Heuristics in Finance

E. Schumann

VIP Value Investment Professionals AG, Wilen (SZ)

6th R/Rmetrics Meielisalp Workshop &
Summer School on Computational Finance and Financial Engineering

Meielisalp, 24–28 June 2012

Outline

- Heuristics
- Single-solution methods: Local Search/Threshold Accepting

Principles

- The application matters most. *(Principle 3)*
- Go experiment. *(Principle 5)*

  how do you *know* how to ...? → how do you *decide* how to ...?

Problems → models

given: a question

- how to allocate wealth?
- how to price a security?
- ...

modelling: objective function $f(\cdot)$ and constraints

- financial considerations (how to measure risk/reward?)
- empirical considerations (estimate/forecast/approximate/simulate...)
- computational considerations

Problems → models

what to model?

- goals – be careful what you wish for
- constraints – empirical, technological, regulatory, manpower, ...

example: asset allocation

- assets, data availability
- investment process (frequency of rebalancing, risk characteristics, stop loss, ...)
- forecasts
- scenarios for risk management
- how to evaluate portfolios?

Heuristics

- used in many fields: mathematics, psychology/judgement and decision making, computer science/artificial intelligence, ...
- associated with optimisation, rules of thumb, search
- optimality cannot be proved

→ used in the sense of numerical optimisation technique

Heuristics

- 'good' stochastic approximation of optimum ('good': solution quality/computing time)
- robust to changes to the given problem and to changes in the parameter settings of the heuristic ([changes in] solution quality/computing time)
- easy and simple
- not subjective

Heuristics

given: optimisation problem min $f(x)$; some solution x

'rule':

- simple
- change x → on average improve $f(x)$

→ apply rule many times over (thus computationally intensive) so that *on average/in the long run* the solution is improved

guidelines for rule (Schumann and Ardia, 2011)

- don't be greedy
- trust your luck

Details matter

- 'in principle' *v* practical implementation (many decisions)
- add representation: matters for speed, but also gives flexibility
- implementation matters

Heuristics: generic iterative methods

generate initial solution x^{c}
evaluate $f(x^{\mathrm{c}})$
while stopping condition not met **do**
  create new solution $x^{\mathrm{n}} \in N(x^{\mathrm{c}})$
  evaluate $f(x^{\mathrm{n}})$
  if $A(x^{\mathrm{n}}, \ldots)$ **then** $x^{\mathrm{c}} = x^{\mathrm{n}}$
end while
return x^{c}

x a solution
f objective function (goal function, fitness function, ...)
N neighbourhood
A acceptance (or selection)
stopping rule

Implementation

- solutions are handled through user-defined functions (f, N, A); can be implemented without side-effects
- solution x can be any data structure (not only a numeric vector)
- when to stop? → trade-off resources/quality
  - check available tools
  - experiment
  - profiler (not compiler)

Heuristics: multiple solutions

generate initial solutions X^{c}
evaluate $f(X^{\mathrm{c}})$
while stopping condition not met **do**
  create new solutions $X^{\mathrm{n}} \in N(X^{\mathrm{c}})$
  evaluate $f(X^{\mathrm{n}})$
  if $A(X^{\mathrm{n}}, \ldots)$ **then** $X^{\mathrm{c}} = X^{\mathrm{n}}$
end while
return X^{c}

x a solution
f objective function (goal function, fitness function, ...)
N neighbourhood
A acceptance (or selection)
stopping rule

Heuristics

good:
all heuristics are based on just a few principles

bad:
all heuristics are based on just a few principles

good:

- precisely-described algorithms exist ('canonical versions')
- algorithms robust for different settings
- heal bad settings through more computing time
- judge for yourself: run experiments – and stop when satisfied

Subset sum problem

- given: a list X of numbers
- aim: find subset $x \in X$ such that $\sum x$ is close to s_0

was discussed here
https://stat.ethz.ch/pipermail/r-help/2010-January/226267.html

Subset sum problem

```
> set.seed(8232)
> X <- runif(100)
> ## Find subset that sums up close to 2.0 !
> i <- sort(c(84,54,11,53,88,12,26,45,25,62,96,23,78,77,66,1))
> sum(X[i])
```

```
[1] 2.0005
```

```
> ## --> should be 2.000451

> xHWB <- logical(100L)
> i <- c(84,54,11,53,88,12,26,45,25,62,96,23,78,77,66,1)
> xHWB[i] <- TRUE
> sum(X[xHWB])
```

```
[1] 2.0005
```

Subset sum problem

find subset of `X` whose sum is 2

```
> set.seed(298007324)
> n <- 100L
> X <- runif(n)
```

create known solution `xTRUE`

```
> sort(which(xTRUE))
```

```
 [1]  1  3 13 19 24 29 34 41 42 48 52 60 70 86
```

```
> sum(X[xTRUE])  ## should be 2
```

```
[1] 2
```


Subset sum problem

- representing a solution
- evaluate a solution: objective function
- modify a solution: neighbourhood function
- accept/reject a solution

Representing a solution

element of X either in subset or not: logical vector

`TRUE FALSE FALSE FALSE FALSE`

no magic numbers → collecting all data in `Data`

```
> Data <- list(X  = X,
               n  = 100L,
               s0 = 2)
```

Evaluating a solution

map a solution into a real number

```
> OF <- function(x, X)
     abs(sum(X[x]) - 2)
> OF(xTRUE, X)
```

```
[1] 0
```

with Data

```
> OF <- function(x, Data)
     abs(sum(Data$X[x]) - Data$s0)
> OF(xTRUE, Data)
```

```
[1] 0
```


Evaluating a solution

```
> sum(numeric(0L))
```

```
[1] 0
```

```
> x <- logical(Data$n)
> x[1:5]
```

```
[1] FALSE FALSE FALSE FALSE FALSE
```

```
> OF(x, Data)
```

```
[1] 2
```

Random solutions

```
> makeRandomSol <- function(Data) {
      x <- logical(Data$n)
      k <- sample(Data$n, size = 1L)  ## random cardinality
      x[sample(Data$n, size = k)] <- TRUE
      x
 }
```

```
> OF(makeRandomSol(Data), Data)
```

```
[1] 40.644
```

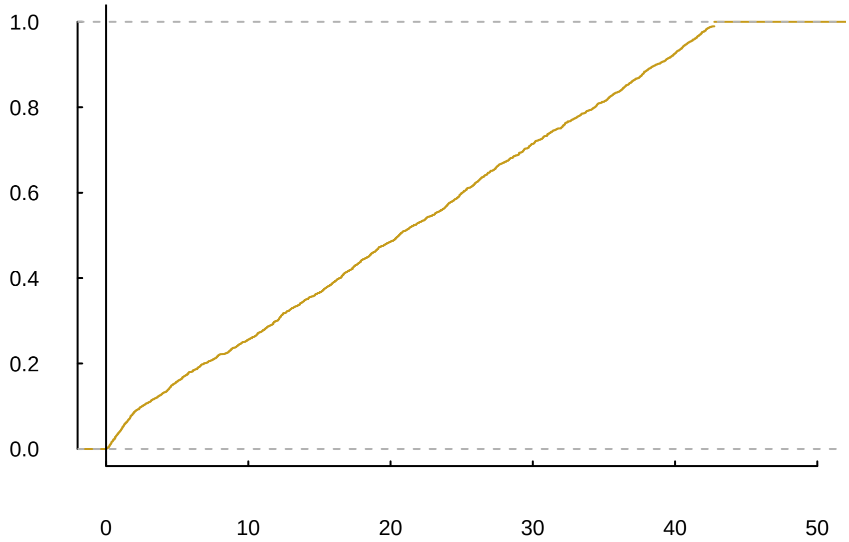
```
> OF(makeRandomSol(Data), Data)
```

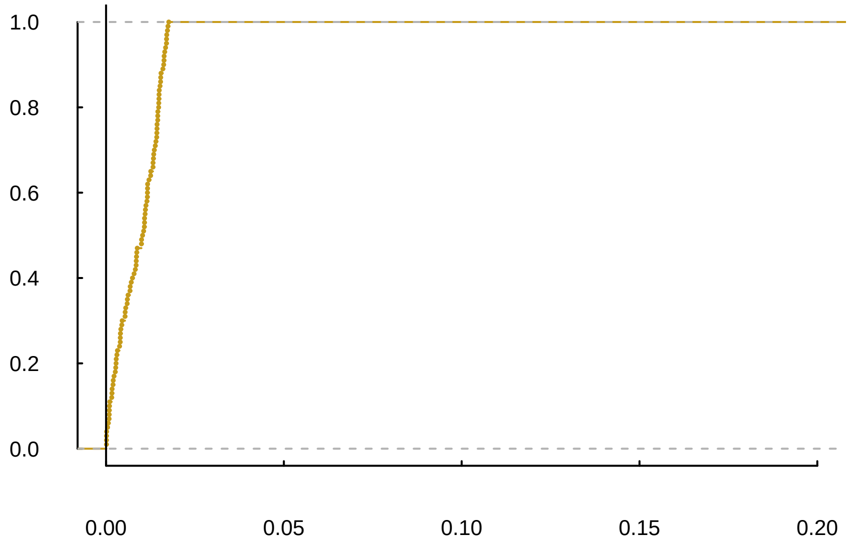
```
[1] 21.632
```

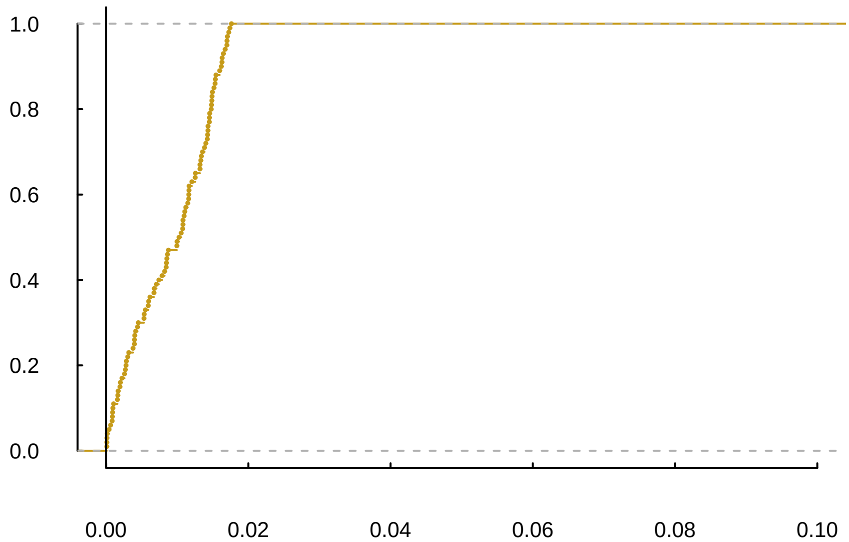

Random solutions

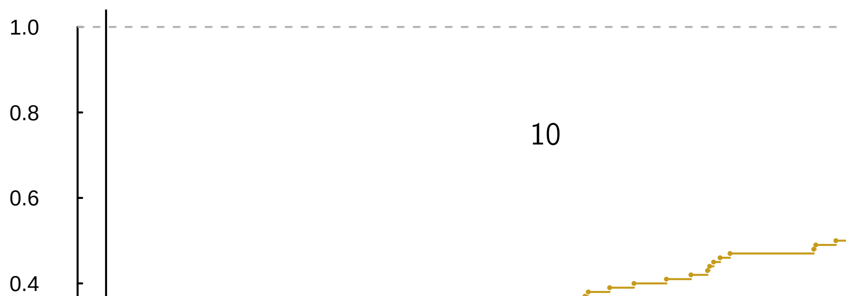
create 100000 of random solutions → keep 100 best solutions
(or use best-of strategy)

Random solutions









Iterative improvement

`TRUE FALSE FALSE FALSE FALSE`

→ change it slightly

`TRUE FALSE TRUE  FALSE FALSE`

→ be greedy: check all neighbours
→ pick one element *randomly* and switch it

```
> Data$size <- 1L
> neighbour <- function(x, Data) {
     p <- sample.int(Data$n, size = Data$size)
     x[p] <- !x[p]
     x
 }
```

But why would that work?

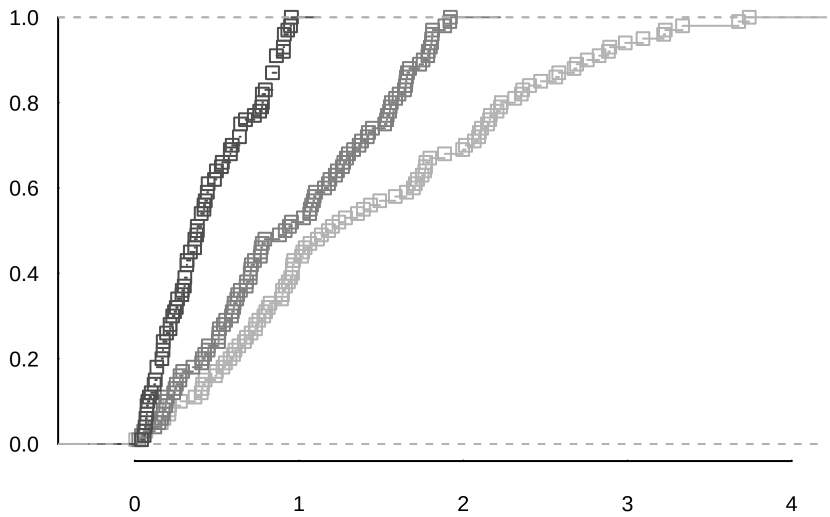
Iterative improvement

create a random solution, create neighbours that differ by

- 1 element
- 3 elements
- 5 elements

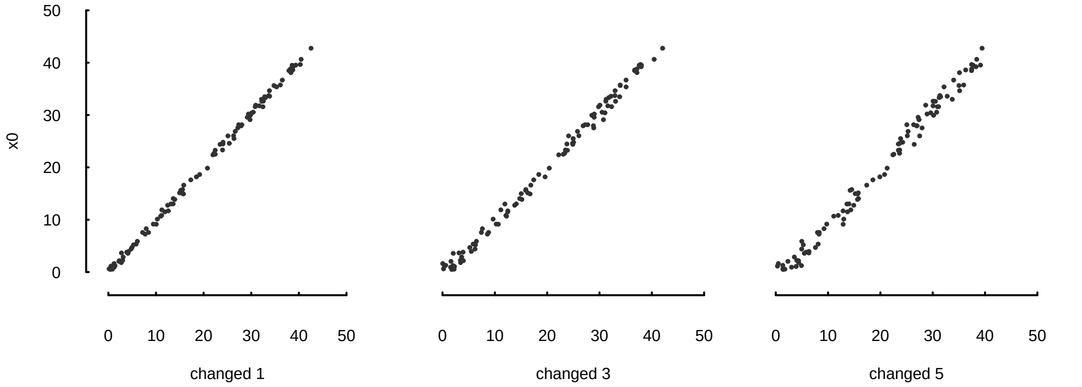
Iterative improvement

larger steps, larger changes in OF



Iterative improvement

quality of solutions is correlated



→

create *meaningful* variation, but keep quality (`OF`) correlated
→ the application determines *meaningful*

Greedy search

while stopping condition not met **do**
  create new solution $x^{\mathrm{n}} \in N(x^{\mathrm{c}})$
  evaluate $f(x^{\mathrm{n}})$
  if $A(x^{\mathrm{n}}, \ldots)$ **then** $x^{\mathrm{c}} = x^{\mathrm{n}}$
end while

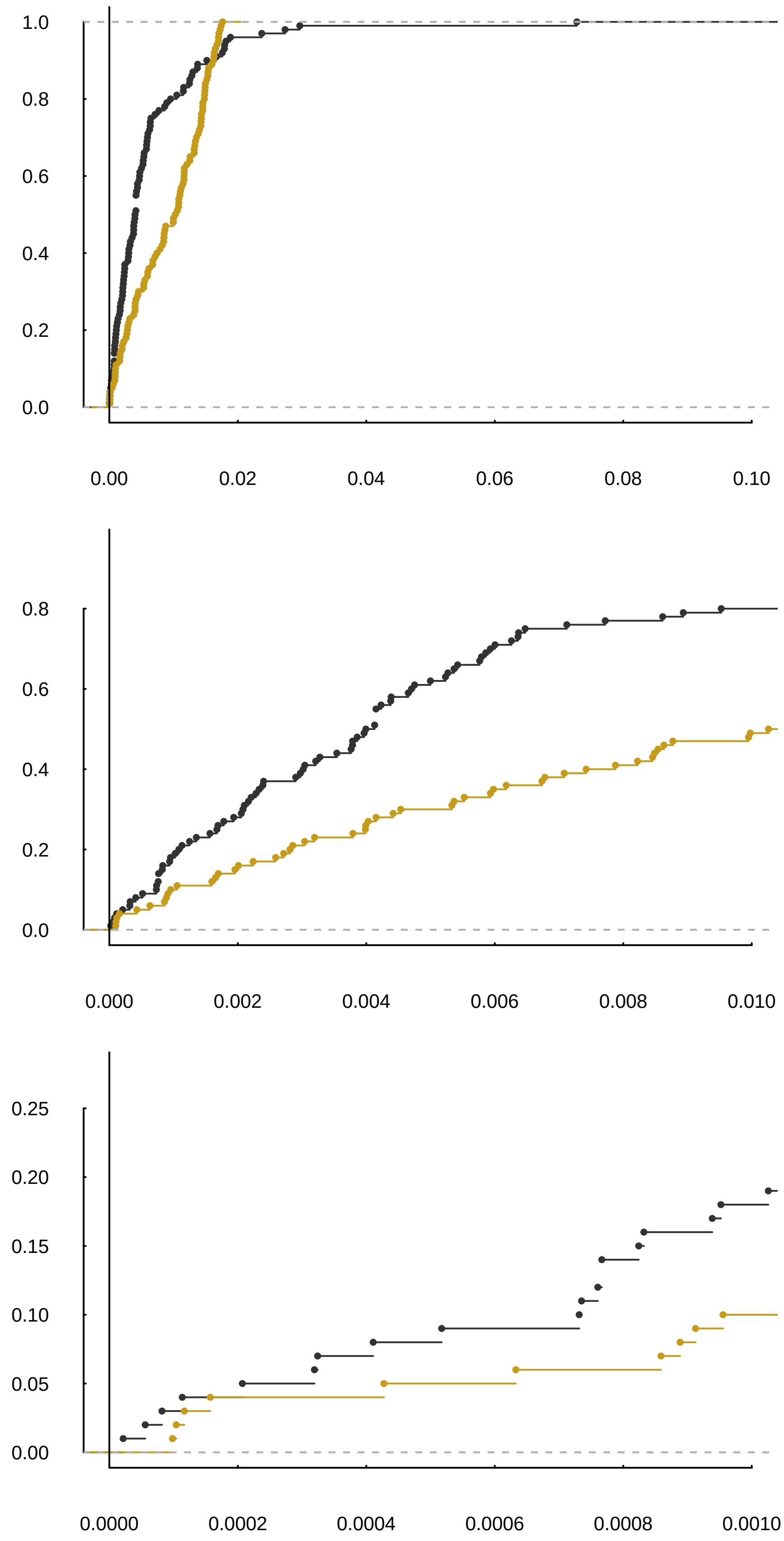
N compute and evaluate all neighbours; return best neighbour

A if *best* neighbour is better, accept it

stop when there is no further improvement

→ depends on starting value

Greedy search



Local Search

1: **while** stopping condition not met **do**
2: create new solution $x^{\mathrm{n}} \in N(x^{\mathrm{c}})$
3: evaluate $f(x^{\mathrm{n}})$
4: **if** $A(x^{\mathrm{n}},\ldots)$ **then** $x^{\mathrm{c}} = x^{\mathrm{n}}$
5: **end while**

N pick one neighbour randomly

A if neighbour is not worse, accept it

stop after a fixed number of iterations

```
LSopt(OF, algo = list(), ...)
```


Local Search

```
> algo <- list(x0 = makeRandomSol(Data),  ## initial solution
               neighbour = neighbour,
               nS = 20000,                ## number of steps
               printBar = FALSE)
> system.time(solLS <- LSopt(OF, algo = algo, Data = Data))
```

```
Local Search.
Initial solution:  16.53
Finished.
Best solution overall: 0.0013902
   user  system elapsed
   0.45    0.00    0.45
```

Threshold Accepting

while stopping condition not met **do**
  create new solution $x^{\mathrm{n}} \in N(x^{\mathrm{c}})$
  evaluate $f(x^{\mathrm{n}})$
  if $A(x^{\mathrm{n}}, \ldots)$ **then** $x^{\mathrm{c}} = x^{\mathrm{n}}$
end while

N pick one neighbour randomly

A if neighbour is *not much worse*, accept it

stop after a fixed number of iterations

not much worse: increase in objective function less than a *threshold*
→ typically, a threshold sequence is used

Threshold Accepting

```
TAopt(OF, algo = list(), ...)
```

```
> algo <- list(x0 = makeRandomSol(Data),  ## initial solution
               neighbour = neighbour,
               nS = 1000,                 ## total iterations:
               nT = 20,                   ##    nS * nT
               printBar = FALSE)
```


Threshold Accepting

```
> system.time(solTA <- TAopt(OF, algo = algo, Data = Data))
```

```
Threshold Accepting.

Computing thresholds ... OK.
Estimated remaining running time: 0.4 secs.


Running Threshold Accepting...
Initial solution:  35.574
Finished.
Best solution overall: 0.000080937
   user  system elapsed
   0.64    0.00    0.64
```

Experiments

Local Search and Threshold Accepting are stochastic

result of optimisation: random variable ϕ with unknown distribution D

(assumption: change seed for each run)

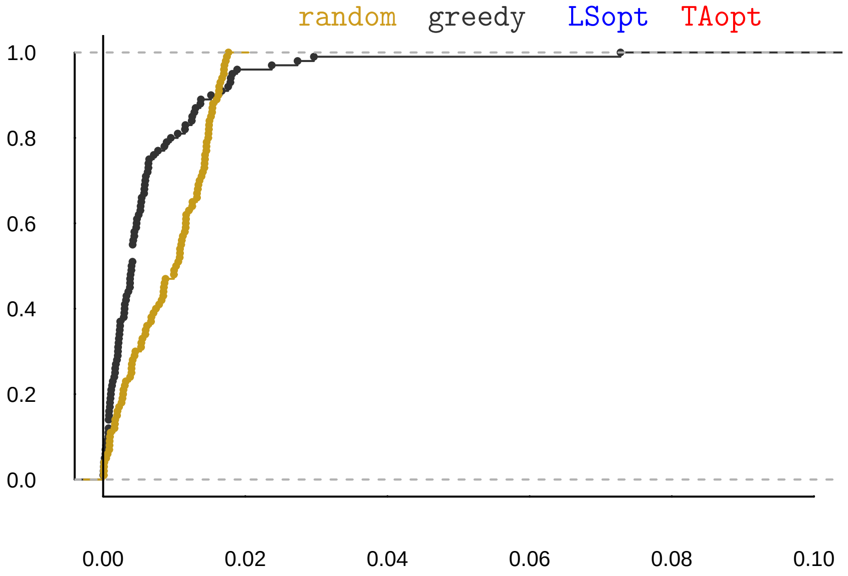
easy to sample from D:
run restarts $i = 1, \dots, n_{\text{restarts}} \rightarrow$ collect ϕ_i

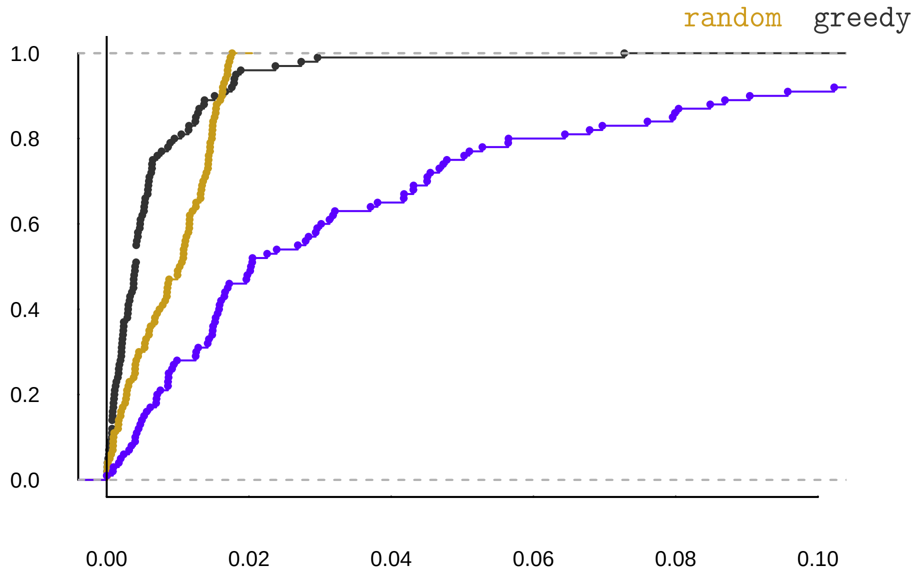
Experiments

```
restartOpt(fun, n, OF, algo, ...,
           method = c("loop", "multicore", "snow"),
           mc.control = list(), cl = NULL)
```

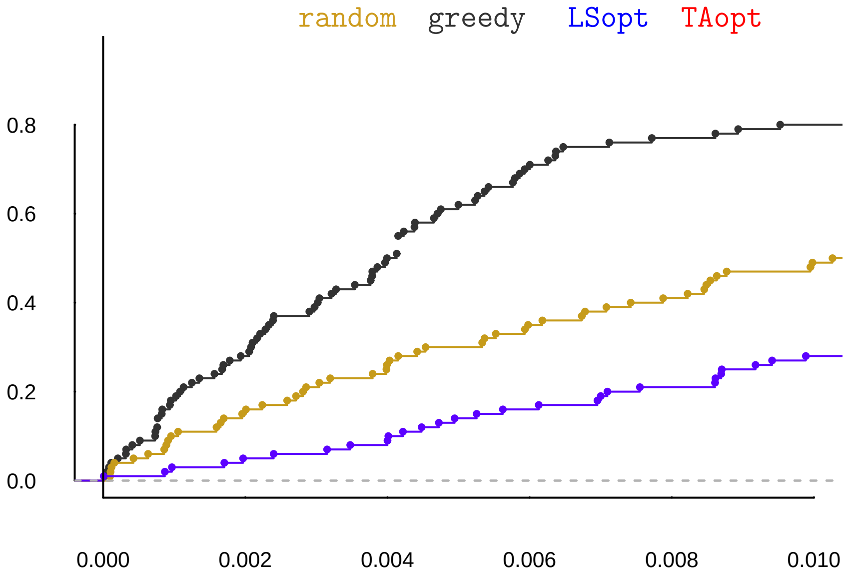
```
> restartOpt(LSopt, n = 100, OF, algo = algo, Data = Data)
> restartOpt(TAopt, n = 100, OF, algo = algo, Data = Data)
```

Experiments





LSopt TAopt



E. Schumann NMOF – 35

Implementation details

representation

objective function: `fun(x)` → *number*

neighbourhood function: `N(x)` → `x.new`

E. Schumann NMOF – 36

Implementation details

solution

- logical vector
- subset sum associated with this vector

iteration 1 $\sum X_{\text{subset1}}$

iteration 2 $\sum X_{\text{subset1}} + \sum (X I_p)$

$$I_p = \begin{cases} = 0 & \text{if not included} \\ = 1 & \text{if added} \\ = -1 & \text{if removed} \end{cases}$$

E. Schumann NMOF – 37

Subset sum with updating

```
> tmp <- makeRandomSol(Data)
> x0 <- list(x  = tmp,
             sx = sum(Data$X[tmp]))
```

```
> OF2 <- function(x, Data)
      abs(x$sx - 2)
> OF2(x0, Data)
```

```
[1] 0.64818
```

```
> OF(tmp, Data)  ## check
```

```
[1] 0.64818
```

E. Schumann NMOF – 38

Subset sum with updating

```
> neighbour2 <- function(x, Data) {
     p <- sample.int(Data$n, size = Data$size)
     x$x[p] <- !x$x[p]
     x$sx <- x$sx + sum(Data$X[p] * ifelse(x$x[p], 1, -1))
     x
 }

     x$sx <- x$sx + sum(Data$X[p] * (x$x[p] * 2 - 1))
```


Subset sum with updating

new data

```
> Data$n <- 50000L
> Data$X <- rnorm(Data$n)
```


Subset sum with updating

```
> set.seed(56447)
> x0 <- makeRandomSol(Data)
> algo <- list(x0 = x0,
               printDetail = FALSE, printBar = FALSE,
               neighbour = neighbour)
> t1 <- system.time(sol1 <- TAopt(OF, algo = algo, Data = Data))

> set.seed(56447)
> tmp <- makeRandomSol(Data)
> x0 <- list(x = tmp, sx = sum(Data$X[tmp]))
> algo <- list(x0 = x0,
              printDetail = FALSE, printBar = FALSE,
              neighbour = neighbour2)
> t2 <- system.time(sol2 <- TAopt(OF2, algo = algo, Data = Data))
```

Subset sum with updating

compare solutions. . .

```
> OF( sol1$xbest, Data)
```

```
[1] 0.000059376
```

```
> OF2(sol2$xbest, Data)
```

```
[1] 0.000059376
```

. . . and speedup

```
> t1[[3L]]/t2[[3L]]
```

```
[1] 12
```


Details

- How to choose the thresholds?
- when to stop?
- constraints?

Constraints

- throw away infeasible solutions
- always construct feasible solutions (example: budget constraint)
- repair solutions
- penalise infeasible solutions

Portfolio optimisation

$$\min_{w} \; \Phi$$

$$\begin{aligned} w'\iota &= 1\,, \\ 0 \leqslant w_j &\leqslant w_j^{\max} \qquad \text{for } j = 1, 2, \ldots, n_A \end{aligned}$$

w weight vector

$w_j^{\max}$ maximum weight 5%

Φ squared portfolio return

s

quared return and variance is similar:

$$\frac{1}{n_S} R'R = \mathrm{Cov}(R) + mm'$$

with m the vector of column means of R

Portfolio optimisation

mean–variance

weights	+	returns	→	portfolio return
w		m		$m'w$
weights	+	covariance matrix	→	portfolio variance
w		Σ		$w'\Sigma w$

scenario optimisation

scenario matrix R (rows: scenarios, columns: assets)

weights	+	scenarios	→	portfolio returns	→	any portfolio statistic
w		R		Rw		$f(Rw)$

Setting up the model

portfolio weights: numeric vector w

objective function: $f(Rw)$

neighbourhood: pick two assets; increase one weight, decrease one weight

1: set ϵ
2: randomly select asset i
3: set $w_i = w_i - \epsilon$
4: randomly select asset i
5: set $w_i = w_i + \epsilon$

→ enforces budget constraint (and possibly `wmin`/`wmax`)

Setting up the model

dataset fundData: 500 weekly return scenarios for 200 funds

```
> Data <- list(R = t(fundData),
               na = dim(fundData)[2L],  ## number of assets
               ns = dim(fundData)[1L],  ## number of scenarios
               eps = 0.5/100,           ## stepsize
               wmin = 0.00,
               wmax = 0.05,
               resample = function(x, ...)
                              x[sample.int(length(x), ...)])
```


Portfolio optimisation

objective function

- compute Rw
- evaluate f(Rw)

```
> OF <- function(w, Data) {
     Rw <- crossprod(Data$R, w)
     crossprod(Rw)
 }
```


Portfolio optimisation

```
> neighbour <- function(w, Data) {
     toSell <- w > Data$wmin
     toBuy  <- w < Data$wmax
     i <- Data$resample(which(toSell), size = 1L)
     j <- Data$resample(which(toBuy),  size = 1L)
     eps <- runif(1L) * Data$eps
     eps <- min(w[i] - Data$wmin, Data$wmax - w[j], eps)
     w[i] <- w[i] - eps
     w[j] <- w[j] + eps
     w
 }
```

Portfolio optimisation

set up and run TAopt

```
> w0 <- runif(Data$na); w0 <- w0/sum(w0)  ## a random solution
> algo <- list(x0 = w0,
               neighbour = neighbour,
               nS = 2000L,
               nT = 10L,
               q = 0.10,
               printBar = FALSE)
```


Portfolio optimisation

```
> res <- TAopt(OF,algo,Data)
```

```
Threshold Accepting.

Computing thresholds ... OK.
Estimated remaining running time: 2.8 secs.


Running Threshold Accepting...
Initial solution:  0.22391
Finished.
Best solution overall: 0.0056666
```

scale solution: divide by ns; take square root; multiply by 100

```
[1] 0.33665
```


Portfolio optimisation

check constraints

```
> min(res$xbest) ## should not be smaller than Data$wmin
```

```
[1] 0
```

```
> max(res$xbest) ## should not be greater than Data$wmax
```

```
[1] 0.05
```

```
> sum(res$xbest) ## should be one
```

```
[1] 1
```

Portfolio optimisation

compare with quadprog

```
OF (scaled) QP: 0.33612
OF (scaled) TA: 0.33665
```

(scaled: divide by ns; take square root; multiply by 100)

Updating

$$
\begin{aligned}
w^{n} &= \quad w^{c} + w^{\Delta}\\
Rw^{n} &= R(w^{c} + w^{\Delta}) = \underbrace{Rw^{c}}_{\text{known}} + Rw^{\Delta}
\end{aligned}
$$

Updating

with updating

```
> OFU <- function(sol, Data)
     crossprod(sol$Rw)
> neighbourU <- function(sol, Data){
     wn <- sol$w
     toSell <- wn > Data$wmin; toBuy  <- wn < Data$wmax
     i <- Data$resample(which(toSell), size = 1L)
     j <- Data$resample(which(toBuy), size = 1L)
     eps <- runif(1) * Data$eps
     eps <- min(wn[i] - Data$wmin, Data$wmax - wn[j], eps)
     wn[i] <- wn[i] - eps; wn[j] <- wn[j] + eps
     Rw <- sol$Rw + Data$R[ ,c(i,j)] %*% c(-eps,eps)
     list(w = wn, Rw = Rw)
 }
```

Updating

```
> w0 <- runif(Data$na); w0 <- w0/sum(w0)  ## a random solution
> Data$R <- fundData
> sol <- list(w = w0, Rw = Data$R %*% w0)
> algo <- list(x0 = sol,
             neighbour = neighbourU,
             nS = 2000L,
             nT = 10L,
             q = 0.10,
             printBar = FALSE,
             printDetail = FALSE)
> res <- TAopt(OFU,algo,Data)
```


Robustness

the weight of asset 200

```
> wqp[200]
```

```
[1] 0.0000000000000001104
```

```
> fundData <- cbind(fundData, fundData[, 200L])
> dim(fundData)
```

```
[1] 500 201
```

```
> qr(fundData)$rank
```

```
[1] 200
```

```
> qr(cov(fundData))$rank
```

```
[1] 200
```


Robustness

```
> cat(try(result.QP <- solve.QP(Dmat = covMatrix,
                                dvec = rep(0, Data$na),
                                Amat = t(rbind(A,B)),
                                bvec = rbind(a,b),
                                meq = 1L)))
```

```
Error in solve.QP(Dmat = covMatrix, dvec = rep(0, Data$na), Amat = t(rbind(A, :
  matrix D in quadratic function is not positive definite!
```

Robustness

```
> res2 <- TAopt(OFU, algo, Data)
```

```
[1] 0.33651
```

weights 200 and 201

```
> res2$xbest$w[200:201]
```

```
[1] 0 0
```


Other objective functions

$$\frac{1}{n_S}\sum_{r_i<\theta}(\theta - r_i)^2$$

```
> OF <- function(w, Data) { ## semi-variance
     Rw <- crossprod(Data$R, w) - Data$theta
     Rw <- Rw - abs(Rw)
     sum(Rw*Rw) / (4 * Data$ns)
 }

> OF <- function(w, Data) { ## Omega
     Rw <- crossprod(Data$R, w) - Data$theta
     -sum(Rw - abs(Rw)) / sum(Rw + abs(Rw))
 }
```

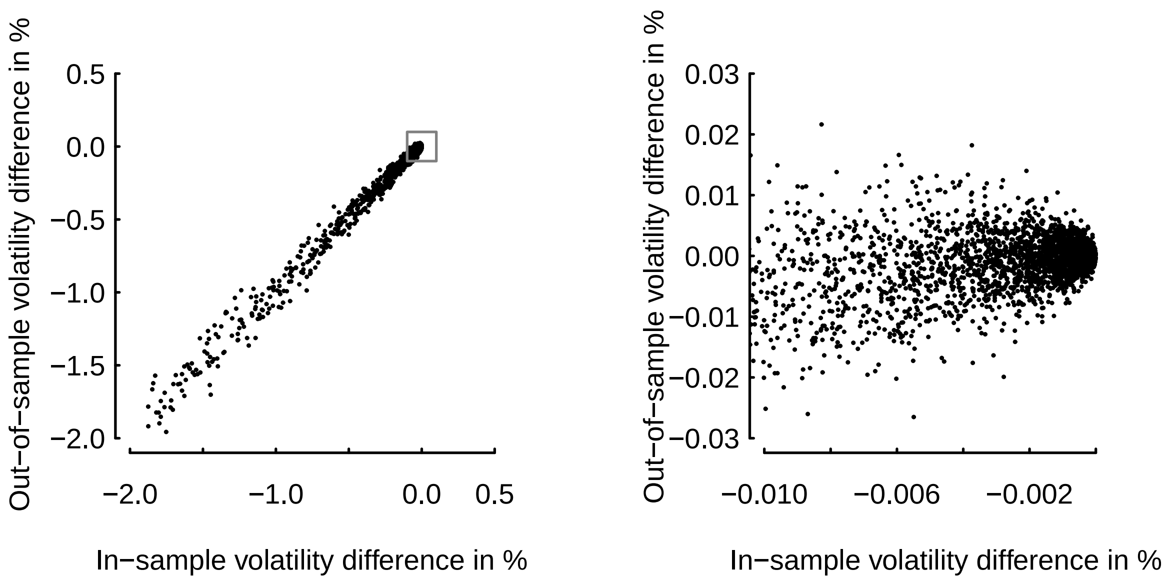
Good enough?

(Gilli and Schumann, 2011; Gilli et al., 2011)

for $i = 10 : 50000$ **do**
    sample 400 scenarios without replacement
    compute optimal portfolio with QP
    set $n_{\text{iterations}} = i$
    compute portfolio with TA, compute in-sample difference between QP/TA
    compute out-of-sample difference for QP and TA on remaining 100 scenarios
end for

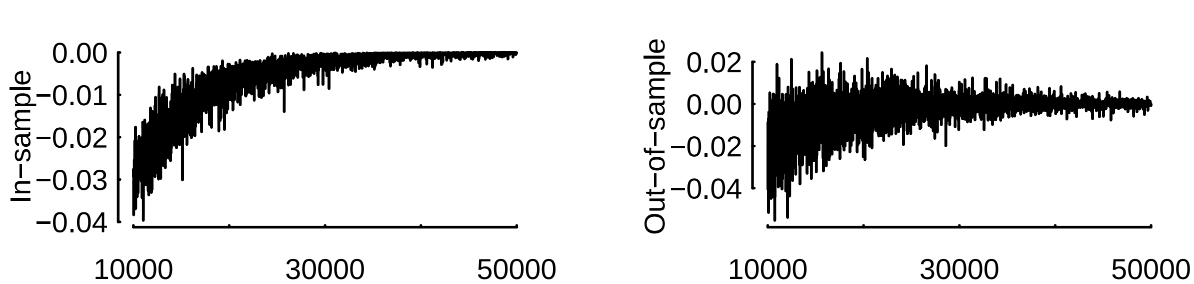
objective function value of QP − objective function value of TA

Good enough?



in-sample versus out-of-sample difference

Good enough?



in-sample versus out-of-sample difference depending on the number of iterations

Conclusion

- 'in principle' *vs* 'details matter':
  compare different methods by implementing them
- parameters (eg, step size in neighbourhood) are determined by application
- required precision is determined by application

More information

the NMOF package is on CRAN/R-Forge

```
> install.packages("NMOF") ## CRAN
> install.packages("NMOF",
                   repos = "http://R-Forge.R-project.org")


> require("NMOF")
> showExamples("tria.R")  ## load code examples from book
```

mailing list: `NMOF-News`
`https://lists.r-forge.r-project.org/cgi-bin/mailman/listinfo/nmof-news`

and also at `gmane.comp.finance.nmof.announce`

references

Manfred Gilli and Enrico Schumann. Optimal enough? *Journal of Heuristics*, 17(4):373–387, 2011. available from `http://dx.doi.org/10.1007/s10732-010-9138-y`.

Manfred Gilli, Dietmar Maringer, and Enrico Schumann. *Numerical Methods and Optimization in Finance*. Academic Press, 2011.

Enrico Schumann and David Ardia. Heuristic methods in finance. *Statistical Computing & Statistical Graphics Newsletter*, 22(1):13–19, 2011.