TMB prelude ♪

Cóilín Minto
Marine and Freshwater Research Centre, GMIT
coilin.minto@gmit.ie

Environmetrics Forum
April 26, 2018

The objective of these notes is to practically introduce some of the concepts behind TMB.

1 Direct differentiation

1.1 A simple function

We often want to find out at what value of x is $f(x)$ maximised? For example, say we have the function

$$y = 2 + 3x - x^2, \tag{1}$$

we would like to know what value of x maximises y? To do so, we can use calculus. We know that at the maximum the first derivative will be zero, so

$$\frac{dy}{dx} = 3 - 2x = 0, \tag{2}$$
$$3 = 2x, \tag{3}$$
$$x = 3/2. \tag{4}$$

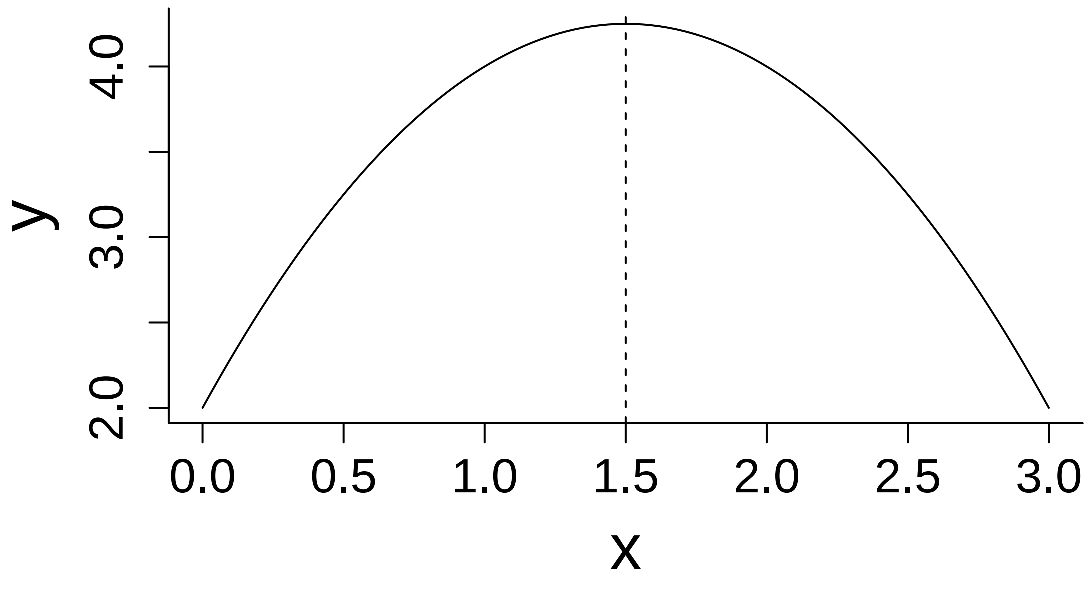

Figure 1: The function $y = 2 + 3x - x^2$ with the value of x that maximises the function shown as a vertical line.

If we didn't know how to differentiate the function, we could use an optimiser to find the maximum. For example, in R, we might write:

```r
fx <- function(x){
  y <- 2 + 3 * x -x^2
  ## return -y as optim automatically does
  ## minimization
  return(-y)
}

nlminb(objective = fx, start = 0)
```

```
## $par
## [1] 1.500001
##
## $objective
## [1] -4.25
##
## $convergence
## [1] 0
##
## $iterations
## [1] 4
##
## $evaluations
## function gradient
##        5        5
##
## $message
## [1] "relative convergence (4)"
```

Note that it took the optimisation algorithm 5 function evaluations to get the maximum (located at $x = 1.5$ where y = 4.25).
Optimisers like the gradient of the function they are optimising as they use this to navigate the surface. For example, let's supply the gradient as well as the function

```r
dydx <- function(x){
  dy <- 3 - 2 * x
  return(-dy)
}

nlminb(objective = fx, gradient = dydx, start = 0)
```

```
## $par
## [1] 1.5
##
## $objective
## [1] -4.25
##
```

```
## $convergence
## [1] 0
##
## $iterations
## [1] 3
##
## $evaluations
## function gradient
##        3        3
##
## $message
## [1] "both X-convergence and relative convergence (5)"
```

Note that when we supply the gradient we get to the maximum in only three steps. This is at the heart of why knowing how to differentiate a function matters when optimising - it will get you there faster (and with more stability).

1.2 A likelihood

Say the function we'd like to maximise was a little more complicated, like a likelihood. Imagine we have a set of observations y_i ($i = 1 \ldots n$) and we'd like to estimate the mean using maximum likelihood, assuming the data are normally distributed. For the moment, imagine we already know the standard deviation ($\sigma = 2$). The problem can then be written

$$L(\mu|\mathbf{y}) = \prod_i \frac{1}{\sigma\sqrt{2\pi}} e^{-(y_i-\mu)^2/2\sigma^2} \tag{5}$$

$$= \prod_i \frac{1}{2\sqrt{2\pi}} e^{-(y_i-\mu)^2/8} \tag{6}$$

As products become sums, it's easier to work with the log-likelihood:

$$\ln L(\mu|\mathbf{y}) = \sum_i -\ln(2) - \ln(2\pi)/2 - (y_i-\mu)^2/8. \tag{7}$$

At what value of μ is the log-likelihood (Equation 7) maximised?

$$\frac{d\ln L(\mathbf{y}|\mu)}{d\mu} = \sum_i \frac{2y_i}{8} - \frac{2\mu}{8} = 0 \tag{8}$$

$$\sum_i \frac{2\mu}{8} = \sum_i \frac{2y_i}{8} \tag{9}$$

$$\sum_i \mu = \sum_i y_i \tag{10}$$

$$n\mu = \sum_i y_i \tag{11}$$

$$\mu = \frac{\sum_i y_i}{n}, \tag{12}$$

which is just the sample mean but we've shown (with some algebra) that it's the value that maximises the likelihood.
What if we try to estimate the mean by direct optimisation. First generate the data and write a log-likelihood function:

```r
## generate some data - 10,000 observations
set.seed(101)
y <- rnorm(1e4, mean = 8)

## log-likelihood function
fmu <- function(mu){
  ll <- dnorm(y, mean = mu, sd = 2, log = TRUE)
  return(-sum(ll))
}
```

Do a quick plot of the likelihood over μ.

```r
fmu2 <- function(mu){
  ll <- dnorm(y, mean = mu, sd = 2, log = TRUE)
  return(sum(ll))
}

h <- Vectorize(fmu2)
curve(h, from = 5, to = 12, ylab = "Log likelihood", xlab = expression(mu),
      cex.axis = 1.5, cex.lab = 1.5, bty = "l", mfrow = c(5, 10, 1, 1))
abline(v = mean(y), col = "grey")
```

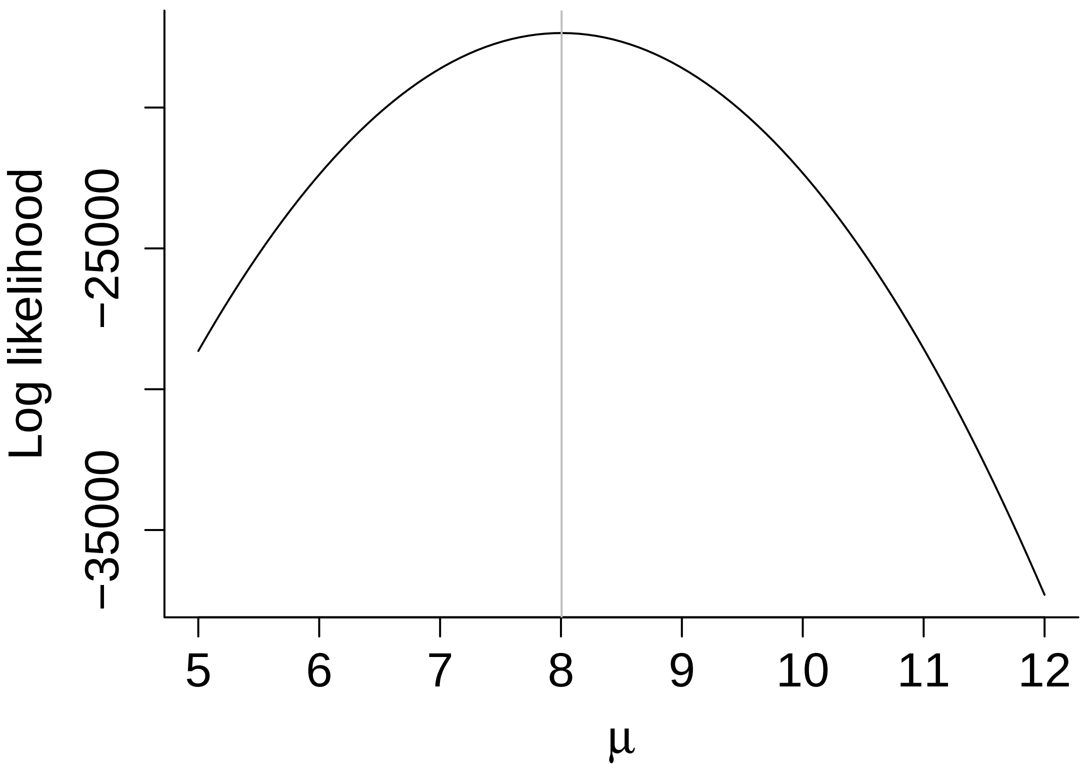

Figure 2: Log-likelihood of $\mathbf{y}$ over μ with the maximum likelihood value of μ shown as a vertical grey line.

Now let's estimate without the analytical gradient.

```
nlminb(objective = fmu, start = 5)


## $par
## [1] 8.005278
##
## $objective
## [1] 17353.73
##
## $convergence
## [1] 1
##
## $iterations
## [1] 2
##
## $evaluations
## function gradient
##       31        2
##
## $message
## [1] "false convergence (8)"
```

The fit didn't converge after 31 function evaluations (for the moment ignore the gradient output, this is an internal approximation). We know the analytical gradient (middle part of Equation 8) so let's use it:

```
dfmu <- function(mu){
  dll <- sum(2 * y / 8 - 2 * mu /8)
  return(-dll)
}

nlminb(objective = fmu, gradient = dfmu, start = 5)


## $par
## [1] 8.005278
##
## $objective
## [1] 17353.73
##
## $convergence
## [1] 0
##
## $iterations
## [1] 2
##
## $evaluations
## function gradient
##        4        2
```

```
##
## $message
## [1] "both X-convergence and relative convergence (5)"
```

It converged with just four function and two gradient evaluations.
The algebra is getting a little cumbersome and will become more difficult when we estimate regression etc. so it would be nice to have code to automatically differentiate the likelihood. This is part of what TMB does for us.

2 Automatic differentiation

2.1 A likelihood

Say we didn't know how to differentiate the likelihood above. Let's get TMB to do it for us. The first thing is to write a .cpp file with the C++ code. Here we'll use the full equation for the normal distribution but TMB already has some built-in functions it recognises.

```
#include <TMB.hpp>
template<class Type>
Type objective_function<Type>::operator() () {
  // DATA
  DATA_VECTOR(y);
  // PARAMETERS:
  PARAMETER(mu);
  // PRELIMINARY CALCULATIONS
  int n = y.size(); // number of observations
  // PROCEDURE
  Type nll = 0.0; // initialize negative log likelihood
  // loop over the observations
  for(int i = 0; i < n; i++){
    nll -= -log(2.0) - log(2.0 * M_PI) / 2.0 - pow(y(i) - mu, 2.0) / 8.0;
  }
  return nll;
}
```

Compile and load the model:

```
## load the TMB library
library(TMB)
```

```
## step takes a while (on my machine anyway)
compile("unknown_mean.cpp")
```

```r
## load the function
dyn.load(dynlib("unknown_mean"))
```

Next step is to create an object for the TMB function with the data and some specifications for optimising.

```r
obj <- MakeADFun(
         data = list(y = y),
         parameters = list(mu = 5),
         DLL = "unknown_mean",
         silent = TRUE)

## compare the log-likelihoods
obj$fn(5); fmu(5)


## [1] 28643.35
## [1] 28643.35


## compare the gradients
obj$gr(5); dfmu(5)


##             [,1]
## [1,] -7513.195
## [1] -7513.195


## quite impressive!
```

Estimate the mean using the AD gradient:

```r
(opt <- nlminb(objective = obj$fn, gradient = obj$gr, start = obj$par))


## $par
##       mu
## 8.005278
##
## $objective
## [1] 17353.73
##
## $convergence
## [1] 0
##
## $iterations
## [1] 2
##
## $evaluations
## function gradient
##        4        2
```

```
##
## $message
## [1] "both X-convergence and relative convergence (5)"
```

which is the same performance as we had with the analytical gradient.
We can also quickly calculate the standard errors:

```
(rep <- sdreport(obj))

## sdreport(.) result
##    Estimate Std. Error
## mu 8.005278       0.02
## Maximum gradient component: 1.475259e-09
```

2.2 A random effects likelihood

Once we have random effects we need to integrate to get the marginal likelihood so it gets more complex. For example, the marginal likelihood for a Gaussian random intercepts model is

$$\prod_{i=1}^{m} \int \prod_{j=1}^{m} \mathrm{N}(y_{ij}|\mu, u_i, \sigma^2, \tau^2)\mathrm{N}(u_i|\tau^2)du_i. \tag{13}$$

Clearly, it's going to get more complex or impossible to obtain an analytical solution for the gradient (non-linear functions). Here we can use TMB to good effect.

```
#include <TMB.hpp>
template<class Type>
Type objective_function<Type>::operator() () {
  // DATA
  DATA_VECTOR(y);
  DATA_IVECTOR(gps);
  DATA_INTEGER(ngps);
  // PARAMETERS:
  PARAMETER(mu);
  PARAMETER(logsigma);
  PARAMETER(logtau);
  PARAMETER_VECTOR(u);
  // PRELIMINARY CALCULATIONS
  int n = y.size(); // number of observations
  Type sigma = exp(logsigma);
  Type tau = exp(logtau);
  vector<Type> gp_means(ngps);
  // PROCEDURE
  Type nll = 0.0; // initialize negative log likelihood
  // likelihood of the random effects
  for(int j = 0; j < ngps; j++){
    nll -= -log(tau) - log(2.0 * M_PI) / 2.0 -
           pow(u(j), 2.0) / (2.0 * pow(tau, 2.0));
    gp_means(j) = mu + u(j);
  }
  // loop over the observations
  for(int i = 0; i < n; i++){
    nll -= -log(sigma) - log(2.0 * M_PI) / 2.0 -
           pow(y(i) - (mu + u(gps(i))), 2.0) / (2.0 * pow(sigma, 2.0));
  }
  ADREPORT(gp_means);
  return nll;
}
```

Compile and load the functions

```
compile("random_intercepts.cpp")
```

```
## load the function
dyn.load(dynlib("random_intercepts"))
```

Generate some random intercepts data

```
ngps <- 20

gp.means <- rnorm(ngps, mean = 8, sd = 2)

y <- c(sapply(gp.means,
              FUN = function(x){rnorm(10, mean = x, sd = 0.5)})
       )
gps <- rep(0:(ngps - 1), each = 10)
library(ggplot2)

dat <- data.frame(gps, y)
```

Fit the data in TMB

```
obj <- MakeADFun(
    data = list(y = y, gps = gps, ngps = ngps),
    parameters =
        list(mu = 5, logsigma = 0.1, logtau = 0.1, u = rep(mean(y), ngps)),
    random = "u",
    DLL = "random_intercepts",
    hessian = TRUE,
    silent = TRUE)

opt <- nlminb(objective = obj$fn, gradient = obj$gr, start = obj$par)

rep <- sdreport(obj)
```

```
rep.summ <- summary(rep)
gp.means <- rep.summ[rownames(rep.summ) == "gp_means",]
row.names(gp.means) <- NULL
pred.dat <- as.data.frame(cbind(gps = 0:(ngps-1), gp.means))
names(pred.dat)[3] <- "SE"
## compare to lme4 in R
library(lme4)

## Loading required package:  Matrix

lme.fit <- lmer(y ~ 1 + (1|gps), REML = FALSE)

## TMB values
c(opt$par[1], exp(opt$par[2:3])); -opt$objective

##        mu  logsigma    logtau
## 7.9942192 0.4785774 1.9092310
## [1] -187.1619

## lme4 values
lme.fit

## Linear mixed model fit by maximum likelihood  ['lmerMod']
## Formula: y ~ 1 + (1 | gps)
##       AIC       BIC    logLik  deviance  df.resid
##  380.3239  390.2188 -187.1619  374.3239       197
## Random effects:
##  Groups   Name        Std.Dev.
##  gps      (Intercept) 1.9093
##  Residual             0.4786
## Number of obs: 200, groups:  gps, 20
## Fixed Effects:
## (Intercept)
##       7.994
```

Plot the data and random effects

```
theme_set(theme_bw())
ggplot(dat, aes(x = gps, y = y)) + geom_point(col = "slategrey", pch = 1) +
  xlab("Group") +
  geom_point(data = pred.dat, aes(x = gps, y = Estimate), col = "purple", size = 2) +
  geom_errorbar(
    data = pred.dat, aes(x = gps, y = Estimate,
      ymin = Estimate - 2 * SE, ymax = Estimate + 2 * SE),
    width = 0.4, col = "purple")

## Warning:  Ignoring unknown aesthetics:  y
```

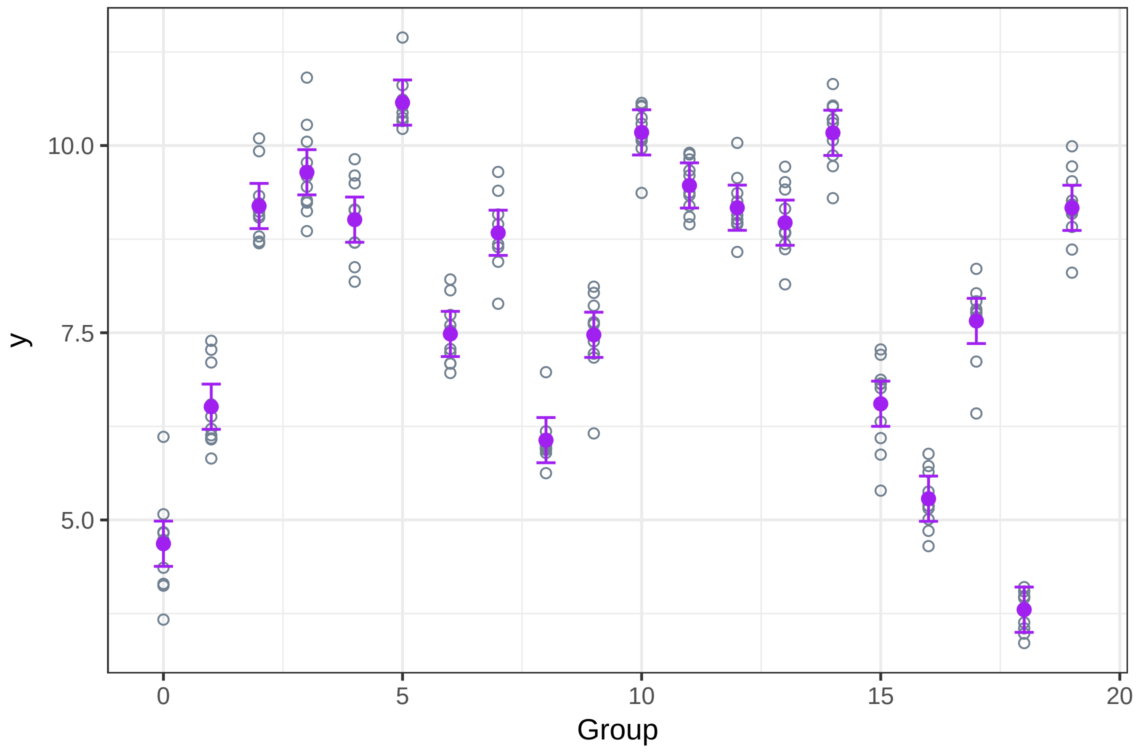

Figure 3: Random intercept data (grey points) and TMB-estimated mixed effect means and approximate 95% confidence intervals.

Now move to dynamic Poisson file