Optimization

Canhong Wen

Agenda

- Optimization
  - Root finding
  - gradient descent
  - Newton's method
  - Fisher scoring
  - Coordinate descent
  - optimization in R: `optimize`, `optim` and `nls`

Examples of Optimization Problems

- Minimize mean-squared error of regression
- Maximize likelihood of distribution
- Maximize output of tanks from given supplies and factories
- Maximize return of portfolio for given volatility
- Minimize cost of airline flight schedule
- Maximize reproductive fitness of an organism

Optimization Problems

- Given an **objective function** f, find

$$\theta^* = arg\min_\theta f(\theta)$$

- Produce sequence of points $\theta^{(k)}, k = 0, 1, 2, \ldots$ with $\theta^{(k)} \to \theta^*$.

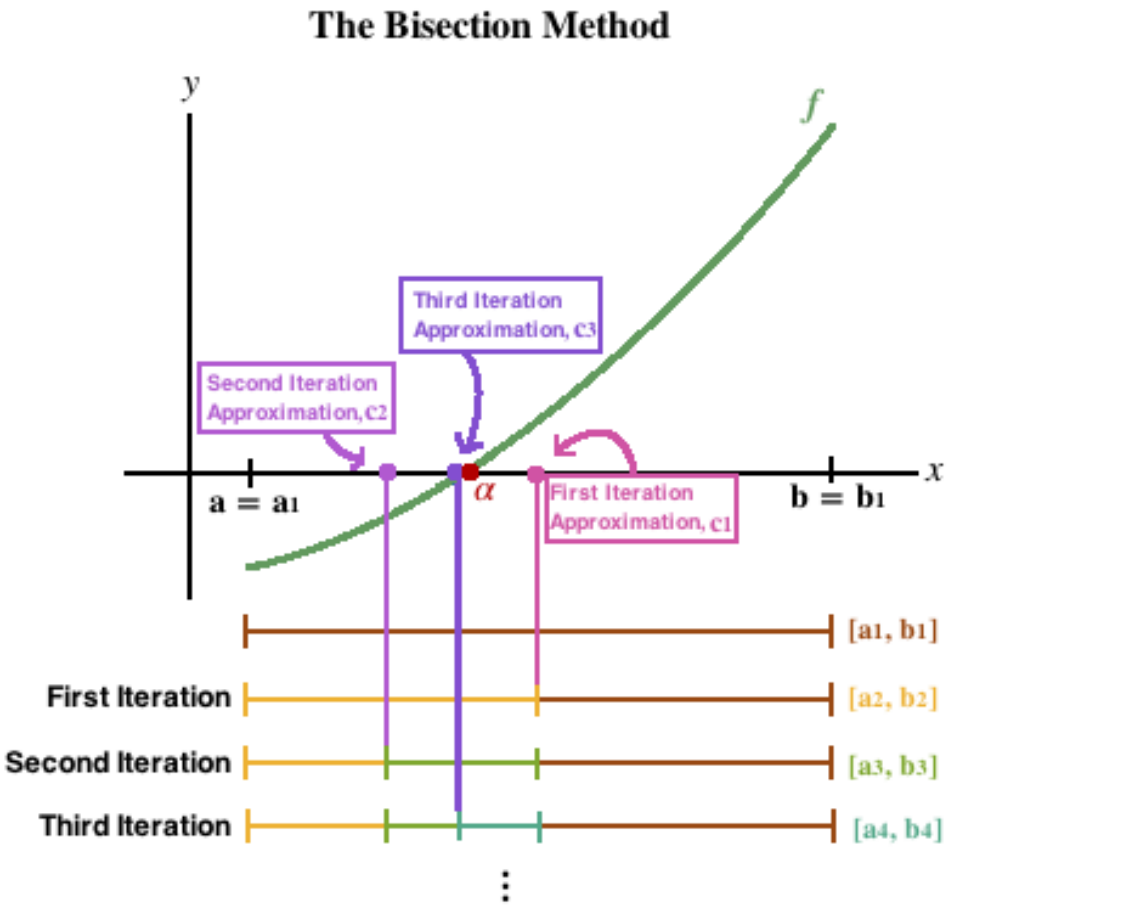
Transform to Root Finding Problems

- Can be interpreted as iterative methods for solving optimality condition

$$\nabla f(\theta^*) = 0$$

- We will introduce two one-dimensional root finding algorithms:
  - Bisection
  - Brent's method

Bisection



After each iteration of the Bisection Method we still have successively smaller intervals $[a_n, b_n]$ for which $f(a_n)f(b_n) < 0$, and so the root of interest in contains in these successively smaller intervals. The more iterations we have, the closer our approximation c_n is to the root.

Example: Bisection

- Solve

$$f(y) = a^2 + y^2 + \frac{2ay}{n-1} - (n-2) = 0.$$

```
f <- function(y, a, n){
  a^2 + y^2 + 2*a*y/(n-1) - (n-2)
}
a <- 0.5
n <- 20
b0 <- 0
b1 <- 5*n
```

```
iter <- 0
eps <- .Machine$double.eps^0.25
y <- seq(b0, b1, length.out = 3)
fv <- c(f(y[1], a, n), f(y[2], a, n), f(y[3], a, n))
if(fv[1]*fv[3]>0)
  stop("f does not have opposite sign at endpoints.\n")
```

```
while(iter < 1000 & abs(fv[2]) > eps){
  iter <- iter + 1
  if(fv[1]*fv[2] < 0){
    y[3] <- y[2]
    fv[3] <- fv[2]
  } else{
    y[1] <- y[2]
    fv[1] <- fv[2]
  }
  y[2] <- (y[1]+y[3])/2
  fv[2] <- f(y[2], a, n)
  print(c(y[1],fv[1], fv[3]-fv[2]))
}
```

```
## [1]    0.000  -17.750 1876.316
## [1]   0.0000 -17.7500 469.4079
## [1]   0.0000 -17.7500 117.5164
## [1]   0.00000 -17.75000  29.46135
## [1]  3.125000 -7.819901 17.172081
## [1]  3.125000 -7.819901  6.754986
## [1]  3.906250 -2.285619  3.530081
## [1]  3.906250 -2.285619  1.650599
## [1]  4.1015625 -0.7113133  0.8348365
## [1]  4.1015625 -0.7113133  0.4102657
## [1]  4.1503906 -0.3058160  0.2057289
## [1]  4.1748047 -0.1012793  0.1030135
## [1]  4.17480469 -0.10127926  0.05139497
## [1]  4.1809082 -0.0499588  0.0257068
## [1]  4.18395996 -0.02427063  0.01285573
## [1]  4.185485840 -0.011419556  0.006428445
## [1]  4.186248779 -0.004992275  0.003214368
## [1]  4.186630249 -0.001778197  0.001607221
## [1]  4.1868209839 -0.0001710497  0.0008036194
## [1]  4.1868209839 -0.0001710497  0.0004018029
## [1]  4.1868209839 -0.0001710497  0.0002008997
```

```
y[2]
```

```
## [1] 4.186845
```

```
fv[2]
```

```
## [1] 2.984885e-05
```

```
iter
```

```
## [1] 21
```

Brent's method

It is a hybridmethod which combines the reliability of **bracketing method** and the speed of open methods

- The approach was developed by Richard Brent(1973)
- The bracketing method used is the **bisection method**
  - The open method counterpart is the **secant method** or the **inverse quadratic interpolation**
  - It fits x as a quadratic function of y. If the three points are $(a, f(a)), (b, f(b)), (c, f(c))$, the next estimate for the root is found by interpolation, setting $y = 0$ in the Lagrange interpolation polynomial

$$x = \frac{[y - f(a)][y - f(b)]c}{[f(c) - f(a)][f(c) - f(b)]} + \frac{[y - f(b)][y - f(c)]a}{[f(a) - f(b)][f(a) - f(c)]} + \frac{[y - f(c)][y - f(a)]b}{[f(b) - f(c)][f(b) - f(a)]}$$

Algorithm for Brent's method

1. Calculate $f(a), f(b), f(c)$
2. Calculate R, S, T, P, Q as follows:

$$R = \frac{f(b)}{f(c)} \quad S = \frac{f(b)}{f(a)} \quad T = \frac{f(a)}{f(c)}$$

$$P = S[T(R - T)(c - b) - (1 - R)(b - 1)]$$

$$Q = (T - 1)(R - 1)(S - 1)$$

3. Let $b = b + P/Q$
4. Repeat until converged

Brent's method in R

```
out <- uniroot(f, lower = 0, upper = 5*n, a = a, n = n)
unlist(out)
```

```
##         root       f.root         iter      init.it   estim.prec
## 4.186870e+00 2.381408e-04 1.400000e+01           NA 6.103516e-05
```

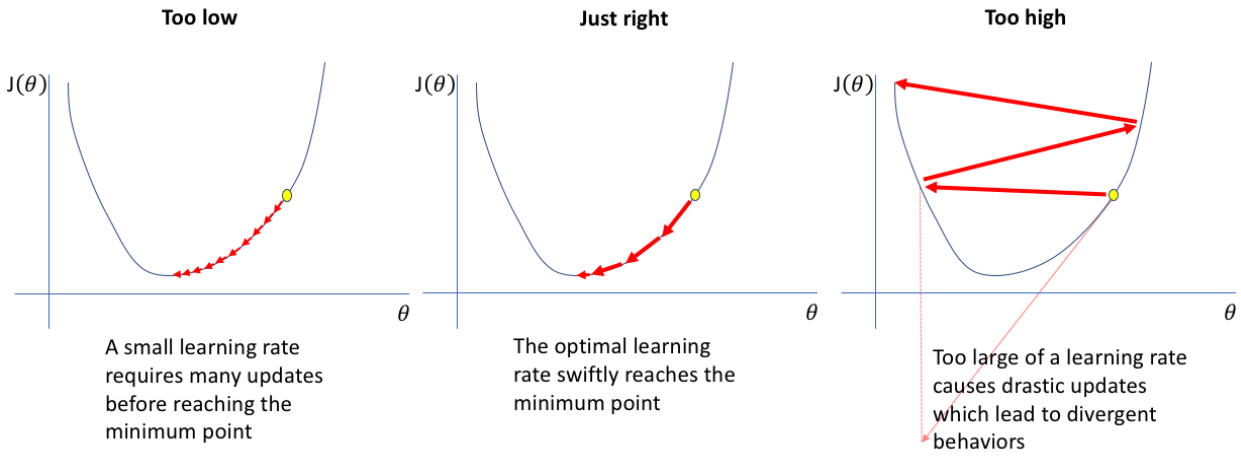
```
out <- uniroot(f, interval=c(-5*n,0), a = a, n = n)
unlist(out)
```

```
##          root        f.root          iter       init.it    estim.prec
## -4.239501e+00  2.382050e-04  1.400000e+01            NA  6.103516e-05
```

Gradient Descent

$$\theta^{(k+1)} = \theta^{(k)} - \eta^{(k)} \nabla f(\theta^{(k)})$$

- $\eta^{(k)}$: step-size for the k-th iteration
  - can be the same across the iteration, i.e., $\eta^{(k)} = \eta$



  - adaptively adjust $\eta^{(k)}$ to make sure of improvement or search along the gradient direction for minimum
- $\nabla f(\theta^{(k)})$: gradient vector
  - $\nabla f(\theta^{(k)})$ points in the direction of fastest ascent at $\theta^{(k)}$
  - for convex optimization it gives the global optimum under fairly general conditions.
  - for nonconvex optimization it arrives at local optimum

Example: solving multiple linear regression

$$\min_{\beta} \|y - X\beta\|^2$$

- $f(\beta) = \|y - X\beta\|^2$
- Gradient $\nabla f(\beta) = 2X^\top(X\beta - y)$
- Gradient descent

$$\beta^{(k+1)} = \beta^{(k)} - 2t^{(k)}X^\top(X\beta^{(k)} - y)$$

```
x <- rnorm(100)
y <- 2*x
iter <- 0
b <- 0
while(abs(sum(x^2)*b - sum(x*y))>1e-3 & iter < 1000){
  iter <- iter + 1
  h <- 2*sum(x^2)
  b <- b - 2*h^-1*(sum(x^2)*b - sum(x*y))
  cat(iter, b, "\n")
}
```

```
## 1 2
```

Choice of step size

Newton's Method

Taylor-expand for the value *at the minimum* θ^*

$$f(\theta^*) \approx f(\theta) + (\theta^* - \theta)\nabla f(\theta) + \frac{1}{2}(\theta^* - \theta)^T \mathbf{H}(\theta)(\theta^* - \theta)$$

Take gradient, set to zero, solve for θ^*:

$$0 = \nabla f(\theta) + \mathbf{H}(\theta)(\theta^* - \theta)\theta^* = \theta - (\mathbf{H}(\theta))^{-1}\nabla f(\theta)$$

Works *exactly* if f is quadratic and $\mathbf{H}^{-1}$ exists, etc.

If f isn't quadratic, keep pretending it is until we get close to θ^*, when it will be nearly true

Newton's Method

$$\theta^{(k+1)} = \theta^{(k)} - (\mathbf{H}(\theta))^{-1}\nabla f(\theta^{(k)})$$

- $(\mathbf{H}(\theta))^{-1}$: gives the step-size for the k-th iteration
  - Step-sizes chosen adaptively through 2nd derivatives, much harder to get zig-zagging, over-shooting, etc.
- Typically many fewer iterations than gradient descent

Getting Around the Hessian

Want to use the Hessian to improve convergence, but don't want to have to keep computing the Hessian at each step

Approaches:

- Use knowledge of the system to get some approximation to the Hessian, use that instead of taking derivatives ("Fisher scoring", see next slide)
- Use only diagonal entries (p unmixed 2nd derivatives)
- Use $\mathbf{H}(\theta)$ at initial guess, hope $\mathbf{H}$ changes *very* slowly with θ
- Re-compute $\mathbf{H}(\theta)$ every k steps, $k > 1$
- Fast, approximate updates to the Hessian at each step (BFGS)
- ...

Fisher scoring for maximum likelihood

- Maximum likelihood $l(\theta)$
$$\max_\theta l(\theta)$$
- Newton's Method $f = -l$
$$\theta^{(k+1)} = \theta^{(k)} - [\mathbf{H}\{-l(\theta)\}]^{-1}\nabla[-l(\theta^{(k)})]$$
Drawback: most of time, $\mathbf{H}\{-l(\theta)\}$ is complex, and may not be positive definite (no guarantee to be invertible)

Fisher scoring

- Remove observations, replace Hessian by the expected Hessian

$$I(\theta) = E\{\mathbf{H}[-\log l(\theta)]\} = -E\left\{\frac{\partial^2 \log l(\theta)}{\partial\theta\partial\theta^\top}\right\}$$

  which is the Fisher information matrix.

- Under mild regularity conditions

$$I(\theta) = E\{\mathbf{H}[-\log l(\theta)]\} = -E\left\{\left(\frac{\partial}{\partial\theta}\log l(\theta)\right)^2\right\}$$

Exponential family

- Exponential family: provides a general framework to parameterize distributions

$$f(x|\theta) = g(x)e^{\beta(\theta)+h(x)^\top v(\theta)}$$

- Example: $\exp(\lambda), N(\mu, \sigma^2)$
- Sufficient statistic: $h(x)$
- Fisher information for general exponential family

$$I(\theta) = \nabla v(\theta)^\top \Sigma(\theta)\nabla v(\theta), \quad \Sigma(\theta) = Var(h(x))$$

Example: Newton's method using Fisher scoring

- Consider multinomial distribution Multi$(n; p_1, \ldots, p_2)$.
- Assume $n = 56, x_1 = 20, x_2 = 9, x_3 = 1$ and $x_4 = 26$.
- Find MLE for p_1, p_2, p_3 and p_4.
- $\nabla[-l(\theta)] = -(x_1, x_2, x_3, x_4)/n$
- $I(\theta)$ is a diagonal matrix with diagonal entries n/p_i
- Newton's updating rule

$$p_i^{(k+1)} = p_i^{(k)} + \frac{x_i}{n}.$$

```
n <- 56
x <- c(20, 9, 1, 26)
p <- rep(1, 4)/4
p0 <- rep(0, 4)
iter <- 0
while(sum(abs(p-p0))>1e-3 & iter < 10){
p0 <- p
iter <- iter + 1
p <- p0 + x/n
p <- p/sum(p)
cat(iter, p, "\n")
}
```

```
## 1 0.3035714 0.2053571 0.1339286 0.3571429
## 2 0.3303571 0.1830357 0.07589286 0.4107143
## 3 0.34375 0.171875 0.046875 0.4375
## 4 0.3504464 0.1662946 0.03236607 0.4508929
## 5 0.3537946 0.1635045 0.02511161 0.4575893
## 6 0.3554688 0.1621094 0.02148438 0.4609375
## 7 0.3563058 0.1614118 0.01967076 0.4626116
## 8 0.3567243 0.1610631 0.01876395 0.4634487
## 9 0.3569336 0.1608887 0.01831055 0.4638672
## 10 0.3570382 0.1608015 0.01808384 0.4640765
```

Coordinate Descent

Gradient descent, Newton's method, etc., adjust all coordinates of θ at once — gets harder as the number of dimensions p grows

Coordinate descent: never do more than 1D optimization

- Start with initial guess θ
- While not converge
- For $i \in (1:p)$
- do 1D optimization over i-th coordinate of θ, holding the others fixed
- Update i-th coordinate to this optimal value
- Return final value of θ

Coordinate Descent

Cons:

- Needs a good 1D optimizer
- Can bog down for very tricky functions, especially with lots of interactions among variables

Pros:

- Can be extremely fast and simple

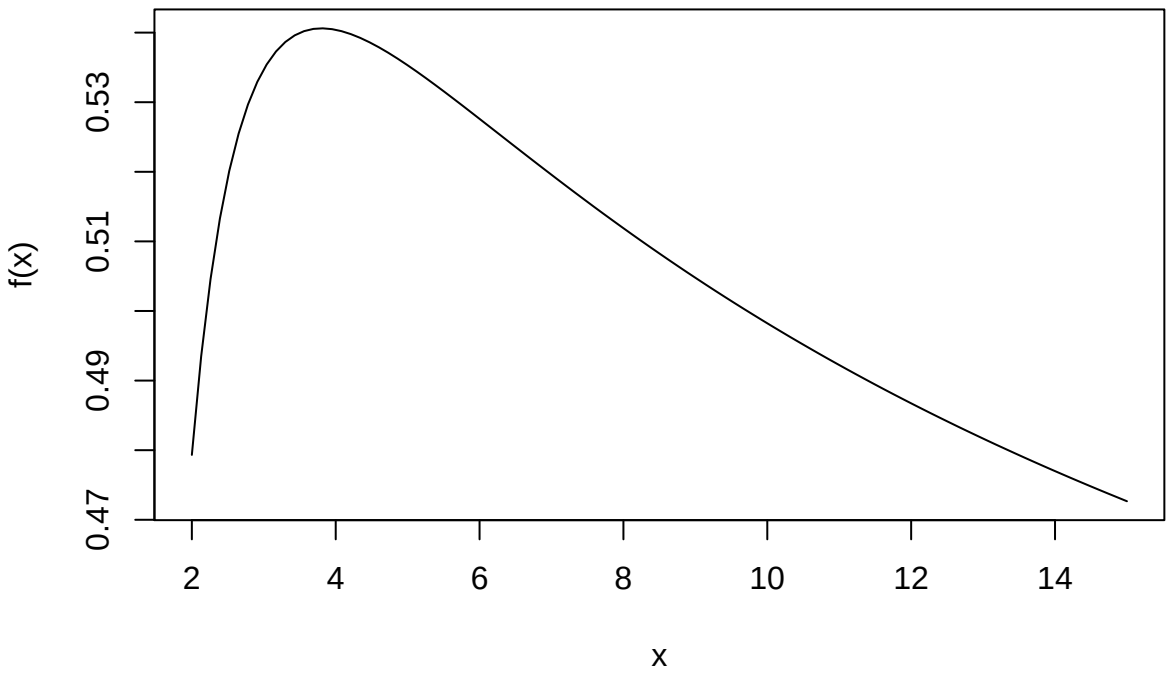
One-dimensional optimization in R: `optimize()`

- Brent's method for optimization
- The method used is a combination of golden section search and successive parabolic interpolation, and was designed for use with continuous functions.

Example

Maximize the function

$$f(x) = \frac{\log(1 + log(x))}{\log(1 + x)}$$




```
f <- function(x) log(1+log(x))/log(1+x)
optimize(f, lower = 4, upper = 8, maximum = TRUE)
```

```
## $maximum
## [1] 4.00006
##
## $objective
## [1] 0.5404008
```

Multi-dimensional optimization in R: `optim()`

`optim(par, fn, gr, method, control, hessian)`

- `fn`: function to be minimized; mandatory
- `par`: initial parameter guess; mandatory
- `gr`: gradient function; only needed for some methods
- `method`: Optimization algorithm, could be BFGS (Newton-ish)
- `control`: optional list of control settings
- (maximum iterations, scaling, tolerance for convergence, etc.)
- `hessian`: should the final Hessian be returned? default FALSE

Return contains the location (`$par`) and the value (`$val`) of the optimum, diagnostics, possibly `$hessian`

Optimization in R: optim()

```
gmp <- read.table("data/gmp.dat")
gmp$pop <- gmp$gmp/gmp$pcgmp
library(numDeriv)
mse <- function(theta) {
mean((gmp$pcgmp - theta[1]*gmp$pop^theta[2])^2)
}
grad.mse <- function(theta) { grad(func=mse,x=theta) }
theta0 <- c(5000,0.15) #Initialization
fit1 <- optim(theta0,mse,grad.mse,method="BFGS",hessian=TRUE)
```

Optimization in R: optim()

fit1: Newton-ish BFGS method

```
fit1[1:3]
```

```
## $par
## [1] 6493.2563738    0.1276921
##
## $value
## [1] 61853983
##
## $counts
## function gradient
##       63       11
```

Optimization in R: optim()

fit1: Newton-ish BFGS method

```
fit1[4:6]
```

```
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##             [,1]         [,2]
## [1,] 5.25021e+01      4422070
## [2,] 4.42207e+06 375729087977
```

Optimization in R: nls()

Nonlinear least squares estimate

```
nls(formula, data, start, control, [[many other options]])
```

- `formula`: Mathematical expression with response variable, predictor variable(s), and unknown parameter(s)

- `data`: Data frame with variable names matching `formula`
- `start`: Guess at parameters (optional)
- `control`: Like with `optim` (optional)
- `algorithm`: Optimization algorithm, the default is a Gauss-Newton algorithm (a version of Newton's method).

Returns an `nls` object, with fitted values, prediction methods, etc.

Optimization in R: nls()

fit2: Fitting the Same Model with nls()

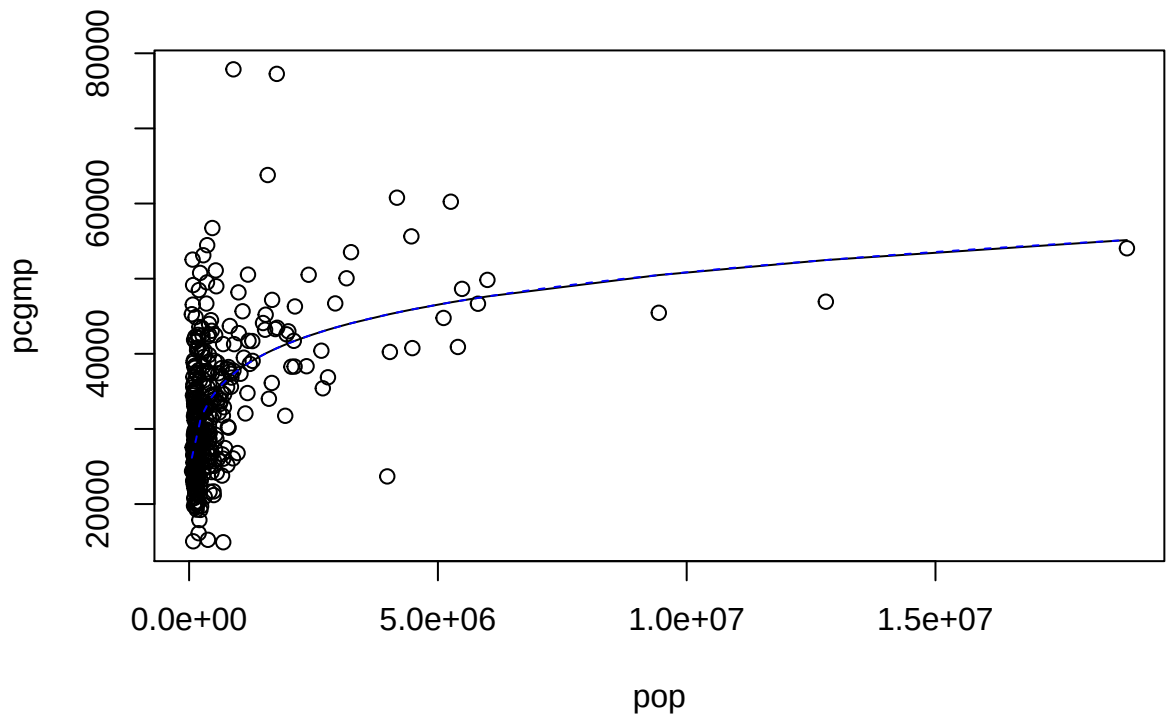
```
fit2 <- nls(pcgmp~y0*pop^a,data=gmp,start=list(y0=5000,a=0.1))
summary(fit2)
```

```
##
## Formula: pcgmp ~ y0 * pop^a
##
## Parameters:
##     Estimate Std. Error t value Pr(>|t|)
## y0 6.494e+03  8.565e+02   7.582 2.87e-13 ***
## a  1.277e-01  1.012e-02  12.612  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 7886 on 364 degrees of freedom
##
## Number of iterations to convergence: 5
## Achieved convergence tolerance: 1.789e-07
```

Optimization in R: nls()

fit2: Fitting the Same Model with nls()

```
plot(pcgmp~pop,data=gmp)
pop.order <- order(gmp$pop)
lines(gmp$pop[pop.order],fitted(fit2)[pop.order])
curve(fit1$par[1]*x^fit1$par[2],add=TRUE,lty="dashed",col="blue")
```



MLE of Gamma distribution

$$logL(\theta|x) = nr\log(\lambda) + (r-1)\sum_{i=1}^{n}\log x_i - \lambda\sum_{i=1}^{n}x_i - n\log\Gamma(r)$$

```
logL <- function(theta, sx, slogx, n){
  r <- theta[1]
  lambda <- theta[2]
  loglik <- n*r*log(lambda) + (r-1)*slogx - lambda*sx - n*log(gamma(r))
  -loglik
}

n <- 200
r <- 5
lambda <- 2
x <- rgamma(n, shape = r, rate = lambda)
```

```
optim(c(1,1), logL, sx = sum(x), slogx = sum(log(x)), n = n)
```

```
## $par
## [1] 4.809551 1.983855
##
```

```
## $value
## [1] 289.2557
##
## $counts
## function gradient
##       71       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
mlests <- replicate(100, expr = {
  x <- rgamma(n, shape = r, rate = lambda)
  optim(c(1,1), logL, sx = sum(x), slogx = sum(log(x)), n = n)$par
})
rowMeans(mlests)
```

```
## [1] 5.049364 2.009791
```