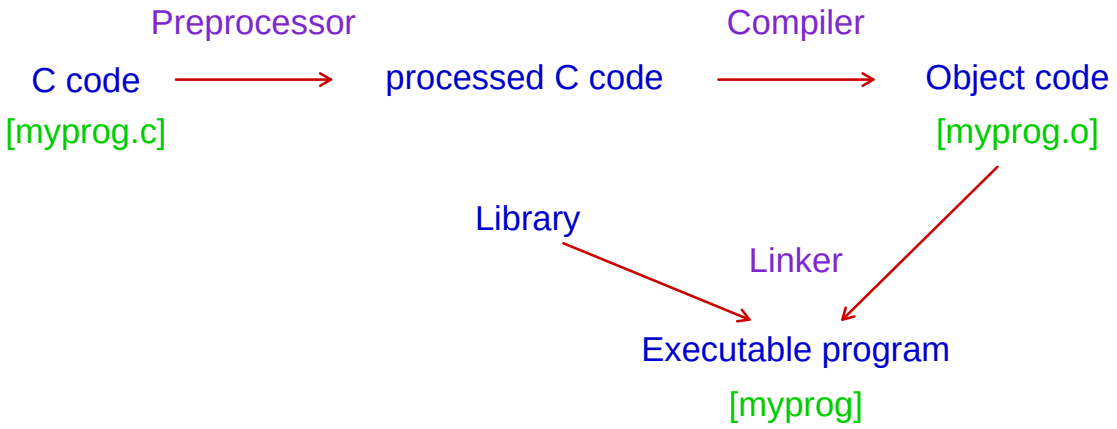
C programming II

Karl W Broman

Department of Biostatistics
Johns Hopkins University

`http://www.biostat.jhsph.edu/~kbroman`

Compilation



The C Preprocessor

...does a fancy search-and-replace on `.c` files before they are sent to the compiler.

1. Include files

```
#include "myheader.h"    /* search current dir'y */
#include <math.h>        /* search standard dir'y */
#include <sys/cdefs.h>
```

The preprocessor replaces these lines with the contents of the files.

You can use the `-I` flag to `gcc` to have other directories searched as well.

```
gcc -I/usr/local/lib/R/include ....
```

The C Preprocessor

2. `#define` statement

```
#define TOL 1e-6
#define ARRAYSIZE 20

int i;
int data[ARRAYSIZE];
int moredata[ARRAYSIZE];

for(i=0; i<ARRAYSIZE; i++)
  if(fabs(data[i]) < TOL) {
    /* do something */
  }
```

Before the code is sent to the compiler, the preprocessor will replace all instances of `TOL` with `1e-6` and all instances of `ARRAYSIZE` with `20`. Note: don't include a semicolon!

See examples of `#define` statements in the header files in `/usr/include` and `/usr/local/lib/R/include`.

The C Preprocessor

2b. Macros via `#define`

You can use `#define` to create macros that take arguments.

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
#define SQR(x) ((x)*(x))
```

`MAX(x*3.0,7.0)` will be replaced with

```
((x*3.0) > (7.0) ? (x*3.0) : (7.0))
```

`SQR(5.0*x + 2.3)` is converted to

```
((5.0*x + 2.3)*(5.0*x + 2.3))
```

Use lots of parentheses!

If we'd written `#define SQR(x) x*x`, then `SQR(5.0*x+2.3)-1.0` would become

```
5.0*x+2.3*5.0*x+2.3 -1.0
```

rather than `((5.0*x+2.3)*(5.0*x+2.3))-1.0`

The C Preprocessor

You can see the results of the preprocessor by typing

```
gcc -E myprog.c | more
```

Note that you can do `#define` at the unix prompt at compilation, as follows:

```
gcc -DTOL=1e-6 -DARRAYSIZE=20 myprog.c
gcc -DDEBUG myprog.c
```

The C Preprocessor

3. Conditional compilation

```
#ifdef DEBUG
    for(k1=0; k1<nparm1; k1++) {
        for(k2=0; k2<nparm2; k2++)
            printf("%8.3lf ", work[k1+k2*nparm1]);
        printf("\n");
    }
#endif /* DEBUG */
```

Only if `DEBUG` has been defined will this code get compiled.

If it's not defined, it'll get deleted before being sent to the compiler.

Define `DEBUG` by:

- `#define DEBUG`
- `gcc -DDEBUG myprog.c`

The C Preprocessor

3. Conditional compilation

```
#ifndef ARRAYSIZE
    #define ARRAYSIZE 20
#endif



#ifdef DEBUG
    printf("Debugging stuff\n");
#else
    printf("Non-debugging stuff\n");
#endif
```

The C Preprocessor

3. Conditional compilation

It's especially useful for commenting out a section of code.

The following doesn't work:

```
/******** commented out this section *****************
      for(k1=0; k1<n_gen1-1; k1++) { /* QTL1 x intxn */
        for(k2=0; k2<n_gen2; k2++)
          Wts12[k1][k2][i] += param[k1+s]*Intcov[j][i];
      }
********* end of commented out bit ******************/
```

It's better to do the following:

```
#ifdef UNDEF
      for(k1=0; k1<n_gen1-1; k1++) { /* QTL1 x intxn */
        for(k2=0; k2<n_gen2; k2++)
          Wts12[k1][k2][i] += param[k1+s]*Intcov[j][i];
      }
#endif /* UNDEF */
```

Example functions

```
double getmax(double *x, int n, int *index)
{
    int i, imax;
    double xmax;

    imax = 0;
    xmax = x[0];

    for(i=1; i<n; i++) {
        if(x[i] > xmax) {
            xmax = x[i];
            imax = i;
        }
    }

    *index = imax;
    return(xmax);
}
```

```
themax = getmax(x, n, &i);
```

Example functions

```
double sum(double *x, int n)
{
    int i;
    double xs;

    xs=0.0;
    for(i=0; i<n; i++) xs += x[i];

    return(xs);
}
```

```
double sum(double *x, int n)
{
    double xs;

    xs=0.0;
    while(n--) { xs += *(x++); }

    return(xs);
}
```

Example functions

```
double *dvector(int n)
{
    double *x;

    x = (double *)calloc(n, sizeof(double));

    if(x==NULL) {
        printf("Error: no space for %d doubles\n", n);
        exit(1);
    }

    return(x);
}
```

```
int n;
double *x;

n=20;
x = dvector(n);
```

Example functions

```
double **reorg_dmatrix(double *x, int nrow, int ncol)
{
  int i;
  double **newx;

  newx = (double **)calloc(ncol, sizeof(double *));
  if(newx==NULL) {
    printf("Error: out of memory in reorg_dmatrix\n");
    exit(1);
  }

  newx[0] = x;
  for(i=1; i<nrow; i++) newx[i] = newx[i-1] + nrow;

  return(newx);
}
```

```
int ncol, nrow;    /* prog10.c */
double *x, **X;

nrow=20; ncol=10;
x = dvector(10*20);
X = reorg_dmatrix(x, nrow, ncol);
```

Communication among functions

1. Function arguments
   - Most readable, reliable, reusable.
2. External or global variables
   - Potentially known by all functions.
   - “Hidden means of communication”
   - Use only as a last resort.
3. Static variables
   - Known by all functions in a single source file.

Variable scope

External variables

- Allows communication among all of the functions in a program, regardless of what source files they reside in.
- In one (and only one) source file, declare the variable in the usual way, but outside the curly braces of any function.

```
double *mydata;
```

- In any other source file (in which it is used), declare it using `extern`.

```
extern double *mydata;
```

Variable scope

Static variables (version A)

- Allows communication among all of the functions in a single source file.
- Declare using `static` and outside the scope of any function.

```
static int xmax;
```

- The variable can be hidden from the calling program/function.

Variable scope

Static variables (version B)

- If you declare a variable as static within the scope of a function, the compiler sets aside space for the variable at compile time. The memory location is the same for each function call.

```
int main()
{
    myfunc();

    ...
}
```

```
void mfunc()
{
    static int ncount=0;

    ncount++;
}
```

- Static variables automatically initialized to 0.
- If you do the initialization of a static variable in the declaration, it will be done only once.

An extended example

m groups
y_{ij}, quantitative values, for i=1 . . . m, j=1 . . . n_i
k_{ij}, underlying (unobserved) counts

(k_{ij}, y_{ij}) mutually independent
$k_{ij} \sim$ Poisson(λ_i)
$y_{ij} \mid k_{ij} \sim$ normal($a + bk_{ij}, \sigma^2$)

Goal: MLEs of $\boldsymbol{\theta}$ = (a, b, σ, $\boldsymbol{\lambda}$) by an EM algorithm.

Log likelihood:

$$l(\boldsymbol{\theta}) = \sum_{ij} \log \left\{ \sum_{k} \Pr(k \mid \lambda_i) \Pr(y_{ij} \mid k, a, b, \sigma) \right\}$$

EM algorithm

1. Pick starting values for the parameters, $\hat{\theta}^{(0)}$

2. E-step

$$v_{ij} = E(k_{ij} \mid \hat{\theta}^{(s)}, y_{ij}) = \frac{\sum_k k \Pr(k \mid \hat{\lambda}_i) \Pr(y_{ij} \mid k, \hat{a}_i, \hat{b}_i, \hat{\sigma}_i)}{\sum_k \Pr(k \mid \hat{\lambda}_i) \Pr(y_{ij} \mid k, \hat{a}_i, \hat{b}_i, \hat{\sigma}_i)}$$

$$w_{ij} = E(k_{ij}^2 \mid \hat{\theta}^{(s)}, y_{ij}) = \frac{\sum_k k^2 \Pr(k \mid \hat{\lambda}_i) \Pr(y_{ij} \mid k, \hat{a}_i, \hat{b}_i, \hat{\sigma}_i)}{\sum_k \Pr(k \mid \hat{\lambda}_i) \Pr(y_{ij} \mid k, \hat{a}_i, \hat{b}_i, \hat{\sigma}_i)}$$

3. M-step

$$\hat{\lambda}_i^{(s+1)} = \sum_{j=1}^{n_i} v_{ij}/n_i$$

$$\hat{b}^{(s+1)} = \frac{\sum y_{ij} v_{ij} - (\sum y_{ij})(\sum v_{ij})/n}{\sum w_{ij} - (\sum v_{ij})^2/n}$$

$$\hat{a}^{(s+1)} = (\sum y_{ij} - \hat{b}^{(s+1)} \sum v_{ij})/n$$

$$\hat{\sigma}^{(s+1)} = (\sum y_{ij}^2 + n\hat{a}^2 + \hat{b}^2 \sum w_{ij} - 2\hat{a} \sum y_{ij} - 2\hat{b} \sum y_{ij} v_{ij} + 2\hat{a}\hat{b} \sum v_{ij})/n$$

The code

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "npmix.h"
/*************************************************************
 * npmix_em
 *
 * The main function for the EM algorithm
 *
 * Input:
 *
 *     y         The data referred to as y[group][obs within group]
 *     n_grps    The number of groups ("m")
 *     n_obs     The number of observations per group
 *               [length n_grps]
 *     est       On entry, the starting points for the parameter
 *                   estimates [length n_grps+3]; on exit, the
 *                   estimates themselves.
 *     lnlik     On exit, the value of the log likelihood evaluated
 *                   at the converged values of the estimates.
 *     ek        On exit, E(k | y, est) [same structure as y]
 *     eksq      On exit, E(k^2 | y, est) [same structure as y]
 *
 *************************************************************/
```

```
void npmix_em(double **y, int n_grps, int *n_obs, double *est,
              double *lnlik, double **ek, double **eksq,
              int maxk, int maxit, double tol, int trace)
{
  int i, j, flag, n_est;
  double *cur, cur_lnlik, maxdiff, temp;

  /* allocate space for current value */
  cur = (double *)calloc(n_grps+3, sizeof (double));
  if(cur==NULL) {
    print("Error: cannot allocate space of %d doubles\n", n_grps+3);
    exit(1);
  }

  n_est = n_grps+3;

  for(j=0; j<n_est; j++) cur[j] = est[j];

  flag = 0;
  cur_lnlik=0.;
  for(i=0; i<maxit; i++) {
    npmix_estep(y, n_grps, n_obs, cur, ek, eksq, maxk);
    npmix_mstep(y, n_grps, n_obs, est, ek, eksq);

    /* calculate maximum change in parameter estimates */
    maxdiff = 0.;
    for(j=0; j<n_est; j++) {
      temp = fabs(est[j] - cur[j]);
      if(temp > maxdiff) maxdiff = temp;
    }




    if(trace) { /* print tracing information */
      *lnlik = npmix_lnlik(y, n_grps, n_obs, est, maxk);
      printf("%4d %11.6lf ", i+1, *lnlik-cur_lnlik);
      cur_lnlik = *lnlik;
      if(trace > 1) {
        for(j=0; j<n_est; j++) printf("%7.4lf ", est[j]);
      }
      else printf("%11.6lf", maxdiff);
      printf("\n");
    }

    if(maxdiff < tol) { /* converged */
      flag = 1;
      break;
    }

    for(j=0; j<n_est; j++) cur[j] = est[j];
  } /* end of loop over EM iterations */

  if(!flag) printf("Didn't converge!\n");

  /* re-do E-step */
  npmix_estep(y, n_grps, n_obs, cur, ek, eksq, maxk);

  /* calculate ln likelihood */
  *lnlik = npmix_lnlik(y, n_grps, n_obs, est, maxk);

  free(cur);
}
```

```
void npmix_estep(double **y, int n_grps, int *n_obs,
                 double *est, double **ek, double **eksq,
                 int maxk)
{
  ...
}


void npmix_mstep(double **y, int n_grps, int *n_obs,
                 double *est, double **ek, double **eksq)
{
  ...
}


double npmix_lnlik(double **y, int n_grps, int *n_obs,
                   double *est, int maxk)
{
  ...
}
```

Two options

1. Write a `main()` function.
   - Call function to read data.
   - Call `npmix_em()`
   - Call function to write data.
2. Call directly from R
   - `.C()`
   - `.Call()`

   Read “Writing R Extensions”.

Calling C via `.C()`

All arguments must be pointers. (I stick with `int` and `double`.)

```
output <- .C("R_npmix_em",
             as.double(unlist(y)),
             as.integer(n.grps),
             as.integer(n.obs),
             est=as.double(start),
             lnlik=as.double(0),
             ek=as.double(rep(0,sum(n.obs))),
             eksq=as.double(rep(0,sum(n.obs))),
             as.integer(maxk),
             as.integer(maxit),
             as.double(tol),
             as.integer(trace))
```

None of the arguments can have `NA`'s unless you use `NAOK=TRUE`.

A simpler example

Convolve two finite sequences.

$$\boldsymbol{a} = (a_1, \ldots, a_m) \qquad \boldsymbol{b} = (b_1, \ldots, b_n)$$

$$c_k = \sum_{i,j:i+j=k} a_i b_j \quad \text{for } k = 1, \ldots, m+n-1$$

The C code:

```
void conv(double *a, double *b, int na, int nb,
          double *ab)
{
  int i, j, nab = na + nb - 1;

  for(i=0; i<nab; i++) ab[i] = 0.0;

  for(i=0; i<na; i++)
    for(j=0; j<nb; j++)
      ab[i+j] += a[i]*b[j];
}
```

Wrapper + R code

A “wrapper” for calling the function from R.

```
    void R_conv(double *a, double *b, int *na, int *nb,
                double *ab)
    {
      conv(a, b, *na, *nb, ab);
    }
```

The R function to call it.

```
    conv <- function(a, b)
    {
      if(!is.loaded(symbol.C("R_conv"))) {
        lib.file <- file.path(paste("convolve", .Platform$dynlib.ext, s
        dyn.load(lib.file)
        cat(" -Loaded ", lib.file, "\n")
      }

      na <- length(a); nb <- length(b)

      .C("R_conv",
         as.double(a),
         as.double(b),
         as.integer(na),
         as.integer(nb),
         ab=as.double(rep(0,na+nb-1)))$ab
    }
```

Back to the big example

```
void R_npmix_em(double *y, int *n_grps, int *n_obs, double *est,
                double *lnlik, double *ek, double *eksq,
                int *maxk, int *maxit, double *tol, int *trace)
{
  int i;
  double **p_y, **p_ek, **p_eksq;

  /* reorganize y, ek, eksq as 2-dim arrays */
  p_y = (double **)R_alloc(*n_grps, sizeof(double *));
  p_ek = (double **)R_alloc(*n_grps, sizeof(double *));
  p_eksq = (double **)R_alloc(*n_grps, sizeof(double *));

  p_y[0] = y;
  p_ek[0] = ek;
  p_eksq[0] = eksq;
  for(i=1; i < *n_grps; i++) {
    p_y[i] = p_y[i-1] + n_obs[i-1];
    p_ek[i] = p_ek[i-1] + n_obs[i-1];
    p_eksq[i] = p_eksq[i-1] + n_obs[i-1];
  }

  npmix_em(p_y, *n_grps, n_obs, est, lnlik, p_ek, p_eksq,
           *maxk, *maxit, *tol, *trace);
}
```

Dynamic memory allocation

1. Send a workspace from R.
   `as.double(rep(0, n.work))`
2. "Transient method"
   R takes care of the garbage collection.
   `R_alloc()`, called like `calloc()`.
3. "User controlled"
   Memory is separate from R's; user must free.
   `Calloc()`, `Realloc()`, `Free()`

I recommend method 2.

```
#include <R.h>
```

Printing / errors / warnings

Use `Rprintf()` rather than `printf()`.

```
#include <R_ext/PrtUtil.h>
```

Use `REprintf()` to write to the error stream (`stderr`).

`error()` and `warning()` are the equivalents of `stop()` and `warning()` in R, and work just like `printf()`.

Calling C from R outside a package

a. Create a shared library

```
R CMD SHLIB npmix.c
R CMD SHLIB -o npmix.so *.c
```

Creates a shared library `npmix.so`.

b. Load the shared library with `dyn.load()`.

(Unload it with `dyn.unload()`.)

```
if(!is.loaded(symbol.C("R_npmix_em"))) {
  lib.file <- file.path(paste("npmix", .Platform$dynlib.ext,
  dyn.load(lib.file)
  cat(" -Loaded ", lib.file, "\n")
}
```

(`symbol.C()` doesn't seem strictly necessary.)

Calling C from R within a package

a. Put source code in the `src` directory.

b. Use `PACKAGE="mypackage"` at the end of the `.C()` call.

c. Create a file `zzz.R` in the `R` subdirectory.

```
.First.lib <-
    function(lib, pkg)
        library.dynam("mypackage", pkg, lib)
```

- The code is compiled (and the shared library is built) when the package is installed.
- The shared library is loaded when `library(mypackage)` is executed.

```
npmix.em <-
function(y, start, maxk=30, maxit=1000, tol=1e-6, trace=FALSE)
{
  # check the input
  if(!is.list(y)) stop("y should be a list.")
  if(length(y) < 2) stop("y should have length >= 2.")
  if(length(start) != length(y)+3)
    stop("length(start) should = length(y)+3.")

  # get rid of NA's
  n.na <- sum(is.na(unlist(y)))
  if(n.na > 0) {
    warning("Omitting ", n.na, " NAs from the input, y.")
    y <- lapply(y, function(a) a[!is.na(a)])
  }

  n.grps <- length(y)     # no. groups
  n.obs <- sapply(y, length) # no. data pts per group

  # load the C code
  if(!is.loaded(symbol.C("R_npmix_em"))) {
    lib.file <- file.path(paste("npmix", .Platform$dynlib.ext, sep=""))
    dyn.load(lib.file)
    cat(" -Loaded ", lib.file, "\n")
  }

  output <- .C("R_npmix_em",

               as.double(unlist(y)),
               as.integer(n.grps),
               as.integer(n.obs),
               est=as.double(start),
               lnlik=as.double(0),
               ek=as.double(rep(0,sum(n.obs))),
               eksq=as.double(rep(0,sum(n.obs))),
               as.integer(maxk),
               as.integer(maxit),
               as.double(tol),
               as.integer(trace))

  # pull out just the named bits
  result <- output[c("est","lnlik","ek","eksq")]

  # make ek and eksq lists like the data, y
  ek <- result$ek
  eksq <- result$eksq
  result$ek <- result$eksq <- vector("list",n.grps)
  cur <- 1
  for(i in 1:n.grps) {
    result$ek[[i]] <- ek[cur:(cur+n.obs[i]-1)]
    result$eksq[[i]] <- eksq[cur:(cur+n.obs[i]-1)]
    cur <- cur + n.obs[i]
  }

  names(result$est) <- c("a","b","sigma",
                         paste("lambda_", 1:length(y),sep=""))
  result
}
```

```
#include <math.h>
#include <stdlib.h>
#include <R.h>
#include <Rmath.h>
#include <R_ext/PrtUtil.h>
#include "npmix.h"

void npmix_em( ... ) { ... }

void R_npmix_em( ... ) { ... }

void npmix_estep(double **y, int n_grps, int *n_obs,
                 double *est, double **ek, double **eksq, int maxk)
{
  int i, j, k;
  double a, b, sigma, *lambda, denom, temp, temp1, temp2, temp3;

  a = est[0];
  b = est[1];
  sigma = est[2];
  lambda = est+3;

  for(i=0; i<n_grps; i++) {
    for(j=0; j<n_obs[i]; j++) {

      ek[i][j] = eksq[i][j] = denom = 0.;
      for(k=0; k<maxk; k++) {
        temp = dpois((double)k, lambda[i], 0) *
               dnorm(y[i][j], a+b*(double)k, sigma, 0);




        denom += temp;
        ek[i][j] += (double)k * temp;
        eksq[i][j] += (double)(k*k) * temp;
      }
      ek[i][j] /= denom;
      eksq[i][j] /= denom;

    }
  }
}
```

```
void npmix_mstep(double **y, int n_grps, int *n_obs,
                 double *est, double **ek, double **eksq)
{
  int i, j, k, n;
  double s_y, s_y_ek, s_ysq, s_ek, s_eksq;

  /* calculate the sufficient statistics */
  s_y = s_y_ek = s_ysq = s_ek = s_eksq = 0.;
  n = 0;
  for(i=0; i<n_grps; i++) {
    n += n_obs[i];
    est[i+3] = 0.;

    for(j=0; j<n_obs[i]; j++) {
      est[i+3] += ek[i][j];

      s_y += y[i][j];
      s_y_ek += (y[i][j]*ek[i][j]);
      s_ysq += (y[i][j]*y[i][j]);
      s_ek += ek[i][j];
      s_eksq += eksq[i][j];
    }

    est[i+3] /= (double)n_obs[i];
  }

  est[1] = (s_y_ek - s_y*s_ek/(double)n) /
             (s_eksq - s_ek*s_ek/(double)n);
  est[0] = (s_y - est[1]*s_ek)/(double)n;




  est[2] = sqrt((s_ysq + (double)n*est[0]*est[0] +
                 est[1]*est[1]*s_eksq - 2.*est[0]*s_y -
                 2.*est[1]*s_y_ek + 2.*est[0]*est[1]*s_ek) /
                     (double)n);
}
```

```
double npmix_lnlik(double **y, int n_grps, int *n_obs,
                   double *est, int maxk)
{
  int i, j, k;
  double lnlik, temp, a, b, sigma, *lambda;

  a = est[0];
  b = est[1];
  sigma = est[2];
  lambda = est+3;

  lnlik = 0.;
  for(i=0; i<n_grps; i++) {
    for(j=0; j<n_obs[i]; j++) {

      temp = 0.;
      for(k=0; k<maxk; k++)
        temp += dpois((double)k, lambda[i], 0) *
          dnorm(y[i][j], a+b*(double)k, sigma, 0);

      lnlik += log(temp);
    }
  }

  return(lnlik);
}
```