

Random effects in AD Model Builder

Anders Nielsen & Arni Magnusson

About random effect models

- In purely fixed effects models we have
 - Random variables we observe
 - Model parameters we want to estimate
- In random effects models we have
 - Random variables we observe
 - Random variables we do NOT observe
 - Model parameters we want to estimate
- This model class is very useful and goes by many names: random effects models, mixed models, latent variable models, state-space models, frailty models, hierarchical models, ...
- Many tools can handle linear Gaussian models.
- No other tool handles non-linear non-Gaussian random effect models like ADMB

Example: Paired observations

- Two methods A and B to measure blood cell count (to check for the use of doping).
- Paired study.

Person ID	Method A	Method B
1	5.5	5.4
2	4.4	4.9
3	4.6	4.5
4	5.4	4.9
5	7.6	7.2
6	5.9	5.5
7	6.1	6.1
8	7.8	7.5
9	6.7	6.3
10	4.7	4.2

- It must be expected that two measurements from the same person are correlated, so a paired t-test is the correct analysis
- The t-test gives a p-value of 5.1%, which is a borderline result...
- But more data is available

- In addition to the planned study 10 persons were measured with only one method
- Want to use all data, which is possible with random effects

- Assume these 20 are randomly selected from a population where the blood cell count is normally distributed

- Consider the following model:

$$C_i = \alpha(M_i) + B(P_i) + \varepsilon_i, \quad i = 1 \dots 30$$

$\alpha(M_i)$ the 2 fixed method effects

$B(P_i) \sim \mathcal{N}(0, \sigma_P^2)$ the 20 random effects

$\varepsilon_i \sim \mathcal{N}(0, \sigma_R^2)$ measurement noise

All $B(P_i)$ and ε_i are assumed independent

- This model uses all data and gives a 95% c. i. for the method bias $\alpha(A) - \alpha(B)$ which is: (0.04; 0.41).

Person ID	Method A	Method B
1	5.5	5.4
2	4.4	4.9
3	4.6	4.5
4	5.4	4.9
5	7.6	7.2
6	5.9	5.5
7	6.1	6.1
8	7.8	7.5
9	6.7	6.3
10	4.7	4.2
11		5.1
12		4.4
13		4.5
14		5.3
15		7.5
16	5.7	
17	6.0	
18	7.5	
19	6.5	
20	4.2	

- Notice that now there is a (slightly) significant method bias.

```

#No rows
30
#No cols
3
#The obs matrix
#P M C
1 1 5.5
2 1 4.4
3 1 4.6
4 1 5.4
5 1 7.6
6 1 5.9
7 1 6.1
8 1 7.8
9 1 6.7
10 1 4.7
16 1 5.7
17 1 6
18 1 7.5
19 1 6.5
20 1 4.2
1 2 5.4
2 2 4.9
3 2 4.5
4 2 4.9
5 2 7.2
6 2 5.5
7 2 6.1
8 2 7.5
9 2 6.3
10 2 4.2
11 2 5.1
12 2 4.4
13 2 4.5
14 2 5.3
15 2 7.5

```

```

DATA_SECTION
  init_int nrow;
  init_int ncol;
  init_matrix obs(1,nrow,1,ncol);

  vector C(1,nrow);
  ivector P(1,nrow);
  ivector M(1,nrow);

  !! C=column(obs,3);
  !! P=(ivector)column(obs,1);
  !! M=(ivector)column(obs,2);
PARAMETER_SECTION
  init_number logSigmaP;
  init_number logSigmaR;
  init_vector alpha(1,2);

  random_effects_vector B(1,20);

  sdreport_number sigmaP;
  sdreport_number sigmaR;
  sdreport_number diffAB;
  vector pred(1,nrow);
  objective_function_value nll;
PROCEDURE_SECTION
  sigmaR=exp(logSigmaR);
  sigmaP=exp(logSigmaP);
  dvariable ss;

  nll=0.0;
  ss=square(sigmaR);
  for(int i=1; i<=nrow; ++i){
    pred(i)=alpha(M(i))+B(P(i));
    nll+=0.5*(log(2*M_PI*ss)+square(C(i)-pred(i))/ss);
  }
  ss=square(sigmaP);
  for(int i=1; i<=20; ++i){
    nll+=0.5*(log(2*M_PI*ss)+square(B(i))/ss);
  }
  diffAB=alpha(1)-alpha(2);

```

Random effects in AD Model Builder

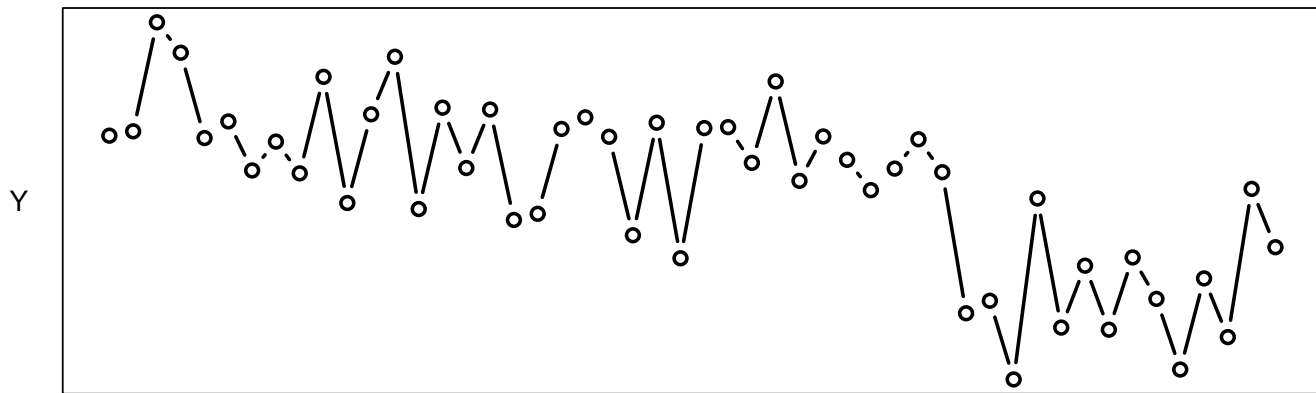
- In random effects models we have
 - Random variables we observe: x
 - Random variables we do not observe: z
 - Model parameters we want to estimate: θ
- If we had observed x and z we would have a joint likelihood $L(x, z, \theta)$
- but z is unobserved so we have to estimate θ in the marginal likelihood:

$$L(x, \theta) = \int L(x, z, \theta) dz$$

- This requires a high dimensional integral — which is difficult
- This is (part of) the reason MCMC methods are so widely used
- MCMC can be slow, difficult to judge convergence, and in tools like winBugs a prior must be assigned to everything — even when you have no prior information.
- AD Model Builder has a better solution

Simplest example: Random walk plus noise

- Observation vector Y generated from:
 - $\lambda_i = \lambda_{i-1} + \eta_i$
 - $Y_i = \lambda_i + \varepsilon_i$
 - where $i = 1 \dots 50$, $\eta_i \sim \mathcal{N}(0, \sigma_\lambda^2)$, and $\varepsilon_i \sim \mathcal{N}(0, \sigma_Y^2)$ all independent.



- Notice λ vector unobserved. Here we wish to estimate both λ and the model parameters ($\lambda_o, \sigma_\lambda$, and σ_ε)

Basic Kalman Filter

```
DATA_SECTION
  init_int N
  init_vector y(1,N)

PARAMETER_SECTION
  init_number logSdLam
  init_number logSdy
  init_number lam0

  vector lam(1,N)
  vector lamVar(1,N)
  vector lamPred(1,N)
  vector lamPredVar(1,N)
  vector lamSmooth(1,N)
  vector lamSmoothVar(1,N)
  vector yPredVar(1,N)
  vector w(1,N)
  number varLam
  number varY
  number varFrac
  objective_function_value nll
```

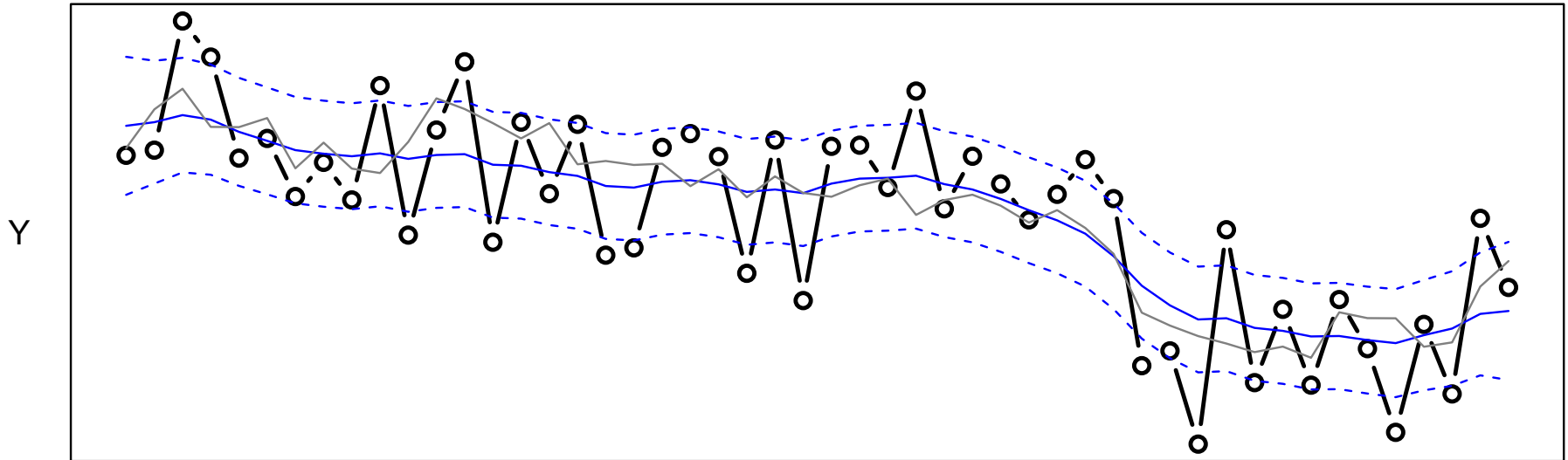
```
PROCEDURE_SECTION
  varLam=exp(2.0*logSdLam);
  varY=exp(2.0*logSdy);

  lamPred(1)=lam0;
  lamPredVar(1)=varLam+1000.0;
  yPredVar(1)=lamPredVar(1)+varY;
  w(1)=y(1)-lamPred(1);
  lam(1)=lamPred(1)+lamPredVar(1)/yPredVar(1)*w(1);
  lamVar(1)=lamPredVar(1)-lamPredVar(1)/yPredVar(1)*lamPredVar(1);
  nll+=0.5*log(yPredVar(1))+0.5*w(1)/yPredVar(1)*w(1);

  for(int i=2; i<=N; ++i){
    lamPred(i)=lam(i-1);
    lamPredVar(i)=lamVar(i-1)+varLam;
    yPredVar(i)=lamPredVar(i)+varY;
    w(i)=y(i)-lamPred(i);
    lam(i)=lamPred(i)+lamPredVar(i)/yPredVar(i)*w(i);
    lamVar(i)=lamPredVar(i)-lamPredVar(i)/yPredVar(i)*lamPredVar(i);
    nll+=0.5*log(yPredVar(i))+0.5*w(i)/yPredVar(i)*w(i);
  }

REPORT_SECTION
  lamSmooth(N)=lam(N);
  lamSmoothVar(N)=lamVar(N);
  for(int i=N-1; i>=1; --i){
    varFrac=lamVar(i)/lamPredVar(i+1);
    lamSmooth(i)=lam(i)+varFrac*(lamSmooth(i+1)-lamPred(i+1));
    lamSmoothVar(i)=lamVar(i)+
      varFrac*(lamSmoothVar(i+1)-lamPredVar(i+1))*varFrac;
  }
  for(int i=1; i<=N; ++i){
    report<<lamSmooth(i)<<" "<<sqrt(lamSmoothVar(i))<<endl;
  }
```


Estimation from the Kalman Filter



- Fast for linear and Gaussian state-space models
- Must be extended to deal with non-linear models
- Non-Gaussian models are difficult and need either simulation based filters or discrete approximation of state-space (only possible in low dimension)
- A different solution will now be demonstrated

Deriving the Laplace approximation

- Want to compute the marginal likelihood for a given θ value:

$$L(x, \theta) = \int L(x, z, \theta) dz$$

- Second order Taylor approximation of $\log(L(x, z, \theta))$ around \hat{z}_θ

$$\ell(\theta, z, Y) \approx \ell(\theta, \hat{z}_\theta, Y) - \frac{1}{2}(z - \hat{z}_\theta)^t (-\ell''_{zz}(\theta, z, Y)|_{z=\hat{z}_\theta}) (z - \hat{z}_\theta)$$

- Using this approximation the integral becomes

$$\int L(\theta, z, Y) dz \approx \sqrt{\frac{(2\pi)^n}{\det(-\ell''_{zz}(\theta, z, Y)|_{z=\hat{z}_\theta})}} \exp(\ell(\theta, \hat{z}_\theta, Y))$$

This can be seen by realizing that $\exp(\ell(\theta, \hat{z}_\theta, Y))$ is a constant w.r.t. z , and the remaining part is recognized as the normalizing constant of a multivariate normal distribution. Here n is the dimension of z .

Laplace approximation in practice

- Want to compute the marginal likelihood for a given θ value:

$$L(x, \theta) = \int L(x, z, \theta) dz$$

- First the joint likelihood $L(x, z, \theta)$ is optimized w.r.t. z .
- This optimization yields \hat{z}_θ , and hessian $\mathcal{H}(\hat{z}_\theta) = -\ell''_{zz}(\theta, z, Y)|_{z=\hat{z}_\theta}$.
- Next a Gaussian approximation is assumed and the result (apart from a constant) is:

$$L(x, \theta) \approx |\det(\mathcal{H}(\hat{z}_\theta))|^{-0.5} L(x, \hat{z}_\theta, \theta)$$

- Notice that when defined in this way \hat{z}_θ and $\mathcal{H}(\hat{z}_\theta)$ and also depend on θ , which makes AD of this pretty difficult, but all solved for us in AD Model Builder.
- Actually this is all very simple to use. All we have to do is:
 - Code up the joint negative log likelihood
 - declare as `random_effects_vector z(1,n);`

```

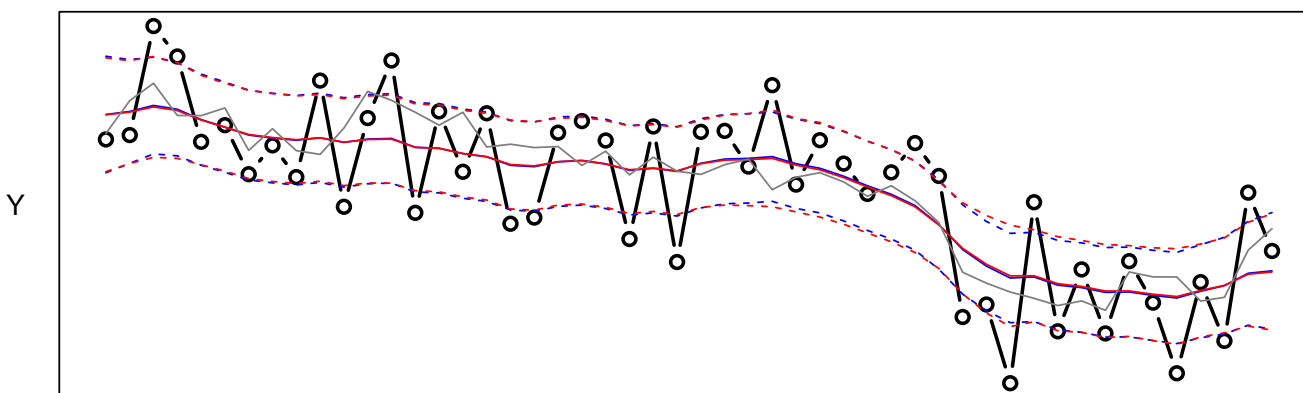
DATA_SECTION
  init_int N
  init_vector y(1,N)

PARAMETER_SECTION
  init_number logSdLam
  init_number logSdy
  init_number lam0
  random_effects_vector lam(1,N);
  objective_function_value jnll;

PROCEDURE_SECTION
  jnll=0.0;
  dvariable var=exp(2.0*logSdLam);
  jnll+=0.5*(log(2.0*M_PI*var)
            +square(lam(1)-lam0)/var);
  for(int i=2; i<=N; ++i){
    jnll+=0.5*(log(2.0*M_PI*var)
              +square(lam(i)-lam(i-1))/var);
  }
  var=exp(2.0*logSdy);
  for(int i=1; i<=N; ++i){
    jnll+=0.5*(log(2.0*M_PI*var)
              +square(lam(i)-y(i))/var);
  }
TOP_OF_MAIN_SECTION
  gradient_structure::set_MAX_NVAR_OFFSET(3000);

```

index	name	value	std dev
1	logSdLam	-2.9761e-01	3.3591e-01
2	logSdy	8.1153e-01	1.1816e-01
3	lam0	9.3888e-01	1.4057e+00
4	lam	9.3882e-01	1.1935e+00
5	lam	1.0511e+00	1.0778e+00
6	lam	1.2683e+00	1.0567e+00
7	lam	1.1166e+00	1.0029e+00
8	lam	7.1842e-01	9.3777e-01
9	lam	4.1948e-01	9.2039e-01
10	lam	1.1140e-01	9.2432e-01
:			
44	lam	-6.1435e+00	9.8562e-01
45	lam	-6.2583e+00	9.6325e-01
46	lam	-6.4506e+00	9.7474e-01
47	lam	-6.4497e+00	9.5186e-01
48	lam	-6.5848e+00	9.7301e-01
49	lam	-6.6817e+00	1.0032e+00
50	lam	-6.4287e+00	9.7369e-01
51	lam	-6.2137e+00	9.9551e-01
52	lam	-5.7458e+00	1.0877e+00
53	lam	-5.6501e+00	1.2210e+00



More efficient coding

```
DATA_SECTION
  init_int N
  init_vector y(1,N)

PARAMETER_SECTION
  init_number logSdLam
  init_number logSdy
  init_number lam0
  random_effects_vector lam(1,N);
  objective_function_value jnll;

PROCEDURE_SECTION
  jnll=0.0;

  step(lam0,lam(1),logSdLam);

  for(int i=2; i<=N; ++i){
    step(lam(i-1),lam(i),logSdLam);
  }

  for(int i=1; i<=N; ++i){
    obs(lam(i),logSdy,i);
  }
```

```
SEPARABLE_FUNCTION void step(const dvariable& lam1, const dvariable& lam2, const dvariable& logSdLam)
  dvariable var=exp(2.0*logSdLam);
  jnll+=0.5*(log(2.0*M_PI*var)+square(lam2-lam1)/var);
```

```
SEPARABLE_FUNCTION void obs(const dvariable& lam, const dvariable& logSdy, int i)
  dvariable var=exp(2.0*logSdy);
  jnll+=0.5*(log(2.0*M_PI*var)+square(lam-y(i))/var);
```

- The idea is to reduce the likelihood calculation to a sum of function calls, where each call only uses a few random effects.
- Each function call must include the parameters needed, and the random effects needed, and not much more (no need to pass data)
- Function headers must be one line — even when they get too long.

Example: Discrete valued time series

- One of the examples from the AD Model Builder site (from Kuk & Cheng (1999)).
- The model:

$$y_i \sim \text{Pois}(\lambda_i) , \text{ where}$$

$$\log(\lambda_i) = X_i b + u_i , \text{ and}$$

$$u_i = a u_{i-1} + \varepsilon_i$$

Here, X_i is a covariate vector, b is a vector of regression parameters and u_i is an AR(1). The dimension of b is 6 and $i=1, \dots, 168$.

	β_1	β_2	β_3	β_4	β_5	β_6	a	σ
ADMB-RE	0.242	-3.81	0.162	-0.482	0.413	-0.0109	0.627	0.538
Std. dev.	0.270	2.76	0.15	0.16	0.13	0.13	0.19	0.15
Kuk & Cheng	0.244	-3.82	0.162	-0.478	0.413	-0.0109	0.665	0.519

Discrete valued time series code

```
DATA_SECTION
  init_int n
  init_vector y(1,n)
  init_int p
  init_matrix X(1,n,1,p)

PARAMETER_SECTION
  init_vector b(1,p,1)
  init_bounded_number a(-1,1,2)
  init_number log_sigma(2)
  random_effects_vector u(1,n,2)
  objective_function_value g

PROCEDURE_SECTION
  g=0.0; int i;

  sf1(log_sigma,a,u(1));

  for (i=2;i<=n;i++){
    sf2(log_sigma,a,u(i),u(i-1),i);
  }

  for (i=1;i<=n;i++){
    sf3(u(i),b,i);
  }

SEPARABLE_FUNCTION void sf1(const dvariable& ls,const dvariable& aa,const dvariable& u_1)
  g += ls - 0.5*log(1-square(aa)) +0.5*square(u_1/exp(ls))*(1-square(aa));

SEPARABLE_FUNCTION void sf2(const dvariable& ls, const dvariable& aa,const dvariable& u_i,const dvariable& u_i1)
  g += ls +.5*square((u_i-aa*u_i1)/exp(ls));

SEPARABLE_FUNCTION void sf3(const dvariable& u_i ,const dvar_vector& bb, int i)
  dvariable eta = X(i)*bb + u_i;
  dvariable lambda = exp(eta);
  g -= y(i)*eta - lambda;

TOP_OF_MAIN_SECTION
  gradient_structure::set_MAX_NVAR_OFFSET(1000);
```

Non-Gaussian random effects

- If the random effects are non-Gaussian, then the Laplace approximation may be inaccurate.
- Can use transformation $g = F^{-1}(\Phi(u))$, where $u \sim \mathcal{N}(0, 1)$.
- E.g. part of a larger example:

```
PARAMETER_SECTION
  ...
  random_effects_vector u(1,nh,2)
PROCEDURE_SECTION
  ...
  for (i=1;i<=nh;i++){
    fun(i,j,u(i),log_theta1,beta);
  }
SEPARABLE_FUNCTION void fun( int i,int & j ,const prevariable& ui, const prevariable& log_theta1,
  f += 0.9189385 + 0.5*square(ui);           // N(0,1) likelihood contribution from u's
  ...
  dvariable z=cumd_norm(ui);                 // z has uniform (0,1) distribution
  z = 0.99999999*z + 0.000000005;           // To gain numerical stability
  dvariable gi = theta1*inv_cumd_gamma(z,1.0/theta1);
  ...
```

- In situations where we fear the Laplace approximation may be inaccurate, we can improve it by **importance sampling**. Simply by:

```
./model -is 100
```


REML via random effects?

- The following non-linear model is assumed to describe the relation between density D within pot and yield Y per plant:

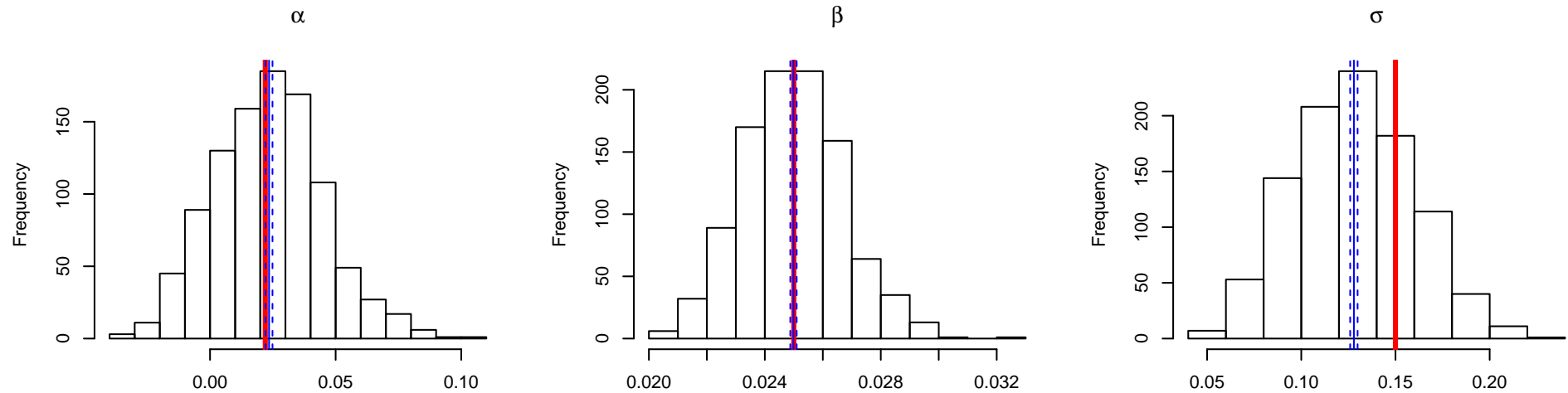
$$\log(Y_i) = -\log(\alpha + \beta D_i) + \varepsilon_i, \quad \text{where } \varepsilon_i \sim \mathcal{N}(0, \sigma^2)$$

```
DATA_SECTION
  init_int N
  init_vector density(1,N)
  init_vector yield(1,N)
  vector logYield(1,N)
  !! logYield=log(yield);
PARAMETER_SECTION
  init_number logA
  init_number logB
  init_number logSigma
  objective_function_value nll
  sdreport_number a
  sdreport_number b
  sdreport_number sigma
  vector pred(1,N)
  number ss
PROCEDURE_SECTION
  b=exp(logB);
  a=exp(logA)-b*min(density);
  sigma=exp(logSigma);
  ss=square(sigma);
  pred=-log(a+b*density);
  nll=0.5*(N*log(2*M_PI*ss)+
    sum(square(logYield-pred))/ss);
```

```
#N
10
#density
5 7 10 15 25 34 51 77 115 173
#yield
6.97 5.569 2.814 2.401 1.89 1.124 0.623 0.592 0.382 0.204
```

index	name	value	std dev
1	logA	-1.9044e+00	1.0966e-01
2	logB	-3.6793e+00	6.7797e-02
3	logSigma	-1.9246e+00	2.2361e-01
4	a	2.2708e-02	2.1107e-02
5	b	2.5241e-02	1.7113e-03
6	sigma	1.4594e-01	3.2633e-02

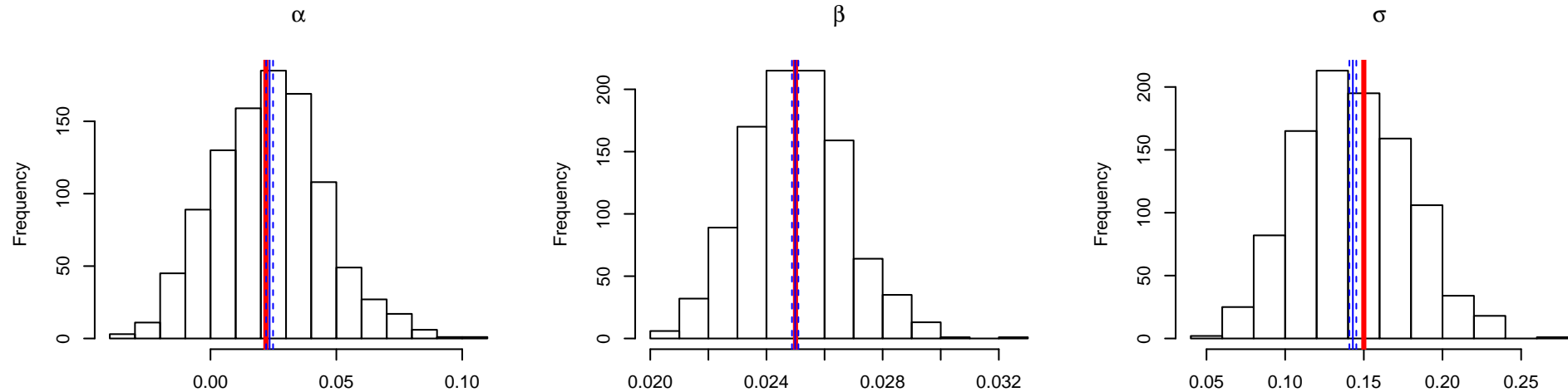
Simulation study



Parameter	True value	low	high
α	0.022	0.022	0.025
β	0.025	0.0249	0.0251
σ	0.150	0.126	0.13

This is an expected result, but can we fix it?

Results with random effects (flat prior) on α and β



Parameter	True value	low	high
α	0.022	0.022	0.025
β	0.025	0.0249	0.0251
σ	0.150	0.141	0.145

Almost!

What else?

- Large collection of examples at <http://www.otter-rsch.com/admbre/examples.html>
- It is possible to add priors on parameters (see exercise)
- The quality of the approximation can be checked and improved by importance sampling — without additional coding!
- Lots of flags for optimizing performance e.g. memory options — see manual.
- ...

Exercise: Random effect logistic regression

- Read through the example at:
<http://mathstat.helsinki.fi/openbugs/Examples/Seeds.html>
- ? Implement the same model in ADMB, but without priors on the hyper parameters.
- ? Compare results.
- The data for this exercise is:

```
#N
21
#r
10 23 23 26 17 5 53 55 32 46 10 8 10 8 23 0 3 22 15 32 3
#n
39 62 81 51 39 6 74 72 51 79 13 16 30 28 45 4 12 41 30 51 7
#x1
0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
#x2
0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1
```

Solution

DATA_SECTION

```
init_int N;
init_vector r(1,N);
init_vector n(1,N);
init_vector x1(1,N);
init_vector x2(1,N);
```

PARAMETER_SECTION

```
init_number alpha0
init_number alpha1
init_number alpha2
init_number alpha12
init_number logSigma
```

```
random_effects_vector B(1,N)
```

```
sdreport_number sigma
```

```
vector logitp(1,N)
```

```
vector p(1,N)
```

```
objective_function_value jnll
```

PROCEDURE_SECTION

```
sigma=exp(logSigma);
logitp=alpha0+alpha1*x1+alpha2*x2+alpha12*elem_prod(x1,x2)+B;
p=elem_div(exp(logitp),(1.0+exp(logitp)));
jnll=0.0;
for(int i=1; i<=N; ++i){
    jnll += -log_comb(n(i),r(i))-log(p(i))*r(i)-log(1.0-p(i))*(n(i)-r(i));
    jnll += 0.5*(log(2.0*M_PI*sigma)+square(B(i)/sigma));
}
```

index	name	value	std dev
1	alpha0	-5.4849e-01	1.6611e-01
2	alpha1	9.7427e-02	2.7739e-01
3	alpha2	1.3368e+00	2.3623e-01
4	alpha12	-8.1003e-01	3.8422e-01
5	logSigma	-1.4497e+00	4.6691e-01
6	B	-1.5854e-01	2.2444e-01
7	B	9.0476e-03	1.8999e-01
8	B	-1.8273e-01	2.0536e-01
9	B	2.4140e-01	2.3428e-01
10	B	9.9434e-02	2.0804e-01
11	B	4.5013e-02	2.3020e-01
12	B	6.2799e-02	1.8985e-01
13	B	1.6606e-01	2.0487e-01
14	B	-1.0436e-01	2.0653e-01
15	B	-2.3195e-01	2.2041e-01
16	B	5.0781e-02	2.2308e-01
17	B	8.0648e-02	2.2664e-01
18	B	-6.6290e-02	2.1167e-01
19	B	-1.1708e-01	2.2198e-01
20	B	1.8894e-01	2.3589e-01
21	B	-8.1461e-02	2.3978e-01
22	B	-1.5248e-01	2.4853e-01
23	B	2.5509e-02	2.0398e-01
24	B	-2.2122e-02	2.0649e-01
25	B	1.7960e-01	2.2983e-01
26	B	-3.1760e-02	2.2604e-01
27	sigma	2.3463e-01	1.0955e-01

Winbugs code

```
model{
  for( i in 1 : N ) {
    r[i] ~ dbin(p[i],n[i])
    b[i] ~ dnorm(0.0,tau)
    logit(p[i]) <- alpha0 + alpha1 * x1[i] + alpha2 * x2[i] +
      alpha12 * x1[i] * x2[i] + b[i]
  }
  alpha0 ~ dnorm(0.0,1.0E-6)
  alpha1 ~ dnorm(0.0,1.0E-6)
  alpha2 ~ dnorm(0.0,1.0E-6)
  alpha12 ~ dnorm(0.0,1.0E-6)
  tau ~ dgamma(0.001,0.001)
  sigma <- 1 / sqrt(tau)
}

list(r = c(10, 23, 23, 26, 17, 5, 53, 55, 32, 46, 10, 8, 10, 8, 23, 0, 3, 22, 15, 32, 3),
      n = c(39, 62, 81, 51, 39, 6, 74, 72, 51, 79, 13, 16, 30, 28, 45, 4, 12, 41, 30, 51, 7),
      x1 = c(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1),
      x2 = c(0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1),
      N = 21
)

list(alpha0 = 0, alpha1 = 0, alpha2 = 0, alpha12 = 0, tau = 10)
```