Optimization

RSMs and Computer Experiments: Part 4

Robert B. Gramacy (rbg@vt.edu (mailto:rbg@vt.edu) : http://bobby.gramacy.com (http://bobby.gramacy.com))
Department of Statistics, Virginia Tech

Goals

To understand the role that GPs can play in optimizing a *blackbox* function,

- i.e., one about which one knows little (it is opaque to the optimizer)
- and which can only be probed through expensive evaluation.

Basically, the idea is to view optimization as an application of sequential design.

The role of "modeling" in optimization has a rich history,

- and we'll barely scratch the surface there.

But the *potential* role of modern *statistical* modeling is just recently being realized by the mathematical programming, statistics, and machine learning communities.

Surrogate-based optimization

An old idea

Statistical methods in optimization, in particular of *noisy* blackbox functions, probably goes back to Box & Draper (https://www.amazon.com/Empirical-Model-Building-Response-Probability-Statistics/dp/0471810339),

- a pre-cursor to "steepest ascent" in classical RSMs.

However, the modern version is closest to methods described by Mockus, et al. (e.g., 1978) (https://www.researchgate.net/publication/248818761_The_application_of_Bayesian_methods_for_seeking_the_extremum), in a paper entitled

- "The application of Bayesian methods for seeking the extremum".

Although it would seem that many of these ideas were overlooked,

- in part because the models involved were too crude (linear)

until the late 1990's, after GPs became established in the computer experiments literature.

Surrogate-based optimization

In the computer experiments literature, folks had been using GPs to optimize functions for some time,

- however, the best reference for the core idea might be Booker, et al. (1999) (http://link.springer.com/article/10.1007/BF01197708).
- They called it *surrogate-based optimization*, and it involved a nice collaboration between optimization and computer modeling researchers.

The methodology is simple:

1. Train a GP on the function evaluations obtained so far.
2. Maximize the fitted predictive mean surface of the GP to choose the next location for evaluation.
3. Repeat.

Observe that Step 2 involves its own inner-optimization,

- but an easy one that can be solved with conventional methods.

The problem

Before we continue, lets be clear about the problem.

We wish to find

$$x^* = \text{argmin}_{x \in \mathcal{B}} f(x).$$

- $\mathcal{B}$ is usually a hyper-rectangle.
- We do not have the derivatives of $f(x)$, nor do we necessarily want them (or want to approximate them).

- The methods we prescribe fall under the class of **derivative-free optimization**, see., e.g., Conn, et al. (2009) (http://epubs.siam.org/doi/book/10.1137/1.9780898718768).

This means that all we get to do is

- evaluate $f(x)$, which for now is *deterministic*,
- and we presume that it is expensive to do so (in terms of computing time, say).
- So a tacit "constraint" on the solver is that it minimize the number of evaluations.

Implementation

Lets consider an implementation of the idea Booker et al. summarized on a re-scaled/coded version of the Goldstein-Price (http://www.sfu.ca/~ssurjano/goldpr.html) function.

```
f <- function(X)
 {
    if(is.null(nrow(X))) X <- matrix(X, nrow=1)
    m <- 8.6928
    s <- 2.4269
    x1 <- 4 * X[,1] - 2
    x2 <- 4 * X[,2] - 2
    a <- 1 + (x1 + x2 + 1)^2 * (19 - 14 * x1 + 3 * x1^2 - 14 *
        x2 + 6 * x1 * x2 + 3 * x2^2)
    b <- 30 + (2 * x1 - 3 * x2)^2 * (18 - 32 * x1 + 12 * x1^2 +
        48 * x2 - 36 * x1 * x2 + 27 * x2^2)
    f <- log(a * b)
    f <- (f - m)/s
    return(f)
 }
```

- We want to minimize this $f(x)$, pretending we can't see how it is comprised.

Initial design

Lets start with a small initial design in the 2d space.

```
library(lhs)
ninit <- 12
X <- randomLHS(ninit, 2)
y <- f(X)
```

Now lets fit a (separable) GP to that data, with a small nugget.

```
library(laGP)
gpi <- newGPsep(X, y, d=0.1, g=1e-6, dK=TRUE)
da <- darg(list(mle=TRUE, max=0.5), X)
mleGPsep(gpi, param="d", tmin=da$min, tmax=da$max, ab=da$ab)$msg
```

```
## [1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

Surrogate-optimize

Just like our ALM/C searches, lets set up an objective that we can search based on the predictive distribution.

```
obj.mean <- function(x, gpi) predGPsep(gpi, matrix(x, nrow=1), lite=TRUE)$mean
```

Now the predictive surface (like the function f) may have many local minima,

- but lets punt on the idea of global optimization for the moment,
- and see where we get with a search initialized at the current best value.

```
m <- which.min(y)
opt <- optim(X[m,], obj.mean, lower=0, upper=1, method="L-BFGS-B", gpi=gpi)
opt$par
```

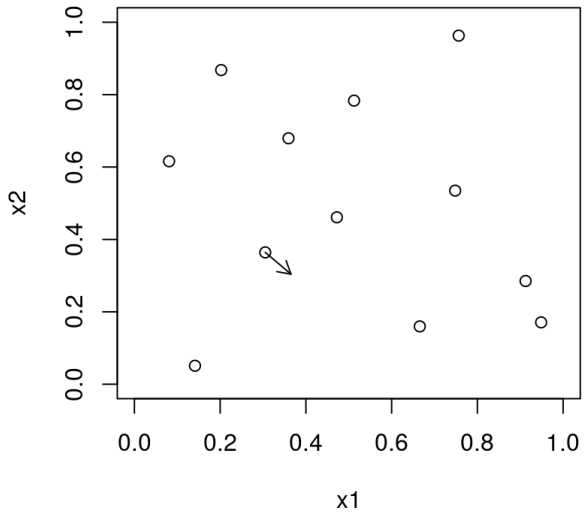
```
## [1] 0.3651836 0.3039873
```

- So this is the next point to try.

One-step of search

Here's what that looks like in the input domain.

```
plot(X[1:ninit,], xlab="x1", ylab="x2", xlim=c(0,1), ylim=c(0,1))
arrows(X[m,1], X[m,2], opt$par[1], opt$par[2], length=0.1)
```



Another iteration

Evaluate f at `opt$par` , update the GP …

```
ynew <- f(opt$par)
updateGPsep(gpi, matrix(opt$par, nrow=1), ynew)
mle <- mleGPsep(gpi, param="d", tmin=da$min, tmax=da$max, ab=da$ab)
X <- rbind(X, opt$par)
y <- c(y, ynew)
```

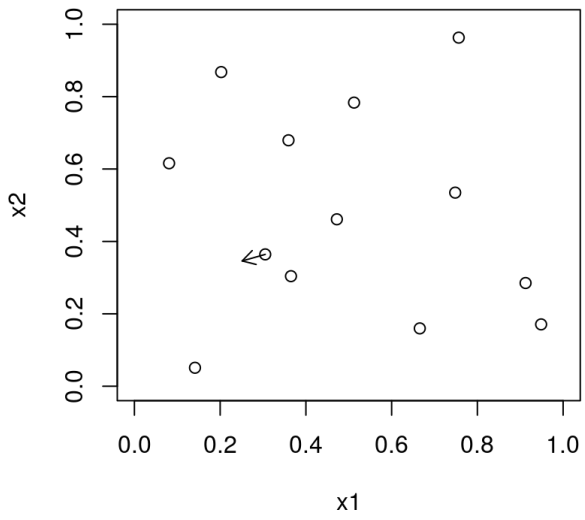
… and solve for the next point.

```
m <- which.min(y)
opt <- optim(X[m,], obj.mean, lower=0, upper=1,  method="L-BFGS-B", gpi=gpi)
opt$par
```

```
## [1] 0.2513159 0.3460500
```

Here's what that looks like in the input domain.

```
plot(X, xlab="x1", ylab="x2", xlim=c(0,1), ylim=c(0,1))
n <- nrow(X)
arrows(X[m,1], X[m,2], opt$par[1], opt$par[2], length=0.1)
```



Five more iterations

Update for the most recent point.

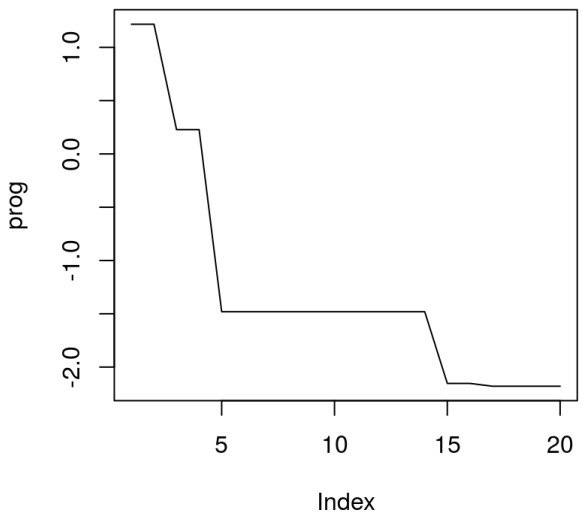
```
ynew <- f(opt$par)
updateGPsep(gpi, matrix(opt$par, nrow=1), ynew)
mle <- mleGPsep(gpi, param="d", tmin=da$min, tmax=da$max, ab=da$ab)
X <- rbind(X, opt$par)
y <- c(y, ynew)
```

Looping: five more iterations.

```
while(1) {
    m <- which.min(y)
    opt <- optim(X[m,], obj.mean, lower=0, upper=1,
        method="L-BFGS-B", gpi=gpi)
    ynew <- f(opt$par)
    if(abs(ynew - y[length(y)]) < 1e-4) break;
    updateGPsep(gpi, matrix(opt$par, nrow=1), ynew)
    mle <- mleGPsep(gpi, param="d", tmin=da$min, tmax=da$max, ab=da$ab)
    X <- rbind(X, opt$par)
    y <- c(y, ynew)
}
```

Progress

```
prog <- y
for(i in 2:length(y))
    if(prog[i] > prog[i-1]) prog[i] <- prog[i-1]
plot(prog, type="l")
```



Encapsulating function

```
optim.surr <- function(f, ninit, stop, tol=1e-4)
 {
    X <- randomLHS(ninit, 2)
    y <- f(X)
    gpi <- newGPsep(X, y, d=0.1, g=1e-7, dK=TRUE)
    da <- darg(list(mle=TRUE, max=0.5), randomLHS(1000, 2))
    mleGPsep(gpi, param="d", tmin=da$min, tmax=da$max, ab=da$ab)
    for(i in (ninit+1):stop) {
        m <- which.min(y)
        opt <- optim(X[m,], obj.mean, lower=0, upper=1,
            method="L-BFGS-B", gpi=gpi)
        ynew <- f(opt$par)
        if(abs(ynew - y[length(y)]) < tol) break;
        updateGPsep(gpi, matrix(opt$par, nrow=1), ynew)
        mleGPsep(gpi, param="d", tmin=da$min, tmax=da$max, ab=da$ab)
        X <- rbind(X, opt$par)
        y <- c(y, ynew)
    }
    deleteGPsep(gpi)

    return(list(X=X, y=y))
 }
```

Average progress under random init

Lets repeatedly solve the problem in this way with 100 random initializations.

```
reps <- 100
prog <- matrix(NA, nrow=reps, ncol=50)
for(r in 1:reps) {
    os <- optim.surr(f, 12, 50)
    prog[r,1:length(os$y)] <- os$y
    for(i in 2:50) {
        if(is.na(prog[r,i]) || prog[r,i] > prog[r,i-1])
            prog[r,i] <- prog[r,i-1]
    }
}
```

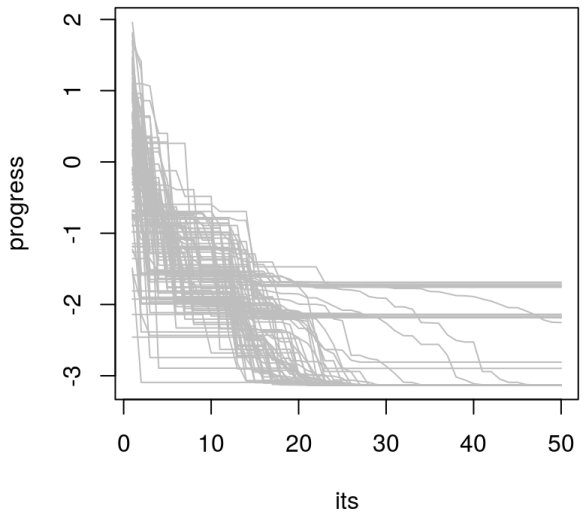
Note that these are random initializations.

- not random searches.
- These searches are completely deterministic.
    - (I.e., this is not *stochastic optimization*.)

Visualizing average progress

Clearly this is not a global optimization tool.

```
matplot(t(prog), type="l", col="gray", lty=1, ylab="progress", xlab="its")
```



How does `optim` compare?

First, we need to modify f so we can keep track of the path of evaluations.

- because `optim` will call f to approximate gradients.

```
fprime <- function(x)
 {
    ynew <- f(x)
    y <<- c(y, ynew)
    return(ynew)
 }
```

Now lets loop a bunch of times.

- Note that `optim` will only allow us to control the “outer” iterations.

Average `optim` progress

Here is the same `for` loop we did with the surrogate-based optimizer,

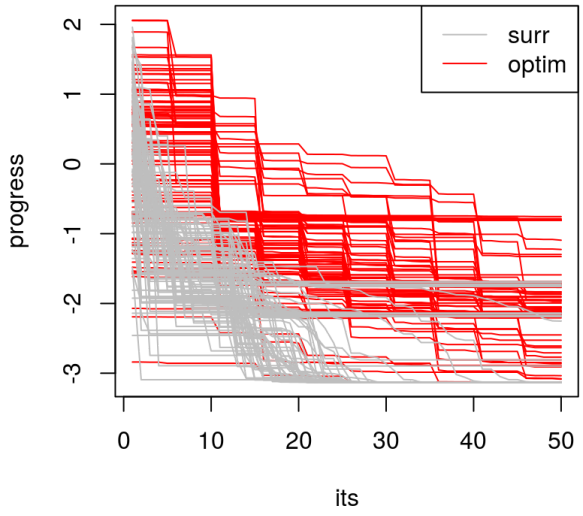
- with a direct `optim` instead,
- randomly initialized.

```
prog.optim <- matrix(NA, nrow=reps, ncol=50)
for(r in 1:reps) {
    y <- c()
    os <- optim(runif(2), fprime, lower=0, upper=1, method="L-BFGS-B")
    prog.optim[r,1:length(y)] <- y <- y[1:min(50, length(y))]
    for(i in 2:length(y)) {
        if(is.na(prog.optim[r,i]) || prog.optim[r,i] > prog.optim[r,i-1])
            prog.optim[r,i] <- prog.optim[r,i-1]
    }
}
```

Clear winner

`optim` progress is much slower on a fixed budget.

```
matplot(t(prog.optim), type="l", col="red", lty=1, ylab="progress", xlab="its")
matlines(t(prog), type="l", col="gray", lty=1)
legend("topright", c("surr", "optim"), col=c("gray", "red"), lty=1)
```



Why do “we” do well?

Our surrogate-based optimization is more efficient because

- it does not need to evaluate the expensive blackbox function, f, to approximate derivatives.
- Rather, the surrogate is providing a sense of derivative for “free”.
- It also can (potentially) take big steps because its knowledge of the response surface is more global than `optim` 's.
    - `optim` only bases evaluations on a local linear approximation.

How can “we” do better?

But surrogate optimization is still mostly a local affair.

- It *exploits*, by moving to the next best spot from where it left off,
- descending with its own `optim` subroutine on the predictive surface.
- It does not *explore* places that cannot easily be reached from the current best value.

We need some way to balance exploration and exploitation.

- BTW, notice that we’re not actually doing statistics,
- because at no point is uncertainty being taken into account.

Expected improvement

Striking a balance

In the mid 90s, Matthias Schonlau was working on his dissertation (http://www.schonlau.net/publication/thesis1side.pdf),

- which basically revisited Mockus’ Bayesian optimization idea from a Gaussian process and computer experiments perspective.

He came up with a heuristic called **expected improvement (EI)**, which is the basis of a so-called

- **efficient global optimization (EGO)** algorithm.

His key insight was that predictive uncertainty was underutilized in the surrogate framework for optimization,

- which is especially a shame when GPs are involved, because they provide such a beautiful predictive variance function.
- The basic idea, however, is not limited to GP surrogates.

Improvement

Schonlau defined a statistic called the **improvement**

$$I(x) = \max\{0, f_n^{\min} - Y(x)\}$$

which is a *random variable* measuring the amount by which an unknown response $Y(x)$ is below the current point known to be the minimum

$$f_n^{\min} = \min\{y_1, \ldots, y_n\}.$$

- That is, measuring potential of $Y(x)$ to "improve" upon the current best minimum.
- If $Y(x)$ has non-zero probability of taking on any value on the real line, then $I(x)$ has nonzero probability of being positive.

Expected improvement

Now there are lots of things you could imagine doing with the improvement,

- but probably the most important thing to do (to make it useful) is to remove the randomness.
- And the simplest way to do that is to take an expectation.

It is easiest to imagine what the **expected improvement** might look like through a Monte Carlo approximation.

- Draw $y^{(t)} \sim \mathcal{N}(\mu(x), \sigma^2(x))$ in the case of Gaussian $Y(x)$, $t = 1, \ldots, T$.
- And average:

$$\frac{1}{T}\sum_{t=0}^{T} \max\{0, f_{\min}^n - y^{(t)}\} \to \mathbb{E}\{I(x)\} \qquad \text{as } T \to \infty.$$

- This works no matter what the distribution of $Y(x)$ is (so long as you can simulate from it).

Analytic expression

The cool thing is that if $Y(x)$ is Gaussian,

- as it is under the predictive equations of a GP,

the EI has a convenient closed form expression.

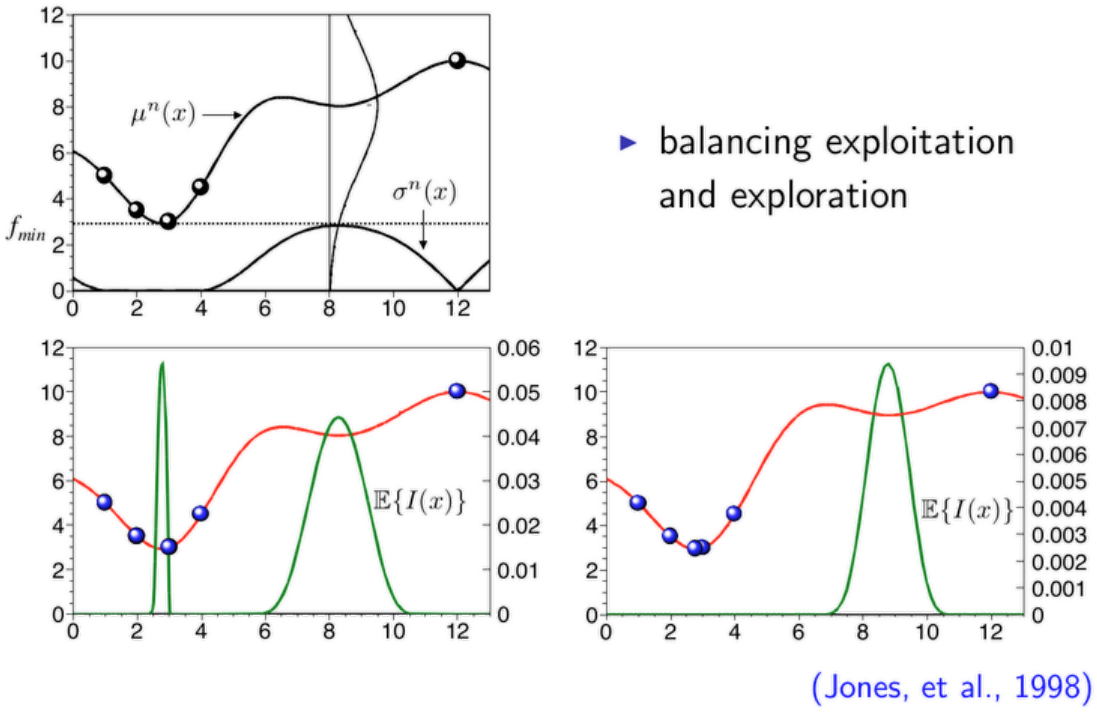
$$\mathbb{E}\{I(x)\} = (f_{\min}^n - \mu_n(x))\,\Phi\left(\frac{f_{\min}^n - \mu_n(x)}{\sigma_n(x)}\right) + \sigma_n(x)\,\phi\left(\frac{f_{\min}^n - \mu_n(x)}{\sigma_n(x)}\right)$$

- Φ and ϕ are the standard normal cdf and pdf, respectively.

Notice how it organically balances

- exploitation: $\mu_n(x)$ below $f_{\min}^n$, and
- exploration: large $\sigma_n(x)$.

– A useful cartoon –



- balancing exploitation and exploration

(Jones, et al., 1998)

Interactive EI demo

See `gp_ei_sin.R` with the course materials.

This code uses a hodge-podge of libraries, and I didn't want to re-write it.

- We'll code our own stuff up in a sec.

Our own EI calculation

The `laGP` package doesn't include an EI calculation,

- but it is easy to use the output of the predict functions to calculate EI.

```
EI <- function(gpi, x, fmin, pred=predGPsep)
 {
    if(is.null(nrow(x))) x <- matrix(x, nrow=1)
    p <- pred(gpi, x, lite=TRUE)
    d <- fmin - p$mean
    sigma <- sqrt(p$s2)
    dn <- d/sigma
    ei <- d*pnorm(dn) + sigma*dnorm(dn)
    return(ei)
 }
```

To use it as an objective in a surrogate-based optimization:

```
obj.EI <- function(x, fmin, gpi) - EI(gpi, x, fmin)
```

Multi-start scheme

Although EI has a "maximizing variance" aspect, which could cause the EI surface to be multi-modal

- just like ALM and ALC,
- it will not be as pathologically so.

The number of EI modes will fluctuate as the algorithm runs,

- but eventually it will resemble the actual (number of) modes in f.

Therefore a sensible multi-start scheme might include

- the best point you have so far ($f^n_{\min}$)
- and a few other points spread around the input space.

A search scheme

Copying from `alm.search`, how about the following?

```
EI.search <- function(X, y, gpi, multi.start=5)
 {
    m <- which.min(y)
    fmin <- y[m]
    start <- matrix(X[m,], nrow=1)
    if(multi.start > 1)
        start <- rbind(start, randomLHS(multi.start-1, ncol(X)))
    xnew <- matrix(NA, nrow=nrow(start), ncol=ncol(X)+1)
    for(i in 1:nrow(start)) {
        if(EI(gpi, start[i,], fmin) <= eps)
            { out <- list(value=-Inf); next }
        out <- optim(start[i,], obj.EI, method="L-BFGS-B",
            lower=0, upper=1, gpi=gpi, fmin=fmin)
        xnew[i,] <- c(out$par, -out$value)
    }
    solns <- data.frame(cbind(start, xnew))
    names(solns) <- c("s1", "s2", "x1", "x2", "val")
    solns <- solns[(solns$val > sqrt(.Machine$double.eps)),]
    return(solns)
}
```

Initializing an EI search

Initializing the GP fit.

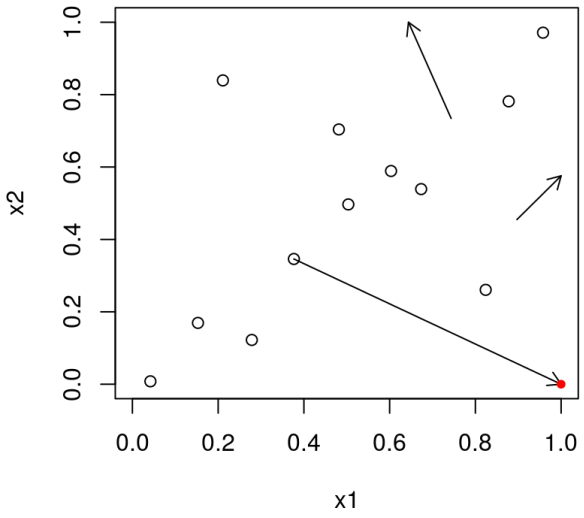
```
X <- randomLHS(ninit, 2)
y <- f(X)
gpi <- newGPsep(X, y, d=0.1, g=1e-6, dK=TRUE)
da <- darg(list(mle=TRUE, max=0.5), X)
```

Performing an EI search.

```
solns <- EI.search(X, y, gpi)
m <- which.max(solns$val)
maxei <- solns$val[m]
```

First iteration visualized

```
plot(X, xlab="x1", ylab="x2", xlim=c(0,1), ylim=c(0,1))
arrows(solns$s1, solns$s2, solns$x1, solns$x2, length=0.1)
points(solns$x1[m], solns$x2[m], col=2, pch=20)
```



Next iteration

Incorporate the new data at the chosen input location.

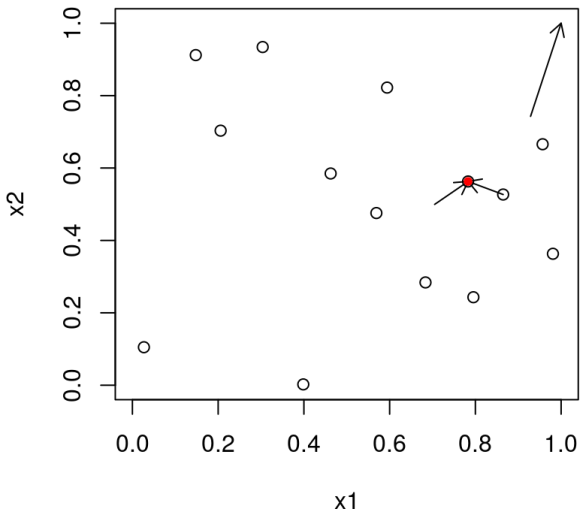
```
xnew <- as.matrix(solns[m,3:4])
X <- rbind(X, xnew)
y <- c(y, f(xnew))
updateGPsep(gpi, xnew, y[length(y)])
mle <- mleGPsep(gpi, param="d", tmin=da$min, tmax=da$max, ab=da$ab)
```

And do another iteration, and update.

```
solns <- EI.search(X, y, gpi)
m <- which.max(solns$val)
maxei <- c(maxei, solns$val[m])
xnew <- as.matrix(solns[m,3:4])
X <- rbind(X, xnew)
y <- c(y, f(xnew))
updateGPsep(gpi, xnew, y[length(y)])
mle <- mleGPsep(gpi, param="d", tmin=da$min, tmax=da$max, ab=da$ab)
```

Clearly a multi-modal criteria.

```
plot(X, xlab="x1", ylab="x2", xlim=c(0,1), ylim=c(0,1))
arrows(solns$s1, solns$s2, solns$x1, solns$x2, length=0.1)
points(solns$x1[m], solns$x2[m], col=2, pch=20)
```



More iterations

Careful, similar y-values is no longer a good measure of convergence.

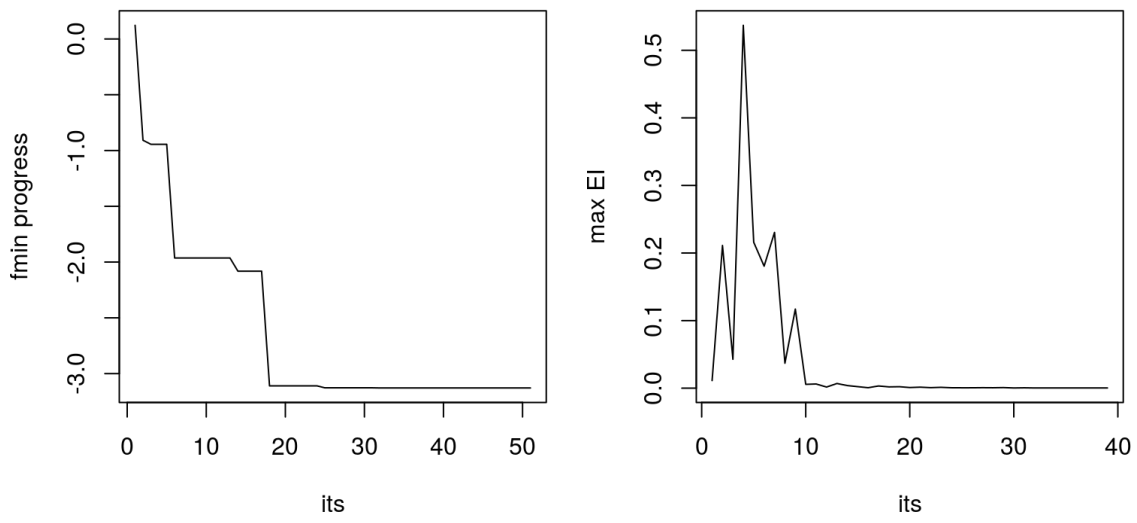
```
for(i in nrow(X):50) {
    solns <- EI.search(X, y, gpi)
    m <- which.max(solns$val)
    maxei <- c(maxei, solns$val[m])
    xnew <- as.matrix(solns[m,3:4])
    ynew <- f(xnew)
    X <- rbind(X, xnew); y <- c(y, ynew)
    updateGPsep(gpi, xnew, y[length(y)])
    mle <- mleGPsep(gpi, param="d", tmin=da$min, tmax=da$max, ab=da$ab)
}
```

Calculating progress …

```
prog.ei <- y
for(i in 2:length(y))
    if(prog.ei[i] > prog.ei[i-1]) prog.ei[i] <- prog.ei[i-1]
```

Two measures of progress

```
par(mfrow=c(1,2))
plot(prog.ei, type="l", ylab="fmin progress", xlab="its")
plot(maxei, type="l", xlab="its", ylab="max EI")
```



Encapsulating function

```
optim.EI <- function(f, ninit, stop)
 {
    X <- randomLHS(ninit, 2); y <- f(X)
    gpi <- newGPsep(X, y, d=0.1, g=1e-7, dK=TRUE)
    da <- darg(list(mle=TRUE, min=eps, max=0.5), X)
    mleGPsep(gpi, param="d", tmin=da$min, tmax=da$max, ab=da$ab)
    maxei <- c()
    for(i in (ninit+1):stop) {
        solns <- EI.search(X, y, gpi)
        m <- which.max(solns$val)
        maxei <- c(maxei, solns$val[m])
        xnew <- as.matrix(solns[m,3:4])
        ynew <- f(xnew)
        updateGPsep(gpi, matrix(xnew, nrow=1), ynew)
        mleGPsep(gpi, param="d", tmin=da$min, tmax=da$max, ab=da$ab)
        X <- rbind(X, xnew); y <- c(y, ynew)
    }
    deleteGPsep(gpi)

    return(list(X=X, y=y, maxei=maxei))
 }
```

Average progress under random init

Lets repeatedly solve the problem in this way with 100 random initializations.

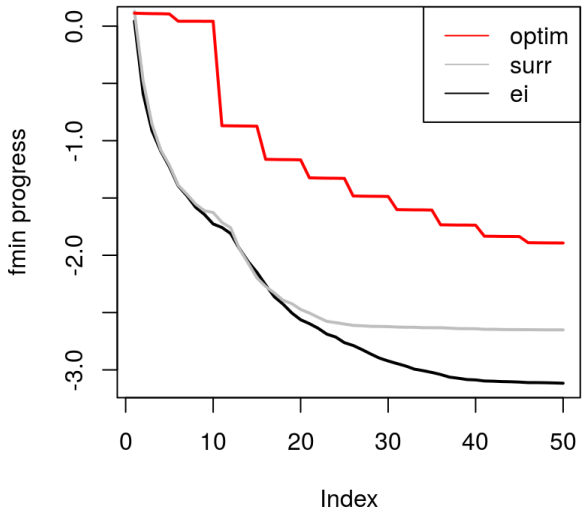
```
reps <- 100
prog.ei <- matrix(NA, nrow=reps, ncol=50)
for(r in 1:reps) {
    os <- optim.EI(f, 12, 50)
    prog.ei[r,1:length(os$y)] <- os$y
    for(i in 2:length(os$y)) {
        if(is.na(prog.ei[r,i]) || prog.ei[r,i] > prog.ei[r,i-1])
            prog.ei[r,i] <- prog.ei[r,i-1]
    }
}
```

The next slide shows the average progress of our three methods so far.

- Its getting to messy to look at all the paths.

Average progress

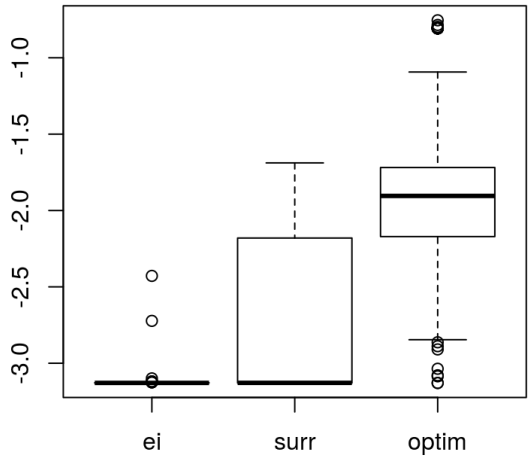
```
plot(colMeans(prog.ei), col=1, lwd=2, ylab="fmin progress", type="l")
lines(colMeans(prog), col="gray",lwd=2)
lines(colMeans(prog.optim, na.rm=TRUE), col=2,lwd=2)
legend("topright", c("optim", "surr", "ei"), col=c(2, "gray", 1), lty=1)
```



Final solution

Once or twice out of 100 repeats did EI not find the global min after 50 iterations.

```
boxplot(prog.ei[,50], prog[,50], prog.optim[,50], names=c("ei", "surr", "optim"))
```



What can you say about it?

Under certain regularity conditions,

- (and when are those really in touch with reality?)
- like that the hyperparameters are known,

the EGO algorithm (i.e., EI searches) will converge to a global optima.

- But only really in the sense that “eventually it will explore everywhere”.

In practice, it does really well

- but there are pathologies;
- just like ALM it can make self-reinforcing decisions, which can be mitigated with strong-ish priors.

You can show that each sequential EI-based decision is optimal

- for the situation where that sample is the last one you'll ever take.

Conditional improvement

That's suggestive of scope for further improvement with additional "look-ahead",

- like ALC does.

Gramacy & Lee (2010) (https://arxiv.org/pdf/1004.4027v2.pdf) describe something called the **integrated expected conditional improvement (IECI)**.

- It was designed for optimization under constraints, which we'll get to in a sec.

The **conditional improvement** involves measuring improvement at one location, a reference location x, *after* another location x_{n+1} is added into the design.

$$I(x \mid x_{n+1}) = \max\{0, f^n_{\min} - Y(x \mid x_{n+1})\}$$

We "deduce" the moments distribution of $Y(x \mid x_{n+1})$ as follows:

- $\mathbb{E}\{Y(x \mid x_{n+1})\} = \mu_n(x)$ since y_{n+1} has not come yet.
- $\mathbb{V}\text{ar}[Y(x \mid x_{n+1})] = \sigma^2_{n+1}(x)$, calculated via partition inverse equations.

IECI

Taking the expectation of $I(x \mid x_{n+1})$ yields the **ECI**

- the analogue of EI for the conditional improvement.

$$\mathbb{E}\{I(x \mid x_{n+1})\} = (f^n_{\min} - \mu_n(x))\,\Phi\left(\frac{f^n_{\min} - \mu_n(x)}{\sigma_{n+1}(x)}\right) + \sigma_{n+1}(x)\,\phi\left(\frac{f^n_{\min} - \mu_n(x)}{\sigma_{n+1}(x)}\right)$$

Then, to obtain a function of x_{n+1} only, i.e., a criteria for sequential design, we integrate over the reference point x

$$\text{IECI}(x_{n+1}) = -\int_{x \in \mathcal{B}} \mathbb{E}\{I(x \mid x_{n+1})\} w(x)\, dx$$

- yielding the **IECI**, for some (possibly uniform) weights $w(x)$.
- In practice we approximate with a sum.

Noisy side-bar

For technical reasons, it is helpful to re-define $f_{\min}$,

- to be your *estimated* rather than *observed* minimum.

This is a standard swap in the literature

- which is usually made in order to handle noisy objective functions.
- In IECI it is helpful whether there is noise or not.

When there is noise,

$$f^n_{\min} = \min_{x \in \mathcal{B}} \mu_n(x) \approx \min_{x \in X_n} \mu_n(x).$$

- Specifically, we'll use $f^n_{\min} = \min_{x \in \mathcal{B}} \mu_n(x)$ going forward.

IECI and noise

Lets illustrate both (IECI and noisy evaluations) at the same time.

Consider the following data in 1d, to ease visualization.

```
f1d <- function(x){ return( sin(x) - 2.55*dnorm(x,1.6,0.45) ) }
X <- matrix(c(seq(0, 4.4, length=15), seq(5.3, 7, length=5)), ncol=1)
y <- f1d(X) + rnorm(length(X), sd=0.15)
```

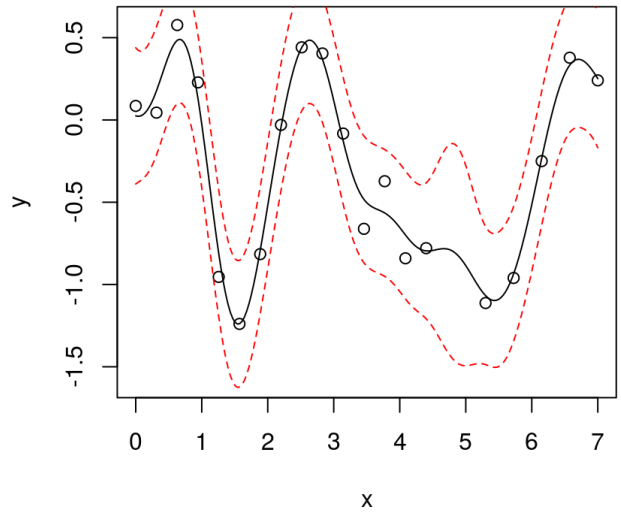
- The design deliberately omits one of the two local minima for illustrative purposes.

The initial fit:

```
gpi <- newGP(X, y, d=0.1, g=0.1*var(y), dK=TRUE)
mle <- jmleGP(gpi)
```

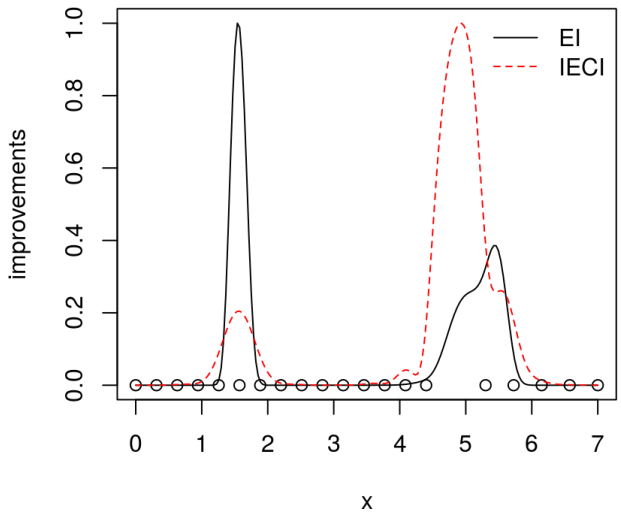
Predictive surface

```
XX <- matrix(seq(0, 7, length=201), ncol=1)
plot(X, y, xlab="x", ylab="y", ylim=c(-1.6,0.6))
p <- predGP(gpi, XX, lite=TRUE); lines(XX, p$mean)
lines(XX, p$mean + 1.96*sqrt(p$s2), col=2, lty=2)
lines(XX, p$mean - 1.96*sqrt(p$s2), col=2, lty=2)
```



Comparing EI and IECI.

```
fmin <- min(predGP(gpi, X, lite=TRUE)$mean)
ei <- EI(gpi, XX, fmin, pred=predGP); ei <- ei - min(ei); ei <- ei/max(ei)
ieci <- ieciGP(gpi, XX, fmin); ieci <- ieci - min(ieci); ieci <- ieci/max(ieci)
plot(XX, ei, type="l", ylim=c(0, max(ei)), xlab="x", ylab="improvements")
points(X, rep(0, nrow(X))); lines(XX, 1-ieci, col=2, lty=2)
legend("topright", c("EI", "IECI"), lty=1:2, col=1:2, bty="n")
```



EI v IECI debrief

IECI is a more aggregate statistic

- pooling together large EI within a wider domain of attraction,
- valuing new design locations by their potential to offer improvement globally in the input space.

Both EI and IECI cope with noise just fine.

I said that IECI was designed for constrained optimization …

Blackbox constraints

Known constraints

First, lets keep it simple and assume the constraints are known.

- That means there is a function $c(x)$ returning
  - zero (or a negative number) if the constraint is satisfied,
  - one (or a positive number) if the constraint is violated
- and we can evaluate it willy-nilly (as much as we want).

The problem is

$$x^* = \text{argmin}_{x \in \mathcal{B}} f(x) \quad \text{subject to} \quad c(x) \leq 0.$$

One simple method is to extend EI to what is called **expected feasible improvement (EFI)** (Schonlau, Jones & Welch, 1998) (https://www.jstor.org/stable/4356058?seq=1#page_scan_tab_contents)

$$\text{EFI}(x) = \mathbb{E}\{I(x)\}\mathbb{I}(c(x) \leq 0),$$

- precluding choosing any point outside the valid set.

IECI adaptation

The IECI adaptation involves deploying the indicator $\mathbb{I}(c(x) \leq 0)$ as a weight $w(x)$:

$$\text{IECI}(x_{n+1}) = -\int_{x \in \mathcal{B}} \mathbb{E}\{I(x \mid x_{n+1})\}\mathbb{I}(c(x) \leq 0)\, dx$$

This down-weights *reference* x values, and thus also x_{n+1} values,

- however it doesn't preclude x_{n+1} values from being chosen in the the invalid region.
- Rather, the value of x_{n+1} is judged by its ability to impact improvement within the valid region.

An alternative implementation, especially when approximating the integral with a sum over `Xref` locations

- is to exclude from `Xref` any input locations which don't satisfy the constraint.

Known constraint illustration

Lets look at the same 1d data as above, except with the invalid region $[2, 4]$.

```
X <- matrix(c(0.01, 0.25, 0.5, 0.75, 1, 1.15, 1.3, 1.5, 5, 5.25, 5.5,
    5.75, 6, 6.25, 6.5, 6.75, 6.99), ncol=1)
y <- f1d(X) + rnorm(length(X), sd=0.15)
lc <- 2; rc <- 4
Xref <- matrix(c(XX[XX < lc,], XX[XX > rc,]), ncol=1)
```

- Notice that the reference locations encode the constraint.

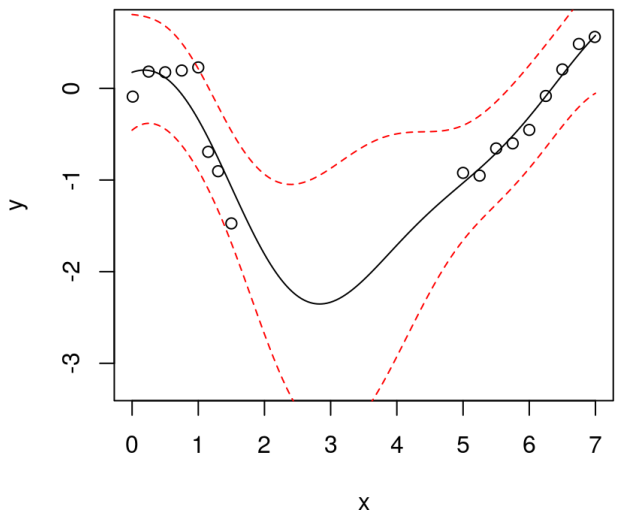
Lets fit the data, but for the sake of this illustration it makes sense to hard-code a longer lengthscale.

```
gpi <- newGP(X, y, d=5, g=0.1*var(y), dK=TRUE)
```

- I suggest tinkering with this (also with `jmleGP`) and see how it effects the results.

Predictive surface

```
XX <- matrix(seq(0, 7, length=201), ncol=1)
plot(X, y, xlab="x", ylab="y", ylim=c(-3.25, 0.7))
p <- predGP(gpi, XX, lite=TRUE); lines(XX, p$mean)
lines(XX, p$mean + 1.96*sqrt(p$s2), col=2, lty=2)
lines(XX, p$mean - 1.96*sqrt(p$s2), col=2, lty=2)
```



Calculating EI and IECI

Calculating EI for all `XX` locations.

```
fmin <- min(predGP(gpi, X, lite=TRUE)$mean)
ei <- EI(gpi, XX, fmin, pred=predGP)
ei <- ei - min(ei); ei <- ei/max(ei)
eiref <- c(ei[XX<lc], ei[XX>rc])
```

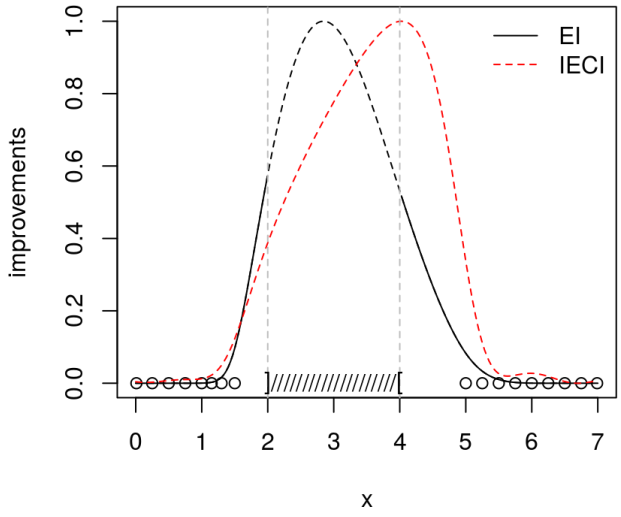
- And extracting those which are in the valid region.

Calculating IECI for all `XX` locations, using `Xref` in lieu of weights.

```
ieci.orig <- ieci <- ieciGP(gpi, XX, fmin, Xref=Xref)
ieci <- ieci - min(ieci); ieci <- ieci/max(ieci)
```

Comparing EI and IECI for a known constraints setup.

```
plot(XX, ei, type="l", ylim=c(0, max(ei)), lty=2, xlab="x", ylab="improvements")
lines(Xref[Xref<lc], eiref[Xref<lc]); lines(Xref[Xref>rc], eiref[Xref>rc])
points(X, rep(0, nrow(X))); lines(XX, 1-ieci, col=2, lty=2)
legend("topright", c("EI", "IECI"), lty=1:2, col=1:2, bty="n")
abline(v=c(lc,rc), col="gray", lty=2); text(lc,0,"]"); text(rc,0,"[")
text(seq(lc+.1, rc-.1, length=20), rep(0, 20), rep("/", 20))
```



Known constraints EI v IECI debrief

I have to admit that, in sequential application, the two methods perform very similarly.

- As do ALM and ALC.
- IECI makes more sense to me but is computationally more demanding (as does/is ALC).
- It depends on how complicated the constraint region is.

But of course, the known constraint setup is very specialized.

- How do we generalize to a unknown (blackbox) constraint setup?

Blackbox constraints

The problem is unchanged

$$x^* = \text{argmin}_{x \in \mathcal{B}} f(x) \quad \text{subject to} \quad c(x) \leq 0,$$

- but now we can't evaluate the constraint function $c(x)$ willy-nilly.

We'll need a model for $c(x)$, and the appropriate model will depend on the nature of the function:

- $c(x) \in \{0, 1\}$ or $c(x) \in \{0, 1\}^m$ for multiple constraints,
  - appropriate for a classification model;
- $c(x) \in \mathbb{R}$ or $c(x) \in \mathbb{R}^m$,
  - appropriate for a regression model.

And we can get as fancy or simple as we want with those models and fitting schemes.

EFI and IECI for blackbox constraints

For now lets generically write $p_n^{(j)}(x)$ for the *predicted* probability that input x satisfies the j^{th} constraint, for $j = 1, \ldots, k$.

- EFI for blackbox constraints is

$$\text{EFI}(x) = \mathbb{E}\{I(x)\} \prod_{j=1}^{m} p_n^{(j)}(x)$$

- and similarly IECI:

$$\text{IECI}(x_{n+1}) = -\int_{x \in \mathcal{B}} \mathbb{E}\{I(x \mid x_{n+1})\} \prod_{j=1}^{m} p_n^{(j)}(x)\, dx$$

- with similar properties as in the known constraints case.

Implementation

Lets revisit our earlier 1d known constraint problem, but now pretend we don't know the constraint.

- To grab a flexible model off the shelf we'll use `randomForest`,
- treating the constraint as binary.
- For visualization it will be cleaner to presume that $c(x) = 1$ is valid and $c(x) = 0$ is invalid.

A setup like this (GPs/EI/ `randomForest`) was first entertained by Lee et al., (2011) (http://www.ybook.co.jp/online2/oppjo/vol7/p467.html).

Initializing EFI

Noisy function; deterministic constraint.

```
ninit <- 7
X <- matrix(randomLHS(ninit, 1)*7, ncol=1)
y <- f1d(X) + rnorm(length(X), sd=0.15)
const <- 1 - as.numeric(X > lc & X < rc)
```

Objective fit.

```
gpi <- newGP(X, y, d=1, g=0.1*var(y), dK=TRUE)
ga <- garg(list(mle=TRUE), y)
mle <- jmleGP(gpi, drange=c(eps, 10), grange=c(ga$min, ga$max), gab=ga$ab)
```

Constraint fit.

```
library(randomForest)
cfit <- randomForest(X, as.factor(const))
```

EFI criteria on a grid

First, calculate $f^n_{\min}$ based on the valid observations.

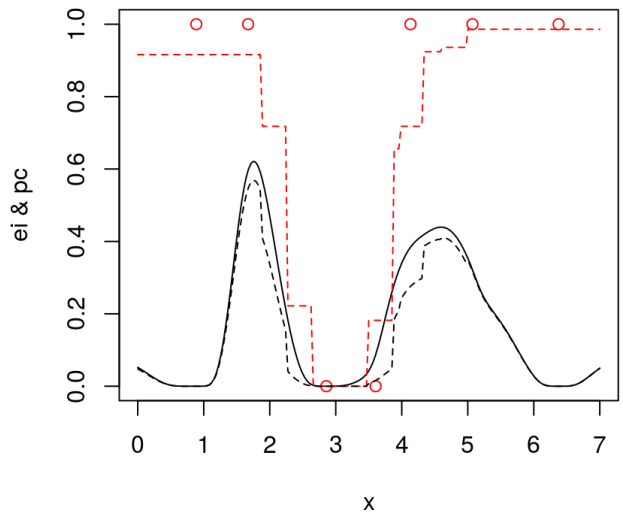
```
Xv <- X[const <= 0,,drop=FALSE]
fmin <- min(predGP(gpi, Xv, lite=TRUE)$mean)
```

Then evaluate EI and the probability of constraint satisfaction on a predictive grid.

```
XX <- matrix(seq(0,7,length=201), ncol=1)
pc <- predict(cfit, XX, type="prob")[,2]
ei <- EI(gpi, XX, fmin, pred=predGP)
```

Visualizing the criteria

```
plot(XX, ei, type="l", ylim=c(0,1), xlab="x", ylab="ei & pc")
lines(XX, pc, col=2, lty=2); points(X, const, col=2)
lines(XX, ei*pc, lty=2)
```



Choosing the next point

Updating fits.

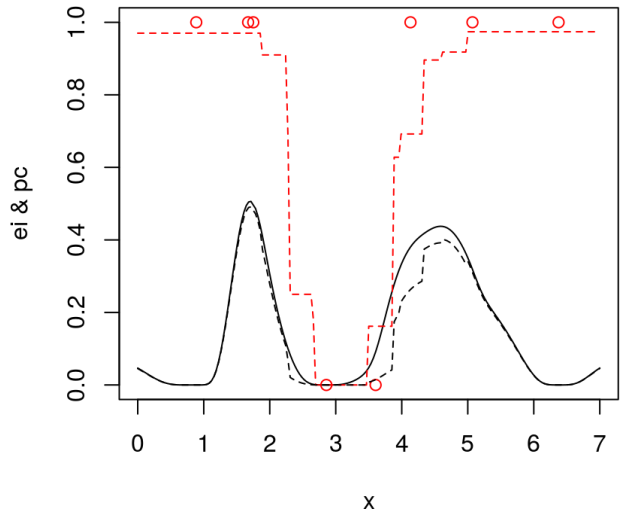
```
m <- which.max(ei*pc)
X <- rbind(X, XX[m,])
y <- c(y, f1d(XX[m,]) + rnorm(1, sd=0.15))
updateGP(gpi, XX[m,,drop=FALSE], y[length(y)])
mle <- jmleGP(gpi, drange=c(eps, 10), grange=c(ga$min, ga$max), gab=ga$ab)
const <- c(const, 1 - as.numeric(XX[m,] > lc && XX[m,] < rc))
cfit <- randomForest(X, as.factor(const))
XX <- XX[-m,,drop=FALSE]
```

Calculating EFI quantities.

```
Xv <- X[const <= 0,,drop=FALSE]
fmin <- min(predGP(gpi, Xv, lite=TRUE)$mean)
pc <- predict(cfit, XX, type="prob")[,2]
ei <- EI(gpi, XX, fmin, pred=predGP)
```

Visualizing the EFI criteria

```
plot(XX, ei, type="l", ylim=c(0,1), xlab="x", ylab="ei & pc")
lines(XX, pc, col=2, lty=2); points(X, const, col=2)
lines(XX, ei*pc, lty=2)
```



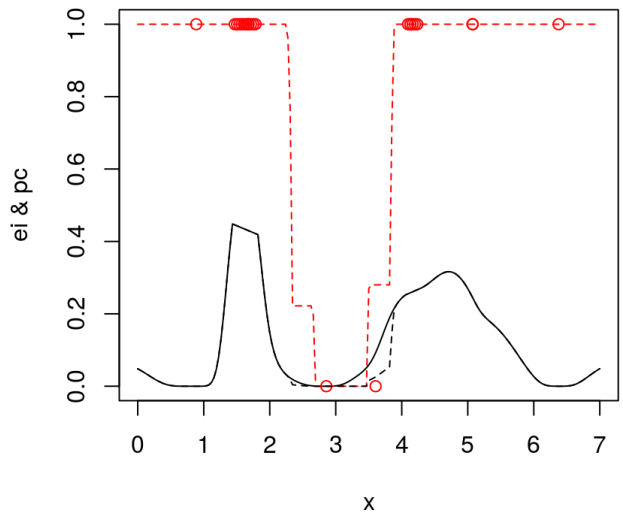
Ten more EFI steps

Lets put all that in a `for` loop and see where we get.

```
for(i in 1:15) {
    m <- which.max(ei*pc)
    X <- rbind(X, XX[m,])
    y <- c(y, f1d(XX[m,]) + rnorm(1, sd=0.15))
    updateGP(gpi, XX[m,,drop=FALSE], y[length(y)])
    mle <- jmleGP(gpi, drange=c(eps, 10), grange=c(ga$min, ga$max), gab=ga$ab)
    const <- c(const, 1 - as.numeric(XX[m,] > lc && XX[m,] < rc))
    cfit <- randomForest(X, as.factor(const))
    XX <- XX[-m,,drop=FALSE]
    Xv <- X[const <= 0,,drop=FALSE]
    fmin <- min(predGP(gpi, Xv, lite=TRUE)$mean)
    pc <- predict(cfit, XX, type="prob")[,2]
    ei <- EI(gpi, XX, fmin, pred=predGP)
}
```

Visualizing the EFI criteria

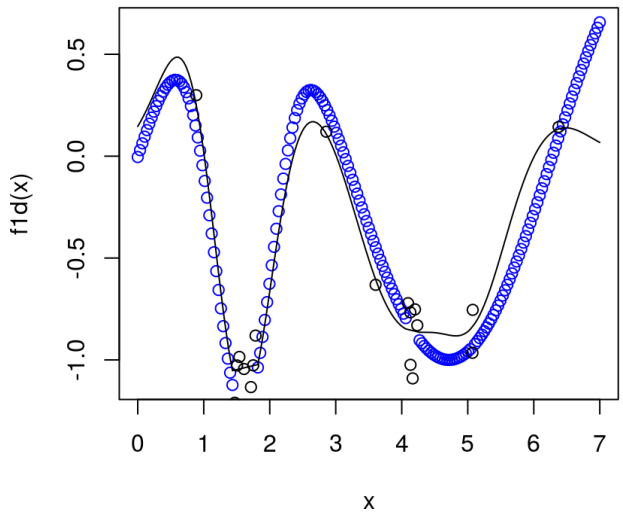
```
plot(XX, ei, type="l", ylim=c(0,1), xlab="x", ylab="ei & pc")
lines(XX, pc, col=2, lty=2); points(X, const, col=2)
lines(XX, ei*pc, lty=2)
```



Visualizing the surface

Blue dots show de-noised values of remaining candidates.

```
p <- predGP(gpi, XX, lite=TRUE)
plot(XX, f1d(XX), col="blue", xlab="x", ylab="f1d(x)")
points(X,y); lines(XX, p$mean)
```



Now IECI

Re-initialize the fit.

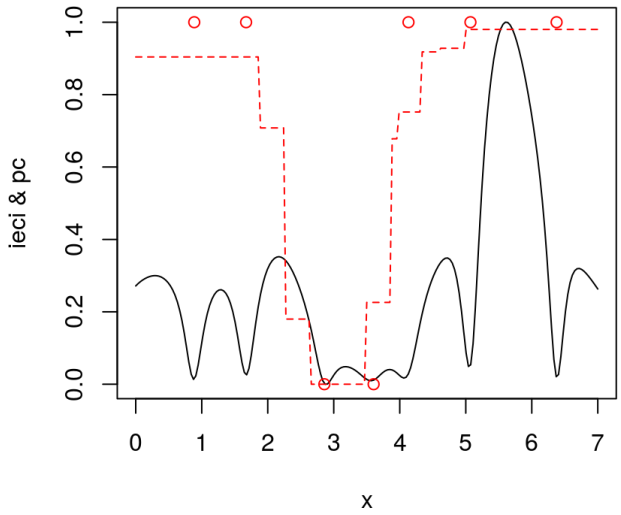
```
X <- X[1:ninit,,drop=FALSE]
y <- y[1:ninit]
const <- const[1:ninit]
gpi <- newGP(X, y, d=1, g=0.1*var(y), dK=TRUE)
mle <- jmleGP(gpi, drange=c(eps, 10), grange=c(ga$min, ga$max), gab=ga$ab)
cfit <- randomForest(X, as.factor(const))
```

Calculate IECI.

```
Xv <- X[const <= 0,,drop=FALSE]
fmin <- min(predGP(gpi, Xv, lite=TRUE)$mean)
XX <- matrix(seq(0,7,length=201), ncol=1)
pc <- predict(cfit, XX, type="prob")[,2]
ieci <- ieciGP(gpi, XX, fmin, w=pc)
ieci <- ieci - min(ieci); ieci <- ieci/max(ieci)
```

Visualizing the IECI criteria

```
plot(XX, 1-ieci, type="l", ylim=c(0,1), xlab="x", ylab="ieci & pc")
lines(XX, pc, col=2, lty=2); points(X, const, col=2)
```



Choosing the next point

Updating fits.

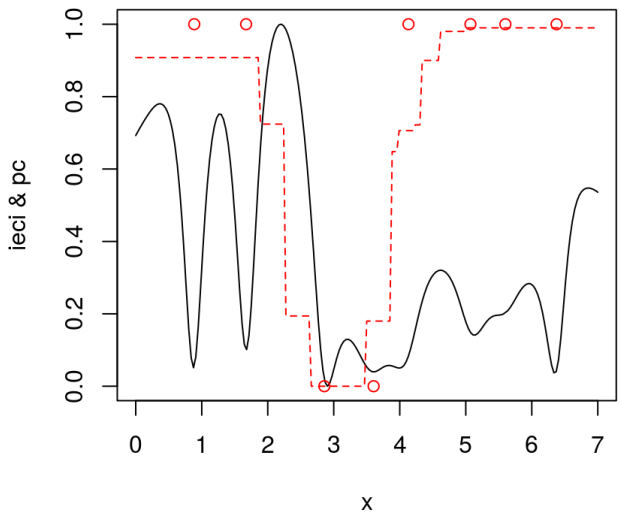
```
m <- which.min(ieci)
X <- rbind(X, XX[m,])
y <- c(y, f1d(XX[m,]) + rnorm(1, sd=0.15))
updateGP(gpi, XX[m,,drop=FALSE], y[length(y)])
mle <- jmleGP(gpi, drange=c(eps, 10), grange=c(ga$min, ga$max), gab=ga$ab)
const <- c(const, 1 - as.numeric(XX[m,] > lc && XX[m,] < rc))
cfit <- randomForest(X, as.factor(const))
XX <- XX[-m,,drop=FALSE]
```

Calculating IECI quantities.

```
Xv <- X[const <= 0,,drop=FALSE]
fmin <- min(predGP(gpi, Xv, lite=TRUE)$mean)
pc <- predict(cfit, XX, type="prob")[,2]
ieci <- ieciGP(gpi, XX, fmin, w=pc)
ieci <- ieci - min(ieci); ieci <- ieci/max(ieci)
```

Visualizing the IECI criteria

```
plot(XX, 1-ieci, type="l", ylim=c(0,1), xlab="x", ylab="ieci & pc")
lines(XX, pc, col=2, lty=2); points(X, const, col=2)
```



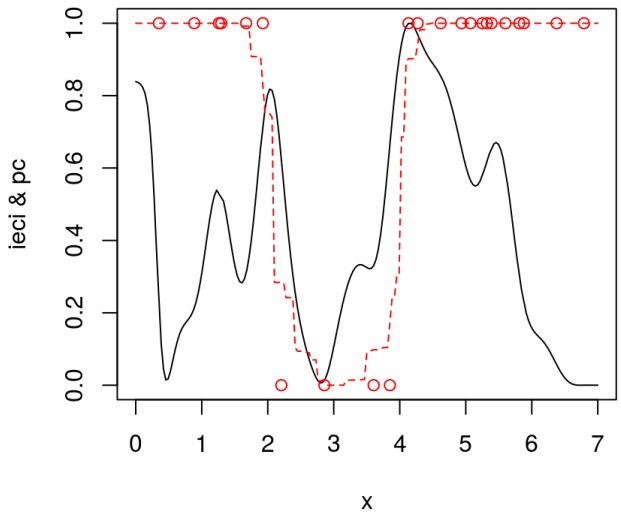
Ten more IECI steps

Lets put all that in a `for` loop and see where we get.

```
for(i in 1:15) {
    m <- which.min(ieci)
    X <- rbind(X, XX[m,])
    y <- c(y, f1d(XX[m,]) + rnorm(1, sd=0.15))
    updateGP(gpi, XX[m,,drop=FALSE], y[length(y)])
    mle <- jmleGP(gpi, drange=c(eps, 10), grange=c(ga$min, ga$max), gab=ga$ab)
    const <- c(const, 1 - as.numeric(XX[m,] > lc && XX[m,] < rc))
    cfit <- randomForest(X, as.factor(const))
    XX <- XX[-m,,drop=FALSE]
    Xv <- X[const <= 0,,drop=FALSE]
    fmin <- min(predGP(gpi, Xv, lite=TRUE)$mean)
    pc <- predict(cfit, XX, type="prob")[,2]
    ieci <- ieciGP(gpi, XX, fmin, w=pc)
    ieci <- ieci - min(ieci); ieci <- ieci/max(ieci)
}
```

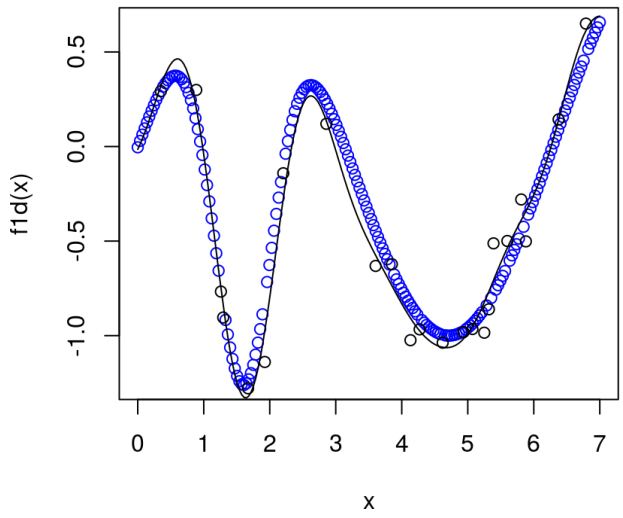
Visualizing the IECI criteria

```
plot(XX, 1-ieci, type="l", ylim=c(0,1), xlab="x", ylab="ieci & pc")
lines(XX, pc, col=2, lty=2); points(X, const, col=2)
```



Visualizing the surface

```
p <- predGP(gpi, XX, lite=TRUE)
plot(XX, f1d(XX), col="blue", xlab="x", ylab="f1d(x)")
points(X,y); lines(XX, p$mean)
```



Real-valued constraint functions

It is perhaps more typical for the constraint functions to be real-valued, and for there to be multiple such constraints.

- This simplifies matters somewhat, because there are generally more modeling choices for real-valued simulations (e.g., GPs).
- Evaluations of $c_n^{(j)}(x) \in \mathbb{R}$ provide extra information (compared to $\{0, 1\}$) comprising of the *distance* to feasibility.
- But multiple constraints obviously makes the overall problem more challenging.

EFI and IECI remain unchanged as general strategies

- suffice it that a probability of constraint satisfaction $p_n^{(j)}(x) = \mathbb{P}(c_n^{(j)}(x) \leq 0)$ can be backed out of the fitted surfaces $c_n^{(j)}(x)$ for each constraint $j = 1, \ldots, m$.
- For GPs this readily available from the CDF Φ, i.e., `pnorm` given $\mu_n^{(j)}(x)$ and $\sigma_n^{(j)2}(x)$.

Focus on constraints

So far we have focused primarily on the objective,

- but in many real world “optimization” problems, the constraints steal the show.

There are plenty of challenging problems where the objective is simple/known (e.g., linear),

- but simulation-based constraints
  - require heavy computation
  - are highly non-linear and non-convex, and therefore create many (deceptive) local minima in the input domain.
- This can make for a hard optimization problem.

For example

Here is a toy problem to fix ideas.

- A linear objective in two variables

$$\min_x \left\{ x_1 + x_2 : c_1(x) \leq 0,\ c_2(x) \leq 0,\ x \in [0,1]^2 \right\}$$

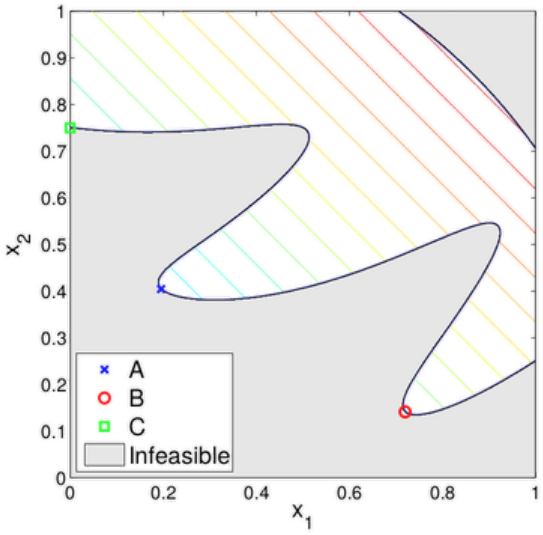
- where two non-linear constraints are given by

$$c_1(x) = \frac{3}{2} - x_1 - 2x_2 - \frac{1}{2}\sin\big(2\pi(x_1^2 - 2x_2)\big)$$
$$c_2(x) = x_1^2 + x_2^2 - \frac{3}{2}$$

Even when treating $f(x) = x_1 + x_2$ as known, this is a hard problem when $c(x)$ is treated as a blackbox.

Visualizing the toy problem

$$
\begin{aligned}
x^A &\approx [0.1954,\ 0.4044], \\
f\left(x^A\right) &\approx 0.5998, \\
x^B &\approx [0.7197,\ 0.1411], \\
f\left(x^B\right) &\approx 0.8609, \\
x^C &= [0,\ 0.75], \\
f\left(x^C\right) &= 0.75,
\end{aligned}
$$



- $c_2(x)$ may seem uninteresting, but it reminds us that solutions may not exist on every boundary.

Hybridization

Math programming has efficient algorithms for non-linear (blackbox) optimization (under constraints) with

- provable *local* convergence properties,
- lots of polished open-source software.

Whereas statistical approaches

- enjoy global convergence properties,
- excel when simulation is expensive, noisy, non-convex,

they offer limited support for constraints.

Augmented Lagrangian

One such framework involves the so-called the **augmented Lagrangian (AL)**:

$$L_A(x;\lambda,\rho) = f(x) + \lambda^\top c(x) + \frac{1}{2\rho}\sum_{j=1}^{m} \max\left(0, c_j(x)\right)^2, \qquad \text{where}$$

- $\rho > 0$ is a penalty parameter
- $\lambda \in \mathbb{R}^m_+$ serves as a Lagrange multiplier

AL-based methods thereby

- transform a constrained problem into a *sequence* of simply constrained ones.

Without the Lagrangian term $\lambda^\top c(x)$,

- one obtains (an example of) a so-called **additive penalty method (APM)**.
- The full AL advantage is automatic updates of the parameters (λ, ρ).

AL sequence

Given $(\rho^{k-1}, \lambda^{k-1})$,

1. approximately solve the **subproblem**

$$x^k = \arg\min_x \left\{L_A(x;\lambda^{k-1},\rho^{k-1}) : x \in \mathcal{B}\right\}$$

2. update:
   - $\lambda_j^k = \max\left(0, \lambda_j^{k-1} + \frac{1}{\rho^{k-1}} c_j(x^k)\right), j = 1, \ldots, m.$
   - If $c(x^k) \leq 0$, set $\rho^k = \rho^{k-1}$; otherwise, set $\rho^k = \frac{1}{2}\rho^{k-1}$
3. … then repeat, incrementing k.

- Functions f and c are only evaluated when solving the **subproblem(s)**, comprising an "inner loop".

Interactive AL demo

See `wildprob.R` with the course material.

Convergence

AL methods are not designed for global optimization, however the convergence results have a certain robustness.

Even if the "inner" sub-problem

$$x^k = \arg\min_x \left\{ L_A(x; \lambda^{k-1}, \rho^{k-1}) : x \in \mathcal{B} \right\}$$

cannot be solved exactly,

- "outer" iterations will converge so long as the "inner" problem makes "progress".
- (Similar to EM or weak learner results.)

How is "outer" convergence determined?

- In our setting, we'll have a maximal computational budget.
- But you could stop when all constraints are satisfied, and the gradient of the Lagrangian is sufficiently small.

Statistical inner solver

The "inner" solver can be anything. Our interactive demo used Nelder–Mead (`optim` default), but approximating the derivative is expensive.

- Derivative-free solvers are an option.
- Of course, we'll focus on methods based on statistical surrogates, and hope for a more global searching flavor.

The idea is to train the "inner" solver with all evaluations

$$(x_1, f(x_1), c(x_1)), \ldots, (x_n, f(x_n), c(x_n))$$

collected over all "inner" and "outer" loops (Gramacy, et al., 2016) (http://amstat.tandfonline.com/doi/full/10.1080/00401706.2015.1014065).

- Whereas in a more conventional approach, each "inner" solver would be independent of the next one.

Separated modeling

Consider a separate/independent GP model each component of the AL.

- f^n emitting $Y_{f^n}(x)$
- $c^n = (c_1^n, \ldots, c_m^n)$ emitting $Y_c^n(x) = (Y_{c_1}^n(x), \ldots, Y_{c_m}^n(x))$

The distribution of the **composite random variable**

$$Y(x) = Y_f(x) + \lambda^\top Y_c(x) + \frac{1}{2\rho} \sum_{j=1}^m \max(0, Y_{c_j}(x))^2$$

can serve as a surrogate for $L_A(x; \lambda, \rho)$.

- Simplifications when f is known.

Tractable surrogate for `optim`

The composite posterior mean is available in closed form, e.g., under GP priors.

$$\mathbb{E}\{Y(x)\} = \mu_f^n(x) + \lambda^\top \mu_c^n(x) + \frac{1}{2\rho} \sum_{j=1}^m \mathbb{E}\{\max(0, Y_{c_j}(x))^2\}$$

A result from generalized EI (Schonlau, Jones & Welch, 1998) (https://www.jstor.org/stable/4356058?seq=1#page_scan_tab_contents) helps us work out the expectation inside that sum above.

$$\mathbb{E}\{\max(0, Y_{c_j}(x))^2\} = \mathbb{E}\{I_{-Y_{c_j}}(x)\}^2 + \mathbb{V}\mathrm{ar}[I_{-Y_{c_j}}(x)]$$
$$= \sigma_{c_j}^{2n}(x)\left[\left(1 + \left(\frac{\mu_{c_j}^n(x)}{\sigma_{c_j}^n(x)}\right)^2\right)\Phi\left(\frac{\mu_{c_j}^n(x)}{\sigma_{c_j}^n(x)}\right) + \frac{\mu_{c_j}^n(x)}{\sigma_{c_j}^n(x)}\phi\left(\frac{\mu_{c_j}^n(x)}{\sigma_{c_j}^n(x)}\right)\right].$$

Expected improvement for AL

The simplest way to evaluate the EI is via Monte Carlo:

- take 100 samples $Y_f^{(i)}(x)$ and $Y_c^{(i)}(x)$
- then $\mathrm{EI}(x) \approx \frac{1}{100}\sum_{i=1}^{100}\max\{0, y_{\min}^n - Y^{(i)}(x)\}$

The "max" in the AL makes analytic calculation intractable.

But you can remove the "max" and obtain an analytic EI with **slack variables**.

- Introduce s_j, for $j = 1, \ldots, m$, i.e., one for each $c_j(x)$,
- convert inequality into equality constraints: $c_j(x) - s_j = 0$
- augment with constraints $s_j \geq 0$, for $j = 1, \ldots, m$.
    - In practice these are subsumed into $\mathcal{B}$.

Slacks also facilitate the only EI-based method for handing mixed (equality and inequality) constraints (Picheny, et al., 2016) (https://arxiv.org/abs/1605.09466).

On our toy data

It is too much to code all this up by ourselves for a real-time run in lecture.

- So we'll borrow the `optim.auglag` implementation in `laGP`.

Here is an implementation of the toy problem in R, using the format required for `optim.auglag`.

```
aimprob <- function(X, known.only=FALSE)
 {
    if(is.null(nrow(X))) X <- matrix(X, nrow=1)
    f <- rowSums(X)
    if(known.only) return(list(obj=f))
    c1 <- 1.5-X[,1]-2*X[,2]-0.5*sin(2*pi*(X[,1]^2-2*X[,2]))
    c2 <- rowSums(X^2)-1.5
    return(list(obj=f, c=cbind(c1,c2)))
 }
```

And we'll work in the following bounding box.

```
B <- matrix(c(rep(0,2),rep(1,2)),ncol=2)
```

Several versions

One non-AL version (EFI), which is also provided by `laGP`.

```
efi <- optim.efi(aimprob, B, end=50, verb=0)
```

One AL version guided by the posterior mean surface of the AL comprised of separated surrogate models.

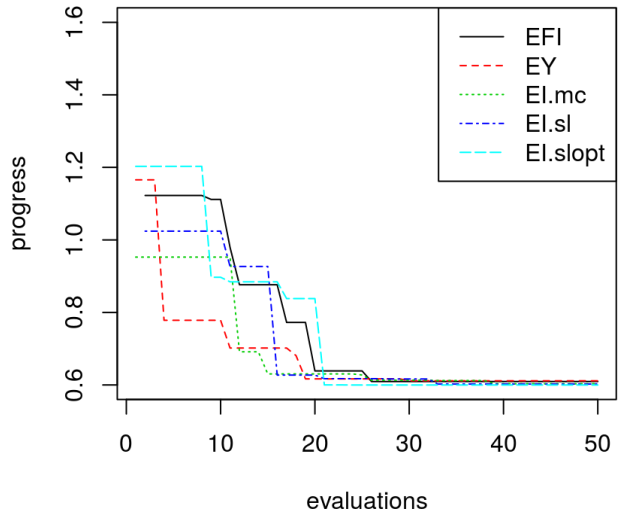
```
ey <- optim.auglag(aimprob, B, end=50, ey.tol=1, verb=0)
```

Three variations with EI on the AL composite.

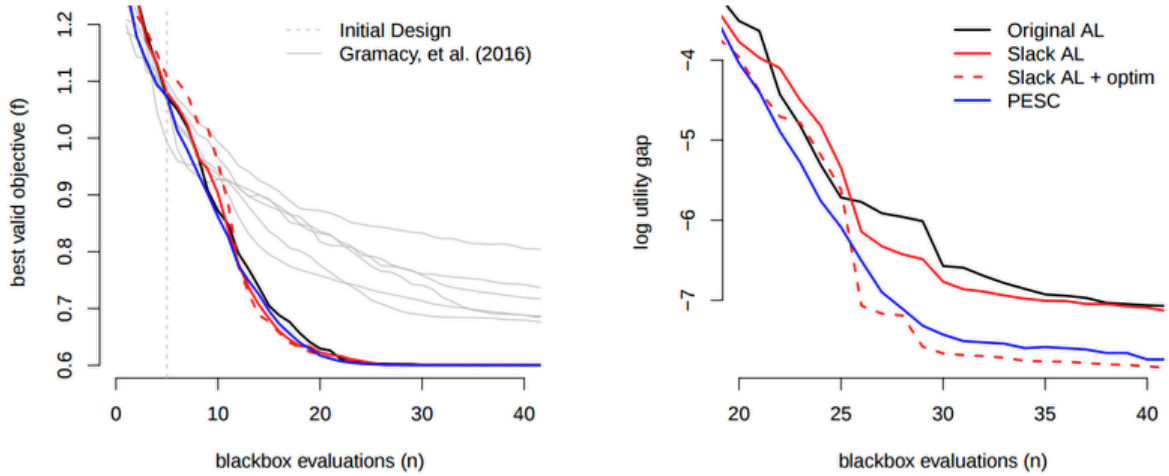
```
ei.mc <- optim.auglag(aimprob, B, end=50, verb=0)
ei.sl <- optim.auglag(aimprob, B, end=50, slack=TRUE, verb=0)
ei.slopt <- optim.auglag(aimprob, B, end=50, slack=2, verb=0)
```

Visualizing progress

```
plot(efi$prog, type="l", ylim=c(0.6, 1.6), ylab="progress", xlab="evaluations")
lines(ey$prog, col=2, lty=2); lines(ei.mc$prog, col=3, lty=3)
lines(ei.sl$prog, col=4, lty=4); lines(ei.slopt$prog, col=5, lty=5)
legend("topright", c("EFI", "EY", "EI.mc", "EI.sl", "EI.slopt"), col=1:5, lty=1:5)
```



Typical behavior



Average results after 100 restarts.

- Pretty speedy.
- Recall that our `optim` -AL required 100+ evaluations for *local* convergence.

Unknown objective?

Sure, no problem. How about this crazy one?

```
f2d <- function(x, y=NULL)
  {
    if(is.null(y)) {
      if(!is.matrix(x)) x <- matrix(x, ncol=2)
        y <- x[,2]; x <- x[,1]
    }
    g <- function(z)
        return(exp(-(z-1)^2) + exp(-0.8*(z+1)^2) - 0.05*sin(8*(z+0.1)))
    return(-g(x)*g(y))
  }

aimprob2 <- function(X, known.only = FALSE)
{
  if(is.null(nrow(X))) X <- matrix(X, nrow=1)
  if(known.only) stop("no outputs are treated as known")
  f <- f2d(4*(X-0.5))
  c1 <- 1.5 - X[,1] - 2*X[,2] - 0.5*sin(2*pi*(X[,1]^2 - 2*X[,2]))
  c2 <- rowSums(X^2)-1.5
  return(list(obj=f, c=cbind(c1,c2)))
}
```

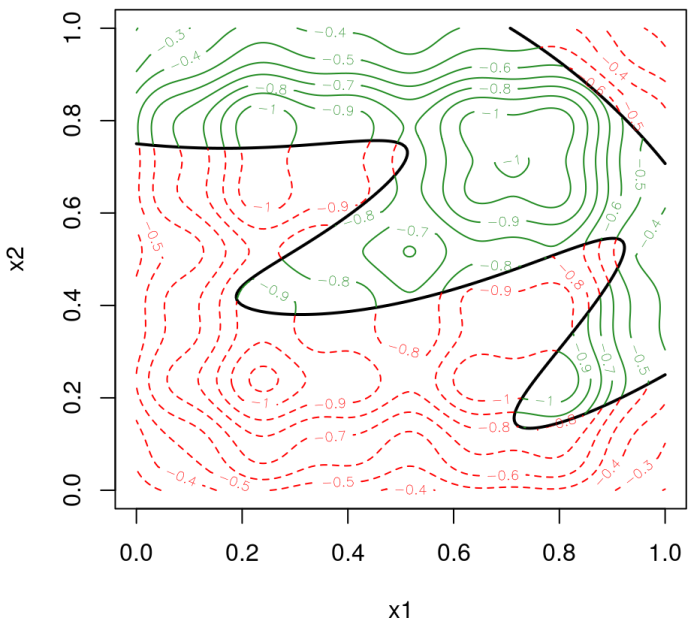
Visualizing sub-routine

Here is a little function to re-draw the surface on-demand.

```
plot.aimprob2 <- function()
 {
    x <- seq(0,1, length=200)
    X <- expand.grid(x, x)
    out <- aimprob2(as.matrix(X))
    fv <- out$obj
    fv[out$c[,1] > 0 | out$c[,2] > 0] <- NA
    fi <- out$obj
    fi[!(out$c[,1] > 0 | out$c[,2] > 0)] <- NA
    plot(0, 0, type="n", xlim=B[1,], ylim=B[2,], xlab="x1", ylab="x2")
    contour(x, x, matrix(out$c[,1], ncol=length(x)), nlevels=1, levels=0,
        drawlabels=FALSE, add=TRUE, lwd=2)
    contour(x, x, matrix(out$c[,2], ncol=length(x)), nlevels=1, levels=0,
        drawlabels=FALSE, add=TRUE, lwd=2)
    contour(x, x, matrix(fv, ncol=length(x)), nlevels=10, add=TRUE,
        col="forestgreen")
    contour(x, x, matrix(fi, ncol=length(x)), nlevels=13, add=TRUE, col=2, lty=2)
 }
```

Visualizing

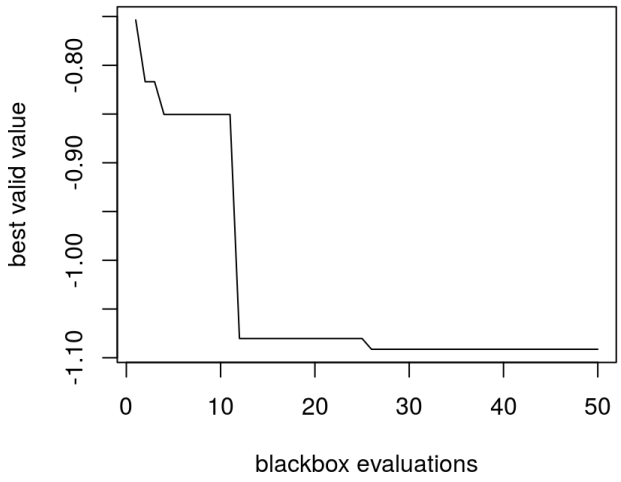
```
plot.aimprob2()
```



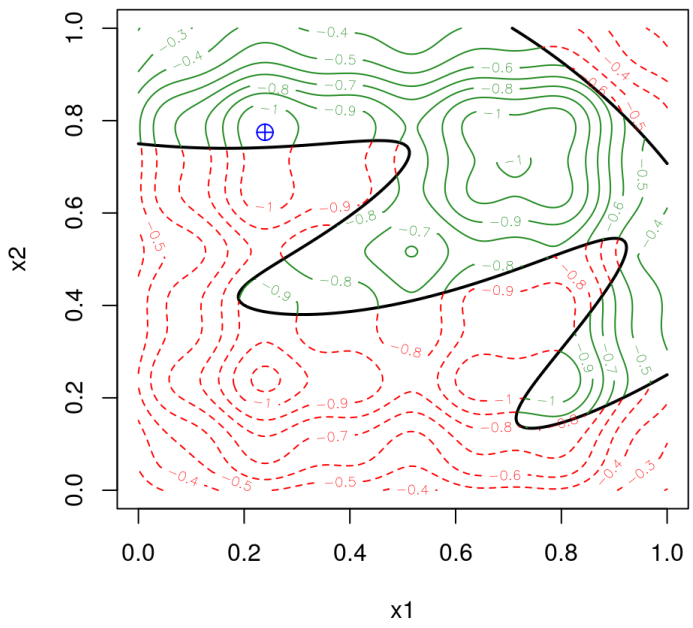
Optimizing via EI with AL

Pretty fast progress.

```
out2 <- optim.auglag(aimprob2, B, fhat=TRUE, start=20, end=50, verb=0)
plot(out2$prog, type="l", ylab="best valid value", xlab="blackbox evaluations")
```




```
plot.aimprob2()
v <- apply(out2$C, 1, function(x) { all(x <= 0) })
X <- out2$X[v,]; obj <- out2$obj[v]; xbest <- X[which.min(obj),]
points(xbest[1], xbest[2], pch=10, col="blue", cex=1.5)
```



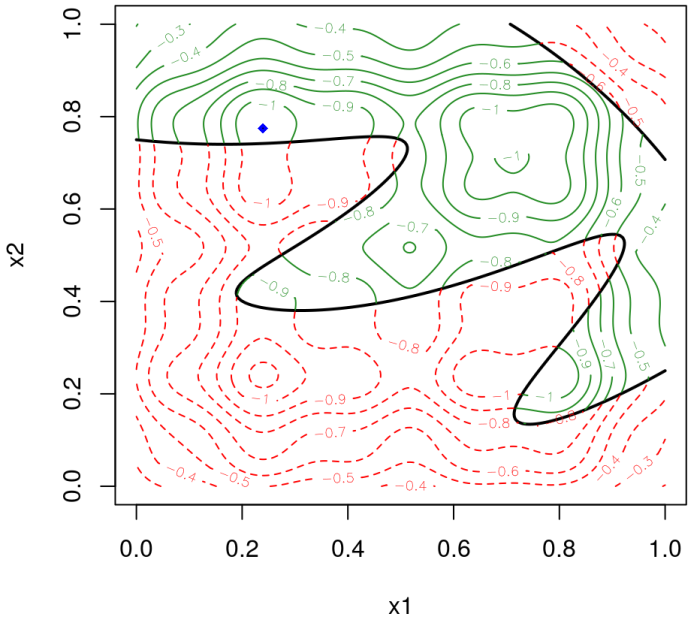
How about initializing an `optim` search from that point to see if we can "drill down" any further?

```
aimprob2.AL <- function(x, B, lambda, rho)
{
  if(any(x < B[,1]) | any(x > B[,2])) return(Inf)
  fc <- aimprob2(x)
  al <- fc$obj + lambda%*%drop(fc$c) + rep(1/(2*rho),2)%*%pmax(0,drop(fc$c))^2
  return(al)
}

## loop over AL updates until a valid solution is found
lambda <- out2$lambda[nrow(out2$lambda),]; rho <- out2$rho[length(out2$rho)]
while(1) {
    o <- optim(xbest, aimprob2.AL, control=list(maxit=15),
      B=B, lambda=lambda, rho=rho)
    fc <- aimprob2(o$par)

    if(all(fc$c <= 0)) { break
    } else {
      lambda <- pmax(0, lambda + (1/rho)*fc$c)
      rho <- rho/2; xbest <- o$par
    }
  }
```

```
plot.aimprob2()
points(o$par[1], o$par[2], pch=18, col="blue")
segments(xbest[1], xbest[2], o$par[1], o$par[2])
```



Other demos

For further comparison with `optim` directly on the AL,

- see `demo("ALfhat")` in the `laGP` package.

Two other demos show a mixed constraints setup

- A 2d problem (`"GSBP"`) involving
  - a Goldstein–Price objective
  - the toy sinusoidal inequality constraint
  - and two equality constraints that together trace out four ribbons of valid region
- A 4d problem (`"LAH"`) with
  - a known linear objective
  - an inequality constraint derived from the “Ackley” function (https://www.sfu.ca/~ssurjano/ackley.html)
  - an equality constraint derived from the “Hartman” function (https://www.sfu.ca/~ssurjano/hart4.html)

`demo("GSBP")` – such a crazy surface!