A comparison between ADMB & TMB

Kasper Kristensen, Anders Nielsen, Casper W. Berg

September, 2013

TMB Intro

- ADMB inspired R-package
- Combines external libraries: CppAD, Eigen, CHOLMOD
- Continuously developed since 2009, $\sim$ 1000 lines of code
- Implements Laplace approximation for random effects
- C++ Template based
- Automatic sparseness detection
- Parallelism through BLAS
- Parallel user templates
- Parallelism through `multicore` package

Example 1: Linear regression

```
DATA_SECTION
init_int N
init_vector Y(1,N)
init_vector x(1,N)
PARAMETER_SECTION
init_number a
init_number b
init_number logSigma
sdreport_number sigmasq
objective_function_value nll
PROCEDURE_SECTION
sigmasq=exp(2*logSigma);
nll=0.5*(N*log(2*M_PI*sigmasq)
    +sum(square(Y-(a+b*x)))/sigmasq);
```

```
#include <TMB.hpp>
template<class Type>
Type objective_function<Type>::operator() ()
{
DATA_VECTOR(Y);
DATA_VECTOR(x);
PARAMETER(a);
PARAMETER(b);
PARAMETER(logSigma);
Type nll=dnorm(Y,a+b*x,exp(logSigma),true).sum();
return nll;
}
```

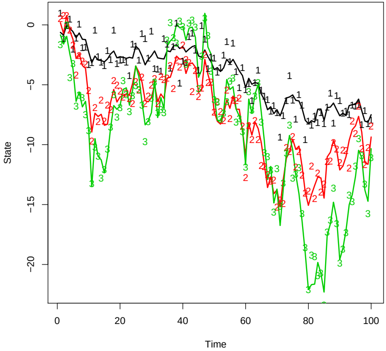
Example 2: Multivariate random walk

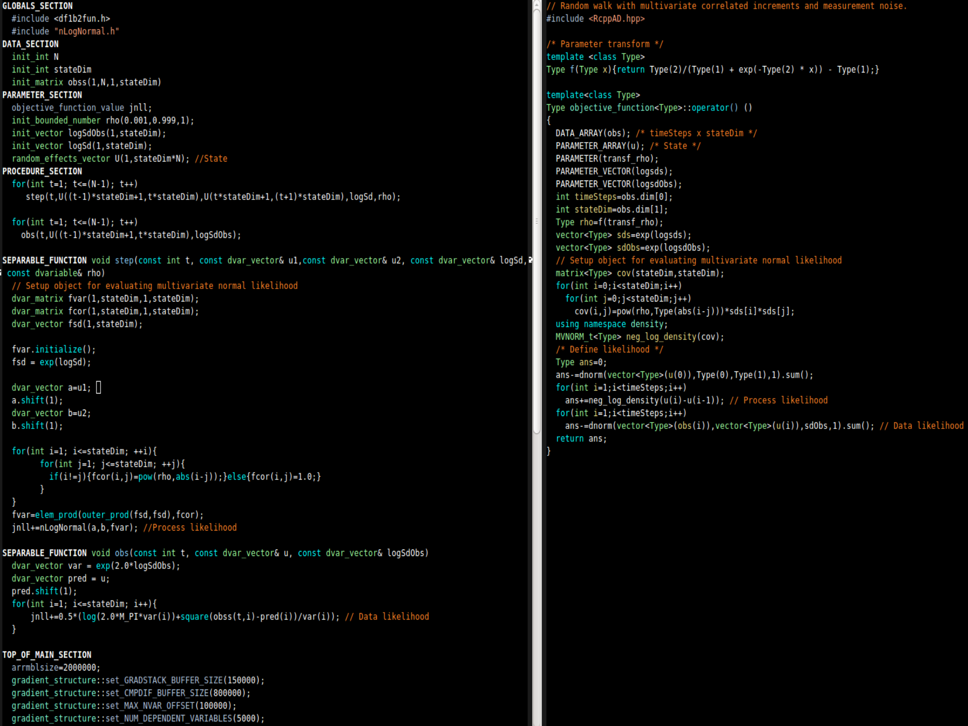
$$\mathbf{X}_{t+1} = \mathbf{X}_t + \varepsilon_{\mathbf{t}} \quad , \varepsilon_{\mathbf{t}} \sim N(\mathbf{0}, \Sigma)$$

$$\mathbf{Y}_t = \mathbf{X}_t + \eta_{\mathbf{t}} \quad , \eta_{\mathbf{t}} \sim N(\mathbf{0}, \sigma_Y^2 \mathbf{I})$$

$$\Sigma_{i,j} = \rho^{|i-j|} \sigma_i \sigma_j$$

States (random effects) **X**, Observations **Y**. Parameters: $\sigma, \sigma_{\mathbf{Y}}, \rho$.





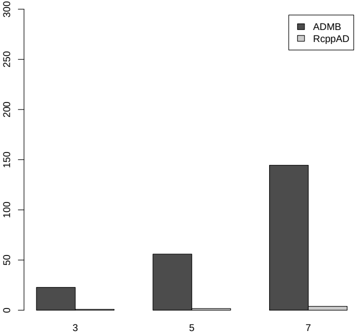
```
GLOBALS_SECTION
  #include <df1b2fun.h>
  #include "nLogNormal.h"
DATA_SECTION
  init_int N
  init_int stateDim
  init_matrix obss(1,N,1,stateDim)
PARAMETER_SECTION
  objective_function_value jnll;
  init_bounded_number rho(0.001,0.999,1);
  init_vector logSdObs(1,stateDim);
  init_vector logSd(1,stateDim);
  random_effects_vector U(1,stateDim*N); //State
PROCEDURE_SECTION
  for(int t=1; t<=(N-1); t++)
     step(t,U((t-1)*stateDim+1,t*stateDim),U(t*stateDim+1,(t+1)*stateDim),logSd,rho);

  for(int t=1; t<=(N-1); t++)
    obs(t,U((t-1)*stateDim+1,t*stateDim),logSdObs);

SEPARABLE_FUNCTION void step(const int t, const dvar_vector& u1,const dvar_vector& u2, const dvar_vector& logSd,
 const dvariable& rho)
  // Setup object for evaluating multivariate normal likelihood
  dvar_matrix fvar(1,stateDim,1,stateDim);
  dvar_matrix fcor(1,stateDim,1,stateDim);
  dvar_vector fsd(1,stateDim);

  fvar.initialize();
  fsd = exp(logSd);

  dvar_vector a=u1;
  a.shift(1);
  dvar_vector b=u2;
  b.shift(1);

  for(int i=1; i<=stateDim; ++i){
        for(int j=1; j<=stateDim; ++j){
          if(i!=j){fcor(i,j)=pow(rho,abs(i-j));}else{fcor(i,j)=1.0;}
        }
  }
  fvar=elem_prod(outer_prod(fsd,fsd),fcor);
  jnll+=nLogNormal(a,b,fvar); //Process likelihood

SEPARABLE_FUNCTION void obs(const int t, const dvar_vector& u, const dvar_vector& logSdObs)
  dvar_vector var = exp(2.0*logSdObs);
  dvar_vector pred = u;
  pred.shift(1);
  for(int i=1; i<=stateDim; i++){
      jnll+=0.5*(log(2.0*M_PI*var(i))+square(obss(t,i)-pred(i))/var(i)); // Data likelihood
  }

TOP_OF_MAIN_SECTION
  arrmblsize=2000000;
  gradient_structure::set_GRADSTACK_BUFFER_SIZE(150000);
  gradient_structure::set_CMPDIF_BUFFER_SIZE(800000);
  gradient_structure::set_MAX_NVAR_OFFSET(100000);
  gradient_structure::set_NUM_DEPENDENT_VARIABLES(5000);
```

```
// Random walk with multivariate correlated increments and measurement noise.
#include <RcppAD.hpp>

/* Parameter transform */
template <class Type>
Type f(Type x){return Type(2)/(Type(1) + exp(-Type(2) * x)) - Type(1);}

template<class Type>
Type objective_function<Type>::operator() ()
{
  DATA_ARRAY(obs); /* timeSteps x stateDim */
  PARAMETER_ARRAY(u); /* State */
  PARAMETER(transf_rho);
  PARAMETER_VECTOR(logsds);
  PARAMETER_VECTOR(logsdObs);
  int timeSteps=obs.dim[0];
  int stateDim=obs.dim[1];
  Type rho=f(transf_rho);
  vector<Type> sds=exp(logsds);
  vector<Type> sdObs=exp(logsdObs);
  // Setup object for evaluating multivariate normal likelihood
  matrix<Type> cov(stateDim,stateDim);
  for(int i=0;i<stateDim;i++)
    for(int j=0;j<stateDim;j++)
      cov(i,j)=pow(rho,Type(abs(i-j)))*sds[i]*sds[j];
  using namespace density;
  MVNORM_t<Type> neg_log_density(cov);
  /* Define likelihood */
  Type ans=0;
  ans-=dnorm(vector<Type>(u(0)),Type(0),Type(1),1).sum();
  for(int i=1;i<timeSteps;i++)
    ans+=neg_log_density(u(i)-u(i-1)); // Process likelihood
  for(int i=1;i<timeSteps;i++)
    ans-=dnorm(vector<Type>(obs(i)),vector<Type>(u(i)),sdObs,1).sum(); // Data likelihood
  return ans;
}
```

Example 2: Results (timings)

	3	5	7
ADMB	22.74	55.93	144.47
TMB	0.91	1.63	3.85
Speed-up	24.88	34.34	37.56

Table: Runtime in seconds for multivariate random walk example.



Parallel user templates intro

- Most objective functions are a result of commutative accumulation ($\theta =$ random and fixed effects):

$$l(\theta) = \sum_{i=1}^{n} l_i(\theta)$$

- If e.g. two cores then let core 1 do AD of the “even terms” and core 2 do AD of the “odd terms”.
- The book keeping is handled by template class `parallel_accumulator<Type>`.
- From user perspective: change one line of template to get parallel version.

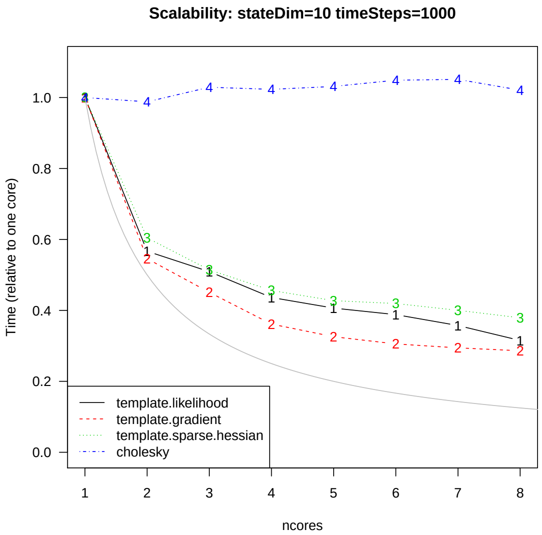
Parallel Code

```
...
Type ans=0;
...
```

```
...
parallel_accumulator<Type> ans(this);
...
```

- Parallel accumulator initialized to zero and has only methods "$+=$" and "$-=$".
- When modified code is compiled from R the template is detected to be parallel and the openmp flag is set.

Results: benchmark plot



Parallel Code with multicore package

- High level parallelization gives best performance.
- Easy with `multicore` package [1].

Examples:

- Parallel likelihood evaluations
  ```
  mclapply(1:10,function(x)obj$fn(obj$par))
  ```
- Parallel gradient evaluations
  ```
  mclapply(1:10,function(x)obj$gr(obj$par))
  ```
- Parallel optimization
  ```
  mclapply(1:10,function(x)do.call("optim",obj))
  ```

---

[1] Note: Before calling `mclapply` do `openmp(1)` to avoid forking a multithreaded process

- Slow compile times
- Standalone applications not possible
- Fewer built-in specialized functionalities (e.g. profile-likelihood, `sd_report_number` etc.)
- Sparse documentation
- Depends on external libraries

- Fast run times
- The use of external libraries means a compact code base that is highly optimized
- Can handle very high dimensional problems ($\sim 10^6$ random effects)
- No `SEPARABLE_FUNCTION` construct needed, fully automatic sparseness detection
- Full R integration – no need for data+results import/export
- No use of temporary files on the disc
- Template based – no code duplication needed as for `df1b2variable`s etc.
- Analytical Hessian for fixed effects.
- High-level parallelization with `multicore` package.