Computational issues

K Kristensen

26.02.2017

TMB Intro

- Template Model Builder (TMB).
- R-package inspired by AD Model Builder (ADMB).
- Laplace approximation using automatic differentiation (AD).
- Interface: User codes the likelihood function in C++ $\Longrightarrow$ flexible !
  - No formula interface.
  - Can use multiple data sources.
  - No limitations on complexity of mean / covariance structures.
- Combines external libraries: CppAD, Eigen, CHOLMOD

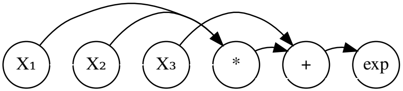
What is automatic differentiation ?

- Given function

$$f : R^n \rightarrow R$$

Apply the chainrule for each step of a computation.

  - Example: $f(x_1, x_2, x_3) = \exp(x_1 x_2 + x_3)$



  - Forward mode. n forward passes through the computational graph to get gradient.
    - Expensive gradient

$$work(\nabla f) > n \times work(f)$$

    - Easy to implement
  - Reverse mode. One forward pass and one reverse pass to get gradient:
    - Cheap gradient

$$work(\nabla f) < 4 \times work(f)$$

    - Memory consuming
    - Difficult to implement

Laplace and gradient

- Given joint negative log likelihood $f(u, \theta)$ of random effects $u \in R^n$ and parameters $\theta \in R^k$.
- Marginal likelihood :

$$L(\theta) = \int e^{-f(u,\theta)}\, du$$

- Recall the Laplace approximation of the negative log-likelihood:

$$-\log L^*(\theta) = -n \log \sqrt{2\pi} + \frac{1}{2} \log \det(H(\hat{u}(\theta), \theta)) + f(\hat{u}(\theta), \theta). \tag{1}$$

$$\hat{u}(\theta) = \arg\min_u f(u, \theta)\ . \tag{2}$$

We use $H(u, \theta)$ to denote the Hessian of $f(u, \theta)$ w.r.t. u

$$H(u, \theta) = f''_{uu}(u, \theta)\ . \tag{3}$$

TMB computational strategy

User implements the joint negative log likelihood $f(u, \theta)$ in C++.

1. Sparsity pattern of Hessian wrt. random effects u is autodetected using a symbolic analysis.
2. Laplace approximation is automatically calculated (involving AD up to order 2).
3. TMB contains a carefully selected scheme for the marginal likelihood gradient that requires a minimum amount of memory while at the same time maintaining the cheap gradient principle. This involves AD up to order 3.

- 1–3 are available as function objects from the R session. Passed to an optimizer by the user.

Theta logistic population model

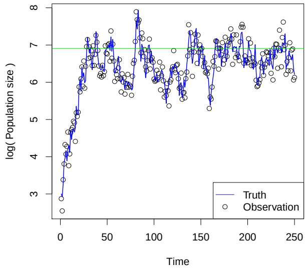
$$X_t = X_{t-1} + r_0 \left(1 - \left(\frac{\exp(X_{t-1})}{K}\right)^{\theta}\right) + e_t,$$

$$Y_t = X_t + u_t,$$

where $e_t \sim N(0, Q)$ and $u_t \sim N(0, R)$.

- Population size (state vector $\sim$ random effects) X
- Observations Y

- Parameters: r_0, K, θ, Q, R.



TMB implementation - C++ template

- Data and parameter section

```
DATA_VECTOR(Y);               // Data
PARAMETER_VECTOR(X);          // Random effects

// Parameters              Transformed parameters
PARAMETER(logr0);          Type r0 = exp(logr0);
PARAMETER(logtheta);       Type theta = exp(logtheta);
PARAMETER(logK);           Type K = exp(logK);
PARAMETER(logQ);           Type Q = exp(logQ);
PARAMETER(logR);           Type R = exp(logR);

int n = Y.size();                // Number of time points
Type f = 0;                      // Initialize summation
```

TMB implementation - C++ template (continued)

- Procedure section

```cpp
// Likelihood for process
for(int t=1; t<n; t++) {    // start at t = 1
  Type mean = X[t-1] +
    r0 * (1.0 - pow( exp(X[t-1]) / K, theta));
  f -= dnorm(X[t], mean, sqrt(Q), true);
}

// Likelihood for observations
for(int t=0; t<n; t++) {    // start at t = 0
  f -= dnorm(Y[t], X[t], sqrt(R), true);
}

// Return result
return f;
```

TMB implementation - R code

```r
library(TMB)

## Compile and load the C++ model
compile("thetalog.cpp")
dyn.load(dynlib("thetalog"))

## Read the data
Y <- scan("thetalog.dat", skip=3, quiet=TRUE)
data <- list(Y=Y)
```

TMB implementation - R code

```r
## All model parmameters with initial values
parameters <- list(
  X = rep(0, length(Y)),
  logr0 = 0,
  logtheta = 0,
  logK = 6,
  logQ = 0,
  logR = 0
)

## Make Function objects and call optimizer
obj <- MakeADFun(data, parameters, random="X")
system.time(opt <- nlminb(obj$par, obj$fn, obj$gr))

## Get standard errors
rep <- sdreport(obj)
```

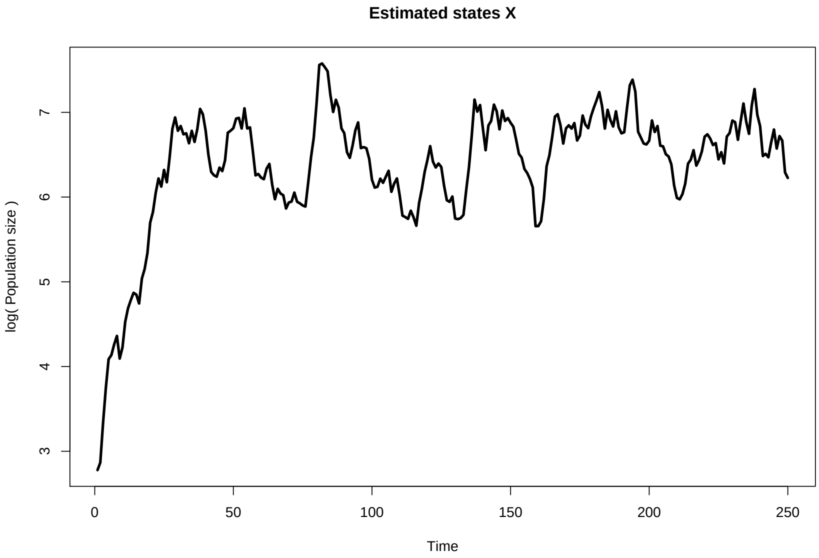
TMB output

```
rep
```

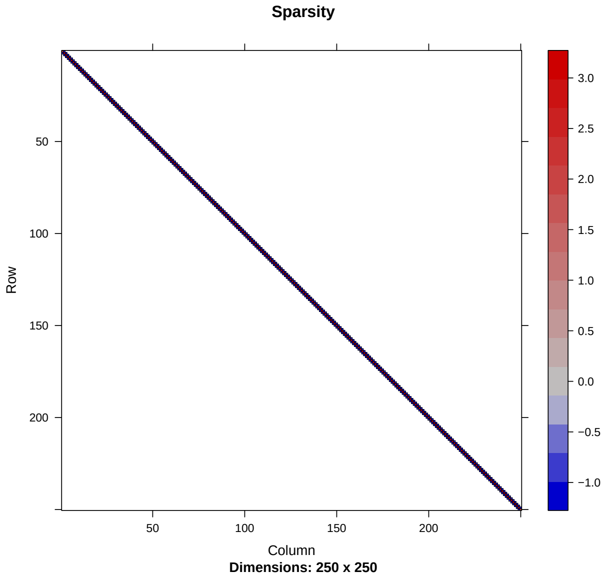
```
## sdreport(.) result
##             Estimate Std. Error
## logr0    -1.5494040  0.5046455
## logtheta -0.2450515  0.6313152
## logK      6.6274196  0.1156406
## logQ     -2.8309443  0.2335364
## logR     -2.9927966  0.2182285
## Maximum gradient component: 1.636775e-05
```

TMB output

```r
plot( as.list(rep, "Est")$X )
```



TMB output

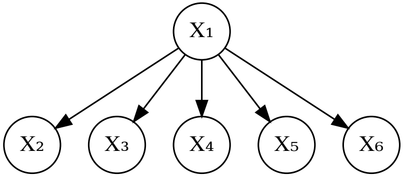

Sparsity
50
100
150
200
Row
50
100
150
200
Column
Dimensions: 250 x 250
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0

Advanced features

- Density constructors `GMRF`, `MVNORM`, `ARk`, `SEPARABLE`, `SCALE`,...
- A fairly complete list of distributions and special functions: `besselK`, `dtweedie`
- Automatic bias correction when reporting a non-linear function of random effects.
- Automatic generate one-step-ahead quantile residuals for model validation.
- Likelihood profiling.
- Automatic differentiation from the user template.
- Parallelization of the user template: `parallel_accumulator`
- Easy simulation experiments from within the user template: `SIMULATE`.

Sparsity

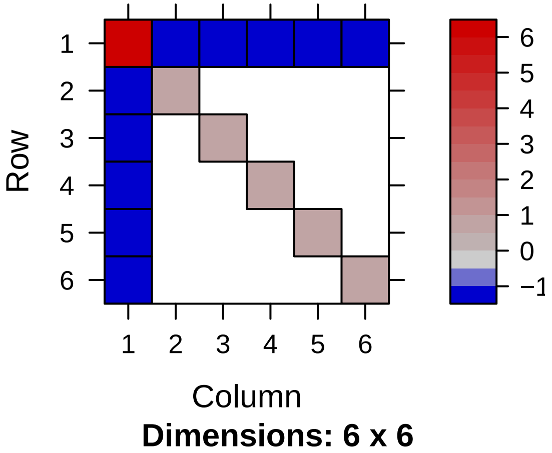
- The good graph: New random effects are linked to **few** past random effects.


$$\begin{tikzcd} & & X_1 \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\ X_2 & X_3 & X_4 & X_5 & X_6 \end{tikzcd}$$

```
nll -= dnorm(X1,  0, 1, true);
nll -= dnorm(X2, X1, 1, true);
nll -= dnorm(X3, X1, 1, true);
nll -= dnorm(X4, X1, 1, true);
nll -= dnorm(X5, X1, 1, true);
nll -= dnorm(X6, X1, 1, true);
```

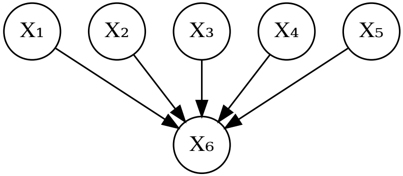
Visualize pattern

```r
image(obj.good$env$spHess())
```



Sparsity

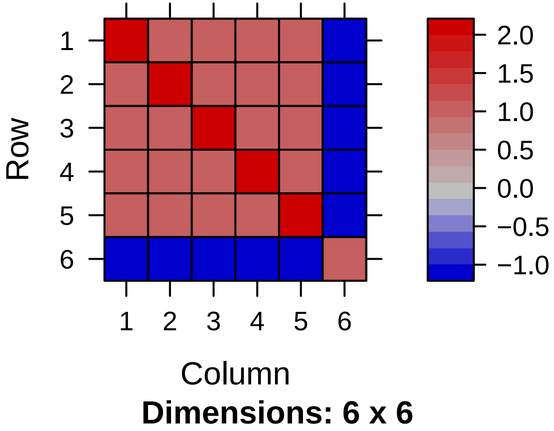
- The bad graph: New random effects linked to many past random effects.



```
nll -= dnorm(X1, 0, 1, true);
nll -= dnorm(X2, 0, 1, true);
nll -= dnorm(X3, 0, 1, true);
nll -= dnorm(X4, 0, 1, true);
nll -= dnorm(X5, 0, 1, true);
nll -= dnorm(X6, X1 + X2 + X3 + X4 + X5, 1, true);
```

Visualize pattern

```
image(obj.bad$env$spHess())
```



A space time TMB example from scratch

- **Poisson data**. Latent log-intensity u is a random field.

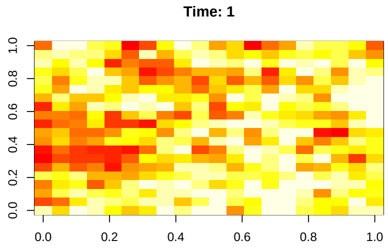
$$counts \sim Pois(su + \mu)$$

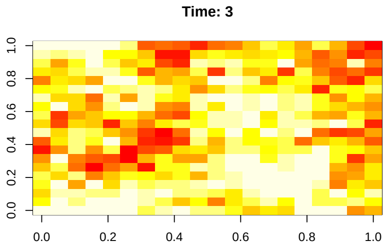
- $u(x, t)$ a 2D array indexed by spatial coordinate x and time index t.

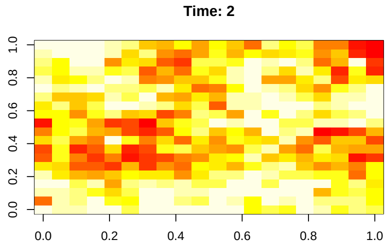
$$u_t = \phi u_{t-1} + \varepsilon_t \qquad \varepsilon_t \sim N(0, Q^{-1})$$

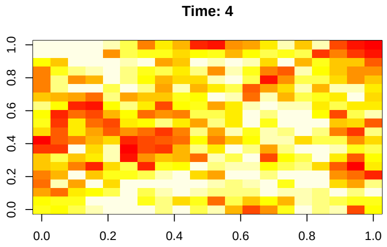
- Time-stationary
- $Q = Q_0 + \delta I$ is the precision of a Gaussian Markov random field.
- $$Q_0(i,j) = \begin{cases} |ne(i)| & i = j \\ -1 & i \sim j \\ 0 & \text{otherwise} \end{cases}$$

Show a simulation









Before we start: Overview of covariance structures

Constructor	Description
GMRF	Gaussian Markov random field
MVNORM	Multivariate normal
UNSTRUCTURED_CORR	Unstructured correlation matrix
AR1	Stationary 1st order autoregressive
ARk	Stationary kth order autoregressive
SEPARABLE	Separable extension of two densities
SCALE	Scale a density

Example

```
// evaluate negative log likelihood at point u.
GMRF(Q)(u);
```

Implementing the model (random effects)

Pass sparse matrices Q_0 and I from R to C++:

```
DATA_SPARSE_MATRIX(Q0);
DATA_SPARSE_MATRIX(I);
```

Pass parameters from R to C++

```
PARAMETER_ARRAY(u);          // Latent variables
PARAMETER(logdelta);         // GMRF Parameter (space)
PARAMETER(logitphi);         // AR1 Parameter  (time)
```

Add negative log likelihood of random field:

```
Type f = 0;                  // Joint neg. log lik.
Type delta = exp(logdelta);
Type phi = invlogit(logitphi);
Eigen::SparseMatrix<Type> Q = Q0 + delta * I;
f += SEPARABLE( AR1(phi), GMRF(Q) ) (u);
```

Implementing the model (data)

- Given u add Poisson observations

```
// Parameters
PARAMETER(mu);
PARAMETER(scale);

// Data
DATA_VECTOR(counts);

// Log-mean of observations
vector<Type> logmu_counts = mu + scale * u;

// Likelihood contribution
f -= dpois(counts, exp(logmu_counts), true).sum();
```

Running the model from R

```r
obj <- MakeADFun(data, parameters, random="u")
fit <- nlminb(obj$par, obj$fn, obj$gr)
sdr <- sdreport(obj)
sdr
```

```
## sdreport(.) result
##             Estimate Std. Error
## mu         0.4299948 0.03617505
## scale      0.9851019 0.01664911
## logdelta -1.9479875 0.13148526
## logitphi  0.3807035 0.05461652
## Maximum gradient component: 0.03692913
```

Performance hints for large models

- The normalizing constant of the GMRF can be expensive to calculate when many spatial nodes. Hint: Drop the normalizing constant in the template using `GMRF(Q, false)` and normalize the prior from R using `TMB::normalize(obj)`.
- Re-order nodes using `TMB::runSymbolicAnalysis(obj)`.
- Use BLAS

Checking the Laplace approximation

From `?TMB::checkConsistency`:

$$\int \int \exp(-f_\theta(u,x))\, du\, dx = 1$$

It follows that the joint and marginal score functions are central:

- $$E_{u,x}\left[\nabla_\theta f_\theta(u,x)\right] = 0$$
- $$E_x\left[\nabla_\theta - \log\left(\int \exp(-f_\theta(u,x))\, du\right)\right] = 0$$

Add simulation code to template

- Process simulation:

```
f += SEPARABLE( AR1(phi), GMRF(Q) ) (u);
// Add this to allow simulation:
SIMULATE {
  SEPARABLE( AR1(phi), GMRF(Q) ).simulate (u);
  REPORT(u);
}
```

- Data simulation:

```
f -= dpois(counts, exp(logmu_counts), true).sum();
// Add this to allow simulation:
SIMULATE {
  counts = rpois(exp(logmu_counts));
  REPORT(counts);
}
```

Checking LA output

```r
chk <- checkConsistency(obj)
chk
```

```
## Parameters used for simulation:
##          mu      scale   logdelta   logitphi
##  0.5000000  1.0000000 -2.0000000  0.4054651
##
## Test correct simulation (p.value):
## [1] 0.5222433
## Simulation appears to be correct
##
## Estimated parameter bias:
##            mu         scale      logdelta      logitphi
##  0.004564256 -0.012517242 -0.004222719  0.030402358
```

The state of MCMC in TMB

- Package adnuts (on CRAN)
- Package tmbstan (on CRAN)

A few use cases

- A full Bayesian analysis (slow in high dimension)

```
tmbstan(obj)
```

- A full Bayesian analysis with Laplace approx for random effects

```
tmbstan(obj, laplace=TRUE)
```

- Sampling random effects *given* the MLE obtained from TMB:

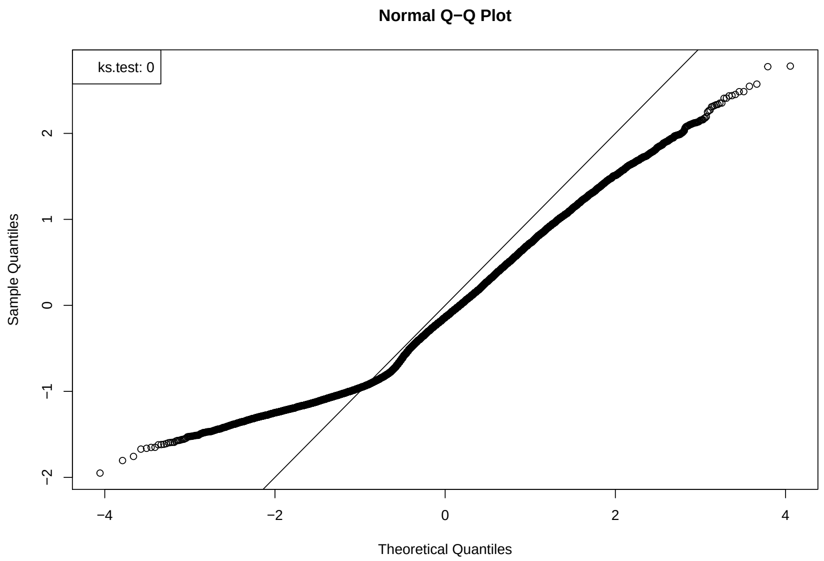
```
obj2 <- MakeADFun(..., map=map)
tmbstan(obj2)
```

Goodness of fit residuals

- How **not** to.
- $$r = \frac{observed - expected}{\sqrt{variance}}$$

  where $expected = exp(s\hat{u} + \mu)$.
- Experiment: Generate residuals based on the **true model**.

Naive residuals - true model



Simulation based residuals

- Given the data generating measure P_θ of the pair $(u, data)$ of random effects and data.
- If we knew u it would be easy to check the distributional assumption of *data*... And that of u itself...
- Can we plug in $\hat{u}$? NO !
- However, we **can** plug in a posterior *sample* u^* given the *data*.
- Works in high dimension in contrast to some other methods.

Getting the residuals

Given the parameter vector $\theta = (\mu, s, \delta, \phi)$.

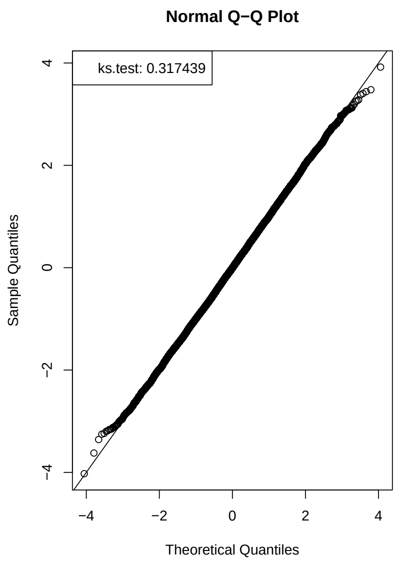
1. Draw posterior sample u^* using MCMC

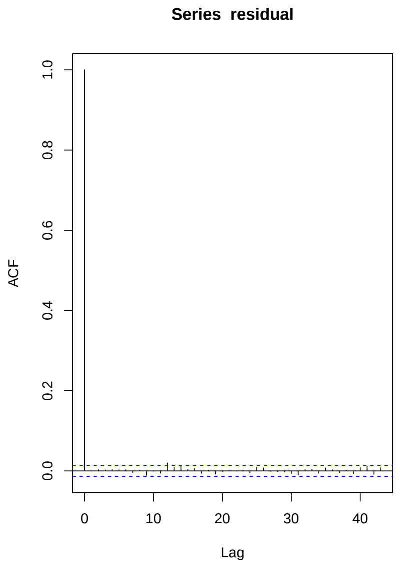
```
FIXED <- factor(NA)
map <- list(mu       = FIXED, scale    = FIXED,
            logdelta = FIXED, logitphi = FIXED)
obj2 <- MakeADFun(data, parameters, map = map)
library(tmbstan)
sample <- extract( tmbstan(obj2) )$u
```

2. Transform with CDF and randomize

```
mean <- exp(sample * scale + mu)
Fx <- ppois(data$counts, mean)
px <- dpois(data$counts, mean)
u <- runif(length(Fx))
residual <- qnorm(Fx - u * px)
```

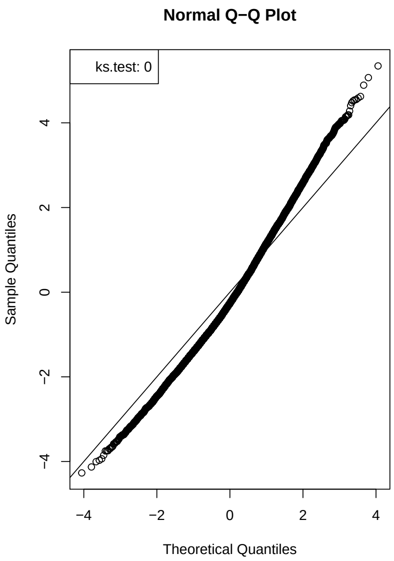
Simulation based residuals - true model

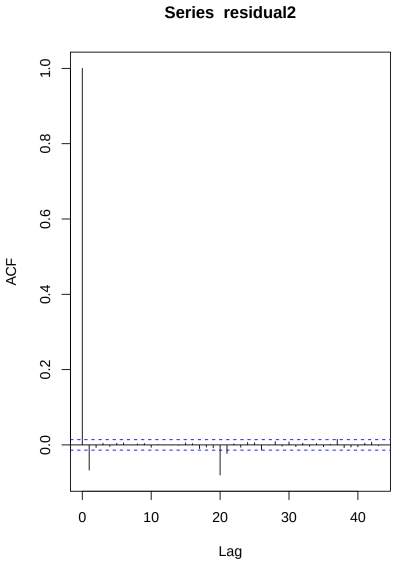




Simulation based residuals - wrong model

- Including nugget effect not accounted for by model.





Comments on residuals

- Pearson like residuals are flawed.
- Simulation based residuals do work.
- However, the power of these residuals may be disappointing when assessing the MLE $\hat{\theta}$. The MLE is often able to compensate for a wrong model specification in unexpected ways.