`mlOSP`: Towards a Unified Implementation of Regression Monte Carlo Algorithms

Mike Ludkovski[*]

December 3, 2020

Abstract

We introduce `mlOSP`, a computational template for Machine Learning for Optimal Stopping Problems. The template is implemented in the `R` statistical environment and publicly available via a `GitHub` repository. `mlOSP` presents a unified numerical implementation of Regression Monte Carlo (RMC) approaches to optimal stopping, providing a state-of-the-art, open-source, reproducible and transparent platform. Highlighting its modular nature, we present multiple novel variants of RMC algorithms, especially in terms of constructing simulation designs for training the regressors, as well as in terms of machine learning regression modules. At the same time, `mlOSP` nests most of the existing RMC schemes, allowing for a consistent and verifiable benchmarking of extant algorithms. The article contains extensive `R` code snippets and figures, and serves the dual role of presenting new RMC features and as a vignette to the underlying software package.

1 Introduction

Numerical resolution of optimal stopping problems has been an active area of research for more than two decades. Originally investigated in the context of American Option pricing, it has since metamorphosed into a field unto itself, with numerous wide-ranging applications and dozens of proposed approaches. A major strand, which is increasingly dominating the subject, is simulation-based methods rooted in the Monte Carlo paradigm. Developed in the late 1990s in [24] and [32] this framework remains without an agreed-upon name; we shall refer to it as Regression Monte Carlo (RMC). The main feature of RMC is its marriage of a probabilistic approach, namely simulation of the underlying stochastic state dynamics, and statistical tools for approximating the quantities of interest: the value and/or continuation functions, and the stopping region. This combination of simulation and statistics brings scalability, flexibility in terms of underlying model assumptions, and a vast arsenal of potential implementations. These benefits have translated into excellent performance which has made RMC popular both in the academic and practitioner (quantitative finance) communities. Indeed, in the opinion of the author, these developments can claim to be the most successful numerical strategy that emerged from Financial Mathematics. RMC has by now percolated down into standard Masters-level curriculum, and is increasingly utilized in cognate engineering and mathematical sciences domains.

Despite hundreds of journal publications addressing various variants of RMC (see e.g. the surveys [20, 28] and the monograph [7]), there remains a dearth of user-friendly benchmarks or unified overviews of the algorithms. In particular, to my knowledge, there is no `R` (or other common programming environments, such as `Python`) package for RMC. Small stand-alone versions exist in [33],[5] but are essentially unknown by the research community; see also [14] that has a broader scope and is primarily written in C++. One reason for this gap is the narrow focus of many research articles that tend to explore one small aspect of RMC and then illustrate their contributions on a small-scale, idiosyncratic example. This makes it hard to evaluate the relative pros and cons or to explore the merit of new ideas.

In the present article, I offer an algorithmic template for RMC, dubbed **mlOSP** (machine learning for optimal stopping problems), coupled with its implementation within `R`. This endeavour is meant to be an ongoing project, offering a platform to centralize and standardize new RMC approaches, as they are proposed. In particular, the associated `{mlOSP}` library [26] is currently in its Version 1.0 incarnation, with further (hopefully regular) updates to be fully expected.

1.1 Contributions

The **mlOSP** template below offers a unified description of RMC via the underlying statistical concepts. The aim of **mlOSP** is a *generic* RMC template that emphasizes the building blocks, rather than specific approaches. This perspective therefore tries to *nest* as many of numerous existing approaches as possible, shedding light on the differences and the similarities between them. To specify **mlOSP**, I utilize as much as possible the language of statistics (in contrast to the language of finance, or of probability). In particular, I attempt to place RMC in the context of modern statistical machine learning, highlighting such aspects as Design of Experiments, Simulation Device, and Sequential Learning.

By providing a templated perspective and tying disparate extant concepts, **mlOSP** allows for numerous new twists, extensions and variants to "classical" RMC. To this end, in this article and in the respective `R` code I discuss and showcase the following *new* RMC flavors:

 i) Multiple variants of space-filling training designs for each time-step of the RMC (Section 3.1);

 ii) Variable simulation design size N_k (Section 3.1);

 iii) Adaptive simulation designs with varying batch sizes generated sequentially (Section 3.3);

 iv) Cross-validated kernel regression emulators (Section 4.1);

 v) Relevance Vector Machine emulators (Section 4.1);

 vi) Heteroskedastic and local Gaussian Process emulators (Section 4.2).

As a further contribution, I moreover extend **mlOSP** to the context of multiple-stopping problems, designing an analogue for valuing Swing options (Section 6.1). To my knowledge, this is the first paper that applies concepts of simulation design and advanced statistical emulators to swing option pricing. This generalization hints at further possibilities of using **mlOSP** to develop an interlocking collection of computational tools for decision-making under uncertainty.

This article also contributes to the literature by providing a comprehensive benchmarking of different RMC schemes across solver types and problem instances. By providing fully reproducible results, I aim to initiate a common collection of testbeds (in the spirit of say the UCI ML repository http://archive.ics.uci.edu/ml) that allow for a transparent, apples-to-apples comparison and can be easily augmented by other researchers.

1.2 Organization

Section 2 presents the **mlOSP** template that is implemented in the eponymous `R` package `{mlOSP}` [26]. The template emphasizes the three pieces of the RMC framework: stochastic simulation, experimental design, and statistical approximation. These pieces are modularized and can be fully mixed-and-matched within the core backward dynamic programming loop. Sections 3 and 4 explore the two key pieces of **mlOSP**: experimental designs and regression emulators. In aggregate, Sections 2-4 serve as a "User Guide" to the package and contain many `R` code snippets to illustrate how to use `{mlOSP}` and interpret the results.

Section 5 benchmarks some of the implemented schemes, running 10 `{mlOSP}` solvers on 10 OSP instances. The latter include Bermudan options in 1-5 dimensions; the solvers range from piecewise linear regression, to RVM kernel regression, to Gaussian process emulators. It also includes additional experimental results related to scheme stability and fine-tuning various algorithm ingredients. Section 6 discusses extensions of the **mlOSP** template. Section 6.1 showcases a generalization to handle multiple-stopping problems, illustrated with Bermudan Swing Put options. In a different vein, Section 6.2 goes through the steps to define a new OSP instance within the package.

2 The mlOSP template

2.1 Regression Monte Carlo methodology

An optimal stopping problem (OSP) is described through two main ingredients: the state process and the reward function. We shall use $X = (X(t))$ to denote the state process, assumed to be a stochastic process indexed by the time t and taking values in $\mathcal{X} \subseteq \mathbb{R}^d$. The reward function $h(t, x)$ represents the net present value (in dollars) of stopping at time t in state x, where the notation emphasizes the common possibility of the reward depending on time, e.g. due to discounting. We assume the standard probabilistic structure of $(\Omega, \mathcal{F}, (\mathcal{F}_t), \mathbb{P})$, where X is adapted to the filtration $(\mathcal{F}_t)$, and $h(t, \cdot) \in L^2(\mathbb{P})$.

We seek the decision rule τ, a *stopping time* to maximize expected reward:

$$\mathbb{E}\left[h(\tau, X(\tau))\right] \to \max! \tag{1}$$

To this end, we wish to evaluate the **value function** $V : [0, T] \times \mathcal{X} \to \mathbb{R}$,

$$V(t, x) := \sup_{\tau \in \mathfrak{S}_t} \mathbb{E}\left[h(\tau, X(\tau))\mid X(t) = x\right], \tag{2}$$

where $\mathfrak{S}_t$ is the collection of all $(\mathcal{F}_t)$-stopping times bigger than t and less than T.

The state $(X(t))$ is typically assumed to satisfy a Stochastic Differential Equation of Ito type,

$$dX(t) = \mu(X(t))\, dt + \sigma(X(t))\, dW(t), \tag{3}$$

where $(W(t))$ is a (multi-dimensional) Brownian motion and the drift $\mu(\cdot)$ and volatility $\sigma(\cdot)$ are smooth enough to yield a unique strong solution to (3). We note that while we use $\mathbb{P}, \mathbb{E}$ to denote the probability and expectation operators, financially we are working under the risk-neutral "$\mathbb{Q}$"-measure that is the relevant one for contingent claim valuation.

For the remainder of the section we adopt the discrete-time paradigm, where decisions are made at K pres-specified instances $t_0 = 0 < t_1 < \ldots < t_k < t_{k+1} < \ldots < t_K = T$, where typically we have $t_k = k\Delta t$ for a given discretization step Δt. Henceforth, we index by k and work with $\mathcal{T} = (t_k)_{k=0}^K$.

It is most intuitive to think of optimal stopping as dynamic decision making. At each exercise step k, the controller must decide whether to stop (0) or continue (1), which within a Markovian structure is encoded via the action strategy $\mathcal{A} = (A_{0:K}(\cdot))$ with each $A_k(x) \in \{0, 1\}$. The action map A_k gives rise to the stopping region

$$\mathcal{S}_k := \{x : A_k(x) = 0\} \subseteq \mathcal{X},$$

where the decision is to stop, and in parallel defines the corresponding first hitting time $\tau_{A_{k:K}} := \inf\{\ell \geq k : A_\ell(X(\ell)) = 0\} \wedge K$ which is an optimal exercise time after k. Hence, solving an OSP is equivalent to classifying each pair $(x \in \mathcal{X}, \sqcup_\| \in \mathcal{T})$ into $\mathcal{S}_k$ or its complement the continuation set. Recursively, the action set A_k is characterized as

$$A_k(x) = 1 \quad \Leftrightarrow \quad \mathbb{E}[h(\tau_{A_{k+1:K}}, X(\tau_{A_{k+1:K}}))\mid X(k) = x] > h(k, x), \tag{4}$$

i.e. one should continue if the expected reward-to-go dominates the immediate payoff. Denoting the step-ahead conditional expectation of the value function by

$$q(k, x) := \mathbb{E}\left[V(k+1, X(k+1))|X(k) = x\right]. \tag{5}$$

and using the Dynamic Programming principle, we can equivalently write $A_k(x) = 1 \quad \Leftrightarrow \quad q(k, x) > h(k, x)$ because we have $V(k+1, X(k+1)) = \mathbb{E}\left[h(\tau_{A_{k+1:K}}, X(\tau_{A_{k+1:K}}))|X(k+1)\right]$. The q-value $q(k, \cdot)$ is known as the continuation value.

The above suggests a recursive construction of approximate $\widehat{A}_k$'s through iterating on either (5) or (4). Thus, the RMC framework generates functional approximations of the continuation values $\hat{q}(k, \cdot)$ in order to build $\widehat{A}_k(\cdot)$. This is initialized with $\widehat{V}(K, x) = h(K, x)$ and proceeds with the following loop:

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

3

i) Learn the q-value $\hat{q}(k, \cdot) \simeq \mathbb{E}\left[\widehat{V}(k+1, X(k+1)) \middle| X(k) = \cdot\right]$;

ii) Set $\widehat{A}_k(x) = 1_{\{\hat{q}(k,x) > h(k,x)\}}$;

iii) Set $\widehat{V}(k, x) = \max(\hat{q}(k, x), h(k, x))$.

The loop makes it clear that RMC hinges on a sequence of probabilistic function approximation tasks that are recursively defined. Note that the principal step i) of learning the continuation function can be represented in multiple ways: of approximating $x \mapsto \mathbb{E}\left[\widehat{V}(k+1, X(k+1)) | X(k) = x\right]$; or of approximating $x \mapsto \mathbb{E}\left[h(\tau_{\widehat{A}_{k+1:K}}, X(\tau_{\widehat{A}_{k+1:K}})) \middle| X(k) = x\right]$; or or generically by picking $1 \leq w \leq K - k$ and approximating

$$x \mapsto \mathbb{E}\left[h(\tau_{\widehat{A}_{k+1:k+w}}, X(\tau_{\widehat{A}_{k+1:k+w}})) + \widehat{V}(k+w, X(k+w)) 1_{\{\tau_{\widehat{A}_{k+1:k+w}} = k+w\}} \middle| X(k) = x\right].$$

Above, the look-ahead horizon $w \in \{1, \ldots, K - k\}$ allows to combines both pathwise rewards based on $\tau_{\widehat{A}}$ and the approximate value function $\widehat{V}$ w-steps into the future [13]. These choices are not numerically identical, because due to the approximation error $\widehat{V}(k+1, x) \neq \mathbb{E}\left[h(\tau_{\widehat{A}_{k+1:K}}, X(\tau_{\widehat{A}_{k+1:K}})) | X(k+1) = x\right]$.

As a final step, once all the action sets are computed, they induce the expected reward on the full horizon:

$$\widetilde{V}(0, x)[\widehat{A}_{0:K}] := \mathbb{E}\left[h(\tau_{\widehat{A}_{0:K}}, X(\tau_{\widehat{A}_{0:K}})) \middle| X(0) = x\right]. \tag{6}$$

The gap between $\tilde{V}(0, x)$ and $V(0, x)$ (which is the maximal possible expected reward) is the performance metric for evaluating $\widehat{A}_{0:K}$.

2.2 Workflow

The key step requiring numeric approximation in the above loop is learning the continuation function $q(k, \cdot)$ in i). In the RMC paradigm it is handled by re-interpreting conditional expectation as the mean response within a stochastic model. Thus, given an input x, there is a generative model $x \mapsto q(k, x)$ which is not directly known but accessible through a pathwise reward simulator $x \mapsto Y(x)$ where $Y(x)$ is a random variable with mean $\mathbb{E}[Y(x)] = q(k, x)$ and finite variance. The aim is then to *predict* the mean output of this simulator for an arbitrary x. Practically, this is done by running some simulations and then utilizing a statistical model to capture the observed input-output relationship. This statistical learning task can be broken further down into three sub-problems:

1. Defining the stochastic simulator;

2. Defining the simulation design;

3. Defining the regression step.

Recasting in the machine learning terminology, for sub-step 1 we need to define a Simulation Device that accepts an input $x \in \mathcal{X}$ (the state at time k) and returns a random sample $Y(x)$, which is a random realization of the pathwise reward starting at (k, x) such that $\mathbb{E}[Y(x)] = q(k, x)$. We have already discussed multiple versions of such simulators, cf. (4)-(5). In sub-step 2, we then need to decide which collection of x's should be applied as a training set. After selecting such Experimental Design of some size N, $x^{1:N}$, we collect the simulation outputs $y^{1:N} = Y(x^{1:N})$ and reconstruct the model

$$Y(x) = f(x) + \epsilon(x), \qquad \mathbb{E}[\epsilon(x)] = 0, \quad \mathbb{V}ar(\epsilon(x)) = \sigma^2(x), \tag{7}$$

where the inferred $f(x) \equiv \hat{q}(t, x)$.

The above 3 sub-steps constitute one iteration of the overall loop; the aggregate RMC is a **sequence** of tasks, indexed by k. With that in mind, the following remarks are helpful to guide the implementation:

- The RMC tasks are recursive, the sub-problem at step k is linked to the previous simulators/emulators at steps $\ell > k$. Therefore, approximation errors will back-propagate.

- While the tasks are inter-related, since they are solved one-by-one, there is a large scope for modularization, adaptation, etc., to be utilized —the solvers need not be identical across k's.

- When deciding how to train $\hat{q}$, there is no "data" per se and the controller is fully in charge of selecting what simulations to run. Judicious choice of how to do so is an important criterion of numerical efficiency.

- In classical machine learning tasks, there is a well-defined loss function that quantifies the quality of the constructed approximation $\hat{q}$. In OSP, this loss function is highly implicit; ultimately we judge algorithm performance in terms of the quality of the overall $\widehat{A}_{0:K}$ in (6).

- The stochasticity in RMC is deeply embedded in all the pathwise simulators and the dependence across time-steps. Understanding the sources of this stochasticity is critical for maximizing performance.

- The "error term" $\epsilon(x)$ in (7) is simulation noise coming from above. As a result, its statistical properties tend to be quite complex and non-Gaussian. For example, in the common scenario where the reward has a lower bound of zero $h(t,x) \geq 0$, $\epsilon(x)$ has a mixed distribution with a point mass.

Finally, we remark that like in any statistical model, for RMC we have a training set that is used to construct the **emulators** – the functional approximations of $\hat{q}$'s, and a test set, that is used to evaluate the quality of the resulting exercise rule $\hat{\tau}$.

2.3 Illustration with Toy Example

To illustrate the recursive construction of $\hat{q}$'s, we proceed with a toy example. We take the point of view of a buyer of a Bermudan Put. The contract has an expiration date T, a strike $\mathcal{K}$ and exercise frequency Δt (such as daily). The state $X(t)$ is interpreted as the *share price* of the asset at date t, with state space $\mathcal{X} = \mathbb{R}_+$ and is assumed to follow Geometric Brownian motion with constant coefficients. Taking into account the time-value of money, the reward function at time t is

$$h_{Put}(t,x) = e^{-rt}(\mathcal{K} - x)_+,$$

where r is the constant interest rate. With this setup, the stopping set $\mathcal{S}_k$ is known as the exercise region. For the Bermudan Put it is known that $\mathcal{S}_k = [0, \bar{s}_k]$, i.e. one should stop as soon as the asset price drops below the exercise thresholds $\bar{s}_k$.

We assume that there are $K = 5$ periods and illustrate how the RMC steps i)-ii)-iii) might be implemented at $k = 4$. Because we are at the penultimate step, the simulation device is simply a one-step-ahead sampling of the terminal payoff, i.e.

$$Y(x) := \left[\mathcal{K} - x \exp\left((r - \sigma^2)\Delta t + \sigma\sqrt{\Delta t}\xi\right)\right]_+, \tag{8}$$

where $\xi \sim \mathcal{N}(0,1)$ is the Brownian shock. For the Experimental Design we draw

$$x^n \sim LogNormal\left(40 \exp[(r - \sigma^2/2)4\Delta t], \sigma\sqrt{4\Delta t}\xi\right)$$

i.i.d., interpreted as the realized values at $k = 4$ obtained by simulating paths $x^n(0:K)$ of X starting with the fixed $X(0) = 40$. Specifically, let us take $N = 20$ inputs $x^{1:N}(k)$ at step 4. The corresponding $y^{1:N}(k)$ are then interpreted as *realized* future payoffs at $K = 5$ if one waits one more period. Because the ξ's inside $Y(x)$ are sampled in an i.i.d. manner, the y^n's are independent draws from the conditional payoff distribution given $X(4)$. (With a path-based perspective, these payoffs are simply $(\mathcal{K} - x^{1:N}(5))_+$ across independent paths of X.) The left panel of Figure 1 visualizes the resulting scatterplot of these input-output pairs $x^{1:N}(k), y^{1:N}(k)$.

Since $V(5,x) = h(5,x)$, the q-value at $k = 4$ is $q(4,x) := \mathbb{E}\left[(\mathcal{K} - X(5))_+ | X(4) = x\right]$, i.e. the expectation of $Y(x)$ conditional on x. To infer the continuation value function $q(4, \cdot)$ we apply regression, specifically we shall fit a quadratic function of the input x to predict the expected value of $Y(x)$. In other words we *postulate* that $\hat{q}(4,x) = \beta_0 + \beta_1 x + \beta_2 x^2$ or

$$Y(x) = \beta_0 + \beta_1 x + \beta_2 x^2 + \epsilon, \tag{9}$$

where the regression coefficients $\beta_{0:2}$ parametrize the unknown continuation function $q(4, \cdot)$ we are after, and $\epsilon \sim \mathcal{N}(0, \sigma_\epsilon^2)$ is assumed to be Gaussian. As is standard, we fit β's via Ordinary Least Squares, i.e by minimizing a quadratic penalty function which is the residual sum of squares between observed and predicted values. In `R` this achieved by the `lm` command as follows:

```
payoff[[4]] <- pmax(40-paths[[5]],0)   # outputs Y(x). Inputs are paths[[4]] (X(4))
rmc[[4]] <- lm(y ~ poly(x,2,raw=TRUE), data.frame(y= payoff[[4]], x = paths[[4]]))
```

Remark: It is instructive to compare the assumed (9) with the exact simulation device in (8). One of the core concepts of RMC is to abstract away from the "innards" of (8) and substitute with a statistical treatment as in (9).

The blue line in the right panel of Figure 1 is our resulting estimate of expected payoff at the terminal time step $K = 5$ given $X(4) = x$. Arithmetically, we have $\mathbb{E}[(\mathcal{K} - X(5))_+|X(4) = x] \simeq \hat{q}(4, x) := 79.107\ \text{-}3.349\ x+ 0.035\ x^2$. So for example $\hat{q}(4, 40)= 1.179$ and $\hat{q}(4, 36) = 3.929$. Proceeding with step ii) of the RMC loop, we may conclude that $\widehat{A}_4(40) = 1$ (continue at $X(4) = 40$) and $A_4(36) = 0$ (stop at $X(4) = 36$).

The $\simeq$ sign is because clearly the right-hand-side is an approximation: the conditional expectation is not *actually* quadratic, but since we do not know it is, we simply *model* it via that fitted quadratic function.

Note that the output of the regression is *not* any particular prediction, but the *object* $\hat{q}(4, \cdot)$, so that the corresponding decision rule $\hat{A}_4(\cdot)$ is implicit. In fact, RMC never requires explicitly characterizing $A_k(\cdot)$; all we need is to be able to evaluate $A_k(x)$ at any given x. In Figure 1 I solve for the stopping boundary $\bar{s}_4$ by simply searching at which threshold stopping becomes more profitable than continuing and shading the resulting stopping region $\mathcal{S}_4$. Obtaining $\bar{s}_4$ is purely "cosmetic" and plays no role in the solver.
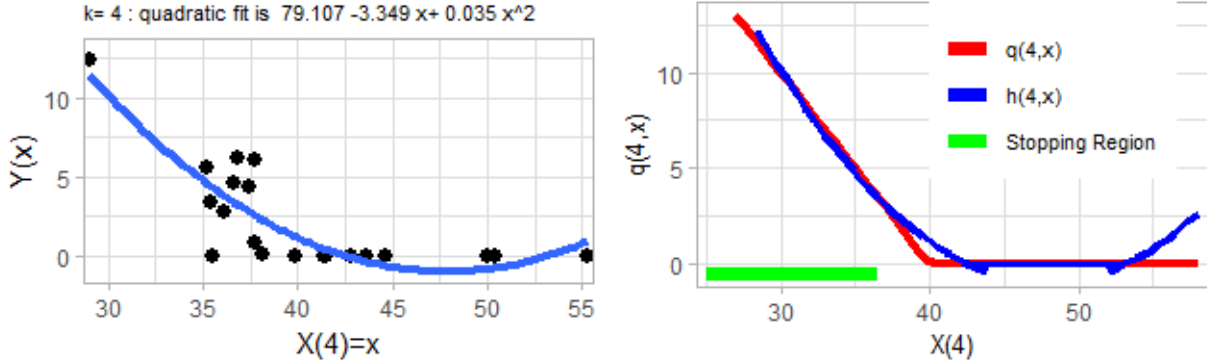

Figure 1: Left: Quadratic fit for the continuation value $q(k, \cdot)$ at $k = 4$. Right: Continuation value $q(k, \cdot)$, payoff $h(k, \cdot)$, value function $\widehat{V}(4, \cdot)$ and the stopping region at $k = 4$ for the Bermudan Put toy example.

The above completes one step of the RMC recursion, defining $\widehat{V}(4, \cdot)$ and (implicitly) $\widehat{A}_4(\cdot)$.

We can then continue with the earlier time-step $k = 3$ and so on. Figure 8 in Appendix B illustrates the sequential nature of RMC, by repeating the above logic over $k = 3, 2, 1$ and using (5), i.e. simulation of one-step-ahead value function to obtain $\hat{q}$'s. We further continue to utilize $N = 20$ training inputs sampled from a log-normal distribution and quadratic fits, though it should be clear to the user that these are rather arbitrary choices for the task at hand.

2.4 Dynamic Emulation Template

In order to abstract from the payoff function specifics, **mlOSP** operates with the *timing value*,

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

Exercising is optimal when the timing value is negative $A_k(x) = 1 \Leftrightarrow T(k, x) \geq 0$ and the stopping boundary is the zero contour of $T(k, \cdot)$. With these substitutions, one can replace $\hat{q}$'s with $\widehat{T}$'s everywhere. Note that given $\widehat{T}(k, x)$ we can recover the value function (and hence the option price) via $\hat{V}(k, x) = h(k, x) + \max\left(0, \widehat{T}(k, x)\right)$.

The other critical specification is the statistical definition of $\widehat{T}(k, \cdot)$. Normally it is taken to be the empirical minimizer in some space $\mathcal{H}_k$ of the mean squared error from the observations,

$$\widehat{T}(k, \cdot) = \arg\min_{f \in \mathcal{H}_k} \sum_{n=1}^{N_k} (f(x^n(k)) - y^n)^2. \tag{10}$$

For instance, in the linear model (aka `lm`) case, $\mathcal{H}_k = span(B_1, \ldots, B_{R-1})$ are the linear combinations of the basis functions $B_r(\cdot)$ and then (10) becomes a finite-dimensional optimization problem in terms of the R respective coefficients $\vec{\beta}$ with $\widehat{T}(k, x) = \beta_0 + \sum_{r=1}^{R-1} \beta_r B_r(x)$.

In addition, I employ the following notation:

- Time steps $k = 1, \ldots, K$, by default $t_k = k\Delta t$;

- $\mathcal{D}_k$: simulation design (can vary across time steps, so indexed by k);

- N_k: Size of $\mathcal{D}_k$ (can again vary across time steps);

- $t_k^{1:N_k}$ pathwise samples of timing value;

- $\mathcal{H}_k$: functional approximation space where $\widehat{T}(k, \cdot)$ is searched within;

- w: lookahead parameter that determines how long are the forward trajectories.

Algorithm 1 Dynamic Emulation Algorithm based on **mlOSP** template.

Require: $K = T/\Delta t$ (time steps), (N_k) (simulation budgets per step)
1: Generate design $\mathcal{D}_{K-1} := (\boldsymbol{X}^{(K)}(K-1))$ of size N_{K-1}
2: Generate one-step paths $x^{(K-1),n}(K-1) \mapsto x^{(K-1),n}(K)$ for $n = 1, \ldots, N_{K-1}$
3: Terminal condition: $t_K^n \leftarrow h(K, x^{(K-1),n}(K)) - h(K-1, x^{(K-1),n}(K-1))$ for $n = 1, \ldots, N_{K-1}$
4: **for** $k = K-1, \ldots, 1$ **do**
5:     Fit $\widehat{T}(k, \cdot) \leftarrow \arg\min_{f(\cdot) \in \mathcal{H}_k} \sum_{n=1}^{N_k} |f(X^{(k),n}(k)) - t_{k+1}^n|^2$
6:     Generate design $\mathcal{D}_{k-1} := (\boldsymbol{X}^{(k-1)}(k-1))$ of size N_{k-1}
7:     Generate w-step paths $x^{(k-1),n}(k-1) \mapsto x^{(k-1),n}(s)$ for $n = 1, \ldots, N_{k-1}$, $s = k, \ldots, k+w-1$
8:     Generate pathwise forward stopping rule $\tau_{k-1}^n = \min\{s \geq k : \widehat{T}(s, x^{(k-1),n}(s)) < 0\} \wedge k-1+w$
9:     Generate pathwise timing value

$$t_k^n = \begin{cases} h(\tau_{k-1}^n, x^{(k-1),n}(\tau_{k-1}^n)) - h(k-1, x^{(k-1),n}(k-1)) & \text{if} \quad \tau_{k-1}^n < k-1+w; \\ \widehat{T}(k-1+w, x^{(k-1),n}(k-1+w) + h(k-1+w, x^{(k-1),n}(k-1+w)) & \text{if} \quad \tau_{k-1}^n = k-1+w. \end{cases}$$

10: **end for**
11: Return fitted objects $\{\widehat{T}(k, \cdot)\}_{k=1}^{K-1}$

The two flexible steps in Algorithm 1 are generating the simulation design $\mathcal{D}_k$ and subsequently fitting the timing value emulator $\widehat{T}(k, \cdot)$. Note that the latter is a statistical model; it is viewed as an object (rather than say a vector of numbers) and passed as a "function pointer" to the pathwise reward simulator in subsequent steps. The latter simulator in turn applies a **predict** method to the fitted model, asking it to furnish a predicted timing value at arbitrary x. These two building blocks of RMC algorithms are explored in Sections 3 and 4 below.

To come back to the Simulation Device, the two most common approaches are TvR (for Tsitsiklis-van Roy [32]) and LS (for Longstaff-Schwartz [24]). In the TvR variant, $w = 1$ and the simulator generates one-step-ahead paths, resulting in the regression of $\widehat{T}(k+1, X(k+1))$ against $X(k)$. In the LS variant, $w = K - k$, and one regresses the pathwise timing values t_k^n against $X(k)$.

By default, the emulator is built only when the decision is non-trivial, i.e. in the in-the-money region $\mathcal{X}_{in} := \{x : h(k, x) > 0\}$. This insight dates back to [24] who noted that learning $\widehat{T}(k, x)$ is only necessary when $h(k, x) > 0$; otherwise it is clear that $T(k, x) > 0$ and hence we may set $\widehat{A}_k(x) = 1$.

2.4.1 Expected Payoff

By default, the output of an RMC algorithm is the approximate decision rules $\widehat{A}_k(\cdot)$. These are functions, taking as input the system state and returning the action to take (stop or continue). Additional computation is needed to evaluate the corresponding *expected payoff* $\tilde{V}$ of the stopping rule $\tau_{\widehat{A}_{0:K}}$ in (6). To do so, one again employs Monte Carlo simulation, approximating with a sample average on a set of forward test scenarios $x^{1:N'}(k), k = 1, \dots, K$,

$$\check{V}(0, X(0)) = \frac{1}{N'} \sum_{n'=1}^{N'} h(\tau^{n'}, x^{n'}(\tau^{n'})), \tag{11}$$

where $\tau^n := \min\{k \geq 0 : x^n(k) \in \widehat{\mathcal{S}}_k\}$ is the pathwise stopping time. Taking $x^{1:N'}$ to be a *fresh* out-of-sample set of scenarios, makes $\check{V}$ an unbiased estimator of $\tilde{V}$. This also yields a confidence interval for the expected payoff by using the empirical standard deviation of $h(\tau^{n'}, x^{n'}_{\tau^{n'}})$'s. Because (11) is a plain MC estimate , 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 designs $\mathcal{D}_k$ define training sets, while $x^{1:N'}$ forms the out-of-sample 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.

Remark: our RMC formulation returns the action map $\widehat{A}_k(\cdot)$ globally, or at least anywhere within the neighborhood of the training sets $\mathcal{D}_k$. As such, for forward simulation one may evaluate the expected payoff with arbitrary initial condition $X(0)$. Traditionally, the LS scheme was based on starting all training paths from a fixed $X(0)$ and was sensitive to "out-of-sample" testing with a different initial condition.

In the LS variant where training is done on paths-based $\mathcal{D}_k$, one additionally has access to an *in-sample* estimator $\hat{V}(0, X_0)$ which is the average payoff on the training paths. As is well known, $\hat{V}$ suffers from look-ahead bias and therefore cannot be reliably compared to the true option price. Heuristically, it has been observed that the in-sample estimator that is available in the global-path design approach tends to give *upper* bounds, and hence can be used to roughly "sandwich" the final estimate of the option price between the lower bound of the test set and the upper bound of the training set, "$V(0, X(0)) \in [\check{V}(0, X(0)), \hat{V}(0, X(0))]$".

2.4.2 What is Random?

The core of RMC is its probabilistic (hence Monte Carlo) nature. Indeed, Algorithm 1 is random and re-running it would yield a different set of functional approximations $\widehat{T}(k, \cdot)$ at each k, and ultimately a different $\check{V}(0, X(0))$. Why does this happen exactly?

In the **mlOSP** template, the immediate place where random samples are generated is during the forward simulation of trajectories (step 8) $x^{(k),1:N}(k : k + w)$. In other words, the "y"s in the regression indeed have a random component to them. Secondly, in the LS variant, the simulation design is random, i.e. the "x"s in $\mathcal{D}_k$ (step 7) are random. Namely, the training set is based on simulated trajectories $(X^{1:N}(k))$ which vary across algorithm runs. Statistically, this reflects the classical regression view of a random design in order to learn the population response. Third, note that different $\widehat{T}(k, \cdot)$ will induce different stopping times even on the same forward trajectories, so a second reason why y's vary is due to back-propagation which couples $\widehat{T}(k, \cdot)$ to $\{\widehat{T}(\ell, \cdot), \ell > k\}$, and therefore as soon as $\widehat{T}(k, \cdot)$ has some probabilistic aspect to it, so will all subsequent (preceding-in-time) emulators. Fourth, observe that many machine learning approaches involve nontrivial nonlinear optimization problems to fit the emulator in step 6 (i.e. minimizing in $\mathcal{H}_k$) and the low-level optimizers frequently invoke randomization. For example neural network and Gaussian process emulators rely on stochastic gradient descent and/or genetic optimization and will return not the true global minimizer but some local minimum in $\mathcal{H}_k$. Fifth, the test set, namely the out-of-sample collection of paths

$X^{1:N'}$ is random. To sum up, hiding all the algorithm details, and fixing the test set, $\check{V}(0, X(0))$ is a random variable, and so are all the intermediate solution outputs, such as $\widehat{A}_k(\cdot)$ or $\widehat{T}(k, \cdot)$. Similarly, the *budget* of the solver in terms of its running time and number of random variates generated is also random, being affected both by $\widehat{T}(k, \cdot)$'s and by the $(x^{(k),1:N})$'s.

With so many random pieces, scheme stability is a key piece of **mlOSP**. The standard way to measure stability is through sampling variance, i.e. the variance of $\check{V}(0, X(0))$ across independent algorithm runs, *holding the test set fixed*. More efficient algorithms would be expected to yield lower sampling variance, i.e. less dependence on the particular training set. Statistically, this dependence is primarily driven by the sensitivity of the regression emulator to the realizations in $t_k^{1:N}$, and thus it is preferable to "regularize" the regression architecture to avoid sensitivity to outliers, or other statistical mis-specifications.

2.5 Getting Started with `{mlOSP}`

The following user guide highlights the key aspects of the `{mlOSP}` package. It is intended to be fully reproducible, and the reader is invited to cut-and-paste (or download the RMarkdown document from GitHub) the `R` code. Since the schemes are intrinsically based on generating random outputs from the underlying stochastic simulator, where possible we fix the random number generator (RNG) seeds. Depending on the particular machine and `R` version, these seeds might nevertheless lead to different results.

The workflow pipeline in `{mlOSP}` consists of:

(i) Defining the `model`, which is a list of (a) parameters that determine the dynamics of $(X(t))$ in (3); (b) the payoff function $h(t, x)$; 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 (6). 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.

The aim of the above architecture is to maximize modularity and to simplify construction of user-defined OSP instances, such as additional system dynamics or bespoke payoffs. The latter piece is illustrated in Section 6.2.

The basic `{mlOSP}` solver is **osp.prob.design**. This is classical RMC that builds a simulation design using a forward simulation of state trajectories. To utilize a particular regression approach `osp.prob.design` has a `method` field which can take a large range of regression methods. Specifics of each regression are controlled through specifying respective `model` fields.

The following example illustrates this workflow. We consider a 1-D Bermudan Put with payoff $e^{-rt}(\mathcal{K} - x)_+$ where the underlying dynamics are given by Geometric Brownian Motion

$$dX(t) = (r - \delta)X(t)dt + \sigma X(t)dW(t), \qquad X(0) = x_0,$$

with scalar parameters r, δ, σ, x_0. Thus, $X(t_k)$ 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:

```
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
          look.ahead=1,pilot.nsims=0,batch.nrep=200,N=25)
```

As a representative solver, we use the {DiceKriging} Gaussian Process (GP) emulator with fixed hyper-parameters and a constant prior mean, selected through `method="km"`. The kernel family is Matérn-5/2, specified via `km.cov`, `km.var` above. For the training design we take $\{16, 17, \ldots, 40\}$ with 200 replications per site, yielding $N = |\mathcal{D}| = 200 \cdot 25$, see Section 3.1. The replications are treated using the SK method [2], pre-averaging the replicated outputs before training the GP.

```
train.grid.1d <- seq(16, 40, len=25)
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 $V(0, X(0))$. Indeed, we have not defined any test set, and consequently are momentarily postponing the computation of $\check{V}(0, X(0))$.

{mlOSP} has several solvers and allows straightforward swapping of the different pieces of the template. Below, for example, we swap the simulation design to be based on forward X-paths (achieved by utilizing the **osp.prob.design** solver) and replace the regression module with a smoothing cubic spline (**smooth.spline**; see [19] for a discussion of using splines for RMC). The latter requires specifying the number of knots. Only 2 lines of code are necessary to make this modification to the above OSP instance.

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

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.

Figure 2 visualizes the fitted models from one time-step. To do so, we predict the timing value 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 actually irrelevant in **mlOSP**, all that matters is the implied $\widehat{A}_k$ which is the zero level set of the timing value, i.e. depends on the *sign* of $\widehat{T}(k, \cdot)$. Therefore, the two regression methods yield quite similar exercise rules at this k, although they do differ (e.g. at $x = 35$ the spline-based rule suggests to continue, while the GP-based one suggests to stop and exercise the Put). To this end, 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 $T(k, \cdot)$ in that region is not statistically significant). A conclusion would be that more training is needed and the outputted $\widehat{\mathcal{A}}$ is probably not very accurate.

{mlOSP} is designed to be dimension-agnostic. 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 $h(t, x) = e^{-rt}(\mathcal{K} - (x_1 + x_2)/2)_+, x \in \mathbb{R}_+^2$. The two assets are assumed to be uncorrelated with identical dynamics, thus the whole solution should be symmetric in x_1, x_2. We continue to use a strike of $\mathcal{K} = 40$ and ATM initial condition $X(0) = (40, 40)$.

```
model2d <- list(look.ahead=1, 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)
```

With the model defined, we can use the same solvers as above, with exactly the same syntax. To illustrate a different regression module, we employ a linear model, `method="lm"`. To do so, we need to first define the basis functions that are passed to the `model$bases` parameter. Below we select polynomial bases of degree ≤ 2, $\mathcal{H}_k = span\{x_1, x_1^2, x_2, x_2^2, x_1 \cdot x_2\}$; by default the R `lm` model also includes the constant term, so there are a total of 6 regression coefficients $\vec{\beta}$.

```
bas22 <- function(x) return(cbind(x[,1],x[,1]^2,x[,2],x[,2]^2,x[,1]*x[,2]))
model2d$bases <- bas22
```
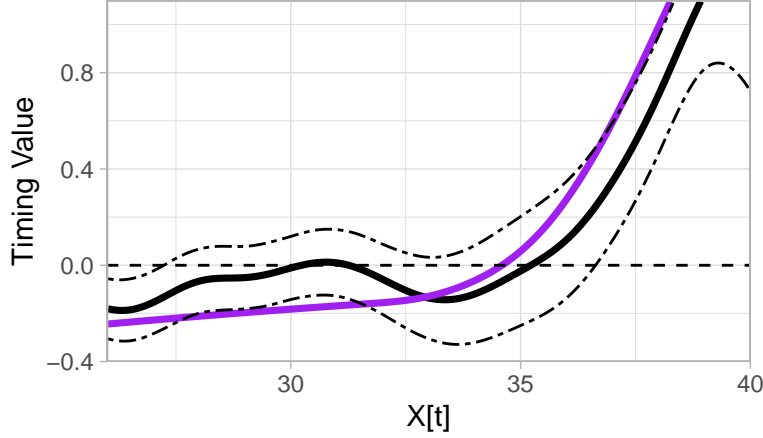
10

Figure 2: Timing value of a Bermudan Put based on GP emulator (black) and a Smoothing Spline emulator (purple). We also display the uncertainty quantification regarding the GP fit of $\hat{T}(k, x)$ (the dashed 95% band)

```
prob.lm <- osp.prob.design(15000,model2d, method="lm")
```

For comparison, we re-solve with a GP emulator that has a space-filling training design of $N_{unique} = 150$ sites replicated with batches of $N_{rep} = 100$ each for a total of $N = 15,000$ simulations again.

```
model2d$N <- 150   # N_unique
model2d$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
sob150 <- sob150[ which( sob150[,1] + sob150[,2] <= 1) ,]
sob150 <- 25+30*sob150  # Lower-left triangle in [25,55]x[25,55], see Fig

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

Figure 3 visualizes the estimated stopping boundary from the above 2 solvers. The underlying `plt.2d.surf` function plots a surface plot of the emulator of $T(k, \cdot)$ at a single time-step k using the `{ggplot}` toolbox. 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. In the left panel, the exercise boundary is a parabola because $\hat{T}(k, \cdot)$ is modeled as a quadratic. On the right panel, the exercise boundary has a much more complex shape since that $\hat{T}(k, \cdot)$ has many more degrees of freedom. For the latter case, `plt.2d.surf` also displays the underlying training design of $N_{unique} = 150$ sites.

2.5.1   Out-of-sample Tests

By default, the **osp.xxx** solvers only create the functional representations of the value function and do not return any explicit estimates of the option price or stopping rule. 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. This is the simulation device to generate a vector of $X(k + 1)$'s conditional on $X(k)$. By applying `payoff.func` to the final column (which stores values of $X(T)$) we can get an estimate of the European option price. By calling `forward.sim.policy` with a previously saved `{mlOSP}` object we then obtain a collection of $h(\tau^n, X^n(\tau^n)), n = 1, \ldots$, that can be averaged to obtain a $\check{V}(0, X_0)$.
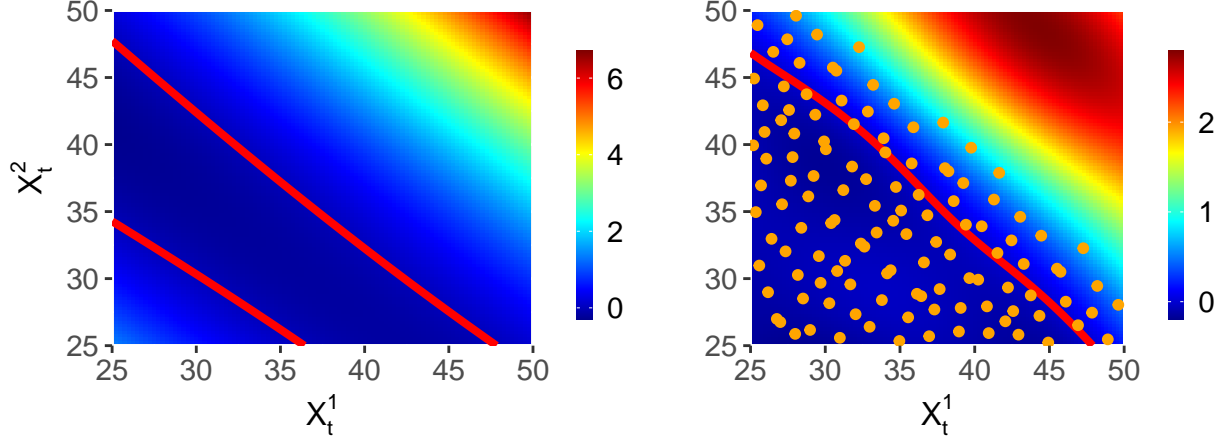
Figure 3: Timing Value of a 2D Basket Put. The contour shows the boundary of the stopping set (bottom-left corner). The colors indicate the value of $\hat{T}(k,x)$.

```r
nSims.2d <- 40000
nSteps.2d <- 25
set.seed(102)
test.2d <- list()
test.2d [[1]] <- model2d$sim.func( matrix(rep(model2d$x0, nSims.2d), nrow=nSims.2d, byrow=T),
                                    model2d, model2d$dt)
for (i in 2:(nSteps.2d+1))
    test.2d [[i]] <- model2d$sim.func( test.2d [[i-1]], model2d, model2d$dt)
oos.lm <- forward.sim.policy( test.2d, nSteps.2d, prob.lm$fit, model2d)
oos.km <- forward.sim.policy( test.2d, nSteps.2d, sob.km$fit,  model2d)
print( c(mean(oos.lm$payoff), mean(oos.km$payoff)) )  # estimates of check{V}(0,X(0))
```

```
## [1] 1.444948 1.437283
```

```r
# sanity check: estimated European option value
print(mean( exp(-model2d$r*model2d$T)*model2d$payoff.func(test.2d [[nSteps.2d]], model2d)))
```

```
## [1] 1.214149
```

Since we evaluated both estimators on the same set of test paths, we can conclude that the LM-Poly emulator leads to a better approximation $\check{V}^{LM}(0, X_0) = 1.444948 > 1.4372826 = \check{V}^{GP}(0, X_0)$ than the GP-km one. 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 $\check{V}^{EU}(0, X_0) - \mathbb{E}[h(T, X(T))|X(0) = X_0] \simeq \check{V}(0, X_0) - \mathbb{E}[h(\hat{\tau}, X(\hat{\tau}))|X(0) = X_0]$.

The `osp.prob.design` implements the Longstaff-Schwartz algorithm and also computes the in-sample estimator $\hat{V}(0, X(0))$. Moreover, it can evaluate in parallel in-sample and out-sample price estimates by splitting the total simulation budget between training and testing paths. This is controlled by the `subset` parameter. For example, we re-run the linear model from above but now using 15,000 training inputs and 15,000 test paths:

```r
ls.lm <- osp.prob.design(30000,model2d,method="lm",subset=1:15000)
```

```
## [1] "in-sample v_0 1.461778; and out-of-sample: 1.446481"
```

This is consistent with the previous discussion that $\hat{V}$ (in-sample) is biased high and $\check{V}$ (out-of-sample) is biased low compared to the true V.

Appendix C lists all the simulation, payoff and solver functions available in `{mlOSP}`.

3 Experimental Designs

The simulation designs $\mathcal{D}_k$ determine the training domain of the regression emulator. In `{mlOSP}` their specification is entirely within the responsibility of the user. While the size $N_k = |\mathcal{D}_k|$ of the training set obviously plays a major role in how accurate the approximation will end up, the *geometry* of $\mathcal{D}_k$ also has a significant impact. In a nutshell, the design geometry is specified through a *training density* $p_\mathcal{D}(t, \cdot)$, with x^n's viewed as samples from that target density

$$x^n \sim p_\mathcal{D}(t, \cdot). \tag{12}$$

In a randomized simulation design, this is precisely how samples are generated. Alternatively, one can generate deterministic representations of $p_\mathcal{D}(t, \cdot)$. For example, if $p_\mathcal{D} \equiv Unif[\tilde{\mathcal{X}}]$ for a given bounded input domain $\tilde{\mathcal{X}}$, then one can empirically replace sampling from $p_\mathcal{D}$ with placing x^n's on a lattice (i.e. grid which is a discrete-uniform target density). Alternatively, we can mimic a Uniform $p_\mathcal{D}$ using a Quasi Monte Carlo (QMC) sequence.

Uniform target densities are common because they correspond to space-filling simulation designs; the latter capture the intuition of learning through exploration, i.e. sampling a diverse set of x's in order to observe the corresponding y's. However, note that a Uniform $p_\mathcal{D}$ requires the user to specify the supporting $\tilde{\mathcal{X}}$.

Replication. Conventionally, a training design of size N consists of N unique sites $x^{1:N}$. In contrast, in a replicated design, all (some) sites appear multiple times. In a most common *batched* design, we have N_{unique} distinct sites, the so-called macro-design, and each unique x^n is then repeated N_{rep} times, so that

$$\mathcal{D} = \{\underbrace{x^1, x^1, \ldots, x^1}_{N_{rep} \text{ times}}, \underbrace{x^2, \ldots}_{N_{rep} \text{ times}}, x^3, \ldots, \ldots, x^{N_{unique}}\}, \tag{13}$$

where the superscripts now index *unique* inputs and the total simulation budget is $|\mathcal{D}| = N_{unique} \times N_{rep}$ The corresponding simulator outputs are denoted as $y^{1,1}, y^{1,2}, \ldots, y^{n,i}, \ldots, y^{N_{unique}, N_{rep}}$.

A replicated design allows to pre-average the corresponding y-values, $\bar{y}^n := \frac{1}{N_{rep}} \sum_{i=1}^{N_{rep}} y^{n,i}$, and then calling the regression module on the reduced dataset $(x^{1:N_{unique}}, \bar{y}^{1:N_{unique}})$. This is especially relevant for non-parametric regressions, where such pre-averaging cuts down on the regression overhead. For example, kernel-type regressions have runtime complexity $\mathcal{O}(N^3)$ and several orders-of-magnitude savings are achieved by training them on only $N_{unique} = N/N_{rep}$ samples rather than N. Clearly, pre-averaging involves loss of information, i.e. an additional approximation error to be incurred.

In the subsections below we describe the various design options that are possible within the **mlOSP** template. Aspects of the simulation design that can be varied include randomized or deterministic; replicated or not; sequential or one-shot. In `{mlOSP}` these choices are controlled through the selection of the top-level solvers, as well as `model` parameters. Most of these ideas are new for RMC training and to my knowledge have not appeared elsewhere except for some tangential mention in [25]. In aggregate they offer a lot of latitude for finetuning RMC solvers.

3.1 Space Filling Designs

Space-filling simulation designs are implemented within the `osp.fixed.design` solver which works very similarly to `osp.prob.design` except that it asks the user to explicitly build the $\mathcal{D}_k$'s. To do so, it has three key parameters: `input.domain` which determines the domain of approximation $\tilde{\mathcal{X}}$, `batch.nrep` which controls the replication amounts, and `method` which controls the regression method. Below we illustrate how these are used on a 2D Basket Put case study.

A space-filling design $\mathcal{D}$ aims to "uniformly" cover the domain of approximation $\tilde{\mathcal{X}}$. In `osp.fixed.design`, this can be done in several ways:

(i) One may directly specify a simulation design $\mathcal{D}$ to be used as-is. The latter can be explicitly chosen to be space-filling, for example through utilizing a lattice. In the next code snippet we generate a fixed lattice over a triangular $\tilde{\mathcal{X}}$ (namely the lower-left triangle since we only need to train in-the-money; based on model parameters a good domain of approximation is $\{x_1, x_2 : 25 \leq x_i \leq 55, x_1 + x_2 \leq 80\}$). We batch each of the resulting 136 inputs with 100 replications/site for a total simulation budget of $N = 13,600$. For the regression module we utilize a GP-km Matérn-5/2 emulator with pre-specified lengthscales.

```
lattice136 <- as.matrix(expand.grid( seq(25,55,len=16), seq(25,55,len=16)))
lattice136 <- lattice136[ which( lattice136[,1] + lattice136[,2] <= 80) ,]

put2d.lattice <- osp.fixed.design(model2d,input.dom=lattice136, method="km")
```

(ii) Second, one can specify a hyper-rectangular $\tilde{\mathcal{X}}$ and then space-fill it. By employing space-filling QMC sequences to place $x^{1:N_k}(k)$ this will yield a space-filling $\mathcal{D}$. This is achieved by providing the ranges for each coordinate x_i (activated if `length(input.domain)==2*model$dim`) and specifying the space-filling method; the latter rely on the {randtoolbox} package.

(iii) Directly specifying an approximation domain $\tilde{\mathcal{X}}$ might be non-obvious to an end user; as a solution {mlOSP} offers an automated 2-step procedure based on *pilot simulations*. The idea is to first (adaptively) construct a bounding hyper-rectangle, and then to space-fill it as above. The pilot simulations are simply forward trajectories $\tilde{x}^{1:N_{pilot}}(0:K)$ from the given initial condition $X(0)$ (their number controlled by `pilot.nsims`). $\tilde{\mathcal{X}}_k$ is then obtained in terms of *quantiles* of $\tilde{x}^{1:N_{pilot}}(k)$, with the latter acting as "scaffolding" to determine an appropriate size of the box. To illustrate, let us set `input.domain=0.04` and use Latin Hypercube Sampling (a randomized space filling method, as implemented in the {tgp} package, selected by `qmc.method=NULL`) to space-fill a rectangle between the 4th and 96th percentiles of the `pilot.nsim=1000` pilot trajectories at each time step. Observe that due to increasing variance of $X(k)$, the pilot trajectories "expand" as k grows, so the resulting $\tilde{\mathcal{X}}_k$ organically grows across the time-steps.

```
model2d$qmc.method <- NULL
model2d$N <- 400   # space-filling inputs to generate. Only in-the-money ones are kept
put2d.lhsAdaptive<- osp.fixed.design(model2d,input.dom=0.04, method="km")
```

Remark: in this setting, $\mathcal{D}_k$ is randomized and since we only keep in-the-money inputs, the resulting N_k is random. For example, in the above run we obtain $N_k = 125$ at $k = 12$, $N_{11} = 128$ and $N_{10} = 115$.

(iv) If one sets `input.domain=-1` then the full range of the pilot scenarios is used, making $\tilde{\mathcal{X}}_k$ larger.

Note that in options (ii)-(iv), $\mathcal{D}_k$ is generated separately for each k based on a given bounding hyper-rectangle. As such, one can straightforwardly vary the design size N_k. In particular, all the space-filling methods can accept arbitrary N_k in contrast to naive lattice designs that force N_k to be a product of marginal lattice lengths. In {mlOSP} this feature is achieved simply by making the `model$N` parameter a vector. One reason why variable N_k is useful is because the region of interest tends to grow over time due to the growing variance $\mathbb{V}ar(X(k)|X(0))$ and so more effort is generally needed to learn $A_k(\cdot)$ for k large. Moreover, backpropagation of approximation errors implies that the algorithm is more sensitive to errors in the middle/end time-steps than in the early ones. While conventional RMC schemes always use a constant N, this restriction is in fact completely superfluous once a non-path-based $\mathcal{D}_k$ is constructed.

In the code snippet below we employ `input.domain=-1` together with a different space-filling tool based on a Halton QMC sequence and a varying `model$N`. Note that the latter is not equal to N_k since the algorithm internally drops all out-of-the-money inputs.

```
model2d$qmc.method <- randtoolbox::halton
model2d$N <- c(rep(300,8), rep(500,8), rep(800,8)) # design size across time-steps
put2d.haltonRange <- osp.fixed.design(model2d,input.dom=-1, method="km")
```

(v) For completeness, let us mention briefly one further option supported by {mlOSP}. If `input.domain`

is omitted entirely, `osp.fixed.design` creates a path-based $\mathcal{D}_k = \tilde{x}^{(0),0:N}(k)$ directly from the pilot paths. This choice still embeds the replication aspect via `batch.nrep`, but otherwise is equivalent to `osp.prob.design`. Thus, `input.domain=NULL` generates a probabilistic, replicated design.

Figure 4 compares designs from option (i) with Lattice space-filling ($N_k = 120$); option (iii) with LHS space-filling ($N_k = 115$), and option (iv) with Halton-sequence space-filling ($N_k = 121$). The latter one has time-varying N_k and we additionally show $\mathcal{D}_k$ at $k = 20$ ($t = 0.8$), where $N_k = 181$.
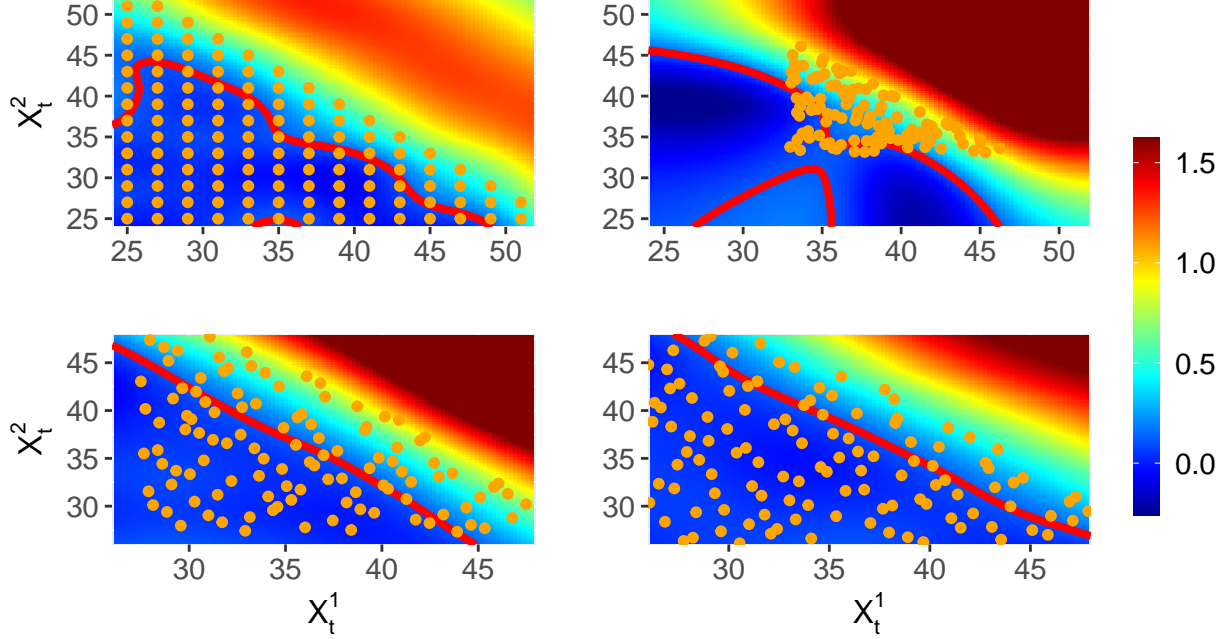

Figure 4: '{DiceKriging}' GP-km emulators with different space-filling simulation designs. Top row: lattice (left) and LHS (right) designs. Bottom row: Halton QMC sequence with time-varying design size: $k = 10$ ($t = 0.4$ left) and $k = 20$ ($t = 0.8$ right). As k increases the input domain $\tilde{\mathcal{X}}_k$ grows.

I end this section by comparing the above three solvers head-to-head in order to assess how the training domain affects the ultimate solution $\check{V}(0, X(0))$:

```
oos.1 <- forward.sim.policy(test.2d, nSteps.2d, put2d.lattice$fit,  model2d)
oos.2 <- forward.sim.policy(test.2d, nSteps.2d, put2d.lhsAdaptive$fit,  model2d)
oos.3 <- forward.sim.policy(test.2d, nSteps.2d, put2d.haltonRange$fit,  model2d)
print( c(mean(oos.1$payoff), mean(oos.2$payoff), mean(oos.3$payoff)) )
```

```
## [1] 1.425510 1.448362 1.444931
```

I note that the hard-coded lattice design does noticeably worse, in particular because its fixed domain of approximation is not adjusting to the time-dependent region of interest of the optimal stopping problem.

Remark: Yet another space-filling possibility was proposed in [[15] and consists in using QMC sampling to draw the ϵ's in the log-normal representation of $X(k)$. This keeps the target training density to be log-normal but simultaneously space-fills the (unbounded) region of interest.

3.2 Sequential Designs

The **osp.seq.design** solver generates simulation designs that are constructed sequentially. In other words $\mathcal{D}_k$ is constructed on-the-fly, gradually adding training samples $x^n(k)$ indexed by n. The goal of such adaptive $\mathcal{D}_k$ is to improve simulation efficiency through targeted placement of $x^n(k)$'s to maximize the learning of

$\widehat{T}(k, \cdot)$. Such *active learning* methods are common in machine learning applications. As discussed in [25], it is most efficient to select inputs around the exercise boundary, i.e. the region where the correct decision rule is most unclear.

To implement active learning, **osp.seq.design** selects $x^n(k)$ one-by-one, through greedily optimizing an *acquisition function* $x \mapsto \mathcal{I}_n(x)$ that is a proxy for the information gain for the respective input location x. Currently, five different acquisition functions are implemented, specified via the **ei.func** parameter: "smcu", "tmse", "sur", "csur", "amcu", cf. [8] and [25]. The acquisition function $\mathcal{I}_n(x)$ is also indexed by n and relies on the posterior uncertainty of the emulator $\widehat{T}^{(n)}(k, \cdot)$, and therefore requires working with a GP-type emulator, namely `method` being one of "km", "trainkm", "hetgp" or "homtp".

Below I illustrate with three representative instances of **osp.seq.design** that among them vary the acquisition function and the emulator type. **osp.seq.design** always starts with an initial design of given size n_0 (`init.size=30` below) and then runs until there are `seq.design.size` inputs. This means the sequential design involves `seq.design.size-init.size` rounds, during each of which $x \mapsto \mathcal{I}_n(x)$ is maximized to pick the next $x^n(k)$. Since each such round has nontrivial computational overhead, it is not tractable to have more than a couple hundred rounds. In order to still have thousands of training samples, we rely on *replication*, i.e. a simulation design of the form (13). In the examples below we start with $n_0 = 30$ space-filling design sites $x^{1:n_0}(k)$, and add an additional 120 with $N_{rep} = 20$ replications each, for a total $|\mathcal{D}_k| = 150 \cdot 20 = 3000$.

The first instance uses the Stepwise Uncertainty Reduction acquisition function `sur` and a `km` emulator.

```
model2d$init.size <- 30    # initial design size
sob30 <- randtoolbox::sobol(55, d=2)  # build a Sobol space-filling design to initialize
sob30 <- sob30[ which( sob30[,1] + sob30[,2] <= 1) ,]
sob30 <- 25+30*sob30
model2d$init.grid <- sob30


model2d$batch.nrep <- 25  # N_rep
model2d$seq.design.size <- 120   # final design size -- a total of 3000 simulations
model2d$ei.func <- "sur"  # Stepwise Uncertainty Reduction acquisition function
model2d$kernel.family <- "matern5_2"
model2d$km.cov <- c(15,15); model2d$km.var <- 1
put2d.sur.km <- osp.seq.design(model2d, method="trainkm")
oos.sur.km <- forward.sim.policy( test.2d, nSteps.2d, put2d.sur.km$fit, model2d)
```

The second instance takes the targeted mean-squared-error acquisition function `tmse` and a `hetGP` emulator.

```
model2d$ei.func <- "tmse"   # targeted mean-squared-error I_n(x)
model2d$tmse.eps <- 0.06    # tMSE parameter
model2d$kernel.family <- "Matern5_2"
put2d.tmse.hetgp <- osp.seq.design(model2d, method="hetgp")
oos.tmse.hetgp <- forward.sim.policy( test.2d, nSteps.2d, put2d.tmse.hetgp$fit, model2d)
```

The third combo utilizes the straddle maximum contour uncertainty (`sMCU`) for $\mathcal{I}_n(\cdot)$ together with a (homoskedastic) Student t-Process emulator.

```
model2d$ei.func <- "smcu"   # straddle maximum contour uncertainty
model2d$ucb.gamma <- 1   # sMCU parameter
model2d$kernel.family <- "Gaussian"
put2d.mcu.tp <- osp.seq.design(model2d, method="homtp") # homoskedastic TP
oos.smcu.tp <- forward.sim.policy( test.2d, nSteps.2d, put2d.mcu.tp$fit, model2d)
print( c(mean(oos.sur.km$payoff), mean(oos.tmse.hetgp$payoff), mean(oos.smcu.tp$payoff)) )
```

```
## [1] 1.422783 1.437392 1.440923
```

The resulting estimates $\check{V}(0, X(0))$ are similar across the three solvers, with the sMCU-TP combo slightly ahead (highest option value of 1.4409).

16

Figure 5 visualizes the resulting designs $\mathcal{D}_k$ (recall that all of them have 150 training sites) at the time step $k = 10$ (i.e. $t = 0.4$). Compared to space-filling designs in Figure 4, **osp.seq.design** yields highly non-uniform placement of $x^n(k)$'s. In all three instances we see that training sites are aggressively placed *around the stopping boundary.* This is especially noticeable for sMCU. SUR is more exploratory but still clearly concentrated along the at-the-money region. This adaptive behavior improves upon the space-filling designs that will often have many simulation sites far from the stopping boundary (e.g. deep in-the-money) and which are therefore not informative about the optimal stopping rule. We refer to [25] for a detailed analysis of sequential designs in RMC, including the trade-off between number of unique inputs N_{unique} and replication count N_{rep} (keeping $N = N_{unique} \cdot N_{rep}$ fixed). The primary take-away is that **osp.seq.design** is highly efficient in terms of keeping N low, but is rather slow due to shifting the work from running simulations to updating the emulator. Thus, we do not recommend that users run it with more than 150 or so sequential rounds.
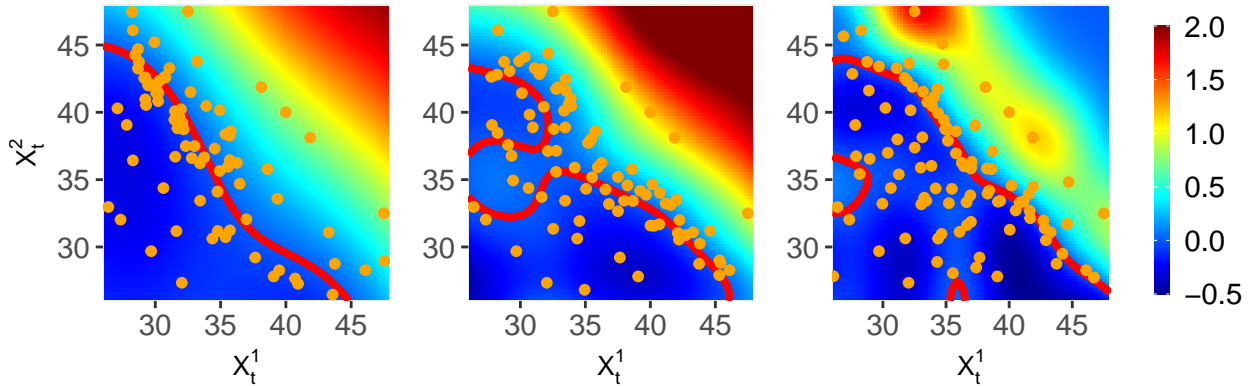

Figure 5: Sequential designs for the 2D basket Put using SUR (left) tMSE (middle) and sMCU (right) acquisition functions. Colors indicate respective timing value $\widehat{T}(k, x)$.

3.3 Deep Dive: Adaptive Batching

Conceptually, active learning will favor locations close to the exercise boundary where the correct decision rule is hardest to resolve. As a result, sequential designs will increasingly concentrate, i.e. the added $x^n(k)$'s cluster as n grows. *Adaptive batching* takes advantage of this by gradually increasing the replication level, in effect replacing clusters of x^n's with a single replicated input. This allows to reduce the number of unique inputs N_{unique} and speeds up the construction of a sequential design. In an adaptively batched design, the constant N_{rep} in (13) is replaced with input-dependent replication counts $r^n(k)$:

$$\mathcal{D}_k = \{\underbrace{x^1, x^1, \ldots, x^1}_{r^1(k) \text{ times}}, \underbrace{x^2, \ldots}_{r^2(k) \text{ times}}, x^3, \ldots, \ldots, x^{N_{unique}}\}, \tag{14}$$

where the algorithm now specifies both the unique inputs $x^1(k), \ldots$ and the respective $r^1(k), r^2(k), \ldots$. This idea was explored in detail in the recent preprint [27] that proposed several strategies to construct $r^n(k)$ sequentially and is implemented in the **osp.seq.batch.design** function. The latter works with a GP-based emulator and includes a choice of several *batching heuristics* that control how $r^n(k)$ is obtained:

- **fb**: Fixed Batching
- **mlb**: Multi-Level Batching
- **rb**: Ratchet Batching
- **absur**: Adaptively Batched SUR

- `adsa`: Adaptive Batching Design with Stepwise Allocation

- `ddsa`: Deterministic Batching Design with Stepwise Allocation

To implement adaptive batching we need to specify the batch heuristic via `batch.heuristic` and the sequential design acquisition function $\mathcal{I}_n(\cdot)$ as in Section 3.2 via `ei.func`. Below we apply Adaptive Design with Sequential Allocation, ADSA (which relies on the AMCU acquisition function) and Adaptive Batching with Stepwise Uncertainty Reduction, ABSUR. The GP emulators are trained using the default `DiceKriging::km` MLE optimizer (method set to `trainkm`). ABSUR jointly optimizes $x^{n+1}(k), r^{n+1}(k)$ while ADSA allows to either add a new training input $x^{n+1}(k)$ *or* allocate additional simulations to existing $x^{1:n}(k)$.

```
model2d$seq.design.size <- 100  # N_unique
model2d$batch.nrep <- 20        # Batch size N_rep
model2d$total.budget <- 2500  # total simulation budget N

model2d$kernel.family <- "gauss"  # GP kernel function: squared-exponential
model2d$update.freq <- 5        # frequency of re-fitting GP hyperparameters
model2d$r.cand <- c(20, 30, 40, 50, 60, 80, 120, 160) # parameter for ABSUR

set.seed(110)
model2d$batch.heuristic <- 'adsa'  # ADSA with AMCU acquisition function
model2d$ei.func <- 'amcu'
oos.obj.adsa <- osp.seq.batch.design(model2d, method="trainkm")

model2d$batch.heuristic <- 'absur'  # Adaptively Batched SUR
model2d$ei.func <- 'absur'
oos.obj.absur <- osp.seq.batch.design(model2d, method="trainkm")
```

The results are in Figure 6 which is in the same format and comparable to Figure 5. The left panel has $N_{unique} = 52$ unique inputs with replication counts ranging up to 115; the panel on the right has 54 $x^n(k)$'s with $r^n(k)$ up to 92. Note that the geometry of $|\mathcal{D}_k|$ is very similar to that in Figure 5, but N_{unique} 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.
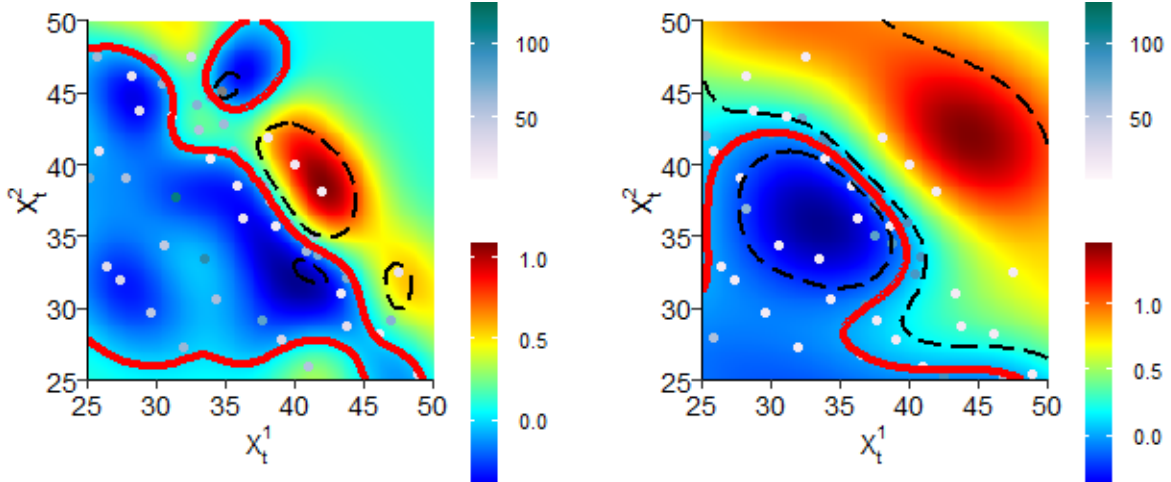

Figure 6: Adaptive batching with ADSA (left) and ABSUR (right) at $t = 0.6$ for the 2D Basket Put example. Replication counts $r^n(k)$ are input-dependent and color coded in grayscale. The surface plot itself is color-coded according to $\widehat{T}(k, \cdot)$.

4 Regression Emulators

The selection of the regression module is the most well known aspect of RMC. Over the past two decades, numerous proposals have been put forth to improve it beyond the original idea of ordinary least squares regression with user-specified basis functions. Among the choices for obtaining $\widehat{T}(k, \cdot)$ or equivalently $\hat{V}(k, \cdot)$ one can mention:

- Piecewise regression with adaptive sub-grids by Bouchard and Warin [9];

- Regularized regression, such as LASSO, by Kohler [21];

- Kernel regression by Belomestny et al. [6];

- Gaussian Process regression by Goudenege et al. [15], [16] and the author [25];

- Neural nets by Kohler et al. [22];

- Dynamic trees by Gramacy and myself [18].

With `{mlOSP}` I take advantage of the standardized `R` regression API to implement more than 10 such emulators. At its core, a regression module is based on the generic `fit` and `predict` methods and is otherwise fully interchangeable in terms of learning $\widehat{T}(k, \cdot)$. Because some of the regression packages do differ in the exact syntax of their **predict** method, e.g. have different ways of inputting new data, or return structures with different fields for the predicted mean response, manual adjustments are sometimes needed. Consequently, `{mlOSP}` implements the different regression variants "under the hood", controlled via the `method` option in the top-level solvers. The implementation allows for a straightforward swapping of regression methods, facilitating comparison and experimentation.

4.1 A potpourri of `{mlOSP}` regressions

The `method` field of the **osp.prob.design** solver supports the following regression modules:

- `lm`: linear model, specified through the set of user-provided basis functions, defined via the `bases` parameter;

- `loess`: local linear regression (only works in 1D and 2D), specified via `lo.span`;

- `rf`: random forest model, specified through the number of trees `rf.ntree` and `rf.maxnode`;

- `earth`: multivariate adaptive regression spline (MARS) model from the `{earth}` package, specified through degree `earth.deg`, number of knots `earth.nk` and backward fit threshold `earth.thresh`;

- `spline`: smoothing splines (only in 1D), specified via number of knots `nk`;

- `km`: Gaussian process model from the `{DiceKriging}` [30] package with fixed GP hyperparameters given through `km.var`, `km.cov` and `kernel.family`;

- `trainkm`: GP model trained via the MLE optimizer as in `{DiceKriging}`;

- `hetgp`: heteroskedastic Gaussian process regression based on the eponymous `{hetGP}` package from [8];

- `homtp`: homoskedastic t-Process regression also implemented in `{hetGP}`;

- `rvm`: relevance vector machine kernel regression from `{kernlab}`. The kernel is specified via `rvm.kernel`; if not specified uses `rbfdot` kernel by default;

- `nnet`: single-layer neural network specified via the number of neurons `nn.nodes`. Uses a linear activation function and the `{nnet}` package;

- `np`: kernel regression using the `{np}` package. The bandwidth is estimated using least squares cross-validation (default `npreg` option). The respective parameters are `np.kertype` (default "gaussian"); `np.regtype` (default "lc") and `np.kerorder` (default "2").

- **dynatree**: dynamic trees using {dynaTree} package that represents $\widehat{T}(k, \cdot)$ via a piecewise regression with adaptively generated spatial partitions. The hierarchical partitioning is similar to random forest, but utilizes a different Bayesian-inspired mechanism.

To my knowledge, most of the above choices are new, or at least partially new (for example there are multiple implementations of kernel regressions) in terms of their application to RMC. Rather than giving an exhaustive comparison of all their nuances, below we illustrate a few of the above regression modules using a 3D OSP instance of a max Call option

$$h_{\text{maxCall}}(t, x) := e^{-rt} \left(\max_{i=1,\dots,d} X_i - \mathcal{K} \right)_+$$

where the underlying assets follow i.i.d. GBMs. As in [1, (MS'04), Table 2 p. 1230] we take $\Delta t = 1/3$ with $T = 3$, i.e. $K = 9$ exercise dates. We take an OTM initial condition $X_0 = (90, 90, 90)$ with strike $\mathcal{K} = 100$. The true price of the max-Call is about $V(0, \mathbf{X}_0) = 11.25$ under continuous exercise optionality.

```
modelBrGl3d <- list(K=100, r=0.05, div=0.1, sigma=rep(0.2,3),T=3, dt=1/3,
  x0=rep(90,3),dim=3, sim.func=sim.gbm,payoff.func=maxi.call.payoff)
```

To allow comparisons between solvers, I generate a shared test set of 20,000 out-of-sample trajectories and report the resulting mean payoff on that same, fixed `test.3d`.

I start with the random forest (RF) regression emulator. RFs are known to be among the most robust (in terms of scalability, resistance to non-Gaussianity, etc.) emulators and generate piecewise constant fits obtained as an ensemble estimator based on a collection of hierarchical partition trees. See also [31] who investigated the use of regression trees for RMC. To specify a RF emulator requires inputting the number of trees (`rf.ntree`) and the maximum size of each tree terminal node (`rf.maxnode`) that are passed to the {randomForest} package.

```
modelBrGl3d$rf.ntree = 200   # random forest parameters
modelBrGl3d$rf.maxnode=200
call3d.rf <- osp.prob.design(100000,modelBrGl3d,method="randomforest")
oos.rf <- forward.sim.policy(test.3d,nSteps.3d,call3d.rf$fit,modelBrGl3d,compact=TRUE)
```

Next option is a neural network solver from {nnet}. The latter package builds a neural network with a single hidden layer with `nn.nodes` nodes (50 below) and the linear activation function.

```
modelBrGl3d$nn.nodes <- 50
call3d.nnet <- osp.prob.design(N=100000,modelBrGl3d, method="nnet")
oos.nnet <- forward.sim.policy( test.3d,nSteps.3d,call3d.nnet$fit,modelBrGl3d)
```

The next emulator is the Relevance Vector Machine (RVM) from the package {kernlab}. This is a kernel type regression, using `rvm.kernel` function, defaulting to the Gaussian radial basis kernel (`rbfdot`). Because RVMs are computationally intensive, we use a replicated design with 800 unique inputs and total simulation budget of $N = 20,000 = 800 \cdot 25$, i.e. $N_{rep} = 25$ replicates per site. This time we apply a space-filling design via `osp.fixed.design` solver.

```
modelBrGl3d$N <- 800   # N_unique
modelBrGl3d$batch.nrep <- 25 # N_rep
lhs.rect <- matrix(0, nrow=3, ncol=2) # domain of approximation
lhs.rect[1,] <- lhs.rect[2,] <- lhs.rect[3,] <- c(50,150)
modelBrGl3d$qmc.method <- randtoolbox::sobol  # space-filling using QMC sequence

call3d.lhsFixed.rvm <- osp.fixed.design(modelBrGl3d,input.domain=lhs.rect, method="rvm")
oos.rvm <- forward.sim.policy(test.3d, nSteps.3d, call3d.lhsFixed.rvm$fit, modelBrGl3d)
```

A different kernel regression package is {npreg}. Below I use it with a Epanechnikov order-2 kernel and local-constant regression `np.regtype=lc`. The bandwidth is estimated using least squares cross-validation (default **npreg** option).

```
modelBrGl3d$np.kertype <- "gaussian"
modelBrGl3d$np.kerorder <- 2
modelBrGl3d$np.regtype <- "lc"

call3d.sobFixed.np <- osp.fixed.design(modelBrGl3d,input.domain=lhs.rect, method="npreg")
oos.np <- forward.sim.policy( test.3d,nSteps.3d,call3d.sobFixed.np$fit,modelBrGl3d)
```

As a final comparator I consider a Gaussian Process emulator from {DiceKriging}. I employ the Matérn-5/2 covariance kernel with fixed lengthscales that are the same (isotropic) in all coordinates to reflect the symmetry of the underlying i.i.d. dynamics. I continue to use a replicated simulation design with $N_{unique} = 800$ training inputs with $N_{rep} = 25$ replicates per site.

```
modelBrGl3d$kernel.family <- "matern5_2"
# covariance function hyperparameters: process variance and lengthscales
modelBrGl3d$km.var=20; modelBrGl3d$km.cov=c(15,15,15)
set.seed(1)
call3d.lhsFixed.km <- osp.fixed.design(modelBrGl3d,input.domain=lhs.rect, method="km")
oos.km <- forward.sim.policy( test.3d,nSteps.3d,call3d.lhsFixed.km$fit, modelBrGl3d)
```

After having built all these models we can do horse racing on a fixed out-of-sample set of scenarios. I repeat this for 25 times each in order to record also the sampling standard error of each $\check{V}(0, X(0))$ estimator, which is a measure of the scheme's stability. The above sampling error reflects sensitivity to the training data, keeping all model tuning parameters and the test set fixed.

Table 1: Comparison of 3D max-Call solvers with different regression emulators. Running times based on a Surface Book laptop with 8GB memory and Inter Core i-5 2.6GHZ processor.

Emulator	Mean Price	Std. Error	Time (secs)
RF	11.098	0.0064	81.6
nnet	11.139	0.0056	462.5
npreg	9.010	0.0043	178.3
RVM	11.146	0.0069	41.9
GP-km	11.111	0.0071	93.3

According to Table 1, RVM and NeuralNet emulators perform best, yielding the highest out-of-sample payoffs. However, looking at running times, we see that NeuralNet is in fact the slowest of the bunch, taking more than 10x extra time compared to RVM, which is the fastest. In terms of standard errors, all emulators are quite close to another, with `npreg` (which performs worst) being the most stable. We note that direct comparisons are not straightforward since the different solvers utilized different training sets, i.e. N is method-specific. In Section 5.3 below, we carry out additional fine-tuning of the LM and neural-net solvers.

Finally, we mention that {mlOSP} also implements the TvR scheme which relies on the one-step-ahead value function $\hat{V}(k + 1, \cdot)$ instead of $h(\tau_{\widehat{A}_{k+1:K}}, \cdot)$ during the regression. This is available via the top-level solver `osp.tvr` that otherwise follows the exact same syntax as `osp.prob.design` and so can be mixed and matched with any of the above regression methods. Below we combine it with a MARS emulator. We set the degree to be `earth.deg=2`, so that bases consist of linear/quadratic hinge functions and allow up to 200 hinges.

```
earthParams <- c(earth.deg=2,earth.nk=200,earth.thresh=1E-8)  # passed to {earth}
call3d.tvr <- osp.tvr(N=80000, c(modelBrGl3d,earthParams), method="earth")
oos.tvr <- forward.sim.policy( test.3d,nSteps.3d, call3d.tvr$fit, modelBrGl3d )
print( mean(oos.tvr$payoff))
```

```
## [1] 10.93269
```

We see that in this example, TvR-MARS performs rather poorly, although it has a pretty fast running time of 2.17 minutes.

4.2 Deep Dive: Specifying bases for a linear model

The most extensively studied RMC approach follows the classical regression paradigm of a linear model with explicitly specified bases $B_1(\cdot), \ldots, B_R(\cdot)$. This means that the approximation space $\mathcal{H}_k$ has R degrees of fredom and $\widehat{T}(k, \cdot) \in span(B_1, \ldots, B_R)$. From a statistical perspective, any set of linearly independent bases will do, although for theoretical analysis one often picks special (orthogonal) basis families, such as Hermite polynomials. In {mlOSP} the user is able to specify any collection of lm bases, giving complete transparency on constructing $\mathcal{H}_k$.

As an illustration, below we provide definitions of polynomial bases of degree up to 2 (bas2) and up to degree 3 (bas3) for the above 3D example. In the latter case, we also append the payoff $h(t, x)$ to the set of bases.

It is well known that linear models are prone to overfitting, in part due to their sensitivity to the Gaussian homoskedastic noise assumption that is strongly violated in RMC. The skewed distribution of $\epsilon(x)$ and state-dependent simulation variance make the resulting fit biased and less stable compared to regularized regression approaches, such as MARS or RF. To avoid overfitting, below we use a large database of $N = 3 \cdot 10^5$ (300 thousand) paths. This example illustrates one of the trade-offs that must be considered for RMC implementations: namely the trade-off between speed (lm is effectively the fastest possible regression emulator) and memory (required very large N in order not to overfit). The memory constraint precludes scalability to very high dimension whereby the needed number of bases grows quickly.

The first expression below takes $\mathcal{H}_k = span(x_1, x_1^2, x_2, x_2^2, x_1 \cdot x_2, x_3, x_3^2, x_3 \cdot x_2, x_3 \cdot x_1)$ and similarly for all degree-3 polynomials.

```r
# polynomials of degree <= 2
bas2 <- function(x) return(cbind(x[,1],x[,1]^2,x[,2],x[,2]^2,x[,1]*x[,2],x[,3],x[,3]^2,
                                 x[,3]*x[,2],x[,1]*x[,3]))
# polynomials up to degree 3 + the payoff
bas3 <- function(x) return(cbind(x[,1],x[,1]^2,x[,2],x[,2]^2,x[,1]*x[,2],x[,3],x[,3]^2,
                                 x[,3]*x[,2],x[,1]*x[,3], x[,1]^3,x[,2]^3,x[,3]^3,
                          x[,1]^2*x[,2],x[,1]^2*x[,3],x[,2]^2*x[,1],x[,2]^2*x[,3],
                          x[,3]^2*x[,1],x[,3]^2*x[,2],x[,1]*x[,2]*x[,3],
                          maxi.call.payoff(x,modelBrGl3d)) )  # include the payoff


modelBrGl3d$bases <- bas2  # 10 coefficients to fit
lm.run2 <- osp.prob.design(300000,modelBrGl3d,method="lm")
oos.lm2 <- forward.sim.policy(test.3d, nSteps.3d, lm.run2$fit, modelBrGl3d)
modelBrGl3d$bases <- bas3  # 21 coefficients to fit
lm.run3 <- osp.prob.design(300000,modelBrGl3d,method="lm")
oos.lm3 <- forward.sim.policy(test.3d, nSteps.3d, lm.run3$fit, modelBrGl3d)
```

We find that the second estimator is significantly better, yielding $\check{V}(0, X_0) = 11.2504$ compared to 10.731 for the first one. In other words, having 21 bases is better than having only 10. At the same time it is also 2.5 times slower (15.67 vs 6.15 seconds).

One may straightforwardly apply more sophisticated bases, for example based on *order statistics* of $X(k)$ which is a common trick in the context of a max-Call payoff [10]. Below we sort x as $x_{(1)} \geq x_{(2)} \geq x_{(3)}$ and then work with $\mathcal{H}_k = span(x_{(1)}, x_{(1)}^2, x_{(1)}^3, x_{(1)}^4, x_{(2)}, x_{(2)}^2, x_{(3)}, x_{(1)} \cdot x_{(2)}, x_{(1)} \cdot x_{(3)})$.

```r
modelBrGl3d$bases <-function(x) {
  sortedx <- t(apply(x, 1, sort, decreasing = TRUE))  # sort coordinates in decreasing order
  return(cbind(sortedx[,1],sortedx[,1]^2, sortedx[,1]^3,sortedx[,1]^4, sortedx[,2],
            sortedx[,2]^2, sortedx[,3], sortedx[,1]*sortedx[,2], sortedx[,1]*sortedx[,3] ))
}
```

```
lm.run4 <- osp.prob.design(300000,modelBrGl3d,method="lm")
```

```
oos.lm.sorted <- forward.sim.policy(test.3d, nSteps.3d, lm.run4$fit, modelBrGl3d)
print(mean(oos.lm.sorted$payoff))
```

```
## [1] 11.27591
```

In other words, the sorted coordinates better convey information about $\mathcal{S}_k$ than the unsorted ones, yielding an excellent (highest among all solvers presented) $\check{V}(0, X(0))$. The above examples allude to the limitless scope for customizing $\mathcal{H}_k$ that is possible with **mlOSP**.

Another approach within the linear model regression framework is to build a piecewise-linear fit for $\widehat{T}(k, \cdot)$ that is defined in terms of partitions of the state space. `osp.probDesign.piecewisebw` function runs the Bouchard-Warin algorithm [9] which constructs $\mathcal{D}_k$ from forward trajectories like `osp.prob.design`, and then adaptively picks the regression sub-domains based on an equi-probable partition of the generated trajectories. Thus the sub-domains are empirically driven and in my implementation contain `model$nChildren` bins in each dimension. Below we take `nChildren=5` and $N = 3 \cdot 10^5$ which implies having $5^3 = 125$ sub-domains with 2400 trajectories in each. The final $\widehat{T}(k, \cdot)$ then has 500 coefficients to be fitted, based on running 125 degree-1 `lm` models across the partitions.

```
modelBrGl3d$nChildren <- 5
bw.run <- osp.probDesign.piecewisebw(300000,modelBrGl3d,test=test.3d)
```

```
## [1] "11.1583031516198  and out-of-sample  11.1085298160948"
```

4.3 Variants of GP Emulators

Gaussian Process emulators for RMC were originally proposed in [25] and also investigated in [15]. One of their benefits is uncertainty quantification of the resulting fit, enabling the use of sequential design via `osp.seq.design` and `osp.seq.batch.design`. Another potential advantage is their expressivity, i.e. ability to fit complex functions based on just a few training points. Thanks to these properties, GP emulators can provide highly accurate fits even with juts a few dozen (well-placed) training sites.

In {`mlOSP`} I implement five different types of GP emulators:

- `km` based on a pre-trained (i.e. with user-specified) hyperparameters;

- `trainkm` based on MLE-optimized GP hyperparameters as implemented in {`DiceKriging`};

- `homGP` which is essentially the same as `trainkm` but with a different MLE optimizer from the {`hetGP`} package;

- `lagp` based on local approximate GP regression proposed in [17] and implemented in {`laGP`};

- `hetgp` from [8] for heteroskedastic GPs.

The latter two choices are new in the RMC literature, and are proposed here for the first time due to their relevance for `mlOSP`, namely handling heteroskedasticity and spatial non-stationarity. The classical GP emulator assumes homoskedastic Gaussian simulation noise in (7) and learn its magnitude it as part of the MLE fitting. However, this is a poor assumption for RMC because the variance in pathwise rewards is highly state-dependent, i.e. heteroskedastic. As a partial solution, the GP-km and GP-trainkm emulators treat noise variance $\sigma^2(x) = \mathbb{V}ar(\epsilon(x))$ as known (rather than a parameter to be estimated). This is achieved through the utilizing a replicated design and relying on the Stochastic Kriging (SK) approach [2] that estimates $\sigma^2(x)$ empirically, Namely, we employ the classical MC variance estimator based on the batch of the N_{rep} pathwise rewards originating at x^n:

$$\hat{\sigma}^2(x^n) = \frac{1}{N_{rep} - 1} \sum_{i=1}^{N_{rep}} (y^{n,i} - \bar{y}^n)^2.$$

Thus, GP-trainkm uses $\hat{\sigma}^2(x^n)$ as a proxy for the true $\sigma^2(x^n)$. To be reliable, this strategy necessitates using large batch sizes N_{rep}; in practice we find that $N_{rep} \gg 20$ is necessary.

The alternative **hetGP** framework directly aims to learn $\sigma^2(\cdot)$ via a second spatial model that is jointly inferred with the model for the mean response. This has been recently implemented [8] in the {hetGP} package and available in {mlOSP} via method="hetgp".

```
modelBrGl3d$pilot.nsims <- 1000
modelBrGl3d$batch.nrep <- 60
modelBrGl3d$kernel.family <- "Matern5_2" # different naming compared to km

modelBrGl3d$N <- 500
modelBrGl3d$qmc.method <- randtoolbox::halton
put3d.hetgp <- osp.fixed.design(modelBrGl3d,input.dom=0.02, method="hetgp")
oos.hetgp <- forward.sim.policy(test.3d, nSteps.3d, put3d.hetgp$fit, modelBrGl3d)
print(round(mean(oos.hetgp$payoff),digits=4))
```

```
## [1] 11.1157
```

A further alternative is to replace a single GP emulator that assumes a global correlation/noise structure with a local GP fit, analogous to replacing a global linear model with LOESS regression. This is achieved via the laGP framework which adopts a prediction-focused approach that builds a new *sparse* GP every time a $\hat{T}(k, x)$ is needed. Those GPs are local, allowing to capture spatial non-stationarity and heteroskedasticity. Below we utilize lagp with a non-replicated design of 5000 unique inputs which would be prohibitively expensive with km.

```
modelBrGl3d$lagp.type="alcray"
modelBrGl3d$lagp.end=40
put3d.lagp <- osp.prob.design(7500,modelBrGl3d,method="lagp",subset=1:2500)
```

```
## [1] "in-sample v_0 12.373710; and out-of-sample: 10.922424"
```

5 Benchmarks

A key motivation for building the **mlOSP** template was to create transparent and verifiable benchmarks for RMC algorithms. While many published works contain detailed comparisons between a particular RMC version and competing approaches, these results are necessarily limited in scope and are often very difficult to reproduce. The numerous nuances that inevitably crop up when implementing RMC make such comparisons fraught, leading to a lack of consensus on what strategies are more efficient. Some of the challenges to benchmarking are:

- RMC algorithms tend to have a slew of tuning parameters that might significantly affect performance (from number of simulations, to the precise choice of the regression specification);

- One must define a complete problem instance, i.e. the payoff function, state dynamics, initial condition, etc. While some test cases have appeared repeatedly in numerous articles, there is no agreed-upon portfolio of benchmarks to apply. Many algorithms are of course very sensitive to the dimensionality of the problem, the geometry of the value function, the initial condition, etc.

- There are multiple metrics one could utilize to compare solution quality across algorithms, including accuracy at a fixed simulation budget, accuracy at a fixed computation time, running time at fixed budget. Again, there is no single consistent set of such criteria to fall back upon;

- The above metrics are heavily affected by the particular implementation, including the lower-level programming environment (say R vs C++), the machine running the benchmark, the operating system, etc.

- A key practical goal of benchmarking is to assess scalability of an RMC scheme, i.e. its ability to work

well for a wide range of OSP instances. However, a working definition of scalability is not easy as it relates to the multiple facets of simulation budget, running time, memory requirements, and of course accuracy, that all change nonlinearly as problems become more complex.

Due to all the above, the only way to create a "level playing field" for the different algorithms is to bring them all under one roof, coded side-by-side within a transparent computing environment. This is precisely what is achieved in {mlOSP}. In the companion software appendix, posted also on GitHub, I provide the RMarkdown code that can be run by any reader or end-user who downloads the {mlOSP} package to fully reproduce our results, figures and tables. To our knowledge, this is the most comprehensive verifiable set of RMC benchmarks (the StochOpt [14] also contains some benchmarks but primarily addresses more general stochastic control problems). The R code also makes completely transparent our chosen set of fully specified problem instances, which would be useful for other researchers in the future. Beyond providing an apples-to-apples comparison of different solvers, we also showcase in more depth the impact of (i) simulation budget; (ii) regression specification; (iii) experimental design specification; (iv) simulator type.

5.1 Benchmarked {mlOSP} instances

Below I present a list of 9 models and 10 solvers. All of them have appeared in previous articles. The OSP instances span a range of case studies (see Table 2 as well as Appendix D for a full specification) in terms of:

- Problem dimension d: 1D, 2D, 3D and 5D;

- Number of time-steps K from 9 to 50;

- Underlying dynamics, including Geometric Brownian motion and stochastic volatility;

- Option payoffs $h(t, x)$, including Puts, basket average Puts, max-Calls;

- Problem geometry: we consider both symmetric settings with i.i.d. assets (coordinates of $X(t)$), as well as asymmetric settings.

Table 2: Benchmarked OSP Instances

Model	dim	steps	Payoff	Dynamics	Notes	Ref
M1	1	25	Put	GBM	Classic 1D Put	[24]
M2	1	25	Put	GBM	Same as M1 but out-of-the-money	[24]
M3	2	25	Basket Put	GBM	2D symmetric Put	[24]
M4	2	9	Max Call	GBM	2D Call with dividends	[24]
M5	2	50	SV Put	SV Heston	Stochastic volatility model with a Put payoff	[29]
M6	3	9	Max Call	GBM	3D Symmetric max-Call	[10]
M7	5	9	Max Call	GBM	5D Symmetric Call	[10]
M8	5	9	Max Call	GBM	5D Asymmetric max-Call out-of-the-money	[10]
M9	5	20	Basket Put	GBM Cor	5D asymmetric correlated Put	[23]

The resulting benchmarks are reproducible (all RNG seeds fixed and provided) and utilize a shared out-of-sample test sets which are available upon request from the author, comprising more than 100Mb of data.

Remark: recently, several articles [15],[3],[4],[12] have explored ultra high dimensional (UHD) OSP instances, with as many as $d = 500$ dimensions. While mlOSP can certainly handle such case studies in principle, it is not currently geared for such use and I do not provide any related benchmarks, limiting to examples in dimension $d \leq 5$. First, handling high-dim. OSP requires different RMC architectures, for example advanced deep learning approaches; the solvers listed in Appendix A are generally inappropriate for $d \gg 5$. Second,

Table 3: Benchmarked Bermudan option prices for a given run. All methods share a common test set for each instance.

Model	S1-LM	S2-RF	S3-MARS	S4-TvR	S5-NNet	S6-BW	S7-GP	S8-ADSA	S9-Seq	S10-hetGP
M1	2.07	2.24	2.30	2.20	2.31	2.30	2.31	2.31	2.31	2.29
M2	1.01	1.04	1.09	1.07	1.09	1.10	1.10	1.10	1.10	1.09
M3	1.23	1.33	1.45	1.37	1.45	1.44	1.46	1.44	1.44	1.45
M4	21.48	21.21	21.39	20.21	21.44	21.36	21.41	21.40	21.32	20.66
M5	16.43	16.44	15.99	16.40	16.35	15.84	16.41	16.32	16.39	15.73
M6	11.15	11.04	11.13	10.85	11.05	11.01	11.06	10.96	11.12	9.90
M7	25.84	25.34	25.32	25.10	25.31	25.00	25.44	25.09	25.31	22.88
M8	11.81	11.51	11.65	11.74	11.69	11.53	11.39	11.10	11.60	10.76
M9	2.71	3.90	4.10	3.50	4.15	3.95	4.11	4.13	4.10	3.26

UHD schemes tend to rely on Python-based frameworks like `TensorFlow` which are less convenient to access via `R` and are not implemented in the present version of `{mlOSP}`. Third, UHD problem instances require a lot of fine-tuning (e.g. through exploiting specific problem structures and through experimenting with solver architectures) and therefore do not fit well into the envisioned automated nature of **mlOSP**. It remains an important open problem to extend the benchmarks to UHD case studies.

5.2 Benchmark Results

Table 3 presents the obtained option prices across the 9 model instances M1-M9 and the solvers S1-S10, see full description in Appendix A. I emphasize that the wide range of tuning parameters of each scheme makes comparisons fraught. For example, just for the linear model emulator, there have been dozens of proposals on how to pick the basis functions and the benchmarked S1-LM is not claimed to be ideal in any way. Therefore, rather than providing authoritative statements about relative performance, I make a few general observations. First, no single scheme clearly dominates in terms of accuracy. This might be expected and reflects the diversity of our problem instances. In 1D (M1-M2), the best performing are the GP methods (S7-S9) which are arguably an overkill for such a straightforward setting, where even PDE methods would work well. S3-MARS and S5-NNet also work very well. In 2D (M3-M5) the situation is similar, except that LM-poly also becomes competitive (and runs extremely fast). In 3D (M6), S1-LM wins on accuracy, followed still by S5-NNet and S3-MARS. In 5D (M7-M9), the best performing is S3-MARS and S8-ADSA. S1-LM does great for Max-Calls in M7-M8 but is terrible for the basket Put in M9.

A few model-solver combinations perform poorly. For instance, M1 with S1-LM is more than 10% below the best benchmark, which is very poor but not surprising since in that case we are attempting to capture the timing value function with just a cubic fit. Another poor choice is S10-hetGP for M7.

Table 4 lists the running times of each solver-model combination. Note the extreme range of the runtimes— from 2 to 3300 seconds. Multiple aspects come together to create this effect, from the choice of N (varies across solvers) to the regression overhead, to the number of time steps K. The solvers with simpler emulators, like S1-LM and S3-MARS run fastest, while complex GP emulators can be more than 100 times slower. The top-performing S5-NNet is also among the slowest. Of note, M5 has 50 time periods and therefore takes the longest to solve.

5.3 Deep dive: finetuning emulators

In this section we present two benchmarks that explore the role of tuning parameters. We concentrate on the M6 model which features the 3D symmetric GBM max-Call.

The left panel of Figure 7 displays the boxplots of out-of-sample estimates $\check{V}(0, X(0))$ using different `lm` settings and LM-Poly emulator. We vary the basis functions: quadratic ($R = 10$ total bases) or cubic ($R = 20$ bases), and the number of paths, running 50 macro-replications for each case. We note that for low N, there

Table 4: Benchmarked Algorithm Running Times (secs)

Model	S1-LM	S2-RF	S3-MARS	S4-TvR	S5-NNet	S6-BW	S7-GP	S8-ADSA	S9-Seq	S10-hetGP
M1	14.5	77.9	14.4	25.2	81.7	17.4	62.5	217.5	206.7	9.7
M2	13.6	50.6	10.5	24.7	42.3	20.6	24.6	214.2	208.4	9.7
M3	15.2	78.7	37.3	114.4	334.7	13.3	43.6	243.9	422.7	26.2
M4	2.0	37.3	24.5	35.4	210.3	2.2	53.7	178.0	521.8	25.6
M5	1.9	249.7	165.5	197.9	1731.6	12.3	1465.4	1593.4	577.4	338.2
M6	3.9	87.0	24.1	54.4	732.1	5.4	323.1	464.4	2023.8	223.8
M7	4.5	299.5	51.4	56.9	2103.1	31.4	1564.8	1670.9	2838.4	2881.7
M8	3.5	114.2	26.4	69.8	455.3	29.8	432.7	1363.8	2011.8	996.8
M9	32.9	348.8	43.3	97.4	1169.2	94.7	340.4	2597.6	3306.7	1037.3

is no gain from having more basis functions, highlighting likely overfitting by the degree-3 polynomials.

The right panel shows a similar fine-tuning for a Neural Network emulator where we vary the number of hidden units (`nodes`). We find that the emulator tends to significantly overfit for low number of paths (very large gap between in-sample and out-of-sample estimators) and that there is no gain from increasing number of hidden units beyond 50.

Detailed search for best tuning parameters in the spirit of the `{caret}` package is left for future research.
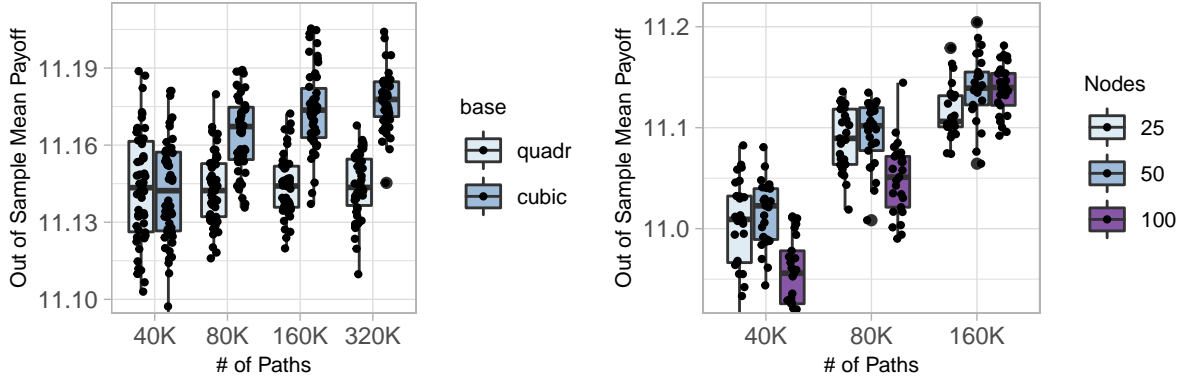

Figure 7: Left: Boxplots showing the joint impact of the number of basis functions R and the number of paths N for an 'lm' emulator. Right: boxplots showing the joint impact of the number of hidden units and the number of paths N for an 'nnet' emulator. Both panels consider the M6 3D Max-Call OSP instance.

6 Extensions

6.1 Valuing Swing Options

Swing options are common in commodity markets, such as natural gas or electricity. They are equivalent to a multiple-stopping problem or a bundle of regular Bermudan options. For example, in a Swing Put contract, the buyer has the right to exercise up to I Puts (e.g $I = 3$) prior to the swing maturity T. This means she has 3 distinct rights, and can dynamically choose to use them one-by-one. We also introduce the *refraction period* Δ which is the minimum amount of time that the holder must wait between two consecutive exercises.

To figure out the exercise strategy and ultimately the price of a Swing option, we need to keep track of the sequence of exercise times. Denote by $\tau^1, \tau^2, \ldots$ the ordered exercise times and by $V^{(i)}(t, x)$ the value of a swing option with i *remaining* exercise rights. Clearly, $V^{(1)}$ is simply the value of a standard Bermudan Put

option with the given strike $\mathcal{K}$ and maturity T. Considering the problem embedded in $V^{(2)}$, the holder has 2 rights remaining and at an instant t has two choices:

– Continue without exercising;

– Exercise one right, i.e. collect $(\mathcal{K} - X(t))_+$; she will then have $2 - 1 = 1$ rights remaining that is worth $V^{(1)}(\cdot, \cdot)$. Moreover, she cannot do anything until $t + \Delta$. Therefore, her overall payoff will be

$$h^{(2)}(t, X(t)) := (\mathcal{K} - X(t))_+ + e^{-r\Delta}\mathbb{E}\left[V^{(1)}(t + \Delta, X(t + \Delta))|X(t)\right],$$

where the first term is the payoff from exercising the Put, and the second term is the net present value of the remaining exercise right once the refraction period passes. We obtain a coupling where the payoff for $V^{(2)}$ depends on $V^{(1)}$, or more generally the payoff of $V^{(i)}$ depends on $V^{(i-1)}$.

More formally to price a Swing option we must solve the multiple-stopping problem (reverting back to discrete-time):

$$V^{(I)}(k, x) := \sup_{k \le \tau^1 < \tau^2 < \ldots < \tau^I \le K} \mathbb{E}\left[\sum_{i=1}^{I} h(\tau^i, X(\tau^i)) \,\middle|\, X(k) = x\right],$$

where each τ^i is a stopping time. It is possible that multiple exercise rights go unused (if the options remain out-of-the-money) in which case we set $\tau^i = \tau^{i+1} = \ldots = K$ for several τ's. The corresponding dynamic programming equation for $i = 1, \ldots$ is

$$q^{(i)}(k, x) = \mathbb{E}\left[V^{(i)}(k + 1, X(k + 1)) \,|\, X(k) = x\right] \tag{15}$$

$$V^{(i)}(k, x) = \max\left(h(k, x) + e^{-r\Delta}\mathbb{E}\left[V^{(i-1)}(k + k_\Delta, X(k + k_\Delta))|X(k) = x\right], q^{(i)}(k, x)\right), \tag{16}$$

with $V^{(0)} \equiv 0$, $V^{(i)}(K, x) = h(K, x) \forall i \ge 1$ and $k_\Delta = \Delta/\Delta t$. The first term in (16) is the expected payoff from exercising and observing a separation of k_δ discrete time-steps, and the second term is the q-value: expected payoff from waiting for one more time-step. Optimal swing times are recursively defined by $\tau^1 = \inf\{k : h(k, X(k)) > q^{(1)}(k, X(k))\}$ and

$$\tau^i = \inf\left\{k > \tau^{i-1} + k_\Delta \; : \; h(k, X(k)) + e^{-r\Delta}\mathbb{E}[V^{(i-1)}(k + k_\delta, X(k + k_\Delta))|X(k)] > q^{(i)}(k, X(k))\right\} \wedge K, \quad i = 2, \ldots. \tag{17}$$

Similarly I define the timing value with i exercise rights $T^{(i)}(k, x)$ which is the difference between the two terms in (16), and $\mathcal{S}_k^{(i)} = \{x : T^{(i)}(k, x) \le 0\}$ the stopping region when i rights remain.

Based on the above discussion, applying RMC to price swing options reduces to replacing the Bermudan payoff $h(k, x)$ with a functional that links $V^{(i)}(k, x)$ to $V^{(i-1)}(k + k_\delta, \cdot)$. This implies that solving for $V^{(i)}$ can be done **iteratively**, by starting with $V^{(1)}$ (which is a regular Bermudan Put) and then recursively solving for $V^{(2)}, V^{(3)}, \ldots$.

Algorithm 1 can now be directly applied by stacking all the regression objects in terms of i and re-defining the payoff structure. Note that although we need to compute multiple $V^{(i)}$'s, we can do so *in parallel* as part of the backward time-stepping over subscript k. Indeed, one can either proceed "layer-by-layer" (i.e. loop over k, then loop over i) or "step-by-step" (loop over i, then loop over k), since solving for $V^{(i)}(k, x)$ requires only knowing $V^{(i-1)}(k + k_\Delta, \cdot)$. In {mlOSP}, I implement the first variant with explicitly specified space-filling training designs in the `osp.swing.fixed` solver. The solver relies on the `swing.policy` function as the simulation device; the latter is called twice for each input $x^n(k)$, in order to compute a pathwise reward in the case of immediate exercise of one right at k, and in the case of no exercise and then acting optimally for steps $k + 1, \ldots, K$.

To illustrate the above, I present a short case study based on Carmona and Touzi [11] who considered a 1D Swing Put with GBM dynamics, $T = 1, X_0 = 100, \mathcal{K} = 100, r = 0.05, \sigma = 0.3$ with $\Delta t = 0.02$ or 50 time steps and refraction period of $\Delta = 0.1$ or $k_\Delta = 5$ time steps. They report results of $V^{(i)}(0, 100) = 9.85, 19.26, 28.802$ for $i = 1, 2, 3$ exercise rights. We see that $V^{(2)} \le 2V^{(1)}$ due to dis-economy of scale: the second swing is less

valuable than the first one due to the inherent refraction between exercises. Moreover, the second swing (first chronologically) will be exercised sooner, the intuition being that as there are fewer and fewer swings left, the holder gets more and more protective about using them.

In the example below, I compute the price of a $I = 3$-right Swing Put with above parameters. Using `swing.fixed.design` is very similar to `osp.fixed.design`, again requiring to specify a `model` list with all the parameters, the regression emulation method, and the training design specification. Two new model parameters are `n.swing` (number of swing rights) and `refract` (refraction period Δ). Below we use a smoothing spline emulator that furthermore requires specifying the number of knots `nk`. For the simulation design we space-fill using option (iii) from Section 3.1

```r
set.seed(10)
swingModel <- list(dim=1, sim.func=sim.gbm, x0=100,
            swing.payoff=put.payoff, n.swing=3,K=100,
            sigma=0.3, r=0.05, div=0,
            T=1,dt=0.02,refract=0.1,
            N=800,pilot.nsims=1000,batch.nrep=25)
swingModel$nk=16  # number of knots for the smoothing spline
spl.swing <- swing.fixed.design(swingModel,input.domain=0.03, method ="spline")
```

We then build a test set as before and evaluate the expected payoff $\check{V}^{(3)}(0, X(0))$ using the fitted `spl.swing` approximators $\widehat{T}^{(i)}(\cdot, \cdot)$ on the out-of-sample paths.

```r
set.seed(10); test.swing <- list()  # 25000 forward scenarios
test.swing[[1]] <- sim.gbm( matrix(rep(swingModel$x0, 25000),nrow=25000), swingModel)
for (i in 2:50)
  test.swing[[i]] <- swingModel$sim.func( test.swing[[i-1]], swingModel)

oos.spl3 <- swing.policy(test.swing,50,spl.swing$fit,swingModel,offset=1,n.swing=3)
mean(oos.spl3$totPayoff)
```

```
## [1] 27.59184
```

Note that the forward payoff evaluator `swing.policy` takes as input the number of swing rights, and we can similarly obtain that the value of having `n.swing=2` rights is 16.429 and of having a single right `n.swing=1` (which is equivalent to the standard Bermudan Put option with same parameters) is 9.33.

6.2 Building a New Model

The **mlOSP** template is highly extensible and accordingly {`mlOSP`} easily accommodates the construction of new OSP instances, allowing straightforward addition of new benchmarks. As an example of how the user can do so, I show the step-by-step process of implementing a new example based on the recent article by Cheridito et al. [3].

Specifically, that reference considers a multivariate GBM model with asymmetric volatilities and constant correlation. The payoff functional is of the max-Call type already described. The dynamics of asset S_i are

$$S_i(t) = s_i(0) \exp\left([r - \delta_i - \frac{\sigma_i^2}{2}]t + \sigma_i W^i(t) \right), \qquad i = 1, \ldots, d,$$

where the instantaneous correlation between W^i and W^j is ρ_{ij}. In the example of Becker et al. [3, Table 2, p16], $d = 5, s_i(0) = X_0, \delta_i = \delta, \rho_{ij} = \rho$ and the payoff is the max-Call.

To import the above into {`mlOSP`} we first define a new simulation function for the multivariate non-i.i.d. correlated Geometric Brownian motion using the `rmvnorm` function in the {`mvtnorm`} package. To avoid passing too many parameters, we introduce new `model` field `rho` (taken to be a constant) and re-define the volatility `sigma` to be a vector of length `model$dim`.

```
sim.corGBM <- function( x0, model, dt)
{    # build a matrix of rho*sigma_i*sigma_j, plus correct the diagonal to be sigma_i^2
     sigm <- model$rho*kronecker(model$sigma, t(model$sigma)) +
             (1-model$rho)*diag(model$sigma^2)

     newX <- x0*exp( rmvnorm(nrow(x0), sig=sigm*dt,
                             mean= (model$r- model$div- model$sigma^2/2)*dt) )
     return (newX)
}
```

Next, we construct the {mlOSP} model. In the referenced setting, the parameter values are $\sigma_i = 0.08i$, as well as $\delta = 10\%, r = 5\%, \rho = 0$ and contract specification $S(0) = 90, T = 3, \mathcal{K} = 100, \Delta t = 1/3$ ($K = 9$ time steps). I set up all the above and pass in the newly defined sim.corGBM as the simulation function.

```
modelBecker <- list(dim=5,sigma=0.08*(1:5), r= 0.05, div=0.1, rho=0,
                    x0 = rep(90,5), T=3, K=100, dt=1/3,
                    sim.func=sim.corGBM, payoff.func=maxi.call.payoff)
```

The package solver is now ready to be utilized. I select a hetGP emulator with Matérn-5/2 covariance and a space-filling design (of size $N_{unique} = 500$, with $N_{rep} = 100$ replicates per site or $N = 50,000$) which requires a few more parameter specifications:

```
hetGP.params <- list(max.lengthscale=rep(40,5),batch.nrep=100,
                    kernel.family="Matern5_2",pilot.nsims=1000,
                    look.ahead=1,N=500)
modelBecker <- c(modelBecker,hetGP.params)
beckerFit <- osp.fixed.design(modelBecker,input.dom=0.02, method="hetgp")
```

All in all, just a few lines of code are necessary to construct such a case study; once done it is trivial to tackle it using the wide range of presented solvers and schemes.

Running on an out-of-sample test set of 10^5 scenarios we obtain a 95% confidence interval for $\check{V}(0, X_0)$ of $[26.862, 27.262]$. This can be compared against the reported interval of $[27.63, 27.69]$ in [3]. We note the very high standard deviation of realized payoffs which leads to a wide credible interval of ± 0.2 of the reported $\check{V}(0, X_0) = 27.062$. Thus, obtaining tight bounds requires a very large out-of-sample test set.

7   Conclusion

With dozens of RMC implementations having been proposed in the literature, there remains a gap in bringing everything "under one roof" to allow proper and reproducible comparisons. The described {mlOSP} package does exactly that via a publicly available, GitHub-hosted, free software. Writing the package opened numerous new possibilities that I have explored above:

- New regression surrogates leveraging the unparalleled range of statistical methods implemented in R;

- New mix-and-match options thanks to the modular nature of **mlOSP** that allow many different combinations once a core template is defined;

- Transparent and reproducible benchmarks that offer apples-to-apples (in terms of the computing environment at least) comparison of different algorithm flavors;

- Transfer of the framework to related problems, such as multiple optimal stopping.

By design, the software library is a work in progress (and might have additional contributors in the future, which is trivial via the GitHub pull requests mechanism). Consequently, while the methodological presentation above will remain, the scope of the implementations will continue to evolve. Similarly, the precise behavior of the benchmarks is relative to the respective package versions, and will change as the utilized packages (such

as {DiceKriging} or {hetGP}) will be updated. As the original author and maintainer of the package, I always welcome feedback and requests for functionality to be added or bugs to be fixed.

Looking ahead, two projects for future work are to investigate more in-depth neural network solvers that have exploded in popularity recently, and to design self-tuning solvers that require as few tuning parameters as possible.

Appendix A: Benchmark Solvers

Solver	Regression Emulator	Solver & Simulation Design	Package
S1	Linear model emulator with polynomial basis functions up to degree 3 in 1-2 dimensions, else up to degree 2	`osp.prob.design` w/size N of $40K/100K$	`lm`
S2	MARS (multivariate adaptive regression splines) with degree=2: bases consist of linear/quadratic hinge functions; 100 knots.	`osp.prob.design` w/size N of $40K/100K$	{earth}
S3	Random Forest emulator with 200 trees and 100/200 nodes per tree	`osp.prob.design` w/size N of $40K/100K/200K$	{randomForest}
S4	Neural Net emulator with single-layer neural nets and 20/40/50 nodes	`osp.prob.design` w/size N of $40K/100K$	{nnet}
S5	MARS with degree=2 and 100 knots using the TvR scheme	`osp.tvr` w/size N of $40K/100K/200K$	{earth}
S6	Treed hierarchical equi-probable partitioning w/constant tree leaf fits	`osp.probDesign.piecewisebw` with `nChildren` of 8/5/4 and $N = 4000/100000/204800$ (5000 sites per leaf in 1D, 625 per leaf in 2D, 800in 3D, 200 in 5D)	Based on [9]
S7	GP-trainkm with Matérn-5/2 kernel	`osp.fixed.design` with replicated design with `batch.nRep=100` design size of $N_{unique} = 400/800/1000$ and LHS space-filling using pilot paths and `input.dom=0.02`	{DiceKriging}
S8	GP-trainkm with Matérn-5/2 kernel	`osp.seq.batch.design` with adaptive sequential batching via ADSA. The initial designs are space-filling; additional sites added using AMCU criterion up to a total of 200/250/500.	{DiceKriging}
S9	GP-trainkm with Matérn-5/2 kernel	`osp.seq.design` with adaptive sequential design via SUR criterion. Initial designs are Sobol sequences. `batch.nrep`, total design size and `cand.len` vary across models.	[25]
S10	GP-hetGP with Gaussian kernel	`osp.fixed.design` with replicated fixed design (pseudo-regression) based on Sobol sequences for ITM regions. Both `batch.nRep` and design size (100-1000) vary across models.	{hetGP}

Appendix B: Details for the Toy RMC Example of a 1D Bermudan Put

Figure 8 illustrates the full backward progression of an RMC solver for the toy 1-D Put example with 5 exercise periods from Section 2.3 . We see that in this case the earlier fits are problematic as we obtain that $\hat{q}(k, x) > h(k, x)$ and so the resulting stopping rule will *never* stop at $k = 2$. This is the direct result of picking a poor approximation space $\mathcal{H}_k$: the quadratic functions are unable to properly fit the true $q(k, x)$.
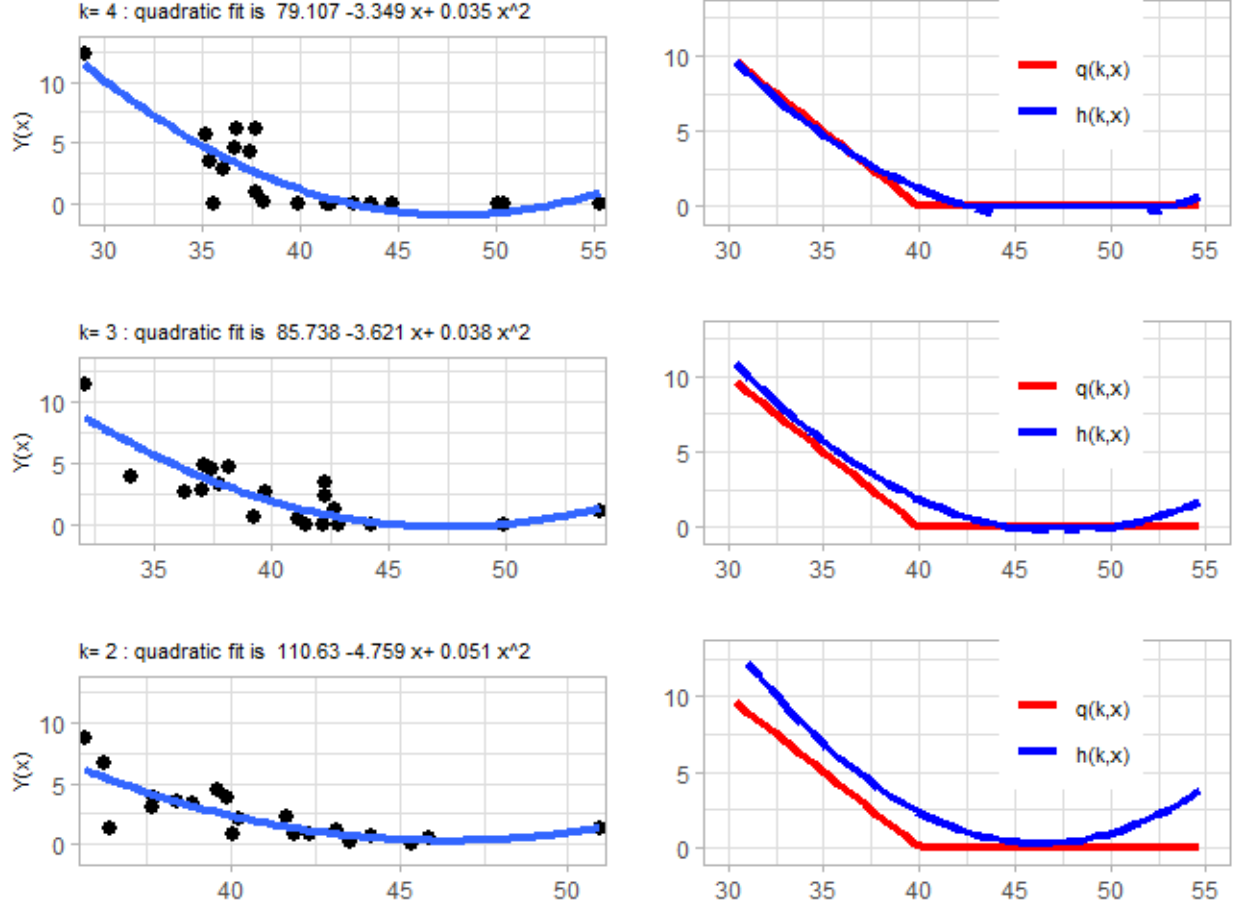
Figure 8: Illustration of the TvR RMC scheme with quadratic fits and 20 training inputs.

```
## [1] 2.43923
```

Appendix C.1: Simulation and payoff functions provided with {mlOSP}

Simulator	Description	Parameters
sim.gbm	Geometric Brownian Motion (GBM)	r (drift), sigma (volatility), div (dividend yield)
sim.gbm.cor	Correlated multi-d GBM with a single constant correlation parameter	rho

Simulator	Description	Parameters
`sim.gbm.matrix`	Correlated multi-d GBM specified via a covariance matrix	`sigma` (matrix)
`sim.ouExp`	Geometric Ornstein Uhlenbeck process	`alpha, meanRev, sigma`
`sim.expOU.sv`	1- or 2-factor Exponential O-U process with stochastic volatility (Vasicek)	
`sim.gbm.asian`	GBM model which also keeps track of the arithmetic and geometric averages of S_k	`r, sigma`
`sim.gbm.moving.ave`	GBM model which also keeps track of the lagged S_k to implement moving-average options	

Payoff	Description	Formula (`K` needed for all)
`put.payoff`	(basket) Put on the arithmetic average	$(\mathcal{K} - mean_i X_i)_+$
`call.payoff`	(basket) Call on the arithmetic average	$(mean_i X_i - \mathcal{K})_+$
`digital.put.payoff`	Digital Put on the geometric average	$\mathbb{1}_{\prod X_i^{1/d} < \mathcal{K}}$
`geom.put.payoff`	(basket) Put on the geometric average	$(\mathcal{K} - \prod X_i^{1/d})_+$
`mini.put.payoff`	Put on the minimum asset	$(\mathcal{K} - \min_i X_i)_+$
`maxi.call.payoff`	Call on the maximum asset	$(\max_i X_i - \mathcal{K})_+$
`sv.put.payoff`	Put within a Stoch Vol model (assumes that asset is the first coordinate)	$(\mathcal{K} - X_1)_+$

Appendix C.2: `{mlOSP}` Solvers

- `osp.prob.design` – the original Longstaff Schwartz scheme. Thus, the simulation design is randomized with the training density being the density of $X(k)$ and is constructed from forward paths $x^{1:N}$. The latter are re-used as the forward paths $X^{(k),n}(k:k+w)$. No replication is used. The solver supports a variety of regression methods.

- `osp.tvr` – the Tsitsiklis van Roy scheme. This is equivalent to `osp.prob.design` except that the regression is applied to the step-ahead value function, rather than pathwise payoffs.

- `osp.fixed.design` – mlOSP with a user-specified simulation design. The design can be replicated using the `batch.nrep` parameter. The typical usage is to create a space-filling design on a given hyper-rectangular domain of approximation $\tilde{\mathcal{X}}$. `osp.fixed.design` generates *fresh* forward paths $X^{(k),n}(k:k+w)$ at each time-step.

- `osp.probDesign.piecewisebw` – the Bouchard Warin implementation of RMC that utilizes a hierarchical (piecewise) linear model based on an equi-probable partition of the forward trajectory sites. The simulation design is the same as in `osp.prob.design`. This solver does not interface with `forward.sim.policy`; instead it directly accepts a collection of test trajectories that are evaluated in parallel with the backward DP iteration.

- `osp.seq.design` – sequential RMC with Gaussian Process-based emulators that are used to construct sequential design acquisition functions via the posterior emulator variance. Due to the overhead of adaptive simulation design, use of replication is strongly encourage.

- `osp.seq.batch.design` – sequential RMC with adaptive batching based on the heuristics in [27].

Appendix D: Benchmarked OSP Instances Specifications: M1-M9

7.0.1   1D

M1: 1D at-the-money Put from [24]

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

M2: 1D out-of-the-money Put

```
BModel[[2]] <- list(dim=1,
             sim.func=sim.gbm,
             K=40,
             payoff.func=put.payoff,
             x0=44,
             sigma=0.2,
             r=0.06,
             div=0,
             T=1,dt=0.04)
```

7.0.2   2D

M3: 2D Basket Put

```
BModel[[3]] <- list(dim=2,
                 K=40,
                 x0=rep(40,2),
                 sigma=rep(0.2,2),
                 r=0.06,div=0,
                 T=1,dt=0.04,
                 sim.func=sim.gbm,
                 payoff.func=put.payoff)
```

M4: 2D Max Call, in-the-money

```
BModel[[4]]<- list(dim=2,
                 K=100,
                 r=0.05,
                 div=0.1,
                 sigma=rep(0.2,2),
                 T=3, dt=1/3,
                 x0=rep(110,2),
                 sim.func=sim.gbm,
                 payoff.func= maxi.call.payoff)
```

M5: Put within a 1-factor Stochastic Volatility model from [29]

```
BModel[[5]] <- list(K=100,
                 x0=c(90, log(0.35)),
```

```
                    r=0.0225,div=0,sigma=1,
    T=50/252,dt=1/252,
    svAlpha=0.015,svEpsY=1,svVol=3,svRho=-0.03,svMean=2.95,
    eulerDt=1/2520, dim=2,
    sim.func=sim.expOU.sv,
    payoff.func =sv.put.payoff)

#putPr <- osp.probDesign.piecewisebw(40000,modelSV5)
 # get putPr$price= 16.81677
```

7.0.3 3D and above

M6: 3D Max Call from [1, Broadie and Cao [10]]

```
BModel[[6]]<- 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)
```

M7: 5D symmetric Max Call symmetric from [10]

```
BModel[[7]] <- list(dim=5,
                sim.func=sim.gbm,
                r=0.05,
                div=0.1,
                sigma=rep(0.2,5),
                x0=rep(100, 5),  # also 70, 130
                payoff.func=maxi.call.payoff,
                K=100,
                T=3,
                dt=1/3)

# paper_result=c(3.892,26.12,59.235) for S_0 = 70, 100, 130
```

M8: 5D asymmetric Max Call. Also X_0 adjusted to be OTM.

```
BModel[[8]] <- list(dim=5,
                sim.func=sim.gbm,
                r=0.05,
                div=0.1,
                sigma=c(0.08,0.16,0.24,0.32,0.4),
                x0=rep(70, 5),
                payoff.func=maxi.call.payoff,
                K=100,
                T=3,
                dt=1/3)

# paper_result=c(11.756,37.730,73.709) for S_0 = 70, 100, 130
```

M9: 5D Basket Average Put from [23]

```
BModel[[9]] <- list(dim=5,
                sim.func=sim.gbm.cor,
                r=0.05,
                div=0,
                sigma=0.2,
                x0=rep(100, 5),
                rho=0.2,
                K=100,
                payoff.func=put.payoff,
                T=3,
                dt=3/20) # 20 steps
#paper_result=4.254
```

M10: Moving Average Call from [10].

```
BModel[[10]] <- list(dim=10,
                sim.func=sim.gbm.moving.ave,
                r=0.05,
                div=0,
                sigma=0.2,
                x0=c(100,0,0,0,0,0,0,0,0,0),
                payoff.func=call.payoff,
                K=100,
                T=1,
                dt=0.02) # 50 steps

# paper_result=c(5.214,11.403,40.440) for S_0 = 90, 100, 130
```

References

[1] Leif Andersen and Mark Broadie. A primal-dual simulation algorithm for pricing multi-dimensional American options. *Management Science*, 50(9):1222–1234, 2004.

[2] Bruce Ankenman, Barry L Nelson, and Jeremy Staum. Stochastic kriging for simulation metamodeling. *Operations Research*, 58(2):371–382, 2010.

[3] Sebastian Becker, Patrick Cheridito, and Arnulf Jentzen. Deep optimal stopping. *Journal of Machine Learning Research*, 20:74, 2019.

[4] Sebastian Becker, Patrick Cheridito, Arnulf Jentzen, and Timo Welti. Solving high-dimensional optimal stopping problems using deep learning. *arXiv preprint arXiv:1908.01602*, 2019.

[5] Mikhail A. Beketov. *LSMonteCarlo: American options pricing with Least Squares Monte Carlo method*, 2013. URL https://CRAN.R-project.org/package=LSMonteCarlo. R package version 1.0.

[6] Denis Belomestny. Pricing Bermudan options by nonparametric regression: optimal rates of convergence for lower estimates. *Finance and Stochastics*, 15(4):655–683, 2011.

[7] Denis Belomestny and John Schoenmakers. *Advanced Simulation-Based Methods for Optimal Stopping and Control: With Applications in Finance*. Palgrave Macmillan, London, 2018.

[8] Mickael Binois, Robert B Gramacy, and Mike Ludkovski. Practical heteroscedastic Gaussian process modeling for large simulation experiments. *Journal of Computational and Graphical Statistics*, 27(4): 808–821, 2018.

[9] Bruno Bouchard and Xavier Warin. Monte-Carlo valorisation of American options: facts and new algorithms to improve existing methods. In R. Carmona, P. Del Moral, P. Hu, and N. Oudjane, editors, *Numerical Methods in Finance*, volume 12 of *Springer Proceedings in Mathematics*. Springer, 2011.

[10] Mark Broadie and Menghui Cao. Improved lower and upper bound algorithms for pricing American options by simulation. *Quantitative Finance*, 8(8):845–861, 2008.

[11] Rene Carmona and Nizar Touzi. Optimal multiple stopping and valuation of swing options. *Mathematical Finance*, 18(2):239–268, 2008.

[12] Yangang Chen and Justin WL Wan. Deep neural network framework based on backward stochastic differential equations for pricing and hedging American options in high dimensions. *Quantitative Finance*, pages 1–23, 2020.

[13] Daniel Egloff, Michael Kohler, and Nebojsa Todorovic. A dynamic look-ahead Monte Carlo algorithm for pricing Bermudan options. *The Annals of Applied Probability*, 17(4):1138–1171, 2007.

[14] Hugo Gevret, Nicolas Langrené, Jerome Lelong, Xavier Warin, and Aditya Maheshwari. *STochastic OPTimization library in C++*, 2018.

[15] Ludovic Goudenège, Andrea Molent, and Antonino Zanette. Variance reduction applied to machine learning for pricing Bermudan/American options in high dimension. *arXiv preprint arXiv:1903.11275*, 2019.

[16] Ludovic Goudenège, Andrea Molent, and Antonino Zanette. Machine learning for pricing American options in high-dimensional Markovian and non-Markovian models. *Quantitative Finance*, 20(4):573–591, 2020.

[17] R.B. Gramacy and D.W. Apley. Local Gaussian process approximation for large computer experiments. *Journal of Computational and Graphical Statistics*, 24(2):561–578, 2015.

[18] Robert B. Gramacy and Michael Ludkovski. Sequential design for optimal stopping problems. *SIAM Journal on Financial Mathematics*, 6(1):748–775, 2015. URL http://arXiv.org/abs/1309.3832.

[19] Michael Kohler. A regression-based smoothing spline Monte Carlo algorithm for pricing American options in discrete time. *Advances in Statistical Analysis*, 92(2):153–178, 2008.

[20] Michael Kohler. A review on regression-based Monte Carlo methods for pricing American options. In *Recent Developments in Applied Probability and Statistics*, pages 37–58. Springer, 2010.

[21] Michael Kohler and Adam Krzyżak. Pricing of American options in discrete time using least squares estimates with complexity penalties. *Journal of Statistical Planning and Inference*, 142(8):2289–2307, 2012.

[22] Michael Kohler, Adam Krzyżak, and Nebojsa Todorovic. Pricing of high-dimensional American options by neural networks. *Mathematical Finance*, 20(3):383–410, 2010.

[23] Jérôme Lelong. Pricing path-dependent Bermudan options using Wiener chaos expansion: an embarrassingly parallel approach. *arXiv preprint arXiv:1901.05672*, 2019.

[24] F.A. Longstaff and E.S. Schwartz. Valuing American options by simulations: a simple least squares approach. *The Review of Financial Studies*, 14:113–148, 2001.

[25] Michael Ludkovski. Kriging metamodels and experimental design for Bermudan option pricing. *Journal of Computational Finance*, 22(1):37–77, 2018.

[26] Mike Ludkovski. *mlOSP: Regression Monte Carlo Algorithms for Optimal Stopping*, 2020. R package version 1.0.

[27] Xiong Lyu and Mike Ludkovski. Adaptive batching for Gaussian process surrogates with application in noisy level set estimation. *arXiv preprint arXiv:2003.08579*, 2020.

[28] Selvaprabu Nadarajah, François Margot, and Nicola Secomandi. Comparison of least squares Monte Carlo methods with applications to energy real options. *European Journal of Operational Research*, 256(1):196–204, 2017.

[29] Bhojnarine R Rambharat and Anthony E Brockwell. Sequential Monte Carlo pricing of American-style options under stochastic volatility models. *The Annals of Applied Statistics*, 4(1):222–265, 2010.

[30] Olivier Roustant, David Ginsbourger, Yves Deville, et al. DiceKriging, DiceOptim: Two R packages for the analysis of computer experiments by kriging-based metamodeling and optimization. *Journal of Statistical Software*, 51(1), 2012.

[31] Stathis Tompaidis and Chunyu Yang. Pricing American-style options by Monte Carlo simulation: Alternatives to ordinary least squares. *Journal of Computational Finance*, 18(1):121–143, 2013.

[32] John Tsitsiklis and Benjamin Van Roy. Regression methods for pricing complex American-style options. *IEEE Transactions on Neural Networks*, 12(4):694–703, July 2001.

[33] Jeremy Yee. rlsm: R package for least squares Monte Carlo. *arXiv preprint arXiv:1801.05554*, 2018.