Bayesian Optimisation for Sequential Experimental Design with Applications in Additive Manufacturing

Mimi Zhang[1,4], Andrew Parnell[2,4], Dermot Brabazon[3,4], Alessio Benavoli[1]

[1]School of Computer Science and Statistics, Trinity College Dublin, Ireland
[2]Hamilton Institute, Maynooth University, Ireland
[3]School of Mechanical & Manufacturing Engineering, Dublin City University, Ireland
[4]I-Form Advanced Manufacturing Research Centre, Science Foundation Ireland

ABSTRACT

Bayesian optimization (BO) is an approach to globally optimizing black-box objective functions that are expensive to evaluate. BO-powered experimental design has found wide application in materials science, chemistry, experimental physics, drug development, etc. This work aims to bring attention to the benefits of applying BO in designing experiments and to provide a BO manual, covering both methodology and software, for the convenience of anyone who wants to apply or learn BO. In particular, we briefly explain the BO technique, review all the applications of BO in additive manufacturing, compare and exemplify the features of different open BO libraries, unlock new potential applications of BO to other types of data (e.g., preferential output). This article is aimed at readers with some understanding of Bayesian methods, but not necessarily with knowledge of additive manufacturing; the software performance overview and implementation instructions are instrumental for any experimental-design practitioner. Moreover, our review in the field of additive manufacturing highlights the current knowledge and technological trends of BO.

Index Terms: Batch optimization, Constrained optimization, Design of experiments, Discrete variables, Multi-fidelity, Multi-objective.

1 Introduction

Engineering designs are usually performed under strict budget constraints. Collecting a single datum from computer experiments such as computational fluid dynamics can potentially take weeks or months. Each datum obtained, whether from a simulation or a physical experiment, needs to be maximally informative of the goals we are trying to accomplish. It is thus crucial to decide where and how to collect the necessary data to learn most about the subject of study. Data-driven experimental design appears in many different contexts in chemistry and physics (e.g. Lam et al., 2018) where the design is an iterative process and the outcomes of previous experiments are exploited to make an informed selection of the next design to evaluate. Mathematically, it is often formulated as an optimization problem of a black-box function (that is, the input-output relation is complex and not analytically available). Bayesian optimization (BO) is a well-established technique for black-box optimization and is primarily used in situations where (1) the objective function is complex and does not have a closed form, (2) no gradient information is available, and (3) function evaluations are expensive (see Frazier, 2018, for a tutorial). BO has been shown to be sample-efficient in many domains (e.g. Vahid et al., 2018; Deshwal et al., 2021; Turner et al., 2021). To illustrate the BO procedure, consider the (artificial) problem in additive manufacturing (AM), where we want to optimize the porosity of the printed product with respect to only one process parameter (e.g. laser power). Figure 1 demonstrates the iterative experiment-design loop, where the upper panels represent performing BO-suggested experiments, and the lower panels represent data-driven design selection.

AM technologies have been developed in the past few years from rapid prototyping to mass scale production-ready approaches. A common AM technique: selective laser melting (aka powder bed fusion), is schematically shown in Figure 2. This process involves spreading a thin layer of loose powder over a platform, and then a fibre laser or electron beam tracing out a thin 2D cross-sectional area of the part, melting and solidifying the particles together. The platform is then lowered, and another powder layer is deposited. The process is repeated as many times as needed to form the full 3D part. The main challenge against the continued adoption of AM by industries is the uncertainty in structural properties of the fabricated parts. Taking metal-based AM as an example, in addition to the variation in as-received powder characteristics, building procedure and AM systems, this challenge is exacerbated by the many involved process parameters, such as laser power, laser speed, layer thickness, etc., which affect the thermal history during fabrication. Thermal history in AM process then affects surface roughness and microstructural,
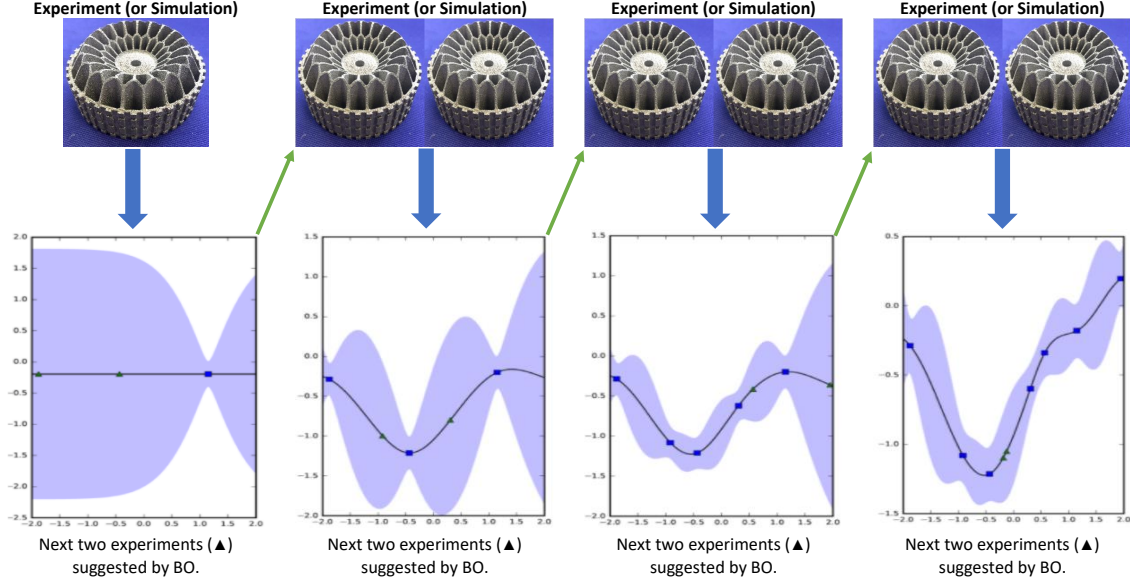
Figure 1: Given preliminary and tentative experimental data, BO will learn a model and decide the next experiment or batch of experiments. After new experimental data are available, BO will update the model and again decide the next experiment or batch of experiments. The procedure iterates until the resource budget is spent. In the above artificial AM example, we initially have one experimental output (indicated by ■); after learning a model (the straight line), BO suggests the next two experimental trials (indicated by ▲). After the two products are printed, BO updates the model (into a curve) according to the three available experimental outputs (indicated by ■ in the 2nd bottom panel), and then suggests another two experimental trials (indicated by ▲ in the 2nd bottom panel). The procedure thus iterates by performing suggested experiments, updating the model, and making new suggestions. The light-blue shaded bands in each bottom panel represent the prediction interval (i.e. uncertainty).

and consequently mechanical behavior of fabricated parts; see Figure 3.

The advent of AM has brought about radically new ways of manufacturing end-use products, by exploiting the space of computer-aided designs and process parameters. However, most previous AM experiments did not use BO to guide the selection of experiments, although simulations have revealed that using BO would be more efficient. Compared with domains like materials science, BO has received very little attention in the AM field. In this work, the experimental design space is a combination of product structural design (e.g. parametric design information, mesh data, material information, etc.) and manufacturing process parameters (e.g. laser power, layer thickness, hatching pitch, etc.). Finding the optimal design of an experiment can be challenging.
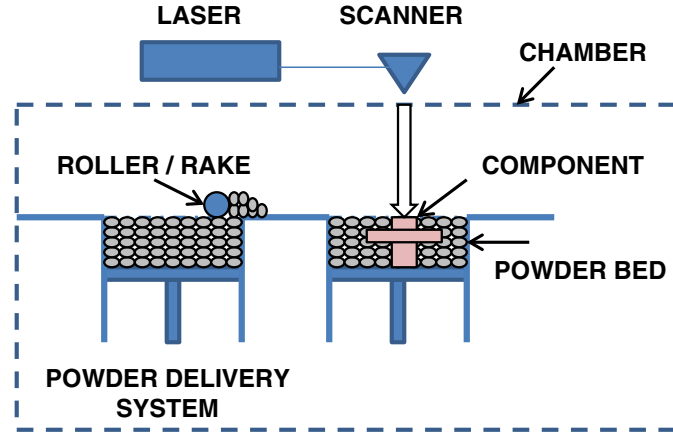
Figure 2: Generic illustration of an AM powder bed system. Reprinted from Frazier (2014).
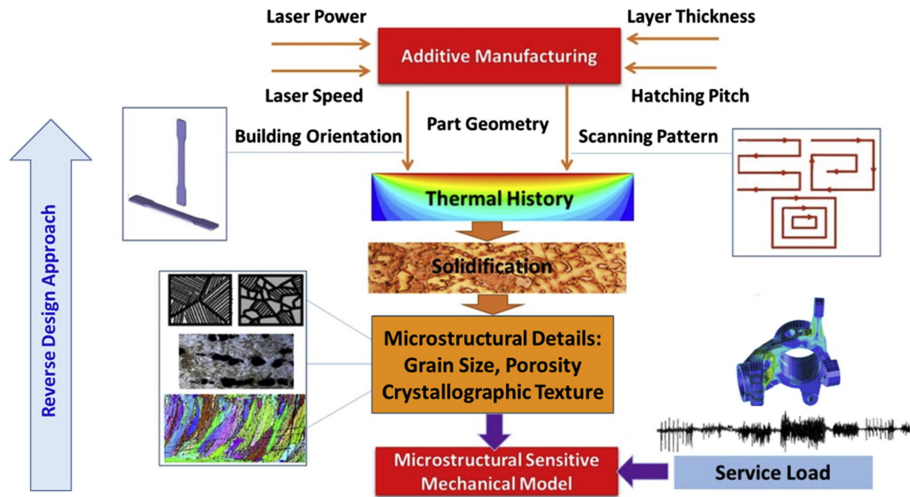

Figure 3: There are many factors (namely design parameters) that will jointly affect the structural properties of fabricated parts. Process parameters affect the thermal history of the AM parts. The thermal history during fabrication governs solidification, and consequently all the resultant microstructural details. Finally, the microstructural features dictate the structural properties of fabricated parts. Reprinted from Yadollahi and Shamsaei (2017), with permission from Elsevier.

A structured and systematic approach is required to effectively search the enlarged design space. Here we aim to bring attention to the benefits of applying BO in AM experiments and to provide a BO manual, covering both methodology and software, for the convenience of anyone (not limited to AM practitioners) who wants to apply or learn BO for experimental design. Our contributions include

4

- A thorough review of the literature on the application of BO to AM problems.

- An elaborate introduction to prominent open-source software, highlighting their core features.

- A detailed tutorial on implementing different BO software to solve different DoE problems in AM.

- An illustration of novel application of BO, where the output data are preference data.

The rest of the paper is organised as follows. In Section 2, we introduce the ingredients of BO: a probabilistic surrogate model and an acquisition function. In Section 3, we provide a review of the varied successful applications of BO in AM. In Section 4, we summarize popular open-source packages implementing various forms of BO. Section 5 provides code examples for bath optimization, multi-objective optimization and optimization with black-box constraints, and further illustrates a recently proposed extension of the basic BO framework to preference data. We summarize our findings in Section 6.

2   Bayesian Optimization

In this section we provide a brief introduction to BO. A more complete treatment can be found in Jones et al. (1998) and Frazier (2018). A review of BO and its applications can be found in Shahriari et al. (2016) and Greenhill et al. (2020). In general, BO is a powerful tool for optimizing "black-box" systems of the form:

$$\mathbf{x}^* = \arg \max_{\mathbf{x} \in \mathcal{X}} f(\mathbf{x}), \tag{1}$$

where $\mathcal{X}$ is the design space, and the objective function $f(\mathbf{x})$ does not have a closed-form representation, does not provide function derivatives, and only allows point-wise evaluation. The value of the black-box function f at any query point $\mathbf{x} \in \mathcal{X}$ can be obtained via, e.g. a physical experiment. However, the evaluation of f could be corrupted by noise. Thus the output y we many receive at point $\mathbf{x}$ is:

$$y = f(\mathbf{x}) + \varepsilon,$$

where the noisy term ε is usually assumed to be normally distributed with mean 0 and variance σ^2. A BO framework has two elements: a surrogate model for approximating the black-box function f, and an acquisition function for deciding which point to query for the next evaluation of f. Here we

employ the Gaussian Process (GP) model as the surrogate due to its popularity, but other models such as Bayesian neural networks are also commonly used as long as they provide a measure of uncertainty; see Forrester et al. (2008) for an overview. The pseudo code of the BO framework is given in Algorithm 1, where we iterate between steps 2 and 5 until the evaluation budget is over. A

Algorithm 1 Bayesian optimization

1: **for** $n = 1, 2, 3, \ldots,$ **do**

2:      select new $\mathbf{x}_{n+1}$ by optimizing the acquisition function α:

$$\mathbf{x}_{n+1} = \arg\max_{\mathbf{x} \in \mathcal{X}} \alpha(\mathbf{x}; \mathcal{D}_{1:n});$$

3:      query the objective function to obtain y_{n+1};

4:      augment data $\mathcal{D}_{1:(n+1)} = \{\mathcal{D}_{1:n}, (\mathbf{x}_{n+1}, y_{n+1})\}$;

5:      update the GP model;

6: **end for**

1D example in Figure 4 further explains Algorithm 1, where the upper panel corresponds to model fitting (steps 4&5), and the lower panel corresponds to deciding the next experiment (steps 2&3).

A GP is determined by its mean function $m(\mathbf{x}) = E[f(\mathbf{x})]$ and covariance function $k(\mathbf{x}, \ddot{\mathbf{x}}) = E[(f(\mathbf{x}) - m(\mathbf{x}))(f(\ddot{\mathbf{x}}) - m(\ddot{\mathbf{x}}))]$. For any finite collection of input points $\mathbf{x}_{1:n} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, the (column) vector of function values $f(\mathbf{x}_{1:n})$ follows the normal distribution $N(m(\mathbf{x}_{1:n}), k(\mathbf{x}_{1:n}, \mathbf{x}_{1:n}))$. Here, we employ compact notation for functions applied to collections of input points: $f(\mathbf{x}_{1:n}) = [f(\mathbf{x}_1), \ldots, f(\mathbf{x}_n)]^T$, $m(\mathbf{x}_{1:n}) = [m(\mathbf{x}_1), \ldots, m(\mathbf{x}_n)]^T$, and $k(\mathbf{x}_{1:n}, \mathbf{x}_{1:n})$ is the $n \times n$ covariance matrix with $k(\mathbf{x}_i, \mathbf{x}_j)$ being the (i, j)th element. The covariance function $k(\mathbf{x}, \ddot{\mathbf{x}})$ controls the smoothness of the stochastic process. It has the property that points $\mathbf{x}$ and $\ddot{\mathbf{x}}$ that are close in the input space have a large correlation, encoding the belief that they should have more similar function values than points that are far apart. A popular choice for the covariance function is the class of Matern kernels. Matern kernels are parameterized by a smoothness parameter $v > 0$, and samples from a GP with a higher v value are more smoother. The following are the most commonly used Matern kernels:

$$
\begin{aligned}
k_{v=\frac{1}{2}}(\mathbf{x}, \ddot{\mathbf{x}}) &= \theta_0^2 \exp(-r), \\
k_{v=\frac{3}{2}}(\mathbf{x}, \ddot{\mathbf{x}}) &= \theta_0^2 \exp(-\sqrt{3}r)(1 + \sqrt{3}r), \\
k_{v=\frac{5}{2}}(\mathbf{x}, \ddot{\mathbf{x}}) &= \theta_0^2 \exp(-\sqrt{5}r)(1 + \sqrt{5}r + \frac{5}{3}r^2),
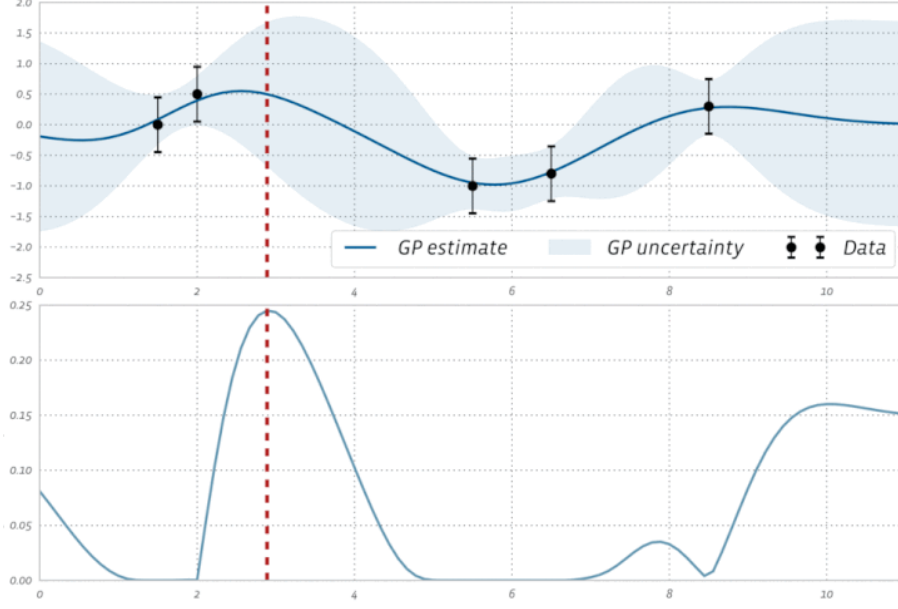\end{aligned}
$$

Figure 4: Upper panel: A surrogate model is fitted to five noisy observations using GPs to predict the objective (solid line) and place uncertainty estimates (proportional to the width of the shaded bands) over the range of possible parameter values. Lower panel: The predictions and the uncertainty estimates are combined to derive an acquisition function which quantifies the utility of running another experiment at a parameter value. The parameter value that maximizes the acquisition function (red dashed line) will be tested in the next experiment. The surrogate model and then the acquisition function will be updated with the results of that experiment. Figure produced by Facebook Ax.

where $r = (\mathbf{x} - \ddot{\mathbf{x}})^T \Lambda (\mathbf{x} - \ddot{\mathbf{x}})$, and Λ is a diagonal matrix of squared length scales θ_i^2.

For the query points $\mathbf{x}_{1:n}$, let $y_{1:n}$ denote the vector of noisy outputs and write $\mathcal{D}_{1:n} = \{(\mathbf{x}_i, y_i) : i = 1, \ldots, n\}$. Conditional on the data $\mathcal{D}_{1:n}$, we now want to infer the value of f at a new point $\mathbf{x} \in \mathcal{X}$. Our prior belief about the black-box function f is the GP, which implies that the prior $p(f(\mathbf{x}), f(\mathbf{x}_{1:n}))$ is normal. The likelihood for the outputs $y_{1:n}$ is a normal distribution $p(y_{1:n}|f(\mathbf{x}_{1:n})) = N(y_{1:n}; f(\mathbf{x}_{1:n}), \sigma^2 I)$, where I is the identity matrix. We can thus create the joint distribution:

$$p(f(\mathbf{x}), f(\mathbf{x}_{1:n}), y_{1:n}) = p(f(\mathbf{x}), f(\mathbf{x}_{1:n})) p(y_{1:n}|f(\mathbf{x}_{1:n})),$$

which is also a normal distribution. Therefore, the prior over outputs $y_{1:n}$ and target $f(\mathbf{x})$ is normal:

$$p(f(\mathbf{x}), y_{1:n}) = N\left(\begin{bmatrix} f(\mathbf{x}) \\ y_{1:n} \end{bmatrix} ; \begin{bmatrix} m(\mathbf{x}) \\ m(\mathbf{x}_{1:n}) \end{bmatrix}, \begin{bmatrix} k(\mathbf{x}, \mathbf{x}) & k(\mathbf{x}, \mathbf{x}_{1:n}) \\ k(\mathbf{x}_{1:n}, \mathbf{x}) & k(\mathbf{x}_{1:n}, \mathbf{x}_{1:n}) + \sigma^2 I \end{bmatrix} \right).$$

Utilizing the rule for conditionals, the distribution $p(f(\mathbf{x})|\mathcal{D}_{1:n}, \mathbf{x})$ is normal, with mean

$$m_n(\mathbf{x}) = m(\mathbf{x}) + k(\mathbf{x}, \mathbf{x}_{1:n})[k(\mathbf{x}_{1:n}, \mathbf{x}_{1:n}) + \sigma^2 I]^{-1}(y_{1:n} - m(\mathbf{x}_{1:n})),$$

and variance

$$\sigma_n^2(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}) - k(\mathbf{x}, \mathbf{x}_{1:n})[k(\mathbf{x}_{1:n}, \mathbf{x}_{1:n}) + \sigma^2 I]^{-1}k(\mathbf{x}_{1:n}, \mathbf{x}).$$

That is, after observing the data, our posterior belief about the black-box function f is still a GP.

The acquisition function is typically a computationally inexpensive function, evaluating how desirable a query point $\mathbf{x}$ is for the optimization problem (1). In other words, we replace our original optimization problem with another optimization problem, but on a cheaper function. One commonly used acquisition function is called expected improvement (EI). By maximizing the EI acquisition function, the next query point at which to evaluate f is the one that, in expectation, improves upon f the most. Let $\hat{f} = \max_{\mathbf{x}_i \in \mathbf{x}_{1:n}} f(\mathbf{x}_i)$ be the maximal value of f observed so far. For any query point $\mathbf{x}$, our utility would be $\max(0, f(\mathbf{x}) - \hat{f})$; that is, we receive a reward equal to the improvement $f(\mathbf{x}) - \hat{f}$ if $f(\mathbf{x})$ turns out to be larger than $\hat{f}$, and no reward otherwise. The EI acquisition function is then the expected utility at $\mathbf{x}$:

$$\begin{aligned}
\alpha(\mathbf{x}; \mathcal{D}_{1:n}) &= \mathrm{E}[\max(0, f(\mathbf{x}) - \hat{f})|\mathcal{D}_{1:n}] \\
&= \int_{\hat{f}}^{\infty} (f - \hat{f}) p(f|\mathcal{D}_{1:n}, \mathbf{x}) df \\
&= \int_{\hat{f}}^{\infty} (f - \hat{f}) \phi\left(\frac{f - m_n(\mathbf{x})}{\sigma_n(\mathbf{x})}\right) df \\
&= [m_n(\mathbf{x}) - \hat{f}]\Phi\left(\frac{\hat{f} - m_n(\mathbf{x})}{\sigma_n(\mathbf{x})}\right) + k(\mathbf{x}, \mathbf{x})\phi\left(\frac{\hat{f} - m_n(\mathbf{x})}{\sigma_n(\mathbf{x})}\right),
\end{aligned}$$

where Φ and ϕ are the CDF and PDF of the standard normal distribution, respectively. The EI acquisition function $\alpha(\mathbf{x}; \mathcal{D}_{1:n})$ has two components. The first can be increased by increasing the mean function $m_n(\mathbf{x})$. The second can be increased by increasing the variance $k(\mathbf{x}, \mathbf{x})$. These two terms can be interpreted as encoding a tradeoff between exploitation (evaluating at points with high mean) and exploration (evaluating at points with high uncertainty).

Another three popular acquisition functions are entropy search (ES), upper confidence bound (UCB) and knowledge gradient (KG). ES attempts to maximize our information about the location of the maximum $\mathbf{x}^*$ given new data. UCB is often described in terms of maximizing f rather than minimizing f. In the context of minimization, the acquisition function is called lower confidence bound, but UCB is ingrained in the literature as a standard term. The UCB acquisition function can be expressed as $\alpha(\mathbf{x}; \mathcal{D}_{1:n}) = m_n(\mathbf{x}) + \beta \sigma_n(\mathbf{x})$, where β is a tuning parameter. The query point

maximizing the UCB corresponds to a (fixed-probability) best case scenario. The KG criterion quantifies the expected increase in the maximum of the surrogate model from obtaining new data. Recently, a flurry of acquisition functions for batch optimization have been developed. These include Thompson sampling, local penalization, and extensions of classical acquisition functions such as qKG, dKG, qEI, bUCB, etc. A short survey of acquisition functions is given by Hunt (2020).[1]

3 Review of Bayesian Optimization in Additive Manufacturing

We performed a systematic search of the literature for the use of BO in AM, using the keywords "Bayesian", "sequential", "adaptive", "additive manufacturing" and "3D printing". Search for the literature was carried out via Scopus and Google Scholar. Research on applying Bayesian optimization to additive manufacturing is still at an early stage with limited literature. Only eight papers were found, and all of them were published in the last five years.

Mechanical metamaterials are artificial structures defined by their geometry rather than their chemical composition. One class of metamaterials is structural lattices. Three previous papers have applied BO for lattice structure design. To reduce the number of design variables, Sharpe et al. (2018) utilized the geometric projection technique and represented the struts in a lattice structure by rod primitives. The authors developed a constrained BO framework for micro-architecture (aka, unit cell) design, to achieve a high stiffness-to-weight ratio. Computationally expensive constraints are approximated by surrogate models, and the probabilities of satisfying the expensive constraints are incorporated as weights into the expected improvement function. Their study used two rods of equal diameter, yielding 13 total design variables. After the optimization process, the optimal design is obtained by collating the endpoints of bars exactly and extending the bars to the boundaries of the unit cell domain where necessary. The BO framework successfully identified a number of interesting unit cell designs that compare favorably with the popular octet truss design. Hertlein et al. (2020) studied both unit cell design optimization (with design variables: height, width, and strut thickness) and lattice structure design optimization (with two more design variables: number of rows and columns of unit cells). Both optimization problems have the same objective function: a weighted sum of the head injury criterion and part volume. The work focuses on two types of unit cells (diamond and honeycomb) which are parameterized by geometric attributes including height,

---

[1]A more complete list of acquisition functions can be found in the webpage "Acquisition Function APIs" of BoTorch (https://botorch.org/api/acquisition.html).

width, and strut thickness. Due to the computational expense of nonlinear finite element analysis and the difficulty of obtaining objective gradient information, the GP model is applied to approximate the objective function. The GP model is initialized with four random training points in the unit cell design optimization, and twelve random points in the lattice structure design optimization. Utilizing multi-material 3D printing techniques (designing novel composite solids by combining multiple base materials with distinct properties), it is possible to design mechanical metamaterials by varying spatial distributions of different base materials within a representative volume element (RVE), which is then periodically arranged into a lattice structure. In Xue et al. (2020), the goal is to design an RVE composed of different base materials so that when the RVEs are periodically arranged into a lattice structure, the macroscopic elastic moduli achieve a desired set of values. They adopted variational autoencoders to compress RVE images that describe the spatial distribution of base materials to a reduced latent space, and then performed BO over the reduced space to find an optimal RVE configuration that fulfills the design goal. The objective function measures the distance to the targeted Young's modulus and Poisson's ratio. The computational cost to evaluate the objective function is not cheap, so they apply a noise-free GP as the surrogate and use expected improvement as the acquisition function.

Pahl et al. (2007) defined a design process with four steps: requirement clarification, conceptual design, embodiment design, and detailed design. In embodiment design (aka, design exploration), designers determine the overall layout design, the preliminary form design, and the production processes based on design feasibility checks. In detailed design (aka, design exploitation), designers narrow down a set of potential solutions to one optimal design and provide ready-to-execute instructions. Xiong et al. (2019) focused on the embodiment design and detailed design steps in the context of AM. They applied the Bayesian network classifier model for embodiment design, and the GP model for detailed design. The first batch of samples, collected by the Latin hypercube sampling method, is utilized in both the embodiment design and the detailed design steps to train the two machine learning models. A computationally-expensive high-fidelity model is used to generate the corresponding responses. In the detailed design step, after training the GP model on the first batch of samples, they applied a Markov chain Monte Carlo (MCMC) sampling method to generate the second batch of design samples, evaluated the design samples via the high-fidelity model, and finally updated the GP model. Informed by the updated GP model, an optimal design is identified to meet certain targets. The method was exemplified by designing an ankle brace that includes tailored horseshoe structures in different regions to achieve stiffness requirements in the rehabilitation process. The initial batch contains 200 design samples distributed in

10

a five-dimensional space, and the MCMC sampling method generates another batch of 35 design samples. We note that the whole design process in Xiong et al. (2019) is not sequential, but the detailed design can be readily modified to be adaptive.

During layer-by-layer fabrication, with the thermal gradient and the solidification velocity in the melt pool changing continuously, the melt pool geometry undergoes substantial changes if the process parameters are not appropriately adjusted in real time. In Mondal et al. (2020), the goal is to dynamically determine the values of process parameters (laser power and linear scan velocity) to minimize the deviation from the target melt pool dimensions (melt pool depth and width). As the true functional form that relates the process parameters to the melt pool dimensions is unknown, GP is applied as the surrogate model. At each control time point, the Latin hypercube sampling method is used to generate 20 training samples, and Eagar-Tsai's model is used to generate the responses; the subsequent samples are determined by the expected improvement criterion; among the evaluated samples, the one with the optimal objective value provides the values of the process parameters. At the next time point of control, the whole process repeats.

Both Gongora et al. (2020) and Deneault et al. (2021) attempted to build an autonomous closed-loop AM system to accelerate the process of learning optimal printing conditions. The system developed by Gongora et al. (2020) consists of five dual extruder fused-deposition-modeling printers, a scale (CP225D, Sartorius) and a universal testing machine (5965, Instron Inc.). The goal is to design the best crossed barrel in terms of toughness. A crossed barrel has two platforms that are held apart by n hollow columns of outer radius r and thickness t and that are twisted with an angle θ; hence, the design space is four-dimension. Gongora et al. (2020) first performed a grid-based experimental campaign in which 600 distinct designs were tested in triplicate (without human intervention). Using the experimental data, they then ran a series of simulations and found that BO can reduce the number of experiments needed by a factor of almost 60. Deneault et al. (2021) built another autonomous system with material extrusion-type printers, which utilizes in-line image capture and analysis for feedback and BO for adaptive design. The design variables are four syringe extrusion parameters (i.e. prime delay, print speed, x-position, and y-position), and the goal is to print lines with the leading segment most closely matching the target geometry. The image of the leading segment of a line is compared to the target geometry, and an "objective score" is returned as the response. The prototype achieved near-ideal production of the user-defined geometry within 100 experiments.

4 Review of Bayesian Optimization Software

As of this writing, there are a variety of Bayesian optimization libraries out there. We here compare certain prominent packages that are available in R or Python.[2] R packages include:

- DiceOptim is a popular R package for sequential or batch optimization. The function *max_qEI* is for batch optimization, *noisy.optimizer* for noisy-output problems, and *EGO.cst* for constrained problems.

- laGP is mainly for GP modelling and inference. It provides a wrapper routine, i.e. *optim.auglag*, for optimizing black-box functions under under multiple black-box constraints.

- mlrMBO allows user-defined acquisition functions, generated with the *makeMBOInfillCrit* function. The functions *initSMBO* and *updateSMBO* are for designing physical experiments (i.e. step-by-step optimization with external objective evaluation).

Python packages include:

- Spearmint (https://github.com/HIPS/Spearmint) is one of the earliest user-friendly BO libraries. It is able to deal with constrained problems (the PESC branch) and multi-objective problems (the PESM branch). However, the latest version comes without the functionality for step-by-step optimization with external objective evaluation, and hence is only suitable for problems with callable objective functions.

- GPyOpt (https://github.com/SheffieldML/GPyOpt) is a popular Python library for optimizing physical experiments (sequentially or in batches). When optimizing the acquisition function, we can fix the value of certain variables. These variables are called context as they are part of the objective but are fixed when the acquisition function is optimized. GPyOpt allows to incorporate function evaluation costs (e.g. time) in the optimization.

- Cornell-MOE (https://github.com/wujian16/Cornell-MOE) is built on the MOE package. It is written in Python, with internal routines in C++. It supports box and simplex constraints. Although Cornell-MOE is easier to install (compared with the MOE package), it is only available for the Linux platform. While the MOE package allows optimization with external objective evaluation (via the function *gp_next_points*), the Cornell-MOE package only

---

[2]There are also Matlab packages like "bayesopt", but in this review we have only focused on open software.

works on callable functions. Cornell-MOE supports continuous-fidelity optimization via knowledge gradient Wu and Frazier (2017).

- GPflowOpt (https://github.com/GPflow/GPflowOpt) is built on the Python library GPflow (pre-2020 versions), a package for running GP modeling tasks on a GPU using Tensorflow. This makes GPflowOpt an ideal optimizer if GPU computational resources are available. It supports implementing user-defined acquisition functions.

- pyGPGO (https://github.com/josejimenezluna/pyGPGO) supports integrated acquisition functions and is able to optimize covariance function hyperparameters in the GP model. More information on this package can be found from one contributor's master thesis Luna (2017).

- Emukit (https://github.com/amzn/emukit) provides a way of interfacing third-part modelling libraries (e.g. a wrapper for using a model created with GPy). When new experimental data are available, it can decide whether hyper-parameters of the surrogate model need updating based on some internal logic.

- Dragonfly (https://github.com/dragonfly/dragonfly) library provides an array of tools to scale up BO to expensive large scale problems. It allows specifying a time budget for optimisation. The ask-tell interface in Dragonfly enables step-by-step optimization with external objective evaluation.

- Trieste (https://github.com/secondmind-labs/trieste) is built on the Python library GPflow (2.x version), serving as a new version of GPflowOpt. As of this writing, it is not recommended to install both Trieste and any other BO library that depends on GPy (e.g. GPyOpt); otherwise, we will get numpy-incompatible errors.

- BoTorch (https://github.com/pytorch/botorch) and Ax are two BO tools developed by Facebook. BoTorch supports both analytic and (quasi-) Monte-Carlo based acquisition functions. It provides an interface for implementing user-defined surrogate models, acquisition functions, and/or optimization algorithms. BoTorch has been receiving increasing attention from BO researchers and practitioners. Compared with BoTorch, Ax is relatively easier to use and targets at end-users.

In Figure 5 we compare the above libraries with respect to nine important features (available as of June/2021). By a "built-in model", we mean a wrapper function is already available for

Library	Built-in model	Acquisition function	Constraints	Multi-objective	Batch optimization	Multi-fidelity	Input type	High dimensional	External evaluation
DiceOptim	GP	EI, EQI, KG	✓	✗	✓	✗	1	✗	✓
laGP	GP	EI	✓	✗	✗	✗	1	✗	✗
mlrMBO	GP, RF	EI, EQI, UCB, PM	✗	✓	✓	✗	1, 2, 3	✗	✓
Spearmint	GP	EI, predictive ES	✓	✓	✓	✗	1, 2, 3	✗	✗
GPyOpt	GP, RF	EI, ES, UCB, PoI	✓	✗	✓	✗	1, 2, 3	✗	✓
Cornell-MOE	GP	EI, KG, predictive ES	✓	✗	✓	✓	1	✓	✗
GPflowOpt	GP	EI, UCB, PoF, PoI, max-value ES	✓	✓	✗	✗	1, 2, 3	✗	✗
pyGPGO	GP, GBM, RF, tSP	EI, PoI, UCB, predictive ES	✗	✗	✗	✗	1, 2	✗	✗
Emukit	GP	EI, ES, UCB, PoF, PoI, max-value ES	✓	✗	✓	✓	1, 2, 3	✗	✓
Dragonfly	GP	EI, PoI, TS, UCB	✓	✓	✓	✓	1, 2, 3	✓	✓
Trieste	GP	EI, UCB, PM, PoF, TS, max-value ES	✓	✓	✓	✗	1	✗	✓
BoTorch	GP	EI, KG, PM, PoI, UCB, max-value ES	✓	✓	✓	✓	1, 2, 3	✓	✓

Built-in model	Acquisition function	Input type
GBM: gradient boosting machine	EI (EQI): expected (quantile) improvement	1: continuous
RF: random forest	ES: entropy search	2: discrete
tSP: student-t process	KG: knowledge gradient	3: categorical
	PM: predictive mean	
	PoF (PoI): probability of feasibility (improvement)	
	TS: Thompson sampling	
	UCB: upper confidence bound	

Figure 5: Comparing different libraries with respect to if they can deal with non-box constrained problems, multi-objective problems, or high dimensional problems, and if they support batch optimization, multi-fidelity optimization, categorical input, or step-by-step optimization with external objective evaluation.

conveniently applying a surrogate model, e.g. the GPy GP wrapper in Emukit. For certain Python libraries (e.g. BoTorch), the acquisition function list is not complete; in particular, most acquisition functions for bath optimization are not included. All libraries can deal with box constraints. Multi-objective problems involve more than one objective function to be optimized simultaneously, and optimal decisions need to be taken in the presence of trade-offs between conflicting objectives. Batch optimization is not to be confused with parallel optimization/computing, where the former recommends multiple experiments at a time. Multi-fidelity optimization algorithms leverage both low- and high-fidelity data in order to speed up the optimisation process.

With many options at hand, a natural question is then which software is the most effective. Well, there is no clear winner. Kandasamy et al. (2020) compared Dragonfly, Spearmint and GPy-

Opt on a series of synthetic functions. On Euclidean domains, Spearmint and Dragonfly perform well across the lower dimensional tasks, but Spearmint is prohibitively expensive in high dimensions. On the higher dimensional tasks, Dragonfly is the most competitive. On non-Euclidean domains, GPyOpt and Dragonfly perform very well on some problems, but also perform poorly on others. Balandat et al. (2020) compared BoTorch, MOE, GPyOpt and Dragonfly through the problem of bath optimization (batch size = 4), on four noisy synthetic functions. The results suggest the ranking of BoTorch (OKG), GPyOpt (LP-EI), MOE (KG) and Dragonfly (GP Bandit), where we include the acquisition function in the parentheses. Vinod et al. (2020) showed that Emukit is efficient for solving constrained problems, where both the objective and constraint functions are unknown and are accessible only via first-order oracles. Li et al. (2021) compared BoTorch, GPflowOpt and Spearmint by tuning hyper-parameters of 25 machine learning models, and the average ranking is BoTorch, Spearmint and GPflowOpt. Note that this comparison does not necessarily show that one package is better than another; it rather compares the performance of their default methods.

5   Case Studies

Practitioners have often avoided implementing BO for DoE, mainly because the lack of tutorials on selecting and implementing available software. After comparing BO libraries in Section 4, we here provide detailed coding examples.

5.1   Batch Optimization with External Objective Evaluation

5.1.1   Background

Here we focus on the 3D printing of NiTi shape memory alloy in cuboidal samples via the laser-powder bed fusion process (Obeidi et al., 2021). The printer is equipped with a 200 W fibre laser with a wavelength of 1068 nm. The build chamber was supplied with argon gas of 99.99% purity. The powder layer thickness was fixed at 90 μm. Four processing parameters were investigated at three levels, producing 81 cubes of NiTi with the dimensions of 3 mm wide $\times$ 3 mm long $\times$ 4 mm high. The processing parameters and their levels are given in Table 1. The 81 cubical samples were evenly distributed on the 140 mm diameter 316L stainless steel build plate with a 10 mm space between consecutive cubes in order to avoid the effect of overheating of adjacent parts on the measured outputs. Archimedes's density measurement method was used to measure the relative

Table 1: Process parameters and their levels.

Process parameter	Units	L1	L2	L3	Lower	Upper
Laser power	W	120	150	180	100	200
Scanning speed	mm/sec	500	1000	1500	500	2000
Spot size	μm	40	60	80	30	80
Hatch spacing	μm	40	60	80	40	100

density of the printed 81 samples. Measurements were averaged from three replicates. Density values obtained by using the ethanol as the fluid are treated as the outputs.

The feasible ranges of the four parameters are not fixed but depend on various factors (such as the powder layer thickness, inert gas type/flow rate, metal powder type, etc.). For example, the input thermal energy cannot be way lower or higher than the melting point of the metal powder; otherwise, both lack-of-melting and over-melting power will result in voids and pore parts. For our printer (with a 316L stainless steel build plate), a laser power of 100 to 200 W, a scanning speed of 500 to 2000 mm/sec, a spot size of 30 to 80 microns, and a hatch spacing of 40 to 100 microns can work efficiently to melt a layer thickness of 20 to 60 microns. Therefore, our problem is box-constrained, with lower bounds {100, 500, 30, 40} and upper bounds {200, 2000, 80, 100}.

5.1.2 Code Examples

We copy the data into a file named "BatchObj.csv", with column names "Laser_power", "Scan_speed", "Spot_size", "Hatch_spacing", and "y" (for density). Given the current experimental data, we want to determine the next batch of experiments, hoping that, after a few more additional experiments, we will find an optimal experimental setting such that the printed parts are of desired density levels. Without loss of generality, we set the batch size to be two. To determine the next batch of experimental settings, we exemplify via the following 7 libraries: {DiceOptim, mlrMBO, GPy-Opt, Emukit, Dragonfly, Trieste, BoTorch}, due to they allowing batch optimization and external objective evaluation. Note that in all the code examples, we have standardized the output data before the optimization. This is mainly because certain libraries assume that the mean function of the GP model is zero.

DiceOptim & mlrMBO: The code for batch optimization with external evaluation is given in Figure 6. For the DiceOptim package, we need to define an object of class 'km' for fitting GP models. The argument 'crit' in the function *max_qEI* is for specifying the method for maximizing

16

```
data<-read.csv("BatchObj.csv", header=TRUE)
data[ ,5]<-(data[ ,5]-mean(data[ ,5]))/sd(data[ ,5])
#-----------------------------------------------------------------------------#
library("DiceOptim")
set.seed(0)
fitted.model<-km(~1, design=data.frame(data[ ,1:4]), response=data[ ,5], covtype="matern5_2",
               optim.method="BFGS", multistart=100, control=list(trace=FALSE, pop.size=50))
BO<-max_qEI(fitted.model, npoints=2, lower=c(100, 500, 30, 40),  upper=c(200, 2000, 80, 100),
           crit="exact", minimization=FALSE, optimcontrol=list(nStarts=10, method="BFGS"))
x_next=BO$par
#-----------------------------------------------------------------------------#
library("mlrMBO")
parameter_space=makeParamSet(makeNumericParam("Laser_power", lower=100, upper=200),
                             makeNumericParam("Scan_speed", lower=500, upper=2000),
                             makeNumericParam("Spot_size", lower=30, upper=80),
                             makeNumericParam("Hatch_spacing", lower=40, upper=100))
ctrl=makeMBOControl(propose.points=2, final.method="best.predicted", store.model.at=1)
ctrl=setMBOControlInfill(ctrl, filter.proposed.points=TRUE)
ctrl=setMBOControlMultiPoint(ctrl, method="moimbo", moimbo.objective="mean.se.dist",
                             moimbo.dist="nearest.better", moimbo.maxit=500L)
set.seed(0)
BO<-initSMBO(par.set=parameter_space, design=data.frame(lapply(data, as.numeric)),
             control=ctrl, minimize=FALSE, noisy=TRUE)
x_next=proposePoints(BO)$prop.points
```

Figure 6: Code example for batch optimization using DiceOptim and mlrMBO.

the qEI criterion. For the mlrMBO package, we need to define an mbo control object, here denoted by 'ctrl'. To define an mbo control object, we need to apply the *makeMBOControl* function, and then the *setMBOControlMultiPoint* function for batch optimization. Here we select 'moimbo' for proposal of multiple infill points. Note that if 'moimbo' is selected, the infill criterion in *setMBOControlInfill* is ignored, and hence we didn't specify any value for the argument 'crit' in the *setMBOControlInfill* function. The method 'moimbo' proposes points by multi-objective infill criteria via evolutionary multi-objective optimization. The other two options are 'cb' that proposes points by optimizing the confidence bound criterion, and 'cl' that proposes points by the constant liar strategy. The functionalities of DiceOptim and mlrMBO can be found from their reference manuals.

GPyOpt: Compared to other Python libraries, GPyOpt is relatively simple to use, and its functionalities are well documented. The code example in Figure 7 applies Thompson sampling for batch optimization, and another option is local penalization. The argument 'maximize' allows us to maximize the objective function.

Emukit: Though Emukit implements the experimental-design feature with the function *ExperimentalDesignLoop*, the function *BayesianOptimizationLoop* from the Bayesian-optimization feature is also suitable for step-by-step optimization with external objective evaluation. Figure 8

17

```python
import numpy
data=numpy.genfromtxt('BatchObj.csv', delimiter=',', skip_header=1)
X_init=data[:,[0,1,2,3]]
Y_init=(data[:,4]-data[:,4].mean())/data[:,4].std()
Y_init.shape=(81, 1)
parameter_space=[{'name': 'Laser_power', 'type': 'continuous', 'domain': (100, 200)},
                 {'name': 'Scan_speed', 'type': 'continuous', 'domain': (500, 2000)},
                 {'name': 'Spot_size', 'type': 'continuous', 'domain': (30, 80)},
                 {'name': 'Hatch_spacing', 'type': 'continuous', 'domain': (40, 100)}]

import GPyOpt
numpy.random.seed(0)
BO=GPyOpt.methods.BayesianOptimization(f=None, domain=parameter_space, X=X_init,
                                       Y=Y_init, normalize_Y=False,
                                       evaluator_type='thompson_sampling',
                                       batch_size=2, maximize=True)
x_next=BO.suggest_next_locations()
```

Figure 7: Code example for batch optimization using GPyOpt.

exemplifies the code. Here we set the GP kernel to be a combination of Matern-5/2 kernel and

```python
import numpy
from emukit.core import ParameterSpace, ContinuousParameter
data=numpy.genfromtxt('BatchObj.csv', delimiter=',', skip_header=1)
X_init=data[:,[0,1,2,3]]
Y_init=-(data[:,4]-data[:,4].mean())/data[:,4].std()
Y_init.shape=(81, 1)
parameter_space=ParameterSpace([ContinuousParameter('Laser_power', 100, 200),
                                ContinuousParameter('Scan_speed', 500, 2000),
                                ContinuousParameter('Spot_size', 30, 80),
                                ContinuousParameter('Hatch_spacing', 40, 100)])

import GPy
from emukit.model_wrappers import GPyModelWrapper
numpy.random.seed(0)
kern=GPy.kern.Matern52(input_dim=4, ARD=True)+GPy.kern.Bias(input_dim=4)
Model_gpy=GPy.models.GPRegression(X=X_init, Y=Y_init, kernel=kern,
                                  normalizer=False, noise_var=0.01, mean_function=None)
Model_gpy.optimize()
Model_emukit=GPyModelWrapper(Model_gpy)

from emukit.bayesian_optimization.acquisitions.expected_improvement import ExpectedImprovement
from emukit.bayesian_optimization.loops import BayesianOptimizationLoop
from emukit.core.optimization import GradientAcquisitionOptimizer
AcqFun=ExpectedImprovement(model=Model_emukit)
AcqFun_opt=GradientAcquisitionOptimizer(space=parameter_space)
BO=BayesianOptimizationLoop(space=parameter_space, model=Model_emukit, acquisition=AcqFun,
                            batch_size=2, acquisition_optimizer=AcqFun_opt)
x_next=BO.get_next_points(None)
```

Figure 8: Code example for batch optimization using Emukit.

bias kernel and apply the available wrapper *GPyModelWrapper* to plug GPy models into Emukit. Models written in GPy, TensorFlow, MXnet, etc. all can be wrapped in Emukit. To be consistent with the configurations of other BO libraries, we choose 'expected improvement' for batch

18

optimization; another popular choice is 'local penalization'. We optimize the acquisition function using a quasi-Newton method (L-BFGS).

Dragonfly: By default, Dragonfly maximizes functions. For a minimization problem, simply take the negative of the objective/outputs and perform the same procedure as below. The ask-tell interface in Dragonfly enables step-by-step optimization with external objective evaluation. Two main components are required in the ask-tell interface: a function caller and an optimizer. Our process parameters are continuous, and hence we adopt the *EuclideanFunctionCaller* and the *EuclideanGPBandit* optimizer. The code example is given in Figure 9. Note that, even though the

```python
import numpy
from dragonfly.exd import domains
data=numpy.genfromtxt('BatchObj.csv', delimiter=',', skip_header=1)
X_init=data[:,[0,1,2,3]]
Y_init=(data[:,4]-data[:,4].mean())/data[:,4].std()
parameter_space=domains.EuclideanDomain([[100, 200], [500, 2000], [30, 80], [40, 100]])

from dragonfly.exd.experiment_caller import EuclideanFunctionCaller
from dragonfly.opt import gp_bandit
from argparse import Namespace
func_caller=EuclideanFunctionCaller(None, parameter_space)
BO=gp_bandit.EuclideanGPBandit(func_caller, ask_tell_mode=True)
numpy.random.seed(0)
BO.initialise()
for i in range(0, 80):
    BO.tell([(X_init[i], Y_init[i])])
x_next=BO.ask(n_points=2)
```

Figure 9: Code example for batch optimization using Dragonfly.

EuclideanGPBandit optimizer includes the 'options' argument (that allows you to select an acquisition function from {TS, UCB, EI, top-two EI, batch UCB}), it is ignored if the 'ask_tell_mode' argument is set to be true. By setting 'n_points = 2', we will obtain the next two experimental settings.

Trieste: Trieste only supports minimizing the objective function, and hence we flip the sign of the output data from positive to negative. The code example is given in Figure 10. Note that we have normalized the inputs by linearly mapping the input space into the unit hypercube. This is for better estimating the length-scale parameters in the Matern kernel. The function *build_model* is to configure the GP model from GPflow, and the function *create_model* returns a trainable probabilistic model built according to the configuration. Note that we need to set the likelihood to be non-trainable. Two batch acquisition functions are supported in Trieste: *BatchMonteCarloExpectedImprovement* and the *LocalPenalizationAcquisitionFunction*. *BatchMonteCarloExpectedImprovement* jointly allocates the batch of points as those with the largest expected improvement

```
import numpy
import tensorflow as tf
import trieste
from trieste.space import Box
data=numpy.genfromtxt('BatchObj.csv', delimiter=',', skip_header=1)
low_bnd=numpy.array([100, 500, 30, 40])
up_bnd=numpy.array([200, 2000, 80, 100])
X_init=(data[:,[0,1,2,3]]-low_bnd)/(up_bnd-low_bnd)
Y_init=-(data[:,4]-data[:,4].mean())/data[:,4].std()
Y_init.shape=(81, 1)
ExpData=trieste.data.Dataset(tf.constant(X_init), tf.constant(Y_init))
parameter_space=Box([0, 0, 0, 0], [1, 1, 1, 1])

import gpflow
from trieste.models import create_model
def build_model(data):
    kern=gpflow.kernels.Matern52()
    gpr=gpflow.models.GPR(data=data.astuple(), kernel=kern, noise_variance=0.01)
    gpflow.set_trainable(gpr.likelihood, False)
    return {
        "model": gpr,
        "optimizer": gpflow.optimizers.Scipy(),
        "optimizer_args": {"minimize_args": {"options": dict(maxiter=1000)}},
        }
numpy.random.seed(0)
tf.random.set_seed(0)
Model=create_model(build_model(ExpData))

from trieste.acquisition import BatchMonteCarloExpectedImprovement
from trieste.acquisition.rule import EfficientGlobalOptimization
AcqFun=BatchMonteCarloExpectedImprovement(sample_size=1000)
BO=EfficientGlobalOptimization(builder=AcqFun, num_query_points=2)
x_next=BO.acquire_single(search_space=parameter_space, dataset=ExpData, model=Model,
                         state=None)
x_next[0].numpy()*(up_bnd-low_bnd)+low_bnd
```

Figure 10: Code example for batch optimization using Trieste.

over our current best solution. In contrast, the *LocalPenalizationAcquisitionFunction* greedily builds the batch, sequentially adding the maximizers of the standard (non-batch) EI criterion. The last line of code projects the query points in the unit hypercube back to the original input space.

BoTorch: BoTorch implements MC acquisition functions that support for batch optimization. They are referred to as q-acquisition functions (e.g. q-EI, q-UCB, and a few others). The code example in Figure 11 applies q-EI, the MC counterpart of the EI criterion. Note that if we apply the EI criterion, then the optimization will only suggest one experimental setting. The function *ExactMarginalLogLikelihood* is to compute the marginal log-likelihood of the GP model, and it works only when the likelihood is Gaussian.

We have set the seed in each code example, so that our results (given in Table 2) are reproducible. From Table 2 we cannot detect any strange result, except that the two query points returned by mlrMBO are close to each other. The main takeaways are: (1) We would not recommend the

20

```python
import numpy
import torch
data=numpy.genfromtxt('BatchObj.csv', delimiter=',', skip_header=1)
X_init=torch.from_numpy(data[:,[0,1,2,3]])
Y_init=numpy.array((data[:,4]-data[:,4].mean())/data[:,4].std())
Y_init.shape=(81, 1)
Y_init=torch.from_numpy(Y_init)
parameter_space=torch.tensor([[100.0, 500.0, 30.0, 40.0], [200.0, 2000.0, 80.0, 100.0]])

from botorch.models import SingleTaskGP
from gpytorch.mlls import ExactMarginalLogLikelihood
from botorch.fit import fit_gpytorch_model
Model=SingleTaskGP(train_X=X_init, train_Y=Y_init)
Model_mll=ExactMarginalLogLikelihood(Model.likelihood, Model)
fit_gpytorch_model(Model_mll)

from botorch.acquisition import qExpectedImprovement
from botorch.sampling import SobolQMCNormalSampler
from botorch.optim import optimize_acqf
qEI_sample=SobolQMCNormalSampler(num_samples=2048, resample=False, seed=0)
AcqFun=qExpectedImprovement(Model, best_f=Y_init.max(), sampler=qEI_sample)
torch.manual_seed(seed=0)
x_next, _=optimize_acqf(acq_function=AcqFun, bounds=parameter_space, q=2,
                        num_restarts=50, raw_samples=512, sequential=False)
```

Figure 11: Code example for batch optimization using BoTorch.

Table 2: Batch optimization results returned by different libraries.

Library	1st query point	2nd query point
DiceOptim	(153.06, 1499.57, 59.63, 57.91)	(171.38, 748.34, 70.28, 72.09)
mlrMBO	(181.71, 1062.62, 70.32, 52.51)	(186.22, 1307.64, 64.26, 57.20)
GPyOpt	(107.77, 1403.14, 45.22, 74.96)	(198.51, 652.70, 39.82, 65.34)
Emukit	(179.29, 1499.60, 79.84, 58.65)	(151.39, 500.12, 39.33, 80.20)
Dragonfly	(189.55, 743.65, 64.80, 80.71)	(176.72, 1801.22, 46.48, 92.42)
Trieste	(175.80, 1166.70, 63.54, 50.33)	(156.84, 846.95, 72.65, 56.63)
BoTorch	(140.11, 1599.53, 68.37, 72.28)	(199.89, 927.42, 63.11, 58.63)

ask-tell interface in Dragonfly, as it is of very limited flexibility. (2) BoTorch is ideal for BO practitioners, yet may be a bit too much for engineers. (3) For the two R libraries, DiceOptim is relatively simpler to use than mlrMBO.

5.2 Multi-objective Optimization with External Objective Evaluation

5.2.1 Background

For the technique of nanoparticle synthesis via laser ablation synthesis in solution (LASiS), a solid target is ablated using high-intensity pulsed laser irradiation while immersed in a liquid. This technique has been used for fabricating various kinds of nanoparticles, including semiconductor quantum dots, carbon nanotubes, nanowires, and core-shell nanoparticles. On the other hand, the properties of nanoparticles depend strongly on the size and size uniformity of the particles. Therefore, two major objectives in LASiS are to achieve a high nanoparticle production efficiency and to achieve a narrow size distribution. In this case study, nano-colloids were generated using a picosecond Nd:YAG laser by irradiating laser energy on the target surface. The laser beam power was 1.46 W. The target material was Ag. A recirculation production mode set up with 100 ml/min liquid flow rate was used for nano-colloids production. Ag nanoparticle solutions were produced by laser ablation of a solid target in deionized water by varying target ablation time. The produced nano-colloids were characterized using dynamic light scattering and UV-Vis spectroscopy. The nanoparticle productivity in deionized water was estimated by evaluating the loss of Ag target mass before and after the laser ablation.

The process parameters of interest are the length of time to run the laser ablation, the scan speed and repetition rate of the laser. We want to quickly determine an optimal parameter setting such that the productivity (ablated mass per ablation time interval) is high and the average particle size is small. To initialize the BO technique, a three-factor Box-Behnken design with five centre points was adopted, resulting in 17 experiments. The factor levels and ranges are given in Table 3. All factors are continuous, accurately set to one decimal place and to the nearest second for time.

Table 3: Process parameters and their levels.

Process parameter	Units	L1	L2	L3	Lower	Upper
Scan speed	mm/s	1.8	2.2	2.6	1.8	2.6
Repetition rate	kHz	10	20	30	10	30
Time	min	10	20	30	10	30

For each of the experiments, the solvent parameters (flow rate and pressure), stage height and laser power were kept constant.

5.2.2 Code Examples

We copy the data into a file named "MultiObj.csv", with column names "Scan_speed", "Repetition_rate", "Time", "y_1" (for ablated mass) and "y_2" (for average particle size). The goal in multi-objective optimization is to learn the Pareto front: the set of optimal trade-offs, where an improvement in one objective means deteriorating another objective. Our two objectives are (1) to maximize the ablated mass and (2) to minimize the average particle size. According to the summary table in Figure 5, we exemplify via the following 3 libraries: {mlrMBO, Trieste, BoTorch}, due to they allowing multi-objective optimization and external objective evaluation. For the python library Dragonfly, multi-objective optimization is currently not implemented in the ask-tell interface, and one has to define a callable function to do multi-objective optimization. Again, in all the following code examples, the outputs are shifted to have zero mean and then scaled to have unit variance.

mlrMBO: The code for multi-objective optimization with external evaluation is given in Figure 12. Compared with single-objective optimization, the main difference here is the additional func-

```
data<-read.csv("MultiObj.csv", header=TRUE)
data[ ,4]<-(data[ ,4]-mean(data[ ,4]))/sd(data[ ,4])
data[ ,5]<-(data[ ,5]-mean(data[ ,5]))/sd(data[ ,5])

library("mlrMBO")
parameter_space=makeParamSet(makeNumericParam("Scan_speed", lower=1.8, upper=2.6),
                             makeNumericParam("Repetition_rate", lower=10, upper=30),
                             makeNumericParam("Time", lower=10, upper=30))
ctrl=makeMBOControl(n.objectives=2, propose.points=2, store.model.at=1)
ctrl=setMBOControlInfill(ctrl, crit=makeMBOInfillCritEI(), filter.proposed.points=TRUE)
ctrl=setMBOControlMultiPoint(ctrl, method="cl")
ctrl=setMBOControlMultiObj(ctrl, method="parego")
set.seed(0)
BO<-initSMBO(par.set=parameter_space, design=data.frame(lapply(data, as.numeric)),
             control=ctrl, minimize=c(FALSE, TRUE), noisy=FALSE)
x_next=proposePoints(BO)$prop.points
```

Figure 12: Code example for multi-objective optimization using mlrMBO.

tion *setMBOControlMultiObj*. There are quite a few options in *setMBOControlMultiObj*, which require knowledge of multi-objective optimization. Here we simply accept all the default values. Optimization of noisy multi-objective functions is not supported in the moment. Setting of 'final.method' and 'final.evals' in the function *makeMBOControl* is not supported for multi-objective optimization either. For illustrative purpose only, here we apply a different acquisition function and a different proposal of multiple infill points. The two suggested query points are (2.54, 20.00, 27.72) and (2.59, 19.64, 16.43).

23

Trieste: Following the tutorials on the documentation site, we model the two objective functions individually with their own GP models. Figure 13 gives the code. The main changes are the

```python
import numpy
import tensorflow as tf
from trieste.data import Dataset
from trieste.space import Box
data=numpy.genfromtxt('MultiObj.csv', delimiter=',', skip_header=1)
low_bnd=numpy.array([1.8, 10, 10])
up_bnd=numpy.array([2.6, 30, 30])
X_init=(data[:,[0,1,2]]-low_bnd)/(up_bnd-low_bnd)
Y_init=(data[:,[3,4]]-data[:,[3,4]].mean(axis=0))/data[:,[3,4]].std(axis=0)
Y_init[:,0]=-Y_init[:,0]
ExpData=Dataset(tf.constant(X_init), tf.constant(Y_init))
parameter_space=Box([0, 0, 0], [1, 1, 1])

import gpflow
from trieste.models import create_model
from trieste.models.model_interfaces import ModelStack
def build_model(ExpData, num_obj)->ModelStack:
    gprs=[]
    kern=gpflow.kernels.Matern52()
    for iobj in range(num_obj):
        single_data=Dataset(ExpData.query_points, tf.gather(ExpData.observations, [iobj], axis=1))
        gpr=gpflow.models.GPR(single_data.astuple(), kernel=kern, noise_variance=0.01)
        gpflow.set_trainable(gpr.likelihood, False)
        gprs.append((create_model({
            "model": gpr,
            "optimizer": gpflow.optimizers.Scipy(),
            "optimizer_args": {"minimize_args": {"options": dict(maxiter=1000)}}
            }), 1))
    return ModelStack(*gprs)
numpy.random.seed(0)
tf.random.set_seed(0)
Model=build_model(ExpData, num_obj=2)

from trieste.acquisition.function import ExpectedHypervolumeImprovement
from trieste.acquisition.rule import EfficientGlobalOptimization
BO=EfficientGlobalOptimization(builder=ExpectedHypervolumeImprovement(), num_query_points=1)
x_next=BO.acquire_single(search_space=parameter_space, dataset=ExpData, model=Model, state=None)
x_next[0].numpy()*(up_bnd-low_bnd)+low_bnd
```

Figure 13: Code example for multi-objective optimization using Trieste.

function *build_model* and the new acquisition function. The function *build_model* simply builds a GP model for each objective and then applies the wrapper *ModelStack* to stack these two independent GPs into a single model working as a multi-output model. The acquisition function *ExpectedHypervolumeImprovement* is for multi-objective optimization. Note that this acquisition function does not support batch optimization, and hence we set the argument 'num_query_points' to be 1. The suggested query point is (2.60, 20.11, 22.70).

BoTorch: Again, we assume that the output features are independent and apply *SingleTaskGP* to fit a GP model for each objective. Figure 14 gives the code example. The acquisition function

```python
import numpy
import torch
data=numpy.genfromtxt('MultiObj.csv', delimiter=',', skip_header=1)
X_init=torch.from_numpy(data[:,[0,1,2]])
Y_init=(data[:,[3,4]]-data[:,[3,4]].mean(axis=0))/data[:,[3,4]].std(axis=0)
Y_init[:,1]=-Y_init[:,1]
Y_init=torch.from_numpy(Y_init)
parameter_space=torch.tensor([[1.8, 10.0, 10.0], [2.6, 30.0, 30.0]])

from botorch.models import SingleTaskGP
from gpytorch.mlls import ExactMarginalLogLikelihood
from botorch.fit import fit_gpytorch_model
Model=SingleTaskGP(train_X=X_init, train_Y=Y_init)
Model_mll=ExactMarginalLogLikelihood(Model.likelihood, Model)
fit_gpytorch_model(Model_mll)

from botorch.utils.multi_objective.box_decompositions.non_dominated import NondominatedPartitioning
from botorch.acquisition.multi_objective.monte_carlo import qExpectedHypervolumeImprovement
from botorch.sampling import SobolQMCNormalSampler
from botorch.optim import optimize_acqf
partitioning=NondominatedPartitioning(ref_point=torch.tensor([-2, -2]), Y=Y_init)
AcqFun=qExpectedHypervolumeImprovement(Model, ref_point=[-2, -2], partitioning=partitioning,
                                       sampler=SobolQMCNormalSampler(1000, seed=0))
torch.manual_seed(seed=0)
x_next, _ =optimize_acqf(acq_function=AcqFun, bounds=parameter_space, q=2,
                         num_restarts=100, raw_samples=512, sequential=True)
```

Figure 14: Code example for multi-objective optimization using BoTorch.

qEHVI requires specifying a reference point, which is the lower bound on the objective used for computing hypervolume. In practice the reference point can be set using domain knowledge to be slightly worse than the minimum acceptable value of interest for each objective. Since we have normalized our outputs, we might set the reference point for each objective to be -2. *qEHVI* also requires partitioning the non-dominated space into disjoint rectangles, here the function *NondominatedPartitioning*. Passing the keyword argument 'sequential=True' to the function *optimize_acqf* specifies that candidates should be optimized in a sequential greedy fashion (suggested by Daulton et al. (2020)). The two suggested query points are (2.22, 20.00, 30.00) and (2.49, 19.84, 29.83).

From the functionality point of view, we have a clear winner: BoTorch, for multi-objective optimization with external evaluation. BoTorch is capable of performing batch multi-objective optimization on noisy data. By contrast, mlrMBO cannot deal with noisy data, and Trieste cannot do batch optimization.

5.3 Optimization with Black-box Constraints

5.3.1 Background

In the LASiS example, the second objective is to minimize the average size of the particles produced in deionized water. However, for many nanoparticle applications, we do not need to minimize the (average) particle size but to keep it within an acceptable range (Albanese et al., 2012). Then we can formulate the second objective into a constraint on the first optimization problem, instead of into an additional optimization problem. Another AM example is fabricating NiTi products (Mehrpouya et al., 2019), where we want to optimize over laser power, laser speed, and hatching space such that the recovery ratio (the ratio of the recoverable strain to the total strain) is as high as possible and that the austenite finish temperature is below a threshold. In both examples, we have two types of goals: one is to optimize certain metrics, and the other is to satisfy requirements on other metrics. In the nomenclature of optimization, we call such problems as constrained optimization problems.

For classical constrained optimization problems, the constraints can be formulated in analytical form. By contrast, a distinguishing feature of the above constrained problems is that the constraints are black-box: just like the objective function, the constraints are unknown and can only be evaluated pointwise via expensive queries to "black boxes" that may provide noise-corrupted values. More formally, we are interested in finding the global maximum of a scalar objective function $f(\mathbf{x})$ over some bounded domain, typically $\mathbf{x} \in \mathcal{X}$, subject to the non-positivity of a set of constraint functions $c_1, ..., c_k$. We write this as

$$\max_{\mathbf{x} \in \mathcal{X}} f(\mathbf{x}), \quad \text{s.t.} \quad c_1(\mathbf{x}) \leq 0, \ldots, c_k(\mathbf{x}) \leq 0, \tag{2}$$

where f and $c_1, ..., c_k$ are all black-box functions. We seek to find a solution to Equation (2) with as few queries as possible.

We here use the data from Mehrpouya et al. (2019) for fabricating NiTi shape memory alloys. A ProX 200 direct metal printer was used to fabricate parts on a NiTi substrate, and the fabrication chamber was purged continuously with Argon. The layer thickness was fixed at 30 μm, and the scanning strategy was bidirectional. 17 cylindrical samples with a diameter of 4.5 mm and a height of 10 mm were fabricated. Laser power, scanning speed and hatch spacing were varied among the samples, and the outputs of interest are transformation temperature and recovery ratio. Transformation temperatures were realized by a Perkin-Elmer DSC Pyris with a heating and cooling rate of 10°C/min, and determined by tangent method from the DSC curve. For the mechanical testing,

a 100 kN MTS Landmark servo-hydraulic was used for the compression test which was done by a 10^{-4}/sec strain rate. For strain measurement, an MTS high-temperature extensometer was attached to the grips. The chamber temperature was kept at Af+ 10°C for mechanical testing.

5.3.2 Code Examples

We copy the data into a file named "BBcon.csv", with column names "Laser_power" (W), "Scan_speed" (mm/s), "Hatch_spacing" (μm), "Recovery_ratio" (%) and "Austenite_finish" (°C). Our objective is to find the optimal design, with as few experiments as possible, such that the recovery ratio is close to 100%, and the Austenite finish temperature is lower than 10°C. The libralies DiceOptim, laGP, Spearmint, GPflowOpt and BoTorch are all capable of dealing with black-box constraints. We here apply the R package DiceOptim and the Python package BoTorch, due to they allowing external objective evaluation.

DiceOptim: A code example is given in Figure 15. Since DiceOptim assumes a minimiza-

```
data<-read.csv("BBcon.csv", header=TRUE)
data[ ,4]<--(data[ ,4]-mean(data[ ,4]))/sd(data[ ,4])
data[ ,5]<-data[ ,5]-10

library("DiceOptim")
set.seed(0)
fitted.model<-km(~1, design=data.frame(data[ ,1:3]), response=data[ ,4], covtype="matern5_2",
                 nugget=0.001, nugget.estim=TRUE, optim.method="BFGS", multistart=1000,
                 control=list(trace=FALSE, pop.size=500))
fitted.cst<-km(~1, design=data.frame(data[ ,1:3]), response=data[ ,5], covtype="matern5_2",
               nugget=0.001, nugget.estim=TRUE, optim.method="BFGS", multistart=1000,
               control=list(trace=FALSE, pop.size=500))
BO<-critcst_optimizer(crit="EFI", fitted.model, fitted.cst, equality=FALSE, lower=c(100, 100, 80),
                      upper=c(300, 1500, 200), type="UK",
                      critcontrol=list(tolConstraints=0.001, n.mc=1000, slack=TRUE),
                      optimcontrol=list(method="genoud", max.generations=100, pop.size=200))
x_next=BO$par
```

Figure 15: Code example for constrained optimization using DiceOptim.

tion problem, we flip the sign of the recovery ratio from positive to negative. We subtract all the temperature measurements by 10 to transform the constraint into the form of Equation (2). In the second block of code, we approximate both the objective function and the constraint function by GP models, where we add a small nugget effect to avoid numerical problems. For the optimization part, we select expected feasible improvement (EFI) as the acquisition function, as suggested by Chaiyotha and Krityakierne (2020). DiceOptim supports equality and inequality black-box constraints. Running the code above, we will get the next query point (230.54, 190.60, 161.19).

Note that, for a callable objective function, one can apply the function *EGO.cst* for sequential constrained optimization and model re-estimation.

BoTorch: The code example given in Figure 16 is similar to that for the bath optimiza-

```python
import numpy
import torch
data=numpy.genfromtxt('BBcon.csv', delimiter=',', skip_header=1)
X_init=torch.from_numpy(data[:,[0,1,2]])
Y_init=data[:,[3,4]]
Y_init[:,0]=(Y_init[:,0]-Y_init[:,0].mean())/Y_init[:,0].std()
Y_init[:,1]=Y_init[:,1]-10
Y_init=torch.from_numpy(Y_init)
parameter_space=torch.tensor([[100.0, 100.0, 80.0], [300.0, 1500.0, 200.0]])

from botorch.models import SingleTaskGP
from gpytorch.mlls import ExactMarginalLogLikelihood
from botorch.fit import fit_gpytorch_model
Model=SingleTaskGP(train_X=X_init, train_Y=Y_init)
Model_mll=ExactMarginalLogLikelihood(Model.likelihood, Model)
fit_gpytorch_model(Model_mll)

from botorch.acquisition.objective import ConstrainedMCObjective
from botorch.sampling import SobolQMCNormalSampler
from botorch.acquisition import qExpectedImprovement
from botorch.optim import optimize_acqf
def obj_callable(Z):
    return Z[..., 0]
def constraint_callable(Z):
    return Z[..., 1]
constrained_obj=ConstrainedMCObjective(objective=obj_callable,
                                       constraints=[constraint_callable])
qEI_sample=SobolQMCNormalSampler(num_samples=2048, resample=False, seed=0)
AcqFun=qExpectedImprovement(Model, best_f=(Y_init[:,0]*(Y_init[:,1]<=0)).max(),
                            sampler=qEI_sample, objective=constrained_obj)
torch.manual_seed(seed=0)
x_next, _=optimize_acqf(acq_function=AcqFun, bounds=parameter_space, q=2,
                        num_restarts=50, raw_samples=512, sequential=False)
```

Figure 16: Code example for constrained optimization using BoTorch.

tion problem. The only difference is the additional function *ConstrainedMCObjective* that computes the feasibility-weighted expected improvement in the objective. The two callable functions, *obj_callable* and *constraint_callable*, define a construct to index the objective function and constraint function. The idea here is similar to that of DiceOptim: fitting a cheaper statistical model to the underlying expensive-to-evaluate constraint function, just like the objective function is modeled by a surrogate model. Running the code above, we will get two query points (165.52, 1470.11, 156.68) and (100.06, 127.79, 114.50).

Compared with DiceOptim, AM practitioners may find it more difficult to understand the BoTorch code. However, after these three case studies, it is clear that BoTorch is the most flexible among all the libraries listed in Figure 5. To conclude, we do not need worry too much about the

performance/efficiency of a BO library. It is the functionality that dominates our choice.

5.4 Preferential Output

The performance of a printing job is usually characterized by one or multiple expensive-to-evaluate metrics: surface roughness, mechanical properties, dimensional accuracy and tolerances, etc. It may not be economical or efficient to collect the exact values of a performance metric. For example, optical measurement techniques such as focus variation, fringe projection, confocal laser scanning microscopy, and profilometers can be used to measure the surface roughness of fabricated products. However, different optical measuring methods produce (significantly) different values (Launhardt et al., 2016); from the optimization point of view, it is the ordering of the measurements (rather than the values of the measurements) that are of importance here. Mechanical-property tests (such as deformation property tests and failure property tests) are for quantifying mechanical properties like tensile strength, hardness, and fatigue strength. However, mechanical-property tests involve specialized test equipment, custom-fabricated fixtures, skilled test engineers, etc., and hence are commonly performed by commercial companies, which leads to a lengthy printing-testing cycle. For void or porosity analysis, X-ray computed tomography (micro-CT) is now the leading technique to analyze AM products non-destructively. However, the availability of micro-CTs and the knowledge of image analysis limit the application of micro-CT for porosity quantification. Again from the optimization point of view, we do not need that much information from micro-CT images but the information of which of two products has a lower porosity.

The idea behind collecting preference data instead of real-valued measurements is to accelerate and economize the DoE procedure, by replacing an advanced measuring technique with a simple one (or with a low-fidelity simulation model) and reaching a preferential judgement (Shahriari et al., 2015; González et al., 2017; Benavoli et al., 2021a). Our output now is not a real value y_i but a pairwise preference of the form $\mathbf{x}_i \succ \mathbf{x}_j$, meaning that the input configuration $\mathbf{x}_i$ is preferred over $\mathbf{x}_j$. We can create a list to store the preference data: $\mathbf{P} = \{(\mathbf{x}_i, \mathbf{x}_j), (\mathbf{x}_k, \mathbf{x}_m)\ldots\}$, where the order implies the preference. Alternatively, we can define a symmetric matrix $\mathbf{P} = [p_{ij}]_{n \times n}$ to represent the preference data:

$$p_{ij} = \begin{cases} 1, & \text{if } \mathbf{x}_i \succ \mathbf{x}_j; \\ -1, & \text{if } \mathbf{x}_i \prec \mathbf{x}_j; \\ 0, & \text{if no preference between } \mathbf{x}_i \text{ and } \mathbf{x}_j. \end{cases}$$

We want to find the optimum of the black-box function f with minimal preferential information.

Note that we are not against collecting real-valued measurements, and preferential BO is of advantage only when obtaining the preference $\mathbf{x}_i \succ \mathbf{x}_j$ is much cheaper than obtaining the real output y_i. In fact, BO can deal with mixed data where we have both preference data and real-valued measurements.

We here give a data-analysis example using the data from Cao et al. (2021). The study investigated the influence of process parameters (laser power, scanning speed, and layer thickness) on surface roughness and dimensional accuracy. Experiments were conducted with a fiber laser operated in continuous mode with a wavelength of 1071 nm and a spot diameter of 50-80 μm. Surface roughness was measured with a surface-roughness tester (Mitutoyo Surftest SJ-210) with 2.5 mm sampling length and 750 μm/s measuring speed. A laser scanning confocal microscope (Smartproof 5) was also applied to analyze the surface topography. The dimensions of the parts were measured using a digital vernier caliper with a precision of 10 μm. Taguchi L16 orthogonal array design was applied with three factor at four levels, and additional five random experiments were conducted, giving in total 21 initial experiments. To reduce randomness in experimental results, three cuboid samples with dimensions 45 mm × 5 mm × 1 mm were fabricated in each experiment with the same design. The lower and upper bounds for the three process parameters are {180 W, 20 μm, 500 mm/s} and {240 W, 35 μm, 800 mm/s}.

We copy the data into a file named "SurfRough.csv", with the original column names: "Laser_power", "Scanning_speed", "Layer_thickness", "Surface_roughness", "Length", "Length_dif", "Width", "Width_dif", "Height", and "Height_dif". In this example, we focus only on the surface roughness, not the dimensional accuracy. We assume that the surface-roughness measurements are of variation, due to aleatory uncertainty and/or epistemic uncertainty, and the error bound is 1 μm. That is, taking the first experiment as an example, the mean surface roughness could take any value in the interval [4.797, 6.797]. According to the 21 surface-roughness measurements, we can create a preference list $\mathbf{P} = \{(\mathbf{x}_1, \mathbf{x}_2), (\mathbf{x}_2, \mathbf{x}_3) \ldots\}$: we have $\mathbf{x}_1 \succ \mathbf{x}_2$, $\mathbf{x}_2 \succ \mathbf{x}_3$, and there is no preference between, e.g., $\mathbf{x}_1$ and $\mathbf{x}_6$. Given the preferential data $\mathbf{P}$ and a GP prior on the black-box function f, the posterior is a skew Gaussian process (SkewGP) (Benavoli et al., 2020, 2021b). Note that, different from the case of real-valued measurements where the surrogate model in the BO framework is GP, for the case of preferential output, the surrogate model in the BO framework is SkewGP. We therefore apply the SkewGP Python library[3] on the preferential data $\mathbf{P}$ to get the next query point.

Running the code in Figure 17, we will get the next query point $(212.93, 28.89, 743.46)$. Here, we created the preferential data $\mathbf{P}$, a list, from the original data, while in practice the preferential

---

[3]The library is available at `https://github.com/benavoli/SkewGP`.

```python
import numpy as np
data=np.genfromtxt('SurfRough.csv', delimiter=',', skip_header=1)
X_init=data[:,[0,1,2]]
Y_init=data[:,3]
Preference=[] #create the preferential data P
for i in range(0, X_init.shape[0]):
    for j in range(i+1, X_init.shape[0]):
        if Y_init[j]-Y_init[i]>=2:
            Preference.append([i,j])
        elif Y_init[j]-Y_init[i]<=-2:
            Preference.append([j,i])
maX=X_init[np.argmax(Y_init)] #best experiment
Class=list(np.ones(X_init.shape[0])) #all experiments have outputs
parameter_space=[[180, 240], [20, 35], [500, 800]]

import GPy, commoncode, BO
#function calls to experiment at X
def query(X):
    return np.array([1.0])
#function returns 0 if X is an invalid setting (experiment no output), 1 otherwise
def valid(X, f):
    return np.array([1.0])
#RBF kernel function
def RBF(X1, X2, hype, diag_=False):
    lengthscale=hype['lengthscale']['value']
    variance=hype['variance']['value']
    kernel=GPy.kern.RBF(X1.shape[1], lengthscale=lengthscale, variance=variance, ARD=True)
    return kernel.K(X1,X2)
kernel=RBF #initialize kernel
#initial values for the hyperparameters of the kernel
hype0={'lengthscale':{'value':np.ones((1, X_init.shape[1]))*0.3,
                      'range':np.vstack([[np.exp(-5.0), np.exp(5.0)]]*X_init.shape[1]),
                      'transform': commoncode.logexp()},
       'variance':{'value':np.array([10.0]),
                   'range':np.vstack([[np.exp(-5.0), np.exp(4.1)]]),
                   'transform': commoncode.logexp()},
       'noise_variance':{'value':np.array([1.0]),
                         'range':np.vstack([[1.0, 1.0001]]),
                         'transform': commoncode.logexp()}}
PBO=BO.BO(X_init, Preference, Class, parameter_space, kernel, hype0, query, valid,
          maX, surrogateM='SGP', acquisition='UCB')
#update the kernel hyperparameters and compute the next point
PBO.find_next(query, 0, update_model='True')
print("next point = ", PBO.X[-1,:])
```

Figure 17: Code example for preferential BO using the SkewGP Python library.

data and the variable "Class" are generally read from external files. The SkewGP library can deal with outputs of mixed types. For example, if an experiment fails to print any product or prints a faulty product, then the experimental setting is invalid, and the output will be like $(\mathbf{x}_i, 0)$, where 0 means invalid (and 1 means valid); in other words, we will get the judgment of invalid, not any pairwise preference. Therefore, in our code example, we define the variable "Class", a vector of all 1's, to indicate that all experiments are valid.

6 Conclusions

BO is a methodology for sample-efficient learning and optimization. By leveraging a probabilistic model, it is able to explore a large design space using only a small number of experimental trials. Our review in the field of AM reveals that BO, although indispensable for a competitive and digital AM, receives few applications until very recently. To facilitate and accelerate the adoption of BO by AM practitioners, we tabulated mature open-source BO packages concerning nine important practical considerations, serving as a guide on software selection. We also summarized the findings of existing comparative studies of BO packages, imparting more information on their relative efficiency. To help engineers dive right into coding their own problems, we provided code examples for several simple yet representative case studies, serving as a convenient one-stop reference. We wound up this work with an example of applying BO to new types of data, hoping to inspire creative applications and increase interest in BO. As a general outlook, there is a trend toward autonomous/digitalized AM, and we believe BO is a key technological element for future intelligent manufacturing.

Acknowledgements

References

Albanese, A., Tang, P. S., and Chan, W. C. (2012). The effect of nanoparticle size, shape, and surface chemistry on biological systems. *Annual Review of Biomedical Engineering*, 14(1):1–16.

Balandat, M., Karrer, B., Jiang, D., Daulton, S., Letham, B., Wilson, A., and Bakshy, E. (2020). Botorch: A framework for efficient monte-carlo bayesian optimization. volume 2020-December.

Benavoli, A., Azzimonti, D., and Piga, D. (2020). Skew gaussian processes for classification. *Machine Learning*, 109:1877–1902.

Benavoli, A., Azzimonti, D., and Piga, D. (2021a). Preferential Bayesian optimisation with Skew Gaussian Processes. In *2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21 Companion), July 10–14, 2021, Lille, France*, New York, NY, USA. ACM.

Benavoli, A., Azzimonti, D., and Piga, D. (2021b). A unified framework for closed-form non-parametric regression, classification, preference and mixed problems with skew gaussian processes. *arXiv preprint arXiv:2008.06677*.

Cao, L., Li, J., Hu, J., Liu, H., Wu, Y., and Zhou, Q. (2021). Optimization of surface roughness and dimensional accuracy in lpbf additive manufacturing. *Optics & Laser Technology*, 142:107246.

Chaiyotha, K. and Krityakierne, T. (2020). A comparative study of infill sampling criteria for computationally expensive constrained optimization problems. *Symmetry*, 12(10).

Daulton, S., Balandat, M., and Bakshy, E. (2020). Differentiable expected hypervolume improvement for parallel multi-objective bayesian optimization. In *Advances in Neural Information Processing Systems*, volume 33, pages 9851–9864. Curran Associates, Inc.

Deneault, J. R., Chang, J., Myung, J., Hooper, D., Armstrong, A., Pitt, M., and Maruyama, B. (2021). Toward autonomous additive manufacturing: Bayesian optimization on a 3d printer. *MRS Bulletin*.

Deshwal, A., Simon, C., and Doppa, J. R. (2021). Bayesian optimization of nanoporous materials. *Cambridge: Cambridge Open Engage*.

Forrester, A. I. J., Sóbester, A., and Keane, A. J. (2008). *Engineering Design via Surrogate Modelling: A Practical Guide*. John Wiley & Sons.

Frazier, P. I. (2018). A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*.

Frazier, W. (2014). Metal additive manufacturing: A review. *Journal of Materials Engineering and Performance*, 23(6):1917–1928.

Gongora, A. E., Xu, B., Perry, W., Okoye, C., Riley, P., Reyes, K. G., Morgan, E. F., and Brown, K. A. (2020). A bayesian experimental autonomous researcher for mechanical design. *Science Advances*, 6(15).

González, J., Dai, Z., Damianou, A., and Lawrence, N. D. (2017). Preferential bayesian optimization. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1282–1291. JMLR. org.

Greenhill, S., Rana, S., Gupta, S., Vellanki, P., and Venkatesh, S. (2020). Bayesian optimization for adaptive experimental design: A review. *IEEE Access*, 8:13937–13948.

Hertlein, N., Vemaganti, K., and Anand, S. (2020). Bayesian Optimization of Energy-Absorbing Lattice Structures for Additive Manufacturing. volume 6 of *ASME International Mechanical Engineering Congress and Exposition*.

Hunt, N. (2020). Batch bayesian optimization. Master thesis, Massachusetts Institute of Technology.

Jones, D. R., Schonlau, M., and Welch, W. J. (1998). Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492.

Kandasamy, K., Vysyaraju, K., Neiswanger, W., Paria, B., Collins, C., Schneider, J., Póczos, B., and Xing, E. (2020). Tuning hyperparameters without grad students: Scalable and robust bayesian optimisation with dragonfly. *Journal of Machine Learning Research*, 21.

Lam, R., Poloczek, M., Frazier, P., and Willcox, K. E. (2018). *Advances in Bayesian Optimization with Applications in Aerospace Engineering*, page 1656.

Launhardt, M., Wörz, A., Loderer, A., Laumer, T., Drummer, D., Hausotte, T., and Schmidt, M. (2016). Detecting surface roughness on sls parts with various measuring techniques. *Polymer Testing*, 53:217–226.

Li, Y., Shen, Y., Zhang, W., Chen, Y., Jiang, H., Liu, M., Jiang, J., Gao, J., Wu, W., Yang, Z., et al. (2021). Openbox: A generalized black-box optimization service. *arXiv preprint arXiv:2106.00421*.

Luna, J. J. (2017). Bayesian optimization in machine learning. Master thesis, The Polytechnic University of Catalonia.

Mehrpouya, M., Gisario, A., Rahimzadeh, A., Nematollahi, M., Baghbaderani, K., and Elahinia, M. (2019). A prediction model for finding the optimal laser parameters in additive manufacturing of niti shape memory alloy. *International Journal of Advanced Manufacturing Technology*, 105(11):4691–4699.

Mondal, S., Gwynn, D., Ray, A., and Basak, A. (2020). Investigation of melt pool geometry control in additive manufacturing using hybrid modeling. *Metals*, 10(5).

Obeidi, M. A., Monu, M., Hughes, C., Bourke, D., Dogu, M. N., Francis, J., Zhang, M., Ahad, I. U., and Brabazon, D. (2021). Laser beam powder bed fusion of nitinol shape memory alloy (sma). *Journal of Materials Research and Technology*.

Pahl, G., Beitz, W., Feldhusen, J., and Grote, K.-H. (2007). *Engineering Design:A Systematic Approach*. Springer-Verlag London.

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., and De Freitas, N. (2015). Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175.

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., and de Freitas, N. (2016). Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175.

Sharpe, C., Seepersad, C. C., Watts, S., and Tortorelli, D. (2018). Design of Mechanical Metamaterials via Constrained Bayesian Optimization. volume 2A of *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*.

Turner, R., Eriksson, D., McCourt, M., Kiili, J., Laaksonen, E., Xu, Z., and Guyon, I. (2021). Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020.

Vahid, A., Rana, S., Gupta, S., Vellanki, P., Venkatesh, S., and Dorin, T. (2018). New bayesian-optimization-based design of high-strength 7xxx-series alloys from recycled aluminum. *JOM*, 70(11):2704–2709.

Vinod, A. P., Israel, A., and Topcu, U. (2020). Constrained, global optimization of functions with lipschitz continuous gradients.

Wu, J. and Frazier, P. I. (2017). Continuous-fidelity bayesian optimization with knowledge gradient.

Xiong, Y., Duong, P. L. T., Wang, D., Park, S.-I., Ge, Q., Raghavan, N., and Rosen, D. W. (2019). Data-Driven Design Space Exploration and Exploitation for Design for Additive Manufacturing. *Journal of Mechanical Design*, 141(10).

Xue, T., Wallin, T. J., Menguc, Y., Adriaenssens, S., and Chiaramonte, M. (2020). Machine learning generative models for automatic design of multi-material 3d printed composite solids. *Extreme Mechanics Letters*, 41:100992.

Yadollahi, A. and Shamsaei, N. (2017). Additive manufacturing of fatigue resistant materials: Challenges and opportunities. *International Journal of Fatigue*, 98:14–31.