Kernels and BO in Julia

David Bindel

2025-06-19

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

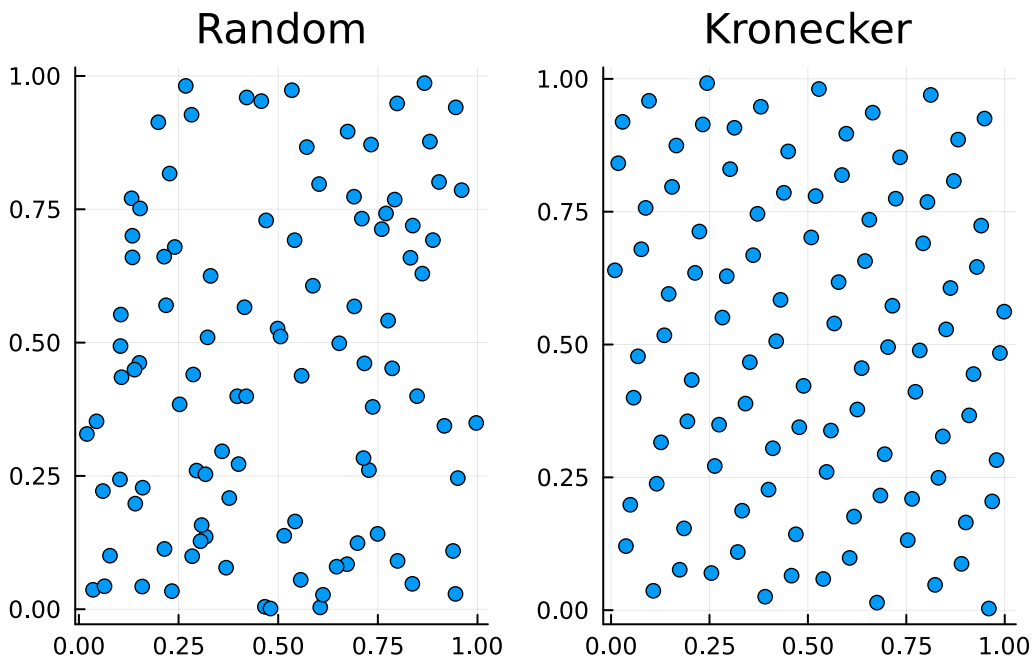
In general, we are interested in functions on a hypercube in $[0,1]^d$, though we can deal with more general regions. For many of us, our first inclination is either random sampling or sampling on a grid. However, choosing independent random samples tends to lead to "clumps" of points in some parts of 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")
      = 1.0+1.0/d
    for k = 1:10
        g  =  ^(d+1)- -1
        dg = (d+1)* ^d-1
          -= g /dg
    end
     s = [mod(1.0/ ^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)* s[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: $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))
```

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

Approximation with a kernel

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

```
_SE(s) = exp(-s^2/2)
_M1(s) = exp(-s)

_M3(s) = let t = sqrt(3)*s
    (1+t)*exp(-t)
end

_M5(s) = let t = sqrt(5)*s
    (1+t*(1+t/3))*exp(-t)
end

_IQ(s) = 1/(1+s^2)
_IM(s) = 1/sqrt(1+s^2)
_RQ(s; α=1.0) = (1+s^2)^-α
```

```
_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 D_SE(s)
      = exp(-s^2/2)
    d_div = -
    d  = d_div*s
    H  = (-1+s^2)*
     , d_div, d , H
end

function D_M1(s)
      = exp(-s)
     , -/s, - ,
end

function D_M3(s)
    t = √3*s
      = exp(-t)
      = (1+t)*
    d_div = -3*
    d  = d_div*s
    H  = 3*(t-1)*
     , d_div, d , H
```

```
end

function D_M5(s)
    t = √5*s
      = exp(-t)
      = (1+t*(1+t/3))*
    d_div = -5/3*(1+t)*
    d  = d_div*s
    H  = -5/3*(1+t*(1-t))*
     , d_div, d , H
end

function D_IQ(s)
      = 1/(1+s^2)
    d_div = -2* ^2
    d  = d_div*s
    H  = 2* ^2*(4* *s^2-1)
     , d_div, d , H
end

function D_IM(s)
      = 1/sqrt(1+s^2)
    d_div = - ^3
    d  = d_div*s
    H  =  ^3*(3*s^2* ^2-1)
     , d_div, d , H
end

function D_RQ(s;  =1.0)
      = (1+s^2)^-
    d_div = - *(1+s^2)^-( +1)*2
    d  = d_div*s
    H  = d_div +  *( +1)*(1+s^2)^-( +2)*4*s^2
     , d_div, d , H
end
```

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

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

```
function fd_check_D(name, D, s, h=1e-6, tol=1e-8; verbose=false, kwargs...)
    , d_div, d, H = D(s; kwargs...)
    p, dp_div, dp, Hp = D(s+h; kwargs...)
    m, dm_div, dm, Hm = D(s-h; kwargs...)
    relerr1 = abs( (d_div*s-d)/d )
    relerr2 = abs( (d-(p-m)/(2h))/d )
    relerr3 = abs( (H-(dp-dm)/(2h))/H )
    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_D("SE", D_SE, s)
    fd_check_D("M1", D_M1, s)
    fd_check_D("M3", D_M3, s)
    fd_check_D("M5", D_M5, s)
    fd_check_D("IQ", D_IQ, s)
    fd_check_D("IM", D_IM, s)
    fd_check_D("RQ", D_RQ, s; =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)
```

With the distance function in place, we can define kernels based on our radial basis functions.

```
k_SE(x, y;  =1.0) =  _SE(dist(x, y)/ )
k_M1(x, y;  =1.0) =  _M1(dist(x, y)/ )
k_M3(x, y;  =1.0) =  _M3(dist(x, y)/ )
k_M5(x, y;  =1.0) =  _M5(dist(x, y)/ )
k_IQ(x, y;  =1.0) =  _IQ(dist(x, y)/ )
k_IM(x, y;  =1.0) =  _IM(dist(x, y)/ )
k_RQ(x, y;  =1.0,  =1.0) =  _RQ(dist(x, y)/ ,  = )
```

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

Frequently there is also a constant multiplier in front. We will treat this case in the next section when we discuss hyperparameters.

Hyperparameter organization

We have written hyperparameters in terms of a named tuple in Julia, which can be passed into functions as keyword arguments. However, it is also useful to be able to think of a vector of hyperparameters that can be optimized. We will call the conversion of a named tuple into a vector *packing* the hyperparameters, and the reverse as *unpacking*.

```
function pack_hypers!(  :: Vector, named :: NamedTuple)
     [:] .= values(named)

end

pack_hypers!(  :: Vector; kwargs...) = pack_hypers!( , kwargs)
pack_hypers(named :: NamedTuple) = pack_hypers!(zeros(length(named)), named)
pack_hypers(; kwargs...) = pack_hypers(kwargs)
unpack_hypers(keys,  ) = NamedTuple{keys}( )
```

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

A demonstration is always useful.

```
let
    named = ( =1.0,  =2.0)
      = pack_hypers(named)
    println( )
    println(unpack_hypers((: , : ),  ))
end
```

```
[1.0, 2.0]
(  = 1.0,   = 2.0)
```

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*:

```
kernel_matrix0(kfun, X :: AbstractMatrix, Y :: AbstractMatrix; kwargs...) =
    [kfun(x,y; kwargs...) for x in eachcol(X), y in eachcol(Y)]
kernel_vector0(kfun, X :: AbstractMatrix, z :: AbstractVector; kwargs...) =
    [kfun(x,z; kwargs...) for x in eachcol(X)]
```

```
kernel_vector0 (generic function with 1 method)
```

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_matrix!(KXX :: AbstractMatrix, kfun,
                        X :: AbstractMatrix; kwargs...)
    for j = 1:size(X,2)
        xj = @view X[:,j]
        KXX[j,j] = kfun(xj, xj; kwargs...)
```

```
        for i = 1:j-1
            xi = @view X[:,i]
            kij = kfun(xi, xj; kwargs...)
            KXX[i,j] = kij
            KXX[j,i] = kij
        end
    end
    KXX
end

function kernel_vector!(KXz :: AbstractVector, kfun,
                        X :: AbstractMatrix,
                        z :: AbstractVector; kwargs...)
    for i = 1:size(X,2)
        xi = @view X[:,i]
        KXz[i] = kfun(xi, z; kwargs...)
    end
    KXz
end

function kernel_matrix!(KXY :: AbstractMatrix, kfun,
                        X :: AbstractMatrix,
                        Y :: AbstractMatrix; kwargs...)
    for j = 1:size(Y,2)
        yj = @view Y[:,j]
        for i = 1:size(X,2)
            xi = @view X[:,i]
            KXY[i,j] = kfun(xi, yj; kwargs...)
        end
    end
    KXY
end

kernel_matrix(kfun, X :: AbstractMatrix; kwargs...) =
    kernel_matrix!(zeros(size(X,2), size(X,2)), kfun, X; kwargs...)

kernel_vector(kfun, X :: AbstractMatrix, z :: AbstractVector; kwargs...) =
    kernel_vector!(zeros(size(X,2)), kfun, X, z; kwargs...)

kernel_matrix(kfun, X :: AbstractMatrix, Y :: AbstractMatrix; kwargs...) =
    kernel_matrix!(zeros(size(X,2), size(Y,2)), kfun, X, Y; kwargs...)
```

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

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

```
function kernel_matrix!(KXX :: AbstractMatrix, kfun,
                        X :: AbstractMatrix,
                        s :: Real; kwargs...)
    for j = 1:size(X,2)
        xj = @view X[:,j]
        KXX[j,j] = kfun(xj, xj; kwargs...) + s
        for i = 1:j-1
            xi = @view X[:,i]
            kij = kfun(xi, xj; kwargs...)
            KXX[i,j] = kij
            KXX[j,i] = kij
        end
    end
    KXX
end

kernel_matrix(kfun, X :: AbstractMatrix, s :: Real; kwargs...) =
    kernel_matrix!(zeros(size(X,2), size(X,2)), kfun, X, s; kwargs...)
```

```
kernel_matrix (generic function with 3 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)
    KXX1 = kernel_matrix0(k_SE, Zk, Zk)
    KXX2 = kernel_matrix(k_SE, Zk)
    KXX3 = kernel_matrix(k_SE, Zk, Zk)
    KXz1 = kernel_vector0(k_SE, Zk, Zk[:,1])
    KXz2 = kernel_vector(k_SE, 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)
    Ktemp = zeros(10,10)
    Kvtemp = zeros(10)
    Zk1 = Zk[:,1]
    KXX1 = @time kernel_matrix0(k_SE, Zk, Zk)
    KXz1 = @time kernel_vector0(k_SE, Zk, Zk1)
    KXX2 = @time kernel_matrix!(Ktemp, k_SE, Zk)
    KXX3 = @time kernel_matrix!(Ktemp, k_SE, Zk, Zk)
    KXz2 = @time kernel_vector!(Kvtemp, k_SE, Zk, Zk1)
    nothing
end
```

```
  0.000005 seconds (2 allocations: 944 bytes)
  0.000004 seconds (2 allocations: 144 bytes)
  0.000003 seconds
  0.000003 seconds
  0.000001 seconds
```

Covariances and Cholesky factors

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, kfun, X :: AbstractMatrix; kwargs...) =
    cholesky!(kernel_matrix!(KXX, kfun, X; kwargs...))

kernel_cholesky(kfun, X :: AbstractMatrix; kwargs...) =
    cholesky!(kernel_matrix(kfun, X; kwargs...))

kernel_cholesky!(KXX :: AbstractMatrix, kfun, X :: AbstractMatrix, s :: Real; kwargs...) =
    cholesky!(kernel_matrix!(KXX, kfun, X, s; kwargs...))

kernel_cholesky(kfun, X :: AbstractMatrix, s :: Real; kwargs...) =
    cholesky!(kernel_matrix(kfun, X, s; kwargs...))
```

```
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, kfun, X :: AbstractMatrix,
                 c :: AbstractVector, z :: AbstractVector; kwargs...)
    kXz = kernel_vector(kfun, X, z; kwargs...)
     z  = dot(kXz, c)
    rXz = ldiv!(KC.L, kXz)
     z  = sqrt(kfun(z,z; kwargs...)-rXz'*rXz)
     z, z
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)

    # Form kernel Cholesky and weights
```

```
    KC = kernel_cholesky(k_SE, Zk)
    c = KC\y

    # Evaluate true function and GP at a test point
    z = [0.456; 0.456]
    fz = testf(z...)
     z,  z = eval_GP(KC, k_SE, Zk, c, z)

    # Compare GP to true function
    zscore = (fz- z)/ z
    println("""
        True value:       $fz
        Posterior mean:   $ z
        Posterior stddev: $ z
        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!(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)
    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, k_SE, Zk)

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

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

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

    # Relative error in computed stddev
    abs( ( z-Kstorage[11,11])/ z )
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 $\log \eta$.

Choosing kernel hyperparameters

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 :: Vector, y :: Vector)
    n = length(c)
      = (dot(c,y) + n*log(2 ))/2
    for k = 1:n
          += log(KC.U[k,k])
    end

end
```

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

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,  A, ΔA, Δ A = randn(10,10), rand(10,10), rand(10,10), rand(10,10)

     inv(A, A) = -A\ A/A
    invA A, invAΔA, invAΔ A = A0\ A, A0\ΔA, A0\Δ A
    Δ invA = (invA A*invAΔA + invAΔA*invA A - invAΔ A)/A0

    check_fd( inv(A0, A), s->inv(A0+s* A), 0),
    check_fd(Δ invA, s-> inv(A0+s*ΔA,  A+s*Δ A), 0)
end
```

```
(6.323307292664657e-10, 1.8517648864636815e-10)
```

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,

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

and hence

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

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

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

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
     A, ΔA, Δ A = randn(10,10), randn(10,10), randn(10,10)

     logdet(A,  A) = tr(A\ A)
    Δ logdet(A,  A, ΔA, Δ A) = tr(A\Δ A)-tr((A\ΔA)*(A\ A))

    check_approx( logdet(A,  A), dot(inv(A'),  A)),
    check_fd( logdet(A, A), s->log(det(A+s* A)), 0),
    check_fd(Δ logdet(A, A,ΔA,Δ A), s-> logdet(A+s*ΔA, A+s*Δ A), 0)
end
```

```
(1.4597051528734925e-16, 1.9523614807889076e-10, 5.4667125412221495e-11)
```

Differentiating the kernel

To differentiate the negative log likelihood with respect to hyperparameters, we need to differentiate the kernel evaluations with respect to hyperparameters. We treat the diagonal variance and the noise variance hyperparameters as special cases, which means that for most of the kernels considered here, we will only really worry about derivatives with respect to length scale.

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

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

For this case, it's easy to write a macro to produce the relevant code.

```
"""
Produce
  d_k!(k, x, y; hypers...)
  H_k!(Hk, x, y; hypers...)
"""

macro d_rbf(D)
      = esc(:)
    :(function(g, x, y, c=1.0;  =1.0)
          s = dist(x,y)/$
           , d_div, d, H  = $D(s)
          g[1] -= c*d*s/$
          g
      end,
      function(H, x, y, c=1.0;  =1.0)
          s = dist(x,y)/$
           , d_div, d, H  = $D(s)
          H[1,1] += c*(H*s + 2*d)*s/$^2
          H
      end)
end

d_k_SE!, H_k_SE! = @d_rbf(D_SE)
```

```
(var"#63#67"(), var"#65#68"())
```

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

```
function d_kernel_matrix!(Ks, d_k!, X; kwargs...)
    n, n, d = size(Ks)
    for j = 1:n
```

```
        xj = @view X[:,j]
         Kjj = @view  Ks[j,j,:]
        d_k!( Kjj, xj, xj; kwargs...)
        for i = j+1:n
            xi = @view X[:,i]
             Kij = @view  Ks[i,j,:]
             Kji = @view  Ks[j,i,:]
            d_k!( Kij, xi, xj; kwargs...)
             Kji[:] .= Kij
        end
    end
     Ks
end

d_kernel_matrix(d_k!, X; kwargs...) =
    d_kernel_matrix!(zeros(size(X,2), size(X,2), length(kwargs)),
                        d_k!, X; kwargs...)
```

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

Per usual, a finite difference check is helpful.

```
let
    x = rand(2)
    y = rand(2)
      = 0.89
     _k_SE(x, y;  =1.0) = d_k_SE!(zeros(1), x, y,  = )[1]
    2_k_SE(x, y;  =1.0) = H_k_SE!(zeros(1,1), x, y,  = )[1]
    check_fd( _k_SE(x,y, = ),  ->k_SE(x,y, = ),  ),
    check_fd( 2_k_SE(x,y, = ),  -> _k_SE(x,y, = ),  )
end
```

```
(1.4710651399032097e-10, 6.007738005416477e-12)
```

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 _nll!(g, d_k!, X, invK, c; kwargs...)
    d, n = size(X)
    for j = 1:n
        xj = @view X[:,j]
        cj = c[j]
        d_k!(g, xj, xj, (invK[j,j]-cj*cj)/2; kwargs...)
        for i = j+1:n
            xi = @view X[:,i]
            ci = c[i]
            d_k!(g, xi, xj, (invK[i,j]-ci*cj); kwargs...)
```

```
        end
    end
    g
end

_nll(d_k!, X, invK, c; kwargs...) =
    _nll!(zeros(length(kwargs)), d_k!, X, invK, c; kwargs...)
```

```
_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  z_nll!(g, d_k!, X, invK, c, s=1e-10; kwargs...)
    _nll!(g, d_k!, X, invK, c; kwargs...)
    g[length(kwargs)+1] = (tr(invK) - c'*c)*s/2
    g
end

z_nll(d_k!, X, invK, c, s=1e-10; kwargs...) =
    z_nll!(zeros(length(kwargs)+1), d_k!, X, invK, c, s; kwargs...)
```

```
z_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,   = 1e-4, 1.0
    z=log(s)

    function k_SE_nll( ,z)
        KC = kernel_cholesky(k_SE, Zk, exp(z);  = )
        nll(KC, KC\y, y)
    end

    KC = kernel_cholesky(k_SE, Zk, s;  = )
    c = KC\y
    g =  z_nll(d_k_SE!, Zk, KC\I, c, s;  = )
```

```
    check_fd(g[1],  ->k_SE_nll( ,z),  ),
    check_fd(g[2], z->k_SE_nll( ,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!( K, KC :: Cholesky)
    ldiv!(KC.L,  K)
    rdiv!( K, KC.U)
     K
end

function whiten_matrices!( Ks, KC :: Cholesky)
    n, n, d = size( Ks)
    for k=1:d
        whiten_matrix!(view( Ks,:,:,k), KC)
    end
     Ks
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 H_nll(k, d_k!, H_k!, X, y, s=1e-10; kwargs...)
    d, n = size(X)
    n  = length(kwargs)

    # Factorization and initial solves
    KC = kernel_cholesky(k, X, s; kwargs...)
    invK = KC\I
    c̃ = KC.L\y
    c = KC.U\c̃
    invKc = KC\c
      = nll(KC, c, y)
     z_nll = (tr(invK)-(c'*c))*s/2

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

    # Add Hessian contribution from kernel second derivatives
     _nll!(H, H_k!, X, invK, c; kwargs...)
    H[n+1,n+1] =  z_nll

    # Set up whitened matrices  K and products  K*c and  K*c̃
     Ks = zeros(n, n, n+1)
    d_kernel_matrix!(Ks, d_k!, X; kwargs...)
    for j=1:n   Ks[j,j,n+1] = s  end
     Ks = whiten_matrices!(Ks, KC)
```

```
    Kc̃s = mul_slices!(zeros(n,n+1), Ks, c̃)
    Kr = reshape(Ks, n*n, n+1)

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

    # And put together gradient gradient
    for j=1:n
        g[j] = tr(view(Ks,:,:,j))/2
    end
    mul!(g, Kc̃s', c̃, -0.5, 1.0)
    g[end] = z_nll

    , g, H
end
```

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

As usual, we sanity check on a simple case.

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

    testf(,z) = H_nll(k_SE, d_k_SE!, H_k_SE!, Zk, y, exp(z); =)
    ref, gref, Href = testf(, z)

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

```
3.392925494606362e-8
```

Fast approximate Hessians

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

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

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

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

Per our custom, we code a short sanity check.

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

end

let
    Zk, y = test_setup2d((x,y) -> x^2 + y)

    # Form kernel Cholesky and weights
    KC = kernel_cholesky(k_SE, Zk)
    c̄ = KC\y
    Copt = (c̄'*y)/10

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

    check_approx(nll(KC, c, y), nllr(KC, c̄, y))
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  _nllr!(g, d_k!, X, invK, c, Copt; kwargs...)
    d, n = size(X)
    for j = 1:n
        xj = @view X[:,j]
        cj = c[j]
        cj_div_Copt = cj/Copt
        d_k!(g, xj, xj, (invK[j,j]-cj*cj_div_Copt)/2; kwargs...)
        for i = j+1:n
```

```
            xi = @view X[:,i]
            ci = c[i]
            d_k!(g, xi, xj, (invK[i,j]-ci*cj_div_Copt); kwargs...)
        end
    end
    g
end

function  z_nllr!(g, d_k!, X, invK, c, Copt, s=1e-10; kwargs...)
     _nllr!(g, d_k!, X, invK, c, Copt; kwargs...)
    g[length(kwargs)+1] = (tr(invK) - c'*c/Copt)*s/2
    g
end

 _nllr(d_k!, X, invK, c, Copt; kwargs...) =
     _nllr!(zeros(length(kwargs)), d_k!, X, invK, c, Copt, s; kwargs...)

 z_nllr(d_k!, X, invK, c, Copt, s=1e-10; kwargs...) =
     z_nllr!(zeros(length(kwargs)+1), d_k!, X, invK, c, Copt, s; kwargs...)
```

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

And our finite difference check:

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

    function k_SE_nllr( ,z)
        KC = kernel_cholesky(k_SE, Zk, exp(z);  = )
        nllr(KC, KC\y, y)
    end

    KC = kernel_cholesky(k_SE, Zk, s;  = )
    c = KC\y
    Copt = (c'*y)/length(c)
    g =  z_nllr(d_k_SE!, Zk, KC\I, c, Copt, s;  = )

    check_fd(g[1],  ->k_SE_nllr( ,z),  ),
    check_fd(g[2], z->k_SE_nllr( ,z), z)
end
```

```
(2.1800075879651277e-8, 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 H_nllr(k, d_k!, H_k!, X, y, s=1e-10; withz=true, kwargs...)
    d, n = size(X)
    n  = length(kwargs)

    # Factorization and initial solves
    KC = kernel_cholesky(k, X, s; kwargs...)
    invK = KC\I
    c̃ = KC.L\y
    c = KC.U\c̃
    Copt = (c'*y)/n
    invKc = KC\c
      = nllr(KC, c, y)
     z_nllr = (tr(invK)-(c'*c)/Copt)*s/2

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

```
    # Add Hessian contribution from kernel second derivatives
     _nllr!(H, H_k!, X, invK, c, Copt; kwargs...)
    if withz
        H[nt,nt] = z_nllr
    end

    # Set up matrices K
    Ks = zeros(n, n, nt)
    d_kernel_matrix!(Ks, d_k!, X; kwargs...)
    if withz
        for j=1:n
            Ks[j,j,nt] = s
        end
    end

    # Set up whitened matrices K and products K*c and K*č
    Ks = whiten_matrices!(Ks, KC)
    Kčs = mul_slices!(zeros(n,nt), Ks, č)
    Kr = reshape(Ks, n*n, nt)

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

    # Last term of the Hessian written via data fidelity part of gradient
    mul!(g, Kčs', č, -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:n
        g[j] += tr(view(Ks,:,:,j))/2
    end
    if withz
        g[end] = z_nllr
    end

    , g, H
end
```

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

And the sanity check on a simple case.

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

    testf( ,z) = H_nllr(k_SE, d_k_SE!, H_k_SE!, Zk, y, exp(z);  = )
     ref, gref, Href = testf( , z)

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

```
2.2745899020429444e-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

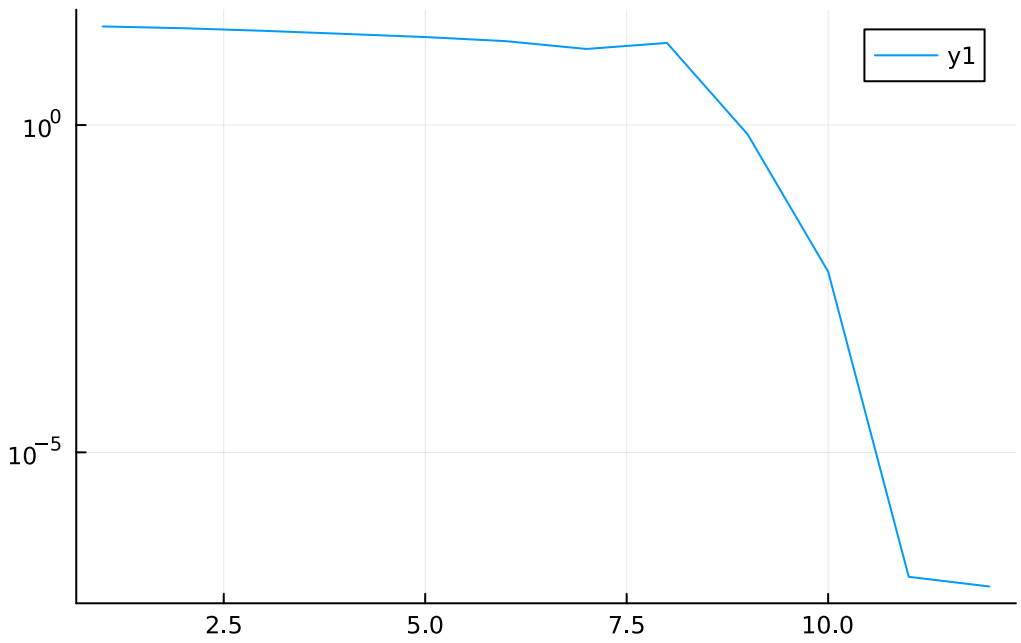
We are now in a position to tune the hyperparameters for a GP example. The problematic bit, which we will deal with in the next section, is the noise variance. Once we have a reasonable estimate for the optimized noise variance, we can get the expected quadratic convergence of Newton iteration. But the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

```
let
    Zk, y = test_setup2d((x,y) -> x^2 + cos(3*y) + 5e-4*cos(100*y), 40)
     , s = 0.7, 1e-4
    normgs = []
```

```
    for j = 1:12
        ¯, ¯, H¯ = H_nllr(k_SE, d_k_SE!, H_k_SE!, Zk, y, s; =)
        u = -H¯\¯
        cut = " "
        if abs(u[2]) > 3
            u /= abs(u[2])
            cut = "*"
        end
        println("$cut ($ ,$s)\t$¯\t$(norm(¯))")
        push!(normgs, norm(¯))
          += u[1]
        s *= exp(u[2])
    end
    plot(normgs, yscale=:log10)
end
```

```
* (0.7,0.0001)  -100.34663467307195 32.14208440052675
* (0.716618084813006,3.678794411714424e-5)  -109.67657182216773 30.14294144303889
* (0.7350441476167393,1.3533528323661272e-5)    -118.45849876322804 27.532576673943836
* (0.7596060755215911,4.978706836786396e-6) -126.61847467803248 24.70449908720823
* (0.7908303320476653,1.8315638888734185e-6)    -134.14606660370762 22.116543349252325
* (0.8269166426646524,6.737946999085469e-7) -140.9679401592967  19.15081161206308
  (0.8666074878980258,2.4787521766663593e-7)    -146.7145722752776  14.537648302653217
  (0.9927687109633535,1.3710075562054298e-8)    -150.9082508976127  17.97744189819667
  (0.9605799688847484,3.438539388188092e-8) -152.11354290436995 0.7242513547247702
  (0.9671531939160823,3.20979825966728e-8)  -152.12017017479076 0.00576881103957636
  (0.9671939952552309,3.208561080777917e-8) -152.1201704484965  1.2549349007530658e-7
  (0.9671939987380738,3.208560933246368e-8) -152.1201704000326  8.93192843347194e-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)
         k = LinearAlgebra.reflector!(x)
        LinearAlgebra.reflectorApply!(x,  k, view(y, k+1:n))
        LinearAlgebra.reflectorApply!(x,  k, view(K, k+1:n, k+1:n))
        LinearAlgebra.reflectorApply!(x,  k, 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
    ℓ = n/2*(log(cTy) + log(2π) + 1 - log(n))
    for j = 1:n
        ℓ += log(alpha[j])
    end

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

    ℓ, dℓ
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)
      = 1e-3

    # Ordinary reduced NLL computation (full and pieces)
    K = kernel_matrix(k_SE, Zk)
```

```
    KC = cholesky(K+ *I)
    c = KC\y
    data1 = c'*y
    logdet1 = sum(log.(diag(KC.U)))
    trinv1 = tr(KC\I)
    ¯1 = nllr(KC, c, y)

    # Reduced NLL computation
    tridiag_reduce!(K, y)
    alpha, beta = tridiag_params(K,  )
    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)
    ¯2, d¯ = nllrT!(K, y,  , alpha, beta, c)

    max(check_approx(data1, data2),
        check_approx(logdet1, logdet2),
        check_approx(trinv1, trinv2),
        check_approx(¯1, ¯2))
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)
      = 1e-3
    K = kernel_matrix(k_SE, Zk)
    tridiag_reduce!(K, y)
    check_fd(nllrT(K, y,  )[2],  ->nllrT(K, y,  )[1],  )
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

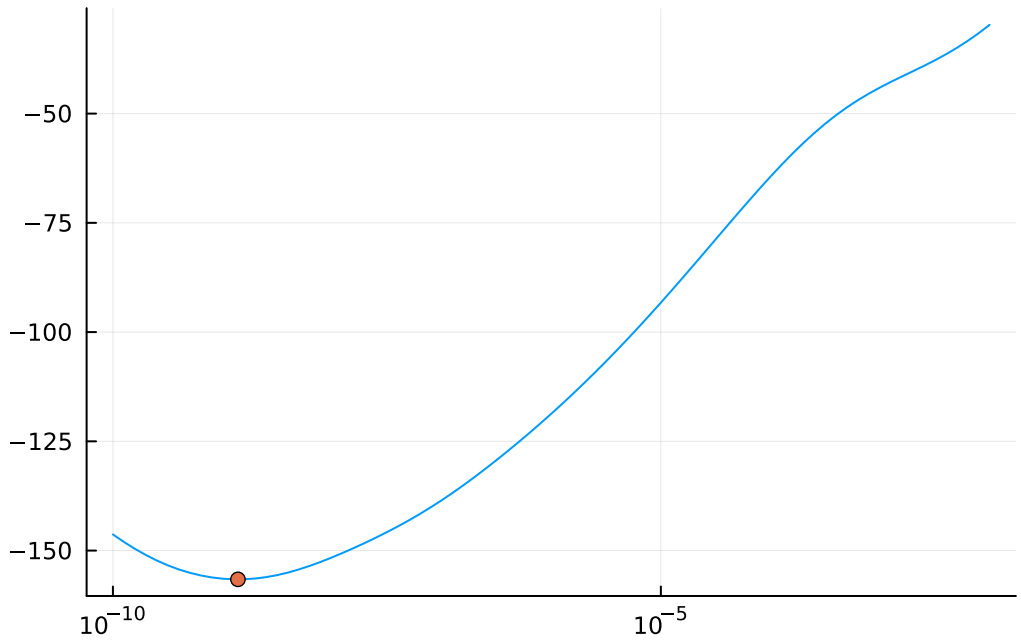
doing evaluations. Therefore, we use an optimizer that starts with a brute force grid search (in $\log \eta$) to find a bracketing interval for a best guess at the global minimum over the range, and then does a few steps of secant iteration to refine the result.

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

    # Sample on an initial grid
    logmin = log( min)
    logmax = log( max)
    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 1 method)
```

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)
      = 1.2

    alpha = zeros(n)
    beta = zeros(n)
    c = zeros(n)
    ỹ = copy(y)
    K = kernel_matrix(k_SE, Zk;  = )
    tridiag_reduce!(K, ỹ)

    ss = 10.0.^(-10:0.1:-2)
    nllss = [nllrT!(K,ỹ,s,alpha,beta,c) for s in ss]
    ss    = [t[1] for t in nllss]
     ss = [t[2] for t in nllss]

     , , d = min_nllrT!(K,ỹ,1e-10,1e-2,alpha,beta,c)
    plot(ss, ss, xscale=:log10, legend=:false)
    plot!([ ], [], marker=:circle)
end
```



Putting it together

While it is clearly possible to have more, most kernels have three hyperparameters: a length scale, a noise variance, and a diagonal variance. We have seen that the diagonal variance can be treated analytically and the (log) noise variance can be treated on its own in a specialized way. This suggests that, at least initially, we might consider a few steps (block) descent iteration in which we alternate between optimizing the length scale and the noise variance. This has the advantage that if we start with a small noise variance term, we are less likely to blunder into a solution in which most of the function behavior is explained as noise. After a few steps of alternating iteration, we can run a few Newton steps to polish up the result, if desired.

```
let
    n = 40
    Zk, y = test_setup2d((x,y) -> x^2 + cos(3*y) + 1e-3*cos(100*x), n)

     =0.5
     =1e-10
    alpha = zeros(n)
    beta = zeros(n)
    c = zeros(n)
    ỹ = copy(y)
    K = zeros(n,n)

    H _SE_nllr( ,  ; withz=false) =
        H _nllr(k_SE, d _k_SE!, H _k_SE!, Zk, y,  ; withz=withz,  = )

    println("--- Start with alternating iteration ---")
    for j = 1:3

        # First adjust the length scale
        g = [0.0]
        for k=1:4
            _, g, H = H _SE_nllr( , )
            u = -(H\g)
              += u[1]
        end

        # Now adjust the noise variance
        kernel_matrix!(K, k_SE, Zk;  = )
        ỹ[:] .= y
        tridiag_reduce!(K, ỹ)
         ,  ,  _  = min_nllrT!(K,ỹ,1e-10,1e-2,alpha,beta,c)
        println("$ \t$ \t$ \t$g\t$ _ ")
```

```
    end

    println("--- Clean up with Newton ---")
    for j = 1:5
         , g, H = H_SE_nllr( , ,withz=true)
        println("$ \t$ \t$ \t$(norm(g))")
        u = -H\g
          += u[1]
          *= exp(u[2])
    end
end
```

```
--- Start with alternating iteration ---
0.4990010685358129  5.384614696373881e-8    -128.0618257870823  [-1.1195050490186986e-5]
0.8489487681511495  6.843489317626243e-8    -145.55760737685605 [-4.003335918351439]    0.01
0.8871649033563495  6.701021043029197e-8    -145.60129738547732 [0.016427827574574394]  -0.0
--- Clean up with Newton ---
0.8871649033563495  6.701021043029197e-8    -145.60129740450816 0.08082515251148191
0.8883042921951915  6.689606200599118e-8    -145.6013431238578  0.0009468013826906623
0.888293068967716   6.689497162497633e-8    -145.6013431249981  2.0230619853422285e-7
0.888293066161236   6.689497192964216e-8    -145.60134313777732 2.8740288764465145e-8
0.8882930668127574  6.68949710935136e-8 -145.60134312463015 4.8949977407923485e-8
```

Spatial derivatives

In order to do gradient-based search for optima of objective functions in Bayesian optimization and the like, we need to be able to compute gradients and Hessians of the pointwise predictive mean and variance. This requires computing gradients and Hessians for the kernel, and assembling those gradients and Hessians into a coherent whole.

Radial basis function gradients and Hessians

For 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

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

These functions are clearly well-behaved away from $r = 0$, but we can run into difficulties at $r = 0$. In particular, in order to have differentiability at $\rho = 0$, we need $\phi'(\rho)/\rho$ to approach a finite value, and in order to have second derivatives we need $\phi'(\rho)/\rho$ to approach $\phi''(0)$.

Note that if $\phi(\rho) = \phi_{\text{ref}}(s)$, with $s = \rho/\ell$, then

$$\begin{aligned} \phi'(\rho) &= \phi'_{\text{ref}}(s) \ell^{-1}, \\ \phi'(\rho)/\rho &= (\phi'_{\text{ref}}(s)/s) \ell^{-2}, \\ \phi''(\rho) &= \phi''_{\text{ref}}(s) \ell^{-2}. \end{aligned}$$

Because we will frequently accumulate gradients and Hessians, we write mutating versions of the computations that add a multiple of a gradient or Hessian into an accumulator argument. Because this code looks the same for all of our radially symmetric kernels, we write a macro to produce it.

```
macro rbf_derivs(D )
     =esc(: )
    :(function(g, x, y, c=1.0;  =1.0, kwargs...)
            = dist(x,y)
          s =  /$
          _, d _div, d , _ = $D (s; kwargs...)
          if   != 0.0
              d  /= $
              C = c*d /
              for i = 1:length(x)
                  g[i] += C*(x[i]-y[i])
              end
          end
          g
```

```
        end,
        function(H, x, y, c=1.0;  =1.0, kwargs...)
              = dist(x,y)
            s =  /$
            _, d_div, _, H  = $D(s; kwargs...)
            H  /= $^2
            d_div /= $^2
            for j = 1:length(x)
                H[j,j] += c*d_div
            end
            if   != 0.0
                C = c*(H -d_div)/ ^2
                for j = 1:length(x)
                    xj, yj = x[j], y[j]
                    for i=1:length(x)
                        xi, yi = x[i], y[i]
                        H[i,j] += C*(xj-yj)*(xi-yi)
                    end
                end
            end
            H
        end)
end

k_SE!, Hk_SE! = @rbf_derivs(D_SE)
```

```
(var"#159#163"(), var"#161#164"())
```

And, per usual, we use a finite difference check to test correctness.

```
let
    x = rand(2)
    y = rand(2)
    dx = rand(2)
     =0.8

     k_SE(x, y;  =1.0) =  k_SE!(0*x, x, y;  = )
    Hk_SE(x, y;  =1.0) = Hk_SE!(zeros(length(x),length(x)), x, y;  = )

    check_fd(dx'* k_SE(x, y;  = ), s->k_SE(x+s*dx,y;  = ), 0.0),
    check_fd(dx'*Hk_SE(x, x;  = )*dx,  s->dx'* k_SE(x+s*dx,x;  = ), 0.0),
```

```
    check_fd(dx'*Hk_SE(x, y;  = )*dx,  s->dx'* k_SE(x+s*dx,y;  = ), 0.0)
end
```

```
(1.9919586300193723e-11, 7.34321340422864e-12, 7.576284508092777e-11)
```

Derivatives of predictive means and variances