℓ,z), z)
end
```

```
(1.9651700832927413e-8, 4.860775501706146e-9)
```

Fast Hessians

If we want to compute with Newton methods, it is useful to rewrite these expressions in a more symmetric way. Let $K = LL^T$ be the (lower triangular) Cholesky factorization of K, and define

$$\tilde{c} = L^{-1}y$$
$$\delta\tilde{K} = L^{-1}(\delta K)L^{-T}$$
$$\Delta\tilde{K} = L^{-1}(\Delta K)L^{-T}$$

This is something we want to do systematically across hyperparameters.

```
function whiten_matrix!(δK, KC :: Cholesky)
    ldiv!(KC.L, δK)
    rdiv!(δK, KC.U)
    δK
end

function whiten_matrices!(δKs, KC :: Cholesky)
    n, n, d = size(δKs)
    for k=1:d
        whiten_matrix!(view(δKs,:,:,k), KC)
    end
    δKs
end
```

```
whiten_matrices! (generic function with 1 method)
```

Then we can rewrite the gradient and Hessian as

$$\delta\phi = \frac{1}{2}\operatorname{tr}(\delta\tilde{K}) - \frac{1}{2}\tilde{c}^T(\delta\tilde{K})\tilde{c}$$
$$\Delta\delta\phi = \left\langle \frac{1}{2}\left(K^{-1} - cc^T\right), \Delta\delta K \right\rangle_F - \frac{1}{2}\langle \Delta\tilde{K}, \delta\tilde{K}\rangle_F + \langle \Delta\tilde{K}\tilde{c}, \delta\tilde{K}\tilde{c}\rangle$$

The cost here is an initial factorization and computation of K^{-1}, an $O(n^3)$ factorization for computing each whitened perturbation ($\delta\tilde{K}$ or $\Delta\tilde{K}$) and then $O(n^2)$ for each derivative component (gradient or Hessian) for a total cost of $O((d+1)n^3 + d^2n^2)$.

We would like to make a special case for the noise variance term. However, after several attempts, I still do not have a method that avoids forming K^{-2} or $L^{-1}L^{-T}$, either of which requires $O(n^3)$ time. Hence, our current code treats $z = \log\eta$ like the other hyperparameters, save that the kernel functions are not called to compute the derivative.

```
function mul_slices!(result, As, b)
    m, n, k = size(As)
    for j=1:k
        mul!(view(result,:,j), view(As,:,:,j), b)
    end
    result
end

function Hθ_nll(gp :: GPPContext)
    ctx, X, y, c, s = gp.ctx, getX(gp), gety(gp), getc(gp), gp.η
    d, n = size(X)
```

```
    nθ = nhypers(ctx)

    # Factorization and initial solves
    KC = getKC(gp)
    invK = KC\I
    c̃ = KC.L\y
    ϕ = nll(gp)
    ∂z_nll = (tr(invK)-(c'*c))*s/2

    # Set up space for NLL, gradient, and Hessian (including wrt z)
    g = zeros(nθ+1)
    H = zeros(nθ+1,nθ+1)

    # Add Hessian contribution from kernel second derivatives
    d, n = size(X)
    for j = 1:n
        xj = @view X[:,j]
        cj = c[j]
        Hθ_kernel!(H, ctx, xj, xj, (invK[j,j]-cj*cj)/2)
        for i = j+1:n
            xi = @view X[:,i]
            ci = c[i]
            Hθ_kernel!(H, ctx, xi, xj, (invK[i,j]-ci*cj))
        end
    end
    H[nθ+1,nθ+1] = ∂z_nll

    # Set up whitened matrices δK̃ and products δK*c and δK̃*c̃
    δKs = zeros(n, n, nθ+1)
    dθ_kernel!(δKs, ctx, X)
    for j=1:n  δKs[j,j,nθ+1] = s  end
    δK̃s = whiten_matrices!(δKs, KC)
    δK̃c̃s = mul_slices!(zeros(n,nθ+1), δK̃s, c̃)
    δK̃r = reshape(δK̃s, n*n, nθ+1)

    # Add Hessian contributions involving whitened matrices
    mul!(H, δK̃r', δK̃r, -0.5, 1.0)
    mul!(H, δK̃c̃s', δK̃c̃s, 1.0, 1.0)

    # And put together gradient gradient
    for j=1:nθ
        g[j] = tr(view(δK̃s,:,:,j))/2
```

```
    end
    mul!(g, δK̃c̃s', c̃, -0.5, 1.0)
    g[end] = ∂z_nll

    ϕ, g, H
end
```

```
Hθ_nll (generic function with 1 method)
```

As usual, we sanity check on a simple case.

```
let
    Zk, y = test_setup2d((x,y) → x^2 + y)
    s, ℓ = 1e-3, 0.89
    z = log(s)

    testf(ℓ,z) = Hθ_nll(GPPContext(KernelSE{2}(ℓ), exp(z), Zk, y))
    ϕref, gref, Href = testf(ℓ, z)

    max(check_fd(gref[1], ℓ→testf(ℓ,z)[1][1], ℓ),
        check_fd(gref[2], z→testf(ℓ,z)[1][1], z),
        check_fd(Href[1,1], ℓ→testf(ℓ,z)[2][1], ℓ),
        check_fd(Href[1,2], ℓ→testf(ℓ,z)[2][2], ℓ),
        check_fd(Href[2,2], z→testf(ℓ,z)[2][2], z),
        check_approx(Href[1,2], Href[2,1]))
end
```

```
3.4786567942664956e-8
```

Fast approximate Hessians

As a final note, we can *estimate* second derivatives using stochastic trace estimation, i.e.

$$\mathrm{tr}(A) = \mathbb{E}[Z^T A Z]$$

where Z is a probe vector with independent entries of mean zero and variance 1. Taking

$W = L^{-T}Z$ gives us the *approximate* first and second derivatives

$$\begin{aligned}
\delta\phi =& \frac{1}{2}\operatorname{tr}(L^{-1}(\delta K)L^{-T}) - \frac{1}{2}c^T(\delta K)c \\
\approx& \frac{1}{2}W^T(\delta K)W - \frac{1}{2}c^T(\delta K)c \\
\Delta\delta\phi =& \frac{1}{2}\operatorname{tr}(L^{-1}(\Delta\delta K)L^{-T}) - \frac{1}{2}\operatorname{tr}(L^{-1}(\Delta K)L^{-T}L^{-1}(\delta K)L^{-T}) \\
& - \frac{1}{2}c^T(\Delta\delta K)c + c^T(\Delta K)L^{-T}L^{-1}(\delta K)c \\
\approx& \frac{1}{2}W^T(\Delta\delta K)W - \frac{1}{2}\langle L^{-1}(\Delta K)W, L^{-1}(\delta K)W\rangle_F \\
& - \frac{1}{2}c^T(\Delta\delta K)c + \langle L^{-1}(\Delta K)c, L^{-1}(\delta K)c\rangle.
\end{aligned}$$

The approximate gradient and approximate Hessian are consistent with each other, which may be helpful if we want to do Newton on an approximate objective rather than approximate Newton on the true NLL. Or we can get an approximate Hessian and an exact gradient at the same cost of $O(n^3 + d^2n^2)$.

Scale factors

We will frequently write our kernel matrix as $K = C\bar{K}$, where $\bar{K}$ is a reference kernel matrix and C is a scaling factor. If $\bar{K}$ has diagonal equal to one, we can interpret it as a correlation matrix; however, this is not necessary. The scaling factor C is a hyperparameter, but it is simple enough that it deserves special treatment. We therefore will separate out the the hyperparameters into the scaling (C) and the rest of the hypers (θ'). The key observation is that given the other hyperparameters, we can easily compute the optimum value $C_{\text{opt}}(\theta')$ for the scaling. This lets us work with a reduced negative log likelihood

$$\bar{\phi}(\theta') = \phi(C_{\text{opt}}(\theta'), \theta').$$

This idea of eliminating the length scale is reminiscent of the *variable projection* approaches of Golub and Pereira for nonlinear least squares problems.

The critical point equation for optimizing ϕ with respect to C yields

$$\frac{1}{2}C_{\text{opt}}^{-1}\operatorname{tr}(I) - \frac{1}{2}C_{\text{opt}}^{-2}y^T\bar{K}^{-1}y = 0,$$

which, with a little algebra, gives us

$$C_{\text{opt}} = \frac{y^T\bar{K}^{-1}y}{n}.$$

Note that if $z = L^{-1}y$ is the "whitened" version of the y vector, then $C_{\text{opt}} = \|z\|^2/n$ is roughly the sample variance. This scaling of the kernel therefore corresponds to trying to make the sample variance of the whitened signal equal to one.

If we substitute the formula for C_{opt} into the negative log likelihood formula, we have the reduced negative log likelihood

$$\begin{aligned}
\bar{\phi} &= \frac{1}{2}\log\det(C_{\text{opt}}\bar{K}) + \frac{1}{2}y^T(C_{\text{opt}}\bar{K})^{-1}y + \frac{n}{2}\log(2\pi) \\
&= \frac{1}{2}\log\det\bar{K} + \frac{n}{2}\log C_{\text{opt}} + \frac{1}{2}\frac{y^T\bar{K}^{-1}y}{C_{\text{opt}}} + \frac{n}{2}\log(2\pi) \\
&= \frac{1}{2}\log\det\bar{K} + \frac{n}{2}\log(y^T\bar{K}^{-1}y) + \frac{n}{2}\left(\log(2\pi) + 1 - \log n\right).
\end{aligned}$$

```
function nllr(K̄C :: Cholesky, c̄ :: AbstractVector, y :: AbstractVector)
    n = length(c̄)
    ϕ = n*(log(dot(c̄,y)) + log(2π) + 1 - log(n))/2
    for k = 1:n
        ϕ += log(K̄C.U[k,k])
    end
    ϕ
end

nllr(gp :: GPPContext) = nllr(getKC(gp), getc(gp), gety(gp))
getCopt(gp :: GPPContext) = ( getc(gp)'*gety(gp) )/gp.n
```

```
getCopt (generic function with 1 method)
```

Per our custom, we code a short sanity check.

```
let
    Zk, y = test_setup2d((x,y) → x^2 + y)
    gp = GPPContext(KernelSE{2}(1.0), 0.0, Zk, y)

    # Form scaled kernel Cholesky and weights
    KC = Cholesky(sqrt(getCopt(gp))*getKC(gp).U)
    c = KC\y

    check_approx(nll(KC, c, y), nllr(gp))
end
```

```
1.5774865384648788e-15
```

Differentiating $\bar{\phi}$ with respect to hyperparameters of $\bar{K}$ (i.e. differentiating with respect to any hyperparameter but the scaling factor), we have

$$\begin{aligned}\delta\bar{\phi} &= \frac{1}{2}\operatorname{tr}(\bar{K}^{-1}\delta\bar{K}) - \frac{n}{2}\frac{y^T\bar{K}^{-1}(\delta\bar{K})\bar{K}^{-1}y}{y^T\bar{K}^{-1}y} \\ &= \frac{1}{2}\operatorname{tr}(\bar{K}^{-1}\delta\bar{K}) - \frac{1}{2}\frac{\bar{c}^T(\delta\bar{K})\bar{c}}{C_{\text{opt}}},\end{aligned}$$

where $\bar{c} = \bar{K}^{-1}y$. Alternately, we can write this as

$$\delta\bar{\phi} = \frac{1}{2}\left\langle \bar{K}^{-1} - \frac{\bar{c}\bar{c}^T}{C_{\text{opt}}}, \delta\bar{K}\right\rangle_F.$$

For the second derivative, we have

$$\begin{aligned}\Delta\delta\bar{\phi} =& \frac{1}{2}\operatorname{tr}(\bar{K}^{-1}\Delta\delta\bar{K}) - \frac{1}{2}\operatorname{tr}(\bar{K}^{-1}\Delta\bar{K}\,\bar{K}^{-1}\delta\bar{K}) \\ &- \frac{1}{2}C_{\text{opt}}^{-1}\bar{c}^T(\Delta\delta\bar{K})\bar{c} + C_{\text{opt}}^{-1}\bar{c}^T(\Delta\bar{K})\bar{K}^{-1}(\delta\bar{K})\bar{c} \\ &- \frac{1}{2n}C_{\text{opt}}^{-2}\bar{c}^T(\delta\bar{K})\bar{c}\bar{c}^T(\Delta\bar{K})\bar{c}.\end{aligned}$$

This is very similar to the formula for the derivative of the unreduced likelihood, except that (a) there is an additional factor of C_{opt}^{-1} for the data fidelity derivatives in the second line, and (b) there is an additional term that comes from the derivative of C_{opt}. Consequently, we can rearrange for efficiency here the same way that we did for the unreduced NLL. If we want the gradient alone, we compute

$$\delta\phi = \left\langle \frac{1}{2}\left(\bar{K}^{-1} - C_{\text{opt}}^{-1}\bar{c}\bar{c}^T\right), \delta\bar{K}\right\rangle_F.$$

As before, our code has slightly special treatment for the case where we also want derivatives with respect to $z = \log\eta$.

```
function gθ_nllr!(g, gp :: GPPContext, invK, Copt)
    ctx, X, c = gp.ctx, getX(gp), getc(gp)
    d, n = size(X)
    for j = 1:n
        xj = @view X[:,j]
        cj = c[j]
        cj_div_Copt = cj/Copt
        gθ_kernel!(g, ctx, xj, xj, (invK[j,j]-cj*cj_div_Copt)/2)
        for i = j+1:n
            xi = @view X[:,i]
```

```
            ci = c[i]
            gθ_kernel!(g, ctx, xi, xj, (invK[i,j]-ci*cj_div_Copt))
        end
    end
    g
end

function gθz_nllr!(g, gp :: GPPContext, invK, Copt)
    ctx, X, c, s = gp.ctx, getX(gp), getc(gp), gp.η
    gθ_nllr!(g, gp, invK, Copt)
    g[nhypers(ctx)+1] = (tr(invK) - c'*c/Copt)*s/2
    g
end

gθ_nllr(gp :: GPPContext, invK, Copt) =
    gθ_nllr!(zeros(nhypers(gp.ctx)), gp, invK, Copt)

gθz_nllr(gp :: GPPContext, invK, Copt) =
    gθz_nllr!(zeros(nhypers(gp.ctx)+1), gp, invK, Copt)

gθ_nllr(gp :: GPPContext) = gθ_nllr(gp, getKC(gp)\I, getCopt(gp))
gθz_nllr(gp :: GPPContext) = gθz_nllr(gp, getKC(gp)\I, getCopt(gp))
```

```
gθz_nllr (generic function with 2 methods)
```

And our finite difference check:

```
let
    Zk, y = test_setup2d((x,y) -> x^2 + y)
    s, ℓ = 1e-4, 1.0
    z=log(s)

    gp_SE_nllr(ℓ,z) = nllr(GPPContext(KernelSE{2}(ℓ), exp(z), Zk, y))
    g = gθz_nllr(GPPContext(KernelSE{2}(ℓ), s, Zk, y))
    check_fd(g[1], ℓ->gp_SE_nllr(ℓ,z), ℓ)
    check_fd(g[2], z->gp_SE_nllr(ℓ,z), z)
end
```

```
1.6370516537939533e-7
```

If we want the gradient and the Hessian, we compute

$$\begin{aligned}
\delta\phi &= \operatorname{tr}(\delta\tilde{K}) - \frac{1}{2}C_{\text{opt}}^{-1}\tilde{c}^T\delta\tilde{K}\tilde{c} \\
\Delta\delta\phi &= \left\langle \frac{1}{2}\left(\bar{K}^{-1} - C_{\text{opt}}^{-1}\bar{c}\bar{c}^T\right), \Delta\delta\bar{K}\right\rangle_F \\
&\quad - \frac{1}{2}\langle\Delta\tilde{K}, \delta\tilde{K}\rangle_F + C_{\text{opt}}^{-1}\langle\Delta\tilde{K}\tilde{c}, \delta\tilde{K}\tilde{c}\rangle \\
&\quad - \frac{1}{2n}C_{\text{opt}}^{-2}\left(\tilde{c}^T(\Delta\tilde{K})\tilde{c}\right)\left(\tilde{c}^T(\delta\tilde{K})\tilde{c}\right)
\end{aligned}$$

This differs from the unscaled version primarily in the last term, which comes from differentiating C_{opt}^{-1} in the gradient. Hence, our Hessian code looks extremely similar to what we wrote before. We note that the last term can be rewritten in terms of the first derivatives of the data fidelity term from before:

$$-\frac{1}{2n}C_{\text{opt}}^{-2}\left(\tilde{c}^T(\Delta\tilde{K})\tilde{c}\right)\left(\tilde{c}^T(\delta\tilde{K})\tilde{c}\right) = -\frac{2}{n}\left(-\frac{1}{2}C_{\text{opt}}^{-1}\tilde{c}^T(\Delta\tilde{K})\tilde{c}\right)\left(-\frac{1}{2}C_{\text{opt}}^{-1}\tilde{c}^T(\delta\tilde{K})\tilde{c}\right).$$

This means the code for the Hessian of the reduced NLL is a very small rearrangement of our code for the unreduced NLL. In addition, we add a small tweak to deal with the case where we don't care about the derivatives with respect to $z = \log(\eta)$.

```
function Hθ_nllr(gp :: GPPContext; withz=true)
    ctx, X, y, s = gp.ctx, getX(gp), gety(gp), gp.η
    d, n = size(X)
    nθ = nhypers(ctx)

    # Factorization and initial solves
    KC = getKC(gp)
    invK = KC\I
    c̃ = KC.L\y
    c = getc(gp)
    Copt = getCopt(gp)
    ϕ = nllr(gp)
    ∂z_nllr = (tr(invK)-(c'*c)/Copt)*s/2

    # Set up space for NLL, gradient, and Hessian (including wrt z)
    nt = withz ? nθ+1 : nθ
    g = zeros(nt)
    H = zeros(nt,nt)

    # Add Hessian contribution from kernel second derivatives
    for j = 1:n
        xj = @view X[:,j]
```

```
        cj = c[j]
        cj_div_Copt = cj/Copt
        Hθ_kernel!(H, ctx, xj, xj, (invK[j,j]-cj*cj_div_Copt)/2)
        for i = j+1:n
            xi = @view X[:,i]
            ci = c[i]
            Hθ_kernel!(H, ctx, xi, xj, (invK[i,j]-ci*cj_div_Copt))
        end
    end
    if withz
        H[nt,nt] = ∂z_nllr
    end

    # Set up matrices δK
    δKs = zeros(n, n, nt)
    dθ_kernel!(δKs, ctx, X)
    if withz
        for j=1:n
            δKs[j,j,nt] = s
        end
    end

    # Set up whitened matrices δK̃ and products δK*c and δK̃*c̃
    δK̃s = whiten_matrices!(δKs, KC)
    δK̃c̃s = mul_slices!(zeros(n,nt), δK̃s, c̃)
    δK̃r = reshape(δK̃s, n*n, nt)

    # Add Hessian contributions involving whitened matrices
    mul!(H, δK̃r', δK̃r, -0.5, 1.0)
    mul!(H, δK̃c̃s', δK̃c̃s, 1.0/Copt, 1.0)

    # Last term of the Hessian written via data fidelity part of gradient
    mul!(g, δK̃c̃s', c̃, -0.5/Copt, 1.0)
    if withz
        g[end] = -(c'*c)/Copt*s/2
    end
    mul!(H, g, g', -2.0/n, 1.0)

    # And finish the gradient
    for j=1:nθ
        g[j] += tr(view(δK̃s,:,:,j))/2
    end
```

```
    if withz
        g[end] = ∂z_nllr
    end

    ϕ, g, H
end
```

```
Hθ_nllr (generic function with 1 method)
```

And the sanity check on a simple case.

```
let
    Zk, y = test_setup2d((x,y) → x^2 + y)
    s, ℓ = 1e-3, 0.89
    z = log(s)

    testf(ℓ,z) = Hθ_nllr(GPPContext(KernelSE{2}(ℓ), exp(z), Zk, y))
    ϕref, gref, Href = testf(ℓ, z)

    max(check_fd(gref[1], ℓ→testf(ℓ,z)[1][1], ℓ),
        check_fd(gref[2], z→testf(ℓ,z)[1][1], z),
        check_fd(Href[1,1], ℓ→testf(ℓ,z)[2][1], ℓ),
        check_fd(Href[1,2], ℓ→testf(ℓ,z)[2][2], ℓ),
        check_fd(Href[2,2], z→testf(ℓ,z)[2][2], z),
        check_approx(Href[1,2], Href[2,1]))
end
```

```
2.059080456905814e-8
```

The vector $c = \bar{c}/C_{\text{opt}}$ can be computed later if needed. However, we usually won't need it, as we can write the the posterior mean at a new point z as $\bar{k}_{Xz}^T \bar{c}$. The posterior variance is $C_{\text{opt}}(\bar{k}_{zz} - \bar{k}_{Xz}^T \bar{K}_{XX}^{-1} \bar{k}_{Xz})$.

Newton solve

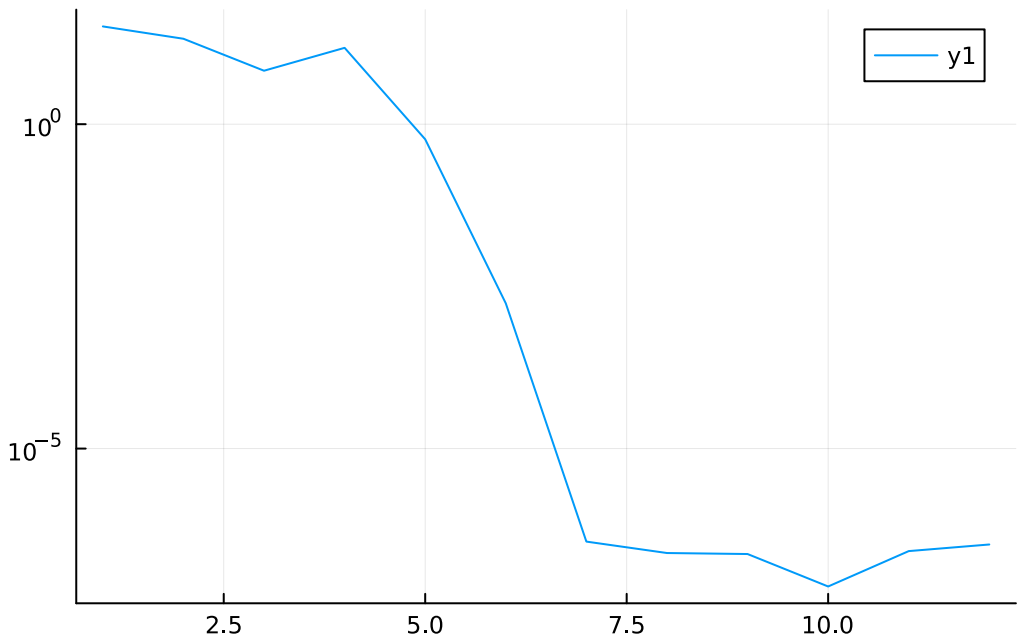
We are now in a position to tune the hyperparameters for a GP example. The problematic bit, which we will deal with in the next section, is the noise variance. Once we have a reasonable estimate for the optimized noise variance, we can get the expected quadratic convergence of Newton iteration. But the basin of convergence is rather small.

This iteration is meant to demonstrate this point – if we were trying to do this for real, of course, we would do a line search! As it is, we just cut the step if we are changing the noise variance by more than a factor of 20 or so.

```
function newton_hypers0(ctx :: KernelContext, η :: Float64,
                        X :: AbstractMatrix, y :: AbstractVector;
                        niters = 12, max_dz=3.0,
                        monitor = (ctx, η, ϕ, gϕ)→nothing)
    θ = getθ(ctx)
    for j = 1:niters
        ϕ, gϕ, Hϕ = Hθ_nllr(ctx, X, y, η)
        monitor(ctx, η, ϕ, gϕ)
        u = -Hϕ\gϕ
        if abs(u[end]) > max_dz
            u *= max_dz/abs(u[2])
        end
        θ[:] .+= u[1:end-1]
        ctx = updateθ(ctx, θ)
        η *= exp(u[end])
    end
    ctx, η
end

function newton_hypers0(gp :: GPPContext;
                        niters = 12, max_dz=3.0,
                        monitor = (ctx, η, ϕ, gϕ)→nothing)
    ctx, η = gp.ctx, gp.η
    θ = getθ(ctx)
    for j = 1:niters
        ϕ, gϕ, Hϕ = Hθ_nllr(gp)
        monitor(ctx, η, ϕ, gϕ)
        u = -Hϕ\gϕ
        if abs(u[end]) > max_dz
            u *= max_dz/abs(u[2])
        end
        θ[:] .+= u[1:end-1]
        ctx = updateθ(ctx, θ)
        η *= exp(u[end])
        gp = change_kernel!(gp, ctx, η)
    end
    gp
end
```

```
let
    Zk, y = test_setup2d((x,y) → x^2 + cos(3*y) + 5e-4*cos(100*y), 40)
    gp = GPPContext(KernelSE{2}(0.7), 1e-4, Zk, y)

    normgs = []
    function monitor(ctx, η, ϕ, gϕ)
        ℓ = ctx.ℓ
        println("($ℓ,$η)\t$ϕ\t$(norm(gϕ))")
        push!(normgs, norm(gϕ))
    end

    gp = newton_hypers0(gp, monitor=monitor)
    println("Finished with ($(gp.ctx.ℓ), $(gp.η))")
    plot(normgs, yscale=:log10)
end
```

```
(0.7,0.0001)    -100.34663467307195 32.14208440052669
(0.7498542544390177,4.978706836786395e-6)   -126.82428640669053 20.742637955268457
(0.8439621011055604,2.478752176666359e-7)   -146.91589825482157 6.671772340200783
(0.9878749215414256,1.586599307582333e-8)   -151.2791860650658  15.027635463007462
(0.9630431836792746,3.3372355028406606e-8)  -152.11784152786728 0.5853529193263738
(0.9671795567713537,3.2090296074910317e-8) -152.12017036528601 0.0017314660913735835
(0.9671939969676034,3.208561034733114e-8)   -152.12017043777809 3.6830691226375475e-
7
(0.9671939983365573,3.2085609402853374e-8) -152.12017043751786 2.4554119137011834e-
7
(0.9671939991118537,3.2085609299671665e-8) -152.12017045531582 2.3693952882918904e-
7
(0.9671939987679161,3.208560922257725e-8)   -152.1201704157499  7.489078421619986e-8
(0.9671939973341431,3.208560979132429e-8)   -152.12017048164074 2.6292035482471436e-
7
(0.9671939998082998,3.2085608954652534e-8)  -152.1201704429588  3.3245986418160274e-
7
Finished with (0.9671939981859833, 3.208560934573076e-8)
```



Noise variance

Unfortunately, the reduced NLL function (viewed as a function of η alone) is not particularly nice. It has a large number of log-type singularities on the negative real axis ($2n-1$ of them), with some getting quite close to zero. So Newton-type iterations are likely to be problematic unless we have a good initial guess, particularly when we are looking for optima close to zero. Fortunately, working with $\log \eta$ rather than η mitigates some of the numerical troubles; and, furthermore, we can be clever about our use of factorizations in order to make computing the reduced NLL and its derivatives with respect to η sufficiently cheap so that we don't mind doing several of them.

Tridiagonal reduction

Fortunately, we can evaluate the reduced NLL and its derivatives quickly by solving an eigenvalue problem – an up-front $O(n^3)$ cost followed by an $O(n)$ cost for evaluating the NLL and derivatives. Better yet, we do not need to get all the way to an eigenvalue decomposition – a tridiagonal reduction is faster and gives the same costs. That is, we write

$$T = Q^T K Q, \tilde{y} = Q^T y$$

and note that

$$Q^T(K+\eta I)Q = T + \eta I,$$
$$\log\det(K+\eta I) = \log\det(T+\eta I),$$
$$y^T(K+\eta I)^{-1}y = \tilde{y}^T(T+\eta I)^{-1}\tilde{y}.$$

Therefore, all the computations that go into the reduced NLL and its derivatives can be rephrased in terms of T and $\tilde{y}$.

The Julia linear algebra library does not include a routine for tridiagonal reduction, but it does include a Householder QR factorization routine, and we can re-use the implementations of the reflector computation and application. The following function overwrites K and y with the entries of T (in the main diagonal and subdiagonal entries) $\tilde{y}$.

```
function tridiag_reduce!(K, y)
    n = length(y)
    for k = 1:n-2
        x = view(K, k+1:n,k)
        τk = LinearAlgebra.reflector!(x)
        LinearAlgebra.reflectorApply!(x, τk, view(y, k+1:n))
        LinearAlgebra.reflectorApply!(x, τk, view(K, k+1:n, k+1:n))
        LinearAlgebra.reflectorApply!(x, τk, view(K, k+1:n, k+1:n)')
    end
end
```

```
tridiag_reduce! (generic function with 1 method)
```

It's helpful to extract the parameters for the tridiagonal

```
function tridiag_params!(K, alpha, beta, s=0.0)
    n = size(K,1)
    for j = 1:n-1
        alpha[j] = K[j,j] + s
        beta[j] = K[j+1,j]
    end
    alpha[n] = K[n,n] + s
end

function tridiag_params(K, s=0.0)
    n = size(K,1)
    alpha = zeros(n)
    beta = zeros(n-1)
    tridiag_params!(K, alpha, beta, s)
    alpha, beta
```

```
end

get_tridiag(K) = SymTridiagonal(tridiag_params(K)... )
```

```
get_tridiag (generic function with 1 method)
```

And we want to be able to compute the Cholesky factorization in place and use it to solve linear systems and to

```
function cholesky_T!(alpha, beta)
    n = length(alpha)
    for j = 1:n-1
        alpha[j] = sqrt(alpha[j])
        beta[j] /= alpha[j]
        alpha[j+1] -= beta[j]^2
    end
    alpha[n] = sqrt(alpha[n])
end

function cholesky_T_Lsolve!(alpha, beta, y)
    n = length(alpha)
    y[1] /= alpha[1]
    for j = 2:n
        y[j] = (y[j]-beta[j-1]*y[j-1])/alpha[j]
    end
    y
end

function cholesky_T_Rsolve!(alpha, beta, y)
    n = length(alpha)
    y[n] /= alpha[n]
    for j = n-1:-1:1
        y[j] = (y[j]-beta[j]*y[j+1])/alpha[j]
    end
    y
end

function cholesky_T_solve!(alpha, beta, y)
    cholesky_T_Lsolve!(alpha, beta, y)
    cholesky_T_Rsolve!(alpha, beta, y)
    y
end
```

```
cholesky_T_solve! (generic function with 1 method)
```

The negative log likelihood in this case involves $\mathrm{tr}(K^{-1}) = \mathrm{tr}(T^{-1})$. Given a Cholesky factorization $T = L^T L$, we have

$$\mathrm{tr}(T^{-1}) = \mathrm{tr}(L^{-T}L^{-1}) = \|L^{-1}\|_F^2$$

Let $X = L^{-1}$ and $x_j = L^{-1}e_j$. Then the equation $XL = I$ gives us the recurrence

$$\begin{aligned} \alpha_n x_n &= e_n \\ \alpha_j x_j + \beta_j x_{j+1} &= e_j, \quad j < n \end{aligned}$$

where α and β denote the diagonal and off-diagonal entries of the Cholesky factor. We can rewrite this as

$$\begin{aligned} x_n &= e_n/\alpha_n \\ x_j &= (e_j - \beta_j x_{j+1})/\alpha_j, \quad j < n \end{aligned}$$

Note that $x_{j+1} \perp e_j$, and so by the Pythagorean theorem,

$$\begin{aligned} \|x_n\|^2 &= 1/\alpha_n^2 \\ \|x_j\|^2 &= (1 + \beta_j^2\|x_{j+1}\|^2)/\alpha_j^2, \quad j < n \end{aligned}$$

Running this recurrence backward and summing the $\|x_j\|^2$ gives us an $O(n)$ algorithm for computing $\mathrm{tr}(T^{-1})$

```
function cholesky_trinvT(alpha, beta)
    n = length(alpha)
    n2x = 1.0/alpha[n]^2
    n2xsum = n2x
    for j = n-1:-1:1
        n2x = (1.0 + beta[j]^2*n2x)/alpha[j]^2
        n2xsum += n2x
    end
    n2xsum
end
```

```
cholesky_trinvT (generic function with 1 method)
```

Putting this all together, we have the following code for computing the reduced negative log likelihood and its derivative after a tridiagonal reduction.

```
function nllrT!(T, y, s, alpha, beta, c)
    n = length(y)
    tridiag_params!(T, alpha, beta, s)
    c[:] .= y
    cholesky_T!(alpha, beta)
    cholesky_T_solve!(alpha, beta, c)
    cTy = c'*y
    Copt = cTy/n

    # Compute NLL
    ϕ = n/2*(log(cTy) + log(2π) + 1 - log(n))
    for j = 1:n
        ϕ += log(alpha[j])
    end

    # Compute NLL derivative
    dϕ = (cholesky_trinvT(alpha, beta) - (c'*c)/Copt)/2

    ϕ, dϕ
end

function nllrT(T, y, s)
    n = length(y)
    alpha = zeros(n)
    beta = zeros(n)
    c = zeros(n)
    nllrT!(T, y, s, alpha, beta, c)
end
```

```
nllrT (generic function with 1 method)
```

Finally, we do a consistency check between our previous computations of the reduced NLL and the version based on tridiagonalization.

```
let
    n = 10
    Zk, y = test_setup2d((x,y) → x^2 + y, n)
    ctx = KernelSE{2}(1.0)
    η = 1e-3

    # Ordinary reduced NLL computation (full and pieces)
```

```
    K = kernel(ctx, Zk)
    KC = cholesky(K+η*I)
    c = KC\y
    data1 = c'*y
    logdet1 = sum(log.(diag(KC.U)))
    trinv1 = tr(KC\I)
    ϕ1 = nllr(KC, c, y)

    # Reduced NLL computation
    tridiag_reduce!(K, y)
    alpha, beta = tridiag_params(K, η)
    cholesky_T!(alpha, beta)
    c = copy(y)
    cholesky_T_solve!(alpha, beta, c)
    data2 = c'*y
    logdet2 = sum(log.(alpha))
    trinv2 = cholesky_trinvT(alpha, beta)
    ϕ2, dϕ = nllrT!(K, y, η, alpha, beta, c)

    max(check_approx(data1, data2),
        check_approx(logdet1, logdet2),
        check_approx(trinv1, trinv2),
        check_approx(ϕ1, ϕ2))
end
```

```
1.082733305815383e-13
```

We will also do a finite difference check on the computed gradient.

```
let
    Zk, y = test_setup2d((x,y) → x^2 + y)
    ctx = KernelSE{2}(1.0)
    η = 1e-3
    K = kernel(ctx, Zk)
    tridiag_reduce!(K, y)
    check_fd(nllrT(K, y, η)[2], η→nllrT(K, y, η)[1], η)
end
```

```
2.0550690950445429e-7
```

Optimizing noise variance

The main cost of anything to do with noise variance is the initial tridiagonal reduction. After that, each step costs only $O(n)$, so we don't need to be too penny-pinching about the cost of doing evaluations. Therefore, we use an optimizer that starts with a brute force grid search (in $\log \eta$) to find a bracketing interval for a best guess at the global minimum over the range, and then does a few steps of secant iteration to refine the result.

```
function min_nllrT!(T, y, ηmin, ηmax, alpha, beta, c; nsamp=10, niter=5)

    # Sample on an initial grid
    logmin = log(ηmin)
    logmax = log(ηmax)
    logx   = range(logmin, logmax, length=nsamp)
    nllx   = [nllrT!(T,y,exp(s),alpha,beta,c) for s in logx]

    # Find min sample index and build bracketing interval
    i = argmin(t[1] for t in nllx)
    if ((i == 1 && nllx[i][2] > 0) ||
        (i == nsamp && nllx[i][2] < 0))
        return exp(logx[i]), nllx[i]...
    end
    ilo = i
    ihi = i
    if nllx[i][2] < 0
        ihi = i+1
    else
        ilo = i-1
    end

    # Do a few steps of secant iteration
    a,  b  = logx[ilo], logx[ihi]
    fa, fb = nllx[ilo], nllx[ihi]
    for k = 1:niter
        dfa, dfb = exp(a)*fa[2], exp(b)*fb[2]
        d = (a*dfb-b*dfa)/(dfb-dfa)
        fd = nllrT!(T,y,exp(d),alpha,beta,c)
        a, b = b, d
        fa, fb = fb, fd
    end

    exp(b), nllrT!(T,y,exp(b),alpha,beta,c)...
end
```

```
min_nllrT! (generic function with 2 methods)
```

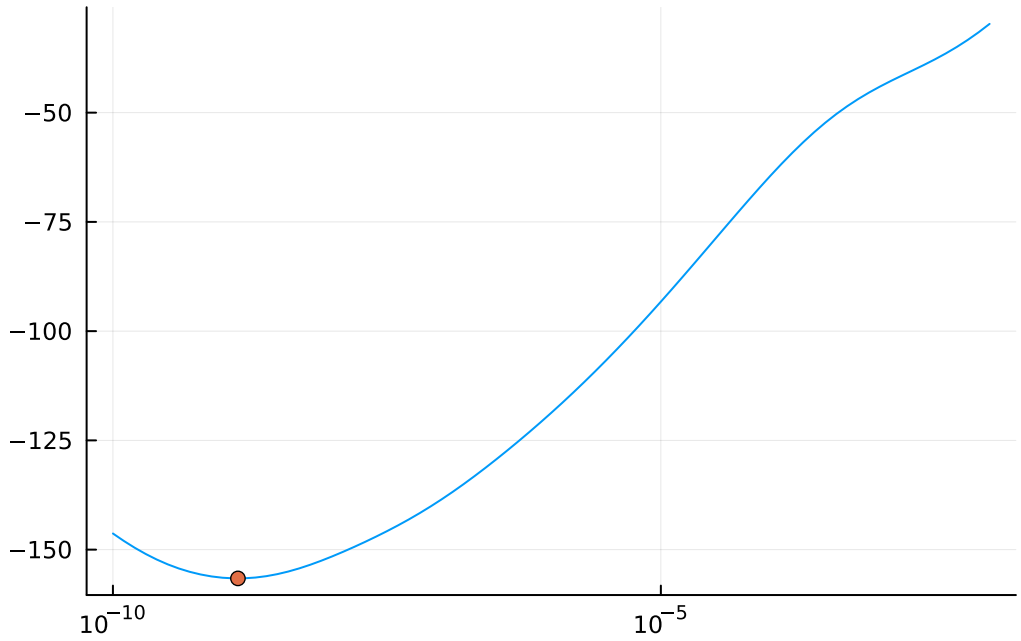
It's useful to us the GP context objects to wrap this up.

```
function tridiagonalize!(gp :: GPPContext)
    K = getK(gp)
    y = view(gp.scratch,1:gp.n,1)
    kernel!(K, gp.ctx, getX(gp))
    copy!(y, gety(gp))
    tridiag_reduce!(K, y)
end

function nllrT!(gp :: GPPContext, η :: Float64)
    K     = getK(gp)
    y     = view(gp.scratch,1:gp.n,1)
    alpha = view(gp.scratch,1:gp.n,2)
    beta  = view(gp.scratch,1:gp.n,3)
    c     = getc(gp)
    nllrT!(K,y,η,alpha,beta,c)
end

function min_nllrT!(gp :: GPPContext, ηmin, ηmax;
                    tridiagonalized=false, nsamp=10, niter=5)
    if !tridiagonalized
        tridiagonalize!(gp)
    end
    K     = getK(gp)
    y     = view(gp.scratch,1:gp.n,1)
    alpha = view(gp.scratch,1:gp.n,2)
    beta  = view(gp.scratch,1:gp.n,3)
    c     = getc(gp)
    ηopt, ϕ, dϕ = min_nllrT!(K,y,ηmin,ηmax,alpha,beta,c,
                             nsamp=nsamp, niter=niter)
    change_kernel!(gp, gp.ctx, ηopt), ϕ, dϕ
end
```

```
min_nllrT! (generic function with 2 methods)
```

An example in this case is useful.

```
let
    # Set up sample points and test function
    n = 40
    Zk, y = test_setup2d((x,y) → x^2 + cos(3*y + 5e-4*cos(100*x)), n)
    ℓ = 1.2

    gp = GPPContext(KernelSE{2}(ℓ), 1e-10, Zk, y)
    tridiagonalize!(gp)
    ss = 10.0.^(-10:0.1:-2)
    ϕss = [nllrT!(gp,s)[1] for s in ss]
    gp, ϕ, dϕ = min_nllrT!(gp,1e-10,1e-2,tridiagonalized=true)

    plot(ss, ϕss, xscale=:log10, legend=:false)
    plot!([gp.η], [ϕ], marker=:circle)
end
```



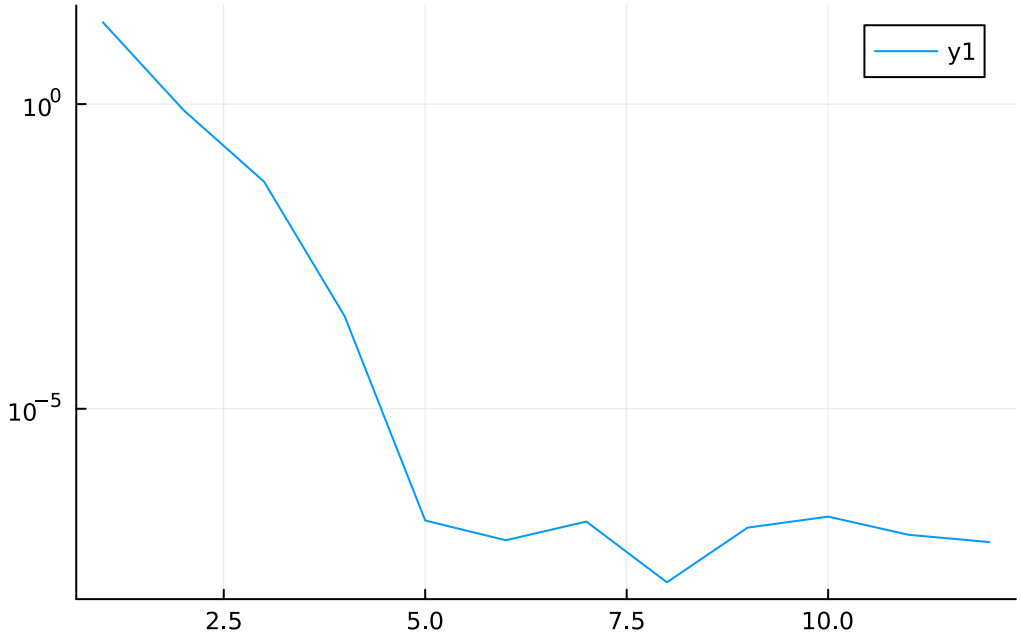
Putting it together

With the fast tuning of the noise variance parameter ready, we can tweak our Newton iteration so that all the other hyperparameters are updated by Newton, and then the noise variance is updated with our tridiagonal solve.

```
function newton_hypers1(gp :: GPPContext;
                        ηmin = 1e-10, ηmax = 1e-2,
                        niters = 12, max_dz=3.0,
                        monitor = (ctx, η, ϕ, gϕ)→nothing)
    ctx, η = gp.ctx, gp.η
    θ = getθ(ctx)
    for j = 1:niters
        ϕ, gϕ, Hϕ = Hθ_nllr(gp)
        monitor(ctx, gp.η, ϕ, gϕ)
        u = -Hϕ\gϕ
        if abs(u[end]) > max_dz
            u *= max_dz/abs(u[2])
        end
        θ[:] .+= u[1:end-1]
        ctx = updateθ(ctx, θ)
        gp = change_kernel_nofactor!(gp, ctx, η)
        gp, _, _ = min_nllrT!(gp, ηmin, ηmax)
    end
    gp
end

let
    n = 40
    Zk, y = test_setup2d((x,y) → x^2 + cos(3*y) + 1e-3*cos(100*x), n)
    gp = GPPContext(KernelSE{2}(1.2), 1e-10, Zk, y)

    normgs = []
    function monitor(ctx, η, ϕ, gϕ)
        println("$(ctx.ℓ)\t$η\t$ϕ\t$(norm(gϕ))")
        push!(normgs, norm(gϕ))
    end
    newton_hypers1(gp, monitor=monitor)
    plot(normgs, yscale=:log10)
end
```

```
1.2 1.0e-10 -119.42163390351996 22.03357398517793
0.8984763999767447  6.566944870255172e-8    -145.59737494792006 0.8003176658372981
0.8890267450522882  6.681776534227681e-8    -145.60132356761338 0.053415846634424516
0.8882976419838108  6.689449595779165e-8    -145.60134311752387 0.0003311806072982448
0.8882930648453994  6.689497205777006e-8    -145.6013431089664  1.4680619757617918e-7
0.8882930663171954  6.68949721320182e-8 -145.60134308390008 6.954155368859481e-8
0.8882930647672218  6.689497180181381e-8    -145.6013431168838  1.4097958959545506e-7
```

```
0.8882930663256122  6.68949715608291e-8 -145.60134312368731 1.4178402196364222e-8
0.8882930662459122  6.689497107835916e-8   -145.60134315547668 1.1188196638635864e-7
0.8882930679061167  6.689497199390357e-8   -145.6013431154924  1.6989506413895535e-7
0.8882930656382045  6.68949714961125e-8 -145.60134312216104 8.544341083122759e-8
0.8882930664304652  6.689497238403028e-8    -145.60134311586737 6.491405739521202e-8
```



Acquisition functions

We are interested in optimizing some objective function f via Bayesian optimization, which means that at each step of the algorithm we will choose a new point x_j at which to sample f by optimizing an *acquisition function* that balances exploration with exploitation. We will focus on two such functions: the lower confidence bound, and the expected improvement.

Lower confidence bound

The lower confidence bound acquisition function is

$$\alpha(x) = \mu(x) - \lambda\sigma(x).$$

We have already worked out spatial derivatives of the predictive mean and variance, so all we need to do to finish is differentiating $\sigma(x) = \sqrt{v(x)}$. Formulas are given below; derivation is

an exercise for the reader:

$$\nabla\sigma = \frac{1}{2\sigma}\nabla v$$
$$H\sigma = \frac{1}{2\sigma}Hv - \frac{1}{4\sigma^3}(\nabla v)(\nabla v)^T$$

```
function Hgx_αLCB(gp :: GPPContext, x :: AbstractVector, λ :: Float64)
    Copt = getCopt(gp)
    μ, gμ, Hμ = mean(gp, x), gx_mean(gp, x), Hx_mean(gp, x)
    v, gv, Hv = Copt*var(gp, x), Copt*gx_var(gp, x), Copt*Hx_var(gp, x)
    σ = sqrt(v)
    α = μ-λ*σ
    gα = gμ - λ/(2σ)*gv
    Hα = Hμ - λ/(2σ)*Hv + λ/(4*σ*v)*gv*gv'
    α, gα, Hα
end

let
    Zk, y = test_setup2d((x,y)→x^2+y)
    gp = GPPContext(KernelSE{2}(0.5), 1e-8, Zk, y)
    z = [0.47; 0.47]
    dz = randn(2)
    λ = 2.3
    g(s)  = Hgx_αLCB(gp, z+s*dz, λ)[1]
    dg(s) = Hgx_αLCB(gp, z+s*dz, λ)[2]
    Hg(s) = Hgx_αLCB(gp, z+s*dz, λ)[3]
    check_fd(dg(0)'*dz, g, 0.0),
    check_fd(Hg(0)*dz, dg, 0.0)
end
```

```
(3.06169624860143e-10, 5.394619940432574e-10)
```

Expected improvement

We will follow the usual perspective of deriving expected improvement as a method of *maximizing* an objective rather than *minimizing* the objective. We can (and will) switch perspectives later by multipling by minus one.

The standard derivation

The expected improvement is

$$\alpha(x) = \mathbb{E}[I(x)],$$

where the improvement (for the minimization problem) is $I(x) = (f(x) - f_*)_+$ where f_* is the largest function value found so far. Equivalently,

$$\alpha(x) = \int_{f_*}^{\infty} (f - f_*) p(f)\, df$$

where $p(f)$ is the predictive probability density for $f(x)$. It is convenient to change variables to $z = (f - \mu)/\sigma$, which is a standard normal random variable, giving us

$$\begin{aligned}
\alpha &= \int_{z_*}^{\infty} \sigma(z - z_*)\phi(z)\, dz = \sigma G(z_*), \\
G(z_*) &:= \int_{z_*}^{\infty} (z - z_*)\phi(z)\, dz,
\end{aligned}$$

where $\phi(z)$ is the pdf or the standard normal. We note that

$$-\phi'(z) = z\phi(z), \quad \Phi'(z) = \phi(z)$$

where $\Phi(z)$ is the standard normal cdf. Therefore,

$$\begin{aligned}
G(z_*) &= (-\phi(z) - z_*\Phi(z))\big|_{z=z_*}^{\infty} \\
&= \phi(z_*) - z_*(1 - \Phi(z_*)) \\
&= \phi(z_*) - z_* Q(z_*),
\end{aligned}$$

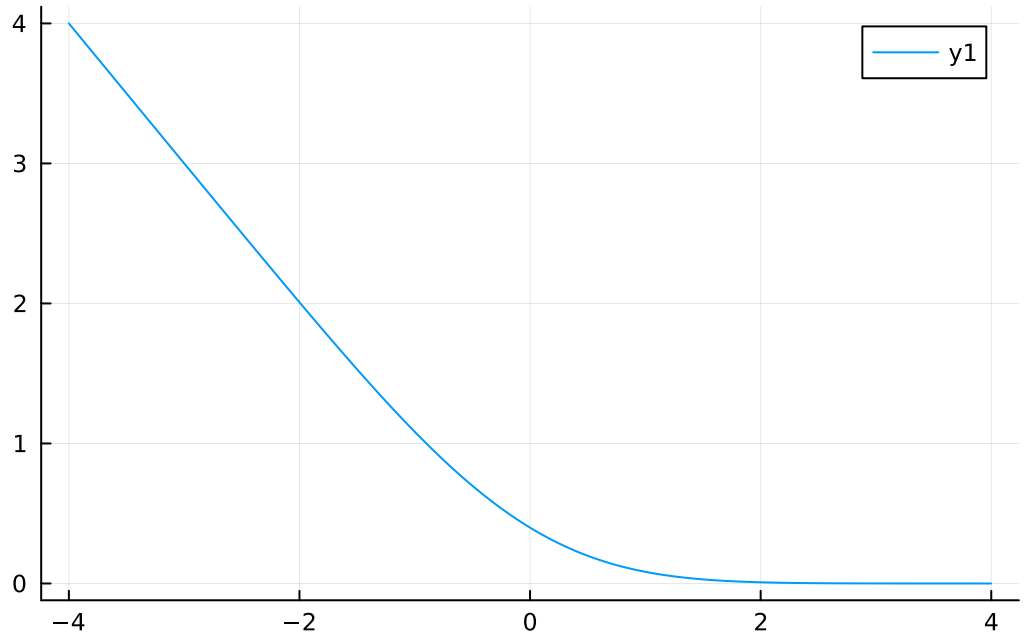
where $Q(z_*) = 1 - \Phi(z_*) = \Phi(-z_*)$ is the complementary cdf for the standard normal. Equivalently, if we let $u = -z_* = (\mu - f_*)/\sigma$, we have

$$G(z_*) = \phi(u) + u\Phi(u),$$

which is the form that we see most often for the expected improvement.

A plot of the G is perhaps useful:

```
let
    z = range(-4.0, 4.0, step=1e-3)
    G(z) = normpdf(z) - z*normccdf(z)
    plot(z, G.(z))
end
```



For sufficiently negative values, we have $G(z) \approx -z$, and for sufficiently positive values we have $G(z) \approx 0$. Near zero, there is a smoothed-out transition region.

For optimization, it's useful to have derivatives as well as a formula for the expected improvement. Fortunately, they are pretty simple: the first and second derivatives of G are

$$G'(z) = -Q(z)$$
$$G''(z) = \phi(z).$$

Per usual, a finite difference check gives us confidence in our ability to do calculus.

```
let
    z = 0.123
    G(z) = normpdf(z) - z*normccdf(z)
    dG(z) = -normccdf(z)
    HG(z) = normpdf(z)
    check_fd(dG(z), G, z),
    check_fd(HG(z), dG, z)
end
```

```
(7.459271484967615e-13, 1.3761144093341995e-11)
```

Negative log EI

One of the challenges of working with expected improvement is that when z_* gets larger than zero, the expected improvement becomes exponentially small. To analyze this, it is helpful to introduce the *Mills ratio*

$$R(z) = Q(z)/\phi(z),$$

so that we can rewrite the scaled EI function $G(z)$ as

$$G(z) = H(z)\phi(z), \quad H(z) = 1 - zR(z).$$

Using asymptotics of the Mills ratio for large z, we have

$$\begin{aligned} R(z) &= z^{-1} - z^{-3} + O(z^{-5}), \\ H(z) &= z^{-2} - O(z^{-4}), \end{aligned}$$

and so $G(z)$ decays asymptotically like $z^{-2}\phi(z)$ — which is rapid decay indeed.

Because it gets so close to zero, it's numerically advantageous to work not with expected improvement, but with the negative log of the expected improvement (NLEI).

$$\psi_{NLEI} = -\log \alpha_{EI} = -\frac{1}{2}\log(\sigma^2) - \log G(z_*).$$

We will write $\psi_{NLG}(z) = -\log G(z)$ for the second term.

We evaluate ψ_{NLG} differently depending on the argument. For z negative (or not too positive), the standard formulas are entirely reasonable. That is,

$$\begin{aligned} \psi_{NLG}(z) &= -\log\left(\phi(z) - zQ(z)\right) \\ \psi'_{NLG}(z) &= \frac{Q(z)}{G(z)} \\ \psi''_{NLG}(z) &= \frac{-\phi(z)G(z) + Q(z)^2}{G(z)^2} \end{aligned}$$

We implement these formulas as "version zero" of the computation.

```
function DψNLG0(z)
    ϕz = normpdf(z)
    Qz = normccdf(z)
    Gz = ϕz-z*Qz
    ψz = -log(Gz)
    dψz = Qz/Gz
    Hψz = (-ϕz*Gz + Qz^2)/Gz^2
    ψz, dψz, Hψz
end
```

```
let
    z = 0.123
    check_fd(DψNLG0(z)[2], z→DψNLG0(z)[1], z),
    check_fd(DψNLG0(z)[3], z→DψNLG0(z)[2], z)
end
```

```
(2.079410599314183e-12, 7.721100664762289e-11)
```

If we believe our model, we are unlikely to have to evaluate ψ_{NLG} for very large arguments. Such an occurrence would mean a very poorly calibrated model! Nonetheless, very poor calibration could potentially happen at some point, and it is worth knowing how to deal with it.

For values of z that are larger than about 26, the standard formulas get perilously close to underflow. In this case, it is more reasonable to write $G(z) = H(z)\phi(z)$ and its derivatives in terms of the Mills ratio $R(z)$ and the related $H(z) = 1 - zR(z)$:

$$\begin{aligned}
\psi_{NLG}(z) &= -\log H(z) + \frac{z^2}{2} + \frac{1}{2}\log(2\pi) \\
\psi'_{NLG}(z) &= \frac{R(z)}{H(z)} \\
\psi''_{NLG}(z) &= \frac{-H(z) + R(z)^2}{H(z)^2}.
\end{aligned}$$

We can write the Mills ratio as $Q(z)/\phi(z)$ when z is not too enormous; alternately, we can write

$$R(z) = \sqrt{\frac{\pi}{2}}\,\mathrm{erfcx}\left(\frac{z}{\sqrt{2}}\right)$$

where erfcx is the scaled complementary error function (assuming one has a library of special functions that includes erfcx).

```
function DψNLG1(z)
    Rz = sqrt(π/2)*erfcx(z/√2)
    Hz = 1-z*Rz
    ψz = -log1p(-z*Rz) + 0.5*(z^2 + log(2π))
    dψz = Rz/Hz
    Hψz = (-Hz+Rz^2)/Hz^2
    ψz, dψz, Hψz
end

let
```

```
    z = 0.123
    z1 = 30.456
    max(check_approx(DψNLG0(z)[1], DψNLG1(z)[1]),
        check_approx(DψNLG0(z)[2], DψNLG1(z)[2]),
        check_approx(DψNLG0(z)[3], DψNLG1(z)[3]),
        check_fd(DψNLG1(z)[2], z→DψNLG1(z)[1], z),
        check_fd(DψNLG1(z)[3], z→DψNLG1(z)[2], z)),
    max(check_fd(DψNLG1(z1)[2], z→DψNLG1(z)[1], z1, h=1e-4),
        check_fd(DψNLG1(z1)[3], z→DψNLG1(z)[2], z1, h=1e-4))

end
```

```
(4.5211247248181733e-10, 5.722663395315026e-9)
```

Were we unwilling to find a library for computing the scaled complementary error function, another approach would be to use Laplace's continued fraction expansion for the Mills ratio:

$$R(z) = \frac{1}{z+}\frac{1}{z+}\frac{2}{z+}\frac{3}{z+\ldots},$$

or we can write

$$R(z) = \frac{1}{z + W(z)}, \quad W(z) = \frac{1}{z+}\frac{2}{z+}\frac{3}{z+\ldots},$$

where $WR = H$. Then we can put everything in terms of W:

$$\begin{aligned}
R(z) &= \frac{1}{z + W(z)} \\
H(z) &= W(z)R(z) \\
\psi_{NLG}(z) &= \log\left(1 + \frac{z}{W(z)}\right) + \frac{z^2}{2} + \frac{1}{2}\log(2\pi) \\
\psi'_{NLG}(z) &= W(z)^{-1} \\
\psi''_{NLG}(z) &= \frac{1 - W(z)(z + W(z))}{W(z)^2}.
\end{aligned}$$

```
function DψNLG2(z)

    # Approximate W by 20th convergent
    W = 0.0
    for k = 20:-1:1
        W = k/(z+W)
    end
```

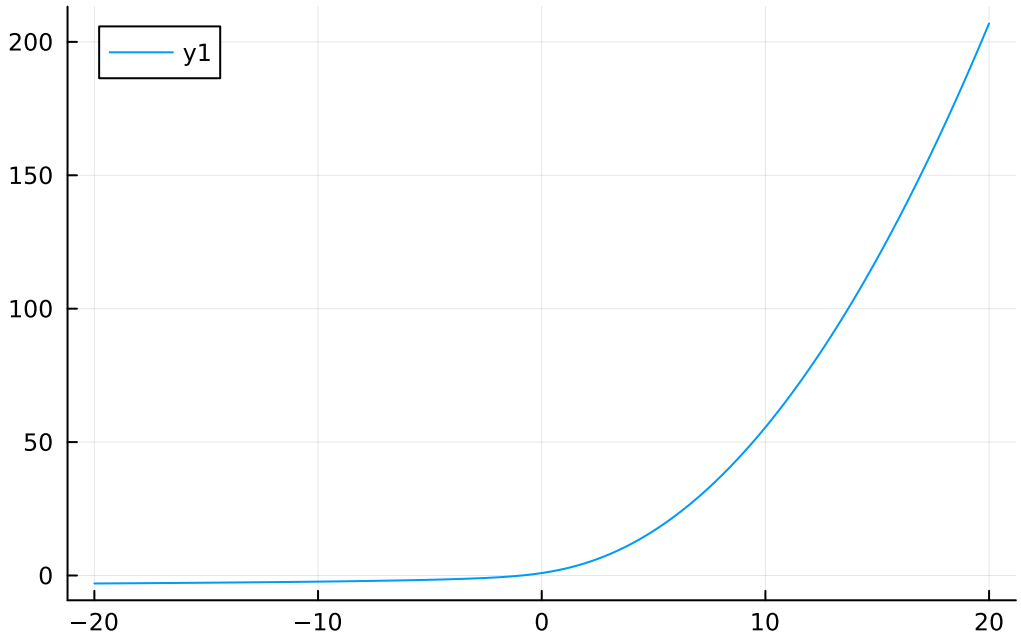
```
    ψz = log1p(z/W) + 0.5*(z^2 + log(2π))
    dψz = 1/W
    Hψz = (1-W*(z+W))/W^2

    ψz, dψz, Hψz
end

let
    z = 5.23
    check_approx(DψNLG0(z)[1], DψNLG2(z)[1]),
    check_approx(DψNLG0(z)[2], DψNLG2(z)[2]),
    check_approx(DψNLG0(z)[3], DψNLG2(z)[3])
end
```

```
(4.341622376621471e-15, 8.3428659267786e-14, 2.920283829794391e-12)
```

The continued fraction doesn't converge super-fast, but that is almost surely fine for what we're doing here. By z values of 5 or so, 20 terms is quite adequate to get good accuracy — and this version does not need any more exotic special functions in our library. If needed, we could do a similar manipulation to get an optimized rational approximation to W from Cody's 1969 rational approximation to the complementary error function (erfc). Or we could use a less accurate approximation — the point of getting the tails right is really to give us enough information to climb out of the flat regions for EI.

Putting everything together, we have

```
# By z = 6, a 20-term convergent of the rational approx is plenty
DψNLG(z) = if z < 6.0 DψNLG0(z) else DψNLG2(z) end

let
    zs = range(-20, 20, step=1e-3)
    ψzs = [DψNLG(z)[1] for z in zs]
    plot(zs, ψzs)
end
```



Of course, if we are interested in minimization rather than maximization, then we want to minimize $\psi_{NLG}(-z_*(x))$ rather than $\psi_{NLG}(z_*(x))$.

Gradients and Hessians

To choose a next point for a BO minimization problem using an EI acquisition function, we minimize $\psi_{NLG}(u(x))$ where

$$u(x) = -z_*(x) = \frac{\mu(x) - f_*}{\sigma(x)}.$$

Taking first and second derivatives, we have

$$\begin{aligned} u_{,i} &= \frac{\mu_{,i}}{\sigma} - u\frac{\sigma_{,i}}{\sigma} \\ u_{,ij} &= \frac{\mu_{,ij}}{\sigma} - \frac{\mu_{,i}}{\sigma}\frac{\sigma_{,j}}{\sigma} - \frac{\mu_{,j}}{\sigma}\frac{\sigma_{,i}}{\sigma} - u\frac{\sigma_{,ij}}{\sigma} + u\frac{\sigma_{,i}\sigma_{,j}}{\sigma^2} \end{aligned}$$

Let $v(x) = \sigma^2(x)$ be the variance; then note that

$$\begin{aligned} v_{,i} &= 2\sigma\sigma_{,i} \\ v_{,ij} &= 2\sigma_{,j}\sigma_{,j} + 2\sigma\sigma_{,ij}, \end{aligned}$$

which we can rearrange to

$$\frac{\sigma_{,i}}{\sigma} = \frac{v_{,i}}{2v}$$
$$\frac{\sigma_{,ij}}{\sigma} = \frac{v_{,ij}}{2v} - \frac{v_{,i}}{2v}\frac{v_{,j}}{2v}.$$

Substituting back in, we have

$$u_{,i} = \frac{\mu_{,i}}{\sigma} - u\frac{v_{,i}}{2v}$$
$$u_{,ij} = \frac{\mu_{,ij}}{\sigma} - \frac{1}{2}\left(\frac{\mu_{,i}}{\sigma}\frac{v_{,j}}{v} + \frac{\mu_{,j}}{\sigma}\frac{v_{,i}}{v}\right) + \frac{1}{2}u\left(\frac{3}{2}\frac{v_{,i}}{v}\frac{v_{,j}}{v} - \frac{v_{,ij}}{v}\right)$$

In vector notation, we have

$$\nabla u = \frac{\nabla\mu}{\sigma} - \frac{u}{2}\frac{\nabla v}{v}$$
$$Hu = \frac{H\mu}{\sigma} - \frac{1}{2}\left(\frac{\nabla\mu}{\sigma}\frac{(\nabla v)^T}{v} + \frac{\nabla v}{v}\frac{(\nabla\mu)^T}{\sigma}\right) + \frac{u}{2}\left(\frac{3}{2}\frac{\nabla v}{v}\frac{(\nabla v)^T}{v} - \frac{Hv}{v}\right).$$

```
function Hgx_u(gp :: GPPContext, x :: AbstractVector, fopt :: Float64)
    Copt = getCopt(gp)
    μ, gμ, Hμ = mean(gp, x), gx_mean(gp, x), Hx_mean(gp, x)
    v, gv, Hv = Copt*var(gp, x), Copt*gx_var(gp, x), Copt*Hx_var(gp, x)

    σ = sqrt(v)
    gμs, Hμs = gμ/σ, Hμ/σ
    gvs, Hvs = gv/v, Hv/v

    u = (μ-fopt)/σ
    gu = gμs - 0.5*u*gvs
    Hu = Hμs - 0.5*(gμs*gvs' + gvs*gμs') + 0.5*u*(1.5*gvs*gvs' - Hvs)
    u, gu, Hu
end
```

```
Hgx_u (generic function with 1 method)
```

Since we have several functions coming with a similar signature, we put together a common tester.

```
function check_derivs3_NLEI(f)
    Zk, y = test_setup2d((x,y)→x^2+y)
    gp = GPPContext(KernelSE{2}(0.5), 1e-8, Zk, y)
    z = [0.47; 0.47]
```

```
    dz = randn(2)
    fopt = -0.1
    g(s)  = f(gp, z+s*dz, fopt)[1]
    dg(s) = f(gp, z+s*dz, fopt)[2]
    Hg(s) = f(gp, z+s*dz, fopt)[3]
    check_fd(dg(0)'*dz, g, 0.0),
    check_fd(Hg(0)*dz, dg, 0.0)
end

check_derivs3_NLEI(Hgx_u)
```

```
(8.938688504606668e-9, 5.717345790763277e-9)
```

The derivatives of $\psi(u)$ are

$$\begin{aligned}\nabla\psi &= \psi'(u)\nabla u\\ H\psi &= \psi''(u)\nabla u(\nabla u)^T + \psi'(u)Hu.\end{aligned}$$

```
function Hgx_ψNLG(gp :: GPPContext, x :: AbstractVector, fopt :: Float64)
    u, gu, Hu = Hgx_u(gp, x, fopt)
    ψ, dψ, Hψ = DψNLG(u)
    ψ, dψ*gu, Hψ*gu*gu' + dψ*Hu
end

check_derivs3_NLEI(Hgx_ψNLG)
```

```
(5.365972459687776e-8, 2.2745601066644508e-9)
```

Getting to the actual acquisition function, we note that

$$\alpha_{NLEI} = -\log\alpha_{EI} = -\frac{1}{2}\log v + \psi_{NLG}(u)$$

and the relevant derivatives of the leading $-(\log v)/2$ are

$$\begin{aligned}-\frac{1}{2}\nabla(\log v) &= -\frac{1}{2}\frac{\nabla v}{v}\\ -\frac{1}{2}H(\log v) &= -\frac{1}{2}\frac{Hv}{v} + \frac{1}{2}\frac{\nabla v}{v}\frac{(\nabla v)^T}{v}.\end{aligned}$$

```
function Hgx_αNLEI0(gp :: GPPContext, x :: AbstractVector, fopt :: Float64)
    Copt = getCopt(gp)
    v, gv, Hv = Copt*var(gp, x), Copt*gx_var(gp, x), Copt*Hx_var(gp, x)
    gvs, Hvs = gv/v, Hv/v

    u, gu, Hu = Hgx_u(gp, x, fopt)
    ψ, dψ, Hψ = DψNLG(u)

    -0.5*log(v) + ψ,
    -0.5*gvs + dψ*gu,
    -0.5*Hvs + 0.5*gvs*gvs' + Hψ*gu*gu' + dψ*Hu
end

check_derivs3_NLEI(Hgx_αNLEI0)
```

```
(3.083860874139411e-9, 2.0784333121053338e-9)
```

And we finally put everything together by combining all the algebra above.

```
function Hgx_αNLEI(gp :: GPPContext, x :: AbstractVector, fopt :: Float64)
    Copt = getCopt(gp)
    μ, gμ, Hμ = mean(gp, x), gx_mean(gp, x), Hx_mean(gp, x)
    v, gv, Hv = Copt*var(gp, x), Copt*gx_var(gp, x), Copt*Hx_var(gp, x)

    σ = sqrt(v)
    gμs, Hμs = gμ/σ, Hμ/σ
    gvs, Hvs = gv/(2v), Hv/v

    u = (μ-fopt)/σ
    ψ, dψ, Hψ = DψNLG(u)

    α = -log(σ) + ψ
    dα = dψ*gμs - (1+u*dψ)*gvs
    Hα = (-0.5*(1.0+u*dψ)*Hvs + dψ*Hμs + Hψ*gμs*gμs' +
        (2.0 + u^2*Hψ + 3.0*u*dψ)*gvs*gvs' +
        -(u*Hψ+dψ)*(gμs*gvs' + gvs*gμs'))

    α, dα, Hα
end

check_derivs3_NLEI(Hgx_αNLEI)
```

```
(3.8643953944885975e-9, 2.1227938996316394e-9)
```

BO loop

The basis of our BO loop is repeated optimization of the acquisition function (in our case log-EI) over the domain. We use the interior-point Newton solver from `Optim.jl` as our basic optimizer. We are not being overly careful about performance, at least for now.

```
function optimize_EI(gp :: GPPContext, x0 :: AbstractVector,
                     lo :: AbstractVector, hi :: AbstractVector)
    fopt = minimum(gety(gp))
    fun(x) = Hgx_αNLEI(gp, x, fopt)[1]
    fun_grad!(g, x) = copyto!(g, Hgx_αNLEI(gp, x, fopt)[2])
    fun_hess!(H, x) = copyto!(H, Hgx_αNLEI(gp, x, fopt)[3])
    df = TwiceDifferentiable(fun, fun_grad!, fun_hess!, x0)
    dfc = TwiceDifferentiableConstraints(lo, hi)
    res = optimize(df, dfc, x0, IPNewton())
end
```

```
optimize_EI (generic function with 2 methods)
```

We will generally do a multi-start solver to find a good solution. For simplicity, we'll use random starts for the moment.

```
function optimize_EI(gp :: GPPContext,
                     lo :: AbstractVector, hi :: AbstractVector;
                     nstarts = 10, verbose=true)
    bestα = Inf
    bestx = [0.0; 0.0]
    for j = 1:10
        z = lo + (hi-lo).*rand(length(lo))
        res = optimize_EI(gp, z, [0.0; 0.0], [1.0; 1.0])
        if verbose
            println("From $z: $(Optim.minimum(res)) at $(Optim.minimizer(res))")
        end
        if Optim.minimum(res) < bestα
            bestα = Optim.minimum(res)
            bestx[:] = Optim.minimizer(res)
        end
    end
```

```
    bestα, bestx
end
```

```
optimize_EI (generic function with 2 methods)
```

Now we do a simple BO loop. We don't bother to tune the length scale or the noise variance in this case, but of course we automatically tune the diagonal variance.

```
let
    testf(x,y) = x^2+y
    Zk, y = test_setup2d(testf)
    gp = GPPContext(KernelSE{2}(0.8), 1e-8, 20)
    gp = add_points!(gp, Zk, y)
    for k = 1:5
        bestα, bestx = optimize_EI(gp, [0.0; 0.0], [1.0; 1.0], verbose=false)
        y = testf(bestx ... )
        gp = add_point!(gp, bestx, y)
        println("$k: EI=$(exp(-bestα)), f($bestx) = $y")
    end
    println("--- End loop ---")
    println("Best found: $(minimum(gety(gp)))")
end
```

```
1: EI=0.12228546386040667, f([0.021943927999207913, 1.5607661413303514e-15]) = 0.0004815359760359817
2: EI=1.1981439462590916e-28, f([0.9999999999999643, 0.9999999999999513]) = 1.9999999999998797
3: EI=8.099940723596572e-6, f([0.03725947303073533, 4.363225999352917e-15]) = 0.0013882683305324569
4:    EI=0.0006397672763878683,    f([2.6506574845390823e-13,    2.96839821419525e-
15]) = 2.96839821426551e-15
5:    EI=3.9206134547188126e-5,    f([2.674804839872237e-12,    1.8874900617772838e-
14]) = 1.8874900624927418e-14
--- End loop ---
Best found: 2.96839821426551e-15
```