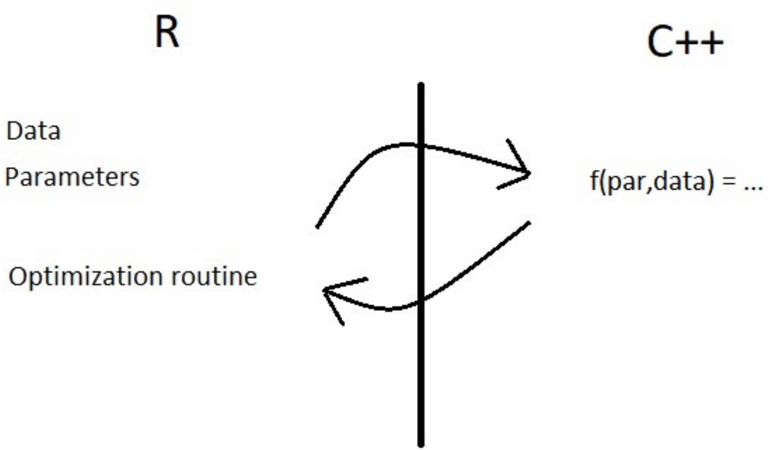
Data and parameters in TMB

Anders Nielsen and Olav Nikolai Breivik

Data and parameters



R

C++

Data

Parameters

Optimization routine

f(par,data) = ...

Data and parameters

Simple example:

```r
library(TMB)
compile("scalar.cpp")
dyn.load(dynlib("scalar"))

data = list()
data$Y = 5

par = list()
par$mu = 0

obj = MakeADFun(data,par,DLL = "scalar")
opt = nlminb(obj$par,obj$fn,obj$gr)
```

```cpp
#include <TMB.hpp>
template<class Type>
Type objective_function<Type>::operator()()
{
  DATA_SCALAR(Y);
  PARAMETER(mu);
  Type nll = pow(Y-mu,2);
  return nll;
}
```

Transfers basic objects

What	R side	C++ side
Number	`1`	`DATA_SCALAR`
Vector	`c(1,2,3)`	`DATA_VECTOR`
Matrix	`matrix(c(1,2,3,4), nrow=2, ncol=2)`	`DATA_MATRIX`
Array	`matrix(c(1,2,3,4), nrow=2, ncol=2)`	`DATA_ARRAY`
Integer	`1`	`DATA_INTEGER`
Integer Vector	`c(1,2,3)`	`DATA_IVECTOR`
Integer Matrix	`matrix(c(1,2,3,4), nrow=2, ncol=2)`	`DATA_IMATRIX`
Integer Array	`matrix(c(1,2,3,4), nrow=2, ncol=2)`	`DATA_IARRAY`
Factor	`factor(c("a","b"))`	`DATA_FACTOR`
String	`"a"`	`DATA_STRING`

Checking what is read

- Report values on C-side

```r
library(TMB)
compile("verify.cpp")
dyn.load(dynlib("verify"))
data = list()
data$V = 1:3
data$M = matrix(1:6,nrow = 2,ncol = 3)
data$A = array(1:6, dim = c(1,2,3))

par = list()
par$mu = 0
obj = MakeADFun(data,par, DLL = "verify")
out = obj$report()
out$M
#     [,1] [,2] [,3]
#[1,]    1    3    5
#[2,]    2    4    6
```

```cpp
#include <TMB.hpp>
template<class Type>
Type objective_function<Type>::operator() ()
{
  DATA_VECTOR(V);
  DATA_MATRIX(M);
  DATA_ARRAY(A);
  REPORT(V);
  REPORT(M);
  REPORT(A);

  PARAMETER(mu);
  Type nll = pow(mu,2);
  return nll;
}
```

Indexing from 0

- In C^{++} the first element is **number 0**
- Different from R, so difficult to remember in the beginning

R

```
data <- list()
data$y <- c(1.1, 2.2)
data$z <- myMatrix

y[1] ...  y[n]
z[1,1] ...  z[m,n]
```

C^{++}

```
DATA_VECTOR(y)
DATA_ARRAY(z)

y(0) ...  y(n-1)
z(0,0) ...  z(m-1,n-1)
```

Parameters

- The parameters are what we want to estimate
- Set up list on R side
- The C side evaluates the function (and derivatives)
- Optimization is performed from R, so values need to be passed from and to many times
- Simple example:

```r
library(TMB)
compile("scalar.cpp")
dyn.load(dynlib("scalar"))

data = list()

par = list()
par$mu = 0

obj = MakeADFun(data,par,DLL = "scalar")
opt = nlminb(obj$par,obj$fn,obj$gr)
```

```cpp
#include <TMB.hpp>
template<class Type>
Type objective_function<Type>::operator()()
{
  PARAMETER(mu);
  Type nll = pow(Type(42)-mu,2);
  return nll;
}
```

Often we have more than one

- The following parameter types are available:

Template Syntax	C++ type	R type
PARAMETER(name)	Type	numeric(1)
PARAMETER_VECTOR(name)	vector<Type>	vector
PARAMETER_MATRIX(name)	matrix<Type>	matrix
PARAMETER_ARRAY(name)	array<Type>	array

- Just like with data we can specify a list of possibly many parameter objects
- Naturally we need to match each parameter object on the C side

Simple bounds from R

- Consider the model:

$$X \sim \text{Bin}(100, p)$$

- Let's say we observe X's equal 2, 0 and 1
- We want to estimate our model parameter p

```r
library(TMB)
compile("p1.cpp")
dyn.load(dynlib("p1"))
data = list()
data$X = c(2,0,1)
par = list()
par$p = 0.5

obj = MakeADFun(data,par,DLL = "p1")
opt = nlminb(obj$par,obj$fn,obj$gr,
lower=c(0),upper=c(1))
rep = sdreport(obj)
summary(rep)
#   Estimate      Std. Error
#p       0.01     0.005716054
```

```cpp
#include <TMB.hpp>
template<class Type>
Type objective_function<Type>::operator()()
{
  DATA_VECTOR(X);
  PARAMETER(p);
  Type nll = -sum(dbinom(X,Type(100),p,true));
  return nll;
}
```

- Now we want to make a 95% confidence interval - see the problem?

Bounds from C

- Consider the model parametrized as:

$$X \sim \text{Bin}(100, p), \qquad \text{where } \text{logit}(p) = \alpha$$

- Our code is then:

```r
library(TMB)
compile("p2.cpp")
dyn.load(dynlib("p2"))
data = list()
data$X = c(2,0,1)
par = list()
par$alpha = 0

obj = MakeADFun(data,par,DLL = "p2")
opt = nlminb(obj$par,obj$fn,obj$gr)
rep = sdreport(obj)
summary(rep)
#         Estimate Std. Error
#alpha -4.59512   0.5802588
```

```cpp
#include <TMB.hpp>
template<class Type>
Type trans(Type x){
  return exp(x)/(Type(1) + exp(x));
}

template<class Type>
Type objective_function<Type>::operator() ()
{
  DATA_VECTOR(X);
  PARAMETER(alpha);
  Type p = trans(alpha);
  Type nll  = -sum(dbinom(X,Type(100),p,true));
  return nll;
}
```

- Now we can make a 95% confidence interval:

```r
a = -4.59512  + 0.5802588 * c(-2,2)
exp(a)/(1 + exp(a))
#[1] 0.003154903 0.031231381
```

Exercise

Exercise: Suggest how to transform parameter:

1. only positive
2. between -1 and 1
3. Increasing positive vector

Exercise

Exercise: Suggest how to transform parameter:

1. only positive
2. between -1 and 1
3. Increasing positive vector

Solution:

1. $\theta = e^{\alpha}, \quad$ where $\alpha \in \mathbb{R}$
2. $\theta = 2/(1 + e^{-\alpha}) - 1, \quad$ where $\alpha \in \mathbb{R}$
3. $\theta = (e^{\alpha_1}, e^{\alpha_1} + e^{\alpha_2}, ..., e^{\alpha_1} + \cdots + e^{\alpha_n}), \quad$ where $\alpha_1, ..., \alpha_n \in \mathbb{R}$

Getting results out

- If estimated standard errors are not needed:
  - `REPORT(X)` in the C^{++} file
  - `obj$report()$X` in the *R* file
- If estimated standard errors are needed:
  - `ADREPORT(X)` in the C^{++} file
  - Reported list:
    `rl <- as.list(sdreport(obj),"Est",report= TRUE)`
  - Reported Sd list:
    `rlsd <- as.list(sdreport(obj),"Std",report= TRUE)`
- Parameter estimates and standard deviations:
  - Parameter list: `pl <- as.list(sdreport(obj),"Est")`
  - Parameter Sd list: `plsd <- as.list(sdreport(obj),"Std")`

Exercise: Beverton-Holt stock recruitment model

- The Beverton-Holt model can be written (slightly re-parametrized) as:

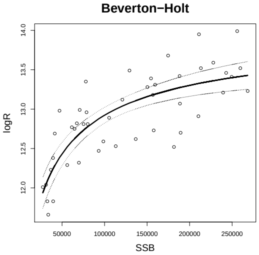
  $$\log R_i = \log(a) + \log(\mathrm{ssb}_i) - \log(1 + \exp(\log(b))\mathrm{ssb}_i) + \varepsilon_i,$$

  where $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$
- A data set of SSB and $\log(R)$ is provided in `bh.dat`
- Code provided in `bh.R` and `bh.cpp`
- Exercise: Estimate the model parameters $\log(a)$ and $\log(b)$ and $\log(\sigma)$.

Solution, R- and C-side

```r
dat<-read.table("bh.dat", header=TRUE)

library(TMB)
compile("bh.cpp")
dyn.load(dynlib("bh"))

data <- list(SSB=dat$SSB,logR=dat$logR)
parameters <- list(
  logA=0,
  logB=0,
  logSigma=0
)

obj <- MakeADFun(data,parameters,DLL="bh")
opt <- nlminb(obj$par,obj$fn,obj$gr)
rep <- sdreport(obj)
```

```cpp
#include <TMB.hpp>

template<class Type>
Type objective_function<Type>::operator() ()
{
  DATA_VECTOR(SSB)
  DATA_VECTOR(logR);

  PARAMETER(logA);
  PARAMETER(logB);
  PARAMETER(logSigma);
  vector<Type> pred=logA+log(SSB)-log(Type(1)+exp(logB)*SSB);
  Type ans=-sum(dnorm(logR,pred,exp(logSigma),true));
  ADREPORT(pred)
  return ans;
}
```



Collapsing parameters, or fixing them

- The `map` argument of the `MakeADFun` can be used to couple elements in a parameter object
- If we have a parameter vector `alpha` of length 4, then the statement:

  ```
  obj <- MakeADFun(data, param, map=list(alpha=factor(c(1,2,3,3))))
  ```

  will collapse the last two parameters.
  - Initialized to the mean of the initializations
- If `NA` is included, as in:

  ```
  obj <- MakeADFun(data, param, map=list(alpha=factor(c(1,2,NA,4))))
  ```

  the optimizer treat that parameter as fixed and equal the initial value.
- Use the map argument to:
  - easily change between different models
  - write the c-side neat

Map exercise

- Consider the data set `InsectSprays`, which is available in R
- We will use the model

$$\text{count}_i \sim \text{Pois}(\lambda_i), \text{ where } \log \lambda_i = \alpha(\text{spray}_i)$$

- This can be implemented as:

```r
library(TMB)
compile("insect.cpp")
dyn.load(dynlib("insect"))

par <- list()
par$logAlpha=rep(0,nlevels(InsectSprays$spray))
obj <- MakeADFun(InsectSprays, par, DLL="insect")
opt <- nlminb(obj$par, obj$fn, obj$gr)
```

```cpp
#include <TMB.hpp>
template<class Type>
Type objective_function<Type>::operator() ()
{
  DATA_VECTOR(count);
  DATA_FACTOR(spray);
  PARAMETER_VECTOR(logAlpha);
  Type nll = 0;
  for(int i=0; i<count.size(); ++i){
    Type lambda = exp(logAlpha(spray(i)));
    nll += -dpois(count(i),lambda,true);
  }
  return nll;
}
```

- Use the `map` argument to test if $\alpha(A) = \alpha(B) = \alpha(F)$
- Test the hypothesis $\lambda(A) = \lambda(B) = \lambda(F) = 15$
- Test the hypothesis without modifying the `cpp` file

Debugging

- Remember that the index starts at 0 in C++ and 1 in R
- Debugging is performed trough `TMB::gdbsource("script.R", interactive = TRUE)`
  - A safe version of `source`
- `debug_tutorial.cpp` fails because of index out of bound
- Exercise: Debug `debug_tutorial.cpp`
- Operating system notes:
  - Linux: Works fine
  - Windows: Behaviour of `gdb` depends on versions of R and Rtools. See `https://github.com/kaskr/adcomp/wiki/Windows-installation#windows-debugging`
    - Debugging of larger programs may not work in Windows, recommend to debug with Linux
- This exercise is borrowed from `https://github.com/skaug/tmb-case-studies/tree/master/debug_tutorial`

Some tips

- Check `obj$gr()` if you don't obtain convergence
    - If one element is zero, something is wrong with the implementation.
- Always use same names on the R and C side
- Set `control = list(trace = 1)` to trace the outer fixed parameters in `nlminb()`
- Don't underestimate the importance of writing the C side neat.
- Overview of vector, matrix and array operation in TMB is provided here: `https://kaskr.github.io/adcomp/matrix_arrays_8cpp-example.html`