R Notebook for Chapter 5: Optimal Stopping

Companion Code to Gaussian Process Models for Quantitative Finance

Mike Ludkovski and Jimmy Risk

4/8/2024

This RMarkdown file presents an illustrative use of Gaussian Process surrogates for Bermudan Option Pricing via Regression Monte Carlo, linking to Chapter 5 of the book. We directly embed `R` code snippets to showcase the straightforward use of the methodology.

This notebook relies on the `mlOSP` package available at https://github.org/mludkov/mlOSP

Regression Monte Carlo methodology

An optimal stopping problem (OSP) is described through two main ingredients: the state process (X_t) taking values in $\mathcal{X} \subseteq \mathbb{R}^d$ and the reward function $G(t, X_t)$.

Below we assume exercise opportunities are discrete according to time step Δt; indexing throughout is by the time steps $k = 1, \ldots, K$, with corresponding $t_k = k\Delta t$, and the following notation:

- N_k: number of training inputs at step k (can again vary across k);
- $\mathcal{D}_k$: simulation design, i.e. the collection of training inputs $x^{(k),1:N_k}$ at step k; $|\mathcal{D}_k| = N_k$
- $\mathcal{H}_k$: functional approximation space where $\widehat{T}(k, \cdot)$ is searched within;
- $y_k^{1:N_k}$ pathwise samples of timing value used as the responses in the regression model.

In order to abstract from the payoff function specifics, we work with the *timing value*,

$$T(k, x) := q(k, x) - h(k, x).$$

Exercising is optimal when the timing value is negative and the stopping boundary is the zero contour of $T(k, \cdot)$. This gives the action strategy: $A_k(x) = 1 \Leftrightarrow T(k, x) \geq 0$.

Regression Monte Carlo implements the following backward recursion loop:

For $k = K - 1, \ldots, 1$ repeat:

i) Learn the timing value $\hat{T}(k, \cdot)$ in the in-the-money region $\mathcal{X}_{in} := \{x : h(k, x) > 0\}$;

ii) Set $\widehat{A}_k(\cdot) := 1_{\{\hat{T}(k,\cdot)>0\}}$ and $\widehat{V}(k, \cdot) := G(k, \cdot) + \max\big(\hat{T}(k, \cdot), 0\big)$.

Dynamic Emulation Algorithm: the **mlOSP** template.

Input: $K = T/\Delta t$ (time steps), (N_k) (simulation budget per step)}, w (path lookahead)

For: $k = K - 1, \ldots, 0$

- Generate training design $\mathcal{D}_k := (x^{(k),1:N_k}(k))$ of size N_k
- Generate w-step paths $x^{(k),n}(k) \mapsto x^{(k),n}(s)$ for $n = 1, \ldots, N_k$, $s = k + 1, \ldots, (k + w) \wedge K$
- Generate pathwise forward stopping rule $\tau_k^n = \min\{s \geq k + 1 : \widehat{T}(s, x^{(k),n}(s)) < 0\} \wedge (k + w) \wedge K$

- Generate pathwise timing value

$$y_{k+1}^n = \begin{cases} h\big(\tau_k^n, x^{(k),n}(\tau_k^n)\big) - h\big(k, x^{(k),n}(k)\big) & \text{if} \quad \tau_k^n < k+w; \\ \widehat{T}\big(k+w, x^{(k),n}(k+w)\big) + h\big(k+w, x^{(k),n}(k+w)\big) - h\big(k, x^{(k),n}(k)\big) & \text{if} \quad \tau_k^n = (k+w) \wedge K. \end{cases} \tag{1}$$

- Fit $\widehat{T}(k,\cdot) \leftarrow \arg\min_{f(\cdot)\in\mathcal{H}_k} \sum_{n=1}^{N_k} |f(x^{(k),n}(k)) - y_{k+1}^n|^2$
- End for-loop
- Return fitted objects $\{\widehat{T}(k,\cdot)\}_{k=0}^{K-1}$

To do so, one again employs Monte Carlo simulation, approximating with a sample average on a set of fresh forward test scenarios $x^{1:N'}(k), k = 1, \ldots, K$, $x^{n'}(0) = X(0)$,

$$\check{V}(0, X(0)) = \frac{1}{N'} \sum_{n'=1}^{N'} h\left(\tau^{n'}, x^{n'}(\tau^{n'})\right)$$

where $\tau^n := \min\{k \geq 0 : x^n(k) \in \widehat{\mathcal{S}}_k\}$ is the pathwise stopping time. Because $\check{V}(0; X(0))$ is based on out-of-sample test scenarios, it is an unbiased estimator of $\tilde{V}$ and hence for large enough test sets it will yield a lower bound on the true optimal expected reward, $\mathbb{E}[\check{V}(0, X(0))] = \tilde{V}(0, X(0); \widehat{A}_{0:K}) < V(0, X(0))$. The interpretation is that the simulation design $\mathcal{D}_k$ defines a training set, while $x^{1:N'}(k)$ forms the test set. The latter is not only important to obtain unbiased estimates of expected reward from the rule $\tau_{\widehat{A}_{0:K}}$, but also to enable an apples-to-apples comparison between multiple solvers which should be run on a fixed test set of X-paths.

The `{mlOSP}` package implements the above RMC template based on the workflow pipeline of:

(i) Defining the `model`, which is a list of (a) parameters that determine the dynamics of (X_t); (b) the payoff function $G(t, x)$; and (c) the tuning parameters determining the regression emulator specification.

(ii) Calling the appropriate top-level solver. The solvers are indexed by the underlying type of *simulation design*. They also use a top-level `method` argument that selects from a collection of implemented regression modules. Otherwise, all other parameters are passed through the above `model` argument. The solver returns a collection of fitted emulators—an `R` list containing the respective regression object of $\widehat{T}(k,\cdot)$ for each time step k, plus a few diagnostics;

(iii) Evaluating the obtained fitted emulators through an out-of-sample forward simulator via `forward.sim.policy` that evaluates @ref(eq:tildeV). The latter is a top-level function that can work with any of the implemented regression objects. Alternatively, one may also *visualize* the emulators through a few provided plotting functions.

One-dimensional example: Figure 5.1

Consider a 1-D Bermudan Put with payoff $e^{-rt}(\mathcal{K} - x)_+$ where the underlying dynamics for the asset price (X_t are given by Geometric Brownian Motion (GBM)

$$dX_t = (r - \delta)X_t dt + \sigma X_t dW_t, \qquad X_0 = x_0,$$

with scalar parameters r, δ, σ, x_0. Thus, the dicrete X_k can be simulated exactly by sampling from the respective log-normal distribution. In the model specification below we have $r = 0.06, \delta = 0, T = 1, \sigma = 0.2$ and the Put strike is $\mathcal{K} = 40$. Exercising the option is possible $K = 25$ times before expiration, i.e., $\Delta t = 0.04$. To implement the above OSP instance just requires defining a named list with the respective parameters (plus a few more for the regression model specified below):

```
put1d.model <- c(K=40, payoff.func=put.payoff,  # payoff function
            x0=40,sigma=0.2,r=0.06,div=0,T=1,dt=0.04,dim=1, sim.func=sim.gbm,
            km.cov=4,km.var=1,kernel.family="matern5_2",  # GP emulator params
            pilot.nsims=0,batch.nrep=200,N=25)
```

As a representative solver, we utilize `osp.fixed.design` that asks the user to specify the simulation design directly. To select a particular regression approach, `osp.fixed.design` has a `method` field which can take a large range of regression methods. Specifics of each regression are controlled through additional `model` fields. As example, the code snippet below employs the `{DiceKriging}` Gaussian Process (GP) emulator with fixed hyperparameters and a constant prior mean, selected through `method="km"`. The kernel family is Matérn-5/2, with hyperparameters specified via `km.cov`, `km.var` above. For the simulation design (`input.domain` field) we take $\{16, 17, \ldots, 40\}$ with 200 replications (`batch.nrep`) per site, yielding $N = |\mathcal{D}| = 200 \cdot 25$. The replications are treated using the SK method (Ankenman et al. 2010), pre-averaging the replicated outputs before training the GP.

```
train.grid.1d <- seq(16, 40, len=25)  # simulation design
km.fit <- osp.fixed.design(put1d.model,input.domain=train.grid.1d, method="km")
```

Note that no output is printed: the produced object `km.fit` contains an array of 24 (one for each time step, except at maturity) fitted GP models, but does not yet contain the estimate of the option price $V(0, X_0$. Indeed, we have not defined any test set, and consequently are momentarily postponing the computation of $\check{V}(0, X_0$.

To compare, we run two another `{mlOSP}` solver, namely the LS scheme, implemented in the **osp.prob.design** solver, where the simulation design is based on forward X-paths. We moreover replace the regression module with a smoothing cubic spline (**smooth.spline**; see Kohler (2008) for a discussion of using splines for RMC). The latter requires specifying the number of knots `model$nk`. Only 2 lines of code are necessary to make all these modifications and obtain an alternative solution of the same OSP:

```
put1d.model$nk=20  # number of knots for the smoothing spline
spl.fit <- osp.prob.design(N=30000,put1d.model,method="spline") #30K training paths
```

Again, there is no visible output; `spl.fit` now contains a list of 24 fitted spline objects that are each parametrized by 20 (number of chosen knots) cubic spline coefficients.

The plot below visualizes the fitted timing value from one time step. To do so, we predict $\widehat{T}(k, x)$ based on a fitted emulator (at $t = 10\Delta t = 0.4$) over a collection of test locations. In the Figure, this is done for both of the above solvers (GP-km and Spline), moreover we also display the 95% credible intervals of the GP emulator for $\widehat{T}(k, \cdot)$. Keep in mind that the exact *shape* of $\widehat{T}(k, \cdot)$ is irrelevant in **mlOSP**, all that matters is the implied $\widehat{A}_k$ which is the zero level set of the timing value, i.e. determined by the *sign* of $\widehat{T}(k, \cdot)$. Therefore, the two regression methods yield quite similar exercise rules, although they do differ (e.g. at $x = 35$ the spline-based rule is to continue, while the GP-based one is to stop and exercise the Put). Since asymptotically both solvers should recover the optimal rule, their difference can be attributed to training errors. As such, uncertainty quantification of $\widehat{T}(k, \cdot)$ is a useful diagnostic to assess the perceived accuracy of $\widehat{\mathcal{A}}_k$. In the Figure the displayed uncertainty band shows that the GP emulator has low confidence about the right action to take for nearly all $x \leq 37$, since zero is inside the 95% credible band and therefore the positivity of $\widehat{T}(k, \cdot)$ in that region is not statistically significant.

```
check.x <- seq(25, 40, len=500)   # predictive sites
km.pred <- predict(km.fit$fit[[10]],data.frame(x=check.x), type="UK")
to.plot.2 <- data.frame(km.fit=km.pred$mean,x=check.x,km.up=km.pred$upper95,km.down=km.pred$lower95,
                        sp.fit=predict(spl.fit$fit[[10]],check.x)$y)
ggplot(to.plot.2, aes(x=x)) +
  geom_line(aes(y=km.fit), size=1.25,color="black") +
  geom_line(aes(y=sp.fit), size=1.25, color="purple") +
  geom_line(aes(y=km.up),size=0.5,color="black",linetype="twodash") +
  geom_line(aes(y=km.down),size=0.5,color="black",linetype="twodash") +
   theme_light() + geom_hline(yintercept=0,linetype="dashed") +
  scale_x_continuous(expand=c(0,0),limits=c(26,40))  + scale_y_continuous(expand=c(0,0), limits=c(-0.5,
   labs(x="x", y=expression(paste("Timing Value ",hat(T)(k,x)))) +
  theme(axis.title = element_text(size = 9))
```

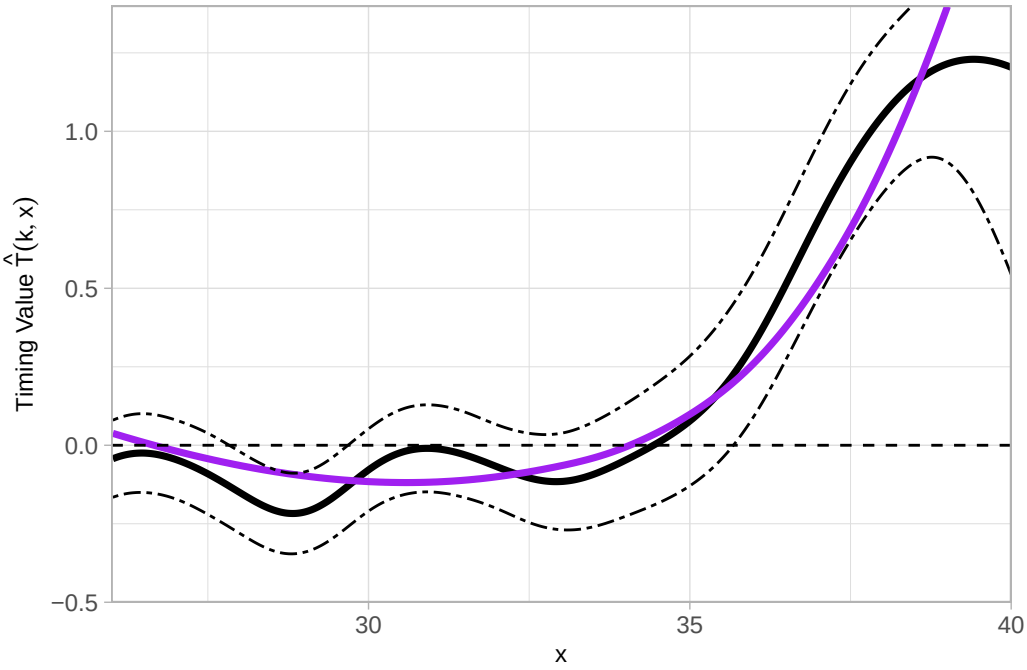

Figure 1: Timing value of a Bermudan Put based on GP emulator (black) and a Smoothing Spline emulator (purple) at $k = 10$ ($t = 0.4$). We also display the uncertainty quantification regarding the GP fit of $\widehat{T}(k, \cdot)$ (the dashed 95% band).

Two-dimensional examples: Figure 5.2

`{mlOSP}` is designed to be dimension-agnostic, so that building a multi-dimensional model follows the exact same steps. For example, the two-line snippet below defines a 2D model with Geometric Brownian motion dynamics for the two assets X_1, X_2 and a basket average Put payoff

$$G_{\text{avePut}}(t, \mathbf{x}) = e^{-rt}(\mathcal{K} - (x_1 + x_2)/2)_+, \quad x \in \mathbb{R}^2_+.$$

The two assets are assumed to be uncorrelated with identical dynamics; the strike is $\mathcal{K} = 40$ and at-the-money initial condition $\mathbf{X}_0 = (40, 40)$.

```
model2d <- list(K=40,x0=rep(40,2),sigma=rep(0.2,2),r=0.06,div=0,
                    T=1,dt=0.04,dim=2,sim.func=sim.gbm, payoff.func=put.payoff)
```

We first solve with a GP emulator that has a space-filling training design of $n = 150$ sites replicated with batches of $a = 100$ each for a total of $N = 15,000$ simulations:

```
model2d$N <- 150   # n
model2d$kernel.family <- "gauss" # squared-exponential kernel
model2d$batch.nrep <- 100
model2d$pilot.nsims <- 0

sob150 <- sobol(276, d=2) # Sobol space-filling sequence
# triangular approximation domain for the in-the-money simulation design X_in
sob150 <- sob150[ which( sob150[,1] + sob150[,2] <= 1) ,]
sob150 <- 25+30*sob150  # Lower-left triangle in [25,55]x[25,55]

sob.km <- osp.fixed.design(model2d,input.domain=sob150, method="trainkm")
```

The next plot created with 'ggplot shows the space-filling training design at step $k = 15$ and an image plot

the estimated timing value $\widehat{T}(k,\cdot)$. Recall that the stopping region is the level set where the timing value is negative, indicated with the red contours that delineate the exercise boundary.

```
g2.km <- plt.2d.surf( sob.km$fit[[15]], x=seq(25,50, len=101), y=seq(25,50,len=101))+ylab("") +
  guides(fill = guide_colourbar(barwidth = 0.4, barheight = 7)) +
  theme(axis.title=element_text(size=9),axis.text=element_text(size=9))
g2.km$layers[[3]]$aes_params$size <- 1.4
g2.km
```

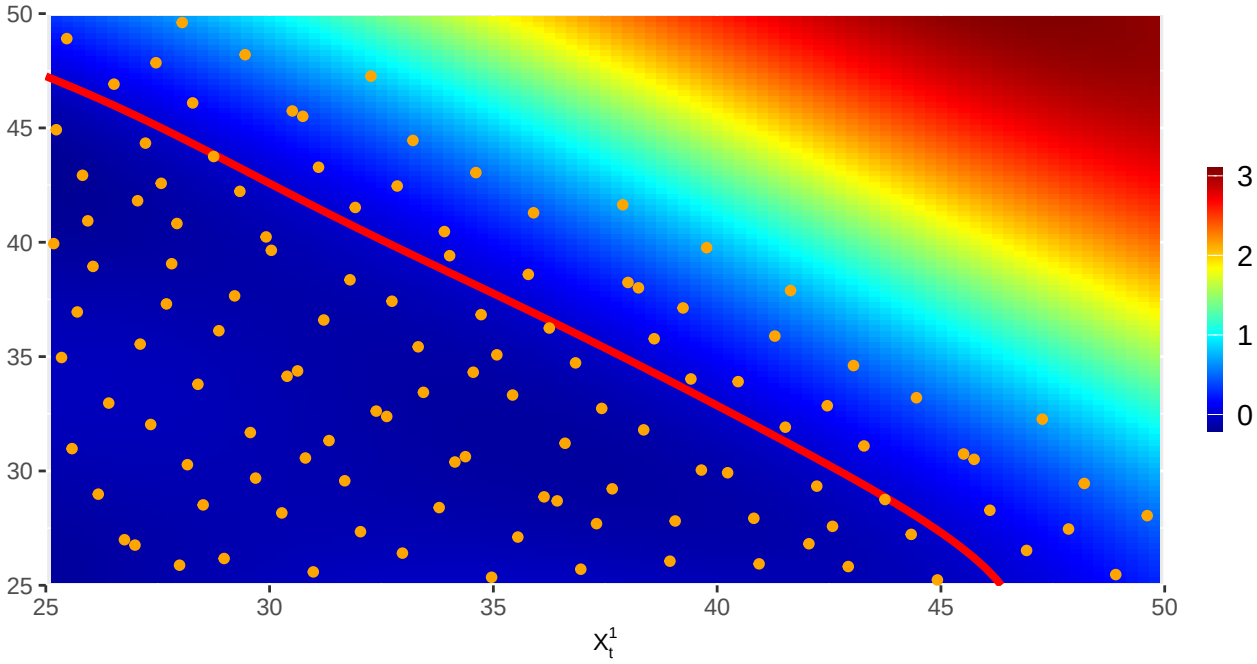

Figure 2: Timing Value of a 2D Basket Put at $k = 15$ ($t = 0.6$). The red contour shows the boundary of the stopping set (bottom-left corner). The colors indicate the value of $\widehat{T}(k,x)$. Left: LM emulator; Right: GP emulator.

```
#grid.arrange(g1.lm,g2.km,nrow=1,widths=c(4,3.7))
```

```
testModel[[3]]$bases <- function(x) return(cbind( poly_base(x, 2), testModel[[3]]$payoff.func(x,testMode

prob.lm2 <- osp.prob.design(15000,testModel[[3]], method="lm")  # Train with 15,000 paths
g3.lm <- plt.2d.surf( prob.lm2$fit[[5]], x=seq(70,130, len=101), y=seq(70,130,len=101),  bases=testModel
  theme(axis.title=element_text(size=9),axis.text=element_text(size=9))
g3.lm
```

Out-of-sample Tests

By default, the RMC solvers only create the functional representations of the timing/value function and do not return any explicit estimates of the option price or stopping rule. This reflects the strict separation between training and testing, as well as the fact that RMC builds a global estimate of $\widehat{T}(k,\cdot)$ and hence can be used to obtain a range of option prices (e.g. by changing X_0).

The following code snippet builds an out-of-sample database of X-paths. This is done iteratively by employing the underlying simulator, already saved under the `model2d$sim.func` field. The latter is interpreted as the function to generate a vector of X_{k+1}'s conditional on X_k. By applying `payoff.func` to the final vector (which stores values of X_T) we can get an estimate of the respective European option price $\mathbb{E}[G(T,X_T]$. By calling `forward.sim.policy` with a previously saved `{mlOSP}` object we then obtain a collection of

$G(\tau^n, X^n_{\tau^n}), n = 1, \ldots,$ that can be averaged to obtain a $\check{V}(0, X_0$.

```
nSims.2d <- 40000  # use N'=40,000 fresh trajectories
nSteps.2d <- 25

# For comparison build a quadratic approximation for each \widehat{T}(k,.)
bas22 <- function(x) return(cbind(x[,1],x[,1]^2,x[,2],x[,2]^2,x[,1]*x[,2]))
model2d$bases <- bas22
prob.lm <- osp.prob.design(15000,model2d, method="lm")  # Train with 15,000 paths

set.seed(102)
test.2d <- list()  # store a database of forward trajectories
test.2d[[1]] <- model2d$sim.func( matrix(rep(model2d$x0, nSims.2d),nrow=nSims.2d,
                                         byrow=T), model2d, model2d$dt)
for (i in 2:(nSteps.2d+1))  # generate forward trajectories
   test.2d [[i]] <- model2d$sim.func( test.2d [[i-1]], model2d, model2d$dt)
oos.lm <- forward.sim.policy( test.2d, nSteps.2d, prob.lm$fit, model2d) # lm payoff
oos.km <- forward.sim.policy( test.2d, nSteps.2d, sob.km$fit,  model2d) # km payoff
cat( paste('Price estimates', round(mean(oos.lm$p),3), round(mean(oos.km$p),3)))
```

```
## Price estimates 1.45 1.444
```

```
# check: estimated European option value
cat( paste('European Put: ', mean( exp(-model2d$r*model2d$T)*model2d$payoff.func(test.2d[[25]], model2d
```

```
## European Put:  1.21414910319116
```

The reported European Put estimate of 1.214 can be used as a control variate to adjust $\check{V}$ since it is based on the same test paths and so we expect that $\check{V}^{EU}(0, x_0) - \mathbb{E}\left[h(T, X(T))|X(0) = x_0\right] \simeq \check{V}(0, x_0) - \mathbb{E}\left[h(\hat{\tau}, X(\hat{\tau}))|X(0) = x_0\right]$.

Sequential Design

The `osp.seq.design` solver construct adaptive 'smart' simulation designs in order to maximize the accuracy of $\widehat{T}(k, \cdot)$. The implementation below using the straddle maximum contour uncertainty heuristic to sequentially select training inputs x^n_k's. It starts with 30 initial $x^{1:N_0}_k$ inputs and adds another 90. All inputs are replicated with $r = 25$.

The GP kernel is Matern-5/2 with ARD, and we utilize the `hetGP` package to handle input-dependent simulation noise.

```
model2d$init.size <- 30   # initial design size
sob30 <- randtoolbox::sobol(55, d=2)  # Sobol space-filling design to initialize
sob30 <- sob30[ which( sob30[,1] + sob30[,2] <= 1) ,]
sob30 <- 25+30*sob30
model2d$init.grid <- sob30
model2d$pilot.nsims <- 1000
model2d$qmc.method <- NULL  # Use LHS to space-fill

model2d$batch.nrep <- 25  # replications
model2d$seq.design.size <- 120  # final design size -- a total of 3000 simulations
model2d$ei.func <- "smcu"  # straddle maximum contour uncertainty
model2d$ucb.gamma <- 1  # sMCU parameter

model2d$kernel.family <- "Matern5_2"
model2d$update.freq <- 5      # frequency of re-fitting GP hyperparameters
```

```
put2d.mcu.gp <- osp.seq.design(model2d, method="hetgp")
oos.mcu.gp <- forward.sim.policy( test.2d, nSteps.2d, put2d.mcu.gp$fit, model2d)
```

Adaptive batching

The idea of adaptive batching was explored in detail in Lyu et al. (2020) that proposed several strategies to construct $r^n(k)$ sequentially and is implemented in the `osp.seq.batch.design` function.

To implement adaptive batching one needs to specify the batch heuristic via `batch.heuristic` and the sequential design acquisition function $\mathcal{I}_n(\cdot)$ via `ei.func`. Below we apply the Adaptive Design with Sequential Allocation (ADSA) scheme. ADSA relies on the AMCU acquisition function and at each batching round either adds a new training input x_k^{n+1} *or* allocates additional simulations to existing $x^{1:n}(k)$. Below I employ ADSA with a heteroskedastic GP emulator (method set to `hetgp`). The other settings match the **osp.seq.design** solver from the previous section to allow for the best comparison.

```
model2d$total.budget <- 3000  # total simulation budget N

set.seed(110)
model2d$batch.heuristic <- 'adsa'  # ADSA with AMCU acquisition function
model2d$ei.func <- 'amcu'
model2d$cand.len <- 1000
put2d.adsa <- osp.seq.batch.design(model2d, method="hetgp")
oos.adsa.gp <- forward.sim.policy( test.2d, nSteps.2d, put2d.adsa$fit, model2d)
```

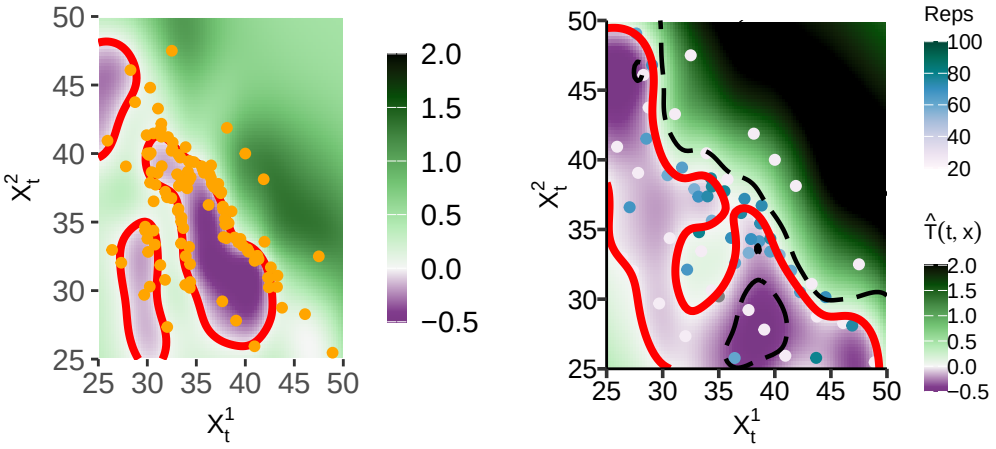

Figure 3: *Left*: Sequential design with sMCU. *Right*: Adaptive batching with ADSA. Both designs are at $k = 15$ ($t = 0.6$) for the 2D Basket Put example. Replication counts $r^n(k)$ are input-dependent and color coded in grayscale. The surface plots are color-coded according to $\widehat{T}(k, \cdot)$.

At $k = 15$ ADSA yields $\mathcal{D}_{15}$ with $N_{unique} = 65$ unique inputs and with replication counts ranging up to $r^n(k) = 79$, see right panel. Note that the geometry of $|\mathcal{D}_k|$ is very similar, but n is noticeably lower. Thus, adaptive batching allows to reduce the number of sequential design rounds and the associated computational overhead, running multiple times faster than a comparable **osp.seq.design** solver. The respective contract prices are 1.4403 for sMCU and 1.4401 for ADSA which are both excellent, however the running times of 48.41 and 14.1 minutes respectively, show that the ADSA-based solver is more than 4 times faster. `osp.seq.batch.design` yields the most (so far) time-efficient GP-based solver in dimensions higher than $d > 2$.

Next steps

The article on {mlOSP} by Ludkovski (2023) contains a multitude additional examples and sample code. One starting point are the max-Call payoffs, with benchmark 2D and 3D configurations given below:

```
testModel <- list()

testModel[[1]]<- list(dim=2,
                      K=100,
                      r=0.05,
                      div=0.1,
                      sigma=rep(0.2,2),
                      T=3, dt=1/3,
                      x0=rep(90,2),
                      sim.func=sim.gbm,
                      payoff.func= maxi.call.payoff)



testModel[[2]]<- list(dim=3,
                      K=100,
                      r=0.05,
                      div=0.1,
                      sigma=rep(0.2,3),
                      T=3, dt=1/3,
                      x0=rep(90,3),
                      sim.func=sim.gbm,
                      payoff.func= maxi.call.payoff)


# construct a test set. Recall that Option price = expected payoff over the test set.

set.seed(44)
all.tests <- list()
nSims <- rep(25000,2)
nSteps <- rep(20,2)

for (j in 1:2) {
  nSteps[j] <- testModel[[j]]$T/testModel[[j]]$dt
  test.j <- list()
  test.j[[1]] <- testModel[[j]]$sim.func( matrix(rep(testModel[[j]]$x0, nSims[j]), nrow=nSims[j], byrow=
  for (i in 2:nSteps[j])
    test.j[[i]] <- testModel[[j]]$sim.func( test.j[[i-1]], testModel[[j]])
  # European option price
  print(mean( exp(-testModel[[j]]$r*testModel[[j]]$T)*testModel[[j]]$payoff.func(test.j[[nSteps[j]]],tes
  all.tests[[j]] <- test.j
}
```

```
## [1] 6.612777
## [1] 9.581096
```

References

1. **mlOSP: Towards a unified implementation of regression Monte Carlo algorithms** by Mike Ludkovski.

2. **Stochastic kriging for simulation metamodeling** (2010) by Bruce Ankenman, Barry L Nelson, and Jeremy Staum.
3. **A regression-based smoothing spline Monte Carlo algorithm for pricing American options in discrete time** (2008) by Michael Kohler
4. **Adaptive Batching for Gaussian Process Surrogates with Application in Noisy Level Set Estimation** (2020) by Xiong Lyu and Mike Ludkovski

---

License