Analytical Version

Maxime Rischard

February 19, 2016

In [ ]:
```
using Distributions
using PDMats
using Optim
using StatsFuns: log2π
import PyPlot; plt=PyPlot
using DualNumbers
using Gadfly
using ToeplitzMatrices
plt.svg(true)
using Mamba
using GraphViz
using ProfileView
using GaussianProcesses
using ProfileView
;
```

In [ ]:
```
# function sqexp{S<:Number,T<:Number}(σ_f::S, L::T)
#     function k{T<:Number}(x::T, xprime::T)
#         return σ_f^2 * exp( - (x-xprime)^2 / (2*L^2))
#     end
#     return k
# end

function kernel{T<:Number,Kern<:Kernel}(kern::Kern, x::AbstractVector{T})
    n = length(x)
    Xmat = reshape(x, (1,n))
    Σ = GaussianProcesses.crossKern(Xmat, kern)
    return Σ
end
```

Sample 200 points uniformly on either side of the discontinuity at $x = 5$.

$$X_1 \sim \text{Uniform}\,(0,5) \tag{1}$$
$$X_2 \sim \text{Uniform}\,(5,10) \tag{2}$$
$$\tag{3}$$

In [ ]:
```
_X1distr = Uniform(0,5)
_X2distr = Uniform(5,10)
_n_per_region = 200
_n = _n_per_region*2
_thresh = 5.0
_X = sort(Float64[rand(_X1distr, _n_per_region); rand(_X2distr, _n_per_region)])
;
```

We want to simulate a GP; we'll simulate from the model

$$y = \tau \, \mathbf{I}\{x > 5\} + X\beta + \epsilon$$

X will be the locations, and y will be the response surface. The shift term $\tau \, \mathbf{I}\{x > 5\}$ will be zero to the left of the boundary, and a constant to the right of the boundary. ϵ is from a GP with a squared exponential covariance with additional iid normal noise with variance σ_y^2.

$$K(x, x') = \operatorname{cov}(y, y') = \sigma_y^2 \delta(x - x') + \sigma_{GP}^2 \exp\left(-\frac{(x - x')}{2L^2}\right)$$

Set the data-generating parameters to the following (arbitrary) values:

$$\sigma_f^2 = 1 \tag{4}$$
$$L^\star = 1.05 \tag{5}$$
$$\tau^\star = 0.75 \tag{6}$$
$$\beta^\star = 1.0 \tag{7}$$
$$\sigma_y^{2\star} = 0.01 \tag{8}$$
$$\tag{9}$$

```
In [ ]: # data-generating parameters
        _σf2_star = 1.0
        _Lstar  = 1.05
        _τstar = 0.75
        _βstar = 1.0
        _σy2_star = 0.01
        _region = _X.>_thresh
        ;
```

We are fitting a Bayesian model with normal priors on β and τ. Let's start by giving them very diffuse normal priors. The normality allows us to use all the tools of multivariate normal theory.

```
In [ ]: function simulate1{T<:Number,S<:Number,Ker<:Kernel}(β::T, τ::T, kern::Ker, σy2::T, thresh::Real
            region = X.>thresh
            n = length(X)
            K = kernel(kern, X) + 1e-5*eye(n)
            fXdistr = MvNormal(zeros(n), K)
            fX = rand(fXdistr)
            In=eye(Diagonal{Float64},n)
            Z_Ydistr = Normal(0,√(σy2))
            Y = fX .+ β*X + τ*region .+ rand(Z_Ydistr, n)
            return Y
        end
```

```
In [ ]: _σβ2 = 1000.0 # extremely diffuse normal prior on β
        _στ2 = 1000.0 # extremely diffuse normal prior on τ
        _kern = SE(log(_Lstar), log(√_σf2_star))
        _II = _region * _region'
        _Y = simulate1(_βstar, _τstar, _kern, _σy2_star, _thresh, _X)
        plt.plot(_X, _Y, "o")
        plt.title("Simulated Dataset")
        plt.xlabel("X")
        plt.ylabel("Y")
        ;
```

Conditionally on σ_y^2, σ_f^2 and L, the posterior of τ (i.e. $\tau \mid Y, \sigma_y^2, \sigma_f^2, L$) can be obtain analytically. We have:

$$
\begin{aligned}
\tau &\sim \mathcal{N}\left(0, \sigma_\tau^2\right) && (10)\\
\beta &\sim \mathcal{N}\left(0, \sigma_\beta^2\right) && (11)\\
Y &= \tau\, \mathbf{I}\left\{x > 5\right\} + X\beta + \underbrace{GP\left(0, K\right)}_{f} + \epsilon_{iid} && (12)\\
\operatorname{cov}\left(Y, \tau\right) &= \sigma_\tau \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ \vdots \\ 1 \end{pmatrix} && (13)\\
\operatorname{cov}\left(Y, \beta\right) &= \sigma_\beta X && (14)\\
\operatorname{cov}\left(Y, \tau\right)\operatorname{cov}\left(Y, \tau\right)^T &= \sigma_\tau^2\, \mathbf{I}\left\{x > 5\right\}\mathbf{I}\left\{x > 5\right\}^T = \sigma_\tau^2 \begin{bmatrix} 0 & 0 & \cdots 0 & 0 \\ 0 & 0 & \cdots 0 & 0 \\ \vdots & \vdots & \ddots \vdots & \vdots \\ 0 & 0 & \cdots 1 & 1 \\ 0 & 0 & \cdots 1 & 1 \end{bmatrix} && (15)\\
\operatorname{cov}\left(Y, \beta\right)\operatorname{cov}\left(Y, \beta\right)^t &= \sigma_\beta^2 X X^T && (16)\\
\operatorname{var}\left(Y\right) &= \operatorname{cov}\left(Y, \tau\right)\operatorname{cov}\left(Y, \tau\right)^T + \operatorname{cov}\left(Y, \beta\right)\operatorname{cov}\left(Y, \beta\right)^T + K + \sigma_y^2 I && (17)\\
Y &\sim \mathcal{N}\left(0, \operatorname{var}\left(Y\right)\right) && (18)\\
& && (19)
\end{aligned}
$$

$$
\begin{pmatrix} Y \\ f \\ \tau \\ \beta \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{bmatrix} \operatorname{var}\left(Y\right) & K & \operatorname{cov}\left(Y, \tau\right) & \operatorname{cov}\left(Y, \beta\right) \\ K & K & 0 & 0 \\ \operatorname{cov}\left(Y, \tau\right)^T & 0 & \sigma_\tau^2 & 0 \\ \operatorname{cov}\left(Y, \beta\right)^T & 0 & 0 & \sigma_\beta^2 \end{bmatrix}\right)
$$

From this we can obtain the posterior on τ:

$$
\tau \mid Y \sim \mathcal{N}\left(\operatorname{cov}\left(Y, \tau\right)^T \operatorname{var}\left(Y\right)^{-1} Y, \sigma_\tau^2 - \operatorname{cov}\left(Y, \tau\right)^T \operatorname{var}\left(Y\right)^{-1}\operatorname{cov}\left(Y, \tau\right)\right)
$$

```
In [ ]: _XXt = _X*_X'
        # marginal variance of Y
        _VarY = kernel(_kern, _X) + _στ2*_II + _σβ2*_XXt + _σy2_star*eye(Diagonal{Float64},_n)
        _YDistr = MvNormal(Distributions.ZeroVector(Float64, _n), PDMat(_VarY))
        _VarYinv = inv(_VarY)
        _Eτ_Y =(_στ2 * _region' * _VarYinv * _Y)[1]
        _Vτ_Y = (_στ2 - _στ2^2 * _region' * _VarYinv * _region)[1]
        _τ_Y = Normal(_Eτ_Y, √(_Vτ_Y))
```

```
In [ ]: function mvnorm_logpdf{T<:Real, S<:Number}(Σ::Array{S,2}, X::Vector{T})
            Σchol = cholfact(Σ)
            c0 = -0.5 * (length(X) * Float64(log2π) + logdet(Σchol))
            sqmahal = dot(X, inv(Σchol) * X)
            return c0 - 0.5*sqmahal
        end
```

We can also get the marginal likelihood of $\mathbb{P}\left(Y \mid \sigma_y^2, \sigma_f^2, L\right)$

```
In [ ]: mvnorm_logpdf(_VarY, _Y),logpdf(_YDistr, _Y)
```

```
In [ ]: @time mvnorm_logpdf(_VarY, _Y);
```

```
In [ ]: @time YDistr = MvNormal(Distributions.ZeroVector(Float64, _n), PDMat(_VarY));logpdf(_YDistr, _Y
```

Maybe not such a great idea.

1 Partial Bayes

These computations are pretty quick, so we can optimize over $\sigma_y^2, \sigma_f^2, L$ (is this partial Bayes?).

```
In [ ]: """
        Marginal variance of Y for fixed `στ2`, `σβ2` and `L`
        """
        function g_VarY{T<:Number, S<:Number}(X::Vector{T}, XXt::Symmetric{T}, thresh::T, στ2::T, σβ2::
            n = length(X)
            kern = SE(log(L), log(√σf2))
            In=eye(Diagonal{Float64},n)
            region = X.>thresh
            II = region * region'
            K = kernel(kern, X)
            VarY = K + στ2*II + σβ2*XXt + σy2*In + 1e-5*eye(n)
            return VarY
        end

        function gen_optim_target{T<:Real}(X::Vector{T}, Y::Vector{T}, thresh::T, στ2::T, σβ2::T)
            XXt = Symmetric(X * X') # = BLAS.syrk('U', 'N', 1.0, X)
            function optim_target{T<:Number}(logx::Vector{T})
                σy2, σf2, L = exp(logx)
                VarY = g_VarY(X, XXt, thresh, στ2, σβ2, σy2, σf2, L)
                loglik = mvnorm_logpdf(VarY,Y)
                return -loglik
            end
        end

        optim_t = gen_optim_target(_X, _Y, 5.0, _στ2, _σβ2)
```

```
In [ ]: isapprox(g_VarY(_X, Symmetric(_XXt), 5.0, _στ2, _σβ2, _σy2_star, _σf2_star, _Lstar), _VarY)
```

```
In [ ]: @time _partial_opt=optimize(optim_t, [0.0, 0.0, 0.0], method=:l_bfgs; ftol=0.01, show_every=tru
```

It takes between 1 and 2 minutes with 200 points on each side of the discontinuity! And it shouldn't be any slower to use the same algorithm in higher dimensions.

```
In [ ]: function τ_posterior{T<:Real, S<:Number}(X::Vector{Float64}, Y::Vector{Float64}, στ2::T, σβ2::T
            XXt = Symmetric(X * X')
            VarY = g_VarY(X, XXt, thresh, στ2, σβ2, σy2, σf2, L)
            VarYinv = inv(VarY)
            region = X.>thresh
            II = region * region' # we're doing this twice. expensive?
            Eτ_Y =(στ2 * region' * VarYinv * Y)[1]
            Vτ_Y = (στ2 - στ2^2 * region' * VarYinv * region)[1]
```

```
    τ_Y = Normal(Eτ_Y, √(Vτ_Y))
    return τ_Y
end
;
```

Using these point estimates of the covariance parameters, and then computing the posterior on τ, we get a good estimate of τ (which hints that our estimator is consistent).

```
In [ ]: _σy2_PB, _σf2_PB, _L_PB = exp(_partial_opt.minimum)
        τ_posterior(_X, _Y, _στ2, _σβ2, _σy2_PB, _σf2_PB, _L_PB, 5.0)
```

2 Slice Sampling

```
In [ ]: _σf2_star, _σf2_PB
```

```
In [ ]: _σf2_prior = Truncated(Cauchy(0.0, 10.0), 0.0, Inf)
        _σy2_prior = Truncated(Cauchy(0.0, 10.0), 0.0, Inf)
        _L_prior   = Truncated(Cauchy(0.0, 10.0), 0.0, Inf)
        _xx = linspace(0.0,10.0,1000)
        plt.plot(_xx, pdf(_σf2_prior, _xx))
```

```
In [ ]: _model = Model(
            σf2 = Stochastic(
                () -> Truncated(Cauchy(0.0, 10.0), 0.0, Inf),
                true
            ),
            σy2 = Stochastic(
                () -> Truncated(Cauchy(0.0, 10.0), 0.0, Inf),
                true
            ),
            L = Stochastic(
                () -> Truncated(Cauchy(0.0, 10.0), 0.0, Inf),
                true
            ),
            Y = Stochastic(1,
                (X, XXt, thresh, στ2, σβ2, σf2, σy2, L) -> MvNormal(
                    Distributions.ZeroVector(Float64, n),
                    PDMat(
                        g_VarY(X, XXt, thresh, στ2, σβ2, σy2, σf2, L)
                    )
                ),
                false
            ),
        )
```

```
In [ ]: Graph(graph2dot(_model))
```

```
In [ ]: _nuts_scheme = [NUTS([:σf2, :σy2, :L])]
        _slice_scheme = [Slice([:σf2, :σy2, :L], 1.0)]
        _gibbsslice_scheme = [Slice([:σf2], 1.0), Slice([:σy2], 1.0), Slice([:L], 1.0)]
        _data = Dict{Symbol, Any}(
            :X => _X,
            :XXt => Symmetric(_X * _X'),
            :Y => _Y,
            :thresh => 5.0,
```

```
    :στ2 => _στ2,
    :σβ2 => _σβ2,
)
_pr = Truncated(Cauchy(0.0, 10.0), 0.0, Inf)
## Initial Values
inits = [
  Dict{Symbol, Any}(
    :Y => _data[:Y],
    :σy2 => rand(_pr),
    :σf2 => rand(_pr),
    :L => rand(_pr),
  )
for i in 1:3
]
;
```

In [ ]:
```
setsamplers!(model, gibbsslice_scheme)
sim1 = mcmc(model, data, inits, 1000, burnin=100, thin=2, chains=3)
```

In [ ]:
```
plot(sim1)[1]
```

In [ ]:
```
plot(sim1)[2]
```

In [ ]:
```
plot(sim1)[3]
```

In [ ]:
```
gelmandiag(sim1)
```

Not converging at all!

3 Toeplitz Structure

Instead of sampling X uniformly from 0 to 10, we will put X on a grid. That way, the Gaussian Process' covariance matrix is Toeplitz, and efficient algorithms exist to invert it.

In [ ]:
```
function kernel{T<:Number,Kern<:GaussianProcesses.Stationary}(kern::Kern, x::LinSpace{T})
    vc = GaussianProcesses.crossKern(x', reshape([0.0,],(1,1)), kern)
    K_ST=SymmetricToeplitz(vec(vc))
    return K_ST
end
```

In [ ]:
```
_Xgrid = linspace(0,10,_n_per_region*2)
_Ygrid = simulate1(_βstar, _τstar, _kern, _σy2_star, _thresh, _Xgrid)
plt.plot(_Xgrid, _Ygrid, ".")
plt.xlabel("X")
plt.ylabel("Y")
;
```

In [ ]:
```
_K_ST = kernel(_kern, _Xgrid)
_K = kernel(_kern, collect(_Xgrid))
;
```

In [ ]:
```
@assert isapprox(_K, _K_ST, atol=1e-4, norm=Base.LinAlg.vecnormInf)
Base.LinAlg.vecnormInf(_K-_K_ST) # maximum distance
```

In [ ]:
```
typeof(_K_ST)
```

```
In [ ]: inv(_K)
        @time inv(_K);
```

```
In [ ]: inv(_K_ST)
        @time inv(_K_ST);
```

Inversion using the Toeplitz structure is about ten times faster. The advantage goes up as the matrix gets larger.

```
In [ ]: function sherman_morrison{T<:Number}(Ainv::AbstractMatrix{T}, u::AbstractVector{T}, v::Abstract
            vT = v'
            α::T = one(T) / (one(T)+(dot(v,Ainv*u)))
            return Ainv - α.*(Ainv*u)*(vT*Ainv)
        end
```

```
In [ ]: # quick test
        @assert isapprox(inv(_K+eye(_n)+3.0*_X*_X'), sherman_morrison(inv(_K+eye(_n)), 3.0*_X,_X))
```

```
In [ ]: # function g_VarY{T<:Number, S<:Number}(X::Vector{T}, XXt::Symmetric{T}, thresh::T, στ2::T, σβ2
        #     n = length(X)
        #     kern = SE(log(L), log(√σf2))
        #     In=eye(Diagonal{Float64},n)
        #     region = X.>thresh
        #     II = region * region'
        #     K = kernel(kern, X)
        #     VarY = K + στ2*II + σβ2*XXt + σy2*In + 1e-5*eye(n)
        #     return VarY
        # end

        function g_VarYInv{T<:Number}(ΣGP::SymmetricToeplitz{T}, X::AbstractVector{T}, thresh::T, στ2::
            vc = copy(ΣGP.vc)
            vc[1] += σy2+1e-5
            ΣGP_obs = SymmetricToeplitz(vc)
            ΣGP_inv = inv(ΣGP_obs)
        #     ΣGP_inv = inv(full(ΣGP)+eye(n)*σy2)
            ΣGPβ_inv = sherman_morrison(ΣGP_inv, σβ2*X, X)
            region = 1.0.*(X.>thresh) # maybe converting a boolean to a float is stupid
            ΣGPβτ_inv = sherman_morrison(ΣGPβ_inv, στ2*region, region)
            ΣGPβτμ_inv = ΣGPβτ_inv
        #     ΣGPβτμ_inv = sherman_morrison(ΣGPβ_inv, σμ2*ones(T,n), ones(T,n))
            return ΣGPβτμ_inv
        end
```

```
In [ ]: g_VarYInv(_K_ST, collect(_Xgrid), _thresh, 2.0, 10.0, 1.5);
        @time g_VarYInv(_K_ST, collect(_Xgrid), _thresh, 2.0, 10.0, 1.5);
```

```
In [ ]: inv(g_VarY(collect(_Xgrid), Symmetric(_Xgrid*_Xgrid'), _thresh, _στ2, _σβ2, 1.5, _σf2_star, _Ls
        @time inv(g_VarY(collect(_Xgrid), Symmetric(_Xgrid*_Xgrid'), _thresh, _στ2, _σβ2, 1.5, _σf2_sta
```

```
In [ ]: # checking that the inverse is correct
        @assert isapprox(g_VarYInv(_K_ST, collect(_Xgrid), _thresh, 2.0, 10.0, 1.5)*g_VarY(collect(_Xgr
```

That's not great. It seems to be doing the right thing, but with large numerical error.

3.1 Optimisation using Toeplitz

In [ ]:
```
function mvnorm_logpdf{T<:Real, S<:Number}(Σ::AbstractMatrix{S}, Σinv::AbstractMatrix{S}, X::Ve
    c0 = -0.5 * (length(X) * Float64(log2π) + logdet(Σ))
    sqmahal = dot(X, Σinv * X)
    return c0 - 0.5*sqmahal
end

function gen_optim_target{T<:Real}(X::LinSpace{T}, Y::Vector{T}, thresh::T, στ2::T, σβ2::T)
    XXt = Symmetric(X * X') # = BLAS.syrk('U', 'N', 1.0, X)
    function optim_target{T<:Number}(logx::Vector{T})
        σy2, σf2, L = exp(logx)
        kern = SE(log(L), log(σf2))
        K = kernel(kern, X)
        VarY = g_VarY(collect(X), XXt, thresh, στ2, σβ2, σy2, σf2, L)
        VarYInv = g_VarYInv(K, collect(X), thresh, στ2, σβ2, σy2);
        loglik = mvnorm_logpdf(VarY,VarYInv,Y)
        return -loglik
    end
end
```

In [ ]:
```
_optim_ST = gen_optim_target(_Xgrid, _Ygrid, 5.0, 10.0, 10.0)
```

In [ ]:
```
_partial_opt_TS=optimize(_optim_ST, [0.0, 0.0, 0.0], method=:l_bfgs; ftol=0.01, show_every=true
Profile.clear()
Profile.init(n=10^7,delay=0.01)
@profile _partial_opt_TS=optimize(_optim_ST, [0.0, 0.0, 0.0], method=:l_bfgs; ftol=0.01, show_e
```

In [ ]:
```
Profile.print(combine=true, format=:flat)
```

In [ ]:
```
_optim_t = gen_optim_target(collect(_Xgrid), _Ygrid, 5.0, 10.0, 10.0)
@time _partial_opt=optimize(_optim_t, [0.0, 0.0, 0.0], method=:l_bfgs; ftol=0.01, show_every=tru
```

In [ ]:
```
_partial_opt.minimum
```

In [ ]:
```
_partial_opt
```

In [ ]:
```
_partial_opt_TS.minimum
```

In [ ]:
```
_partial_opt_TS
```

In [ ]:
```
_optim_ST(_partial_opt_TS.minimum)
```

In [ ]:
```
_optim_ST(_partial_opt.minimum)
```

In [ ]:
```
_optim_t(_partial_opt_TS.minimum)
```

In [ ]:
```
_optim_t(_partial_opt.minimum)
```

4 Autodiff

Auto-differentiation using the DualNumbers package doesn't work well for multivariate analysis.

In [ ]:
```
using ForwardDiff
x = rand(3)
h{T<:Number}(x::Vector{T}) = sum(sin, x) + prod(tan, x) * sum(sqrt, x);
∇h = ForwardDiff.gradient(h)
∇h(x)
```

In [ ]:
```julia
x = [Dual(2.0,1.0), Dual(3.0,1.0), Dual(2.5, 1.0), Dual(1.3, 1.0)]
XXt = x * x'
cholfact(Symmetric(XXt))
```

In [ ]:
```julia
realpart(XXt)
```

In [ ]:
```julia
@time optim_t(x);
∇opt = ForwardDiff.gradient(optim_t)
@time ∇opt(x)
```

In [ ]:
```julia
using Base.LinAlg: chol, chol!
# Linear algebra on dual matrices

# Solve the matrix system
#
#   U'*M + M'*U = B
#
# for M, where B is symmetric and U is upper triangular.  This looks a bit like
# the *-Sylvester equation, but with a lot more structure.  The solution
# algorithm is basically a forward substitution method.
function tri_ss_solve!(M, U, B)
    n = size(U,1)
    # Compute M row by row.  The expression for the i'th row is broken into a
    # part involving matrix products of the previously computed rows of M with
    # parts of U, and a part involving the current row.
    for i = 1:n
        a = B[i,i:end]
        if i > 1
            # Uses only parts of M computed in previous iterations.
            a -= M[1:i-1,i]'*U[1:i-1,i:end] + U[1:i-1,i]'*M[1:i-1,i:end]
        end
        M[i,i] = a[1]/(2*U[i,i])
        println("Uii: ", U[i,i])
        println("Mii: ", M[i,i])
        println("Ui,*:", U[i,i+1:end])
        M[i,i+1:end] = (a[1,2:end] - M[i,i]*U[i,i+1:end]) / U[i,i]
    end
    return M
end

# Version of tri_ss_solve!() optimized for BLAS types
# function tri_ss_solve!{T<:Base.LinAlg.BlasFloat}(M::AbstractMatrix{T},
#                                     U::AbstractMatrix{T}, B::AbstractMatrix{T})
#     n = size(U,1)
#     a = zeros(n)
#     mi = zeros(n)
#     ui = zeros(n)
#     unit = one(T)
#     for i = 1:n
#         m = n-i+1
#         # a[1:m] = B[i,i:n]
#         for k=1:m
#             a[k] = B[i,i+k-1]
#         end
#         if i > 1
```

```
#             # a -= M[1:i-1,i]'*U[1:i-1,i:end] + U[1:i-1,i]'*M[1:i-1,i:end]
#             for k=1:i-1
#                 mi[k] = M[k,i]
#                 ui[k] = U[k,i]
#             end
#             # Call BLAS directly to avoid temporary array copies.
#             BLAS.gemv!('T', -unit, sub(U,1:i-1,i:n), mi, unit, a)
#             BLAS.gemv!('T', -unit, sub(M,1:i-1,i:n), ui, unit, a)
#         end
#         M[i,i] = a[1]/(2*U[i,i])
#         # M[i,i+1:end] = (a[1,2:end] - M[i,i]*U[i,i+1:end]) / U[i,i]
#         for k=2:m
#             M[i,i+k-1] = (a[k] - M[i,i]*U[i,i+k-1]) / U[i,i]
#         end
#     end
#     return M
# end

# Cholesky factorization of arrays of dual numbers
#
# The returned matrix is upper triangular
function chol42{T}(A :: AbstractMatrix{Dual{T}})
    U = chol(real(A))
    B = epsilon(A)
    M = zeros(size(A))
    tri_ss_solve!(M, U, B)
    return dual(U,M)
end
```

In [ ]:
```
Σhat = cov(rand(Normal(), (10000,5)))
Σdual = Array(Dual{Float64}, size(Σhat))
for i in 1:5
    for j in 1:5
        Σdual[i,j] = Dual(Σhat[i,j], 1.0)
    end
end
chol42(Σdual)
```