Optimization problem

- In statistics **Maximum-likelihood estimation (MLE)** is a method of estimation the parameters of statistical model.

- For a fixed set of data and underlying statistical model, MLE selects values of the model parameters that produces a distribution that gives the observed data the greatest probability.

- R has a great build-in function **glm()** that allows to find solution to this problem for many standard distributions

Optimization problem

$$\nabla_\theta \hat{l}(\hat{\theta}|x) = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta \ln f(x_i|\hat{\theta}) = 0$$

- The problem of solving this optimization problem leads to the problem of solving the system of ***n*** equations with ***n*** variables.

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ \ldots & \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \end{aligned}$$

Optimization problem

- One of the well known method to solve this system of equations is a ***Newton – Raphson*** method, which is one of so called Householder's methods in numerical analysis.

- For the function of one variable it is based on the fact that for a differentiable function ***f(x)*** we have the following approximation:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Optimization problem

- Similarly, for the system of ***n*** functions of ***n*** variables:

$$X_{n+1} = X_n - [F'(x_n)]^{-1} F(X_n)$$

- $F'(x_n)$, often called **Jacobean matrix,** is a matrix of first order partial derivatives of all the functions.

Optimization problem

One of the possible ways to implement this algorithm

1. Define a function that calculates values at a given location;
2. Define a function that evaluates a Jacobean matrix;
3. Select a "*best guess*" starting value;
4. Evaluate the function and Jacobean at the current location;
5. Find inverse Jacobean matrix;
6. Calculate the next position;
7. Iterate through steps 4 – 6 until the root is found with desired precision.

optimization.c

- void getF () {…}
- void findJacobian() {…}
- void matrixInverse () {…}
- void matrixVectMult () {…}
- int isConverge() {…}
- main () {…}

optimization.c

Function definition:

We would like to find the intersection between a circle and a parabola.

$$f_1(x, y) = x^2 + y^2 - 4$$
$$f_2(x, y) = y - \frac{1}{3}x^2$$

```
// Return the values of the function at a given location
void getF ( double *X, double *F) {

        F[0] =  X[0] * X[0] + X[1]*X[1] - 4.;
        F[1] = -X[0] * X[0] / 3. + X[1] ;
        return;
}
```

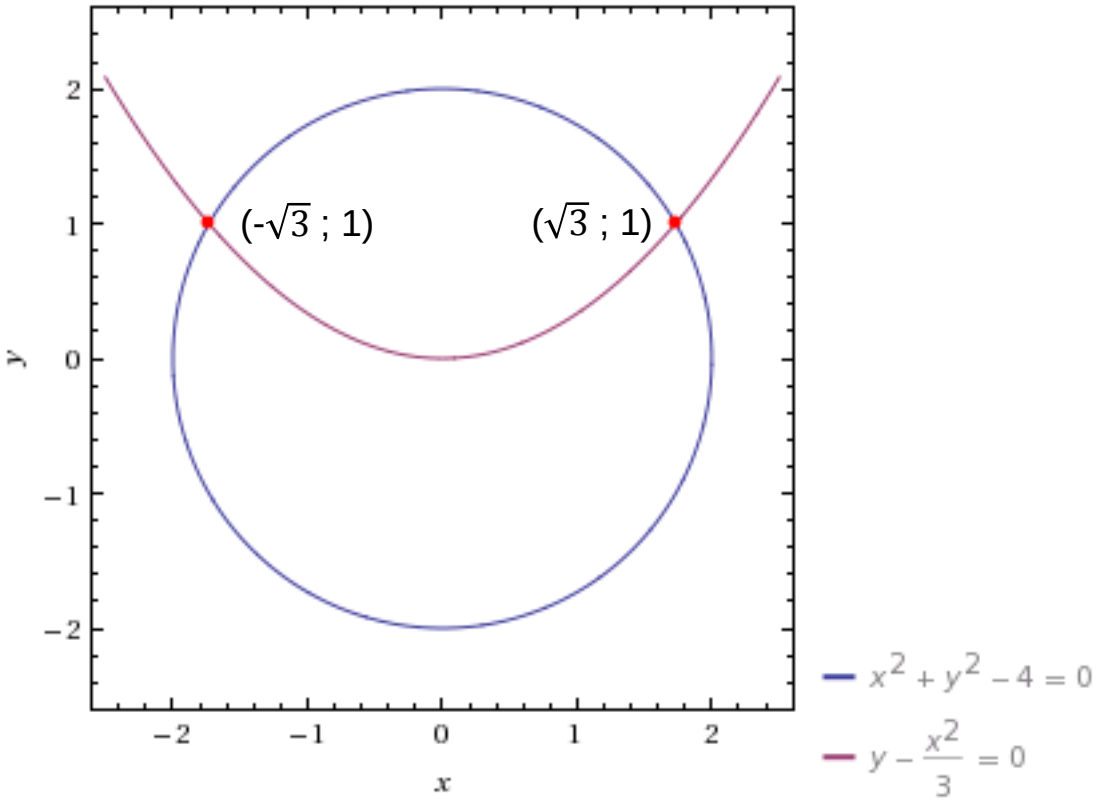
optimization.c

Jacobean matrix:

$$\frac{df_1(x,y)}{dx} = 2x$$
$$\frac{df_1(x,y)}{dy} = 2y$$
$$\frac{df_2(x,y)}{dx} = -\frac{2}{3}x$$
$$\frac{df_2(x,y)}{dy} = 1$$

```
// Evaluate Jacobean matrix
void findJacobian( double *X, double H[2][2]){

        H[0][0] =  2. *  X[0];

        H[0][1] =  2. *  X[1];

        H[1][0] = -2. *  X[0]/3.;

        H[1][1] =  1.;

        return;
}
```

Optimization



Optimization

```
// Calculate inverse matrix using Gauss-Jordan Elimination Process
// Input Matrix: H                ( Jacobean matrix)
// Output Matrix: HT              ( inverse Jacobean matrix)
int matrixInverse( double H[K][K], double HT[K][K]){
 …
}

// Multiply matrix by a vector
// Input Matrix: HT               ( inverse Jacobean matrix)
// Input Vector: F                ( functions' values at a current location)
// Output Vector: delta           ( step toward the solution)
void matrixVectMult( double HT[K][K], double *F, double *delta) {
…
}
```

Optimization

```
#define K   2   // number of variables
#define EPSILON  0.000001


// Check for convergence
int isConverge( double *delta){
    double p = 0;

    for( int i = 0; i < K; i++) p += delta[i] * delta[i];
    return ( pow(p,0.5) < EPSILON)
}
```

Optimization

```
int main (int argc, char **argv){
    …
    int maxSteps = 1000;          // maximum number of steps (to avoid infinite loop )
    double X[2] = {30, 30.};      // starting location
    int n_steps = 0;              // local variable to calculate the number of steps
    do {
            findJacobian(X,H);                  // Jacobian matrix
            matrixInverse(H, HT);               // Inverse matrix
            getF(X, F);                         // value of the function in the new location
            matrixVectMult( HT, F, delta);      // multiply matrix by the  vector

            for ( i=0; i < K; i++) X[i] -= delta[i];        // move to the next location
            n_steps++;                                      // count the number of steps
            if (n_steps == maxSteps) break;  // check if maximum number of steps reached

      } while ( ! isConverge(delta) );                      // check if solution is found
      printf(" Solution: %f %f is found in %d steps\n", X[0], X[1], n_steps);
      return 0;
}
```

Optimization

This algorithm is actually widely used in many different areas, so we can expect, that a well debugged and efficient routine has been already written and might be available. So instead of “reinventing the wheel” we can search for the library that we can simply link to.

One of such libraries that can be used for many *statistics* and *linear algebra* problems is the **GNU Scientific Library** (or **GSL**).

- Written **in C** for numerical calculations in applied mathematics and science
- Can be used for C and C++ projects
- Free software under the GNU General Public License
- Available on both Katana and BlueIce clusters
- Provides over 1000 functions

The GNU Scientific Library

A partial list of the subject areas, covered by the library :

- Random numbers generators
- Random Distributions
- Quasi-Random sequences (in arbitrary dimensions)
- Permutations
- Eigen Systems
- Statistics
- Histograms
- Minimization
- Monte Carlo Integration
- Root-Finding
- Least-Squares Fitting (including non-linear fitting)

…

The GNU Scientific Library

Documentation, download instructions, many examples can be found

on the web:

http://www.gnu.org/software/gsl/

Each function has its own “man” – page.

So lets see how we could solve the same system of equations using this library.

The GNU Scientific Library

We first need to add GSL header files, where all GSL functions are declared:

```
#include <stdlib.h>
#include <stdio.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_multiroots.h>
```

The GNU Scientific Library

```
// define the functions
int rosenbrock_f (const gsl_vector * x, void *params,  gsl_vector * f){
  double a = ((struct rparams *) params)->a;
  double b = ((struct rparams *) params)->b;

  const double x0 = gsl_vector_get (x, 0);
  const double x1 = gsl_vector_get (x, 1);

  const double y0 = a * (x0*x0 + x1*x1 - 4);
  const double y1 = b * (x1 - x0*x0/3.);

  gsl_vector_set (f, 0, y0);
  gsl_vector_set (f, 1, y1);

  return GSL_SUCCESS;
}
```

The GNU Scientific Library

```
int main (void) {
      const gsl_multiroot_fsolver_type *T;
      gsl_multiroot_fsolver *s;
    …
      gsl_multiroot_function f = { &rosenbrock_f, n,  &p };
    …
      double x_init[2] = {30.0, 3.0};
      gsl_vector *x = gsl_vector_alloc (n);
      gsl_vector_set (x, 0, x_init[0]);       gsl_vector_set (x, 1, x_init[1]);

      // this example uses “hybrid” algorithm, others (including Newton’s) are available
      T = gsl_multiroot_fsolver_hybrids;
      s = gsl_multiroot_fsolver_alloc ( T,  2 );
      gsl_multiroot_fsolver_set (s, &f, x);
         …
}
```

The GNU Scientific Library

```
int main (void) {
…
 do {
            iter++;
            status = gsl_multiroot_fsolver_iterate (s);
            if (status)  break                    /* check if solver is stuck */
            status = gsl_multiroot_test_residual (s->f, 1e-7);
       } while (status == GSL_CONTINUE &&  iter < 1000);

     printf ("status = %s\n", gsl_strerror (status));

     gsl_multiroot_fsolver_free (s);
     gsl_vector_free (x);
     return 0;
  }
```

The GNU Scientific Library

Now we need to link the source to the library:

gcc -o gsl_optim **gsl_optim.c -lgsl -lgslcblas**

R and C integration

We would like to use interactive style of R as well as its many easy to use built-in functions and graphics, but take advantage of fast C computations.

It is especially useful for iterative calculations, such as **Monte-Carlo** algorithms. *Metropolis – Hastings* algorithm is one of them.

Metropolis – Hastings algorithm

- In Statistics it is a most popular **Markov chain Monte Carlo** (**MCMC**) method for obtaining a sequence of a random samples from a probability distribution for which direct sampling is difficult
- Can be used to **approximate the distribution** (generate a histogram)
- **Compute an integral** (such as expected value)
- Usually used for **multi-dimensional** spaces
- Was first proposed in **1953** ("… for fast computing machines")
- The **Gibbs** sampling is a special case

Metropolis – Hastings algorithm

- An outline of the algorithm:

- **Initialization:** Start with some value of $\mathbf{X_0}$ (best guess)

Metropolis – Hastings algorithm

- An outline of the algorithm:

- **Initialization:** Start with some value of $\mathbf{X_0}$ (best guess)
- **Proposal Step:** For each *i* –th step sample "candidate" from some proposal distribution $\mathbf{Z} \sim \boldsymbol{q}(\quad z \mid \mathbf{X}(i-1))$.

  - This distribution ***q*** depends on current state of the Markov chain $\mathbf{X_i}$
  - The algorithm works the best if it close in shape to the target distribution
  - Often a normal distribution is used, centered on the current state $\mathbf{X_i}$
  - User provides parameters for the proposed distribution

Metropolis – Hastings algorithm

- An outline of the algorithm:

- **Initialization:** Start with some value of $\mathbf{X_0}$ (best guess)
- **Proposal Step:** For each *i* –th step sample "candidate" from some proposal distribution $\mathbf{Z} \sim \boldsymbol{q}(\ z \mid \mathbf{X}(i-1)\ )$.
- **Acceptance Step:**
  - Calculate acceptance probability:

$$A = \min\{\ 1,\ \frac{P(\mathbf{Z})\ \ \boldsymbol{q}(\mathbf{X}_{i-1} \mid \mathbf{Z}\ )}{P(\mathbf{X}_{i-1})\ \boldsymbol{q}(\mathbf{Z} \mid \mathbf{X}_{i-1})}\ \}$$

  - Accept value with probability A: generate a random number u between 0 and 1: accept $\mathbf{X}_i = \mathbf{Z}$, if $u \le A$ and $\mathbf{X}_i = \mathbf{X}_{i-1}$, otherwise.

Metropolis – Hastings algorithm

- *Possible Enhancements*:

- **Burn-in period**: If the start value is chosen at a very bad location, a number of first iterations should be ignored (let algorithm run for a while before starting collecting values).

- **Lag**: If the acceptance rate of a proposal distribution is very low, the chain can get "stuck" in one location for some time. We can "skip" some number of iterations in between successive samples.

Metropolis – Hastings algorithm

- *Consider Bivariate distribution*:

$$f(x_1, x_2) = \frac{\nu(\nu+1)e^{-x_1}e^{-x_2}}{\left(1+e^{-x_1}+e^{-x_2}\right)^{\nu+2}}$$

Metropolis – Hastings algorithm

- *Consider Bivariate distribution*:

$$f(x_1, x_2) = \frac{\nu(\nu+1)e^{-x_1}e^{-x_2}}{\left(1+e^{-x_1}+e^{-x_2}\right)^{\nu+2}}$$

- *Conditional distribution*:

$$f(x_1|x_2) = \frac{(\nu+1)e^{-x_1}(1+e^{-x_2})^{\nu+1}}{\left(1+e^{-x_1}+e^{-x_2}\right)^{\nu+2}}$$

Metropolis – Hastings algorithm

- *Consider Bivariate distribution*:

$$f(x_1, x_2) = \frac{\nu(\nu+1)e^{-x_1}e^{-x_2}}{\left(1+e^{-x_1}+e^{-x_2}\right)^{\nu+2}}$$

- *For proposal distribution we will use Gaussian*:

$$\mathbf{X} = \mathbf{X^{t-1}} + \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \end{pmatrix}, \quad \varepsilon_\mathbf{j} \sim \mathbf{N}(\mathbf{0}, \mathbf{3^2})$$

Metropolis – Hastings algorithm

To improve numerical stability we rewrite the density function:

$$f(x_1, x_2) = \frac{\nu(\nu+1)}{\left(e^{\frac{x_1+x_2}{\nu+2}} + e^{\frac{x_1+x_2}{\nu+2} - x_1} + e^{\frac{x_1+x_2}{\nu+2} - x_2} \right)^{\nu+2}}$$

Metropolis – Hastings algorithm

To improve numerical stability we rewrite the density function:

$$f(x_1, x_2) = \frac{\nu(\nu+1)}{\left(e^{\frac{x_1+x_2}{\nu+2}} + e^{\frac{x_1+x_2}{\nu+2}-x_1} + e^{\frac{x_1+x_2}{\nu+2}-x_2} \right)^{\nu+2}}$$

Logarithm of density function:

$$\ln f(x_1, x_2) = \ln(\nu) + \ln(\nu+1) -$$

$$(\nu+2) \ln \left(e^{\frac{x_1+x_2}{\nu+2}} + e^{\frac{x_1+x_2}{\nu+2}-x_1} + e^{\frac{x_1+x_2}{\nu+2}-x_2} \right)$$

Metropolis – Hastings algorithm

R Implementation:

$$\ln f(x_1, x_2) = \ln(\nu) + \ln(\nu+1) - (\nu+2) \ln\left(e^{\frac{x_1+x_2}{\nu+2}} + e^{\frac{x_1+x_2}{\nu+2} - x_1} + e^{\frac{x_1+x_2}{\nu+2} - x_2} \right)$$

```
# Create a function to calculate the logarithm of density
log.dens <- function (x1, x2, nu ){
    a<- (x1+x2) /(nu+2)
    log(nu) + log(nu+1) - (nu+2) * log(exp(a) + exp(a - x1) + exp(a - x2))
}
```

Metropolis – Hastings algorithm

Initialization:

```
nu <– 1

n <–1e5

x <– matrix(nrow = n,  ncol = 2)

cur.x <– c(0, 0)

cur.logf<–log.dens(cur.x[1],cur.x[2],nu=nu)

n.accepted <– 0

sigma <– sqrt(9)
```

- # set parameter value
- # set the sample size
- # create matrix to hold sample
- # set the initial value
- # evaluate density at cur.x
- # count the accepted values
- # normal distribution st.d.

Metropolis – Hastings algorithm

```
# Iterations
for ( i  in  1 : n ) {                                  # repeat n times ...

   new.x <– cur.x + sigma * rnorm(2)                    # create proposed  value
   new.logf <– log.dens( new.x[1], new.x[2],nu = nu )   # evaluate density at proposed value

   prob.acceptance <– exp( new.logf - cur.logf )        # probability of acceptance

   if ( runif(1) < prob.acceptance) {                   # if we accept the proposed value ...
      n.accepted <– n.accepted + 1                      # incr. the counter of accepted values
      cur.x <– new.x                                                # accept the new value
      cur.logf <– new.logf                              # retain the density at the accepted
value
   }
   x[i,] <– cur.x                                       # store current value
}
```

Metropolis – Hastings algorithm

```
#load R source
source("mcmcR.r")

# call the function
s<–mcmcR (10 000)

# Plot the results
plot(s)
```