

# Empirical research **using R:** in economics, finance, and management

## Essentials, real examples, and troubleshooting

---

Compiled from session06.tex @ 2023-12-28 02:48:15+01:00.

### **Day 6: Numerical optimisation in R**

Andrei V. KOSTYRKA  
4<sup>th</sup> of October 2023



# Presentation structure

1. Basics of numerical optimisation
2. Gradient-free methods
3. Gradient-based methods
4. Stochastic methods
5. Common issues in real applications

# Quick recap

We learned:

- How to create various 2D plots
- How to customise plots and compute arbitrary statistics from the data
- How to render 3D plots and encode videos of animations

Today, we learn how to find solutions of various optimisation problems in the most general form.

# **Basics of numerical optimisation**

# What is numerical optimisation

**Optimisation:** Finding a parameter value that minimises or maximises the chosen objective function.

**Objective function:** A *scalar* function that is to be minimised or maximised.

Default behaviour in software: **minimisation**.

- Some plots look better with maximisation problems: easier to visualise hills than valleys
- Equivalence:  $\max f(\theta) = -[\min -f(\theta)]$

# Optimisation problems in economics

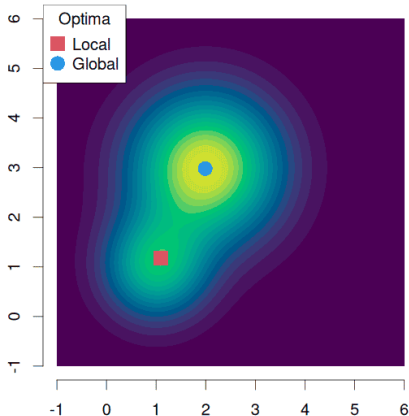
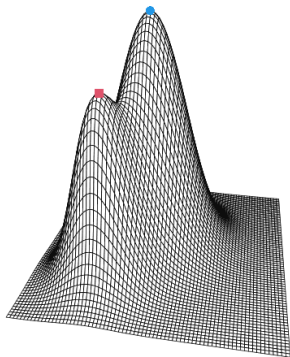
- Minimising model penalty to find the best fit:  $S(\ell(U))$ , where  $U$  is the model **residual** (observed minus predicted),  $\ell$  is the **loss** function,  $S$  is the aggregating statistic
  - Least squares:  $\ell(U) = U^2$ ,  $S = \sum_{i=1}^n$
  - Median absolute deviations:  $\ell(U) = |U|$ ,  $S = \text{Median}$
  - Least trimmed squares (at 90%): Exclude 10% largest  $U^2$ , or  $S(\ell(U)) = \sum_{i=1}^{\lfloor 0.9n \rfloor} U_{(i)}^2$ , where  $U_{(i)}$  are *ordered* penalties
- Maximising goodness of fit:
  - Likelihood: Probit / logit, Heckman selection, parametric conditional density (GARCH variants), empirical likelihood
  - Bayesian models: Maximum posterior utility expectation
  - Machine learning: Percentage of correctly classified cases

# Unconstrained optimisation

$$\hat{\theta} := \arg \min_{\theta \in \mathbb{R}^d} f(\theta)$$

- Finding the minimum of the objective function  $f$ 
  - $\hat{\theta}$  is a **local minimiser** of  $f$  if  $f(\hat{\theta}) \leq f(\theta) \forall \theta \in \Theta$ , where  $\Theta$  is a neighbourhood
  - If  $\Theta = \mathbb{R}^d$ ,  $\hat{\theta}$  is a **global minimiser**
- Example: ordinary least squares (for any data set with  $n \geq d$  observations and a linear model without linearly dependent regressors, one shall get an OLS estimate)

# Local and global optima





# Constrained optimisation

$$\min_{\theta \in \Theta} f(\theta) \quad \text{subject to } g(\theta) = 0$$

- Finding the minimum of the objective function  $f$ 
  - Equality constraint: Certain functional relationships between the coordinates of  $\hat{\theta}$  should hold
  - Subspace constraint:  $\theta \in \Theta$  is often replaced by  $h(\theta) \geq 0$  (inequality constraint)
- Example: Maximise the Sharpe ratio,  $\frac{\mathbb{E}(R-R_f)}{\sigma_R}$ , of a portfolio of 10 stocks with weights  $0.05 \leq w_i \leq 0.40$ ,  $\sum_i w_i = 1, i = 1, \dots, 10$
- Hard to tackle in general, but several solutions exist
  - R packages: `nloptr`, `Rsolnp`

# Converting constrained problems

Re-define the objective function to *bake in* the constraints:

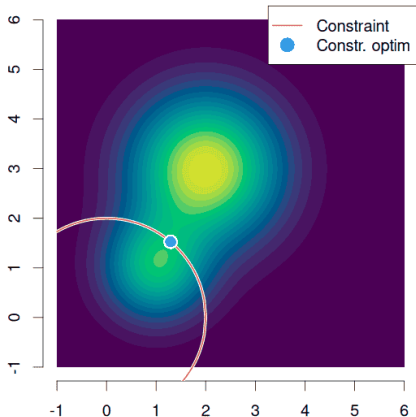
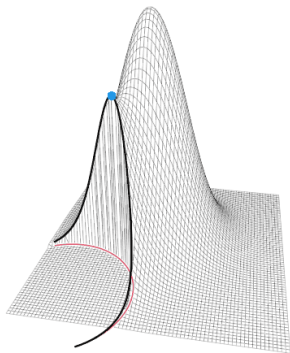
$$\min_{\theta \in \Theta} f(\theta) \quad \text{s.t.} \quad g(\theta) = 0 \iff \min_{\theta \in \mathbb{R}^d} f^*(\theta),$$

$$f^*(\theta) := \begin{cases} f(\theta), & \theta \in \Theta \text{ and } g(\theta) = 0, \\ \infty, & \text{otherwise.} \end{cases}$$

- Checking if  $\theta \in \Theta$  is easy
  - But  $\theta$  near the boundary  $\Rightarrow$  algorithms may behave poorly
- Incorporating  $g(\theta) = 0$  is **hard** as it usually restricts the solution space to a subspace of lower dimensionality
  - Solution: re-parametrise the problem (e.g. if  $\sum_{i=1}^n w_i = 1$ , use  $1 - \sum_{i=1}^{n-1} w_i$  instead of  $w_n$ )
  - Expressing  $\theta^{(i)}$  through  $g$  and  $\theta^{(-i)}$  can be impossible – add penalty for  $g(\theta) \neq 0$ , e.g.  $\min_{\theta} [f(\theta) + 100\|g(\theta)\|_2^2]$

# Constrained optimisation visualisation

Constraint:  $[\theta^{(1)}]^2 + [\theta^{(2)}]^2 = 4$



# Derivative of a function

**Derivative:** The immediate rate of change of a function.

$$f'(\theta) = \frac{df}{d\theta} := \lim_{h \rightarrow 0} \frac{f(\theta + h) - f(\theta)}{h}$$

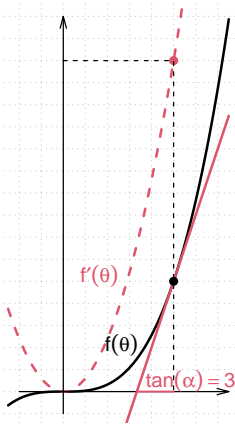
$f'(\theta)$  is the slope of the tangent line to the graph at  $\theta$ .

Illustration:  $f(\theta) := \theta^3$ ,  $f'(\theta) = 3\theta^2$ .

$f(1) = 1$ ,  $f'(1) = 3$ .

The tangent equation at  $\theta = 3$  is  $3\theta - 2$ .

A differentiable function **must be** continuous (the opposite is not true:  $f(\theta) = |\theta|$  is not differentiable at 0).



# Gradient of a function

---

**Gradient:** vector of partial derivatives of a differentiable scalar function.

$$\nabla_f(\theta) := \begin{pmatrix} \frac{\partial f}{\partial \theta^{(1)}}(\theta) \\ \vdots \\ \frac{\partial f}{\partial \theta^{(d)}}(\theta) \end{pmatrix}$$

At any point  $\theta$  (a vector), the gradient – the  $d$ -dimensional slope of  $f$  – is the **direction and rate of the steepest growth** of  $f$ .

*‘A source of anxiety for non-mathematics students.’  
J. C. Nash, ‘Nonlinear Parameter Optimization’ (2014).*

# Jacobian of a function

**Jacobian:** Matrix of gradients for a vector-valued function  $f$ .

If  $\dim \theta = d$ ,  $\dim f = m$ ,

$$J_f(\theta) := \left( \frac{\partial f}{\partial \theta^{(1)}}(\theta) \quad \dots \quad \frac{\partial f}{\partial \theta^{(d)}}(\theta) \right) = \begin{pmatrix} \nabla_{f^{(1)}}^T(\theta) \\ \vdots \\ \nabla_{f^{(m)}}^T(\theta) \end{pmatrix}$$

The Jacobian is often required for constrained optimisation problems, i. e.  $J_g(\theta)$  if  $g(\theta) = 0$ .

*Including incorrectly computed derivatives (mostly gradients or Jacobian matrices) <...> explains almost all the 'failures' of optimisation codes I see. (Idem.)*

# Hessian of a function

**Hessian:** Square matrix of second-order partial derivatives of a twice-differentiable scalar function.

$$\nabla_f^2(\theta) := \left\{ \frac{\partial^2 f}{\partial \theta^{(i)} \partial \theta^{(j)}} \right\}_{i,j=1}^d = \begin{pmatrix} \frac{\partial^2 f}{\partial \theta^{(1)} \partial \theta^{(1)}}(\theta) & \cdots & \frac{\partial^2 f}{\partial \theta^{(1)} \partial \theta^{(d)}}(\theta) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial \theta^{(d)} \partial \theta^{(1)}}(\theta) & \cdots & \frac{\partial^2 f}{\partial \theta^{(d)} \partial \theta^{(d)}}(\theta) \end{pmatrix}$$

The Hessian is the transpose Jacobian of the gradient:

$$\nabla_f^2(\theta) = J_{\nabla_f}^T(\theta)$$

If  $\nabla_f$  is differentiable,  $\nabla_f^2$  is symmetric.

Hessians are sometimes useful in numerical optimisation, but are too slow or unreliable (approximations are used).

# Numerical derivatives

---

When analytical derivatives are not available, the derivative definition gives a hint:

$$f'(x) := \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

Remove the limit:

$$f'_{\text{FD}}(x, h) := \frac{f(x+h) - f(x)}{h}$$

One can choose a sequence of decreasing step sizes  $h_i$  (e.g.  $\{0.1, 0.01, 0.001, \dots\}$ ), and observe the sequence

$f'_{\text{FD}}(x, 0.1), f'_{\text{FD}}(x, 0.01), f'_{\text{FD}}(x, 0.001), \dots$  converge to  $f'$ .



# Higher-order accuracy of derivatives

Central derivatives are more accurate than one-sided ones:

$$f'_{\text{CD}} := \frac{0.5f(x+h) - 0.5f(x-h)}{h}$$

The Taylor expansion yields:

- $f'(x) - f'_{\text{FD}} = -\frac{f''(x)}{2}h + O(h^2) = O(h)$
- $f'(x) - f'_{\text{CD}}(x) = -\frac{f'''(x+\alpha h)}{6}h^2 = O(h^2)$ , where  $\alpha \in [-1, 1]$

However, if  $f(x)$  is already known, it requires 2 more computations than  $f'_{\text{FD}}$ , which is 2 times slower.

Better accuracy is achievable with more terms and careful choice of  $h$ .

# Higher-order derivatives

Higher-order derivatives may be obtained by repeatedly differencing first derivatives. However, doing the differencing in one step is more accurate:

$$f''(x) = \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} + O(h^3)$$

Derivatives of order  $m$  may be computed as a weighted sum of  $f$ . For a **stencil**  $\{b_i\}_{i=1}^n$ , define the **evaluation grid**  $x + b_1h, x + b_2h, \dots, x + b_nh$ . If  $m < n$ , then,  $\exists\{w_i\}_{i=1}^n$  that yield the  $a^{\text{th}}$ -order-accurate approximation of  $f^{(m)}$ :

$$\frac{d^m f}{dx^m}(x) = \frac{1}{h^{-m}} \sum_{i=1}^n w_i f(x + b_i h) + O(h^a)$$

# Numerical Hessians

Compute Hessian via finite differences of finite differences w. r. t. two indices. Define  $h_j$  to be a vector of zeros with 1 in the  $i^{\text{th}}$  position.

4 evaluations of  $f$  are required to approximate  $H_{ij}$  via CD:

$$\begin{aligned} H_{ij} &\approx \frac{\nabla_i f(x + h_j) - \nabla_i f(x - h_j)}{2h} \approx \\ &\approx \frac{f(x+h_i+h_j) - f(x-h_i+h_j) - f(x+h_i-h_j) + f(x-h_i-h_j)}{4h^2} \end{aligned}$$

Use the symmetry: if  $x = \theta$ , compute  $H_{ij}$  for all  $i = 1, \dots, \dim \theta$  and  $j \geq i$ .

# Numerical accuracy matters

---

The choice of size  $h$  is crucial for accuracy.

- $h$  too large → **truncation error** from the truncated Taylor-series term
- $h$  too small → **rounding error**: catastrophic cancellation, division of something small by something small – finite machine accuracy (machine  $\varepsilon$ )

Exact expressions for optimal  $h^*$  exist. A package for auto-selection of  $h$  and parallel computation of gradients is being developed. General rules:

- $h^* \sim \varepsilon^{1/2}$  for forward,  $h^* \sim \varepsilon^{1/3}$  for central differences
- $h^* \sim \varepsilon^{1/4}$  for central second derivatives and Hessians

# Minimisation in the language of calculus

---

The problem for a continuously differentiable  $f$ :

$$\hat{\theta} := \arg \min_{\theta \in \mathbb{R}^d} f(\theta)$$

can be reformulated via 1<sup>st</sup>- and 2<sup>nd</sup>-order conditions:

$$\{\hat{\theta}: \nabla_f(\hat{\theta}) = 0 \quad \text{and} \quad \nabla_f^2(\hat{\theta}) \text{ is pos. semi-def.}\},$$

where  $\nabla_f$  is the gradient and  $\nabla_f^2$  is the Hessian.

If multiple such points exist, declare  $\hat{\theta}$  to be the candidate yielding the smallest value  $f$ .

# Gradient visualisation

---

Live demonstration time!

# Minimisation / FOC solving

---

- Which one is easier: to search for  $\hat{\theta}$  by checking if (1) it yields a reduction of  $f$ , or (2) if brings  $\nabla_f$  closer to 0?
- It is easy to get an idea about the **general shape** of the function by directly evaluating it at some points
  - If  $f(\theta_2) < f(\theta_1)$ , then,  $\theta_2$  is a better candidate solution
- The gradient information, on the other hand, is hard to interpret or manipulate
  - The curvature of  $f$  may differ w. r. t. coordinates of  $\theta$
- Solving  $\nabla_f(\theta) = 0$  is equivalent to **minimising the length of the gradient**:  $\min_{\theta} \|\nabla_f(\theta)\|$  in some metric (Euclidean, Manhattan, variable etc.)
  - If  $\hat{\theta}$  solves the FOC equation system, then,  $\|\nabla_f(\hat{\theta})\| = 0$

# Types of optimisation

---

- Unconstrained vs constrained
  - Budget constraints; positive portfolio weights adding up to unity; constrained parameter space (negative price elasticity, positive foreign price elasticity of demand)
- Global vs local
  - Global optimisation is much harder as it cannot be guaranteed in the general case; we focus on **local optimisation** and on multiple-local-optima checks
- Deterministic vs stochastic
  - Random initial values or iteration rules have large benefits but come at a computational cost
- Low-dimensional vs high-dimensional
  - High-dimensional problems deserve a separate course



# Common issues in optimisation

---

- Sensitivity to function/parameter scaling
  - Multiplying  $f$  by 1000 should not affect the algorithm
  - Recall session 1: computers do not like mixing small and big numbers; preferred order of magn. of  $f$  and  $\theta$  is 0.1–10
- Non-smooth functions
  - Quantile regression, maximum score for non-parametric discrete choice, non-smooth utility, conditional VaR (expected shortfall), binary predictions
- Premature convergence to local optima
  - Arises if theoretically the parameters are identified but the data available do not produce an objective function of the shape that guarantees global convergence
- Too many parameters
  - Slow convergence, ill-conditioned matrices

# Ideas behind optimisation algorithms

---

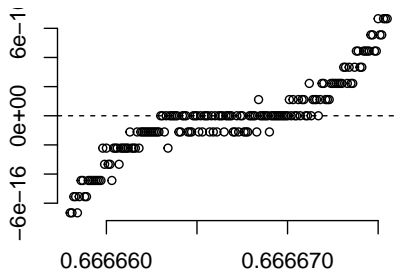
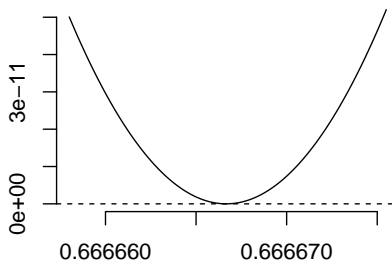
1. Start somewhere
  - The user **must** supply an initial value  $\theta_0$  for which  $f(\theta) \in \mathbb{R}$
2. Determine which coordinates of  $\theta_0$  should be changed and by how much to get  $f(\theta_1) < f(\theta_0)$  for some  $\theta_1$
3. Repeat many times until the stopping criterion is met (recall Session 1, 'Loop termination'):
  - grad. tol, the gradient is close to zero:  $\|\nabla_f(\theta_k)\|_2 \leq \varepsilon$
  - rel. tol, the relative improvement is close to zero:  
 $f(\theta_k) - f(\theta_{k+1}) < \varepsilon$
  - rel. xtol, the relative change of  $\theta_{k+1}$  dictated by the algorithm is close to zero:  $\max \left| \frac{\theta_{k+1} - \theta_k}{\theta_k} \right| < \varepsilon$

**NB.** Maximum iterations reached  $\neq$  convergence!

# Accuracy limits in optimisation

Minimise  $f(\theta) := \frac{1}{4}\theta^4 - \frac{2}{3}\theta^3 + \frac{2}{3}\theta^2 - \frac{8}{27}\theta + \frac{4}{27}$  and find the local optimum around  $2/3$  (left).

Derivative:  $\nabla_f(\theta) = \theta^3 - 2\theta^2 + \frac{4}{3}\theta - \frac{8}{27} = (\theta - 2/3)^3$  (right).



Numerical methods can return  $\hat{\theta} \in 0.66666 + [0.1, 1.2] \cdot 10^{-5}$ .

# Error magnification factor

---

Requesting the optimiser to stop if some tolerance is  $< \varepsilon$  does **not** guarantee solution accuracy up to  $\pm\varepsilon$ . The maximum accuracy that one can obtain does not exceed the **error magnification factor** (EMF).

EMF depends on the problem. In economics, many problems are written as linear equations  $A\theta = b$  (or  $\approx b$ ).

$$\text{EMF} = \|A\|_{\infty} \cdot \|A^{-1}\|_{\infty},$$

$\|A\|_{\infty} := \max_i \sum_{j=1}^n |A_{ij}| = \text{max. row sum of abs. values.}$

If  $\text{EMF} \approx 10^k$ ,  $k$  accuracy digits are lost.  $k \approx 11$  (previous problem)  $\Rightarrow 16 - 11 = 5$  digits are accurate.

## Example: EMF for bare-bones OLS

An economist estimates a regression of mpg on all other variables from the mtcars data set using OLS via matrices:

$$\text{mpg} = \beta_0 + \beta_1 \text{cyl} + \dots + \beta_{10} \text{carb} + U = \tilde{X}'\beta + U, \quad A = \frac{1}{n} \sum_{i=1}^n \tilde{X}_i \tilde{X}_i'$$

```
X <- cbind(1, as.matrix(mtcars[, -1]))
A <- crossprod(X) / nrow(mtcars)
max(rowSums(abs(A)))           # 117619
max(rowSums(abs(solve(A))))    # 1895
```

Manual approach yields max. rel. error  
 $\text{mach.eps}/2 \cdot \|A\|_{\infty} \cdot \|A^{-1}\|_{\infty} \approx 2 \cdot 10^{-8}$ .

If possible, use functions from well-tested libraries (LAPACK, Armadillo etc.); do not re-invent the wheel.

# Example: Wilkinson polynomials

Some functions have poor convergence in optimisation.

Searching for the roots of the polynomial of the form

$W_n(\theta) = (\theta - 1) \cdot (\theta - 2) \cdot \dots \cdot (\theta - n) = \prod_{k=1}^n (\theta - k)$ , one can get poor numerical solutions. (May arise in repeated derivation:  $\frac{\partial^n}{\partial x^n} x^\theta$  has these components.)

Obviously,  $W_{20}(16) = 0$ , but the numerical solution  $\hat{\theta} \approx 16.00003$  (default tol =  $1.2e-4$ ) has  $w(\hat{\theta}) \gg 0$ :

```
w <- function(x, deg = 20) prod(x - 1:deg)
r <- uniroot(w, interval = c(15.9, 16.2))
print(r$root, 12) # 16.0000256241
w(r$root) # 804218432, not a typo
```

# Linear programming

---

Finding the optimum of a linear function subject to linear constraints:

$$\min_{\theta} c'\theta \quad \text{subject to } A\theta \geq b,$$

where  $\dim c = \dim \theta = d$ ,  $\dim b = m$ ,  $\dim A = (m \times d)$ .

- Operations research
- Company management
- Maximise profits and minimise costs with limited resources




Dantzig's simplex algorithm, criss-cross algorithm etc.

*'Programming'* really means *'optimisation'*.

# Linear programming example

A construction company erects bunkhouses and cabins. Each building requires beams, wall panels, and planks.



Material	Bunkhouse	Cabin	Material stock, €
 Beams	1	2	100
 Wall panels	3	1	150
 Planks	0	4	160
Sell price	10	30	

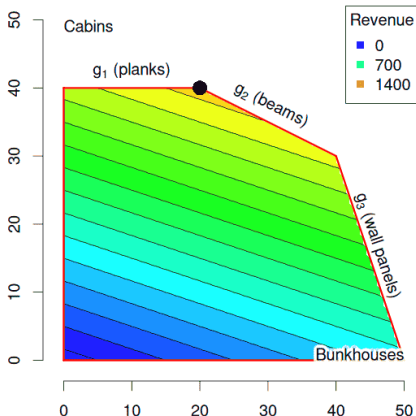
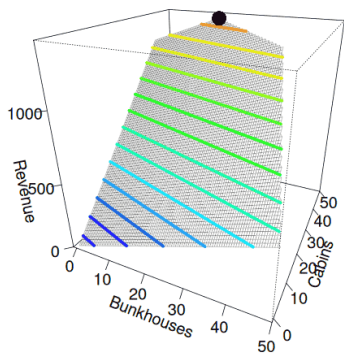


# Linear programming questions

---

1. What is the construction plan (bunkhouses and cabins) that maximises the revenue?
2. If there are leftover materials after construction, which short-supply materials are to be bought to use them up with maximum profit?

# Linear programming visualisation



It is not really about optimisation, it is about finding which constraints are actively binding.

# Linear programming solution

---

$$\max 10b + 30c \quad \text{s.t.} \quad g(b, c) := \begin{pmatrix} b + 2c - 100 \\ 3b + c - 150 \\ c - 40 \\ -b \\ -c \end{pmatrix} \leq 0$$

- $(\hat{b}, \hat{c}) = (20, 40)$
- Constraint 2 is not binding:  $3\hat{b} + \hat{c} = 100$ , therefore, €50 of panels are left over
- At  $(\hat{b}, \hat{c})$ , +1 bunkhouse requires +€1 of beams and uses €3 of panels; with 50 unused WP, one could buy €16 of beams and build 16 more  $b$  (profit  $16 \cdot 9 = 144$ )

# Linear programming in R

The `lp()` function from `lpSolve` does the job:

```
revenue <- c(10, 30)
cost <- cbind(c(1, 3, 0, 1, 0),
              c(2, 1, 4, 0, 1))
budget <- c(100, 150, 160, 0, 0)
f.dir <- c(rep("<=" , 3), ">=", ">=")
o <- lpSolve::lp(objective.in = revenue,
                 const.mat = cost, const.dir = f.dir,
                 const.rhs = budget, direction = "max")
o[c("objval", "solution")]

#> $objval $solution
#>    1400      20 40
```

It is an interface to the `lp_solve 5.5` C library (can do more than the `lpSolve` R interface).

# Quadratic programming

---

Finding the optimum of a quadratic function subject to linear constraints:

$$\min_{\theta} \frac{1}{2} \theta' Q \theta + c' \theta \quad \text{subject to } A \theta \geq b,$$

where  $Q$  is symmetric and  $\dim Q = (d \times d)$ .

The solution is analytical; finding the active constraints is the main job.

Examples: OLS, LASSO, convex non-parametric regression.

- Goldfarb–Idnani dual method (`quadprog::solve.QP`)
- Interior point (`ipoptr::ipoptr`)
- Augmented Lagrangian (`nloptr::auglag`)

# Quadratic programming example

Suppose: due to the lack of free labour and equipment, the revenue function is tapering off with respect to  $b$  and  $c$ .

$$\min_{b,c} [-f(b,c)] = -[10b - 2(b - 15)^2 + 30c - (c - 32)^2]$$

quadprog: `solve.QP()` has a slightly different syntax:  
solve  $\min(-c'\theta + 0.5\theta'Q\theta)$  with the constraints  $A'\theta \geq b$ :

$$\min_{\theta} \frac{1}{2}\theta' \begin{pmatrix} 4 & 0 \\ 0 & 2 \end{pmatrix} \theta - \begin{pmatrix} 70 \\ 94 \end{pmatrix} \theta, \quad \begin{pmatrix} -1 & -3 & 0 & 1 & 0 \\ -2 & -1 & -4 & 0 & 1 \end{pmatrix}' \theta \geq \begin{pmatrix} -100 \\ -150 \\ -160 \\ 0 \\ 0 \end{pmatrix}$$

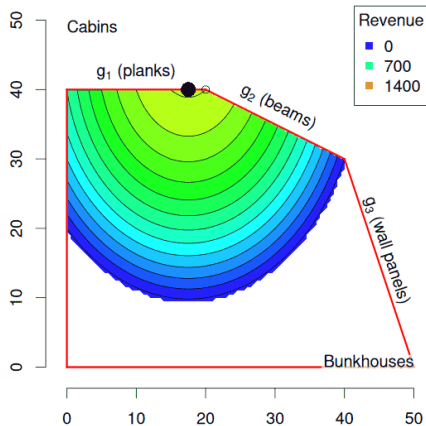
# Quadratic programming implementation

```
Q <- 2 * diag(c(2, 1))
c <- c(70, 94)
cost <- rbind(c(-1,-3,0,1,0), c(-2,-1,-4,0,1))
budget <- c(-100, -150, -160, 0, 0)
o <- quadprog::solve.QP(Dmat = Q, dvec = c,
  Amat = cost, bvec = budget)
o[c("solution", "unconstrained.solution")]
#> 17.5 40.0 constr, 17.5 47.0 unconstr
which(o$Lagrangian != 0) # 3rd active constraint
```

One can construct only integer houses; check the remaining budget – 3 beams and 59 panels remain:

```
crossprod(cost, floor(o$solution)) - budget
# 3 59 0 17 40
```

# Quadratic programming visualisation



QP is also about finding the binding constraints.



# Convex optimisation

---

Finding the optimum of a convex function with constraints:

$$\min_{\theta} f(x) \quad \text{subject to } g(x) \geq 0,$$

where both  $f$  and  $g$  are convex:

$$f(\alpha\theta_1 + (1 - \alpha)\theta_2) \leq \alpha f(\theta_1) + (1 - \alpha)f(\theta_2)$$

Every local minimum is a global minimum.

- Descent methods (gradient, steepest, Newton's)
  - $\nabla$ -based,  $\nabla^2$ -based (exact or approximate)
- Interior-point methods

# Linear vs. non-linear problems

---

If the problem is linear in parameters ( $A\theta = b$ ) and

1. There are as many equations as there are parameters,
2. All equations are consistent,
3. The matrix  $A$  is full-column-rank,

then,  $\hat{\theta} := A^{-1}b$  is the unique solution.

If the problem is non-linear:

- Try many-many reasonable candidate points,
- Or consider a simpler problem (e.g. assume some parameter values),
- Or linearise the problem around a candidate  $\tilde{\theta}$ .

Solve a sequence of simpler problems, check the FOC+SOC.

# General optimisation solution

---

If solving the FOC to obtain **the solution** is impossible, we start from any candidate  $\tilde{\beta}$  and look for **an improvement**.

If  $f(\tilde{\beta}) > f(\tilde{\beta}) > f(\tilde{\beta}) > \dots$ , we continue guessing until there is little to no improvement to the value of  $f$ .

Once a good enough candidate  $\check{\beta}$  has been found, check the FOC (necessary) and SOC (sufficient):

$$\nabla_f(\check{\beta}) = 0, \quad \nabla_f^2(\check{\beta}) = \text{pos. def.}$$

If these two conditions hold, declare  $\check{\beta}$  to be a **strict local minimum** (or simply a local minimum if  $\nabla_f^2(\check{\beta})$  is PSD).

Any questions on the mathematical concepts behind optimisation?

# **Gradient-free methods**

# Derivative-free optimisation is fun

---



Make a joke about derivative-free optimisation.



Why did the derivative-free optimizer get invited to all the parties?

Because it always knew how to find the maximum fun without taking any directions!

# Grid search

---

If a reasonable range for the optimal parameter is known, one can try all the variants!

- Generate a lattice of parameter values
- Evaluate the objective function at all points
- Pick the point with the minimum value
- Refine the grid around the optimum if necessary

**Pros:** exhaustive; with fine enough grid, finds the optimum; with a large enough grid, finds the global optimum.

**Cons:** very costly curse of dimensionality: requires  $n_{\text{grid}}^{\dim \theta}$  evaluations.

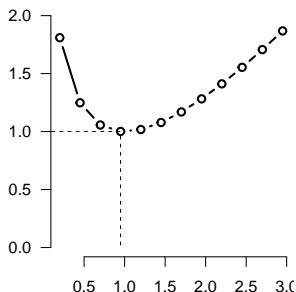
# One-dimensional grid search

- Generate a sequence from  $\theta_{\min}$  to  $\theta_{\max}$  of chosen length  $n$  (as many as the resources allow)
- Plot  $(\theta, f(\theta))$
- If the optimal value is in the interior, choose it
- If the value is on the boundary, extend the grid

```
f <- \(x) -log(x) + x
xseq <- seq(0.2, 3, 0.25)
yseq <- sapply(xseq, f)
plot(xseq, yseq)
xseq[which.min(yseq)] # 0.95
```

**NB.** No guarantee of exactitude –  
refine on

```
xseq[which.min(yseq)+c(-1,1)].
```





# Two-dimensional grid search

---

Without vectorising the function  $f(x, y)$ , the quickest way to obtain a 2D lattice is as follows:

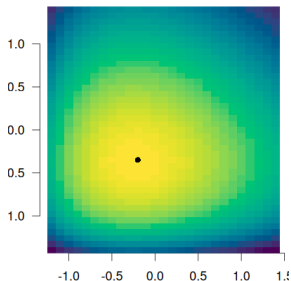
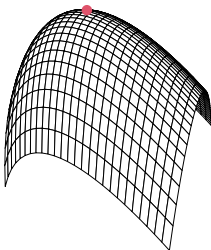
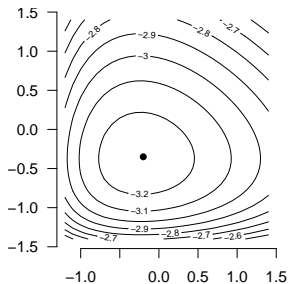
- Create a matrix of all combinations of 2 parameters from two vectors
- Create a vector of values  $f(x, y)$ 
  - Parallelise this computation over all rows of the matrix
- Wrap this vector into a matrix and visualise it

## 2D grid search example

---

```
f <- function(x) -(2 + log(x[1]+1.9) - 0.6*x[1]
  + log(x[2]+1.8) - 0.7*x[2])
xvec <- seq(-1.2, 1.4, length.out = 51)
yvec <- seq(-1.4, 1.4, length.out = 41)
xy <- as.matrix(expand.grid(x=xvec, y=yvec))
zvec <- unlist(parallel::mclapply(
  1:nrow(xy), \(i) f(xy[i, ]), mc.cores = 4))
# zvec <- apply(xy, 1, f) # No parallel
zmat <- matrix(zvec, nrow = length(xvec))
opt <- xy[which.min(zvec), ]
```

# 2D grid visualisations



```
contour(xvec, yvec,  
        zmat)  
points(opt[1], opt[2])
```

```
p <- persp(xvec,  
           yvec, -zmat)  
points(  
  trans3d(  
    opt[1], opt[2],  
    -f(opt), p))
```

```
image(xvec, yvec,  
      zmat^2,  
      col = hcl.colors(51))  
points(opt[1], opt[2])
```

# Partial grid search

---

The objective function may behave better w. r. t. some parameters and fluctuate wildly w. r. t. other 'wicked' ones.

If certain parameters are fixed, the function may become linear / quadratic in a sub-set of parameters – easy solution.

- Generate a lattice of 'wicked' parameters
- For each combination of those, keep them as fixed and optimise the function w. r. t. 'nice' parameters
- Compare the optima and choose the 'wicked' parameters that yielded the best solution

# Application #1: hyper-parameter search

XGBoost has many tuning parameters: number of boosting rounds, learning rate, minimum loss reduction for partition, maximum tree depth, regularisation term on weights etc.

Generate a grid of all combinations, run XGBoost for each of them, and choose the combination that minimises the out-of-sample prediction error on test data.

```
pars <- expand.grid(  
  nrounds = c(2, 5, 8), eta = c(.2, .3, .4),  
  gamma = c(0, 0.01, 0.03), maxdepth = (1:5)*2,  
  lambda = c(0.5, 1), alpha = c(0, 0.5))
```

```
#> nrounds eta gamma maxdepth lambda alpha  
#>      2 0.2      0         2     0.5     0  
#>      5 0.2      0         2     0.5     0  
#> <...>  
#>      8 0.4  0.03        10         1     0.5
```

## Application #2: estimating returns to scale

A researcher interested in returns to scale in a Cobb-Douglas-like production function is fitting the curve

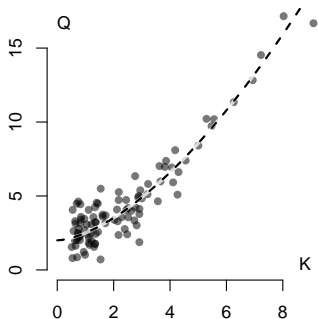
$$Q = \beta_0 + \beta_1 K^{\beta_2} + U, \quad K > 0.$$

(AKA 'Box-Cox transformed regressors'; assume that  $K$  is exogenous.)

$\beta_2 > 1$ : increasing return to scale.

The coefficients  $(\beta_0, \beta_1, \beta_2)$  are estimated with 100 data points via non-linear least squares:

$$\min f(\beta) := \sum_{i=1}^n (Q_i - \beta_0 - \beta_1 K_i^{\beta_2})^2$$



# NLS problem, necessary conditions

---

$$\min f(\beta) := \sum_{i=1}^n (Q_i - \beta_0 - \beta_1 K_i^{\beta_2})^2$$

Deriving the first-order conditions,  $\nabla_f(\beta) = 0$ :

$$\begin{cases} \frac{\partial}{\partial \beta_0} f(\beta) = -2 \sum_i (Q_i - \beta_0 - \beta_1 K_i^{\beta_2}) = 0 \\ \frac{\partial}{\partial \beta_1} f(\beta) = -2 \sum_i (Q_i - \beta_0 - \beta_1 K_i^{\beta_2}) K_i^{\beta_2} = 0 \\ \frac{\partial}{\partial \beta_2} f(\beta) = -2 \sum_i (Q_i - \beta_0 - \beta_1 K_i^{\beta_2}) \beta_1 K_i^{\beta_2} \log K_i = 0 \end{cases}$$

3 equations with 3 unknowns... but they are non-linear! No general solution available. (Only  $\hat{\beta}_0 = \bar{Q} - \hat{\beta}_1 \bar{K}^{\hat{\beta}_2}$ .)

# Partial grid search example for NLS

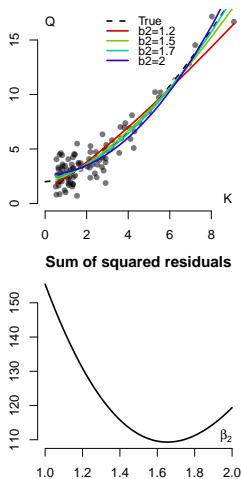
$$Q = \beta_0 + \beta_1 K^{\beta_2} + U, \beta = (2, 0.5, 1.6)'$$

Note that if  $\beta_2$  is fixed, the problem can be solved by OLS!

**Step 1.** Use a grid of values  $\tilde{\beta}_2 = (1.00, 1.01, \dots, 1.99, 2.00)$ . For each  $\tilde{\beta}_2$ , minimise  $f(\beta_0, \beta_1, \tilde{\beta}_2)$  w. r. t.  $(\beta_0, \beta_1)$  (using OLS).

**Step 2.** Choose the  $\beta_2$  that minimises the sum of squared residuals.

Here,  $\hat{\beta}_2 = 1.65$ , yielding  $(\hat{\beta}_0, \hat{\beta}_1) = (2.21, 0.43)$ .





# Grid-evaluation implementation

---

```
set.seed(1)
n <- 100
X <- sort(rchisq(n, df = 2)) + 0.5
f <- function(x, b) b[1] + b[2]*x^b[3]
Y <- f(X, c(2, .5, 1.6)) + rnorm(100)

b2.grid <- seq(1, 2, 0.01)
rss <- function(b2) sum(resid(lm(Y ~ I(X^b2))))^2
rss.grid <- sapply(b2.grid, rss)
b2.opt <- b2.grid[which.min(rss.grid)] # 1.65
# Now computing b0 and b1 for this fixed b2
b01 <- coef(lm(Y ~ I(X^b2.opt))) # 2.213 0.434
```

# Drawback of grid-assisted optimisation

---

1. The FOC might be inexact if the grid is not fine enough
  - In the example above,  $\nabla_f(\hat{\beta}) = (0, 0, -0.74) \neq 0$
  - Solution: refine the grid
2. Not suitable for problems with too many parameters responsible for complex behaviour of  $f$ 
  - The grid in high dimensions is very sparse
3. Requires inspired guesses about the plausible values of fixed parameters
  - Unlucky guesses may result in non-global local optima
4. For economists: Inference is complicated (how to compute standard errors?)

# Deterministic derivative-free optimisation

---

If one function evaluation is costly, then, instead of a grid, only one initial value must be chosen.

- Choose an initial value  $\theta_0$
- Evaluate  $f$  at several points around  $\theta_0$
- If the function value becomes lower at some  $\theta_1 \neq \theta_0$ , evaluate  $f$  at several points around  $\theta_1$
- Continue evaluating in the vicinity of new 'best' points until no improvement can be found or the maximum number of iterations has been reached

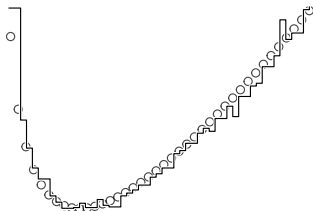
# One-dimen. optimisation on an interval

If  $f$  is differentiable in  $[a, b]$ , then, minimising  $f$  is equivalent to finding all  $\theta^*$ :  $f'(\theta^*) = 0$  and checking the second-order conditions.

Numerical derivatives can be unstable or inaccurate:

- $x + h$  is rounded towards the nearest representable  $[x + h]$
- $f([x + h])$  is rounded towards the nearest representable  $\hat{f}([x + h])$
- If  $f$  is unimodal,  $\hat{f}$  may be not

Therefore, only  $\hat{f}$  should be used to locate the optimum.

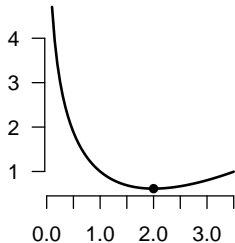


# One-dimensional minimisation in R

- Choose an interval *known to contain the minimum*
- Use bisection / golden section / parabolic interpolation

`optimise()` returns a list with two values: estimated minimum and function value at the minimum.

```
f <- \(x) x - 2*log(x)
o <- optimise(
  interval=c(.1, 3.5),
  f = f, tol = 1e-8)
print(unlist(o), 16)
#>           minimum
#> 2.000000000025565
#>           objective
#> 0.61370563888011
```



# Issues with tolerance

---

Tolerance means different things in different optimisers: gradient tolerance,  $\Delta f$  tolerance, or  $\Delta\theta$  tolerance.

In `optimise()`, `tol` determines the contraction limit:  $f$  is never evaluated at two points closer than  $\theta^*\sqrt{\epsilon} + \text{tol}/3$ .

Therefore, setting `tol = 1e-12` or `1e-15` yields no change.

- Always check the stopping criterion in a method
- Regardless of the method, requesting anything tighter than  $\epsilon$  makes no sense
- In most applications,  $\sqrt{\epsilon} \approx 1.5e-8$  is sufficient
  - Numerical optimisers can rarely do better than  $\sqrt{\epsilon}$  relative accuracy; therefore, all subsequent calculations will be only *this* accurate

# Nelder-Mead simplex method idea

---

It is so good, it is the default in R `optim()`.

- Given a starting point in  $d$ -dimensional space, construct a simplex of a small radius
- Reflect, compact, expand, or shrink the simplex depending on their configuration relative to the centroid
- Terminate when the simplex size is small enough

# Nelder-Mead visualisation

---

Live demonstration time!

Line 685 in `session06.R`



# Nelder-Mead implementation

---

Default method in `optim()`:

```
library(mvtnorm)
f <- \(x) # Our bimodal function
  -0.25*dmvnorm(x, mean=c(1,1), sigma = diag(2)/2)
  -0.75*dmvnorm(x, mean=c(2,3), sigma=diag(2))
o <- optim(par = c(0, 0), fn = f)
```

Track the progress via control list:

```
optim(c(0, 0), f,
      control = list(trace = 2, reltol = 1e-6))

#> Nelder-Mead direct search function minimizer
#> function value for initial parameters = -0.010949
#> Stepsize computed as 0.100000
#> BUILD          3 -0.010949 -0.013264
#> EXTENSION     5 -0.013241 -0.019132
#> LO-REDUCTION  7 -0.013264 -0.019132
#> ...
```

# Coordinate descent

---

Popular approach (e. g. used in LASSO regression).

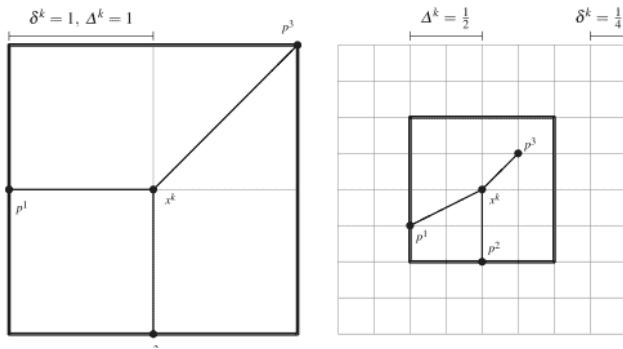
- Choose an initial value  $\theta_0$
- For all coordinates  $\theta_0^{(1)}, \dots, \theta_0^{(k)}$ :
  1. Find the direction of the decrease – either left or right
  2. Move in that direction until the function starts growing again
  3. Do the same for the next coordinate
- Loop until convergence

Not too hard to implement manually.

# Mesh adaptive direct search (MADS)

Remember grid search? This is an improvement: create variable-size meshes and refine them.

`dfoptim::mads()` implements this method.



Credit: Audet, C. & Hare, W. (2017). Derivative-free and blackbox optimization..

Any questions on gradient-free methods?

# **Gradient-based methods**

# Descent methods

---

Suppose that  $f$  is to be minimised and one has an initial value  $\theta_0$ .

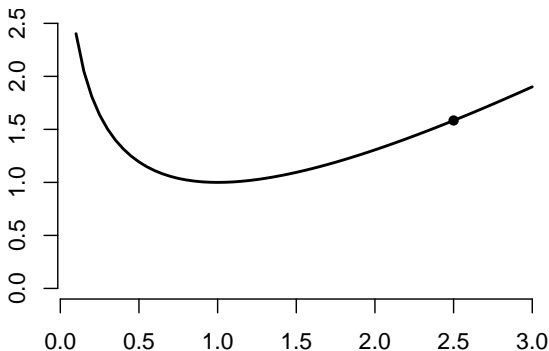
1. Find a direction  $t$  (a vector) in which  $f$  decreases (i. e.  $f(\theta_0 + \varepsilon t) < f(\theta_0)$  for a very small  $\varepsilon$ );
2. Try making a step in that direction, evaluate  $f(\theta_0 + t)$ 
  - If an improvement has been attained, let  $\theta_1 := \theta_0 + t$ , go to step 1
  - If  $f(\theta_0 + t) > f(\theta_0)$ , try shrinking the step: take some  $\beta \in (0, 1)$  such that  $f(\theta_0 + \beta t) < f(\theta_0)$ ; go to step 1

Stopping rule: if the gradient, function change, or step is close to zero, terminate.

# Example: Gradient descent in 1D

$$f(\theta) := \theta - \log \theta, \quad \nabla_f = 1 - 1/\theta$$

Initial value and step:  $\theta_0 = 2.5$ ,  $-\nabla_f(\theta_0) = -0.6$ .



## Example: Gradient descent in 1D (steps)

$$f(\theta) := \theta - \log \theta, \quad \nabla_f = 1 - 1/\theta$$

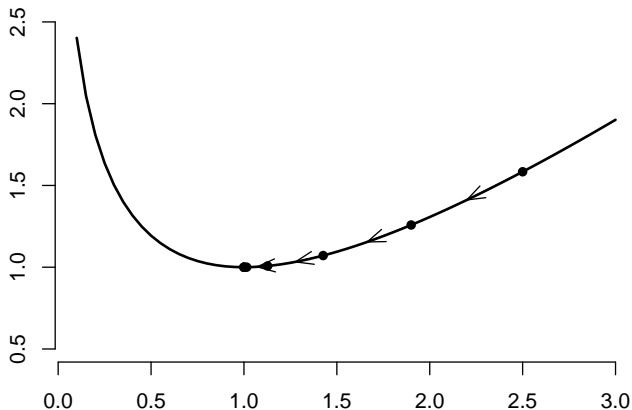
We are to the right of the global minimum, therefore, the correct direction is **left** (negative steps).

Iter	$\theta$	$f(\theta)$	step = $-\nabla_f(\theta)$
1	2.50000000	1.58370927	-0.60000000
2	1.90000000	1.25814611	-0.47368421
3	1.42631579	1.07122104	-0.29889299
4	1.12742280	1.00748848	-0.11302131
5	1.01440149	1.00010272	-0.01419703
6	1.00020446	1.00000002	-0.00020442
7	1.00000004	1.00000000	-0.00000004
8	1.00000000	1.00000000	1.78e-15



# Example: Gradient descent in 1D (plot)

$$f(\theta) := \theta - \log \theta, \quad \nabla_f = 1 - 1/\theta$$



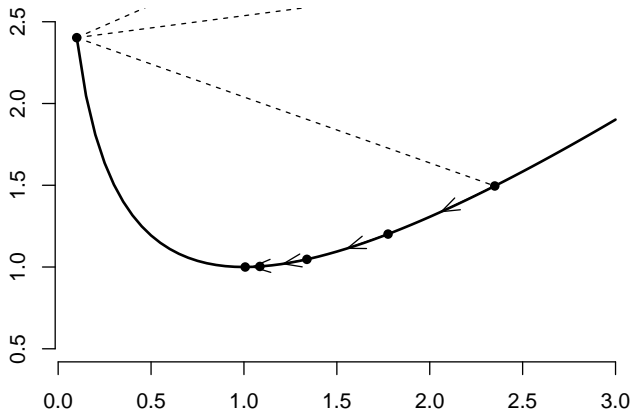
# Initial value matters

$$f(\theta) := \theta - \log \theta, \quad \nabla_f = 1 - 1/\theta$$

Now we start from  $\theta_0 = 0.1$ .

Iter	$\theta$	$f(\theta)$	step = $-\nabla_f(\theta)$
1	0.10000000	2.40258509	9.00000000
Substep: step too large ( $f = 6.89$ ), halving.			
Substep: step too large ( $f = 3.07$ ), halving.			
2	2.35000000	1.49558467	-0.57446809
3	1.77553191	1.20143187	-0.43678850
4	1.33874342	1.04701199	-0.25303088
5	1.08571254	1.00347605	-0.07894589
6	1.00676665	1.00002279	-0.00672117
7	1.00004548	1.00000000	-0.00004548
8	1.00000000	1.00000000	2.07e-09

# Backtracking



If the search direction is correct but the step size does not yield a reduction, **backtracking** (step shrinking) is needed.

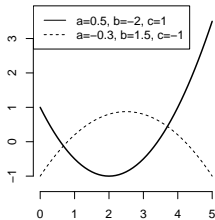
# One-dimensional quadratic functions

The equation of a parabola:

$$y = ax^2 + bx + c$$

Its vertex is located at  $x^* = -\frac{b}{2a}$ .

$$\nabla_y = y' = 2ax + b, \nabla_y^2 = y'' = 2a.$$



- At any point  $x_0$ , it suffices to make a step of size  $-(x_0 + \frac{b}{2a})$  to end up at the vertex:  $x_0 - (x_0 + \frac{b}{2a}) = -\frac{b}{2a}$
- Expressing the step size via  $\nabla_y$  and  $\nabla_y^2$ :

$$-\frac{\nabla_y(x)}{\nabla_y^2(x)} = -\frac{2ax + b}{2a} = -x - \frac{b}{2a}$$

This is true  $\forall x \in \mathbb{R}$ !

# Quadratic function optim. via derivatives

If at any point  $x_0 \in \mathbb{R}$ , one computes  $\nabla_y(x_0)$  and  $\nabla_y^2(x_0)$ ,

$$\underbrace{x_0}_{\text{initial value}} + \underbrace{\left( -\frac{\nabla_y(x_0)}{\nabla_y^2(x_0)} \right)}_{\text{step}}$$

is the global optimum of  $y(x)$ !

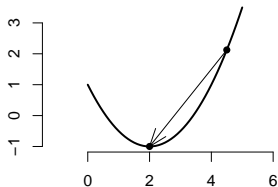
If one-dimensional  $y(\theta)$  is defined in the software and turns out to be quadratic, then, it suffices to compute the first two derivatives to find the global optimum.

Numerical algorithms for evaluating derivatives are usually available in good statistical packages (not EViews).

# Example: one-dimensional quadratic

Let  $y$  be a quadratic function with mysterious coefficients.

- Ask R to evaluate  $y'$  and  $y''$  at  $x_0 = 4.5$
- R returns  $y'(4.5) = 2.5$ ,  $y''(4.5) = 1$
- Step size:  $-y' / y'' = -2.5 / 1 = -2.5$
- Therefore,  $x^* = 4.5 - 2.5 = 2$



Forsooth!

# Multi-dimensional quadratic functions

---

Now consider a multi-variate quadratic function:

$$f(\theta) := \theta' A \theta + b' \theta + c,$$

where  $A$  is a symmetric matrix of full column rank.

$$\nabla_f(\theta) = 2A\theta + b, \quad \nabla_f^2(\theta) = 2A$$

Solving  $\nabla_f(\theta) = 2A\theta + b = 0$  yields  $\theta^* = -0.5A^{-1}b$ .

At any  $\theta_0$ , the global optimum is found by evaluating

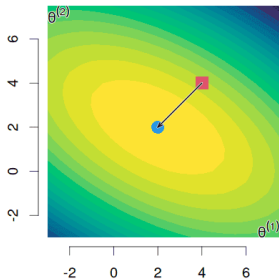
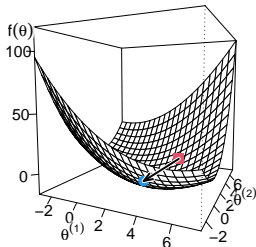
$$\underbrace{\theta_0}_{\text{initial value}} + \underbrace{\left(-[\nabla_f^2(\theta_0)]^{-1} \cdot \nabla_f(\theta_0)\right)}_{\text{step}}$$

# Example: multi-dimensional quadratic

Let  $f$  be a quadratic function of two parameters with mysterious coefficients.

- Evaluate  $\nabla_f$  and  $\nabla_f^2$  at  $\theta_0 = \begin{pmatrix} 4 \\ 4 \end{pmatrix}$
- R returns  $\nabla_f \begin{pmatrix} 4 \\ 4 \end{pmatrix} = \begin{pmatrix} 7 \\ 11 \end{pmatrix}$ ,  
 $\nabla_f^2 \begin{pmatrix} 4 \\ 4 \end{pmatrix} = \begin{pmatrix} 2 & 1.5 \\ 1.5 & 4 \end{pmatrix}$
- Step size:  $-\begin{pmatrix} 2 & 1.5 \\ 1.5 & 4 \end{pmatrix}^{-1} \begin{pmatrix} 7 \\ 11 \end{pmatrix} =$   
 $-\frac{4}{23} \begin{pmatrix} 4 & -1.5 \\ -1.5 & 2 \end{pmatrix} \begin{pmatrix} 7 \\ 11 \end{pmatrix} = -\frac{4}{23} \begin{pmatrix} 11.5 \\ 11.5 \end{pmatrix} = \begin{pmatrix} -2 \\ -2 \end{pmatrix}$
- Hence,  $\theta^* = \begin{pmatrix} 4 \\ 4 \end{pmatrix} - \begin{pmatrix} 2 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$

The computer returned the correct answer with just 2 vectors and 1 matrix!





# Multi-dimensional convex functions

---

Suppose that the objective function  $f$  is not quadratic, but just **strictly convex**.

Then, the local minimum is the global minimum, and the problem has *at most* one optimum.

If  $f$  is also strongly smooth (i. e.  $\nabla_f^2$  is bounded), one can use gradient descent to find improvements such that  $f(\theta_{k+1}) < f(\theta_k)$ , and the algorithm is guaranteed to converge geometrically.

If only  $\nabla_f$  is bounded but not  $\nabla_f^2$ , convergence is simply slower.

# Quadratic approx. of convex function

---

Suppose that we approximate a convex function  $f$  in the neighbourhood of some value  $\theta_0$  with a quadratic function  $q$  using second-order Taylor series:

$$f(\theta) \approx q(\theta) := f(\theta_0) + \nabla_f^T(\theta_0)(\theta - \theta_0) + \frac{1}{2}(\theta - \theta_0)' \nabla_f^2(\theta_0)(\theta - \theta_0)$$

Then, we can solve the 'wrong' problem of minimising  $q$  and (hopefully) become closer to minimising  $f$ .

# Newton–(Raphson) method

---

Take a valid initial guess  $\theta_0$ .

1. Compute  $\nabla_f$  and  $\nabla_f^2$  at the current guess via numerical differentiation
2. Let  $\theta_{k+1} := \theta_k - [\nabla_f^2(\theta_k)]^{-1} \nabla_f(\theta_k)$
3. Repeat until the stopping criterion is met

# Newton method example

---

Live demonstration time!

# Problems with Newton–(Raphson) method

---

- Can break down if  $\nabla_f^2$  is degenerate (i. e.  $f$  not *strictly* convex or even non-convex)
  - Therefore, can diverge if the step is huge
  - Remedy 1: diagonalise  $\nabla_f^2$ , keep the eigenvectors, replace the negative eigenvalues with  $\varepsilon > 0$ , convert back
  - Remedy 2: add  $\lambda I$  for a sufficiently large  $\lambda$  (as  $\lambda \rightarrow \infty$ ,  $\lambda I + \nabla_f^2 \approx \lambda I \Rightarrow$  the NR method becomes gradient descent with step  $1/\lambda$ ); also known as Tikhonov regularisation (used in ridge regression)
- Can be trapped in a loop
  - Quite rare with real-world data

Improvement: **back-tracking**, i. e. multiply the step at each iteration by  $\beta \in (0, 1)$

# Exact line search

---

If a step is too large, we shrink it by a factor of  $\beta$ , but how do we know if it could be too small?

For any  $\theta_0$ , for any valid descent direction  $t$  such that  $f(\theta_0 + \varepsilon t) < f(\theta_0)$ , find the optimal step size  $s$  that yields the lowest  $f(\theta_0 + st)$ .

This method transforms a multivariate optimisation problem to a univariate one.

# Simplest line search variant

---

At any step of the descent, start with  $s = 1$  and span multiplier  $\alpha = 0.2$ .

1. Evaluate  $f_1 = f(\theta_0 + (1 - \alpha)st)$ ,  $f_2 = f(\theta_0 + st)$ ,  
 $f_3 = f(\theta_0 + (1 + \alpha)st)$ 
  - If  $f_1 < \min(f_2, f_3)$ , then,  $s$  is too large; multiply  $s$  by  $(1 - \alpha)$ ;
  - If  $f_3 < \min(f_1, f_2)$ , then,  $s$  is too small; multiply  $s$  by  $(1 + \alpha)$
  - If  $f_2 < \min(f_1, f_3)$ , the bracket is too wide; multiply  $\alpha$  by 0.8
2. Repeat until  $\alpha < 0.02$  (at most 10 times)

This is quick, dirty, and inaccurate for the sake of clarity. There are much better implementations (Brent's method, golden-section search etc.).

# Dropping the Hessian

---

In applied work, Hessians are usually not computed directly:

- If  $\dim \theta = k$ , computing  $\nabla^2 f(\theta)$  takes  $k^2$  operations – slow
- If  $f(\theta)$  exhibits near-linear behaviour, inverting  $\nabla^2 f(\theta)$  is problematic
- Numerical  $\hat{\nabla}^2 f(\theta)$  may be very inaccurate
  - Ill-conditioned
  - Non-symmetrical
  - Inaccurate due to repeated (instead of one-time) differencing

$\nabla^2 f$  is typically substituted with a completely different object to allow more iterations in the same time.



# Inexact line search

---

Steepest descent makes updates of the form

$$\theta_{k+1} = \theta_k - \alpha_k \nabla_f(\theta_k)$$

The greedy exact search chooses  $\alpha_k$  as

$$\arg \min_{\alpha \geq 0} f(\theta_k - \alpha \nabla_f(\theta_k))$$

This search method may behave poorly in practice. If the contour lines of  $f$  resemble long valleys, the sequence  $\{\theta_k\}$  displays a zig-zagging trajectory; the speed of convergence is very slow.

# Weak Wolfe conditions for step size $\alpha$

Define **directional derivatives** for a *unit vector*  $v$ :

$$\nabla_v f(\theta) := \lim_{h \rightarrow 0} \frac{f(\theta + hv) - f(\theta)}{h} = \nabla f(\theta)'v$$

**1. Sufficient decrease** (Armijo condition),  $c_1 \approx 0.0001$ :

$$f(\theta_k + \alpha v_k) \leq f(\theta_k) + c_1 \alpha \nabla_v f(\theta)$$

**2. Curvature condition** ( $c_2 > c_1$ ,  $c_2 < 1$ ,  $c_2 \approx 0.9$ ):

$$\nabla_v f(\theta_k + \alpha v_k)'v_k \geq c_2 \nabla_v f(\theta_k)$$

(1): The decrease should be at least a small fraction ( $c_1$ ) of the gradient. (2): The slope at the new point should be at least  $c_2$  times the original slope.

# BFGS optimiser

---

Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm: the most popular gradient-based technique. (One of the best ones in practice for problems in economics, too.)

Take  $\theta_0, \hat{H}_0 = \beta I$ . Let  $A_k = \widehat{H}_k^{-1}$  be the *approximation of the inverse Hessian*. Iterate:

- Choose the line search direction  $v_k = -A_k \nabla_f(\theta_k)$
- Choose the step size  $\alpha_k$  satisfying the weak Wolfe conditions and set  $\theta_{k+1} = \theta_k + \alpha_k v_k$
- Apply a *very clever* formula to update  $A_k$  directly without any inverses and without computing  $H_k$  directly

Stop when  $\frac{|f(\theta_{k+1}) - f(\theta_k)|}{f(\theta_k)} < \text{rel. tol}$  (typically  $1e-8$ ).

# BFGS with memory or box constraints

---

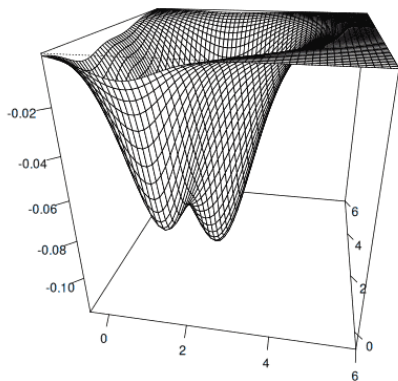
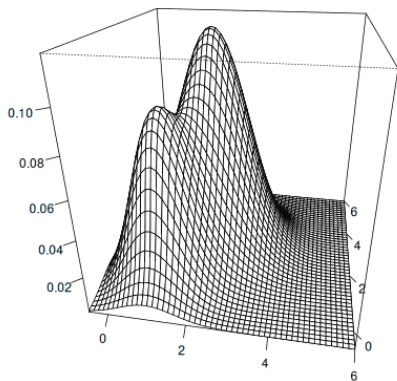
**L-BFGS:** limited-memory BFGS. Does not store the matrix  $A_k$ , computes it from the last few  $m$  values of  $f$  and  $\nabla_f$ .

**BFGS-B:** BFGS with box constraints, i. e.  $\underline{\theta} \leq \theta \leq \bar{\theta}$ . Identifies fixed and free variables at every step, updates free variables only.

R has `optim(..., method = "BFGS")` and `optim(..., method = "L-BFGS-B")`. In practice, go for vanilla BFGS (unless the dimensionality of the problem consumes the RAM): more accurate.

The `lbfgsb3c` package provides an updated (2011) L-BFGS-B version.

# BFGS example function shape



Left: original function. Right: input to `optim()`.

# Calling BFGS without gradients

---

```
f <- \(x) # Our bimodal function
  -0.25*mvtnorm::dmvnorm(x, mean=c(1,1), sigma = diag(2)/2) -
  0.75*mvtnorm::dmvnorm(x, mean=c(2,3), sigma=diag(2))

ctrl <- list(reltol = 1e-8, trace = 2,
  REPORT = 1, ndeps = rep(1e-5, 2))
o1 <- optim(c(0, 0), f, method = "BFGS",
  control = ctrl)
o2 <- optim(c(4, 4), f, method = "BFGS",
  control = ctrl)
```

**NB:** ndeps is very important for numerical accuracy. Recall that  $h^* \sim \sqrt[3]{\epsilon}$  for central differences, but the default is 0.001 – too large.

# Initial value matters

---

```
str(o1)
```

```
#> $ par           : num [1:2] 1.09 1.18  
#> $ value         : num -0.0915  
#> $ counts        : Named int [1:2] 28 26  
#> $ convergence: int 0
```

```
str(o2)
```

```
#> $ par           : num [1:2] 1.99 2.98  
#> $ value         : num -0.12  
#> $ counts        : Named int [1:2] 40 36  
#> $ convergence: int 0
```

Which one is better? Are we sure that there are no other local optima?

# Multi-start

---

Generate starting values randomly in a hyper-cube (other methods may be more appropriate). Here, we try 10 points:

```
set.seed(1)
initval <- matrix(runif(2*10, 0, 10), ncol = 2)
doOpt <- function(x) {
  ret <- optim(par = x, f, method = "BFGS",
              control = ctrl)
  c(start = x, end = ret$par, val = ret$value)
}
res <- t(apply(initval, 1, doOpt))
res[order(res[, "val"]), ]

start1 start2  end1  end2  val      # <-- best start
2.6551 2.0597 1.9902 2.9804 -0.1199
3.7212 1.7656 1.9902 2.9804 -0.1199
6.2911 3.8004 6.2874 3.7997  0.0000
<...>
9.4468 7.1762 9.4468 7.1762  0.0000
```



# Importance of convergence codes

---

Always check the convergence code. `code = 1`  $\Rightarrow$  maximum iterations reached = convergence not achieved. **The exit code must be zero.**

However, the zero exit code is a **necessary but not sufficient condition** for declaring success and moving on.

- If the exit code is 0 but the counts are 1, the optimiser did not do any work at all and did not move anywhere
- Function with no global minima might still yield convergence = 0 when the  $f$  tapers out into a plateau

# Silent failure example

---

```
optim(c(10, 10), f, method = "BFGS",
      control = ctrl)
#> $ par           : num [1:2] 10 10
#> $ value          : num -3.46e-26
#> $ counts         : Named int [1:2] 1 1
#> $ convergence: int 0
```

**Counts:** the optimiser stopped after the first iteration.

**Reason:** the function is almost flat in that area.

# False convergence example

---

The user may forget to multiply the 'good' function for maximisation by  $-1$ , and the optimiser minimises, moving away from the desired maximum.

```
f2 <- \(x) -f(x)
optim(c(4, 4), f2, method = "BFGS",
      control = ctrl)
#> $ par      : num [1:2] 7.54 5.77
#> $ value    : num 5.65e-10
#> $ counts   : Named int [1:2] 27 26
#> $ convergence: int 0
```

# Nonsensical convergence example

---

Linear functions cannot be globally optimised:

```
g <- function(x) x[1] - x[2]
optim(c(4, 4), g, method="BFGS", control=ctrl)

#> initial value 0.000000
#> iter 2 value -2.000000
#> iter 3 value -4.000000
#> iter 4 value -180143084373.758240
#> iter 4 value -180143084376.809998
#> final value -180143084376.809998
#> converged
#> $ par : num [1:2] -9.01e+10 9.01e+10
#> $ value : num -1.8e+11
#> $ counts : Named int [1:2] 5 4
#> $ convergence: int 0
```

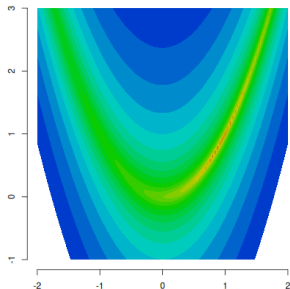
However, the exit code is still zero.

# BFGS with analytical gradients

Optimise the famous Rosenbrock 'banana' function:

```
f <- \(x) 100 * (x[2] - x[1]^2)^2 + (1 - x[1])^2
nablaf <- \(x) {
  c(-400 * x1*(x[2]-x[1]^2) - 2*(1 - x[1]),
    200 * (x[2] - x[1]^2))
}
```

```
optim(c(0.9, 0.9),
      fn = f,
      gr = nablaf,
      method = "BFGS",
      control = ctrl)
#> $ par      : 1 1
#> $ value    : 1.54e-19
#> $ counts   : 33 16
#> $ convergence: 0
```



# Non-convergence example

---

The Rosenbrock function is famously slow to converge if the initial value is not close to the optimum (1, 1).

The conjugate gradient in `optim()` fails:

```
optim(c(-1.2, 1), fn = f, method = "CG")
```

```
#> $ par      : -0.765 0.593  
#> $ value    : 3.11  
#> $ counts   : 402 101  
#> $ convergence: 1
```

Increasing the number of iterations helps:

```
optim(c(-1.2, 1), fn = f, method = "CG",  
      control = list(maxit = 10000))
```

```
#> $ par      : 1 0.999  
#> $ value    : 8.12e-08  
#> $ counts   : 19938 4975  
#> $ convergence: int 0
```

# Diagnosing non-convergence

---

- Increase the number of iterations
- If  $\dim \theta = 2$ , simply visualise the function in 2D via `contour()` or 3D via `persp()`
- Try a wide range of starting values

# Other gradient-based optimisers

---

- `nlm()`: more fragile, requires gradients, but for functions looking like quadratics, beats BFGS in terms of convergence speed
  - Best convergence check: gradient tolerance
- `nlmminb()`: similar to BFGS, supports box constraints
  - Can rely on the supplied Hessian
  - `Rsolnp`: `solnp()`: supports box constraints,  $g(\theta) = 0$ , and  $\underline{h} \leq h(\theta) \leq \bar{h}$
- `optimx`: a package containing multiple Quasi-Newton optimisers: conjugate gradient (Fletcher & Reeves), quadratic approximations (Powell)
  - `nlm` support is not fully featured



# optimx: unifying framework

Try many optimisers and compare their performance:

```
library(optimx)
o <- optimx(par = c(-1.2, 1), fn = f,
  gr = nablaf,
  control = list(kkt=TRUE, all.methods=TRUE))
```

	p1	p2	value	fevals	gevals	niter	convcode	kkt1	kkt2	xtime
BFGS	1.00000	1.00000	9.5949e-18	110	43	NA	0	TRUE	TRUE	0.000
CG	-0.76483	0.59275	3.1065e+00	402	101	NA	1	FALSE	FALSE	0.055
Nelder-Mead	1.00026	1.00050	8.8252e-08	195	NA	NA	0	FALSE	TRUE	0.000
L-BFGS-B	0.99999	0.99999	2.2675e-13	47	47	NA	0	TRUE	TRUE	0.000
nlm	1.00000	1.00000	1.1820e-20	NA	NA	24	0	TRUE	TRUE	0.000
nlmminb	1.00000	1.00000	4.2919e-22	43	36	35	0	TRUE	TRUE	0.000
spg	0.99980	0.99960	3.8980e-08	141	NA	112	0	TRUE	TRUE	0.004
ucminf	1.00000	1.00000	2.7256e-17	38	38	NA	0	TRUE	TRUE	0.000
Rcgmin	0.99999	0.99999	2.6795e-14	111	54	NA	0	TRUE	TRUE	0.001
Rvmin	1.00000	1.00000	1.2325e-32	59	39	NA	2	TRUE	TRUE	0.003
newuoa	1.00000	0.99999	2.1777e-15	257	NA	NA	0	TRUE	TRUE	0.001
bobyqa	1.00000	1.00001	2.8448e-13	290	NA	NA	0	TRUE	TRUE	0.001

# Constrained optimisation via NLOpt

---

NLOpt is a great cross-platform library for optimisation with inequality and equality constraints.

$$\begin{aligned} \min f(\theta), \quad \theta \in \mathbb{R}^k \\ \text{s. t. } g(\theta) \geq 0, \quad h(\theta) = 0, \quad \underline{\theta} \leq \theta \leq \bar{\theta} \end{aligned}$$

Inputs:

- $f$  and  $\nabla_f$ 
  - If analytical  $\nabla_f$  is unavailable, provide a numerical one
- Optional:  $g$  and  $\nabla_g$
- Optional:  $h$  and  $\nabla_h$

# NLopt in action

---

Maximising the function from Slide 9 via augmented Lagrange multiplier method:

```
library(mvtnorm)
f <- \(x)
  -0.25*dmvnorm(x, mean = c(1, 1), sigma = diag(2)/2)
  -0.75*dmvnorm(x, mean = c(2, 3), sigma = diag(2))
fp <- \(x) numDeriv::grad(func = f, x = x)
g <- \(x) x[1]^2 + x[2]^2 - 4
gp <- \(x) matrix(2*x, nrow = 1)
o <- nloptr::nloptr(x0 = c(2, 2),
  eval_f = f, eval_grad_f = fp,
  eval_g_eq = g, eval_jac_g_eq = gp,
  opts = list(algorithm = "NLOPT_LD_AUGLAG_EQ",
  local_opts =
    list(algorithm = "NLOPT_LD_MMA", maxeval = 100),
  maxeval = 1000, print_level = 3))
```

# NLopt output

---

```
print(o)
#> Minimization using NLopt version 2.7.1
#> NLopt solver status: 4 ( NLOPT_XTOL_REACHED:
  ↳ Optimization stopped because xtol_rel or xtol_abs
  ↳ (above) was reached. )
#> Number of Iterations.....: 105
#> Termination conditions: maxeval: 1000
#> Number of inequality constraints: 0
#> Number of equality constraints: 1
#> Optimal value of objective function:
  ↳ -0.0867539596779702
#> Optimal value of controls: 1.292447 1.526297
```

Any questions on gradient-based methods?

# **Stochastic methods**

# Stochastic optimisers

---

Deterministic algorithms suffer from the common malady: they need a **good initial value**. Without a good initial value, optimisation is doomed: the solver may diverge, or the function may be not defined.

Knowing reasonable range enables the following heuristic:

- Randomly generate many-many points in a subset of  $\mathbb{R}^{\dim \theta}$ , evaluate  $f$  at them
- Discard failed / bad solutions
- Next time, generate many-many points in the range where  $f$  was taking reasonable values
- Hope for improvement!

# How does one multistart?

---

Many deterministic searches are the poor man's stochastic search:

- Generating many initial values and running many optimisation problems is costly
- Use some information from the literature to have a range of plausible values
- `optimr::multistart()` provides a nice wrapper
- `Rsolnp::gosolnp()` retries if the solver fails



# Benefits of randomised search methods

---

- If  $f$  is non-smooth, discontinuous, or is numerically unstable, can still work
  - Gradient-based methods fail immediately
  - Deterministic gradient-free methods may fail, too
- Should not get stuck in local optima
  - Under certain robustness conditions, given enough time
- May find the global optimum if it lies in the chosen range
  - Under certain conditions, given enough time

# Drawbacks of randomised search methods

---

- Very slow
  - High-dimensional space is incredibly sparse
  - The number of iterations is more important than the 'population' size at each iteration
- The FOC is not checked – may result in false convergence or corner solutions
- An inspired guess about the range of parameter values is still required

# Pure random search

---

Naïve approach with very little *a priori* knowledge.

- Generate a cloud of points with a certain radius  $\sigma$  at centre  $\mu$  (e. g.  $\mathcal{N}(\mu, \sigma^2)$ )
- Evaluate  $f$ , sort by value
- Find the best candidate
- Generate the next point cloud around the best candidate with a slightly smaller radius

May be useful in picking initial values for deterministic optimisers.

# Simulated annealing

---

**Annealing:** heating a material and cooling it down to reduce internal stresses / hardness.

Terminology: 'bad' solution = high energy.

- Start with a random population and a 'high temperature'
- Perturb the population proportionally to the 'energy'
- Lower the temperature

**In simple words:** shuffle the worst points strongly, the best points mildly.

`optim(..., method = "SANN")` is outdated – use `GenSA::GenSA()` or `optimization::optim_sa()`.

# Simulated annealing example

---

```
f <- \(x) -0.25*mvtnorm::dmvnorm(x, mean=c(1, 1), sigma = diag(2)/2) -  
0.75*mvtnorm::dmvnorm(x, mean = c(2, 3), sigma = diag(2))
```

```
library(GenSA)
```

```
set.seed(1)
```

```
o <- GenSA(fn = f,  
  lower = rep(-10, 2), upper = rep(10, 2),  
  control = list(verbose = TRUE))
```

```
o  
#> $ value      : -0.12  
#> $ par        : 1.99 2.98  
#> $ trace.mat  : 1 1 1 1 2 2 2 2 3 3 ...  
#> $ counts     : int 48120
```

Note: almost 50k evaluations!

# Particle swarm

---

Each point represents a 'firefly' flying through a high-dimensional space.

- Each 'firefly' can travel a certain distance in one iteration
- Each 'firefly' has memory of its best position
- Each 'firefly' knows the position of each other firefly

Algorithm:

- Start with a random population moving in random directions
- Update the vector speed of each member based on 3 components: (1) inertia, (2) current global best point, (3) personal best point
- Move the swarm one step and repeat

# Particle swarm methods

---

- Standard PSO: the particles balance between the global and best-known optima based on the topology
- Improved PSO: only the best  $n_g$  particles are used for speed updates
- Fully informed PS: all particles impact all particles
- Weighted FIPS: the contribution of each particle is proportional to its goodness-of-fit

Try all of them, see what works.

# Particle swarm tuning parameters

---

- Number of iterations – **crucial** (the more, the better!)
- Population size – not less than  $10 + 2\sqrt{\dim \theta}$
- Method and topology
- $c_1, c_2$  – importance of current global best and personal best
  - Can be adaptive and time-varying

Less frequency used: inertia weights, constriction factor, regrouping, boundary behaviour.



# Particle swarm in R

---

The most complete package is hydroPSO.

Enable plotting to examine convergence visually:

```
f2 <- function(x) {Sys.sleep(0.02); f(x)}
```

```
library(hydroPSO)
```

```
o <- hydroPSO(fn = f2,  
  lower = rep(-10, 2), upper = rep(10, 2),  
  control = list(parallel = "parallel",  
    REPORT = 1, verbose = T, plot = T))
```

On Windows, use `parallel = "parallelWin"`.

hydroPSO writes all the information to disk – any dimensions can be plotted.

# Differential evolution

---

Each point represents an animal living through a high-dimensional space.

Algorithm:

- Start with a random population
- Compute the fitness of each animal
- Every point produces an offspring
  - Cross-over: each 'child' is a linear combination of 'parents'
  - Mutation: this combination is imperfect
- The least fit 'animals' may die
- Repeat over multiple generations

# Differential evolution rules

---

Updates all points:

$$\theta_{i,t+1} = \theta_{i,t} + F \cdot (\theta_{i_2,t} - \theta_{i_1,t})$$

Update the best point:

$$\theta_{i,t+1} = \theta_{\text{best},t} + F \cdot (\theta_{i_2,t} - \theta_{i_1,t})$$

Update current to best:

$$\theta_{i,t+1} = \theta_{i,t} + F \cdot (\theta_{\text{best},t} - \theta_{i,t}) + F \cdot (\theta_{i_2,t} - \theta_{i_1,t})$$

Differencing  $\Rightarrow$  differential evolution.

# Differential evolution parameters

---

- Number of iterations – **crucial** (the more, the better!)
- Population size – not less than  $10 \cdot \dim \theta$
- Mutation rule
- $CR$  – cross-over probability (0.5)
- $F$  – differential weighting factor (0.8)

Less frequency used: % of surviving points.

# Differential evolution in R

---

```
library(DEoptim)
library(parallel)
cl <- makeCluster(4)
clusterExport(cl, "f")
o <- DEoptim(fn = f2,
  lower = rep(-10, 2), upper = rep(10, 2),
  control = list(strategy = 2,
    cluster = cl, storepopfrom = 1))
```

# Many optima in high dimensions

---

Live demonstration time!

# Stochastic hill-climbing

---

Stochastic and gradient-based? Yes!

- Stochastic hill climbing: choose the descent direction at random (with some useful probability)
- Stochastic gradient descent: compute the gradient using only a sub-set of data

# Stochastic gradient descent

---

$$\theta_{k+1} = \theta_k + \eta \tilde{\nabla}_f(\theta_k)$$

$\eta$ : learning rate,  $\tilde{\nabla}_f$  uses a random batch of data.

- Works well with large data sets with big data where the full data set cannot be possibly used
- $\eta$  is adaptive, starts small, tends to zero for convergence
- Can use back-tracking for optimal updates
- Pure SGD uses derivatives of the objective function with only one observation, but the variance is too high.  
Mini-batch gradient descent benefits from vectorised operations (SIMD + parallelism)



Any questions on stochastic methods?

# **Common issues in real applications**

# Numerical accuracy and optimisation

---

- Modern economic models rely heavily on numerical computations
  - The trend seems to be unidirectional, everything is getting even more computationally intensive
- Qualitative economic reasoning depends on the numeric output
- If the output is numerically wrong, statistical inference is meaningless
- More numbers = more sources of error at each step
  - The errors do not 'average each other out'

# Step size and numerical Hessians

In many economic applications, standard errors are obtained by inverting the numerical Hessian. Default method settings can be bad. Always check the routine under the hood. Wrong inference = paper might be retracted.

```
ugarchfit(spec = ..., data = ...)
```

	Estimate	Std. Error	t value	Pr(> t )
omega	0.000010	0.000000	49.1672	0.000000
alpha1	0.126679	0.014911	8.4956	0.000000
beta1	0.731779	0.027181	26.9220	0.000000

```
ugarchfit(..., numderiv.control =  
  list(grad.eps = 1e-7, hess.eps = 1e-7))
```

	Estimate	Std. Error	t value	Pr(> t )
omega	0.000010	0.000005	1.9747	0.048307
alpha1	0.126679	0.037332	3.3933	0.000690
beta1	0.731779	0.095098	7.6950	0.000000

# Tunnelling through local optima

---

- Stochastic tunnelling: randomly hop to a different solution with probability depending on the function difference (and its variants: TRUST 1997, STP 1999 etc.)
  - Detrended fluctuation analysis:  
nonlinearTseries::dfa() to determine long-range correlations and phenomena that might cause multiple optima
- Acceptance (final stage of grief): in certain applications where identification is not the point (e. g. multi-layered neural networks), local minima can produce predictions as accurate as those from the global optimum

# Optimisation speed-up

---

- Parallelise the gradient
  - A package will be uploaded to CRAN soon
- Parallelise function evaluation in stochastic methods
- In nested optimisation, parallelise chunks to reduce overhead
  - Save the progress in chunks as well

# Optimisation with hundreds of parameters

---

- Simultaneous perturbation stochastic approximation: compute approximate gradients by shifting many parameters at once
- Barzilai-Borwein spectral projected gradients
- Update subspaces of parameters
  - Truncated Newton-like methods: symmetric rank 1, truncated BFGS

# Barzilai-Borwein SPG method

Recall the NR update formula:  $\theta_{k+1} := \theta_k - [\nabla_f^2(\theta_k)]^{-1} \nabla_f(\theta_k)$ .

Idea: Assume that  $\nabla_f^2(\theta_k) \approx \sigma_k I$ .

Unlike line search (scale the gradient), in **spectral-projected-gradient** methods, the Hessian is scaled instead using the *previous-step* information:

$$\theta_{k+1} = \theta_k + \alpha_k B_k^{-1} \nabla_f(\theta_k),$$

where  $B_k = \sigma_k I$  is a diagonal matrix satisfying

$$B_k(\theta_k - \theta_{k-1}) = \nabla_f(\theta_k) - \nabla_f(\theta_{k-1})$$

Step size:  $-\nabla_f(\theta_k) / \sigma_k$  (still line search).



# SPG implementation in R

Optimise the 100-dimensional Rosenbrock banana function:

$$f(x) := \sum_{i=2,4,6,\dots} 100(x_i - x_{i-1}^2)^2 + (1 - x_{i-1})^2, \quad x^* = (1, \dots, 1)$$

```
banana <- function(x) {  
  j <- 2 * (1:(length(x)/2))  
  sum(100 * (x[j] - x[j-1]^2)^2 + (1 - x[j-1])^2)  
}
```

```
library(BB)  
set.seed(1)  
BBoptim(par = rnorm(100), fn = banana)  
spg(par = rnorm(100), fn = banana)
```

# Optimisation with millions of parameters

---

- All about stochastic gradient descent in batches
- SGD with momentum, SGD with adaptive delta, SGD with custom learning rates for each parameters
- If necessary, assume sparsity, penalise non-zero parameters

Thank you for your attention!