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, ~ 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

K. Kristensen, A. Nielsen, C.W. Berg
Example 1: Linear regression

```cpp
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
report_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);
```

```cpp
#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

\[ X_{t+1} = X_t + \varepsilon_t, \quad \varepsilon_t \sim N(0, \Sigma) \]
\[ Y_t = X_t + \eta_t, \quad \eta_t \sim N(0, \sigma_Y^2 I) \]
\[ \Sigma_{i,j} = \rho^{|i-j|} \sigma_i \sigma_j \]

States (random effects) \( X \),
Observations \( Y \). Parameters:
\( \sigma, \sigma_Y, \rho \).
include "df1b2fun.h"
#include "LogNormal.h"

GLOBAL_SECTION
include <Rcpp.h>

DATA_SECTION
init_int N
init_int stateDim
init_matrix obs(1, N, 1, stateDim)

PARAMETER_SECTION
objective_function_value Jnll;
init bounded_number rho(0.001, 0.9991);
init_vector logSdObs(1, stateDim);
init_vector logSd(1, stateDim);
random_effects_vector U1(stateDim, N); // State

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

SEPARABLE_FUNCTION void obs(const int t, const dvar_vector & u, const dvar_vector & u2, const dvar_vector & logSd, const dvarable & rho)
// Setup object for evaluating multivariate normal likelihood

// 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 = 0; i < stateDim; i++)
for (int j = 0; j < stateDim; j++)
cov(i, j) = pow(rho, abs(1 - j));
endfor
fcor[i, j] = 1.8;

fvar.elem_prod(outer_prod(fsd, fsd), fcor);
jnll = -logNormal(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 * PI) + var(i)) + square(obs(t, i) - pred(i)); // Data likelihood

TOP_OF_MAIN_SECTION
armblsize = 2000000;
gradients: set GRADSTACK BUFFER SIZE (1500000);
gradients: set CMBDIF BUFFER SIZE (1000000);
gradients: set MAX NVAR OFFSET (1000000);
gradients: set NUM DEPENDENT VARIABLES (500000);
Example 2: Results (timings)

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADMB</td>
<td>22.74</td>
<td>55.93</td>
<td>144.47</td>
</tr>
<tr>
<td>TMB</td>
<td>0.91</td>
<td>1.63</td>
<td>3.85</td>
</tr>
<tr>
<td>Speed-up</td>
<td>24.88</td>
<td>34.34</td>
<td>37.56</td>
</tr>
</tbody>
</table>

**Table:** Runtime in seconds for multivariate random walk example.
Parallel user templates intro

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

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

- 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

- 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.

```cpp
Type ans=0;
... parallel_accumulator<Type> ans(this);
...
```
Results: benchmark plot

Scalability: stateDim=10 timeSteps=1000

K. Kristensen, A. Nielsen, C.W. Berg
Parallel Code with multicore package

- High level parallelization gives best performance.
- Easy with `multicore` package \(^1\).

Examples:

- Parallel likelihood evaluations
  \[
  \text{mclapply}(1:10, \text{function}(x) \text{obj}$fn(\text{obj}$par))
  \]

- Parallel gradient evaluations
  \[
  \text{mclapply}(1:10, \text{function}(x) \text{obj}$gr(\text{obj}$par))
  \]

- Parallel optimization
  \[
  \text{mclapply}(1:10, \text{function}(x) \text{do.call("optim", obj))}
  \]

\(^1\) 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, \texttt{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 df1b2variables etc.
+ Analytical Hessian for fixed effects.
+ High-level parallelization with multicore package.